POLITECNICO DI TORINO

MASTER's Degree in Computer Engineering

MASTER's Degree Thesis

Propagation pattern mining algorithm: a parallel approach

Supervisors

Prof. Paolo GARZA

Dr. Luca COLOMBA

Candidate

Gaetano Riccardo RICOTTA

April 2023

Summary

The main objective of this project is to re-implement a centralized state-of-the-art algorithm for spatio-temporal patterns extraction using the distributed Apache Spark framework. Spatio-temporal data refers to data that contains information about the location and time in which the event took place. Throughout this thesis, a public dataset containing information related to traffic and weather events in the USA was considered. This dataset contains approximately 37 million events. Traffic events include congestion, accidents, and construction, while weather events include rain, snow, and storms.

The codebase was developed using PySpark, a Python library for big data processing that runs on top of the Apache Spark framework. Spark is a powerful open-source framework for big data processing that allows for fast and efficient data processing and analysis.

To extract frequent patterns from the dataset, we are using SLEUTH, a tree mining algorithm, which is able to extract frequent subtrees from a tree database. In this context, spatio-temporal correlation of events is encoded through tree-based structures according to different space and time constraints. SLEUTH is a sequential algorithm that can handle large datasets and extract patterns in a highly efficient manner.

In this work, we have primarily used three configurations, each of which differs from the others by a temporal threshold of 10, 15, and 20 minutes. This time threshold is used to search and build the aforementioned parent-child relationships between events with the goal of creating a tree database from the original dataset. The experiments were mainly conducted in three US cities: Los Angeles, Boston, and New York City. We remark the fact that the SLEUTH algorithm adopted in this study was not subject of parallelization, and such was used in its original implementation made available by the authors, which is used to extract patterns from the dataset.

Acknowledgements

I dedicate this achievement to the people who have been close to me during these years.

In particular, I would like to thank my advisor, Professor Paolo Garza, and my co-advisor, Dr. Luca Colomba, thank you for your endless availability and patience. Your suggestions and experience allowed me to complete this thesis work in the best possible way.

I am infinitely grateful to my parents, brother, and sister for supporting me throughout my academic journey. You have always believed in my abilities and have done everything possible to help me achieve every goal. Thanks to your support, I have been able to overcome even the most difficult moments when I wanted to give up everything. For this, I will always be grateful to you.

I would like to thank my friends and my girlfriend for supporting me and being there for me during difficult times, as well as for sharing the joy of the achievements I have made.

Finally, I would like to thank my university colleagues with whom I have shared the struggles and joys of this academic journey. Without you, this path would have been much less interesting and stimulating.

Gaetano Riccardo Ricotta

Table of Contents

List of Tables

List of Figures

Acronyms

GPS
: Global Positioning System

RDD
: Resilient Distributed Dataset

SCPM
: Spatial Colocation Pattern Mining

SLEUTH
: anagram of the bold letters in the phrase: **L**isting “**H**idden” or **E**mbedded **U**nordered **S**ub **T**rees

STInv
: SpatioTemporally Invariant Pattern

Chapter 1

Introduction

1.1 Background

Spatio-temporal data refers to information that is linked to both geographic location and time. This can include a wide range of type of data, such as traffic and weather data. For example, traffic data might include information about accidents, congestion, or road construction, while weather data might include information about rain or other weather events.

The analysis of spatio-temporal data is important for a variety of reasons, including urban planning, disaster prediction, and traffic management. One key aspect of this analysis is the discovery of patterns within the data. This can include identifying co-locations, co-occurrences, or cause-and-effect relationships. For example, a pattern might be discovered that links a rain event to an increase in the number of accidents on a particular road.

The extraction of these patterns is crucial for making data-driven decisions in various fields. For instance, in urban planning, the patterns can be used to identify areas where accidents are more likely to happen, and thus design safer roads. In the field of disaster prediction, the patterns can be used to predict the likelihood of certain types of natural disasters in certain areas. Finally, in traffic management, the patterns can be used to predict traffic congestion, and take appropriate actions.

1.2 Objective

The goal of this thesis is to parallelize the analysis of dataset containing traffic and weather events and run the extraction of spatio-temporal patterns, as described by [1]. The current implementation of the code is written in Python and is executed in a sequential manner. However, in order to make it more efficient, my task is to re-implement it using Spark, a powerful framework that enables parallel processing of data, which can result in significant speed improvements.

Spark is a popular big data processing framework that allows to perform parallel processing of large data sets. By leveraging the power of Spark, the code for spatio-temporal pattern extraction can be executed much faster, even when dealing with large amounts of data. It is a powerful tool that allows to distribute data across multiple machines, so that large data sets can be processed in a much shorter time.

1.3 Organisation of the work

The remainder of this thesis is organized as follows. In Chapter 2, related work is reviewed, providing context and background information for the research. In Chapter 3, our framework is described, providing detailed information about the methods and techniques used, it includes pseudocode that illustrates the key steps and processes of the framework. Additionally, in Chapter 3, the dataset that has been used for the study will be presented, including information about the source and the characteristics of the data. Chapter 4 will present the results of the experiments and the analyses that have been conducted. Finally, in Chapter 5, the conclusions summarize the main findings and contributions of the work presented, and possible further developments.

Chapter 2

Related Work

The concept of spatio-temporal patterns, which involves both spatial and temporal components, have been explored in previous works. Earlier studies tended to focus more on the spatial component, omitting or ignoring the temporal information. Instead, more recent works focused on the development of algorithms which are able to handle both the spatial and temporal dimensions. These frameworks can be used to identify patterns of co-occurrence or colocation, cascading patterns, or cause and effect relationships.

When working with data that pertains to both space and time, or even just space alone, there are a variety of patterns that can be extracted. Some of these include **colocation patterns**, **trajectory patterns**, and **spatio-temporal patterns**.

2.1 Colocation pattern

Colocation pattern focuses on the location of objects or entities in relation to one another, and how they are grouped or clustered together. For example, a colocation pattern could be used to identify areas where certain types of businesses are most likely to be located. This could be done by analyzing business data and identifying locations where multiple businesses of the same type occurred within a certain distance of each other. For example, if there is a high concentration of coffee shops within a one-mile radius, it could indicate a potential high-demand area for coffee shops in the future. Entrepreneurs could use this information to open a new coffee shop in that area or to implement other business strategies.

In literature, a work that utilizes colocation patterns is [2]. The main objective of this paper is to extract colocation patterns in a parallel manner. The authors introduce a new parallel mechanism for spatial colocation pattern mining

(SCPM) based on neighbor-dependency partitions and propose a method to calculate the participation index without generating the table instance. This method is based on the research of Shekhar et al. [3], which uses a distance-based measure to evaluate the prevalence of colocation patterns. The algorithm divides the neighbor relationships into partitions, allowing the SCPM task to be performed on each partition independently, thus the entire mining process can be done in parallel to find all colocation patterns. To improve the efficiency of the participation index calculation, the authors propose a column-based method that eliminates the need for the whole table instance of a pattern. The proposed parallel SCPM algorithm is implemented on the MapReduce platform.

2.2 Trajectory pattern

Trajectory pattern focuses on the movement or path of an object or entity over time. For example, a person's movement through a city can be represented as a trajectory pattern. It can be represented using a series of GPS coordinates recorded at regular intervals. The pattern can be analyzed to understand the person's behavior, such as their preferred routes, frequented locations, and duration of stay at each location.

A trajectory can be represented in a formal manner as a sequence of positions, where each position is represented by a set of attributes, including the position's identifier, spatial location, timestamp, and possibly additional descriptive data. The spatial location of the position can be represented in different forms, depending on the technology used for recording, such as GPS coordinates or cell-based identifiers.

In literature, a work that utilizes trajectory pattern is the paper [4] that presents a method for identifying individuals who are traveling together using streaming GPS data. The method is based on the assumption that individuals who are traveling together will have similar trajectories, and uses a combination of spatial and temporal clustering techniques to group together individuals whose trajectories are similar. To discover individuals who are traveling together, the system should cluster objects and perform an intersection of objects within a cluster. Both clustering and intersection operations are heavy, therefore, a key point of the paper is to make this process more efficient. They propose a new method for intersection to accelerate the process.

2.3 Spatio-temporal pattern

Spatio-temporal patterns combine both space and time, providing insights into the interactions between events or objects across both dimensions. As mentioned, recent works take into account both the spatial and temporal aspect, but often they are based on a simple definition of spatio-temporal neighborhood that involves spatial proximity using a Euclidean or Cartesian system and temporal overlap. As a result, these solutions may not be suitable for analyzing applications such as traffic, transportation, or weather, where the impact of events may not be limited to a specific geographic area or may persist beyond the duration of the event. For example, an accident that occurs in one lane of a highway will not necessarily affect the opposite lane, but these frameworks would consider both lanes to be in the same neighborhood. Similarly, the impact of a snow event on traffic may continue even after the event has ended, but the constraints of these frameworks, which only take into account temporal overlap, would not capture this type of phenomenon. A paper that takes into consideration these more hidden aspects is [1].

Another work that works with spatio-temporal patterns and that we have taken into account is [5]. The goal of this paper is to generalize geospatial event relations through the introduction of the concept of spatio-temporal invariance. The authors center on discovering sequences of occurrences that are commonly linked to a trigger event of concern and that display similar spatial and temporal relative gaps. To tackle this problem, the authors suggest a new type of pattern called SpatioTemporally Invariant Patterns (STInvs). The authors describe a trigger event followed by a sequence of spatiotemporally related events, which are identified by a constrained spatial and temporal gap with respect to the trigger event.

In order to gain a deeper understanding of the concept of SpatioTemporally Invariant Patterns, I will reiterate the example provided in the paper. The manager of a bike-sharing system is interested in keeping track of and analyzing the occupancy levels of stations. Experts are asked to identify a set of trigger events. For instance, if the occupancy level of a specific station becomes too high, an event can be recorded as a trigger. When an event takes place, the location and timing of it are recorded. The location is the geographical spot where the event of type e occurred and is also defined by the surrounding area, which implies the potential area of impact of the event. They set different levels of neighborhoods. The occurrence time of a specific event t is divided into equal time intervals td. td represents the temporal level of detail at which they examine the occurrences of specific events. As an example, in Figure 2.1, the temporal resolution is 15 minutes. All events that happen within the same time interval (e.g., e_1 and e_2) are considered to be happening simultaneously.

The figure illustrates an example of sequences of events related to the monitoring of bike-sharing station occupancy in an urban area. The events being monitored, such as fully occupied, almost full, and increasing occupancy, indicate the occupancy levels of the bike-sharing stations. The illustration uses 15-minute time intervals as consecutive frames. To visualize the neighborhood of the trigger events e_1 and e_4, the spatial resolution is set to $r = 100m$ and circles are plotted around them, representing spatial distances of $1 \cdot r = 100m$, $2 \cdot r = 200m$, and $3 \cdot r = 300m$. In Figure 2.1, two STInvs are shown as lines connecting various spatio-temporal events. These lines are represented as a dashed line and a solid line respectively.

One example of a pattern can be seen in the STInv with trigger event "Almost full", which occurs at the time slot [10.00am, 10.15am]. This pattern demonstrates a correlation between events e_1 ("Almost full") and e_2 ("Increase"). More specifically, e_2 happens within a distance of $200m$ to $300m$ from the trigger event. Additionally, 15 minutes after the occurrence of the trigger event e_1, event e_5 ("Full") can also be seen in the same location. This sequence of spatio-temporal events (e_1-e_2 and then e_5) is noteworthy because another spatiotemporally invariant occurrence can be observed in the upper right corner of the spatial areas in time slots [10.15am, 10.30am] and [10.30am, 10.45am] (events e_7-e_8 and then e_{11}). This STInv can be expressed as: [$\langle$ Almost Full, $\delta s = 0m$, $\delta t = 0$ $\rangle$, $\langle$ Increase, $\delta s = 200m - 300m$, $\delta t = 0$ $\rangle$, $\langle$ Full, $\delta s = 0m$, $\delta t = 15mins$ $\rangle$]. It can be interpreted as: *When the occupancy level of a bike-sharing station is close to capacity and the usage of a neighboring station within a distance of 200m to 300m is on the rise, it can be expected that the original station will reach full capacity within 15 minutes.*

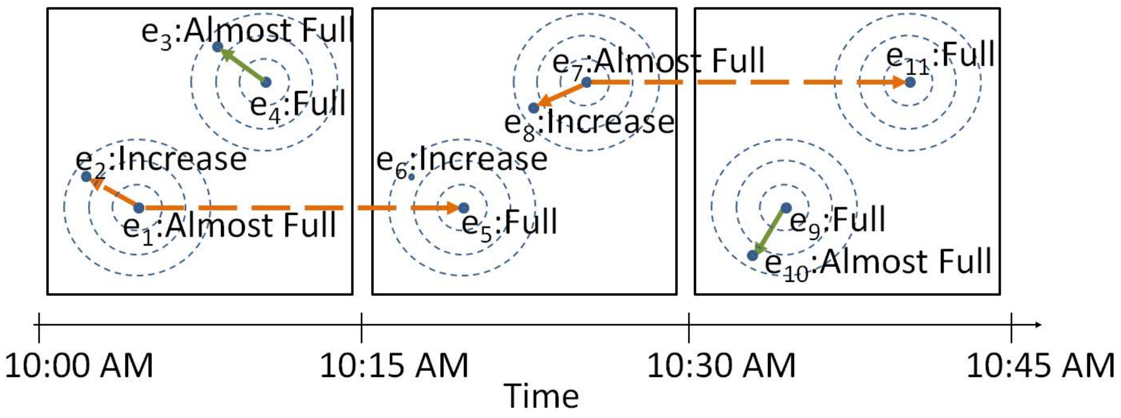


Figure 2.1: Examples of SpatioTemporally Invariant patterns.

The authors propose a new algorithm called STInv-Miner to identify SpatioTemporally Invariant Patterns. This algorithm utilizes a technique known as PrefixSpan to accomplish this task.

2.4 Overview of Apache Spark and PySpark

The Apache Spark framework has several components, including Spark Core, Spark SQL, Spark Streaming, MLlib, and GraphX. Spark Core is the underlying execution engine of Apache Spark and provides the basic functionality of Spark, including the Resilient Distributed Datasets (RDDs), task scheduling, and memory management. Spark Core provides a foundation for the other Spark components and is responsible for the distributed processing of data across a cluster of machines.

Spark Core implements the RDD abstraction, which is the fundamental data structure of Spark. RDDs allow Spark to efficiently process large data sets in parallel by dividing the data into smaller partitions that can be processed in parallel across multiple machines in a cluster. RDDs also provide a fault-tolerant mechanism for storing and processing large data sets, as RDDs can automatically recover from failures in the cluster.

In addition to RDDs, Spark Core also provides the task scheduling and resource management functions needed to run Spark applications. Spark Core uses a scheduler to manage the execution of tasks on a cluster and to ensure that the necessary resources are available to run those tasks. It also manages the allocation of memory and other resources to Spark applications and optimizes the use of resources in a cluster.

Spark also provides several other important features such as:

- Spark SQL: Spark provides a SQL interface that allows developers to run SQL queries on Spark data sets.
- Machine learning library: Spark includes a machine learning library, MLlib, which provides a set of algorithms for common machine learning tasks.
- Graph processing: Spark provides a graph processing library, GraphX, which allows developers to perform graph-based processing on large-scale data.
- Spark Streaming: Spark Streaming is a real-time data processing engine for processing live data streams.

The Apache Spark Architecture is shown in Figure 2.2.

PySpark is a Python library for big data processing that runs on top of the Apache Spark framework. PySpark provides a Python API for Spark, allowing developers to use Spark's distributed processing capabilities in Python. With PySpark, developers can write Spark applications in Python, taking advantage of

Spark's powerful distributed processing capabilities and Python's ease of use and high-level abstractions.

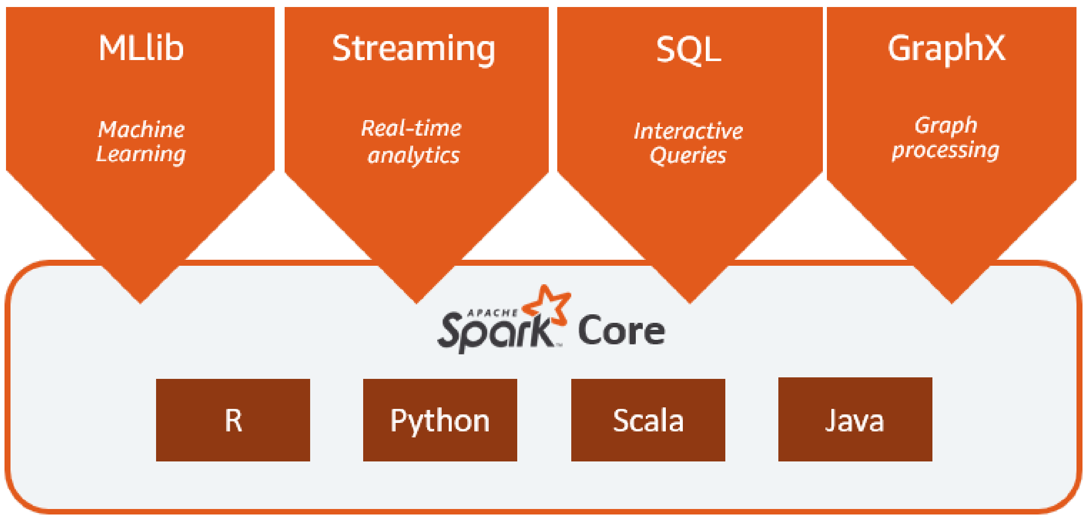


Figure 2.2: Apache Spark Architecture ([6])

Chapter 3

Methodology

To study the more particular cases that were discussed in Chapter 2, we relied on the framework developed by Sobhan Moosavi's team [1] for identifying Spatio-temporal patterns. This framework consists of two parts, one concerning the exploration of propagation patterns and the other to reveal influential patterns. In this thesis, we focused only on the first part. These patterns involve spatio-temporal entities that are both spatially co-located and temporally co-occurring, such as a snow event causing traffic congestion and resulting in an accident. To analyze these propagation patterns, the framework employs a specific definition of spatial neighborhood and a definition of temporal co-occurrence tailored to the characteristics of spatio-temporal data and the constraints of the application domain.

To test their framework, Sobhan Moosavi's team [1] used a large dataset containing spatio-temporal traffic and weather data. In our own framework, we used an updated version of the same dataset, which includes three times as many records. This dataset covers the contiguous United States from August 2016 to the end of December 2020 and includes 31.4 million traffic events and 5.6 million weather events. Propagation patterns, also known as short-term patterns, are identified using a process that involves representing spatiotemporal colocation and co-occurrence in the form of tree structures and then using an existing algorithm called SLEUTH [7] to find the most frequent patterns. This framework is particularly well-suited for domain such as traffic because it employs a strict definition of neighborhood. SLEUTH is a highly efficient algorithm designed to identify frequent, unordered subtrees within a database of trees. It is able to perform this task with performance that is comparable to TreeMiner [8], which is another algorithm that is specifically designed to mine ordered trees.

SLEUTH [7] is an algorithm that was developed by Zaki. It is a growth-based approach, which means that it works by gradually adding nodes to a tree structure until a desired pattern is found. SLEUTH is particularly designed for identifying frequent, unordered, embedded subtrees in a database of trees, which can be useful for understanding relationships between events or phenomena and for analyzing spatio-temporal patterns. SLEUTH has been applied to a variety of domains, including traffic analysis, urban planning, and land use dynamics. It works by starting with a seed node and then adding nodes to the tree structure based on the patterns that are observed in the data. It is able to identify patterns that are both frequent and embedded, which means that they occur frequently within the data and are nested within other patterns. SLEUTH is also able to identify unordered patterns, which means that the order of the nodes in the tree structure does not matter.

The SLEUTH algorithm is written in C++ and it takes as input a dataset represented as a set of encoded trees. The input trees must be in a specific format for the algorithm to be able to process them. The format of the input trees must be in the following format: **id**, **id**, **length**, **string_encoding**
Where:

- **id** is the unique identifier of the tree and it is repeated twice (the same value for the tree number)
- **length** is the number of items to follow on the line
- **string_encoding** is the coding of the tree.

An example of encoded input trees in this format could be:

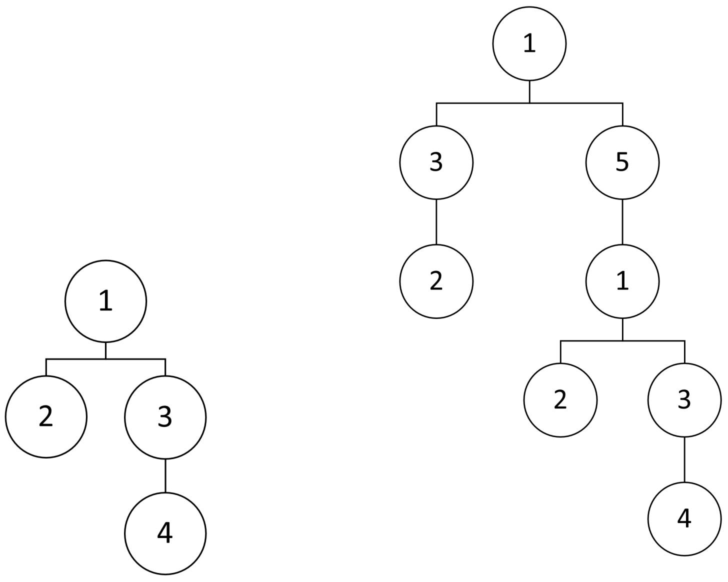
- 0 0 7 1 2 -1 3 4 -1 -1
- 1 1 11 2 1 2 -1 4 -1 -1 2 -1 3 -1
- 2 2 15 1 3 2 -1 -1 5 1 2 -1 3 4 -1 -1 -1 -1

This database has 3 trees, the first tree's string encoding has a length of 7 (including -1's), the second one has a length of 11 and the third one has a length of 15.

It's important to note that the string encoding, it's a representation of the tree structure, where where the '-1' represents the 'back to parent node' operation. The visual representation of the first and third string-encoded trees is shown in Figure 3.1.

3.1 Dataset

The dataset used in the development of this framework covers a period of four years and four months, from August 2016 to the end of December 2020, and includes



(a) 1→2→-1→3→4→-1→-1 **(b)** 1→3→2→-1→-1→5→1→2→-1→3→4→-1→-1→-1→-1

Figure 3.1: Visual representation of some string-encoded trees.

over 31.4 million traffic events and over 5.6 million weather events. The region covered by the dataset is the contiguous United States.

The dataset used in this study includes a taxonomy of different types of traffic entities. These entities are organized into the following categories:

- Accident: This category refers to traffic incidents that involve one or more vehicles and may result in fatalities. Accidents can range in severity, and may involve cars, trucks, motorcycles, or other types of vehicles.

- Broken-Vehicle: This category refers to situations in which one or more vehicles are disabled on a road. This could be due to a mechanical issue, an accident, or some other cause.

- Congestion: This category refers to situations in which traffic is moving slower than expected. The severity of congestion is classified as light, moderate, or heavy using TMC codes, which are standardized codes used to describe traffic conditions.

- Construction: This category refers to ongoing construction or maintenance projects on a road that may impact traffic flow. These projects may involve lane closures, detours, or other disruptions to the normal flow of traffic.

- Event: This category refers to situations such as sports events, concerts, or demonstrations that could potentially impact traffic flow. These events may attract large crowds and result in increased traffic in the area.
- Lane-blocked: This category refers to situations in which one or more lanes are blocked due to traffic or weather conditions. This could be due to an accident, a broken-down vehicle, or some other cause. Lane closures can significantly impact traffic flow, especially on busy roads or highways.
- Flow-incident: This category refers to all other types of traffic entities that do not fit into the above categories. Examples of flow-incidents might include a broken traffic light or an animal in the road.

The traffic dataset includes a variety of different types of traffic entities, as described in the previous list. According to the table 3.1, congestion is the most frequent entity type, making up around 80% of the data. Accidents are the second most frequent entity type. The table provides more detailed information about the traffic dataset, including the number of occurrences for each type of event and the relative frequency of each type.

The weather dataset used in this study includes a taxonomy of different types of weather entities, which are organized into the following categories:

- Cold: This category refers to extremely cold conditions, with temperatures below -23.7°C.
- Fog: This category refers to low visibility conditions due to fog or haze. Fog can make it difficult to see while driving and may require reduced speeds.
- Hail: This category refers to solid precipitation in the form of ice pellets or hail. Hail can cause damage to vehicles, buildings, and crops and may require protective measures to be taken.
- Rain: This category refers to rain of any intensity, ranging from light to heavy. Rain can impact traffic flow and may require adjustments to driving behavior to ensure safety.
- Snow: This category refers to snow of any intensity, ranging from light to heavy. Snow can cause slippery roads and reduced visibility, and may require special precautions to be taken while driving.
- Storm: This category refers to extremely windy conditions, with wind speeds of at least 60 km/h. Storms can be dangerous and may cause damage to buildings, power outages, and other disruptions.

- Precipitation: This category refers to any type of solid or liquid deposit that is not classified as snow or rain. This is a generic label that was frequently observed in raw weather data and may include a wide range of different types of precipitation.

The diagram 3.2 illustrates the frequency distribution of traffic events across 50 different states in the United States. It should be noted that for the following states, there is no traffic data available from August 2016 to August 2017: AL, AR, AZ, CO, ID, KS, KY, LA, ME, MN, MS, MT, NC, ND, NH, NM, NV, OK, OR, SD, TN, UT, VT, WI, and WY.

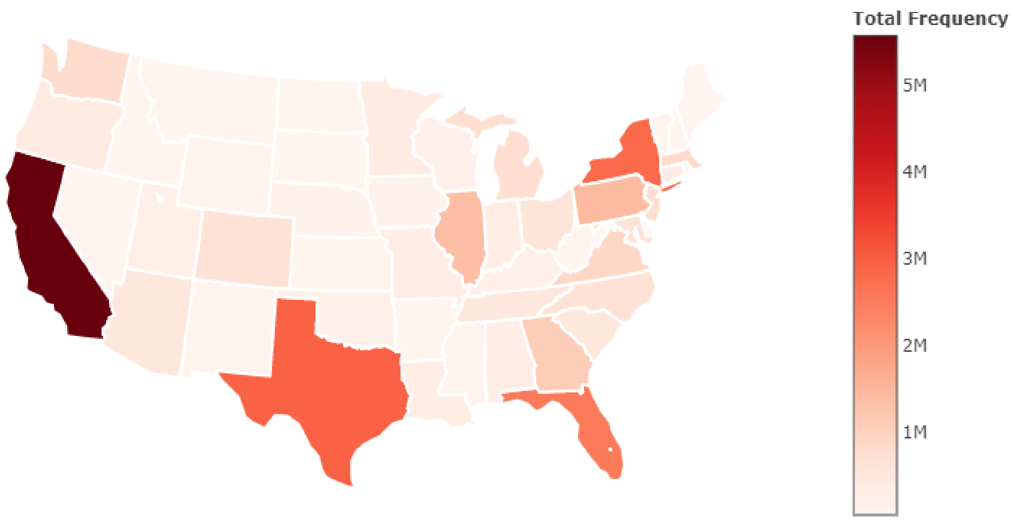


Figure 3.2: Frequency Distribution of Traffic Events from August 2016 to Dec 2020

The diagram 3.3 illustrates the frequency distribution of weather events in the same set of states.

According to the table 3.2, the most frequent weather entity types are rain, fog, and snow. The table provides more detailed information about the weather data, including the number of occurrences for each type of event and the relative frequency of each type.

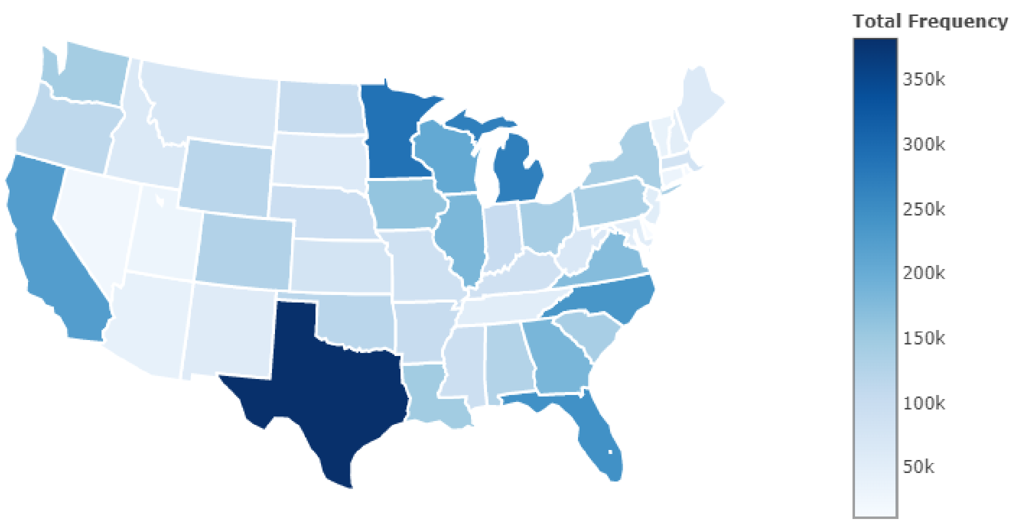


Figure 3.3: Frequency Distribution of Weather Events from August 2016 to Dec 2020

Entity Type	Raw Count	Relative Frequency
Congestion	25,100,017	80.05%
Accident	2,638,575	8.42%
Flow-Incident	1,693,415	5.4%
Construction	728,257	2.32%
Broken-Vehicle	645,985	2.06%
Lane-Blocked	501,829	1.6%
Event	47,497	0.15%
Total	**31,355,575**	**100.0%**

Table 3.1: Details on Traffic Dataset, collected for the contiguous United States from Aug 2016 to Dec 2020.

3.2 Map Reduce: An Overview

For a complete understanding of Spark parallelization in our work, it is essential to delve into the workings of the **Map Reduce** model.

Map Reduce is a distributed computing model that is used to process large amounts of data on clusters of computers. The model consists of two main phases: Map and Reduce. In the Map phase, I take input a set of events represented as key-value

Entity Type	**Raw Count**	**Relative Frequency**
Rain	3,320,935	59.55%
Fog	1,257,543	22.55%
Snow	723,547	12.98%
Cold	147,288	2.64%
Precipitation	89,365	1.6%
Storm	35,444	0.64%
Hail	2,312	0.04%
Total	**5,576,434**	**100.0%**

Table 3.2: Details on Weather Dataset, collected for the contiguous United States from Aug 2016 to Dec 2020.

pairs, where each event is a tuple of the form $(EventId, EventData)$. For example, an event in my project might have the following form:

$(EventId, Type, Severity, TMC, Description, StartTime(UTC),$
$EndTime(UTC), TimeZone, LocationLat, LocationLng, Distance(mi),$
$AirportCode, Number, Street, Side, City, County, State, ZipCode)$

In the **Map** phase, I transform each event into a set of intermediate key-value pairs that associate each event with a specific geographic area (zipCode). For example, I might produce the following intermediate pairs:

$(ZipCode, Type, Severity, TMC, Description, StartTime(UTC),$
$EndTime(UTC), TimeZone, LocationLat, LocationLng, Distance(mi),$
$AirportCode, Number, Street, Side, City, County, State, ZipCode)$

In the **Reduce** phase, I take input the intermediate set of key-value pairs produced by the Map phase and produce a final set of key-value pairs, where each pair associates a specific zipCode with the number of events that took place there. For example, the Reduce phase might produce the following final pairs:

$(ZipCode, NumberOfEvents)$

The pseudocode for the Map phase and the Reduce phase is shown in Algorithms 1 and 2, respectively.

3.3 The Algorithm

The main objective of our work is to identify cause-and-effect relationships from spatio-temporal data. This is mainly achieved through the use of a well-known

Algorithm 1 Map Phase Pseudocode

```
1: procedure MAP($EventId, EventData$)
2:     ▷ Extract the ZipCode from the EventData
3:     $ZipCode \leftarrow$ extractZipCode($EventData$)
4:
5:     ▷ Emit the intermediate key-value pair
6:     emit($ZipCode, EventData$)
7: end procedure
```

Algorithm 2 Reduce Phase Pseudocode

```
1: procedure REDUCE($ZipCode, EventDataList$)
2:     ▷ Count the number of events that took place in each ZipCode
3:     $NumberOfEvents \leftarrow$ len($EventDataList$)
4:
5:     ▷ Emit the final key-value pair
6:     emit($ZipCode, NumberOfEvents$)
7: end procedure
```

tree-mining algorithm, SLEUTH. In order to achieve this goal, raw input data must be analyzed and processed in order to construct an appropriate input for SLEUTH, a Tree DataBase.

This section will describe all the steps undertaken in the algorithm that lead to the obtaining of frequent spatio-temporal patterns. To give an overview, the algorithm starts with the integration of similar weather and traffic events in order to eliminate any duplicates, then parent-child correlations are sought between the events, which are necessary to start building the trees in the next step. This is because we want to encode the spatio-temporal correlation of events in the form of trees in order to be able to exploit the SLEUTH algorithm to carry out the identification of frequent patterns.

The main steps are as follows:

1. De-duplication of events
2. Parent-Child correlation
3. Construction of final trees
4. Extraction of frequent patterns

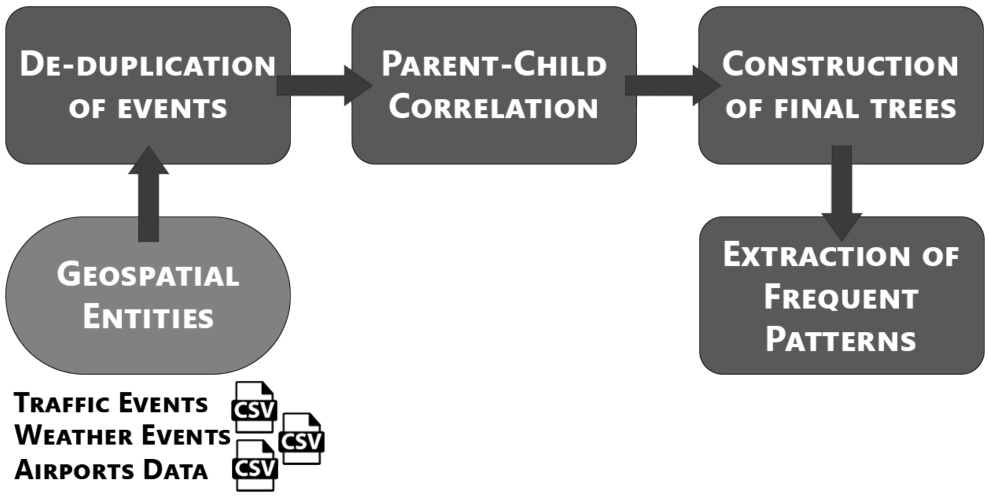


Figure 3.4: Phases of the Algorithm.

3.3.1 De-duplication of events

In order to identify the most frequent patterns in the dataset, the process begins by integrating similar traffic and weather events. To be considered similar, traffic events must be located within a certain distance between each other (measured with the Haversine distance), must occur within a 5-minute time frame of each other, must be of the same type (e.g. both Congestion), must happen on the same street and on the same side of the street. Weather events are considered similar if they are of the same type (e.g. Rain), if they occur within a certain time frame between each other, with the specific time frame varying depending on the type of event. The pseudocode for integrating similar traffic events is shown in Algorithm 3, while the pseudocode for integrating similar weather events is shown in Algorithm 4. For simplicity, the pseudocode only shows how the events are integrated. In the interest of simplicity, the pseudocode only presents the method by which the events are integrated. Instead of performing a comparisons between all possible couples of events, with a computational cost of $O(N^2)$, being one of the requirements related to spatial closeness between events, we identify for each event its own neighborhood by means of the BallTree data structure, reducing the overall computational cost to $O(NlogN)$.

Algorithm 3 Pseudocode of integrateSimilarTrafficIncidents

```
1:  if $e_1.type = e_2.type$ then
2:      $timeDiff \leftarrow abs((e_1.startTime - e_2.startTime)$
3:      $th \leftarrow 5$
4:
5:      if $timeDiff < (th * 60)$ and $e_1.street = e_2.steet$ and $e_1.side = e_2.side$ then
6:          $e_1.startTime \leftarrow min(e_1.startTime, e_2.startTime)$
7:          $e_1.endTime \leftarrow max(e_1.endTime, e_2.endTime)$
8:          $e_1.distance \leftarrow max(e_1.distance, e_2.distance)$
9:          delete $e_2$
10:     end if
11: end if
```

3.3.2 Parent-Child correlation

The second step involves searching for parent and child events among both traffic and weather events. The only strict rule in this process is that weather events can only be parent events and cannot be child events, while traffic events can be both parent and child events. This means that a weather event can be the cause of a traffic event, but a traffic event cannot be the cause of a weather event.

Algorithm 4 Pseudocode of integrateSimilarWeatherEvents

```
 1: if $e_1.type = e_2.type$ then
 2:     $timeDiff \leftarrow max((e_1.startTime - e_2.endTime), (e_2.startTime - e_1.endTime))$
 3:
 4:     if $e_1.type =$ 'snow' then
 5:         $th \leftarrow 30$
 6:     else if $e_1.type =$ 'rain' then
 7:         $th \leftarrow 15$
 8:     else
 9:         $th \leftarrow 10$
10:     end if
11:
12:     if $timeDiff < (th * 60)$ then
13:         $e_1.startTime \leftarrow min(e_1.startTime, e_2.startTime)$
14:         $e_1.endTime \leftarrow max(e_1.endTime, e_2.endTime)$
15:         $delete\, e_2$
16:     end if
17: end if
```

To perform this analysis, there are two important parameters that need to be considered: a spatial threshold, which represents the physical distance between two traffic events, and a temporal threshold, which represents the time difference between the start times of the two events. Overall, this process is designed to identify patterns of spatio-temporal entities that are both spatially co-located and temporally co-occurring. These patterns can be useful for understanding how events or phenomena interact and spread over time and space, and can be particularly useful for domains such as traffic analysis.

To perform the identification of parent and child events we first iterated over all of the weather events to store their start and end times within an Interval tree data structure, along with the corresponding event ID as the label. This allowed us to efficiently store and search for time intervals within the tree.

Next, we compared traffic events that belonged to the same neighborhood and determined if they occurred on the same street and side of the street, and if they were at most a certain temporal threshold apart. If these conditions were met, we set the event that started first as the parent event and the one that started afterwards as the child event.

After this step, we needed to search for weather events that occurred at the same start time as the traffic event. To do this, we used Interval Tree to search for weather events that overlapped with the start time of the traffic event. If any weather events were found, they were set as the parent events of the traffic event.

In these initial steps, namely, integrating similar events and searching for parent and child events, we used a very efficient algorithm called BallTree for searching for nearby events. BallTree is an algorithm used for efficient nearest-neighbor search. It works by dividing the search space into a hierarchy of "balls" (hyper-spheres) around data points, and only considering points within a certain radius of the query point. This allows the algorithm to quickly narrow down the search space and find the nearest neighbors more efficiently than a brute-force search. BallTree requires two main parameters: the data on which the tree is built, and the distance metric used to measure the distance between points. The data should be a $N \times M$ array, where N is the number of data points and M is the number of dimensions or features of each point. In our framework, M is 2. This means that the data array passed to the BallTree has shape $N \times 2$, where N is the number of data points. Each data point consists of two features or dimensions, which represent the latitude and longitude of an event.

The distance metric specifies how the distance between two points is calculated. Common choices include Euclidean distance (the straight-line distance between two points), Manhattan distance (distance between two points in a grid-like structure, such as a city grid), Haversine and others. The choice of distance metric can affect the performance and accuracy of the BallTree. We decided to use the Haversine distance. The Haversine formula is a way to calculate the great-circle distance between two points on a sphere based on their longitudes and latitudes. It is often used in geographic applications because it takes into account the curvature of the Earth and provides more accurate distances compared to simpler methods that assume a flat Earth. The pseudocode of BallTree application is shown in Algorithm 5. In this code, the values of the *distanceThresh* parameter and the Earth's radius are expressed in miles. Specifically, *distanceThresh* is set to 0.2 miles, which corresponds to about 320 meters, and the Earth's radius is set to 3958.7564 miles, which corresponds to about 6371 kilometers. By dividing a distance by the radius of the Earth, it is possible to convert the distance to an angular distance. The angular distance between two points on the surface of a sphere is the angle formed by two lines drawn from the center of the sphere to the two points. It is a measure of how far apart the points are on the surface of the sphere, and it is usually expressed in radians or degrees.

Algorithm 5 Pseudocode of the BallTree application

```
$bt \leftarrow BallTree(traffic\_events, haversine)$

▷ maximum distance to search for neighboring events
$distanceThresh \leftarrow 0.2$
$dist\_unit\_sphere \leftarrow distanceThresh/3958.7564$

▷ find the indices of the events within dist_unit_sphere of each event
$indices \leftarrow tree.query\_radius(traffic\_events, dist\_unit\_sphere)$
```

3.3.3 Construction of final trees

The next step in the process is to create relation trees from the child-parent relationships that have been identified. A relation tree is a type of tree structure that represents the relationships between events, with the root of the tree representing the parent event and the branches representing the child events. These relation trees are rooted, meaning that they have a specific starting point, labeled, meaning that they are labeled with information about the events they represent, and unordered, meaning that the order of the branches does not matter. After creating these relation trees, the process results in a forest of relation trees, which is a collection of multiple relation trees.

3.3.4 Extraction of frequent patterns

The final step in the process is to perform frequent tree pattern mining, which involves extracting all of the frequently observed unordered subtrees in the database of relation trees. To achieve this goal, we use the SLEUTH algorithm. By using this algorithm, we are able to identify the most frequently occurring subtrees in the relation tree database, which can be used to understand the relationships between different events and phenomena and to analyze spatio-temporal patterns.

We are trying to discover patterns of causation that occur over short periods of time between geospatial entities. To do this, we represent a set of weakly dependent geospatial entities as trees. Given a group of these trees (represented as $F = T_1, T_2, ..., T_k$), the problem we are trying to solve is to find all of the embedded sub-trees within F that occur relatively frequently. In other words, we want to identify patterns of causation between geospatial entities that happen often within these trees.

Two geospatial entities that occur at the same time and in the same location are considered weakly dependent. To be considered co-occurring, the difference in start times between the two entities (e_1 and e_2) must be within a certain time threshold (T-thresh). For example, if T-thresh is set to 5 minutes, and e_1 starts at 10:00 and e_2 starts at 10:03, they would be considered co-occurring. For two traffic entities to be considered co-located, their locations must match (all fields except for GPS coordinates must be the same) and they must be close to each other (the Haversine distance between them, as calculated using GPS coordinates, must be within a certain distance threshold). When trying to match a weather and traffic entity, colocation means that the airport station where the weather entity is reported must match the airport station closest to the location of the traffic entity. In other words, the weather entity and traffic entity must both be related to the same airport in order for them to be considered co-located.

3.4 Definitions

We will now provide a series of definitions in order to make it easier to understand the concepts that will be discussed later.

The **support** of an element or set of elements is the frequency with which it appears in the data. For example, if we are analyzing a dataset of products purchased in a store, the support of a particular product would be the number of times it was purchased in the time period being examined. Support can be calculated as a percentage of the total data or as an absolute number.

To formally define the relationship between a subtree (S) and a tree (T), we use the following equation to calculate the support (S,T):

$$support(S,T) = \begin{cases} 1 & \text{if S is a subtree of T} \\ 0 & otherwise \end{cases} \tag{3.1}$$

To define the relationship between a subtree (S) and a set of trees (F), we use the following equation to calculate the support (S,F):

$$support(S,F) = \frac{\sum_{T \in F} support(S,T)}{\mid F \mid} \tag{3.2}$$

For a given subtree (S), we can calculate its support (S, F) within a set of trees (F) using the previous equation. If the calculated support is greater than or equal to a minimum support threshold (min_sup), we consider the subtree to be a frequent embedded subtree in F. In other words, if the subtree occurs frequently enough within the set of trees, we consider it to be a significant pattern.

The **minimum support** is a threshold used in association rule mining to identify frequently occurring patterns in a dataset. It is defined as the minimum number of times a pattern must appear in the dataset for it to be considered a frequent pattern.

For example, if the minimum support is set to 2, then a pattern that appears in the dataset only once will not be considered frequent and will be excluded from the analysis.

Determining an appropriate value for the minimum support threshold, also known as *min_sup*, is a critical aspect. This value plays a significant role in the identification of frequently occurring patterns within a dataset.

- On one hand, if the *min_sup* value is set too high, it may result in the exclusion of interesting but rare patterns, which could be valuable for identifying meaningful patterns in the data.
- On the other hand, if the *min_sup* value is set too low, it may lead to a high computational cost and an excessive number of frequent patterns being extracted.

We have also extended the concept of **Confidence** to trees, but to better understand the explanation we first give the classic definition of Confidence:

$$confidence(A \rightarrow B) = \frac{support(A, B)}{support(A)} \tag{3.3}$$

Confidence, measures the strength of the relationship between two elements. For example, if we are analyzing a dataset of products purchased in a store, the confidence between products A and B would indicate how often product B is purchased when product A is also purchased.

Now we are going to try to give the definition of Confidence extended to trees. Starting from a tree, we extract certain sub-trees. These sub-trees are extracted by removing a leaf node from the starting tree, therefore we define the confidence of the starting tree in the following way:

$$confidence(sub\_tree \rightarrow starting\_tree) = \frac{support(starting\_tree)}{support(sub\_tree)} \tag{3.4}$$

In Figure 3.5, an example of the subtrees obtained from the string-encoded tree $1 \rightarrow 3 \rightarrow 2 \rightarrow -1 \rightarrow -1 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow -1 \rightarrow 3 \rightarrow 4$ is shown.

A numerical example is shown below:
Suppose that our starting_tree is $1 \rightarrow 3 \rightarrow 2 \rightarrow -1 \rightarrow -1 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow -1 \rightarrow 3 \rightarrow 4$ and its *support* is 0.011.

According to the methodology explained a few lines earlier, the subtrees we get are:

- $1 \rightarrow 3 \rightarrow -1 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow -1 \rightarrow 3 \rightarrow 4$
- $1 \rightarrow 3 \rightarrow 2 \rightarrow -1 \rightarrow -1 \rightarrow 5 \rightarrow 1 \rightarrow 3 \rightarrow 4$
- $1 \rightarrow 3 \rightarrow 2 \rightarrow -1 \rightarrow -1 \rightarrow 5 \rightarrow 1 \rightarrow 2 \rightarrow -1 \rightarrow 3$

Let's call them $subtree\_1$, $subtree\_2$, and $subtree\_3$, respectively.
Suppose that:

- $support(subtree\_1) = 0.054$
- $support(subtree\_2) = 0.016$
- $support(subtree\_3) = 0.025$

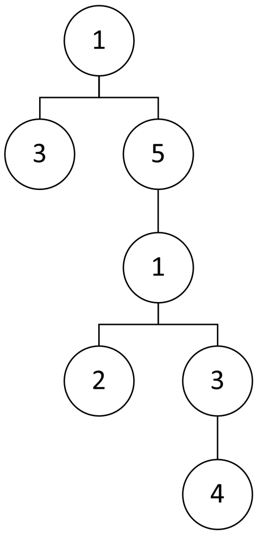
Then:

$$conf(subtree\_1 \Rightarrow starting\_tree) = \frac{support(starting\_tree)}{support(subtree\_1)} = \frac{0.011}{0.054} = 0.203 \tag{3.5}$$

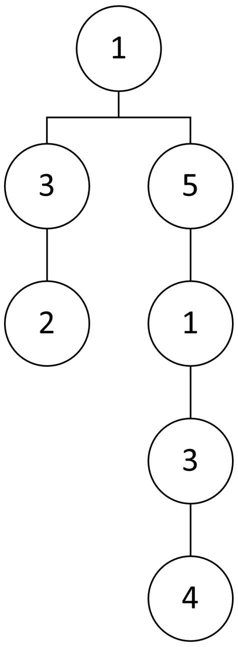
$$conf(subtree\_2 \Rightarrow starting\_tree) = \frac{support(starting\_tree)}{support(subtree\_2)} = \frac{0.011}{0.016} = 0.687 \tag{3.6}$$

$$conf(subtree\_3 \Rightarrow starting\_tree) = \frac{support(starting\_tree)}{support(subtree\_3)} = \frac{0.011}{0.045} = 0.244 \tag{3.7}$$

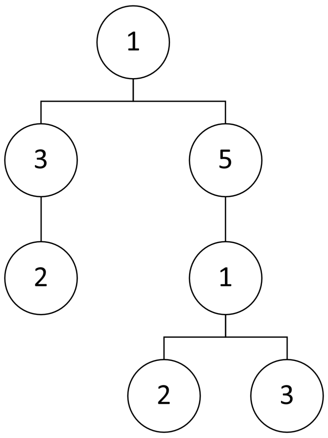
Support and confidence metrics are useful for identifying associations between elements of a dataset and for making decisions based on this information. For example, if a store observes that a particular product has high support and high confidence with another product, it might decide to place these products near each other in the store or to promote them together in a marketing campaign.



(a) 1→3→-1→5→1→2→-1→3→4



(b) 1→3→2→-1→-1→5→1→3→4



(c) 1→3→2→-1→-1→5→1→2→-1→3

Figure 3.5: Visual representation of the subtrees extracted from the *starting_tree* shown is Figure 3.1b.

Chapter 4

Experimental evaluation

In this chapter, the main improvements over the original framework will be explained, the results obtained from various experiments will be presented, and the execution times of each phase of the new algorithm will be discussed.

4.1 Improvements

This section explains the main improvement made to the framework developed by Sobhan Moosavi's team [1]. The new framework is significantly faster than the previous version, due in part to the fact that the code was rewritten using PySpark, leveraging on distributed computation over multiple nodes. In fact, PySpark is designed specifically for distributed data processing and is well-suited to handling large datasets. It is built on top of the Apache Spark platform, which is a powerful open-source data processing engine that is widely used for data processing, machine learning, and analytics tasks.

One of the key advantages of PySpark is its ability to scale horizontally across multiple machines, making it well-suited to handling large datasets. It is also highly flexible and can be used with a variety of data storage systems, including Hadoop Distributed File System (HDFS), Amazon S3, and Cassandra. PySpark also integrates seamlessly with other popular data processing libraries and frameworks, such as Pandas and NumPy, making it easy to use in a wide range of data processing pipelines.

As an example of the improved performance of the new framework, it is noted that the old framework took approximately 6 hours to process just 1% of the data for the integration phase of similar traffic events (3.3.1). In contrast, the new framework is able to integrate the entirety of similar events, both traffic and weather, in just over 2 hours. Based on this comparison, it is estimated that the old framework would have taken about 60 hours to complete the integration of similar traffic events. This demonstrates a significant improvement in the speed and efficiency of the new framework, likely due in part to the use of PySpark for the implementation.

The new framework developed in this study includes a number of improvements over the previous version developed by Sobhan Moosavi's team [1], including increased speed and efficiency. However, there is one section of the framework that remains unchanged in terms of efficiency. This is the final part of the process, in which the most frequent patterns are searched for by applying the SLEUTH algorithm. This algorithm is designed to identify patterns of immediate or short-term impacts within a dataset.

Another reason why the new framework is faster is due to the use of the BallTree algorithm in the parts of the code where the neighborhood of an event was considered needed to be calculated.

While the other sections of the framework have been optimized and parallelized using PySpark in order to improve performance, the implementation of the SLEUTH algorithm has not been modified. PySpark allows data to be processed in parallel across multiple machines. However, the SLEUTH algorithm is not well-suited to this type of parallelization and so it has been left unchanged in the new framework. Despite this limitation, the overall performance of the new framework is significantly improved over the previous version due to the optimization and parallelization of the other sections of the code.

4.2 Experiments and Results

In this section, the different experiments that were performed using the proposed framework are detailed. For each experiment, the specific configurations that were used are described, along with the results that were obtained.

The different configurations varied in the temporal threshold that was considered. This threshold was used in the search for parent and child events and was expressed in minutes. The values that were evaluated were 10, 15, and 20 minutes. Larger temporal thresholds were not considered because it was believed that 20 minutes was already a sufficient amount of time to consider a delay in a cause and effect relationship.

In order to search for frequent patterns using SLEUTH, certain cities were selected. Specifically, two of the most populous cities in the United States, New York City and Los Angeles, and another less populous city, Boston, were chosen.

Figure 4.1 shows the number of patterns that were extracted in New York City for each temporal threshold. The data suggests that as the temporal threshold increases, the number of extracted patterns also increases. However, this trend is not consistently observed as demonstrated in Figure 4.2, which shows the results for Boston. In regards to the city of Los Angeles, with a threshold value of 20 minutes, there is no increase in the number of patterns extracted compared to the configuration with 15 minutes as can be seen in Figure 4.3.

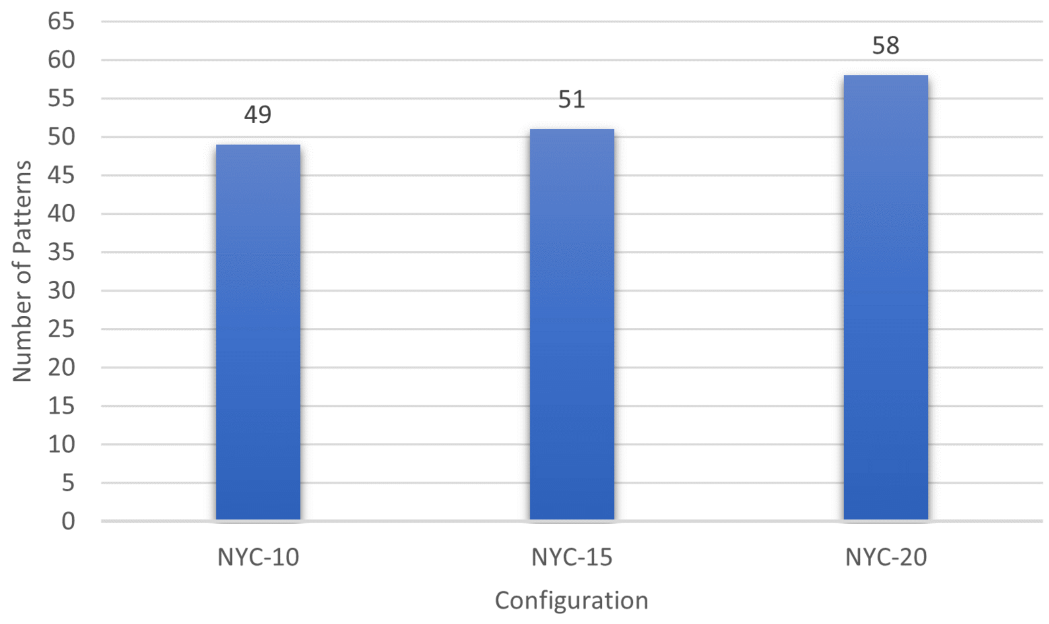


Figure 4.1: The image shows a bar graph representing the number of patterns extracted in the city of New York with different temporal thresholds

The graphs shown in Figure 4.4, 4.5, and 4.6 present a study on the initiation of frequent patterns in the three cities taken into consideration.

In Boston, with a threshold of 10 minutes, 59.18% of patterns were initiated by a weather event, while 40.82% were initiated by a traffic event. As the threshold increases, the percentage of patterns initiated by a weather event decreases and the percentage of patterns initiated by a traffic event increases.

In Los Angeles, with a threshold of 10 minutes, 52.78% of patterns were initiated by a weather event and 47.22% were initiated by a traffic event. As the threshold increases, the percentage of patterns initiated by a weather event and traffic event both decrease and eventually reach 50% each at a threshold of 15 and 20.

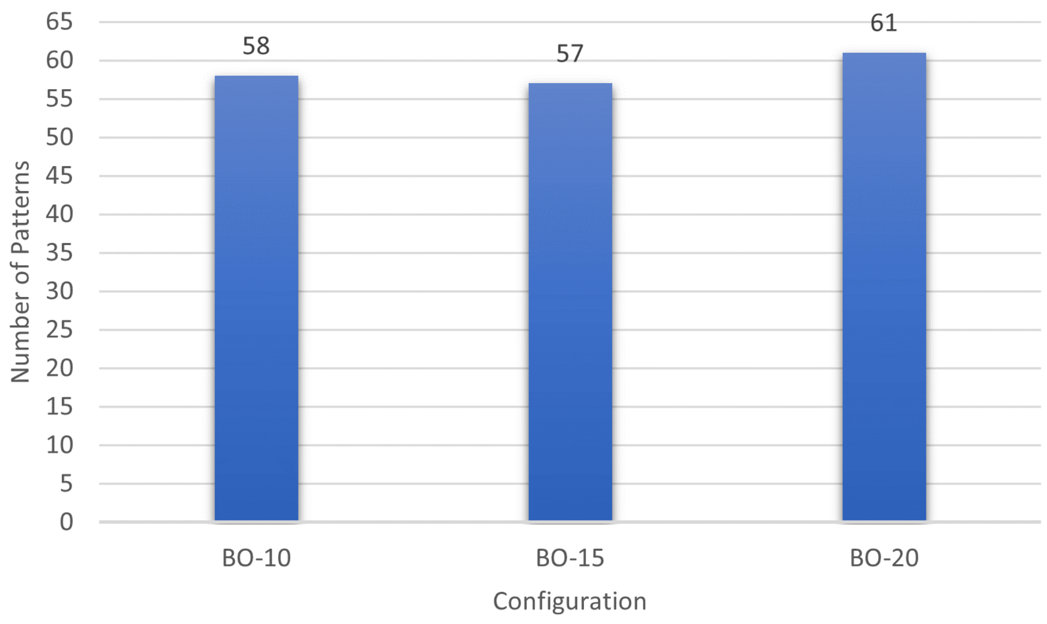


Figure 4.2: The image shows a bar graph representing the number of patterns extracted in the city of Boston with different temporal thresholds

In New York, with a threshold of 10 minutes, 52.50% of patterns were initiated by a weather event and 47.50% were initiated by a traffic event. At a threshold of 15 and 20, the percentage of patterns initiated by both weather events and traffic events decreases to 50% each. However, at a threshold of 20, there is a shift towards patterns initiated by traffic events, with 57.14% initiated by traffic and 42.86% initiated by weather events.

Furthermore, we conducted a more in-depth analysis of the data. The results shown in Figures 4.7, 4.8 and 4.9 show the most frequent events that initiated patterns in each city based on the different thresholds.

The analysis was performed by examining the percentage of occurrences of various events, such as Snow, Rain, Construction, Congestion, Fog, Incident-flow, Accident, Cold, and Precipitation, and averaging the results across three different thresholds (10, 15, and 20).

The event types that initiated frequent patterns in Boston, based on the average of the thresholds, were Rain and Congestion. Rain contributed approximately 44.08% to the frequent patterns, while Congestion accounted for approximately

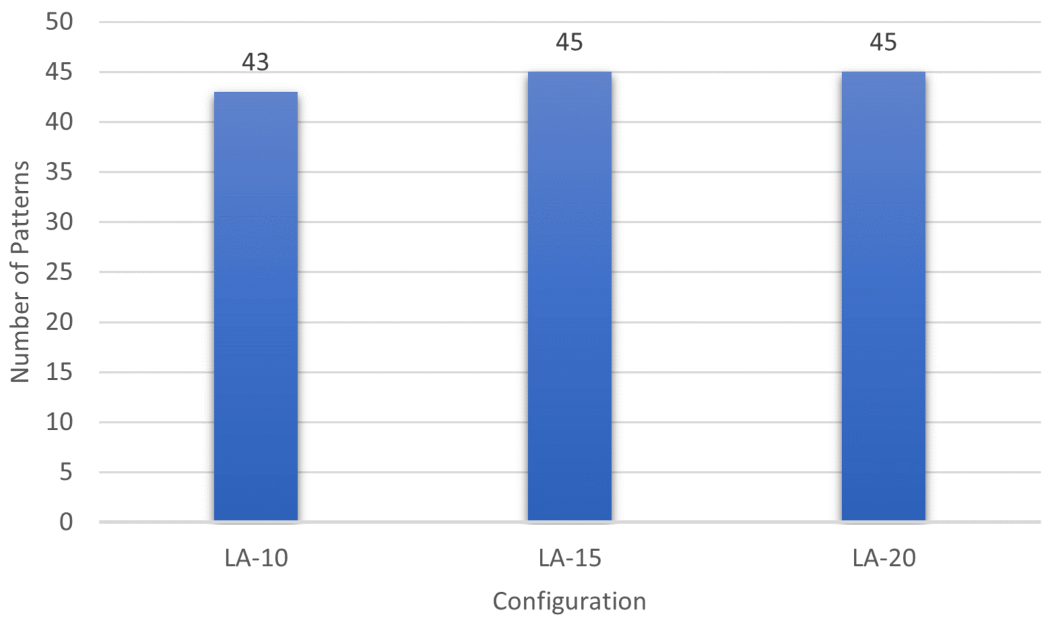


Figure 4.3: The image shows a bar graph representing the number of patterns extracted in the city of Los Angeles with different temporal thresholds

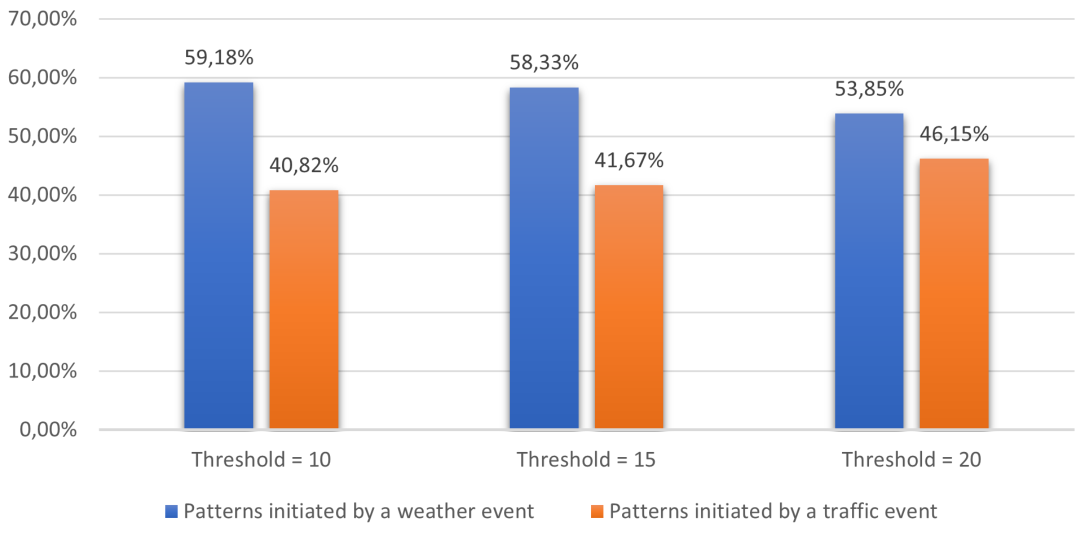


Figure 4.4: Patterns initiated by a weather/traffic event in Boston.

31.37%. Construction, Fog, and Incident-Flow also played a role, but to a lesser extent. Snow contributed the least, with an average of 5.97%.

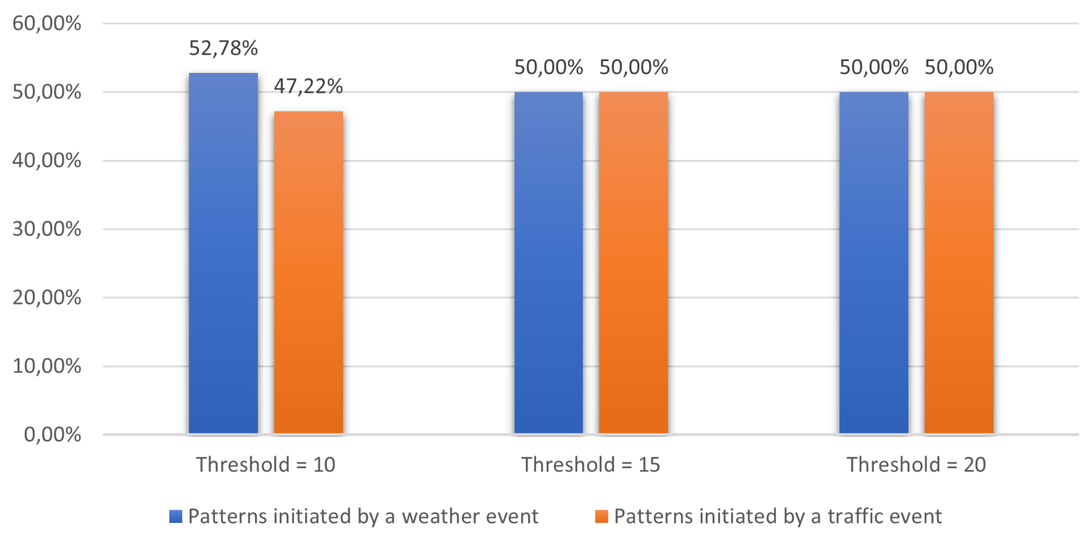


Figure 4.5: Patterns initiated by a weather/traffic event in Los Angeles.

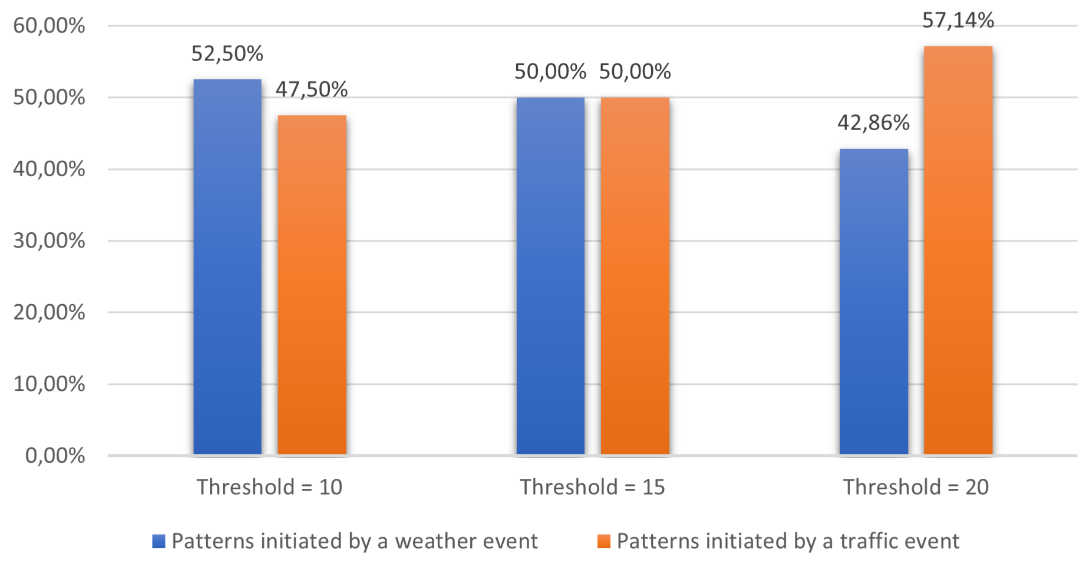


Figure 4.6: Patterns initiated by a weather/traffic event in New York City.

In Los Angeles, the top two events that initiated frequent patterns were Fog and Congestion. On average, Fog accounted for 37.44% of the frequent patterns, while Congestion made up 33.73%. Rain and Incident-Flow also played a significant role, with 11.11% and 13.16% respectively. Accidents made up a small portion of the

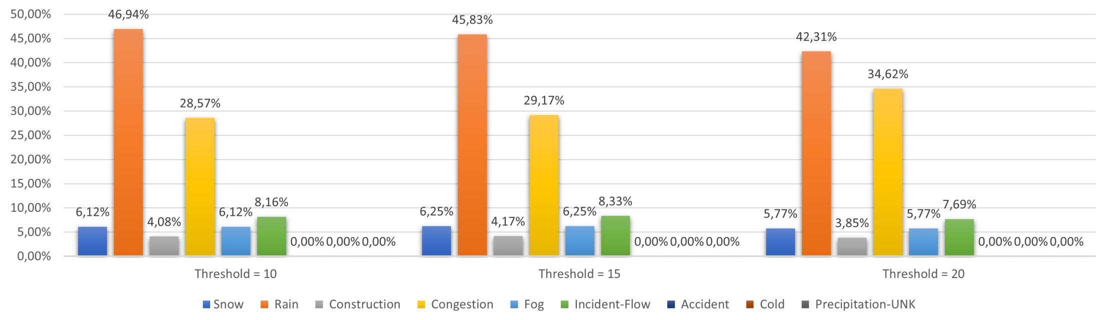


Figure 4.7: Events type that initiated frequent Patterns in Boston.

frequent patterns, with 2.7%.

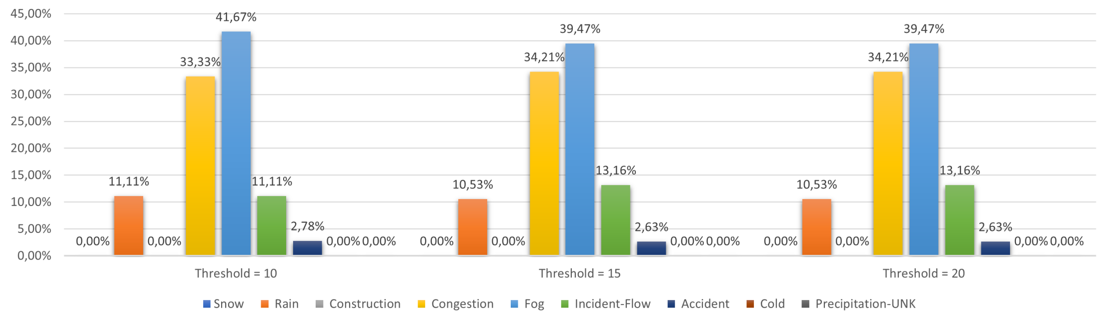


Figure 4.8: Events type that initiated frequent Patterns in Los Angeles.

In New York City, the most significant events that initiated frequent patterns were Congestion and Rain. Congestion accounted for an average of 37.38% of the frequent patterns, while Rain contributed to an average of 30.71%. Snow, Fog, Incident-Flow, and Precipitation also played a role, but to a lesser extent. Accidents did not contribute to any of the frequent patterns in New York City.

The metrics that were taken into consideration to evaluate the quality of frequent patterns are **Support** and **Confidence**. Support and confidence are commonly used in data mining and data analysis to identify the strength of a pattern.

The support of an element or set of elements is the frequency with which it appears in the data and Confidence, measures the strength of the relationship between two elements. The more detailed definitions of these metrics are explained in Section 3.4.

Figures 4.10, 4.11, and 4.12 show the most frequent patterns in each of the three

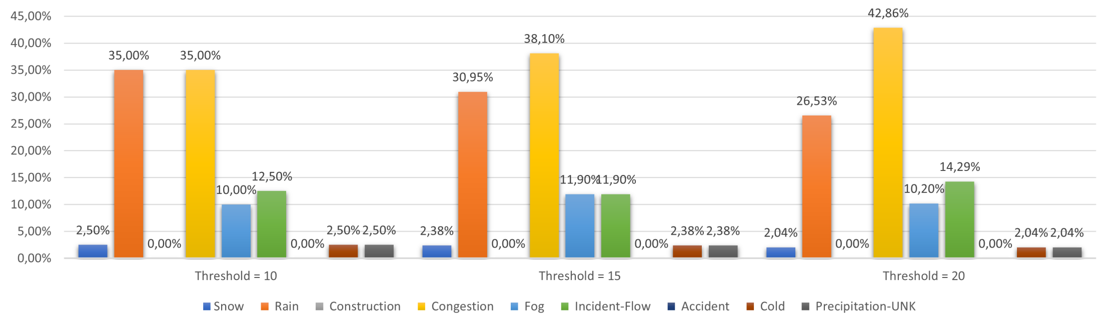


Figure 4.9: Events type that initiated frequent Patterns in New York City.

configurations with the highest Support for the city of Boston. It can be noted that as the temporal threshold increases, the support of traffic events slightly increases while the support of weather events remains constant. This behavior can be more clearly seen in Tables 4.1, 4.2 and 4.3 which show the absolute supports of patterns composed of a single event in the cities of Boston, New York City and Los Angeles as the temporal threshold varies.

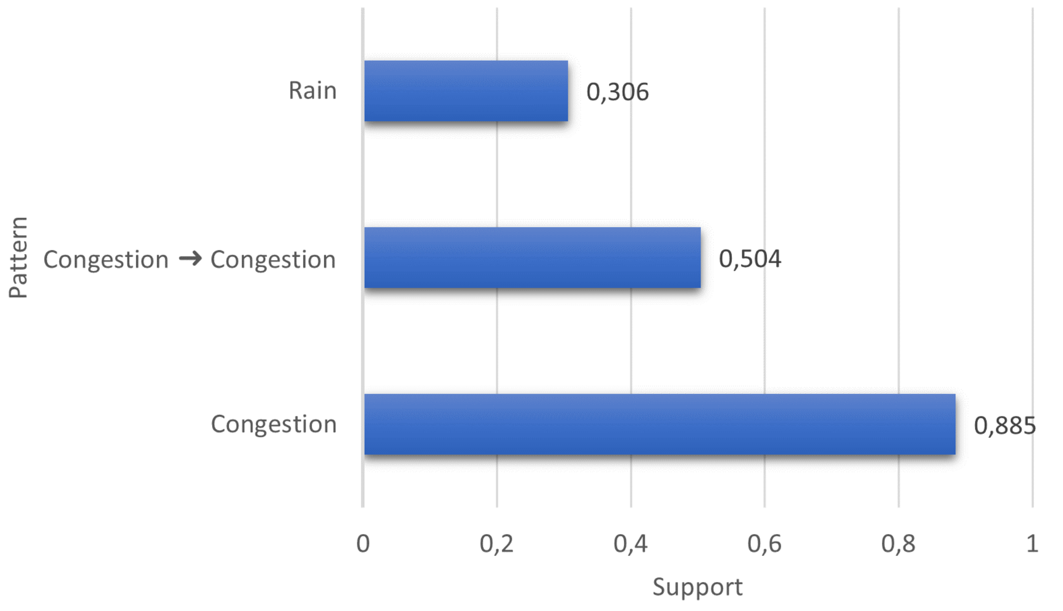


Figure 4.10: The bar chart displays the most frequent patterns in the city of Boston with the highest Support and a temporal threshold of 10 minutes.

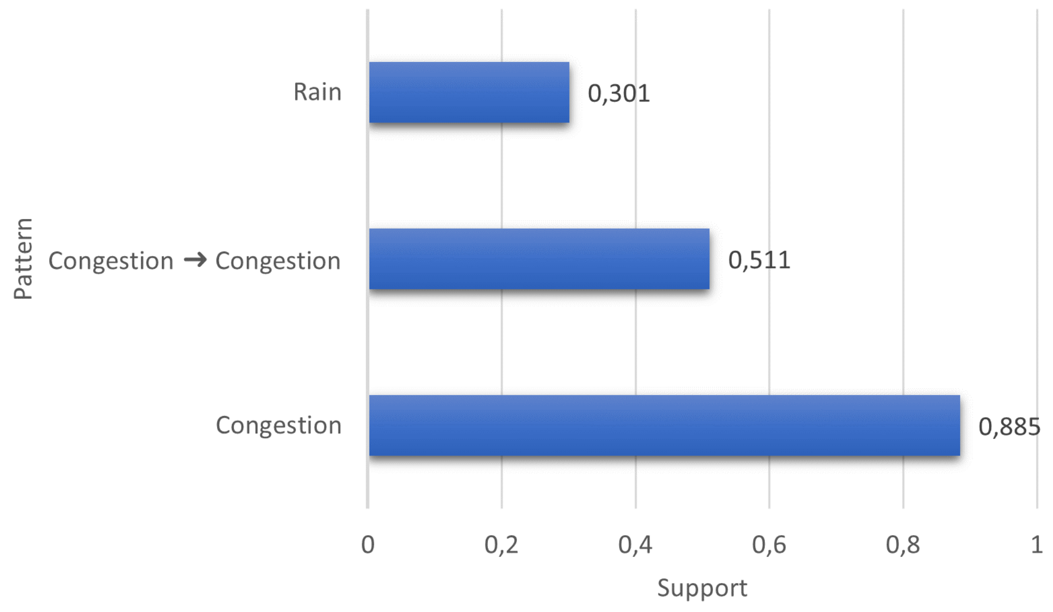


Figure 4.11: The bar chart displays the most frequent patterns in the city of Boston with the highest Support and a temporal threshold of 15 minutes.

Pattern	**Support_10**	**Support_15**	**Support_20**
Congestion	9.813	10.010	10.219
Rain	3.386	3.407	3.394
Incident-Flow	1.369	1.382	1.400
Construction	1.067	1.068	1.073
Snow	593	598	592
Fog	538	537	533
Accident	524	530	539
Broken-Vehicle	162	171	178
Event	142	168	168

Table 4.1: This table shows the absolute supports for patterns composed of a single event in the city of Boston as the temporal threshold varies.

Figures 4.13, 4.14, and 4.15 show the most frequent patterns in each of the three configurations with the highest Confidence for the city of Boston. For a clearer representation of the patterns, Figure 4.16 displays the pattern trees from Figure 4.13.

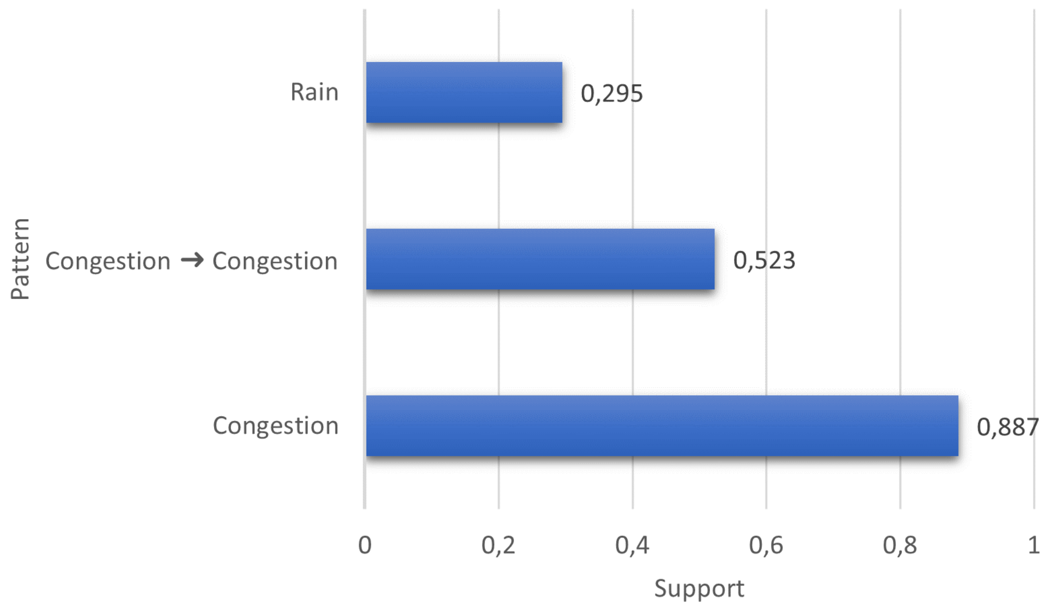


Figure 4.12: The bar chart displays the most frequent patterns in the city of Boston with the highest Support and a temporal threshold of 20 minutes.

Pattern	**Support_10**	**Support_15**	**Support_20**
Congestion	71.455	73.811	75.876
Incident-Flow	15.769	16.373	16.882
Rain	14.712	14.599	14.492
Fog	3.868	3.850	3.823
Precipitation-UNK	3.003	2.974	2.934
Snow	1.641	1.627	1.611
Construction	1.609	1.624	1.633
Accident	1.324	1.353	1.384
Cold	950	945	929

Table 4.2: This table shows the absolute supports for patterns composed of a single event in the city of New York City as the temporal threshold varies.

In the graphs that represent patterns with the highest confidence, the child node is underlined. For example, in Figure 4.13, the Pattern $\underline{Snow} \rightarrow Congestion$ indicates that the *Snow* event is the child node. The complete notation should be $Snow \Longrightarrow Snow \rightarrow Congestion$.

Pattern	Support_10	Support_15	Support_20
Congestion	93.064	96.473	99.532
Incident-Flow	21.261	21.950	22.672
Fog	14.806	14.741	14.690
Accident	6.961	7.115	7.240
Rain	6.809	6.793	6.763
Lane-Blocked	2.642	2.712	2.767
Broken-Vehicle	2.339	2.403	2.457

Table 4.3: This table shows the absolute supports for patterns composed of a single event in the city of Los Angeles as the temporal threshold varies.

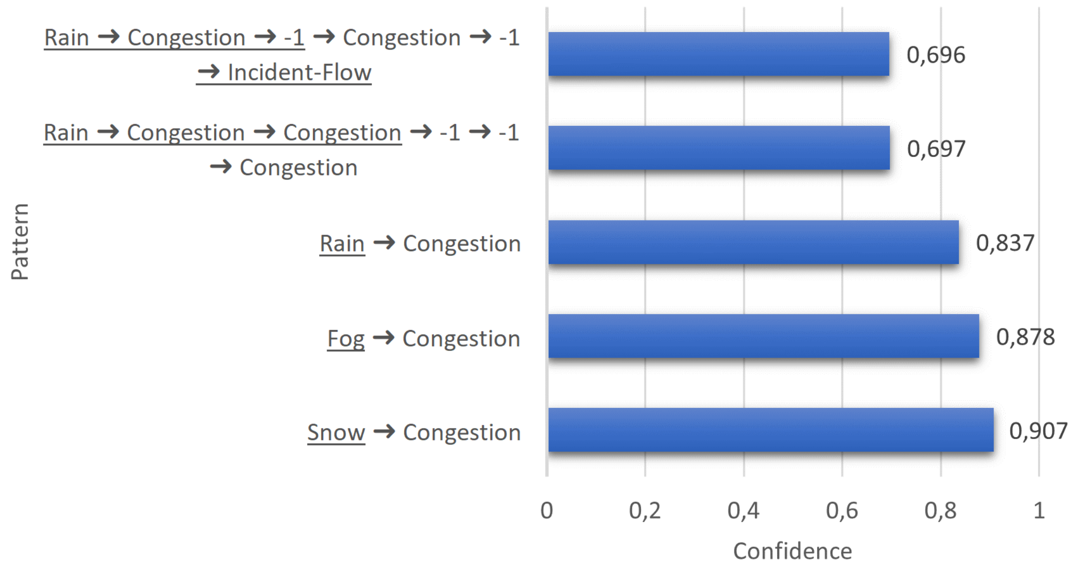


Figure 4.13: The bar chart displays the most frequent patterns in the city of Boston with the highest Confidence and a temporal threshold of 10 minutes. These patterns are represented by trees in Figure 4.16

We also present the patterns with high absolute support in the three cities: Boston, Los Angeles, and New York City. Each city is represented by two graphs, one showcasing the patterns with a length of 2 that have the highest absolute support and the other graph showcasing the patterns with a length of 3. Figures 4.18 and 4.19 are related to Boston, Figures 4.20 and 4.21 to Los Angeles, and Figures 4.22 and 4.23 to New York City.

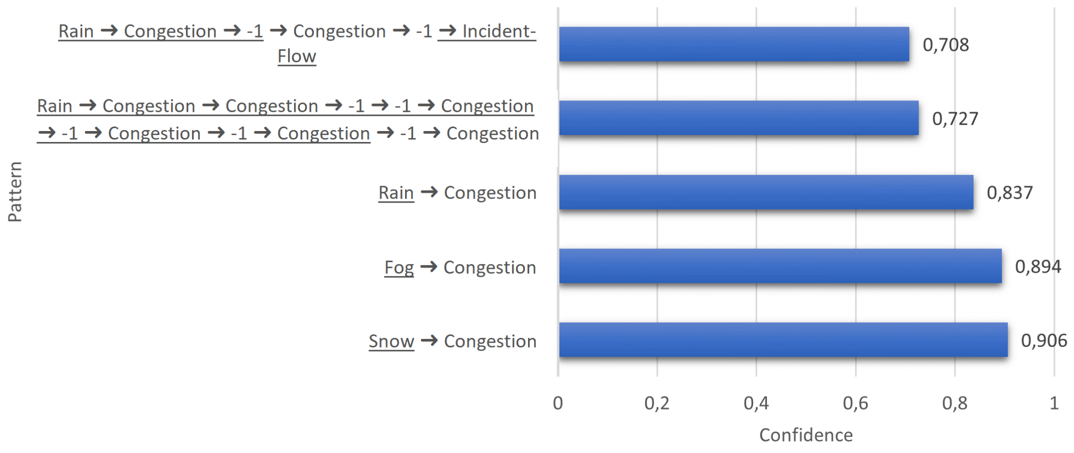


Figure 4.14: The bar chart displays the most frequent patterns in the city of Boston with the highest Confidence and a temporal threshold of 15 minutes.

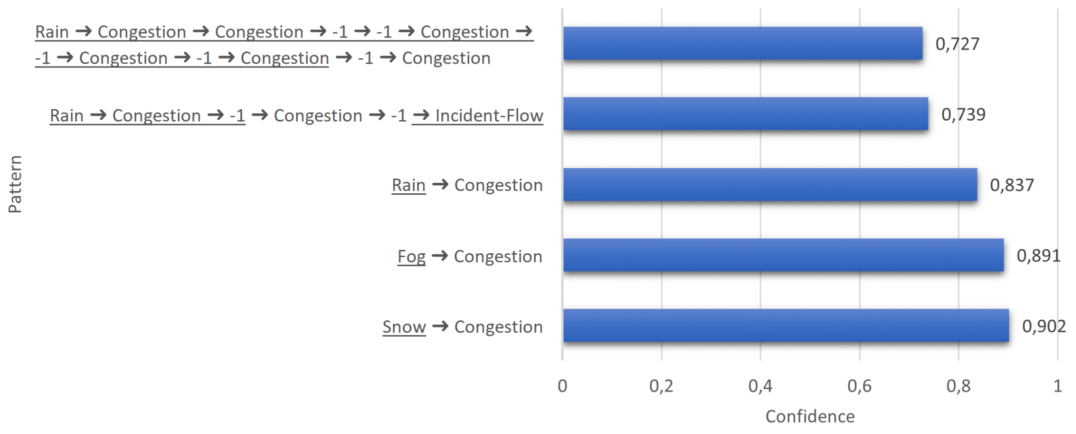
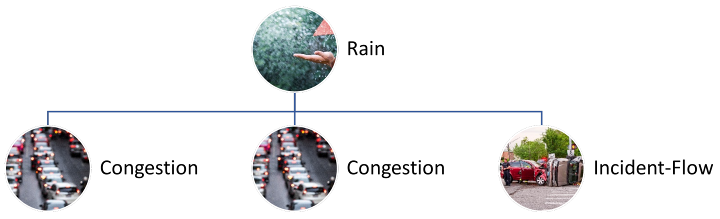


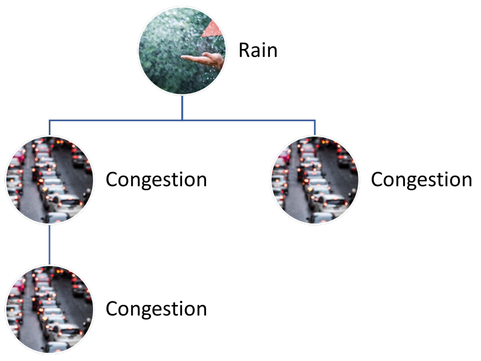
Figure 4.15: The bar chart displays the most frequent patterns in the city of Boston with the highest Confidence and a temporal threshold of 20 minutes.

For example, in 4.18, there is the pattern $Incident - Flow \rightarrow Congestion$. In this pattern, an incident-flow, such as a traffic accident, a road closure, or roadwork, can cause disruptions in the flow of traffic, leading to congestion. This congestion can cause further delays and reduce overall mobility. The incident-flow acts as the trigger for congestion, which can have a cascading effect on the rest of the transportation system.

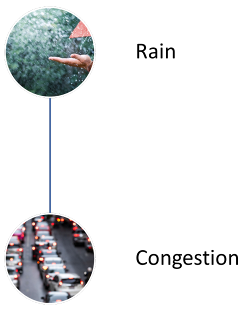
The graphs regarding the 15 and 20 minute thresholds have been omitted as



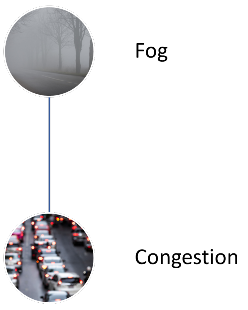
(a) Rain→Congestion→-1→Congestion→-1→Incident-Flow



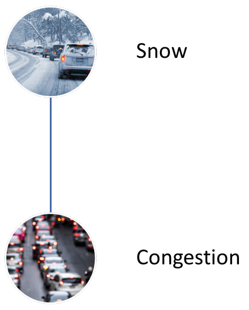
(b) Rain→Congestion→Congestion→-1→-1→Congestion



(c) Rain→Congestion



(d) Fog→Congestion



(e) Snow→Congestion

Figure 4.16: Visual representation of the string-encoded tree presented in Figure 4.13.

the patterns do not change.

Based on the patterns presented earlier, one may get the impression that our work is limited to identifying only simple and straightforward patterns. However, that is not the case. Our work is actually capable of extracting a wide range of patterns, including those that are much more complex.

Certainly, these more advanced patterns have lower support levels. They abide by the antimonotonicity property, which states that as the size of the tree increases, the support decreases. This is because the more complex the pattern, the more variables are involved, making it harder to find instances where the pattern is supported by the data. The pattern shown in Figure 4.17 is a prime example of these complex patterns that were identified in the city of Los Angeles.

The pattern could represent a real-life scenario as follows:

- The Fog event could refer to a dense fog that descends upon a city, reducing visibility and making it difficult for drivers to see the road ahead.
- The first instance of $Congestion$ refers to the traffic that builds up on the roads as drivers slow down or stop due to the poor visibility.
- The second and third instances of $Congestion$ also refer to additional traffic congestions that develop in different parts of the city, independently of one another, as drivers adjust their speeds and routes due to the dense fog.
- The $Incident - Flow$ event refers to the consequences of the congestions, such as increased wait times, detours, and frustration for drivers. The incident flow could also potentially cause additional incidents, such as accidents or road closures.

In this scenario, the Fog event is the cause of each instance of $Congestion$ and the $Incident - Flow$, and the pattern shows how a single, initial event can have multiple, independent effects that can result in a final outcome. The pattern illustrates the idea that even a small change, like a dense fog, can have a big impact and cause a cascade of events, demonstrating the complexity of cause-and-effect relationships in real-life situations.

4.3 Execution times

In this section, we present the execution times of each phase of the algorithm when applied to the entire dataset presented in Section 3.1.

The first phase, integrating similar traffic and weather events, took around 2 hours to complete. The second phase, which consists of searching for parent and

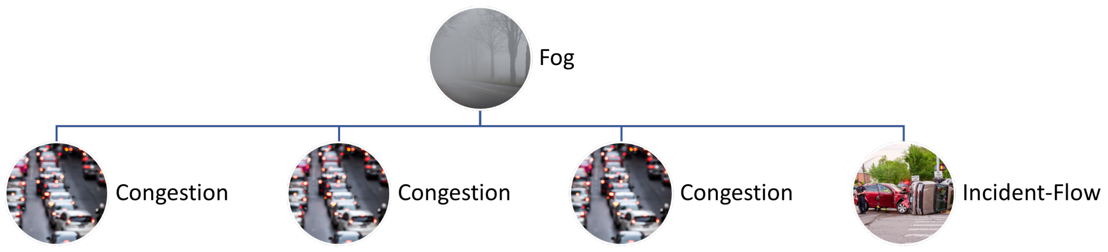


Figure 4.17: Example of a complex pattern extracted for the City of Los Angeles.

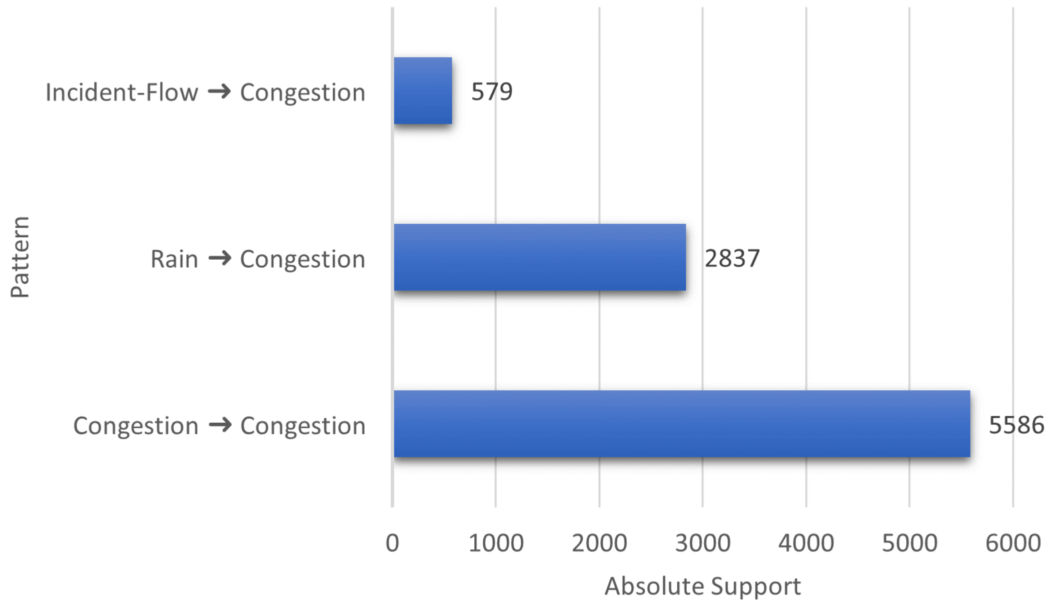


Figure 4.18: Boston Patterns with High Absolute Support: Length 2, temporal threshold 10 minutes.

child events, is the heaviest phase of the algorithm, it took about 17 hours and 50 minutes. The third phase, creating relation trees from the child-parent relationships, had varying execution times depending on the city chosen. For example, for the city of Boston, this phase took 16 minutes to complete. Finally, the last phase, extracting frequent patterns, also had varying execution times depending on the

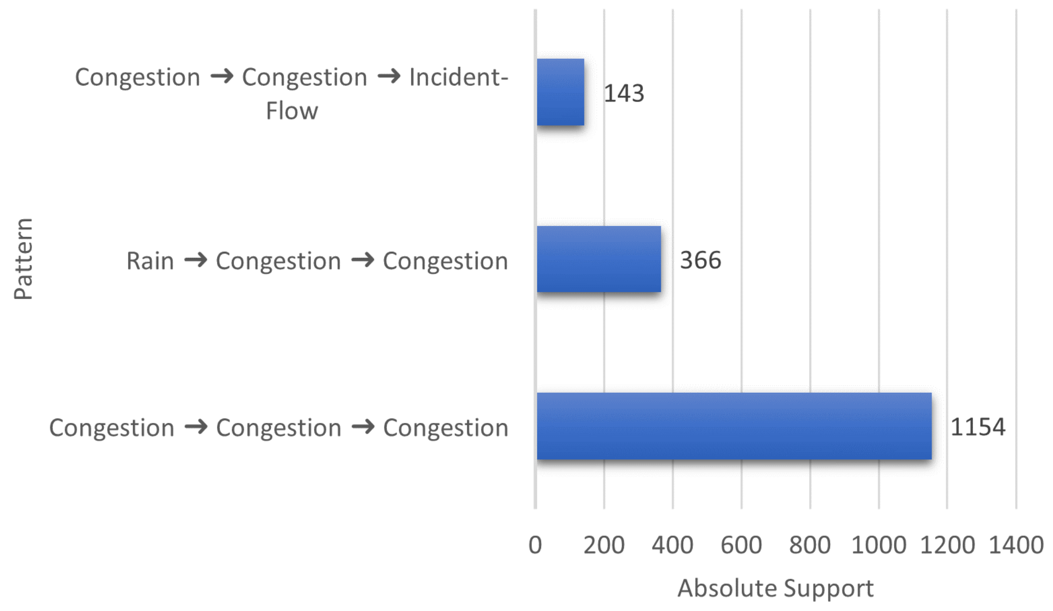


Figure 4.19: Boston Patterns with High Absolute Support: Length 3, temporal threshold 10 minutes.

city and the minimum support set. For example, for the city of Boston with a minimum support of 75, this phase took around 6 hours to complete, for the City of Los Angeles with a minimum support of 1095 took around 1 hour and 30 minutes, and for the City of New York with a minimum support of 700 took around 50 minutes.

As seen in Table 4.4, the minimum support chosen for each city is related to the number of input trees that were given to the SLEUTH algorithm, representing around 1% of the total number of input trees.

4.4 Scalability Test

An scalability test was conducted to evaluate the performance of the code with an increasing amount of data. The test was performed using data from the city of Los Angeles, which had the highest number of traffic events in the dataset, approximately 850,000. The data was selected and sorted in descending order, from the most recent to the oldest event. Five tests were conducted, each with a different amount of events: 10,000, 50,000, 100,000, 200,000, and 500,000. The execution times for each test are presented in Graph 4.24, the x-axis displays the number of events, ranging from 10,000 to 500,000, while the y-axis displays the elapsed time

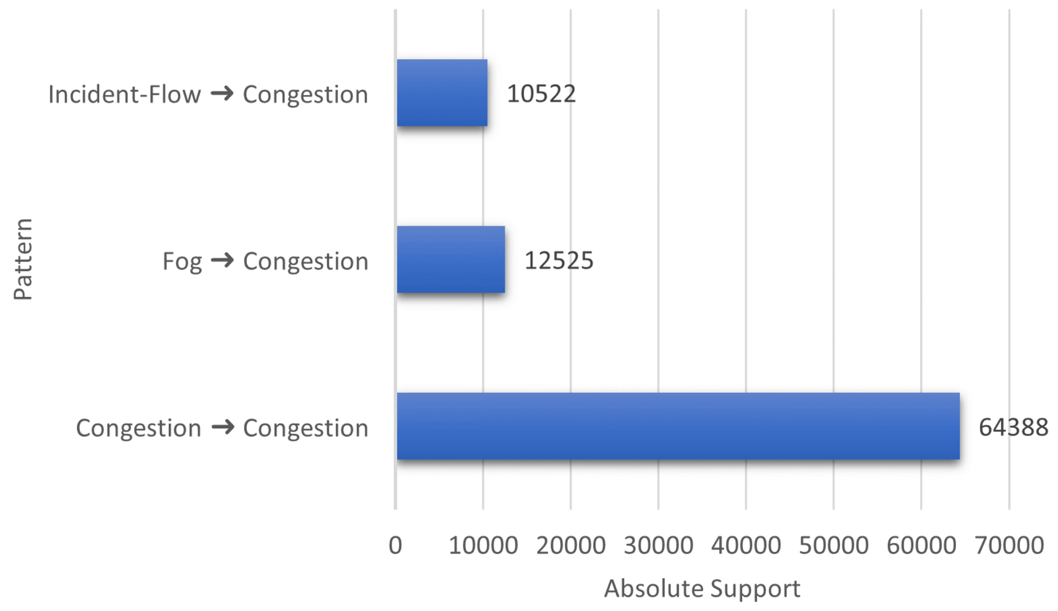


Figure 4.20: Los Angeles Patterns with High Absolute Support: Length 2, temporal threshold 10 minutes.

City	MinSup	T-threshold	Input trees	Support %
Boston	75	10	11.083	0,68 %
		15	11.308	0,66 %
		20	11.521	0,65 %
New York City	700	10	74.569	0,94 %
		15	76.943	0,91 %
		20	79.045	0,89 %
Los Angeles	1095	10	99.578	1,10 %
		15	103.078	1,06 %
		20	106.254	1,03 %

Table 4.4: Table displaying the utilized parameters for SLEUTH including the minimum support values and the corresponding number of input trees for each configuration.

in seconds, represented on a logarithmic scale. Furthermore, the elapsed time in seconds for each of the experiments can be viewed in greater detail in Table 4.5.

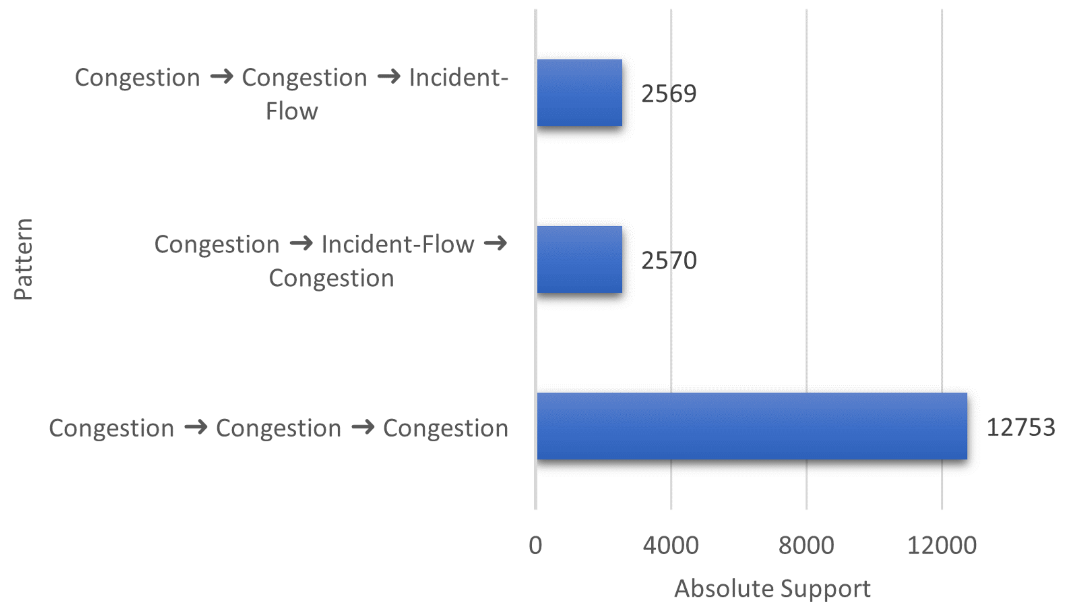


Figure 4.21: Los Angeles Patterns with High Absolute Support: Length 3, temporal threshold 10 minutes.

Record	Elapsed time (seconds)
10.000	60 s
50.000	120 s
100.000	285 s
200.000	890 s
500.000	8000 s

Table 4.5: Table displaying the elapsed time in seconds for each of the 5 experiments with the city of Los Angeles.

4.5 Comparison of Results with STInvMiner

Furthermore, we also compared the results obtained from our work with the results obtained from the STInvMiner work [5], in order to understand if the patterns had some kind of similarity. This comparison was carried out through code that transforms the results obtained from our work into sequences, in order to be of the same type as those of STInv and subsequently performs the inverse operation, that is, transforms the STInv sequences into trees, that is the suitable data type for our work.

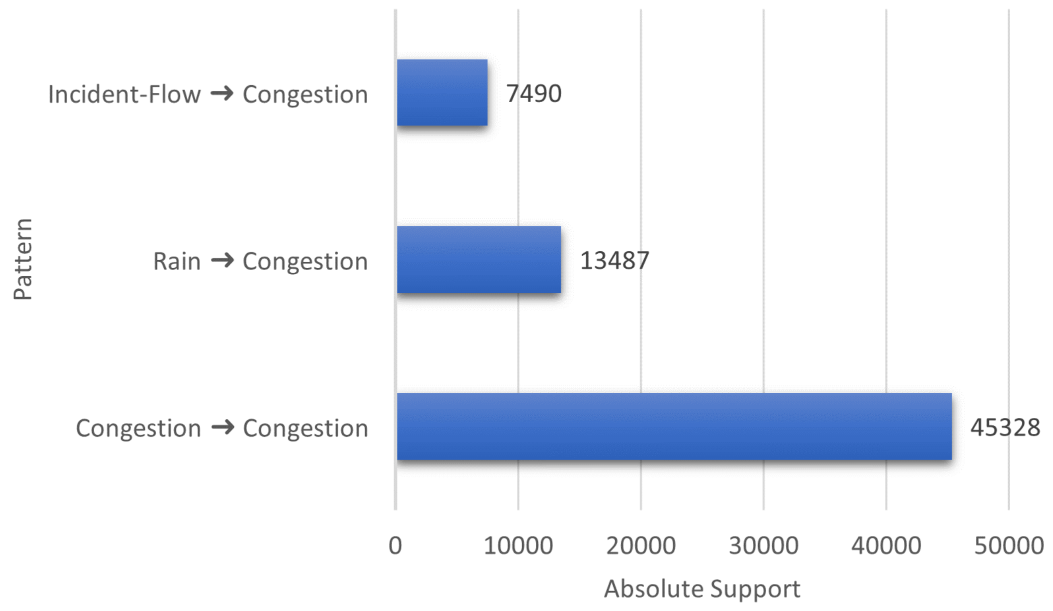


Figure 4.22: New York City Patterns with High Absolute Support: Length 2, temporal threshold 10 minutes.

The code used for comparison contains a parameter named $offset$, which plays a crucial role in determining the behavior of the comparison process. When the offset is set to 1, it mandates that the STInv patterns must be sequential. On the other hand, when the offset is set to 0, it signifies that sequentiality in the STInv patterns is not mandatory.

The graphs in Figures 4.25 and 4.26 show the comparison between our results and the results obtained from the STInvMiner work for the cities of Boston and Los Angeles, respectively, with the $offset$ parameter set to 0. The y-axis shows the coverage value, with a range from 0 to 1. A value of 1 on the y-axis indicates that all the patterns extracted by one particular algorithm are covered by the other algorithm as well. For example, a coverage value of 0.65 for STInv/SLEUTH indicates that the patterns extracted by STInv cover 65% of the patterns extracted by our method using SLEUTH. A value of 0 on the y-axis indicates that none of the patterns are covered. As you can see from the graph, the coverage value is low for most of the configurations, indicating that our results do not fully cover all the patterns extracted by STInvMiner.

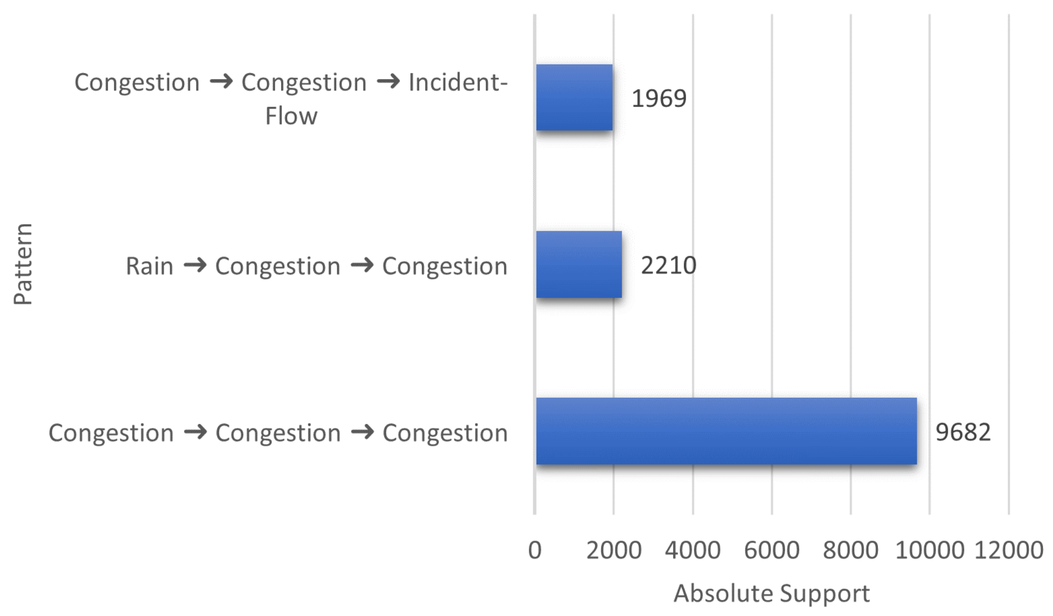


Figure 4.23: New York City Patterns with High Absolute Support: Length 3, temporal threshold 10 minutes.

The graphs in Figures 4.27 and 4.28 show the comparison between our results and the results obtained from the STInvMiner work for the cities of Boston and Los Angeles, respectively, with the $offset$ parameter set to 1.

As can be seen from the graphs, we observe a decrease in coverage from the STInv patterns. This is due to the fact that the $offset$ was set to 1, which mandates sequentiality in the STInv patterns. The decrease in coverage is therefore justified.

In this thesis, the reimplemented approach leverages on PySpark framework to extract tree-based patterns which are comparable to those presented in STInvMiner algorithm, without the need of time discretization. In fact, the algorithm adopted in this thesis work analyzes continuous time, without performing any windowing operation on the temporal dimension. Discretizing time involves dividing time into fixed intervals, such as seconds or minutes. This results in each event being assigned to a specific interval and time information being represented in a discrete manner. However, this process can lead to a loss of information regarding continuous time. My approach avoids this issue by using timestamps to maintain the continuity of time information, placing emphasis on considering the sequentiality of events, which can provide valuable information about the order in which events occur.

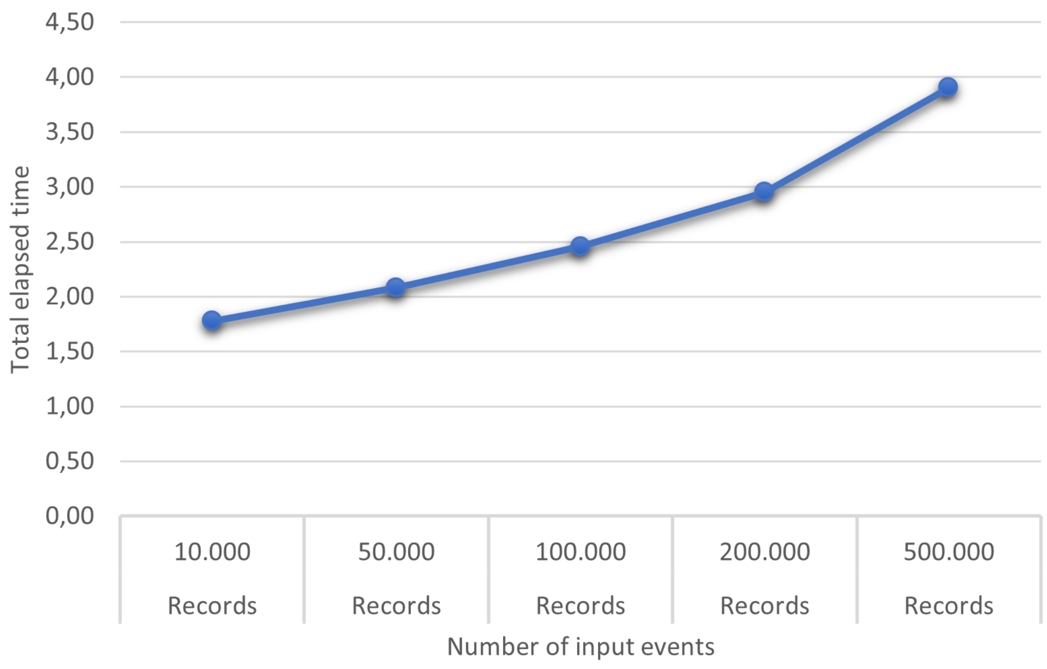


Figure 4.24: Graph showing the elapsed time in seconds for each of the 5 experiments with the city of Los Angeles, the y-axis is on a logarithmic scale.

To summarize, both approaches are valid and can address different problems, depending on the nature of the issue. Additionally, it is worth mentioning that these are two distinct event mining algorithms: the SLEUTH-based approach identifies event trees, which can potentially be far from the starting event, while STInv is limited to being close to the reference event. Event trees are easier to interpret than the structured sequences produced by STInv and allow for better highlighting of parent-child relationships and cause-effect connections.

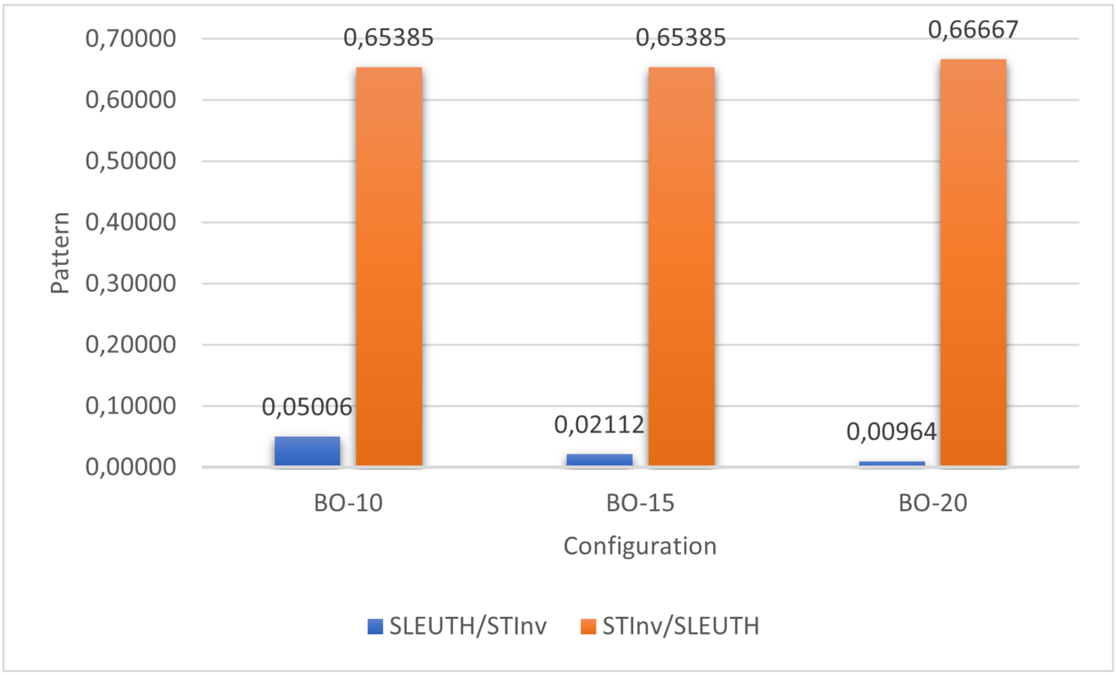


Figure 4.25: Comparison of Results between our work and STInv for the City of Boston with $offset = 0$

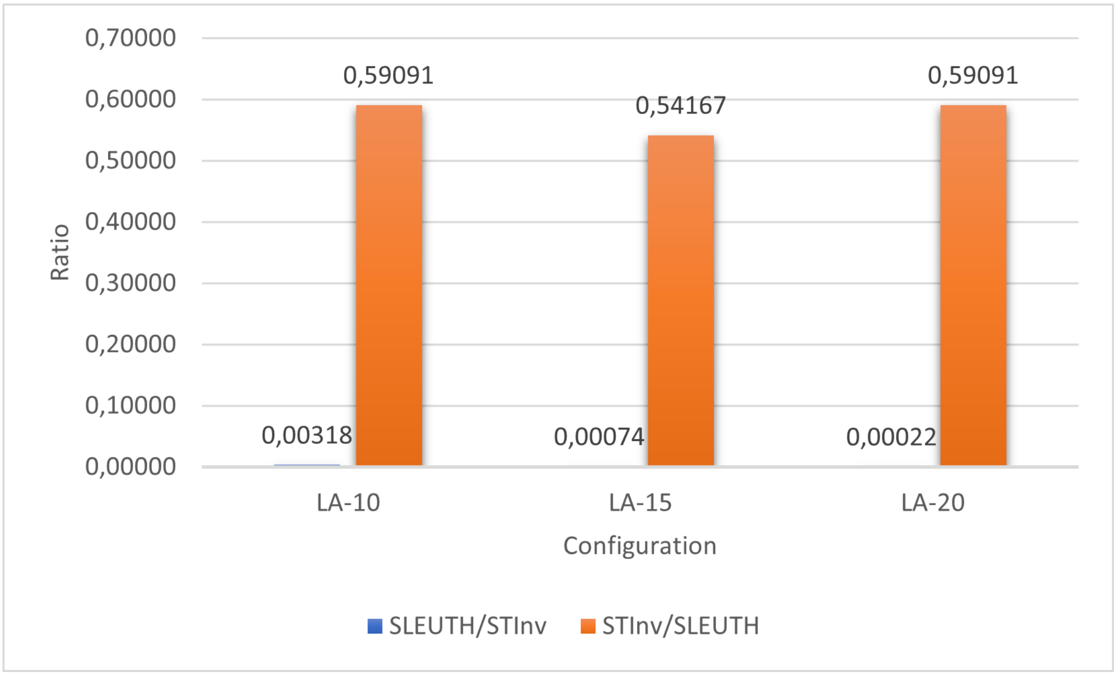


Figure 4.26: Comparison of Results between our work and STInv for the City of Los Angeles with $offset = 0$

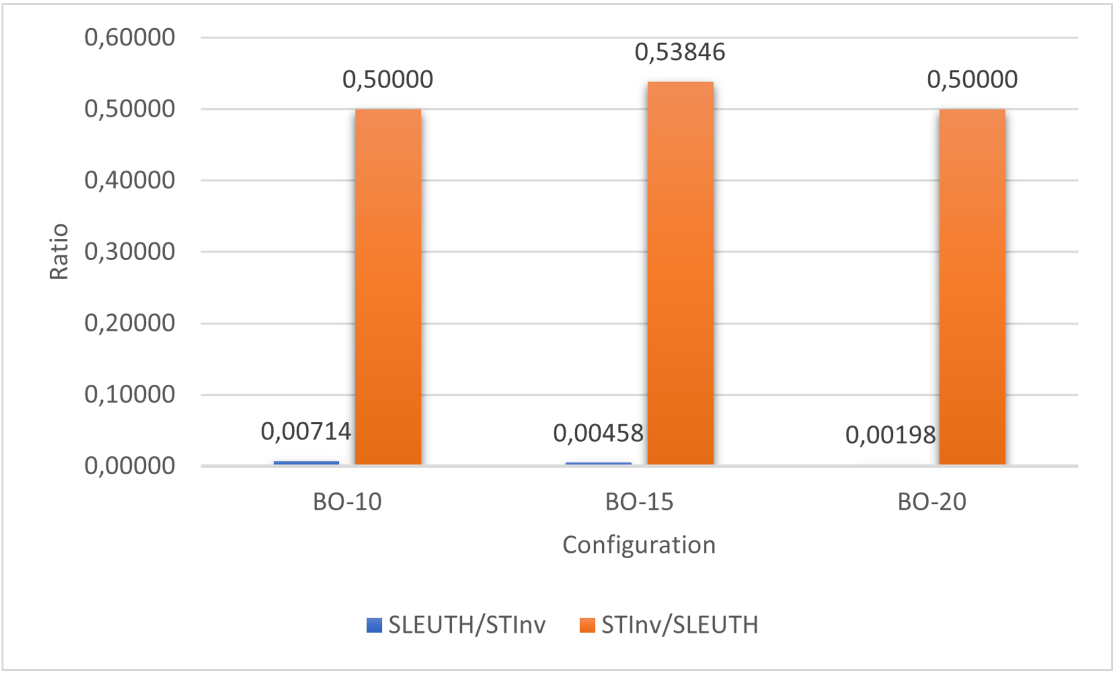


Figure 4.27: Comparison of Results between our work and STInv for the City of Boston with $offset = 1$

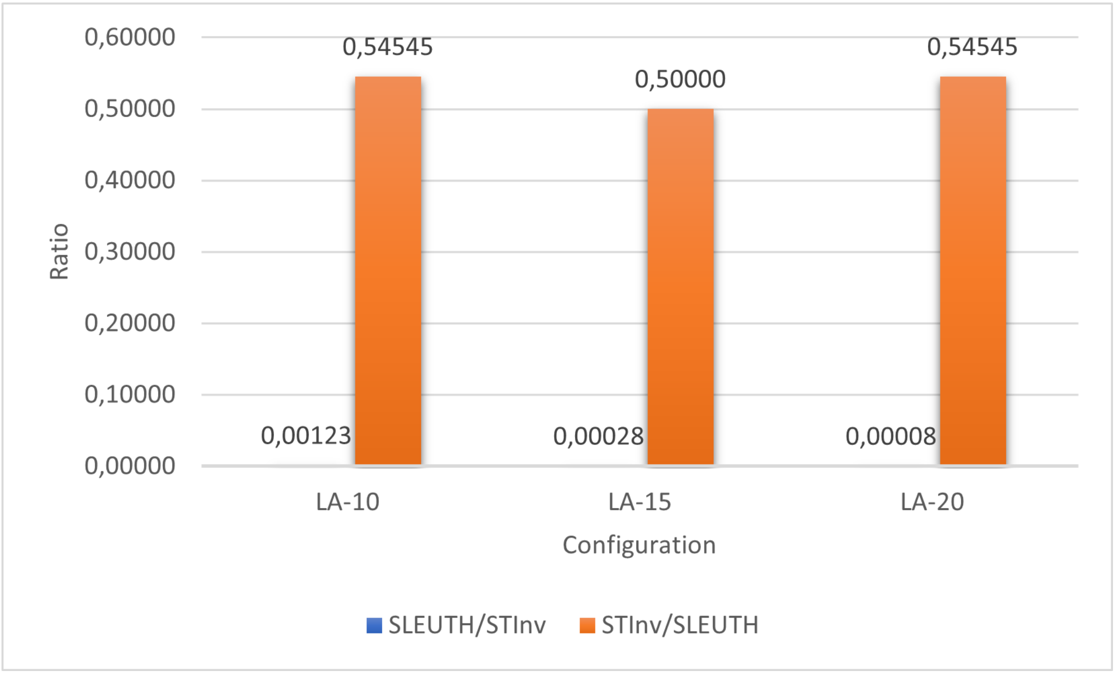


Figure 4.28: Comparison of Results between our work and STInv for the City of Los Angeles with $offset = 1$

Chapter 5

Conclusions and future developments

The new framework developed in this study represents a significant improvement over the previous version developed by Sobhan Moosavi's team. One of the main differences between the two frameworks is the increased speed and efficiency of the new version, which is achieved through the use of PySpark for the implementation and the optimization of certain parts of the code.

Looking to the future, there may be opportunities to further improve and optimize the new framework. One potential area of development could be to parallelize the implementation of the SLEUTH algorithm, which is currently the only part of the code that is not optimized for parallel processing. This could help to further increase the speed and efficiency of the framework and make it even more effective for analyzing large datasets. Overall, the new framework represents an important step forward in the development of tools for identifying patterns of causation between geospatial entities and is likely to be an important tool for a wide range of applications.

Bibliography

[1] Sobhan Moosavi, Mohammad Hossein Samavatian, Arnab Nandi, Srinivasan Parthasarathy, and Rajiv Ramnath. «Short and Long-term Pattern Discovery Over Large-Scale Geo-Spatiotemporal Data». In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery &amp Data Mining.* ACM, July 2019. DOI: 10.1145/3292500.3330755. URL: https://doi.org/10.1145%5C%2F3292500.3330755 (cit. on pp. 1, 5, 9, 27, 28).

[2] Peizhong Yang, Lizhen Wang, Xiaoxuan Wang, Lihua Zhou, and Hongmei Chen. «Parallel Co-Location Pattern Mining Based on Neighbor-Dependency Partition and Column Calculation». In: *Proceedings of the 29th International Conference on Advances in Geographic Information Systems.* SIGSPATIAL '21. Beijing, China: Association for Computing Machinery, 2021, pp. 365–374. ISBN: 9781450386647. DOI: 10.1145/3474717.3483984. URL: https://doi.org/10.1145/3474717.3483984 (cit. on p. 3).

[3] Shashi Shekhar and Yan Huang. «Discovering Spatial Co-location Patterns: A Summary of Results». In: *Advances in Spatial and Temporal Databases.* Ed. by Christian S. Jensen, Markus Schneider, Bernhard Seeger, and Vassilis J. Tsotras. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 236–256. ISBN: 978-3-540-47724-2 (cit. on p. 4).

[4] Lu-An Tang, Yu Zheng, Jing Yuan, Jiawei Han, Alice Leung, Chih-Chieh Hung, and Wen-Chih Peng. «On Discovery of Traveling Companions from Streaming Trajectories». In: *2012 IEEE 28th International Conference on Data Engineering.* 2012, pp. 186–197. DOI: 10.1109/ICDE.2012.33 (cit. on p. 4).

[5] Luca Colomba, Luca Cagliero, and Paolo Garza. «Mining Spatiotemporally Invariant Patterns». In: SIGSPATIAL '22. Seattle, Washington: Association for Computing Machinery, 2022. ISBN: 9781450395298. DOI: 10.1145/3557915.3560998. URL: https://doi.org/10.1145/3557915.3560998 (cit. on pp. 5, 44).

[6] Amazon. *Apache Spark Architecture.* https://d1.awsstatic.com/Data%20Lake/what-is-apache-spark.b3a3099296936df595d9a7d3610f1a77ff0749df.PNG (cit. on p. 8).

[7] Mohammed Zaki. «Efficiently mining frequent embedded unordered trees». In: *Fundamenta Informaticae* 65 (June 2005), pp. 1–20 (cit. on p. 9).

[8] Mohammed J. Zaki. «Efficiently Mining Frequent Trees in a Forest». In: *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '02. Edmonton, Alberta, Canada: Association for Computing Machinery, 2002, pp. 71–80. ISBN: 158113567X. DOI: `10.1145/775047.775058`. URL: `https://doi.org/10.1145/775047.775058` (cit. on p. 9).