ASSOCIATION PATTERN MINING IN SPATIO-TEMPORAL DATABASES

WANG JUNMEI

(M.Eng. XI'AN JIAOTONG UNIVERSITY, CHINA)

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgements

I wish to express my deep gratitude to my supervisors Dr. Wynne Hsu and Dr. Lee Mong Li. I thank them for their continuous encouragement, confidence and support, for sharing with me their knowledge and experience, and for their insightful comments and advice.

I wish to thank Dr. Tay Seng Chuan for his support and providing the dataset for our experiments. My gratitude and appreciation also go to Dr. Tan Chew Lim and Dr. Huang Zhiyong for serving as examiners of my thesis. I also wish to thank Ms Alexia Leong for proofreading of my thesis.

I want to thank my parents and my husband, Wang Jianjun for their continuous moral support and encouragement. I am also very grateful to my brothers and sisters for their continuous encouragement and concern. I hope I will make them proud of my achievements as I am proud of them. Their love accompanies me wherever I go.

Last but not least, I would also like to thank many people in our faculty for always being helpful over the years. I thank my friends at the National University of Singapore for their help.

Contents

Acknowledgements **i**

Contents **ii**

Abstract **vi**

List of Tables **viii**

List of Figures **ix**

List of Publications **xiv**

1 Introduction **1**

1.1 Motivation and Contribution 2

1.2 Organization of the Thesis 8

2 Related Work **9**

2.1 Mining Association Patterns in Spatial Databases 10

2.1.1 Mining of Spatial Association Rules 11

2.1.2 Mining of Spatial Collocation Patterns 13
2.2 Mining Sequence Patterns 14
2.3 Mining Spatio-temporal Databases 17
2.3.1 Mining Evolution Patterns 18
2.3.2 Mining Frequent Movements of Objects 19

3 Mining Topological Patterns 21

3.1 Problem Statement 23
3.1.1 Topological Patterns 24
3.1.2 Geographical Features 27
3.2 Pattern Growth Approach 29
3.3 Algorithm TopologyMiner 31
3.3.1 Summary structure 31
3.3.2 Mining Topological Patterns 35
3.3.3 Mining Geographical Features 41
3.4 TopologyMiner Algorithm 42
3.5 Experimental Study 46
3.5.1 Synthetic Data Generation 46
3.5.2 Effect of Prevalence Threshold 50
3.5.3 Effect of Database Size 50
3.5.4 Effect of Distance Thresholds 52
3.5.5 Effect of Number of Features 52
3.5.6 Comparative Study on Finding Interesting Geographical Features 55

3.5.7 Comparative Study on Finding Clique Patterns 57

3.6 Summary 60

4 Mining Spatial Sequence Patterns 61

4.1 Framework of Spatio-temporal Databases 62

4.1.1 Interesting Patterns in Spatio-temporal Databases 65

4.2 FlowMiner: Finding Flow Patterns in Spatio-temporal Databases 66

4.2.1 Problem Statement 66

4.2.2 Candidates Generation 68

4.2.3 Support Counting 78

4.2.4 Pruning Techniques 80

4.2.5 FlowMiner Algorithm 82

4.2.6 Performance Study 85

4.3 GenSTMiner: Mining Generalized Spatio-temporal Patterns 98

4.3.1 Problem Statement 99

4.3.2 Projection-based Sequential Pattern Mining 102

4.3.3 GenSTMiner Algorithm 103

4.3.4 Performance Evaluation 113

4.4 Summary 120

5 Mining Arbitrary Spatio-temporal Patterns 122

5.1 Preliminary Concepts 126

5.2 Partition-based Graph Mining 128

5.2.1 Dividing Graph Database into Units 129
5.2.2 Mining Frequent Subgraphs in Units 135
5.2.3 Combining Frequent Subgraphs 137
5.2.4 Framework of PartMiner 143
5.2.5 Handle Updates Using PartMiner 146
5.3 Experimental Study 151
5.3.1 Performance Study on Static Datasets 152
5.3.2 Performance Study on Dynamic Datasets 159
5.4 Experiments on Real-life Dataset 164
5.5 Summary 165

6 Conclusions and Future Work 167

6.1 Future Research Directions 169

Bibliography 180

Abstract

With the explosive growth of spatio-temporal applications and spatio-temporal databases, there is increasing need for spatio-temporal data mining. Spatio-temporal data mining has the ability to uncover insightful knowledge in spatio-temporal data that is of increasing relevance in a variety of applications such as homeland security, surveillance, epidemiological and environmental protection. With the knowledge of spatio-temporal data, decision makers can understand the underlying process that controls changes to perform accurate prediction. To date, a limited number of works have been proposed for mining patterns in spatio-temporal databases. Moreover, most of them are simply adaptations of existing techniques for either spatial or temporal data mining. Yet, in spatio-temporal databases, each object is related to other objects in complex interactions, which cannot be discovered by looking at spatial information or temporal information independently. Methods for the extraction of complex relationships in spatio-temporal data are clearly required.

This thesis studies the techniques for discovering association patterns in spatio-temporal databases by combining spatial and temporal information together. Specifically, we first investigate the problem of mining topological patterns by imposing tem-

poral constraints into spatial collocation pattern mining. We design and develop an efficient algorithm to find topological patterns. Next, we study the problem of mining spatial sequence patterns by incorporating spatial information into sequence mining. We introduce two new classes of spatial sequence patterns, called flow patterns and generalized spatio-temporal patterns, and develop two algorithms to find them. A comprehensive performance study shows that the proposed algorithms are efficient and scalable in finding spatial sequence patterns. Finally, we study the problem of mining arbitrary spatio-temporal patterns by modeling spatio-temporal data as graphs. We introduce a partition-based approach to graph mining. Our extensive experimental results indicate that the proposed algorithm is effective and scalable in finding frequent subgraphs in the databases, and outperforms existing algorithms in the presence of updates.

List of Tables

3.1 Data generation parameters 48

3.2 Observed common habits 56

3.3 Interesting patterns found 57

4.1 Parameters 85

4.2 Real-life dataset characteristics 86

4.3 Comparison of candidates generated 97

5.1 Meaning of symbols 146

5.2 Parameters of synthetic data generator 151

List of Figures

1.1 Example of a spatio-temporal database 3

1.2 Graph representation of spatio-temporal patterns 7

2.1 Summary of techniques for mining spatial association patterns 11

2.2 Summary of techniques for mining sequence patterns 15

2.3 Summary of the techniques for mining patterns in spatio-temporal databases 18

3.1 Example of two topological patterns 25

3.2 Relationship of distance to geographical feature 28

3.3 Projection sequential pattern mining 30

3.4 Example of a spatio-temporal database 33

3.5 Example of a summary-structure 34

3.6 The projected database of f_1 37

3.7 The projected databases of $\langle f_1, f_2 \rangle$ 38

3.8 Outline of the TopologyMiner algorithm 43

3.9 Procedure MiningPDB 44

3.10 Runtime vs. prevalence threshold 49

3.11 Runtime vs. number of points N 51

3.12 Runtime vs. distance thresholds 53

3.13 Runtime vs. number of features 54

3.14 Runtime vs. the distance relation (clique patterns) 58

3.15 Runtime vs. number of points (clique patterns) 59

4.1 Example of a spatio-temporal database 63

4.2 Example of flow patterns 67

4.3 Candidates validation with length-2 sequences and neighborhood constraints 69

4.4 Summary tree for the dataset in Figure 4.1 71

4.5 Temporal relationships of length-2 sequences 74

4.6 Example of insert positions 75

4.7 Procedure of candidate generation 77

4.8 Hash tree for varying flow patterns length 79

4.9 Framework of the FlowMiner algorithm 83

4.10 Optimized algorithm 84

4.11 Varying parameter C (synthetic dataset) 87

4.12 Varying parameter T (synthetic dataset) 87

4.13 Varying parameter R (synthetic dataset) 88

4.14 Varying parameter D (synthetic dataset) 88

4.15 Runtime vs. parameter minsup (real-life dataset) 90

4.16 Runtime vs. spatial neighbor relation R (real-life dataset) 91

4.17 Scalability (real-life dataset) 91

4.18 Flow patterns [Trend 1: from West to East in March and April] 93

4.19 Flow patterns [Trend 2: from South to Northwest in April and May] . . 94

4.20 Effect of optimizations 95

4.21 Comparative study (sequence patterns) 96

4.22 Example spatio-temporal database (W = 15days, $R = 1$) 99

4.23 Projected database of event a 105

4.24 Generalized projected database of event a 106

4.25 The GenSTMiner algorithm 109

4.26 a-conditional projected database 111

4.27 Example of pseudo-projection 113

4.28 Runtime vs. parameter R 115

4.29 Runtime vs. parameter t-minsup 116

4.30 Runtime vs. parameter s-minsup 117

4.31 Scalability 117

4.32 Comparison of flow patterns and generalized spatio-temporal patterns . 119

5.1 Framework for mining arbitrary spatio-temporal patterns 123

5.2 Example of the DFS tree and DFS code 128

5.3 Overview of partition-based graph mining 129

5.4 Example of graph bi-partitioning 130

5.5 Example of partitioning criteria 131

5.6 Algorithm to partition a graph 133

5.7 Dividing a graph database into units 134

5.8 Partitioning the graph database into k units 135

5.9 Outline of ADIMINE algorithm 136

5.10 Example of recovering the original database from the units 137

5.11 Example of the merge-join operation 140

5.12 Base case 141

5.13 Induction step 141

5.14 Outline of the PartMiner algorithm 144

5.15 Outline of the MergeJoin procedure 145

5.16 Outline of the IncPartMiner algorithm 149

5.17 Outline of the IncMergeJoin procedure 150

5.18 Example of transformed graphs 152

5.19 Effect of partitioning criteria 154

5.20 Runtime vs. parameter minsup 154

5.21 Runtime vs. parameter k 155

5.22 Varying parameter T 157

5.23 Varying parameter I 157

5.24 Varying parameter D 158

5.25 Effect of partitioning criteria 160

5.26 Runtime vs. parameter minsup 160

5.27 Runtime vs. parameter k 162

5.28 Updating the node/edge labels 163

5.29 Adding new edges between two vertices 163

5.30 Adding new vertex with an edge to existing vertices 164

5.31 Interesting patterns found in real-life dataset 165

List of Publications

1. Junmei Wang, Wynne Hsu, and Mong Li Lee. *Discovering Geographical Features for Location-Based Services*, in 9th International Conference on Database Systems for Advanced Applications (DASFAA), Korea, March 2004.

2. Junmei Wang, Wynne Hsu, Mong Li Lee, and Jason Wang. *FlowMiner: Finding Flow Patterns in Spatio-temporal Databases,* in 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Florida, November, 2004

3. Junmei Wang , Wynne Hsu, and Mong Li Lee. *Mining in Spatio-Temporal Databases, Book Chapter in Spatial Databases: Technologies, Techniques and Trends*, Yannis Manalopoulos, Apostolos N. Papadopoulos, Michael Gr. Vassilakopoulos (Eds.), ISBN: 159140388-X, Idea Group Publishing, 2005

4. Junmei Wang, Wynne Hsu, and Mong Li Lee. *Mining Generalized Spatio-Temporal Patterns*, in 10th International Conference on Database Systems for Advanced Applications (DASFAA), Beijing China, April 18-20, 2005.

5. Junmei Wang, Wynne Hsu, and Mong Li Lee. *A framework for mining topological patterns in spatio-temporal databases*, in 2005 ACM CIKM International

Conference on Information and Knowledge Management, Bremen, Germany, October 31 - November 5, 2005. ACM 2005.

6. Junmei Wang, Wynne Hsu, and Mong Li Lee. *A Partition-Based Approach to Graph Mining*, accepted in the 22nd International Conference on Data Engineering April 3-7, Atlanta, GA, 2006 .

Chapter 1

Introduction

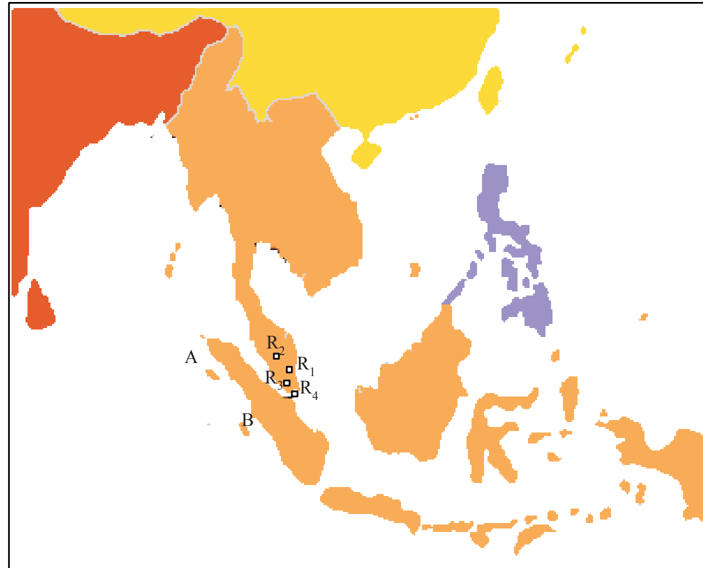
Spatio-temporal databases have been an active area of research since the early 1990s. This surge in interest has resulted in recent advances such as modeling, indexing, and querying of moving objects and spatio-temporal data [GBE$^+$00, SJLL00, TPS02, TTPL04, CN04, SPTL04]. These advances suggest that database technologies will play a central role in the development and deployment of spatio-temporal applications. Accordingly, advanced data mining capabilities should become increasingly important to spatio-temporal databases. Spatio-temporal data mining has the ability to disclose insightful knowledge embedded in spatio-temporal phenomena and enable decision makers to understand the underlying process that controls changes and patterns of changes. Compared to the conventional data mining areas, e.g., spatial data mining and temporal data mining, spatio-temporal data mining is more complicated and presents a number of challenges due to the complexity of geographical domains, the mapping of data in spatial and temporal frameworks, and spatial and temporal autocorrelation [MH01]. In

spatio-temporal databases, each object is related to other objects in complex interactions which are captured in the form of past, present and future states in the modeled environment. Data mining in spatio-temporal databases must consider the multi-states of spatio-temporal data. It must integrate spatial information and temporal information together to find meaningful spatio-temporal patterns.

1.1 Motivation and Contribution

In the last decade, we have witnessed increased attention on spatial data mining and temporal data mining. Many algorithms have been proposed to find either *spatial patterns* [HKS97, SH01, Mor01, ZMCS04] or *time varying patterns* [AS96, PHMAP01, WH04, Zak98]. Both spatial patterns and time varying patterns can reveal interesting information from data, but they either focus on the spatial dimension or on the temporal dimension. Very few of them handle both.

As spatio-temporal data becomes more prevalent, researchers [SNMM95, MSM95, TSK01, STK^{+}01, TG01, PC03, MCK^{+}04] have re-focused their attention to the discovery of interesting patterns in spatio-temporal databases. Initially, most of the work in spatio-temporal data mining is simply adaptations of techniques from the spatial or temporal data mining field for use on spatio-temporal data. However, spatio-temporal data contains complex relationships that cannot be discovered simply by looking at the spatial dimension or the temporal dimension independently. We illustrate this with a simple example.



(a) Space-view

ID	Time	Location	Event
101	July 26, 1965	R_2	forest fire
102	July 28, 1965	R_1	haze
103	July 30, 1965	R_3	atmospheric pressure $\downarrow$
104	August 2, 1965	R_4	rainfall
...	...	...	...
19998	February 26, 2005	A	earthquake
19999	February 26, 2005	A	tsunami
20000	March 28, 2005	B	earthquake

(b) Database

Figure 1.1: Example of a spatio-temporal database

Assume that we have a spatio-temporal database of the weather system in Southeast Asia. The information stored in the database includes events, such as $atmospheric$ $pressure$, $forest\ fire$, $haze$, $rainfall$, $earthquake$, $tsunami$, etc., locations of the events, and time of the events. With the spatio-temporal databases, we want to study the interaction relationships of these events in different areas in Southeast Asia. Figure 1.1 shows an example of the spatio-temporal database.

Using the spatial data mining techniques, we discover the following spatial association patterns:

S1: If an $earthquake$ occurs in the place close to sea, there is high probability of the occurrence of $tsunami$.

S2: There is a higher confidence of $earthquakes$ in a region if there is $high\ atmospheric$ $pressure$ in the nearby regions.

S3: There is high probability of $haze$ in region R_1 if there is $forest\ fire$ occurring in the nearby region R_2.

S4: If there is a $drop$ in $atmospheric\ pressure$ in region R_3, $rainfall$ will always occur in the nearby region R_4.

S5: There is high probability of a $drop$ in $atmospheric\ pressure$ in region R_3 if there is $haze$ in the nearby region R_2.

However, these spatial rules do not tell us us any information about the temporal relationships of the events.

To discover the temporal relationships among these events, we have to use temporal data mining techniques. Examples of temporal rules we have found are listed below:

T1: $Earthquakes$ always happen during or soon after periods of $high\ atmospheric\ pressure$.

T2: If there is a $forest\ fire$, soon after there will be $haze$, then a $drop$ in $atmospheric\ pressure$, then $rainfall$.

Once again, these temporal rules seem to have some information missing. Ideally, we should link the location and precedence relationships together in our spatio-temporal rules. For example:

ST1: There is a higher incidence of $earthquakes$ in a region during or soon after high $atmospheric\ pressure$ in the nearby region.

ST2: $Forest\ fire$ always occurs at region R_1 prior to the occurrence of $haze$ in the nearby region R_2, then a drop in $atmospheric\ pressure$ at region R_3, and then $rainfall$ at region R_4.

ST3: From March to April, if there is a $forest\ fire$ in a region in South Asia, $haze$ and $rainfall$ will subsequently occur in its Southeastern neighbors.

Clearly, patterns ST1-ST3 are much more informative than spatial patterns and temporal patterns. Moreover, these spatio-temporal patterns not only link events in different locations, but also establish the sequence of changes of events in these locations. Hence,

they are more useful and helpful for decision makers in understanding the evolving process and making accurate predictions.

We investigate the discovery of interesting spatio-temporal patterns from two aspects:

- First, we impose temporal constraints on the mining of spatial collocation patterns to discover *topological patterns* such as: "*There is higher incidence of earthquakes in a region during or soon after periods of high atmospheric pressure in the nearby regions.*" Topological patterns aim to discover the intra-relationships of events in a time period. We design an efficient algorithm to find topological patterns in a depth-first manner.

- Second, we search for spatial sequence patterns, such as: "*Forest fire always occurs at region R_1 prior to the occurrence of haze in the nearby region R_2.*" and "*A drop in atmospheric pressure at a region always precedes rainfall in the nearby regions.*" by incorporating spatial information into the process for mining sequence patterns. Spatial sequence patterns aim to find the inter-relationships of events in different time windows. In the thesis, we introduce two new classes of spatial sequence patterns, called *flow patterns* and *generalized spatio-temporal patterns*. These two classes of spatial sequence patterns are useful to the understanding of many real-life applications. Algorithms designed to discover these two classes of spatial sequence patterns have shown to be efficient and scalable.

Some complex relationships among spatio-temporal data cannot be captured with

these two simple approaches. To further discover complex relationships in spatio-temporal data, we model data as graphs. Each vertex in a graph represents a variable labeled by an attribute or event, and each edge represents the spatial relationship, the temporal relationship, or both. With this, we transform the problem of mining *arbitrary* spatio-temporal patterns into the problem of finding frequent subgraphs. Figure 1.2 shows the possible graph structures representing the spatio-temporal patterns ST1, ST2, and ST3.

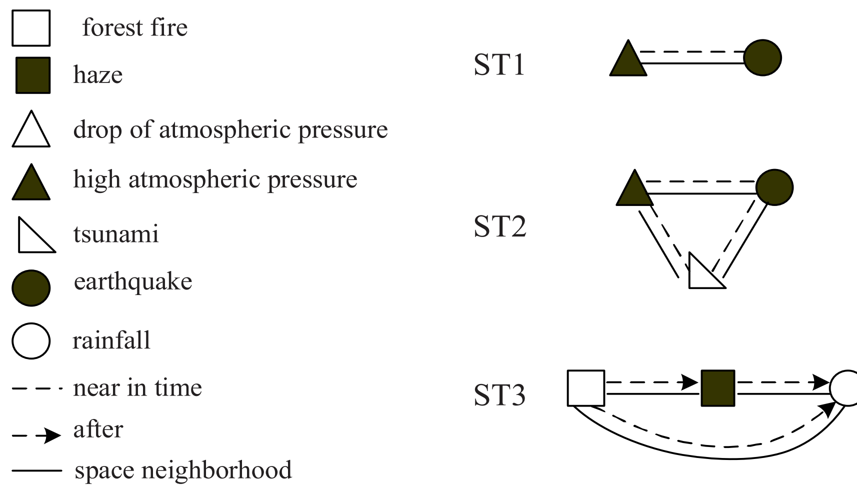


Figure 1.2: Graph representation of spatio-temporal patterns

Unfortunately, extending existing algorithms to find these spatio-temporal patterns is not feasible due to the large search space of both the spatial and temporal dimensions.

To find these patterns, we instead design and develop a partition-based graph mining algorithm. These algorithms work by discovering frequent subgraphs in the graph database. The proposed algorithm is effective and scalable in finding frequent subgraphs, and outperforms existing algorithms in the presence of updates.

1.2 Organization of the Thesis

This thesis is organized as follows. Chapter 2 reviews the related work on mining interesting association patterns in spatial, temporal and spatio-temporal databases. In Chapter 3, we study the problem of finding topological patterns in spatio-temporal databases and illustrate the algorithm in detail. Next, we introduce two new classes of spatial sequence patterns and illustrate the algorithms designed for mining these two classes of spatial sequence patterns in detail in Chapter 4. The work for mining arbitrary spatio-temporal association patterns is described in Chapter 5. We conclude the thesis in Chapter 6.

Chapter 2

Related Work

Spatial data mining is the process of discovering relationships between spatial data and nonspatial data by using spatial proximity relationships. Spatial data is self-autocorrelated and exhibits a unique property known as Tobler's first law of geography [Tob79]: *"Everything is related to everything else but nearby things are more related than distant things."* Mining patterns from spatial datasets is more difficult than extracting the corresponding patterns from traditional numeric and categorical data due to the complexity of spatial data. Spatial data mining covers a wide spectrum, including spatial clustering [GRS98, NH94, SEKX98], spatial characterization and trend detection [EFKS98], spatial classification [KHS98], etc. Among them, the problem of mining interesting association patterns in spatial databases is most related to our work.

Similar to spatial data mining, temporal data mining has also received much attention [RS02]. Two types of temporal data are dominant in the development of temporal data mining. They are *time-series data* and *sequence data*. Time-series data

is a sequence of real numbers that vary with time, e.g., stock prices, exchange rates, biomedical measurements data, etc. Sequence data is a list of *transactions*, and a transaction time is associated with each transaction, e.g., web page traversal sequences. Mining patterns from temporal databases is complex due to the existence of time. Time implies an ordering, and this ordering affects the statistical properties of the data and the semantics of the rules being extracted from them. Temporal data mining also covers a wide spectrum, including time series similarity [Keo01], sequence mining [AS96, Zak98, PHMAP01, AGYF02], temporal classification [AC01], clustering [OSC00, WWYY02] etc., where the problem of mining sequence patterns is considered to be more related to our work.

In this chapter, we review the work for mining spatial association patterns in Section 2.1 and the techniques for mining sequence patterns in Section 2.2. Finally, we describe the early attempts on spatio-temporal data mining in Section 2.3.

2.1 Mining Association Patterns in Spatial Databases

In the context of spatial data mining, spatial association patterns reflect the relationships of spatial/spatial data or spatial/nonspatial data. To date, two formats of association rules in spatial databases have been introduced:

1. *Spatial association rules* are the natural extension of classic association rules in spatial databases. They incorporate spatial predicates into either the antecedent or the consequent. For example, a spatial association rule "*80% of schools are*

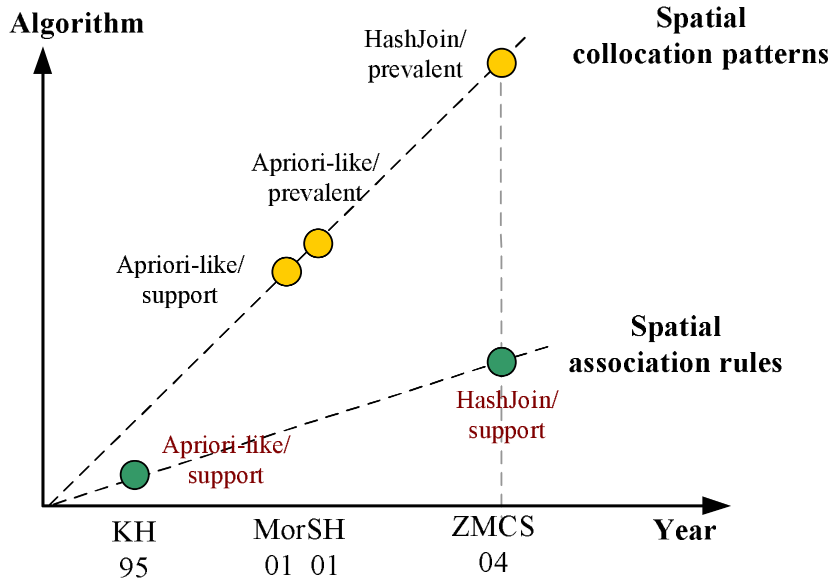


Figure 2.1: Summary of techniques for mining spatial association patterns

close to parks";

2. *Spatial collocation patterns* seek to find the set of spatial features with instances that are located in the same neighborhood. For example, a collocation rule can be described as "*76% of the occurrences of smoke aerosols implies the occurrence of rainfall in a nearby region*".

Here, we briefly review the techniques to extract these spatial association patterns in spatial databases. Figure 2.1 summarizes the techniques for mining association patterns in spatial databases.

2.1.1 Mining of Spatial Association Rules

The problem of mining spatial association rules based on spatial relationships (e.g., adjacency, proximity) of events or objects is first discussed in [KH95], where spa-

tial data are converted to transactions according to a centric reference feature model. Consider a spatial database D, which consists of n number of spatial sub-datasets $D = \{R_1, R_2, \ldots, R_n\}$, such that each R_i contains all objects that have a particular nonspatial feature f_i. Given a feature f_i, we define a transactional database as follows. For each object o_i in R_i, a spatial query is issued to derive a set of features $I = \{f_j : fj \neq fi \wedge \exists o_j \in R_j, dist(o_i, o_j) \leq \epsilon\}$. The collection of all feature sets I for each object in R_i defines a transactional table T_i. T_i is then mined using some itemsets mining method [AS94, HP00]. The frequent feature sets I in this table, according to a minimum support value, can be used to define rules of the form:

$$o.label = f_i \Rightarrow o \text{ close to some } o_j \in R_j, \forall f_j \in I$$

The support of a feature set I defines the confidence of the corresponding rule.

The major limitation of the spatial association rule is that it depends on the concept of explicit transactions in databases. However, due to the continuity of the underlying space, this may not be possible or appropriate in spatial databases. Moreover, many duplicate counts of association rules may result if we define transactions around locations of instances of features.

Further, it is difficult to extend the algorithm for mining spatial association rules to find association rules in spatio-temporal databases. In spatio-temporal databases, association rules should satisfy both spatial proximity relationships and temporal proximity relationships. Since spatio-temporal databases are 3D, instead of 2D, the computational cost of processing candidate patterns and computing the interestingness of these patterns is much higher than that of spatial databases. As a result, existing techniques are

difficult and not scalable for use to find association rules in spatio-temporal databases.

2.1.2 Mining of Spatial Collocation Patterns

Recently, research on spatial association pattern mining has shifted towards mining collocation patterns that are the set of spatial features with instances located in the same neighborhood.

[SH01] first defines the problem for mining spatial collocation patterns using neighborhoods in place of transactions. The work defines a new spatial measure of *conditional probability* as well as a monotonic measure of *prevalence* to allow iterative pruning. Based on these concepts, an Apriori-like approach called Co-location Miner is developed to find all the frequent collocation patterns. Co-location Miner initially performs a spatial join to retrieve object pairs which are close to each other, and then it uses the Apriori-based candidate generation algorithm to generate the candidates of length $(k+1)$-pattern from k-patterns and validate the candidates by joining the instances of the k-patterns which share the first $k-1$ feature instances. They further study the problem of mining confident co-location rules without a support threshold in their continuous work [HXSP03]. Similarly, [Mor01] studies the same problem to find sets of services located close to each other. This work also presents an Apriori-like algorithm. Different from Co-location Miner, it uses a Voronoi diagram and a quaternary tree to improve running time. However, the method can only be used to do approximation.

[ZMCS04] introduces a method to discover maximal collocation patterns by combining the discovery of spatial neighborhoods with the mining process. Specifically, it

extends a hash-based spatial join algorithm to operate on multiple feature sets in order to identify such neighborhoods. The algorithm divides the map and partitions the feature sets using a regular grid. While identifying object neighborhoods in each partition, at the same time, the algorithm attempts to discover prevalent and confident patterns by counting their occurrences at production time. However, the approach has to enumerate all combinations of the spatial features, and the performance decreases dramatically as the number of spatial features increases.

From the above, we note that most of the methods [KH95, SH01, Mor01] proposed in spatial databases follow the candidates-maintenance-and-test methodology. Their performances suffer from maintaining many candidates and the need for multiple database scans. Hence, it is difficult to extend them to the discovery of spatio-temporal patterns due to the high computational cost of candidate patterns in higher dimension space.

2.2 Mining Sequence Patterns

The problem of discovering sequence patterns is to discover and infer relationships of contextual and temporal proximity in the data. Since it was first introduced in [AS95], sequence mining has become an essential data mining task with broad applications, such as in market and customer analysis, etc. Efficient mining methods have been studied extensively, including general sequence pattern mining [AS96, Zak98, PHMAP01, AGYF02], constraint-based sequence pattern mining [GRS99, PHW02],

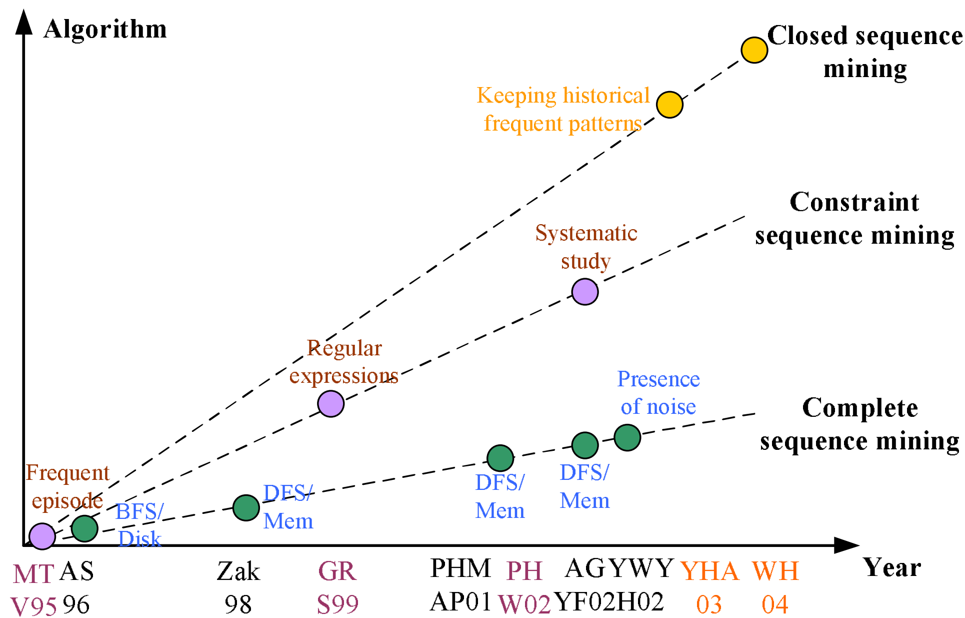


Figure 2.2: Summary of techniques for mining sequence patterns

frequent episode mining [MTV95], long sequence pattern mining in noisy environment [YWYH02], and closed sequence pattern mining [WH04]. Figure 2.2 shows the techniques for mining sequence patterns.

First, we review the methods proposed for mining the complete set of frequent sequences. [AS96] introduces a breadth-first disk-based algorithm, which follows the *candidate-maintenance-and-test* paradigm to find frequent sequence patterns. Subsequently, [Zak98], [PHMAP01] and [AGYF02] investigate depth-first memory-based methods to mine sequence patterns. The depth-first approaches generally perform better than the breadth-first approaches if the data resides in memory. Recently, [YWYH02] has studied the problem for mining frequent sequences in the presence of noise with the help of the compatibility matrix, which provides a probabilistic connection from the observation to the underlying true value. However, the limitation of these methods is

that their performances degrade dramatically when the length of the sequences is long and the minimum support threshold is low. This is not surprising since a long sequence contains a combinatorial number of frequent subsequences. Such mining generates an explosive number of subsequences for long sequences.

Currently, an interesting solution, called *mining closed sequence patterns*, is proposed to overcome this difficulty. The problem of mining closed sequences is to find the set of sequences such that there is no sequence which has a super-sequence with the same support. [YHA03] is the first to present an algorithm CloSpan to mine closed sequence patterns. It introduces the concept of equivalence of projected databases, which unifies two pruning optimizations: *Backward Sub-pattern* and *Backward Super-pattern* in a single step. However, CloSpan still follows the candidate-maintenance-and-test paradigm and has to maintain the set of already mined closed sequence candidates. To overcome this problem, [WH04] introduces the BI-directional extension checking scheme, a new closure checking and ScanSkip optimization technique. Based on the technique, the authors present a solution BIDE, which can find the set of closed sequences without keeping track of any single historical frequent closed sequences for a new pattern's closure checking.

At the same time, many researchers [MTV95, GRS99, PHW02] have shifted their attention towards mining sequences by incorporating constraints to reduce search space. [MTV95] studies the problem of finding a frequent episode in a sequence of events by posing constraints on the event in the form of acyclic graphs. [GRS99] proposes regular expressions as constraints for sequence pattern mining and develops a family of SPIRIT

algorithms while members in the family achieve various degrees of constraint enforcement. Following that, [PHW02] conducts a systematic study on constraint sequence pattern mining and classifies various kinds of constraints into two categories according to their application semantics and roles in sequence pattern mining.

2.3 Mining Spatio-temporal Databases

As a significant subset of data mining, spatio-temporal data mining is an emerging research area dedicated to the development and application of novel computational techniques for the analysis of very large spatio-temporal databases. Knowledge of spatio-temporal data is of increasing relevance in a variety of applications, such as homeland security, global environment change, etc. However, mining in spatio-temporal databases is still in its infancy. In this section, we introduce the early attempts at spatio-temporal data mining and review the techniques presented to find various interesting spatio-temporal patterns. Figure 2.3 shows the techniques for mining patterns in spatio-temporal databases.

In short, the previous work on spatio-temporal data mining has mainly focused on two types of patterns:

- *Evolution patterns* of natural phenomena, such as forest coverage, and
- *Frequent movements of objects over time*.

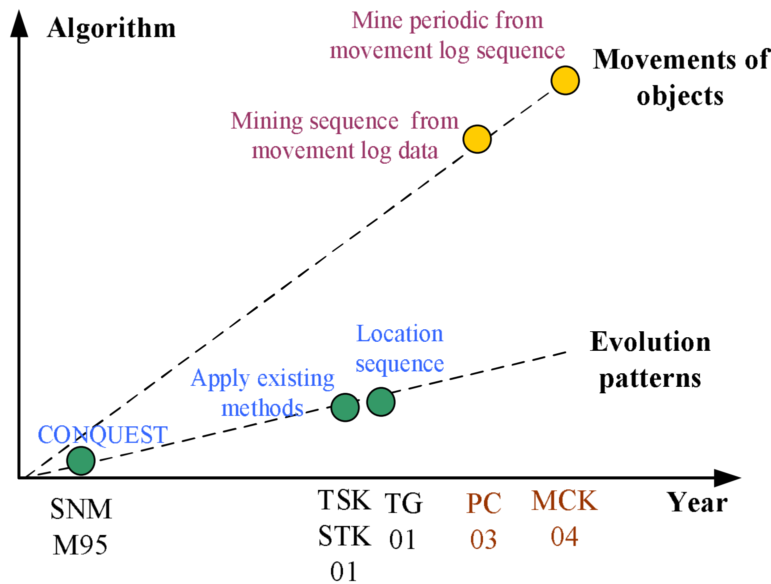


Figure 2.3: Summary of the techniques for mining patterns in spatio-temporal databases

2.3.1 Mining Evolution Patterns

In finding the evolution patterns of natural phenomena, a system called CONQUEST [SNMM95, MSM95] has first been developed to allow some means of accessing and interpreting spatio-temporal data. It provides an environment that enables geophysical scientists to easily formulate queries of spatio-temporal patterns on massive data, such as cyclones, hurricanes and fronts.

Following that, many researchers [TSK01, STK^{+}01] have attempted to mine interesting spatio-temporal patterns in earth science data. They apply existing data mining techniques to find clusters, predictive models and trends, and they state that existing data mining algorithms cannot discover all the interesting patterns in spatio-temporal data [TSK01].

Recently, [TG01] has presented an algorithm to discover frequent sequences in a

depth-first manner over all locations in spatio-temporal databases. This is essentially a sequence mining algorithm whereby each location is treated as a transaction. The algorithm is able to find the common temporal relationships of events in some locations, but not the relationships of events among these locations.

2.3.2 Mining Frequent Movements of Objects

With the development of the global positioning system, moving object databases have received considerable attention. Many research efforts have been focused on finding efficient indexing and querying methods in such databases. However, data mining in moving object databases is still in its infancy.

[PC03] has first proposed a method to optimize mobile systems by finding the frequent motion patterns of objects. It first converts the movement log data into multiple subsequences, each of which represents a maximal moving sequence. With this, finding frequent moving patterns means finding frequently occurring consecutive subsequences among maximal moving sequences. With the mining results of user moving patterns, the authors further develop data allocation schemes that can utilize the knowledge of user moving patterns for proper allocation of both personal and shared data.

[MCK$^+$04] studies the problem of optimizing spatio-temporal queries through the discovery of spatio-temporal periodic patterns, which are the sequence of object locations that reappear in the movement history periodically. This work uses the concept of *dense cluster* to identify a valid region instead of a district in the map from the object trajectory. To find spatio-temporal periodic patterns, the study develops a two-phase

top-down method. First, it uses a hash-based method to retrieve all frequent 1-patterns (i.e., a set of valid clusters), and replaces the trajectories in the database using cluster-ids. Next, it uses the same methodology of maxsubpattern-tree algorithm to discover all the frequent patterns. After getting all the frequent spatio-temporal periodic patterns, it introduces an index structure, called Period Index, to manage the trajectories of objects by exploiting the discovered periodic patterns.

From the above, we note there is a limited number of works on spatio-temporal data mining. Most of them have been regarded as the generalization of pattern mining in temporal databases. In other words, they map data (i.e., locations of objects or the changes of natural phenomena over time) to sequences of values. Then, the algorithms that discover frequent sequences or find frequent subsequences in a long sequence are applied. Although these techniques can discover some interesting patterns in spatio-temporal databases, they cannot be used to discover patterns that disclose the interactions of the events or objects in different locations.

Chapter 3

Mining Topological Patterns in Spatio-temporal Databases

Spatial data mining is an interesting area and has received a lot of attention [NH94, SEKX98, GRS98, KHS98]. Recently, some researchers have shifted their attention towards mining topological patterns, also called collocation patterns. Mining topological patterns is an interesting research problem with broad applications, such as mining topological patterns in an E-commerce company, a location-based service, an ecology dataset and so forth. However, most existing work typically ignores the temporal aspect and focuses on mining spatial patterns, such as: "*There is high probability of the occurrence of earthquakes in a region if there is high atmospheric pressure in the nearby region.*" With the prevalence of spatio-temporal databases, mining of topological patterns with temporal information, such as: "*There is a higher incidence of earthquakes in a region* ***during or soon after*** *a high atmospheric pressure occurs in the nearby re-*

gion." will be much more useful and helpful for data analysts and decision makers in understanding the underlying process that controls the changes.

Existing techniques for finding topological patterns [KH95, Mor01, SH01, HXSP03, ZMCS04] do not scale in spatio-temporal databases since they follow the *candidate-generation-and-test* [AS94] methodology; these methods have to generate and store a potentially large number of candidate patterns. Further, the computational cost of processing candidate patterns and testing the interestingness of the patterns is high. In spatio-temporal databases, topological patterns should satisfy not only spatial proximity relationships but also temporal proximity relationships. Since spatio-temporal databases are three-dimensional, unlike spatial databases which are two-dimensional, the computational cost of processing candidate patterns and computing the interestingness of these patterns are higher than that in spatial databases. We therefore need to explore new methods to solve the problem.

In addition, we note that the spatial features in topological patterns are always prompted by the surrounding geographical objects. If we can identify a set of spatial features that always happen together when certain geographical features are present, then decision makers or area developers can have the means to issue a warning ahead of a disaster or consider the available alternatives..

In this chapter, we study the problem of mining topological patterns by imposing temporal constraints into the process of mining collocation patterns. We first introduce a summary-structure that summarizes the database with the instances' count information of a feature in a region within a time window. Next, based on the summary structure, we

design an algorithm, called TopologyMiner, to find interesting topological patterns in a depth-first manner, and following the pattern growth methodology. Finally, we extend TopologyMiner to find the geographical features of topological patterns. Our extensive experimental study indicates that our proposed algorithm can discover topological patterns efficiently and scalably.

The rest of this chapter is organized as follows. We define preliminary concepts in Section 3.1. Section 3.2 explains the pattern growth method. We illustrate the main steps of the algorithm TopologyMiner in Section 3.3, and give its framework in Section 3.4. The experimental results are reported in Section 3.5. Finally, we summarize the chapter in Section 3.6.

3.1 Problem Statement

Given a spatio-temporal database $\mathcal{D}$, let $F = \{f_1, ..., f_n\}$ be a set of *spatial features* and a lexicographic order $\preceq_f$ be among the spatial features. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m instances in the spatio-temporal database $\mathcal{D}$, where each instance is a vector $\langle$ *instance-id, spatial feature, position, time-stamp* $\rangle$. The spatial feature f, the position (x, y) and the time-stamp ts of an instance i are denoted as $i.f$, $i.x$, $i.y$ and $i.ts$ respectively.

Let R be a neighborhood relation over the positions of the instances in the spatio-temporal database $\mathcal{D}$. Here, we define R as a distance threshold. The distance between two instances i_1 and i_2 is computed as $sdist = \sqrt{(i_1.x - i_2.x)^2 + (i_1.y - i_2.y)^2}$. We

say i_1 and i_2 are located close to each other if and only if $sdist \leq R$. Similarly, let W be a closeness relation over the time-stamps of instances in $\mathcal{D}$. We define W as a time window threshold. The distance between the time-stamps of two instances is computed as $tdist = |i_1.ts - i_2.ts|$. Two instances are said to be near in time if and only if $tdist \leq W$.

To capture the concept of "nearby", a ***neighbor set*** N is defined as a set of instances such that not only all pairwise **positions** of the instances in N are neighbors, but they are also near in **time**.

3.1.1 Topological Patterns

A *topological pattern* S of length k or k-pattern for short, is a set of spatial features, denoted as $S = \{f_1, f_2, \ldots, f_k\}$. All the features in S are ordered according to $\preceq_f$. A valid instance of S is a set of instances $\{i_1, i_2, \ldots, i_k\}$ such that the spatial feature of the instance i_j is f_j, i.e., $i_j.f = f_j$. Note that all the features' instances in S must be ***near in time***. A topological pattern P is called a *sub-pattern* of Q if $\forall f_j \in P$, $f_j \in Q$; and Q is a *super-pattern* of P, denoted as $P \preceq Q$.

[KH95, ZMCS04] define the concept of star-like and clique patterns. We extend these concepts by imposing temporal constraints in them. Based on the star-like patterns and clique patterns, we further introduce another interesting topological patterns, called star-clique patterns.

A topological pattern S is a *star-like pattern* if in a valid instance of S, the instance i_j of the feature f_j is located close to other instances while the instances of other features

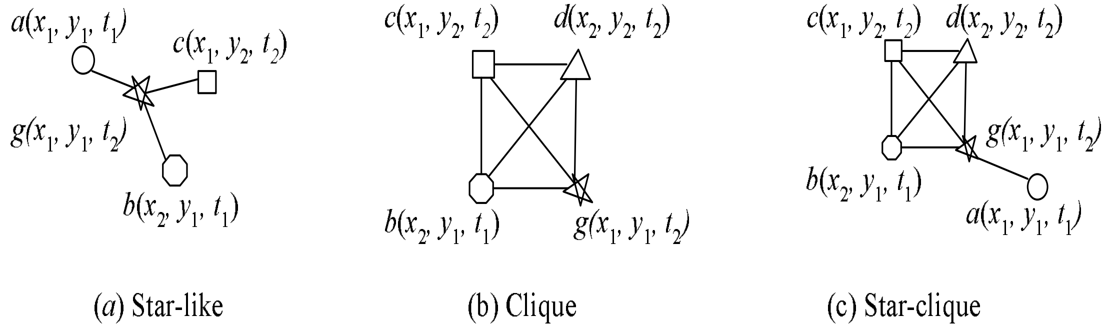


Figure 3.1: Example of two topological patterns

are not required to be located close to each other. We represent a star-like pattern with $\langle f_j : \{f_1, ..., f_k\}\rangle$. Figure 3.1(a) shows an example of a star-like pattern $\langle g : \{a, b, c\}\rangle$.

A topological pattern S is said to be a *clique pattern* if and only if in a valid instance of S, all pairs of the features' instances are located close to *each other* (i.e., they are located close both in positions and time). In other words, the closeness relationships of the instances form a *clique graph*. A clique pattern is denoted as $\langle f_1, f_2, \ldots, f_k\rangle$. For example, Figure 3.1(b) shows a clique pattern where the instances of the features in the set $\langle b(x_2, y_1, t_1), c(x_1, y_2, t_2), d(x_2, y_2, t_2), g(x_1, y_1, t_2)\rangle$ are close to each other.

A topological pattern S is a *star-clique pattern* if S contains a sub-clique pattern S' (i.e. $S' \subset S$), and there is a feature $f_j \in S'$ such that the instance of f_j is close to the instances of the features in $S \backslash S'$ and the instances of the features in $S \backslash S'$ are not required to be close to each other. A star-clique pattern is denoted as $\langle S' | f_i : S \backslash S'\rangle$. In essence, star-clique patterns can be generated by combining the star-like patterns with the clique patterns on a common feature. Figure 3.1(c) shows a star-clique pattern $\langle\langle b, c, d, g\rangle | g : \{a\}\rangle$.

Two measurements, *support* [KH95] and *participation ratio* [SH01], have been in-

troduced to measure the implication strength of a spatial feature in a topological pattern.

The *support* of a pattern S is defined as the number of instances of S found in the database. The support of a pattern S also defines the *confidence* of the corresponding rule in the form of $f_i \Rightarrow \{f_1, \ldots, f_s\}$. For example, Figure 3.1(a) defines a rule $g \Rightarrow \{a, b, c\}$, which means that if there is an instance of g, there is high confidence that it is close to the instances of features a, b and c while the instances of features a, b and c do not need to be close to each other.

Different from support, *participation ratio* is used to capture the probability that whenever an instance feature $f_i \in S$ appears on the map, it will participate in an instance of S. The participation ratio of a feature f_i in a pattern S, denoted as $pr(f_i, S)$, is defined by the following equation:

$$pr(f_i, S) = \frac{\sharp \text{ instances of } f_i \text{ in any instance of } S}{\sharp \text{ instances of } f_i} \tag{3.1}$$

In order to characterize the strength of a topological pattern in implying the co-occurrence of features, the *prevalence* [ZMCS04] of a pattern S , denoted as $prevalence(S)$, is further defined as the *minimum* probability among all the features of S, that is $prevalence(S)$ = $min\{pr(f_i, S), f_i \in S\}$. The prevalence is *monotonic*; if $S \preceq S'$, then $prevalence(S) \geq prevalence(S')$. Additionally, with the prevalence of S, we can define the topology rule with the form $A \Rightarrow B$, where A and B are subsets of spatial features. For example, a rule $g \Rightarrow \{b, c, d, g\}$ can be obtained in Figure 3.1(b), which means that if there is an instance of the feature g, there is a high probability that it participates in the instances of the clique pattern $\langle b, c, d, g\rangle$. In this chapter, we use the prevalence threshold as our interestingness measure.

3.1.2 Geographical Features

With the concept of topological patterns, we now define the geographical features of topological patterns.

Geographical features of topological patterns are entities in the physical world, such as park, school, zoo, etc. These geographical features can be extracted from maps in geographical information systems and are kept in geographical feature databases with the format ⟨*geographical feature identifer, geographical feature type, minimum bounding rectangles*⟩.

A geographical feature, denoted as g, is indicated by a polygon or a minimum bounding rectangle (MBR) that describes its boundary. A geographical feature is said to be interesting with respect to a topological pattern S if it is always close to the spatial features in an instance of S. We define the distance between an instance i_{jk} of a spatial feature in S and the MBR of g as the minimum Euclidean distance from i_{jk} to g, denoted as $mindist(i_{jk}, g)$. A geographical feature g is *frequent* if the number of valid instances exceeds a user specified minimum support value.

Let R_g be the distance threshold for measuring the closeness of a geographical feature and the spatial features in a topological pattern. Here, we assume that $R_g \gg R$. This is to ensure that the instances' centroid of a length-k topological pattern S can be used to represent the positions of the spatial features' instances in S such that the interesting geographical features of S are also the interesting geographical features of spatial features in S.

Figure 3.2 shows the distance relation between the centroid and the instances of

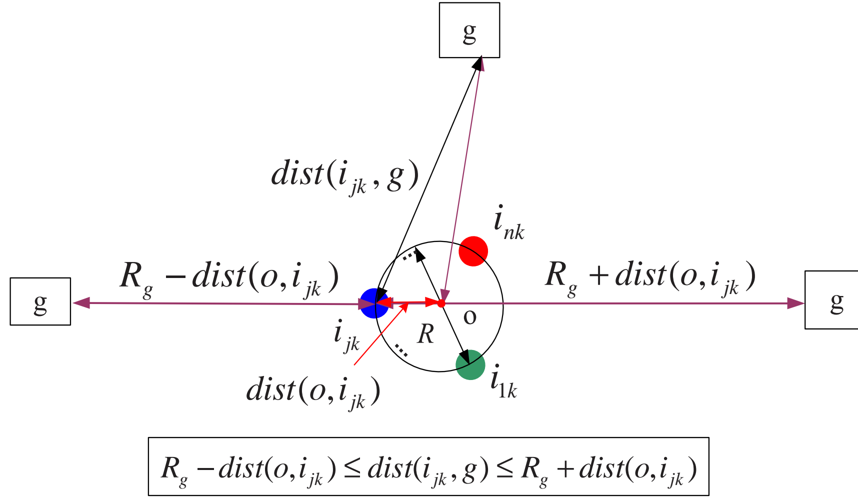


Figure 3.2: Relationship of distance to geographical feature

spatial features in a length-k topological pattern. Let $dist(o, i_{jk})$ be the distance from the centroid o to the instance i_{jk}, $dist(o, i_{jk})$ = μR where $\mu \leq 1$. Let $dist(o, g)$ be the distance from the centroid o to the geographical feature g and $dist(o, g) = R_g = \sigma R, \sigma \gg 1$.

Based on the triangle inequality, we have $(R_g - \mu R) \leq dist(i_{jk}, g) \leq (R_g + \mu R)$. If we regard $dist(o, g)$ as $dist(i_{jk}, g)$, then the error ϵ obtained is:

$$\begin{aligned} \epsilon &= \frac{|dist(o,g) - dist(i_{jk},g)|}{dist(i_{jk},g)} \approx \frac{|dist(o,g) - dist(i_{jk},g)|}{dist(o,g)} \\ &\leq \frac{dist(o,i_{jk})}{dist(o,g)} = \frac{\mu R}{\sigma R} = \frac{\mu}{\sigma} \\ &\leq \frac{1}{\sigma} \end{aligned} \tag{3.2}$$

From the above equation, we observe that the error is only related to $\sigma = \frac{R_g}{R}$. When σ is big, the distance from the spatial features in an instance of a topological pattern to the geographical feature approximates the distance from the centroid of the instance to the geographical feature, that is $dist(i_{jk}, g) \approx dist(o, g)$.

Note that finding the geographical features of topological patterns involves mining patterns across two types of databases, that is, spatio-temporal databases (e.g., mobile service databases) and geographical feature databases.

With the above definition, we can now define the problem to find topological patterns as follows: *Given a spatio-temporal database* $\mathcal{D}$ *and a geographical feature database* $\mathcal{D}_g$*, the distance thresholds* R *and* R_g*, a time window threshold* W*, and the minimum prevalence threshold* minprev*, we aim to find all frequent topological patterns, i.e., star-like, clique and star-clique patterns, and their geographical features.*

3.2 Pattern Growth Approach

Before we illustrate the algorithm in detail, we first explain the framework of the pattern growth method.

The pattern growth method has been shown to be one of the most effective methods for frequent pattern mining and is superior to the candidate-maintenance-test approach, especially on a dense database or with low minimum support threshold [HP00]. As a divide-and-conquer method, the pattern growth method partitions the database into subsets recursively, but does not generate candidate sets. It also makes use of the *Apriori* property to prune search space and counts frequent patterns in order to decide whether it can assemble longer patterns.

The sequential pattern mining algorithm PrefixSpan [PHMAP01] provides a general framework of the pattern growth method. The basic idea is to use a set of locally

Seq ID.	Sequence
s_1	$\langle a, c, f\rangle \rightarrow \langle a, c, d, f, g\rangle \rightarrow \langle d, g\rangle$
s_2	$\langle a, f, g\rangle$
s_3	$\langle a, c, d, f\rangle \rightarrow \langle a, c, d, f, g\rangle \rightarrow \langle a, c, d, g\rangle$

(a) Sample sequence database

Seq ID.	Sequence
s_{1a}	$\langle \ddagger, c, f\rangle \rightarrow \langle a, c, d, f, g\rangle \rightarrow \langle d, g\rangle$
s_{2a}	$\langle \ddagger, f, g\rangle$
s_{3a}	$\langle \ddagger, c, d, f\rangle \rightarrow \langle a, c, d, f, g\rangle \rightarrow \langle a, c, d, g\rangle$

(b) a-projected database

Figure 3.3: Projection sequential pattern mining

frequent items to grow patterns.

For example, Figure 3.3(a) shows a sample sequence database. The set of frequent items $F_1 = \{a, c, d, f, g\}$ when $minsup$ = 2. Let us find the sequential patterns with prefix a. Figure 3.3(b) shows the a-projected database, where only the subsequence prefixed with the first occurrence of a is considered. Note that $\langle \ddagger, c, f\rangle$ in s_{1a} means that the last element in the prefix which is a, together with c and f form one element. By scanning the a-projected database once, we get the locally frequent items of the item a, that is $LF_a = \{\langle \ddagger c\rangle, \langle \ddagger f\rangle, a, c, d, f, g\}$. We can generate the corresponding 2-sequences with prefix a, i.e., $\langle a, c\rangle$, $\langle a, f\rangle$, $a \rightarrow a$, $a \rightarrow c$, $a \rightarrow d$, $a \rightarrow f$ and $a \rightarrow g$. Then, we can further partition the set of frequent patterns prefixed with a into $|LF_a|$ = 7 subsets (i.e., the subset prefixed with $\langle a, c\rangle$, the subset prefixed with $\langle a, f\rangle$, etc.), construct their corresponding projected databases, and mine them recursively.

Unfortunately, spatio-temporal databases, like spatial databases, are very different from conventional databases (e.g., sequence databases). There is no explicit transaction concept in a spatio-temporal database. It is difficult to extend existing algorithms such

as [HP00] to find topological patterns in spatio-temporal databases. We need to design new algorithms that follow the pattern growth methodology to find topological patterns in spatio-temporal databases.

3.3 Algorithm TopologyMiner

In this section, we present the algorithm TopologyMiner for finding topological patterns. TopologyMiner finds frequent patterns in a depth-first manner. It divides search space into a set of partitions. In each partition, it uses a set of locally frequent features to grow patterns. It consists of two phases:

- In the first phase, it divides the space-time dimensions into a set of smaller disjoint cubes. Then, it scans the database once to build a summary-structure that records the instances' count information of the features in a cube. It further constructs two indices on the summary-structure to facilitate the mining of topological patterns.
- The second phase utilizes the count information stored in the summary-structure to discover frequent topological patterns in a depth-first manner.

3.3.1 Summary structure

First, we introduce the *summary-structure*. Let $\mathcal{D}$ be the spatio-temporal database, R be the distance threshold, and W be the time window threshold. We divide the database $\mathcal{D}$ into a set of disjoint cubes $\{\langle c_{x_1,y_1}, w_1\rangle, \ldots, \langle c_{x_1,y_1}, w_q\rangle$, ..., $\langle c_{x_p,y_p}, w_1\rangle$, ..., $\langle c_{x_p,y_p}, w_q\rangle\}$ where $\{c_{x_1,y_1}, \ldots, c_{x_p,y_p}\}$ are 2D cells with width $\frac{R}{\sqrt{2}}$, and $\{w_1, w_2, ..., w_q\}$

are 1D time periods with width $\frac{W}{2}$.

For the instances in a cube $\langle c_{x_k,y_k}, w_t \rangle$, we can easily determine their close neighbors, which must be the instances in one of the cubes $\langle c_{x_i,y_i}, w_s \rangle$ in the set $N_{c_{x_k,y_k},w_t} = \{\langle c_{x_i,y_i}, w_s \rangle | \, |y_k - y_i| \leq 2 \wedge |t-s| \leq 2 \wedge |x_k - x_i| \leq 2$ and if $|x_k - x_i| = 2, |y_k - y_i| \neq 2)\}$. We call $N_{c_{x_k,y_k},w_t}$ the *neighbor-set* of the cube $\langle c_{x_k,y_k}, w_t \rangle$. Note that two instances are near in position if and only if their cubes are neighbors.

Let $L = \{\langle c_{x1,y1}, w_1 \rangle, \ldots, \langle c_{x_i,y_i}, w_s \rangle\}$ be a list of cubes. The neighbor-set of L, denoted as N_L, is the join of the neighbor-set of each cube in L, i.e., $N_L = N_{c_{x_1,y_1},w_1} \cap \ldots \cap N_{c_{x_i,y_i},w_s}$.

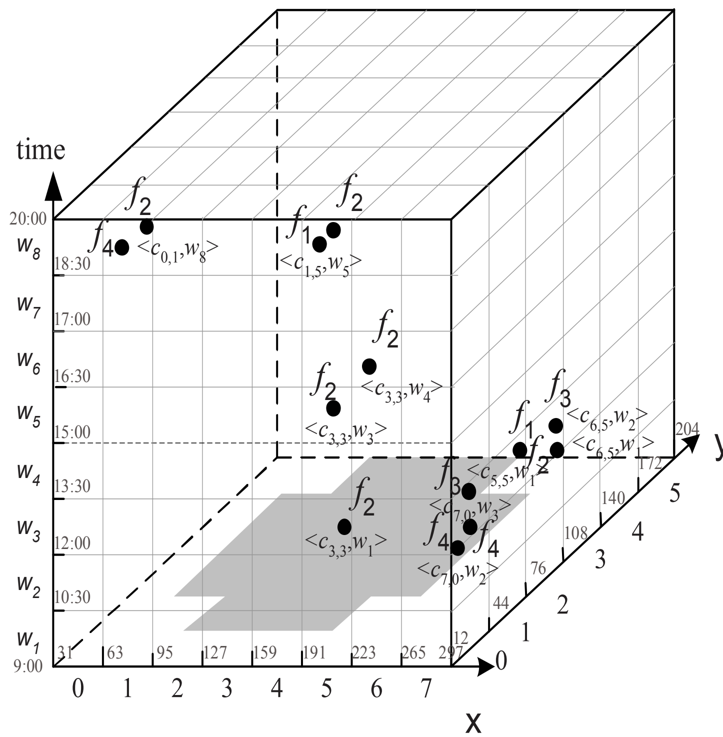
Figure 3.4 shows an example of the spatio-temporal database with $R = 45$ and $W = 90\, mins$. The space is divided into 48 cells and the time is divided into 8 time periods. The neighbor-set of the cell $c_{3,3}$, i.e., $N_{c_{3,3}}$, is marked in grey. The neighbor-set of the time period w_4 consists of the time periods w_2, w_3, w_4, w_5 and w_6. Hence, the neighbor-set of the cube $\langle c_{3,3}, w_4 \rangle$ is the join of the neighbor-set $N_{c_{3,3}}$ with $\{w_2, w_3, w_4, w_5, w_6\}$.

After dividing time and space into a set of cubes, we scan the database once, and hash the instances of the features into the corresponding cubes. For each cube $\langle c_{x_i,y_i}, w_s \rangle$, we keep the instances' count of a feature f_j in the main memory. Note that only those cubes that contain at least one feature instance will be stored in the summary-structure. Compared to the original database of N instances, the size of the summary-structure is $O(\frac{N}{k})$, assuming each cube contains $k (k \geq 1)$ instances.

To facilitate information retrieval operations in the summary-structure, we construct two hash-based indices, called *Cube-Feature Index* (CFI) and *Feature-Cube Index*

tid	fid	position	time
1	f_1	(68,185)	15:32:01
2	f_1	(200,180)	9:05:31
3	f_2	(70,202)	15:45:01
4	f_2	(57, 59)	19:25:31
5	f_2	(130, 120)	13:03:33
6	f_2	(235, 200)	9:25:31
7	f_3	(240, 180)	11:19:07
8	f_3	(263, 15)	12:29:54
9	f_4	(31, 62)	19:05:45
10	f_4	(268, 28)	11:55:14
11	f_4	(275, 12)	11:29:54
12	f_2	(128, 125)	10:21:43
13	f_2	(135, 115)	14:05:26

(a) spatio-temporal db



(b) space-time view

Figure 3.4: Example of a spatio-temporal database

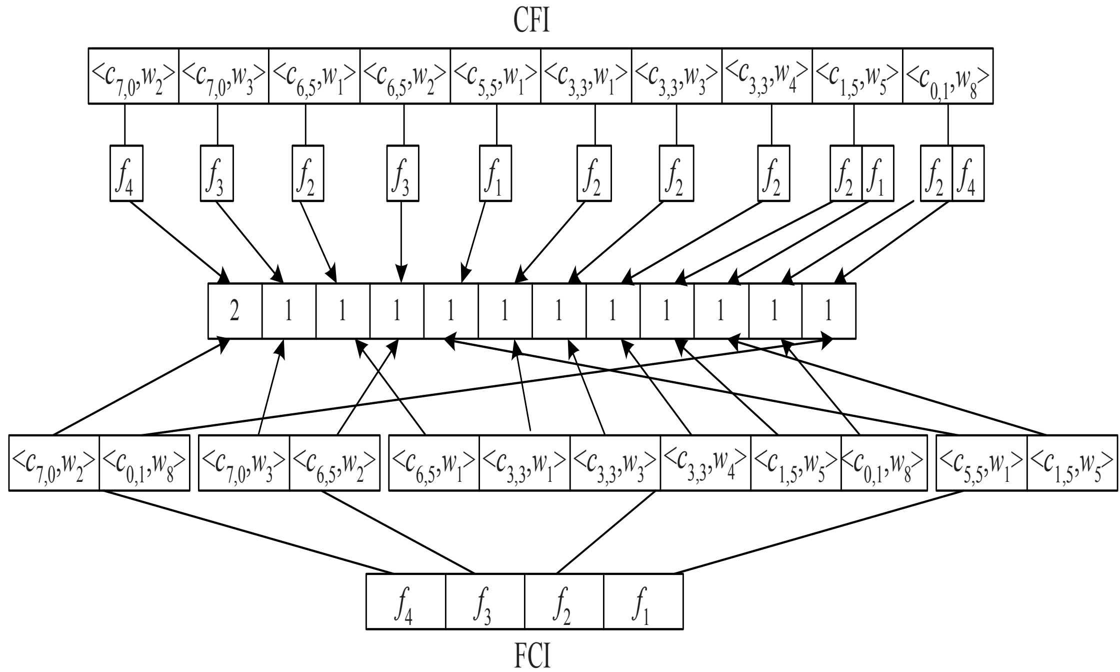


Figure 3.5: Example of a summary-structure

(FCI). Both indices are two-level structures. Specifically, CFI is built with the composite key $(\langle c_{x_i,y_i}, w_s\rangle, fid)$, and its first level is used to index the cube with the identifier $\langle c_{x_i,y_i}, w_s\rangle$, and its second level is utilized to index features with the identifier fid. With CFI, we can obtain features that occur in a cube $\langle c_{x_i,y_i}, w_s\rangle$, and retrieve their correspondingly instances' count in the cube in constant time.

FCI is built using the composite key (fid, $\langle c_{x_i,y_i}, w_s\rangle$). The first level of FCI is used to index features with the identifier fid, and the second level indexes cubes with the identifier $\langle c_{x_i,y_i}, w_s\rangle$ respectively. FCI helps to determine corresponding cubes in which a feature fid occurs and obtain its instances' count in constant time.

Figure 3.5 gives an example of the summary-structure with the two indices, CFI and FCI, for the database in Figure 3.4.

With these two indices, we can approximate the number of instances of a topological pattern. Recall we consider two instances are near in the position if and only if their cubes are neighbors. This means that the instances of a feature f_i in a cube $\langle c_{x_1,y_1}, w_1 \rangle$ and the instances of a feature f_j in the neighboring cube $\langle c_{x_2,y_2}, w_2 \rangle$ form the valid instances of a topological pattern $\langle f_i, f_j \rangle$. In other words, the instances' count of a feature in a topological pattern can be obtained from the summary-structure directly. With this in mind, we now proceed to explain the process of finding frequent topological patterns.

3.3.2 Mining Topological Patterns

Now, we discuss the steps to find topological patterns. We first define the projected database of a length-k topological pattern S, and then explain how to construct the projected database of S from the summary-structure. Finally, we illustrate the process of mining frequent topological patterns in the projected database of S.

Concept of Projected Database

We define the projected database of a topological pattern as a collection of sets of cubes in which instances of the features in a topological pattern occur, and a set of related features.

Let $S = \{f_1, ..., f_k\}$ be a length-k topological pattern. The *projected database* of S, denoted as P_S, is the collection of entries $\langle L, R_p \rangle$, where L is a list of cubes and R_p is a pointer pointing to a list of features that are related to the pattern S. These features are

found either in L or the neighboring cubes of L. For brevity, we use $P_S.L$ and $P_S.R_p$ to represent the cube-list and the feature-list respectively.

The *cube-list* $P_S.L$, denoted as $(\langle c_{x_1,y_1}, w_1\rangle$, ..., $\langle c_{x_k,y_k}, w_k\rangle)$, is used to store the cubes in which the instances of the features in S occur, and all the cubes in the cube-list must be neighbors. In other words, the ith cube $\langle c_{x_i,y_i}, w_i\rangle$ contains the instances of the ith feature $f_i \in S$, $1 \leq i \leq k$. With the cube-list, we could obtain the instances' count of a feature participating in pattern S, and approximate the number of instances of S.

The *feature-list* $P_S.R_p$ stores the features that are related to the pattern S and the cubes where the features' instances occur. Each element in the feature-list has the format $(f_r : \langle c_{x_m,y_m}, w_m\rangle)$, where $f_r \geq f_k$, $f_k \in S$, and the cube $\langle c_{x_m,y_m}, w_m\rangle$ that contains the instances of the feature f_r, is a neighboring cube of the cube-list $P_S.L$. The feature-list stores the potential features that can be used to combine with S to generate longer patterns. Figure 3.6(a) shows an example of the projected database of f_1.

Construction of Projected Database

The projected database of a topological pattern can be obtained from the summary-structure directly. Consider the construction of the projected database of pattern f_i. First, we obtain the cube-lists of f_i by scanning FCI. For example in Figure 3.6(a), we get two cube-lists of f_1 by scanning FCI in Figure 3.5, that is $L_1 = (\langle c_{1,5}, w_5\rangle)$ and $L_2 = (\langle c_{5,5}, w_1\rangle)$.

For each cube-list L in P_{f_i}, we obtain its neighbor-set N_L. Then, for each cube in the neighbor-set N_L, we get the set of related features (i.e., RF) from CFI. With the

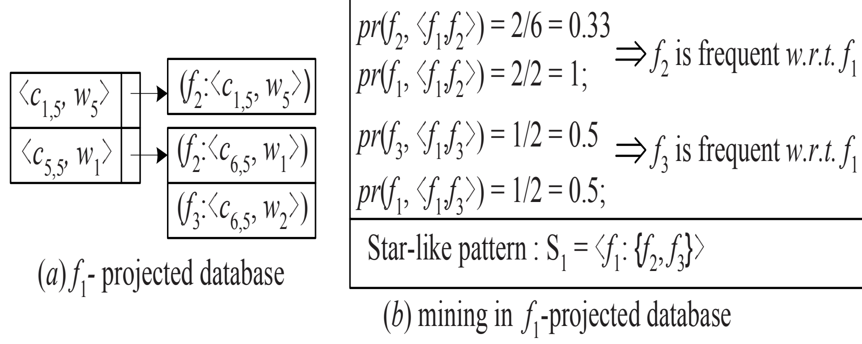


Figure 3.6: The projected database of f_1

related features and the neighboring cubes, we generate new entries for the feature-list R_P in P_{f_i}. Figure 3.6(a) shows sample feature-lists in P_{f_1}. Initially, for the cube-list $L_1 = (\langle c_{1,5}, w_5\rangle)$, there is only one valid neighboring cube, i.e., neighbor-set $N_{L_1} = \{\langle c_{1,5}, w_5\rangle\}$. From $c_{1,5}$, we obtain the set of related features of f_1 i.e., $RF = \{f_2\}$. Finally, we add the entry $(f_2 : \langle c_{1,5}, w_5\rangle)$ into the feature-list R_{p_1}. Similarly, for the cube-list L_2, we obtain two entries for the feature-list R_{p_2}, that is $(f_2 : \langle c_{6,5}, w_1\rangle)$ and $(f_3 : \langle c_{6,5}, w_2\rangle)$.

The projected database of a length-k pattern $S_k = \{f_1, f_2, \ldots, f_{k-1}, f_k\}$ ($k \geq 2$) can be derived from the projected database of its prefix (k-1)-subpattern $S_{k-1} = \{f_1, f_2, \ldots, f_{k-1}\}$. Note that the feature f_k is a related feature of the pattern S_{k-1} and $f_k \geq f_{k-1}$. We construct P_{S_k} from $P_{S_{k-1}}$ as follows: For each entry $\langle L, R_p\rangle$ of $P_{S_{k-1}}$ where the feature-list $P_{S_{k-1}}.R_p$ contains an element of the feature f_k, e.g., $(f_k : \langle c_{x_m,y_m}, w_m\rangle)$, we yield a new entry $\langle L, R_p\rangle$ for P_{S_k}, where $P_{S_k}.L = P_{S_{k-1}}.L \cup \{\langle c_{x_m,y_m}, w_m\rangle\}$ and the feature-list $P_{S_k}.R_p$ is a subset of the feature-list $P_{S_{k-1}}.R_p$, that is each element $(f_r, \langle c_{x_s,y_s}, w_s\rangle)$ in the feature-list $P_{S_k}.R_p$, where $f_r \geq f_k$ and $\langle c_{x_s,y_s}, w_s\rangle$

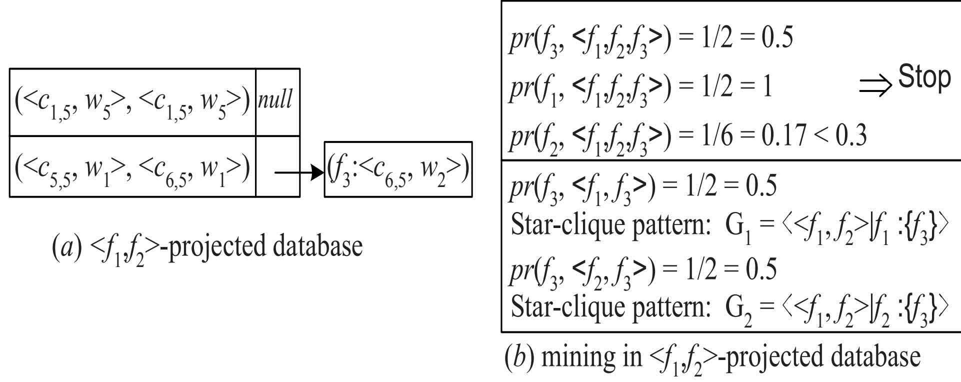


Figure 3.7: The projected databases of $\langle f_1, f_2\rangle$

is a neighbor of the cube $\langle c_{x_m,y_m}, w_m\rangle$.

Figure 3.7(a) shows the projected database of the pattern $\langle f_1, f_2\rangle$, which is derived from P_{f_1}. In Figure 3.6(a), we know that the feature f_2 is contained in both entries of P_{f_1}. Hence, we generate two entries for $P_{\langle f_1,f_2\rangle}$, that is $L_1 = (\langle c_{1,5}, w_5\rangle, \langle c_{1,5}, w_5\rangle)$ and $L_2 = (\langle c_{5,5}, w_1\rangle, \langle c_{6,5}, w_2\rangle)$. Since the feature f_3 is only contained in $P_{f_1}.R_{p2}$ and $\langle c_{6,5}, w_2\rangle$ is a neighbor of $\langle c_{6,5}, w_1\rangle$, we add the entry $(f_3 : \langle c_{6,5}, w_2\rangle)$ into the feature-list $P_{\langle f_1,f_2\rangle}.R_{p2}$.

Mining Projected Databases

Let S_k be a frequent length-k topological pattern and P_{S_k} be constructed already. Now, we see the process to mine topological patterns from the projected database of S_k.

Mining Star-like Patterns. To get star-like patterns, we directly mine the projected database of the features (i.e., 1-topological patterns). For a feature f_i, the feature f_j is said to be a frequent related feature of f_i if and only if $pr(f_j, \langle f_i, f_j\rangle) \geq minprev$. All the frequent related features of f_i form a star-like pattern $S = \{f_i : \langle f_{r1}, \ldots, f_{rm}\rangle\}$.

For example in Figure 3.6(a), since $pr(f_2, \langle f_1, f_2\rangle)$ and $pr(f_3, \langle f_1, f_3\rangle)$ are greater than 0.3, we can generate a star-like pattern $S_1 = \langle f_1 : \{f_2, f_3\}\rangle$.

Mining Clique Patterns. The process to discover clique patterns is a little complicated. Our main idea is to check whether a related feature f_r of S_k can be combined with S_k to generate a longer clique pattern $S_{k+1} = S_k \cup \{f_r\}$ ($f_r \geq f_k$). To achieve this, we need to determine whether the prevalence of S_{k+1}, which is the minimum participation ratio among all features in S_{k+1}, is equivalent to or greater than $minprev$. In other words, we not only need to compute the participation ratio of related feature f_r w.r.t. S_{k+1}, but also the participation ratio of the features in S_k w.r.t. S_{k+1}. This is due to the incorporation of the feature f_r in S_k, $pr(f_i, S_{k+1}) \neq pr(f_i, S_k)$ for each $f_i \in S_k$.

Suppose the set RF contains all the related features of S_k, The features in RF are ordered according to $\prec_f$. For each feature f_r in RF, we first compute the participation ratio $pr(f_r, S_{k+1})$ in the projected database S_k. The main step is to obtain the instances' count of f_r through the FCI. Specifically, for each feature-list R_p in P_{S_k}, which has an element containing the feature f_r, e.g., (f_r : $\langle c_{x_m,y_m}$, $w_m\rangle$), we obtain the instance's count of f_r in the cube by indexing FCI with the key (f_r, $\langle c_{x_m,y_m}$, $w_m\rangle$). Note that for each key, we only query FCI once. After obtaining all the instances' count of f_r in P_{S_k}, we compute $pr(f_r, S_{k+1})$ according to Equation 3.1. If $pr(f_r, S_{k+1}) \geq minprev$, we continue to compute $pr(f_i, S_{k+1})$ for each $f_i \in S$. Otherwise, we remove f_r from RF, since it cannot be combined with the pattern S_k to generate any frequent topological pattern.

The process to compute the participation ratio $pr(fi, S_{k+1})$ for each feature $f_i \in S_k$

proceeds as follows. For each entry $\langle L, R_p \rangle$ in P_{S_k} such that the feature-list $P_{S_k}.R_p$ contains the feature f_r, we get the cube $\langle c_{x_i,y_i}, w_i \rangle$ in L because it contains the instances of feature $f_i \in S_k$. We obtain the instances' count of f_i through FCI with the key $(f_i, \langle c_{x_i,y_i}, w_i \rangle)$. After scanning all entries in P_{S_k}, we compute $pr(f_i, S_{k+1})$ using Equation 3.1. Once there is a feature in S_k whose participation ratio with respect to S_{k+1} is less than $minprev$, we stop the process because the prevalence of S_{k+1} cannot be greater than $minprev$. Only when $prevalence(S_{k+1}) \geq minprev$, we output the pattern S_{k+1}, construct the projected database of S_{k+1}, and mine it recursively.

Figure 3.6(b) shows the process of finding topological patterns with $minprev$ = 0.3 in Figure 3.6(a). In P_{f_1}, the features f_2 and f_3 are related features of f_1. To determine whether f_2 is a frequent related feature of f_1, we need to compute $prevalence(\langle f_1, f_2 \rangle)$. In other words, we need to compute the participation ratios $pr(f_2, \langle f_1, f_2 \rangle)$ and $pr(f_1, \langle f_1, f_2 \rangle)$. As we know, there are in total six instances of f_2 in the spatio-temporal database, and only two instances of f_2 participate in the instances of $\langle f_1, f_2 \rangle$, i.e., one instance in cube $\langle c_{1,5}, w_5 \rangle$ and one in cube $\langle c_{6,5}, w_1 \rangle$. Hence, $pr(f_2, \langle f_1, f_2 \rangle)$ = $\frac{2}{6}$ = 0.33. Since all instances of f_1 participate in the instances of the pattern $\langle f_1, f_2 \rangle$, we have $pr(f_1, \langle f_1, f_2 \rangle)$ = $\frac{2}{2}$ = 1. Finally, $prevalence(\langle f_1, f_2 \rangle)$ = $min\{0.33, 1\}$ = 0.33 $>$ 0.3. Hence, f_2 is a frequent related feature of f_1 and can be combined with f_1 to generate a longer frequent pattern $\langle f_1, f_2 \rangle$. Similarly, we have the $prevalence(\langle f_1, f_3 \rangle)$ = 0.5 $>$ 0.33, and hence, f_3 is also a frequent related feature of f_1. After mining the f_1-projected database, we construct the $\langle f_1, f_2 \rangle$-projected database and mine it recursively. Figure 3.7(b) shows the corresponding mining process.

Mining Star-clique Patterns. Finally, we examine the process to get star-clique patterns. The mining of star-clique patterns is invoked after a clique pattern S_k is yielded. A star-clique pattern is generated by combining the clique pattern S_k with the star-like pattern of the features f_i, $f_i \in S_k$. Specifically, for each feature $f_i \in S_k$, denoted as $S_k|f_i$, we first obtain the set of cubes in which f_i occurs. In other words, we do the projection of P_{S_k} on the feature f_i, denote it as $P_{S_k}|f_i$. Next, with the cubes in $P_{S_k}|f_i$, we recompute the participation ratio $pr'(f_r, \langle f_i, f_r \rangle)$ for each related feature f_r in the star-like pattern of the feature f_i. If $pr'(f_r, \langle S_k|f_i, f_r \rangle) \geq minprev$, f_r is said to be a frequent related feature of $S_k|f_i$. All the frequent related features of $S_k|f_i$ form a star-clique pattern with respect to $S_k|f_i$. For example in Figure 3.7(b), we can generate two star-clique patterns, i.e., $G_1 = \langle \langle f_1, f_2 \rangle | f_1 : \{f_3\} \rangle$ and $G_2 = \langle \langle f_1, f_2 \rangle | f_2 : \{f_3\} \rangle$.

3.3.3 Mining Geographical Features

After mining the frequent topological patterns, we can find the geographical features of these patterns. The process to find the interesting geographical features of topological patterns is very simple. Here, we assume that the R-tree [Gut84] index structure is built on the geographical feature databases.

To find interesting geographical features of a topological pattern S, we first compute the centroid for each instance of S. Next, we retrieve all the geographical features in the geographical feature database that lie within the user specified distance of R_g from the centroid. This is achieved by constructing a query window whose center is the centroid and whose radius is equal to R_g. Finally, we update the frequencies of the retrieved

geographical features of S. Note that a geographical feature is only counted once for S in its one instance search, no matter how many instances of this geographical feature are close to the S instance.

3.4 TopologyMiner Algorithm

Figure 3.8 shows the framework of TopologyMiner. It takes as input the spatio-temporal database $\mathcal{D}$, the distance threshold R, the time window threshold W and the prevalence threshold $minprev$, and outputs the set of frequent topological patterns. Line 1 scans the database once and constructs the summary-structure. We then discover the topological patterns at lines 4-25. Line 5 constructs the projected database for each feature f_i in $\mathcal{D}$ with CFI. Lines 7-17 discovers the star-like patterns with respect to f_i by scanning P_{f_i}. For each feature f_i in $\mathcal{D}$, lines 19-25 calls the procedure MiningPDB to find the longer frequent topological patterns.

The procedure MiningPDB (see Figure 3.9) works as follows: For each related feature f_r of S, line 2 computes the participation ratio $pr(f_r, S \cup \{f_r\})$ using FCI. If it is equivalent to or greater than $minprev$, lines 5-11 are executed to compute the participation ratio $pr(f_i, S \cup \{f_r\})$ for each feature $f_i \in S$. If there is a feature f_p such that $pr(f_p, S \cup \{f_r\})$ is less than $minprev$, the procedure terminates with the extension f_r (lines 8-10). If the participation ratios of all the features in S are greater than or equivalent to $minprev$, a new clique pattern $S' = S \cup \{f_r\}$ is generated. This is followed by a search for its geographical features, the construction of the projected database S'

Algorithm TopologyMiner

Input: $\mathcal{D}$, $\mathcal{D}_g$: the spatio-temporal database and geographical feature database;

R, W: the distance and time window threshold;

$minprev$: prevalence threshold.

Output: $\mathcal{S}, \mathcal{C}, \mathcal{G}$: the set of frequent star-like, clique and star-clique patterns

1: Scan $\mathcal{D}$ and construct CFI and FCI with R and W;

2: RF = {all the features in $\mathcal{D}$};

3: $\forall f_i \in \mathcal{D}, Fs_i = \emptyset$;

4: **for each** feature f_i in RF {

5: constructing the projected database of f_i;

6: RF_i = {related features in P_{f_i}};

7: **for each** related feature $f_j \in RF_i$ {

8: $pr_j = pr(f_j, \langle f_i, f_j \rangle)$ and $pr_i = pr(f_i, \langle f_i, f_j \rangle)$

9: compute pr_j and pr_i through CFI and FCI;

10: **if**($pr_j <$ minprev$||pr_i <$ minprev)

11: $RF_i = RF_i \backslash \{f_j\}$;

12: **if**($pr_j \geq minprev$) $Fs_i = Fs_i \cup \{f_j\}$;

13: **if**($pr_i \geq minprev$) $Fs_j = Fs_j \cup \{f_i\}$;

14: }

15: $S = \langle f_i : Fs_i \rangle$;

16: $\mathcal{S} = \mathcal{S} \cup \{S\}$;

17: }

18: **for each** feature f_i in RF {

19: **for each** feature $f_j \in RF_i$ {

20: $S' = \langle f_i, f_j \rangle$

21: $\mathcal{C} = \mathcal{C} \cup \{S'\}$;

22: construct $P_{S'}$ based on P_{f_i};

23: $RF_{S'}$ = {related features in $P_{S'}$};

24: call **MiningPDB**($P_{S'}$,$RF_{S'}$,$minprev$);

25: }

26: }

Figure 3.8: Outline of the TopologyMiner algorithm

```
Procedure MiningPDB(P_S, RF, minprev)
1:  for each feature f_r in RF {
2:      compute pr(f_r, S ∪ {f_r});
3:      if pr(f_r, S ∪ {f_r}) < minprev continue
4:      else{
5:          flag = 1;
6:          for each f_i ∈ S {
7:              compute pr(f_i, S ∪ {f_r});
8:              if(pr(f_i, S ∪ {f_r}) < minprev)
9:                  flag = 0;
10:                 break;
11:         }
12:         if(flag) {
13:             S' = S ∪ {f_r};
14:             finding geographical features of S';
15:             C = C ∪ {S'};
16:             construct P_S' based on P_S;
17:             RF_S' = {related features in P_S'};
18:             call GenGenericPtns(S', minprev);
19:             MiningPDB(P_S', RF_S', minprev)
20:         }
21:     }
22: }
Procedure GenGenericPtn(C, minprev)
1:  Fs = ∅;
2:  for each feature f_i ∈ C {
3:      get S ∈ 𝒮 s.t. S.f_i = f_i
4:      for each frequent related feature f_r ∈ S.Fs
5:          if (pr(f_r, ⟨C|f_i, f_r⟩) ≥ minprev) Fs = Fs ∪ {f_r};
6:      G = {C|f_i : Fs};
7:      𝒢 = 𝒢 ∪ {G};
8:  }
```

Figure 3.9: Procedure MiningPDB

using CFI, invoking the procedure GenGenricPtn to find star-clique patterns, and then mining the projected database S' recursively (lines 13-19). The process continues until there are no more frequent topological patterns.

We now show that Algorithm TopologyMiner is correct and complete.

Theorem 1 *TopologyMiner is correct and complete.*

Proof: At the beginning, TopologyMiner initializes the length-1 topological patterns to all spatial features in the database and obtains the corresponding projected databases. This is correct and complete as the features' participation ratios are equivalent to 1.

Next, we assume that TopologyMiner can correctly discover all the length-k topological patterns. Let α be a length-k topological pattern and $\{\beta_1, \beta_2, \ldots, \beta_m\}$ be the set of all length-(k+1) topological patterns having prefix α. The complete set of topological patterns having prefix α is divided into m disjoint subsets. The jth subset $(1 \leq j \leq m)$ is the set of topological patterns having prefix β_j. Each subset of topological patterns can be further divided when necessary. To mine the subsets of topological patterns, TopologyMiner constructs the corresponding projected databases. In other words, TopologyMiner can correctly discover all the frequent length-(k+1) topological patterns.

The completeness of the mining in the projected database of a topological pattern can be argued as follows. The projected database of a topological pattern S_k consists of cube-lists and feature-lists. With the cube-lists, we can obtain the instances of the features participating in S_k. With the feature-lists, we can get all the related features and the cubes containing the instances of the related features. This means we can deter-

mine the complete set of the frequent related features of the pattern using the projected database. In other words, we can find all frequent topological patterns.■

3.5 Experimental Study

In this section, we evaluate the effectiveness and efficiency of TopologyMiner in finding all frequent topological patterns in spatio-temporal databases. As described in Section 2.1, there are two Apriori-like algorithms [KH95, SH01] proposed for finding all the frequent star-like or clique patterns in spatial databases. To compare them with TopologyMiner, we extend the algorithm in [SH01] to find topological patterns in spatio-temporal databases by incorporating the temporal aspect into the mining process. We use spatial join using neighbor relationship to generate the patterns of size 2. For the patterns of size 3 and more, we use the neighbor-based pruning to generate the instances of the candidates.

The algorithms are implemented in C++, and run on a Pentium 4, 3GHZ, 1G main memory PC.

3.5.1 Synthetic Data Generation

Recently, the author in [ZMCS04] presented a data generator for generating spatial datasets. While the algorithm TopologyMiner is to find topological patterns with instances near both in position and time in spatio-temporal databases. We cannot use it directly.

In order to generate spatio-temporal datasets, we modify the synthetic data generator described in [ZMCS04] for generating spatio-temporal data. Table 3.1 summarizes the parameters used in the data generator. First, we set L features, which we call non-noise features and which can appear in the longest collocation pattern generated. We also set n noise features. The number of points for noise features is $r \times N$. We assign these points to the noise features uniformly. The remaining points are assigned to non-noise features uniformly. The participation ratio of a feature in the longest pattern which has a participation ratio larger than the confidence threshold is $\delta_{max} + \theta$. The number of points N_i, which must appear in the instances of the longest pattern of a feature f_i is $(\delta_{max} + \theta) \times \frac{N \times (1-r)}{L}$. For other features, the participation ratios are $\delta_{min} + \theta$ and the number of points in instances of the longest pattern is $(\delta_{min} + \theta) \times \frac{N \times (1-r)}{L}$.

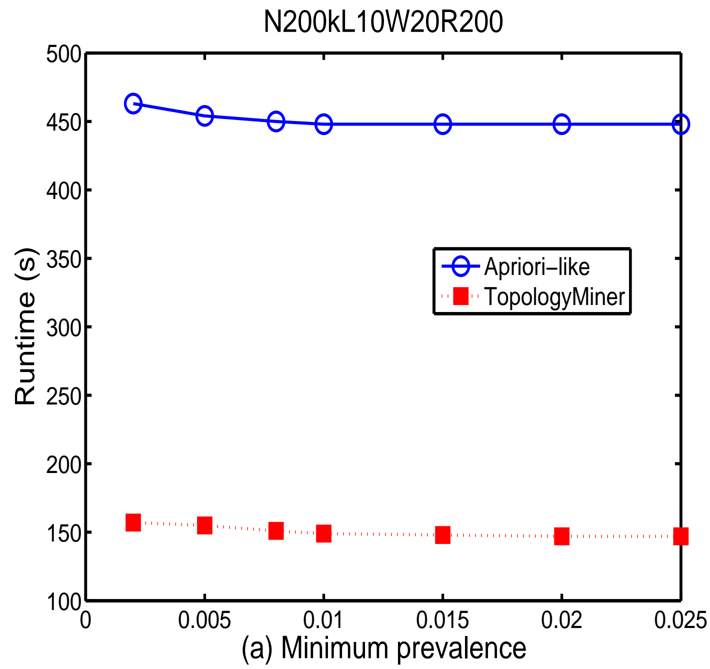
We generate instances of the longest pattern as follows. We divide the space into regular cells by dividing the map using a regular grid of cell-side length R and dividing the time using a regular window of length λ. At first, we generate a point randomly. We use the point as the center and generate points for a feature in the longest pattern around a cube, where r_s is the radius of a circle in space and r_t is the radius in time. The coordinates for the i-th point of the feature f_j is $(x_c + r_s \times sin\frac{2\pi i}{n_j}, y_c + r_s \times sin\frac{2\pi i}{n_j}, t_c + r_t \times \frac{i}{n_j}), n_j = \frac{N_i}{d}$. Because r_s is in $(0, \frac{R}{2})$ and r_t is in $(0, \frac{\lambda}{2})$, we can assign r_s and r_t the values $\sigma(\frac{R}{2L})$ and $\sigma(\frac{\lambda}{2L})$, $(1 \leq \sigma \leq L)$. In this way, any point in the cube can participate in an instance of the longest pattern. After selecting the first center point, we mark the cubes that intersect the cube centered at it such that no other longest pattern instances can be generated in them. Next, we continue to generate pattern instances from random

Table 3.1: Data generation parameters

Parameter	Meaning	Default
N	♯ points on the map ($\times$ 1000)	200k
L	♯ features in the longest pattern	10
m	♯ prevalent features in the longest pattern	8
n	♯ noise features	2
r	percentage of points with noise features	0.1
d	♯ generated longest pattern instances	1500
θ	minimum prevalence threshold	0.05
δ_{min}	minimum difference between prevalence of longest pattern and θ	-0.023
δ_{max}	maximum difference between prevalence of longest pattern and θ	0.08
R	distance threshold	200
λ	time window threshold	20
map	x - and y- extent of the map	8000 $\times$ 8000
T	extent of the time dimension	1000

points whose extended cube does not intersect the used cubes. After generating pattern instances d times, the process ends. The remaining points of the features which appear in the longest pattern are generated randomly on the map. Finally, we generate the points of noise features randomly on the map.

The generator described above generates instances of a long pattern with length L. The number of features which have participation ratios larger than the prevalence threshold is set to m.



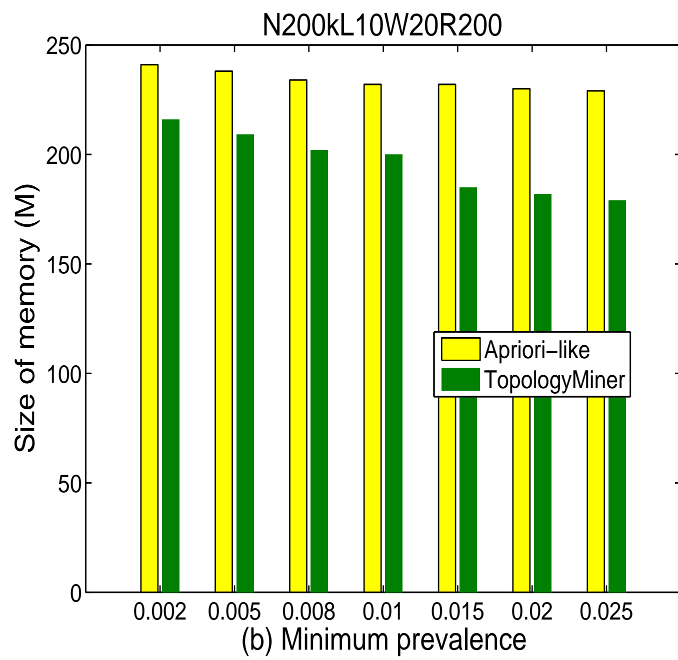


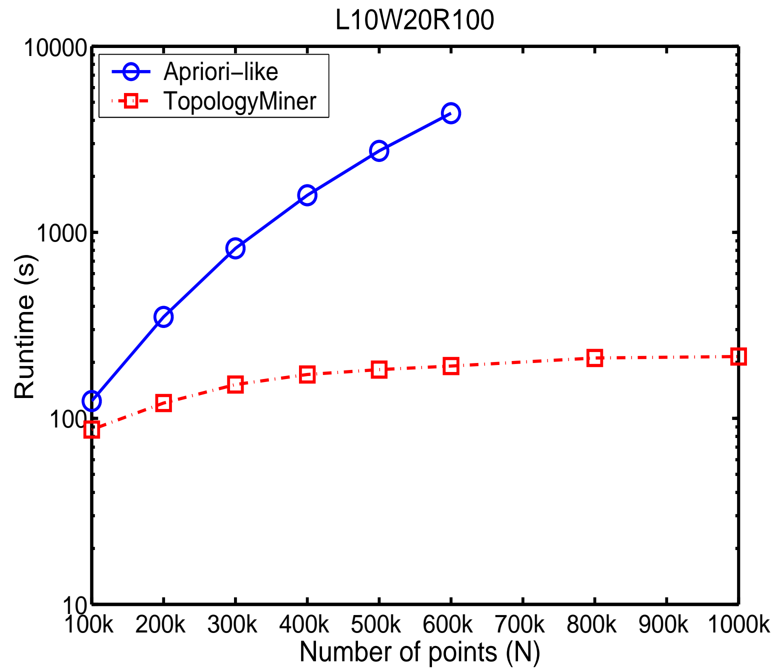
Figure 3.10: Runtime vs. prevalence threshold

3.5.2 Effect of Prevalence Threshold

We first evaluate TopologyMiner by varying the prevalence threshold. The results are shown in Figure 3.10. Compared to the Apriori-like algorithm, TopologyMiner needs less time and space to find the topological patterns. This is expected because when the prevalence threshold decreases, more topological patterns become frequent and the length of the frequent patterns tend to be longer. As a result, the Apriori-like algorithm needs more time to compute the instances of topological patterns and count their frequency. In contrast, TopologyMiner uses the summary-structure to approximate instances' count of a topological pattern and eliminate the generation and computation of the instances, which is very costly.

3.5.3 Effect of Database Size

Next, we study the effect of number of points N in a dataset. Figure 3.11 shows the results by varying N from $100k$ to $1000k$. Figure 3.11(a) indicates that TopologyMiner scales linearly with number of points. Compared to the Apriori-like algorithm, TopologyMiner requires less time to find frequent topological patterns with the increase of N. Note that the Apriori-like algorithm runs out of memory when N is greater than $600k$. This is because Apriori-like algorithm needs more time to test the interestingness of the candidates' instances. As N increases, the number of the instances of the frequent patterns become larger. As a result, the Apriori-like algorithm needs more time to compute the instances of the patterns and more memory to maintain these instances (see Figure 3.11(b)). Unlike the Apriori-like algorithm, TopologyMiner finds frequent



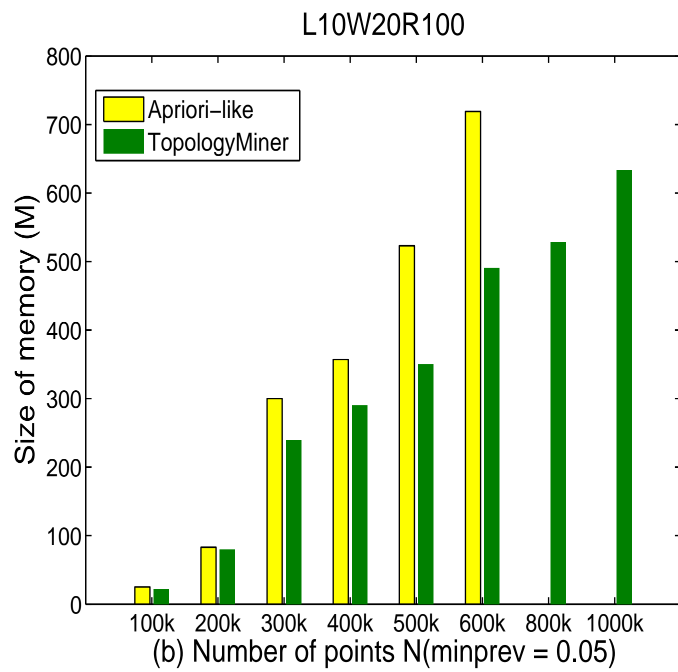


Figure 3.11: Runtime vs. number of points N

topological patterns in a depth-first manner. For each frequent pattern, it just needs to maintain the corresponding projected database which reduces with the length of the patterns.

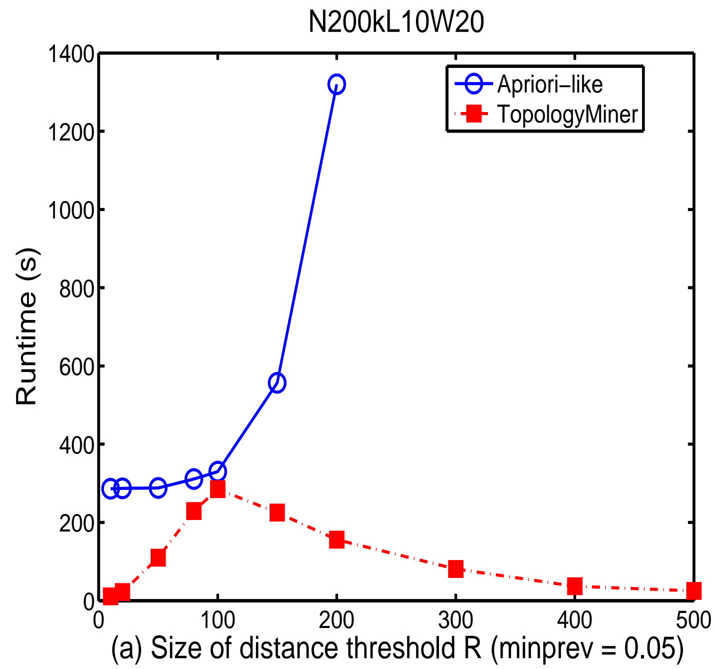
3.5.4 Effect of Distance Thresholds

We test the performance of TopologyMiner of varying the distance threshold R and the time window threshold W. Figure 3.12 (a) shows the results by varying the distance threshold R. A peak is reached when $R = 100$. This is because the performance of TopologyMiner is dependent on number of cubes, size of frequent patterns and length of frequent patterns. As the distance threshold R increases, the number of cubes decreases while the size of frequent patterns becomes larger and the length of frequent patterns increases. There is a tradeoff among the three factors with the maximum runtime recorded when $R = 100$. The figure also shows that TopologyMiner has an advantage over the Apriori-like algorithm. Note that in Figure 3.12(a), when R is larger than 200, the Apriori-like algorithm runs out of memory.

A similar trend is also observed when we vary the time window threshold W (see Figure 3.12(b)). Here, the longest runtime is recorded when W = 20. The same explanation applies here.

3.5.5 Effect of Number of Features

Finally, we study the performance of the algorithms by varying the parameters L (i.e., number of features in the longest pattern) and m (i.e., the prevalent features in the



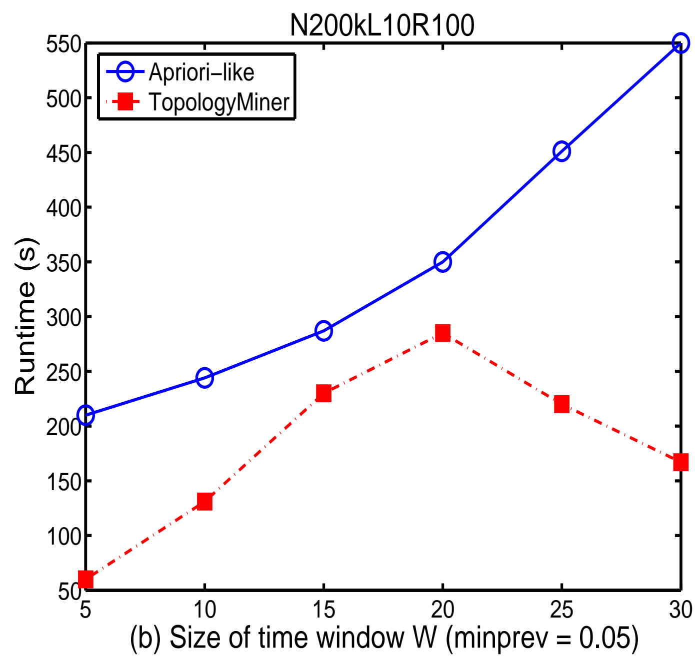


Figure 3.12: Runtime vs. distance thresholds

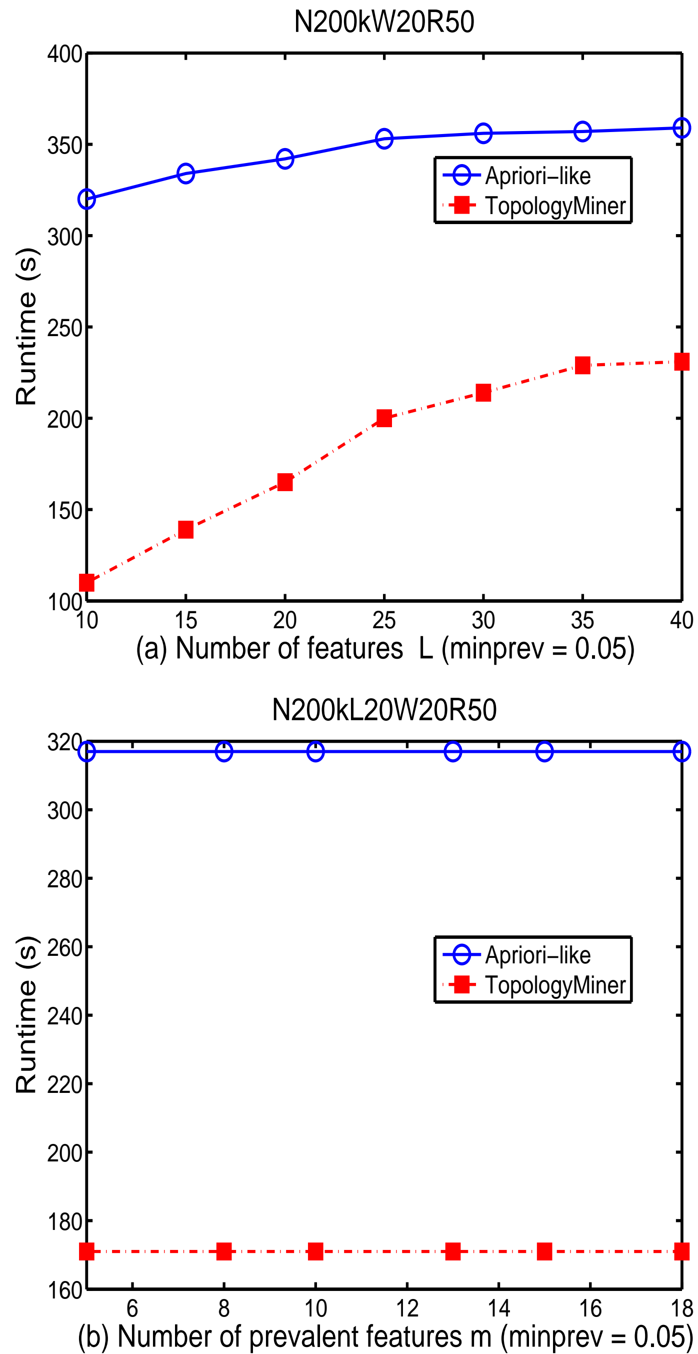


Figure 3.13: Runtime vs. number of features

longest pattern). Both parameters have an effect on the length of frequent patterns. Figure 3.13(a) shows the results of varying the parameter L with values ranging from 10 to 40. We observe that TopologyMiner scales linearly with number of features in the longest pattern and outperforms the Apriori-like algorithm by 50%.

Similarly, figure 3.13(b) shows the results of the algorithm when the parameter m is varied with values ranging from 5 to 18. As expected, the parameter m has little effect on TopologyMiner.

This is because the advantage of TopologyMiner is lies in the use of the summary-structure that not only enables TopologyMiner to count the frequency of a pattern efficiently but also allows it to find the pattern in a depth-first manner, thus eliminating the generation of a huge number of candidates.

3.5.6 Comparative Study on Finding Interesting Geographical Features

This set of experiments aims to show the usefulness of geographical based topological patterns compared to the topological patterns. Here, we use the neighboring requested service sets (i.e., service request sets that are located close to each other) as corresponding topological patterns.

We generate service requests according to some common habits we observe in real life. Table 3.2 lists the 10 top habits that we have observed. It includes geographical features, service requests that are close by, and their respective correlation coefficients. The correlation coefficients determine the probability that a service request is close to a

geographical feature.

Table 3.2: Observed common habits

Geo-Feature	Service Requests	Corr-Coefficients
Shopping Complex	Fun Place, Restaurant, Hairdressing Salon, ATM	0.2
	Nearest Friend, Taxi	0.1
Hotel	Restaurant, Taxi, Fun Place	0.25
Clinic	ATM, Pharmacy	0.3
Freeway	Petrol Kiosk, Direction Guide	0.3
	Restaurant, McDonald's	0.1
Airport	Taxi, Hotel	0.3
Residential Area	Taxi, Restaurant, Nearest Friend, Fun Place, Technicians	0.1
Mass Rapid Transit (MRT)	Nearest Friend, McDonald's	0.1
Office Building	Restaurant, MRT, Taxi, Client's Office, Technicians	0.1

The data generation proceeds as follows. We use 100 service request types, each of which occurs with a probability ranging from 0.1 to 0.7. We use a correlation coefficient to indicate the percentage of service requests of a given type that are close to a geographical feature. This coefficient E varies from 0 to 70. For each service request type, $E\%$ of its requests are generated close to a geographical feature based on Gaussian distribution. The locations of these requests are varied according to a deviation parameter $V = 50$ from the centroid of the geographical feature. The remaining $1 - E\%$ requests are generated using uniform distribution. A total of 100,000 requests is generated.

We use the Singapore map as our geographical feature database which contains 50 types of geographical features. Table 3.3 shows some of the interesting patterns we find with the support counts. When we show both sets of patterns to decision makers, they

Table 3.3: Interesting patterns found

Topological Patterns
{ATM, Pharmacy, Fun Place, 2761}
{Restaurant, Petrol Kiosk, Direction Guide, 1734}
{**Taxi**, ATM, Shopping Mall, Hairdressing Salon, 1845}
{Client's Office, Direction Guide, Restaurant, 3421}
{ATM, **Taxi**, Restaurant, Client's Office, 2142}
{Restaurant, Fun Place, **Taxi**, Hotel, 1421}
Geographical-based Topological Patterns
{{Clinic}::{ATM, Pharmacy}, 1323}
{{Shopping Complex, Hotel}::{Restaurant, Fun Place}, 1123}
{{Freeway Exit}::{Petrol Kiosk, Direction Guide}, 2312}
{{Airport}::{**Taxi**, Hotel}, 889}

unanimously prefer the geographical-based topological patterns.

3.5.7 Comparative Study on Finding Clique Patterns

In this experiment, we compare the TopologyMiner algorithm with the FastMining algorithm on mining clique patterns. We extend the algorithm FastMining [ZMCS04] to find the clique patterns in three dimensions and realize it according to [ZMCS04]. We implement the FastMining as a memory-based algorithm by dividing the space and feature sets using a regular grid. To identify the frequent clique patterns, we maintain the maximal clique patterns in the memory and mark the corresponding patterns and all their subpatterns, such that each distinct patterns where o_i takes part is marked only once.

Figure 3.14 shows the results for the synthetic dataset by varying the distance relation R. Compare to the FastMining algorithm, TopologyMiner needs less time and space to find the frequent topological patterns. This is expected because the set of fre-

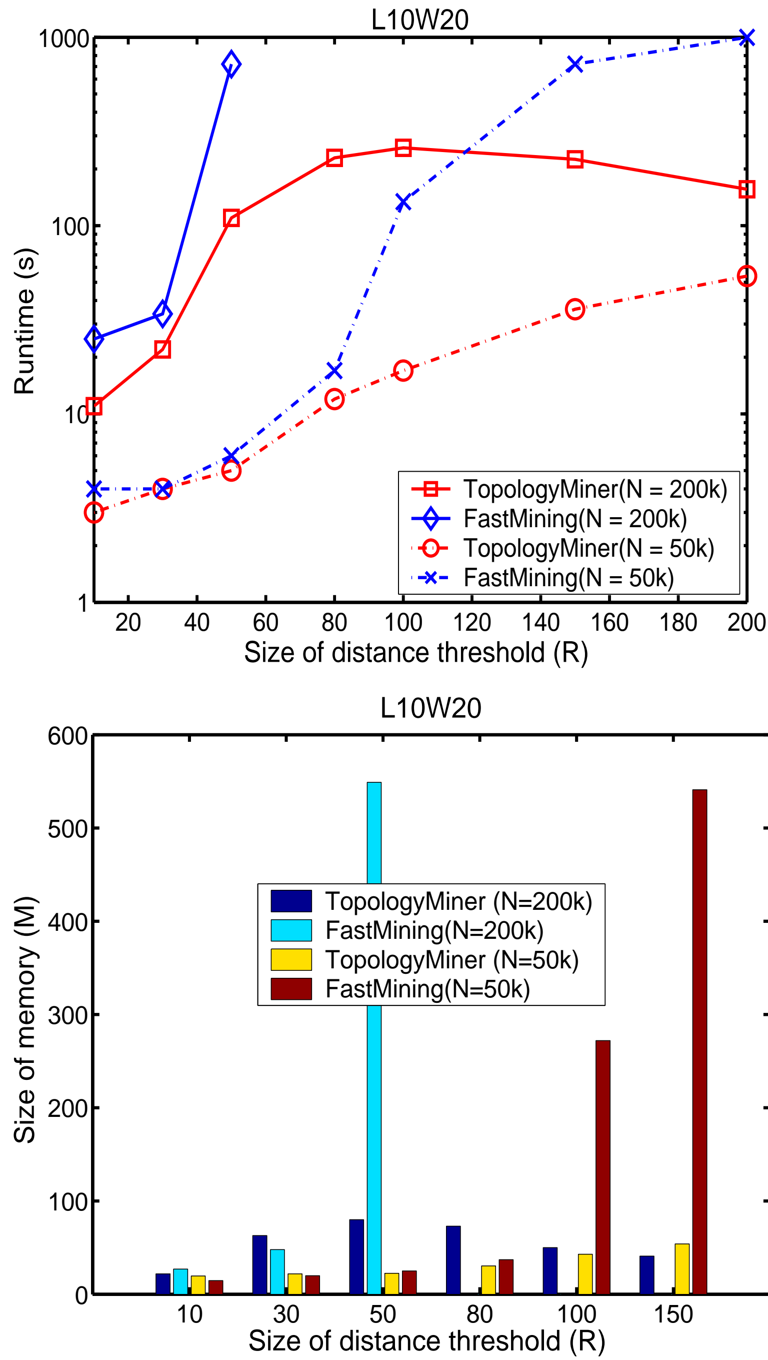


Figure 3.14: Runtime vs. the distance relation (clique patterns)

quent topological patterns becomes larger and the length of the frequent patterns tends to be longer as the distance threshold increases. As a result, FastMining algorithm needs more time to enumerate and computer the instances of topological patterns and more space to maintain the candidates and the instances of the topological patterns. In contrast, TopologyMiner uses the summary structure to approximate instances' count of a topological pattern so as to eliminate the generation and computation of the instances. Note that in the Figure 3.14(b), FastMining algorithm is out of memory when R is greater than 50.

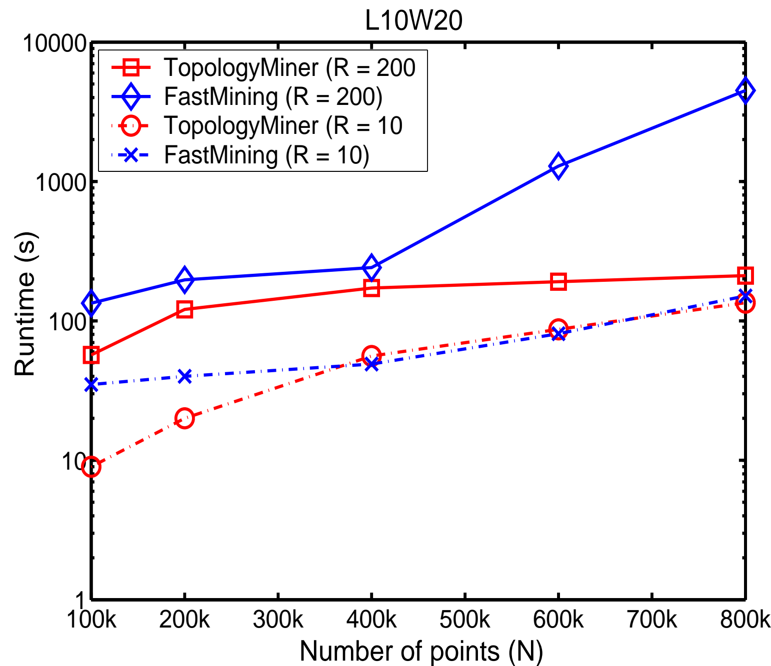


Figure 3.15: Runtime vs. number of points (clique patterns)

Figure 3.15 shows the results by varying the number of points in the database. From the figure, we observe that TopologyMiner needs less time to find the frequent topological patterns, compared to the FastMining algorithm. Since FastMining attempts to

count at one scan the instances of the powerset of all possible patterns, the computation cost tends to be much higher as the number of points increases, especially when the total number of features is large.

3.6 Summary

In the chapter, we have introduced an algorithm, TopologyMiner, for finding topological patterns in spatio-temporal databases. We have presented a summary-structure that summarizes spatio-temporal databases by recording instances' count information of a feature in a cube. Based on the summary-structure, TopologyMiner finds topological patterns in a depth-first manner and eliminates the generation of too many candidates and frequency tests. We have also studied the problem of finding the geographical features of topological patterns. The experimental studies indicate that TopologyMiner is effective and scalable in finding topological patterns and outperforms Apriori-like algorithms by a few orders of magnitude. Moreover, compared to topological patterns, geographical-based topological patterns are more informative.

Chapter 4

Mining Spatial Sequence Patterns

Besides topological patterns, another class of useful spatio-temporal patterns is spatial sequence patterns. Unlike topological patterns which aim to find the intra-relationships of events in a time window, spatial sequence patterns aim to disclose the inter-relationships of events in different time windows. For example: "*Forest fire always occurs at region* R_1 *prior to the occurrence of haze in nearby region* R_2." or "*Forest fire always occurs at a region prior to the occurrence of haze in its Northeastern nearby regions.*" In the above example, clearly the spatial sequence patterns are more informative and useful as they disclose both the spatial and temporal relationships of the events *fire* and *haze*. Moreover, they also link the event *fire* at R_1 to the event *haze* in R_2, which cannot be obtained by spatial patterns, temporal patterns or topological patterns. However, discovering such spatial sequence patterns is challenging because of the potentially large search space and the large number of candidates. This calls for new data mining algorithms.

In this chapter, we study the problem of finding spatial sequence patterns by incorporating spatial information into the process for mining sequence patterns. We introduce two new classes of spatial sequence patterns: *flow patterns* and *generalized spatio-temporal patterns*, which link the changes of events in one location to another location in order to reveal insights that cannot be obtained otherwise. We design an algorithm called *FlowMiner* to find flow patterns. FlowMiner incorporates a new candidate generation algorithm and employs various optimization techniques for better efficiency. We also propose an algorithm called GenSTMiner to discover generalized spatio-temporal patterns by exploring the pattern growth approach. We also present two optimization techniques to enhance the efficiency of GenSTMiner.

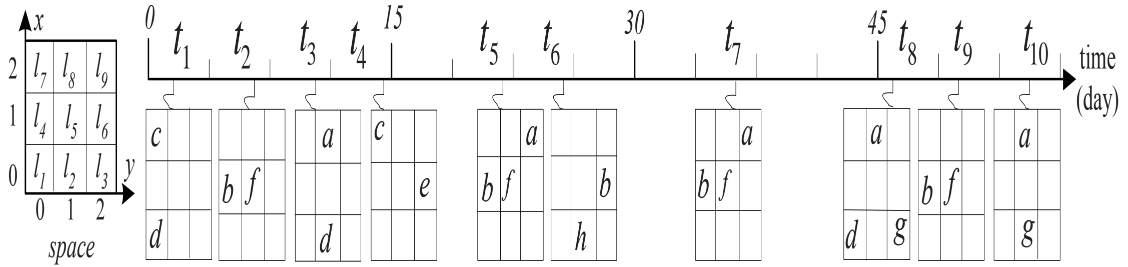
The chapter is organized as follows. We first present the framework of spatio-temporal databases and the preliminary concepts in Section 4.1. Section 4.2 illustrates the concept of flow patterns and the algorithm FlowMiner. Section 4.3 introduces the concept of generalized spatio-temporal patterns and illustrates the algorithm GenST-Miner. We summarize the chapter in Section 4.4.

4.1 Framework of Spatio-temporal Databases

Spatio-temporal databases capture both the time and space dimensions. First, we divide time into disjoint time windows of length W. Each time window denotes a time period. Time t_1 and t_2 are said to be near if they are in the same time period.

Next, we divide the space into a set of disjoint grid cells, $\mathcal{S} = \{l_1, l_2, \ldots, l_q\}$, where

each grid cell represents a location, denoted as $l_i = (x, y)$. Let R be a *neighbor relation* over the locations in $\mathcal{S}$. Location l_1 and l_2 are said to be neighbors if $(l_1, l_2) \in R$. The *neighborhood* of a location l is defined as a set of locations $N(l)$ = $\{l_1, \ldots, l_k\}$ such that each l_j in $N(l)$ is a neighbor of l.



(a) Space-time view

window id (wid)	time	eventsets
1	t_1	$d(l_1)$, $c(l_7)$
	t_2	$b(l_4)$, $f(l_5)$
	t_3	$d(l_2)$, $a(l_8)$
	t_4	$e(l_6)$, $c(l_7)$
2	t_5	$b(l_4), f(l_5), a(l_9)$
	t_6	$h(l_2), b(l_6)$
3	t_7	$b(l_4)$, $f(l_5)$, $a(l_9)$
4	t_8	$d(l_1)$, $g(l_3)$, $a(l_8)$
	t_9	$b(l_4)$, $f(l_5)$
	t_{10}	$g(l_2)$, $a(l_8)$

(b) Dataset sorted by window id and time

wid	sequences
1	$\langle d(l_1), c(l_7)\rangle \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow \langle d(l_2), a(l_8)\rangle \rightarrow \langle e(l_6), c(l_7)\rangle$
2	$\langle b(l_4), f(l_5), a(l_9)\rangle \rightarrow \langle h(l_2), b(l_6)\rangle$
3	$\langle b(l_4), f(l_5), a(l_9)\rangle$
4	$\langle d(l_1), g(l_3), a(l_8)\rangle \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow \langle g(l_2), a(l_8)\rangle$

(c) Sequences for time windows

Figure 4.1: Example of a spatio-temporal database

A *location-based event*, or *event* for short, denoted as $e(l, t)$, is a spatial feature e (such as drought, rain) occurring in location l at time t. Two events $e_1(l_1, t_1)$ and

$e_2(l_2, t_2)$, $t_1 \leq t_2$, are said to be *related* or *CloseNeighbor*s if and only if $(l_1, l_2) \in R$ and t_1 is near t_2. For convenience, we simply write the location-based event as $e(l)$ when the sequential context is clear.

A set of location-based events that occur at the same time is called an *eventset*, denoted as $E = \langle e_1(l_1), \ldots, e_m(l_m) \rangle$. Two eventsets E_1 and E_2 are said to be *CloseNeighbor*s if and only if every event in E_1 is related to every event in E_2. An eventset E_p at time t_1 is said to *flow to* an eventset E_q at time t_2, $t_1 \leq t_2$, if and only if E_p and E_q are CloseNeighbors. We denote it as $E_p \rightarrow E_q$. In addition, an eventset E_t is said to be *reflexive* if and only if E_t flows to itself.

Figure 4.1 shows an example of a spatio-temporal database, where time is divided into four time windows (i.e., W = 15 days), and space is divided into nine locations, and the literals $\{a, b, ..., h\}$ represent some spatial features.

Suppose R is the unit length of a square and $R = 1$, then we could see that the two events $d(l_1, t_1)$ and $b(l_4, t_2)$ are CloseNeighbors, that is $d(l_1, t_1)$ and $b(l_4, t_2)$ are related. The eventset $E_1 = \langle b(l_4), f(l_5) \rangle$ at time t_2 and the eventset $E_2 = \langle d(l_2), a(l_8) \rangle$ at time t_3 are CloseNeighbors. Moreover, E_1 flows to E_2. In particular, the eventset $\langle b(l_4), f(l_5) \rangle$ at time t_2 is a reflexive eventset, but the eventset $\langle d(l_2), a(l_8) \rangle$ is not since $(l_2, l_8) \notin R$.

A *sequence* is a list of eventsets sorted by time within a time window. Figure 4.1(c) shows an example.

4.1.1 Interesting Patterns in Spatio-temporal Databases

In this section, we present various sequence patterns that can be found in spatio-temporal databases using existing data mining techniques. Suppose we are given a spatio-temporal database as in Figure 4.1, we can observe at least three types of sequence patterns:

1. **Global sequence patterns**

   A global sequence, $s = \{(s_1 \rightarrow \ldots \rightarrow s_t) :: (L)\}$, where s_i $(1 \leq i \leq t)$ is a set of spatial features and L is a set of locations and $L \subseteq \mathcal{S}$, is a frequently occurring pattern if there are at least sup different locations containing s.

2. **Local sequence patterns**

   A local sequence is a sequence $s = \{(s_1 \rightarrow \cdots \rightarrow s_t) :: (l_k)\}$, where s_i $(1 \leq i \leq t)$ is a set of spatial features and $l_k \in \mathcal{S}$. Given a time window with width W, we say that s is frequent if there are at least sup different windows at location l_k containing s.

3. **Location-sensitive sequence patterns**

   A location-sensitive sequence pattern is a list of events sorted by time, denoted as $s = (E_1 \rightarrow \cdots \rightarrow E_t)$, where E_i $(1 \leq i \leq t)$ is an eventset. Given a time window with width W, s is said to be frequent if there are at least sup different windows containing s.

We can discover these three types of sequence patterns with existing association rule mining techniques or sequence mining techniques.

4.2 FlowMiner: Finding Flow Patterns in Spatio-temporal Databases

We have presented three types of sequence patterns that can be discovered in spatio-temporal databases. Although these three types of sequence patterns can reveal some interesting information of the events, that is the spatial relationships or the temporal relationships of the events, none of them can be used to link the changes in one location to the changes in a nearby location. In this section, we introduce the concept of flow patterns that are intended to describe the changes of events over space and time. We design an algorithm, called *FlowMiner*, which utilizes temporal relationships and spatial relationships amid events to generate flow patterns.

4.2.1 Problem Statement

The concept of the *flow pattern* is defined as follows:

Definition 1 *(***Flow Pattern***)*

A flow pattern *is a sequence of* reflexive *eventsets sorted by time such that for any two consecutive eventsets, E_p at time t_i and E_q at time t_{i+1}, E_p flows to E_q.*

Consider Figure 4.1. To simplify discussion, we define R as the unit length of a square. Let R = 1, then $d(l_1) \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow a(l_8)$ at time window 1 is a flow pattern, but $\langle b(l_4), f(l_5), a(l_9)\rangle$ at time window 3 is not since $(l_4, l_9) \notin R$.

We limit flow patterns to reflexive eventsets to provide a more meaningful interpretation of the patterns discovered. This is because a reflexive eventset guarantees that all

the events within the set are related to each other. For example, in Figure 4.2, $d(l_2)$ and $a(l_8)$ are not related in the pattern $d(l_1) \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow \langle d(l_2), a(l_8)\rangle \rightarrow e(l_6)$. Thus, we will consider $d(l_1) \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow a(l_8) \rightarrow e(l_6)$, and $d(l_1) \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow d(l_2) \rightarrow e(l_6)$ as two independent patterns since they indicate two opposite trends.

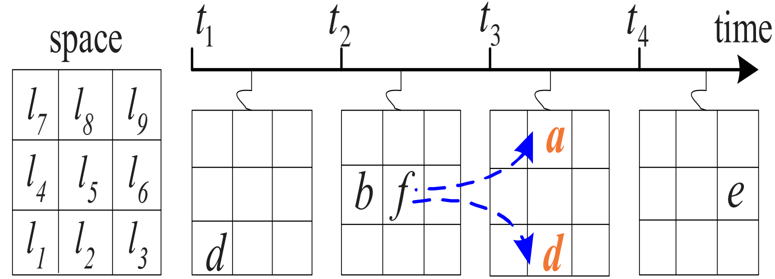


Figure 4.2: Example of flow patterns

A flow pattern with k events is called k-$flow$. A k-$flow$ pattern is frequent if there are at least $minsup$ different occurrences of the pattern over time, where $minsup$ is a user-specified threshold. Let $P = E_{P_1} \rightarrow \cdots \rightarrow E_{P_s}$ and $Q = E_{Q_1} \rightarrow \cdots \rightarrow E_{Q_t}$ be two flow patterns. P is called a *sub-flow* of Q, and Q a *super-flow* of P, denoted as $P \sqsubseteq Q$, if there exist integers $1 \leq j_1 < j_2 < \cdots < j_n \leq t$ such that $E_{P_1} \subseteq E_{Q_{j_1}}$, $E_{P_2} \subseteq E_{Q_{j_2}}, \ldots, E_{P_s} \subseteq E_{Q_{j_n}}$. A flow pattern, P, is *maximal* if there does not exist any flow pattern Q such that $P \sqsubseteq Q$.

Lemma 1 *Flow patterns satisfy Apriori property: Any sub-flow of a frequent flow pattern must be frequent.*

Proof: The set of flow patterns is a subset of sequences that satisfy the additional neighborhood constraint. We know that if a sequence is frequent, all its subsequences

must be frequent. Since the set of flow patterns is a subset of sequences, we conclude that flow patterns also satisfy the Apriori property.

With the above definitions, we can now define the problem to find flow patterns as follows: *Given a spatio-temporal database $\mathcal{D}$, a temporal window of length W, a neighbor relation R, and a user specified threshold $minsup$, the problem of mining flow patterns in spatio-temporal databases is equivalent to finding the set of all frequent flow patterns.*

In the following parts, we will discuss the process of discovering flow patterns. The first step in the mining process is to scan the database to find all frequent events (i.e., 1-$flow$s). These events are sorted according to their support in descending order. Next, based on the sorted event order, we proceed from left-to-right to find all frequent length-2 sequences. Following that, we mine the frequent k-$flow$s ($k > 2$) in a depth-first manner. This involves two main sub-tasks: candidate generation and support counting. The pruning techniques are presented in Section 4.2.4

4.2.2 Candidates Generation

A key observation in mining flow patterns is that a length-2 sequence specifies a temporal relationship that must be maintained in the higher-order sequences.

Let $\{d(l_1) \rightarrow b(l_4)\}$, $\{d(l_1) \rightarrow f(l_5)\}$ and $\{d(l_1) \rightarrow a(l_8)\}$ be three frequent length-2 sequences. Suppose we want to extend $\langle b(l_4), f(l_5)\rangle \rightarrow a(l_8)$ by inserting event $d(l_1)$. An enumeration-based candidate generation method will generate five length-4 sequences as shown in Figure 4.3 (column 1). Note that an eventset corresponds to only

enumerated candidates	length-2 sequences			neighborhood
	$\{d(l_1) \rightarrow b(l_4)\}$	$\{d(l_1) \rightarrow f(l_5)\}$	$\{d(l_1) \rightarrow a(l_8)\}$	constraints
$d(l_1) \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow a(l_8)$	$\surd$	$\surd$	$\surd$	$\surd$
$\langle b(l_4), d(l_1), f(l_5)\rangle \rightarrow a(l_8)$	$\times$	$\times$	$\surd$	$\times$
$\langle b(l_4), f(l_5)\rangle \rightarrow d(l_1) \rightarrow a(l_8)$	$\times$	$\times$	$\surd$	$\times$
$\langle b(l_4), f(l_5)\rangle \rightarrow \langle a(l_8), d(l_1)\rangle$	$\times$	$\times$	$\times$	$\times$
$\langle b(l_4), f(l_5)\rangle \rightarrow a(l_8) \rightarrow d(l_1)$	$\times$	$\times$	$\times$	$\times$

Figure 4.3: Candidates validation with length-2 sequences and neighborhood constraints

one insert position because adding an event in different positions of an eventset only indicates the same fact that all the events occur at the same time. We assume that the events in an eventset are sorted alphabetically.

However, if we take into consideration the temporal constraints implied by the frequent length-2 sequences, then it is clear that $d(l_1)$ can only be inserted into $\langle b(l_4), f(l_5)\rangle \rightarrow a(l_8)$ at the position before $\langle b(l_4), f(l_5)\rangle$ in order to generate valid sequence candidates, i.e., $d(l_1) \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow a(l_8)$. Moreover, with the neighborhood constraints of flow patterns, we can further remove sequence candidates which are not flow patterns.

The above example indicates that it is possible to avoid generating infrequent and invalid candidate flow patterns by taking into consideration the temporal relationships specified by length-2 sequences and the spatial constraints specified by flow patterns.

Summary Tree

We introduce a *summary tree* to keep track of all frequent flow patterns that have been generated and to capture the temporal relationships of length-2 sequences.

The structure of a summary tree is defined as follows:

1. A *root* node, at level 0 of the tree, is denoted as $null$. This node has no incoming edge, and corresponds to the initial state.

2. Each node n at level k, consists of *a set of frequent k-flows*, and is associated with an *extension set*, denoted as $Ext(n)$, which stores events that can be combined with k-flows to form the children of node n.

Figure 4.4 shows the summary tree that has been constructed from the dataset in Figure 4.1(b) with $sup = 50\%$, $W = 15days$ and R denoting a square of unit length 1. Node 0 is the root node and its extension set consists of all the frequent events, i.e., $\{b(l_4), f(l_5), a(l_8), a(l_9), d(l_1)\}$. The nodes at level 1 in the tree are the frequent events that are 1-$flow$s. A frequent event is included in the extension set of node n if this event occurs on the right of node n in the tree.

The children of a level k node n, $(k \geq 1)$, are generated by combining all frequent flow patterns in node n with the events in $Ext(n)$. For example, the children of node 1, i.e., node 6, node 7, node 8, node 9 and node 10, are generated by combining 1-$flow$ $b(l_4)$ with its extension set elements $b(l_4)$, $f(l_5)$, $a(l_8)$, $a(l_9)$ and $d(l_1)$ respectively.

Algorithm

When we have found all the frequent events and length-2 sequences (i.e., level 1 and 2 nodes), we move on to the next step. Based on the level 1 and level 2 nodes in the summary tree, we can now construct the level k nodes, $k > 2$, by extending the level $k-1$ nodes with extension elements. This consists of four main steps which we will

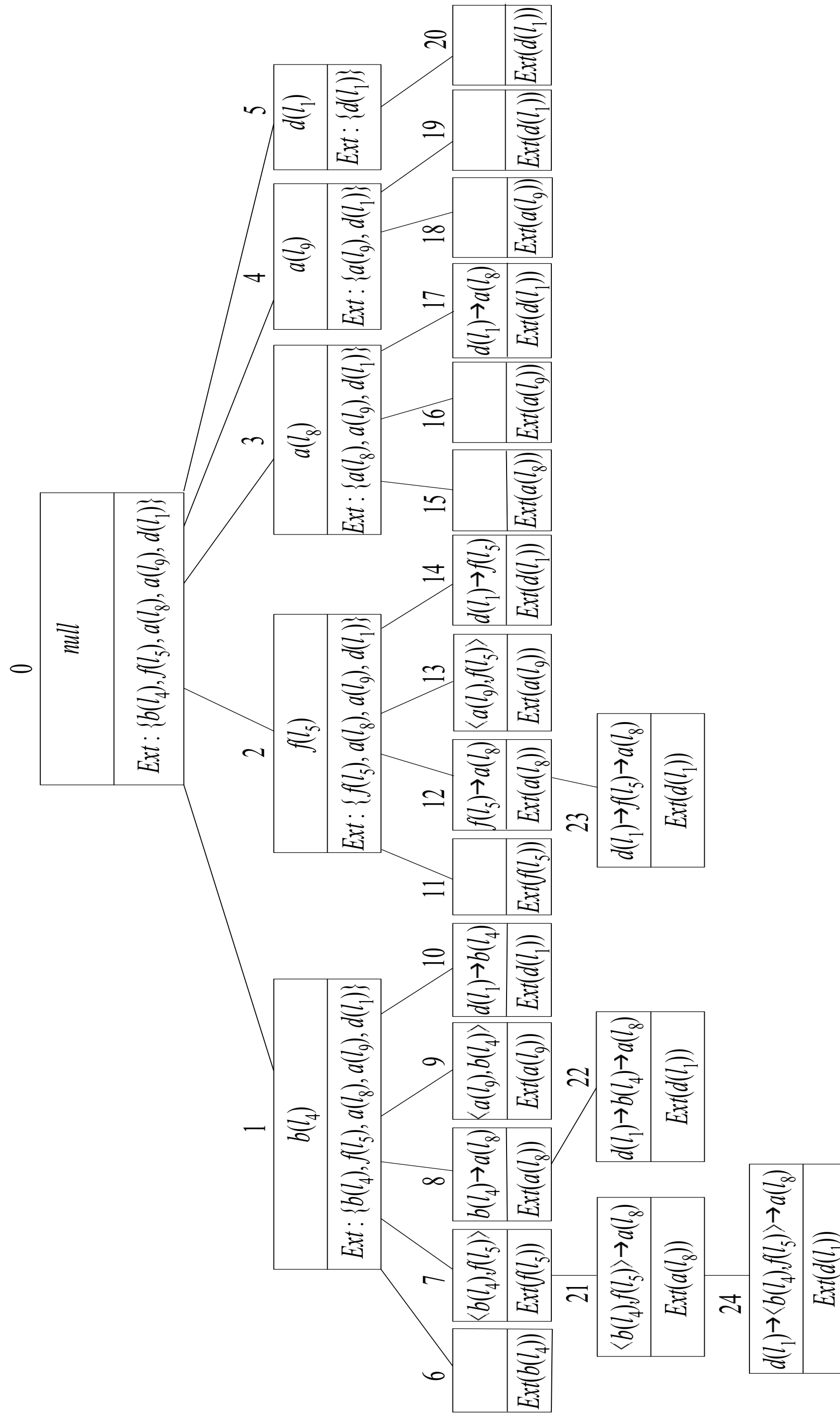


Figure 4.4: Summary tree for the dataset in Figure 4.1

illustrate using node 24 in Figure 4.4.

Step 1. *Determine Relevant Temporal Constraints*

When extending a node n at level $k-1$ with the extension element β, we need to limit the number of sequences generated by eliminating infeasible sequences through the use of relevant temporal constraints. These temporal constraints are in the form of length-2 sequences that involve events in node n and the extension element β.

In Figure 4.4, node 24 is generated by extending node 21 with $d(l_1)$. The events in node 21 are $\{b(l_4), f(l_5), a(l_8)\}$. Hence, the relevant temporal constraints are those length-2 sequences involving an event in $\{b(l_4), f(l_5), a(l_8)\}$ with $d(l_1)$. They are contained in nodes 10, 14 and 17.

Step 2. *Find Feasible Insert Positions Based on Temporal Constraints*

Given a (k-1)-$flow$ consisting of t eventsets $\alpha_1 \rightarrow \cdots \rightarrow \alpha_t$ where $\alpha_i = \langle e_{i_1}, e_{i_2}, ...e_{i_m} \rangle$ ($1 \leq i \leq t$) (e_{i_j}, $1 \leq j \leq m$, is an event) and an extension element β, there is a total of $2t + 1$ insert positions in which the extension element β can be inserted as an explicit eventset or as an element of the eventset α_i ($1 \leq i \leq t$) to form a k-$flow$ (see Figure 4.5).

Not all the $2t + 1$ insert positions are feasible. To determine the feasible insert positions, we use the relevant length-2 sequences to determine the set of insert positions that do not violate the corresponding temporal constraints. Let us examine how the temporal relationships of length-2 sequences can be used to obtain the feasible insert positions.

A careful study reveals that there are seven ways in which a temporal constraint can affect the insert positions. Figure 4.5 summarizes the seven cases. Note that the position of β relative to e_{i_j} is the position of β relative to eventset α_i where $e_{i_j} \in \alpha_i$. Since an eventset corresponds to an insert position, we say that the temporal relationship between the events e_{i_j} and β is also the temporal relationship between the eventset α_i and β.

Case 1: *β occurs at the same time as e_{i_j}.*

In this case, we only have one possible insert position. That is, β must be inserted at the same position as e_{i_j}.

Case 2: *β occurs before e_{i_j}.*

Here, all the insert positions before e_{i_j} are feasible. In other words, we have $2i-1$ insert positions.

Case 3: *β occurs after e_{i_j}.*

This is similar to Case 2 except in this case, all the insert positions after e_{i_j} are feasible. In total, there are $2(t-i)+1$ insert positions.

Case 4: *β occurs before or at the same time as e_{i_j}.*

This is a combination of Cases 1 and 2. In this case, there are $2i$ possible insert positions for β to be inserted into α.

Case 5: *β occurs at the same time as or after e_{i_j}.*

This is a combination of Cases 1 and 3. In this case, we have a total of $2(t-i)+2$ insert positions.

$(k-1)-$ **flow** : $\alpha_1 \to \cdots \to \alpha_i \to \cdots \to \alpha_t$ **with** β

insert positions: 1 2 3 2i-1 2i 2i+1 2t-1 2t 2t+1

Cases	Node[e_{i_j}+β]	k-$flows$
1	$\{\langle e_{i_j}, \beta\rangle\}$	$\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i\beta \to \ldots \to \alpha_t$
2	$\{\beta \to e_{i_j}\}$	$2i-1$: $\beta \to \alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t$ $\alpha_1\beta \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to \ldots \to \beta \to \alpha_i \to \ldots \to \alpha_t$
3	$\{e_{i_j} \to \beta\}$	$2(t-i)+1$: $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \beta \to \ldots \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \beta \to \alpha_t$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t\beta$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \cdots \to \alpha_t \to \beta$
4	$\langle e_{i_j}, \beta\rangle$ $\beta \to e_{i_j}$	$2i$: $\beta \to \alpha_1 \to \alpha_2 \to .... \to \alpha_i \to ... \to \alpha_t$ $\alpha_1\beta \to \alpha_2 \to ... \to \alpha_i \to ... \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to ... \to \beta \to \alpha_i \to ... \to \alpha_t$ $\alpha_1 \to \alpha_2 \to ... \to \alpha_i\beta \to ... \to \alpha_t$
5	$\langle e_{i_j}, \beta\rangle$ $e_{i_j} \to \beta$	$2(t-i)+2$: $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i\beta \to \ldots \to \alpha_t$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \beta \to \ldots \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \beta \to \alpha_t$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t\beta$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t \to \beta$
6	$\beta \to e_{i_j}$ $e_{i_j} \to \beta$	$2t$: $\beta \to \alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t$ $\alpha_1\beta \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to \ldots \to \beta \to \alpha_i \to \ldots \to \alpha_t$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \beta \to \ldots \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \beta \to \alpha_t$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t\beta$ $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_i \to \ldots \to \alpha_t \to \beta$
7	$\langle e_{i_j}, \beta\rangle$ $\beta \to e_{i_j}$ $e_{i_j} \to \beta$	$2t+1$: $\beta \to \alpha_1 \to \alpha_2 \to ... \to \alpha_i \to ... \to \alpha_t$ $\alpha_1\beta \to \alpha_2 \to ... \to \alpha_i \to ... \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to ... \to \beta \to \alpha_i \to ... \to \alpha_t$ $\alpha_1 \to \alpha_2 \to ... \to \alpha_i\beta \to ... \to \alpha_t$ $\vdots$ $\alpha_1 \to \alpha_2 \to ... \to \alpha_i \to ... \to \beta \to \alpha_t$ $\alpha_1 \to \alpha_2 \to ... \to \alpha_i \to ... \to \alpha_t\beta$ $\alpha_1 \to \alpha_2 \to ... \to \alpha_i \to ... \to \alpha_t \to \beta$

Figure 4.5: Temporal relationships of length-2 sequences

Case 6: *β occurs before or after e_{i_j}.*

This is a combination of Cases 2 and 3. It has $2t$ insert positions.

Case 7: *Combination of Cases 1, 2 and 3.*

In this case, none of the insert positions can be eliminated, and we have to generate all $2t+1$ sequences (see Figure 4.5).

After Step 1, nodes 10, 14 and 17 will store the relevant length-2 sequences for node 24. The summary tree will capture the corresponding temporal constraints. Since these constraints fall under Case 2, we can determine the feasible insert positions of event $d(l_1)$ in node 21 (see column 2 in Figure 4.6).

3-$flow$ $\langle b(l_4), f(l_5)\rangle \rightarrow a(l_8)$ with ext $d(l_1)$		
Length 2-sequences	insert position by temporal constraints	insert position by spatial constraints
Node 10 = $\{d(l_1) \rightarrow b(l_4)\}$	1	1
Node 14 = $\{d(l_1) \rightarrow f(l_5)\}$	1	1
Node 17 = $\{d(l_1) \rightarrow a(l_8)\}$	1, 2, 3	1

Figure 4.6: Example of insert positions

Step 3. *Reduce Feasible Insert Positions Based on Spatial Constraints*

Having decided on the insert positions based on temporal constraints, we can further optimize the set of insert positions based on the spatial constraint specified by flow patterns. This is realized by considering the neighborhood constraints between the extension element and the corresponding eventsets.

Let p be the position in α where event β may be inserted, $1 \leq p \leq 2t+1$. The position p is said to be a valid insert position if:

1. β is inserted into α as an explicit eventset such that $(\beta, \alpha_{\lfloor \frac{p}{2} \rfloor}) \in R$ and $(\beta, \alpha_{\lceil \frac{p}{2} \rceil}) \in R$ hold, or;

2. β is inserted into α as an element of the eventset $\alpha_{\lfloor \frac{p}{2} \rfloor}$ where $(\beta, \alpha_{\lfloor \frac{p}{2} \rfloor}) \in R$, $(\beta, \alpha_{\lfloor \frac{p}{2} \rfloor - 1}) \in R$ and $(\beta, \alpha_{\lfloor \frac{p}{2} \rfloor + 1}) \in R$ hold.

Insert positions that do not satisfy the above two conditions can be removed. Figure 4.6 (column 3) shows the final insert positions obtained for node 24.

Step 4. *Generate New Flow Patterns*

Step 3 yields a list of possible insert positions in which an extension element β can be inserted into an existing (k-1)-$flow$ to form a new k-$flow$. If a (k-1)-$flow$ includes m unique events, then there are m level 2 nodes that can be used to decide m sets of feasible insert positions in α. The actual insert positions are determined by finding the intersection of these m sets of feasible insert positions. This process is repeated for a node until all the (k-1)-$flow$s in the node have been examined.

In our running example, we have obtained the insert positions based on three nodes: nodes 10, 14 and 17. The intersection of these insert positions results in only one final insert position $\{1\}$, i.e., a new 4-$flow$ $d(l_1) \rightarrow \langle b(l_4), f(l_5) \rangle \rightarrow a(l_8)$ being generated as node 24. Since all the 3-$flow$s in node 21 have been checked, the candidate generation process for node 24 terminates.

Figure 4.7 shows the candidate generation algorithm. Its input includes node N at level $k-1$, a upper triangle Tbl, where we store the case number of length-2 sequences, and the extension element β. The child of the node N, N' (i.e. $Node[N + ext]$) at level

Procedure FlowCandGen

Input: N: the node at level k-1 in the Summary tree;
β: extension element;
Tbl: the cases table of two events

Output: N': the child of the node N at level k

1: $C_k = \emptyset$;
2: $Cases$ = $\{Tbl[e_{i_j}][\beta] | e_{i_j} \in N\}$;
3: **for each** (k-1)-$flow$ $\alpha \in N$ {
4: **for each** $c_i \in Cases$ {
5: pos_i is the set of insert positions imposed by c_i;
6: $finalpos_i = \emptyset$;
7: **for each** $p \in pos_i$ {
8: **if** (((β is an explicit eventset) && $(\beta, \alpha_{\lfloor \frac{p}{2} \rfloor}) \in R$ && $(\beta, \alpha_{\lceil \frac{p}{2} \rceil}) \in R$)
|| ((β is an element of $\alpha_{\lfloor \frac{p}{2} \rfloor}$) && $(\beta, \alpha_{\lfloor \frac{p}{2} \rfloor}) \in R$&& $(ext, \alpha_{\lfloor \frac{p}{2} \rfloor - 1}) \in R$
&& $(ext, \alpha_{\lfloor \frac{p}{2} \rfloor + 1}) \in R$)) {
9: $finalpos_i$ = $finalpos_i \cup \{p\}$;
10: }
11: }
12: $FinalPos = \bigcap_i finalpos_i$;
13: }
14: C_k=$C_k \bigcup \{k$-$flow$s generated using $FinalPos\}$;
15: }
16: **return** C_k

Figure 4.7: Procedure of candidate generation

$k+1$, is the output. Initially, the $Cases$ of the relevant length-2 sequences are obtained from the Tbl (line 2). Then, we find the initial insert positions using the $Cases$ for each (k-1)-$flow$ in node N (lines 4-5), and further optimize the insert positions using the neighborhood constraints at lines 6-9. The final positions are decided at line 12. The process is continued until all (k-1)-$flow$s in node N are examined. Finally, the k-$flow$s in node N' are generated.

4.2.3 Support Counting

Having generated the candidate patterns, we need to determine the frequencies of these candidates. Our algorithm makes use of a hash tree. Each node in the hash tree is associated with a hash table, where items of a candidate are hashed via some standard hash function. Each entry of the hash table is a list of $(item, pointer)$, where $item$ denotes the item that has been hashed to this entry, and $pointer$ points to the node containing the next item in the candidate.

When we add a candidate, we start from the root and descend the hash tree. At each depth p of the interior node, we apply the hash function to the pth item of the candidate and insert the corresponding $(item, pointer)$ to the hash entry. The depth of the root node is 1 and the node at depth p points to the node at depth $p+1$. Figure 4.8 shows an example of the construction of a hash tree for flow patterns.

In conventional algorithms [AS96], the hash tree is built for the candidates of each level k, i.e., the leaf nodes are of the same depth. In our case, flow patterns of different lengths are found simultaneously. To allow for this difference in lengths, we augment

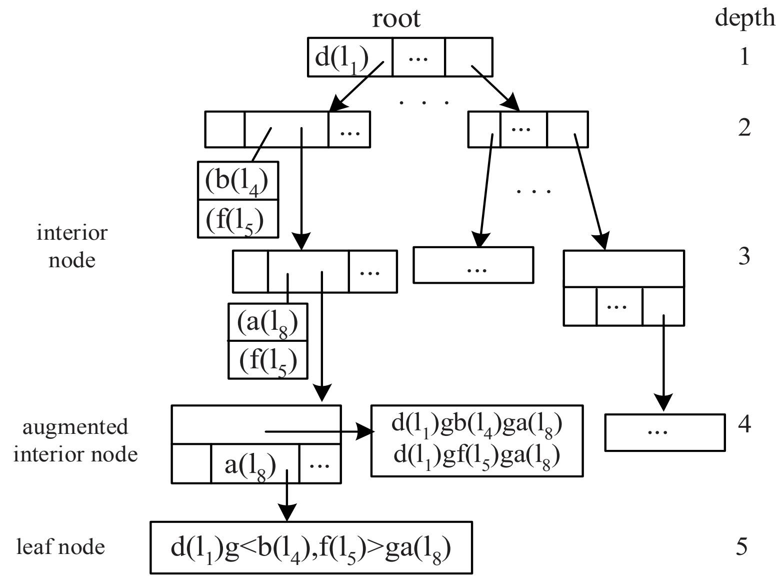


Figure 4.8: Hash tree for varying flow patterns length

some interior nodes of the hash tree with a list to store flow patterns. Figure 4.8 shows a sample hash tree with two different flow pattern lengths.

After constructing the hash tree for all the candidate flow patterns, we scan the database to count the number of occurrences of these candidate flow patterns. For each sequence found in the database, we check whether the sequence S is a superflow of any of the candidate flow patterns as follows:

For each event in S, we start at the root node, and recursively apply the hash function to determine the corresponding hash table entry with a pointer to the child node. Once we reach a leaf node or an augmented node, we check if S is a superflow of one of the flow patterns stored in the leaf node/augmented node. If it is, we increment the support count of the corresponding flow pattern.

4.2.4 Pruning Techniques

In this section, we discuss various optimization techniques that are used to improve the efficiency of FlowMiner.

Prune Infrequent Candidates

The summary tree is constructed in a depth-first manner. As we expand a node to generate its children, some of the children may turn out to be unpromising as none of their flow patterns are frequent. Thus, we can immediately prune off their descendants.

For example, if node 10 in Figure 4.4 is infrequent, that is to say the combination of events $b(l_4)$ and $d(l_1)$ is infrequent. Hence, all the nodes in the summary tree that contain the two events $b(l_4)$ and $d(l_1)$ can be pruned, e.g., node 22 and node 24. This is achieved by removing $d(l_1)$ from the extension set of node 1.

To realize this pruning strategy efficiently, we associate a vertical bitmap with each node n in the summary tree, denoted as $bitmap(n)$. Each bit in the vertical bitmap indicates the occurrence of flow patterns in node n in each time window in the database, that is, if a flow pattern α appears in time window j, then bit j of $bitmap(n)$ is set to one; otherwise, it is set to zero.

With the vertical bitmap, we can quickly eliminate the unpromising nodes from being generated by updating the appropriate extension set. When generating a new node by extending node n with the extension item β, we only need to check whether there is $minsup$ ones in the result bitmap of $bitmap(n) \cap bitmap(\beta)$. If the sum of ones is less than $minsup$, then we can conclude that the combination of events in node

n with β is infrequent. Hence, we can eliminate β from subsequent extensions of node n.

Eliminate Hashing Non-promising Events

Another optimization lies in the support counting process. We observe that certain events do not need to be hashed in the hash tree because they cannot be flow patterns, and will not be able to contribute to the support counts of any of the flow patterns in the hash tree. Suppose we have $S = \langle d(l_1), g(l_3), a(l_8)\rangle \rightarrow \langle b(l_4), f(l_5)\rangle \rightarrow \langle g(l_2), a(l_8)\rangle$ in Figure 4.1. It is clear that $g(l_3), a(l_8)$ are not related to $d(l_1)$, i.e., their locations are not neighbors of each other. Thus, we do not need to search the corresponding hash tree entries for them since there is no flow pattern involving these events. Based on this observation, we introduce a check prior to an event being hashed to ensure that this event satisfies both the reflexive and flow conditions in the flow pattern definition.

Delay Database Scans

A database scan is typically needed to determine whether a flow pattern α is frequent. However, if we know that a superset of α is frequent, then we can immediately conclude that α is frequent. This allows us to delay database scan, which ultimately minimizes the number of database scans needed.

To achieve this, we store a list of the maximal frequent flow patterns, called *Max-Filter* in the main memory. When a new node n is generated, we first check whether a flow pattern in n is a $sub\text{-}flow$ of some of the flow patterns in MaxFilter. If it is, then it

is frequent and we keep it in node n; otherwise, instead of scanning the database immediately to decide its frequency, we keep its pointer in a list. A database scan is carried out only when the number of flow patterns in the list exceeds the memory threshold, or all descendants of node n have been generated.

In real-life applications, we may have thousands of maximal frequent flow patterns which makes the $sub\text{-}flow$ check expensive. In order to reduce the cost, instead of storing all the maximal flows in MaxFilter, we only store the maximal sets that can potentially be supersets of candidates generated from node N. We call this list the *local maximal frequent flow patterns,* $LMFS_N$. In this way, we can eliminate many comparisons with the maximal sets that are not super- or sub- flows of node N.

4.2.5 FlowMiner Algorithm

Figure 4.9 shows the framework of FlowMiner. The database $\mathcal{D}$ is sorted first by time, then by location. Lines 2-3 find all the frequent 1-$flow$s and extend them to 2-$flow$s in a left-to-right order. Lines 6-11 call the procedure DFS-PathScan to generate k-$flow$s ($k > 2$).

DFS-PathScan (see Figure 4.10) generates nodes in a depth-first manner: Lines 2-3 generate node N's child N' and prune it using $LMFN_N$. We delay the counting of flow patterns and minimize the number of database scans needed at Lines 5-14. Line 15 updates $LMFS_{N'}$ with those maximal sets in $LMFN_N$ who contains all the events in node N'. Then we prune those unpromising extension events in $Ext(N')$ at line 16 accordingly. Finally, we call DFS_PathScan recursively to generate N' descendants

Algorithm FlowMiner

Input: $\mathcal{D}$, Database;
 R, spatial relation;
 W, temporal relation;
 $minsup$, minimum support;
Output: M, set of frequent flow patterns
1: $M = \emptyset$;
2: F_1=$\{$all frequent location-based events (i.e. 1-$flows$)$\}$;
3: F_2=$\{$all frequent 2-$flows$$\}$;
4: Filling the Tbl with frequent length-2 sequences;
5: $i = 1$;
6: **for each** level 2 node N and $N \neq \emptyset$ {
7: $LMFN_N = \emptyset$;
8: F_i = DFS-PathScan(N,$LMFN_N$,Tbl,$minsup$);
9: $M = M \cup F_i$;
10: $i++$;
11: }
12: **Answer** = M

Figure 4.9: Framework of the FlowMiner algorithm

Procedure DFS-PathScan(N, $LMFS_N, Tbl, minsup$)

0: Pre-condition: $F = \emptyset$
1: **for each** $ext_i \in Ext(N)$ {
2: N' = **FlowCandGen**(N, ext_i, Tbl);
3: Prune N' using $LMFS_N$;
4: $LMFS_{N'} = \emptyset$;
5: **if** (k-$flows$ in N' need to count)**then**
6: Add these k-$flows$ to $path$;
7: **if** (k-flows in $path \geq$ memory threshold)**then**
8: Scan database for $path$;
9: **for each** $\alpha \in path$ {
10: **if** $(sup(\alpha) \geq minsup)$ **then**
11: $F = F \cup \{\alpha\}$;
12: **if** $(\exists \beta \in N, \beta \preceq \alpha)$ **then**
13: Update $LMFS_N$ s.t. $\{\gamma \in LMFS_N | \gamma \npreceq \alpha, \alpha \npreceq \gamma\}$;
14: }
15: $LMFS_{N'} = LMFS_{N'} \cup \{\alpha \in LMFS_N | ext_i \in \alpha\}$;
16: $Ext(N')$ = **Ext-combine**(N', ext_i, N, $minsup$);
17: **if** there are frequent/uncertain $flows$ in N' **then**
18: DFS-PathScan(N', $LMFS_{N'}$, Tbl, $minsup$);
19: }
20: **return** F;

Procedure Ext-combine(N'**,**ext_i**,** N**,** $minsup$**)**

1: C = $\emptyset$
2: **for each** $ext_j \in Ext(ext_i)$ {
3: **if** (($ext_j \in Ext(N)$)
&& ($bitmap(ext_j) \cap bitmap(N') \geq minsup$)) **then**
4: $C = C \cup ext_j$;
5: }
6: **return** C

Figure 4.10: Optimized algorithm

at lines 17-18. This process is continued until all the nodes in the summary tree are generated or no additional frequent flow patterns are found.

4.2.6 Performance Study

In this section, we present the results of experiments to evaluate the effectiveness and efficiency of FlowMiner on both synthetic and real-life datasets. The experiments are carried out on a Pentium 4, 1.6 GHZ processor with 256MB memory running Windows XP. The algorithm is implemented in C++.

Table 4.1: Parameters

Par.s	Meaning	Range
$\|D\|$	$\sharp$ of time windows($\times$ 10,000)	1,2,4,6,8,10
$\|C\|$	Avg. $\sharp$ of eventsets per time window (W)	5,10,15,20
$\|T\|$	Avg. $\sharp$ of events per Eventsets	2,4,6,8

- *Synthetic Dataset.* We augment the Quest synthetic dataset generator in [AS95] to include spatial information by generating N item using F spatial features and L locations. Our synthetic datasets are generated by setting N=10,000, F=1000 and L=100. Other parameters used are listed in Table 4.1.

- *Real-life Datasets.*

  1. *Meteorological dataset.* We retrieve two years' worth of standard meteorological data for eight closely located stations from the Nation Data Buoy Center[1]. The data consists of 10 continuous features being recorded at an

[1] http://www.ndbc.noaa.gov/rmd.shtml

hourly interval. We first discretize the features and then divide the space into grids for the locations to distribute uniformly.

2. *Forest Fire dataset.* Two years of forest fire satellite images, which include 2,495,097 forest fire occurrences, are obtained from a remote imaging center. We divide the region into 49 grids whereby each region is 10 degrees in the longitudinal direction and 10 degrees in the latitudinal direction.

The characteristics of the datasets are shown in Table 4.2.

Table 4.2: Real-life dataset characteristics

Dataset	♯ loc.	♯ features.	avg.len.of eventsets	♯ eventsets
Meteorological	8	30	20	17520
Forest Fire	49	10	13	16650

Experiments on Synthetic Dataset

We first examine how the various parameters listed in Table 4.1 affect the performance of FlowMiner. Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14 show the results for the synthetic dataset. In general, we observe that the runtime of FlowMiner increases when the minimum support is small. This is because many flow patterns become frequent and the length of the frequent flow patterns tends to be long when minimum support is small.

Figure 4.11 shows the effect when the parameter C (i.e., time window length W) varies from 5 to 20. The runtime of FlowMiner grows as C increases. This is expected as an increase in time window length implies a longer data sequence, which in turn

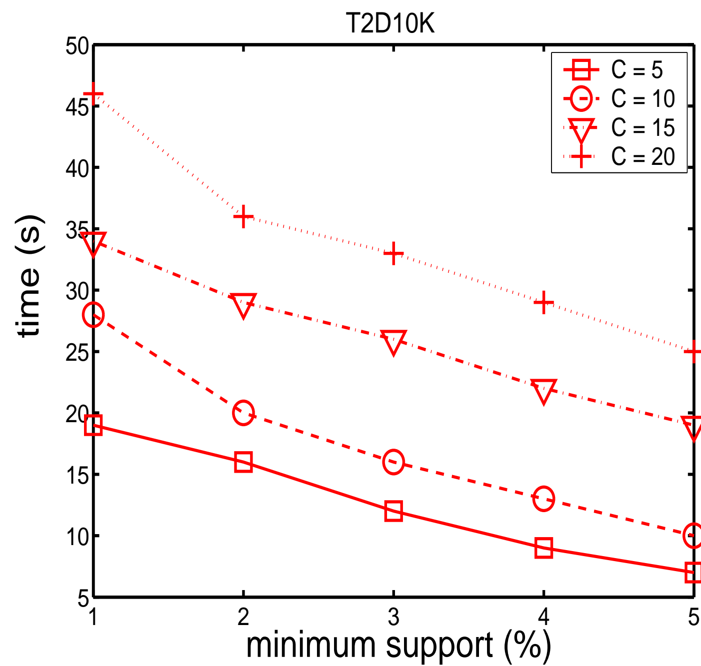


Figure 4.11: Varying parameter C (synthetic dataset)

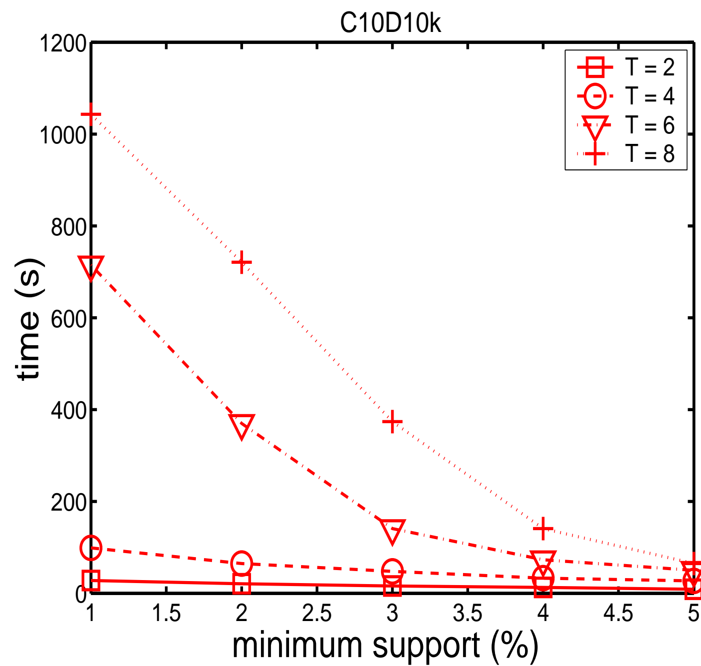


Figure 4.12: Varying parameter T (synthetic dataset)

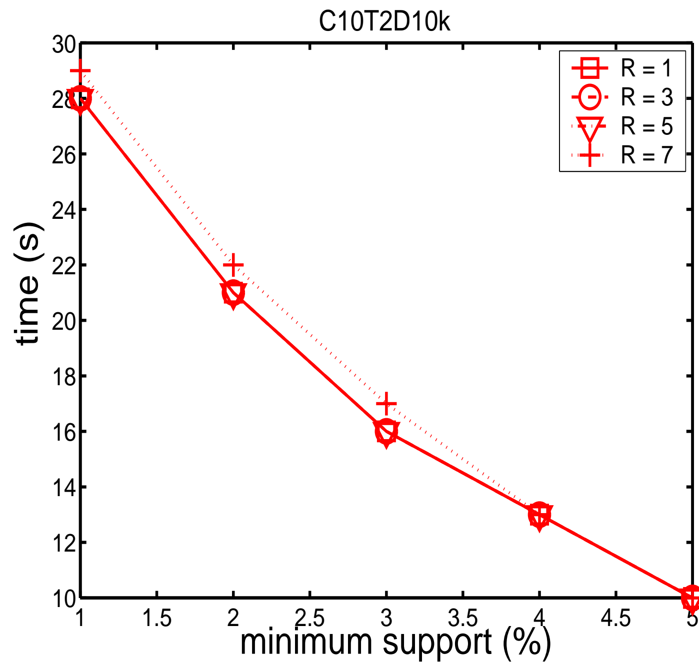


Figure 4.13: Varying parameter R (synthetic dataset)

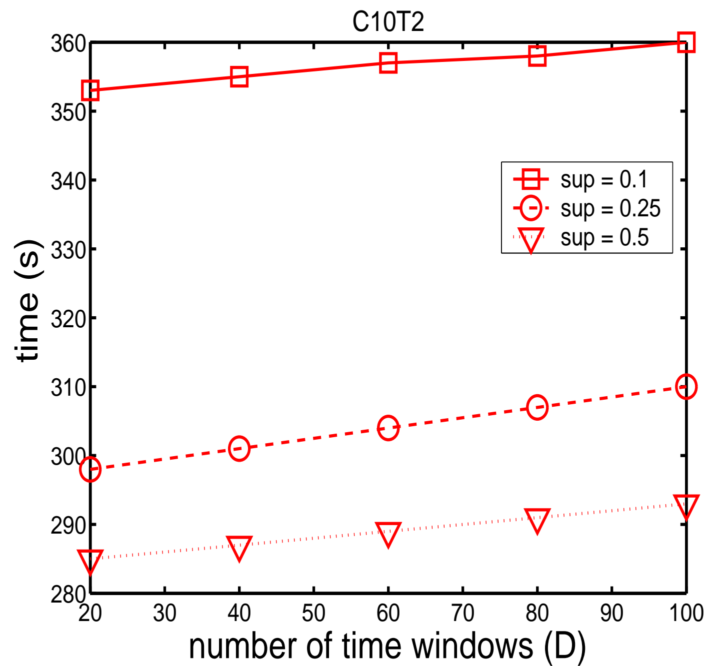


Figure 4.14: Varying parameter D (synthetic dataset)

implies longer flow patterns.

Figure 4.12 shows the runtime of FlowMiner for varying values of T. We observe that the runtime of FlowMiner grows as T increases because the size of an eventset and the length of the sequences in the databases become large. Hence, many flow patterns tend to be frequent and longer.

Figure 4.13 shows the performance of FlowMiner when the size of neighborhood relation R (i.e., length of a unit square) varies from 1 to 7. We observe that the runtime of FlowMiner remains almost constant, and grows slightly when R is 7. The explanation is: As the size of R increases, more events qualify as belonging to the spatial neighborhood of an event. As a result, the length of frequent flow patterns tends to increase, and the number and size of candidates grow rapidly.

Finally, we evaluate the performance of FlowMiner by varying the parameter D that is the number of time windows from 20,000 to 100,000. Figure 4.14 shows that the time taken by FlowMiner scales well with the increase in number of time windows.

Experiments on Real-life Datasets

Next, we examine the performance of FlowMiner on real-life datasets. Figure 4.15 shows the results on the meteorological dataset for time window length of 6, 9 and 12 time units respectively, Figure 4.16 shows the results when the spatial neighbor relation R is varied, and Figure 4.17 shows the results when the size of the database is varied. The results are consistent with those obtained on the synthetic dataset. Note that in Figure 4.17, the replication factor is a value to inflate the database size to test the

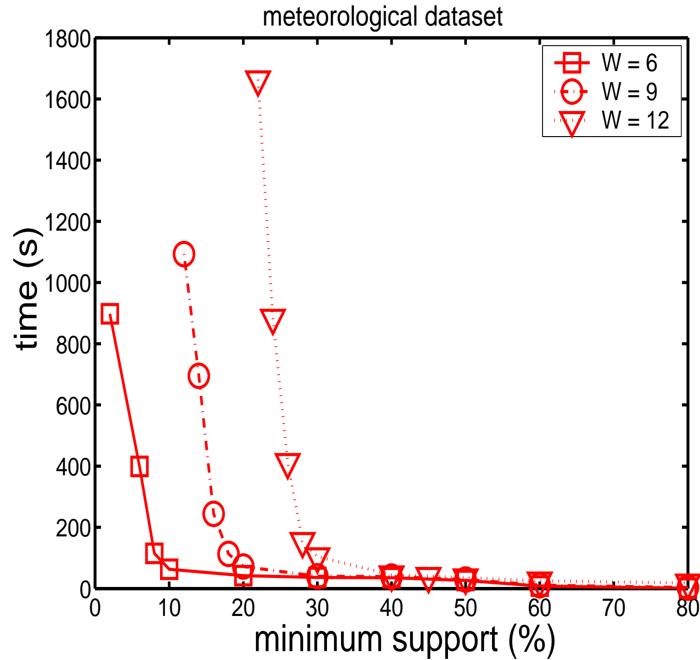


Figure 4.15: Runtime vs. parameter minsup (real-life dataset)

scalability of our algorithms.

We also discovered meaningful flow patterns in the real-life datasets. Figure 4.19 shows a sample of the flow patterns found in the forest fire dataset by setting the minimum support with the value 0.17%. There are in total 2908 frequent flow patterns found in the dataset. To identify the interesting frequent flow patterns, we cross-match the flow patterns with the corresponding weather maps. The corresponding interesting flow patterns are shown below the graphs. The events related to the $fire$ in the patterns are indicated as the rectangle floated in the space according to time, and the corresponding locations are indicated using the white color in the space. The fire spots depict two distinct spread patterns: the first is from West to East as shown in Figure 4.19(a), which occurs mostly in March and the beginning of April; and the second is from South to

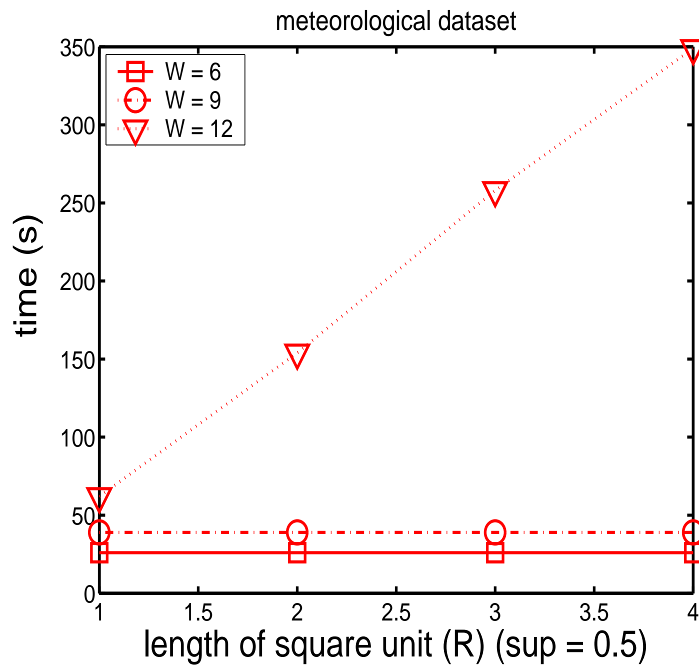


Figure 4.16: Runtime vs. spatial neighbor relation R (real-life dataset)

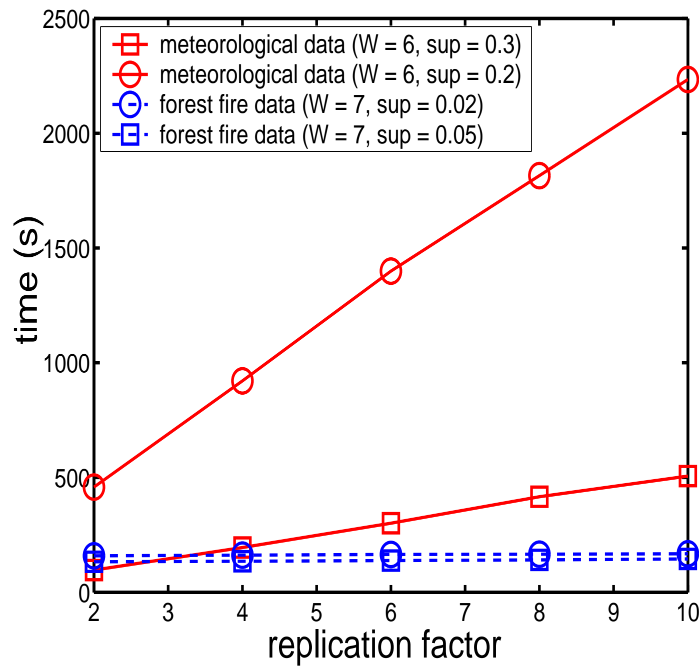


Figure 4.17: Scalability (real-life dataset)

Northwest as shown in Figure 4.19(b), which happens mostly in April and May.

Evaluation of Optimization Techniques

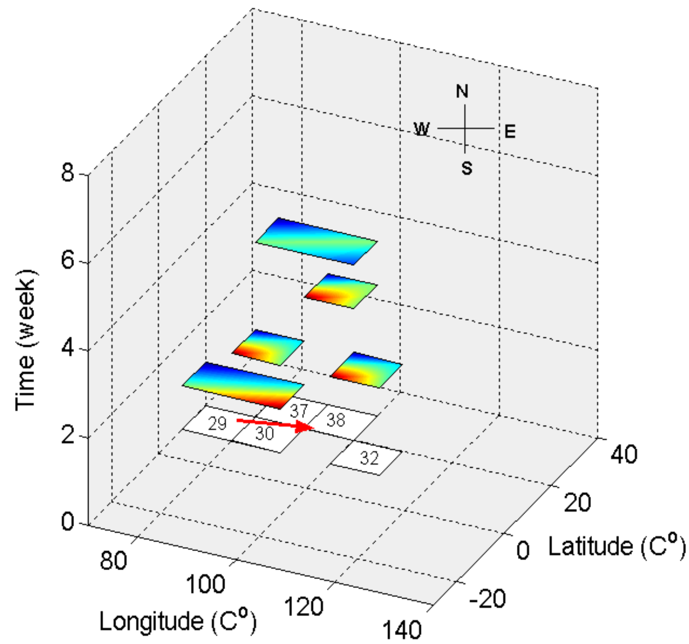
Finally, we investigate how the three optimization techniques, namely, prune infrequent candidates (Opt1), eliminate hashing non-promising events (Opt2), and delay database scan (Opt3) enhance the efficiency of FlowMiner. We use the synthetic dataset $C10T2D10K$, and set R to be a square of unit length 3 for this experiment.

Figure 4.20 shows that the greatest gain is obtained by delaying database scans. This is to be expected because the number of database scan plays an important role in the performance of FlowMiner.

Comparative Study

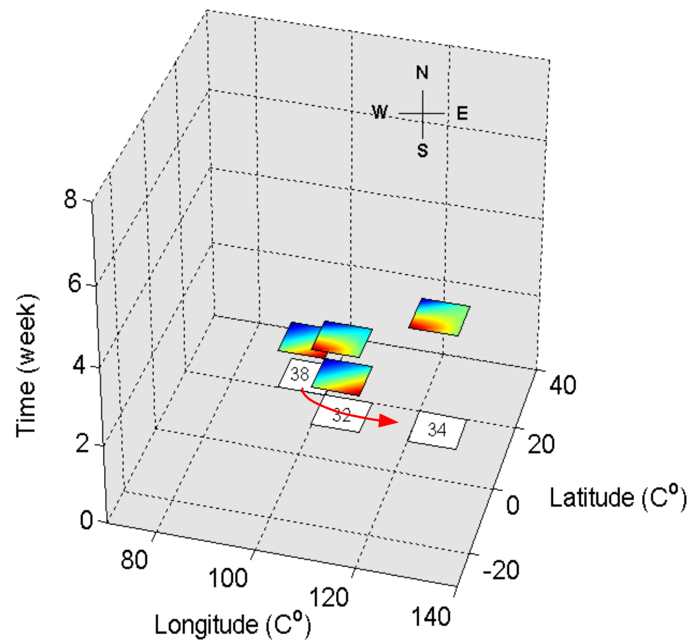
We observe that FlowMiner reduces to a sequence mining algorithm when the spatial relation R is the whole space. This allows us to compare FlowMiner with existing sequence mining algorithms. We compare it with GSP and $PrefixSpan$. We implement GSP and PrefixSpan according to [AS96, PHMAP01]. In this set of experiments, we compare FlowMiner algorithm with others for mining the complete set of sequences.

Figure 4.21 shows the results for the synthetic dataset $C10T2D10k$. Figure 4.21(a) gives the runtime when the minimum support is varied, and Figure 4.21(b) records the amount of memory used by the three algorithms. We observe that FlowMiner outperforms GSP. PrefixSpan outperforms FlowMiner when minimum support is large; but when minimum support is low, FlowMiner outperforms PrefixSpan. The amounts of



$$\langle F(l_{29}), F(l_{30})\rangle \rightarrow \langle F(l_{30}), F(l_{32})\rangle \rightarrow F(l_{38}) \rightarrow \langle F(l_{37}), F(l_{38})\rangle$$

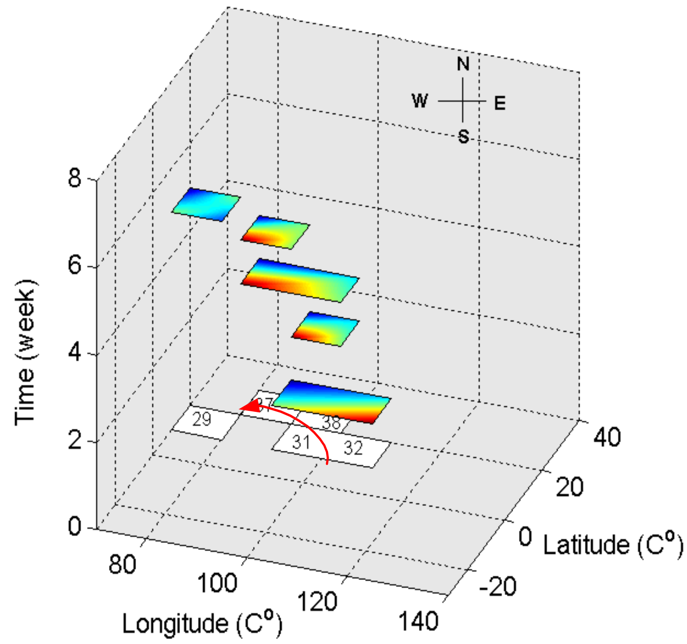
(a)



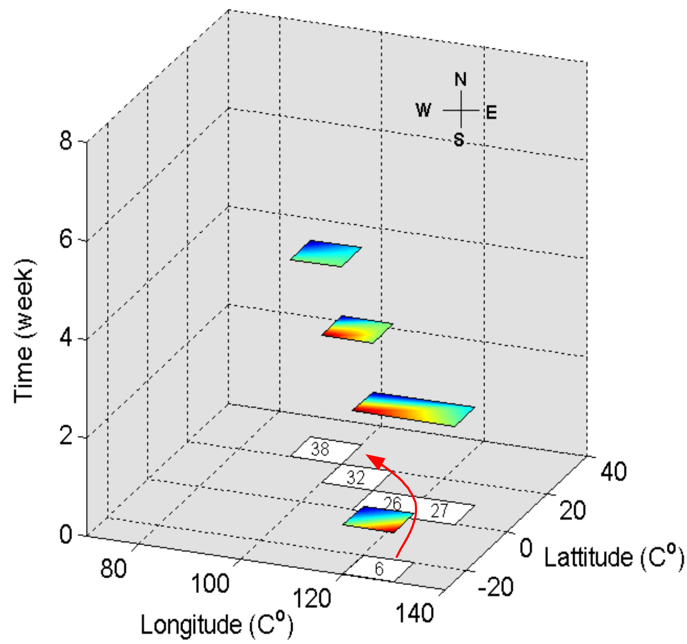
$$\langle F(l_{32}), F(l_{38})\rangle \rightarrow F(l_{32}) \rightarrow F(l_{34})$$

(b)

Figure 4.18: Flow patterns [Trend 1: from West to East in March and April]



$\langle F(l_{31}), F(l_{32})\rangle \rightarrow F(l_{38}) \rightarrow \langle F(l_{37}), F(l_{38})\rangle \rightarrow F(l_{37}) \rightarrow F(l_{29})$



$F(l_6) \rightarrow \langle F(l_{26}), F(l_{27})\rangle \rightarrow F(l_{32}) \rightarrow F(l_{38})$

(b)

Figure 4.19: Flow patterns [Trend 2: from South to Northwest in April and May]

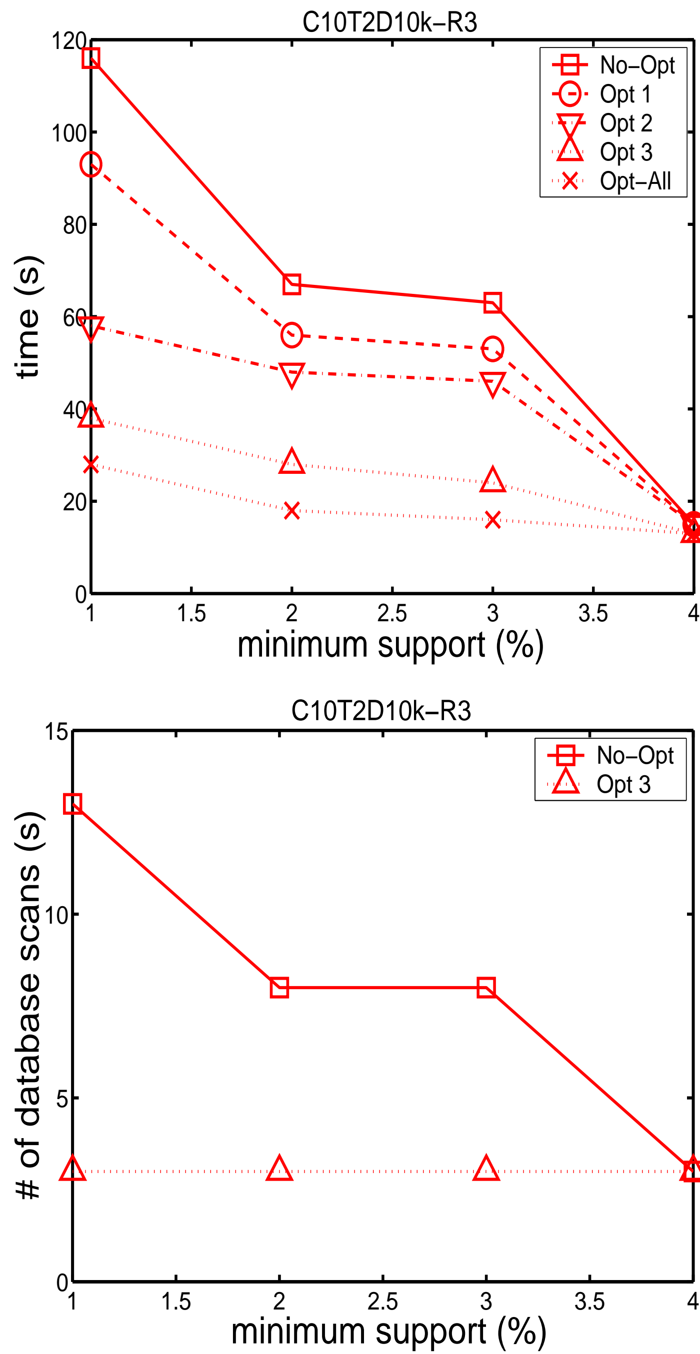


Figure 4.20: Effect of optimizations

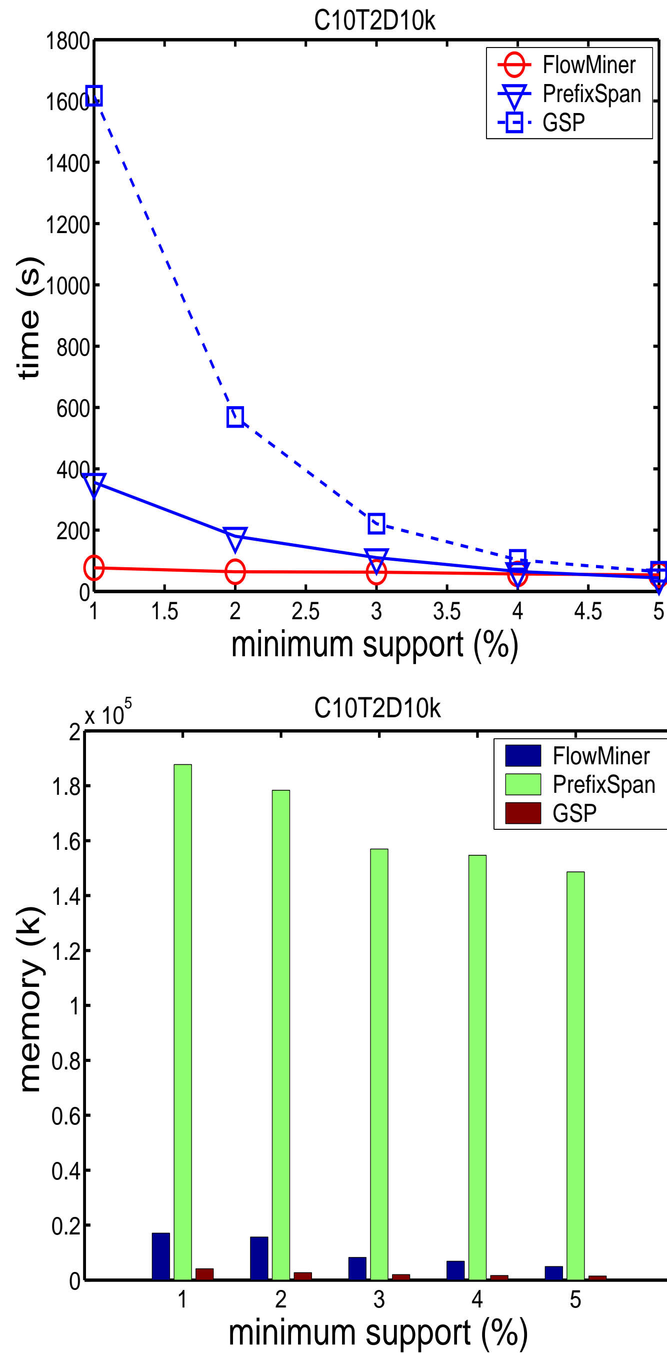


Figure 4.21: Comparative study (sequence patterns)

Table 4.3: Comparison of candidates generated

Number of Candidates Generated						
Algorithm	Support threshold					
	1	2	3	4	5	6
FlowMiner	15679	7573	3731	1832	891	428
FlowMiner (counted support)	**515**	**62**	**35**	**33**	**31**	**29**
GSP	15648	7543	3719	1820	879	416
enumeration (-based)	60926	30251	14962	7347	3582	1699

memory used by both FlowMiner and GSP are much smaller than that used by PrefixSpan.

The results confirm that FlowMiner is more scalable as compared to PrefixSpan. When minimum support is large, the set of frequent sequences is small. Hence, PrefixSpan runs slightly faster than FlowMiner. However, when minimum support is low, the set of frequent sequences becomes larger and the length of the frequent sequences tends to be longer. As a result, PrefixSpan needs more time and memory to obtain and store the large number of projection databases. In contrast, FlowMiner, with its small set of candidates generated and its optimization techniques, is able to save much time and memory. Hence, FlowMiner excels at finding frequent sequences when minimum support is small.

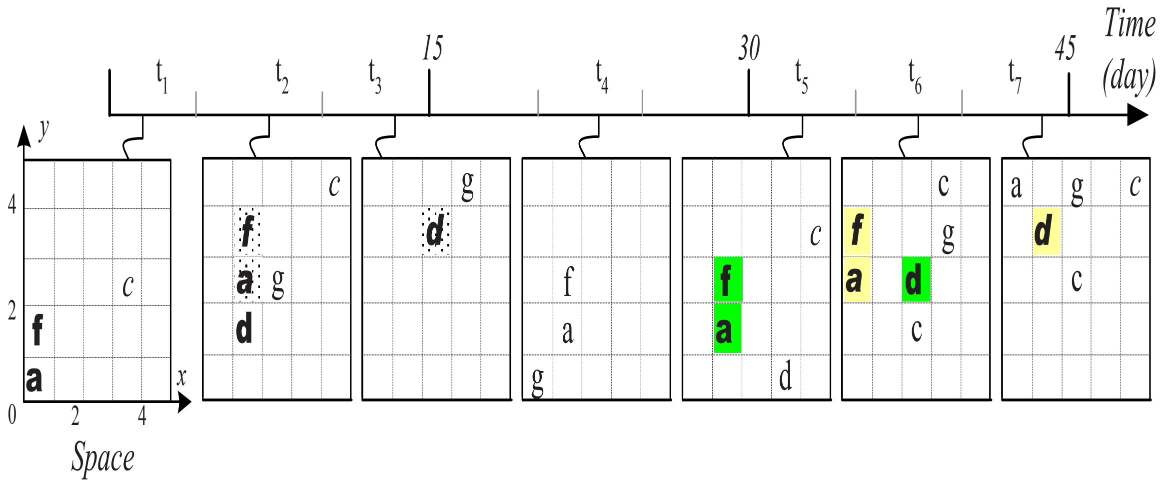
We also compare the candidate generation of FlowMiner with the Apriori-like candidate generation for BFS algorithms such as GSP[AS96], and the enumeration-based candidate generation for DFS algorithms such as DFS_Mine[TG01], SPADE[Zak98]. Table 4.3 shows the results based on the synthetic dataset $C10T2D10k$. We observe that the total number of candidates generated by FlowMiner is comparable to that by

GSP algorithm, but much fewer than that by the enumeration-based process. Moreover, we see that the size of candidates really needed for counting support in FlowMiner is much smaller. This is due to FlowMiner's adoption of a depth-first search strategy, which could find long frequent patterns quickly. Another reason is the further improvement realized through using the filter $LMFN$, which helps decide the frequency of candidates quickly.

4.3 GenSTMiner: Mining Generalized Spatio-temporal Patterns

In Section 4.2, we introduced the *flow patterns*, which aim to capture the evolution of events in neighboring regions over time. While flow patterns can clearly capture the flow of events to some degree, they rely heavily on the assumption that these events will repeat themselves in exactly the same locations. However, in some applications, the absolute locations in which an event e has occurred are not important. Rather, it is the relative locations of events with respect to event e that are interesting.

In this section, we introduce a new class of spatio-temporal patterns called *generalized spatio-temporal patterns* to summarize the sequential relationships between events that are prevalent in sharing the same topological structures. We adopt the pattern growth approach and develop an algorithm called GenSTMiner to discover generalized spatio-temporal patterns. To increase efficiency of the mining process, we also present two optimization techniques. The first is the use of *conditional projected databases* to



(a) Space-time view

Window ID	Time	Eventsets
1	t_1	$a(0,0), c(3,2), f(0,1)$
	t_2	$a(1,2), c(4,4), d(1,1), f(1,3), g(2,2)$
	t_3	$d(2,3), g(3,4)$
2	t_4	$a(1,1), f(1,2), g(0,0)$
3	t_5	$a(1,1), c(4,3), d(3,0), f(1,2)$
	t_6	$a(0,2), c(2,1), c(3,4), d(2,2), f(0,3), g(3,3)$
	t_7	$a(0,4), c(2,2), c(4,4), d(1,3), g(2,4)$

(b) Datasets of Events

Figure 4.22: Example spatio-temporal database (W = 15days, $R = 1$)

prune infeasible events and sequences, and the second is *pseudo projection* to reduce memory requirement.

4.3.1 Problem Statement

We introduced the framework of spatio-temporal databases in Section 4.1. Here, we assume that a location is denoted as $l = (x, y)$, and a location-based event as $e(x, y, t)$.

For example, Figure 4.22 shows an example of a spatio-temporal database which records the various locations where cyclones and storm occur over time. The space

(shown in Figure 4.22(b)) is partitioned into 25 disjoint locations, and the time is divided into three disjoint time windows. Figure 4.22(a) shows the events $\{a, b, c, d, etc\}$ that are observed at various locations over time.

Some sequences in Figure 4.22(a) that satisfy the *flow pattern* definition are as follows:

$$\langle a(0,0), f(0,1)\rangle \rightarrow d(1,1)$$
$$\langle a(1,2), f(1,3)\rangle \rightarrow d(2,3)$$
$$\langle a(1,1), f(1,2)\rangle \rightarrow d(2,2)$$
$$\langle a(0,2), f(0,3)\rangle \rightarrow d(1,3)$$

Each of the above flow patterns occurs only once and will be discarded by most mining algorithms. However, a closer examination reveals that these patterns actually convey some interesting behavior of the cyclones, i.e., *"Event a in an area that has been hit by the storm always leads to event f in its Northern neighbors and event d in its Northeastern neighbors*." In other words, the absolute locations in which event a has occurred are not important. Rather, it is the relative locations of event d or f with respect to event a that are interesting.

We have noted that relative addresses play an important role in capturing the invariant topological relationships of a pattern. In order to incorporate the concept of relative addresses, we first select a *reference location* denoted as $l_{ref} = (x_{ref}, y_{ref})$. We map each occurring event $e_1(x_1, y_1), e_2(x_2, y_2), \ldots, e_m(x_m, y_m)$ to its corresponding relative occurring location as $e_1(x_1 - x_{ref}, y_1 - y_{ref}), e_2(x_2 - x_{ref}, y_2 - y_{ref}), \ldots, e_m(x_m - x_{ref}, y_m - y_{ref})$.

A *RelativeEventset* is a set of mapped events that occur at the same time t, denoted as $\overrightarrow{E} = E\langle e_1(x_1 - x_{ref}, y_1 - y_{ref}), e_2(x_2 - x_{ref}, y_2 - y_{ref}), \ldots, e_m(x_m - x_{ref}, y_m - y_{ref})\rangle$. We assume that all events in a RelativeEventset are listed alphabetically. A RelativeEventset $\overrightarrow{E}_p$ is a *CloseNeighbor* of a RelativeEventset $\overrightarrow{E}_q$ if every event in $\overrightarrow{E}_p$ is a *CloseNeighbor* of every event in $\overrightarrow{E}_q$, denoted as $(\overrightarrow{E}_p, \overrightarrow{E}_q) \in (R, W)$.

Definition 2 *(***Generalized spatio-temporal pattern***)*

A generalized spatio-temporal pattern is a sequence of RelativeEventsets*, and all the RelativeEventsets are* CloseNeighbor*s of each other, denoted as,* $\overrightarrow{E}_1 \rightarrow \overrightarrow{E}_2 \rightarrow \cdots \rightarrow \overrightarrow{E}_m$*, s.t.* $\forall i, j \in (1..m), (\overrightarrow{E}_i, \overrightarrow{E}_j) \in (R, W)$.

Note that generalized spatio-temporal patterns can be specialized to spatial patterns and sequential patterns. When the space is reduced to a single location (i.e., $S \rightarrow 0$), the spatio-temporal pattern is simply the sequential pattern. On the other hand, if we limit the time window to a snapshot (i.e., $t \rightarrow 0$), we will have the co-located events among the spatial neighborhoods [SH01].

A generalized spatio-temporal pattern is said to be frequent if there are at least t-$minsup$ (i.e., temporal support) different occurrences of the pattern over time, and in each time window, there are at least s-$minsup$ (i.e., spatial support) patterns occurring in the space. A generalized spatio-temporal pattern involving k different events is called a k-generalized spatio-temporal pattern.

Given two generalized spatio-temporal patterns, $P = \overrightarrow{E}_1 \rightarrow \ldots \rightarrow \overrightarrow{E}_m$ and $Q = \overrightarrow{E}'_1 \rightarrow \ldots \rightarrow \overrightarrow{E}'_m$. Let P' be generated by concatenating P with Q, denoted as $P' = P \cdot Q$. P is called the *prefix* of P', and Q the *suffix* of P'. Q can be concatenated with P

in two ways, namely Q is an *eventset extension*, i.e. $\overrightarrow{E}_1 \to \ldots \to (\overrightarrow{\boldsymbol{E}}_m \cup \overrightarrow{\boldsymbol{E}}'_1) \to \overrightarrow{E}'_m$ or Q is a *sequence extension*, that is $\overrightarrow{E}_1 \to \ldots \to (\overrightarrow{\boldsymbol{E}}_m \to \overrightarrow{\boldsymbol{E}}'_1) \to \ldots \to \overrightarrow{E}'_m$.

Suppose there is a lexicographic ordering $\leq$ among the set of events in the spatio-temporal database. For example, given two events $e_1(x_1, y_1)$ and $e_2(x_2, y_2)$, $e_1(x_1, y_1) \leq e_2(x_2, y_2)$ if and only if (i) $e_1 \leq e_2$, or (ii) $e_1 = e_2, x_1 \leq x_2$, or (iii) $e_1 = e_2, x_1 = x_2, y_1 \leq y_2$.

In this section, we focus on finding frequent generalized spatio-temporal patterns by exploiting their similarity to sequence patterns. Among many sequence mining algorithms, we focus on the pattern growth method because it has been shown to be one of the most effective methods for frequent pattern mining and is superior to the candidate-maintenance-test approach, especially on a dense database or where there is low minimum support threshold [HP00]. As a divide-and-conquer method, the pattern growth method partitions the database into subsets recursively, but does not generate candidate sets. It also makes use of the Apriori property to prune the search space and count the frequent patterns in order to decide whether it can assemble longer patterns.

4.3.2 Projection-based Sequential Pattern Mining

We explain the pattern growth approach and illustrate the algorithm PrefixSpan[PHMAP01], which provides a general framework for the pattern growth method in Section 3.2. The basic idea of PrefixSpan is to use a set of locally frequent items to grow patterns.

We examine how PrefixSpan can be used to discover generalized spatio-temporal patterns. It first needs to find all frequent sequential patterns that satisfy $t\text{-}minsup$.

Next, it scans each time window and checks if there are $s\text{-}minsup$ spatial-sequences which are instances of frequent sequential patterns and all eventsets in a spatial-sequence are CloseNeighbors, and adds them into the candidate sets. Finally, all frequent generalized spatio-temporal patterns are obtained by mapping the spatial-sequences in the candidate sets into their relative addresses, where their support is larger than or equal to $s\text{-}minsup$.

Although PrefixSpan could find all the frequent generalized spatio-temporal patterns, it is neither efficient nor scalable as it needs to maintain a lot of sequence or spatial-sequence patterns in memory before it finds the final set of generalized spatio-temporal patterns. Further, it needs to scan the database more than twice.

4.3.3 GenSTMiner Algorithm

In this section, we describe an efficient algorithm called GenSTMiner that follows the framework of pattern growth methods and finds the complete set of generalized spatio-temporal patterns directly without maintaining a large number of candidates. We also devise optimization techniques that eliminate redundant candidates, and reduce the size of the projected database so that it fits into the memory. The GenSTMiner algorithm consists of the following three steps:

1. First, find the set of frequent events F_1 (1-general spatial-sequences) by scanning the database once, and sort them according to their lexicographic order.

2. Next, divide the set of frequent patterns into $|F_1|$ partitions and retrieve the pro-

jected database PDB_e of each event $e \in F_1$ from the database $\mathcal{D}$. Then, for each sequence in PDB_e, choose its reference location and map events in it into their relative locations. The transformed PDB_e is called the generalized projected database, denoted as GDB_e.

3. Finally, based on GDB_e, find all frequent k-generalized spatio-temporal patterns prefixed with e by constructing and mining the projected databases of the length-k generalized spatio-temporal patterns recursively.

The spatial support of an event in a time window can be determined by the number of different locations where it occurs, that is, an event is frequent in a time window if it occurs in at least $s\text{-}minsup$ locations. The temporal support of an event is decided by the number of different time windows where it is spatially frequent. Only when the temporal support of an event is larger than $t\text{-}minsup$, it is considered frequent.

Note that in the second step, we use all the instances of an event e regardless of their locations in an input sequence in the database to retrieve the projection database of the event e. For the same instances of the event e (same location), we only consider the first occurrence of it. This differs from PrefixSpan which considers only the first occurrence of the event e in an input sequence.

Consider Figure 4.22(c). Suppose $R = 1$, $W = 15days$, and the input sequence $s_1 = \langle \mathbf{a(0,0)}, c(3,2), f(0,1)\rangle \rightarrow \langle \mathbf{a(1,2)}, c(4,4), d(1,1), f(1,3), g(2,2)\rangle \rightarrow \langle d(2,3), g(3,4)\rangle$. Suppose we want to retrieve the projection of s_{1a}. Since there are two instances of a in s_1, namely $a(0,0)$ and $a(1,2)$, the projection of s_{1a} consists of two subsequences: $\langle \ddagger, c(3,2), f(0,1)\rangle \rightarrow \langle a(1,2), c(4,4), d(1,1), f(1,3), g(2,2)\rangle \rightarrow \langle d(2,3), g(3,4)\rangle$ and

Wid	Sid	Sequences	Prefix
1	1	$\langle \ddagger, c(3,2), f(0,1)\rangle \rightarrow \langle a(1,2), c(4,4), d(1,1), f(1,3), g(2,2)\rangle \rightarrow \langle d(2,3), g(3,4)\rangle$	$a(0,0)$
	2	$\langle \ddagger, c(4,4), d(1,1), f(1,3), g(2,2)\rangle \rightarrow \langle d(2,3), g(3,4)\rangle$	$a(1,2)$
2	1	$\langle \ddagger, f(1,2), g(0,0)\rangle$	$a(1,1)$
3	1	$\langle \ddagger, c(4,3), d(3,0), f(1,2)\rangle \rightarrow \langle a(0,2), c(2,1), c(3,4), d(2,2), f(0,3), g(3,3)\rangle \rightarrow \langle a(0,4), c(2,2), c(4,4), d(1,3), g(2,4)\rangle$	$a(1,1)$
	2	$\langle \ddagger, c(2,1), c(3,4), d(2,2), f(0,3), g(3,3)\rangle \rightarrow \langle a(0,4), c(2,2), c(4,4), d(1,3), g(2,4)\rangle$	$a(0,2)$
	3	$\langle \ddagger, c(2,2), c(4,4), d(1,3), g(2,4)\rangle$	$a(0,4)$

Figure 4.23: Projected database of event a

$\langle \ddagger, c(4,4), d(1,1), f(1,3), g(2,2)\rangle \rightarrow \langle d(2,3), g(3,4)\rangle$. Figure 4.23 shows the a-projected database obtained from Figure 4.22(c).

Having obtained the projected database of the frequent events, we need to choose the reference locations of the sequences, and then map events in a sequence into their relative locations. Here, the problem is how to choose the reference location.

Choice of Reference Location

We can either use the location of the event e or the base location of a sequence as the reference location. The base location of a sequence s is given by $\{(x,y)|\forall x_{i_j} \in s, y_{i_j} \in s, x = min(x_{i_j}), y = min(y_{i_j})\}$. If we use the base location of a sequence as the reference location, then we may change the center of the topological structure to another event since the base locations of the sequences in the projected database may not be the locations of the event e.

For example, the a-projected database consists of two sequence $s_1 = d(0,1) \rightarrow$

Wid	Sid	Sequences	LF
1	1	$\langle \ddagger, \mathbf{c(3,2)}, \mathbf{f(0,1)} \rangle \rightarrow \langle a(1,2), c(4,4), \mathbf{d(1,1)}, f(1,3), \mathbf{g(2,2)} \rangle \rightarrow \langle d(2,3), g(3,4) \rangle$	$\langle \ddagger, c(3,2) \rangle$, $\langle \ddagger, f(0,1) \rangle$, $d(1,1)$, $g(2,2)$
	2	$\langle \ddagger, \mathbf{c(3,2)}, d(0,-1), \mathbf{f(0,1)}, g(1,0) \rangle \rightarrow \langle \mathbf{d(1,1)}, \mathbf{g(2,2)} \rangle$	
2	1	$\langle \ddagger, f(0,1), g(-1,-1) \rangle$	
3	1	$\langle \ddagger, \mathbf{c(3,2)}, d(2,-1), \mathbf{f(0,1)} \rangle \rightarrow \langle a(-1,1), c(1,0), c(2,3), \mathbf{d(1,1)}, f(-1,2), \mathbf{g(2,2)} \rangle \rightarrow \langle a(-1,3), c(1,1), c(3,3), d(0,2), g(1,3) \rangle$	
	2	$\langle \ddagger, c(2,-1), \mathbf{c(3,2)}, d(2,0), \mathbf{f(0,1)}, g(3,1) \rangle \rightarrow \langle a(0,2), c(2,0), c(4,2), \mathbf{d(1,1)}, \mathbf{g(2,2)} \rangle$	
	3	$\langle \ddagger, c(2,-2), c(4,0), d(1,-1), g(2,0) \rangle$	

Figure 4.24: Generalized projected database of event a

$\langle a(1,2), g(2,2) \rangle$ and $s_2 = a(1,2) \rightarrow \langle f(1,3), g(2,2) \rangle$, and $base(s_1) = (0,1)$, $base(s_2) = (1,2)$. If we choose the base locations of the sequences as the reference locations, then the center of the topological structure of s_1 is changed to the event d, instead of a.

Hence, to keep all the events in the generalized spatio-temporal patterns consistent in their topological structure, we use the location of the event e as the reference location. Figure 4.24 shows the generalized projected database of the event a.

Mining k-Generalized Spatio-Temporal Patterns

Having obtained the *generalized projected database* of an event e, we proceed to discover the frequent k-generalized spatio-temporal patterns ($k \geq 2$) that are prefixed with it. We first find the set of the locally frequent events LF_e. Then, for each valid event in LF_e, we generate the $(k+1)$-generalized spatio-temporal patterns, construct its projected database, and mine it recursively. Note that in the projected database of a length-k generalized spatio-temporal pattern, the spatial support of a local event at time

window i is decided by the number of sequences in the projected databases that contain it, and the temporal support is up to the number of time windows where it is spatially frequent.

For example, let $R = 1$, $W = 15days$, $s\text{-}minsup$ = 2 and $t\text{-}minsup$ = 2. We want to retrieve all the frequent generalized spatio-temporal patterns prefixed with a in Figure 4.24. First, by mining GDB_a, we can obtain the set of its locally frequent events, i.e., $LF_a = \{\langle \ddagger, c(3,2)\rangle, \langle \ddagger, f(0,1)\rangle, d(1,1), g(2,2)\}$. Note that $\langle \ddagger, c(3,2)\rangle$ means $c(3,2)$ is an eventset extension, and $d(1,1)$ a sequence extension. With LF_a, we can generate four frequent patterns: $\langle a(0,0), c(3,2)\rangle$, $\langle a(0,0), f(0,1)\rangle$, $a(0,0) \rightarrow d(1,1)$ and $a(0,0) \rightarrow g(2,2)$. However, since $\langle a(0,0), c(3,2)\rangle \notin R$ and $\langle a(0,0), g(2,2)\rangle \notin R$, only $P_{21} = \langle a(0,0), f(0,1)\rangle$ and $P_{22} = a(0,0) \rightarrow d(1,1)$ are frequent 2-generalized spatio-temporal patterns.

Next, the frequent generalized spatio-temporal patterns prefixed with a can be further partitioned into two subsets: one prefixed with P_{21}, and the other prefixed with P_{22}. We can construct their projected database respectively and mine them recursively. The process continues until there are no more valid locally frequent events found.

In a similar way, we can find the k-generalized spatio-temporal patterns prefixed with the events c, d, f and g respectively. The final set of frequent generalized spatio-temporal patterns is the collection of patterns found in the above recursive mining process.

Figure 4.25 shows the GenSTMiner algorithm. It first scans the database once to find the frequent 1-generalized spatio-temporal patterns F_1 (line 2), treats each $e_k \in F_1$ as a

prefix, builds its projected database PDB_{e_k}, and then transforms PDB_{e_k} into GDB_{e_k} (lines 3-5). Next, it calls the subroutine Ptn-growth method (line 6). The subroutine Pattern-growth method recursively calls itself and works as follows: For prefix e, it scans its projected database once to find its locally frequent events (line 9), it grows e with each valid locally frequent event to get a new prefix e', it builds the projected database for the new prefix, and it calls itself recursively (lines 12-16).

Compared to PrefixSpan, GenSTMiner can find the complete set of generalized spatio-temporal patterns by generating a much smaller set of candidates. The following section describes optimization techniques to reduce further the number of candidates generated and further reduce memory requirement.

Conditional Projected Database

We observe that not every event or eventset in the sequence in the database participates in generalized spatio-temporal patterns. In order to eliminate those non-promising events or eventsets in the projected database of an event, we introduce the concept of the conditional database with respect to an event e.

Given an input sequence s in the database $\mathcal{D}$, *the conditional spatial-sequences* w.r.t. *an event* e of s is the set of subsequences of s prefixed with e, and each of them is a spatial-sequence and all the eventsets in these subsequence are CloseNeighbors of the event e.

For example, given the input sequence $s = \langle \mathbf{a(0,0)}, c(3,2), f(0,1)\rangle \rightarrow \langle \mathbf{a(1,2)}, c(4,4), d(1,1), f(1,3), g(2,2)\rangle \rightarrow \langle d(2,3), g(3,4)\rangle$, and $R = 1$, $W = 15 days$. We

Algorithm GenSTMiner

Input: $\mathcal{D}$, the spatio-temporal database;
R, size of spatial neighbor relation;
W, size of time window;
s-minsup, minimum spatial support;
t-minsup, minimum temporal support.

Output: M: set of generalized spatio-temporal patterns

1. $M = \emptyset$;
2: F_1 = {all the frequent events};
3: **for each** $e_k \in F_1$ **do**
4: PDB_{e_k} = projected database($\mathcal{D}, e_k$);
5. Convert PDB_{e_k} into GDB_{e_k};
6: Call **Ptn-growth**(e_k,GDB_{e_k},s-minsup,t-minsup);
7: **return** M

Procedure Ptn-growth(α, PDB_α,s-minsup,t-minsup)

8: $M = M \cup \alpha$;
9: Scan PDB_α once and get all frequent LF;
10: **if** LF is empty **then**
11: **return**;
12: **for each** $e_j \in LF$ **do**
13: **if** $\forall e_i \in \alpha, (e_i, e_j) \in (R, W)$ **then**
14: $\alpha' = \alpha \cdot e_j$;
15: $PDB_{\alpha'}$ = projected database (PDB_α, α');
16: Call **Ptn-growth**(α',$PDB_{\alpha'}$,s-minsup,t-minsup);
17: **return**

Figure 4.25: The GenSTMiner algorithm

want to retrieve the conditional spatial sequence of the event a. Notice that there are only two instances of a in s, i.e., $a(0,0)$ and $a(1,2)$. First, for the instance $a(0,0)$, there are only two events $f(0,1)$ and $d(1,1)$ in s that can form a conditional spatial sequence together with $a(0,0)$; for the instance $a(1,2)$, the events $d(1,1), f(1,3), g(2,2)$ and $d(2,3)$ are valid. Hence, the final s_a consists of two sequences $\langle a(0,0), f(0,1)\rangle \rightarrow d(1,1)$ and $\langle a(1,2), d(1,1), f(1,3), g(2,2)\rangle \rightarrow d(2,3)$.

The collection of all the conditional spatial-sequences w.r.t. an event e in the database $\mathcal{D}$ forms the *conditional database* w.r.t. *an event* e. All the conditional spatial-sequences are ordered according to their time.

GenSTMiner obtains the projected database of an event e from its conditional database, instead of database $\mathcal{D}$. This effectively removes unpromising events from the projected databases of the event e. For simplicity, we call the projected databases of an event e retrieved from the conditional database as the *conditional projected databases* w.r.t the event e or e-*conditional projected databases*.

For example, Figure 4.26 shows the a-conditional projected database retrieved from Figure 4.22(c) by setting $R = 1$. Clearly, the a-conditional projected database is much more compact than the a-projected database (see Figure 4.23).

While the conditional database can be used to remove unpromising events from the event e-projected database, there are still unpromising events when we further construct the projected database of length-k ($k > 2$) generalized spatio-temporal patterns. We use the Apriori checking as in [PHMAP01] to prune events during the construction of the projected databases of length-k generalized spatio-temporal patterns.

Wid	Sid	Conditional spatial-sequences	Prefix
1	1	$\langle \ddagger, f(0,1)\rangle \rightarrow \langle d(1,1)\rangle$	$a(0,0)$
	2	$\langle \ddagger, d(1,1), f(1,3), g(2,2)\rangle \rightarrow d(2,3)$	$a(1,2)$
2	1	$\langle \ddagger, f(1,2), g(0,0)\rangle$	$a(1,1)$
3	1	$\langle \ddagger, f(1,2)\rangle \rightarrow \langle a(0,2), c(2,1), d(2,2)\rangle \rightarrow c(2,2)$	$a(1,1)$
	2	$\langle \ddagger, f(0,3)\rangle \rightarrow d(1,3)$	$a(0,2)$
	3	$\langle \ddagger, d(1,3)\rangle$	$a(0,4)$

Figure 4.26: a-conditional projected database

To construct the P-conditional projected database, where P is a length-l generalized spatio-temporal pattern, let E be the last element of P and P' be the prefix of P such that $P = P' \cdot E$.

If $P' \cdot x$ is not frequent, then event x can be excluded from projection. For example, if we know that $a(0,0) \rightarrow g(0,1)$ is not frequent, then event $g(0,1)$ can be excluded from the construction of $a(0,0) \rightarrow d(1,1)$-conditional projected databases.

However, if $P' \cdot x$ is frequent, but there $\exists e \in P, s.t. \langle e, x\rangle \notin R$, then event x can be excluded from the projection. For example, let $R = 1$ and $\langle a(0,0), g(-1,-1)\rangle$ is frequent, but since $\langle f(0,1), g(-1,-1)\rangle \notin R$, we can remove g(-1,-1) from the construction of $\langle a(0,0), f(0,1)\rangle$-conditional projected database.

Moreover, let E' be formed by substituting any item in E by x. If $P' \cdot E'$ is not frequent, then event x can be excluded from the first element of the suffix of the element that is a superset of e. For example, suppose $a(0,0) \rightarrow \langle b(0,1), f(1,1)\rangle$ is not frequent. To construct $a(0,0) \rightarrow \langle b(0,1), c(0,1)\rangle$-projected database, conditional spatial-sequence $a(0,0) \rightarrow \langle (b(0,1), c(0,1), f(1,1), g(1,1)\rangle \rightarrow d(0,1)$ should be projected to $\langle \ddagger, g(1,1)\rangle \rightarrow d(0,1)$.

Pseudo-projection

In general, we obtain projected databases by scanning the sequences at each time window in the databases. However, after scanning the projected databases, we have known the time windows in which the locally frequent event e is not spatially frequent. Hence, there is no need to scan such time windows to get the projection sequences of the event e. For example in Figure 4.22(c), we know that event a is not spatially frequent at time window 2. Hence, we can stop retrieving projection of sequences prefixed with a at time window 2. To realize this, for each locally frequent event e, we use a bitmap to record the time windows in which it is frequent. We only retrieve the projection of the sequences from the time windows where its corresponding value in the bitmap is set to 1. Moreover, we could get the frequent period of a generalized spatio-temporal pattern by scanning the bitmap once.

Additionally, when we retrieve the conditional projected database w.r.t. an event e, we observe that an event e_k in the sequence in the database may appear many times in the e-conditional projected database. The cost of the projection (constructing the conditional projected database recursively) becomes a major cost in GenSTMiner. We can use the pseudo-projection technique in PrefixSpan to reduce the cost of the projection.

In PrefixSpan, pseudo-projection is used to avoid physically copying suffixes. When the database can be held in main memory, instead of constructing a physical projection by collecting all suffixes, pseudo-projection uses pointers referring to the sequences in the database. Every projection consists of two pieces of information $\langle$*pointer, offset*$\rangle$, where *pointer* points to the sequence in the database and *offset* indicates the start

$s = \langle a(0,0), c(3,2), \mathbf{f(0,1)}\rangle \rightarrow \langle a(1,2), c(4,4), \mathbf{d(1,1)}, f(1,3), g(2,2)\rangle \rightarrow \langle d(2,3), g(3,4)\rangle$			
	prefix	physical projection	pseudo projection
s_a	$a(0,0)$	$\langle \ddagger, f(0,1)\rangle \rightarrow d(1,1)$	$\langle$*pointer to* s, 3, 10010000, (0,0)$\rangle$
	$a(1,2)$	$\langle \ddagger, d(1,1), f(1,3), g(2,2)\rangle \rightarrow d(2,3)$	$\langle$*pointer to* s, 6, 11110, (1,2)$\rangle$

Figure 4.27: Example of pseudo-projection

position of the suffix in the sequences.

Unlike in PrefixSpan where only the first occurrence of an item is considered, GenSTMiner needs to consider the suffixes in an input sequence prefixed with different instances of the event e. Hence, the problem of pseudo-projection becomes more complicated.

In GenSTMiner, every projection consists of four pieces of information: $\langle$*pointer, offset, bitmap, refloc*$\rangle$, where *pointer* points to the sequence in the database, *offset* indicates the start position of the suffixes in the sequence, *bitmap* indicates the appearance of the events in the suffixes of the sequence in the conditional spatial sequence w.r.t. the event e, size of $bitmap$ is equivalent to the number of events in the suffixes of the sequence, and *refloc* stores the reference location of the conditional spatial sequence. Figure 4.27 shows an example of the pseudo-projection of a sequence in the database.

4.3.4 Performance Evaluation

We implement the algorithms in C++ and evaluate their performance on both synthetic and real-life datasets. The experiments are carried out on a Pentium 4, 1.6 GHZ processor with 256MB memory running Windows XP.

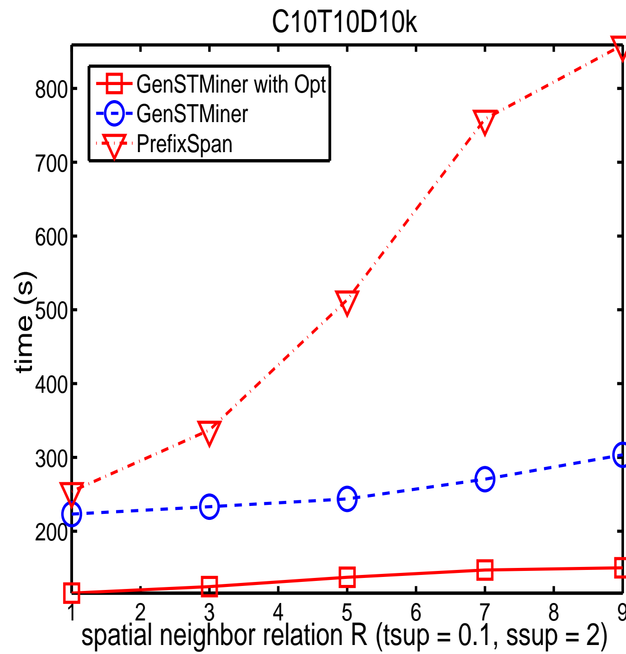
Experiments on Synthetic Dataset

We augment the IBM Quest synthetic data generator [2] to include spatial information by generating N items using F spatial features and L locations. We generate datasets by setting $N = 10,000$, $F = 1,000$, $L = 100$. The other parameters are D, number of sliding windows (= size of Database); C, average number of eventsets in a sliding window; and T, average number of events in an eventset. We evaluate the performance of GenSTMiner on the synthetic dataset $C10T10D10k$ by varying the parameters R, $t\text{-}minsup$, and $s\text{-}minsup$. We test the performance of GenSTMiner with and without optimization techniques and compare it with PrefixSpan. The results are shown in Figures 4.28, 4.29 and 4.30. The results indicate that GenSTMiner outperforms PrefixSpan, especially when it uses optimization techniques. This is expected as the pruning techniques we use in GenSTMiner not only reduce the size of the projected databases, but also eliminate infeasible events and sequences.

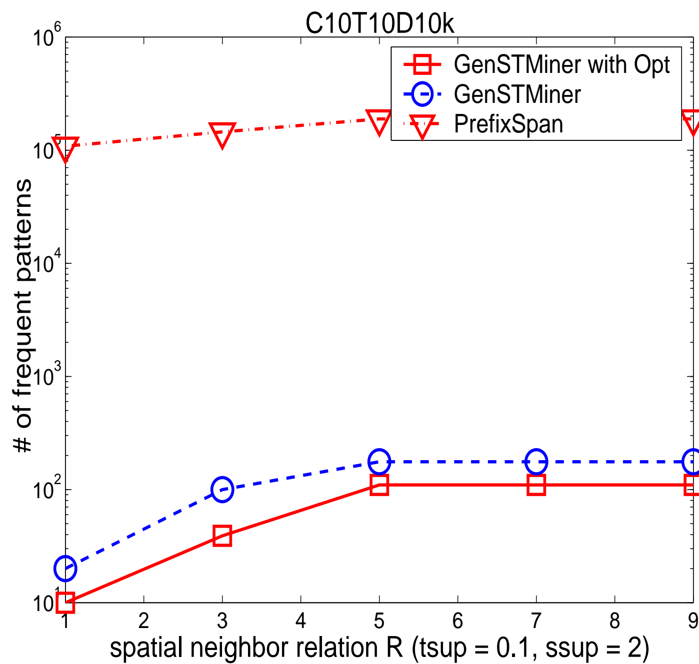
Figure 4.28 shows the efficiency of the algorithms when the size of the spatial neighbor relation R is varied. Figure 4.28(a) shows that the runtime of GenSTMiner grows linearly as spatial neighbor relation R increases. When R is large, the number of spatial neighborhoods of an event tends to be large, and the length of frequent patterns increases (see Figure 4.28(b).

Figure 4.29 shows that GenSTMiner requires more time to find frequent patterns when t-minsup is small. This is due to more local frequent patterns becoming globally frequent when t-minsup is small. As a result, the size of frequent patterns become

[2] http://www.almaden. ibm.com/software/quest

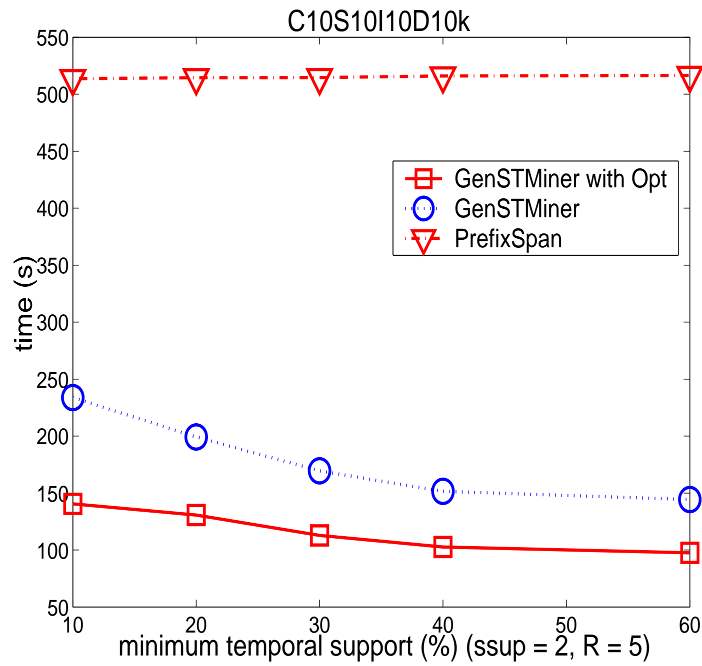


(a)

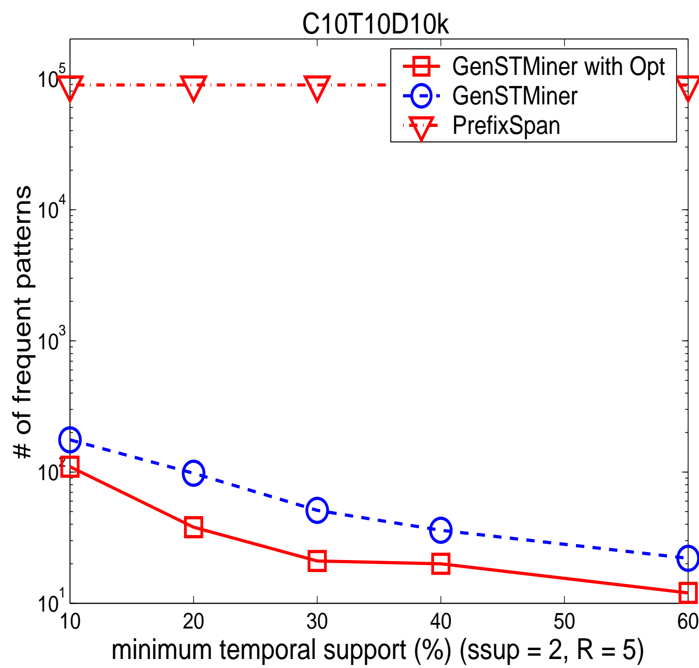


(b)

Figure 4.28: Runtime vs. parameter R



(a)



(b)

Figure 4.29: Runtime vs. parameter t-minsup

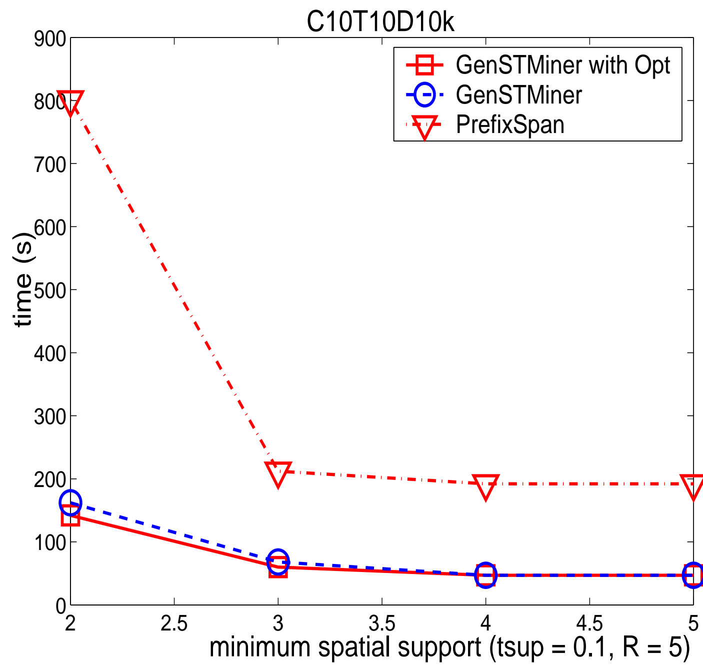


Figure 4.30: Runtime vs. parameter s-minsup

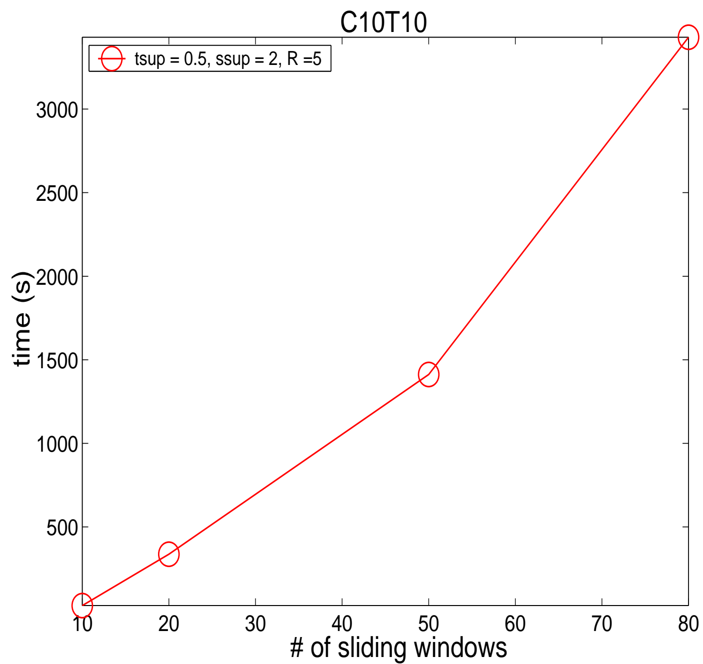


Figure 4.31: Scalability

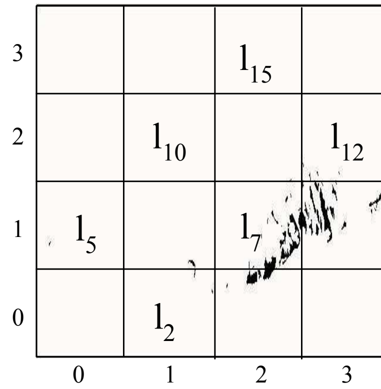
larger. This is also verified by Figure 4.29(b). Similarly, Figure 4.30 indicates that GenSTMiner requires more time to find frequent patterns when s-minsup is small.

We also investigate the scalability of GenSTMiner in terms of database size. Figure 4.31 shows the runtime of GenSTMiner when the parameter D (i.e. number of sliding windows) is varied from $20k$ to $100k$. Figure 4.31 indicates that GenSTMiner grows linearly with database size. In other words, GenSTMiner is able to scale well for large datasets.

Comparative Study

This set of experiments aims to show the usefulness of generalized spatio-temporal patterns as compared to flow patterns using a real-life dataset. We obtain three years of standard meteorological data from five stations that are closely located in space from the Nation Data Buoy Center. The dataset has 10 features that are recorded hourly. After discretization, the final dataset contains 30 features. With these 30 features, we define a set of meteorological events. A sample of the events defined are as follows. $A \uparrow (l_a)$ (or $A \downarrow (l_a)$): denotes the event that the air temperature at location l_a has increased (or decreased); $S \uparrow (l_a)$ (or $S \downarrow (l_a)$): denotes the event that the wind speed at location l_a has increased (or decreased); and $G \uparrow (l_a)$ (or $G \downarrow (l_a)$): denotes the event that the gust speed at location l_a has increased (or decreased).

We divide the whole space into 4×4 grids so that the five locations are distributed uniformly. Figure 4.32(a) shows the geographical positions of the five locations, namely l_2, l_7, l_{10}, l_{12} and l_{15}. FlowMiner and GenSTMiner are applied on this dataset with t-



(a) Neighbor relations

Flow Patterns
$A\uparrow(l_2) \rightarrow G\uparrow(l_7), S\uparrow(l_7)\langle\rightarrow \langle G\downarrow(l_{10}), S\downarrow(l_{10}), G\downarrow(l_{12}), S\downarrow(l_{12})\rangle$
$\langle A\uparrow(l_{10}), A\uparrow(l_{12})\rangle \rightarrow \langle G\uparrow(l_{15}), S\uparrow(l_{15})\rangle$
$\langle G\uparrow(l_{10}), S\uparrow(l_{10})\rangle \rightarrow A\uparrow(l_{15})$
$\langle G\uparrow(l_7), S\uparrow(l_7)\rangle \rightarrow \langle A\uparrow(l_{10}), A\uparrow(l_{12})\rangle$
$\langle G\uparrow(l_{10}), S\uparrow(l_{10})\rangle \rightarrow \langle G\downarrow(l_{15}), S\downarrow(l_{15})\rangle$
Generalized Spatio-Temporal Patterns
$A\uparrow(0,0) \rightarrow \langle G\uparrow(1,1), S\uparrow(1,1)\rangle$
$\langle G\uparrow(0,0), S\uparrow(0,0)\rangle \rightarrow A\uparrow(1,1)$
$\langle G\uparrow(0,0), S\uparrow(0,0)\rangle \rightarrow \langle G\downarrow(1,1), S\downarrow(1,1)\rangle$

(b) Interesting frequent patterns

Figure 4.32: Comparison of flow patterns and generalized spatio-temporal patterns

minsup = 10, s-minsup = 2, $W = 6days$ and R = 2. Figure 4.32(b) summarizes some of the interesting patterns we find.

We observe that flow patterns are able to capture the flow of events such as: an increase in air temperature at location l_2 leads to an increase in wind speed and gust speed at location l_7; and an increase in air temperature at location l_{10} leads to an increase in wind speed and gust speed at location l_{15}. However, the usefulness of these flow patterns is rather limited as they are unable to provide a general trend. On the other hand, the generalized spatio-temporal pattern reveals the trend that whenever there is an increase in air temperature at a specific location, we can expect an increase in wind speed and gust speed at the Northeastern neighbor of the location. By knowing the general trend, the meteorologist can perform more accurate forecast of the weather.

4.4 Summary

In the chapter, we have studied the problem of mining spatial sequence patterns. We have presented two new classes of spatial sequence patterns, *flow patterns* and *generalized spatio-temporal patterns*, to describe the changes of events over space and time.

We have developed a depth-first algorithm FlowMiner to find flow patterns. We have also designed a new candidate generation algorithm that could quickly remove infeasible candidates by using both temporal and spatial relationships among events. In addition, we have proposed some optimization techniques to enhance the efficiency of FlowMiner. A comprehensive performance study shows that FlowMiner can find the

complete set of flow patterns efficiently, and it possesses linear scalability.

Further, we have presented a framework *GenSTMiner* based on the methodology of pattern growth approach to discover generalized spatio-temporal patterns. Some optimization techniques are proposed to boost the efficiency of GenSTMiner. The experimental study indicates that with the optimization techniques, GenSTMiner is effective and scalable in mining frequent generalized spatio-temporal patterns.

Chapter 5

A Partition-based Approach for Mining Arbitrary Spatio-temporal Patterns in the Presence of Updates

In Chapter 3 and Chapter 4, we find spatio-temporal patterns such that objects in valid instances of a pattern satisfy specific types of spatial and temporal relationships. For patterns with arbitrary relationships, we represent the relationships with a graph, where each vertex in a graph represents a variable labeled by an attribute or event, and each edge represents a spatial relationship, temporal relationship, or both. As a result, the spatio-temporal database is correspondingly transformed into a graph database, and the problem of mining frequent spatio-temporal patterns becomes the problem of finding frequent subgraphs in the graph database. Figure 5.1 shows the framework for mining arbitrary spatio-temporal patterns.

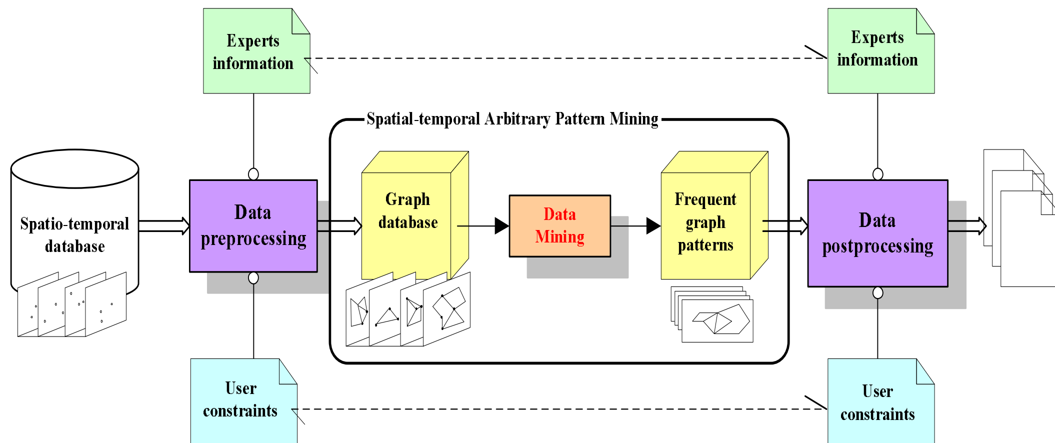


Figure 5.1: Framework for mining arbitrary spatio-temporal patterns

Data mining in graph databases has received much attention. We have witnessed many algorithms proposed for mining frequent graphs. [IWNM01, KK01] introduce the apriori-like algorithms, *AGM* and *FSG*, to mine the complete set of frequent graphs. However, both algorithms are not scalable as they require multiple scans of databases and tend to generate many candidates during the mining process. Subsequently, [YH03, NK04] propose the depth-first graph mining approaches, *gSpan* and *Gaston*. These approaches are essentially memory-based and their efficiencies decrease dramatically if the graph database is too large to fit into the main memory. Recognizing this problem, [WWP$^+$04] present an effective index structure *ADI* to represent the graph databases and to facilitate the major graph mining operations. Based on the ADI structure, they present an algorithm ADIMINE, which has the advantageous to mine various graphs over large databases that cannot be held into main memory. However, this solution does not work well when the graph database is still evolving. This is because the ADI structure has to be rebuilt each time the graph database is updated.

In short, while previous studies have made excellent progress in mining graph databases, many of them assume that the graphs in the databases are relatively static and simple. They do not scale well for mining graphs in a dynamic environment. For example, a spatio-temporal database can contain millions of different structures and the number of different labels in the graphs is easily in the range of hundreds. Changes to spatio-temporal databases cause changes to the graph structures that model the relationships in the spatio-temporal data. Re-execution of the mining algorithm each time the graphs are updated is costly, and may result in an explosion in the demand for computational and I/O resources. Consequently, there is an urgent need to find an algorithm that is scalable and can incrementally mine from only those parts of the graph databases that have been changed.

We propose a partition-based approach to graph mining. Our idea is to isolate update changes to a small set of subgraphs and re-execute the graph mining algorithm only on the isolated subgraphs. Instead of finding frequent graph patterns on complex graphs, we recursively partition complex graphs into smaller, more manageable subgraphs until these subgraphs can fit into the main memory. With this, existing memory-based graph mining algorithms can be utilized to discover frequent patterns in the subgraphs. The discovered patterns are then joined via a merge-join operation to recover the final set of frequent patterns that exist in the original complex graphs.

Mining frequent patterns in databases using the partition-based approach is not new. As early as the 1990s, [SON95] has introduced a partition-based method to find association rules. Since then, many partition-based algorithms have been developed to solve

the various problems in data mining, such as classification[HCB98, SAM96], clustering [BFR98, NH02] and incremental mining [LLC01], etc. The data partitioning approach involves splitting a dataset into subsets, learning/mining from one or more of the subsets, and possibly combining the results. The advantage of the data partitioning approach is the ability to avoid thrashing by memory management systems which frequently occurs when algorithms try to process huge datasets in the main memory. To date, there has been no work on mining frequent subgraphs using the partition-based method.

In this chapter, we design a partition-based algorithm to divide graphs into k smaller subgraphs (k is determined by the size of the main memory) with the goal of reducing connectivity among subgraphs while localizing most, if not all, the updated nodes to a minimal number of subgraphs. Once divided, we can utilize existing efficient memory-based graph mining algorithms to discover frequent patterns in these subgraphs. We develop a merge-join operation to losslessly recover the complete set of frequent subgraphs in the database from the set of subgraphs found in the partitions. We also give a theoretical proof to ensure that mining of frequent subgraphs in the partitions will be equivalent to mining in the original graph database. In the following, we present the details of our partition-based graph mining algorithm, called *PartMiner*. Here, we make use of the cumulative information obtained during the mining of previous subgraphs to effectively reduce the number of candidate graphs. *PartMiner* is inherently parallel in nature and can be parallelized with minimal communication and synchronization among processing nodes. Finally, we extend PartMiner to handle updates in

the graph database. The *IncPartMiner*, an extended version of PartMiner, makes use of the pruned results of the pre-updated database to eliminate the generation of unchanged candidate graphs, thus leading to tremendous savings.

This chapter is organized as follows. Section 5.1 discusses some preliminary concepts. Section 5.2 presents the partition-based graph mining approach. Section 5.3 reports the experimental results. We conclude the chapter with Section 5.5.

5.1 Preliminary Concepts

We represent a labeled graph by $G = (V, E, L_V, L_E)$ where V is the set of vertices, E is the set of edges denoted as pair of vertices, L_V is a set of labels associated with the vertices, and L_E is a set of labels for the edges. A graph G is connected if a path exists between any two vertices in V. The size of a graph is the number of edges in it, and a graph G with k edges is called a *k-edge graph* or a graph of size k.

A graph G_1 is *isomorphic* to a graph G_2 if there exists a bijective function f : $V_1 \rightarrow V_2$ such that for any vertex $u \in V_1$, $f(u) \in V_2 \wedge L_u = L_{f(u)}$, and for any edge $(u, v) \in E_1, (f(u), f(v)) \in E_2 \wedge L_{(u,v)} = L_{(f(u),f(v))}$. An *automorphism* of a graph G is an isomorphism from G to G. A *subgraph isomorphism* from G_1 to G_2 is an isomorphism from G_1 to a *subgraph* G_2, and G_1 is called a *supergraph* of G_2, denoted as $G_2 \subseteq G_1$.

A graph database is a set of tuples (gid, G), where gid is a graph identifier and G is an undirected labeled graph. Given a graph database $\mathcal{D}$, the *support* of a graph G is the

number of graphs in $\mathcal{D}$ that are supergraphs of G.

To find all frequent subgraphs in a database, we need to encode the structure of a graph such that if two graphs have identical encoding, they are isomorphic. We use the method proposed in [YH02] to encode a graph. The method in [YH02] performs a depth-first search on a graph G to order the vertices and construct the *DFS-tree* T of G. An edge (v_i, v_j) is called a *forward edge* if $i < j$; otherwise, it is called a *backward edge*.

A linear order $\prec$ on the edges in G is defined as follows: Given two edges $e = (v_i, v_j)$ and $e' = (v_{i'}, v_{j'})$, $e \prec e'$ if either one of the following conditions is true:

1. both e and e' are forward edges, $j < j'$ or $(i > i' \wedge j = j')$;
2. both e and e' are backward edges, $i < i'$ or $(i = i' \wedge j < j')$;
3. e is a forward edge and e' is a backward edge, $j \leq i'$;
4. e is a backward edge and e' is a forward edge, $i < j'$.

By ordering the edges in a graph G, the structure of G can be encoded using the *DFS code*. Given a graph G, and a DFS-tree T, the *DFS code* of G w.r.t T, denoted by $code(G, T)$, is a list of all edges E in the order of $\prec$, where an edge (v_i, v_j) is written as $(v_i, v_j, l_{v_i}, l_{(v_i, v_j)}, l_{v_j})$.

Since a graph can have many different DFS-trees, [YH02] defines the notion of the *minimum DFS code*, which is the minimum of all the DFS codes of a graph G, to encode the graph.

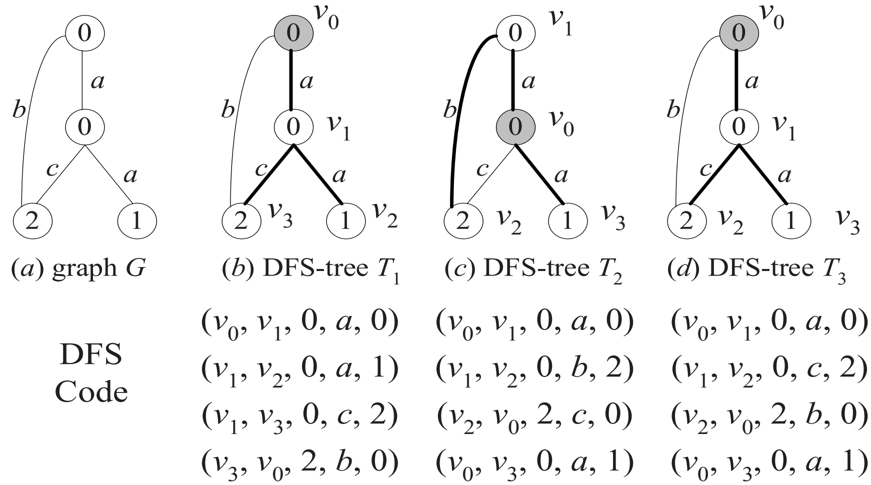


Figure 5.2: Example of the DFS tree and DFS code

Figure 5.2 shows a graph G, and three DFS trees of G, together with their corresponding DFS codes. The $code(G, T_1)$ in Figure 5.2(b) is the minimum DFS code.

5.2 Partition-based Graph Mining

Figure 5.3 shows the framework of the proposed partition-based graph mining approach. It consists of two phases. In the first phase, we iteratively call a graph partitioning algorithm to partition each of the graphs in the graph database into smaller subgraphs. Then we group the subgraphs into units.

The second phase applies any existing memory-based graph mining algorithm to discover the frequent subgraphs in each unit. The set of frequent subgraphs in each unit are then merged via a merge-join operation to recover the complete set of frequent subgraphs. The proposed framework can be easily extended to handle incremental mining when updates occur in the graph database (see Section 5.2.5).

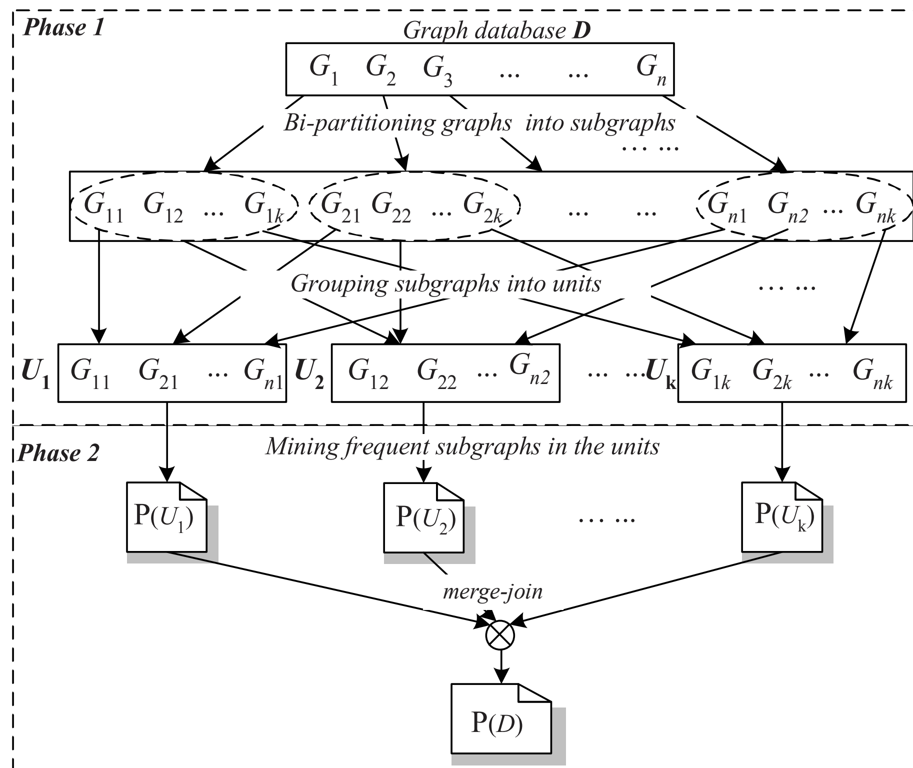


Figure 5.3: Overview of partition-based graph mining

5.2.1 Dividing Graph Database into Units

The motivation for the proposed partition-based graph mining approach is to effectively deal with graphs in the presence of frequent updates. By partitioning the graphs, we can reduce graph complexity as well as size so that existing memory-based graph mining algorithms can be applied. However, the frequent subgraphs obtained from each unit need to be combined using a merge-join operation which is costly. To minimize the number of units involved in the merge-join operation, it is important to minimize connectivity (i.e., the number of connective edges) among subgraphs in the units. Moreover, in the presence of updates, it is also important to isolate those vertices and edges that are changed frequently, and localize them in a minimal number of units to reduce the

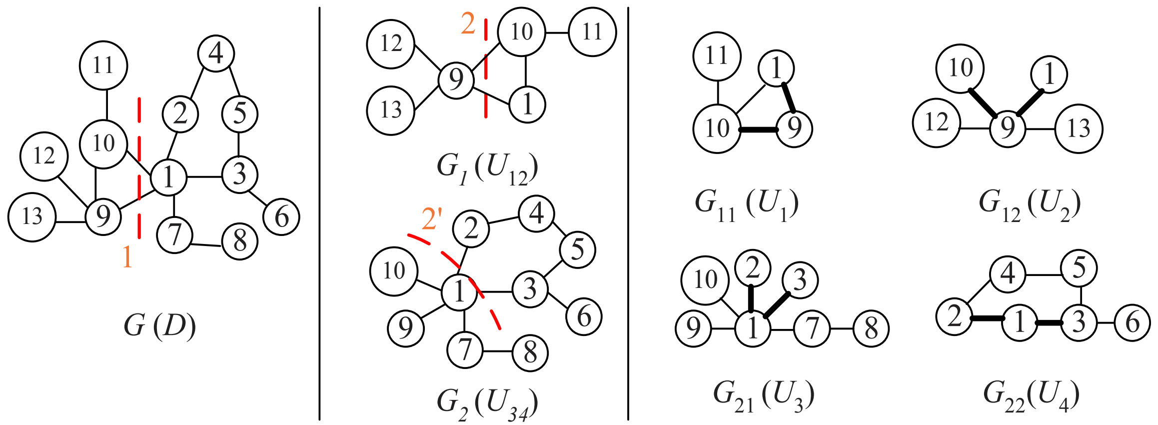


Figure 5.4: Example of graph bi-partitioning

number of units that need to participate in the incremental mining process.

To achieve this goal of minimizing connectivity among units, each graph in the database must be carefully partitioned and organized into units. If we randomly partition the graphs and group them into units, then the connectivity among the subgraphs in the units will not be clear, and a merge-join operation will be needed on each pair of units. Therefore, we adopt an approach that repeatedly bi-partitions each of the graphs in the database.

Figure 5.4 shows a graph G which is first divided into two subgraphs G_1 and G_2. G_1 (G_2) is again further divided into two subgraphs G_{11} and G_{12} (G_{21} and G_{22}). This bi-partitioning process yields a total of four subgraphs for G. By applying this bi-partitioning procedure on each of the graphs G_i in the database, we have four subgraphs G_{i1}, G_{i2}, G_{i3}, and G_{i4} for each G_i. Each of the subgraphs G_{ij}, $1 \leq j \leq 4$, is grouped into one unit U_j.

The bi-partitioning approach facilitates the recovery of the complete set of graphs in

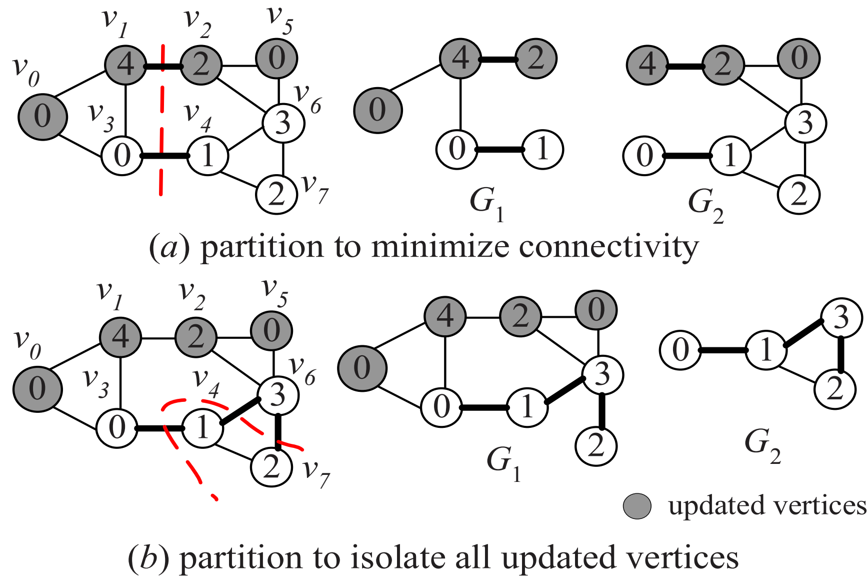


Figure 5.5: Example of partitioning criteria

a database from the subgraphs in the units. In our example, we just need to combine the set of subgraphs in U_1 and U_2 to get the set of subgraphs in U_{12}, and the set of subgraphs in U_3 and U_4 to get the set of subgraphs in U_{34}. The set of subgraphs in U_{12} and U_{34} are subsequently combined to obtain the original graph database. This also indicates the sequence of combining the frequent subgraphs mined in each unit to obtain the final set of frequent subgraphs for the database.

We use two criteria to carry out bi-partitioning. The first criteria is to minimize connectivity between subgraphs, and the second criteria is to isolate frequently updated vertices to a subgraph. Figure 5.5 illustrates how a graph G can be partitioned using these two criteria. Note that subgraphs should include the connective edges between them so that we can recover the original graph later. For example, edges (v_1, v_2) and (v_3, v_4) in Figure 5.5(a), and edges (v_3, v_4), (v_4, v_6) and (v_6, v_7) in Figure 5.5(b) are

connective edges.

We associate each vertex v in a graph G with a value $ufreq$ to indicate its update frequency, denoted as $v.ufreq$. The vertices of G are sorted in descending order according to their update frequencies. Suppose that the vertex set V of the graph G is divided into two subsets V_1 and V_2, we define a weight function $w(V_1)$ to reflect the average update frequencies of the vertices in the vertex set V_1 and its connectivity to the vertex set V_2 (see Equation (1)).

$$w(V_1) = \lambda_1 \frac{\sum_{v_i \in V_1} v_i.ufreq}{|V_1|} - \lambda_2 |E_{V_1,V_2}| \quad (5.1)$$

where E_{V_1,V_2} is the set of connective edges $e(v_i, v_j)$ between vertex sets V_1 and V_2, i.e., $v_i \in V_1, v_j \in V_2$ or $v_i \in V_2, v_j \in V_1$. The first term in $w(V_1)$ computes the average update frequencies of the vertices in the subset V_1, and the second term counts the number of connective edges. We use two parameters λ_1 and λ_2 to set the weight between these two terms.

Figure 5.6 shows the graph partitioning algorithm, called $GraphPart$. Line 1 sorts the vertex set V of the graph G according to their update frequency. Let v_c be the centroid of V. Then v_c divides V into two subsets V_1 and V_2, where V_1 contains the vertices $v_i \in V \wedge v_i.ufreq \geq v_c.ufreq$, and V_2 contains vertices $v_j \in V \wedge v_j.ufreq < v_c.ufreq$. For each vertex $v_i \in V_1$, we traverse the graph G in depth-first manner to construct the vertex subset V_i, and compute the weight function $w(V_i)$ (lines 4-12). The vertex set with the largest weight function is the final subset V^*. Note that when scanning unvisited neighbors of a vertex, the vertex with the highest frequency should be the next visited node (line 21). Finally, we obtain two subgraphs G_1 and G_2. G_1

Algorithm GraphPart

Input: G, the graph

Output: G_1, G_2, the two subgraphs of G

1: V = {vertices sorted according to their update frequency};

2: $V^* = \emptyset$;

3: $w(V^*) = -\infty$

4: **for**(i = 0; $i < |V|/2$; i++) {

5: $V_i = \emptyset$;

6: call **DFSScan**(V, i, V_i);

7: Compute $w(V_i)$;

8: **if** $(w(V_i) > w(V^*))$ {

9: $w(V^*) = w(V_i)$;

10: $V^* = V_i$;

11: }

12: }

13: $G_1 = \{e_{ij} = (v_i, v_j) | v_i \in V^*, v_j \in V^*\} \cup \{e_{ij} = (v_i, v_j) | v_i \in V^*, v_j \notin V^*\}$

14: $G_2 = \{e_{ij} = (v_i, v_j) | v_i \notin V^*, v_j \notin V^*\} \cup \{e_{ij} = (v_i, v_j) | v_i \in V^*, v_j \notin V^*\}$

Procedure DFSScan(V, i, V_i)

15: $stack$ = $\emptyset$, m = 0;

16: $stack$.push(v_i);

17: **while**($stack \neq \emptyset \wedge m \leq |V|/2$){

18: v = $stack$.pop();

19: $V_i = V_i \cup \{v\}$;

20: m ++;

21: choose the neighbor vertex v_h, s.t. $v_h.visited = 0$, and $\forall v_s$,$v_s.visited = 0 \wedge (v, v_s) \in E, v_s.ufreq < v_h.ufreq$;

22: $stack$.push(v_h);

23: }

Figure 5.6: Algorithm to partition a graph

```
Procedure DBPartition(𝒟, k)
𝒟, graph database;
k: number of units
1:   D_{0,0} = 𝒟;
2:   i = 1;
3:   l = ⌊log_2 k⌋;
4:   while (i ≤ l) {
5:       for (j = 0; j < 2^{i-1}; j++)
6:           DivideDBPart(D_{i-1,j}, D_{i,2j}, D_{i,2j+1});
7:       i++;
8:   }
9:   for(j = 0; j < k - 2^l; j + +)
10:      DivideDBPart(D_{i-1,j}, U_{2j}, U_{2j+1});

Function DivideDBPart(D_s, D_{1,0}, D_{1,1})
1:   D_{1,0} = ∅;
2:   D_{1,1} = ∅;
3:   for each graph G ∈ D_s {
4:       G_1, G_2 = calling GraphPart(G);
5:       D_{1,0} = D_{1,0} ∪ {G_1};
6:       D_{1,1} = D_{1,1} ∪ {G_2};
7:   }
```

Figure 5.7: Dividing a graph database into units

contains all vertices in V^*, and G_2 contains all vertices in V/V^*. Both G_1 and G_2 also include the connective edges (lines 13-14).

After partitioning each graph in the database into a set of subgraphs, the next step is to group the subgraphs into units such that each unit can fit into the main memory. Figure 5.7 shows the algorithm to divide the database into units. We use a parameter k to indicate the number of units that the database will be divided into. The value of k is determined by the availability of the main memory and the size of the database.

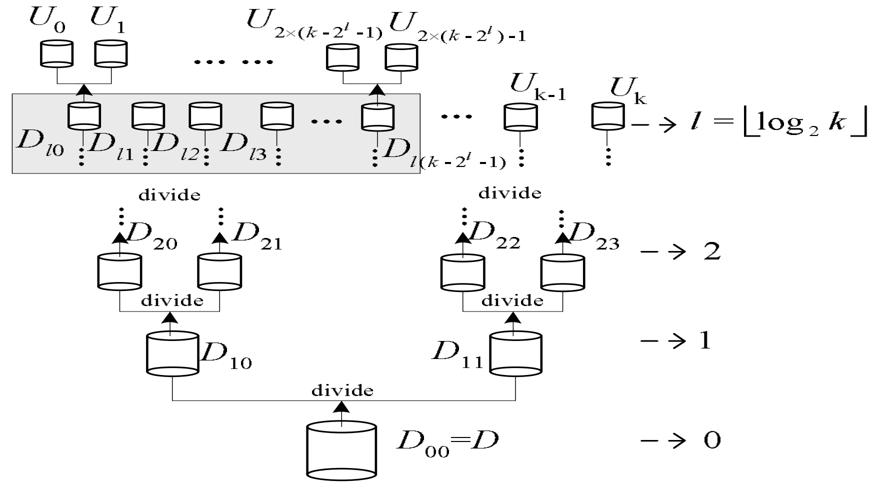


Figure 5.8: Partitioning the graph database into k units

For each graph in the database, we repeatedly call Algorithm GraphPart to partition it. The subgraphs generated during the partitioning process are kept in the database $D_{i,j}$, $1 \leq i \leq \lfloor log_2 k \rfloor$, $0 \leq j \leq 2^{i-1}$. Finally, the resulting k subgraphs are then distributed to the k units, i.e., $U_1, U_2, \ldots, U_k$. Figure 5.8 shows the structure by dividing the graph databases into k units.

5.2.2 Mining Frequent Subgraphs in Units

After dividing the graph database into k units such that each unit can fit into the main memory, we can now use any existing memory-based algorithms to find frequent subgraphs in the units.

Many memory-based algorithms have been proposed to discover frequent graphs. In this work, we use the ADIMINE algorithm [WWP+04] to find the set of frequent subgraphs in the units. ADIMINE uses an effective index structure ADI to index the

Algorithm ADIMINE

Input: U, one of the units of the database
sup, minimum support.
Output: $\mathcal{P}(U)$, the set of frequent subgraphs in U
1: construct the ADI structure if it is not available
2: F_1 = {frequent edges in edge table};
3: for each $e \in F_1$ {
4: F_e = {set of frequent adjacent edges for e};
5: call *subgraph-mine*(e, F_e)
6: }
Procedure *subgraph-mine* (G, F_e
7: for each $e \in F_e$ {
8: let G' be the graph by adding e into G
9: if DFS code of G' is not minimum
10: return;
11: update F_e of adjacent edges;
12: call *subgraph-mine*(G', F_e)
13: }

Figure 5.9: Outline of ADIMINE algorithm

graph databases. Further, with the ADI structure, the major operations in graph mining can be facilitated efficiently.

Figure 5.9 gives an outline of the ADIMINE algorithm. Line 1 constructs the ADI structure if it is unavailable. Line 2 obtains all frequent edges from the ADI structure. For each frequent edge, the algorithm retrieves the set of frequent adjacent edges from the ADI structure and uses them to growth the patterns by calling the pattern growth procedure *subgraph-mine*. The procedure subgraph-mine tries every frequent adjacent edge e and check whether e can be added into the current frequent graph pattern G to form a larger pattern G'. Once a larger pattern G' is found, the set of frequent adjacent

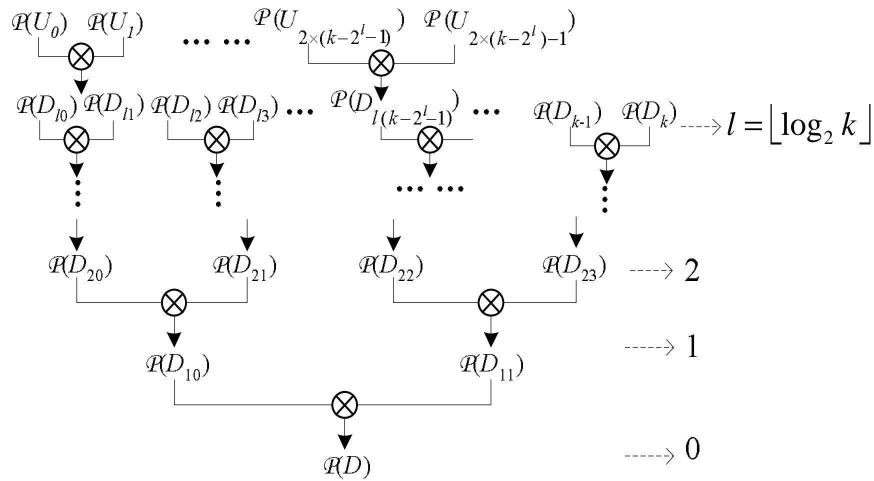


Figure 5.10: Example of recovering the original database from the units

edges is updated and will be considered in the future growth from G'.

5.2.3 Combining Frequent Subgraphs

When we have computed the set of frequent subgraphs in the units, we need to recover the complete set of frequent subgraphs in the original database. We design a **merge-join** operation to accomplish this. We first illustrate the idea behind the merge-join operation before presenting the algorithm. We also give theoretical proof to show the complete set of frequent subgraphs in the database can be losslessly recovered by the merge-join operation on the set of frequent subgraphs found in the units. Figure 5.10 shows the example to recover the original database from the units.

Suppose unit U is partitioned into two units U_1 and U_2. Let $\mathcal{P}(U_1)$ and $\mathcal{P}(U_2)$ be the set of frequent subgraphs found in the units U_1 and U_2. We want to recover the set of frequent subgraphs in the unit U, that is, P(U). We first sort the frequent subgraphs

in each unit according to their number of edges. We shall use $\mathcal{P}^k(U_i)$ to denote the set of subgraphs of size k.

First, frequent 1-edge subgraphs in the units U_1 and U_2 are simply merged since they do not share any common connective edges. We denote the resulting set of 1-edge subgraphs by $\mathcal{P}^1(U) = \mathcal{P}^1(U_1) \cup \mathcal{P}^1(U_2)$.

Next, we merge frequent 2-edge subgraphs, that is $\mathcal{P}^2(U) = \mathcal{P}^2(U_1) \cup \mathcal{P}^2(U_2)$, and join the 2-edge subgraphs based on the common connective edges to produce a set of candidate 3-edge subgraphs C^3. A subgraph isomorphism check is then carried out to remove the infrequent 3-edge subgraphs in C^3, resulting in the set of *frequent* 3-edge subgraphs, i.e., F^3.

For frequent k-edge subgraphs, $k > 2$, we obtain the set of k-edge subgraphs by merge $\mathcal{P}^k(U_1)$, $\mathcal{P}^k(U_2)$ and F^k, that is, $\mathcal{P}^k(U) = \mathcal{P}^k(U_1) \cup \mathcal{P}^k(U_2) \cup F^k$. Then, the merge-join operation is applied to find the set of candidate $(k+1)$-edge subgraphs C^{k+1}. We obtain C^{k+1} in three steps:

Step 1. Join the subgraphs in the set $\mathcal{P}^k(U_1)$ with the subgraphs in the set F^k to obtain the candidate set ${C_1}^{k+1}$;

Step 2. Join the subgraphs in the set $\mathcal{P}^k(U_2)$ with those in the set F^k to obtain the candidate set ${C_2}^{k+1}$;

Step 3. Join the subgraphs in the set F^k with themselves to get the candidate set ${C_3}^{k+1}$.

The set of candidate subgraphs C^{k+1} is the union of the set ${C_1}^{k+1}$, ${C_2}^{k+1}$ and ${C_3}^{k+1}$, that is, $C^{k+1} = {C_1}^{k+1} \cup {C_2}^{k+1} \cup {C_3}^{k+1}$. We remove the infrequent (k+1)-edge subgraphs

from C^{k+1}, resulting in F^{k+1}.

This process continues until all frequent subgraphs in the original database are discovered.

Figure 5.11 shows an example of the merge-join operation. Figure 5.11(a) shows the unit U with one graph G and its two subgraphs G_1 in U_1 and G_2 in U_2. The process of recovering $\mathcal{P}(G)$ from $\mathcal{P}(G_1)$ and $\mathcal{P}(G_2)$ is illustrated in Figure 5.11(b), where the left light grey region marks the set of subgraphs of G_1, i.e., $\mathcal{P}(G_1)$, and the dark grey region marks the set of subgraphs of G_2, $\mathcal{P}(G_2)$.

Proof of Completeness

In this section, we prove that the complete set of frequent graphs in the database $\mathcal{D}$ can be recovered when the merge-join operation is performed on the set of frequent subgraphs found in all the units. We first prove that it is possible to recover all subgraphs of a graph G from its partitioned subgraphs. Then we introduce the Apriori property of subgraphs, and proceed to show that the complete set of frequent subgraphs in the original database can be losslessly recovered even when the mining is performed on the individual smaller units.

Theorem 2 *The set of subgraphs of a graph G (i.e. $\mathcal{P}(G)$) with size of n, $n \geq 2$, can be losslessly recovered when the merge-join operation is recursively applied on its bi-partitioned subgraphs G_1 and G_2.*

Proof: We prove this by induction. Let H_n denote the hypothesis that the set of subgraphs of a graph G of size n can be losslessly recovered from its partitions G_1 and

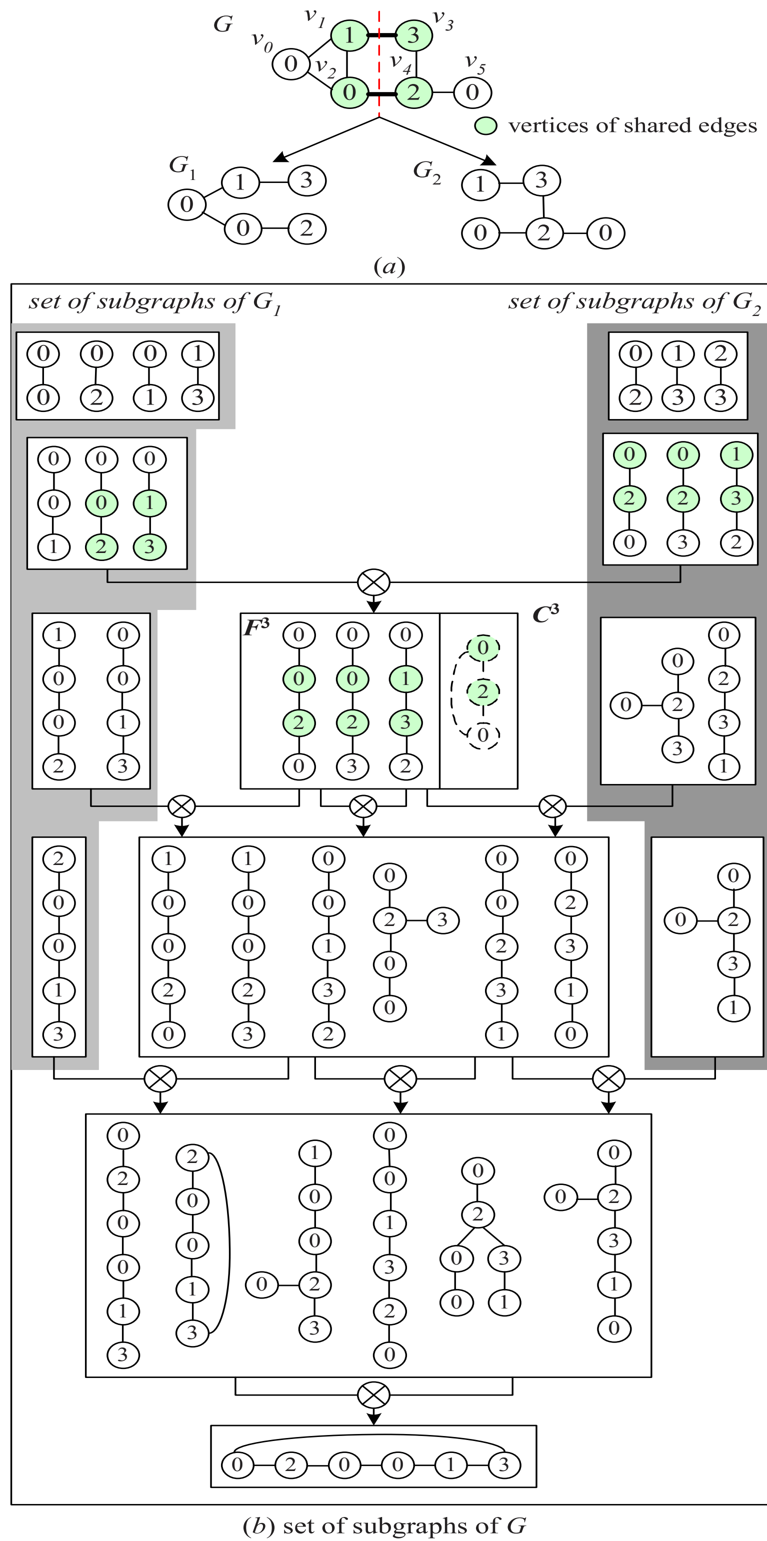


Figure 5.11: Example of the merge-join operation

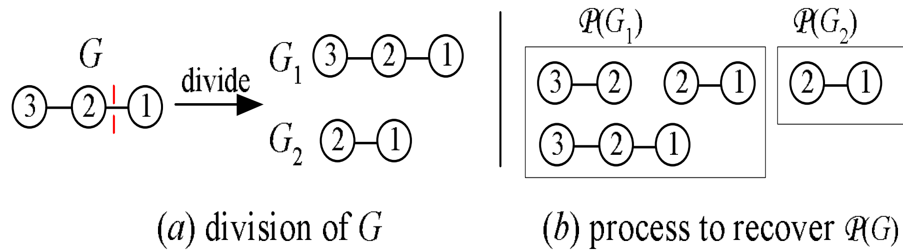


Figure 5.12: Base case

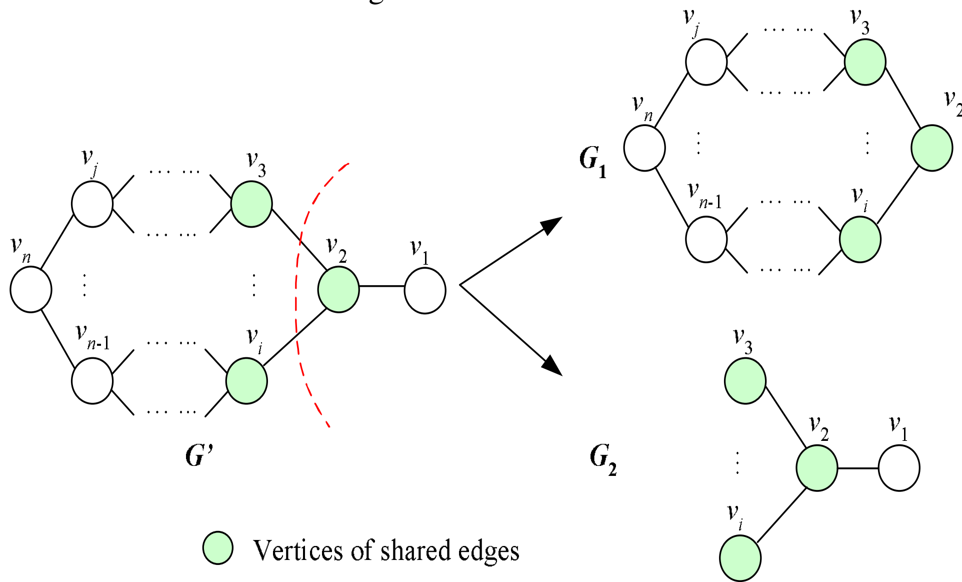


Figure 5.13: Induction step

G_2, $n \geq 2$

Base Case: $n = 2$. This is trivially true as shown in Figure 5.12. Figure 5.12(a) shows the division of the graph G into two subgraphs G_1 and G_2. The process to recover $\mathcal{P}(G)$ from $\mathcal{P}(G_1)$ and $\mathcal{P}(G_2)$ is shown in Figure 5.12(b).

Induction Step: Suppose H_n is true, we want to show that H_{n+1} is also true. If H_n is true, we know that we are able to recover all subgraphs of a graph G of size n. Now we have a graph G' of size $n + 1$. We partition graph G' into two subgraphs. Let G_1 denote the partition of size n, and let G_2 denote the partition of size i-2, $3 < i < n$

(see Figure 5.13). By our assumption, we know we can recover all subgraphs from G_1 (because G_1 is of size n). Hence, the only missing subgraphs are those involving the edge (v_1, v_2) in G_2. These subgraphs are formed by the joining of the subgraphs of G_1 and G_2, which share one of the common edges $(v_2, v_3), ..., (v_2, v_i)$ marked as grey in Figure 5.13. This step is in fact included in the merge-join operation. In other words, if H_n is true, then H_{n+1} must be true.■

Theorem 3 (Apriori Property) *If a graph G is frequent, all of its subgraphs are frequent.*

Proof: A graph G is frequent implies the number of occurrences of G exceeds some minimum support threshold. A subgraph of G must, by definition, occur in G. Hence, the number of occurrences of the subgraph must be equal to or exceed G's occurrences. In other words, the subgraph is frequent.■

Theorem 4 *Let $\mathcal{D}$ be a graph database that has been divided into k smaller units U_i, $k \geq 2$, $1 \leq i \leq k$. If we know the complete set of frequent subgraphs $\mathcal{P}(U_i)$ in each unit U_i, $1 \leq i \leq k$, we can determine the complete set of frequent subgraphs $\mathcal{P}(\mathcal{D})$ in $\mathcal{D}$.*

Proof: Let H_{k-1} denote the hypothesis that the complete set of frequent subgraphs of the unit U can be losslessly recovered from the set of frequent subgraphs in its k-1 subunits U_i $(1 \leq i \leq k-1)$.

Base case: $k = 2$, that is $\mathcal{D}$ is divided into two units U_1 and U_2. Given $\mathcal{P}(U_1)$ and $\mathcal{P}(U_2)$, according to the **Theorem 1** and **Theorem 2**, we can losslessly recover the set

of frequent subgraphs in $\mathcal{D}$, that is we can get the complete set of frequent graphs $\mathcal{P}(\mathcal{D})$.

Induction step: $k > 2$. Suppose H_{k-1} is true, we want to show that H_k is also true. Given the unit U is divided into k subunits U_i ($0 \leq 0 \leq k$). If H_{k-1} is true, we know that we are able to recover the complete set of frequent subgraphs of the unit U' from the set of frequent subgraphs in its $k-1$ subunits U_j ($0 \leq\leq k-1$). Now we have the set $\mathcal{P}(U')$ and $\mathcal{P}(U_k)$. According to the base case, we know that we can losslessly recover $\mathcal{P}(U)$ from $\mathcal{P}(U')$ and $\mathcal{P}(U_k)$. In other words, if H_{k-1} is true, then H_k must be true.■

From **Theorem 3**, we note that the completeness of the frequent graphs in the database $\mathcal{D}$ depends on the completeness of the frequent subgraphs in the units. Hence, we can reduce the support threshold used to discover frequent subgraphs in units to get the complete set of frequent graphs in the units.

5.2.4 Framework of PartMiner

Figure 5.14 shows the outline of PartMiner. It takes as input the database $\mathcal{D}$, the minimum support sup, and the number of units k, and outputs the set of frequent subgraphs in $\mathcal{D}$. Algorithm PartMiner works in two phases. In the first phase, it divides the database $\mathcal{D}$ into set of units of proper and manageable size(line 1). In the second phase (lines 2-17), PartMiner first calls the algorithm ADIMINE to find the set of frequent subgraphs in the k units with the support threshold sup/k (lines 2-17). The reason that we use the lower support threshold for mining the units is to guarantee that the subgraphs that are frequent in the original database are also frequent in the units. After mining the units, line 14 recursively calls the procedure *MergeJoin* (see Figure 5.15)

Algorithm PartMiner

Input: $\mathcal{D}$, graph database;

sup: minimum support;

k: number of units

Output: $\mathcal{P}(\mathcal{D})$: set of frequent subgraphs in $\mathcal{D}$.

*/*Phase1: dividing the database into k units*/*

1: DBPartition($\mathcal{D}, k$);

*/*Phase2: combining the results of k units*/*

2: $l = \lfloor log_2 k \rfloor$;

3: $i = l + 1$;

4: **for**($j = 0; j < k - 2^l; j++$) {

5: Mining U_{2j} and U_{2j+1} using Gastion;

6: $\mathcal{P}(D_{i-1,j})$ = MergeJoin($D_{i-1,j}, \mathcal{P}(U_{2j}), \mathcal{P}(U_{2j+1}), \frac{sup}{k}$)

7: }

8: $i--$;

9: **while** ($i > 0$) {

10: **for**($j = 0; j < 2^i; j = j + 2$) {

11: **if**($i == log_2 k \wedge j > k - 2^l - 1$)

12: Mining D_{ij} and $D_{i,j+1}$ using ADIMINE;

13: $S = D_{i-1,\frac{j}{2}}$;

14: $\mathcal{P}(S)$ = MergeJoin($S, \mathcal{P}(D_{i,j}), \mathcal{P}(D_{i,j+1}), \frac{sup}{2^i}$)

15: }

16: $i--$;

17: }

Figure 5.14: Outline of the PartMiner algorithm

Procedure MergeJoin

Input: S, the graph dataset; $\mathcal{P}(S_0)$, set of frequent subgraphs in S_0;
$\mathcal{P}(S_1)$, set of frequent subgraphs in S_1; sup, minimum support.

Output: $\mathcal{P}(S)$, the set of frequent subgraphs in the dataset S.

1: $\mathcal{P}^1(S)$ = {frequent 1-edge subgraphs in S};
2: $P = \mathcal{P}(S_0) \cup \mathcal{P}(S_1) \backslash \mathcal{P}^1(S)$;
3: Pruning graphs in $\mathcal{P}(S_0)$ and $\mathcal{P}(S_1)$ with P;
4: $\mathcal{P}^2(S) = \mathcal{P}^2(S_0) \cup \mathcal{P}^2(S_1)$;
5: C^3 = Join($\mathcal{P}^2(S_0), \mathcal{P}^2(S_1)$);
6: F^3 = CheckFrequency(C_3, sup);
7: **for** ($k = 3; F^k \neq \emptyset; k++$){
8: $\mathcal{P}^k(S) = \mathcal{P}^k(S_0) \cup \mathcal{P}^k(S_1) \cup F^k$;
9: C_1^{k+1} = Join($\mathcal{P}^k(S_0), F^k$);
10: C_2^{k+1} = Join($\mathcal{P}^k(S_1), F^k$);
11: C_3^{k+1} = Join(F^k, F^k);
12: $C^{k+1} = C_1^{k+1} \cup C_2^{k+1} \cup C_3^{k+1}$;
13: F^{k+1} = CheckFrequency(C^{k+1}, sup);
14: }
15: $\mathcal{P}(S) = \bigcup \mathcal{P}^k(S)$

Figure 5.15: Outline of the MergeJoin procedure

Table 5.1: Meaning of symbols

Symbol	Meaning
U_i	original unit
U_i'	updated unit
$D_{i,j}$	set of intermediate subgraphs generated during the graph partitioning process
$\mathcal{P}(U_i)$	set of frequent subgraphs in the unit U_i
$\mathcal{P}^k(U_i)$	set of frequent k-edge subgraphs in the unit U_i
C^{k+1}	set of candidate (k+1)-edge subgraphs
F^{k+1}	set of frequent (k+1)-edge subgraphs
$\overline{\mathcal{P}}$	difference between $\mathcal{P}(U_i)$ and $\mathcal{P}(U_i')$
P	prune set

to combine the results of $D_{i,j}$ and $D_{i,j+1}$ $(0 \leq i \leq log_2 k, 0 \leq j \leq k)$ together. The process continues until the set of frequent subgraphs of $\mathcal{D}$ (i.e., $\mathcal{P}(\mathcal{D})$) is found. The symbols and their meaning used in the algorithm are listed in Table 5.1.

5.2.5 Handle Updates Using PartMiner

The motivation for the proposed approach is to effectively deal with graphs in the presence of updates. It is important to isolate those vertices and edges that are changed frequently into a small set of subgraphs so that the number of subgraphs that need to participate in the incremental mining process is minimized.

Recall that PartMiner finds the set of frequent subgraphs by first partitioning the database into several units, then mining the set of frequent subgraphs in each of these units, and finally merging the results of the units with the merge-join operation. If we are able to isolate the updated vertices and/or edges of a graph to a small set of subgraphs, we will be able to focus only on this set of subgraphs instead of mining on

the entire database each time an update occurs.

Patterns that are affected by updates can be classified into three categories:

1. UF (unchange frequencies): the set of patterns whose frequencies remain unchanged;

2. FI (frequent to infrequent): the set of previously frequent patterns that have become infrequent; and

3. IF (infrequent to frequent): the set of previously infrequent patterns that have become frequent.

Algorithm PartMiner can be easily extended to discover UF and IF.

The extension idea is as follows: When the database $\mathcal{D}$ is updated, for each updated unit U_i' (U_i is the original unit before the updates), we re-execute the main memory algorithm to find the new set of frequent subgraphs $\mathcal{P}(U_i')$. We then compare the set $\mathcal{P}(U_i')$ against the set $\mathcal{P}(U_i)$. If they are different, we do the following:

1. We keep the subgraphs that appear in the set $\mathcal{P}(U_i)$ but not in the set $\mathcal{P}(U_i')$ in the prune set P. For each subgraph in the prune set P, we check to see if it exists in any other $\mathcal{P}(U_j')$ $(0 \leq j \leq k \wedge j \neq i)$. If it exists, we remove it from P. Otherwise, we do nothing. Note that P keeps track of all subgraphs that may potentially change from frequent to infrequent.

2. Next, we check the set of subgraphs in the pre-updated database $\mathcal{D}$, i.e., $\mathcal{P}(\mathcal{D})$ against the prune set P. We remove those subgraphs in $\mathcal{P}(\mathcal{D})$ that are the super-

graphs of some graphs in P, and add them to the set FI. The pruned $\mathcal{P}(\mathcal{D})$ is denoted as $\mathcal{P}(\mathcal{D})'$.

When all of the updated units are checked, we perform a final merge-join operation to obtain the updated results. However, since there are some graphs whose frequencies are not changed during the updates, we do not need to check them. This can result in further optimization.

The *IncPartMiner* algorithm for handling updates is shown in Figure 5.16. Line 1 scans the updated database $\mathcal{D}'$ to get the set of frequent edges, i.e., $\mathcal{P}^1(\mathcal{D}')$. Line 2 compares the set of frequent edges in the original database $\mathcal{D}$, i.e., $\mathcal{P}^1(\mathcal{D})$ with the set $\mathcal{P}^1(\mathcal{D}')$, and add the subgraphs that exist in $\mathcal{P}^1(\mathcal{D})$ but not in $\mathcal{P}^1(\mathcal{D}')$ into the prune set P.

Next, for each unit U_i' that consists of the updated vertices, we re-execute the ADIMINE algorithm to mine the set of frequent subgraphs (i.e., $\mathcal{P}(U_i')$), and compare it with the set of subgraphs in the unit U_i, i.e., $\mathcal{P}(U_i)$. Those potentially infrequent subgraphs are added to the prune set P (lines 3-9). Line 10 then prunes the subgraphs in the set $\mathcal{P}(\mathcal{D})$ with the prune set P, which results in the pruned set $\mathcal{P}(\mathcal{D})'$. Then, lines 11-12 further use the pruned set $\mathcal{P}(\mathcal{D})'$ to prune the candidate graphs when carrying out the merge-join operation on the updated results of the units (see Figure 5.17). Finally, the three sets of the subgraphs, i.e., UF, FI and IF, are output (lines 13-15), where IF consists of the graphs in $\mathcal{P}(\mathcal{D}')$ but not in $\mathcal{P}(\mathcal{D})$, UF is the set of graphs in $\mathcal{P}(\mathcal{D}')$ but not in IF, and FI contains all the graphs G in $\mathcal{P}(\mathcal{D})$ such that there is a graph G' in P that is a subgraph of G.

Algorithm IncPartMiner

Input: $\mathcal{D}'$, the updated database

$\mathcal{P}(U_i)$, the set of frequent subgraphs in the unit U_i $(0 \leq i \leq k)$

$\mathcal{P}(\mathcal{D})$, old set of the frequent subgraphs in $\mathcal{D}$

sup, minimum support;

set, a setword to indicate the reexamined units

Output: UF, IF, FI: 3 sets of patterns;

1: $\mathcal{P}^1(\mathcal{D}')$ = {frequent 1-edge subgraphs in $\mathcal{D}'$};

2: $P = \mathcal{P}^1(\mathcal{D}) \backslash \mathcal{P}^1(\mathcal{D}')$;

3: **for** ($i = 0$; $i < k$; i++) {

4: **if** $(set(i) \neq 1)$ **continue**;

5: $\mathcal{P}(U_i')$ = mining the unit U_i using ADIMINE;

6: **if** $(\mathcal{P}(U_i') \backslash \mathcal{P}(U_i) \neq \emptyset)$ recombine = 1

7: $\overline{\mathcal{P}} = \mathcal{P}(U_i) \backslash \mathcal{P}(U_i')$;

8: $P = P \cup \{G \in \overline{\mathcal{P}} | \forall j = 0..k \wedge j \neq i, G \notin \mathcal{P}(U_j)\}$;

9: }

10: $\mathcal{P}(\mathcal{D})' = \{G \in \mathcal{P}(\mathcal{D}) | \forall G' \in P, G' \nprec G\}$;

11: **if** (recombine)

12: $\mathcal{P}(\mathcal{D}')$ = calling IncMergeJoin to join the units' results ;

13: $IF = \mathcal{P}(\mathcal{D}') \backslash \mathcal{P}(\mathcal{D})$;

14: $UF = \mathcal{P}(\mathcal{D}') \backslash IF$

15: $FI = \{G \in \mathcal{P}(\mathcal{D}) | \exists G' \in P, G' \prec G\}$;

Figure 5.16: Outline of the IncPartMiner algorithm

Procedure IncMergeJoin ($\mathcal{D}', \mathcal{P}(S_0), \mathcal{P}(S_1), \mathcal{P}(\mathcal{D})$)

1: $\mathcal{P}^2(\mathcal{D}') = \mathcal{P}^2(S_0) \cup \mathcal{P}^2(S_1)$;
2: C^3 = Join($\mathcal{P}^2(S_0), \mathcal{P}^2(S_1)$);
3: **for each** $G \in C^3 \wedge G \in \mathcal{P}(\mathcal{D})'$ {
4: $C^3 = C^3 - \{G\}$;
5: $F^3 = F^3 \cup \{G\}$;
6: }
7: $F^3 = F^3 \cup \{G \in C^3 | G.sup \geq sup)\}$;
8: **for** $(k = 3; F^k \neq \emptyset; k++)${
9: $\mathcal{P}^k(\mathcal{D}') = \mathcal{P}^k(S_0) \cup \mathcal{P}^k(S_1) \cup F^k$;
10: C_1^{k+1} = Join($\mathcal{P}^k(S_0), F^k$);
11: C_2^{k+1} = Join($\mathcal{P}^k(S_1), F^k$);
12: C_3^{k+1} = Join(F^k, F^k);
13: $C^{k+1} = C_1^{k+1} \cup C_2^{k+1} \cup C_3^{k+1}$;
14: **for each** $G \in C^{k+1} \wedge G \in \mathcal{P}(\mathcal{D})'$ {
15: $C^{k+1} = C^{k+1} - \{G\}$;
16: $F^{k+1} = F^{k+1} \cup \{G\}$;
17: }
18: $F^{k+1} = F^{k+1} \cup \{G \in C^{k+1} | G.sup \geq sup)\}$;
19: }
20: **return** $\bigcup \mathcal{P}^k(\mathcal{D}')$

Figure 5.17: Outline of the IncMergeJoin procedure

5.3 Experimental Study

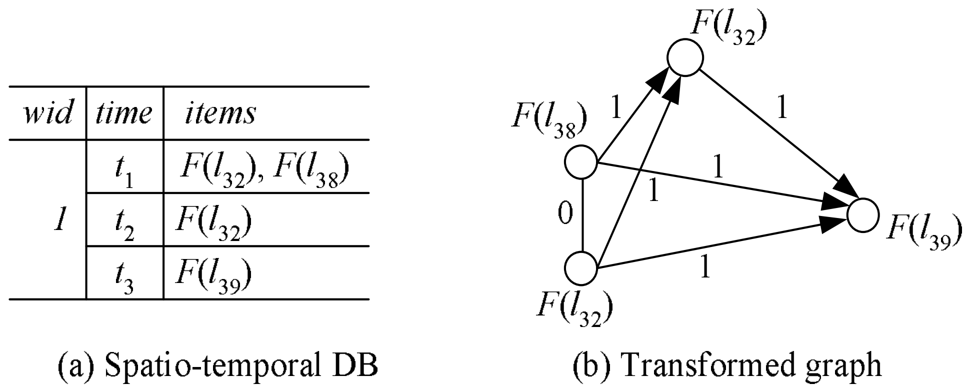
In this section, we report the performance study of the proposed algorithms. The algorithms are implemented in C++. All the experiments are conducted on a P4 2.8GHZ CPU, 2.5GB RAM and 73GB hard disk. The operating system is Redhat Linux 9.0.

- **Synthetic Dataset:** We use the synthetic data generator described in [WWP$^+$04]. The data generator takes as input five parameters D, N, T, I and L, whose meanings are shown in Table 5.2. For example, the dataset $D50kT20$ $N20L200I5$ indicates that the dataset is made up of 50k graphs, the average number of edges in each graph is 20, and there are 20 possible labels and 200 potentially frequent kernels. The average number of edges in the frequent kernels is 5.

Table 5.2: Parameters of synthetic data generator

Parameter	Meaning	Range
D	total number of graphs in the data set	100k - 1000k
N	number of possible labels	20, 30, 40, 50
T	average number of edges in graphs	10, 15, 20, 25
I	average number of edges in potentially frequent graph patterns	2, 3, 4, 5, 6, 7, 9
L	number of potentially frequent kernels	200

- **Real-life Dataset:** We transformed the forest fire dataset in Chapter 4 into graphs. Each node corresponds to a forest fire event occurring in a location. Two nodes are connected via an edge if and only if the event $F(l_i)$ is related to event $F(l_j)$, that is the event $F(l_i)$ and the event $F(l_j)$ are close in time and location. We define $R = 1$ and $W = 20$ days. The spatio-temporal database have been transformed into



wid	*time*	*items*
1	t_1	$F(l_{32}), F(l_{38})$
	t_2	$F(l_{32})$
	t_3	$F(l_{39})$

Figure 5.18: Example of transformed graphs

a graph database with 40 graphs. The graphs represents non-overlapping region. Figure 5.18 shows an example of the transformed graphs from the spatio-temporal database.

5.3.1 Performance Study on Static Datasets

In this section, we study the performance of PartMiner in a static environment. We compare it with the ADIMINE algorithm [WWP+04][1]. In the following sets of experiments, the parameter L is fixed at 200, and the parameter k is set to 2. The size of the memory is set to unlimited.

Effect of Partitioning Criteria

We first study the effect of using different partitioning criteria. There are three ways to partition graphs: (1) Isolate the updated vertices into the same subgraphs, i.e., $\lambda_1 = 1$, $\lambda_2 = 0$ (Partition1); (2) Minimize connectivity between subgraphs, i.e., $\lambda_1 = 0$, $\lambda_2 = 1$

[1] We obtain the executable ADIMINE from the authors

(Partition2); and (3) Isolate the updated vertices *AND* minimize the connectivity, i.e., λ_1 = 1, λ_2 = 1 (Partition3). We also use METIS [KK98] to partition the graphs before mining the graphs in the database. The algorithms in METIS are based on multilevel graph partitioning. They reduce the size of the graph by collapsing vertices and edges, partition the smaller graphs, and then uncoarsen it to construct a partition for the original graph.

Figure 5.19 shows the results. The proposed graph partitioning algorithms perform better than METIS. Specifically, Partition2 gives the best performance. This is because Partition2 minimizes the number of connective edges between subgraphs, which in turn reduces the number of joining graphs in the merge-join operations. This means that in the static dataset, the criteria to partition graphs based on frequency of updated vertices has minimal effect on the performance of PartMiner.

Varying Minimum Support

Next, we study the performance of PartMiner by varying minimum support from 1% to 6%. The results are shown in Figure 5.20. Compared to ADIMINE, we observe that PartMiner needs less time to find the complete set of frequent subgraphs when minimum support is greater than 1.5%. When minimum support is less than 1.5%, we find that PartMiner needs more time than ADIMINE does to find the complete set of frequent graphs. This is because when minimum support decreases, more subgraphs become frequent, and the subgraphs also become more complex. As a result, PartMiner needs more time to count the frequency of the subgraphs. In contrast, the index structure of

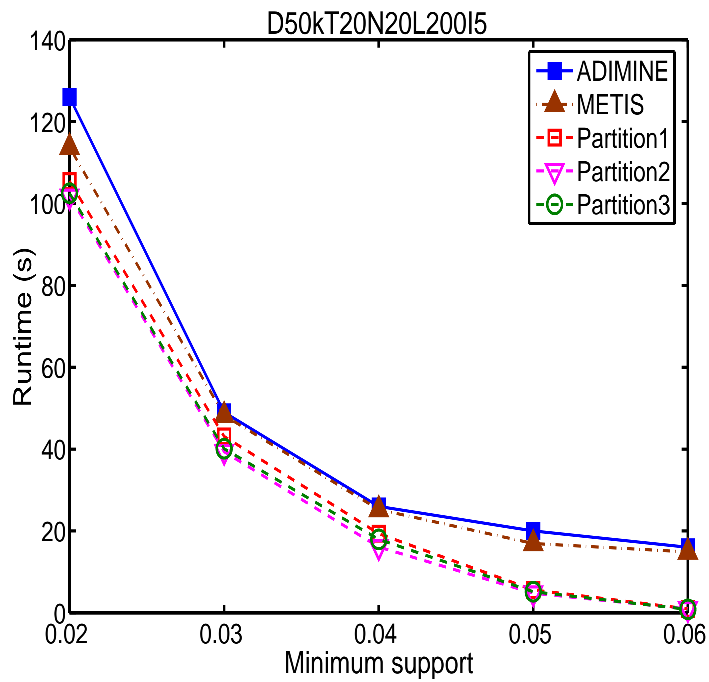


Figure 5.19: Effect of partitioning criteria

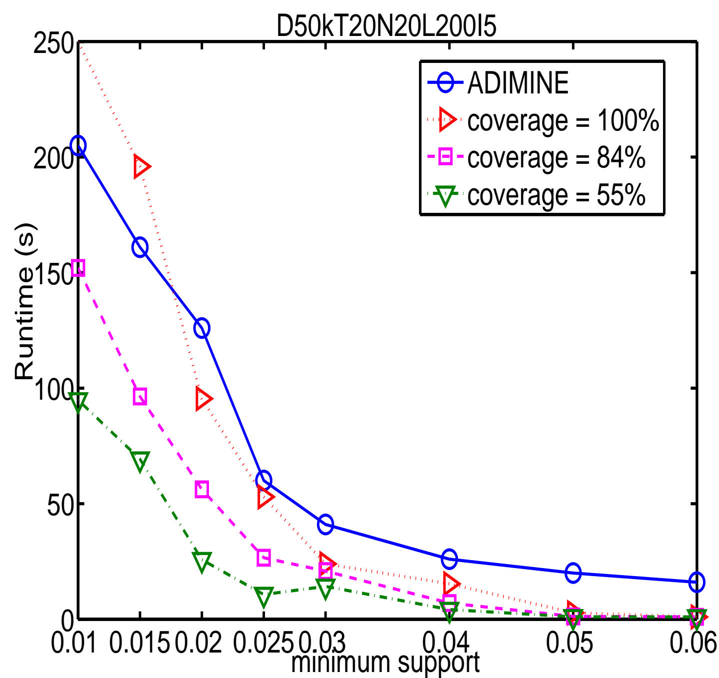


Figure 5.20: Runtime vs. parameter minsup

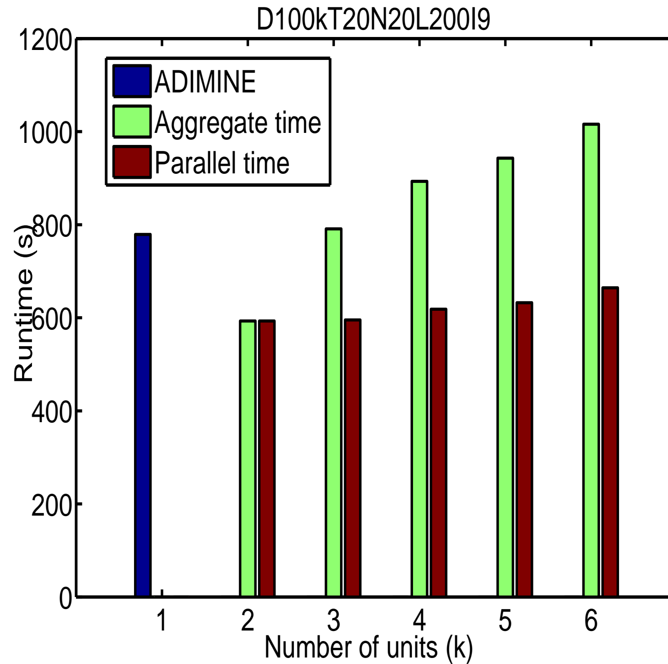


Figure 5.21: Runtime vs. parameter k

the ADIMINE is advantageous at this time. However, if we only want to retrieve an approximate result of the frequent subgraphs, e.g., coverage = 84% or 55% (coverage indicates the percentage of the frequent subgraphs found in the complete set), then PartMiner outperforms ADIMINE.

Effect of Number of Units k

We test the performance of the PartMiner by varying the number of units from 2 to 6. Recall that we have divided the database into units and these units can be processed in parallel. This implies that our PartMiner can be executed either in the serial mode or the parallel mode. In the serial mode, we measure aggregate time which is computed by adding the time spent in all the units together. In the parallel mode (with 1 CPU),

the units are executed concurrently and we simply take the maximum of the time spent in the units.

From the results in Figure 5.21, we see that the parallel running PartMiner is faster than ADIMINE in finding the complete set of frequent subgraphs. In other words, we could achieve better performance to mine the frequent subgraphs by partitioning the databases into the subunits than that in the original databases.

We also observe that more time is needed to find the complete set of subgraphs in the database as the parameter k increases. The reason is that with the increasing of the number of units, we need more merge-join operations to recover the complete set of frequent subgraphs. For example in Figure 5.10, we can see that we only need one merge-join operation to recover the complete set of frequent subgraphs for 2 units. If we divide the database into 6 units, we need four merge-join operations to recover the complete set of the frequent subgraphs. Hence, we need more time to get the final set of frequent subgraphs when we divide the original database into more units.

Scalability

In this set of the experiments, we evaluate the performance of PartMiner by varying the parameters T, I and D of the synthetic data generator, which will affect the size of the graphs. Figure 5.22, Figure 5.23 and Figure 5.24 show that PartMiner scales linearly with the parameters T, I and D, and is faster in finding the complete set (or approximate results) of the frequent subgraphs compared to ADIMINE.

Figure 5.22 shows the results when the parameter T varies from 10 to 25. We

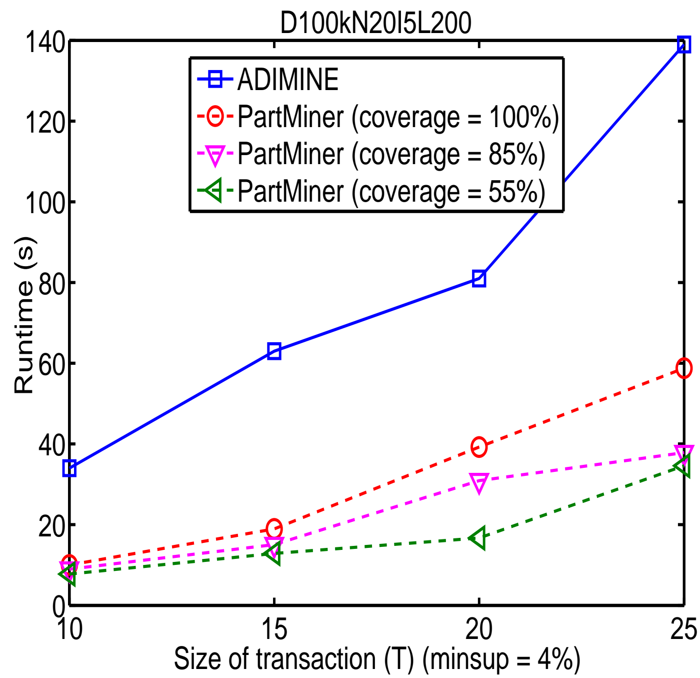


Figure 5.22: Varying parameter T

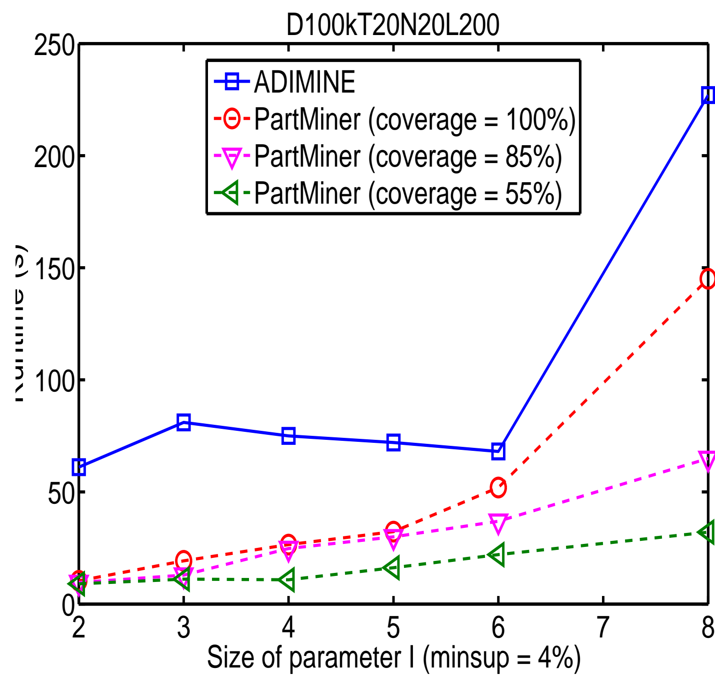


Figure 5.23: Varying parameter I

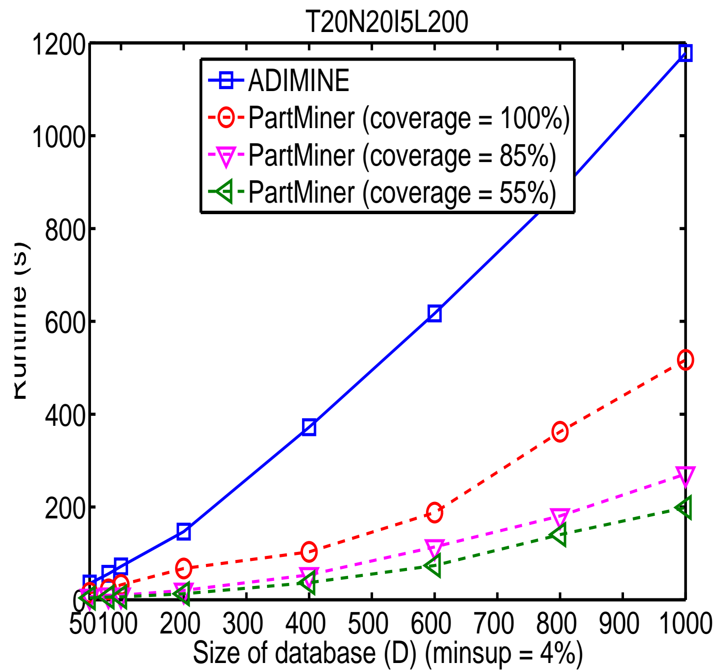


Figure 5.24: Varying parameter D

observe that PartMiner needs more time to find the set of frequent graphs as T increases. This is to be expected since the frequent subgraphs in the final results tend to be more complex with the growth of T.

Figure 5.23 shows the performance of PartMiner when the parameter I varies from 2 to 8. The results indicate that to find the set of frequent graphs, PartMiner is slower when I increases to more than 7. This is because the size of the potentially frequent graphs in the dataset becomes larger. As a result, the size of the frequent subgraphs in the database tends to be larger, and PartMiner needs more time to find the larger frequent subgraphs.

The effect of varying the size of the database from 50,000 to 1000,000 is shown in Figure 5.24. We observe that PartMiner scales linearly with the size of the database.

5.3.2 Performance Study on Dynamic Datasets

In this section, we study the performance of IncPartMiner When updates occur. We implement a data generator which updates the database in three different ways:

- Updating the vertex/edge labels with existing or new labels. For example, updating the vertices/edges with label l in the graphs to the label l';
- Adding a new edge with an existing or a new label between two vertices v_i and v_j in the graph G. For example, adding an edge with the label l' between the vertices v_0 and v_1 if there is no edge between them.;
- Adding a new vertex v, and a new edge $e(v, v_i)$ with the existing or new labels on the vertex v_i in the graph G.

We update the dataset D50kT20N20L200I5 using the three different ways, and evaluate IncPartMiner by varying the percentage of the graphs updated in the database from 20% to 80% with the number of units in the database fixed at 2.

Effect of Partitioning Criteria

We first test the performance of IncPartMiner by varying the partitioning criteria. Figure 5.25 shows the results. We observe Partition3, that is isolating the updated vertices AND minimizing the connective edges, yields the best performance in the dynamic dataset. This is because Partition3 not only reduces connectivity among units, but also tries to isolate updated vertices into a minimum number of units, and minimizes the number of units needed to be re-mined and re-examined in the merge-join operation.

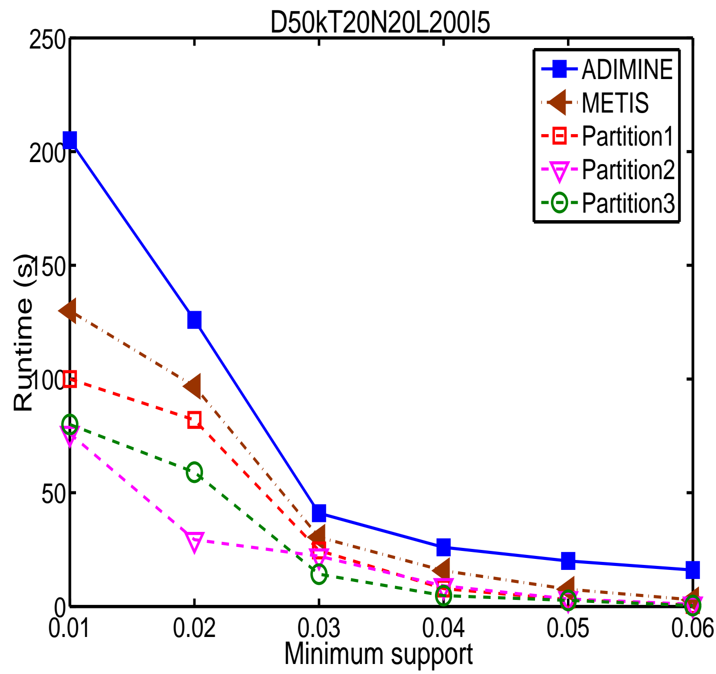


Figure 5.25: Effect of partitioning criteria

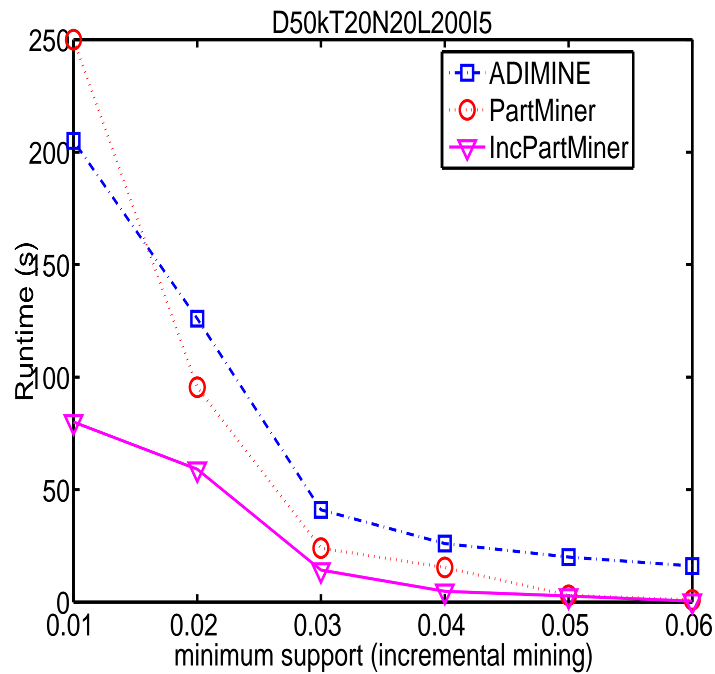


Figure 5.26: Runtime vs. parameter minsup

Compared to Figure 5.19 about the static datasets, we note that the criteria to partition graphs based on frequency of updated vertices takes its advantages on the performance of IncPartMiner in the dynamic dataset.

Varying Minimum Support

We evaluate the IncPartMiner algorithm by varying minimum support from 1% to 6%. Figure 5.26 shows the results of finding the complete set of graphs in the database. We note that IncPartMiner is more efficient in finding the new set of frequent graphs compared to ADIMINE and PartMiner.

This is because IncPartMiner makes use of the pruning results of the pre-updated database to prune those candidate graphs that remain unchanged. It also uses the differences between the pre-updated results and the updated results of the units to prune those subgraphs that have become infrequent. This results in much savings. In contrast, ADIMINE and PartMiner have to re-mine the database to find the subgraphs that have been changed, and need to re-examine both the changed and unchanged subgraphs.

Effect of Number of Units k

Next, we test IncPartMiner with different number of units. We vary the parameter k from 2 to 6. Figure 5.27 indicates that IncPartMiner runs faster than ADIMINE when mining dynamic datasets both in serial mode and in parallel mode. We believe this is due to the fact that IncPartMiner uses the pruned results of the pre-updated database to avoid generating candidate graphs that remain unchanged. This enables the IncPart-

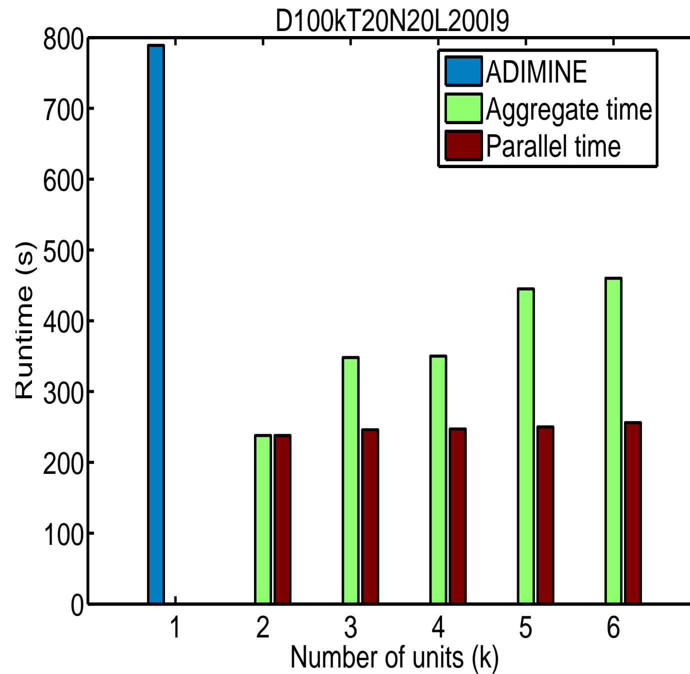


Figure 5.27: Runtime vs. parameter k

Miner to check only those subgraphs that were infrequent but have become frequent in the updated database. In contrast, ADIMINE has to rebuild the ADI index structure and re-mine the subgraphs, including changed subgraphs and unchanged subgraphs.

Effect of Various Types of Updates

In this set of the experiments, we evaluate the performance of IncPartMiner by varying the updating coverage in the database from 20% to 80%. Figure 5.28 shows the results of updating the labels of the vertices (or edges) to existing and new labels. Figure 5.29 and Figure 5.30 show the results of adding new edges/vertices to the existing/new labels. The results confirm that IncPartMiner outperforms ADIMINE in mining frequent graphs in dynamic datasets.

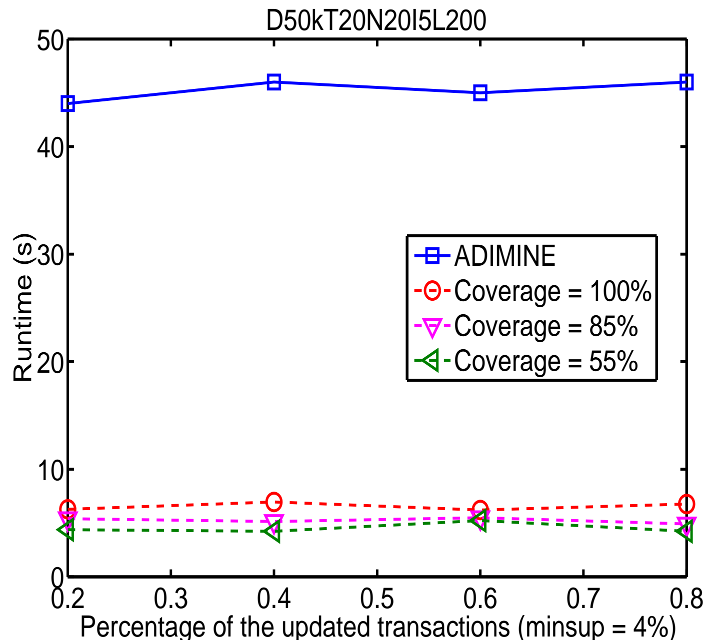


Figure 5.28: Updating the node/edge labels

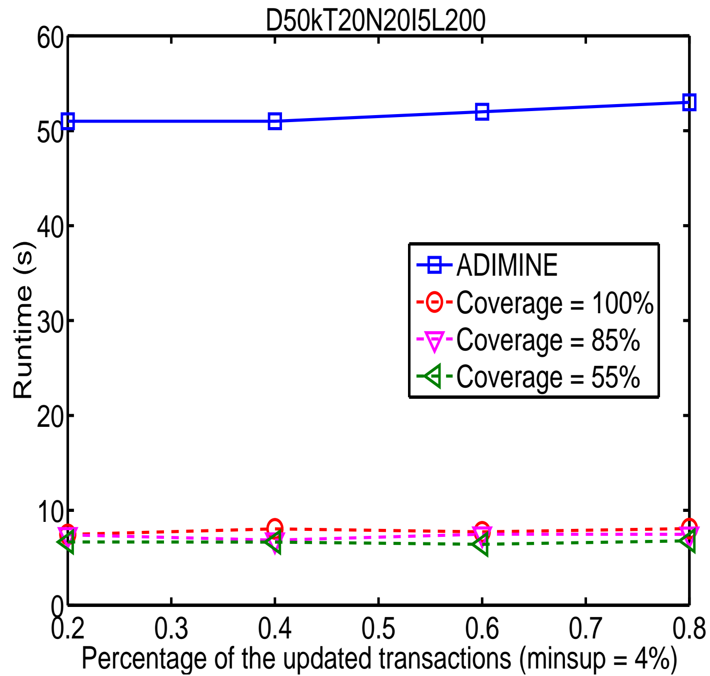


Figure 5.29: Adding new edges between two vertices

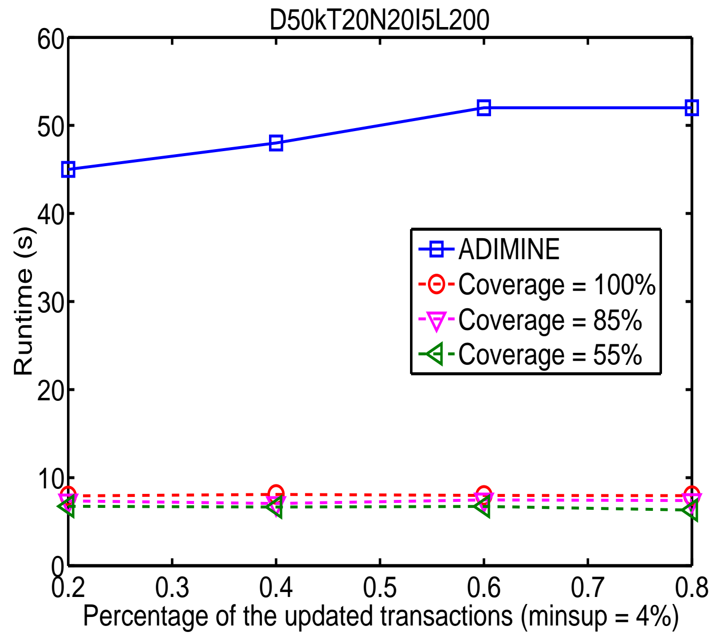


Figure 5.30: Adding new vertex with an edge to existing vertices

We observe that the updating coverage of the database has little effect on the performance of IncPartMiner. In other words, IncPartMiner scales linearly with the size of the graph and the number of labels in the database. These advantages of IncPartMiner are due to the pruning techniques we employ. IncPartMiner focuses on patterns in the set IF, that is, patterns which were infrequent but now tend to be frequent. The partitioning process handles the patterns in UF while patterns in FI can be determined from the results of the pre-updated databases.

5.4 Experiments on Real-life Dataset

We also discovered meaningful frequent subgraphs in the real-life forest fire dataset. Figure 5.31 shows a sample of the frequent subgraphs found in the real-life dataset.

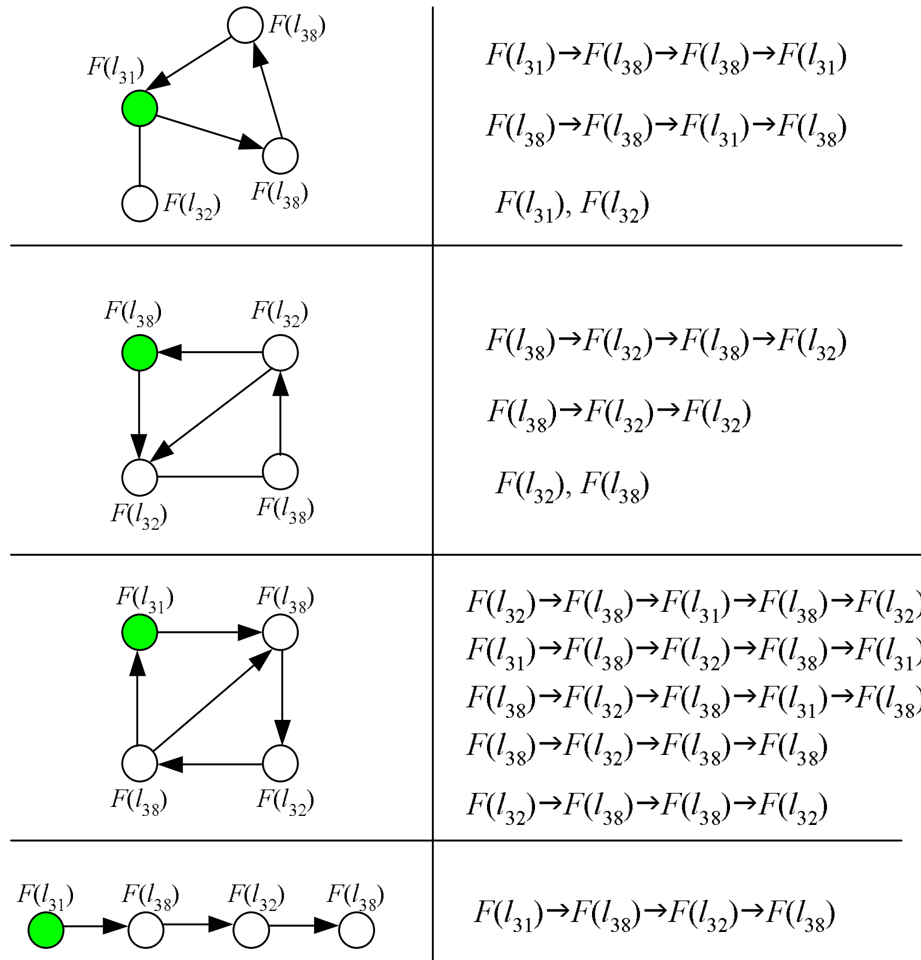


Figure 5.31: Interesting patterns found in real-life dataset

From the frequent subgraphs, we also mapped them back into flow patterns. The corresponding flow patterns are shown in the right column of Figure 5.31.

5.5 Summary

In this chapter, we have presented a partition-based algorithm *PartMiner* for discovering the set of frequent subgraphs. Each graph in the database is partitioned into smaller subgraphs. This enables PartMiner to avoid the thrashing of memory-based algorithms.

Moreover, by exploring the cumulative information of units, PartMiner can effectively reduce the number of candidate graphs. Experimental results verify that PartMiner can find frequent subgraphs efficiently and scalably.

We also present IncPartMiner, an extended version of PartMiner that handles updates in graph databases. The IncPartMiner uses pruning results of pre-updated databases to avoid generating candidate graphs that remain unchanged. It only checks those subgraphs that were infrequent but tend to be frequent in updated databases, instead of re-examining both changed and unchanged subgraphs as existing algorithms do. This leads to tremendous cost savings. The experimental results also verify that IncPartMiner performs much better than ADIMINE and PartMiner in finding graphs when updates occur.

Chapter 6

Conclusions and Future Work

Association rule mining in spatial databases and temporal databases have been studied extensively in data mining research. Most previous studies have found interesting patterns in either spatial information or temporal information; few studies have handled both efficiently. Meanwhile, developments in spatio-temporal databases and spatio-temporal applications have prompted data analysts to turn their focus to spatio-temporal patterns that explore both spatial and temporal information. In this thesis, we have introduced new classes of spatio-temporal patterns by incorporating spatial information or temporal information into existing work, and we have developed efficient and effective algorithms for mining these spatio-temporal patterns. We summarize our contributions as follows:

- First, we have devised a method to discover topological patterns by imposing temporal constraints into the process for mining collocation patterns. We have designed an algorithm called TopologyMiner to find topological patterns, and

presented a summary structure to summarize a database by recording instances' count information in a cube. With the summary structure, TopologyMiner finds topological patterns in a depth-first manner and follows the pattern-growth methodology. We have also studied the problem of mining the geographical features of topological patterns. Experimental studies indicate that TopologyMiner could find topological patterns efficiently and scalably, outperforming existing Apriori-like algorithms by a few orders of magnitude.

- Second, we have studied the problem of discovering spatial sequence patterns. We have presented two new classes of spatial sequence patterns, called *flow patterns* and *generalized spatio-temporal patterns* to describe the change of events over space and time, which are useful to the understanding of many real-life applications. We have designed two algorithms, FlowMiner and GenSTMiner, to find these two classes of spatial sequence patterns. FlowMiner utilizes temporal relationships and spatial relationships amid events to generate flow patterns. GenSTMiner is based on the idea of the pattern growth approach and finds generalized spatio-temporal patterns in a depth-first manner. Our performance studies show that the proposed algorithms are both scalable and efficient. Experiments on real-life datasets also reveal some interesting flow patterns and generalized spatio-temporal patterns.

- Finally, we have studied the problem of mining arbitrary spatio-temporal patterns by modeling spatio-temporal data as graphs. We have designed a partition-based

approach called *PartMiner* for graph mining. PartMiner utilizes the cumulative information of partitions to effectively reduce the number of candidate graphs. We have also extended PartMiner to handle frequent updates in the database. The extended version called IncPartMiner uses the pruning results of pre-updated databases to avoid generating unchanged candidate graphs. IncPartMiner only checks those subgraphs that were infrequent but tend to be frequent in updated databases, instead of re-examining both changed and unchanged subgraphs as existing algorithms do. This leads to tremendous cost savings. The experimental results indicate that PartMiner is effective and scalable in finding frequent subgraphs, outperforming existing algorithms in updated databases.

6.1 Future Research Directions

While this thesis has shown association rule mining to be a promising tool for spatio-temporal data analysis, there are a number of issues that need to be further investigated:

- *Data integration and data classification.* Real world spatio-temporal data tends to be large and is obtained from heterogeneous data sources. How to integrate data from different data sources at different levels is an increasing problem that extends beyond spatio-temporal association rule mining and into many types of spatio-temporal statistical analysis. Hence, discovering knowledge from real-world spatio-temporal applications calls for data integration and data classification.

- *Representation and calculation of spatial relationships*. In this thesis, we have focused on relationships of spatial coincidence and distance. However, there are other types of spatial relationships that may be used in spatio-temporal association rule mining, such as direction, topological relationships, etc. A structured experiment comparing different spatial relationship types in association rule mining would illustrate the impact of choice of spatial relationship type on mining results.

- *Representation of spatio-temporal data*. Developing spatio-temporal mining methods should go hand in hand with efficient and effective spatio-temporal data mining. Each spatio-temporal representation approach and the corresponding data structures may impose some unique challenges on data mining algorithms/methods.

- *Application in special types of spatio-temporal database*. We consider to extend our algorithms to special types of spatio-temporal databases, such as those collected from sensor networks (for environmental monitoring) etc.

Bibliography

[AC01] J. Aach and G.M. Church. Aligning gene expression time series with time warping algorithms. *Bioinformatics*, pages 495–508, 2001.

[AGYF02] J. Ayres, J. Gehrke, T. Yiu, and J. Flannick. Sequential pattern mining using a bitmap representation. *Proceedings of the ACM SIGKDD international conference on Knowledge discovery and data mining*, 2002.

[AS94] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proceedings of the International Conference on Very Large Databases*, pages 487–499, 1994.

[AS95] R. Agrawal and R. Srikant. Mining sequential patterns. *Proceedings of the International Conference on Data Engineering*, 1995.

[AS96] R. Agrawal and R. Srikant. Mining sequential patterns: Generalizations and performance improvements. *Proceedings of the International Conference on Extending Database Technology*, pages 3–17, 1996.

[BFR98] P. Bradley, U. Fayyad, and C. Reina. Scaling clustering algorithms to large databases. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 9–15, 1998.

[CN04] Y. Cai and R. Ng. Indexing spatio-temporal trajectories with chebyshev polynomials. *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 599 – 610, 2004.

[EFKS98] M. Ester, A. Frommelt, H.P. Kriegel, and J. Sander. Algorithms for characterization and trend detection in spatial databases. *Proceedings of International Conference on Knowledge Discovery and Data Mining*, pages 44–50, 1998.

[GBE^{+}00] R. H. Gting, M.H. Bhlen, M. Erwig, C.S. Jensen, N.A. Lorentzos, M.Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *The Transactions on Database Systems, volume 25(1)*, pages 1–42, 2000.

[GRS98] S. Guha, R. Rastogi, and K. Shim. Cure: An efficient clustering algorithm for large databases. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 73–84, 1998.

[GRS99] M. Garofalakis, R. Rastogi, and K. Shim. Spirit: Sequential pattern mining with regular expression constraints. *Proceedings of the International Conference on Very Large Data Bases*, pages 223–234, 1999.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of Annual Meeting, Boston, Massachusetts, SIGMOD'84*, pages 47–57, 1984.

[HCB98] L. Hall, N. Chawla, and K.W. Bowyer. Combining decision trees learned in parallel. *ACM SIGKDD workshop on distributed data mining*, 1998.

[HKS97] J. Han, K. Koperski, and N. Stefanovic. Geominer: A system prototype for spatial data mining. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 553 – 556, 1997.

[HP00] J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. *ACM SIGKDD Explorations (Special Issue on Scaleble Data Mining Algorithms), 2(2)*, 2000.

[HXSP03] Y. Huang, H. Xiong, S. Shekhar, and J. Pei. Mining confident co-location rules without a support threshold. *Proceedings of the ACM Symposium on Applied Computing*, pages 497–501, 2003.

[IWNM01] Akihiro Inokuchi, Takashi Washio, Kunio Nishimura, and Hiroshi Motoda. A fast algorithm for mining frequent connected subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, 2001.

[Keo01] E. Keogh. Mining time series data. *IEEE International Conference on Data Mining*, 2001.

[KH95] K. Koperski and J. Han. Discovery of spatial association rules in geographic information databases. *Proceedings of the International Symposium on Large Spatial Databases*, pages 47–66, 1995.

[KHS98] K. Koperski, J. Han, and N. Stefanovic. An efficient two-step method for classification of spatial data. *Proceedings of The International Symposium on Spatial Data Handling SDH'98*, 1998.

[KK98] G. Karpis and V. Kumar. Multilevel algorithms for multi-constraint graph partitioning. *Proceedings of ACM/IEEE International Conference on Supercomputing*, pages 1–13, 1998.

[KK01] Michihiro Kuramochi and George Karypis. An efficient algorithm for discovering frequent subgraphs. *IEEE Transactions on Knowledge and Data Engineering*, pages 1038 – 1051, 2001.

[LLC01] Chang-Huang Lee, Cheng-Ru Lin, and Ming-Syan Chen. Sliding window filtering: an efficient algorithm for incremental mining. *Proceedings of the International Conference on Information and Knowledge Management*, pages 263–270, 2001.

[MCK+04] N. Mamoulis, H. Cao, G. Kollios, M. Hadjieleftheriou, Y. Tao, and D. W. Cheung. Mining, indexing, and querying historical spatiotemporal data. *Proceedings of the ACM SIGKDD International Conference on Knowledge discovery and data mining*, 2004.

[MH01] S. Ma and J.L. Hellerstein. Mining partially periodic event patterns with unknown periods. *Proceedings of International Conference on Data Engineering*, page 205C214, 2001.

[Mor01] Y. Morimoto. Mining frequent neighboring class sets in spatial databases. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 353 – 358, 2001.

[MSM95] E. Mesrobian R. Muntz, E. C. Shek, and C. R. Mechoso. Exploratory data mining and analysis using conquest. *IEEE Pacific Conference on Communications, Computers, Visualization, and Signal Processing*, pages 281–286, 1995.

[MTV95] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 210–215, 1995.

[NH94] R.T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. *Proceedings of the International Conference on Very Large Databases*, pages 144–155, 1994.

[NH02] Raymond T. Ng and Jiawei Han. CLARANS: A method for clustering objects for spatial data mining. *IEEE Transactions on Knowledge and Data Engineering*, pages 1003–1016, 2002.

[NK04] Siegfried Nijssen and Joost N. Kok. A quickstart in frequent structure mining can make a difference. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 647–652, 2004.

[OSC00] T. Oates, M.D. Schmill, and P.R. Cohen. A method for clustering the experiences of a mobile robot that accords with human judgments. *American Association for Artificial Intelligence*, 2000.

[PC03] W.C. Peng and M.S. Chen. Developing data allocation schemes by incremental mining of user moving patterns in a mobile computing system. *IEEE Transactions on Knowledge and Data Engineering*, 2003.

[PHMAP01] J. Pei, J. Han, B. Mortazavi-Asl, and Helen Pinto. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. *Proceedings of the International Conference on Data Engineering*, pages 215–224, 2001.

[PHW02] J. Pei, J. Han, and W. Wang. Mining sequential patterns with constraints in large databases. *Proceedings of the ACM CIKM International Conference on Information and Knowledge Management*, pages 18–25, 2002.

[RS02] J. Roddick and M. Spiliopoulou. A survey of temporal knowledge discovery paradigms and methods. *IEEE Transactions on Knowledge and Data Engineering, Volume 14*, pages 750–767, 2002.

[SAM96] Johs Shafer, Rakesh Agrawal, and Manish Mehta. Sprint: A scalable parallel classifier for data mining. *Proceedings of the International Conference on Very Large Data Bases*, pages 544–555, 1996.

[SEKX98] J. Sander, M. Ester, H.P. Kriegel, and X. Xu. Density-based clustering in spatial databases: A new algorithm and its applications. *Data Mining and Knowledge Discovery*, pages 2(2):169–194, 1998.

[SH01] S. Shekhar and Y. Huang. Discovery of spatial co-location patterns. *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases*, pages 236–256, 2001.

[SJLL00] S. Saltenis, C. Jensen, S. Leutenegger, and M. Lopez. Indexingthe positions of continuously moving objects. *Proceedings of the 2000 ACMSIGMOD Conference on Management of Data*, pages 331–342, 2000.

[SNMM95] P. Stolorz, H. Nakamura, E. Mesrobian R. R. Muntz, and C. R. Mechoso. Fast spatio-temporal data mining of large geophysical datasets. *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, pages 300–305, 1995.

[SON95] Ashok Savasere, Edward Omiecinski, and Shamkant Navathe. An efficient algorithm for mining association rules in large databases. *Proceedings of the International Conference on Very Large Data Bases*, pages 432–444, 1995.

[SPTL04] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present and the future in spatio-temporal databases. *Proceedings of the 20th IEEE International Conference on Data Engineering*, pages 202–213, 2004.

[STK+01] M. Steinbach, P. N. Tan, V. Kumar, S. Klooster, C. Potter, and A. Torregrosa. Clustering earth science data: Goals, issues and results. *KDD 2001 Workshop on Mining Scientific Dataset*, 2001.

[TG01] I. Tsoukatos and D. Gunopulos. Efficient mining of spatiotemporal patterns. *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases*, pages 425–443, 2001.

[Tob79] W.R. Tobler. *Cellular Geography, Philosophy in Geography*. Gale and Olsson (Eds), 1979.

[TPS02] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. *Proceedings of the Very Large Data Bases Conference*, pages 287–298, 2002.

[TSK01] P. N. Tan, M. Steinbach, and V. Kumar. Finding spatio-temporal patterns in earth science data. *KDD 2001 Workshop on Temporal Data Mining*, 2001.

[TTPL04] Y. Tao, C. Taloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. *Proceedings of the 2004*

ACM SIGMOD International Conference on Management of Data, pages 611 – 622, 2004.

[WH04] J. Wang and J. Han. Bide: Efficient mining of frequent closed sequences. *Proceedings of the International Conference on Data Engineering*, 2004.

[WWP+04] Chen Wang, Wei Wang, Jian Pei, Yongtai Zhu, and Baile Shi. Scalable mining of large disk-based graph databases. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 316–325, 2004.

[WWYY02] H. Wang, W. Wang, J. Yang, and P.S. Yu. Clustering by pattern similarity in large data sets. *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 394–405, 2002.

[YH02] Xifeng Yan and Jiawei Han. gspan: Graph-based substructure pattern mining. *IEEE Transactions on Knowledge and Data Engineering*, page 721, 2002.

[YH03] Xifeng Yan and Jiawei Han. Closegraph: Mining closed frequent graph patterns. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 286–295, 2003.

[YHA03] X. Yan, J. Han, and R. Afshar. Clospan: Mining closed sequential patterns in large datasets. *Proceedings of the SIAM International Conference On Data Mining*, 2003.

[YWYH02] J. Yang, W. Wang, P. S. Yu, and J. Han. Mining long sequential patterns in a noisy environment. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 406 – 417, 2002.

[Zak98] M. Zaki. Efficient enumeration of frequent sequences. *Proceedings of the International Conference on Information and Knowledge Management*, pages 68 – 75, 1998.

[ZMCS04] X. Zhang, N. Mamoulis, D.W. Cheung, and Y. Shou. Fast mining of spatial collocations. *Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004.