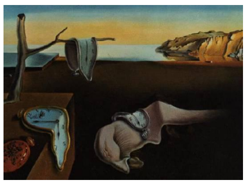


Journal of Statistical Software

July 2019, Volume 90, Issue 1. doi: 10.18637/jss.v090.i01

Efficient Code for Second Order Analysis of Events on a Linear Network

Suman Rakshit
Curtin University

Adrian Baddeley
Curtin University

Gopalan Nair
The University of Western Australia

Abstract

We describe efficient algorithms and open-source code for the second-order statistical analysis of point events on a linear network. Typical summary statistics are adaptations of Ripley's K-function and the pair correlation function to the case of a linear network, with distance measured by the shortest path in the network. Simple implementations consume substantial time and memory. For an efficient implementation, the data structure representing the network must be economical in its use of memory, but must also enable rapid searches to be made. We have developed such an efficient implementation in C with an R interface written as an extension to the R package **spatstat**. The algorithms handle realistic large networks, as we demonstrate using a database of all road accidents recorded in Western Australia.

Keywords: geometric correction, K-function, pair correlation function, point process, R, shortest-path distance, **spatstat**.

1. Introduction

The study of events that occur along a network of lines, such as traffic accidents recorded on a road network, requires the development of advanced statistical techniques and computational algorithms (Okabe and Sugihara 2012; Ver Hoef, Peterson, and Theobald 2006; Baddeley, Rubak, and Turner 2015, Chapter 17). Because a linear network is not a homogeneous space, even elementary statistical tools can be difficult to implement. Kernel smoothing of point events, which is simple to define and very fast to compute in two dimensions (Diggle 1985), is mathematically complicated and can be extremely time-consuming to perform on a network (Okabe, Satoh, and Sugihara 2009). Similar difficulties arise in second-order (correlation) analysis of point patterns, which is straightforward in two dimensions using Ripley's K-function (Ripley 1977) and the pair correlation function (Okabe and Yamada 2001; Ang, Baddeley, and Nair 2012; Baddeley, Jammalamadaka, and Nair 2014).

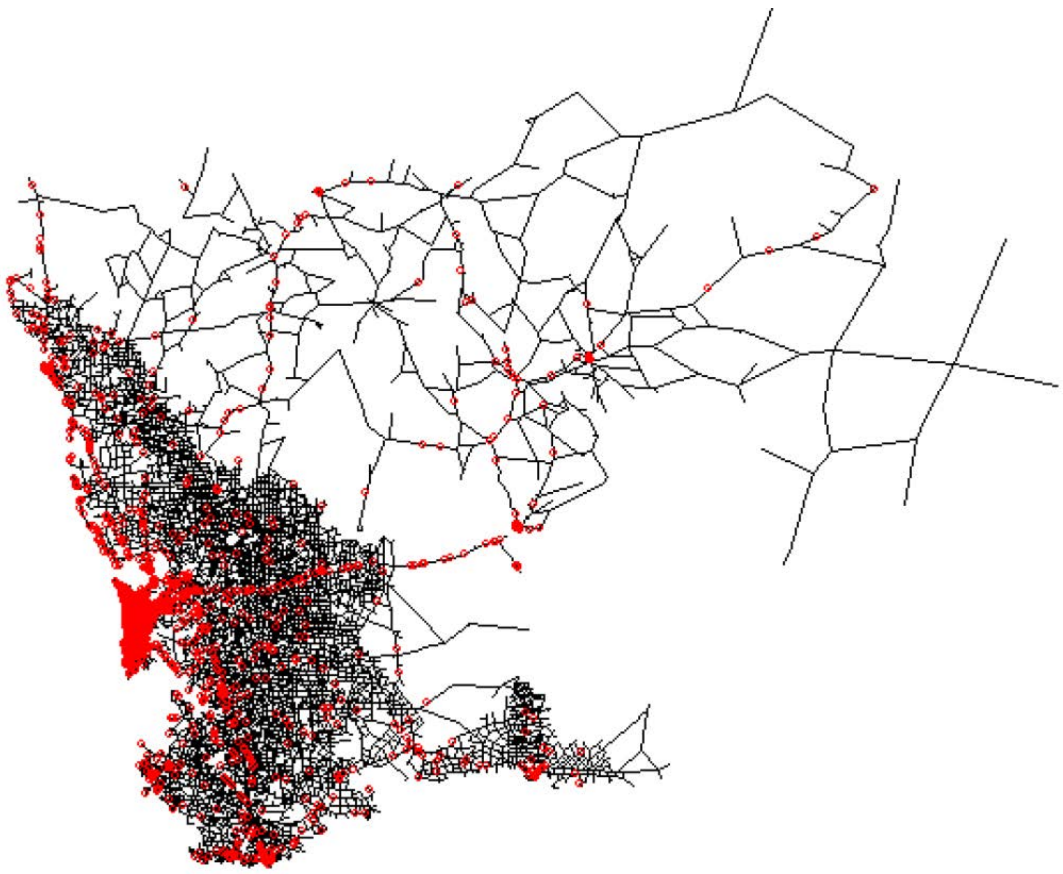


Figure 1: Traffic accidents (red dots) recorded in the year 2011 on the entire road network (black lines) of the state of Western Australia.

This geometrical complexity militates against the statistical analysis of real data sets of moderate size. Figure 1 shows the locations of road accidents recorded on the road network in the state of Western Australia for the year 2011. In this area, about 2000 km across, the network consists of $115,169$ road segments and there are $14,562$ accident locations. Kernel smoothing and second-order analysis of these accident data are prohibitively expensive (both in computer time and memory) using current implementations of these methods (Baddeley *et al.* 2015, Chapter 17; Okabe and Sugihara 2012) as we demonstrate below. Figure 2 shows a much smaller data set that can easily be handled with simple R code (R Core Team 2019): It contains 116 points on a network with 503 line segments. For kernel smoothing on a network, a fast algorithm capable of handling very large data sets was recently developed by McSwiggan, Baddeley, and Nair (2016) and is now implemented in the R package **spatstat** (Baddeley *et al.* 2015) as the function `density.lpp`.

In this paper, the main focus is the second-order (correlation) analysis of point patterns on a linear network. We develop efficient algorithms and open-source code for computing general second-order summary functions which include the K-function and pair correlation function.

Suppose we have observed point events $x_1, \ldots, x_p$ on a linear network L. Let $d_L(x_i, x_j)$ denote the *shortest-path distance* between data points x_i and x_j in the network. The objective is to

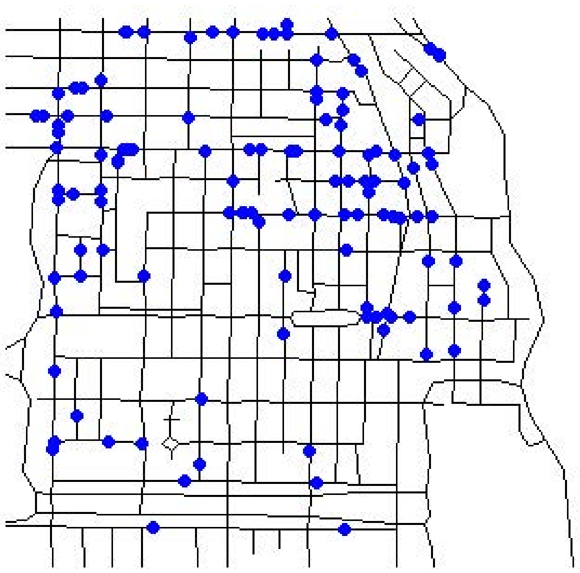


Figure 2: Chicago crime data. Street address locations of 116 crimes recorded in a two-week period around the University of Chicago. Extracted from a report in the Chicago Weekly News, 2002 and analyzed in Ang *et al.* (2012).

calculate second-order summary statistics of the general form

$$S(r) = \sum_{i=1}^{p} \sum_{j \neq i} h(x_i, d_L(x_i, x_j), r), \qquad r \geq 0, \tag{1}$$

where h is a chosen function. A simple example of Equation 1 is the empirical cumulative distribution function of the shortest-path distances

$$\widehat{F}(r) = \frac{1}{p(p-1)} \sum_{i=1}^{p} \sum_{j \neq i} \mathbf{I}(d_L(x_i, x_j) \leq r), \tag{2}$$

where $\mathbf{I}(A)$ is the indicator that equals 1 when A is true, and 0 otherwise. Other examples of the general form (1) include the observed network K-function (Okabe and Yamada 2001; Okabe and Sugihara 2012, Chapter 6), empirical estimators of the geometrically-corrected network K-function and pair correlation function (Ang *et al.* 2012), and various generalizations involving spatially-varying weights, auxiliary variables and local statistics (Ang *et al.* 2012; Baddeley *et al.* 2014; Boots and Okabe 2007). A detailed description of these estimators and their applications can be found in Baddeley *et al.* (2015, Chapter 17). Computation of the summary statistics of the form (1) is important, not only for exploratory data analysis, but also for fitting models to point pattern data by maximum composite likelihood (Guan 2006; Tanaka, Ogata, and Stoyan 2008; Baddeley *et al.* 2015, Chapter 12).

Simple code for calculating any statistic of the form (1) is available in the R package **spatstat** (Baddeley and Turner 2005; Baddeley *et al.* 2015). Table 1 (under the column heading *Adjacency matrix*) shows the computation time (in seconds) and the total memory (sum of all memory allocation requests) used by this implementation to compute the geometrically-corrected K-function (Ang *et al.* 2012, Equation 12) for three small example data sets supplied in the **spatstat** package.

			Algorithm			
			Adjacency matrix		Linked list	
Data set	Points	Lines	Time	Memory	Time	Memory
`spiders`	48	203	0.1	20	0.1	0.07
`chicago`	116	503	0.3	180	4.0	0.16
`dendrite`	566	639	5.0	2853	34.0	0.70

Table 1: Performance comparison of two algorithms for computing the geometrically-corrected K-function. *Adjacency matrix:* Algorithm M using adjacency matrices, described in Section 4, as implemented in **spatstat**; *Linked list:* Algorithm L using linked lists, described in Sections 5–6, as implemented in the supplied package **spatstat.Knet**. Row names refer to three example data sets supplied in the **spatstat** package. Column headings are as follows: *Points:* Number of data points; *Lines:* Number of line segments; *Time:* Elapsed computation time in seconds, reported by `system.time`; *Memory:* Sum of all memory allocations in megabytes, reported by the function `profmem` in package **profmem** (Bengtsson 2018).

Extrapolating to the Western Australian road accident data (Figure 1), under simple assumptions, gives a predicted computation time of at least 2 hours and total memory allocation of at least 1.4 terabytes. On a standard PC, such large amounts of memory are not available, and the algorithm will not run successfully.

For the simplest case of the network K-function (Okabe and Yamada 2001, Equation 7), which is equivalent to the computing in Equation 2, an efficient algorithm has been described by Okabe and Yamada (2001); see also Okabe and Sugihara (2012, Chapter 6). The code is not open-source, although compiled executables are available (Okabe, Okunuki, and Shiode 2006).

This paper presents an alternative, open-source, C implementation for computing any statistic of the general form introduced in Equation 1. Detailed pseudocode is included; the full source code is available within the R package **spatstat.Knet** (Rakshit and Baddeley 2019) from the Comprehensive R Archive Network (CRAN) at `https://CRAN.R-project.org/package=spatstat.Knet`. Our implementation uses a simple data structure and efficient code, which are easily adaptable to different choices of the function h in Equation 1 including the geometrically-corrected K-function (Section 3.2). This implementation adapts many of the ideas of Okabe and Yamada (2001), including the key concept of the (extended) shortest-path tree.

The traffic accident data in Figure 1 have been included as a data set named `wacrashes` in the **spatstat.Knet** package. The following code can be used to create Figure 1.

```
R> library("spatstat.Knet")
R> data("wacrashes", package = "spatstat.Knet")
R> plot(wacrashes, cols = "red", cex = 0.5, main = " ")
```

Figure 2 can be plotted as follows

```
R> plot(unmark(chicago), cols = "blue", pch = 16, main = " ")
```

Section 2 introduces necessary mathematical and computational structures such as adjacency matrices and linked-lists, for representing linear networks and events on networks. Section 3 gives some notation associated with the shortest-path distance on a network, different versions of the K-function, and their current implementation in **spatstat**. Section 4 describes the existing algorithm, which we call Algorithm M, for computing the K-function using the adjacency matrix. Sections 5 and 6 present a new algorithm, which we dub Algorithm L, using linked-lists. Section 7 gives a worked example of the key part of Algorithm L. Timings, as a function of the number of observed points on the network, are reported in Section 8. The Western Australian road accident data are analyzed in Section 9. We end with a discussion.

2. Representation of events on a linear network

In this section, we introduce the terminology associated with a linear network (Section 2.1), describe the traditional adjacency-matrix and adjacency-list data structures that are used for storing simple networks (Section 2.2), and propose a new data structure for storing point patterns on a linear network (Section 2.3).

2.1. Linear networks

A *straight line segment* in the two-dimensional plane with endpoints v and v' is the set $s = \{tv + (1-t)v' : \ 0 \leq t \leq 1\}$. The length of s is $\ell(s) = \|v - v'\|$, the Euclidean distance between its endpoints.

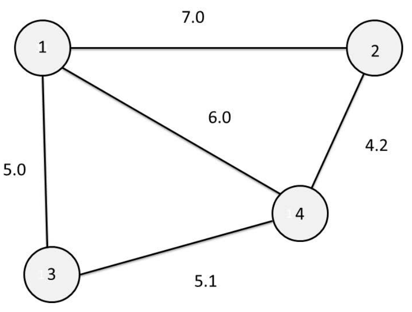
A *linear network* is the union $L = \bigcup_{i=1}^{n} s_i$ of a finite collection of straight line segments $s_1, \dots, s_n$; the total length of the network is $|L| = \sum_{i=1}^{n} \ell(s_i)$. The representation of L as a union of line segments is not unique: we assume that a representation is chosen so that any two distinct segments s_i, s_j with $j \neq i$ either do not intersect, or intersect at a common endpoint of s_i and s_j. Then the network can be considered as an embedded planar graph, whose vertices are the endpoints of the segments.

In a planar graph setting, it is common to refer the line segments as *edges* and their endpoints as *nodes*. However, we make a small distinction between a segment and an edge, which will be explained in the next subsection. The set of nodes (vertices) is denoted by $V = \{v_1, \dots, v_m\}$ and the set of segments by $S = \{s_1, \dots, s_n\}$. Both V and S are indexed sets with the subscript of an element representing its integer identifier in the set. Therefore, though the nodes $v_i \in L \subset \mathbb{R}^2$, we often use the labeling $1, \dots, m$ to denote m nodes. In what follows, without loss of generality, we assume $i < j$ in the representation $[v_i, v_j]$ of a segment in S.

2.2. Data structure for storing a network

Given a spatial linear network L we can construct a *weighted undirected graph* $G = (V, E)$, where V is the set of nodes as before, and E is the set of weighted edges. Each weighted edge is an ordered triple (v, v', w), where v and v' are endpoints of some segment $s \in S$ and w is the positive weight associated with the segment s; in this paper, we take the weight to be the segment length $w = \ell(s)$. More details on the connection between linear networks and weighted graphs can be found in Kolaczyk and Csárdi (2014).

Note that, corresponding to each segment $s = [v, v'] \in S$, there are two weighted edges (v, v', w) and (v', v, w) in G. In the ordered representation (v, v', w) of an edge, we shall refer to v as the *starting node* and v' as the *ending node*. Because all algorithms in this paper are developed assuming such double representation of the segments, in what follows, we use data



	1	2	3	4
1	0	7.0	5.0	6.0
2	7.0	0	∞	4.2
3	5.0	∞	0	5.1
4	6.0	4.2	5.1	0

Figure 3: A weighted undirected graph (left) with circles representing nodes with node identifiers, and the corresponding weighted adjacency matrix (right) with positive values indicating the weights.

structures that store information about all $2n$ weighted edges. Two standard data structures for representing a weighted graph are the *adjacency list* and *adjacency matrix* (see Cormen, Leiserson, Rivest, and Stein 2009, p. 589), which are discussed below.

Adjacency matrix

The (weighted) adjacency matrix of G is the $m \times m$ matrix $A = (a_{ij})$ in which a_{ij} is equal to the weight of the edge joining vertices i and j, or $a_{ij} = \infty$, if there is no such edge. Note that a_{ij} is finite if and only if $i \sim j$, i.e., i and j are adjacent to each other. The left panel of Figure 3 shows an example of an undirected graph, and the right panel shows the corresponding adjacency matrix with $V = \{1, \dots, 4\}$ and $E = \{(1, 2, 7.0), (2, 1, 7.0), (2, 4, 4.2), (4, 2, 4.2), (1, 4, 6.0), (4, 1, 6.0), (3, 4, 5.1), (4, 3, 5.1), (1, 3, 5.0),(3, 1, 5.0)\}$.

When a network is represented by its adjacency matrix, software coding becomes relatively straightforward for computing functions on the network. For example, the task of finding the immediate neighbors (or adjacent nodes) of a given node can easily be implemented by extracting the relevant row of the adjacency matrix and finding all finite entries. However, the main drawback of the adjacency matrix is the high memory usage when it is represented as a full matrix with m^2 entries. For the Western Australian road network, shown in Figure 1, there are $m = 88,512$ nodes; since $m^2 > 2^{32}$, the full adjacency matrix would exceed the array size limits in many 32-bit software systems, and would be too large to fit into random-access memory (RAM) on a typical 64-bit PC.

For a weighted undirected graph constructed from a road network, the adjacency matrix is usually *sparse* in the sense that the number of edges is much less than $m(m-1)$, the maximum possible number of finite entries in the matrix. In such cases one can use a *sparse matrix* representation (Wilkinson and Reinsch 1971; Tewarson 1973; Pissanetzky 1984; Golub and Van Loan 1996). In a sparse representation of an adjacency matrix A, only the finite positive entries of A are recorded, essentially as a list of triples (i, j, a_{ij}) giving the endpoints and weight associated with each weighted edge in G. This reduces the storage requirement to the minimum possible. Table 2 compares the memory storage requirements of the full and sparse matrix representations for four example networks, as reported by the R utility `object.size`.

Network data	Full matrix	Sparse matrix
`spiders`	314	33
`chicago`	1389	60
`dendrite`	4899	112
`wacrashes`	`NA`	9238

Table 2: Memory storage requirements (`kb`) for linear networks using the adjacency matrix representation in the **spatstat** package, with or without sparse matrix encoding. Network data are example network data sets provided in the **spatstat** and **spatstat.Knet** packages.

Element	Description
`data`	Pointer to a data object that is stored in the list.
`next`	Pointer to the next list entry in the linked-list.

Table 3: Components of a linked-list entries and their description.

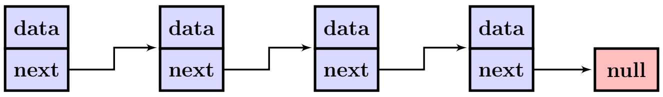


Figure 4: A linked-list.

Computations involving sparse matrices are supported by fast code in low-level languages, usually Fortran libraries (Dongarra, Moler, Bunch, and Stewart 1979; Anderson *et al.* 1999) with interfaces to higher-level languages such as R (Koenker and Ng 2003; Bates and Maechler 2019). However, most of the functionality provided by sparse matrix libraries is for linear algebra, rather than graph topology. Furthermore, the representation of a graph structure by an unstructured list of edges (i, j, a_{ij}) leads to computational inefficiencies. For example, the task of finding the immediate neighbors of a given node i^* requires a search through the entire list to find all entries (i, j, a_{ij}) where either $i = i^*$ or $j = i^*$. This can be accelerated by sorting the list appropriately, but the data structure is inherently inefficient for our purposes.

Adjacency list

Hopcroft and Tarjan (1973) advocated the adjacency list representation for graphs in terms of *linked-lists*. A linked-list is a standard data structure for representing a list of objects which are related in some way, e.g., nodes that are all connected to a given node or road accidents recorded on the same road segment. Each entry in a linked-list is a pair of pointers, namely, `data` and `next`, whose description is given in Table 3. The end of the list is indicated by assigning a null value to the pointer to the next entry (see Figure 4). List entries can easily be inserted or deleted at any position by changing the relevant pointers, so that linked-list data structures are well suited to applications where the connections between the list entries are required to be changed frequently (Cormen *et al.* 2009, p. 236; Louden 1999, Chapter 5).

The adjacency list representation of a weighted graph G consists of a list of m linked-lists, one linked-list for each node in the graph. The linked-list corresponding to a particular node stores all its adjacent nodes, along with the associated edge weights. Recall that each line segment $s = [v, v']$ in L corresponds to two weighted edges in G; the edges $(v, v', \ell(s))$ and $(v', v, \ell(s))$ appear, respectively, in the adjacency lists for nodes v and v'.

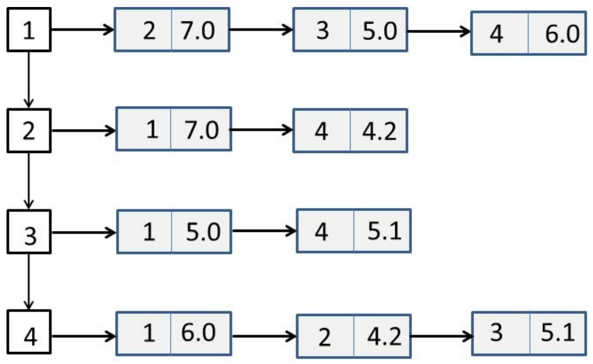


Figure 5: Adjacency list representation of the graph in Figure 3.

Figure 5 shows the adjacency list representation of the graph in Figure 3. The first column represents the list of four nodes, and for a given node, the corresponding row represents the linked-list containing its adjacent nodes along with the edge weights. For ease of illustration, in Figure 5 we omitted the `next` elements of the linked-list structures and the crooked arrows in Figure 4 representing pointers were replaced with the straight arrows.

For sparse graphs, the adjacency list representation is compact, utilizing only $O(n + m)$ space for storing the graph, and is more efficient than the adjacency matrix representation for implementing graph searching algorithms (Cormen *et al.* 2009; Even 1979; Tarjan 1983). An adjacency matrix representation is efficient only when the graph is *dense*, i.e., when the number of edges n is of the same order as m^2 (see Cormen *et al.* 2009; Even 1979).

2.3. Events on a network

A data set of events on a linear network L will be represented as $\mathbf{x} = \{x_1, \ldots, x_p\}$, where $x_i \in L$ is the location of the ith event on the network, and $p \geq 0$ is the total number of points, which is not fixed in advance. Figure 6 depicts a simple illustrative example of data giving the spatial locations $\{x_1, \ldots, x_5\}$ of events along a network.

Although the planar Cartesian coordinates are sufficient to locate point events, the data structure will be more computationally efficient if it makes an explicit connection between the point event and the segment on which it lies. This can be done using the elements `x`, `y`, `seg` and `tp` corresponding to a point event; description of these elements are given in Table 4.

The coordinate system with both `(x, y)` and `(seg, tp)` is mathematically redundant, but allows efficient addressing of different databases of spatial information. In road accident analysis, some explanatory variables, such as shoulder width, road curvature, and road condition, are part of a database of road information. For any event on the network, it is convenient to query this database using the road name or number (i.e., `seg`) and position along the road (i.e., `tp`). Other explanatory variables, such as terrain elevation, are spatially-referenced images, which are most conveniently queried using spatial coordinates `(x, y)`.

There is an important distinction between explanatory variables and 'marks', which could be part of the database. An explanatory variable is potentially observable at any spatial location: examples are terrain height and road width. A mark, such as crash severity or the number of passengers, is an additional attribute of the observed event. For simplicity of presentation, this paper does not consider marks.

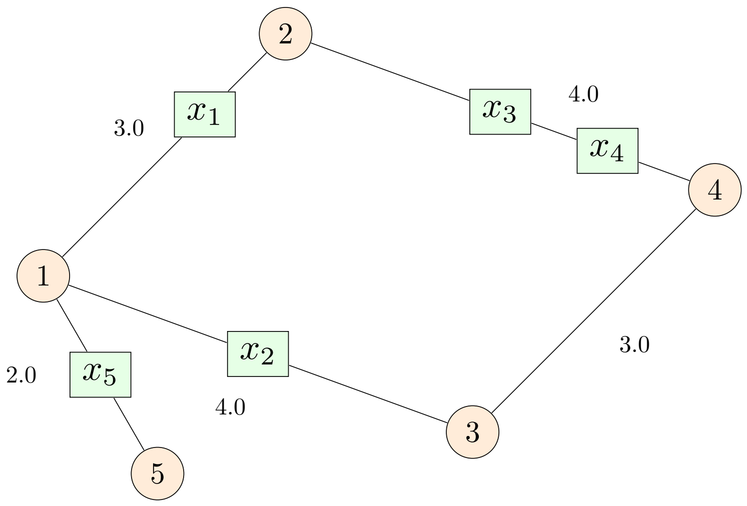


Figure 6: Illustrative example data set of events on a network. Circles are nodes of the network; boxes are event locations; real numbers are segment lengths (distances between nodes).

Element	Description
`(x, y)`	Planar Cartesian coordinates of the point event.
`seg`	Integer label of the segment containing the event.
`tp`	Relative distance of the point event along the segment; `tp` values 0 and 1, respectively, correspond to the starting node and ending node of the segment.

Table 4: Elements for storing information about a point event.

Data structure for a linear network with events

For efficient statistical investigation of the point patterns on a linear network, we have developed a data structure that extends the adjacency list representation by including storage for events and additional data about them. The elementary components of this data structure, namely `Adjlist`, `sNode`, and `aNode`, are sketched in Figure 7, and Table 5 provides a brief description of the members of these different components. Figure 7 also includes the `Crash` structure, which is used for storing point events.

The `sNode` objects are used for storing all the nodes in a weighted graph. For a given node in V, the `node` member of `sNode` stores the integer identifier of the node, and the members `d` and `parent` are used for storing, respectively, the shortest-path distance estimates and information about some relevant nearest node while computing the shortest path route from some source point in the network (see Procedure 1 in Section 5 for details).

For a given node $v \in V$, the `Adjlist` object connects v to all its adjacent nodes and keeps track of the resulting edges. The `data` member is a pointer to the `sNode` object that stores the starting node v, and the `adjacent` member is a pointer that keeps reference of the start of the linked-list that stores all the adjacent nodes of v. In each entry of this linked-list, the `data` member is a pointer to the `aNode` structure that contains the integer identifier of the adjacent node and all information about the edge connecting v to that adjacent node.

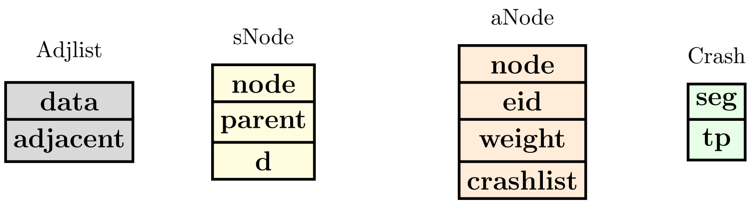


Figure 7: Different components of the data structure representing a network.

Component	Description
`sNode`	
`node`	Integer identifier of a given starting node
`parent`	Pointer to the parent node in the shortest-path route
`d`	Shortest-path distance from some source
`aNode`	
`node`	Integer identifier of a given ending node
`eid`	Integer identifier of the edge
`weight`	Length of the edge
`crashlist`	List of point events (`Crash` structures) on the edge.
`Adjlist`	
`data`	Pointer to the starting node (`sNode` object) of an edge
`adjacent`	Pointer to the beginning of the list of all adjacent edges corresponding to the starting node

Table 5: Components of the adjacency list data structure for storing events on a network.

At the top level, the entire network is accessed as a linked-list whose entries correspond to the nodes of the network. Figure 8 shows the data structure representing the network and events in Figure 6. The first column of the figure shows five linked-list entries corresponding to the five nodes in the network. For this top level linked-list, the `data` member of the ith list entry ($i = 1, \dots, 5$) points to an `Adjlist` object, which holds all the information related to the ith (starting) node v_i and its outgoing edges. The third column in Figure 8 shows the use of `sNode` structures by `Adjlist` structures for storing information about the nodes of the network.

Columns four to six in Figure 8 show the use of the `aNode` structure in representing the adjacent nodes (corresponding to the weighted edges) of the network graph in Figure 6. Due to space limitations, we have omitted `eid` from the `aNode` structure in the figure. The `crashlist` is again a linked-list of `Crash` structures (see Figure 7, right) containing information about each point event on a given edge. The members `seg` and `tp` of the `Crash` structure are described in Table 4.

Time complexity of creating the data structure

Recall that for a network with point events on it, m, n, and p are the numbers of nodes, segments, and events respectively. In order to build the data structure (in Figure 8) for a network, first we insert all the nodes in the top level linked-list. The time complexity of this

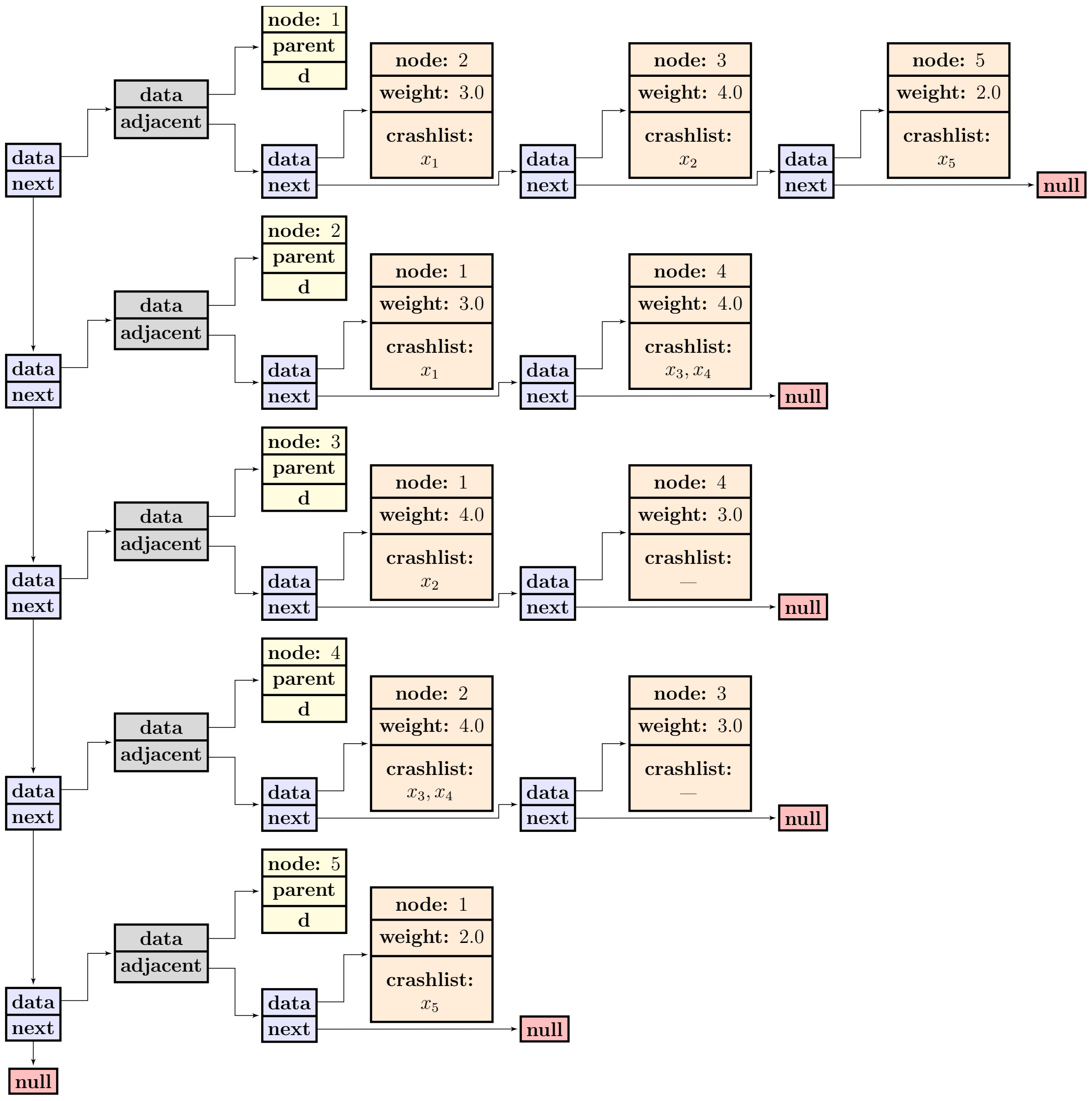


Figure 8: Data structure representing the network and events in Figure 6.

operation is $O(m)$. Then for each segment $s_i, i = 1, \dots, n$, two weighted edges are inserted, so in total we insert $2n$ network edges in the data structure. Every edge in our representation is a line segment of a given length joining a starting node and an adjacent node. For a given segment $s = [v, v']$, we insert the nodes v' and v, respectively, in the adjacency lists of the nodes v and v'. The time complexity of this operation is $O(n)$, and for each segment insertion we spend $O(m)$ time to search the nodes in the top level linked-list. Consequently, it easily follows that the time complexity of inserting all the edges in the network is $O(nm)$.

3. Inter-event distances and the K-function

In this section, we first introduce further terminology related to a linear network and point pattern data, then describe the K-functions introduced by Okabe and Yamada (2001) and

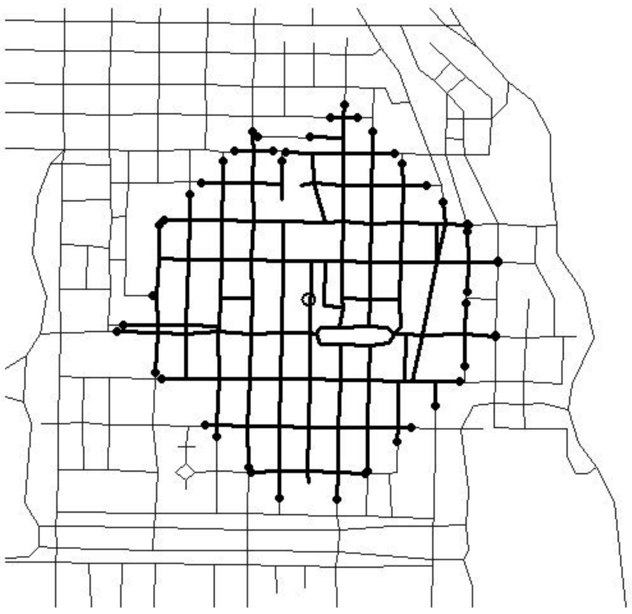


Figure 9: A disc in the shortest-path metric, on the Chicago street network. Open circle: center point. Bold lines: disc of radius 425 feet around the center point. Filled circles: disc endpoints counted by the function $m(u, r)$.

Ang *et al.* (2012) for analyzing such data, and finally provide examples of the current implementation (using adjacency matrix data structure) of these functions in the R package **spatstat** (Baddeley *et al.* 2015).

3.1. Shortest path distance

A *path* between two points w and w' on a linear network L is a sequence $\pi = (w_0, w_1, \ldots, w_N)$ of points joining $w_0 = w$ to $w_N = w'$ such that each line segment $[w_i, w_{i+1}]$ $(i = 0, \ldots, N-1)$ is a subset of some edge of the network. For most purposes $w_1, \ldots, w_{N-1}$ can be taken as nodes of the network. The length of the path is the sum of the step lengths, $\ell(\pi) = \sum_{i=1}^{N} \|w_i - w_{i-1}\|$. The *shortest-path distance* $d_L(w, w')$ between w and w' is the minimum of the lengths of all possible paths from w to w'. If there are no paths from w to w' (implying that the network is not path-connected) then we define $d_L(w, w') = \infty$.

The *disc* of radius $r > 0$, with center $u \in L$, is the set of all points v in the network that lie no more than a distance r from the location u, by the shortest path: $b_L(u, r) = \{v \in L : d_L(u, v) \leq r\}$. Figure 9 gives the street network around the University of Chicago (Ang *et al.* 2012). Bold lines show the disc of radius 425 feet centered at the location marked by the open circle.

The *perimeter* $c_L(u, r) = \{v \in L : d_L(u, v) = r\}$ is the set of points lying exactly r units away from u by the shortest path. The *perimeter count*

$$m(u, r) = \#c_L(u, r) \tag{3}$$

is the number of points on the perimeter $c_L(u, r)$. Points contributing to the count $m(u, r)$ are displayed as filled-circle in Figure 9. Note that a segment of $b_L(u, r)$ may terminate without contributing to $m(u, r)$ if its terminal endpoint lies at a distance less than r from u. Two such cases are visible in Figure 9.

A *subnetwork* $\tilde{L}$ of L is a linear network with a collection of line segments $\tilde{S} \subset S$ and set of nodes $\tilde{V} \subseteq V$. If the network $\tilde{L}$ is path-connected (i.e., if any two points on $\tilde{L}$ can be joined by a path inside $\tilde{L}$), we call it a *connected subnetwork*.

3.2. K-functions on a linear network

A standard tool for the analysis of point patterns in two-dimensional space is the well-known K-function introduced by Ripley (1977). Several counterparts and generalizations of this function have been defined for linear networks (Okabe and Yamada 2001; Boots and Okabe 2007; Ang *et al.* 2012).

Suppose we are given a linear network L with observed events $\mathbf{x} = \{x_1, \ldots, x_p\}$. Okabe and Yamada (2001) defined the (empirical) network K-function as

$$\widehat{K}_{\mathsf{net}}(r) = \frac{|L|}{p(p-1)} \sum_{i=1}^{p} \sum_{j \neq i} \mathbf{I}(d_L(x_i, x_j) \leq r). \tag{4}$$

As explained in Ang *et al.* (2012), a severe drawback of $\widehat{K}_{\mathsf{net}}(r)$ is its dependence on the network geometry, even for a completely random point pattern. This makes it difficult to compare point patterns on different networks.

Ang *et al.* (2012) proposed the following geometrically-corrected empirical network K-function:

$$\widehat{K}_{\mathsf{L}}(r) = \frac{|L|}{p(p-1)} \sum_{i=1}^{p} \sum_{j \neq i} \frac{\mathbf{I}(d_L(x_i, x_j) \leq r)}{m(x_i, d_L(x_i, x_j))}, \tag{5}$$

where the m-function in the denominator, defined in (3), is the weight to compensate for the network geometry. The empirical function $\widehat{K}_{\mathsf{L}}(r)$ is given as an estimator of the theoretical K-function $K_{\mathsf{L}}(r)$ (Ang *et al.* 2012, p. 598) and it is shown that $K_{\mathsf{L}}(r) = r$ $(r > 0)$ for a homogeneous Poisson process. This property can be used to compare any point pattern to a completely random point pattern. It is also permissible to compare K-functions obtained from different networks (Ang *et al.* 2012).

Ang *et al.* (2012) also introduced a version of (5) for inhomogeneous point processes:

$$\widehat{K}^{ih}_{\mathsf{L}}(r) = \frac{1}{\sum_{i=1}^{p} 1/\hat{\lambda}(x_i)} \sum_{i=1}^{p} \sum_{j \neq i} \frac{\mathbf{I}(d_L(x_i, x_j) \leq r)}{\hat{\lambda}(x_i)\hat{\lambda}(x_j) m(x_i, d_L(x_i, x_j))}, \tag{6}$$

where $\hat{\lambda}(\cdot)$ is some estimate of the spatially varying intensity function of the point process on L.

In the analysis of two-dimensional point patterns, it is standard practice to restrict the computation of $K(r)$ to distances r that are less than a specified maximum $r_{\max}$. This is due to the fact that bias and variance increase dramatically with r and that there is usually a maximum expected range of spatial dependence (Baddeley *et al.* 2015, Chapter 7). The same statements are true on a linear network. For example, in the Western Australian accident data, any spatial dependence between accident events is unlikely to extend over more than a few kilometers. Accordingly, we adopt the same practice: our algorithms are designed to evaluate the K-functions (4)–(6) for $0 \leq r \leq r_{\max}$, where $r_{\max}$ is considerably smaller than

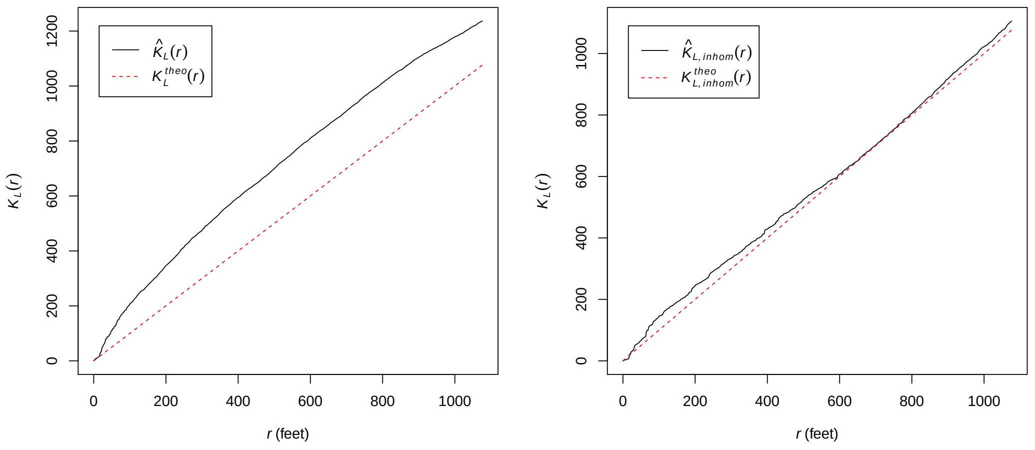


Figure 10: Homogeneous (left) and inhomogeneous (right) geometrically-corrected K-function estimates for the `chicago` crime data. Solid line: empirical estimate of K-function. Dashed line: theoretical K-function for completely random pattern.

$R = \min_{u \in L} \max_{v \in L} d_L(u, v)$, called the *inradius* of the network L. We show in Section 8 that this restriction substantially increases the computational efficiency.

3.3. Existing implementations of K-function

For computing the network K-functions (4)–(6), the R package **spatstat** at present offers the only open source capabilities, to our knowledge. The **spatstat** function `linearK` computes $\widehat{K}_{\mathsf{net}}$ and $\widehat{K}_{\mathsf{L}}$ when the arguments `correction="none"` and `correction="Ang"` are provided, respectively; the default value for the argument `correction` is `"Ang"`. The function `linearKinhom` computes the inhomogeneous K-function $\widehat{K}_{\mathsf{L}}^{ih}$ when the intensity estimates $\hat{\lambda}$ are provided. See Baddeley *et al.* (2015, Chapter 17).

Figure 10 shows the homogeneous and inhomogeneous K-functions computed and plotted by the following R code:

```
R> library("spatstat.Knet")
R> X <- unmark(chicago)
R> plot(linearK(X))
R> fit <- lppm(X ~ polynom(x, y, 2))
R> plot(linearKinhom(X, fit))
```

At the time of writing, linear networks in **spatstat** are represented using adjacency matrices, and a matrix-based algorithm, which we refer to as Algorithm M, is used to compute the network K-functions (4)–(6). This is described in Section 4.

4. Algorithm M, using adjacency matrix

If the data set is small, or computer memory is unlimited, then it is possible to use the adjacency matrix representation of the network (Section 2.2) and a relatively straightforward algorithm can be used to compute second-order summary statistics of the form (1). This approach is followed in the **spatstat** package at the time of writing (**spatstat** versions 1.23-0 to 1.51-0). In this section, we give details of this approach for computing the geometrically-corrected K-function (5). Algorithms for (4) and (6) can be obtained as simple modifications.

4.1. Algorithm M specification

The algorithm is described in four sequential steps (*M1*)–(*M4*).

(M1) Shortest-path distances between nodes

The first step is to compute the matrix of shortest-path distances $d_{ij} = d_L(v_i, v_j)$ between each pair of nodes. Recall that the adjacency matrix (a_{ij}) has entries $a_{ij} = \|v_i - v_j\|$, if v_i and v_j are joined by an edge and $a_{ij} = \infty$, otherwise. The algorithm initializes $d_{ij} = a_{ij}$, then iteratively applies the update

$$d_{ij} := \min\{d_{ik} + d_{kj} : \ v_k \sim v_i\}, \tag{7}$$

where $v \sim v'$ denotes a pair of nodes joined by an edge. This update is similar to the 'relaxation' step used in the famous Dijkstra shortest-path algorithm (Gallo and Pallottino 1988). This iterative procedure finishes after a finite time, giving the matrix (d_{ij}). Careful coding is needed to avoid numerical error associated with floating-point comparisons, which could otherwise cause the iterations to continue indefinitely.

(M2) Shortest-path distances between events

Next, for each pair (x_i, x_j) of events, the algorithm computes the shortest-path distance $h_{ij} = d_L(x_i, x_j)$ as follows. We identify the segments s_i and s_j containing x_i and x_j, respectively. If $s_i = s_j$ then $h_{ij} = \|x_i - x_j\|$. Otherwise, we find the endpoints v, v' of s_i and the endpoints w, w' of s_j, and then compute

$$h_{ij} = \min\{h_{vw}, h_{vw'}, h_{v'w}, h_{v'w'}\}, \tag{8}$$

where

$$\begin{aligned}
h_{vw} &= \|x_i - v\| + d_L(v, w) + \|w - x_j\|, \\
h_{vw'} &= \|x_i - v\| + d_L(v, w') + \|w' - x_j\|, \\
h_{v'w} &= \|x_i - v'\| + d_L(v', w) + \|w - x_j\|, \quad \text{and} \\
h_{v'w'} &= \|x_i - v'\| + d_L(v', w') + \|w' - x_j\|
\end{aligned}$$

are the path lengths of the shortest paths from x_i to x_j passing through the specified end-points.

(M3) Counting points on the disc perimeter

The shortest-path distance matrix h_{ij} $(i, j = 1, \dots, p)$ described above provides the necessary data for the network K-function (4). The geometrically corrected function (5) requires additional computation of the weighting factors $m_{ij} = m(x_i, h_{ij})$ for each pair (i, j). For any arbitrary point u on the network and a given distance r, the steps to compute $m(u, r)$ are as follows:

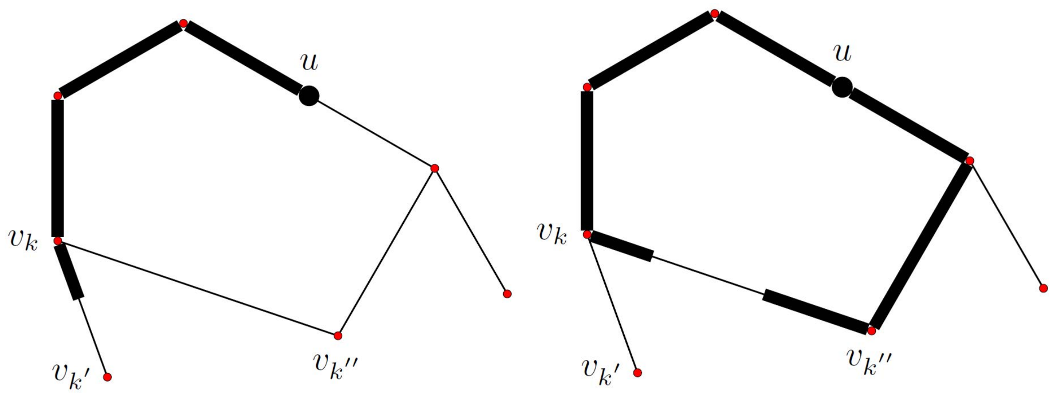


Figure 11: The bold lines give part of the disc $b_L(u, r)$ relevant to scenarios 3(a) (left) and 3(b) (right) of step *M3* for counting perimeter points.

1. Find the endpoints v, v' of the segment containing u.

2. For each node v_k compute the shortest-path distance from u,

$$t_k = d_L(u, v_k) = \min\{\|u - v\| + d_L(v, v_k),\ \|u - v'\| + d_L(v', v_k)\}.$$

   Since v and v' are nodes of the network, the values $d_L(v, v_k)$ and $d_L(v', v_k)$ can readily be extracted from the shortest-path distance matrix (d_{ij}).

3. For all nodes v_k for which $t_k \leq r$, i.e., for all nodes lying inside $b_L(u, r)$, consider all adjacent line segments of v_k of the form $[v_k, v_{k'}]$, $[v_k, v_{k''}]$, etc. These are all the line segments that may contain a point of $c_L(u, r)$. Now, any one of the following two scenarios may arise while investigating the segment $[v_k, v_{k'}]$:

   (a) if $d_L(u, v_{k'}) = t_{k'} \geq r$ then $v_{k'}$ lies outside $b_L(u, r)$. Hence $[v_k, v_{k'}]$ crosses the perimeter of the disc, and contains one perimeter point. This is illustrated in the left panel of Figure 11. The bold line in the figure gives a part of $b_L(u, r)$ where $t_k < r$, $v_k \sim v_{k'}$, and $t_{k'} > r$. This gives one perimeter point on $[v_k, v_{k'}]$.

   (b) if $t_{k'} < r$ then $v_{k'}$ lies inside $b_L(u, r)$. Consider $c = \|v_k - v_{k'}\| - (r - t_k) - (r - t_{k'})$. Then $[v_k, v_{k'}]$ contains 0, 1 or 2 perimeter points according as $c < 0$, $c = 0$ or $c > 0$ respectively. This is illustrated in the right panel of Figure 11. The bold line in the figure gives a part of $b_L(u, r)$ where $t_k < r$, $v_k \sim v_{k''}$, $t_{k''} < r$. In this case $c > 0$, giving two perimeter points on $[v_k, v_{k''}]$.

(M4) Computation of K-functions

The final step is to compute the Okabe-Yamada network K-function $\widehat{K}_{\mathsf{net}}$ defined in (4) or the geometrically corrected K-function $\widehat{K}_{\mathsf{L}}$ defined in (5). For $\widehat{K}_{\mathsf{net}}$, we simply compute all the pairwise shortest-path distances h_{ij}, form a histogram, compute the cumulative distribution function of the distances, and renormalize to obtain (4). For $\widehat{K}_{\mathsf{L}}$, we compute pairs (h, w) of distances $h = h_{ij}$ and corresponding weights $w = 1/m(x_i, h_{ij})$, compute the *weighted* histogram, cumulate and renormalize to obtain (5).

4.2. Implementation in spatstat

Algorithm M was implemented in the R package **spatstat**. The initial implementation of steps *M1–M4* was coded in the R language using the basic facilities for matrix operations. This is very convenient for testing and cross-testing purposes, but is slow and memory-hungry. The algorithm was later accelerated by re-coding the update (step *M1*) and the perimeter-counting rule (step *M3*) in C to achieve the speeds reported in Table 1.

At the time of writing, **spatstat** represents a linear network using an adjacency matrix which may be either sparse or full (using the **Matrix** package of Bates and Maechler (2019) for sparse matrices). The sparse representation is memory efficient, even for the Western Australian data. However, the Algorithm M requires computation of the *full* matrix (d_{ij}) of distances between all pairs of nodes. Hence the fundamental limitation of this algorithm is the $O(m^2)$ storage requirement.

In Section 5 and 6, we develop a new algorithm for computing summary statistics of the general form (1), in particular (4)–(6).

5. Tree structures for the new algorithm

As explained in Section 4, Algorithm M computes the shortest-path distances between events by first computing the matrix (d_{ij}) of pairwise distances between nodes. The new algorithm, which we refer to as Algorithm L, avoids this memory-intensive task of computing the distance matrix (d_{ij}); instead, it computes the shortest-path distances from x_i $(i = 1, \ldots, p)$ to other events in the network by performing a network search starting at the source point x_i. Searching a graph is a standard procedure, which is generally accomplished by constructing a *tree* from the source point, taking it as the root node (Cormen *et al.* 2009). In this section, we describe some important tree structures used in developing Algorithm L. A pseudocode for the Algorithm L is presented in the Section 6.

The adjacency list algorithm divides the problem of computing a network statistic of the general form (1) into sub-problems. In order to develop this idea for the network K-function $\widehat{K_{\mathsf{L}}}$ defined in (5), we use the concept of the *local K-function* (Getis and Franklin 1987; Anselin 1995, Baddeley *et al.* 2015, Chapter 7) adapted to linear networks. The network K-function can be decomposed as

$$\widehat{K_{\mathsf{L}}}(r) = \frac{1}{p} \sum_{i=1}^{p} \widehat{K_{\mathsf{L}}}(x_i, r), \tag{9}$$

where

$$\widehat{K_{\mathsf{L}}}(x_i, r) = \frac{|L|}{(p-1)} \sum_{j \neq i} \frac{\mathbf{I}(d_L(x_i, x_j) \leq r)}{m(x_i, d_L(x_i, x_j))} \qquad (i = 1, \ldots, p) \tag{10}$$

are the local K-functions. Hence the computation of $\widehat{K_{\mathsf{L}}}$ reduces to the computation of p local K-functions.

Here we introduce some important graphical structures for the new algorithm. For the computation of local K-functions in (10), we need the notions of "breakpoint", "shortest-path tree", and "extended shortest-path tree" introduced by Okabe and Yamada (2001). The next three subsections give, respectively, a method of constructing an extended shortest-path tree (using a shortest-path tree and breakpoints) based on a subnetwork, a procedure for con-

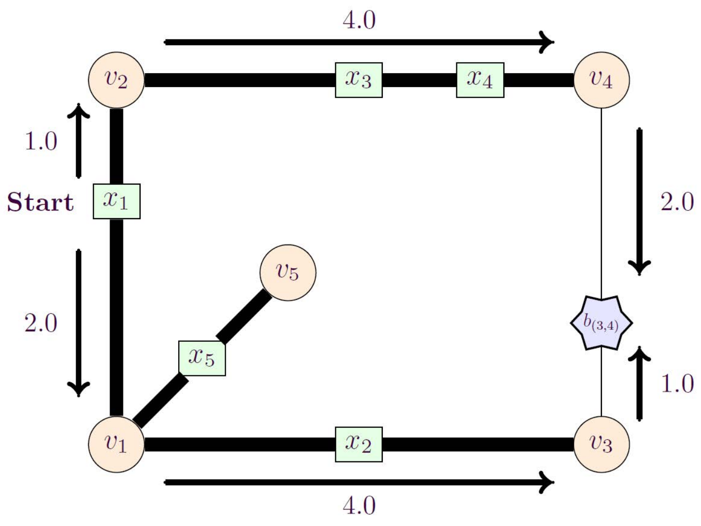


Figure 12: Shortest path tree (solid black line) starting from crash point x_1 in Figure 6. Six-pointed star represents a breakpoint. Arrows indicate the two different paths of equal length from x_1 to the breakpoint $b_{(3,4)}$; arrow direction shows the travel path.

structing such a subnetwork from the original network, and the time complexity analysis of this procedure.

5.1. Shortest-path tree, breakpoints, and extended shortest-path tree

For a given starting point $u \in L$, a *breakpoint* (corresponding to u) is defined as a point $v \in L$ for which the shortest path from u to v is not unique, i.e., there exist two different paths from u to v which achieve the minimum possible path length. There can only be finitely many breakpoints for any starting point $u \in L$.

Let L_u^* be the subset of L formed by the union of all segments which do not contain a breakpoint corresponding to u. Then L_u^* is a directed weighted graph without any loops, and is equivalent to a tree rooted at u, called the *shortest-path tree.* A data structure representing the shortest-path tree L_u^* can be built by first including all the nodes in the network along with the additional node representing the root u of the tree and then recursively adding adjacent edges to an adjacency-list data structure.

Figure 12 gives a topologically equivalent representation of the network in Figure 6, and the bold lines in it represent the shortest-path tree rooted at the point event x_1. The segment $[v_3, v_4]$ contains the breakpoint $b_{(3,4)}$, so this segment is not included in the shortest-path tree.

When there is a breakpoint corresponding to some $u \in L$, an *extended shortest-path tree* can be constructed as follows. For each segment $s = [v_i, v_j]$ that contains a breakpoint $b_{(i,j)}$, two new nodes $b'_{(i,j)}$, $b''_{(i,j)}$ are created with the same spatial coordinates as $b_{(i,j)}$ but are treated as distinct nodes. Then the two weighted edges corresponding to s are replaced by the two new edges $(v_i, b'_{(i,j)}, \|v_i - b'_{(i,j)}\|)$ and $(v_j, b''_{(i,j)}, \|v_j - b''_{(i,j)}\|)$, which are treated as having no common intersection. These edges are added to the shortest-path tree L_u^* by inserting $b'_{(i,j)}$ and $b''_{(i,j)}$ in the adjacency lists of v_i and v_j, respectively. Continuing the process for all breakpoints of u, the final tree, denoted as L_u^{**}, thus obtained is called the extended shortest-path tree

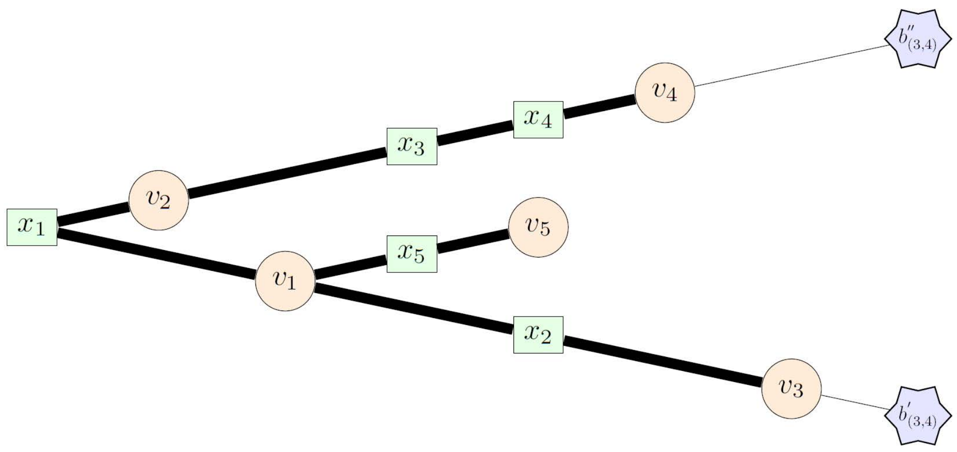


Figure 13: Extended shortest-path tree from point event x_1 in Figure 6. Six-pointed stars represent terminal nodes, which are duplicates of the breakpoints. Solid black lines represent the shortest-path tree in Figure 12.

rooted at u. The extended shortest-path tree is a one-to-one representation of all minimal paths starting from u in L.

Figure 13 shows the extended shortest-path tree corresponding to Figure 12. The segment $[v_3, v_4]$ has been replaced by the edges $(v_3, b'_{(3,4)}, \|v_3 - b'_{(3,4)}\|)$ and $(v_4, b''_{(3,4)}, \|v_4 - b''_{(3,4)}\|)$.

To compute the local K-functions (10), first an extended shortest-path tree, $L^{**}_{x_i}$, needs to be constructed from x_i for $i = 1, \ldots, p$. Then, $\widehat{K_{\mathsf{L}}}(x_i, r)$ can be computed for any $r \le t_{x_i}(L^{**}_{x_i})$, where for a connected subnetwork $\tilde{L}(\subset L)$ and $u \in \tilde{L}$, $t_u(\tilde{L})$ is defined by

$$t_u(\tilde{L}) = \max\{d_L(u, v) : v \in \tilde{L}\}. \tag{11}$$

Note that $t_{x_i}(L^{**}_{x_i})$ (denoted by $t^*_{x_i}$ hereafter) is the maximum possible shortest-path distance from x_i to any point on $L^{**}_{x_i}$. However, as discussed in Section 3.2, in the case of a large network, the computation of $\widehat{K_{\mathsf{L}}}(x_i, r)$ is restricted to distances $r < r_{\max}$, for a prespecified value $r_{\max} < \min\{t^*_{x_1}, \ldots, t^*_{x_p}\}$.

5.2. Local subnetwork

Since the local K-functions $\widehat{K_{\mathsf{L}}}(x_i, r)$ $(i = 1, \ldots, p)$ are restricted to $r \in [0, r_{\max})$, we do not require to construct the extended shortest-path trees based on the entire network L. Here we note a straightforward but important fact that $|L^{**}_{x_i}| = |b_L(x_i, t^*_{x_i})|$. Hence, for the computation of the local K-function $\widehat{K_{\mathsf{L}}}(x_i, r)$ $(0 \le r < r_{\max})$, it is sufficient to consider an extended shortest-path tree based on any connected subnetwork $\tilde{L}$ such that $b_L(x_i, r_{\max}) \subset \tilde{L}$.

Accordingly, for each $i = 1, \ldots, p$, we construct a *local connected subnetwork* $L(x_i, r_{\max})$ corresponding to x_i as follows. First, we insert all the nodes that are within a distance $r_{\max}$ from x_i by the shortest path. Next, we insert all the edges connected to these nodes and all the nodes corresponding to the endpoints of these edges (if not already inserted). It can be shown that $L(x_i, r_{\max})$ is the smallest connected subnetwork of L with the property $b_L(x_i, r_{\max}) \subset L(x_i, r_{\max})$.

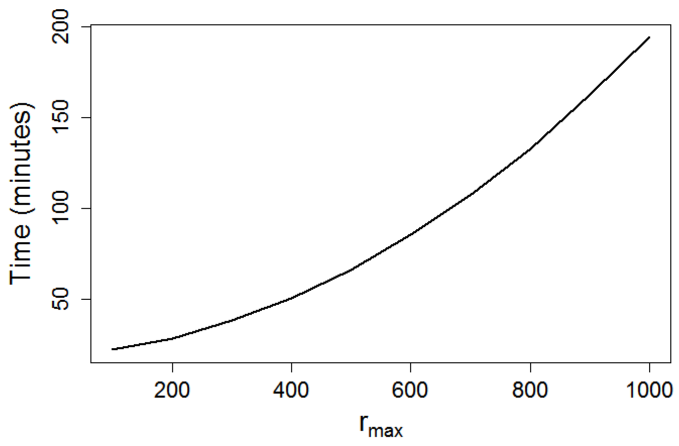


Figure 14: Execution time (in minutes) of Algorithm L for computing $\widehat{K}_{\mathsf{L}}(r)$ for the 2011 accident data on the Western Australian road network with different $r_{\max}$ values (in meters).

The use of these local subnetworks in the construction of the extended shortest-path trees greatly reduces the computing time for the K-function. This is evident in Figure 14 which plots the computation time of the K-function $\widehat{K}_{\mathsf{L}}(r)$ against different $r_{\max}$ values for the Western Australian network data.

5.3. Procedure for constructing local subnetwork

The following algorithm to construct $L(x, r_{\max})$ corresponding to a point event $x \in \mathbf{x}$ is a modification of Dijkstra's shortest-path algorithm (Gallo and Pallottino 1988).

Procedure 1 (Local subnetwork construction)**.**

Input data: *Network-graph L and the point event x.*

1. *Identify the segment $[v_1, v_2]$ that contains x.*
2. *Insert x as the $(m+1)$th node in L, where m is the number of nodes in L.*
3. *Delete both weighted edges corresponding to the segment $[v_1, v_2]$.*
4. *Insert two new edges $(x, v_1, \|x - v_1\|)$ and $(x, v_2, \|x - v_2\|)$ in L.*
5. *Create an empty graph $L(x)$.*
6. *Assign a distance value to* `d` *in the* `sNode` *structure of every node in L: set it to zero for x and infinity for all other nodes. Label all the nodes with color white signifying they are all unvisited. Assign the* `parent` *member of every node a label equal to* `NULL`*.*
7. *Select the white node with the minimum distance* `d`*. If there is no white node, go to Step* 13.
8. *If the minimum distance is greater than r_{max}, go to Step* 13.

9. *Change the unvisited status of the selected node by changing its color label to black. The corresponding value of* `d` *is the shortest-path distance from x to this node.*

10. *Insert the black node and all its adjacent edges in $L(x)$.*

11. *Suppose that the most recently visited node is v. Then the reason we are in this step is because $d_L(x, v) < r_{max}$. Let v_{adj} denote the adjacent node under consideration.*

    (a) If the current distance value `d` *of v_{adj} (from x) is smaller than $d_L(x, v)+d_L(v, v_{adj})$, then do nothing and move to the next adjacent node if one exists.*

    (b) If there are no adjacent nodes left to be scanned, move to Step 12.

    (c) On the other hand, if `d` *corresponding to v_{adj} is greater than $d_L(x, v) + d_L(v, v_{adj})$, set* `d` $= d_L(x, v) + d_L(v, v_{adj})$*, and assign node v to the* `parent` *component of v_{adj}.*

12. *Repeat Steps 7 to 11.*

13. *Restore L to its original form after it was changed in Steps 2–4.*

When Procedure 1 terminates, the empty graph $L(x)$ becomes $L(x, r_{\max})$, and every node that are within $r_{\max}$ distance from x in this subnetwork stores its shortest-path distance from x and reference to its parent node in the shortest-path route. The shortest-path distances from x to these nodes in $L(x, r_{\max})$ play a very important role in the computation of the m-function (3), the denominator in (10).

5.4. Time complexity in computing local subnetwork

Let m_x denote the number of nodes that are within a distance $r_{\max}$ from x by shortest path and n_x denote the number of edges in $L(x, r_{\max})$. At the center of Procedure 1 is a single conditional loop (Steps *7–11*) that iterates m_x times, once for each of the m_x nodes. The conditional loop is originally set to iterate over all m nodes in the network L with the condition given in Step *8* of the Procedure 1 to exit the loop. Because the nodes that we encounter after first m_x iterations all have shortest-path distances (from x) more than $r_{\max}$, the loop terminates after m_x iterations.

Each iteration starts by selecting the node with the smallest shortest-path estimate among the nodes labeled as white, and the node is selected by traversing through m nodes in L and checking their distance estimates. This part of the iteration is $O(mm_x)$.

Next, we visit the nodes adjacent to the selected node. As we visit each adjacent node, we update the distance estimates `d` and the parent label `parent` of the adjacent node (Step *11*). The update process for an adjacent node v_{adj} of v requires the distance estimate corresponding to v_{adj}. It is obtained by going through m nodes in the node-list. For all the m_x nodes, we go through the node-list n_x times, once for each of the edges in $L(x, r_{\max})$. Consequently, this part of the iteration is $O(mn_x)$. Therefore, the main conditional loop overall is $O(m(m_x+n_x))$.

6. Algorithm L, using adjacency list

In this section, we describe Algorithm L for computing the geometrically-corrected K-function (5) based on the adjacency list structure in Figure 8 and the concepts developed in Section 5.

Implementations for (4) and (6) are simple modifications of this algorithm. One feature of our algorithm is that it computes $\widehat{K}_{\mathsf{L}}(r)$ on a finite grid of distance values $0 \le r_1, \dots, r_l \le r_{\max}$ without additional computational cost. This efficiency is achieved by computing *interval sums* (defined in (12)), which are related to the local K-functions, for every point event in the network; details are in Section 6.1.

After constructing the extended shortest-path tree from the root node x_i, two remaining steps in the computation of $\widehat{K}_{\mathsf{L}}(r)$ are:

(1) computing the weights $1/m(x_i, d_L(x_i, x_j))$ corresponding to the neighboring point events x_j of x_i, and

(2) computing the interval sums for a specified grid of distance values by performing a search from the root node x_i.

Section 6.2 outlines the computational details of $m(x, r)$, required in (1) for $x \in \mathbf{x}$, and Section 6.3 provides a depth-first search algorithm (Cormen *et al.* 2009; Even 1979; Tarjan 1983) for computing the interval sums in (2). The pseudocode for Algorithm L is given in Section 6.4.

6.1. Interval sums

Although $\widehat{K}_{\mathsf{L}}(r)$ is a function of a continuous argument r, in practice we compute it on a finite grid with stepsize ε, obtaining $\widehat{K}_{\mathsf{L}}(r_j)$, where $r_j = j\varepsilon$, for $j = 0, \dots, l$; $l = \lfloor r_{\max}/\varepsilon \rfloor$ is the largest integer less than or equal to $r_{\max}/\varepsilon$.

For $x \in \mathbf{x}$ and $j = 1, \dots, l$, let

$$I_j(x) = \sum_{y \in \mathbf{x} \setminus \{x\}} \frac{\mathbf{I}(r_{j-1} < d_L(x, y) \le r_j)}{m(x, d_L(x, y))} \qquad (12)$$

be the interval sums corresponding to x. In what follows we assume $I_0(x) = 0$ for all $x \in \mathbf{x}$, corresponding to locations without multiple events. Then the local K-function and the K-function can be expressed as

$$\frac{(p-1)\widehat{K}_{\mathsf{L}}(x, r_i)}{|L|} = \sum_{j=0}^{i} I_j(x) \qquad (13)$$

$$\frac{p(p-1)\widehat{K}_{\mathsf{L}}(r_i)}{|L|} = \sum_{x \in \mathbf{x}} \sum_{j=0}^{i} I_j(x) \qquad (14)$$

for $i = 1, \dots, l$.

An alternative way of expressing the K-function, by interchanging the sums in (14), is

$$\frac{p(p-1)\widehat{K}_{\mathsf{L}}(r_i)}{|L|} = \sum_{j=0}^{i} \sum_{x \in \mathbf{x}} I_j(x) \quad \text{for} \quad i = 1, \dots, l. \qquad (15)$$

An intuitive way to compute the K-function is to use (14) by first computing the local K-functions using (13). However, it is more efficient to do this by summing the computed values

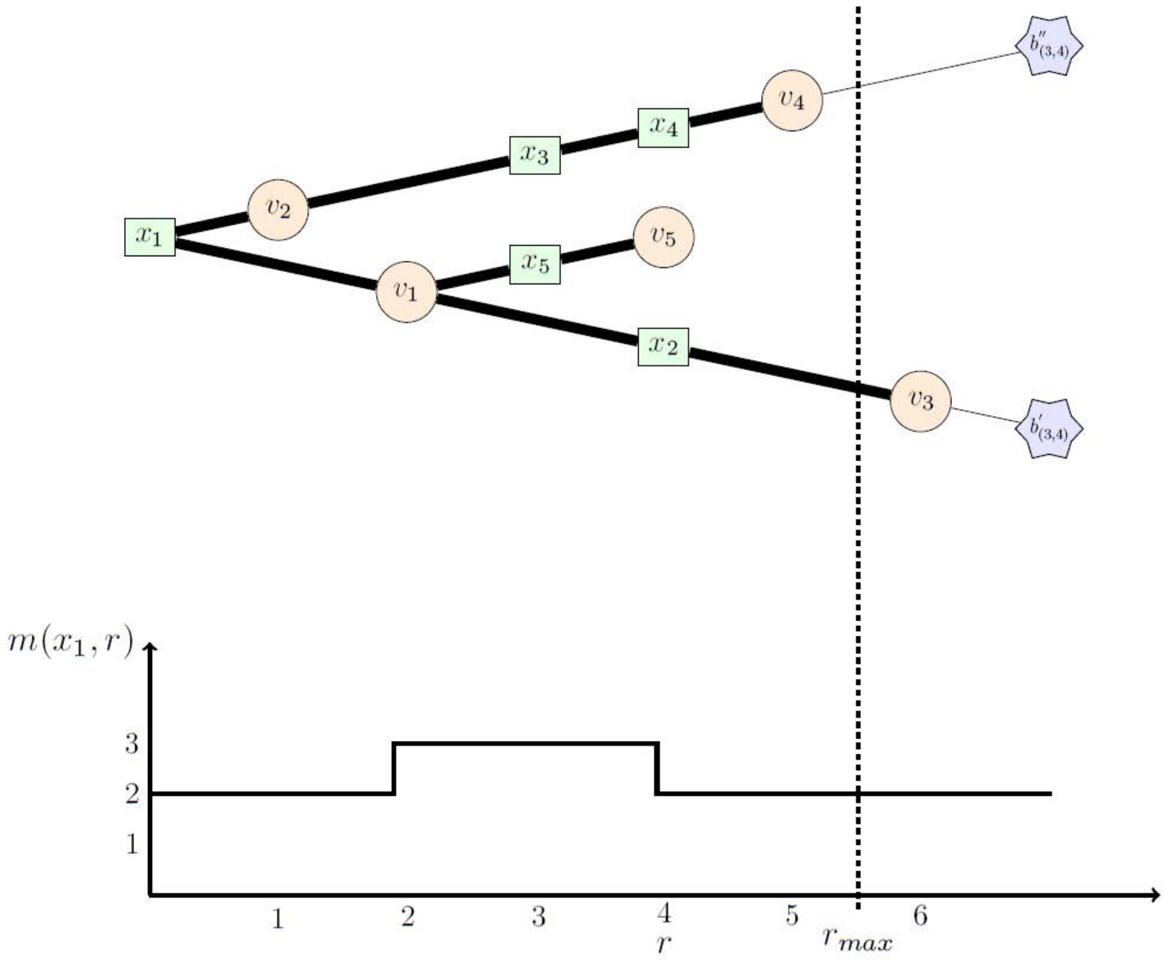


Figure 15: Top: the extended shortest-path tree from point event x_1 in Figure 6, restricted to $r_{\max} = 5.5$. Below: the function $m(x_1, r)$ for computing local K-function of x_1.

of $\sum_{x \in \mathbf{x}} I_j(x)$ and using (15). We first compute the function $m(x, r)$ and then compute $I_j(x)$ using a depth-first search algorithm.

6.2. Computation of perimeter count $m(x, r)$

We start with some notation used in the rest of this section. Let $L_x^{**}(r_{\max})$ denote the extended shortest-path tree that is constructed from the subnetwork $L(x, r_{\max})$ defined in Section 5.2. Let $V_x(r_{\max})$ denote the set of nodes that are within $r_{\max}$ distance from x by shortest path. Let V_x^* denote the set of all nodes in $L_x^{**}(r_{\max})$, except the nodes that do not have any adjacent node in $V_x(r_{\max})$.

It is easy to verify that $V_x(r_{\max}) \subset V_x^*$, and for any $v \in V_x^*$, $d_L(x, v) > r_{\max}$ if and only if $v \sim v'$ for some $v' \in V_x(r_{\max})$. Note that, V_x^* may not include all breakpoints in $L_x^{**}(r_{\max})$, e.g., in Figure 15 the extended shortest path tree $L_{x_1}^{**}(r_{\max})$, for a given $r_{\max} = 5.5$, does not include the breakpoint $b'_{(3,4)}$.

To compute $m(x, r)$, we first order the shortest-path distances based on the non-decreasing values of $d_L(x, v)$, $v \in V_x^*$. The sorted list of distances is given by

$$D_x = \{d_{[1]}, \dots, d_{[m_x^*]}\}, \text{ with } m_x^* = |V_x^*|, \tag{16}$$

corresponding to the list of nodes $\{v_{[1]}, \dots, v_{[m_x^*]}\}$. Note that we shall always have $v_{[1]} = x$ and $d_{[1]} = 0$. Let $\delta_{[j]}$ denote the degree of the node $v_{[j]}$, for $j = 1, \dots, m_x^*$. Then, for a given r determine j such that $d_{[j]} \le r < d_{[j+1]}$, and compute $m(x, r)$ as

$$m(x, r) = \delta_{[1]} + \sum_{k=2}^{j} (\delta_{[k]} - 2). \tag{17}$$

The m-function is a step-function with possible jumps at distinct $d_{[j]}$ values. If there is a tie, e.g., $d_{[j]} = d_{[j+1]}$, the value of the m-function remains unchanged between $d_{[j-1]}$ and $d_{[j+1]}$. More generally, if $d_{[j-1]} < d_{[j]} = \cdots = d_{[j+k]} < d_{[j+k+1]}$ for some $j \ge 2$ and $k \ge 2$, we have

$$m(x, r) = \delta_{[1]} + \sum_{k=2}^{j+k-1} (\delta_{[k]} - 2) \quad \text{for } d_{[j-1]} \le r < d_{[j+k]}.$$

The m-function $m(x, r)$ is stored in the computer memory using two arrays `dval` and `mval` of equal size M_x, where M_x is the number of distinct values (except $d_{[1]} = 0$) in D_x. In general, $M_x \le m_x^*$, with equality holding when there are no ties in D_x. For $j = 2, \dots, M_x$,

$\texttt{dval}[j] = j$th smallest value amongst the distinct values in D_x;

$\texttt{mval}[j] = m(x, r)$ if $\texttt{dval}[j-1] \le r < \texttt{dval}[j]$.

Therefore, for a given r, we have,

$$m(x, r) = \begin{cases} 2, & \text{if } r < \texttt{dval}[1]; \\ \texttt{mval}[j], & \text{if } \texttt{dval}[j-1] \le r < \texttt{dval}[j]; \\ \texttt{mval}[M_x], & \text{if } r \ge \texttt{dval}[M_x]. \end{cases} \tag{18}$$

6.3. Depth-first search algorithm

Here we present a variant of the classical depth-first search algorithm (Tarzan 1972) for computing the interval sums $I_j(x)$ $(j = 1, \dots, l)$ in (12) for a given $x \in \mathbf{x}$. When searching a tree, the most recently visited node, say v, is called the *current node*, and its adjacent nodes away from the root node are the *children nodes*, with v as their *parent node*.

A depth-first search begins at the root node x, taking it as the current node, in the extended shortest-path tree $L_x^{**}(r_{\max})$. The search then explores an unvisited outgoing edge (a child node) of the current node and then updates that child node as the current node. When there are no unvisited outgoing edge from the current node, the search backtracks to the parent node, thereby updating the parent node as the current node. The search finishes when there are no more unvisited outgoing edges from x. In our variation, we backtrack from a node, without exploring outgoing edges from the node, if the shortest-path distance from x to that node is greater than $r_{\max}$ and the shortest distance to its parent node from x is less than $r_{\max}$.

We have implemented the depth-first search algorithm in a recursive C function. To give an overview of our implementation, below we provide a pseudofunction `intervalSum` outlining the steps involved in computing the interval sums (12). The pseudofunction has six arguments, which are described in Table 6.

Procedure 2 (Depth-first search)**.** `intervalSum(G, D, M, x, dist, K, r)`

```
1. Find the node x in G.
2. Check if there exists any adjacent node xAdj of x.
    (a) IF (xAdj == NULL), then EXIT.
    (b) WHILE (xAdj ≠ NULL)
        i. IF (xAdj.weight > dist)
            A. crashList = xAdj.crashlist
            B. Assign to nCrash the number of Crash objects in the crashList that are
               less than dist distance away from x
            C. FOR($i = 1; i \leq$ nCrash; $++i$){
                • crash = crashList[$i$]
                • Compute the shortest-path distance from the root of the tree to the crash:
                  rootToCrashDistance = xAdj.d + xAdj.weight * crash.tp
                • Find the m-value corresponding to rootToCrashDistance using D and
                  M using (18) in Section 6.2, and assign it to mValue.
                • Find $j$ such that r[$j-1$] < rootToCrashDistance ≤ r[$j$].
                • Perform K[$j$] = K[$j$] + 1/mValue } END FOR LOOP
        ii. ELSE
            A. Compute remainingDistance = dist − xAdj.weight
            B. crashList = xAdj.crashlist
            C. Assign to nCrash the total number of Crash objects in crashList.
            D. FOR($i = 1; i \leq$ nCrash; $++i$){
                • crash = crashList[$i$]
                • rootToCrashDistance = xAdj.d + xAdj.weight * crash.tp
                • Find the m-value corresponding to rootToCrashDistance and assign it
                  to mValue.
                • Find $j$ such that r[$j-1$] < rootToCrashDistance ≤ r[$j$].
                • Perform K[$j$] = K[$j$] + 1/ mValue} END FOR LOOP
            E. intervalSum(G, D, M, xAdj, remainingDistance, K, r)
        iii. Go to the next adjacent node, and assign it to xAdj.
    (c) END WHILE LOOP
3. EXIT
```

The function `intervalSum` should be executed with arguments `dist` $= r_{\max}$ and `K` equal to **0**, an l-length array of zeros. The recursive calls of `intervalSum` update the array `K` upon

Argument	Description
`x`	A node identifier for node x whose outgoing edges are to be explored.
`G`	The extended shortest-path tree $L_x^{**}(r_{\max})$.
`D`	The array `dval` of the distance values.
`M`	The array `mval` for accessing the m-values.
`dist`	Numeric value representing the distance required to be explored from node x on the tree $L_x^{**}(r_{\max})$.
`K`	An array of size l for storing the interval sums $I_j(\mathtt{x})$ for $j = 1, \dots, l$.
`r`	Vector of distances $(r_1, \dots, r_l)$.

Table 6: The function arguments for `intervalSum` and their descriptions.

every call, and at the end when `intervalSum` terminates, the jth entry of the array `K` contains $I_j(x)$, for $j = 0, \dots, l$.

6.4. Algorithm L based on an adjacency list representation

Here we present a pseudocode of our adjacency list algorithm for computing the network K-function $\widehat{K}_{\mathsf{L}}(r)$.

Algorithm 1 (L)**.** ***Input data:*** *Network-graph L, vector of distances $\mathbf{r}$=$(r_1, \dots, r_l)$.*

```
1. Assign Isum an array of size $l$ with all entries equal to zero for storing the interval sums.
2. FOR($i$ = 1; $i \le p$; ++$i$){
     (a) Construct $L(x_i, r_{max})$ corresponding to the point event $x_i$ using Procedure 1.
     (b) Construct the extended shortest-path tree $L_{x_i}^{**}(r_{max})$.
     (c) Compute $m(x_i, r)$ using (17) and store it using two arrays dval and mval.
     (d) Call intervalSum(G = $L_{x_i}^{**}(r_{max})$, D = dval, M = mval, x = $x_i$, dist = $r_{max}$,
         K=Isum, r = $\mathbf{r}$).
   } END FOR LOOP
3. The array Isum now contains all the information for computing the K-function $\widehat{K}_{\mathsf{L}}(r)$
   for $r = r_1, \dots, r_l$. Assign Kval an empty array of size $l$.
4. FOR($j$ = 1; $j \le l$; ++$j$){ Kval[$j$] = $\{p(p-1)\}^{-1}|L| \sum_{s=1}^{j}$ Isum[$s$] }
5. RETURN Kval.
```

The most important step in Algorithm L is the computation of the interval sums (12). In Section 7, we demonstrate the computation of this part of the algorithm with an example.

7. Worked example of an interval sum

Here we calculate the interval sums (12) for the point event x_1 in Figure 6. To do this, we first compute the m-function $m(x_1, r)$ and then describe the steps in the search procedure `intervalSum` using the extended shortest-path tree shown in Figure 15.

The extended shortest-path tree is constructed for a given $r_{\max} = 5.5$. For this example, $m^*_{x_1} = 7$, and the ordered distances are

$$d_{[1]} = 0, d_{[2]} = 1.0, d_{[3]} = 2.0, d_{[4]} = 4.0, d_{[5]} = 5.0, d_{[6]} = 6.0, d_{[7]} = 7.0,$$

and the nodes corresponding to these ordered distances are

$$v_{[1]} = x_1, v_{[2]} = v_2, v_{[3]} = v_1, v_{[4]} = v_5, v_{[5]} = v_4, v_{[6]} = v_3, v_{[7]} = b''_{3,4}.$$

The graph in Figure 15 gives a plot of the step-function corresponding to $m(x_1, r)$, whose computation is explained below using (17).

$$m(x_1, r) = \begin{cases} \delta_{[1]} = 2, & \text{if } d_{[1]} = 0 < r < d_{[2]} = 1.0; \\ \delta_{[1]} + (\delta_{[2]} - 2) = 2, & \text{if } d_{[2]} = 1.0 < r < d_{[3]} = 2.0; \\ \delta_{[1]} + (\delta_{[2]} - 2) + (\delta_{[3]} - 2) = 3, & \text{if } d_{[3]} = 2.0 < r < d_{[4]} = 4.0; \\ \delta_{[1]} + (\delta_{[2]} - 2) + (\delta_{[3]} - 2) + (\delta_{[4]} - 2) = 2, & \text{if } d_{[4]} = 4.0 < r < d_{[5]} = 5.0; \\ 2, & \text{if } d_{[5]} = 5.0 < r < r_{\max} = 5.5. \end{cases}$$

Let $r_1 = 0.5, r_2 = 1.0, \ldots, r_{10} = 5.0$ be the values of r for which the interval sums will be computed. Accordingly, we set the initial value for `Isum` as the zero vector of length 10. Then, the depth-first search routine as explained in Procedure 2 works as follows.

1. The search begins at root node x_1, visits v_2, one of the x_1's adjacent nodes, and then checks for any point events on the edge $[x_1, v_2]$. Since the edge is empty, the search visits v_4, one of the unexplored adjacent nodes of v_2.

2. The edge $[v_2, v_4]$ holds two point events x_3 and x_4. The algorithm then computes $d_L(x_1, x_3) = 3.0$, $d_L(x_1, x_4) = 4.0$, and the corresponding m-values $m(x_1, 3.0) = 3$ and $m(x_1, 4.0) = 3$.

3. Because $r_5 < d_L(x_1, x_3) \leq r_6$ and $r_7 < d_L(x_1, x_4) \leq r_8$, the algorithm assigns `Isum`[6] $= 1/m(x_1, 3.0) = 0.333$ and `Isum`[8] $= 1/m(x_1, 4.0) = 0.333$, respectively.

4. The search now proceeds toward the adjacent node of v_4, and it stops at the vertical line signifying $r_{\max}$ in Figure 15. Because the line segment $[v_4, r_{\max}]$ contains no point events and v_4 does not have any unexplored adjacent node, the search track backs to v_2.

5. Because all the adjacent nodes of v_2 have already been explored, the search now track backs to x_1.

6. The search now visits v_1, the last unvisited node in the adjacency list of x_1. Since the edge $[x_1, v_1]$ is empty, the search next visits v_5, one of the unexplored adjacent nodes of v_1.

7. The edge $[v_1, v_5]$ contains a point event x_5. The algorithm computes $d_L(x_1, x_5) = 3.0$ and the corresponding m-value $m(x_1, 3.0) = 3$. Subsequently, the algorithm performs `Isum`[6] = `Isum`[6] $+1/m(x_1, 3.0) = 0.666$.

8. Since the adjacency list of v_5 is empty, the search track backs to v_1.

9. The search now proceeds toward v_3, the last unvisited adjacent node of v_1, and it stops at $r_{\max}$. The line segment $[v_1, r_{\max}]$ contains a point event x_2. The algorithm computes $d_L(x_1, x_2) = 4.0$ and $m(x_1, 4.0) = 3$. Subsequently, the algorithm performs `Isum`$[8] =$ `Isum`$[8] + 1/m(x_1, 4.0) = 0.666$.

10. Finally, the search track backs to x_1, and since there present no unexplored outgoing edges of x_1, the algorithm terminates.

When the above search finishes, the jth element of the array `Isum` is equal to the interval sum $I_j(x_1)$ for $j = 1, \ldots, 10$.

8. Execution time against the number of point events

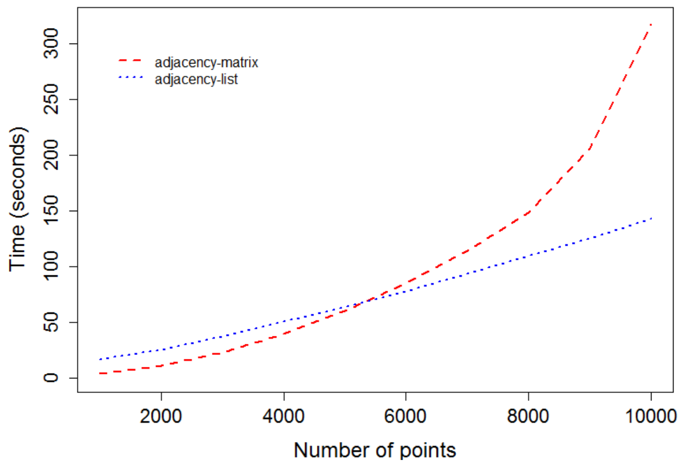
The computational time of both algorithms M and L crucially depends on p, the number of events observed on the network. Simulation experiments in this section suggest that the computational times of algorithms M and L are, respectively, quadratic and linear functions of p (see Figure 16a). A heuristic behind these timings is that Algorithm M computes $p(p-1)/2$ shortest-path distances for all distinct event pairs, whereas Algorithm L only iterates over p times (see Step *2* of Algorithm L).

Figure 16a compares execution times (in seconds) for the Algorithm M and the Algorithm L as a function of p. For this plot, first, we generated independent, uniformly-distributed random points on the `chicago` street network for $p = 1000, \ldots, 10000$, and then for each simulated data, we recorded the execution times of both the algorithms, taken for computing $\widehat{K}_{\mathsf{L}}$. The plot confirms that our proposed algorithm's computation time is linear in p, while that of the adjacency matrix algorithm is quadratic in p. The other drawback of the Algorithm M is that the R-program breaks down, even for a small network such as the `chicago` network, due to memory allocation problem when the number of points are greater than or equal to $11,000$.

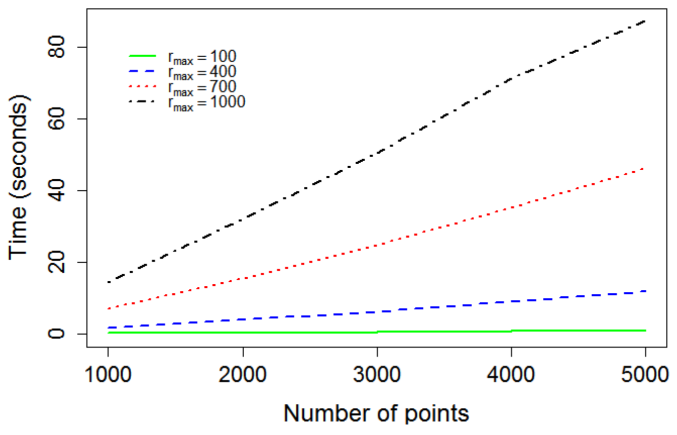
Figure 16a also shows that for small number of point events Algorithm M can outperform Algorithm L – although the differences in the timings are often negligible. Therefore, it is okay to use Algorithm M if the observed number of events on a network is relatively small and the network itself can be stored using the adjacency matrix data structure. Baddeley *et al.* (2015, Section 17.8) provides example illustrating the use of Algorithm M in applications to point patterns on linear networks. Ang *et al.* (2012) also used the same implementation for analyzing point patterns created by spider webs on the mortar lines of a wall and by crime events on the street network in Chicago, USA.

Although the adjacency matrix algorithm has satisfactory performance for small data sets, it is not feasible for larger data sets such as the Western Australian road network in Figure 1. Not only is the time complexity of order p^2, but also the memory storage requirement of Algorithm M is prohibitive, as discussed in Section 2.2. Based on the adjacency list structure in Figure 8 and the Algorithm L in Section 6.4, we created R interfaces for computing the summary statistics (4)-(6). This implementation is memory efficient; it can store and analyze large networks with more than 10^5 nodes and edges on a PC with only 8 Gb RAM.

The execution time of Algorithm L also depends on the chosen value of $r_{\max}$. However, the linear time complexity of the algorithm with respect to p holds true for any choice of $r_{\max}$. Figure 16b plots the computation times of the network K-function (5) against the number of



(a)



(b)

Figure 16: (a) Execution times of the adjacency matrix and adjacency list algorithms, plotted against the number of point events used to compute $\widehat{K_L}(r)$ on the `chicago` street network. (b) Execution times of the adjacency list algorithm for computing $\widehat{K_L}(r)$ (on the `chicago` network) corresponding to different $r_{\max}$ values. In both cases, uniform random points were generated on the network.

points on the `chicago` network for different choices of $r_{\max}$. It is evident from the plot that the choice of $r_{\max}$ only affects the slope of the linear relationship.

To evaluate performance on real data, we used the large Western Australian road network shown in Figure 1, and generated data sets with different numbers of point events by sampling without replacement from the 2011 road accident record. Figure 17 plots the timings (in minutes) of computing $\widehat{K_L}(r)$ for the data sets with $p = 1000, 3000, \ldots, 13000$. As expected, the plot shows an approximately linear relationship between the execution time and the number of points.

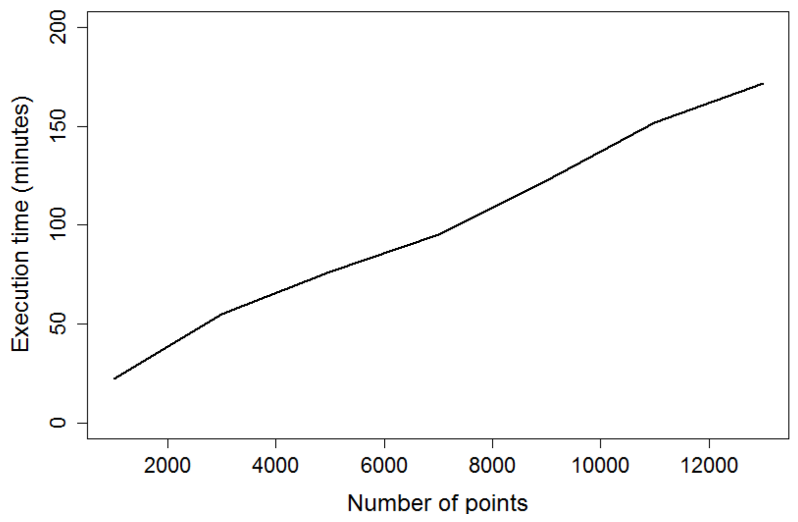


Figure 17: Execution time (in seconds) of the adjacency list algorithm for computing $\widehat{K}_{\mathsf{L}}(r)$ with different number of points sampled randomly from the accident data on the Western Australian road network.

9. Western Australia road network and road accident data

Here we compute the homogeneous and inhomogeneous network K-functions using Algorithm L for the road accident data set shown in Figure 1. The accidents are recorded on the road network of the state of Western Australia for the calendar year 2011. These data, provided by the Western Australian state government department of Main Roads, are made available for this publication as part of the Western Australian Whole of Government Open Data Policy. The primary data corresponding to the Western Australian road network and accidents can be accessed publicly from the Main Roads Data Portal. The network graph has $88,512$ nodes and $115,169$ edges with a total length of approximately $97,165$ km. There are $14,562$ accident locations recorded.

The geometrically-corrected homogeneous K-function $\widehat{K}_{\mathsf{L}}(r)$, equation (5), for the Western Australian accident pattern is shown in Figure 18 (left panel). The summary function $\widehat{K}_{\mathsf{L}}(r)$ is often compared with the theoretical K-function of the Poisson process ($K_{\mathsf{L}}(r) = r$) to assess whether the distribution of point pattern is different from a completely random point pattern. In Figure 18, the large difference between the empirical K-function $\widehat{K}_{\mathsf{L}}(r)$ and the theoretical benchmark value (dashed line) suggests a departure from the completely random point process model. One can use $\widehat{K}_{\mathsf{L}}(r)$ to formally test whether the accident pattern exhibits clustering, using a one-sided Monte Carlo test based on simulation envelopes (Baddeley *et al.* 2015, Chapter 10). The following lines of code can be used to compute and plot the homogeneous K-function in Figure 18.

```
R> library("spatstat.Knet")
R> data("wacrashes", package = "spatstat.Knet")
R> r_grid <- seq(0, 1000, length = 101)
R> Khom <- Knet(wacrashes, r = r_grid)
R> plot(Khom, legend = FALSE, lwd = 2, main = " ")
```

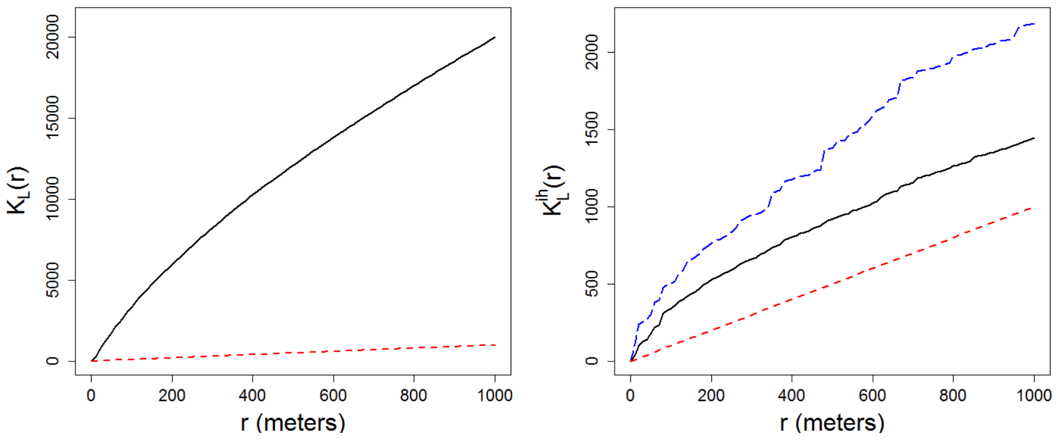


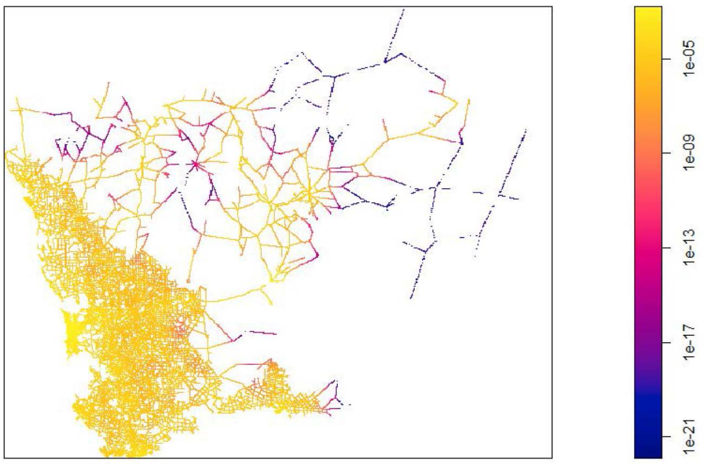
Figure 18: Left: Geometrically-corrected homogeneous K-function for the Western Australian road accident data set (solid line). Right: Geometrically-corrected inhomogeneous K-functions with intensity estimates computed using a fixed bandwidth smoothing method (large dashed-line) and using a variable bandwidth smoothing method (solid line). In both plots, the theoretical K-function for a completely random pattern is plotted using red dashed line and the horizontal axis is road distance in metres.

Although inference based on the homogeneous K-function is straightforward, its computation for the accident pattern assumes that the accident rate is constant across the entire road network. This assumption is clearly fallacious in this case, as the spatially varying accident rates are visible in Figure 1. If the underlying point process is inhomogeneous with a spatially varying intensity function $\lambda(u)$, $u \in L$, the inhomogeneous K-function $\widehat{K}_{\mathsf{L}}^{ih}(r)$ defined in (6) is typically used to examine the second-order properties, such as clustering or interactions amongst points, of the pattern (Ang *et al.* 2012; Baddeley, Møller, and Waagepetersen 2000). This function adjusts for the varying intensity by using the intensity estimates at the event locations as weights for the estimator $\widehat{K}_{\mathsf{L}}^{ih}(r)$.

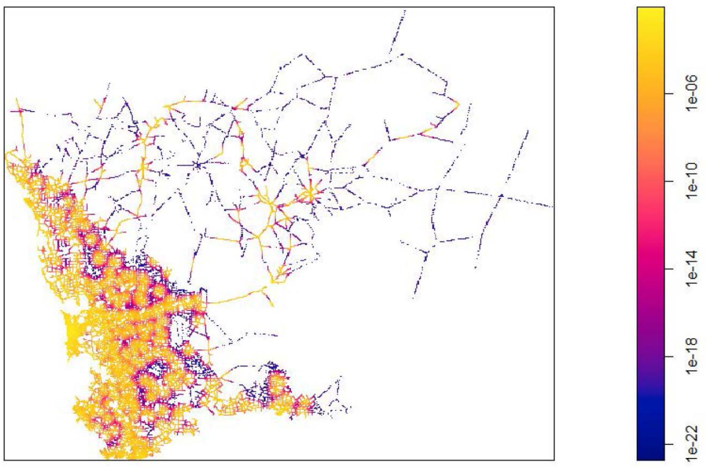
The accuracy of estimation of $\widehat{K}_{\mathsf{L}}^{ih}$ depends on how well we estimate the intensity function $\lambda(u)$. In case of two-dimensional point patterns, kernel smoothing is a standard nonparametric technique for estimating the intensity function. However, kernel smoothing is time-consuming on a linear network due to its non-homogeneous spatial structure and complex boundary configurations at different locations (McSwiggan *et al.* 2016). We estimated the spatially varying accident rates using fixed and variable bandwidth smoothing methods for point patterns on a linear network. Details on these methods will be given in a sequel paper (Rakshit *et al.* 2019).

Figures 19a and 19b show, respectively, heatmaps of the fixed and variable bandwidth intensity estimates (with color map on a logarithmic scale) on the Western Australian road network. The estimated intensity values are provided in supplementary files `waCrashIntensity.rda` and `waCrashIntensityAdaptive.rda`, respectively. The following code plots both the heatmaps.

```
R> library("spatstat.Knet")
R> load("waCrashIntensity.rda")
R> plot(waCrashIntensity, log = TRUE, main = " ")
R> load("waCrashIntensityAdaptive.rda")
```



(a)



(b)

Figure 19: (a) Fixed and (b) variable bandwidth estimate of the intensity for the accidents on the Western Australian road network. Color map is on a logarithmic scale.

```
R> plot(waCrashIntensityAdaptive, log = TRUE, main = " ")
```

Using these two estimates, we have computed two geometrically-corrected inhomogeneous K-functions, which are plotted in the right panel of Figure 18. The following code can be used for computing and plotting the inhomogeneous K-functions.

```
R> inten_est <- waCrashIntensity[as.ppp(wacrashes)]
R> inten_est_adap <- waCrashIntensityAdaptive[as.ppp(wacrashes)]
R> Kin <- Knetinhom(wacrashes, lambda = inten_est, r = r_grid)
R> Kin_adap <- Knetinhom(wacrashes, lambda = inten_est_adap, r = r_grid)
R> plot(Kin, lty = c(5, 2), lwd = 2, col = c("blue", "red"), legend = FALSE)
R> plot(Kin_adap, est ~ r, add = TRUE, lty = 1, lwd = 2, col = "black")
```

The Western Australian accident pattern is dense in some parts of the network and very sparse in other parts. In such a scenario, the fixed bandwidth estimator performs unsatisfactorily because it over-smooths the densely populated parts of the network while producing intensity estimates close to zero for network parts that are sparsely populated. This is evident from the over-smoothing of the densely-populated western part of the state and from several missing pixels in Figure 19a, appearing in the sparsely populated north western part of the network, which corresponds to intensity estimates very close to zero. The variable bandwidth intensity estimates in Figure 19b are computed after adjusting for the underlying density of the point events – relatively large bandwidths are used in the sparsely populated areas than the dense areas. This reduces the over-smoothed nature of the heatmap and decreases the number of zero-valued pixels.

If we contrast the two inhomogeneous K-functions in Figure 18, we observe that, although both plots suggest some form of clustering in the accident pattern, the K-function with the adaptively smoothed intensity estimates reveals a lesser degree of clustering than its counterpart with fixed bandwidth estimates. However, this is expected as the adaptive smoothing based on variable bandwidth provide better estimates of the underlying intensity function than the fixed bandwidth estimates.

10. Discussion

This paper examined two general approaches to the computation of statistical summaries of events on a network. An approach based on the incidence matrix (Algorithm M) is straightforward to implement, and quite fast to execute, but is severely limited by its very large memory requirements. The adjacency matrix is wasteful because the vast majority of entries are zero. A sparse matrix representation reduces storage requirements but is not efficient for graph topology operations.

The alternative approach using adjacency lists (Algorithm L) results in substantial memory savings. This is evident from Table 1, which gives a comparison of memory usage between these two implementations of the K-function in (5) for three different network data sets available in the R package **spatstat** (Baddeley and Turner 2005). This efficient use of memory allows Algorithm L to be applied to very large networks, such as the entire road network of Western Australia. Algorithm L also lends itself easily to the calculation of other quantities such as the local K-functions.

Although Algorithm L adapts some ideas from Okabe and Yamada (2001), a direct comparison between our implementation and that of Okabe and Yamada (2001) does not seem appropriate. The latter computes the uncorrected, unweighted K-function (4), whereas our implementation is designed to compute any summary function of the general form (1) with special emphasis on the geometrically-corrected empirical K function (5).

Many improvements are possible. In very large and complex networks it might be more efficient to use quad-trees or other geometric hashing methods to divide the network into manageable pieces (cf. Okabe and Sugihara 2012, Chapter 3). Furthermore, it would be possible to improve parts of Algorithm L by using a priority queue. The operation of extracting the minimum value from a priority queue is $O(1)$, and maintaining the heap property of the priority queue is $O(\log M)$, where M is the number of elements in the queue.

The representation of a road network as a graph is a substantial over-simplification of the real physical network (Okabe and Sugihara 2012). Roads have complicated geometry including curvature and camber, multiple lanes, complicated intersections, overpasses, and structures which separate different lanes. The analysis of road accidents must take into account many of these covariates associated with the road network. The `aNode` components could easily be extended in order to make these covariates accessible in the adjacency-list data structure.

Supplementary materials

The supplementary materials provide the source code of our implementation of Algorithm L; the data analyzed in the paper; and R scripts needed to reproduce our results.

Computation times reported in the paper were measured on a 2.67 GHz Windows laptop with 8 Gb RAM.

The R package **spatstat.Knet** contains our implementation of Algorithm L. The implementation is written in C with an R interface through functions named `Knet` and `Knetinhom`. The **spatstat.Knet** package also contains the point pattern data sets analyzed in the paper, and the two estimated intensity functions depicted in Figures 19a and 19b.

The file `v90i01.R` is a stand-alone R script for reproducing all the results and figures in the paper.

The **spatstat.Knet** package will be updated from time to time; the latest version can be installed from the authors' Github repository using the R package **remotes** (Hester, Csárdi, Wickham, Chang, Morgan, and Tenenbaum 2019):

```
R> library("remotes")
R> install_github("spatstat/spatstat.Knet")
```

Acknowledgments

We gratefully acknowledge the collaboration, participation, and support of Main Roads Western Australia, in particular Ms. Thandar Lim and Dr. Sanath Jayamanna. We also thank Mr. Isaac Gravestock for his collaboration in early stages of the project. This work was supported by Australian Research Council grant DP130102322. We thank CSIRO and the University of Western Australia for computing resources and related support.

References

Anderson E, Bai Z, Bischof C, Blackford S, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999). ***LAPACK** Users' Guide.* 3rd edition. SIAM, Philadelphia.

Ang QW, Baddeley A, Nair G (2012). "Geometrically Corrected Second Order Analysis of Events on a Linear Network, with Applications to Ecology and Criminology." *Scandinavian Journal of Statistics*, **39**(4), 591–617. `doi:10.1111/j.1467-9469.2011.00752.x`.

Anselin L (1995). "Local Indicators of Spatial Association – LISA." *Geographical Analysis*, **27**(2), 93–115. `doi:10.1111/j.1538-4632.1995.tb00338.x`.

Baddeley A, Jammalamadaka A, Nair G (2014). "Multitype Point Process Analysis of Spines on the Dendrite Network of a Neuron." *Journal of the Royal Statistical Society C*, **63**(5), 673–694. doi:10.1111/rssc.12054.

Baddeley A, Møller J, Waagepetersen R (2000). "Non- and Semi-Parametric Estimation of Interaction in Inhomogeneous Point Patterns." *Statistica Neerlandica*, **54**(3), 329–350. doi:10.1111/1467-9574.00144.

Baddeley A, Rubak E, Turner R (2015). *Spatial Point Patterns: Methodology and Applications with R*. Chapman and Hall/CRC, Boca Raton.

Baddeley A, Turner R (2005). "**spatstat**: An R Package for Analyzing Spatial Point Patterns." *Journal of Statistical Software*, **12**(6), 1–42. doi:10.18637/jss.v012.i06.

Bates D, Maechler M (2019). ***Matrix**: Sparse and Dense Matrix Classes and Methods*. R package version 1.2-17, URL https://CRAN.R-project.org/package=Matrix.

Bengtsson H (2018). ***profmem**: Simple Memory Profiling for R*. R package version 0.5.0, URL https://CRAN.R-project.org/package=profmem.

Boots B, Okabe A (2007). "Local Statistical Spatial Analysis: Inventory and Prospect." *International Journal of Geographical Information Science*, **21**(4), 355–375. doi:10.1080/13658810601034267.

Cormen TH, Leiserson CE, Rivest RL, Stein C (2009). *Introduction to Algorithms*. 3rd edition. The MIT Press, London.

Diggle PJ (1985). "A Kernel Method for Smoothing Point Process Data." *Journal of the Royal Statistical Society C*, **34**(2), 138–147. doi:10.2307/2347366.

Dongarra J, Moler C, Bunch JR, Stewart GW (1979). ***LINPACK** Users' Guide*. 1st edition. SIAM.

Even S (1979). *Graph Algorithms*. 1st edition. Computer Science Press.

Gallo G, Pallottino S (1988). "Shortest Path Algorithms." *The Annals of Operations Research*, **13**(1), 1–79. doi:10.1007/bf02288320.

Getis A, Franklin J (1987). "Second-Order Neighbourhood Analysis of Mapped Point Patterns." *Ecology*, **68**(3), 473–477. doi:10.2307/1938452.

Golub GH, Van Loan CF (1996). *Matrix Computations*. 3rd edition. Johns Hopkins, Baltimore.

Guan Y (2006). "A Composite Likelihood Approach in Fitting Spatial Point Process Models." *Journal of the American Statistical Association*, **101**(476), 1502–1512. doi:10.1198/016214506000000500.

Hester J, Csárdi G, Wickham H, Chang W, Morgan M, Tenenbaum D (2019). ***remotes**: R Package Installation from Remote Repositories, Including GitHub*. R package version 2.0.4, URL https://CRAN.R-project.org/package=remotes.

Hopcroft JE, Tarjan RE (1973). "Algorithm 447: Efficient Algorithms for Graph Manipulation." *Communications of the ACM*, **16**(6), 372–378. doi:10.1145/362248.362272.

Koenker R, Ng P (2003). "**SparseM**: A Sparse Matrix Package for R." *Journal of Statistical Software*, **8**(6), 1–9. doi:10.18637/jss.v008.i06.

Kolaczyk ED, Csárdi G (2014). *Statistical Analysis of Network Data with R*. Springer-Verlag, New York.

Louden K (1999). *Mastering Algorithms with C*. 1st edition. O'Reilly.

McSwiggan G, Baddeley A, Nair G (2016). "Kernel Density Estimation on a Linear Network." *Scandinavian Journal of Statistics*, **44**(2), 324–345. doi:10.1111/sjos.12255.

Okabe A, Okunuki K, Shiode S (2006). "The **SANET** Toolbox: New Methods for Network Spatial Analysis." *Transactions in GIS*, **10**(4), 535–550. doi:10.1111/j.1467-9671.2006.01011.x.

Okabe A, Satoh T, Sugihara K (2009). "A Kernel Density Estimation Method for Networks, Its Computational Method and a GIS-Based Tool." *International Journal of Geographical Information Science*, **23**(1), 7–32. doi:10.1080/13658810802475491.

Okabe A, Sugihara K (2012). *Spatial Analysis Along Networks*. John Wiley & Sons, New York.

Okabe A, Yamada I (2001). "The K-Function Method on a Network and Its Computational Implementation." *Geographical Analysis*, **33**(3), 271–290. doi:10.1111/j.1538-4632.2001.tb00448.x.

Pissanetzky S (1984). *Sparse Matrix Technology*. 1st edition. Academic Press.

Rakshit S, Baddeley A (2019). ***spatstat.Knet**: Extension to **spatstat** for Large Datasets on a Linear Network*. R package version 1.11-2, URL https://CRAN.R-project.org/package=spatstat.Knet.

Rakshit S, Davies T, Moradi M, McSwiggan G, Nair G, Mateu J, Baddeley A (2019). "Fast Kernel Smoothing of Point Patterns on a Large Network Using 2D Convolution." *International Statistical Review*. Forthcoming.

R Core Team (2019). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. URL https://www.R-project.org/.

Ripley BD (1977). "Modelling Spatial Patterns." *Journal of the Royal Statistical Society B*, **39**(2), 172–212.

Tanaka U, Ogata Y, Stoyan D (2008). "Parameter Estimation and Model Selection for Neyman-Scott Point Processes." *Biometrical Journal*, **50**(1), 43–57. doi:10.1002/bimj.200610339.

Tarjan RE (1983). *Data Structures and Network Algorithms*. 1st edition. Society for Industrial and Applied Mathematics.

Tarzan R (1972). "Depth-First Search and Linear Graph Algorithms." *SIAM Journal on Computing*, **1**(2), 146–160. doi:10.1137/0201010.

Tewarson RP (1973). *Sparse Matrices*. Mathematics in Science and Engineering, 1st edition. Academic Press.

Ver Hoef JM, Peterson E, Theobald D (2006). "Spatial Statistical Models That Use Flow and Stream Distance." *Environmental and Ecological Statistics*, **13**(4), 449–464. doi:10.1007/s10651-006-0022-8.

Wilkinson JH, Reinsch C (eds.) (1971). *Linear Algebra*, volume II of *Handbook for Automatic Computation*. Springer-Verlag, Berlin.

Affiliation:

Suman Rakshit
SAGI-West, School of Molecular and Life Sciences
Curtin University
GPO Box U1987
Perth WA 6845, Australia

Adrian Baddeley
Department of Mathematics and Statistics
Curtin University
GPO Box U1987
Perth WA 6845, Australia
E-mail: adrian.baddeley@curtin.edu.au

Gopalan Nair
School of Mathematics and Statistics
University of Western Australia
35 Stirling Highway
Crawley WA 6009, Australia

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
July 2019, Volume 90, Issue 1 *Submitted:* 2017-06-04
doi:10.18637/jss.v090.i01 *Accepted:* 2018-03-08