UNIVERSITA DEGLI STUDI DI PADOVA

Corso di laurea Magistrale in Ingegneria Informatica

Periodic subgraph mining in dynamic networks

Laureando: *Barbares Manuel.*

Matricola: *604065.*

Relatore: *Apostolico Alberto.*

Correlatore: *Pizzi Cinzia.*

Data: *07/12/2010.*

Sommario

Recenti ricerche hanno dimostrato che la quantità di informazioni nel mondo raddoppia ogni 20 mesi. Questa crescita esponenziale di dati ha fatto emergere la necessità di progettare e sviluppare algoritmi in grado di automatizzare il processo di analisi dei dati. Il data mining è la disciplina che studia le tecniche per estrarre informazioni dai dati. Il formato dei dati agli inizi era limitato a relazioni tra tabelle e transazioni dove ogni istanza era rappresentata da un insieme. Nel corso del tempo le ricerche iniziarono ad estendersi verso classi di dati strutturati quali sono i grafi. Il data mining di grafi iniziò quindi a diventare sempre più popolare.

Lo scopo principale degli algoritmi di data mining è la scoperta di pattern interessanti da un dataset. La definizione di cosa si intenda per interessante varia a seconda del contesto. Una comune assunzione è quella di considerare interessanti i pattern che si ripetono frequentemente nei dati. Le ricerche si sono focalizzate anche nell'estrarre delle sottostruttre che si ripetono periodicamente nel dataset.

Il problema dell'estrazione di sottografi periodici, che è l'argomento di questa tesi, è una combinazione di entrambi questi aspetti e trova applicazioni in diversi domini applicativi. Infatti, questo problema si prefigge di scoprire interazioni periodiche frequenti tra i membri di una popolazione il cui comportamento viene studiato in un certo arco di tempo. Le interazioni tra i membri della popolazione sono rappresentate da archi E tra vertici V di un grafo. Una rete dinamica consiste in una serie di T timestep per ciascuno dei quali esiste un grafo che rappresenta le interazioni attive in quel dato istante.

Questa tesi presenta *ListMiner*, un algoritmo per il problema dell'estrazione di sottografi periodici. La complessità computazionale di tale algoritmo è $O((V+E) T^2 \ln (T/\sigma))$, dove σ è il minimo numero di ripetizioni periodiche necessarie per riportare il sottografo estratto in output. Questa complessità propone un miglioramento di un fattore T rispetto all'unico algoritmo noto in letteratura, PSEMiner. Gli esperimenti condotti su dataset reali ed artificiali, hanno confermato l'analisi teorica dimostrando che ListMiner è più veloce nella pratica quando vengono analizzati dataset ad alta e media densità.

É stato inoltre proposto un altro algoritmo per risolvere una variante del problema in cui la periodicità ha vincoli più restrittivi, come per le stringhe, permettendo che le istanze dei sottografi abbiano un limitato grado di variabilità. I risultati sperimentali ottenuti utilizzando due reti dinamiche reali dimostrano l'applicabilità del nostro approccio.

Infine è stata effettuata un'analisi qualitativa dei sottografi estratti dai due algoritmi proposti.

Abstract

Recent researches estimate that the amount of information in the world doubles every 20 months. The growth of datasets produced a consequent need to design and development of methodologies to be able to analyze such data through automated processes.

Data mining is the discipline that studies techniques to extract useful information from data. The format of data in the beginning of the field were limited to relational tables and transactions where each instance is represented by one row in a table or one transaction, represented as a set. However, the studies within the last several years began to extend the classes of considered data to semi-structured or structured data such as graphs, making the graph based data mining popular. The main goal of data mining algorithms is the identification of interesting patterns in a dataset. The definition of what is interesting might vary depending on the contexts. However, a common assumption is that patterns that are particularly frequent must hold some valuable information. Moreover, in the last years, the data mining community showed increasing interest and research effort on mining patterns that repeat periodically.

The Periodic subgraph mining problem, which is the subject of this thesis, is a combination of both these aspects, and finds its application in several domains. In fact, this problem aims at the discovery of frequent periodic interactions among the members of a population whose behavior was observed over time. The interactions among members of the population are modeled as edges E between vertexes V of a graph. The dynamic network consists of a series of T timesteps for each of which there is a corresponding graph that describes the set of active interactions.

This thesis presents *ListMiner*, an algorithm for the basic Periodic subgraph mining problem. Its time complexity is $O((V+E)\ T^2 \ln (T/\sigma))$, where σ is the minimum number of periodic repetitions to report a candidate subgraph in output, showing an improvement of a factor T with respect to the state-of-the-art PSEMiner. Experimental evaluation of the performances of the two algorithms, on both real world and artificial datasets, confirmed the theoretical analysis, and also showed ListMiner to be much faster in practice when high and medium density datasets are analyzed.

Another algorithm is proposed to solve a slightly different problem, where the periodicity has stronger requirements, as in the string framework, and some disruption in the composition of the subgraphs is allowed. Experimental results on two real dynamic networks demonstrate the applicability of this approach.

To conclude, the results of a qualitative analysis of the subgraphs that were extracted by the two proposed algorithms are reported and discussed.

Index

1 Introduction

Since early Eighties, the so called *Digital Revolution* has dramatically changed our life style taking advantage of major breakthroughs in all fields related to information and communication technology.
From an engineering perspective, the way information is stored, retrieved, discovered, kept secure or shared, are subject of great research interests for both the Academy and the Industry.
Recent researches estimate that the amount of information in the world doubles every 20 months [33]. The growth in the size of digital datasets naturally produced a compelling need for the design and development of methodologies to be able to analyze such data using automated processes.
Data mining is the discipline that studies techniques to extract useful information from data. Several approaches can be used for fulfill the analysis, each of which might be more indicated than others depending on the data to mine. Among the most important techniques one can find neural networks, clustering, genetic algorithms, decision trees, and support vector machines.
According to Washio e Takoda "*the field of data mining has developed as a novel field of research with the purpose of checking remarkable research issues, and then creating real life applications*" [38].
It is then important to underline this strong connection to real life applications among which one can cite surveillance, fraud detection, marketing, and scientific discovery.
While early data mining techniques were mainly relying on database tables, in the last years many studies have been devoted to different types of data, such as semi-structured data (HTML and XML), symbolic sequences, ordered trees and relations.
The main goal of data mining algorithms is knowledge discovery. Knowledge discovery is based on the identification of interesting patterns in a dataset. The definition of what is interesting might vary depending on the contexts. However, a common assumption is that patterns that are particularly frequent must hold some valuable information. For example, in Market Basket analysis [6], where the purpose is to study the purchase behavior of customers, the patterns of interest are defined by frequent itemsets that represent the basket of items that are usually bought.
In the past few years researchers dealt especially with the need for mining structured data, and graphs are probably one of the best studied structure in the field of computer science and discrete mathematics, which helped graph based data mining to become so popular.
The field of graph mining is very vast. Here follows an overview of the main techniques and of the most interesting problems that have been raised in this very interesting field.
The first studies on graph mining dates back to the middle of the 1990s, when Cook and Holder (SUBDUE) [34] and Yoshida and Motoda (GBI) [35] proposed

some methods to discover concepts from graph representations of some structures.
SUBDUE was based on searching, at each iteration, the best subgraph to compress an input graph *G*. The algorithm uses a bottom up approach, beginning with a subgraph *S* composed by a single vertex that grows incrementally by adding new nodes. When the best subgraph is found, the subgraph is flushed in output and the next iteration starts using the G\S as a new input. GBI (*Graph based induction*) was also based on the idea of deriving a minimal size graph, similarly to SUBDUE, by replacing each interesting subgraph with one vertex, thus compressing the graph at each iteration. At every step the algorithm finds a pair of connected vertexes to join in a single node.
However, both these works used greedy search reporting in output solutions that could not be guaranteed to be optimal.
In 1998, Dehaspe and Toivonen suggested the use of an Inductive Logic Programming (see below for a description of this method) based algorithm (WARMR) [36] that enabled a complete search for frequent subgraphs from graph data. The subsequent work of Nijssen and Kok in [21] proposed a more efficient algorithm for the same problem.
In 2001, De Raedt and Kramer presented MolFea [37], an algorithm to find characteristic paths from a given graph. MolFea was based on a complete search of the paths in the input graph using a lattice structure.
Inokuchi et al. [15] and Kuramochi et al. [16] studied the problem of finding subgraphs that are shared by a set of input graphs applying concept deriving from itemset datamining.
Several other algorithms have been developed for the purpose of graph mining, and they can be classified in five categories: greedy search, inductive logic programming, inductive database, support vector machine and complete search and direct methods.
Early days techniques were mostly based on greedy search [34][35], that falls in the *Heuristic search and direct matching* category. The algorithms that belong to this category can be further classified into two sub-categories depending on the order follow to analyze the elements: depth-first search (DFS), which is used because it can save memory consumption, and breath first search (BFS), which is used because is less time consuming although it uses more memory.
Inductive Logic Programming (ILP) [39] is based on formulating some hypotheses and then seeking the hypotheses to justify the observed fact. The main advantage of this method is the introduction of background knowledge to derive other knowledge represented by "*first order predicate logic*".
Inductive database [40] uses mining approaches to pre-generate inductive rules, relations or patterns. The results are stored in a database which is then queried by using a query language designed to express conditions to retrieve patterns from the database.

Support Vector Machines (SVM) [41] include a set of supervised learning methods that consists on classifying the data input in a multidimensional feature space.

The last category is represented by *Complete search and direct methods,* that perform a complete level-wise search of the dataset. This approach is often used, for example, in market basket analysis. The most popular method of this category is the Apriori based Graph Mining (AGM) [15]. This algorithm starts from frequent graphs composed by a single vertex, and then it builds larger graphs in bottom up manner combining graphs of smaller sizes.

In the last ten years, the data mining community showed increasing interest and research effort on the problem of *Periodic pattern mining*. This kind of analysis is of great interest in several domains [3][4][5], among which transactional datasets, daily traffic patterns, meteorological data, stock data, event logs, web logs, power consumptions.

Therefore, mining periodic patterns is one of the most important tasks in data mining and knowledge discovery.

Among the most interesting results on periodic pattern mining, Ozden et al. [7] analyzed the problem of discovering cyclic association rules that could display regular cyclic variation over time, while Bettini et al. [9] proposed an algorithm to discover temporal patterns in time sequences.

Partial periodic pattern mining is another very interesting problem since it deals with approximation. After the seminal work of Han et al. [8] several others followed [15][20][29]. In particular, Yang et al. in [10] proposed an algorithm to mine all asynchronous periodic patterns, both in a sequence of events and in a temporal datasets with multiple eventsets.

Another general model to mine partial periodic patterns is proposed by Huang and Chang in [11]. In their model, each valid pattern must have a maximum number of disruptions and must contain a minimum number of contiguous matches. SMCA [12] is a suite of four algorithms which enumerates complex patterns.

Another topic of research in pattern mining is the introduction of a probabilistic model to assign a value for the degree of surprise of every occurrence of a pattern. In [14] Yang et al. proposed an algorithm, InfoMiner, to mine surprise patterns according to a new measure of surprise called *information gain*.

Pattern mining problems arise also in the contexts of dynamic networks. Dynamic networks are a sequence of graphs that represent the change in time of the behavior of a fixed population. The members of the population are represented by vertexes. An interaction between two members at some particular time is indicated by the presence of an edge between the two vertexes in the corresponding graph.

The population involved in mining dynamic networks can be of disparate nature: humans [22][23][24], animals [13], networked computers [22]. Social network [21] analysis is probably the most famous example of dynamic network

analysis. It is also worth mentioning that the concept of interaction depends on the application.
Among the analysis that can be performed on dynamic network, finding periodicity is one for the most interesting. For example, periodic subgraphs can correspond to seasonal associations of animals that are of great interest in biological studies [19][13]. Another example of application is the discovery of human periodic behavior in order to understand how humans communicate using current technologies: computer networks [22], mails [23][24], phones [25], social networks[21].
In this context Lahiri and Berger-Wolf in [1] [2] proposed a new mining problem to find predictable behaviors in dynamic social networks. To this purpose it is necessary to identify periodically recurring interaction patterns in networks that change over time. Their solution involved mining all periodic subgraphs that occur a minimum number of times. Here they tackled the notion of *closed* subgraph mining to lower the redundancy in the definition of a frequent pattern, a concept that has been widely exploit in frequent pattern mining [26]. For this purpose they followed Occam's Razor principle of parsimony.
The work of this thesis focuses on periodic mining in dynamic networks. The main contribution is the design and development of ListMiner, an online algorithm that improves the worst case time performances of the algorithm [1][2] by a factor proportional to the number of timesteps in the dynamic network.
A further contribution is the development of an algorithm to extract subgraphs that are periodic in the same sense as periodic strings are defined [30]. Experiments confirmed the theoretical analysis and allowed also for qualitative analysis of the extracted periodic subgraphs.

This thesis is organized as follows:

- Chapter 2 presents the periodic minim problem in dynamic networks as defined in [1][2]. This section contains some preliminary definitions related to dynamic networks, as well as some graph theoretic properties.
- Chapter 3 shows the proof contained in [1][2] that the problem is in the computational class P. The proof consists on deriving the upper bound on the maximum number of possible periodic subgraphs in the worst case. The chapter also contains a proof that in the case of "approximate" periodicity the number of patterns becomes exponential.
- Chapter 4 describes the details of the Lahiri and Berger-Wolf's algorithm, PSEMiner.
- Chapter 5 presents ListMiner, the algorithm proposed in this thesis that reduces the worst case complexity by a linear factor in the number of timesteps of the dynamic network.

- Chapter 6 reports experimental results comparing and analyzing the performance of the two algorithms (PSEMiner and ListMiner) using real and artificial datasets.
- Chapter 7 presents a new algorithm for string-like periodicity mining in dynamic networks.

2 *Problem definition*

2.1 Preliminaries

This chapter presents the periodic graph mining problem as defined by Lahiri and Berger-Wolf [1][2].
The purpose is to study periodic interactions between elements that belong to a population. This can be modeled as a graph where the set of vertexes $V \in \mathbf{N}$ are the entities of population, and the set of edges E represents the interactions between elements. The key observation is that in this context vertexes are uniquely labeled (since they represent a specific member of the population). This is very important from a computational point of view because various hard graph problems such as maximum common subgraph and subgraph isomorphism are reduced to polynomial time [17][27].

Definition 2.1 For a graph $G = (V, E)$ with unique vertex labels, the set representation R for G is formed by mapping each vertex and edge to a unique element in R, where $R \subseteq \mathbf{N}$.

Since each vertex is uniquely identified by its label, it follows that each edge is also uniquely identified by its endpoints. This allows each vertex and edge to be coded as a unique integer, even across different graphs over the same vertex set.

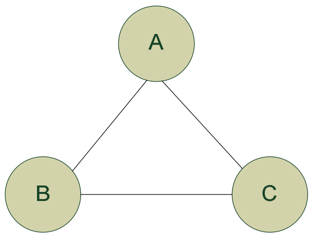


Fig. 2.1

The graph in figure 2.1 can be mapped in the following set representation R: {1,2,3,4,5,6} where, for example, 1 can be vertex A, 2 vertex B, 3 vertex C, 4 edge A-B, 5 edge A-C and 6 edge B-C.
It can be trivially shown that two graphs (or timesteps) will result in the same set R if and only if they have identical vertex and edge sets. Given two graphs G_1 and G_2, with unique vertex labels, testing whether G_1 is a subgraph of G_2 or vice versa is equivalent to check whether the corresponding set representations R_1 and R_2 are subsets of each other.
We observe that for a set of graphs with vertexes unique labels, finding the maximal common subgraph (MCS) is equivalent to calculate the maximal intersection of their set representations (see figure 2.2).
For a set of graphs $<G_1, \ldots, G_T>$, a vertex or an edge is part of the MCS if it is part of every G_t. As a result, the MCS always exists, is unique and well-defined,

but could possibly be the empty graph with no vertexes or edges. We use the intersection operator ∩ to denote the MCS of two or more graphs.

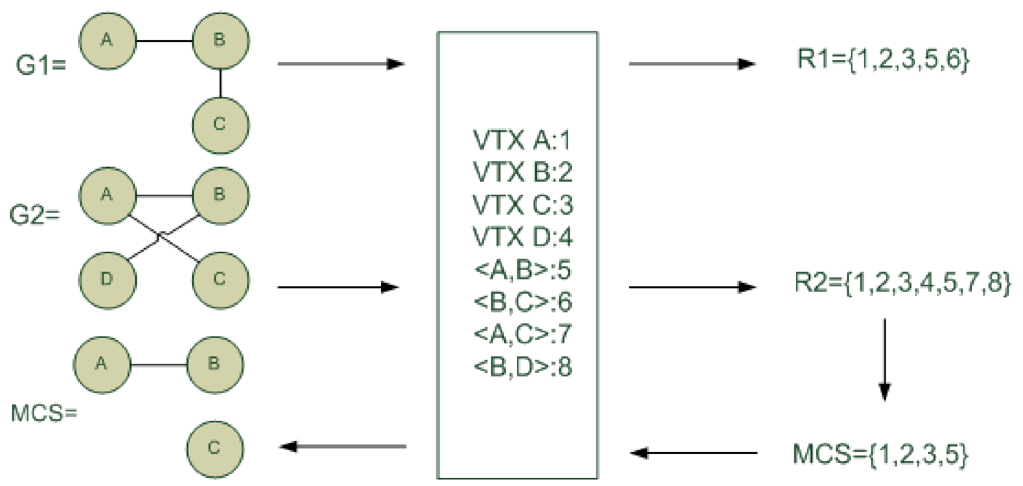


Fig. 2.2: in this example G1 and G_2 are encoded into two sets of integer R1 and R_2. The example shows the calculation of the MCS using the set representation.

Interactions are recorded over a period of time in which the population is observed. The time span is divided into T discrete timesteps of equal duration. These data constitutes a *dynamic network*.

Definition 2.2 A *dynamic network* **G**=$<G_1,...,G_T>$ is a time-series of graphs, where $G_{t=}<V_t,E_t>$ is the graph of interactions E_t observed at timestep t, among the set of uniquely labeled entities $V_t \subseteq V$.

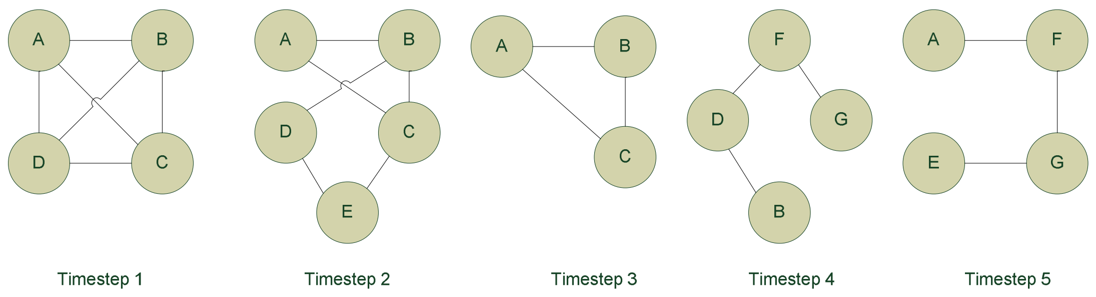


Fig. 2.3: an example of a dynamic network with 5 timesteps.

Definition 2.3 Given a graph $G=(V_g \subseteq V, E_g \subseteq V \times V)$, G is *periodic* with period p, if G is a subgraph of $<G_x, G_{x+p},....,G_{x+np}>$, where $0 \leq x \leq T$, $1 \leq n \leq \lceil (T-x)/p \rceil$.

For example in Figure 2.3 the graph with vertexes B and D and edge B-D is periodic with period 1 because it occurs in timestep 1 and timestep 2. It is also periodic with period 3 because it occurs in timestep 1 and timestep 4.

Studying large populations, the number of periodic patterns could be very large. Moreover, graphs that appear few times are not significant for the context. For these reasons we are interested in patterns that appear at least a minimum number of times.

Definition 2.4 For an arbitrary graph $F=(V_f \subseteq V, E_f \subseteq V \times V)$, its *support set* in G, $S(F)$ is the set of timesteps where F is a subgraph of G_t ($F \subseteq G_t$). F is a *frequent subgraph* of G if $|S(F)| \geq \sigma$, where $1 \leq \sigma \leq T$ is a parameter defined by user.

For example in Figure 2.3 if we set $\sigma=3$ then graph *B-D* is frequent because it occurs in 3 timesteps, but graph *E-G* is not frequent.
An important property is the so called *downward closure property*: if a graph F is frequent, then all its subgraphs are also frequent.
This property is exploited by several algorithms in pattern mining, such as, for example the well known Apriori algorithm[6].
Given a dynamic network, the number of subgraphs in it is exponential in the number of vertexes. So a straightforward enumeration and subsequent check for frequency and periodicity is not a feasible solution. The notion of closed frequent subgraphs is introduced to minimize the redundancy in the network.

Definition 2.5 A graph $F=(V_f \subseteq V, E_f \subseteq V \times V)$ is a *closed frequent subgraph* if there is no other subgraph Y where F is a proper subgraph of Y that $S(Y) = S(F)$.

For example the graph *A-B* in Figure 2.3 is not closed because it is a subgraph of *A-B-C* that is a graph with the same support set.

Definition 2.6 Given a dynamic network G and an arbitrary subgraph $F = (V, E)$, a *periodic support set* of F in G, denoted $SP = (i, p, s)$, is a maximal, ordered set of s timesteps starting at t_i with every two consecutive timesteps differs of p positions.
$SP = (i, p, s) = \langle t_i, t_{i+p}, \ldots, t_{i+p(s-1)} \rangle$ is subject to the following constraints:

1. **Existence in** G**:** F must exist at all timesteps in SP, i.e., $\forall t\, (t \in SP \rightarrow F \subseteq G_t)$;
2. **Minimum size:** A periodic support set has to have at least two elements, i.e., $|SP| = s \geq 2$;
3. **Temporal maximality:** The support set cannot be extended in time to contain F and still be periodic.

The *phase offset* of a periodic support set is defined as $m = t_i \mod p$. Hence, $0 \leq m < p$.

A key difference in the definition of a support set for frequent pattern mining and periodic pattern mining is that a single graph *F* can have multiple periodic support sets to allow for multiple, disjoint, or overlapping periodic behavior. Hence the notion of *periodic subgraph embeddings* is introduced.

2.2 Basic formulation

Definition 2.7 Given a dynamic network *G*, a *periodic subgraph embedding (PSE)* is a pair <*F, SP*>, where *F* is an arbitrary graph that is closed over a periodic support set *SP* with $|SP| \geq \sigma$. The following list summarizes the properties of a PSE:

1. **Minimum support:** $|SP| \geq \sigma \geq 2$.
2. **Structural maximality:** *F* is closed over *SP*, i.e., *F* is the MCS of *SP*.
3. **Temporal maximality:** *SP* is temporally maximal for *F,* i.e. there are not other timesteps *T* where $F \subseteq T$.

These properties allow the development of efficient mining algorithms, and justify an independent treatment of the problem. Mining frequent closed subgraphs is an elegant solution to the redundancy of the general frequent pattern mining problem.

Definition 2.8 Given a dynamic network *G* and a minimum support threshold $\sigma \geq 2$, the *Periodic Subgraph Mining problem* is to identify all frequent closed subgraphs in *G***.**

2.3 Parsimonious formulation

There is the possibility that a periodic subgraph embedding carries information contained in other periodic subgraph embeddings. If a graph is periodic of period *p*, is also periodic of period 2*p* and for every multiple of *p* and depending on the threshold value, if they are frequent they will be put in output.
For example in Figure 2.4 the graph *A-B* is periodic with periods 1, 2, 3, 4. To reduce the size of the output we want to eliminate PSEs with periods 2, 3, 4 because they are redundant.

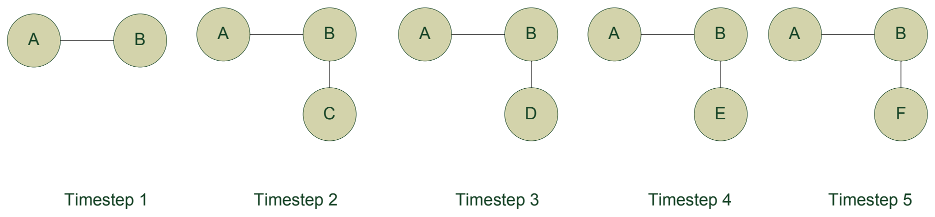


Fig. 2.4: the edge A-B is periodic in this network with period p = 1, 2, 3, 4. In a parsimonious representation of the output the PSE with periods p = 2, 3, 4 should not be reported.

For this reason, the concept of *subsumption* is introduced.

Definition 2.9 For two periodic subgraphs F_1 and F_2 with respective periodic support sets $SP_1 = (i_1, p_1, s_1)$ and $SP_2 = (i_2, p_2, s_2)$, $<F_1, SP_1>$ completely contains or *subsumes* $<F_2, SP_2>$ if all of the following conditions hold:
1. $F_2 \subseteq F_1$
2. $t_{i2} \geq t_{i1}$
3. $t_{i2} + p_{2} \cdot (s_{2} - 1) \leq t_{i1} + p_{1} \cdot (s_{1} - 1)$
4. $p_2 = k \cdot p_1$ for some integer $k > 0$
5. $t_{i,2} = t_{i,1} + l \cdot p1$ for some integer $l \geq 0$

Condition 1 ensures that no information is lost. Condition 2 and condition 3 require that the support set of F_2 is contained within the boundaries of support set of F_1, although they could be of different phase offsets and not overlapping at all, or partially overlapping but of different periods. Condition 4 requires that p_2 is a multiple period of p_1. Condition number 5 guarantees that the initial timestep of F_2 is a timestep where F_1 appears. This means that F_1 and F_2 have compatible offset which ensure that they overlap.

Definition 2.10 A PSE that is not subsumed by any another PSE is a *parsimonious periodic subgraph embedding* (PPSE).

Another condition can be added to the previous problem definition:

Definition 2.11 Given a dynamic network G and a minimum support threshold $\sigma \geq 2$, the *Periodic Subgraph Mining problem* is to list all parsimonious periodic subgraphs embeddings in G that satisfy the minimum support.

Since real-world networks are unlikely to contain perfectly periodic patterns, [1][2] presented a definition of what constitutes 'near' periodicity.

Definition 2.12 A *noisy subgraph can* exhibit a *jitter* in its period, that is, its period is near-constant. Given a jitter value of $J \geq 0$, the periodic graph mining problem can be extended to account for noise as follows: $SP(F) = <t : F \subseteq G_t>$, and $\forall$ i : $|t_{i+1} - t_i - p| \leq J$.

As it will be shown in chapter 3 the subgraph mining problem is in P. However, including jitter the size of the output could become exponential in the number of patterns, making the problem intractable.

3 Complexity of Mining Periodic Subgraph

This chapter presents the proof (taken from [1][2]) that the problems under analysis, as they were stated in definition 2.8 and definition 2.11 are in P. Moreover, a proof is proposed to show that allowing jitter, as explained in definition 2.12, the number of discovered patterns could become exponential in the number of timestep, making this version of the problem intractable.
To this purpose we want to highlight the importance of labeling the vertex with unique labels, allowing graphs to be modeled as set of integers. As a direct consequence, some NP-hard problems in graph mining become polynomial in this context [18].
Starting with a dynamic network *G*, we must build the worst case network. For this purpose the network must contain the maximum number of periodic subgraphs at minimum support $\sigma=2$. This upper bound is a polynomial function of the number of timesteps and the minimum support value.
The proof leads to the conclusion that mining all closed PSEs can be done in polynomial time in the size of the input, proving that the mining (enumeration) problem is in the complexity class *P*, when the graphs have unique vertex labels.
For the proof, the concept of *projection* of a discrete time sequence is introduced to count the maximum number of PSEs in this class of dynamic networks [28].

Definition 3.1 Given a dynamic network *G*, a *projection* $\pi_{p,m}$ of *G* is a subsequence of graphs $\pi_{p,m} = <G_{1+m}, G_{1+m+p}, G_{1+m+2p}, \dots>$, where *p* is the *period* of the projection and $0 \leq m < p$ is the *phase offset*.

It should be clear from the definitions of periodicity and projection that any periodic support set at minimum support σ is embedded in at least σ consecutive positions of some projection $\pi_{p,m}$.

Proposition 3.2 Let *F* be the Maximum Common Subgraph (MCS) of any $s \geq \sigma$ consecutive positions of any projection $\pi_{p,m}$. If *F* is not empty, then it is a periodic subgraph, and the *s* consecutive timesteps from $\pi_{p,m}$ are part of a PSE for *F*.

Proof: if the MCS F of any $s \geq \sigma$ consecutive positions is not empty, this implies that *F* is maximal over a support set of at least σ periodic timesteps, which in turn might or might not be temporally maximal for *F*. However, in either case, the *s* timesteps are part of some valid periodic support set of size at least σ. This is a sufficient condition to satisfy Definition 2.7 (excluding temporal maximality), and thus *F* is a periodic subgraph.

□

Corollary 3.3 In the worst case of the Periodic Subgraph Mining Problem, the MCS of every s ≥ σ consecutive positions of every projection is not empty and contains a unique PSE.

Proof: we now show that this is attainable using an explicit construction. We place a different edge in each $s \geq \sigma$ consecutive positions of every projection to ensure that each edge is part of a unique PSE. Let e be an edge created in this way with support set *SP* in some $\pi_{p,m}$. Considering only *SP*, we know that it is temporally maximal for the edge e because e does not exist in any other timesteps.
Furthermore, the MCS of *SP* is non empty because it contains at least the edge e. Thus, each edge is part of a unique PSE whose support set is *SP*. Since a different edge was placed in every $s \geq \sigma$ consecutive positions of every projection, the number of PSEs is equal to the number of edges created. No additional PSEs can be created since every permissible support set, i.e., with support greater than σ, is already part of a unique PSE. Therefore, the described structure is a worst case instance for its size.

□

Example
We now present an example of worst case. The example has T=5 timesteps, and threshold σ=2. We describe each graph with its integer mapping for ease of explanation.

G_1= <1, 6, 10, 13, 16, 17, 19, 23, 25, 27>
G_2= <1, 2, 6, 7, 10, 11, 13, 14, 16, 20, 22, 24, 26>
G_3= <2, 3, 6, 7, 8, 10, 11, 12, 13, 14, 16, 17, 18, 19>
G_4= <3, 4, 7, 8, 9, 10, 11, 12, 13, 14, 16, 20, 21, 22, 23>
G_5= <4, 5, 8, 9, 11, 12, 13, 14, 16, 18, 19, 24, 25>
G_6= <5, 9, 12, 14, 16, 21,22, 26, 27>

In this dataset we can note that for every period p $1 \leq p \leq 5$, for every sequence we have a different element, hence a different PSE.
The next step is to explicitly calculate the upper bound on the number of PSEs in the worst-case network instance. From Corollary 3.3, we only need to count the number of $s \geq \sigma$ consecutive positions of every projection to derive this bound. In order to do this, we first state the bounds on several other parameters.

Proposition 3.4 In a dynamic network with T timesteps, the maximum period of any periodic subgraph F with support at least σ is $P = \lfloor (T-1)/(\sigma - 1) \rfloor$.

Proof: For a given period p, we can have $F \subseteq G_1$. In the other T-1 timesteps, for every periodic embedding we have that $F \subseteq G_j$ in σ–1 consecutive timesteps

$<T_{1+p}, T_{1+2p}, \ldots, T_{1+p(\sigma-1)}>$. The last index $1+p(\sigma - 1)$ is smaller or equal than T, because T is the index of the last timestep. From this inequality we derive $p \le (T-1)/(\sigma-1)$.

□

Proposition 3.5 In a dynamic network with T timesteps, the length of any projection *is* $|\pi_{p,m}| = \lceil (T - m)/p \rceil$.

Proof: since $\pi_{p,m} = <G_{1+m}, G_{1+m+p}, G_{1+m+2p}, \ldots>$, the projection starts after m elements, and so there are T-m timesteps remaining. Since indexes of two following timesteps differ by p positions, we have that the number of elements in π_p is $\lceil (T - m)/p \rceil$.

□

Given the above expressions, an exact bound for the number of closed PSE can be obtained by construction [1].

Theorem 3.6 In a dynamic network with T timesteps, there are at most $O(T^2 \ln (T/\sigma))$ closed PSEs at minimum support σ.

Proof: From Corollary 3.3, the maximum possible number of PSEs in a dynamic network at minimum support σ is equal to the number of $s \ge \sigma$ length windows over all possible projections of the network. For a given projection $\pi_{p,m}$ and value of s, the number of length-s windows over the projection is $|\pi_{p,m}| - s + 1$, where $|\pi_{p,m}|$ is the length of the projection as defined in Proposition 3.5. Thus, for a given value of s, the number of windows, of length s, over all projections can be obtained by substituting the expressions from Propositions 3.4 and 3.5:

$$\sum_{p=1}^{\left\lfloor \frac{T-1}{s-1} \right\rfloor} \sum_{m=0}^{p-1} \left(\left\lceil \frac{T-m}{p} \right\rceil - s + 1 \right)$$

In the expression for the maximum period of a pattern from Proposition 3.4, the parameter σ was replaced by s since we only want projections which contain at least one length s window for any s. This constitutes the outer summation; the inner summation is over all possible phase offset values m for a given period p. Finally, the term inside the summation is the number of length s windows in any projection, where $|\pi_{p,m}|$ is equal to $\frac{\lceil \mathrm{T-m} \rceil}{p}$ as proved in Proposition 3.5.

This expression is next summed over all possible values of s, which run from σ to T, and the floor and ceiling expressions for an asymptotic closed form approximation are relaxed.

$$\sum_{s=\sigma}^{T} \sum_{p=1}^{\left\lfloor \frac{T-1}{s-1} \right\rfloor} \sum_{m=0}^{p-1} \left(\left\lceil \frac{T-m}{p} \right\rceil - s + 1 \right)$$
$$\sim \sum_{s=\sigma}^{T} \sum_{p=1}^{\frac{T-1}{s-1}} \sum_{m=0}^{p-1} \left(\frac{T-m+p}{p} - s + 1 \right)$$

From the above formula, we obtain the expression:

$$O(T^2 \sum_{p=1}^{(T-1)/(\sigma - 1)} \frac{1}{p})$$

where $\sum_{p=1}^{(T-1)/(\sigma - 1)} \frac{1}{p}$ is approximated by $\ln(\frac{T}{\sigma})$.

Hence, the number of closed PSEs at minimum support σ is $O(T^2 \ln \frac{T}{\sigma})$

□

Theorem 3.7 Periodic subgraph mining in dynamic networks is in P.

*Proof: s*uppose to have an algorithm that outputs the maximal common subgraph of every σ length window of every projection. Since the maximal common subgraph can be found in time O(V+E) [27], the algorithm runs in time $O((V+E)T^2 \ln \frac{T}{\sigma})$, and it is guaranteed to output every closed periodic subgraph. Thus, the mining problem is in *P*, and the exact bound on the number of closed PSEs is given in summation form in Theorem 3.6.

□

An alternative proof of Theorem 3.6 was proposed in [2].

Theorem 3.8 In a dynamic network with *T* timesteps, there are at most $O(T^2/\sigma)$ closed periodic subgraphs at support exactly equal to σ.

Proof: For each projection $\pi_{p,m}$ the maximal common subgraph, of any σ consecutive timesteps, if not empty, can be a unique maximal periodic subgraph. Let *s* be the length of the projection. There are at most $s-\sigma+1$ possible windows of size σ in any projection. For every of these windows in the worst case, there is a maximal periodic subgraphs that is also frequent. Summing this expression over all possible values of *m* and *p*, we obtain the following upper bound on the total number of possible maximal periodic subgraphs:

$$\sum_{p=1}^{P}\sum_{m=0}^{p-1}(s-\sigma+1)=\sum_{p=1}^{\lfloor\frac{T-1}{\sigma-1}\rfloor}\sum_{m=0}^{p-1}\left(\left\lceil\frac{T-m}{p}\right\rceil-\sigma+1\right)$$

Solving this expression, we obtain that there are at most $O(T^2 / \sigma)$ closed PSEs at support σ.

□

Since we are interested in closed PSE at minimum support, to prove Theorem 3.6 we have to sum all closed PSEs for every support σ'≥ σ.

$$\sum_{\sigma'=\sigma}^{T}\sum_{p=1}^{\lfloor\frac{T-1}{\sigma'-1}\rfloor}\sum_{m=0}^{p-1}\left(\left\lceil\frac{T-m}{p}\right\rceil-\sigma'+1\right)$$

Solving the previous expression we obtain $O(T^2\ln\frac{T}{\sigma})$ closed PSEs at minimum support σ.

As mentioned in chapter 2, to avoid redundancy a new formulation of the problem is proposed including the concept of parsimony.
As explained in [1] a naïve algorithm to mine parsimonious subgraphs compares each of the $O(T^2\ln\frac{T}{\sigma})$ closed PSEs at minimum support σ with all other PSE taking time $O((V+E)(T^2\ln\frac{T}{\sigma})^2)$.
These complexity bounds do not hold in case jitter is allowed (see definition 2.12). In fact in this case the number of mined patterns could be exponential in the number of timesteps. Below I propose two possible proofs for this observation.

Theorem 3.9: The number of PSE when jitter is allowed is exponential in the number of timesteps.

Proof: let us consider a dynamic network $<G_0, \dots , G_T>$, and fix a period p. The following reasoning can be done: for every graph G_i, $0\le i\le T$, the maximum common subgraph $MCS(G_i,G_{i+p})$, if not empty, is a periodic subgraph (in this case i represent the offset of a given projection).
Allowing jitter j, for every timestep G_x, where $i+p-j \le x \le i+p+j$, the MCS M_x between G_i and G_x is a valid periodic subgraph. If every G_x has a different common edge with G_i, every MCS M_x is different from the others. So there are exactly $2j+1$ different periodic subgraphs. Now, for each of these different 2j+1 subgraphs the same reasoning can be repeated.
In particular, for a single graph M_x, the MCS with every graph G_y, where $x+p-j\le y\le x+p+j$, if not empty, is a valid periodic subgraph. As previously described,

if every G_y has a different common edge with M_x, there are other $2j+1$ different periodic subgraphs. This can be repeated for each previously identified graph M_x, thus obtaining $(2j+1)^2$ different periodic subgraphs.
For a given projection $\pi_{p,i}$ this process can be repeated for the maximum length of the projection $\pi_{p,i}$ that is $\lceil T-i/p \rceil$. Therefore the number of patterns discovered could be $(2j+1)^{\lceil T-i/p \rceil}$ for a fixed projection $\pi_{p,i}$.

□

The proof could be done also by induction in the number of elements in a given projection. The following proof describes the construction of the worst case for the projection $\pi_{1,0}$.

Base case:
The proof starts from G_{j+1}. Following the previous considerations, for every graph G_x, where $1 \leq x \leq 2j+2$, the MCS M_x can be calculated with every graph G_y, where $x+p-j \leq y \leq x+p+j$ having, in the worst case, $2j+1$ different graphs. Repeating this procedure for every G_x, we have in total $(2j+1)^2$ periodic graphs.

Induction:
Assuming the property true for $t-1$, there are $(2j+1)^{t-1}$ different periodic graphs. Repeating the process described above, for each of the $(2j+1)^{t-1}$ graphs, in the worst case, when all MCS are not empty and different from each other, further $2j+1$ periodic graphs are found. So, in total there are $(2j+1)^t$ different periodic subgraphs.

□

The following figures present an example of what was explained above.

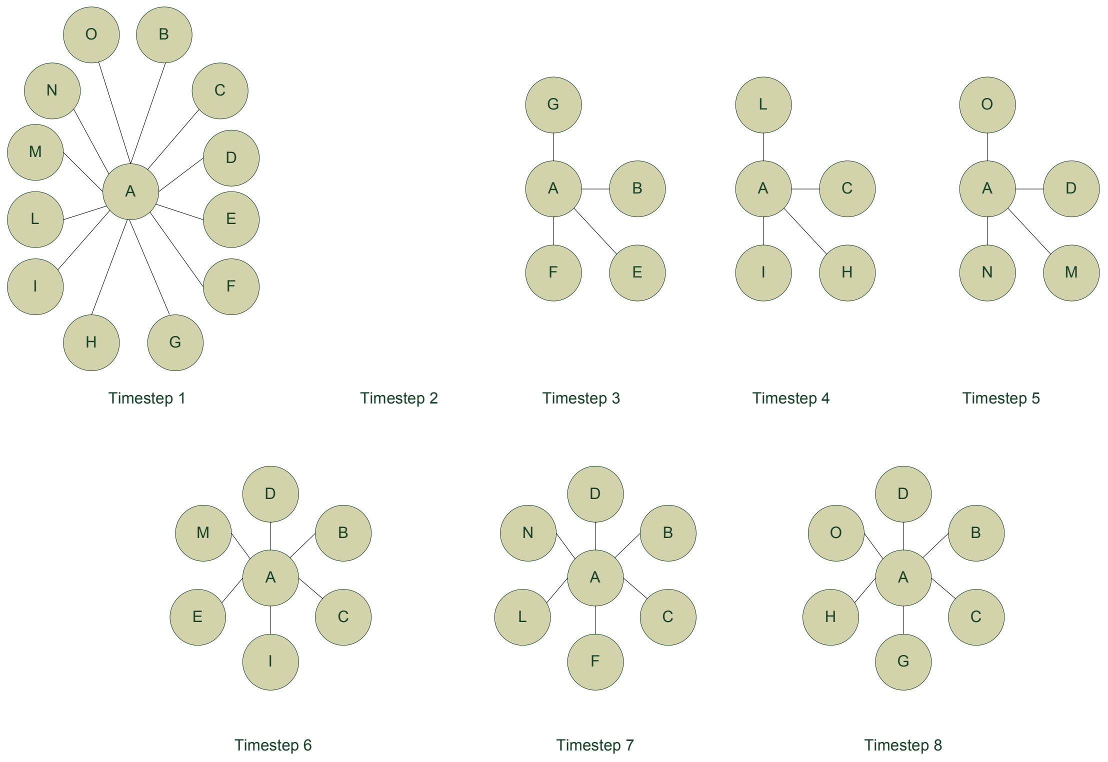


Fig. 3.1: an example of dynamic network in which, setting jitter j=1 and σ=2, the number of frequent and periodic pattern is exponential in the number of timesteps.

Suppose we are looking for periodic patterns with period p=3 and σ=2*,* allowing jitter with j=1. Considering the offset i=1, both G_1 and G_2 are possible candidates to be the first timestep of a PSE with period 3 and offset 1. For brevity we consider only the case with G_1 as first timestep. The next expected timestep is 4, since we want find periodic patterns with period 3. Allowing jitter also timesteps 3 and 5 are possible candidates. The MCS G_{13} between G_1 and G_3 is equal to G_3, the MCS G_{14} between G_1 and G_4 is equal to G_4*,* and the MCS G_{15} between G_1 and G_5 is equal to G_5. After timestep 4 the next expected timestep is timestep 7. Allowing jitter also timesteps 6 and 8 are possible candidates. Therefore, for each of the three periodic patterns (G_{13}, G_{14}, G_{15}) the MCS between timesteps 6, 7, and 8 is a periodic pattern that is different from the others. In total, with period 3, we have $(2j+1)^{[(T-1)/p]-1}=(3)^2=9$ periodic patterns. With respect to the expression calculated in theorem 3.9, the exponent is decremented by one because, for brevity, we do not have considered the case with G_2 as first timestep. Nevertheless, the number of patterns remains exponential in the number of timesteps.

The following figures list the 9 periodic patterns described above:

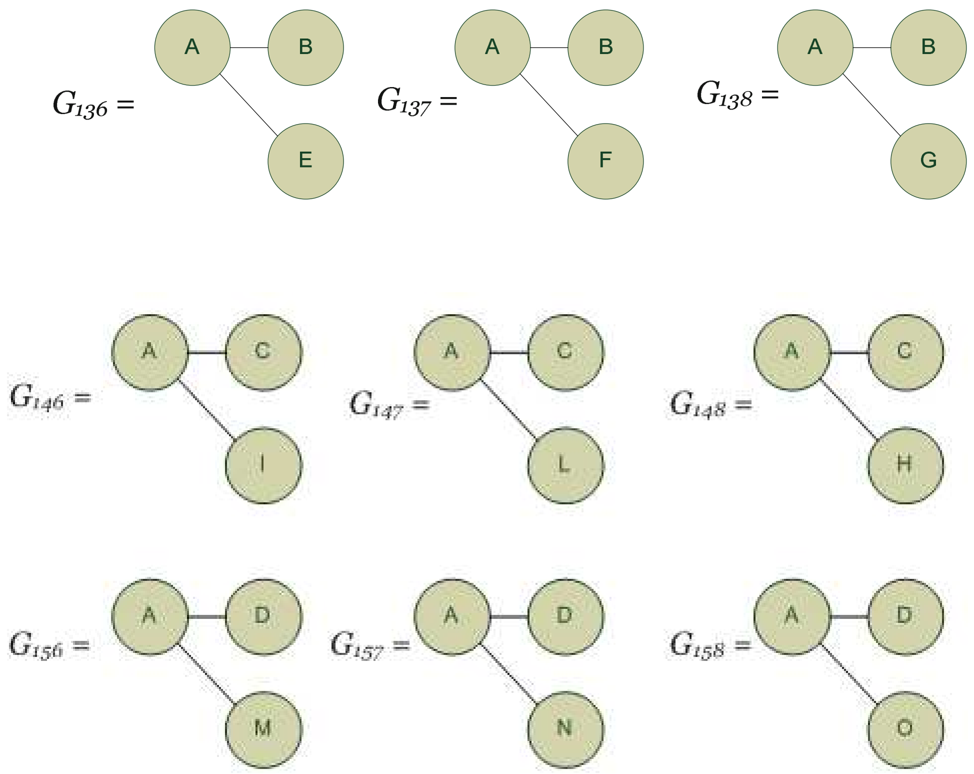


Fig 3.2: all frequent and periodic patterns of projection $\pi_{3,1}$ in figure 3.1 with σ=2 and j=1.

4 PSEMiner (Lahiri and Berger-Wolf's algorithm)

The main characteristic of the algorithm proposed in [1][2] is the use of a data structure called *pattern tree*. This structure maintains all PSEs seen up to timestep *t,* and it also tracks subgraphs that might become periodic at some point in the future.
At each timestep *t*, the graph G_t is red, and the pattern tree is updated with the new information, which could involve modifying, adding and deleting tree nodes.
The most important parameter of the algorithm is the maximum period P_{max}.
When $P_{\max}$ is restricted, the algorithm functions as an online algorithm, retaining in memory only the parts of the dataset that it requires to calculate periodicities.
However, in many situations this information is not available or relevant, such as in streaming sensor data. In such cases, an unrestricted maximum period value must be set. The unrestricted period places a large computational burden on the algorithm, and requires that the entire dataset is retained in memory.

4.1 Data structures

The algorithm actually make use of five data structures:

- *Pattern tree*. It maintains all PSEs seen up to timestep *t,* and also tracks subgraphs that might become periodic at some point in the future. The nodes of the pattern tree are called *treenodes*.
- *Treenode*. Each treenode contains a different subgraph *G* and a list of *descriptors* (see below for a detail description), one for each PSE observed for subgraph G. There is a constraint that every treenode, except the root, must observe: all descendants of a treenode for a graph *F* are associated with proper subgraphs of *F*, but not all subgraphs of *F* are necessarily its descendants in the tree.
- *Subgraph hash map*. It allows direct access to treenodes by associating subgraphs with their corresponding treenode. This can be done because a hashing function exists for graphs since the set representation R has a global ordering by virtue of $R \subset N$.
- *Descriptor*. It is a representation of a periodic support set. Each descriptor is associated with a treenode and it defines a unique PSE. Each descriptor is a triplet *(i,p,s)* that stores the initial position of the observed PSE, the period and the number of steps for which the periodicity hold (support). The last element in the support set is the timestep $t_j=t_i+p\ (s-1)$. The next expected timestep is $t_e=t_j + p$.
  A descriptor, at timestep *t*, is alive if $t_e \geq t$. A descriptor that is not alive must be reported in output if it has enough support, and then discarded, since it cannot change state and return alive in the future. A descriptor

where $t_i = t_j$ is a special case called an *anchor descriptor*, as it does not represent a periodic support set, but it could potentially become a PSE if the associated subgraph is observed at some future timestep. An anchor descriptor is always alive, unless $P_{\max}$ is defined and $t - t_i > P_{\max}$, in which case the anchor can never lead to a valid PSE with period at most $P_{\max}$, and is no longer needed.

- The algorithm also uses a two-dimensional arrays to store periods and phase offsets of live descriptors.

4.2 Description of the basic algorithm

Initially an empty treenode is set at the root of the pattern tree. At each timestep t, with associated graph G_t, the treenode is traversed with a breadth-first visit. Only treenodes that can be modified by the newly acquired information are the ones that are actually traversed. This excludes every treenode with subgraph F for which the *MCS(G_t,F)* is empty, and all its descendant. The algorithm first searches in its hashtable if the node already exists in the pattern tree for G_t. If it does not exist, a new node is created in a position that does not violate the subgraph constraint, or it is added as a new child of the root.
An anchor descriptor for graph G_t is then added to the corresponding treenode.
During the breadth-first traversal of the tree, one of the following three conditions holds at each treenode N with graph F. Let $C = F \cap G_t$ be the MCS of G_t and F, and consider each descriptor of the treenode.

- **Update descriptors:** If $F \subseteq G_t$ then F has appeared in its entirety at timestep t. Let t_e the next expected timestep.
    - If $t_e=t$ then t is added to the support to ensure temporal maximality;
    - If $t_e<t$ then the expected timestep has already been processed. If the support of D is greater than σ, then D is written in output and removed from the treenode.
    - If $t_e>t$ then the expected timestep has not been processed yet, so nothing is done.
    - If D is an anchor descriptor then timestep t can be considered as a second occurrence and the descriptor D is updated: the period is set to $p = t - t_i$ and the phase offset to $m = (t_i - 1) \bmod p$. If N does not contain a living descriptor with the same period and phase offset, D is added as new descriptor to the list of descriptors at N.

  For every treenode N' with graph F' that is a child of N, since $F \subseteq G_t$ and $F' \subseteq F$ for the property of treenodes, we have $F' \subseteq G_t$. So the algorithm can update all descriptors of the subtree with root F without calculating the MCS, thus saving computational time.

- **Propagate descriptors:** If C is not empty then a subgraph C of F occurs at timestep t. Using the subgraph hash map, the algorithm controls if a treenode for C exists, otherwise a treenode is created as a child of *N* with subgraph *F*. The descriptors of *N* are copied in the new node, and timestep *t* is added to their support sets if the next expected timestep is *t*. If the treenode exists, then for each descriptor *D*, if the next expected timestep $t_e=t$, *t* is added to support set. Otherwise, if $t_e<t$, *D* is written in output if its support is greater than, or equal to σ, and *D* is removed. If $t_e>t$ no action is taken.

- **Dead subtree:** If *C* is empty, then G_t and *F* have no common subgraph, and no descriptors at *N* are directly affected by the observation of G_t. Furthermore, no treenode that is a descendant of *N* will have any common subgraph with G_t either, since they are all subgraphs of *F*. The subtree rooted at *N* is therefore eliminated from the rest of the tree traversal.

```
Algorithm 1 UPDATETREE($G_t$)
Require: $G_t$ is the graph of timestep $t$
1: $Q \leftarrow$ new queue
2: push($Q$, root.children)
3: while $N \leftarrow$ pop_front($Q$) do
4:   $C \leftarrow G_t \cap N$
5:   if $C$ is not empty then
6:     if $N \subseteq G_t$ then
7:       UPDATEDESCRIPTORS($N$)
8:     else
9:       $W \leftarrow$ FINDNODE($N$) or NEWNODE($N$, $C$)
10:      PROPAGATEDESCRIPTORS($N$, $W$)
11:    end if
12:    push($Q$, children($N$))
13:  end if
14: end while
15: $W \leftarrow$ FINDNODE($G_t$) or NEWNODE(root, $G_t$)
16: Add anchor descriptor for $G_t$ to $W$.
```

Fig 4.0 shows the pseudocode of the function to update the tree.

Example

Here we show an example of the algorithm using the dataset in figure 2.3 (that is reported again in figure 4.7) as input, and setting σ=2.

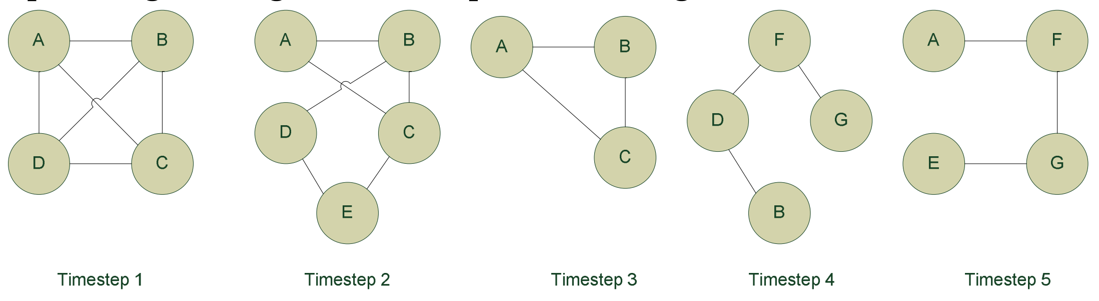


Fig 4.7: the input dataset.

We start with the first timestep creating a single node that contains the graph of the first timestep, and an anchor descriptor.

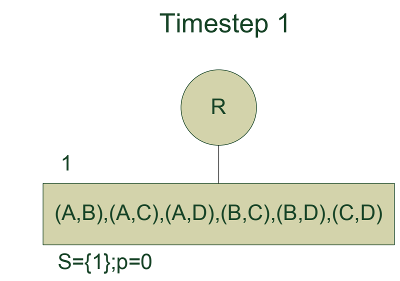
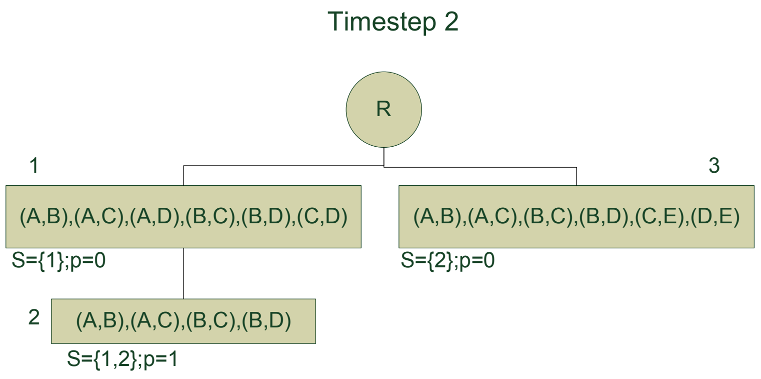


Fig. 4.1: description of timestep 1

We continue by processing the second timestep. First, node 2 is created, with an anchor descriptor for the graph contained in the second timestep. Then another node is created, which contains the MCS between the first two timesteps.


Timestep 2
R
1
3
(A,B),(A,C),(A,D),(B,C),(B,D),(C,D)
(A,B),(A,C),(B,C),(B,D),(C,E),(D,E)
S={1};p=0
S={2};p=0
2
(A,B),(A,C),(B,C),(B,D)
S={1,2};p=1
Fig. 4.2: description of timestep 2

In the next step of the algorithm a new node (node 2 of Figure 4.3) is created for the graph at timestep 3, holding an anchor descriptor starting at *t=3*. This node must be a child of node 1, since it is a subgraph of the graph in node 1. From node 1, node 2 inherits the descriptor starting at *t=1*, which can be extended both with period *p=1*, and with period *p=2*. Finally, the old node 2 (Figure 4.2) is deleted because its next expected time is *t=3*, but the subgraph does not appear in its entirety at this timestep. However, since its support equals the threshold, it must be reported in output before being discarded.

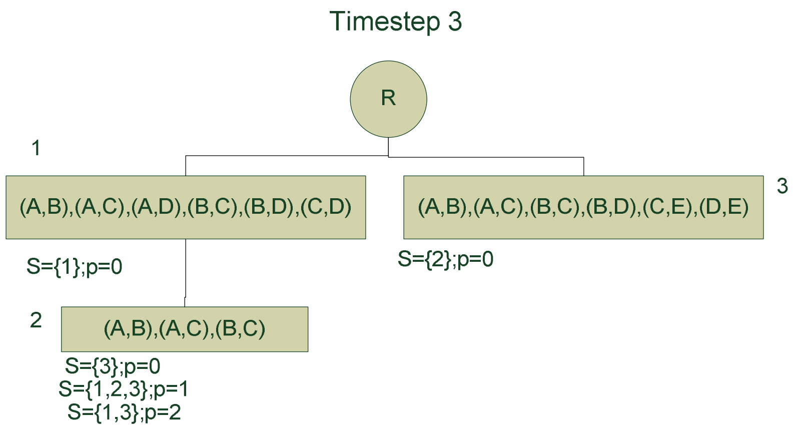


Fig 4.3: description of timestep 3

Then node 5 (fig 4.4), with the MCS between node 1 in fig 4.3 and G_4, is created, with the descriptor with period $p=3$ (timesteps 1 and 4). The descriptor of the node 2 in fig 4.3 that refers to period p=1 is no longer valid, so the subgraph is reported in output (its support is 3) and the descriptor is discarded.
The MCS between node 2 in fig 4.3 and G_4, leads to the creation of node 6, where only the vertex B occurs with period $p=1$. The MCS with node 3 in fig 4.3 gives again the subgraph with just the edge (B,D). Since this graph is already in the pattern tree the algorithm does not create a new node but only the descriptor with period 2 and support {2,4}.

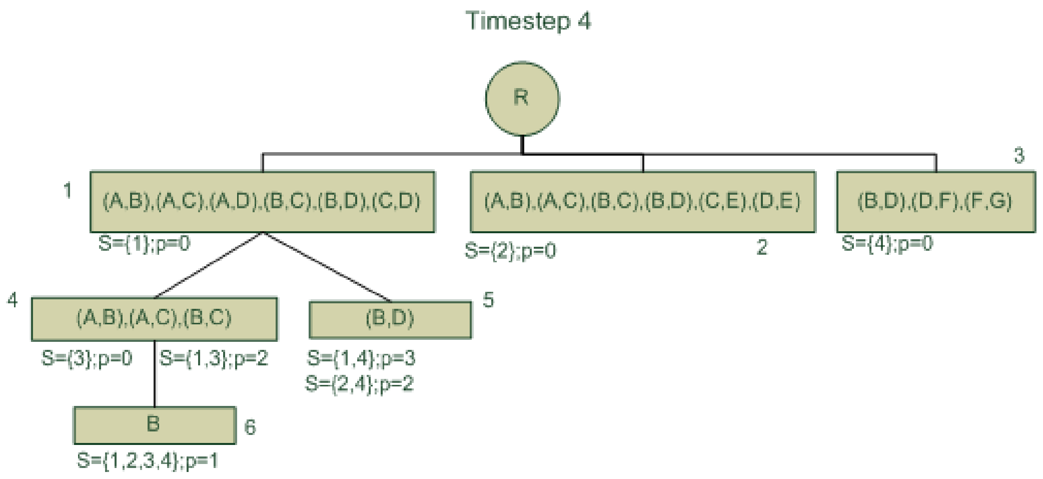


Fig 4.4: description of timestep 4

The last timestep refers to figure 4.5. A new node with graph G_5, and an anchor descriptor, is inserted as a child of the root. The MCS with node 1 in figure 4.4, gives the vertex A. The newly created node 6 in figure 4.5 will inherit a descriptor for a subgraph occurring at t=1 and t=5, therefore with period p=4. The MCS with node 2 in figure 4.4 gives a new node with vertexes A and E. Node 7 in figure 4.5 is therefore inserted as child of node 3 (figure 4.5) with the corresponding descriptor.

The MCS with node 3 in figure 4.4 is the edge (F,G). A new node 8 (figure 4.5) is created, with inherited descriptor for *p=1*.
Apart from the anchor descriptor, the other descriptor of node 4 in figure 4.4 must be flushed in output if frequent, and then deleted, because the graph at node 4 in fig 4.4 does not appear in its entirety.
The MCS with node 5 in figure 4.4 is the empty set, however the next expected times for its descriptors are *t*=7 and *t*=6, so nothing happens.
The MCS with node 6 in figure 4.4 gives the empty set. Since its only descriptor is no longer valid, it must be reported in output because frequent, and then the node is discarded.

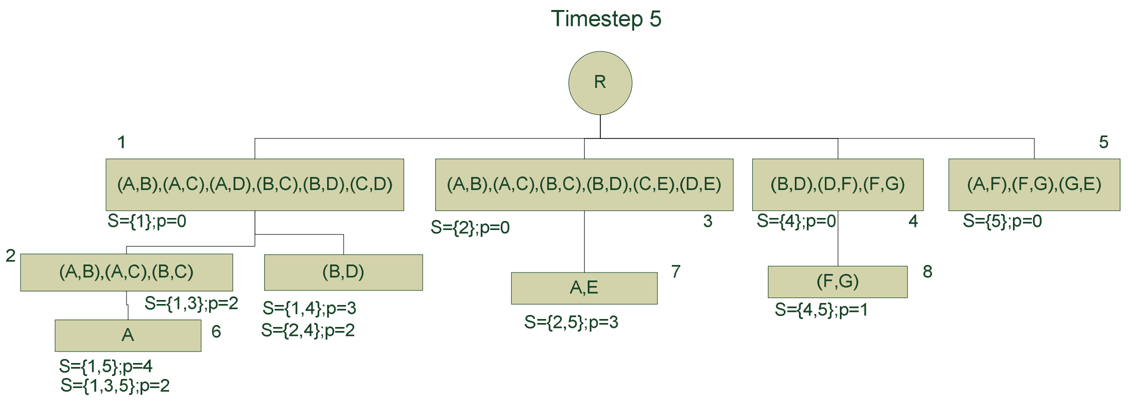


Fig 4.5: description of timestep 5

Finally, the tree is traversed and the frequent descriptors are flushed in output along with their subgraph.

4.3 Extension to the basic algorithm

4.3.1 Mining parsimonious PSEs.

For mining parsimonious PSEs an indicator bit is added to each descriptor to indicate if the descriptor is subsumed. This bit is initially cleared when the descriptor is created. When a descriptor *D* that belongs to a treenode *N* have to be put in output, the indicator bit is checked: if it is clear then *D* is compared to all other descriptor at *N*. If *D* is subsumed by another descriptor, it is not written to the output. If *D* subsumes some other descriptor *D'*, the subsumed bit for *D'* is set to 1. If the support of *D* increases in the future, its subsumed bit is cleared. If the indicator bit is set, the descriptor is not written in output.

4.3.2 Including smoothing

Since real-world networks are unlikely to contain perfectly periodic patterns, Lahiri and Berger-Wolf [1][2] used smoothing as a mechanism for accommodating imperfect periodicity. Given a user-defined smoothing

parameter $S \geq 1$, the dynamic network $G=<G_1,G_2,\ldots G_T>$ is mapped in a new network G', in which each element $G_i' = G_i \cup \ldots \cup G_{i+S}$
In addition, the following two conditions handle the removal of artifacts introduced by the smoothing process.

1. The minimum period $P_{\min}$ is set to S.
2. PSEs of the same subgraph that share the same period, and that differ in their starting positions by at most $S-1$ timesteps, are merged together. In other words, only the PSE with the highest support is retained. This can be done as a post-processing step, or it can be incorporated into the mining algorithm itself.

By introducing this smoothing mechanism, they allow a window of timesteps within which the order of events does not matter. No smoothing is performed at $S = 1$.

4.3.3 Sorted descriptor list

The list of descriptors can be sorted by the next expected timestep. So, for a given timestep *t*, only descriptors which are expected at or before *t* will be examined, cutting down the number of descriptors that need to be examined during each tree update. The added computational cost is that of having to sort the list of descriptors after each update. Since the number of descriptors per treenode is generally not very large, the computational overhead is minimal in practice.

4.3.4 Lazy tree update

Most of the running time is due to the computation of the intersection between graphs. Although the maximum common subgraph of two graphs is calculated in linear time in the number of vertexes and edges, the size of the graphs is such that this operation results in a relatively expensive computation. Thus, to improve the practical efficiency of the algorithm, it is possible to delay the computation of intersections until it is absolutely necessary. In a treenode *N* an intersection at timestep *t* is absolutely necessary if there is a descriptor *D* in which the expected timestep is *t*. This variant of Lahiri and Berger-Wolf algorithm is shown in figure 4.6.

Algorithm 2 LazyUpdateTree(G_t)

```
Require: $G_t$ is the graph of timestep $t$
$Q \leftarrow$ new queue
push($Q$, root.children)
while $N \leftarrow$ pop_front($Q$) do
  lazy $\leftarrow$ true
  while lazy = true do
    $D \leftarrow$ next descriptor at $N$
    next $\leftarrow$ last($D$) + period($D$)
    if D is an anchor or next = $T$ then
      lazy $\leftarrow$ false
    else
      if next < $T$ then
        flush $D$ to output and delete
      else
        break
      end if
    end if
  end while
  if lazy = false then
    $C \leftarrow G_t \cap N$
    if $C$ is not empty then
      if $N \subseteq G_t$ then
        UpdateDescriptors($N$)
      else
        $W \leftarrow$ FindNode($N$) or NewNode($N, C$)
        PropagateDescriptors($N, W$)
      end if
      push($Q$, children($N$))
    end if
  else
    push($Q$, children($N$))
  end if
end while
$W \leftarrow$ FindNode($G_t$) or NewNode(root, $G_t$)
Add anchor descriptor for $G_t$ to $W$.
```

Fig 4.6: the algorithm for lazy tree update described in section 1.3.4

4.3.5 Using a timeline

The timeline is a mechanism that associates each future timestep with a list of treenodes that have at least one descriptor expected at that timestep. It can be dynamically updated at a not significant cost (constant or logarithmic) per treenode update, and stored in linear space in the number of treenodes. After the tree update for timestep *t*, all treenodes that are still associated with timestep *t* are guaranteed not to have been visited during the tree update, and have at least one descriptor which is no longer periodic. These treenodes can then be visited and the invalid descriptors can be removed. The time required is proportional to the number of descriptors to be removed. Thus, at the end of each tree update operation, the treenode only contains descriptors that are alive at the next timestep. This ensures that the pattern tree contains a minimal number of descriptors and treenodes at any given timestep.

4.4 Space and time complexity

Let N the number of nodes in the treenode, P_{max} the maximum period, and G be the number of frequent periodic subgraph.
From corollary 3.3, the worst case is when we have the maximum number of periodic subgraph at minimum support $\sigma=2$.

Space complexity: let N be the number of treenodes and G the number of descriptors in the tree. Every node of the tree has an associated graph of size $O(V+E)$. Moreover the algorithm uses a two-dimensional array of size P^2_{max} to store periods and phase offsets of live descriptors (P_{max} offset for every of the P_{max} period). The overall space complexity is therefore $O((V+E)N+P^2_{max}+G)$. From proposition 3.6, in the worst case we have $O(T^2 \ln (T/\sigma))$ number of different periodic subgraphs. Since in the worst case we have that each descriptor corresponds to a unique PSE, and in every treenode there is only one descriptor, we have $N=G=O(T^2 \ln (T/\sigma))$. Since at most one descriptor is added per timestep, the asymptotic bound on the total number of nodes and descriptors does not change.
In the worst case the space complexity is $O((V+E+ P^2_{max}) T^2 \ln (T/\sigma))$, and if P_{max} is unresctricted ($P_{max}=O(T/\sigma)$) the total space complexity is $O((V+E) T^2 \ln (T/\sigma)+ T^4 \ln (T/\sigma))$.

Time complexity: For every timestep t the tree is completely traversed. Thus, time complexity of the algorithm involves traversing each descriptor in the tree once for each timestep, and calculating the MCS at each treenode. So the time complexity is $O((V+E)T N)$. In the worst case the number of the nodes is equal to the number of PSEs that are $O(T^2 \ln (T/\sigma))$, so the time complexity in the worst case is $O((V+E) T^3 \ln (T/\sigma))$.

5 ListMiner

This chapter presents the main contribution of the thesis: the design and development of ListMiner, an algorithm that improves the worst case time complexity of PSEMiner by a factor T.

From the previous description in Chapter 4 it can be observed that at each timestep t PSEMiner must traverse every node of the pattern tree. At each node u the following operations occur:

1) the MCS between the graph under analysis G_t, and the graph described at node u is computed;

2) the corresponding node v in the pattern tree is searched for;

3) each descriptor at node v is then checked for consistency of periodicity in t (basically its next expected time must be equal to t). If so, the descriptor is updated, otherwise either it is deleted or no action takes place. If no action is taken, the time consuming computation of the MCS was useless. This problem is due to the fact that, with this approach, at the time of the computation of the MCS it is not known if the resulting subgraph would be periodic or not.

The key idea of the algorithm that I propose in this thesis is to calculate the MCS only when necessary. To this end I have been inspired by proposition 3.2 to consider only timesteps in which the graphs to intersect can contain a periodic subgraph. Consider a fixed period p, every timestep t belongs to a single projection $\pi_{p,m}$, where $m=(t \bmod p)$. Therefore every projection can be considered separately, since graphs that belong to different projections cannot be periodic with the same period p.

More precisely, for a fixed period p, the T timesteps are partitioned into p projections. For example, setting $p=1$, there is a single projection that contains all the timesteps. Setting p=2, there are two projections: $\pi_{2,1}=<G_1,G_3,G_5,G_7,\ldots,>$ and $\pi_{2,0}=<G_2,G_4,G_6,G_8,\ldots.>$, etc.

Afterwards, for every projection, a list is created. This list contains the intersection between every possible sequence of consecutive graphs. For example if the projection is $\pi_{2,1}$ the list is composed by $\{G_1\}$, $\{G_3\}$,...,$\{G_1 \cap G_3\}$,$\{G_3 \cap G_5\}$,.....,$\{G_1 \cap G_3 \cap G_5\}$ and so on. In this way every possible PSE is generated, as explained in proposition 3.2. By iterating the process for all possible choices of period p, all PSE will be eventually found.

In the following sections these concepts are formalized, and a detailed description of the algorithm is given.

5.1 Preliminaries

Definition 5.1: For a given projection $\pi_{p,m}=<G_1',G_2',\ldots,G_x'>$, where $G_j'= G_{pj+m}$ and $x=\lceil (T-m)/p \rceil$, we call run $S_{i,j}$ every subsequence of consecutive graphs from $\pi_{p,m}$. For two fixed indexes i and j, $1 \le i \le j \le x$ we have $S_{i,j}=< G_i',\ldots,G_j'>$.

Proposition 5.2: For a given period p, every timestep G_t belongs to a single projection $\pi_{p,m}$ where $m=t \bmod p$.

Proof: by definition, $G_t \in \pi_{p,m}$ if $t=qp+m$ for some q. It is known that for every $t \in \mathbf{Z}$ there is a unique remainder $m \in \mathbf{N}$ such that $t=qp+m$ where $p,q \in \mathbf{Z}$ and $p>0$. Since m is unique, G_t belongs only to $\pi_{p,m}$.

□

Proposition 5.3: For a given period p, there are exactly p projections.

Proof: from the previous proposition the number of all possible values of remainders m is p.

□

It follows from proposition 3.2 that in a projection $\pi_{p,m}$, the MCS of any $s \geq \sigma$ consecutive positions, if not empty, is a periodic subgraph, and the s consecutive timesteps are part of a PSE.
Therefore the purpose of the algorithm is to consider, for every period p, every projection $\pi_{p,m}$ and computing the MCS M between all graphs of every run $S_{i,j}$ of length at least σ, and saving only subgraphs which are temporally maximal.

Using the following property the MCS of every run can be calculated in time $V+E$.

Property 5.4: Given a run of graphs $<G_i, G_{i+1},\ldots,G_x>$ where $1 \leq i \leq x \leq t$, the MCS of this run is:

$$\text{MCS of } <G_i, G_{i+1},\ldots,G_x> = \text{MCS of } < \text{MCS of } < G_i, G_{i+1},\ldots,G_{x-1}>, G_x>.$$

Proof: this property can be proved using the associative property of intersection between sets. Since from definition 2.1 every graph can be considered as a set of natural numbers, the intersection of $<G_i, G_{i+1},\ldots,G_x>$ is equal to the intersection between $< (G_i, G_{i+1},\ldots,G_{x-1}), G_x >$.
Using this theorem, the MCS of a given run $S_{i,j}$, can be obtained calculating the MCS between $S_{i,j-1}$ and the j-graph of run $S_{i,j}$. The time needed for such intersection is $V+E$.

□

5.2 Data structures

In order to mine all periodic, frequent subgraphs, the algorithm uses three primary data structures: *lists, listnodes* and a *bidimensional array* that contains every list. To mine only parsimonious subgraphs another data structure is necessary: a *hash map*.
Every list is composed by some listnodes and it is associated to a specific projection $\pi_{p,m}$. Every listnode describes a run $S_{i,j}$ of the projection and it is used to describe a single temporally maximal PSE.

5.2.1 List

Every projection $\pi_{p,m}$, where $1 \leq p \leq P_{max}$, $0 \leq m < p$, is associated to a specific list. Every node of the list contains the MCS between all graphs of a specific run in projection $\pi_{p,m}$. Precisely, at timestep t, every list L contains in its listnodes in reverse order ($S_{t,t}$ in the first node of the list and $S_{1,t}$ in the last node) the MCS of all runs $S_{x,t}$ $1 \leq x \leq t$ which graph is temporally maximal. Therefore, by construction, the list is subjected to a single constraint: each node N in the list has a graph that is properly contained in the graph of its predecessor.
This property allows efficient traversal of every list by the mining algorithm, and also allows the list to be built and manipulated quickly.

5.2.2 Listnode

Given a list L for a single projection $\pi_{p,m}$, every listnode describes a single PSE and it is composed by:

- *Start index:* this is the index of the first timestep of the PSE;
- *End index:* this is the index of the last timestep of the PSE;
- *Graph G:* it is the graph that is associated to the PSE. This is the MCS between all graphs from timestep T_{start} to timestep T_{end} whose indexes differ by p (the graph is periodic with period p);
- *Support:* it is the number of elements of the support set. It can be obtained in the following way: $Support = (T_{end} - T_{start})/p$, and is pre-computed because it is used several times during the processing.

This data structure is equivalent to the *descriptor* used in Berger Wolf 's algorithm.

5.2.3 Bidimensional array

The bidimensional array A is used to store all lists. The list associated to the projection $\pi_{p,m}$ is stored at position $A[m][p]$. Using A allows to perform list lookup in constant time.

5.2.4 Subgraph hash map

This data structure is used for mining parsimonious PSE only. It uses a graph as a key. For every key the associated object is a list of descriptors. Descriptor is a triple $<p,s,e>$ where p is the period, s is the first timestep and e is the last timestep of the PSE. This information is added to the hash map when a listnode is flushed out of its list.
Since two PSE could have the same graph, the object associated to every key is a list of descriptors. Before writing in output a PSE P with graph G, G is used as a key to access to the hash map. If it exists in the hash map, the corresponding list is traversed, and for every descriptor D the algorithm controls if D subsumes P. If so, P is subsumed, otherwise it is printed in output and its descriptor is added to the list. If it does not exist in the hash map then P is the first PSE with graph equal to G. This means that all other PSEs with graph equal to G will have a

period greater than the period of P. Therefore P can be safely printed in output because it cannot be subsumed.

5.3 Parameters

The algorithm is a single-pass, polynomial time and space algorithm for mining all closed PSE in a dynamic network. It does not require any parameters, but it optionally accepts the following:

- Minimum support threshold $\sigma \geq 2$ (default: 2).
- Maximum period P_{max} (default: unrestricted).

The optional $P_{\max}$ parameter limits the maximum period of mined patterns, thus reducing the number of projection to consider, with a consequent speed up of the algorithm.
Minimum support threshold σ is a parameter that is used for mining frequent subgraphs (see definition 2.4). Its value depends on the context and on the dataset. By increasing σ the size of the output is reduced and the algorithm is faster.

5.4 Description of the algorithm

The algorithm starts creating a bidimensional array A that contains, in every cell, an empty list. Timesteps are red and stored in an array called *input*.
For each timestep t the algorithm finds, for each period p, the list L stored in the bidimensional array in position $A[p][m]$ where $m=t \bmod p$.
To begin with, a new listnode $N=(G_t,t,t,0)$ is added at the head of the list because it could be the first element of a future PSE. Thereafter, the function *update* is called. This function calculates the MCS between the graph in each listnode and G_t. Whenever a PSE of a subgraph node is detected, it is checked for subsumption, and eventually printed in output.
When all timesteps have been elaborated, some listnodes could remain in the lists. This is because the next expected timestep for some graphs could be equal to or greater than T. Therefore the algorithm must further control if every PSE associated to these nodes is subsumed by another PSE previously calculated and, if it is not subsumed, it must report it in output.
The algorithm is implemented using three functions: *Miner* is the main program; *update* is the function to update a single list; *subsumed* is a function to control if a specific PSE can be printed in output (the subgraph is frequent and not subsumed).
This is the complete version that discovers all frequent periodic and parsimonious subgraphs. For mining all frequent and periodic graphs without subsumption, the function *subsumed* must not be called. In this case the algorithm only controls if the support of every subgraph is greater than, or equal to σ.

Algorithm Miner(*input*)

Require: *input* is a vector that in position *input* [i] = G_{i+1}. We have *T* timesteps, from 0 to *T*-1. Let *L.begin()* and *L.end()* be the first and the last element of the list, *L.iterator()* the element pointed by the iterator.

```
1:  A ← new matrix;
2:  H ← new subgraph hash map;
3:  for (t ← 0 to T-1) do
4:      G_t ← input[t]
5:      for (p←1 to min (t, P_max)) do
6:          phase ← t mod p
7:          L←A[p][phase]
8:          N= new ListNode(G_t,t,t, 0)
9:          L.push_head(N)
10:         update(L).
11:     end for
12: end for
13: for (i ← 0 to P_max ) do
14:   for (j ← 0 to i ) do
15:     L←A[i][j];
16:     iterator ← L.begin()
17:      for iterator to L.end() do
18:        G_x ← L.iterator()
19:        if(!Subsumed (G_x,i)and G_x.support() ≥ σ) then
20:              print G_x
21:        endif
22:     end for
23:   end for
24: end for
```

5.4.1 *Update algorithm*

This section describes the update algorithm, which is the core of the mining process. For every timestep *t* all the lists associated to projections $\pi_{p,m}$, where $p \leq \min(t, P_{max})$ and $m=t \bmod p$, are updated with the new information contained in G_t. The update process starts by adding a listnode for G_t at the head of the list *L*. This listnode is built as follows: the graph is set to G_t , start and end indexes are set to *t* because *t* is the first and the last index of the run, and the support is set equal to 1. This accounts for the possibility that G_t in its entirety is the first occurrence of a (future) periodic subgraph.

During the traversal of the list, one of the following three conditions holds at each listnode *N* with graph *F*. Let $C = F \cap G_t$ be the MCS of G_t and *F*.

- If $F \subseteq G_t$, i.e. $F = C$, then F has appeared in its entirety. Therefore the MCS is F, and the listnode is updated in the following way:
    - *Graph* is unchanged
    - *Start index* is unchanged.
    - *End index* is set to t because the last timestep where the MCS (C) occurs is t.
    - *Support* is incremented by one unit because there is another timestep (t) where C appears.

  Since all successors of a node N must have a graph G_d that is a subgraph of F, and F is subgraph of G_t ($G_d \subseteq F \subseteq G_t$), then G_d is a also subgraph of G_t. Therefore the algorithm updates all successors of node N in the same way without calculating the MCS, thus saving computational time.
- If $C = \emptyset$, then G_t and F have no common subgraph. Furthermore, no listnode that is a successor of N will have any common subgraph with G_t either, since they are all subgraphs of F. N and all its successors are therefore eliminated from the rest of the list and, if they are frequent and not subsumed, they are flushed in output.
- If $C \neq \emptyset$ and $F \not\subseteq G_t$, then a subgraph C of F is present at timestep t. This happens, for example, when a formerly periodic subgraph F fractures into a smaller subgraph C that continues F's periodic behavior.
  In this case the algorithm first check if the listnode parameters describe a subgraph that is frequent and not subsumed. If it is so, it is printed in output. Then the algorithm updates the listnode N in the following way:
    - *Graph* is set to C.
    - *Start index* is unchanged.
    - *End index* is set to t because t is the last timestep where C appears.
    - *Support* is equal to support(N)+1.

  The next listnode in the list is then considered.

Moreover, whenever the update involves not just the start/end indexes, but also the *Graph* variable, or whenever a new node is inserted, the new graph is compared with the one at the previous node. If they are equal, the previous node is deleted, since it would represent the same graph within a smaller periodic interval, therefore it would not respect the condition of temporal maximality (see definition 2.7). For example let us consider a dynamic network composed by a unique graph G that is repeated in every timestep. For every timestep the algorithm inserts a new listnode with graph equal to G. If the algorithm does not control if the new graph is equal to the one at the previous node then all listnodes of every list contain the same graph (G) but not all are temporally maximal (only the last node of every list is temporally maximal).
The pseudocode of the update function is reported in the next page.

Algorithm Update(*L*)

Require: *L* is a list of listnode. Let *L.begin()* and *L.end()* be the first and the last element of the list, *L.iterator()* the element pointed by the iterator, *L.iterator().graph()* the graph of the element pointed by the iterator and *iterator.next()* a function for forwarding the iterator. *F* and *C* are graphs, *N* is a node.

```
1:  G_t ← L.begin()
2:  iterator ← L.begin()
3:  iterator.next() //the first node is (G_t,t,t,0) therefore the update starts at the next node.
4:  while iterator ≤ L.end()  do
5:    N ← L.iterator()              // current node
6:    F ← L.iterator().graph()    // current node graph
7:    C ← G_t ∩ F
8:    if (F ⊂ G_t) then
9:        while iterator ≤ L.end()  do
10:         N ← L.iterator()
11:         N.update_end_index(t) // the end index of N is set equal to t
12:         N.update_support(support(F)+1) // the support of N is incremented by one
13:       end while
14:   else if C = ∅ then
15:         while iterator ≤ L.end()  do
16:           N ← L.iterator()
17:           if (!Subsumed (N.graph(),p) and N.support() ≥ σ ) then
18:             print N
19:           endif
20:           delete N
21:         end while
22:       else // case C != ∅ and C!=F
23:         if (!Subsumed (F,p) and N.support() ≥ σ ) then
24:             print N
25:         endif
26:         N.update_graph(C) //the graph of N is set to C
27:         N.update_end_index(t) // the end index of N is set equal to t
28:         N.update_support(support(N)+1) //the support of N is incremented by one
29:         if (N.graph()  is equal to the graph of the previous node) then
30:           delete the previous node
31:         endif
32:         iterator.next()
33:       endif
34:   endif
35: endfor
```

5.4.2 Subsumed algorithm

This procedure controls if a given PSE, that is represented by a listnode, is subsumed by another PSE. In order to do this the algorithm uses a subgraph hash map *H*.
For a given PSE *P* with graph *F* the procedure checks if a list associated to *F* exists in *H*. If not, then *P* is not subsumed because it is the first PSE with graph *F*. Therefore *P* is printed in output and stored in the hash map. Otherwise, for every descriptor of the list the algorithm verifies if there exists another descriptor that respects all the conditions given in definition 2.9. If there is, then *P* is subsumed and it is not printed in output, otherwise *P* is memorized in the hash map and flushed in output.

The pseudocode of the function subsumed is presented below.

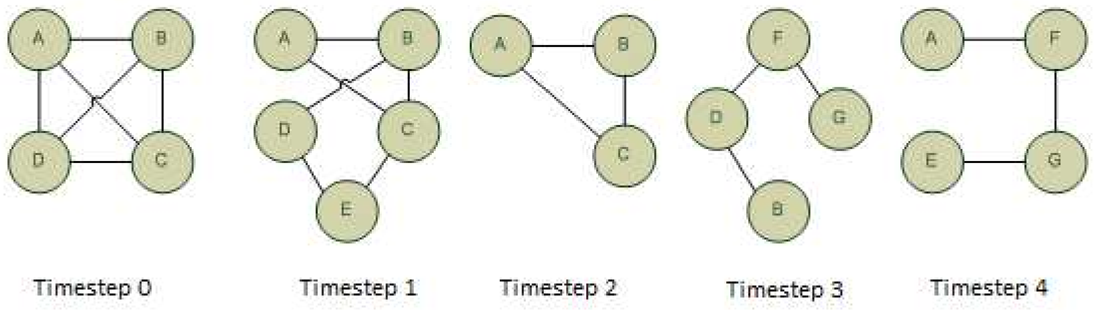
Algorithm Subsumed(*G*,*p*)

Require: *G* is a listnode, *p* the period, *H* the subgraph hash map. *H*.search(*G*.graph()) is a function of *H* that returns true if the graph of *G* is in *H*, otherwise it returns false. *H*.insert(graph, start,end, period) is a function that insert in *H* a graph with his relative period and support set.

```
1:  subsumed=false;
2:  if (H.search(G.graph())=false) then
3:     if (G.support() $\geq \sigma$) then
4:       H.insert(G.graph, G.start(),G.end(),p)
5:     endif
6:  else
7:       for each descriptor $D \in$ H.search(G.graph()) do
8:          if (p mod D.period()=0 and G.start() $\geq$ D.start() and G.end() $\leq$ D.end()) then
9:             subsumed=true
10:            break
11:        end if
12:      end for each
13:      if (subsumed=false and G.support() $\geq \sigma$) then
14:        H.insert(G.graph, G.start(),G.end(),p)
15:      endif
16: endif
```

5.4.3 Example

Consider as input dataset the dynamic network in figure 2.3 that we report here for convenience.



Each column represents a single step of the algorithm. Each row is a single list that represents the elaboration of the specified projection. The algorithm in this example does not perform subsumption, the threshold σ is equal to 2 and P_{max} is unrestricted. Listnodes are represented by grey boxes, S is the support and indexes near boxes are just used to reference in the text.
In the next pages a figure shows an example step by step of the execution of ListMiner.

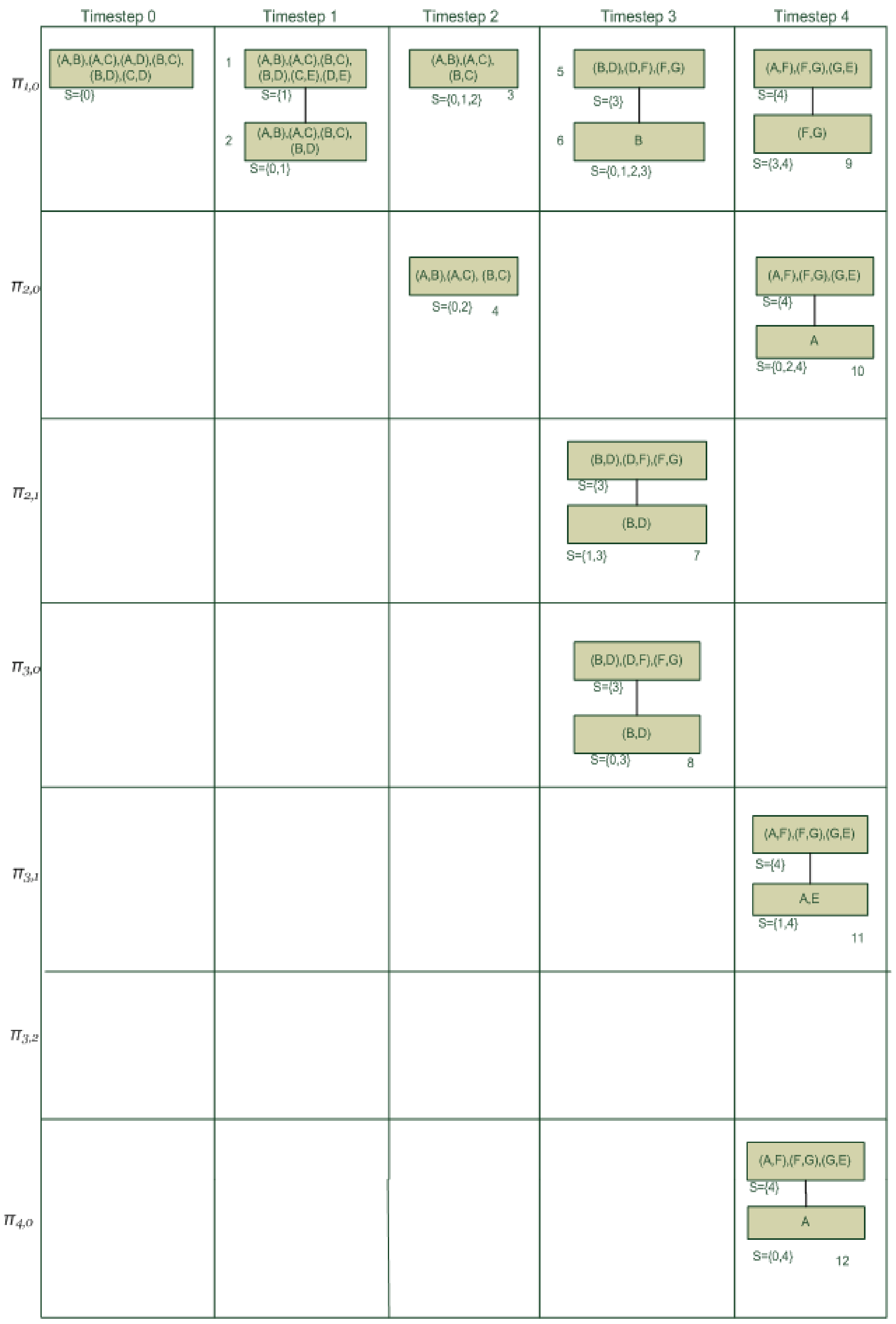


Fig 5.1 - A step by step example of computation. Each box represent a node. The edges in each box represent the current MCS. Single vertexes are not explicitly shown, unless they are the only components of an MCS. Each column represent a timestep, and each row a different List: $\pi_{i,j}$ is the list for period *i* and phase *j*.

It should be noted that at each timestep *t* (with the exception of the initial timestep 0), only the lists corresponding to a projection $\pi_{i,j}$ with period *i* (up to *t*) and phase *j=t mod i* are subject to update. Moreover, at each timestep *t* there will be the initialization of those lists for which *i+j=t+1*. The initialization simply consists in the insertion of a node with *Graph*=G_j, *start=end=j,* support=1. The algorithm also output single nodes that are periodic (when they are maximal). For ease of representation such subgraphs will be considered implicitly represented by the arcs involving them. They will be explicitly represented, by a box containing only the label of the vertex, only when they are the maximal periodic subgraph at that step.

- Timestep 0: the first list $\pi_{1,0}$ is initialized to contain G_0.
- Timestep 1: after two steps only periodicity *p*=1 can occur, so $\pi_{1,0}$ is updated. The first step consists in the insertion of a new listnode (G_1, 1, 1, 1) at the head of the list. This is node 1 in the table. Then the following node is analyzed. The intersection between its graph (which is G_0) and the current graph G_1 is not empty (and different from G_0), so case (3) of the procedure *update* applies. The support of the node is less than the threshold, so we can safely update the content to (MCS(G_0,G_1), 0, 1, 2). This node is called node 2 for future reference.
- Timestep 2: after three steps we can have periodicity 1 and 2 (starting in 0). So the algorithm must:
    - o Update $\pi_{1,0}$. A node (G_2,2,2,1) is inserted at the head of the list. Then the MCS between G_2, and node 1 is computed. The result of the intersection is G_2, hence condition (3) of the procedure update applies. Since node 1 has not enough support we just update it to (G_2,1,2,2). Now, node 1 and its predecessor share the same graph, hence its predecessor is deleted. Then node 2 is considered. Its support set is 2, hence it should be reported in output (or put in the hashtable H for subsequent processing).
      Next, node 2 is similarly updated to hold (G_2,0,2,3). Since node 2 and its predecessor share the same graph its predecessor is deleted. Node 2 is relabeled to node 3 to avoid confusion in the further description of the processing.
    - o Create list $\pi_{2,0}$, which also requires to be first initialized to hold (G_0,0,0,1). Then the new node for G_2, (G_2,2,2,1), is inserted at the head of the list. The MCS between G_2 and the following node (G_0,0,0,1) is computed. The intersection is not empty, and not equal to G_0, hence we are in case (3) of *update*. Node (G_0,0,0,1) is not frequent, so it can be safely updated to (MCS(G_0,G_2), 0, 2, 2) and named node 4.
- Timestep 3: the lists to consider at this step are $\pi_{1,0}, \pi_{2,1}, \pi_{3,0}$.
    - o Update $\pi_{1,0}$. A node (G_3,3,3,1) is inserted at the head of the list, and labeled with number 5. Then the MCS between G_3, and node 3

is computed. The intersection contains no edges between different vertexes, however, vertex B is still present, so the MCS is not empty. Since node 3 has enough support it should be reported in output (or put in the hashtable H for subsequent processing). Then it is updated to contain ({B}, 0, 3, 4), and relabeled as node 6.

    - List $\pi_{2,1}$ is considered for the first time, and initialized to contain (G_1,1,1,1). Node (G_3,3,3,1) is then inserted at the head of the list. The processing continues computing the MCS(G_1, G_3). This is not empty and not equal to G_1(condition (3) of *update*). Since the current node is not frequent, it is updated to (MCS(G_1, G_3),1,3,2), and labeled with number 7.
    - Similarly, list $\pi_{3,0}$ is considered for the first time, and initialized to contain (G_0,0,0,1). Node (G_3,3,3,1) is then inserted at the head of the list. Then the MCS(G_0, G_3) is computed. This is not empty, and not equal to G_0 (condition (3) of *update*). Since the current node is not frequent, it is updated to (MCS(G_0, G_3),1,3,2) and labeled it with number 8.

- Timestep 4: the lists to consider at this step are $\pi_{1,0}, \pi_{2,0}, \pi_{3,1}, \pi_{4,0}$.
    - Update $\pi_{1,0}$. A node (G_4,4,4,1) is inserted at the head of the list. Then the MCS between G_4, and node 5 is computed. The result of the intersection is not empty, and not equal to the graph of node 5, so condition (3) of the update procedure holds. Since the node has not enough support, it can be safely updated to (MCS(G_3, G_4),3,4,2), and labeled node 9.
    - Update $\pi_{2,0}$. A node (G_4,4,4,1) is inserted at the head of the list. Then the MCS between G_4, and node 4 is computed. The result of the intersection is the vertex A. Since the node has enough support, it should be reported in output (or put in the hashtable H for subsequent processing) before updating it. The result of the update ({A},0,4,2) is named node 10.
    - List $\pi_{3,1}$, is considered for the first time, and initialized to contain (G_1,0,0,1). Node (G_4,4,4,1) is then inserted at the head of the list. The processing continues computing MCS(G_1, G_4), which is vertexes A and E. Since the current node is not frequent, it can be safely updated to ({A,E},1,4,2), and named node 11.
    - List $\pi_{4,0}$, is considered for the first time, and initialized to contain (G_0,0,0,1). Node (G_4,4,4,1) is then inserted at the head of the list. The processing continues computing MCS(G_0, G_4), which is vertex A. Since the current node is not frequent, it can be safely updated to ({A},0,4,2) and named node 12.
- The subgraphs that should be reported in output are those with labels 2,3,4,6,7,8,9,10,11,12. If subsumption is checked then the nodes to report in output should be those with labels 2,3,6,7,8,9,10,11,12.

5.5 Correctness

Every PSE is represented by a listnode. Let $\pi_{p,m}$=<G_1',G_2',...,G_x'>, where G_j'=G_{pj+m} $1 \leq j \leq x \leq t$, be a given projection. The algorithm uses a list to calculate the MCS of every possible run. For each element G_j of the projection, $i \leq j \leq x-\sigma$, the list is traversed: a new listnode with the graph G_j is inserted at the head of the list. After that every element *L* of the list is replaced by a new listnode *N* as explained in section 5.4.1.
If the MCS between the node of the list and G_j is equal to the MCS that is stored at the previous node, the previous node is deleted because its support set is not temporally maximal (see property 3 of definition 2.7). Therefore all PSE flushed in output are temporally maximal.

Here follows a series of theorems that prove the correctness of *ListMiner*.

Theorem 5.5: Given a projection $\pi_{p,m}$=<G_i, G_{i+1},....,G_n> and the corresponding list *L*, the algorithm calculates and stores in *L*, in reverse order, ($S_{n,n}$ is the first node of the list and $S_{1,n}$ is the last node) the MCS of all runs $S_{x,n}$ $1 \leq x \leq n$, for a fixed *n*.

Proof: the proof is an induction on the number of elements *n*.

Base: *n*=1.
In this case the list is empty. The listnode (G_i,*t*,*t*,0) is added to the head of the list. This node contains the MCS for $S_{1,1}$ which is the only possible run.

Induction: suppose the property is true for *n-1*.
Since the proposition is true for *n-1* the list contains the MCS of all runs $S_{x,n-1}$ $1 \leq x \leq n-1$. For the next element G_n, the *n*-element of the projection is inserted at the head of the list. Therefore the MCS for run $S_{n,n}$ is found. Afterwards, for every run $S_{x,n-1}$ $1 \leq x \leq n-1$ that is represented by a single listnode with graph *G*, the algorithm replaces it with a new listnode with the MCS between *G* and G_n. Therefore, from theorem 5.4, in every node of the list there is the MCS of all runs $S_{x,n}$ $1 \leq x \leq n$ in reverse order.

□

Theorem 5.6: The algorithm calculates the MCS of every possible run $S_{x,j}$ $1 \leq x \leq j \leq n$ that belongs to a given projection $\pi_{p,m}$ =<G_i, G_{i+1},....,G_n> for all values of *n*.

Proof: this can be proved using theorem 5.5. Since every element of the projections is processed, for every $1 \leq x \leq n$ theorem 5.5 is valid. Therefore the MCS of every possible run $S_{x,j}$ $x \leq i \leq j \leq n$ is calculated.

□

Theorem 5.7: The algorithm calculates all temporally maximal PSE.

Proof: theorem 5.6 proves that the algorithm calculates the MCS of all possible runs of a given projection. If two consecutive runs have the same graph then the run with support set not temporally maximal is deleted. Therefore for a single period every temporally maximal PSE is found. By iterating this process for all the possible choices of period *p*, all PSE are calculated.

□

Theorem 5.8: Let *L* be a list for a given projection $\pi_{p,m}$. For every node *N* of the list with graph *G*, all successors have a graph *G'* that is a proper subgraph of *G*.

Proof: the proof is an induction on the number of timestep *T*.

Base: $T=1$
With only one element the proof is trivial because a single node has not successors.

Induction: first we prove that for every node *N* of the list with graph *G*, all successors have a graph *G'* that is a subgraph of *G*.
Suppose that the property is true for *T*-1. Therefore for every listnode *N* of the list, at timestep *T*-1, all successors have a graph that is a proper subgraph of graph of *N*.
The algorithm at timestep *T*, for every period *p,* updates every list associated to projection $\pi_{p,m}$ where $m= T \bmod p$, inserting at the head of the list a new listnode with graph G_T and substituting the graph G_j of each node with $G_j \cap G_T$.
Since the first element of the list is G_T and the graph of every node *N* is updated with $G_j' = G_j \cap G_T$, where G_j is the graph of the node *N* at the previous timestep, all successors of the first node have a graph $G_j' \subseteq G_T$.
Now we prove the theorem for the other nodes of the list, from second to the last. For these nodes, before the update process, at timestep *T*-1, every listnode with graph *G*, have all successors with graph *G'* that is a subgraph of *G* for inductive hypothesis. The algorithm, for every node *N,* updates its graph *G* with $(G \cap G_T)$. Since for every graph G' that is a successor of *N* G' is a subgraph of *G* for inductive hypothesis, then also $(G' \cap G_T) \subseteq (G \cap G_T)$. Therefore for all successors of a node *N* with graph *G*, $G' \subseteq G$.
To complete the proof we have to prove that all successors of a node with graph *G* have a graph that is a *proper* subgraph of *G*. This is due by construction because the algorithm always controls if the graph in a node is equal to the graph of the previous node. If it is then the previous node is deleted. Therefore we cannot have two consecutive nodes with the same graph.

□

Theorem 5.9: Every listnode in each list represents a unique PSE.

Proof: for a given period *p*, a single timestep G_t belongs to a single projection. For every different projection there is a different list. Therefore for every list, every node cannot represent a PSE which is equal to another node of another list because they belong to different projections. In the same list, from theorem 5.6, we see that every node is the MCS of a given run, and so it cannot represent a PSE equal to another node of the same list.

□

The *subsumed* function is correct because it fulfill all the conditions listed in definition 2.9.
From theorem 5.8 and 5.9 every list is always in a consistent state. In theorem 5.7 is proved that the algorithm calculates all temporally maximal PSE. Therefore the algorithm is correct.

5.6 Description of the implementation

The algorithm is implemented in C++.
In the next section all the most important elements of the program are presented.

- **Graph:** this class describes a graph. The graph is represented by a vector of integers in which entries edges and vertexes are mapped as integers (see definition 2.1).
- **ListNode:** this class represents a PSE. It contains the graph, the period, the start and end indexes, and the value of the support.
- **Subgraph hash map:** the subgraph hash map was implemented using the Google dense_hash_map library optimized for speed over memory usage. Dense_hash_map (*key,data,hash_function,equalkey*) is a Unique Pair Associative Container that associates objects of type Key with objects of type Data. In this hash map two elements cannot have keys that compare equal using *EqualKey*. Dense_hash_map is different from other hash-map implementations for its speed and for the ability to save and restore contents to disk. On the other hand, this hash-map implementation can significantly use more space than other hash-map implementations. We use a class *Graph* as key, a class *Descriptor* as data, an implementation of Daniel J. Bernstein's hash_djb2 function as *hash function*, and we create a function that compares two graphs returning true if the two graphs are equal and false otherwise.
- **Descriptor:** it is the object stored in the list contained in the hash map. It describes a PSE that is flushed in output and therefore it has period, support, start and end indexes.
- **Hash_djb2 function:** djb2 is the algorithm which was first reported by Dan Bernstein many years ago in comp.lang.c. The function is the following:

```
unsigned long hash(unsigned char *str)
(const vector<int> s) const
unsigned long hash = 5381;
for  c ← 0 to  s.size()
hash = ((hash << 5) + hash) + s.at(c);
return hash;
```

- **MCS:** for every timestep G_t we have to calculate the MCS between elements of some lists. For every list *L,* every node *N* of the list with graph *G* has all successors with graphs *G'* that are proper subgraphs of *G*. Therefore when at timestep *t* we have to update a single list calculating the MCS between every node and G_t, instead of using G_t every time, we can use the MCS computed at the previous node, thus saving some comparisons.

5.7 Time and space complexity

Time complexity: from proposition 5.3, there are exactly *p* projections for a given period *p*. Now from proposition 3.5 the length of every projection is $|\pi_{p,m}| = \lceil (T - m)/p \rceil$.
Since the algorithm creates a new listnode for every element of the projection the maximum number of listnode is the length of the projection.
For every timestep *t* and for every list, in the worst case the algorithm calculates the MCS for every node of the list and G_t.
Therefore the number of MCS is:

$$\sum_{p=1}^{\mathrm{Pmax}} \sum_{m=0}^{p} \sum_{j=0}^{\lceil (T-m)/p \rceil} j$$

The summations are: for every period *p*, for every projection with period *p* the algorithm creates a list. The number of elements in the list is increased by one at every step. Therefore, also the number of MCS to calculate is increased by one at every step, from 0 to the maximum length of the projection.
The solution of the first summation is $O(T^2/p^2)$. Solving the second summation we have $O(T^2/p^2)p$ that is $O(T^2/p)$.
The last summation is: $O(T^2 \sum_{p=1}^{\mathrm{Pmax}} 1\backslash p$) that is $O(T^2 \ln (P_{max}))$.
Since the MCS can be computed in $O(V+E)$ time, the total complexity of the basic algorithm (without subsumption) is $O((V+E)\, T^2 \ln (P_{max})$.
Since P_{max} is unrestricted in the worst case, its maximum value is $O(T/\sigma)$. Therefore the complexity time in the worst case is $O((V+E)\, T^2 \ln (T/\sigma)$ that is smaller by a factor *T* than PSEMiner [1][2].
It should be observed that only in the worst case the innermost summation is completely calculated because the number of elements could be less than the upper bound for three reasons:

- If the MCS between G_t and the graph *F* at node *N* is empty, then node *N* is (eventually) printed, and deleted. Since the followers of *N* all have a graph that is a subset of *F,* their intersection with G_t is also empty, and no

MCS computation is needed. The nodes are (eventually) printed, and deleted;

- if two consecutives nodes of a list have the same graph, then the node with support set that is not temporally maximal is deleted;
- the MCS between G_t and the graph *F* at node *N* is equal to *F*. Since the graphs of the following nodes in the list are all subsets of *F*, the MCS will be equal to those graphs as well, without need to be calculated.

Space complexity: for every period *p* there are *p* projections with $O(T/p)$ elements. Therefore the number of listnodes for every period is $O(T)$.
Every listnode contains an associated graph. Therefore the total space complexity is $O(P_{max}(V+E)T)$. In the worst case P_{max} is unresctricted ($P_{max}=O(T/\sigma)$), so the space complexity is $O((V+E)T^2/\sigma)$.

6 Experimental evaluation

In this section the performances and the behavior of ListMiner are compared with those of PSEMiner. Two real-world dynamic social networks are used for the evaluation. Artificial datasets are also created to better understand the characteristics of every algorithm, highlighting differences, weak and strong points.
Both algorithms are implemented in C++. The experiments were run on a dual-core Intel Core(TM)2 duo T7300 2.0 GHz, 2 GB of RAM, running Linux Ubuntu. Both algorithms use the google sparsehash library, that therefore be installed in the system.
In all the experiments, the reported computation time is the sum of the user (computation) and kernel (I/O, etc.) CPU time. Memory usage is the maximum resident set size reported by the Linux proc filesystem.

6.1 Datasets description

Here follow a detailed description of the datasets used in the experiments. Table 6.1 summarizes their parameters (number of timesteps, number of vertexes, and the maximum tested period).

6.1.1 Real data

Dynamic networks were collected from two sources, covering a range of human interaction dynamics.
Enron e-mails. The Enron e-mail corpus is a publicly available database of e-mails sent by and to employees of the now defunct Enron corporation. Timestamps, senders and lists of recipients were extracted from message headers for each e-mail on file. The quantization timestep is a day, with a directed (unweighted) interaction present if at least one e-mail was sent between two individuals on a particular day.
Reality mining. Cellphones with proximity tracking technology were distributed to 100 students at the Massachusetts Institute of Technology over the course of an academic year. The timestep quantization was chosen to be 4 h.

6.1.2 Artificial data

Artificial data are used to better understand the performances of the algorithms. The aim of this set of experiments was to understand why and when an algorithm outperforms the other. The following datasets were created using a java program based on the function Java.util.random().
Worst case: this dataset contains 150 timesteps and represents the worst case described in proposition 3.3. In this dataset the maximum number of periodic patterns is generated. Therefore in this dataset the MCS of any $s \geq \sigma$ consecutive positions of any projection $\pi_{p,m}$ is a different PSE. To do this a different edge is inserted in every of the $s \geq \sigma$ consecutive positions of any projection $\pi_{p,m}$.

Maximal case: this dataset contains 800 timesteps and represents the case described in theorem 3.8: for every s =σ =2 consecutive positions of every projection there is a different PSE. The difference from the worst case is that only in sequences of s=σ=2 consecutive positions of every projection there is a different PSE. Differently, in worst case, every sequence of s ≥ σ=2 consecutive graphs of every projection is a PSE.

Experiment 1.1: this is a dataset with 800 timesteps. For every timestep, a random graph with 15 different elements from edges and vertexes is created. The graph is mapped into a sequence of 15 different random numbers from 0 to 300.

Experiment 1.2: this is a dataset with 800 timesteps. For every timestep, a random graph with 15 different elements from edges and vertexes is created. The graph is mapped into a sequence of 15 different random numbers from 0 to 50.

Experiment 1.3: this is a dataset with 800 timesteps. For every timestep, a random graph with 15 different elements from edges and vertexes is created. The graph is mapped into a sequence of 15 different random numbers from 0 to 3000.

Experiment 2.1: this is a dataset with 2000 timesteps. For every timestep, a random graph with 15 different elements from edges and vertexes is created. The graph is mapped into a sequence of 15 different random numbers from 0 to 300.

Experiment 2.2: this is a dataset with 2000 timesteps. For every timestep, a random graph with 15 different elements from edges and vertexes is created. The graph is mapped into a sequence of 15 different random numbers from 0 to 50.

Experiment 2.3: this is a dataset with 2000 timesteps. For every timestep, a random graph with 15 different elements from edges and vertexes is created. The graph is mapped into a sequence of 15 different random numbers from 0 to 3000.

Dataset	**Timestep**	**Vertexes**	**P_{max}**
Enron	2588	82614	40
Reality mining	2940	100	40
Maximal case	800	140802	50
Worst case	150	65486	unrestricted
Experiment 1.1	800	300	30
Experiment 1.2	800	50	30
Experiment 1.3	800	3000	30
Experiment 2.1	2000	300	50
Experiment 2.2	2000	50	50
Experiment 2.3	2000	3000	50

Table 6.1: parameters of the datasets.The number of vertexes is the total size of the population.

The choice of the parameters for the artificial networks was specifically thought so to have networks with low density (experiments 1.3 and 2.3), medium density (esperiments 1.1 and 2.1), and high density (experiments 1.2 and 2.2) of periodic patterns. As the experiments in the next subsection will show, pattern density is a parameter that has a high influence on the performances of the algorithms.

6.2 Experimental Time Analysis

This section shows the analysis of the comparison of execution times between ListMiner and PSEMiner.
Table 6.2 reports the execution times of two experiments (parameters: σ=3, unrestricted P_{max}) without subsumption (All patterns) and with subsumption (Parsimonious).

	ListMiner		**PSEMiner**	
Dataset	**All patterns**	**Parsimonious**	**All patterns**	**Parsimonious**
Enron	21 s	25 s	0,8 s	1 s
Reality mining	32 s	114 s	720 s	864 s
Maximal case	26,4 s	28,5 s	465 s	472 s
Experiment 1.1	3,4 s	4,7 s	7,4 s	8,5 s
Experiment 1.2	6,6 s	6,8 s	29,2 s	30,9 s
Experiment 1.3	3,0 s	3,2 s	0,8 s	0,7 s
Experiment 2.1	21,1 s	22, 5 s	200 s	206 s
Experiment 2.2	39, 5 s	530 s	712 s	800 s
Experiment 2.3	16, 3 s	16,8 s	7,5 s	7,2 s

Table 6.2: execution times of the two algorithms, the proposed ListMiner, and the state-of-the-art PSEMiner, with (Parsimonus) and without (All patterns) subsumption for support threshold σ=3 and P_{max} unrestricted.

As showed in Table 6.2, ListMiner is faster than PSEMiner in all the experiments, with exception of Enron, experiment1.3 and experiment2.3. This is due to the different density of the dataset under analysis, as explained below.
The Reality mining dataset has a high density of periodic patterns (the number of vertexes is low (100) and the number of timesteps is high (2940)). Similarly, in the datasets of experiment1.2 and experiment2.2, where every graph is composed by a sequence of 15 casual numbers from 0 to 50, the probability that the MCS between two graphs is not empty is high. As a consequence, the density of periodic patterns is also high.
In this *high-density context* PSEMiner is much slower than ListMiner. In particular, the comparison between the results of experiment1.2 and

experiment2.2, tell us that the higher the number of timesteps, the bigger is the difference of the time performances in favor of ListMiner.
The decrease of speedup in experiment2.2, when the control of subsumption is included (18x for all patterns, and 1,5x including subsumption), is due to the large computational time need to check subsumption. The complexity time of the *subsumed()* function, that verifies if a specific PSE is subsumed, is the same for both algorithms. Since in experiment2.2 there is a high number of periodic patterns, the subsumption function takes most of the execution time. In this case, the time complexity for checking subsumption prevails on the time to mine all PSE. Hence, the difference of execution time between PSEMiner and ListMiner is relatively small.
Although the theoretical bound of the time complexity of ListMiner is better than that of PSEMiner, in Enron dataset, experiment1.3, and experiment2.3 PSEMiner is faster.
These are datasets that have a common property: the density of periodic patterns is low.
In fact, the two artificial datasets have been created to highlight the behavior of the two algorithms when the number and density of periodic patterns are low by choosing 15 random vertexes among 3000 at each timestep. Since the number of vertexes is very high, the probability that the MCS between two graphs is not empty is low. This is confirmed observing the number of mined patterns in table 6.3 (when σ=3), and in table 6.4 (when σ=2). As a conclusion, *low-density* experimental settings favor PSEMiner.

Dataset	**Subsumption**	**No subsumption**	**Pattern Theoretical bound**
Enron	84017	84056	47992640
Reality mining	102669	102745	63037927
Maximal case	87780	1*	3575039
Experiment 1.1	5292	5292	3575039
Experiment 1.2	95677	95687	3575039
Experiment 1.3	53	53	3575039
Experiment 2.1	33231	33231	26009160
Experiment 2.2	595162	595172	26009160
Experiment 2.3	393	393	26009160

Table 6.3 shows the number of patterns mined with and without subsumption, and the theoretical upper bound on the number of patterns for σ=3. (*) maximal case is a dataset that is built in the following way: every timestep contains the vertex “1”. Now, as described in theorem 3.8: for every s = σ =2 consecutive positions of every projection there is a different vertex and thus a different PSEs. So the only graph that appears more than 3 times is the graph with only vertex “1”.

Dataset	Subsumption	No subsumption	Pattern Theoretical bound
Enron	125088	125428	45276939
Reality mining	630798	629496	59533249
Maximal case	372001	372001	2139075
Experiment 1.1	167937	167910	2139075
Experiment 1.2	400935	400624	2139075
Experiment 1.3	23071	23063	2139075
Experiment 2.1	1051439	1051378	27631021
Experiment 2.2	2502547	2501844	27631021
Experiment 2.3	144420	144436	27631021

Table 6.4: shows the number of patterns mined with and without subsumption, and the theoretical upper bound on the number of patterns for σ=2.

The datasets of experiment1.1 and experiment2.1 are composed by a sequence of graphs where, at each timestep, 15 random vertexes from 0 to 300 were selected as active. Therefore, the probability that the MCS between two graphs is not empty is in between that of the previous experiments.
In this *medium-density* contexts, ListMiner runs faster than PSEMiner, but the speed-up is lower (2x and 10x vs 4x and 18x) than in experiment1.2 and experiment2.2. This is due to the smaller number of patterns to mine.
These results can be explained from the analysis of theoretical time complexity. The upper bound of PSEMiner is *O((V+E)T N)* where *N* is the number of nodes in the pattern tree. Table 6.5 shows the number of nodes and descriptors created by PSEMiner during the elaboration. The complexity time of PSEMiner strongly depends on the number of nodes in the pattern tree. In fact, observing Table 6.5, it can be noticed that in datasets where PSEMiner have a low ratio (with respect to the maximum) of treenodes, it is faster than ListMiner. A node remains in the tree until it has at least one descriptor *D* such that $next(D) \leq P_{max}$ and $next(D) \geq t$, where *t* is the timestep that we are processing. PSEMiner creates a node in the tree when the MCS between two graphs is a new subgraph in the pattern tree. Therefore *N* is higher when in the dataset there are a lot of different periodic subgraphs. For these reasons the higher the density of periodic patterns, the higher is the number of nodes in the tree, and the worst are the performances of PSEMiner.

Dataset	Number of nodes	Number of descriptors	Nodes Theoretical bound	% number of nodes
Enron	24583	41095	45276939	0,05%
Reality mining	87780	277613	59533249	0,1%
Maximal case	160153	213084	2139075	7,4%
Experiment 1.1	17135	85245	2139075	0,8%
Experiment 1.2	84350	192461	2139075	0,4%
Experiment 1.3	3408	11518	2139075	0,16%
Experiment 2.1	61480	532315	27631021	0,22%
Experiment 2.2	347616	1199625	27631021	1,26%
Experiment 2.3	7355	72373	27631021	0,03%

Table 6.5: reports the number of nodes and descriptors of PSEMiner when P_{max} is unrestricted and σ=3.

In particular, analyzing the *maximal* case the value of N is $O(T^2/\sigma)$, and the value of P_{max}, when it is unrestricted, is $O(T/\sigma)$. Therefore the time complexity of PSEMiner becomes $O((V+E)\ T^3/\sigma)$. On the other hand, ListMiner has a time complexity which is $O((V+E)T^2 \ln(T/\sigma))$. This explains why ListMiner is faster than PSEMiner.
From the characteristics of datasets it is known that in Enron, in experiment1.3 and in experiment2.3 the density of periodic patterns is low. Since the number of nodes in the pattern tree depends on the density of periodic patterns, the complexity of PSEMiner is much lower than the upper bound. In fact, it is $O((V+E)\ T\ N)$ with $N << O(T^2\ ln\ (T/\ \sigma))$. The complexity time of ListMiner is not strongly influenced by the number of the periodic patterns. In this low-density case, the time complexities are: $O((V+E)TN)<O((V+E)T^2 ln(T/\sigma))$ since $N<<O(T^2 \ln (T/\ \sigma))$. For this reason the execution time of PSEMiner is lower than that of ListMiner.
On the contrary, in Reality mining, experiment2.2, and experiment3.2 the density of periodic patterns is very high. Consequently, also the number of nodes in the pattern tree is high. The complexity time of PSEMiner is $O((V+E)TN)$ with N near to the upper bound $O(T^2 \ln (T/\ \sigma)$. Hence, in a *high-density* setting the time complexities become:
$O((V+E)T\ N) > O((V+E)\ T^2\ ln\ (T/\ \sigma))$ since $N \approx O(T^2\ ln\ (T/\ \sigma))$.

In the worst case PSEMiner ends in 587 s and ListMiner ends in 216 s as showed in table 6.6. Therefore, as expected from theoretical analysis, the experiments confirm that ListMiner is actually more efficient than PSEMiner in the worst case hypothesis.

	Execution time	**Number of nodes**	**Number of descriptors**
PSEMiner	587 s	10508	11026
ListMiner	216 s	/	/

Table 6.6: execution time of the two algorithms in the worst case with P_{max} unrestricted and σ=2. For PSEMiner also the number of nodes and the descriptors are reported.

We conclude by observing that in the typical online analysis scenario with a restricted P_{max}, ListMiner takes few seconds to execute and uses less than 15 MB of memory in all cases as we can see in table 6.7.

Dataset	**Time**	**Memory**
Enron	1,8 s	340 KB
Reality mining	2,8 s	1,1 MB
Maximal case	6 s	256 KB
Experiment 1.1	0,8 s	604 KB
Experiment 1.2	1 s	2,8 MB
Experiment 1.3	0,5 s	472 KB
Experiment 2.1	1,1 s	14,7 MB
Experiment 2.2	7 s	2,3 MB
Experiment 2.3	0,9 s	10,8 MB

Table 6.7: reports the execution time and the memory usage of ListMiner with σ=3, with subsumption, and with restricted P_{max}.(see table 6.1) .

6.3 Experimental Space Analysis

This section presents the analysis of the memory requirements of ListMiner and PSEMiner. Table 6.8 shows the results for the comparison of the memory usage of the algorithms with P_{max} unrestricted, and σ=3.
ListMiner uses less memory than PSEMiner in experiment1.1, experiment1.2, experiment2.1, and experiment2.2. In experiment1.3 and in experiment2.3 the memory used by ListMiner is slightly higher, but approximately of the same order. In the other datasets ListMiner uses more memory than PSEMiner.
This behavior can be justified by theoretical analysis of the space complexity. The space complexity of PSEMiner is $((V+E)N+ P^2_{max} + G)$ where N is the number of nodes in the tree, G is the number of descriptor, and V, E are the number of vertexes and edges, respectively. The space complexity of ListMiner is always $((V+E)\ T^2/\sigma)$. Since the most part of the memory is used to store graphs, the dominant term of the space complexity expression in PSEMiner is

(V+E)N. Therefore, when the number of treenodes is low, the space complexity of PSEMiner is lower than the space complexity of ListMiner.

Dataset	**ListMiner**	**PSEMiner**
Enron	1 GB	150 MB
Reality mining	353,6 MB	157 MB
Maximal case	503,6 MB	200 MB
Worst case	790 MB	400 MB
Experiment 1.1	12,7 MB	35 MB
Experiment 1.2	19,6 MB	94 MB
Experiment 1.3	10,3 MB	6 MB
Experiment 2.1	78,4 MB	298 MB
Experiment 2.2	118 MB	881 MB
Experiment 2.3	63,8 MB	52 M

Table 6.8: reports the memory usage of the two algorithms with σ=3 and P_{max} unrestricted.

6.4 Mined Patterns Analysis

This section reports the analysis of the patterns mined by the algorithms. The analysis was performed in two directions: 1) number of periodic patterns vs support; 2) number of periodic patterns vs period.

6.4.1 Periodic Pattern distribution and Support values

Figures from 6.9 to 6.14 in the next page show, for each experiment on an artificial dataset, the distribution of the number of patterns (y-axis) with respect to a given support (x-axis).

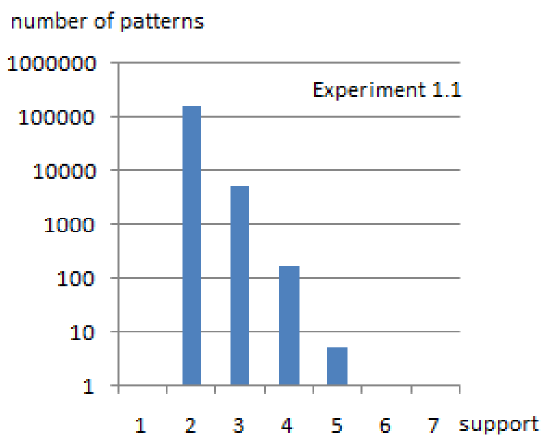


Fig 6.9: Experiment 1.1
(T=800,V=300,P_{max} unrestriced)

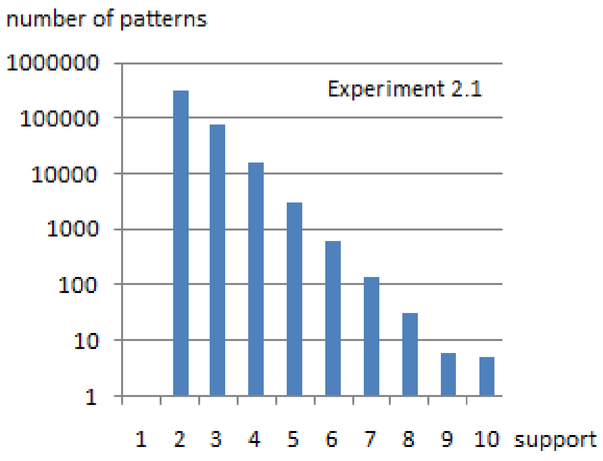


Fig 6.10: Experiment 2.1
(T=2000,V=300,P_{max} unrestriced)

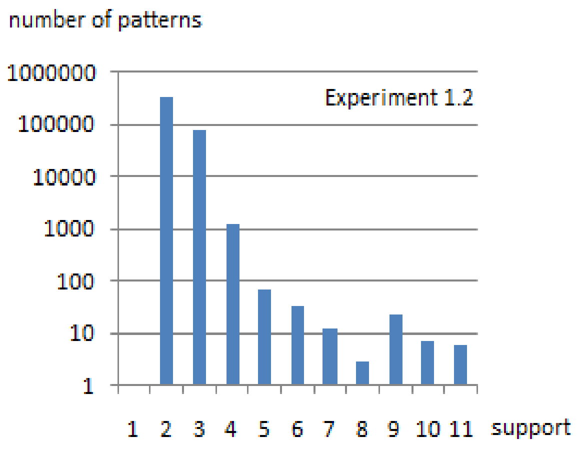


Fig 6.11: Experiment 1.2
(T=800,V=50,P_{max} unrestriced)

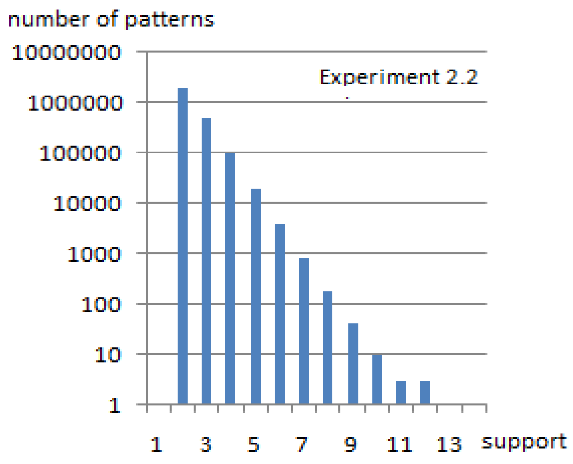


Fig 6.12: Experiment 2.2
(T=2000,V=50,P_{max} unrestriced)

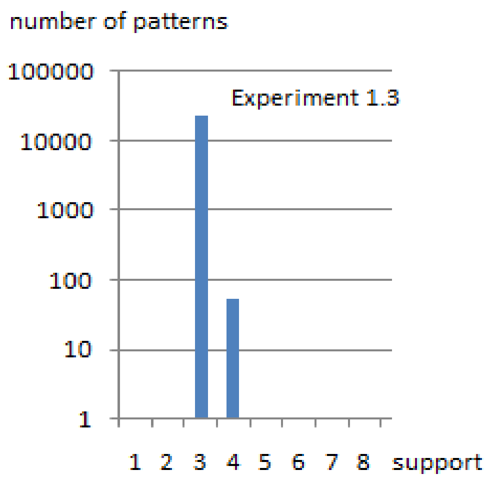


Fig 6.13: Experiment 1.3
(T=800,V=3000,P_{max} unrestricted

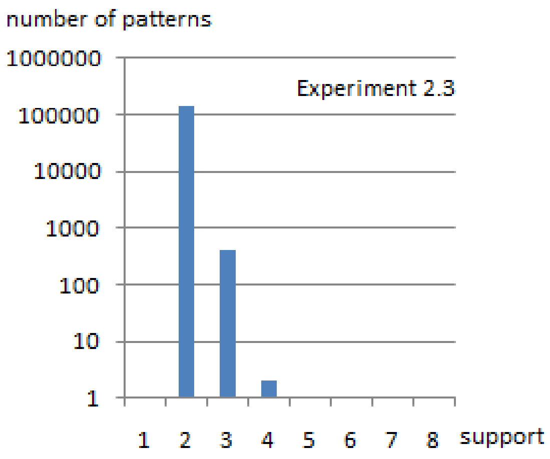


Fig 6.14: Experiment 2.3
(T=2000,V=3000,P_{max} unrestricted)

Experiments showed that the highest number of periodic patterns, and the highest values of support occurs on *high-density* periodic patterns frameworks (experiments 1.1 and 2.1 in Fig.6.11 and 6.12, respectively). This is expected since the probability of vertexes to occur at some timestep is higher than in the other cases.

Fig. 6.13 and 6.14 shows the results for *low-density* experiment1.3 and experiment2.3. Here the values of support are limited to 3, 4 and 5. This is because the number of vertexes is high, and therefore the probability of a particular subgraph to be periodic for a long time is low.

Fig. 6.9 and 6.10 show the experiment for the medium-density setting. In these datasets the number of patterns is smaller than in the datasets with high density (experiment1.2 and experiment2.2), but bigger than low density datasets (experiment1.1 and experiment2.1). Moreover, the behavior with 800 timesteps is more similar to the low-density case, while with 2000 timesteps the figures looks much more the high-density context but the number of patterns is different. This depends on the higher number of periodic patterns of experiment2.1. In fact the higher the number of timesteps, the higher is the probability to have a high number of different values of support.

Figure 6.15 shows a comparison between the two real datasets, Enron and Reality Mining. It can be seen that Reality presents a behavior that is more periodic than Enron (number of periodic patterns is higher at all supports).

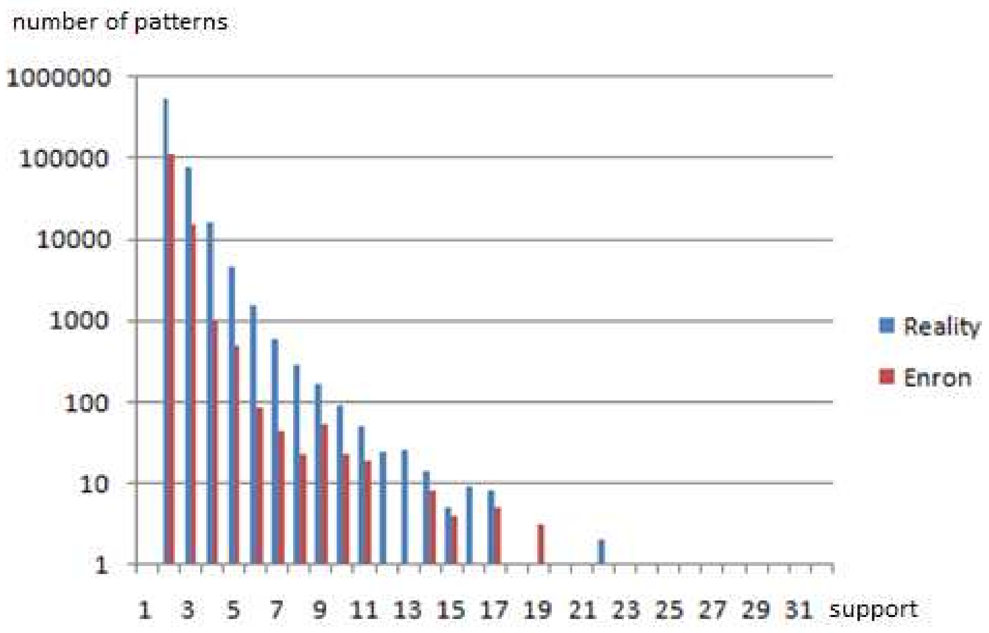


Fig 6.15: Reality(T=2940,V=100, P_{max} unrestriced),Enron(T=2588,V=82614, P_{max} unrestriced)

The experiments from Fig. 6.16 to 6.22 shows the distribution of patterns for each possible period, when P_{max} is unrestricted and σ=3.

Figures 6.16 and 6.17 show the distribution for experiment1.2 and experiment2.2. Experiments confirmed that these are the datasets with the highest number and density of periodic patterns, as expected.

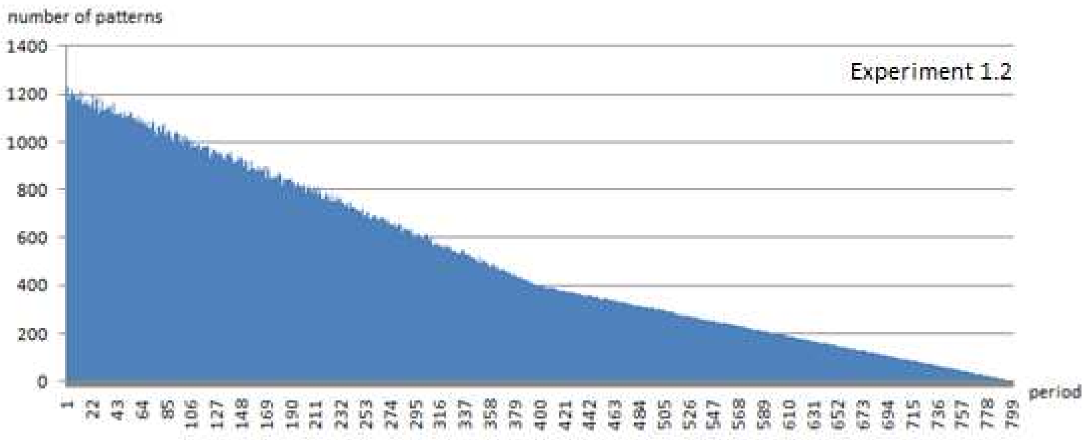


Fig 6.16: Experiment 1.2(T=800,V=50,P_{max} unrestriced)

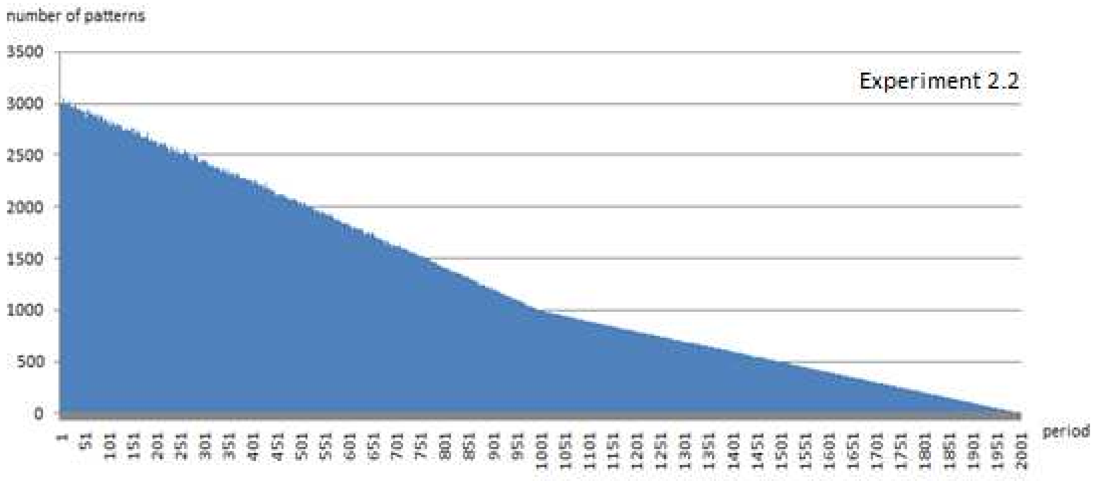


Fig 6.17: Experiment 2.2(T=2000,V=50,P_{max} unrestriced)

On the contrary, experiment1.3 and experiment2.3 are the datasets with the lowest number and density of periodic patterns (see figures 6.18, 6.19).

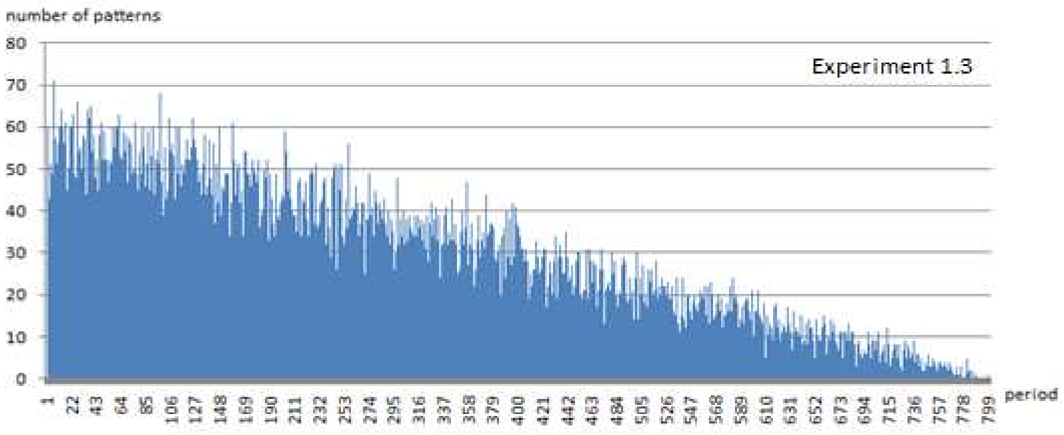


Fig 6.18: Experiment 1.3(T=800,V=3000,P_{max} unrestriced)

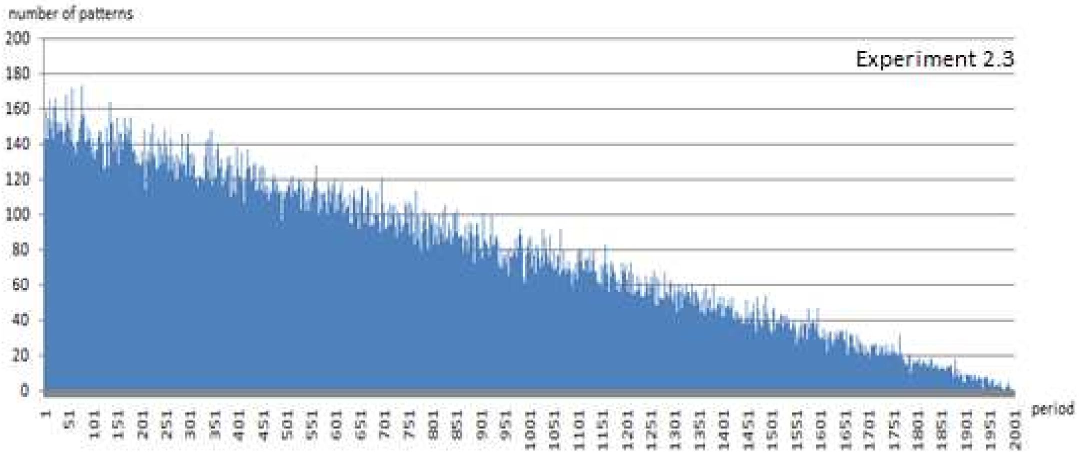


Fig 6.19: Experiment 2.3(T=2000,V=3000,P_{max} unrestriced)

Finally, experiments1.1 and exxperiments2.1 (see figures 6.20 and 6.21) showed to have a distribution of periodic patterns that is in between.

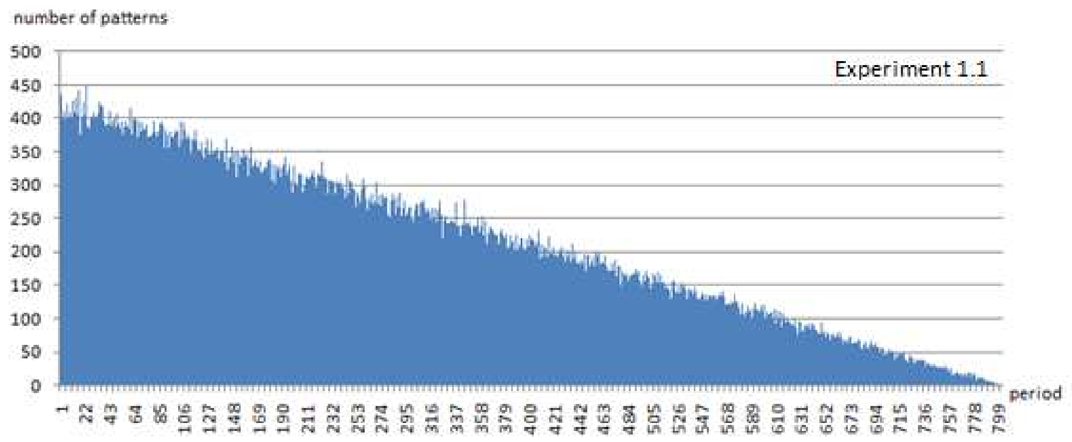


Fig 6.20: Experiment 1.1(T=800,V=300,P_{max} unrestriced)

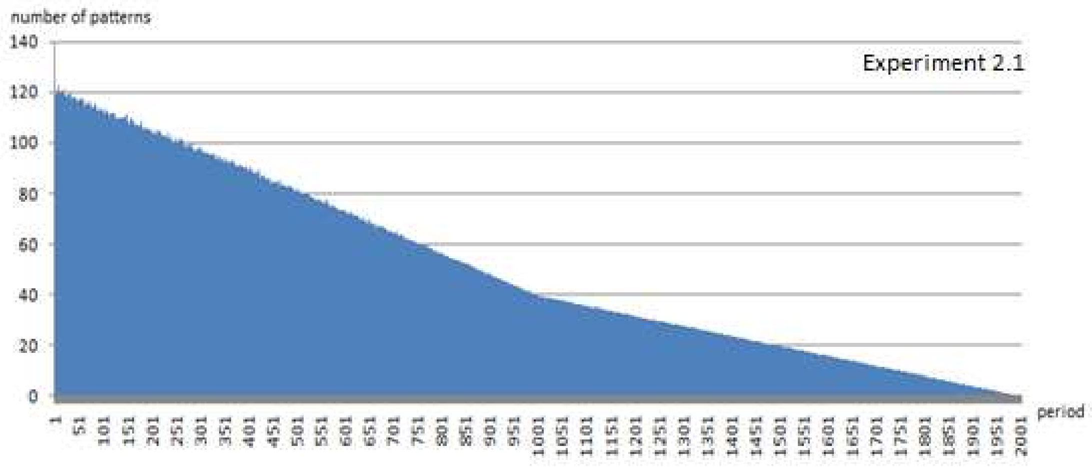


Fig 6.21: Experiment 2.1(T=2000,V=300,P_{max} unrestriced)

Figures 6.22 and 6.23 show the results for the two real datasets. Enron appears to be less periodic than Reality, in fact both the number of periodic patterns, and the number of different periods is lower. The density of periodic patterns in Reality is higher.

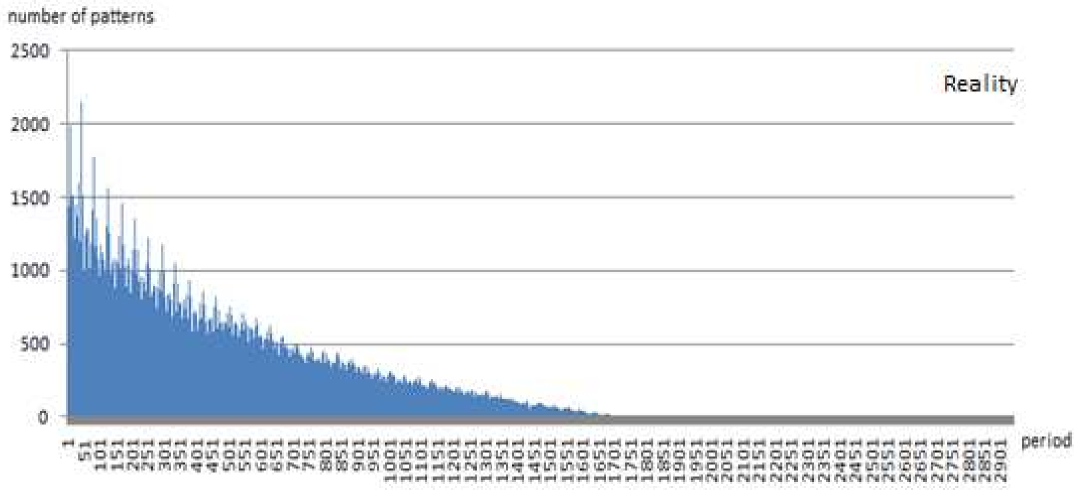


Fig 6.22: Reality(T=2940,V=100, P_{max} unrestriced)

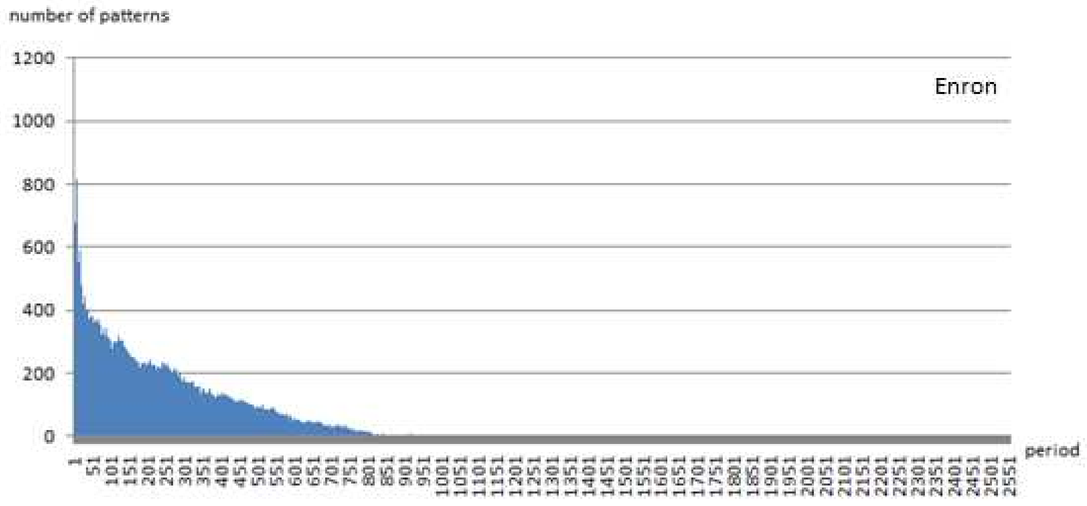


Fig 6.23: Enron(T=2588,V=82614, P_{max} unrestriced)

Figure 6.24 in the next page shows global periodicities in real networks when P_{max} is restricted. It can be observed that Enron and Reality datasets have strong daily and weekly periodicities, as might be expected from human interactions despite the fact that the interactions occur through different mechanisms in each dataset, e-mail in the Enron dataset, and physical proximity in the Reality mining dataset. In fact in figure 6.23 we can observe that we have two peaks for p=1 and p=7 for the Enron dataset. In Reality, where the quantization time is 4 hours, we can note that there are peaks for every multiple of p=6 (one day). In p=42 (one week) there is an high number of periodic patterns that confirms the weekly periodicity.

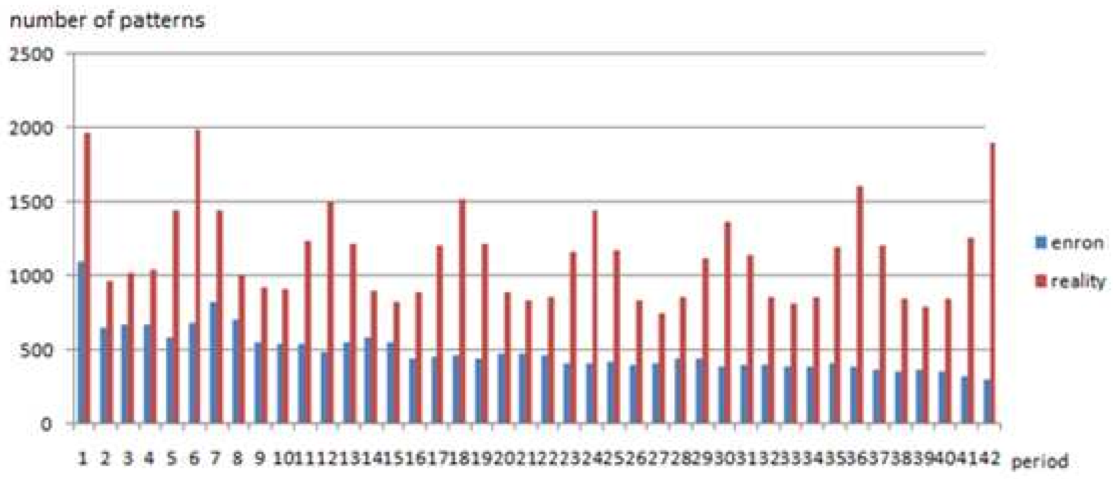


Fig 6.24:Global periodicity for restricted maximal periods in real dynamic networks. Reality P_{max}=42, Enron P_{max}=42,σ=3.

7 *Period mining*

7.1 Problem definition

In this chapter a different mining problem is discussed. Let us model a dynamic network as a string in which each symbol corresponds to a subgraph. What was called a *period* in previous chapters, in this framework is called a *cadence*, while the concept of period requires further constrains. This chapter presents an algorithm to mine periodic (as in the string framework) patterns in dynamic networks. Since perfect periodicity is rare in this context, some imperfections between instances of subgraphs will be allowed. Here follows some related definitions taken from [30].

Definition 7.1: Integers $t_1<t_2<t_3,<...,<t_n$ are a cadence for word $x_1x_2x_3...x_r$ if $x_{t1}= x_{t2}=...= x_{tn}$. In this case n is called the order of the cadence.

It can be seen that what was called a period in the previous chapters is indeed a type of cadence. Let S be a finite subset of the alphabet, a cadence of type S is a cadence of the form $\alpha S + \beta$ (i.e., an arithmetic cadence with common difference α when $\alpha,\beta>0$).

In this framework a period is defined as follows:

Definition 7.2: A string z has a *period* w if z is a prefix of w^k for some integer k. Alternatively a string w is a period of a string z if $z = w^l v$ and v is a possibly empty prefix of w.

Often, when this does not cause confusion, we will use the word “period” also to refer to the length or size $|w|$ of a period w of z.

Definition 7.3: A non-empty string w is a *border* of a string z if z starts and ends with an occurrence of w, i.e. $z=uw$ and $z=wv$ for some possibly empty strings u and v.

Clearly a string is always a period (resp. border) of itself. This period (resp. border) is called *trivial* period (resp. border). It is immediate to see that two consecutive occurrences of a word may overlap only if their distance equals one of the periods of w. A string can have several periods, and corresponding borders. The smallest (resp. longest border) period is the period (resp. the border) of the string.

Definition 7.4: A *sliding window* of length s for a dynamic network G is a sequence of s consecutives timesteps $G_x, G_{x+1},...., G_{x+s-1}$.

When considering datasets based on real interactions within a population, it is unlikely that subgraphs repeat themselves exactly, and periodically, for a long time. Therefore, some relaxation is needed, and the concept of distance between two graphs is used to account for the presence of such variability.

Definition 7.5: Given two graphs $G_1=(V_1,E_1)$ and $G_2=(V_2,E_2)$, the *distance* between G_1 and G_2 is the cardinality of the set composed by all vertexes and edges of G_1 and G_2 that are not in the MCS between G_1 and G_2.
Since by definition 2.1 the set representation for every graph is formed mapping each vertex and edge to a unique integer, $d(G_1,G_2)=|(V_1 \cup V_2)-(V_1 \cap V_2)|+|(E_1 \cup E_2)-|E_1 \cap E_2)|$.

A new parameter d_{max} is introduced in the problem to represent the maximum allowed distance between two graphs. The problem becomes:

Definition 7.6: Given a dynamic network G and the maximum distance d_{max} between two graphs, for every possible window G_x, G_{x+1},...., G_{x+s-1} $1 \le x \le T-s$, of length *s,* we want to calculate the period *p* of the graphs inside the window allowing some imperfections. Precisely, the distance $d(G_i, G_{i+p})$ must be less than d_{max} for every $x \le i \le s-p$.

7.2 Complexity of periods mining

In this section the problem is shown to belong to the complexity class *P* and it is therefore tractable.

Theorem 7.7: The periods mining problem is in *P*.

Proof: since a dynamic network can be considered like a string of subgraphs, the problem is reduced to find the period of a given string. It is obvious that in a dynamic network with *T* timesteps there are $T-s+1$ windows of length *s*. To find the period of a given string with length *s*, the maximum number of comparison is $O(s^2)$. Since the distance between two graphs can be found in $O(V+E)$ (we have to calculate the union and intersections between edges and vertexes of the graphs), the cost of a single comparison is $O(V+E)$. Therefore the total time complexity is $O((V+E)Ts^2)$.

□

7.3 Description of the algorithm

For every fixed size window, the algorithm calculates the period by finding the maximum border of graphs in the window. The trivial border is not considered. The algorithm starts controlling if the "string" in the window matches, against left shifts of *i* position of itself $1 \le i \le s-1$. Two graphs match if their distance is less than or equal to d_{max}. This problem can be easily reduced to find the maximum border of a given string, which is a well known problem. To solve this problem the algorithm implemented to calculate the *failure function* of the KMP algorithm [31] is adapted. The *failure function* calculates, for all prefixes of the string, the longest prefix that is also a suffix. Therefore, the algorithm, for every window, calculates the longest border of the window. The key difference between the *failure function* and the proposed algorithm is that, for the problem at hand, the concept of match between symbols is substituted by the distance between two graphs being below a given threshold. Here follow the pseudocode of the algorithm.

Algorithm Period(*input*)

```
Require: input is a vector that in position input[i] = $G_{i+1}$. We have T timesteps.
Let window be the length of the window.
1:  for (i ← 0 to T-1-s ) do
2:      $G_{new}$= input [i]…..input[i+window-1]
3:      maxborder= FailureFunction ($G_{new}$)
4:      print all graphs from input[i] to input[i+maxborder]
5:  end for
```

Algorithm FailureFunction(G_{new})

```
Require: $G_{new}$ is a sequence of graph length windows; dmax is the maximum distance
between two graphs, distance($G_1$,$G_2$) is a procedure that calculates the distance between
two graphs as explained in definition 8.5 and f(i), $0 \leq i \leq s$, contains the values of the
failure function.
1:  i ← 0
2:  j ← 0
3:  while i < window do
4:      if (distance ($G_{new}$[i], $G_{new}$[ j]) ≤ dmax) then
5:          f (i) ← j+1
6:          i ← i+1
7:          j ← j+1
8:      else if (j>0) then
9:          j ← f (j-1)
10:     else
11:         f (i) ← 0
12:         i ← i+1
13:     end if
14:     end if
15: end while
16: return f (window-1)
```

7.3.1 Correctness

The only difference between the algorithm to compute the period of a graph sequences and the period of a string is that the former uses the *distance* operation between graphs rather than the comparison between symbols of an alphabet. Hence, the correctness of the algorithm follows from the correctness of the algorithm in [31].

7.3.2 Space and time complexity

Space complexity: the algorithm is implemented in place. Other data structures are not necessary, therefore the space complexity is $O(T)$.

Time complexity: the maximum border of the window is calculated for each of the T-s+1 windows of the dynamic network. The distance between two graphs can be calculated in $O(V+E)$. The maximum border can be found in $O(s)$ [31] where s is the length of the window. Therefore the total time complexity is $O((V+E)Ts)$.

7.4 Experimental evaluation

To evaluate the performances of the algorithm the two real-world dynamic social networks, Enron and Reality, were analyzed. The description of these datasets is in section 6.1.1. The algorithm is implemented in C++. The experiments were run on a dual-core Intel Core(TM)2 duo T7300 2.0 GHz, with 2 GB of RAM, running Linux Ubuntu. In all cases, time computation is reported as the sum of the user (computation) and kernel (I/O, etc.) CPU time. Memory usage is the maximum resident set size reported by the Linux proc filesystem. The algorithm was tested for several lengths of the window and the values of the maximum distance d_{max}.

7.4.1 Running time and space occupation

Tables 7.1, 7.2 and 7.3 show the execution time and the memory usage of the algorithm for windows of length 8, 31, 62 days. The parameter d_{max} is set equal to 10.

s=8:

Dataset	**Time**	**Memory usage**
Enron	0,8s	14,6MB
Reality	1,1s	17,1 MB

Table 7.1: execution time when the length of the window is 8 and the maximum distance is 10.

s=31

Dataset	Time	Memory usage
Enron	1,8s	14,6MB
Reality	2,2s	17,1 MB

Table 7.2: execution time when the length of the window is 31 and the maximum distance is 10.

s=62

Dataset	Time	Memory usage
Enron	3,5s	14,6MB
Reality	3,8s	17,1 MB

Table 7.3: execution time when the length of the window is 62 and the maximum distance is 10.

The tables above show that the algorithm takes few seconds (maximum 3,8 s when *s*=62) to execute and uses less than 18 MB of memory in all cases. As expected, increasing the size of the window the execution time increases. The space occupation is not affected by the size of the window because only the dataset is stored in the memory. The execution time and the memory usage in Enron are higher than Reality because Reality has more timesteps.

7.4.2 Patterns and periods distributions

This subsection reports just the experimental results. Global analysis and observations will be reported in the next subsection.

Enron

Figure 7.4 shows the number of subgraphs for a given period value (x-axis) mined when *s*=8 for the three values of d_{max}.

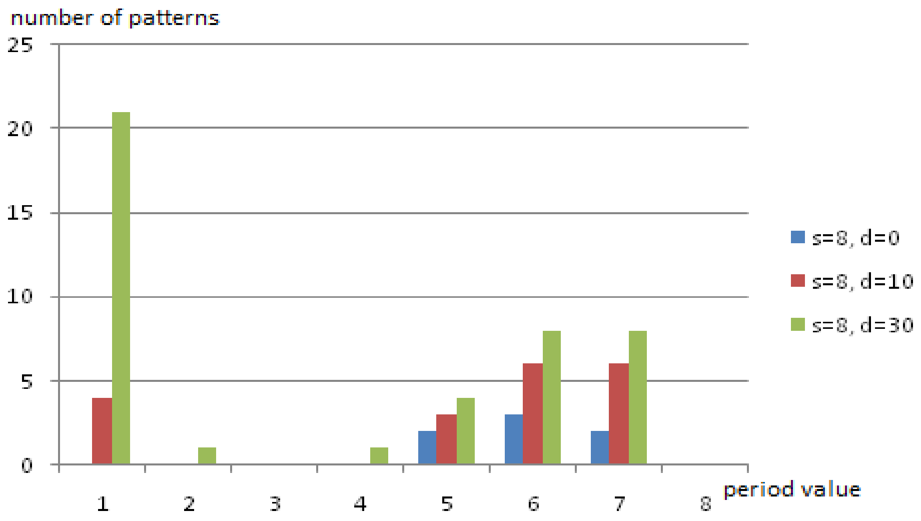


Fig 7.4: the number of subgraphs for every period value when the length of the window is 8 and the value of the distance is 0, 10, 30.

Figure 7.5 shows the number of subgraphs for a given period value (x-axis) mined when *s*=31 for the three values of *d*.

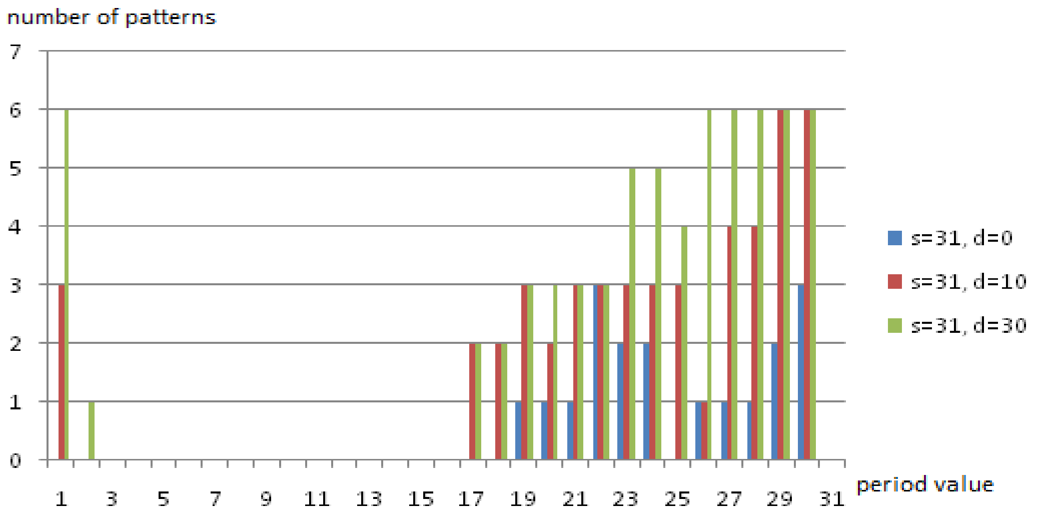


Fig 7.5: the number of subgraphs for every period value when the length of the window is 31 and the value of the distance is 0, 10, 30

Figure 7.6 shows the number of subgraphs for a given period value (x-axis) mined when *s*=62 for the three values of d_{max}.

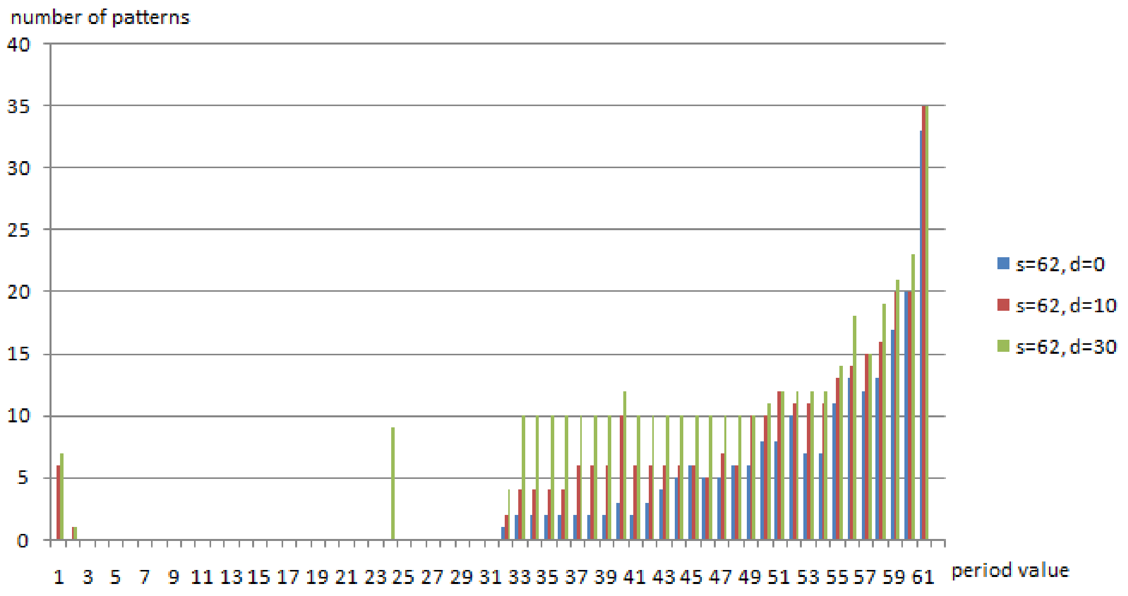


Fig 7.6: the number of subgraphs for every period value when the length of the window is 62 and the value of the distance is 0,10,30

Reality

For Reality mining the timestep quantization was set to one day, not 4 h like in section 6 for a more significant analysis. Therefore the number of the timesteps is 368.

Figure 7.7 shows the number of subgraphs for a given period value (x-axis) mined when *s*=8 for the three values of d_{max}.

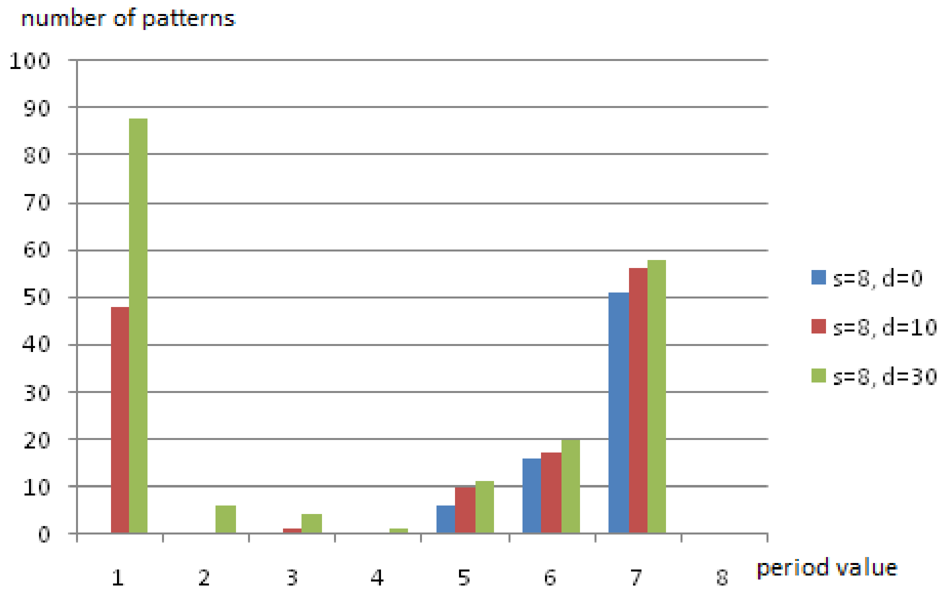


Fig 7.7: the number of subgraphs for every period value when the length of the window is 8 and the value of the distance is 0, 10, 30

Figure 7.8 shows the number of subgraphs for a given period value (x-axis) mined when *s*=31 for the three values of *d*.

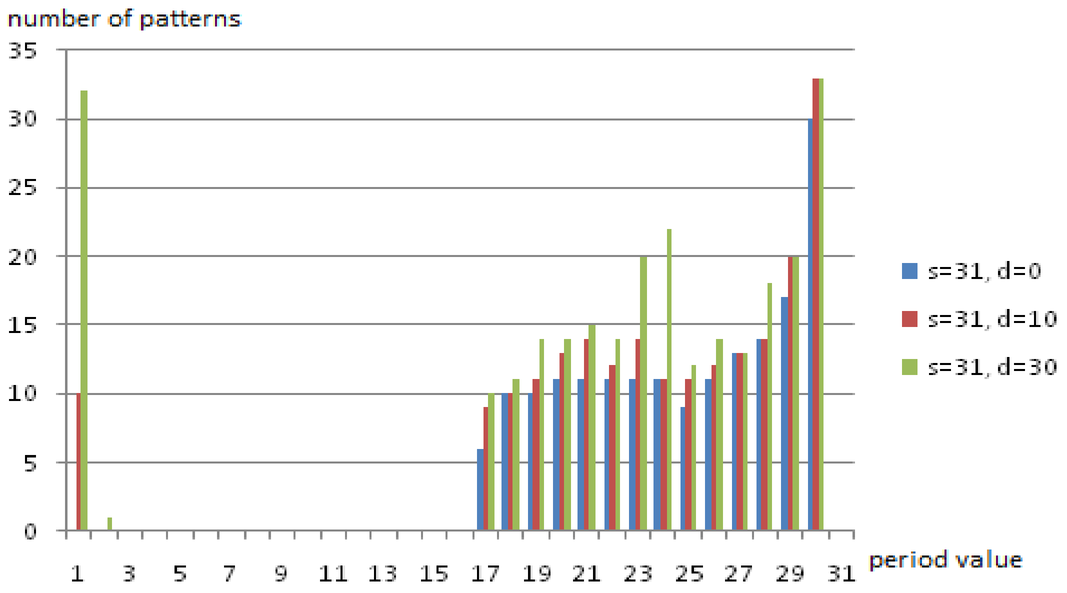


Fig 7.8: the number of subgraphs for every period value when the length of the window is 31 and the value of the distance is 0,10,30

Figure 7.9 shows the number of subgraphs for a given period value (x-axis) mined when *s*=62 for the three values of d_{max}

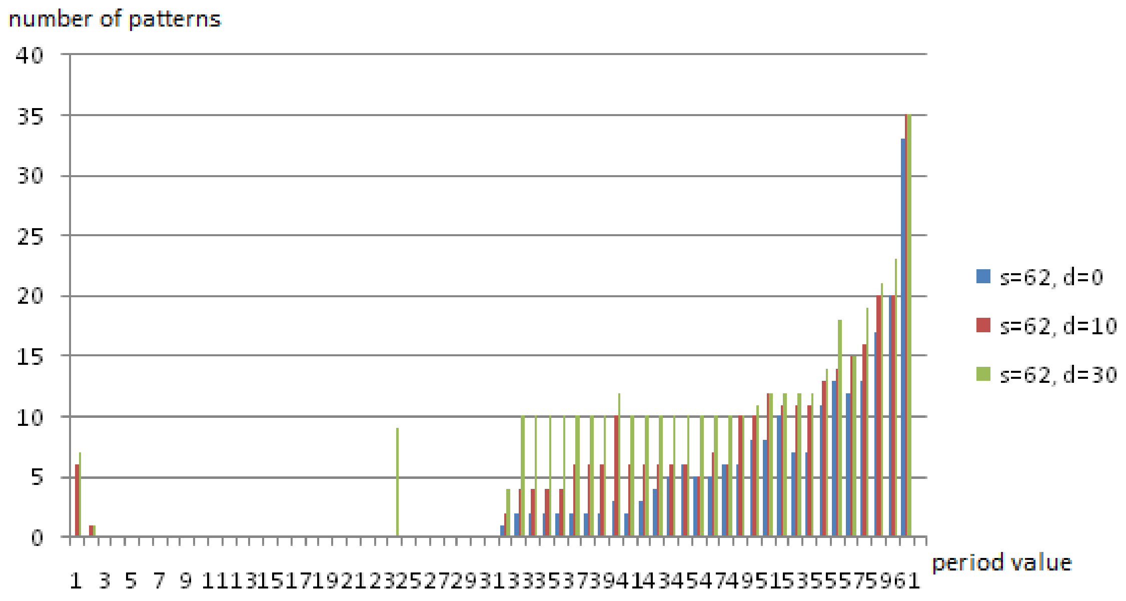


Fig 7.9: the number of subgraphs for every period value when the length of the window is 62 and the value of the distance is 0,10,30

7.4.3 Analysis of results

Analyzing all figures in section 7.4.2 it can be seen that the majority of subgraphs periodicity is equal to 1 (in every figure there is a peak in p=1). This means that there is an high number of subgraphs that are repeated in *s* consecutive timesteps. Therefore the Enron and Reality mining datasets show strong daily behavior, as might be expected by human interactions. The experiments also show that the number of subgraphs for a given period increases when the maximum distance is higher, as expected because some mismatches between subgraph instances are allowed. Another observation that emerges from the analysis is that in Enron dataset the number of periods is similar to the Reality although this last dataset has less timesteps. This means that Reality dataset has more regularities than Enron, which was also confirmed by the analysis in Section 6.4. The number of non trivial periods is low in Enron. This is due to the fact that this dataset represents interactions between a very large number of individuals. Every timestep has a graph with a lot of vertexes and edges, therefore it is difficult to have long periods. This can be understood thinking of the relationships that people have in a company, where every person communicates for short periods of time with a small group of colleagues. In Reality there are more no trivial periods in proportion the total number of the timesteps. This is due to the fact that students usually have friendly relationships that are more durable than work relationships. Another aspect that can be noticed is that, except for period 1, the other periods are near to the trivial period. This means that people either have stable relationships or have relationships after lots of days, for example two weeks or one month.

7.5 From data to knowledge

In this section I want to turn the attention to some qualitatively interesting aspects of the subgraphs that were extracted. Analyzing the output of Enron and Reality datasets, the first interesting observation is that there are some similar patterns that are repeated periodically. This means that humans are divided in groups where they interact among each other. Small patterns (4-12 individuals), that represent a small group, have high supports. Large patterns (30-40 individuals), that represent a large group, have low values of support. This fact can be explained referring to real life where humans have stable and durable interactions with few people, for example friends or family. Individuals have intimate relationships toward what sociologists call *reference groups*. Reference groups are groups which people refer to when evaluating their [own] qualities, circumstances, attitudes, values and behaviors [32]. This can be observed because periodic entities of the dataset occur in similar patterns that represent the reference groups. Groups are not disjointed but they have some common entities just like in real life.
Observing the patterns extracted it can be observed how groups change during the time. We can see that we do not have large changes. This is noted in the output of ListMiner because there are patterns, that partially overlap, that are similar. These similar patterns represent the interactions between individuals of the same group. There are also some groups that disappear especially in Reality dataset. There could be different reasons that justify this phenomenon, for example, they might represent that leave their groups when complete their university career.
Another interesting result shows that people have daily and weekly interactions. Especially in Enron, weekly emails seem to be particularly popular in a corporate enviroment. This can be observed in figure 6.24 observing the high number of patterns for the period equal to 7.
The last observation is that, in Reality and Enron datasets, there are some patterns that are cliques. In Enron dataset there are also some hierarchical interactions. This is somewhat expected because in large companies individuals have hierarchical structures. On the contrary, in Reality dataset the interactions between students are typically peer-to-peer and so we do not have hierarchical patterns.

8 Conclusion

The main contribution of this thesis is the design and development of *ListMiner,* an efficient algorithm for solving the *Periodic subgraph mining problem* in dynamic networks. This problem was introduced by Lahiri and Berger-Wolf [1][2] to discover frequent periodic interactions among the members of a population whose behavior was observed over time. Lahiri and Berger-Wolf also proposed and developed an algorithm for this problem, called *PSEMiner*. The time complexity of ListMiner is $O((V+E)\,T^2 \ln(T/\sigma))$, where V is the size of the population under analysis, E is the set of interactions among its members, T is the number of observations (timesteps) and σ is the minimum number of periodic repetitions that a subgraph must show to be reported in output. Listminer improves the worst case time complexity of PSEMiner by a factor T.
There are several variants of the periodic subgraph problem that can be studied. Most of these variants belong to the P complexity class. However, when *jitter* is allowed, this is no longer true. In fact, the problem of mining closed periodic subgraphs at minimum support σ, allowing jitter, was proved to be intractable because the number of patterns is exponential in the number of timesteps in the worst case.
Another contribution of this thesis is an algorithm to solve a slightly different problem in which the periodicity is defined as in the *string* contexts. This definition is stronger than the one in [1][2] (which in this framework is actually defined as *cadency*). Since real-world networks are unlikely to be "fully" periodic, some limited disruption in the composition of instances of the mined subgraphs was allowed. Experiments showed that the algorithm was capable of extracting meaningful patterns from real world networks.
Theoretical analysis of the proposed solutions was followed and supported by experimental validation. The performances and the behavior of ListMiner and PSEMiner were compared using two real-world dynamic social networks and several artificial datasets. The experiments showed that the performances of the algorithms are somewhat affected by the composition of the input dataset. Precisely, in datasets with high density of patterns, PSEMiner is much slower than ListMiner. Contrarily in a low-density context PSEMiner is faster than ListMiner. However, experiments on a worst case dataset confirm that ListMiner is actually more efficient than PSEMiner in this case, as expected from the theoretical analysis.
Moreover, in real scenarios, where the maximum period P_{max} is restricted, ListMiner took few seconds to execute and uses less than 15 MB of memory. ListMiner efficiently mines all periodic patterns, and it is a concrete alternative to PSEMiner for frequent subgraph mining.
Finally, a qualitative analysis of the mined patterns was done to understand the periodicities of the interactions between college students

and corporate executives. The experiments show the daily and weekly behavior of interactions among people. Moreover, the mined patterns reflected the characteristics of the interactions between the elements of the population under analysis. In particular, the patterns characterizing interactions between college students showed a peer-to-peer trend, while and those of between corporate individuals were mostly hierarchical.
Studying the Periodic subgraph mining problem, along with some of its variants that were the subject of this thesis, can be seen as initial steps into the uncovering of interesting relationships in dynamic network analysis. In fact, there is a number of interesting directions that can be the subject of future research. Among these, for example, particularly fascinating appears to be the introduction of the concept of noise to discover noisy subgraphs and the introduction of a probabilistic background model to assign a degree of *surprise* to the occurrence of each candidate pattern in a dynamic network.

9 *Acknowledgements*

I am very grateful to my supervisors Alberto Apostolico and Cinzia Pizzi for their guidance, especially Dr. Cinzia Pizzi for the help to write this document, and encouragement throughout this thesis.
I am thankful to Tania Berger-Wolf and Mayank Lahiri for sharing their datasets and their software PSEMiner.
Finally I would to express my grateful to my family, to my girlfriend Giulia and to all my friends for their support.

10 References

[1] M. Lahiri and T.Y. Berger-Wolf. Periodic subgraph mining in dynamic networks. Knowledge and Information Systems, Volume 24, Issue 3 (2010), p. 467.
[2] M. Lahiri and T.Y. Berger-Wolf. Mining Periodic Behavior in Dynamic Social Networks. Proc. of the 8th IEEE International Conference on Data Mining (ICDM 2008), Pisa, Italy. December 2008.
[3] K.-Y. Huang and C.-H. Chang: Mining Periodic Patterns in Sequence Data, In the Proceedings of the 6th International Conference on Data Warehousing and Knowledge Discovery (DaWaK04), Zaragoza, Spain, 2004. LNCS 3181 (SCI expanded), pp. 401-410.
[4] Minghua Zhang, Ben Kao, David W. Cheung, Kevin Y. Yip. Mining periodic patterns with gap requirement from sequences. ACM Transactions on Knowledge Discovery from Data (TKDD) Volume 1 , 2007. ISSN:1556-4681.
[5] David Lo, Siau-Cheng Khoo and Chao Liu. Efficient Mining of Iterative Patterns for Software Specification Discovery. In proceedings of the 13th SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD'07). San Jose, California. Aug 12-15, 2007.
[6] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In Proc. of the 20th Intl. Conf. on Very Large Data Bases, pg. 487–499, 1994.
[7] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In Proceedings of the 14th International Conference on Data Engineering, (ICDE'98), pages 412-421, 1998.
[8] J. Han, G. Dong, and Y. Yin. Efficient mining partial periodic patterns in time series database. In Proceedings of the 15th International Conference on Data Engineering, (ICDE'99), pages 106-115, 1999.
[9] C. Bettini, X. S.Wang, S. Jajodia, and Jia-Ling Lin. Discovering frequent event patterns with multiple granularities in time sequences. IEEE Transaction on Knowledge and Data Engineering, 10(2), 222-237, 1998.
[10] J. Yang, W. Wang, and P.S. Yu. Mining asynchronous periodic patterns in time series data. IEEE Transaction on Knowledge and Data Engineering, 15(3):613-628, 2003.
[11] K.Y. Huang and C.H. Chang. Asynchronous periodic patterns mining in temporal databases. In Proceedings of the IASTED International Conference on Databases and Applications (DBA'04), pages 43-48, 2004.
[12] Huang K-Y, Chang C-H (2005) SMCA: a general model for mining asynchronous periodic patterns in temporal databases. IEEE Trans Knowledge Data Eng 17(6):774–785
[13] I. R. Fischhoff, S. R. Sundaresan, J. Cordingley, H. M. Larkin, M.-J. Sellier, and D. I. Rubenstein. Social relationships and reproductive state influence leadership roles in movements of plains zebra, Equus burchellii. Animal Behaviour, 73(5):825–831, May 2007.

[14] Jiong Yang, Wei Wang, Philip S. Yu: Infominer: mining surprising periodic patterns. KDD 2001: 395-400.
[15] Inokuchi A, Washio T, Motoda H (2000) An apriori-based algorithm for mining frequent substructures from graph data. In: Proceedings of the 4th European conference on principles of data mining and knowledge discovery, pp 13–23.
[16] Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: Proceedings of the 2001 IEEE international conference on data mining, pp 313–320.
[17] Boros E, Gurvich V, Khachiyan L, Makino K (2002) On the complexity of generating maximal frequent and minimal infrequent sets. In: Proceedings of the 19th annual symposium on theoretical aspects of Computer Science (London, UK, 2002). Springer-Verlag, pp 133–141.
[18] Yang G (2004) The complexity of mining maximal frequent itemsets and maximal frequent patterns. In: Proceedings of the tenth ACM SIGKDD international conference on knowledge discovery and data mining (New York, NY, 2004). ACM, pp 344–353
[19] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet. ACM SIGPLAN Notices, 37(10):96–107, 2002.
[20] Juang P, Oki H, Wang Y, Martonosi M, Peh LS, Rubenstein DI (2002) Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with ZebraNet. ACM SIGPLAN Notices 37(10):96–107.
[21] Wasserman S, Faust K (1994) Social network analysis: methods and applications. Cambridge University Press, Cambridge.
[22] Faloutsos M, Faloutsos P, Faloutsos C (1999) On power-law relationships of the internet topology. In: Proceedings of the conference on applications, technologies, architectures, and protocols for computer communication (New York, NY, 1999). ACM, pp 251–262.
[23] Chapanond A, KrishnamoorthyMS, Yener B (2005) Graph theoretic and spectral analysis of Enron email data. Comput Math Organ Theory 11(3):265–281.
[24] Diesner J, Carley KM (2005) Exploration of communication networks from the Enron Email corpus. In: Proceedings of the 2005 SIAM workshop on link analysis, counterterrorism and security, pp 3–14.
[25] Nanavati AA, Gurumurthy S, Das G, Chakraborty D, Dasgupta K, Mukherjea S, Joshi A (2006) On the structural properties of massive telecom call graphs: findings and implications. In: Proceedings of the 15th ACM international conference on Information and knowledge management (New York, NY, USA,2006). ACM, pp 435–444.
[26] 18. Han J, Cheng H, Xin D, Yan X (2007) Frequent pattern mining: current status and future directions. Data Min Knowl Discov 15(1):55–86.

[27] Dickinson PJ, Bunke H, Dadej A, Kraetzl M (2003) On graphs with unique node labels, vol 2726 of Lecture Notes in Computer Science. Springer, Berlin, pp 409–437.
[28] Elfeky MG, ArefWG, Elmagarmid AK (2005) Periodicity detection in time series databases. IEEE Trans Knowl Data Eng 17(7):875–887.
[29] Ma S, Hellerstein JL (2001) Mining partially periodic event patterns with unknown periods. In: Proceedings of the 17th international conference on data engineering (Washington, DC, USA, 2001). IEEE Computer Society, pp 205–214.
[30] A. Apostolico and M. Crochemore String Pattern Matching for a Deluge Survival Kit, Handbook of Massive Data Sets, J. Abello et al, Eds. Kluver Acad. Publishers, Kluver Acad. Publishers, pp. 151--194 2002.
[31] Donald E. Knuth, James H. Morris and Vaughan R. Pratt, Fast Pattern Matching in Strings, SIAM Journal on Computing, 6(2):323-350, 1977.
[32] Thompson, William; Joseph Hickey (2005). Society in Focus. Boston, MA: Pearson. ISBN 0-205-41365-X.
[33] William J. Frawley, Gregory Piatetsky-Shapiro, Christopher J. Matheus: Knowledge Discovery in Databases: An Overview. AI Magazine 13(3): 57-70 1992.
[34] J. Cook and L. Holder. Substructure discovery using minimum description length and background knowledge. J. Artificial Intel. Research, 1:231-255, 1994.
[35] K. Yoshida, H. Motoda, and N. Indurkhya. Graph based induction as a unified learning framework. J. of Applied Intel., 4:297-328, 1994.
[36] L. Dehaspe and H. Toivonen. Discovery of frequent datalog patterns. Data Mining and Knowledge Discovery, 3(1):7-36, 1999.
[37] L. De Raedt and S. Kramer. The levelwise version space algorithm and its application to molecular fragment finding. In IJCAI'01: Seventeenth International Joint Conference on Artificial Intelligence, volume 2, pages853-859, 2001.
[38] Takashi Washio and Hiroshi Motoda. State of the Art of Graph-based Data Mining, ACM SIGKDD Explorations Newsletter pages: 59–68, 2003.
[39] S. Muggleton and L. De Raedt. Inductive logic programming: Theory and methods. J. Logic Programming, 19(20):629-679, 1994.
[40] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. Communications of the ACM, 39(11):58-64, 1996.
[41] V. Vapnik. The Nature of Statistical Learning Theory. Springer Verlag, New York, 1995.