EFFICIENT MINING AND EXPLORATION OF MULTIPLE INTERSECTING AXIS-ALIGNED OBJECTS

TILEMACHOS PECHLIVANOGLOU

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO
JULY 2019

EFFICIENT MINING AND EXPLORATION OF MULTIPLE INTERSECTING AXIS-ALIGNED OBJECTS

by **Tilemachos Pechlivanoglou**

a thesis submitted to the Faculty of Graduate Studies of York University in partial fulfilment of the requirements for the degree of

MASTER OF SCIENCE

EFFICIENT MINING AND EXPLORATION OF MULTIPLE INTERSECTING AXIS-ALIGNED OBJECTS

by **Tilemachos Pechlivanoglou**

Abstract

Identifying and quantifying the size of multiple overlapping axis-aligned geometric objects is an essential computational geometry problem. The ability to solve this problem can effectively inform a number of spatial data mining methods and can provide support in decision making for a variety of critical applications. Currently, the state-of-the-art approach for addressing such intersection problems resorts to an algorithmic paradigm, collectively known as the *sweep-line* or *plane sweep* algorithm. However, its application inherits a number of limitations including lack of versatility and lack of support for ad hoc intersection queries. With these limitations in mind, we design and implement a novel, exact, fast and scalable yet versatile, sweep-line based algorithm, named *SLIG*. The key idea of our algorithm lies in constructing an auxiliary data structure when the sweep line algorithm is applied, an *intersection graph*. This graph can effectively be used to provide connectivity properties among overlapping objects and to inform answers to ad hoc intersection queries. It can also be employed to find the location and size of the common area of multiple overlapping objects. *SLIG* performs significantly faster than classic sweep-line based algorithms, it is more versatile, and provides a suite of powerful querying capabilities. To demonstrate the versatility of our *SLIG* algorithm we show how it can be utilized for evaluating the importance of nodes in a trajectory network - a type of dynamic network where the nodes are moving objects (cars, pedestrians, etc.) and the edges represent interactions (contacts) between objects as defined by a proximity threshold. The key observation to address the problem is that the time intervals of these interactions can be represented as *1-dimensional axis-aligned geometric objects*. Then, a variant of our *SLIG* algorithm, named *SLOT*, is utilized that effectively computes the metrics of interest, including node degree, triangle membership and connected components for each node, over time.

Table of Contents

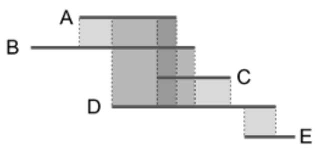
1 Introduction

In computational geometry, a *sweep-line* algorithm is an algorithmic paradigm that is commonly used in problems related to identifying pairs of intersecting geometric objects. This process can be critical for a number of spatial data mining and querying methods, as well as various critical applications, including spatial databases [47], VLSI physical design [25], computer graphics and simulation collision detection [55], to name a few.

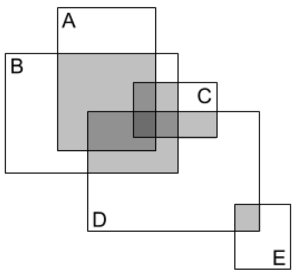
While there are many variations, depending on the type of intersecting objects to be examined, all sweep-line algorithms employ a conceptual line that is swept or moved across the plane, ”scanning” over an entire dataset while only stopping at some points. Any algorithm operations are restricted to points that either intersect or are in the immediate vicinity of the sweeping line at those points, with the results being available at the end of the single pass. Because of this fact, sweep-line-based algorithms don't need to compare every possible pair of objects, only checking those in close proximity instead.

An essential computational geometry problem that belongs to the category mentioned above is that of identifying intersections of a large number of axis-aligned geometric objects in multiple dimensions [8]. The axis-aligned requirement constrains the objects to be located on axes that align with each other (i.e., run in the same direction); that includes *line segments* in 1-D, *rectangles* or *boxes* in 2-D, *cuboids* in 3-D, and so on (see examples in Fig. 1.1a, 1.1b, 1.1c). This problem is well-suited to the sweep-line algorithmic approach, with fast and efficient state-of-the-art algorithms able to find pairs of intersections in 1, 2 or 3 dimensions [20].

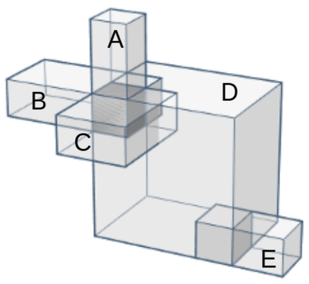
In this work, we wish to expand the scope of the axis-aligned object intersection problem mentioned, as



(a) 1-D line segments



(b) 2-D rectangles



(c) 3-D cuboids

Figure 1.1: Example of axis-aligned geometric objects in 1-D, 2-D and 3-D.

well as demonstrate how another, significantly different problem can be mapped to it. We wish to show the versatility and usefulness of the sweep-line algorithm, as it can be used to provide meaningful insights in the analysis of data from a wide variety of domains.

Specifically, we first modify the sweep-line algorithm so that it can identify not just pairs of intersecting objects, but also *multiple* object intersections, i.e. triplets, quadruplets and larger intersecting sets of objects, as well as compute the size of their common overlap. Secondly, we present a problem related to spatial trajectories and temporal graphs, and show it is possible to map it to the object intersection problem, specifically to the $1 - D$ case of intervals. This enables us to use the sweep-line algorithm in order to quickly and efficiently calculate a number of metrics related to the importance of each object in the dataset.

1.1 Contributions

In summary, the major contributions of this work are as follows:

- We present MultipleIntersect, a novel and challenging problem related to intersections of geometric objects in multiple dimensions. This problem can appear in diverse applications and domains, but cannot be easily addressed using traditional implementations of modified state-of-the-art algorithms.

- We design and implement SLIG, a novel sweep-line based algorithm that can efficiently address the introduced problem MultipleIntersect. The algorithm is utilizing information coming from an auxiliary data structure, an *intersection graph*, and is *fast*, *exact*, *versatile* and *scalable*.

- We perform additional exploratory analysis of this problem by answering different types of queries related to multiple region intersections, namely MultiRegionQuery and SingleRegionQuery.
- We present a thorough experimental evaluation of SLIG against state-of-the-art algorithms that demonstrate that our algorithm is superior for a range of conditions.
- We demonstrate the capabilities and versatility of SLIG by using it in a data analysis application on a real-world extreme-weather-related dataset.
- We introduce a novel framework for network-based trajectory analysis.
- We design and implement SLOT, a novel one-pass algorithm, for fast and accurate mining of node importance in trajectory networks.
- We perform a thorough evaluation of node importance methods on large-scale synthetic data, for a range of conditions.
- We make *source code* and *data* publicly available to encourage reproducibility of results.

1.2 Structure

In Chapter 2, several key concepts and ideas related to existing techniques and methodologies are presented in more detail, along with the related work for each of the problems we tackle. In Chapter 3, we present the problem of identifying *multiple* object intersections, along with the various necessary modifications to the sweep-line algorithm in order to achieve this. In Chapter 4, we outline the problem of importance in trajectory networks, along with how it can be mapped to the interval intersection problem. Chapter 5 contains some final comments and potential future steps related to this work.

2 Related Work

Our research is related to the computational geometry problem of *object intersection*, the sweep-line algorithm, as well as spatial data structures. Furthermore, in this work we employ techniques and concepts from the fields of graph theory and trajectory mining, such as *clique enumeration*, *trajectory data mining*, *dynamic* and *temporal networks.* Here we present a more comprehensive coverage of these topics.

2.1 Object Intersection

A great number of data structures and algorithms have been developed that deal with finding and performing queries on intersecting objects [8, 20]. One of the most common problems related to that is the axis-aligned (or iso-oriented) rectangles in $\mathbb{R}^d$ problem [22, 15], or the very similar orthogonal range search problem [16]. In most prior research, the objective is to identify and report pairs of intersecting objects with speed and accuracy, usually for purposes or collision detection in computer simulations [65] or for object placement problems [1]. Some methods exist for problems conceptually similar but fundamentally different from ours, such as the proposed technique to pre-process data in order to quickly find intersecting pairs that overlap a query rectangle, a.k.a. triple intersections [45, 18]. In this case, some computation speed-up is sacrificed in order to preserve space requirements of $O(1)$, while it is hinted that without this constraint lower computation costs are easily possible. However, this is still constrained to only exactly triple intersections ($k = 3$). The state-of-the-art techniques used in related research belong to one of two families of algorithms: either a *sweep-line* (also known as *plane-sweep* or *Bentley-Ottman* algorithm) or a *divide-and-conquer* algorithm, which have been shown to be equivalent in computation cost [28]. These algorithms are commonly tasked with the purpose of identifying all pairs of intersecting objects, and using that information to construct data structures that can accommodate spatial access queries [3, 64]. While these algorithms have been extensively studied and there are many improvements and optimised implementations proposed [42], they can be problematic when trying to apply them to identify more than just pairs of intersections. A specific, fixed implementation logic must be employed depending on the intended result, e.g. if the goal is identifying intersecting triplets in $2 - D$ or quadruplets in $1 - D$. For our work, we select the sweep-line approach due to its implementation simplicity, although similar results could be obtained using a divide-and-conquer approach or related implementations, such as *R-trees* [29].

2.2 Sweep-line Algorithm

The existing state-of-the-art sweep-line algorithm finds pairs of intersecting objects. Given a set of d-dimensional objects (e.g., line segments, rectangles, cuboids, etc.), the first step of the algorithm involves constructing a list that includes the *start* and *stop* points of all objects in each dimension and sorting them, typically in a pre-processing phase. Then, a *conceptual line*, L moves (sweeps) from left to right across the plane, examining the objects, one by one, in order. During the sweep, the *active objects* (i.e., the ones that line L is currently traversing over) are maintained. When a new region is encountered by L, it is marked as *intersecting* all the currently active objects in the current sweep dimension. The process consists of a single pass in each dimension. Regions that are marked as *intersecting* in all dimensions are actually *intersecting objects*. An illustrative example of the process for 1-D objects (line segments) can be seen in Figure 2.1.

2.3 Spatial Data Structures

Tree-based data structures are commonly used in spatial access methods for indexing information such as geographical coordinates, rectangles or polygons. In 1-D problems, interval trees [9] are used to efficiently find all intervals that overlap with any given interval or point. In more than one dimensions, *bounding volume hierarchy trees*, more commonly known as *Axis-Aligned Bounding Box (AABB) trees* [10] manage to

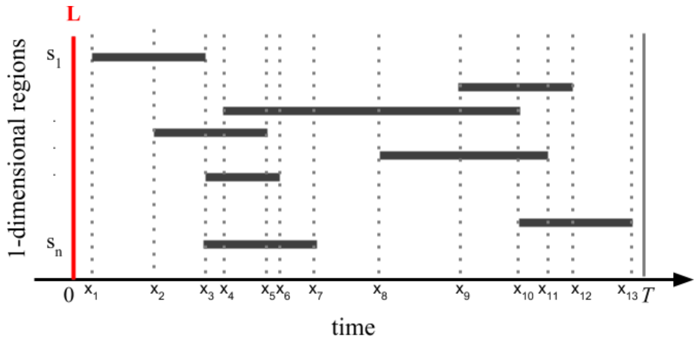


Figure 2.1: Illustrative example of a sweep-line algorithm that can determine intersections of 1-D objects (line segments).

significantly reduce computation costs of spatial queries by using a hierarchy of larger rectangular bounding volumes that contain smaller objects. A similar concept, *R-trees* group objects by using minimum bounding rectangles [29]. Along with several variants such as *R*, R+* and *Hilbert R-trees [7, 52, 32]* they allow significantly fast answers to spatial access queries and similar collision detection problems.In all these tree-based data structures, the intervals or rectangles are represented by *leaf nodes* and they are incrementally merged to form a tree with a common *root node*, with different grouping heuristics. While this method is very effective for answering access queries, which effectively require the traversal of the tree from the root to the leaves, it is inadequate to answer queries related to the intersections between groups of objects, since they require traversing through the leaf nodes of the tree. A data structure that is more capable of answering these types of queries is the intersection graph [49]. In intersection graphs, each *vertex* represents a single rectangle or interval and each *edge* represents an intersection between two of these objects [41]. In the case of 1-D regions, this graph is known as an *interval graph* [39]. As these data structures maintain information on the intersections and connections between individual objects, it is possible to use intersection graphs in order to provide answers to queries involving groups of objects [31]. Common graph queries such as identifying all neighbors of a node or finding connected components in a graph now translate to queries related to the actual geometric objects and their intersections in space.

2.4 Clique enumeration

A *clique* is a fully-connected subgraph in a larger, simple graph. That means that every node within a clique has a connection with every other node within it. Furthermore, a *maximal clique* is one that can't be augmented by adding additional vertices, while a *maximum clique* is the largest maximal clique in the graph. We are interested in the problem of finding a maximum clique [4] and the problem of enumerating all maximal cliques [14]. As one of the best known and most widely studied combinatorial problems in graph theory, there is an exhaustive list of related work. These problems are NP-hard, given that they subsume the classic version of the NP-complete clique decision problem. But, because of their significance in real applications, many practical algorithms have been proposed [12]. In relation to our problem, in Section 3.3, we first establish the relationship between the multiple intersection problem and the *clique enumeration* problem. Then, we explain how existing clique enumeration algorithms can be employed to enumerate sets of intersecting regions. As such, clique enumeration algorithms are orthogonal to our methods and improved versions can be adapted as needed.

2.5 Trajectory Data Mining

Trajectory problems have been extensively studied in the data mining area. Of particular interest are problems related to similarity in trajectory data [36, 56] and trajectory clustering [38]. While there is research on the behavior and trajectories of moving objects [21], we wish to focus on the interactions between them, and specifically the way objects come in close proximity to each other and the groups they form. This is roughly similar to the concept of contact networks commonly studied in epidemiology and infection spreading, although it's not strictly related to spatial proximity [50].

2.6 Spatial Networks

The notion of objects distributed in space and interacting with each other has been heavily explored in graph theory. Graph theory concepts such as proximity graphs [26] and geometric intersection graphs [24] are characteristic examples of this. Specifically, several variations of proximity graphs have been developed over time to better fit different problems. For instance, relative neighbor graphs [57] and Gabriel graphs [26] connect nearest neighbors if no other vertexes are nearby, while Delaunay triangulations [19] maximize the minimum angles of all triangles formed. These, however, mostly deal with static data, while our goal is to examine cases of proximity graphs where all objects/nodes are moving and their relationships are evolving over time.

2.7 Temporal Networks

There has also been significant research on networks that are evolving over time, or temporal networks. The nature of these systems introduces a number of issues and obstacles, which make it necessary to adapt for the problem basic graph theory concepts and algorithms such as shortest paths [61], motifs [35] and other metrics [44]. Furthermore, with the addition of temporal information, several concepts can be extended to take advantage of the additional data. Examples of this are the temporal node centrality [34] and the network reachability [30] metrics. As part of this project we wish to develop techniques that provide accurate values for these metrics in a fast and accurate way for all nodes, with detailed results over time, then apply these for the analysis of trajectory temporal networks. We intend to make use of the node degree and reachability or connectedness metrics, as well as examine the participation of every node in triangles over time. Earlier research on this topic, including *network temporal motifs*, focused mostly on faster approximations for streaming graphs [13, 6, 58]. An accurate triangle counting algorithm without approximation exists [46], but merely returns the total *count* of all triangles rather than *enumerating* them.

3 Mining of Multiple Axis-aligned Intersecting Objects

3.1 Motivation

In the previous chapter, we outlined the details of the sweep-line algorithm, and showed how the reported results are the pairs of intersecting objects. In this chapter, we want to explore the related problem of identifying *multiple* intersections of such objects (e.g. triplets, quadruplets of objects intersecting at the same point) and computing the size of their common overlap. While the current state-of-the-art sweep-line algorithm mentioned before has been successfully used in a number of problems, its application inherits a number of limitations that are relevant here: *(i) lack of versatility*: classic sweep-line algorithms cannot easily be adapted to different instances of the intersection problem, such as multiple intersections or arbitrary dimensions. This is because they are typically designed and coded with clear and definite specifications of an intersection problem and therefore the logic is bound to specifics of the implementation; *(ii) lack of support for ad hoc intersection queries*: classic sweep-line algorithms are not designed to support ad hoc intersection queries, such as queries that aim to answer whether a set of geometric objects (query) are intersecting (or not), and if yes, how much they are overlapping.

With these limitations in mind, we design and propose SLIG[1], a sweep-line based algorithm that can be employed in a number of problems and applications involving the efficient computation of multiple object intersections or intersection queries, in multiple dimensions. The key idea of our algorithm lies in constructing an auxiliary data structure when the sweep line algorithm is applied, an *intersection graph*. This graph can effectively be used to provide connectivity properties among overlapping objects and to inform answers to

[1]Sweep Line (assisted by an) Intersection Graph

ad hoc intersection queries. In addition, it can be employed to inform the much harder problem of finding the location and size of the common area defined by multiple overlapping objects by reducing it to the clique enumeration problem, a problem thoroughly explored in graph theory.

Table 3.1: Summary of Notations

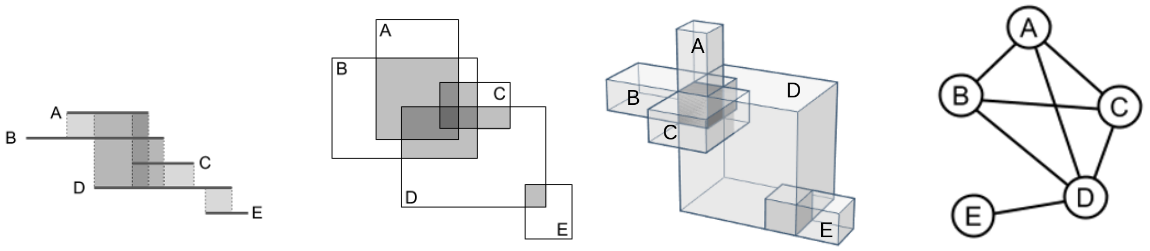
Notation	**Description**
d	Number of dimensions
s_i	An axis-aligned region in $\mathbb{R}^d$
S	A set of regions $\{s_1, s_2, \ldots, s_n\}$
I	A subset of S, whose regions all intersect with each other
Z	The common overlap region of all regions in I
$\|Z\|$	The size of the common overlap region Z
k	Intersection cardinality of I
l	Number of fully intersecting sets I_i in S
l_{max}	Number of maximal intersecting sets I_i in S, i.e. only the largest possible sets
q	A region in S used as a query
Q	A subset of regions S forming a query
r	The average size of regions in the dataset as a ratio of the total space in each dimension

3.2 Problem of Interest

In this section, we introduce notation and formally present the problems of interest.

3.2.1 Preliminaries

It is important to note that the methods we present generalize to multiple dimensions. Where necessary, we explain the extra steps required to apply a method in higher dimensions. This typically involves first operating on each dimension in isolation and then combining results from multiple dimensions to formulate the final outcome. Therefore, for the rest of this document, we use the term *regions* to refer to geometric objects in $\mathbb{R}^d$, where $d \geq 1$ (i.e., *line segments* in 1-D, *rectangles* or *boxes* in 2-D, *cuboids* in 3-D, etc.). Note as well that even though our implementations are general, we only report values for up to three dimensions



(a) 1-D line segments (b) 2-D rectangles (c) 3-D cuboids (d) Intersection graph

Figure 3.1: Example of axis-aligned geometric objects in 1-D, 2-D and 3-D, and their corresponding intersection graph. Note that A, B, C and D are all intersecting with each other, forming a *common region*. That common region is represented in the intersection graph as a *maximal clique*.

in the experiments, as that should be enough to demonstrate the behavior of the methods in most common scenarios and applications.

Related to the problem we aim to address is the way that geometric objects are overlapping with each other. Consider for example the regions A and B in Figure 3.1b. As they are overlapping with each other, they form a *common region* Z_{AB} that consists of all points in space that belong to both regions A and B. We also define the *common region size* $|Z_{AB}|$ to be the size of that overlap. Depending on the application dimensions, the size can represent the *length* of a line segment, the *surface* of the overlapping rectangles/boxes, or the *volume* of the cuboid formed by the overlapping cuboids.

In order to generalize the concept of *common region* to more than two regions, we need to consider all the different ways that overlaps can occur. For example, in Figure 3.1b, regions A, D and E have some pairwise overlaps (Z_{AD}, Z_{DE}), but they do not all overlap with each other forming a single common region (Z_{ADE}). On the other hand, for example, regions A, B, C, D are all overlapping with each other forming Z_{ABCD}, each point of which belongs to (is covered by) all four regions. We introduce the concept of *intersection cardinality* k to refer to the type of overlapping regions (triplets, quadruplets, etc.) that we are interested in detecting and reporting.

Formally, given a set of regions $S = \{s_1, s_2, \dots, s_n\}$ that individually may or may not overlap with each other, we can construct sets of regions $I_i \subseteq S$ where each region in I intersects with every other in I, i.e. they are all overlapping together and forming a common region Z. Any such subset I will have cardinality k equal to its size $|I|$, since all regions in that I intersect. For example, in Fig. 1.1b, $\{D, E\}$ has $k = 2$, $\{A, B, C\}$ has $k = 3$, $\{A, B, C, D\}$ has $k = 4$, but $\{A, B, C, D, E\}$ has $k = 4$, since the largest subset with a common region is $\{A, B, C, D\}$.

A worst-case scenario exists where $k = n$, where $n = |S|$ is the total number of objects, meaning that all regions are overlapping with each other; this, however, is a degenerate case for most real-world scenarios or applications.

3.2.2 Problem

We are now in position to formally define the problems of interest.

MultipleIntersect: Given a set $S = \{s_1, s_2 \dots, s_n\}$ of n regions in $\mathbb{R}^d$,where the number of dimensions $d \geq 1$, enumerate all sets of intersecting regions $I_i \subseteq S$. For each of the sets, report the common region Z_i and its respective size $|Z_i|$.

For example, given the set of regions $S = \{A, B, C, D, E\}$ of Figure 3.1b, we seek to find the intersecting sets *AB, AC, AD, BC, BD, CD, DE, ABC, ABD, ACD, BCD, ABCD*, as well as their respective *common regions* Z_{AB}, Z_{AC}, …, Z_{ABCD} and *common region sizes* $|Z_{AB}|$, $|Z_{AC}|$, …, $|Z_{ABCD}|$.

Intersection Queries: Given a set $S = \{s_1, s_2, \dots, s_n\}$ of n regions in $\mathbb{R}^d$, $d \geq 1$, construct a data structure capable of answering different types of queries related to S, such as:

- SingleRegionQuery: For a region $q \in S$, report all sets of intersecting regions $I_i \subseteq S$ containing q.
- MultiRegionQuery: For a subset of the regions $Q = \{q_1, q_2, \dots, q_m\} \subseteq S$ report all sets of intersecting regions $I_i \subseteq S$ containing any region $q_i \in Q$.

In the same example as before for $q = D$, SingleRegionQuery will report the following sets and their respective *common regions* and *sizes*: *AD, BD, CD, DE, ABD, ACD, BCD, ABCD*. Similarly, if $Q = \{A, E\}$, then MultiRegionQuery will enumerate the sets, *common regions* and *sizes* for: *AB, AC, AD, DE, ABC, ABD, ACD, ABCD.*

Note that these problems are difficult to compute using the existing state-of-the-art sweep-line algorithm algorithm presented in Chapter 2. Although it's possible to use customized versions of the sweep-line or equivalent algorithm to accommodate a specific instance of the problem, it won't be versatile enough to accommodate all problems within the same implementation, or an arbitrary number of dimensions. Furthermore, any possible solutions to the presented problem are inherently expensive even for 2- and 3-D cases [18], let alone for more dimensions.

3.3 Methodology

In this section, we describe methods for addressing the MultipleIntersect problem defined in the previous section. We start with the description of a naive implementation to address the problem, and we explain why the naive approach performs poorly. Then, we describe in detail of how one would utilize the state-of-the-art sweep-line paradigm instead; eventually that would provide a sensible baseline to compare against our more sophisticated algorithm. Afterwards, we present details of our proposed algorithm, SLIG, a sweep-line based algorithm that constructs and utilizes an intersection graph to greatly simplify and speed up the solution to the problem. Finally, we show how the data structures constructed using SLIG can be used to quickly and efficiently answer the SingleRegionQuery and MultiRegionQuery queries outlined in Section 3.2. It is important to note that all methods described below compute *exact* results (not approximated in any way), therefore discussion about the accuracy of the methods isn't necessary. Table 3.2 provides a summary of the computational complexity of different methods for addressing each of the problems of interest. The exact implementation of the methods shown can be found in Section 3.4.

3.3.1 MultipleIntersect Problem

Recall that in this problem we wish to report all the sets of intersecting regions $I_i \subseteq S$. Finding whether a pair of regions in $\mathbb{R}^d$ intersect or not is straightforward, as it's only necessary to compare the corner points of both regions in all dimensions. There are, however, different methods to select which of the possible regions should be compared with each other, resulting in different numbers of unnecessary comparisons.

The Naive Method: The simplest approach is comparing every region in S with every other, finding intersecting pairs, and then proceed to compare the common region defined by every intersecting pair with every other region to find triple intersections, and so on. This iterative process has to be continued until no more intersections are found or the maximum cardinality $k = n$ has been reached. As is apparent, the computational cost of such a method increases exponentially with the number of regions in S. For a specific k, the k combinations of the n regions in S are given by $\binom{n}{k}$ and enumerating all k-combinations (i.e., $k = \{2, 3, ..n\}$) would be $\sum_{2 \leq k \leq n} \binom{n}{k} = 2^n$. Moreover, as the same process has to be followed for every dimension, the computational cost of Naive becomes $O(d2^n)$. Unless the dataset is extremely small or there are very few intersections occurring, the cost of that computation would be prohibitively large.

The SweepLine Method: The Naive method has several shortcomings, and as we mentioned in the previous chapter a better alternative for addressing intersection problems exists in the form of the sweep-line algorithm. It is straight-forward to modify the classic sweep-line algorithm so that at every stopping point, it performs a comparison of all *active* regions (i.e. regions currently under the line) in a way similar to the Naive method mentioned above. This would still perform only a single pass, but it would still introduce unnecessary repetition of checks and comparisons between objects, similar to the Naive method.

In order to address this problem in a way that is efficient and fair, we had to modify the sweep-line algorithm and come up with a variant method that for the purpose of this problem we will refer to as SweepLine. Specifically, SweepLine maintains and returns all sets of intersecting regions with intersection cardinality of up to a specified k_{max} using multiple passes. The fist pass consists of the classic sweep-line algorithm that returns pairs of intersections, which are then used as the input for the second pass, which will now output all the triple ($k = 3$) intersections, which is used as input for the next pass, etc. until either k_{max} is reached or no other intersections are possible. This approach maximizes computation efficiency and avoids repeated comparisons.

The computational cost of the original sweep-line (Bentley-Ottmann) algorithm for a single dimension is $O((n + l) \log n)$, where l is the number of intersections found (in non-degenerate cases, $l \ll 2^n$). This includes the cost of sorting which is $O(n \log n)$(for one dimension) and comparisons which is $O(l \log l)$. After modifying the algorithm to support multiple intersections, the sorting cost is the same while the comparison cost becomes $l \log l$ for a single specific k, and therefore to report all possible k-combinations (i.e., $k = \{2, 3, ..l\}$) it would be $\sum_{2 \leq k \leq l} l \log l = l \log l^k$. Therefore, for all dimensions d, the computational cost of our modified SweepLine would be in the order of $O(dn \log n + d(l \log l)^k)$. Note that since $l \ll 2^n$, this represents a significant improvement over the NAIVE methods, but it still remains significantly expensive.

<u>S</u>weep <u>L</u>ine <u>I</u>ntersection <u>G</u>raph (SLIG): Although SWEEPLINE employs a state-of-the-art algorithm and produces much better results than NAIVE, it still inherits a number of limitations. For instance, intermediate results (e.g., the computation of pair-wise intersecting regions), are not well utilised; they are taken into account in subsequent computations that could speed up the whole process. Towards that end, we propose SLIG, a novel method for efficiently solving the problems of interest. The method we devise is based on the following two **key observations**:

- Current best approaches to the problem depend on an expensive process of sequentially examining regions to determine if they intersect with previously visited regions or sets of regions. However, this limitation can be overcome by constructing a *region intersection graph* data structure.
- The multiple intersection problem can be mapped to that *clique problem* (and variants of it) on the region intersection graph. This can suggest huge time performance savings, since it is possible to address the problems of interest just by operating on the region intersection graph, without the need to operate on the original regions or to re-apply a sweep-line algorithm multiple times.

We elaborate on these key observations in the next paragraphs.

Region Intersection Graph (RIG) is a graph where each *vertex* represents a region in S and each *edge* represents an intersection between two regions. In the case of 1-dimensional regions, this graph is known as an *interval graph.* Note that the idea of the intersection graph generalizes to multiple dimensions. Now, with a region intersection graph in place, it is easy to interpret intersection queries as connectivity queries in RIG. For example, if two vertices are connected in RIG, then the regions represented by the vertices are intersecting; or, obtaining all *neighbouring vertices* of a vertex in RIG is equivalent to finding all regions intersecting with a specific region, and so on. These graph operations are typically fast, while the construction of the *RIG* is straightforward. We employ the classic Bentley-Ottmann sweep-line algorithm for detecting pair-wise intersections of regions. Whenever a new region is encountered, a *new node* added to the RIG; for a pair-wise intersection, a *new edge* is added. The time complexity of this process is $O(n \log n)$ and the space complexity is $O(n)$ (due to the classic sweep-line). The space complexity of storing RIG in memory is $O(V + E)$, where $V = n$ is the set of vertices (regions) and E is the set of edges (intersections).

The clique problem refers to the computational problem of finding cliques in a graph. A *clique* is a subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent; that is, its induced subgraph is complete. To make a connection to our problem, it is well known that in the case of 1-dimensional regions (line segments), a set of pair-wise intersecting regions would have a common point or range, a *common region*, where they all overlap (see for example Figure 3.2). Effectively, these intersecting regions directly correspond to a clique in the intersection graph (see also result in [5, Lemma 3.3]). This means that, once a clique has been identified in the intersection graph, it is possible to quickly identify and report information about the common region of the multiple intersecting regions. It is possible to take this analysis further, by taking advantage of significant prior research related to cliques, and specifically the *maximal clique enumeration* problem. A *maximal clique* is a clique that cannot be extended by including an additional adjacent vertex (i.e., it isn't part of a larger clique). Enumerating only the maximal cliques can be significantly faster than enumerating all cliques, while a large number of highly optimised algorithms are available for the task [12, 23]. In our problem, a maximal clique represents a set of regions that are all intersecting with each other, or otherwise it represents the fact that there exists a *common region* among a set of intersecting regions. Once a maximal clique is identified, by definition, all subset combinations of all regions participating in the maximal clique are cliques and therefore form a common region themselves. For example, in Fig. 3.1d, $ABCD$ forms a maximal clique, meaning that ABC, ABD, ACD and BCD are also cliques.

As a result of the previous discussion, we can provide a solution to this problem by implementing an out-of-the-box state-of-the-art algorithm that enumerates all maximal cliques. The Bron-Kerbosch algorithm has a worst-case computation cost of $O(3^{n/3})$ [12], which is better than the NAIVE method and better than the *Sweep-line* method for very large values of l. However, another option is the algorithm by Tsukiyama et al. [59], which has a computational cost relative to the number of the resulting maximal cliques in the graph,

$O(n \cdot l_{max})$, where l_{max} is the number of maximal cliques. Notice that the dimensions cost multiplier d is only included in the sorting; this is because that cost would only incur during the construction of the intersection graph, but never again, suggesting large computation time savings. Therefore, the computational cost of SLIG would be $O(dn \log n + n \cdot l_{max})$.

Table 3.2: Summary of Computational Complexities

MultipleIntersect Problem	
NAIVE	$O(d2^m)$
SWEEPLINE	$O(d \cdot n \log n + d(l \log l)^k)$
SLIG	$O(d \cdot n \log n + n \cdot l_{max})$
Intersection Queries	
SINGLEREGIONQUERY	$O(m \cdot l_{max})$
MULTIREGIONQUERY	$O(m \cdot l_{max})$

3.3.2 Intersection Queries

Having constructed the region intersection graph mentioned previously, it is now possible to quickly and efficiently answer a number of different queries related to multiple intersections on the dataset that are difficult to do otherwise. In the case of the NAIVE and SWEEPLINE algorithms, the entire dataset or a substantial portion of it would need to be reexamined and the algorithms completely repeated to provide the necessary answers. Using the SLIG, however, it is possible to perform graph-based queries that only involve the necessary regions, and furthermore SLIG itself needs only be executed once, avoiding significant preprocessing cost. We examine the following queries mentioned in the problem description:

SingleRegionQuery: : Using the RIG, this problem can be easily addressed. First, given $q \in S$ we need to obtain all regions that intersect with it. This is performed by obtaining all the direct connections (neighbors) of the vertex represented by q in the intersection graph, which is a simple fast operation. Combining q and

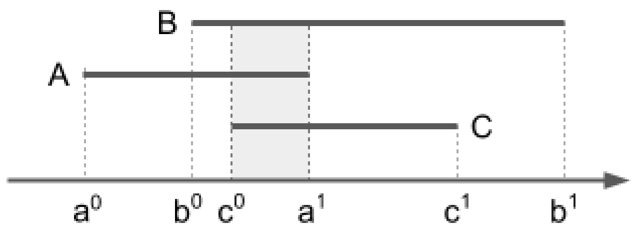


Figure 3.2: Illustrative example of 1-D line segment intersections. The shaded area is the common area $z = [c^0, a^1]$ and the common region size $z = a^1 - c^0$.

its graph neighbors into the set $Q \subseteq S$, we solve the problem by applying the clique enumeration algorithm mentioned in Chapter 2. The query rectangle q is guaranteed to only intersect with the regions with which the corresponding vertex is connected. Therefore, the computational cost would be $O(m \cdot l_{max})$, where $m = |Q|$.

MultiRegionQuery: : This can be answered in a similar way as SINGLEREGIONQUERY. We simply need to first combine all the all the nodes in Q and all their neighbors into a single subgraph. Afterwards, we need only apply once again the clique enumeration algorithm to that subgraph and filter out all the results that don't contain a region in Q. The computational cost of SLIG would be $O(m \cdot l_{max})$, with m this time being $|Q|$ plus the number of neighbors found. Except for the trivial worst case where $m \gg |Q|$, the post-processing cost should be at most $O(l_{max})$.

3.3.3 Calculation of Common Region Size

In this paragraph we describe how we can efficiently compute the *common region size* $|Z|$ of the newly defined common region Z, in multiple dimensions (i.e., the length of a line segment, surface of a rectangle/box, volume of a cuboid, and so on). We have already established that a set of regions that form a clique in RIG are defining a *common region* Z, which itself is a new region in the same dimension as the original intersecting regions (e.g., the common region of 2-D rectangles will be a 2-D rectangle). This region can be defined using the *start* and *end* coordinates in the case of 1-D line segments, the coordinates of the two opposite corners of a 2-D rectangle, and so on. First, we show how we can efficiently compute $|Z|$ in one dimension, and then we generalize to multiple dimensions.

Since the geometric objects we examine in this work are all convex sets, according to Helly's theorem [48] if all regions s_i in a set of regions I are pair-wise overlapping, then they have a non-empty intersection, i.e. they form a common region Z. Furthermore, since these regions are axis-aligned objects, it is trivial to show that the position and size of the common region can be found as such:

Corollary 1. *Let a set of k regions $S = \{s_1, s_2, \ldots, s_k\}$ that form a common region. Now, assume that for each region s_i we represent its start point as s_i^0 and its end point as s_i^1, where $s_i^1 \geq s_i^0$. Therefore $s_i = [s_i^0, s_i^1]$. Let $Z : [z^0, z^1]$ be the common region defined by regions in S. Then, the start point z^0 and the end point z^1 of Z will be given by:*

$$z^0 = max(s_1^0, s_2^0, \ldots, s_k^0), \;\; z^1 = min(s_1^1, s_2^1, \ldots, s_k^1)$$

Corollary 2. *The common region size $|Z|$ in 1-D is given by:*

$$|Z| = min(s_1^1, s_2^1, \ldots, s_k^1) - max(s_1^0, s_2^0, \ldots, s_k^0)$$

Consider the example of Figure 3.2. The common region of the intersecting regions A, B, C is shown shaded. The common region will be $z = [c^0, a^1]$ and the common region size will be $|Z| = a^1 - c^0$. For higher dimensions ($d \geq 2$), we need to repeat the same process for each dimension d and obtain its common region size, say $|z_d|$.

Corollary 3. *The common region size in higher dimensions is the product of common region sizes in each dimension:*

$$|Z| = \prod^{d} |Z_d|$$

3.4 Algorithms

In order to evaluate the performance of the different methods, we implemented the Naive algorithm, a modified version of the SweepLine algorithm that is able to accommodate the various problems of interest, and our proposed SLIG algorithm. The Naive algorithm simply consists of nested loops where each object is compared with each other and, if the two are intersecting, the latter is compared with every previous object that intersects the first one. For the SweepLine and SLIG, the implementation follows the methods outlined in Section 3.3, and the details are provided in Alg. 1 and Alg. 2, respectively. In all cases, the `intersects()` function is a simple geometric comparison in all dimensions.

Algorithm 1: SweepLine

```
Input: Set $S$ of regions
Output: Set $O$ of intersecting sets of regions, grouped by intersection cardinality $k$ in the form
        $O = \{k_2 : [\{s_1, s_2\}, \ldots], k_3 : [\{s_1, s_2, s_3\}, \ldots], \ldots\}$
Points ← sort($x^0, x^1 \; \forall \; s_i, d \leftarrow 1$)
O ← [ ], k ← 2
LastIntersects ← [ [region] ∀ region in $S$ ]
while O[k-1] not empty do
    Intersects ← GetKIntersects(k, LastIntersects)
    O[k] ← Intersects
    LastIntersects ← Intersects
    k ← k + 1
```

```
Procedure GetKIntersects(k, LastIntersects)

Actives ← [], Intersects ← {}
for point in Points do
    if point.type = start then
        Intersects[point.region] ← []
        for activeIntersect in Actives do
            if activeIntersect.intersects(point.region) then
                intersection ← activeIntersect
                intersection.append(point.region)
                Intersects[point.region].append(intersection)
        for intersection in LastIntersects[point.regions] do
            Actives.append(intersection)
    else
        for intersection in LastIntersects[point.regions] do
            Actives.remove(intersection)
```

Algorithm 2: SLIG

```
Input: Set $S$ of regions
Output: Set $O$ of intersecting sets of regions, grouped by intersection cardinality $k$ in the form
        $O = \{k_2 : [\{S_1, S_2\}, \ldots], k_3 : [\{S_1, S_2, S_3\}, \ldots], \ldots\}$
Points ← sort($x^0, x^1 \; \forall \; s_i, d \leftarrow 1$)
O ← [ ]
GetIntersectionGraph($S$)
GenerateKCliques(CliqueList,Graph)
for clique in CliqueList do
    k ← len(clique)
    for i in [2,k-1] do
        O[i].append(all_combinations(clique, i))
```

Procedure GetIntersectionGraph(S)

```
Actives ← [], Graph ← []
for point in Points do
    if point.type = start then
        for activeRegion in Actives do
            if activeRegion.intersects(point.region) then
                Graph.addEdge(activeRegion, point.region)
        activeRegion.append(point.region)
    else
        activeRegion.remove(point.region)
```

3.5 Experiments

In this Section, we experimentally evaluate the performance of our proposed method SLIG against SWEEP-LINE. As we described SWEEP-LINE serves as a sensible state-of-the-art method for addressing the problem of interest. On the other hand, the NAIVE method is not sophisticated enough and it performs at a level that is considerably inadequate for the problems, so we don't consider it in the experimental evaluation and omit any related discussion.

Before presenting the results, we provide details of the computational environment and the data sets employed.

Environment: All experiments are conducted on a PC with 8x Intel(R) Core$^{\text{TM}}$ i7-7700 CPU @ 3.60GHz and 64GB memory using Python 3.7. For each algorithm or parameter effect evaluation, we execute the algorithm ten (10) independent times and report the average execution time or other results.

Data: In order to evaluate the behavior of the algorithms under certain conditions, we had to resort to synthetic data. Towards this end, a data generator was implemented that allows to generate data sets of specific characteristics by making fine adjustments to a controlled number of parameters. We use d to control the number of dimensions. Then, we define a d-dimensional space where each dimension has size T, effectively ranging in $[0, T]$; unless otherwise noted $T = 1000$. Within a d-dimensional space, we uniformly generate n d-dimensional regions at random. The size of each dimension of a region is randomly selected from the uniform range $t : [0, t_{max}]$, where $t_{max} = r \cdot T$ and $r \in [0, 1]$ represents a ratio of the total length T. For example, if $r = 0.01$ and $T = 1000$, then the size for each dimension of a region would be bound by $0 \leq t \leq 10$. Therefore, the configurable parameters of the data generator are *number of dimensions* d, *number of objects* n and *ratio* r. For experimental evaluation purposes, various datasets were created, ranging from 10^1 to 10^5 regions and resulting in 10^2 to 10^8 intersections.

Figure 3.3 provides illustrative examples of two small sample data sets of regions in 1-D and 2-D. The parameters used are $\{d = 1, n = 100, T = 1000, r = 0.01\}$ for the 1-D and $\{d = 2, n = 500, T = 1000, r =$

0.01} for the 2-D, respectively. Moreover, we present the region intersection graphs (RIGs) that our SLIG method constructs to assist in addressing the problems of interest. The images are provided in color to help distinguish the different clusters of intersecting regions.

Experiments: We aim to evaluate the following aspects:

- **Effect of Parameters n and r on RIG** How does the number of regions n and the size of regions (as defined by the ratio r) affect the properties of the region intersection graph (RIG)? In particular, what is the size (number of edges) and overall level of clustering (as measured by the average clustering coefficient) of the RIG obtained, in different dimensions?
- **SLIG Comparative Performance** How does our proposed SLIG method compare to the SWEEPLINE method for the MULTIPLEINTERSECT problem?
- **Effect of RIG Topology on SLIG** How does the structure of the RIG influence the performance of SLIG?
- **SLIG Query Performance** How does SLIG perform when utilized to answer SINGLEREGIONQUERY and MULTIREGIONQUERY?
- **SLIG Flexibility** SLIG has a number of real-world applications in various domains, as explained in the Chapter 1. To demonstrate the easiness of adapting our method to new datasets and scenarios, we apply SLIG to a real-world dataset, not typically discussed in the context of intersections.

3.5.1 Effect of Parameters n and r on RIG

The region intersection graph RIG constructed by our method SLIG depends on the number of regions that are intersecting in the data set. In principle, the number of intersections depend on two parameters: the number of the regions n in the data set and the size of these regions, as determined by the ratio r. Therefore, in this experiment we aim to examine the effect of these parameters to the properties of RIG. For the first set of experiments, we set the ratio to be $r = 0.01$ and vary the number of regions n. The results can be seen in Figures 3.4a and 3.4b. For the second set of experiments, we fix $n = 1000$ and vary the values of the ratio r. The results can be seen in Figures 3.4c and 3.4d. In all instances, since the generated regions are generated uniformly at random with a region size $t : 0 \leq t \leq t_{max}$ in each dimension, the average region size in each dimension corresponds to $\frac{t_{max}}{2}$. As can be seen in these figures, the size of the RIG increases linearly with both n and r ($O(n \cdot r)$). That is to be expected, as in both cases the probability of any two regions intersecting is increasing with the relative size of the region in the d-dimensional space. On the other hand, the average clustering coefficient's value remains low until a critical point is reached where the RIG is no longer sparse. After that, the coefficient's value increases logarithmically, until the graph becomes very dense, where eventually it plateaus out. This happens because, while the RIG is sparse, the number of its edges is relatively small compared to the total number of edges possible (i.e., the likelihood of triangles forming is low). However, as the number of edges in the RIG increases faster than the total number of edges possible, more and more triangles form. Eventually, the graph becomes saturated when the average clustering coefficient approaches a value of 0.8. Similar trends are demonstrated for all dimensions reported $d = \{1, 2, 3\}$. However, the higher the dimension, the more sparse the RIG would be. This is because regions need to intersect in all dimensions simultaneously to be actually intersecting and with more dimensions considered, the probability of any two regions intersecting decreases.

3.5.2 SLIG Comparative Performance

We evaluated the time performance of SLIG against the Sweep-line method, as a function of the number of regions n in the dataset. Based on the parameter sensitivity experiments presented earlier, we set the parameter values as follows: $d = 2$, $r = 0.01$. These values provide a balance between generating a sufficiently large number of intersections, but at the same time avoiding degenerate cases where most regions overlap with each other. Figure 3.5a presents the results for the MultipleIntersect problem. It is apparent from the results that our SLIG algorithm is **multiple orders of time faster** than Sweep-line. Note that the Sweep-line algorithm would require an estimated *one day* to process the $5 \cdot 10^4$ regions and an estimated *eleven days* for processing 10^5 regions, therefore the exact elapsed time is not reported (estimated times reported using dashed lines).

3.5.3 Effect of RIG Topology on SLIG

We have already established that the parameters n and r will have an important effect to the characteristics of the RIG, since they effectively control the number of intersections. However, it is also well known that networks of the same number of nodes and edges can exhibit different network topology, as demonstrated by different overall average clustering coefficients. For the RIG specifically, different network topologies effectively correspond to different distributions of the regions in the original space. This observation is essential to the MULTIPLEINTERSECT problem, since the number of cliques to be enumerated is directly related to a network's average clustering coefficient. In order to evaluate the impact of the network topology, we designed experiments with networks generated by using the Watts—Strogatz small-world model [60]. This model employs a re-wiring probability $p \in [0, 1]$ that can control the overall clustering of the network of specific number of nodes and edges. Small values of p will result in networks with high clustering coefficient and high values of p will result in networks with low clustering coefficient, respectively. Figure 3.5b reports on the time performance of SLIG in networks of different size (each network has the same number of nodes $n = 10^5$, but different number of edges $[10^5, 2x10^5, 5x10^5, 10^6, 2x10^6, 5x10^6, 10^7]$) and for varying values of re-wiring probability $p = \{0.1, 0.5, 0.9\}$. For each network, it can be seen that as the value of p decreases (i.e., higher clustering), the algorithm requires significantly more time to process the regions, since more cliques (of greater size) are present in the graph. For the most extreme values of $p = 0.1$ and 10^7 edges, large sections of the graph become fully connected, resulting in exponentially many cliques, which led to the available memory being exceeded.

3.5.4 SLIG Query Performance

We evaluated the time performance of SLIG when performing MULTIREGIONQUERY and SINGLEREGION-QUERY. For the needs of the MULTIREGIONQUERY problem we set the number of query regions to be $|Q| = 0.1 \cdot n$ (i.e., it always represents 10% of the total number of regions in the dataset n). This guarantees that the evaluation covers queries with various requirements regarding the volume of the returned results. The time performance of those queries can be seen in Figure 3.5c, where it is apparent that MultiRegionQuery executes much slower, although both are completed in < 1 second, even in the case of 10^5 regions.

3.5.5 Real-world experiment

To evaluate the SLIG algorithm's performance and demonstrate its usability, we applied it to *real-world historical climate data* coming from three different sources: *WorldClim.org*, *ECMRWF*[2] and *NWS*[3]. The data represents aggregated seasonal values over the period of 1980-1990 and are grid-based — data values correspond to cells on a global coordinate grid. We consider these cells as the input 2-*D regions* of our problem. Note however that each service is using a different coordinate system (10 arc-min, 45 arc-min and HRAP, respectively), thus the cells/regions do not match. In reality, these regions would represent separate sets of *overlapping rectangles.* Now, we would like to use these data to identify areas that have been through *different types of extreme weather conditions in that decade.* According to the NYC Emergency Services, extreme weather conditions for California, US can be of the following types:

- HOT: Extreme heat, season mean temperature $> 30°$ C
- COLD: Freezing cold, season mean temperature $< 0°$ C
- RAIN: Extreme rain, season total precipitation > 400mm
- DROUGHT: Drought, season total precipitation < 50mm
- WIND: Extreme wind, season mean speed > 3.6km/h

So, we filter out any regions that never experienced any extreme weather and keep only the regions that satisfy at least one of these types. That way, we obtained 5 sets of regions, one set per type. Each region

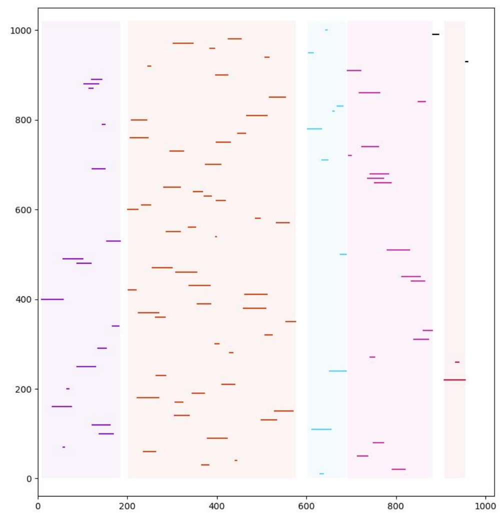
[2]https://apps.ecmwf.int/datasets/data/interim-full-moda/levtype=sfc/

[3]https://www.ncdc.noaa.gov/swdi

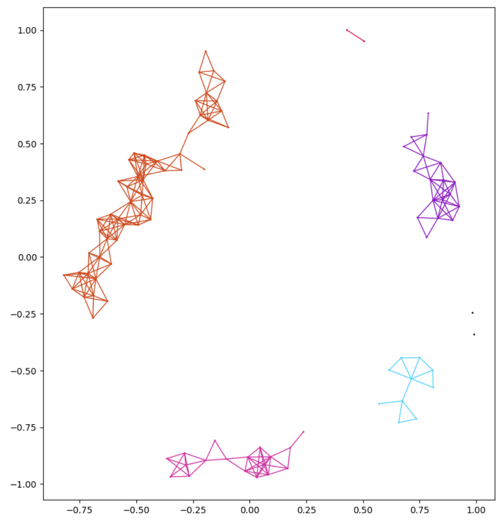
represents the geographic area where an extreme weather condition occurred. The areas where these regions overlap correspond to the areas we are looking for (areas that had several different types of extreme weather). For example, regions corresponding to HOT, WIND and RAIN overlap ($k = 3$), have been through at least one heatwave season, one extreme wind season and one extreme rain season. A visualization of these regions can be seen in Figure 3.5. SLIG can rapidly identify these areas, and also report other interesting findings as can be seen in Table 3.3. For example, the 4 overlapping areas that had experienced 4 different types of extreme weather conditions in that decade correspond to Reno, Nevada.

Table 3.3: Extreme weather region overlaps

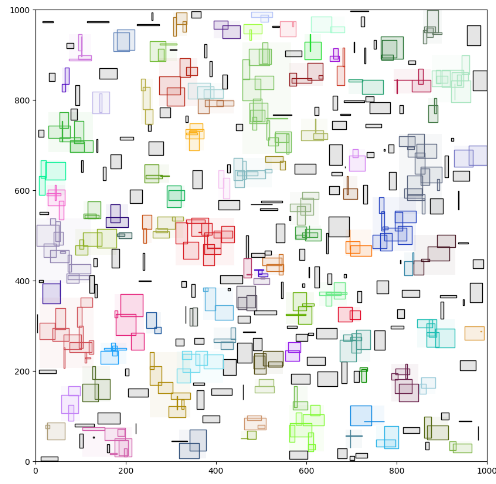
Property	**Value**
#intersections, $k = 2$	5841
#intersections, $k = 3$	2658
#intersections, $k = 4$	4
most frequent combination	COLD & DROUGHT (2338#)
highest total overlap	DROUGHT & WIND $(352,304km^2)$



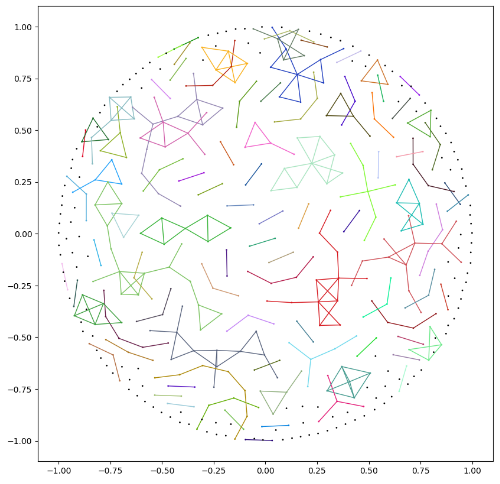
(a) Sample 1-*D* regions



(b) RIG of the regions in (a)

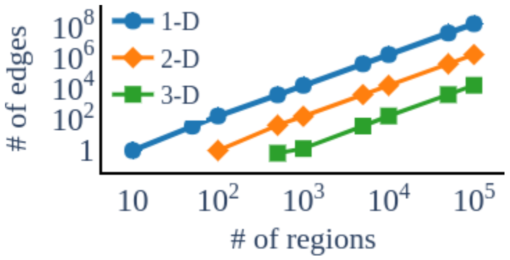


(c) Sample 2-*D* regions

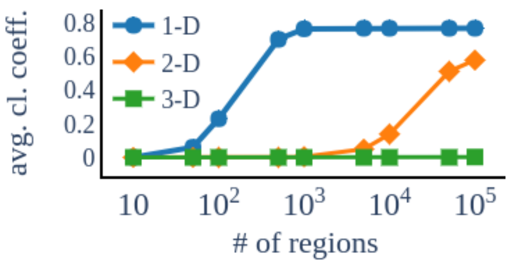


(d) RIG of the regions in (c)

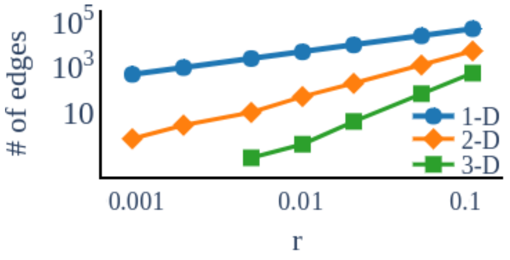
Figure 3.3: Sample 1-*D* and 2-*D* regions created by our synthetic data generator (left). The resulting region intersection graphs (RIG) of the sample regions are also shown (right).



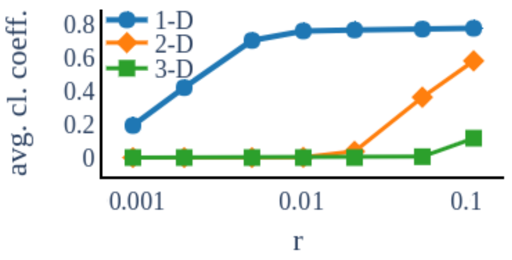
(a) #edges × #regions



(b) Avg cl. coeff. × #regions

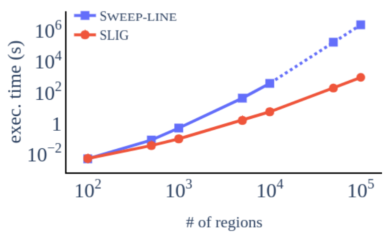


(c) #edges × region size

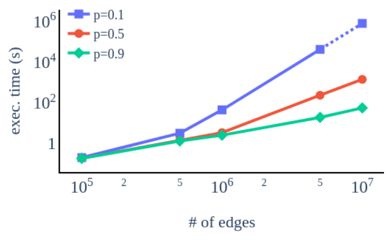


(d) Avg cl. coeff. × region size

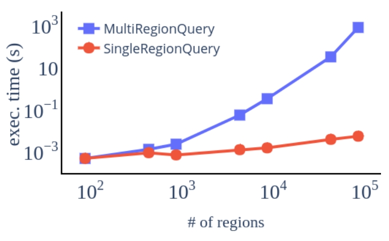
Figure 3.4: Effect of the number of regions and region size on the region intersection graph (RIG) size (number of edges) and overall level of clustering (average clustering coefficient), for different dimensions.



(a) SLIG vs. Sweep-line



(b) RIG Topology Effect



(c) SLIG Query Performance

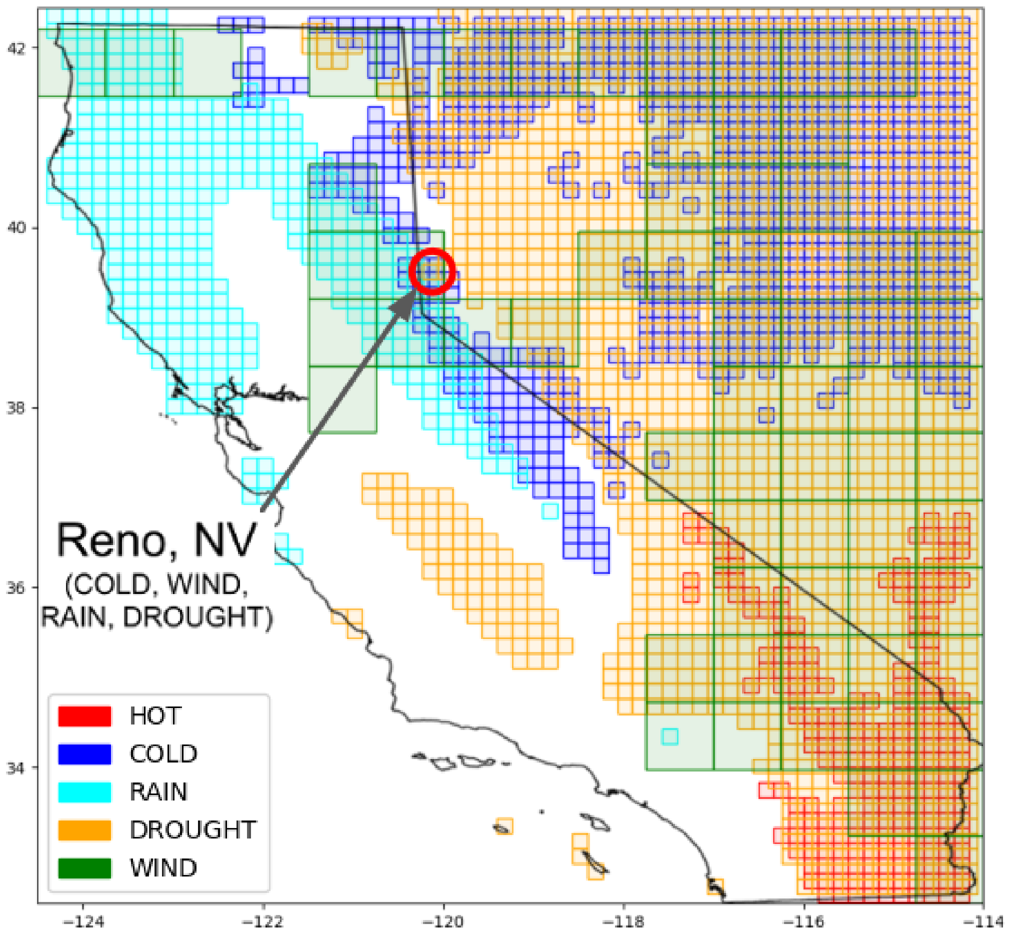


Figure 3.5: Overlapping areas of extreme weather in CA, US

4 Mining of Node Importance in Trajectory Networks

4.1 Motivation

Advances in location acquisition and tracking devices have given rise to the generation of enormous trajectory data consisting of *spatial* and *temporal* information of moving objects, such as persons, vehicles or animals [62]. Mining trajectory data to find interesting patterns is of increased research interest due to a broad range of useful applications, including analysis of transportation systems, location-based services, and crowd behavior analysis [51, 27, 54, 63]. Concepts from the fields of trajectory data mining and graph theory can be combined and, with the help of the object intersection problem and the sweep-line algorithm, can be used in *mining the importance of moving objects*, represented as nodes in a network.

4.1.1 Trajectory Mining

Mining trajectory data to find interesting patterns is of increased research interest due to a broad range of useful applications, including analysis of transportation systems, location-based services, and crowd behavior analysis [51, 27, 54, 63].

As multiple objects are continuously moving in an area they can be found in close proximity (spatial distance) to each other, forming *contacts*. A *contact* between two objects can be seen as an *event* that lasts for as much as their spatial distance remains consistently smaller than a specified proximity threshold. Therefore, any event has a duration, simply defined as the amount of time that passes from the beginning of the contact until it ends. Note that any two objects might encounter each other multiple times, leading to the same contact occurring multiple times. So, while each event is unique, multiple events might refer to the same contact of two objects, occurring at different times.

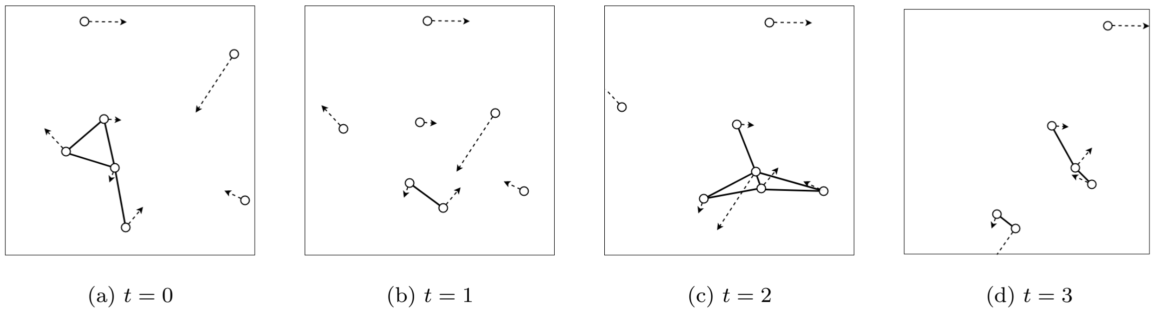

(a) $t = 0$ (b) $t = 1$ (c) $t = 2$ (d) $t = 3$

Figure 4.1: Example evolution of the system over discrete times. As objects move over trajectories they form contacts with other objects they are in close proximity to. The collection of contacts at time t represents a proximity network; over a period of time $[0, T]$ it represents a temporal network, or *trajectory network*.

4.1.2 Proximity and Trajectory Networks

At any point in time t, we can consider *a proximity network*, defined as a graph with nodes representing objects and edges representing contacts that are currently active, i.e. the objects are currently in close proximity. The proximity network satisfies particular distance requirements and its topology depends on the proximity threshold employed. Given a proximity threshold, the trajectories of the moving objects continuously form new contacts, while other contacts are dissolved, leading to a collection of proximity networks. These networks can be understood as a dynamic network the topology of which is changing over time, *a temporal network*. We refer to this temporal network defined by the trajectories of moving objects over time as *a trajectory network*.

4.1.3 Node Importance

The concept of node importance has been rigorously studied in the case of *static graphs*, due to its numerous applications. These include measuring the influence of individuals in a social network, understanding the role of infrastructure nodes in transportation networks, urban networks, the Internet, or assimilating the role of a given node in spreading a contagious disease, to name a few.

Most of the metrics employed to characterize the importance of a node in a static graph depend on the number of direct connections that this node has to other nodes in the graph (i.e., the node degree). In the case of a trajectory network, these connections are evolving over time, therefore the definition of node importance needs to be redefined. For example, the number of unique connections or the average number of connections over time might be of interest. In addition, while existence (or not) of a node's connections remains important, the temporal dimension adds more quantities of interest that need to be examined, including metrics of *frequency* and *duration* of these connections. These metrics can be equally important in understanding a node's role in the network and being able to evaluate them allows for an overall more comprehensive analysis.

While the temporal dimension adds richness to the analysis, it also adds to the complexity of the graph representation and demands for efficient methods for evaluating the metrics of interest. The naive approach to the problem requires to evaluate the various node importance metrics over a number of *proximity graphs* (assuming observation time is discretized), using traditional static graph algorithms and then aggregating quantities in a meaningful way. This is in addition to applying expensive trajectory similarity methods over multiple points to construct the proximity network at each point in time. There are also streaming versions of algorithms that focus on efficient computation of single network metrics over time. In contrast to these approaches, we devise a novel method that is able to simultaneously evaluate a number of network metrics of interest for all moving objects (i.e., all trajectories), over time. Our proposed method is based on two phases. First, it efficiently computes contacts of moving objects over a period of time and represents them as a set of time intervals. Then, the sweep-line algorithm we presented in Chapter 2 is used to evaluate the metrics of interest, all at once, by efficiently processing the sets of time intervals of contacts. In the cases we present, moving objects have either constant velocities or follow random trajectories while other parameters that can dramatically affect the results and the algorithms' performance are considered, such as the proximity threshold τ. Furthermore, our method is utilized to evaluate the membership of an object to interesting network motifs, allowing for a more comprehensive clustering analysis of objects over time.

4.2 Related Work

To better understand the intuitions behind more advanced concepts utilized in this project, a review of core literature is pertinent. We have already outlined the details of the interval intersection problem and the sweep-line algorithm. Apart from those, our work is related to *trajectory data mining*, *dynamic* and *temporal networks*. Here we present a more comprehensive view of existing work on these topics.

4.2.1 Trajectory Data Mining

Trajectory problems have been extensively studied in the data mining area. Of particular interest are problems related to similarity in trajectory data [36, 56] and trajectory clustering [38]. While there is research on the behavior and trajectories of moving objects [21], we wish to focus on the interactions between them, and specifically the way objects come in close proximity to each other and the groups they form. This is roughly similar to the concept of contact networks commonly studied in epidemiology and infection spreading, although it's not strictly related to spatial proximity [50].

4.2.2 Spatial Networks

The notion of objects distributed in space and interacting with each other has been heavily explored in graph theory. Graph theory concepts such as proximity graphs [26] and geometric intersection graphs [24] are characteristic examples of this. Specifically, several variations of proximity graphs have been developed over time to better fit different problems. For instance, relative neighbor graphs [57] and Gabriel graphs [26] connect nearest neighbors if no other vertexes are nearby, while Delaunay triangulations [19] maximize the minimum angles of all triangles formed. These, however, mostly deal with static data, while our goal is to examine cases of proximity graphs where all objects/nodes are moving and their relationships are evolving over time.

4.2.3 Temporal Networks

There has also been significant research on networks that are evolving over time, or temporal networks. The nature of these systems introduces a number of issues and obstacles, which make it necessary to adapt for the problem basic graph theory concepts and algorithms such as shortest paths [61], motifs [35] and other metrics [44]. Furthermore, with the addition of temporal information, several concepts can be extended to take advantage of the additional data. Examples of this are the temporal node centrality [34] and the network reachability [30] metrics. As part of this project we wish to develop techniques that provide accurate values for these metrics in a fast and accurate way for all nodes, with detailed results over time, then apply these for the analysis of trajectory temporal networks. We intend to make use of the node degree and reachability or connectedness metrics, as well as examine the participation of every node in triangles over time. Earlier research on this topic, including *network temporal motifs*, focused mostly on faster approximations for streaming graphs [13, 6, 58]. An accurate triangle counting algorithm without approximation exists [46], but merely *counts* all triangles rather than *enumerating* them.

Table 4.1: Summary of Notations

Notation	**Description**
$[0, T]$	Observation time interval
$\mathscr{N}$	A set of moving objects
$\mathbf{P}_i$	Trajectory of an object i
(x, y, t)	Coordinates of an object at time t
τ	Proximity threshold
$d_{u,v}$	Spatial distance of u and v
$c_{u,v}$	Contact of u and v
$e_{u,v}$	Event about a contact between u and v
Δt	Duration of an event
$G_{[0,T]}(V, E)$	Trajectory network of V nodes and E edges
V_t	Set of nodes at time t
E_t	Set of edges at time t
$G_t(V_t, E_t)$	Proximity network at time t

4.3 Definitions and Preliminaries

Consider a set of objects $\mathscr{N} = \{1, 2, \ldots, N\}$ moving in the Euclidean plane $\mathbb{R}^2$ for a finite *observation time interval* $[0, T]$, forming trajectories $\mathbf{P}_i, i \in \{1, 2, \ldots, N\}$. As the objects are continuously moving, they can at times encounter each other, forming *contacts*.

Definition 1. *(Contact) A* contact *c between two moving objects $u, v \in \mathscr{N}$ occurs when the physical proximity (spatial distance) $d_{u,v}$ of the two objects is smaller than or equal to a threshold τ ($d_{u,v} \leq \tau$). A contact is represented as a pair of nodes $c_{u,v} = (u, v)$.*

Several approaches can be used to estimate the spatial distance of two points in Euclidean plane. In this work, we employ its simplest form, the Euclidean distance, given by:

$$d_{u,v} = \sqrt{(x_u - x_v)^2 + (y_u - y_v)^2}$$

where (x_u, y_u) and (x_v, y_v) are the spatial coordinates of objects u and v at a time t, where $0 \leq t \leq T$ respectively.

As mentioned earlier, a contact is considered *active* for as much as the spatial distance between the two objects u and v remains consistently smaller than a proximity threshold τ. To better describe this concept, we introduce the concept of an *event*.

Definition 2. *(Event) An* event *e occurs when two moving objects $u, v \in \mathscr{N}$ form a contact $c_{u,v}$ and is represented by a pair $e_{u,v} = (c_{u,v}, [t_s, t_e])$, where $c_{u,v} = (u, v)$ and $[t_s, t_e]$ is a time interval between the time at which the contact became active t_s and the time that the contact dissolved t_e. We also refer to the times t_s and t_e as endpoints of an event, the one representing the starting point and the other the ending point of the event e. Endpoints are critical for our proposed methods. For simplicity, we sometimes represent an event as a quadruple $e_{u,v} = (u, v, t_s, t_e)$. An event has also a duration $\Delta t = t_e - t_s$.*

Note that, in our setting, we do not preclude the case that two objects contact each other multiple times over the observation time interval $[0, T]$. In this case, a contact between two moving objects u and v is represented by a sequence of events $E_{u,v} = \{e^1_{u,v}, \ldots, e^n_{u,v}\}$ or $E_{u,v} = \{(c_{u,v}, [t^1_s, t^1_e]), \ldots, (c_{u,v}, [t^n_s, t^n_e])\}$, and the respective durations of the events as a set $E^{\Delta t}_{u,v} = \{\Delta t^1, \ldots, \Delta t^n\}$.

In this work, we employ a universal proximity threshold τ, so the contacts will always be *reciprocal*, meaning that $(c_{u,v}, [t_s, t_e])$ is equivalent to $(c_{v,u}, [t_s, t_e])$. While we can assume that the reciprocity property is valid in many applications (e.g., vehicle-to-vehicle proximity, human-to-human proximity, to name a few), there are interesting cases and applications where the reciprocity property might not always be satisfied. For example, if proximity is defined as the ability of a node to perceive another object, then we can assume that nodes u and v might have a different degree of that skill, and therefore $(c_{u,v}, [t_s, t_e])$ is not necessarily equal to the $(c_{v,u}, [t_s, t_e])$. These cases are out of the scope of this work.

4.3.1 Trajectory Networks

Monitoring the physical proximity of moving objects, can be represented as a *trajectory network*. Formally, a trajectory network $G_{[0,T]}$ defined in an *observation time interval* $[0, T]$ consists of a set of vertices $V_{[0,T]}$ and a set of edges $E_{[0,T]}$. It is easy to see that $V_{[0,T]}$ represents all moving objects $\mathscr{N}$ and $E_{[0,T]}$ represents all the *events* that occurred in $[0, T]$.

A trajectory network is inherently dynamic and can also be thought of as a *temporal network*, also referred to as a *time-varying network*. Most characterizations of temporal networks discretize time by converting temporal information into a sequence of n network "snapshots". We use w to denote the time duration of each snapshot (time window size), where $w = T/n$, expressed in some time unit (e.g., seconds, minutes, hours, etc.). For simplicity we assume that $w = 1$. In other words, a temporal network can be represented as a series of static graphs G_1, G_2, …, G_n. The notation $G_t(V_t, E_t)$, $\forall t \in \{1, 2, \ldots, n\}$ represents the temporal network snapshot at time t, where V_t, E_t are the sets of vertices and edges at time t, respectively. It is easy to see that $G_t(V_t, E_t)$ represents a proximity network at time t, where V_t represents moving objects and E_t represents all active contacts at time t.

4.3.2 Edge Stream Representation of a Trajectory Network

A trajectory network $G_{[0,T]}$ can be represented as an *edge stream* —a sequence of all events $e \in E_{[0,T]}$ ordered by their starting time t_s. For example, if $E_{[0,T]}$ has the following events: $\{(u_1, u_5, \underline{3}, 7)$, $(u_2, u_7, \underline{2}, 4)$, $(u_1, u_2, \underline{5}, 7)\}$, then the edge stream appears as follows: $\{(u_2, u_7, \underline{2}, 4)$, $(u_1, u_5, \underline{3}, 7)$, $(u_1, u_2, \underline{5}, 7)\}$. If two events start at the same time, their ordering is considered arbitrary. Our algorithms assume that the edges are given in chronological order; if not, they can be sorted in $O(mlogm)$ time, where $m = |E_{[0,T]}|$. The *edge stream* is a natural way to represent a trajectory network, e.g. representing walking individuals over time, vehicles moving in city over time, and more. The edge stream of G can be modeled as a set of n continuous line segments (i.e, event time intervals) with freely defined starting and ending points placed along an horizontal axis representing time (see Fig. 4.2).

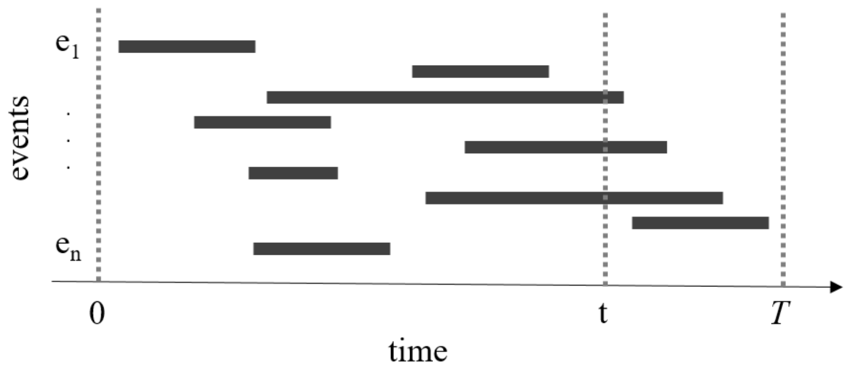


Figure 4.2: A set of continuous line segments representing time intervals can be used to model the events that occur in a trajectory network.

4.3.3 Node Importance in Trajectory Networks

In this paragraph, we define metrics that relate to the temporal importance of nodes. Note that we abstain from the term *node centrality* to refer to node importance that is common in static network analysis. This is because measures of node centrality in the traditional setting of a static network are commonly based on *shortest paths* (e.g., betweenness centrality [43, 11]), but shortest paths in temporal networks take a different character [33]. For example, in [61], the authors define *minimum temporal paths* to capture the different characterizations of time-constraint shortest paths including cases of earliest-arrival paths, latest-departure paths, or fastest paths. It is possible to evaluate a notion of temporal betweenness [34], but in our setting, we focus on more versatile notions of importance that are critical in the context of trajectories and network-based trajectory analysis. That includes metrics that relate to the *trajectory node degree*, *trajectory node connectedness*, and *trajectory node triangle membership*, as described below. In addition, we describe global metrics that offer insights into the state of the observed system of trajectories, over the observation time period $[0, T]$, including descriptive analysis of the number of events per time and space and network community profiling analysis.

Definition 3. *(Trajectory Node Degree) We generalize the concept of a node importance to that of* node profiling *in trajectory networks. For a moving object $u \in \mathscr{N}$, we assume that the object might appear and disappear during the n time units of the observation time interval $[0, T]$. We represent the γ sequences of the continuous periods of presence as: $\Gamma_u = \{[t_a^1, t_\omega^1], \ldots, [t_a^\gamma, t_\omega^\gamma]\}$. Each appearance i of an object u spans $|\Gamma_u^i| = (t_\omega^i - t_a^i + 1)$ time units, and the total number of observation time units T_u of an object u will be $T_u = \sum_{i=1}^{\gamma} |\Gamma_u^i|$, where $T_u \leq n$. Then, we define the following metrics:*

- *C_u: a set of all the contacts of u during the observation time interval $[0, T]$.*
- *deg_u^{Max}: the maximum number of concurrent contacts at some time t, $0 \leq t \leq T$.*
- *deg_u^{Min}: the minimum number of concurrent contacts at some time t, $0 \leq t \leq T$.*
- *$deg_u^{Avg} = \frac{\sum_1^{|\Gamma_u|} \sum_{i=1}^{|\Gamma_u^i|} deg_u^t}{T_u}$: the normalized mean temporal node degree, where deg_u^t is the degree of u at time t and $t \in [t_a^i, t_\omega^i]$, $i = \{1, \ldots, \gamma\}$.*
- *$D_{deg_u}(k)$: the time series that represents the fraction of the time $[0, T_u]$ that u has node degree k.*

In a static undirected graph, a node v is *reachable* from a node u if there is a sequence of adjacent nodes (i.e., *a path*) starting at u and ending at v. In addition, a *connected component* is a subgraph of an undirected graph in which any two nodes are reachable to each other. Node reachability and connectedness are important since they allow to characterize the topology of a network and to investigate the dynamics of processes occurring in it. In the case of a trajectory network $G_{[0,T]}$, node adjacency is a function of time t and a proximity threshold τ, therefore the concepts of reachability and connectedness need to be redefined.

Definition 4. *(Trajectory Node Connectedness) Two nodes u and v are* reachable *at time t if there is a sequence of adjacent nodes connecting them in the proximity network at time t. Similarly, a* connected component *cc is a subgraph of the proximity network at time t in which any two nodes are reachable to each other. We also define as* node connectedness *cc_u^t of u the connected component containing u at time t. Then, we can define the following metrics:*

- *CC_u: a set of all the connected components that contained u during the observation time interval [0,T].*

- $D_{CC_u}(k)$*: a time series of the size of connected components that represents the fraction of the time* $[0, T_u]$ *that* u *is a member of components of size* k*.*

In a static undirected network, the clustering coefficient of a node u is a fundamental measure that quantifies how close its neighbours are to being a fully connected clique. To compute it, all that is needed is to count the number of *triangles* that include the particular node u in the network, where a triangle is a set of three nodes u, v, w such that (u, v), (v, w), (u, w) are edges in the graph. In the case of a trajectory network $G_{[0,T]}$, the number of triangles incident on a particular node u is a function of time t and a proximity threshold τ, therefore the concept of a triangle and membership in a triangle needs to be redefined. Duration of membership to each triangle is also imported.

Definition 5. *(Trajectory Node Triangle Membership) A node* u *is a member of a triangle* $\{u, v, w\}$ *at time* t *if there are nodes* $u, v, w \in V_t$*, such that* (u, v)*,* (v, w)*,* $(u, w) \in E_t$ *in the proximity network* G_t*. Then, we can define the following metrics:*

- $\lambda_u^{G[0,T]}$*: the number of triangles during the observation time interval* $[0, T]$ *that* v *is a member of.*

- $D_{\lambda_u}(k)$*: a time series that represents the fraction of the time* $[0, T]$ *that* u *is a member of* k *triangles.*

It is easy to see that a number of *global trajectory network analytics* are possible in a post-processing phase. For instance, it is possible to perform an enumeration of all the connected components, along with the time duration of each, during the observation time interval $[0, T]$ leading to a complete *network community profiling* [40] analysis for the trajectory network. We provide some example analysis of this in the experimental evaluation section.

4.3.4 The Problem

In this work, we are interested in mining the network importance of moving objects in trajectory networks. In the previous section, we have explained how the semantics of network importance had to be redefined to consider the spatio-temporal notion of node degree, node connectedness and node membership in triangles. We collectively refer to the problem of interest as *the Moving Object Network Profiling problem*, or simply *MONetPro*, and we formally define it as follows:

Problem 1. *(MONetPro) Given the trajectories* $\mathbf{P}_i$*,* $i \in [1, \ldots, N]$ *of* $\mathscr{N}$ *moving objects, an observation time interval* $[0, T]$ *and a proximity threshold* τ *that defines a* non-negligible contact *between two objects, compute the metrics that define:*

(i.) the trajectory node degree of each object.

(i.) the trajectory node connectedness of each object.

(i.) the trajectory node triangle membership of each object.

4.4 Methodology

The following sections depict the process by which the proposed method is implemented and test. The research problem is outlined, followed by the processes used for constructing a trajectory network. Finally, we present methods that, given a trajectory network, address the subproblems of Problem 1.

4.4.1 Construction of the Trajectory Network

Given a set of trajectories of moving objects, the first task is to construct the trajectory network. The result of this process is an edge stream representation of a trajectory network — a sequence of all events $e \in E_{[0,T]}$. Typically, the construction of the trajectory network requires considerable time. This time depends on the type of motion of the objects that is expected/allowed. Below we cover the case of *random trajectories* and the case of *trajectories of constant velocities.* Both have interesting real-world applications.

Random Trajectories: In the general case, a trajectory is a random walk in the Euclidean space. In that case, in order to construct the trajectory network $G_{[0,T]}(N, E)$ (without assuming any advanced approximation method) we need to compute the distances between all pairs of objects $(u, v) \in V_t \times V_t$ that are present at time t. If the distance is less than or equal to the proximity threshold τ, then an edge is added to the trajectory network that connects the two nodes at that time. The process is continued for subsequent times, eventually finding all events $e \in E_{[0,T]}$. The computation cost of this process for the entire observation time [0, T] is $O(T \cdot |V_t|^2)$.

Trajectories of Constant Velocity: In many applications, we can assume that objects are moving with constant velocity (i.e., constant speed and direction), forming trajectories that can be represented as straight lines in the Euclidean space. In that special case, the position of the objects as a function of time can be described using a linear equation. This case is interesting because we can resort to an algebraic way of determining *whether* and *when* an edge between two moving objects exists. In practice, we need to determine the *switching times* when the distance of each pair of objects is smaller or larger than the proximity threshold

τ, without the need to compute their distance again and again for each time unit. Formally,the position of an object i moving with constant velocity, as a function of time $t \in [0, T]$ can be described by the following two equations:

$$p_{x,i}(t) = s_{x,i} \cdot t + x_{0,i}$$

$$p_{y,i}(t) = s_{y,i} \cdot t + y_{0,i}$$

where $s_{x,i}$ is the speed of object i and $x_{0,i}$ is its initial position in the x axis. Similarly, $s_{y,i}$ is the speed of object i and $y_{0,i}$ is its initial position in the y axis. Then, we can express the distance between two points a and b as a function of time, as:

$$d_{a,b}(t) = \sqrt{(p_{x,a}(t) - p_{x,b}(t))^2 + (p_{y,a}(t) - p_{y,b}(t))^2}$$

where $d_{a,b}(t)$, is the distance of points a and b over time. Now, the solutions (if any) of this quadratic polynomial for $d_{a,b}(t) = \tau$ are the times when a contact is formed or dissolved. This process requires a single distance calculation between any pair of objects $(u, v) \in V \times V$, so the required time is $O(|V|^2)$.

4.4.2 Methods for Moving Object Network Profiling

Given a representation of all events over time that define t proximity networks G_t, $t \in [0, T]$, we need to compute the metrics of interest that define the subproblems of Problem 1. Towards this end, we present three approaches: a *naive* approach of applying standard graph algorithms on every proximity network G_t, a *streaming approach* of computing metrics over a stream of edges, and our proposed method that its key idea is based on applying a $\underline{\text{S}}$weep $\underline{\text{L}}$ine $\underline{\text{O}}$ver $\underline{\text{T}}$rajectories (*SLOT*). A sweep line algorithm [53] is an algorithmic paradigm that uses a conceptual sweep line to solve various problems in Euclidean space — it is one of the key techniques in computational geometry.

Naive Approach: The naive way to address the problem is to first construct a set of all the proximity networks $G_t(V_t, E_t)$ for each time unit $t \in [0, T]$. Then, visit each G_t independently and compute the metrics of interest for each node $u \in V_t$ by applying standard graph algorithms on static graphs. Once all networks have been examined, a post-processing step is required that would collect and aggregate the independent results in order to compose the final results. The post-processing phase could be dropped if the G_t networks are examined in a temporal order. In that case, it is possible to update the metrics of interest on-the-fly (progressively) as they are computed in subsequent proximity networks. But, the naive approach has serious drawbacks. First, it will be very inefficient, because each proximity network still needs to be constructed for every time t. Then, the standard algorithms need to be run for T time units, so computations would grow linearly to the size of the observation period $[0, T]$. Given a trajectory network $G_{[0,T]=(V,E)}$, the worst-case computational complexity of the different subproblems is as follows:

- Trajectory Node Degree: $O(T \cdot (|V_t| + |E_t|))$. Every vertex and every edge of each proximity network G_t need to be explored in the worst case, for $t \in [0, T]$.

- Trajectory Node Connectedness: $O(T \cdot (|V_t| + |E_t|))$. The connected components of a proximity network G_t can be found by applying a breadth-first or depth-first search algorithm, which are known to have a computational complexity of $O(|V_t| + |E_t|)$ in worst case [17]. They need to be applied for every $t \in [0, T]$.

- Trajectory Node Triangle Membership: $O(T \cdot (|V_t|^3))$. The trivial approach of counting the number of triangles in a proximity network G_t is to check for every triple $(u, v, z) \in \binom{|V_t|}{3}$ if nodes u, v, z form a triangle. This procedure has a worst-case complexity of $O(|V_t|^3)$. Faster algorithms are also known for finding and counting triangles that rely on fast matrix product and have a computational complexity of $O(|V_t|^{\omega})$, where $\omega < 2.376$ [37, 2]. They need to be applied for every $t \in [0, T]$.

Streaming Approach: Since a trajectory network can be represented as an edge stream, an alternative approach to evaluating the metrics of interest is to consider streaming versions of the graph algorithms. The main idea is that at each time $t \in [0, T]$ some of the metrics of interest are computed and this information is carried over to subsequent time units. As a result, unnecessary computations are dropped. The main drawback of the streaming approach is that computations still have to take place at every t. Streaming algorithms would have a similar worst case computational complexity to static algorithms examined in the naive approach. It is easy to see that if one considers the case where all vertices and edges are becoming available in a single time t. As we show in Section 4.6, in practice, the streaming algorithm would always outperform the naive approach, since it only needs to account for the updates that occur among subsequent times, but its cost would still be dominated by the requirement to run for each $t \in [0, T]$. In Section 4.6 we only experiment with streaming versions for computing the trajectory node degree related metrics, but skip streaming algorithms for more advanced metrics that are not readily available. These are adequate to demonstrate the relative performance of streaming versions to the naive method and our proposed methods.

Sweep Line Over Trajectories (SLOT): We can further improve the performance of the streaming algorithms by avoiding a large number of unnecessary computations. The key idea of our proposed method is that even if an event $e_{u,v} = (u, v, t_s, t_e)$ has a duration of $\Delta t = t_e - t_s$ time units, the metrics of interest for each node need only be updated at the **endpoints** t_s and t_e of each event $e_{u,v}$. Recall that the endpoints define when an edge in the network (i.e., a contact) is created or dissolved. When the time intervals (i.e., durations) of multiple events that involve the same node u are overlapping (it is easy to imagine such cases in the example of Fig. 4.2), then we need to consider these events simultaneously and inform the correct update of the related metrics.

Processing events only at the endpoints (instead of all the time units) has the premise of improving the computational performance of the method by orders of time. However, there is a challenge. In many of our metrics, the duration of the overlapping time of events is increasingly important. In the cases of the *streaming approach* and *the naive approach*, where all units are examined, we had to simply increment the duration values by one (1) time unit, when needed. Now that endpoints are examined in arbitrary times, computing the duration of overlapping times (without examining all time units) is challenging.

To address this problem, we adopt the *sweep line approach*. Using the techniques mentioned in Chapter 2, we execute the state-of-the-art sweep-line algorithm to identify pairs of intersections and multiple simultaneous intersections of intervals, or in this case events. We already showed that we can represent the edge

Table 4.2: Summary of Time Complexities

	Naive	**Streaming**	**SLOT**
i	$O(T \cdot (\lvert V_t\rvert + \lvert E_t\rvert))$	$O(T \cdot (\lvert V_t\rvert + \lvert E_t\rvert))$	$O(\lvert E\rvert)$
ii	$O(T \cdot (\lvert V_t\rvert + \lvert E_t\rvert))$	$O(T \cdot (\lvert V_t\rvert + \lvert E_t\rvert))$	$O(\lvert E\rvert)$
iii	$O(T \cdot (\lvert V_t\rvert^3))$	$O(T \cdot (\lvert V_t\rvert^3))$	$O(\lvert E\rvert)$

streams as intervals in Section 4.3, now all that is necessary is to calculate our metrics of interest by using the sweep-line algorithm. Given a trajectory network $G_{[0,T]=(V,E)}$, the number of events is the same as the number of edges E, thus the number of event endpoints will be $T_L = 2|E|$. It is sensible to assume that the maximum node degree of our network is much less than the total number of edges ($max(deg_u) \ll |E|$). Therefore, the worst-case computational complexity for all three trajectory node metrics is $O(|E|)$.

Our algorithms assume that the edges are given in chronological order; if not, they can be sorted in $O(|E| \cdot log|E|)$ time.

Table 4.2 provides a summary of all the time complexities of different methods for each subproblem of Problem 1. We provide implementation details of the algorithms that compute the various metrics in Chapter 4.5 and demonstrate the efficiency of the method in Chapter 4.6.

4.5 Algorithms

The SLOT algorithm receives a set of all endpoints t_s and t_e of all events $e_{u,v}$ and computes the relevant metrics for all nodes $u \in V$, in a single pass. The algorithm has two parts. In the first part, all variables are initialized using `InitializeMetrics`. In the second part, event endpoints are iteratively processed to compute metrics of interest. For each metric, a separate metric procedure is invoked. Details of the metric procedures are provided in the next paragraphs. There is also an auxiliary procedure, `StoreNodeMetric`, which is used by all metric procedures of the algorithm. This procedure computes the elapsed time between the last and current endpoint for a particular node and incrementally updates metric and duration values.

4.5.1 Trajectory Node Degree

For the computation of the trajectory node degree, we provide the procedure `CalculateDegree`. As endpoints are processed, we monitor the current node degrees and increment or decrement them according to whether an edge attached to a node is added or removed. A time series of the node degree values of each node is stored and returned at the end.

4.5.2 Trajectory Node Connectedness

For the computation of the trajectory node connectedness, we provide the `CalculateConnectedNess` procedure. As endpoints are processed, we monitor the currently available components in the network, information about the component each node participates in, and its size. When a new edge is added, there are two possible outcomes: either the two nodes attached to the new edge are already members of the same component (no need to update), or they are members of two different components that now need to be merged into one. In either case, the current connectedness values of all nodes participating in any of the affected components are updated. When an edge is removed, we must perform a search to check if this has caused a disconnection, or the component remains connected. We perform a breadth-first search to determine if the two nodes are still connected. If after the removal of the edge, the two nodes are still connected, there is no need to update. Otherwise, the current component needs to be split into two components and the values of all nodes participating in any of the affected components are updated. A time series of the node connectedness values of each node is stored and returned at the end, along with the times each component found started or stopped existing.

4.5.3 Trajectory Node Triangle Membership

For the computation of the trajectory triangle membership, we provide the `CalculateTriangles` procedure. As endpoints are processed, we monitor the current number of triangles each node participates in. When a new edge is added (or removed), we check whether the two newly connected (or disconnected) nodes have any common neighbors. If they do, then a new triangle is formed or an existing one no longer exists, respectively. We update the metric values accordingly for each of the three nodes, and we update the lists of active neighbors for the two nodes of the new edge. A time series of the node triangle membership values of each node is stored and returned at the end, along with the times each triangle found started or stopped existing.

Algorithm 3: Main SLOT algorithm that scans over the sequence of edge events and at the endpoints of each event, calculates the relevant metrics based on user-selected flags.

```
Input: Set $V$ of nodes, Set $Ends$ of all event endpoints $t_s$, $t_e$ of all events $e_{u,v} = (u, v, t_s, t_e)$
Output: Set $V$ of nodes, each with a set representing the node's distributions of metric values over
        time $(u, D_{deg_u}, D_{CC_u}, D_{\lambda_u})$
if not sorted(Ends) then
    Sort(Ends)
InitializeMetrics($V$)
for endpoint in Ends do
    CalculateDegree($V$, endpoint)
    CalculateTriangles($V$, endpoint)
    CalculateConnectedness($V$, endpoint)
```

Procedure InitializeMetrics(V)

```
components ← []
for node in V do
    node.last_time ← 0
    node.degree.value ← 0
    node.degree.history ← []
    node.neighbors ← []
    node.triangles.value ← 0
    node.triangles.history ← []
    components.append(node)
    node.component ← components[node]
    node.connectedness.value ← 0
    node.connectedness.history ← []
```

Procedure StoreNodeMetric(V, endpoint, node, metric)

```
new_duration ← endpoint.time - node.last_time
if new_duration> 0 then
    if node.metric.value not in node.metric.history then
        append(node.metric.value) to node.metric.history
        node.metric.value.duration ← 0
    node.metric.value.duration + = new_duration
    node.last_time ← endpoint.time
```

```
Procedure CalculateDegree($V$, endpoint)

StoreNodeMetric($V$, endpoint, endpoint.node_u, degree)
StoreNodeMetric($V$, endpoint, endpoint.node_v, degree)
if $event.type = start$ then
    endpoint.node_u.value $+= 1$
    endpoint.node_v.value $+= 1$
else
    endpoint.node_u.value $-= 1$
    endpoint.node_v.value $-= 1$
```

Procedure CalculateConnectedness(V, endpoint)

```
node_u←endpoint.node_u, node_v←endpoint.node_v
com_u←node_u.component, com_v←node_v.component
if endpoint.type=start then
    if com_u != com_v then
        new_com← MergeComponents(com_u, com_v)
        new_size = length(new_com)
        for node in new_com do
            node.component← new_com
            StoreNodeMetric($V$, endpoint, node, connectedness)
            node.connectedness.value = new_size
else
    connected, com_u, com_v = BFS(V,node_u,node_v)
    if not connected then
        size_u = length(com_u)
        size_v = length(com_v)
        for node in com_u do
            node.component← com_u
            StoreNodeMetric($V$, endpoint, node, connectedness)
            node.connectedness.value = size_u
        for node in com_v do
            node.component← com_v
            StoreNodeMetric($V$, endpoint, node, connectedness)
            node.connectedness.value = size_v
```

Procedure CalculateTriangles(V, endpoint)

```
node_u←endpoint.node_u, node_v←endpoint.node_v
if endpoint.type = start then
    append(node_v) to node_u.neighbors
    append(node_u) to node_v.neighbors
else
    remove(node_v) from node_u.neighbors
    remove(node_u) from node_v.neighbors
for node_i in endpoint.node_u.neighbors do
    if node_i in endpoint.node_v.neighbors then
        StoreNodeMetric($V$, endpoint, node_u, triangles)
        StoreNodeMetric($V$, endpoint, node_v, triangles)
        StoreNodeMetric($V$, endpoint, node_i, triangles)
        if endpoint.type=start then
            node_u.triangles.value $+=$ 1
            node_v.triangles.value $+=$ 1
            node_i.triangles.value $+=$ 1
        else
            node_u.triangles.value $-=$ 1
            node_v.triangles.value $-=$ 1
            node_i.triangles.value $-=$ 1
```

4.6 Experimental Evaluation

In this Section, we experimentally evaluate the performance of SLOT compared to the static and streaming methods. We also examine the effect of some critical parameters on its computation cost. We aim to answer the following questions:

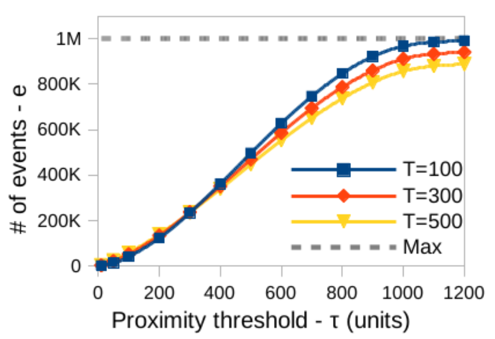
- **Q1 Effects of the Proximity Threshold.** How does the proximity threshold τ affect the final number of edges and events in the trajectory network?
- **Q2 SLOT Comparative Performance.** How much faster is SLOT compared to the naive and streaming methods?
- **Q3 SLOT Scalability.** How SLOT scales to larger number of objects or events in the trajectory network?

4.6.1 Setup

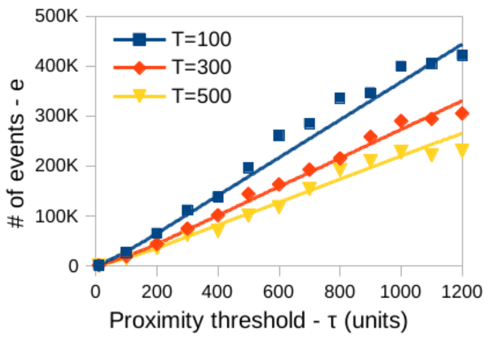
Before presenting the results, we provide details of the computational environment and the data sets employed.

Environment: All experiments are conducted on a PC with 8x Intel(R) Core$^{\text{TM}}$ i7-7700 CPU @ 3.60GHz and 64GB memory. Python 3.6 is used and the static graph calculations use the state-of-the-art algorithms for the relevant metrics provided by the *networkx* package. For the algorithm performance evaluation, we measure the average time for 10 executions of each algorithm, and for the study of simulation parameter effects, we take the average results of 10 random seed executions.

Data: In order to evaluate the various parameters of our algorithms under a very wide variety of conditions, we utilized synthetic data. Towards this end, a generator was implemented that allows the simulation of random or constant velocity trajectories over a Euclidean plane. Given the size T of an observation period $[0, T]$ every object is set in motion for T discrete time units forming trajectories. Objects moving outside



(a) Constant trajectories



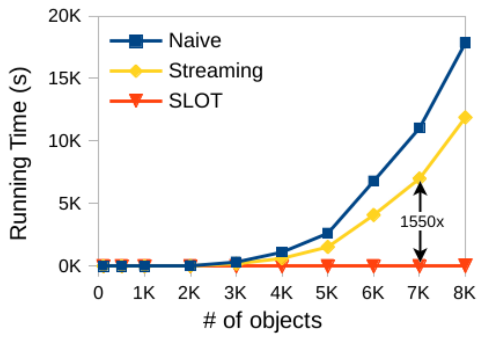
(b) Random trajectories

Figure 4.3: Effect of proximity threshold on number of events, $N = 1000$.

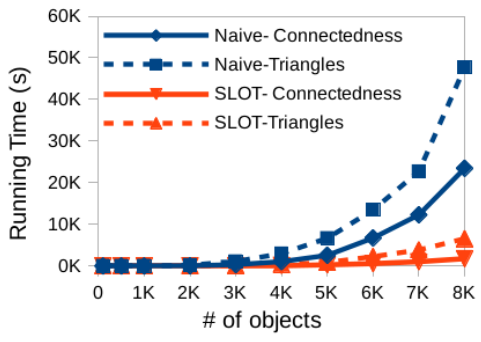
the boundaries are deleted, while new objects are generated at a steady rate to counteract the loss. The resulting generator has a number of configurable parameters, including *space size*, *min and max speed of an object*, *new object generation rate*, *observation time*. Different combinations of these parameters lead to different trajectory networks. We examine the effect of some of these properties on the results and the algorithm performance. For the majority of the experiments and without loss of generality, we fix the following parameters $space\_size = 1000 \times 1000$, $minimum\_speed = 0$ and $maximum\_speed = \pm 1$. For experimental evaluation purposes, various datasets were created, ranging from 10^2 to 10^5 objects and 10^4 to 10^{10} individual measurements. Additionally, to evaluate the scalability of *SLOT* to larger datasets, we had to resort to a random generator of events (instead of trajectories). This will skip the computations required to construct the trajectory network, a more computationally expensive task.

4.6.2 Effects of the Proximity Threshold

We examined the impact of the threshold parameter during the simulation phase on the number of events produced, and therefore on the resulting data size. With fixed values for the rest of the parameters, the results for various threshold values can be seen in Figure 4.3a for the single-contact case, and Figure 4.3b for the multi-contact one.



(a) Node degree



(b) Connectedness, Triangles

Figure 4.4: Performance of SLOT versus naive, streaming algorithms, $T = 100$.

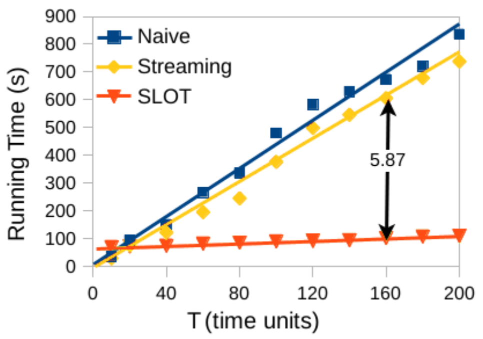
Constant Trajectories: For proximity threshold $\tau < 300$, the number of connections increases quadratically with τ. This happens because the τ defines a circle with radius $r = \tau$ around every object, where any other objects found will be connected to the first. As τ increases, the area of that circle increases relative to the square of τ. Around $\tau = 300$, many objects are limited by the space boundaries and the number of new connections slows, converging to the theoretical maximum of all possible connections $N(N-1)$. Because of objects moving out of bounds, the actual maximum is lower for larger observation times T.

Random Trajectories: In the case of random trajectories, the exact number of events is unpredictable, but it increases in a roughly linear fashion with the proximity threshold.

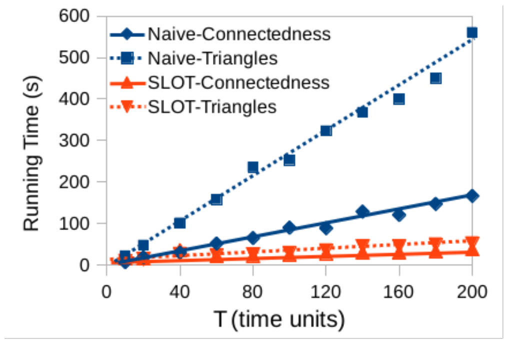
4.6.3 SLOT Comparative Performance

SLOT is an exact algorithm, so it will always find the correct values of the metrics of interest that it computes. Here, we evaluate the time performance of SLOT against the naive and streaming methods, as a function of the number of objects in the trajectory network. We report results for the trajectory node degree metric. For the rest two metrics (connectedness, triangles), we only compare SLOT to the static method, since streaming versions of algorithms that compute these metrics are not readily available. Fig. 4.4a presents the results for for trajectory node degree, and Fig. 4.4b for the rest of the metrics.

SLOT outperforms both the naive and streaming methods by a significant margin, up to 1550$\times$ in the



(a) Node degree



(b) Connectedness, Triangles

Figure 4.5: Performance of SLOT versus naive, streaming algorithms, $N = 1000$.

case of $7 \cdot 10^3$ objects. This happens because SLOT scales with the number of events $|E|$, which is relative to the number of edges $|E_t|$, but the other algorithms scale with the product of time and number of edges $T \cdot |E_t|$, as mentioned in chapter 4.4. It can also be seen that the streaming method performs better than the naive. This is to be expected, as the streaming method avoids many unnecessary computations by maintaining information over time. We also experiment with different observation times T, and the results can be seen in Fig. 4.5. SLOT once more outperforms the naive and streaming algorithms by a large margin, for example 5.87× for $T = 160$. The naive and streaming algorithms behave this way because they scale linearly with time T, while its effect on the calculation cost of SLOT is negligible.

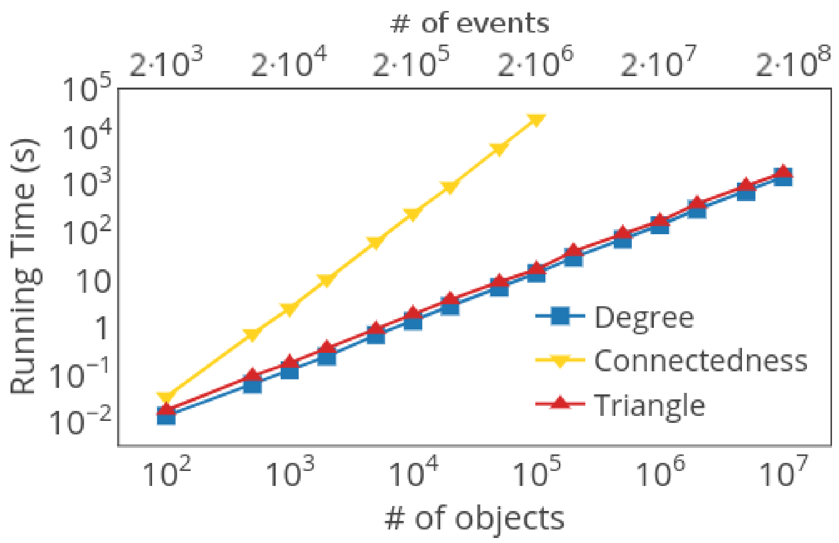


Figure 4.6: Performance of SLOT for large scale data, $T = 10000$.

4.6.4 SLOT Scalability

Using the synthetic data generated by the trajectory network generator, we examine the performance of the SLOT algorithm itself for different numbers of objects. To obtain consistent results that scale with the number of nodes, the data generated had on average of 20 events per node. This means that the number of events scales linearly with the number of objects, and as a result the execution time of SLOT does the same. This can be seen in Figure 4.6.

5 Conclusions and Future Steps

In this work we explored specific cases and uses of the object intersection problem from the field of computational geometry.

We have introduced MultipleIntersect, a novel and computationally challenging problem that arises in the context of *identifying* and *quantifying the size* of multiple intersections of a large number of axis-aligned multi-dimension geometric objects (*regions*). To address this problem we designed and implemented an efficient algorithm, named SLIG. SLIG is a versatile sweep-line based method that operates with the help of an auxiliary data structure, a *region intersection graph* (RIG), which is effectively a graph-based data structure that is easily constructed during the one-pass traversal of the sweep-line. RIG provides fast access to important information regarding the connectivity of regions (whether regions intersect or not) that is otherwise difficult to obtain directly from the data. As a result, our proposed method SLIG that utilizes RIG, is able to address the problem of interest and demonstrate a performance that is many orders of time faster than sensible state-of-the-art approaches, while also enabling further exploration of the data through intersection-related queries, namely SingleRegionQuery and MultiRegionQuery. Extensive experiments were performed to demonstrate the effectiveness of our algorithm, in a wide range of conditions, including a real-world dataset of extreme weather conditions. We also demonstrated that our method is able to scale to very large amounts of regions (while running in a single PC).

In this work, In order to evaluate the metrics of interest, we proposed $SLOT$, a fast and accurate algorithm for mining node importance in trajectory networks based on the sweep-line approach. Extensive experiments were performed to demonstrate the effectiveness of our algorithm, in a wide range of conditions. Furthermore, our algorithm outperformed naive and sensible streaming approaches to address the problem, by many orders of time. We also demonstrated that our method can scale to very large amounts of trajectories. We are confident that the novel problem and method presented will prove useful and find interesting application in a number of real-world applications and solutions.

Furthermore, we have represented trajectories of moving objects as a *trajectory network*. Based on this representation, we have introduced metrics of network importance of moving objects, specifically *trajectory node degree*, *trajectory node connectedness* and *trajectory node triangle membership*. These metrics can be used to better understand the behavior of a moving object or a group of them through their interactions with the environment and other objects, over time. They can also be used to reveal interesting network dynamics of moving objects, not easily observable before. In order to evaluate these metrics, we developed *SLOT*, a fast and accurate algorithm for mining node importance in trajectory networks based on the sweep-line approach found in the object intersection problem. Extensive experiments were performed to demonstrate the effectiveness of our algorithm, in a wide range of conditions. Furthermore, our algorithm outperformed naive and sensible streaming approaches to address the problem, by many orders of time. We also demonstrated that our method can scale to very large amounts of trajectories. This work is a substantial first step in understanding network dynamics of moving objects. We are confident that, as large amounts of trajectory data become available, these methods will prove useful and find interesting application in a number of real-world problems and solutions.

This research can be further extended in a number of ways. It may be of interest to evaluate the algorithms developed in various real-world datasets, or studying how they can be implemented to work in a distributed way.

For the multiple intersection problem, it may be interesting to explore alternatives utilizing different algorithmic paradigms, such as interval trees, and compare the algorithms' performance and space requirements. Furthermore, it may be of great interest to extend our approach and methodology to other object intersection problems, specifically objects of different shapes. Although different version of the sweep-line algorithm are available for other basic shapes such as circles or regular polygons, we believe our approach could be used with arbitrary, irregularly shaped objects, by approximating their shape with axis-aligned regions of different sizes (an indication of this can be seen in Fig. 3.5, if the areas of different type in that grid are considered single, irregularly shaped objects).

For the node importance in trajectory networks problem, it could be useful to compare the insights this approach provides with those of more traditional approaches. It could be of interest, for example, to apply a conventional trajectory mining technique such as trajectory classification on a real-world dataset, and then try to enhance those results using the information the trajectory network metrics can provide.

Furthermore, it is possible to make direct use of these novel algorithms in order to develop various tools, such as data visualization applications and trajectory analysis platforms. To encourage reproducibility we provide details of our data generator and methods' pseudo-code, while we make the source code and data sets publicly available.

Bibliography

[1] Pankaj K. Agarwal, Marc Van Kreveld, and Subhash Suri. Label placement by maximum independent set in rectangles. *Computational Geometry*, 11(3-4):209–218, 1998.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[3] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, and Jeffrey Scott Vitter. Scalable sweeping-based spatial join. In *Proc. of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 570–581, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[4] Egon Balas and Chang Sung Yu. Finding a maximum clique in an arbitrary graph. *SIAM Journal on Computing*, 15(4):1054–1068, 1986.

[5] R. Bar-Yehuda, M. M. Halldrsson, J. (S.) Naor, H. Shachnai, and I. Shapira. Scheduling split intervals. *SIAM Journal on Computing*, 36(1):1–15, 2006.

[6] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[7] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, May 1990.

[8] Bentley and Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, C-29(7):571–577, July 1980.

[9] Mark De Berg, Marc Van Kreveld, Mark Overmars, and Otfried Schwarzkopf. *Computational geometry: algorithms and applications*. Springer, 2000.

[10] Gino Van Den Bergen. Efficient collision detection of complex deformable models using aabb trees. *Journal of Graphics Tools*, 2(4):1–13, 1997.

[11] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of mathematical sociology*, 25(2):163–177, 2001.

[12] Coen Bron and Joep Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, 1973.

[13] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting triangles in data streams. In *Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 253–262. ACM, 2006.

[14] Frédéric Cazals and Chinmay Karande. A note on the problem of reporting maximal cliques. *Theoretical Computer Science*, 407(1-3):564–568, 2008.

[15] Timothy M. Chan. A note on maximum independent sets in rectangle intersection graphs. *Information Processing Letters*, 89(1):19–23, 2004.

[16] B. Chazelle, Jacob E. Goodman, and Richard Pollack. *Advances in discrete and computational geometry: proceedings of the 1996 AMS-IMS-SIAM Joint Summer Research Conference, Discrete and Computational Geometry.* American Mathematical Society, 1999.

[17] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms.* MIT press, 2009.

[18] Mark de Berg, Joachim Gudmundsson, and Ali D. Mehrabi. Finding pairwise intersections inside a query range. *Algorithmica*, 80(11):3253–3269, Nov 2018.

[19] Boris Delaunay. Sur la sphre vide. a la mmoire de georges vorono. *Bulletin de l'Acadmie des Sciences de l'URSS*, 6:793–800, 1934.

[20] F. Dévai and L. Neumann. A rectangle-intersection algorithm with limited resource requirements. In *2010 10th IEEE International Conference on Computer and Information Technology*, pages 2335–2340, Berlin, Germany, June 2010. IEEE.

[21] Somayeh Dodge, Robert Weibel, and Ehsan Forootan. Revealing the physics of movement: Comparing the similarity of movement characteristics of different types of moving objects. *Computers, Environment and Urban Systems*, 33(6):419–434, 2009.

[22] Herbert Edelsbrunner. A new approach to rectangle intersections. *International Journal of Computer Mathematics*, 13(3-4):221–229, 1983.

[23] David Eppstein, Maarten Löffler, and Darren Strash. Listing all maximal cliques in sparse graphs in near-optimal time. *Algorithms and Computation Lecture Notes in Computer Science*, abs/1006:403–414, 2010.

[24] Thomas Erlebach, Klaus Jansen, and Eike Seidel. Polynomial-time approximation schemes for geometric intersection graphs. *SIAM Journal on Computing*, 34(6):1302–1323, 2005.

[25] J. Fang, J.s.l. Wong, K. Zhang, and P. Tang. A new fast constraint graph generation algorithm for vlsi layout compaction. *1991., IEEE International Sympoisum on Circuits and Systems*, 1991.

[26] K. Ruben Gabriel and Robert R. Sokal. A new statistical approach to geographic variation analysis. *Systematic Zoology*, 18(3):259, 1969.

[27] Diansheng Guo, Shufan Liu, and Hai Jin. A graph-based approach to vehicle trajectory analysis. *Journal of Location Based Services*, 4(3-4):183–199, 2010.

[28] Ralf Hartmut Güting and Werner Schilling. A practical divide-and-conquer algorithm for the rectangle intersection problem. *Information Sciences*, 42(2):95–112, 1987.

[29] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[30] Petter Holme. Network reachability of real-world contact sequences. *Physical Review E*, 71(4):046119, 2005.

[31] Hiroshi Imai and Takao Asano. Finding the connected components and a maximum clique of an intersection graph of rectangles in the plane. *Journal of Algorithms*, 4(4):310–323, 1983.

[32] Ibrahim Kamel and Christos Faloutsos. On packing r-trees. *Proceedings of the second international conference on Information and knowledge management - CIKM 93*, 1993.

[33] David Kempe, Jon Kleinberg, and Amit Kumar. Connectivity and inference problems for temporal networks. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 504–513. ACM, 2000.

[34] Hyoungshick Kim and Ross Anderson. Temporal node centrality in complex networks. *Physical Review E*, 85(2):026107, 2012.

[35] Lauri Kovanen, Márton Karsai, Kimmo Kaski, János Kertész, and Jari Saramäki. Temporal motifs in time-dependent networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2011(11):P11005, 2011.

[36] Marc Van Kreveld and Jun Luo. The definition and computation of trajectory and subtrajectory similarity. *Proc. of the 15th ACM int. symposium on Advances in geographic information systems - GIS 07*, 2007.

[37] Matthieu Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1-3):458–473, 2008.

[38] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering. *Proceedings of the ACM SIGMOD international conference on Management of data - SIGMOD 07*, 2007.

[39] C. Lekkeikerker and J. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51(1):45–64, 1962.

[40] Jure Leskovec, Kevin J Lang, Anirban Dasgupta, and Michael W Mahoney. Statistical properties of community structure in large social and information networks. In *Proceedings of the 17th international conference on World Wide Web*, pages 695–704. ACM, 2008.

[41] Terry A. McKee and F. R. McMorris. *Topics in intersection graph theory.* Society for Industrial and Applied Mathematics, 1999.

[42] Mark McKenney and Tynan McGuire. A parallel plane sweep algorithm for multi-core systems. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 392–395, New York, NY, USA, 2009. ACM.

[43] Mark EJ Newman. A measure of betweenness centrality based on random walks. *Social networks*, 27(1):39–54, 2005.

[44] Vincenzo Nicosia, John Tang, Cecilia Mascolo, Mirco Musolesi, Giovanni Russo, and Vito Latora. Graph metrics for temporal networks. In *Temporal networks*, pages 15–40. Springer, 2013.

[45] Eunjin Oh and Hee-Kap Ahn. Finding pairwise intersections of rectangles in a query rectangle. *CoRR*, abs/1801.07362, 2018.

[46] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, pages 601–610. ACM, 2017.

[47] Jignesh M. Patel and David J. DeWitt. Partition based spatial-merge join. *SIGMOD Rec.*, 25(2):259–270, June 1996.

[48] James Peters. Hellys theorem and strongly proximal helly theorem. *Intelligent Systems Reference Library Computational Proximity*, 2016.

[49] C. S. Rim and K. Nakajima. On rectangle intersection and overlap graphs. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 42(9):549–553, Sept 1995.

[50] C. S. Riolo. Methods and measures for the description of epidemiologic contact networks. *Journal of Urban Health: Bulletin of the New York Academy of Medicine*, 78(3):446457, 2001.

[51] Abdullah Sawas, Abdullah Abuolaim, Mahmoud Afifi, and Manos Papagelis. Tensor methods for group pattern discovery of pedestrian trajectories. In *Proceedings of the 19th IEEE International Conference on Mobile Data Management*, pages 76–85, 2018.

[52] Timos Sellis, Nick Roussopoulos, and Christos Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th VLDB Conference, Brighton 1987*, pages 507–518, 1987.

[53] Michael Ian Shamos and Dan Hoey. Geometric intersection problems. In *17th annual symposium on foundations of computer science*, pages 208–215. IEEE, 1976.

[54] Katarzyna Siła-Nowicka, Jan Vandrol, Taylor Oshan, Jed A Long, Urška Demšar, and A Stewart Fotheringham. Analysis of human mobility patterns from gps trajectories and contextual information. *International Journal of Geographical Information Science*, 30(5):881–906, 2016.

[55] T.d. Tang, Erik L.j. Bohez, and Pisut Koomsap. The sweep plane algorithm for global collision detection with workpiece geometry update for five-axis nc machining. *Computer-Aided Design*, 39(11):10121024, 2007.

[56] Kevin Toohey and Matt Duckham. Trajectory similarity measures. *SIGSPATIAL Special*, 7(1):43–50, 2015.

[57] Godfried T. Toussaint. The relative neighbourhood graph of a finite planar set. *Pattern Recognition*, 12(4):261–268, 1980.

[58] Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 837–846. ACM, 2009.

[59] Shuji Tsukiyama, Mikio Ide, Hiromu Ariyoshi, and Isao Shirakawa. A new algorithm for generating all the maximal independent sets. *SIAM Journal on Computing*, 6(3):505–517, 1977.

[60] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of small-world networks. *Nature*, 393:440–442, 2011.

[61] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. Path problems in temporal graphs. *Proceedings of the VLDB Endowment*, 7(9):721–732, 2014.

[62] Guan Yuan, Penghui Sun, Jie Zhao, Daxing Li, and Canwei Wang. A review of moving object trajectory clustering algorithms. *Artificial Intelligence Review*, 47(1):123–144, 2017.

[63] Francesco Zanlungo, Tetsushi Ikeda, and Takayuki Kanda. Potential for the dynamics of pedestrians in a socially interacting group. *Physical Review E*, 89(1):012811, 2014.

[64] Fa Zhang, Xiang-Zhen Qiao, and Zhi-Yong Liu. A parallel smith-waterman algorithm based on divide and conquer. In *Fifth International Conference on Algorithms and Architectures for Parallel Processing*, pages 162–169. IEEE, Oct 2002.

[65] Yunhong Zhou and Subhash Suri. Collision detection using bounding boxes: Convexity helps. *Algorithms - ESA 2000 Lecture Notes in Computer Science*, pages 437–448, 2000.