CSPM: Discovering Compressing Stars in Attributed Graphs

Jiahong Liu[a], Philippe Fournier-Viger[b,c,*], Min Zhou[d], Ganghuan He[a], Mourad Nouioua[c]

[a]*School of Computer Sciences and Technology, Harbin Institute of Technology (Shenzhen), Shenzhen, Guangdong, China, 518055*
[b]*College of Computer Science and Software Engineering, Shenzhen University, Shenzhen, Guangdong, China, 518060*
[c]*School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen, Guangdong, China, 518055*
[d]*Noah's Ark Lab, Huawei, Shenzhen, China, 518055*

Abstract

Graphs, also known as networks, are an expressive data representation used in many domains. Numerous algorithms have been designed to find interesting patterns in graphs. However, they have at least one of two major drawbacks. First, most algorithms focus on finding complex subgraphs but ignore relationships between attributes. But attributes play a major role in several real-life applications such as for profile completion in social networks. Second, the user must generally set multiple parameters that greatly influence results but are unintuitive to set. To provide a solution to these issues, this paper introduces a novel algorithm named CSPM (Compressing Star Pattern Miner) for discovering compressing patterns in an attributed graph. The algorithm is parameter-free and discovers star-shaped attribute patterns that reveal strong relationships between attribute values by utilizing the concept of conditional entropy and the minimum description length principle. Extensive experiments on real data show that CSPM is efficient and can find insightful patterns. In particular, it is observed that the discovered patterns can boost the accuracy of state-of-the-art graph attribute completion models on social network data by up to 15.39%. Moreover, it is found that CSPM can uncover many interesting patterns describing alarm correlations in data from a large-scale industrial telecommunication network.

Keywords: graph, network, attributed graph, pattern mining, compressing patterns, stars, graph completion

1. Introduction

To discover interesting knowledge in graphs, several graph pattern mining algorithms have been developed. They can find informative patterns (subgraphs) that match some constraints specified by users. For example, a popular data mining task is frequent subgraph mining, which consists of finding all connected subgraphs that appear at least some minimum number of times in graph(s) [19, 31]. Graph patterns are useful as they can give insights to the user about the structure of graphs in terms of the connections between edges and vertices. Moreover, graph patterns can also be used to support decision making and tasks such as clustering and classification [12, 19]. While some algorithms can identify subgraphs without any restrictions on the number of edges and vertices and how they are connected, discovering complex subgraphs has a very high time complexity and is often unnecessary for users. For these reasons, algorithms have also been developed to find more simple types of subgraphs such as stars, trees and paths [12, 26].

Subgraph pattern mining algorithms have two major limitations. First, most algorithms can only process graphs having nodes described using a single attribute [12, 19]. This means that a node can be annotated with at most one label (also called attribute value). For instance, a social network graph could only describe persons (vertices) using a single attribute such as gender that can take an attribute value such as male or female. Considering a single attribute value per vertex reduces the complexity of subgraph pattern mining but results in missing interesting information about how multiple attribute values (labels) are correlated. For many real-world graphs, several attributes should be jointly analyzed as they play a key role in understanding a graph's structure. For example, in social network analysis,

*Corresponding author
Email addresses: jiahong.liu21@gmail.com (Jiahong Liu), philfv@szu.edu.cn (Philippe Fournier-Viger), zhoumin27@huawei.com (Min Zhou), heganghuan@gmail.com (Ganghuan He), mouradnouioua@gmail.com (Mourad Nouioua)

it is known that multiple attributes (e.g. age, gender, city) influence how users are connected [11]. Discovering the relationships between attribute values of social network users can help understanding their preferences and doing tasks such as inferring missing profile information [4, 11].

Second, graph pattern mining algorithms typically require to set multiple parameters to select patterns. But finding appropriate values for these algorithm parameters is time-consuming and often unintuitive. For instance, to apply a frequent subgraph mining algorithm, the user must choose a value for a parameter named the minimum support (*minsup*) threshold. Then, the algorithm returns to the user all subgraphs appearing at least *minsup* times. But choosing a suitable *minsup* value is hard for the user as it depends on dataset characteristics that are initially unknown to the user [19]. Hence, a user will typically run an algorithm multiple times with different parameter values to find enough but not too many patterns. If the constraints defined by the algorithm parameters are too strict, many useful patterns are missed, while if they are too loose, a huge amount of spurious patterns may be found, and combing through these patterns becomes very time-confusing for the user [14]. There is thus a need to reduce the number of algorithm parameters, while ensuring that interesting or useful patterns are discovered [14].

Recently, some researchers have done some steps towards solving the first problem, as algorithms have been proposed for mining patterns in **attributed graphs** [1, 5, 13, 21, 33]. In such graphs, each vertex may be described using multiple attributes that are generally assumed to be categorical. Though, these algorithms were shown to be useful for some applications [5, 13], they still suffer from the problem of algorithm parameter selection.

The *research problem* of this paper is to address the two aforementioned limitations of current graph pattern algorithms. The *objective* is to design an algorithm that can identify patterns revealing complex relationships between multiple attribute values in an attributed graph and that is parameter-free.

Attaining this objective requires to address three challenges: (1) designing a pattern format that is meaningful and practical for some applications, (2) devising an algorithm that can efficiently find these patterns, and (3) showing through experiments that this new pattern format can reveal interesting patterns in real data, and be useful for some downstream tasks such as graph completion.

To tackle these challenges, this paper draws inspiration from an emerging approach in the field of pattern mining, which is to find **compressing patterns** [2, 7, 27, 38, 41]. The key idea in compressing pattern mining is to try to identify the set of patterns that provides the best compression of a database according to the MDL (Minimum Description Length) principle. The rationale is that patterns that compress well a database are also representative of the key features of that database. Algorithms for mining compressing patterns have been defined for various pattern mining tasks such as mining patterns in transactions [38, 41] and sequences [27]. And it was shown that compressing patterns can be more useful than traditional frequent patterns for several tasks such as classification [41], outlier detection [27, 38] and event prediction [7]. However, to avoid the problem of parameter selection, it is desirable to design an algorithm that is truly parameter-free for mining compressing patterns [38].

The contribution of this paper is a novel and truly parameter-free algorithm named CSPM (Compressing Star Pattern Miner) for discovering compressing patterns in an attributed graph. Rather than finding attribute-patterns with complex structures, CSPM identifies simple star-shaped patterns called *attribute-stars* (a-stars) which are easily understandable, have practical applications, and can reveal strong relationships between one or more attribute values of connected nodes. An attribute-star indicates that attribute values of a vertex (called core) are correlated to those of some connected nodes (called leaves). An attribute-star is general in the sense that it can match with multiple vertices of a graph. The interpretation of an attribute-star is that if the core attribute values are observed for a vertex there is a high chance that connected vertices will have the attribute values of its leaves. To illustrate the concept of attribute-star, Fig. 1 (a) depicts part of a music social network where attribute values indicate musical preferences and Fig. 1 (b) shows an attribute-star found in that social network. This star indicates that the friends of a user that prefers pop music tend to like rock and jazz music. This a-star pattern appears several times in the graph of Fig. 1 (a).

As it will be described in the experimental evaluation section of this paper, such attribute-stars are useful to understand the graph and can also boost the performance of state-of-the-art graph attribute completion models. For social networks, it means that attribute-stars can be used to infer the profiles of users based on their relationships with other people with more accuracy. Inferring user profiles is an important task in social network analysis to provide targeted advertisement [11]. Moreover, it will be also shown that the proposed attribute-stars found in data from a large-scale industrial telecommunication network revealed interesting alarm correlations that can be used for network fault management.

To obtain the proposed a-star patterns, the designed CSPM algorithm applies a greedy-search to quickly find an

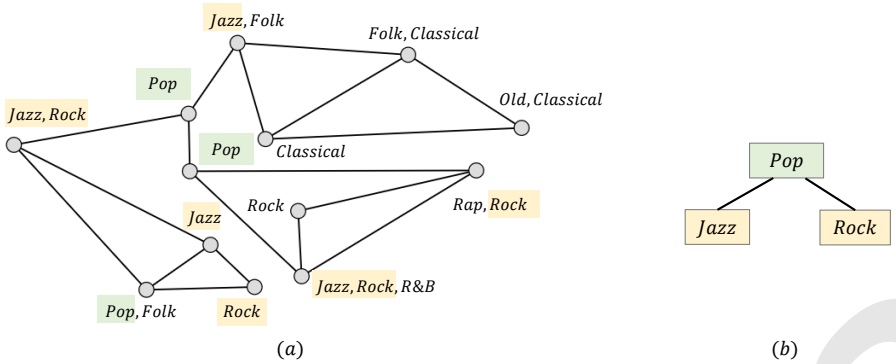


Figure 1: An (a) attributed graph and (b) an attributed-star

approximation of the best set of patterns that maximizes compression according to the MDL principle. CSPM also relies on the concept of conditional entropy to assess how strong the relationships between attribute values are.

The rest of this paper is organized as follows. Section 2 reviews related work. Then, preliminaries are described in Section 3. Thereafter, the designed CSPM algorithm is introduced in Section 4. Then, variants of CSPM are presented for different application scenarios in Section 5. Experiments are then described in Section 6 and a conclusion is drawn in Section 7.

2. Related Work

This section surveys relevant research studies on (1) discovering patterns in graphs, (2) finding compressing patterns in various types of data, and (3) techniques for compressing or summarizing graphs.

2.1. Mining patterns in graphs

Many pattern mining algorithms have been designed to find interesting patterns in graphs. The aim of these algorithms is to find patterns that meet constraints, defined by the user. Frequent subgraph mining (FSM) [19] is one of the key tasks, which aims at enumerating all connected subgraphs that have a number of occurrences (support) that exceeds or is equal to some user-specified minimum support threshold. FSM can be applied to find subgraphs that are common to several graphs or that appear many times in a single graph. The search space of FSM is huge because connected subgraphs can have a complex topological structure with multiple edges and vertices. Several algorithms have been designed for FSM [14]. A typical FSM algorithm will traverse the search space of subgraphs and may have to evaluate a very large number of subgraphs to find the frequent ones [31, 43]. Besides, another reason for the high time complexity of FSM is the cost of subgraph isomorphism checking, that is to verify if a subgraph is equivalent to another subgraph that was previously checked to avoid redundant work [14, 19]. FSM has many applications and variations [12, 19]. Nevertheless, discovering frequent subgraphs having complex topological structures is sometimes unnecessary. For this reason, more simple subgraph types have also been studied such as frequent trees [20] and frequent paths [32]. But a key limitation of traditional frequent subgraph mining algorithms is that each vertex is not allowed to have more than one label. This is a major problem for applications such as social network analysis where users (vertices) are typically described using multiple labels (attribute values) such as as their age, gender and location. To address this problem, some FSM algorithms were proposed to mine subgraph patterns in attributed graphs (graphs having multiple labels per node) [21], frequent trees in databases of attributed trees [33], and temporal patterns in dynamic attributed graphs [12, 13, 15].

Although much work has been done on discovering subgraphs, traditional techniques often require setting multiple algorithm parameters and finding appropriate values for these parameters is often unintuitive [14]. If constraints for selecting patterns are too strict, many important patterns may be missed, while if constraints are too loose, millions of spurious patterns may be found and algorithms may even fail to terminate or run out of memory. As a consequence users often try to find appropriate parameter values by running graph pattern mining algorithms numerous times using a trial and error process, which is time-consuming [14].

Another issue with traditional subgraph mining algorithms is that frequent subgraphs may contain graph elements (vertices, edges or labels) that appear frequently together but are weakly correlated (may appear together by chance) [13, 34]. For example, a subgraph appearing frequently in a social network may indicate that a smoker is friend with persons who likes pop music but this pattern may simply be frequent because many people like pop music and many people are smoking. Thus, it is desirable to not only consider frequency as a criterion to select patterns but also some measure of correlation to filter spurious patterns [13, 15, 34].

2.2. Mining compressing patterns

An emerging research direction in pattern mining is to find *compressing patterns*. The aim is to identify a small set of patterns that describe well the content of a database. The concept of compressing patterns was first proposed in the Krimp [41] algorithm for mining itemsets (sets of values) in a binary table (also called a transaction database). Krimp takes frequent itemsets as input and relies on the MDL principle to evaluate different subsets of patterns and select the subset that provides the best compression. The MDL principle states that some data D is best encoded using the model M that has the smallest Description Length (DL). In this context, a model M is a set of patterns and the data D is the input database. A simple way of calculating the DL of a model M for some data D is as $L(M, D) = L(M) + L(D|M)$ where $L(M)$ is the length (size) of M and $L(D|M)$ is the length of the data D when encoded using M.

The motivation for using the MDL is that the model having the smallest MDL contains patterns representing important aspects of the data. To compress a database using a model M, a unique code is assigned to each pattern, and the occurrence of each pattern is replaced by that code. For easy lookup, all the codes are stored in a structure called the *code table*. Since there is generally a huge number of possible models (sets of patterns), finding the best one is often impractical. Indeed, it is not unusual that real datasets contain millions of patterns, and if there are x patterns, an exhaustive search would require to evaluate $2^x - 1$ models to find the best one. Hence, Krimp utilizes an approximate algorithm to find a good model. To do this, the user must first provide a set of candidate patterns extracted using a traditional frequent itemset mining algorithms. Then, Krimp recursively applies a greedy search to test different subsets of candidate patterns. Krimp returns the set M that has the smallest DL, i.e. $L(M, D)$. This is done iteratively by starting from an empty model M, and repeatedly adding to the model the candidate pattern that provides the greatest reduction of the DL. The search ends, when Krimp cannot add more patterns to further reduce the DL [41]. Krimp is an approximate algorithm but typically can find good models using its greedy search. However, a drawback of Krimp is that the database format (a binary table) is too simple for many applications.

To extend this approach to more complex data types, algorithms inspired by Krimp were designed. For sequential data, the DITTO [3] and SeqKrimp [27] algorithms allow extracting compressing patterns from a discrete sequence and a set of sequences, respectively. For graph data, the GraphMDL algorithm [2] mines a set of compressing subgraphs in labelled graphs where vertices are described using at most one label per vertex. GraphMDL first extracts frequent subgraphs from input graphs using a traditional FSM algorithm [19]. Then, GraphMDL applies a greedy search inspired by Krimp to find a set of subgraphs that compresses well the data in terms of the minimum description length. GraphMDL can reveal interesting patterns but has two important drawbacks. First, GraphMDL can only handle graphs where each vertex is described using a single attribute, that is a vertex cannot have more than one label (attribute value). Hence, GraphMDL cannot be used on attributed graphs.Second, GraphMDL relies on a traditional pattern mining algorithm to find the initial set of candidate patterns (similarly to Krimp). Thus, to run GraphMDL, the user must set a value for a parameter called the minimum support threshold. Because of this, GraphMDL is not parameter-free, and results may be of poor quality while runtime and memory usage can dramatically increase if the user does not set the parameter to an appropriate value.

This work is different from the above studies in three ways:

1. This paper aims at discovering compressing patterns in an attributed graph rather than in a graph described using a single attribute.
2. A different type of patterns is discovered. While many studies focus on finding patterns with complex topological structure, this paper suggests mining a novel pattern type called attributed stars. This pattern type is simple in terms of topological structure but can reveal strong correlations between multiple attribute values of a vertex and its neighbors. As it will be shown in the experiments, this pattern type is useful for tasks such as user profile completion on social networks and analyzing telecommunication network alarm data.

3. The proposed algorithm is entirely parameter-free and does not depend on traditional FSM algorithms for mining an initial set of candidate patterns. This is achieved by drawing inspiration from an improved version of Krimp, named SLIM [38] that relies on an iterative procedure to identify a good set of compressing patterns. Using this approach, the proposed algorithm generates candidate patterns on-the-fly rather than first mining a fixed set of candidate patterns (as Krimp, GraphMDL and several other algorithms do).

2.3. Summarizing and compressing a graph

Another research area that is related to this work is algorithms for data compression and summarization. Graph compression algorithms reduce the space for storing very large graphs, which can also speed-up their processing. Graph summarization algorithms have a similar aim but are generally lossy and provide a summary that may also help users understand key features of the graph.

A representative graph compression algorithm is SlashBurn [29]. It first finds hubs (nodes connected to many edges) that if removed could result in obtaining many spokes (large connected subgraphs). Then, SlashBurn obtains a high compression by reordering and transforming the graph's adjacency matrix representation based on the hubs and spokes. Another graph compression algorithm is VOG [26]. VOG first applies a graph decomposition technique (e.g. SlashBurn) or a community detection algorithm to divide a large graph into many subgraphs. Then VOG relies on the MDL principle to approximate the topology of each subgraph using a vocabulary consisting of six pattern types (stars, near or full bipartite cores, chains, and near or full cliques). For each subgraph, the model that has the smallest DL is selected. VOG is parameter-free but only focuses on compressing a graph's topology. It ignores graph attributes.

A few studies have been done on summarizing and compressing attributed graphs based on their attributes and topology. The Greedy-Merge algorithm [30] finds nodes having the same attribute values, and then links them using virtual edges. Thereafter, Greedy-Merge applies a clustering algorithm to find groups of vertices that have similar attribute values and topological structures. The QB-ULSH [23] algorithm compresses an attributed graph by creating super-nodes to replace vertices that have similar edges based on the MDL principle. But a major limitation of QB-ULSH is that many parameters must be set by the user, which greatly influence its output.

The purpose of the algorithms reviewed in this subsection is to compress or summarize a graph. This is different from the CSPM algorithm presented in this paper, where compression is not the end goal but is a mean of finding a good set of patterns indicating strong relationships between attribute values.

3. Preliminaries

This section introduces important preliminaries about attributed graphs and describes the basic framework for discovering compressing patterns.

3.1. Graphs, Attributed graphs and Stars

Key concepts about graphs, used in this paper, are the following:

Definition 1 (Graph). *A* graph $G = (V, E)$ *consists of two sets: a vertex set V and an edge set E. The set of vertices contains elements called* vertices *and must be non-empty ($V \neq \emptyset$). The set of edges consists of zero or many* edges*. An edge is a tuple of the form (u, v) where u and v are vertices ($u, v \in V$). In other words, $E \subseteq V \times V$.*

Definition 2 (Connected graph). *Let there be a* graph $G = (V, E)$*. If there exists an edge $(u, v) \in E$, then it is said that vertex u* is adjacent to *vertex v, or simply that u* can reach *v. Two edges e_1 and e_2 are adjacent if they have a vertex in common. A graph G is a* connected graph *if for any pairs of vertices u, v, there is a* path *(a sequence of adjacent vertices) such that u can be reached from v.*

The concept of graph can be generalized as that of attributed graph to be able to describe properties of vertices.

Definition 3 (Attributed graph). *An* attributed graph *is a tuple $G = (A, \lambda, V, E, F)$ consisting of a finite and non-empty vertex set V, a finite edge set E, a finite attribute set A, a finite attribute value set F, and a function $\lambda : A \times V \mapsto F$ that maps pairs of attributes and vertices to attribute values.*

Example 1. *Fig. 2 (a) depicts an attributed graph that will be used as running example. The vertex set is* $V = \{v_1, v_2, v_3, v_4, v_5\}$ *and the edge set is* $E = \{(v_1, v_2), (v_1, v_3), (v_1, v_4), (v_3, v_5), (v_4, v_5)\}$. *The set of attributes is* $A = \{\alpha, \beta\}$. *The set of attribute values is* $F = \{a, b, c\}$. *The function* λ *assigns attribute values to vertices, and is defined as:* $\lambda = \{(\alpha, v_1, a), (\alpha, v_2, a), (\alpha, v_5, a), (\beta, v_2, c), (\beta, v_3, c), (\beta, v_4, b), (\beta, v_5, b)\}$. *The vertex* v_1 *has attribute value a for the attribute* α, *while the vertex* v_5 *has values a and b for the attributes* α *and* β, *respectively. Such graph may be used to represent data from various domains. For instance, that graph may represent a social network where vertices are persons and edges are friendship links. The attribute* α *may indicate if a person is known to be an adult with the value a, while the attribute* β *may indicate the gender as either male (value b), female (value c), or unknown (no value).*

In the rest of this paper, it will be considered that an attributed graph cannot contain an edge (u, v) such that $u = v$. In other words, *self-loops* are not allowed. This is a reasonable assumption to represent data such as a social graph as a person cannot be friend with himself.

With respect to attributes, it should be noted that an attributed graph can have both numerical and categorical attributes. The algorithm designed in this paper treats all attribute values as nominal [5, 13]. To handle numerical attributes, numbers such as the age of a person could be treated as nominal values or discretized [13].

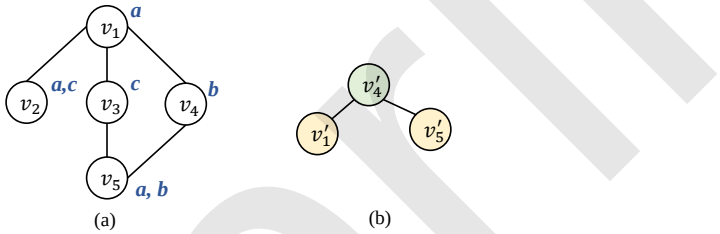


Figure 2: An example of (a) attributed graph and (b) a star pattern

A convenient way of representing a graph to process it is using a *vertex adjacency list*. The vertex adjacency list of an (attributed) graph is a list of tuples, where each tuple contains a vertex followed by the set of its adjacent vertices.

Example 2. *For the graph of Fig. 2 (a), the adjacency list is* $\{(v_1, \{v_2, v_3, v_4\}), (v_2, \{v_1\}), (v_3, \{v_1, v_5\}), (v_4, \{v_1, v_5\}), (v_5, \{v_3, v_4\})\}$.

Various algorithms have been designed to identify many different types of patterns in graphs such as bipartite cores, trees, subgraphs, clique, paths, and stars [12, 19, 26]. While finding patterns with complex topological structures can be useful for some applications, patterns having simple topological structures such as stars and chains have many applications and have the benefit of being easy to interpret. Thus, it was decided for this study to focus on discovering patterns that have a simple topological structure but can reveal complex relationships between attribute values. The type of patterns that is proposed in this paper, called attributed-star, is inspired by the concept of star pattern [26] but redefined to handle multiple attributes in an attributed graph.

Definition 4 (Star). *A* star *X is a graph, represented by a tuple* $X = (V, E, L, c)$, *where V is a set of vertices and E is a set of undirected edges. Among the vertices, a single vertex* $c \in V$ *is called the core, while the remaining vertices* $L = V - \{c\}$ *are called the leaves. Furthermore, there is an edge between each leaf and the core, i.e.* $E = \{(c, w) | w \in V \wedge w \neq c\}$.

Example 3. *Consider the graph of Fig. 2. A star pattern found in this graph is depicted in Fig. 2(b), which contains three vertices* $V = \{v'_1, v'_4, v'_5\}$, *two edges* $E = \{(v'_1, v'_4), (v'_4, v'_5)\}$, *a core* $c = v'_4$ *and two leaves* $L = \{v'_1, v'_5\}$.

Star patterns are useful in many applications. For example, in social network analysis, it is well-known that a person (core) often influences his friends (leaves) [11]. Such relationships are naturally captured by stars. This application as well as another one for telecommunication network alarm analysis will be discussed in the experimental evaluation section.

3.2. The Basic Framework for Mining Compressing Patterns

The basic framework for mining compressing patterns in a database using the MDL is described next. The main steps were introduced in the Krimp algorithm [41] to select the best model M for compressing a database. Compression is quantified by the total description length, that is the sum of a model length and the length of the data compressed using that model. A shorter description length is better.

Compressing patterns can be discovered in various types of data. Generally, a model M is a set of patterns that follow a given format defined for some applications. For example, in the Krimp algorithm, a pattern is an itemset (a set of values) that appears in a transaction database (a binary table) [41], while in the SeqKrimp algorithm, a pattern is a sequential pattern (a sequence of values) that appears in discrete sequences [27].

To utilize a model M for compressing a database, a unique code is given to each pattern of that model. Then, the compressed database is obtained by replacing each pattern occurrence by the pattern's code. To maximize the compression obtained using a model, it is important to define an appropriate strategy to assign codes to patterns. In particular, it is desirable to give smaller codes to patterns that have a higher frequency. Besides, another challenge for mining compressing patterns is the very large number of possible models. It is thus necessary to adopt an appropriate strategy for exploring that search space to find a good model quickly.

This is explained in more details by taking the Krimp algorithm as example. The length of a code for a pattern (an itemset) X is the minimum number of bits that can be used to encode the pattern without information loss. The optimal code length of a pattern is calculated using the Shannon Entropy [28] as $L(X) = -log_2 p(X)$, where $p(X)$ is the relative occurrence frequency of the pattern X in the data.

To search for models efficiently, Krimp builds a structure named the *Standard Code Table* (ST). That table contains an entry (row) for each item (value) that indicates its code length. The Standard Code Table is then used to calculate the length of the original database without compression, denoted as $L(D|ST)$. This is done by replacing in the database each item occurrence by its code in ST and then adding the database length to that of ST. Then, the possible patterns (itemsets) are obtained by a traditional frequent itemset mining algorithm. Then, based on these itemsets, Krimp starts to search for a better model M (a better set of patterns). This is done using an iterative process where M is initialized as the empty set. Then, during each iteration, Krimp adds to M the pattern that is not already in M and that increases the most the compression. Iterations stop when the compression cannot be improved by adding another pattern. Each model M is represented by a structure called a Code Table, denoted as CT. Each row of the code table stores a pattern as (1) the items (values) that it contains and (2) its code length. The total length of the database compressed using a code table CT is calculated as $L(CT, D) = L(CT|D) + L(D|CT)$, where $L(CT|D)$ is the length of the code table, and $L(D|CT)$ is the length of the database encoded using that code table. To find a good model (a code table), Krimp employs a greedy search.

For example, Fig. 3 (left) shows a database D consisting of eight records (transactions). Each record (row) is a set of items denoted by upper case letters (A, B and C). For instance, the first line of D indicates that items A, B and C appear in that record. From that database, the Standard Code-Table ST is created, which is shown in Fig. 3 (top). In the ST, a code is given to each item using the Shannon Entropy. In that figure, a code is represented as a colored rectangle, where a smaller rectangle indicates a shorter code. A shorter code is assigned to items that appear more frequently (e.g. A and B) than other items (e.g. C) in D. Then, a model CT is created based on the MDL principle. For instance, Fig. 3 (bottom, center) shows a model consisting of five patterns, stored in the CT. The CT contains the patterns, which are represented using codes from the ST. For example, the first row of the CT is the pattern A, B, C which is made of the codes of A, B and C in the ST. Furthermore, each pattern in the CT has a new code. These codes are then used to encode the original database so as to evaluate the compression obtained using the model CT. The result is shown in Fig. 3 (right). The description length obtained using the model CT is simply calculated as the sum of the lengths of the two columns in CT, and the length of the encoded database. As example, Fig. 4 (left) shows the length of the original database encoded with ST, and Fig. 4 (right) depicts the description length using the model CT. Krimp builds a model iteratively by adding patterns until no further compression is achieved. When the iterative process ends, the result is the set of compressing patterns stored in the *Itemset* column of CT.

Krimp has attracted much attention in recent years. Its iterative process to find a model is generally fast. However, a major limitation of Krimp is that to apply it, a user must first run a traditional frequent itemset mining algorithm to extract frequent itemsets. This step is very time-consuming and it requires that the user choose a value for a parameter called the minimum support threshold. Depending on how this parameter is set, the runtime and final results can

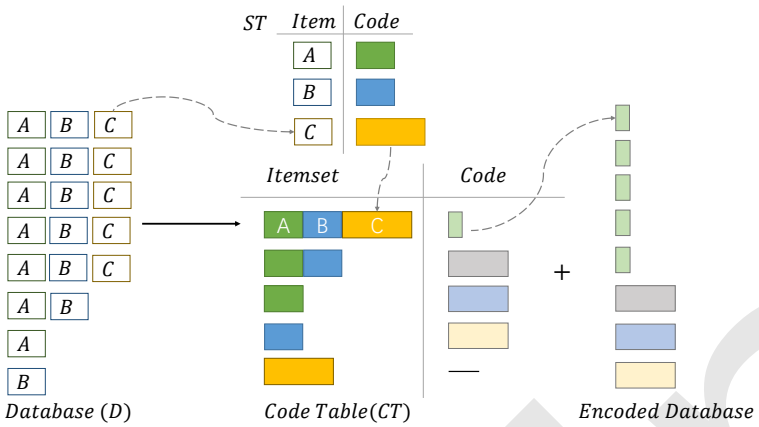


Figure 3: An example model of transaction database

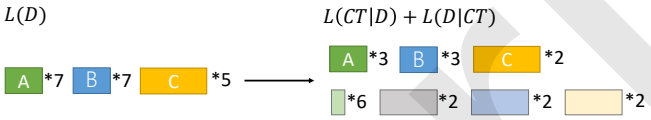


Figure 4: An example of description length calculation

greatly vary. To solve this problem, the SLIM algorithm [38] was proposed which is a modified version of Krimp that mines patterns on-the-fly during the iterative process. The key advantage of SLIM over Krimp is that the user does not need to set a threshold. SLIM mines the same type of compressing patterns as Krimp but is more efficient.

4. The CSPM Algorithm

This section presents the designed CSPM (Compressing Star Pattern Mining) algorithm for identifying patterns in an attributed graph that have a star shape and can reveal strong correlations between attribute values. To design such algorithm based on the MDL principle, there are four challenges to be addressed. First, it is necessary to select a pattern format that provides information about a core and its leaves. Second, a model must be designed to encode a database using these patterns. Third, a method must be put forward to calculate the description length using this model type. Fourth, a search strategy must be designed to quickly find the best model (or an approximation) without having to check all possible patterns and models.

This section provides solutions to these challenges. Subsection 4.1 describes the notation and pattern type. Subsection 4.2 presents the overall framework of CSPM. Then, subsection 4.3 introduces an optimized version of CSPM to increase its efficiency, called CSPM-Batch. Lastly, subsection 4.4 analyzes the algorithm's complexity.

4.1. Pattern Format and Notation

While there are many different pattern types, this study focuses on discovering star shaped patterns because they have a simple topological structure that is easy to understand and is meaningful for various applications. For instance, in social network analysis, relationships between a person and its close friends are known to be meaningful [11]. An interesting property of stars is also that they have a higher occurrence frequency than more complex subgraph structures such as chains, bipartite graphs and cliques [10]. In fact, the simple structure of stars can be used to represent any other types of subgraphs [10]. In other words, large and complex subgraph patterns can always be broken down into one or more meaningful stars. Besides, considering patterns having a simple topological structure allows focusing on discovering complex relationships between attribute values in an attributed graph.

Hence, the objective of this study is to find a novel type of star patterns, called attributed stars. As it will be shown in the experimental evaluation section, there are several applications of this new pattern type such as alarm correlation rule analysis [16] and user profile completion for social networks [4].

To extend the concept of *star* presented in Definition 4 with attribute values, the concept of *extended star* is defined.

Definition 5 (Extended star). *An* extended star *X is an attributed graph, represented as a tuple $X = (V, E, L, c, A, F, \lambda)$, such that (V, E, L, c) is a star, A is an attribute set, F is an attribute value set, and λ is a mapping $\lambda : A \times V \mapsto F$ that associates attribute values to pairs of attributes and vertices.*

Example 4. *An extended star is depicted in Fig. 5 (a), which is the star of Fig. 2 (a), where attribute values have been added. There are two attribute $A = \{\alpha, \beta\}$ that can take values from the set $F = \{a, b\}$. The mapping λ from vertices and attributes to attribute values is defined as: $\lambda = \{(\alpha, v_1', a), (\beta, v_4', b), (\beta, v_5', b)\}$. Thus, the vertex v_1' has attribute value $\{a\}$ for attribute α, while vertices v_4' and v_5' have attribute value $\{b\}$ for attribute β.*

To be able to count the number of occurrences of extended stars in an attributed graph, the concept of appearance is defined.

Definition 6 (Appearance of a star). *Let there be an attributed graph $G = (A_y, \lambda_y, V_y, E_y, F)$ and an extended star $X = (V_x, E_x, L, c, A_x, F, \lambda_x)$. X has an* appearance *in G if each vertex from X can be matched to a distinct vertex from G such that edges and attribute values of X are preserved by this mapping. A formal definition is given next. An appearance of X in G is a bijective mapping $f : V_x \rightarrow V_y$ that satisfies two conditions. Firstly, for each triple $(a, v, z) \in \lambda_x$, there exists a corresponding triple $(a, f(v), z) \in \lambda_y$ (attribute values from X are preserved by the mapping f). Second, for each edge $(u, v) \in E_x$, there is a corresponding edge $(f(u), f(v)) \in E_y$ (edges from X are preserved by the mapping f). Note that generally, an extended star can have zero, one or multiple appearances in an attributed graph.*

Example 5. *In the attributed graph of Fig. 2, there is an appearance of the extended star of Fig. 5 (a), which is defined by the mapping function $f = \{(v_1, v_1'), (v_4, v_4'), (v_5, v_5')\}$. This appearance satisfies the two conditions since that mapping preserves the attribute values and edges of the extended star. For instance, v_1 has the attribute value b and has an edge to a vertex v_4 just like v_1' has the attribute value b and an edge to a vertex v_4'.*

Extended stars can describe interesting relationships between vertices, edges and attribute values. But to be able to summarize multiple extended stars, a more general pattern type is proposed. It is called *attribute-star*, or shortly *a-star*. This pattern type is designed to focus more on discovering strong relationships between attribute values of a core and its leaves than on the exact topological structure of the star. The definition of a-star is given next, and then the relationship with extended stars is illustrated with an example.

Definition 7 (Attribute-star, a-star). *Let there be the set F of all possible attribute values. An attribute-star (a-star) S is a pair of sets of attribute values $S = (S_c, S_L)$, such that $S_c, S_L \subseteq F$. The set S_c contains the attribute values of a core node c, while the set S_L contains attribute values of leaf nodes (nodes directly connected to c). The sets S_c and S_L of an attribute-star are called the* coreset *and* leafset *of S. The attribute values in the coreset S_c are called the* core-values*, while those in the leafset S_L are called the* leaf-values*.*

Example 6. *Let there be a set of possible attribute values $F = \{a, b\}$. The a-star $S = (\{b\}, \{a, b\})$ indicates that a core has an attribute value $\{b\}$ and that some connected vertices (leaves) have attribute values $\{a\}$ and $\{b\}$. The coreset of that a-star is $\{b\}$. Thus, b is said to be a core-value. And the leafset is $\{a, b\}$, which contains two leaf-values that are a and b.*

In the following, an a-star that has a single core-value ($|S_c| = 1$) and a single leaf-value ($|S_L| = 1$) is called a *basic a-star*. The proposed CSPM algorithm uses the basic a-stars as building blocks to create larger patterns. CSPM starts to search for patterns from basic a-stars and combines them to generate a-stars having more attribute values.

The following definition explains the relationship between an attribute-star and a star:

Definition 8 (Relationship between an attribute-star and a star). *Let there be an extended star* $X = (V, E, L, c, A, F, \lambda)$ *and an attribute-star* $S = (S_c, S_L)$. *S is said to be* matching with *X, if two conditions are satisfied: (1) there is an attribute* $\alpha \in A$, $\forall z \in S_c$ *such that there is a triple* $(\alpha, c, z) \in \lambda$; *(2) there is a leaf* $u \in L$, $\forall y \in S_L$ *such that* $(\beta, u, y) \in \lambda$.

By the above definition, it can be seen that multiple stars may be summarized by a single a-star.

Example 7. *The a-star* $S = (\{b\}, \{a, b\})$ *presented in Fig. 5(c) matches with the extended stars depicted in Fig. 5(a) and (b) which are extracted from the graph illustrated in Fig. 2(a).*

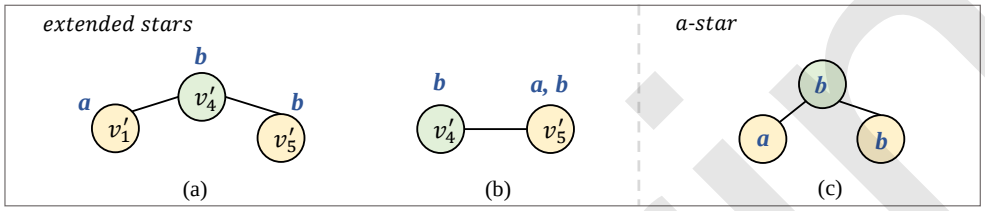


Figure 5: Two extended stars and an a-star

The proposed CSPM algorithm is designed to find a small set of a-stars in an attributed graph, such that each a-star indicates a strong correlation between a coreset and a leafset. For this purpose, a possibility is to search for a-stars where leafsets and coresets frequently co-occur. However, just looking for frequent co-occurrences of attribute values would not allow to understand the influence relationship between values in an a-star. As a solution, it was decided to view core-values and leaf-values as playing a different role. More precisely, the core-values of an a-star are considered as influencing the leaf-values. Hence, the interpretation of an a-star is that if core-values appear in a core vertex, there is a high chance that the leaf-values appear in its neighbor(s). This influence relationship between a coreset and a leafset is assessed using the conditional entropy (as it will be explained) and has meaningful applications for tasks such as graph attribute completion as it will be shown in the experimental evaluation.

For the convenience of the reader, the notations used through the rest of this paper is summarized in Table 1.

4.2. Compressing Star Pattern Miner

The input of CSPM is an attributed graph $G = (V, E, A, F, \lambda)$ and the output is a model M containing a set of compressing a-star patterns. This subsection first presents the overall framework of CSPM and then each step is described in detail.

4.2.1. Overall Framework

The overall framework of CSPM is illustrated in Fig. 6. CSPM has two main sub-procedures, named *Initialization* and *Pattern generation*. The first procedure encodes all core-values and identifies all basic a-stars that appear in the input graph G. This allows finding an initial model M (a set of compressing a-stars). Then, the second procedure performs a greedy search to improve the model M according to the MDL by not only considering basic a-stars but also merging leafsets to obtain a-stars with multiple leaf-values. The procedure stops when the model M cannot be improved to achieve more compression. Finally, the model is returned to the user. The first procedure has three steps, while the second one has two steps for a total of five steps.

Step 1: Encoding attribute values and core-values. CSPM first reads the mapping function λ of the input graph G to obtain information about the attribute values of each vertex.

These values are treated as nominal values. Then, there are two cases. If the user wants to find a-stars that may have more than one core-value, preprocessing is applied. For this, a traditional compressing itemset mining algorithm is applied by considering that the attribute values of each vertex are a transaction. Several algorithms can be used such as Krimp [41] and SLIM [38]. The output of these algorithms are itemsets representing sets of core-values that appear together. Then, each of those itemsets are then treated by CSPM as an atomic coreset. In the other case, if the user only wants to find stars having a single core-value, no preprocessing is done. Then, all attribute values are optimally

Table 1: Notations

Notation	Description
S	an a-star
S_c	coreset; the set of core-value(s) of an a-star S
S_L	leafset; the set of leaf-value(s) of an a-star S
S_c^M	the set of all coresets in the model M
S_L^M	the set of all leafsets in the model M
I	the inverted database
M	the model
CT_c	the code table for coresets in M
CT_L	the code table for leafsets in M
$L(M, I)$	total description code length encoded by the model M
$L(M)$	description length of the model M
$L(I\|M)$	description length of the new inverted database encoded by M
S_{code}	the code of S
f_L	the frequency of the corresponding a-star (line) in CT_L
f_c	sum of the frequency of coreset S_c in CT_L
s	total frequency of a-stars (lines) in I; the sum of f_L
m	total number of coresets (S_c)
n	total number of a-stars(lines) in I
l_{ij}	the frequency f_L of the a-star S with the i^{th} leaf-values and j^{th} core-values
c_j	the frequency f_c of the j^{th} core-values
ΔL	the gain for a merge operation; the difference between the description length (DL) before and after merging
P	the set of all possible pairs of leafsets to be merged
p	the pair of leafsets to be merged
S_{Lx}, S_{Ly}	the leafsets to be merged of pair p
C	the set of the coresets that S_{Lx} and S_{Ly} are both connected with
x_e	the frequency of the a-star (e, S_{Lx})
y_e	the frequency of the a-star (e, S_{Ly})
xy_e	the co-occurrence frequency of the a-stars (e, S_{Lx}) and (e, S_{Ly})
f_e	the frequency of coreset e

encoded according to their global frequencies. Thereafter, a standard code table ST of attribute values is built that will be used for calculating the length of the original database without compression. This table assigns a code to each attribute value. Moreover, a code table is created, called CT_c, that assigns a code to each coreset. This code table CT_c can be viewed as part of the initial model M that will be iteratively improved in the next steps. For the sake of simplicity, in the following examples, only stars having a single core-value are considered. In this case, the code table CT_c is identical to the standard code table.

Step 2: Construct the inverted database. Then, CSPM builds a data structure called the inverted database (I) from G that will be used to facilitate the construction of larger a-stars from smaller a-stars during the process of pattern generation, and to quickly find their coverage of the database. The inverted database stores two types of attribute values, that is the coresets and leafsets, as well as the topology information linking core-values to leaf-values. In the initial inverted database, each line records the positions of an a-star having a single edge. More details about the inverted database will be given in Subsection 4.2.2.

Step 3: Initialize the model M and calculate the description length. Then, CSPM builds an initial model M. While traditional compressing pattern mining algorithms take a single variable into account, CSPM considers two variables, that is coresets and leafsets. Hence, CSPM utilizes two code tables called CT_c and CT_L to encode all a-stars without any loss. CT_c was built in Step 1 and stores information about coresets, while CT_L contains information about leafsets. Subsection 4.2.3 describes the structure of the model M and Subsection 4.2.4 explains how to calculate the total description length of the model M using the conditional entropy.

Step 4: Generate leafset merge candidates. Thereafter, CSPM generates leafsets that could be used to make new patterns for reducing the description length of M. A possible new set of leaf-values, called a *merge candidate*, is created by combining the leafsets of two known patterns. CSPM evaluates the quality of all merge candidates. More

precisely, CSPM calculates the gain ΔL of all possible pairs of leafsets in the inverted database, a process explained in Subsection 4.2.5. Then, pairs having a gain $\Delta L > 0$ are sorted in descending order of ΔL.

Step 5: Select a-stars and update the database. Finally, new patterns are generated using the leafset pairs among merge candidates that provide the maximal gain. Then, the code table CT_L and the inverted database are updated. At this point, Steps 3 and 4 are repetitively performed until there is no pair in merge candidates that can help to further compress the database.

This approach of generating patterns on-the-fly by merging leafsets has the advantage of not requiring to mine a set of patterns beforehand as many other algorithms such as Krimp [41] and GraphMDL [2] do. This is important to have an algorithm that is truly parameter-free.

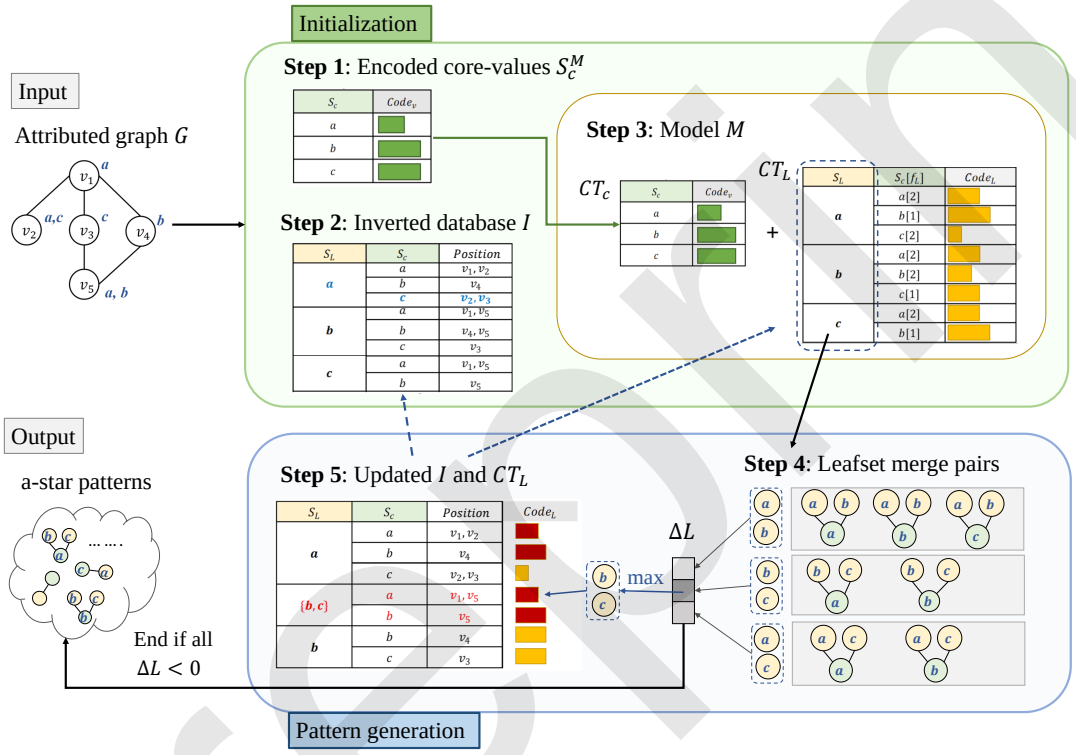


Figure 6: Overall framework of CSPM algorithm

4.2.2. The Inverted Database Structure

The CSPM algorithm relies on an *inverted database* (I) data structure to facilitate the search for a-stars that compress the input graph G. The design of the inverted database structure is based on the observation that an attributed graph can be described as the combination of a mapping function λ that assigns attribute values to vertices, and a vertex adjacency list that encodes the topology.

A vertex adjacency list contains multiple tuples, where each one can be considered as a star. The first element of a tuple is a vertex (core), which is followed by a list of adjacent vertices (leaves). For instance, the tuple $(v_4, \{v_1, v_5\})$ in the adjacency list of the graph of Fig. 2(a) is a star.

To represent attributed stars using an adjacency list, it is important to recall that an a-star describes only relationships between attribute values of vertices, and that its core can match with many different vertices of a graph. Hence, replacing vertices in an adjacency list by their attribute values can produce an a-star. As example, consider the tuple $(v_1, \{v_2, v_3, v_4\})$, which is a star. By replacing those vertices by the corresponding attribute values, the a-star $(\{a\}, \{a, b, c\})$ is obtained, which can match with other vertices, as well. This representation is interesting but since core and leaves play different roles in an a-star, it is important to label attribute values as either core-values or leaf-values.

The inverted database structure used by CSPM is based on the concept of adjacency list. It is a three-column table, where each line provides information about an a-star. The first column indicates the leafsets S_L, the second column lists the coresets S_c that are connected to the corresponding leafset in the first column to form an a-star, and the third column describes the list of vertices where the coresets appear, which are called the *positions* of an a-star. The vertices in the *Position* column can be obtained through a mapping table, which is built to map each core to its list of positions.

Example 8. *Fig. 7(a) shows the mapping table corresponding to the graph of Fig. 2(a). The inverted database generated from this graph is depicted in Fig. 7(b). It can be found from the mapping table that the coreset* S_c : $\{c\}$ *appears at vertices* v_2 *and* v_3*. Moreover, it can be observed from the graph that attribute value* $\{a\}$ *is adjacent to* $\{c\}$ *for both* v_2 *and* v_3 *as core. Hence, there is an a-star* $(\{a\},\{c\})$ *with position* $\{v_2, v_3\}$ *in the inverted database of Fig. 7(b) (third line).*

The inverted database structure is designed to support the efficient generation of a-stars to explore the search space. CSPM does this by merging pairs of lines in the inverted database where two leafsets (S_L) have the same coreset (S_c). In other words, two a-stars having the same coreset are merged to obtain a larger a-star by combining their leafsets. In CSPM, finding the a-stars that provide the best compression of the database according to the MDL is thus viewed as a problem of merging appropriate pairs of lines to produce new a-stars. Because each line of the inverted database is an a-star structure with one edge (core-value, leaf-value), all a-star patterns can be generated by merging two or more lines together. Note that the occurrence information of a-stars is stored in the *Position* column in CT_L, and the length of the position set in a line indicates the frequency of the corresponding a-star.

S_c	*Position*
a	v_1, v_2, v_5
b	v_4, v_5
c	v_2, v_3

(a)

Inverted database

S_L	S_c	*Position*
a	a	v_1, v_2
	b	v_4
	c	$\boldsymbol{v_2, v_3}$
b	a	v_1, v_5
	b	v_4, v_5
	c	v_3
c	a	v_1, v_5
	b	v_5

(b)

Figure 7: Mapping table and inverted database

Using the inverted database structure to search for a-stars has three benefits. First, the frequency of each a-star is easily obtained by calculating the length of the *Position* set of its line. And for an a-star obtained by merging two lines, the frequency can be quickly calculated by intersecting their position sets. This can be done because leaf-values that co-occur are adjacent to the same coreset. The details will be explained in subsection 4.2.5. Second, A-star patterns can be generated and updated from the inverted database structure without having to scan the whole database or the original attributed graph again. Third, the inverted database allows storing a-stars in a non-redundant way, where each line of the inverted database represents a distinct a-star pattern. As a result CSPM can encode each a-star pattern by appending a distinct code to each line, and it is unnecessary to build an additional translation table.

In terms of efficiency, building the inverted database structure is fast. It only requires reading the input attributed graph once. If the input graph is already in this format, this step is avoided.

Having explained the inverted database used by CSPM, the next subsection explains how CSPM uses this data structure to encode a-stars under the MDL principle.

4.2.3. Model Representation and Description Length Calculation

The framework of CSPM is inspired by that of Krimp [41] for discovering compressing patterns using the MDL. To calculate the description length of a model or database, Krimp utilizes a two-column code table and a standard code table. This is appropriate to mine compressing itemsets (sets of values) in a transaction database (a binary table).

But the problem studied in this paper is more complex. The goal is to mine a-star patterns in an attributed graph, where there are two types of values: core-values and leaf-values.

Due to these differences, this paper presents a novel model type for mining a-star patterns. It is composed of two code tables. The first one, called CT_c, is a traditional two-column code table, that gives a unique code to each coreset. The second code table, named CT_L, is a three-column table, indicating codes for leafsets of a-stars. The structure of CT_L is similar to that of the inverted database. But the third column of CT_L contains leafsets' codes instead of positions. To make it easy to calculate the description length of an a-star using both the codes of a coreset and a leafset, each line in CT_L contains a pointer to the corresponding coreset in CT_c. Then, the code of an a-star is the concatenation of the codes of its leafset in CT_L and that of its coreset in CT_c.

Explaining in more detail how the description length is calculated requires to first describe the *standard code table* ST. This table is the optimal encoding of attribute values. The table stores a code for each single attribute value, which is calculated based on its frequency in the attributed graph. The lengths of codes in the standard code table contribute to the description length of core-values in coresets and leaf-values in leafsets in code tables. If the user wants to find a-stars each having a single core-value in its coreset S_c, the code table CT_c has the same format as the standard code table. And if the user wants to find a-stars where each coresets may contain multiple core-values, then an algorithm is first applied to find compressing coresets (sets of core-values), as discussed in Step 1 of Subsection 4.2.1.

In the following the first case is discussed for the sake of simplicity. A-stars having a single core-value in their coresets are formed by combining two leafsets S_L having the same adjacent coresets. For instance, Fig. 8(a) shows the code tables CT_c and CT_L for the graph depicted in Fig. 2(a). To the right of CT_c, numbers under the word *Usage* give the frequency of each coreset S_c in the input attributed graph. Note that the frequency information is not a part of the CT_c table but is used to calculate the code lengths of core-values. Shorter codes are given to coresets that are more frequent in CT_c .

In CT_L, each line represents an a-star. In Fig. 8(a), the fraction besides each line of CT_L under the label f_L/f_c provides frequency information. f_L denotes the frequency of an a-star (line), while f_c is the frequency of the coreset (set of core-values) S_c according to the inverted database. The f_L/f_c ratio also indicates the relative frequency of an a-star supposing that its coreset S_c is known. Note that this frequency information f_L/f_c is not a part of CT_L but is used to calculate the code $Code_L$ of each leafset (set of leaf-values) S_L in CT_L. Then, the description length of each a-star (line) in CT_L is calculated by adding the codes $Code_c$ and $Code_L$ of S_L and S_c. For instance, consider the a-star $(\{a\}, \{b\})$ having the coreset $\{a\}$ and leafset $\{b\}$, which is the fourth line of CT_L. The code of this a-star is created by appending the code $Code_c$ of the coreset $\{a\}$ with the corresponding code $Code_L$ of its leafset $\{b\}$, as illustrated in Fig. 8(b). To facilitate the lookup of $Code_c$ in CT_L, pointers link coresets S_c in CT_L to the corresponding core-values S_c in CT_c.

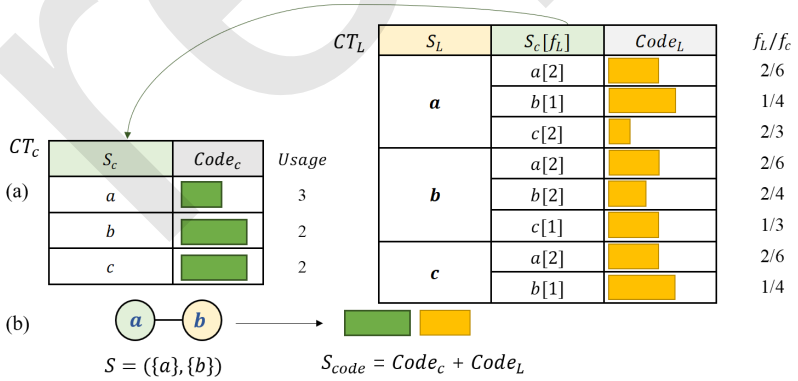


Figure 8: Code tables (1) CT_c and (2) CT_L for the graph of Fig. 2(a), and (3) the encoding of a-star $S = (\{a\}, \{b\})$

The description length $L(M, I)$ of a model and inverted database is calculated using equation (1):

$$L(M, I) = L(M) + L(I|M). \tag{1}$$

$$L(M) = L(CT_c|I) + L(CT_L|I). \tag{2}$$

The calculation of $L(M)$ is done by Equation (2), in a similar way to the Krimp algorithm. $L(M)$ is calculated by adding the code lengths of all columns in CT_L and CT_c.

The description length of the encoded database $L(I|M)$ is derived directly from the proposed inverted database structure by adding all code lengths in I together, as shown in Equation (3) and Equation (4):

$$L(I|M) = \sum_{S \in I_{new}} L(S_{code}) \times S(f_L). \tag{3}$$

$$L(S_{code}) = L(Code_c) + L(Code_L). \tag{4}$$

where f_L is the frequency of each a-star (line) S in the new inverted database I_{new} that is described using M, and $L(S_{code})$ is the code length of S, which is calculated by summing up the code lengths of all coresets $L(Code_c)$ and leafsets $L(Code_L)$.

CSPM is designed to identify a-stars that indicate strong correlations between a leafset and a coreset by applying the MDL principle. To evaluate how relevant a set of leaf-values is for a set of core-values, the conditional entropy is used, as described next.

4.2.4. Code Length Calculation Based on the Conditional Entropy

This subsection describes how CSPM calculates the optimal code lengths of a-stars. Unlike traditional methods such as Krimp, CSPM does not use the Shannon entropy to compute the optimal code lengths of a-stars but instead the conditional entropy. There are two reasons. First, the Shannon entropy is only suitable for a single variable but the proposed model has two variables (a coreset and a leafset). Second, a goal of this work is to evaluate the relevance between leafsets and coresets instead of merely focusing on their frequencies.

For a model M, CSPM calculates the code $Code_c$ of coreset S_c using Equation (5) where $p(S_c)$ is the relative occurrence frequency of S_c in the input attributed graph.

$$L(S_c) = -\log(p(S_c)). \tag{5}$$

To optimally encode how leaf-values are connected to core-values in an a-star, the conditional entropy is used as measure. In information theory, the conditional entropy $H(Y|X)$ measures the cost of describing a random variable Y if the value of another random variable X is known [6]. For CSPM, the set S_c^M of all coresets in a model can be regarded as X and the set of all leafsets S_L^M in M as Y. Based on this fact, the relevance of core-values to leaf-values in an a-star can be assessed using the conditional entropy. According to the principle of conditional entropy, the length of the code $Code_L$ of each line for a leafset S_L and coreset S_c is given by Equation (6). Hence, $Code_L$ is only determined by the frequencies f_L and f_c.

$$L(Y = S_L|X = S_c) = -\log \frac{p(S_L, S_c)}{p(S_c)} = -\log \frac{f_L}{f_c}. \tag{6}$$

Then, $H(Y|X)$ is calculated by Equation (7), which represents the average encoding cost of each line in CT_L after using the conditional entropy encoding method.

$$\begin{aligned} H(Y|X) &= -\sum_{S_c \in X, S_L \in Y} p(S_c, S_L) \log \frac{p(S_c, S_L)}{p(S_c)} \\ &= \sum_{j=1}^{m} \sum_{i=1}^{n} \frac{l_{ij}}{s} \log \frac{l_{ij}}{c_j}, \end{aligned} \tag{7}$$

In that equation, s is the total frequency of a-stars (lines) in the inverted database, which is the sum of f_L; m is the total number of coresets S_c in CT_c; n is the total number of a-stars (lines) in the inverted database; l_{ij} is the frequency f_L of the i^{th} leafset and j^{th} coreset; and c_j is the frequency f_c of the j^{th} coreset.

Based on the above equation, the total encoding cost of the inverted database is obtained by Equation (8). Note that $\sum_{i=1}^{n} l_{ij}$ is equal to the frequency of the j^{th} coreset. Thus, the first term of Equation (8) is only related to the frequencies of the coresets.

$$
\begin{aligned}
L(I|M) = s \times H(Y|X) &= \sum_{j=1}^{m} \sum_{i=1}^{n} l_{ij} \log \frac{l_{ij}}{c_j} \\
&= \sum_{j=1}^{m} \underbrace{\left(\sum_{i=1}^{n} l_{ij} \right)}_{=c_j} \log c_j - \sum_{j=1}^{m} \sum_{i=1}^{n} l_{ij} \log l_{ij} \\
&= \sum_{j=1}^{m} c_j \log c_j - \sum_{j=1}^{m} \sum_{i=1}^{n} l_{ij} \log l_{ij}
\end{aligned}
\tag{8}
$$

4.2.5. Pattern Generation with the Merge Operation

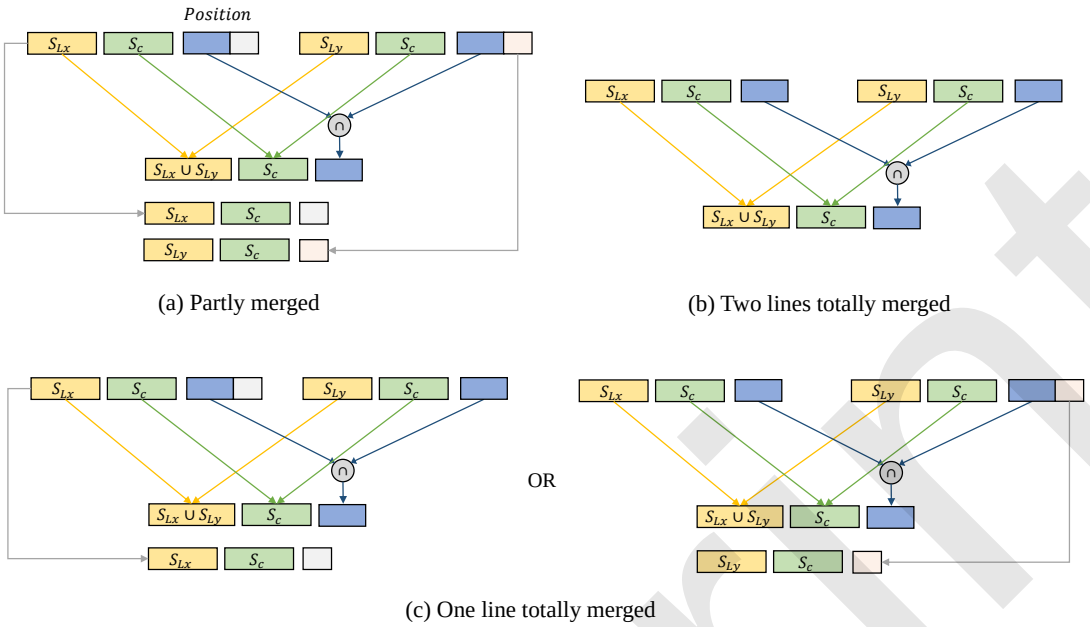
This subsection explains how CSPM generates a-stars to build a model M. CSPM updates a model M iteratively. During each iteration, CSPM selects a pair of leafsets and then merges the corresponding a-stars in the inverted database I to improve the model. This can be viewed as a problem of selecting the set of a-stars that contributes to the smallest description cost i.e. the set of leafsets that have the smallest code length and compress the original data best given some coresets. To find a-stars to be added to M, a straightforward way is to compute the description lengths of all combinations of all possible sets of leaf-values and core-values. However, this approach is inefficient for large databases due to the large number of possible combinations. To provide a more efficient approach CSPM utilizes a merge operation. It consists of merging pairs of leafsets in CT_L that provide the best gain ΔL, where ΔL is the difference between the DL (Description Length) before merging and after merging (a greater gain is better). This merge operation is used during each iteration to find a-stars to be added to M. Compared to traditional algorithms such as Krimp this greedy approach based on a merge operation has the advantage of not requiring to mine a set of candidate patterns beforehand using an external algorithm. This is important so that CSPM is a parameter-free algorithm. The details about how to merge are presented next, and then how to calculate the gains.

(1) How to merge? Merging requires to update two structures, which are the **inverted database** and the **code table** CT_L. The next paragraphs first explain how to merge leaf-values pairs based on the inverted database, and how to update the inverted database.

For ease of explanation, let P be the set of all possible leafsets pairs $P = \{(S_{Lx}, S_{Ly}) | \, S_{Lx}, S_{Ly} \in CT_L\}$ that can be merged. Merging a pair p, which consists of S_{Lx} and S_{Ly}, means to create a new leafset $\{S_{Lx} \cup S_{Ly}\}$ that covers all situations where S_{Lx} and S_{Ly} are adjacent to the same coresets. The set of all coresets where the two leafsets are both adjacent to is denoted as C where $C \subseteq S_c^M$. The occurrences of p for each coreset $S_c \in C$ are easily obtained from the inverted database by intersecting the corresponding positions of the leafsets S_{Lx} and S_{Ly} with the coreset S_c. More precisely, three cases can be encountered when intersecting the positions, which are called *partly merged*, *two lines totally merged*, and *one line totally merged*.

Fig. 9 illustrates these three cases. In this figure, blue rectangles represent the positions where the new a-star $(S_c, S_{Lx} \cup S_{Ly})$ occurs. After these positions are calculated, they are removed from the positions of a-stars (S_c, S_{Lx}) and (S_c, S_{Ly}). The first case is *partly merged* (Fig. 9 (a)), which means that there are two a-stars (S_c, S_{Lx}) and (S_c, S_{Ly}) that remain after the merge (their position sets are not empty). The second case is *two lines totally merged* (Fig 9 (b)), where the original a-stars are perfectly merged. This means that the leafsets S_{Lx} and S_{Ly} always appear together around the coreset S_c. Thus, the result is a single a-star $(S_c, S_{Lx} \cup S_{Ly})$. The third case is *one line totally merged* (Fig 9 (c)), where either the a-star (S_c, S_{Lx}) or (S_c, S_{Ly}) remain after the merge.

Example 9. *Consider the inverted database shown in Fig. 7 (b). It has three leaf-values* $\{a\}$, $\{b\}$, *and* $\{c\}$. *During the first iteration, there are three possible merge pairs, that are* $P = \{(\{a\}, \{b\}), (\{a\}, \{c\}), (\{b\}, \{c\})\}$. *Suppose that the merge pair* $p = (\{b\}, \{c\})$ *is selected and merged, i.e, a new leafset* $\{b, c\}$ *is generated. Fig.10 (a) illustrates the merging process. First, CSPM identifies* C, *that is all coresets that* p *is adjacent to. In this case,* $C = \{\{a\}, \{b\}\}$, *which means that the leafset* $\{b, c\}$ *appear both around the coresets* $\{a\}$ *and* $\{b\}$. *Then, the positions of the values in the*



(a) Partly merged

(b) Two lines totally merged

(c) One line totally merged

Figure 9: The three cases when merging leafsets S_{Lx} and S_{Ly} with some coreset S_c to build an a-star $(S_c, S_{Lx} \cup S_{Ly})$

coreset C are calculated by intersecting the corresponding positions. For the coreset $\{a\}$*, the positions of a-stars with leaf-values* $\{b\}$ *and* $\{c\}$ *are both* $\{v_4, v_5\}$*. Thus, it is the case of two lines totally merged. The positions of the new a-star are* $\{v_4, v_5\} \cap \{v_4, v_5\} = \{v_4, v_5\}$*, and the two merged lines are removed at the same time. For the coreset* $\{b\}$*, it is the case of one line totally merged. The intersection of the positions is* $\{v_4, v_5\} \cap \{v_5\} = \{v_5\}$*. Thus, the leafset* $\{c\}$ *with coreset* $\{b\}$ *is perfectly matched and removed, while* $\{v_4, v_5\} - \{v_5\} = \{v_4\}$ *for the leafset* $\{b\}$ *and thus the line of* $\{b\}$ *is kept but updated to contain only* $\{v_4\}$*. To further help understanding this process, Fig. 10 (b) and Fig. 10 (c) depicts the a-stars corresponding to the inverted database shown in Fig. 10 (a) in the input attributed graph, using the same colors. It can be observed that the patterns are mined and replaced as the inverted database is updated.*

When doing a merge operation, CSPM also updates CT_L by adding or removing the lines that map patterns to their codes. The first two columns of CT_L are updated according to the first columns of I to record the changes made to a-star patterns.

Example 10. *Fig. 11 shows the updated inverted database and code lengths* $Code_L$ *after merging the pair* $(\{b\}, \{c\})$*. The code lengths are calculated by Equation (6). It can be observed in Fig. 11 that leafsets* $\{b\}$ *and* $\{c\}$ *are both connected to the same coreset* $\{a\}$ *of vertices* $\{v_1, v_5\}$ *and that they are connected to coreset* $\{b\}$ *of vertex* $\{v_5\}$*. Interestingly, the code length of leafset* $\{b, c\}$ *in* CT_L *corresponding to the coreset* $\{a\}$ *is much shorter than previously. Thus, the description length is also reduced by compression. Besides, if there is no attribute value a that appears at* v_1 *and* v_2 *in the attributed graph, the code length of the leafset* $\{b, c\}$ *in* CT_L *corresponding to the coreset* $\{a\}$ *will be 0. This can be interpreted as indicating that in all situations where the core-value* $\{a\}$ *appears, there must be the leaf-values* $\{b, c\}$ *in some adjacent vertices. This means that the more strong the relationship is, the smaller the code length will be.*

It can be observed that not all variables need to be changed after a merge operation in Equation (10). For the above example presented in Fig. 11, only the frequencies of coresets $\{a\}$ and $\{b\}$ and leafsets $\{b\}$ and $\{c\}$ have been updated, because these variables are directly related to the code length.

(2) How to calculate the gains? To calculate the gains, a simple approach would be to do the merge for each pair to compute the differences of the description lengths. Thought, this would work, it would be inefficient because the inverted database and the code table CT_L are changed by merging, and these changes would have to be reverted after each gain calculation. This problem can be solved by observing that there are only few parts of the DL calculation that are influenced by a merge operation.

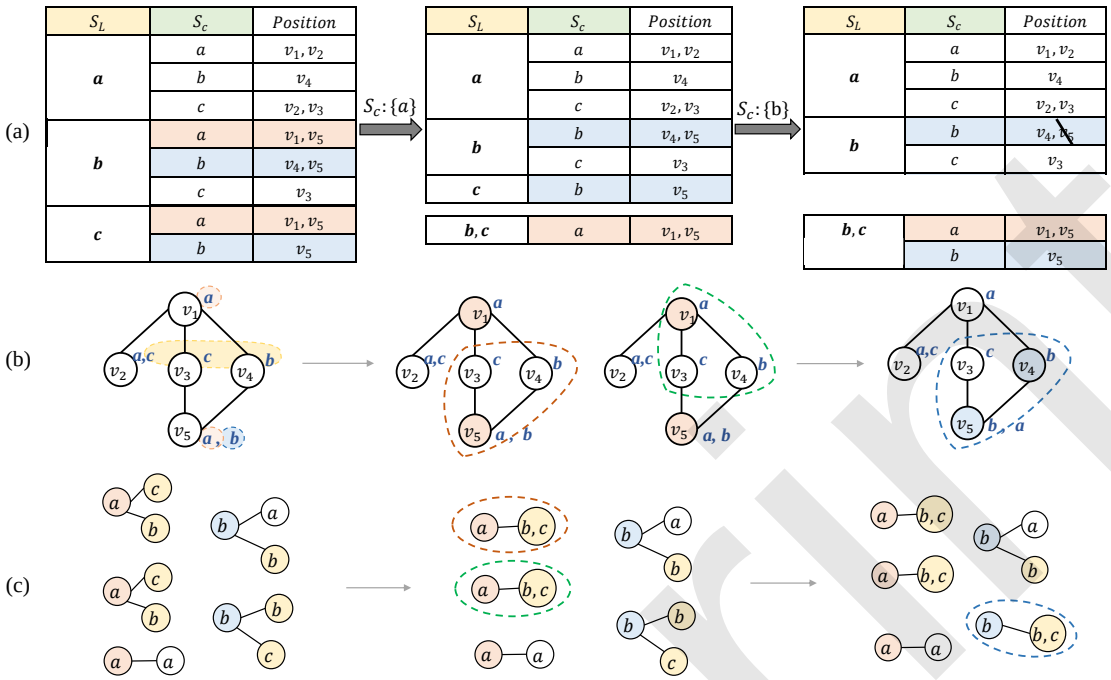


Figure 10: An example of merging leafsets pair $(\{b\},\{c\})$ in the inverted database

S_L	S_c	Position	$Code_L$	f_L/f_c
a	a	v_1, v_2		2/4
	b	v_4		1/3
	c	v_2, v_3		2/3
{b, c}	a	v_1, v_5		2/4
	b	v_5		1/3
b	b	v_4		1/3
	c	v_3		1/3

Figure 11: Updated inverted database and $Code_L$ after merging the leafsets pairs $(\{b\},\{c\})$

According to Equation (1), the gain ΔL for generating a pattern $\{S_{Lx} \cup S_{Ly}\}$ can be separated into two parts as Equation (9), which are $\Delta L(M)$ and $\Delta L(I|M)$. The first one is the updated cost of the model M, and the other one is the encoded inverted database. The next paragraphs explain these two parts in detail.

The first part, $\Delta L(M)$, is related to the changes of CT_c and CT_L, based on Equation (2). Note that $\Delta L(CT_c|I)$ is equal to zero because CT_c is not changed, and that $\Delta L(CT_L|I)$ could be obtained by adding or removing lines of a-stars and updated code lengths. The cost of the representation of the first two columns in CT_L can be easily obtained through the standard code table, and the code length of the third column is also calculated through Equation (6).

$$\Delta L = \Delta L(M) + \Delta L(I|M) = \Delta L(CT_c) + \Delta L(CT_L) + \Delta L(I|M) \tag{9}$$

The second part, $\Delta L(I|M)$, is the gain resulting from merging the pair of leaf-values. Through Equation (8), a simplified version of the $\Delta L(I|M)$ calculation is obtained as Equation (10):

$$\Delta L(I|M) = \underbrace{\left(\sum_{i=1}^{m} c_j \log c_j - \sum_{i=1}^{m} c_j^{'} \log c_j^{'}\right)}_{P_1} - \underbrace{\left(\sum_{j=1}^{m}\sum_{i=1}^{n} l_{ij} \log l_{ij} - \sum_{j=1}^{m}\sum_{i=1}^{n^{'}} l_{ij}^{'} \log l_{ij}^{'}\right)}_{P_2}, \tag{10}$$

where, variables with prime as superscript indicate the properties of the new code table $CT_L^{'}$ resulting from the merge operation. It is to be noted that the total number of core-values m does not change during the whole process. This is because the set of coresets S_c^M is known and fixed.

To better explain Equation (10), a detailed analysis of its two main terms P_1 and P_2 is given. P_1 is related to the coresets while P_2 is related to the leafsets. In the following, for the sake of brevity, the lower case variable x is used to denote the frequency f_L of the leafset S_{Lx}, i.e. $x = S_{Lx}.f_L$. Similarly, the variable y is used to denote $S_{Ly}.f_L$.

Recall the observation that not all frequencies for each line f_L and each core-value f_c will be changed by a merge operation. Suppose that C is the set of coresets that S_{Lx} and S_{Ly} are both connected with. For each coreset $e \in C$, f_e is the frequency of coreset e and xy_e is the co-occurrence frequency for a-stars (e, S_{Lx}) and (e, S_{Ly}). In particular, if xy_e is equal to zero, there is nothing to do because the two lines cannot be merged, and the gain is equal to zero. P_1 in Equation (10) could be derived as shown in Equation (11):

$$P_1 = \sum_{e \in C} \left(f_e \log f_e - (f_e - xy_e) \log(f_e - xy_e)\right). \tag{11}$$

Due to the fact that not all values of $l_{ij}^{'}$ will become different from l_{ij} after merging, the term P_2 is calculated by only summing up all relative merged lines using Equation (12), where P_e means the gain for the merged lines with core-value e.

$$P_2 = \sum_{e \in C} P_e. \tag{12}$$

The three cases discussed above are considered based on the relationship between xy_e, x_e and y_e.

(1) **Partly merged**. In this case, part of the positions where the newly merged pattern appears are such that $xy_e \neq x_e, xy_e \neq y_e$. The formulation P_e can be derived as Equation (13).

$$\begin{aligned} P_e &= (x_e \log x_e + y_e \log y_e) - [(x_e - xy_e) \log(x_e - xy_e) + (y_e - xy_e) \log(y_e - xy_e) + xy_e \log xy_e] \\ &= x_e \log \frac{x_e}{x_e - xy_e} + y_e \log \frac{y_e}{y_e - xy_e} - xy_e \log \frac{xy_e}{(y_e - xy_e)(x_e - xy_e)}. \end{aligned} \tag{13}$$

(2) **Two lines totally merged**. In this case, $xy_e = x_e$ and $xy_e = y_e$. The two lines (a-stars) are merged as a new pattern because they always appear at the same positions. After merging, there is no a-star (e, S_{Lx}) or (e, S_{Ly}) in the inverted database or code table anymore. Thus, P_e can be calculated as Equation (14).

$$P_e = (x_e \log x_e + y_e \log y_e) - (xy_e \log xy_e) = xy_e \log xy_e. \tag{14}$$

(3) **One line totally merged**. In this case, only one a-star is removed by merging. Hence, $xy_e = x_e, xy_e \neq y_e$ or $xy_e \neq x_e, xy_e = y_e$, as shown as Equation (15) and (16), respectively.

$$\begin{aligned} P_e &= (x_e \log x_e + y_e \log y_e) - (y_e \log y_e + xy_e \log xy_e) \\ &= y_e \log \frac{y_e}{y_e - xy_e} + xy_e \log(y_e - xy_e). \end{aligned} \tag{15}$$

$$\begin{aligned} P_e &= (x_e \log x_e + y_e \log y_e) - (y_e \log y_e + xy_e \log xy_e) \\ &= x_e \log \frac{x_e}{x_e - xy_e} + xy_e \log(x_e - xy_e). \end{aligned} \tag{16}$$

4.3. Optimization

The main procedure of CSPM is illustrated in Algorithm 1. First, all the coresets of S_c^M are encoded (line 1). Then, in lines 2-4, the algorithm constructs the inverted database I and initializes the two parts (CT_c and CT_L) of the model M, which correspond to Steps 2 and 3 of the overall framework described in Subsection 4.2.1. Next, Step 4 and Step 5 are completed in lines 5-10. These two steps iteratively generate candidates and merge the best pair with the maximum gain. Details about candidate generation are shown in Algorithm 2. The algorithm calculates the gains of all possible pairs and selects pairs with positive gains as candidates.

Algorithm 1: The CSPM-Basic algorithm

input : An attributed graph G with two parts: the adjacency list and the mapping function
output: Compressing star attribute patterns: a set of a-stars

1 Encode all coresets S_c^M;
2 $CT_c \leftarrow (S_c, Code_c)$;
3 $I \leftarrow (S_L, S_c, Position)$;
4 $CT_L \leftarrow (S_L, S_c, Code_L)$;
5 *candidates* $\leftarrow$ generate_candidate(S_L^M); `// Algorithm 2`
6 **repeat**
7 $\quad p \leftarrow$ *candidates.pop*(); `// select pair with the maximum gain` ΔL
8 $\quad I, CT_L \leftarrow$ merge_operation(p); `// update` I `and` CT_L
9 $\quad$ *candidates* $\leftarrow$ generate_candidate(S_L^M);
10 **until** *candidates* $= \emptyset$;
11 **return** *all a-stars in M*

Algorithm 2: Generate leafsets merge candidates

input : All leafsets S_L^M in CT_L, inverted database I
output: A candidate list of merge pairs

1 *candidates* = [];
2 $P \leftarrow$ enumerate(S_L^M, 2);
3 **for** $p \in P$ **do**
4 $\quad$ *gain* $\leftarrow$ calculate_gain(p, I) `// Equation (9)`
5 $\quad$ **if** *gain* > 0 **then**
6 $\quad\quad$ candidates.append(p);
7 $\quad$ **end**
8 **end**
9 Sort *candidates* by descending order of gain;
10 **return** *candidates*

The basic CSPM algorithm calculates many gains and only merges one pair in each iteration, which may not be very efficient when there are many possible pairs. To solve this problem and improve the performance of the process of candidate generation and pair merging, an optimized version of CSPM, named CSPM-Batch, is designed. The main idea of CSPM-Batch is to merge as many pairs in one iteration as possible, and at the same time try to ensure the quality of the patterns that are mined. In this way, a good trade-off may be reached between efficiency and quality of patterns found.

To be more precise, pairs that are merged in batch during an iteration need to meet the requirement that leafsets of all pairs are disjoint. This restriction ensures that pairs are independent and thus that there is no conflict when intersecting positions. Algorithm 3 shows the pseudocode of the CSPM-Batch algorithm. The differences with the basic version are from line 7 to line 13. A set named (used_set) is used to store all leafsets of pairs that have been merged in the current iteration. Then, the algorithm checks all the candidate pairs in order and a pair can be merged if its two leafsets have not been used in this iteration.

Algorithm 3: The CSPM-Batch algorithm

input : An attributed graph G with two parts: the adjacency list and the mapping function
output: Compressing star attribute patterns: a set of a-stars

```
Encode all coresets S_c^M;
CT_c ← (S_c, Code_c) ;
I ← (S_L, S_c, Position) ;
CT_L ← (S_L, S_c, Code_L);
candidates ← generate_candidate(S_L^M); // Algorithm 2
repeat
    used_set = set();
    while candidates do
        p ← candidates.pop();
        if p[0] ∉ used_set and p[1] ∉ used_set then
            I, CT_L ← merge_operation(p); // update I and CT_L
            used_set.add(p[0]); // S_Lx
            used_set.add(p[1]); // S_Ly
        end
    end
    candidates ← generate_candidates(S_L^M);
until candidates = ∅;
return all a-stars in M
```

Next, the impact of the optimization method are analyzed briefly on two aspects: the merge operation and the pattern generation order. Fig. 12(a) illustrates a conflicting case where two pairs are merged with the same leafset at the same time. Suppose that the pairs $(\{b\},\{c\})$ and $(\{b\},\{d\})$ are both in the candidate list, and the pair of $(\{b\},\{c\})$ has a higher gain. First, the pair $(\{b\},\{c\})$ is merged, and a new leafset $\{b,c\}$ is generated. After that, the leafset appearing around the coreset $\{a\}$ are $\{b,c\}$, $\{d\}$, and $\{e\}$, which means that $(\{b\},\{d\})$ are not candidates that could be merged together anymore for the core-value $\{a\}$. However, the pair $(\{e\},\{d\})$ could still be merged because the two operations do not affect each other on pattern generation, as shown in Fig. 12(b). Thus, by selecting merge pairs in batch can ensure that there is no conflict that may influence the final results.

A point that needs to be emphasized is that merging pairs in batches may change the merge order compared with the basic method, and as a result the patterns mined by these two greedy strategies may be slightly different. For example, Fig. 12(b) shows a case where the CSPM-Batch algorithm merges two leafsets pairs, $(\{b\},\{c\})$ and $(\{e\},\{d\})$, during the same iteration. There are two compressing a-stars obtained, which are $(\{a\},\{b,c\})$ and $(\{a\},\{e,d\})$. However, when the CSPM-Basic algorithm is applied to the same case, the result is different. The procedure of the first iteration also merges the pair $(\{b\},\{c\})$, but the resulting leafset $\{b,c\}$ will be considered in the next iteration. That means that $(\{b,c\},\{d\})$ may be the best pair to compress the data. In this way, the final leafset is $\{b,c,d\}$ and the corresponding a-star is $(\{a\},\{b,c,d\})$, depicted in Fig. 12(c).

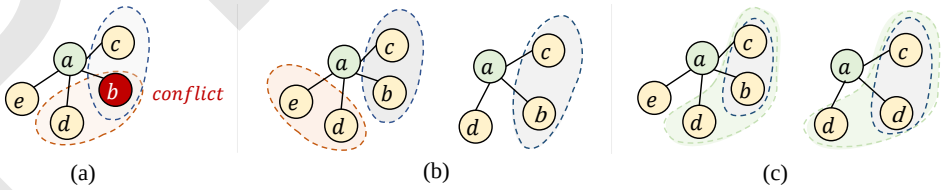


Figure 12: Merge examples of CSPM-Batch and CSPM-Basic

In conclusion, the CSPM-Batch algorithm is an optimization to improve the efficiency of the basic CSPM algorithm. It modifies the greedy strategy by merging one or several independent pairs during each iteration. The quality of the results is ensured by selecting the candidate pairs by order of gains. Because the order of pattern generation is not exactly the same for the two algorithms, the results of CSPM-Batch and CSPM-Basic may be different.

4.4. Complexity

This subsection analyzes the complexity of the CSPM algorithm step-by-step. Suppose that an attributed graph G has $|E|$ edges, $|V|$ vertices, $|A|$ distinct attribute values, and the average number of values per attribute is $|\bar{A}|$. First, the algorithm needs to encode all the coresets. In this step, $O(1)$ time is required to insert elements in CT_c. Next, to construct the inverted database, all the attributes are scanned in all vertices, which takes at most $O(|E| \times |\bar{A}|^2)$. After that, the patterns are generated by doing merges. For each iteration, candidates generation (Algorithm 2) takes at most $O(|S_L^M|^2)$ steps. Suppose there are at most $|F|$ iterations. The complexity of a merge operation is $O(1)$. For CSPM-Basic, the merge operation is executed once in an iteration, and thus the complexity of the mining procedure is $O(|S_L^M| \times |F|)$. While for CSPM-Batch, the complexity is $O(|\bar{S_L^M}| \times |F|)$, where $|\bar{S_L^M}|$ is the average number of merge pairs in an iteration, which is at worst equal to $\lfloor \frac{1}{2} \times |S_L^M| \rfloor$. Note that the number of iterations $|F|$ of CSPM-Batch is much smaller than CSPM-Basic, and this explains why CSPM-Batch is more efficient in runtime. It could be observed that the efficiency of the algorithm is relevant to the average attribute dimension $|\bar{A}|$ and the number of iterations $|F|$, which are quite small in practice.

5. Variants of CSPM and their applications

The CSPM algorithm can mine compressing patterns in an attributed graph. CSPM and its inverted database structure are designed so that they can be adapted to meet the needs of various applications with only minor changes. To illustrate this, this section presents two variants of CSPM. First, Subsection 5.1 introduces a variant named CSPM$_w$, which adds weights to leaf nodes around each core node of a-stars to mine more specific patterns. This is beneficial for tasks such as graph completion, as it is explained in Subsection 5.2. Second, Subsection 5.3 explains how CSPM can be used for discovering patterns in dynamic attributed graphs by adapting its inverted database representation. An industrial application of this variant is described to analyze the correlation between alarms in a telecommunication network. Finally, how to adapt CSPM to other types of graphs is briefly discussed in subsection 5.4.

5.1. Compressing Star Pattern Miner with Position Weights

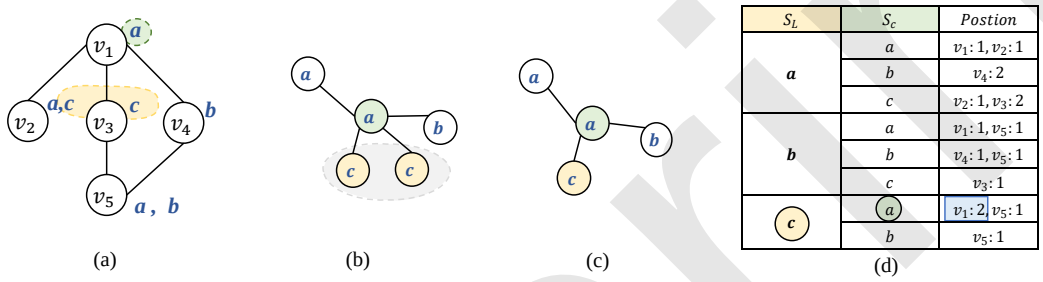
The CSPM algorithm is designed to mine a type of patterns that is quite general, indicating that some leaf-values appear near some core-values. Although this is useful, a limitation of CSPM is that it does not consider that a leafset may appear multiple times around a coreset. This is illustrated with an example. Consider the toy graph of Fig. 2. The coreset $\{a\}$ appears in vertex v_1, which is highlighted in green color in Fig. 13(a). To form a-stars using this coreset, there are three basic leafsets that can be used, which are $\{a\}$, $\{b\}$ and $\{c\}$. It can be observed in Fig. 13(a) that the leafset $\{c\}$ appears twice around the coreset $\{a\}$ for vertex v_1. But this information is not taken into account by CSPM. As a result, CSPM considers that the a-star depicted in Fig. 13(b) is the same as the a-star illustrated in Fig. 13(c). For some applications such as graph completion, it is desirable to distinguish between such cases, and to have explicit information about the number of occurrences of a leafset around a coreset. For instance, in social network analysis, a person having two friends who likes pop music can be viewed as a different situation from a person having two friends or ten friends who like pop music.

To address this limitation, a variation of CSPM is proposed named CSPM$_w$, which adds weights to each vertex in the *Position* column of the inverted database. The weights are used to store information about how many times a leafset appear around a coreset for each vertex where the coreset appears. Thus the frequency of each a-star is the sum of weights in its *Position* column. This modification is illustrated in Fig. 13(d), which shows the inverted database built by CSPM$_w$ for the graph of Fig. 2(a). It can be observed for the a-star composed of $S_L : \{c\}$ and $S_c : \{a\}$ that the weight is 2 for v_1 and that the weight is 1 for v_5. This indicates that the leafset $\{c\}$ appears twice around v_1 but only once around v_5, thus distinguishing between these two situations.

To handle these weights, the main procedure of CSPM$_w$ is slightly different from CSPM. The main difference is how the co-occurrence frequency of two positions is calculated to find the gain of a merge pair. When two lines are merged their *Position* sets are compared. If a vertex is found to appear in the two *Position* sets, then the smallest weight is selected. Then, the vertex is added to the *Position* set of the new line with that weight. Finally, that weight is subtracted from weights of that vertex in the *Position* sets of the two lines that are merged.

This is illustrated with an example. Suppose that there are two lines $(S_c : \{a\}, S_L : \{b\})$ and $(S_c : \{a\}, S_L : \{c\})$ to be merged. The CSPM algorithm does not handle weights and thus simply merges these lines as shown in the first

step (pink part) of Fig. 10(a). The *Position* column is updated by CSPM as follows. First, the *Position* sets of the two lines are intersected as $\{v_1, v_5\} \cap \{v_1, v_5\} = \{v_1, v_5\}$ to obtain the *Position* set of the new pattern. Then, the *Position* sets of the merged lines are updated by removing the *Position* set of the new line from those of the merged lines. In this case, we have $\{v_1, v_5\} - \{v_1, v_5\} = \emptyset$ and thus, the two lines are totally merged and removed. For CSPM$_w$ the process is slightly different because the *Position* sets of $(S_c : \{a\}, S_L : \{b\})$ and $(S_c : \{a\}, S_L : \{c\})$ are not the same due to weights. They are $\{v_1 : 2, v_5 : 1\}$ and $\{v_1 : 1, v_5 : 1\}$, respectively. Fig. 14 shows the detailed process of updating the *Position* sets by CSPM$_w$. The vertices that are common to the two lines to be merged are also $\{v_1, v_5\}$. However, the difference with CSPM is that CSPM$_w$ selects the smallest weight of each vertex to obtain the new pattern's *Position* set. Then, that weight is subtracted from the vertex's weights in the two merged *Position* sets. For the vertex v_1, the weight in the new *Position* set is 1 because the smallest weight of v_1 in the two merged lines is 1, and similarly, the weight of v_5 is 1. To update the merged *Position* sets, we need to subtract the corresponding weight from the original *Position* sets. As a result, the line of a-star $(\{a\}, \{c\})$ is not perfectly merged and the remaining position is $\{v_1 : 1\}$.



S_L	S_c	Postion
a	a	v_1:1, v_2:1
	b	v_4:2
	c	v_2:1, v_3:2
b	a	v_1:1, v_5:1
	b	v_4:1, v_5:1
	c	v_3:1
c	a	v_1:2 v_5:1
	b	v_5:1

Figure 13: A graph, two a-stars, and an inverted database built by the CSPM$_w$ algorithm

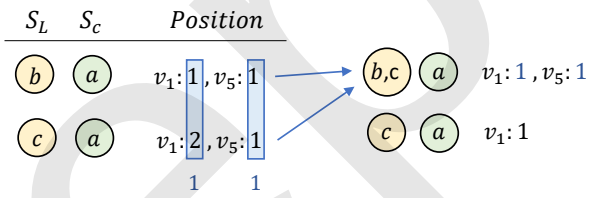


Figure 14: An example of how CSPM$_w$ updates the *Position* sets when merging two lines

5.2. *CSPM$_w$ for Attribute Completion*

This subsection describes a practical application of CSPM$_w$, which is *graph completion*. Graph completion is a popular machine learning problem. The input is a graph where some attribute values of nodes are missing. The output is a set of predicted attribute values that are expected to have a high accuracy and be ranked by importance for each node. The motivation of this application is the following. Numerous real-world graphs contains nodes that have missing attribute values. These graphs are said to be *attribute-missing graphs*. For example, in a social network, information about some users may be unknown. For a social network business, it is beneficial to infer this information as it can play a role for advertisement and also to provide personalized recommendations and services to users.

Graph completion is not an easy problem. Making predictions at the attribute level is a nontrivial task. It requires to build models with a higher prediction capability than for tasks such as node classification and link prediction. Therefore, few methods can achieve excellent results for this task [4], and there is much room for improvement.

To be able to use CSPM for graph completion, a simple attribute value scoring method based on a-stars found by CSPM$_w$ was designed. It uses attribute value scores generated by the scoring module to improve the performance of the state-of-the-art graph attribute completion algorithms.

CSPM scoring module. The designed scoring module is based on the idea that attribute values from a node may be viewed as core-values and those of its neighbors as leaf-values. Furthermore, the missing values of a node should be core-values in a-stars. The code lengths of a-stars are easily obtained through the model M produced by CSPM$_w$.

According to the MDL principle, a shorter code length means a higher possibility that the a-star appears. Therefore, a core-value in an a-star that is encoded with a lower score is more likely to be an attribute value of the vertex. Note that because we want to score all the possible single attribute values, CSPM$_w$ was configured to found coresets containing only one core-value for this application.

Algorithm 4: The scoring module of CSPM

input : An attributed graph G with n attribute values, the model M, and a node v with missing attribute values
output: Scores for all possible attribute values

1 $scores \leftarrow [n]$; `// Initialize an array to store the scores of the` n `attribute values`
2 $neighbors \leftarrow$ neighbor_attributes(v);
3 **foreach** *a-star* S *in* M **do**
4 $\quad S_{code} \leftarrow$ code_length(S); `// Equation (4)`
5 $\quad w \leftarrow$ distance(S_L, $neighbors$); `//` S_L`:leafsest of` S
6 $\quad cl \leftarrow w \times S_{code}$;
7 $\quad$ **if** $cl < scores[S_c]$ **then**
8 $\quad\quad scores[S_c] \leftarrow cl$; `//` S_c`:coreset of` S
9 $\quad$ **end**
10 **end**
11 **return** $scores$

The proposed scoring module is described in Algorithm 4. It takes as input an attributed graph G having n attribute values, a model M generated by CSPM, and a node v with missing attribute values. The algorithm returns scores for all possible attribute values for v based on the core-values in the model M. The algorithm is applied as follows. First, the attribute values appearing around the vertex v are obtained (line 2). Next, the algorithm scans the leafsets of all a-stars in model M to find the minimum code length for each possible core-value (line 3-10). In general, it is difficult to perfectly match the leaf-values in the leafset of an a-star with the attribute values of a neighbor due to differences between vertices. Thus, a weight is introduced to allow partial matches. The weight represents the similarity between leaf-values in leafset of an a-star and that of the neighbors of v. There are multiple choices to calculate the weight in line 5, such as the Euclidean distance and cosine distance. Intuitively, a leafset that is not similar to a set of neighbors will have a larger weight w. And this will lead to a higher cl (score), which means that the corresponding core-values are less likely to be an attribute value of v.

Note that although the CSPM algorithm is parameter-free, we add a parameter γ to allow adjusting the influence of S_c and S_L when scoring the attribute values. The function code_length in line 4 of Algorithm 4 is then $L(S_{code}) = \gamma \times L(Code_L) + (1 - \gamma) \times L(Code_c)$.

The scores generated by the scoring module can then be used to improve the results of state-of-the-art machine learning algorithms for graph attribute completion. Fig. 15 illustrates the whole process. Given an attribute-missing graph, nodes with attributes are used to train an existing graph completion model (e.g. a neural network), and at the same time the CSPM algorithm is applied to mine the a-star patterns. Then, the trained graph completion model and CSPM model are jointly used to predict the missing attributes of the test vertices. The output of the graph completion model for each vertex is a vector containing values representing the possibilities that the attribute values appear, while the vector obtained by the CSPM model records the scores of its scoring module (Algorithm 4). Note that, for the vector of possibilities, higher means better, which is opposite to the scores of CSPM. Thus, the probabilities output by the graph completion model can negate the scores of the CSPM's scoring module. The two vectors are normalized separately and then multiplied to combine them together. Experimental results for this application will be presented in Section 6.

5.3. CSPM$_t$ to Handle Dynamic Attributed Graphs with Application to Alarm Analysis

A second variant of CSPM named CSPM$_t$ has been developed for discovering a-stars in a dynamic attributed graph. A *dynamic attributed graph* is an attributed graph that is observed at different timestamps, where attribute values can change, and vertices and edges may be added and removed. Dynamic attributed graphs can be used to encode various types of data such as co-authorship networks and airport traffic data [12]. However, the primary

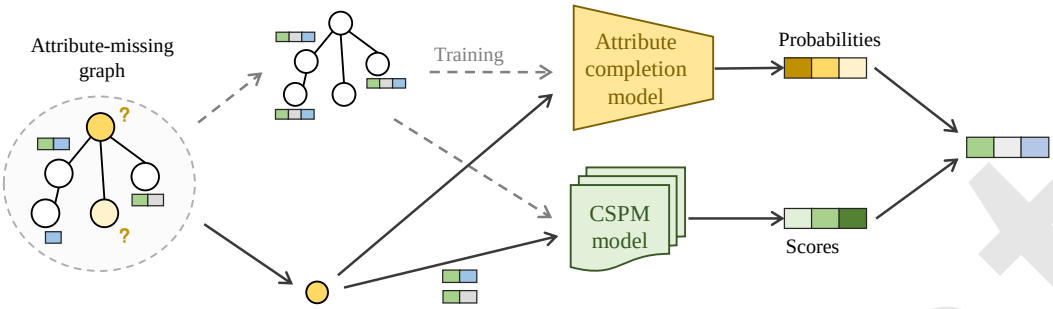


Figure 15: The whole process of the attribute completion task using CSPM

motivation for developing CSPM$_t$ is to analyze telecommunication network data to find strong correlations between device alarms. The next paragraphs give a brief overview of this application and then explain how a variant of CSPM is used to find interesting alarm patterns.

Network fault management. In a large scale telecommunication network, millions of alarms may be triggered by network devices every day, which can have a bad effect on performance and user satisfaction. Hence, it is important to investigate alarms to identify, prioritize and fix important network problems, which is known as *network fault management* [16]. For this purpose, telecommunication companies record logs about the alarms triggered by network devices. Then, maintenance workers investigate and fix problems based on these logs [16]. However, understanding and investigating a large number of alarms is not easy and it is time-consuming because faults are often caused by complex interactions between network devices. To improve fault management, recently, a promising approach is to find patterns that reveal the relationships between network alarms. The aim is to help experts understand how alarms are propagating on a network to make maintenance decisions and fix network problems more quickly. Several pattern mining algorithms have been applied for this purpose. However, they generally have two important limitations [16]: they ignore the network topology or require that the user sets algorithm parameters. To address these limitations, a variant of CSPM named CSPM$_t$ is designed and applied to analyze alarm data.

An overview of how CSPM is applied for alarm analysis is provided in Fig. 16. The output of the CSPM variant is star-shaped alarm rules. An example of alarm rules is shown in Fig. 16, where *Low signal* is the alarm in the coreset and alarms in the leafset are *Link interruption*, *Ethernet port failure*, *Link degrader* and *Microwave stripping*. This pattern can be interpreted as: if alarm *Low signal* appears, there is a high chance the leafset alarms appear nearby. Thus, the devices triggering the *Low signal* alarm should have a higher priority for maintenance operators, since that alarm is likely to be the **root cause** of other alarms. For real-world industrial applications, this approach based on a-stars mined by CSPM has the potential of greatly reducing the number of alarms presented to maintenance operators. Specifically, alarms in leafsets can be hidden to only show the most important alarms which are in the coresets. Next, how CSPM mines star-shaped alarm rules is explained in detail.

How to mine alarm patterns? A telecommunication network has two important types of data for network fault management: an alarm log and the network topology. This is the input of this application, which is represented as a dynamic attributed graph where vertices represent the devices, edges indicate the connections between devices, and all alarms triggered by devices are encoded as attribute values of vertices with timestamps indicating their appearances.

For instance, Fig. 17 (a) shows a simple alarm network observed at two timestamps t_1 and t_2 ($t_1 < t_2$). There are five vertices (v_1 to v_5), and three attributes (a_1, a_2 and a_3) representing three alarm types. For instance, at timestamp t_1, the alarm a_1 appears in device v_1.

The goal of mining alarm correlation rules is to find patterns that reveal the inner relationships and propagation properties of alarms in a network. There are two principles that help to address this issue, which were confirmed by domain experts: (1) alarms that appear together during a short time interval have a high chance of being relevant to each other; (2) it is generally unimportant to take the IDs of vertices (devices) into account to understand the relationships between alarms, while the edges (connections) between vertices are important as they imply the possible flow directions of alarms.

(1) Inverted database construction (Step 1 in Fig. 16). Based on these principles, CSPM$_t$ was designed. A key

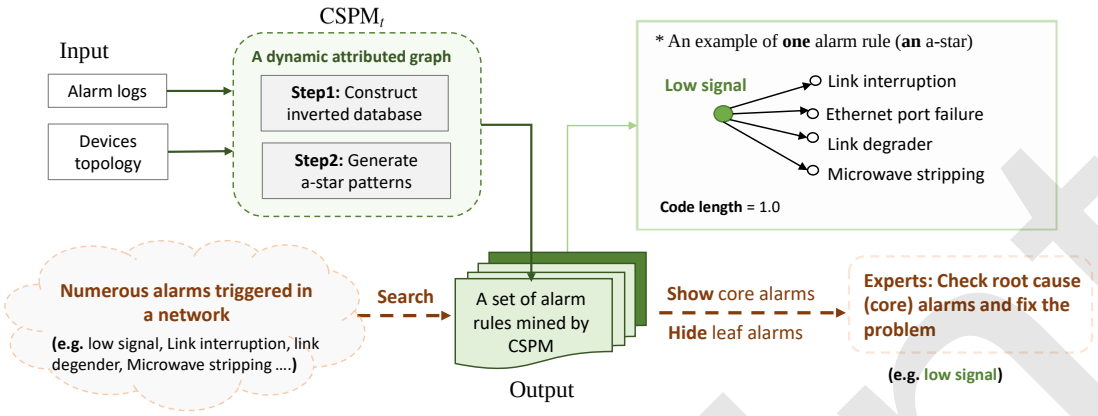


Figure 16: Overview of CSPM's application for alarm correlation analysis.

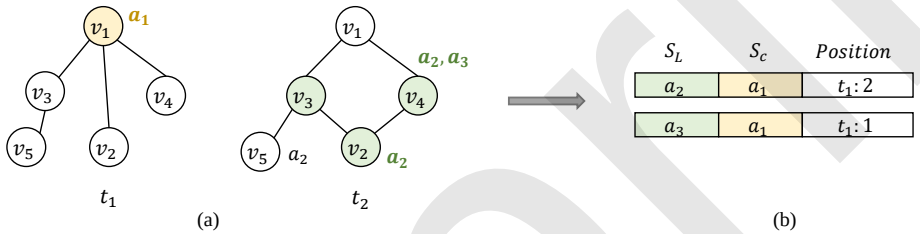


Figure 17: An (a) alarm network and (b) a part of the corresponding inverted database

modification is how the inverted database is designed. First, to weaken the influence of specific vertices, the third column *Position* is changed to store the occurrence frequencies of timestamps instead of vertices. Second, a *maxtime* threshold is introduced, which is used to determine whether some core-values and leaf-values appear together in a short time interval. In CSPM$_t$, a basic a-star is added to the inverted database, if the alarm in the leafset S_L occurs in a device that is connected to a device where alarms S_c appear. And these alarms must appear together in a time interval that is no longer than *maxtime*. For each such occurrence, a weight of 1 is added to the timestamp t_c in the *Position* set.

For instance, Fig. 17 (b) shows a part of the inverted database built from the graph of Fig. 17 (a). For the coreset $\{a_1\}$ triggered by v_1 at timestamp t_1, there are two leafsets $\{a_2\}$ and $\{a_3\}$ that appear at a nearby timestamp (i.e. appear at timestamp t_2 such that $|t_2 - t_1| < maxtime$). These two leafsets are added to the inverted database. The weight of $\{a_2\}$ is 2 since it has two occurrences near $\{a_1\}$. Note that $\{a_2\}$ in v_5 is not considered because there is no edge (network flow path) between v_1 and v_5 at timestamp t_1.

(2) A-star generation (Step 2 in Fig. 16). To find a-stars from basic a-stars, the CSPM$_t$ algorithm relies on CSPM$_w$. The inverted database of CSPM$_t$ is directly fed into CSPM$_w$. The resulting a-star patterns can then be viewed as alarm correlation rules. Such pattern can be interpreted as if a core alarm appears, there is a high chance that the leaf alarm(s) will be triggered. These rules can then be used by network maintenance experts to prioritize alarms that should be fixed first based on the hypothesis that a core alarm is likely

Experimental results for this application with real telecommunication alarm logs are presented in section 6.

5.4. Adapting CSPM for other graph types

The previous subsections have described a variant and two applications of CSPM. Besides this, CSPM is flexible and can be used to handle various types of attributed graphs. To demonstrate this, this subsection briefly explains how to adapt CSPM to handle directed graphs and weighted graphs (Fig. 18). The only modification that needs to be done is how to build the inverted database.

Mining directed graphs. In a directed graph, an edge's direction represents the direction of information flow. Recall that one of our goals is to mine relationships between attribute values, i.e., the possibilities that the leaf-values occur when the core-values are known. Thus, semantically, only the attribute values appearing at the start node of a directed edge can be regarded as core-values, and those appearing at an edge's end as leaf-values. Enforcing the above constraint for directed edges is the only modification that must be made to CSPM to handle directed graphs.

An example is shown in Fig. 18, where there are only two core-values, which are a_1 at v_1 and a_2 at v_3, since only these two nodes are the starting points of edges. More precisely, the core-value a_1 has two leaf-values a_2 and a_3, which are determined based on the directions of edges.

Mining weighted graphs. To handle weighted graphs, it is only necessary to add the real weights of the connected edge(s) between core-value and leaf-value to the weight of each a-star. Fig. 18 shows how the weights can be stored in the inverted database. We only give an example of how to add the a-stars with core-value a_1 in v_1. Similarly, it has two leaf-values, which are a_2 and a_3, respectively. The weight of a-star $(\{a_1\}, \{a_2\})$ is w_2, which is obtained from the weight of edge (v_1, v_3). And for leaf-value a_3, the weight $w_1 + w_3$ is obtained by summing up the weights of edges (v_1, v_2) and (v_1, v_4).

Others. Several other variations could be considered. For instance, CSPM can be adapted to mine only a subset of relationships between attribute values if needed.

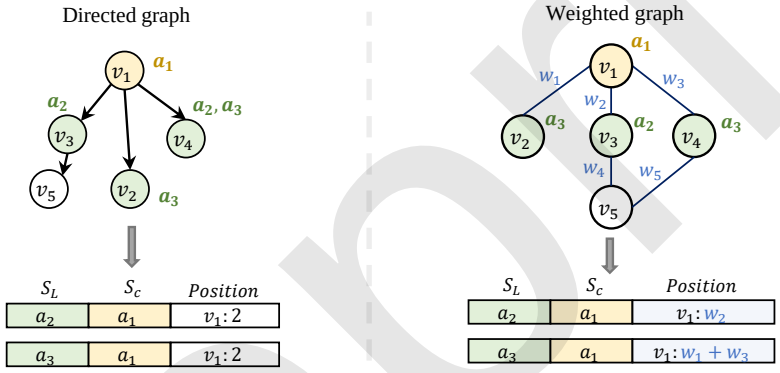


Figure 18: Inverted database construction for directed and weighted graphs.

6. Experiments

To evaluate the proposed CSPM algorithm, experiments have been performed using Python 3.7 on a computer having 32 GB of RAM and an Intel(R) Core i7-8700 CPU. This section is divided into three parts. First, Subsection 6.1 presents an evaluation of CSPM's performance and that of the optimized CSPM-Batch version in terms of runtime and pattern distribution. Moreover, the compression rate and some patterns are also analyzed. Then, to further assess pattern quality, an experiment is described in Subsection 6.2 where CSPM$_w$ is combined with state-of-the-art graph attribute completion models to improve their accuracy. Finally, Subsection 6.3 describes an experiment to verify the quality of alarm correlation rules extracted by CSPM$_t$ from telecommunication network data.

6.1. Analysis of the CSPM algorithm

Compared algorithms. Three algorithms are compared in this subsection. First, the baseline is CSPM without optimizations, which is called *CSPM-Basic*. Second, the optimized version of CSPM, called *CSPM-Batch* is compared. Third, to provide a point of reference for the experimental results, the SLIM algorithm [38] was also included in the experiments. Although SLIM ignores a graph's topology, it can be applied to find correlations between attribute values. It is interesting to compare CSPM with SLIM as both uses a compression-based approach to find patterns. However, it should be noted that SLIM's output is different from that of the two CSPM variants. We do not compare with other graph pattern mining algorithms as they are not suitable to be used as baselines for runtime analysis as the

patterns and/or focus of these algorithms are very different from CSPM. Thus, it would be unfair and meaningless to compare with them.

Datasets. Four real-life graphs having various characteristics were used, namely *DBLP*, *DBLP-Trend*, *USFlights* and *Pokec*.

The *DBLP* dataset [8] is a citation network indicating co-author relationships (edges) between researchers (vertices) during the years 2006-2010. Attributes of a person represents the conferences/journals where s/he has published. *DBLP* contains 3,464 edges, 2,723 vertices and 27 attributes representing conferences or journals related to databases such as ICDE, VLDB and JMLR.

The *DBLP-Trend* dataset has the same topology as the above DBLP dataset. The only difference is that attribute values are discrete and they are represented as trends to focus on the changes in terms of publication count between year 1990 and 1996. For example, the attribute values ICDE+, ICDE-, ICDE= indicate that the number of publications in ICDE has increased, decreased, and stayed the same since the previous year, respectively. Overall, there are 132 attribute values on this dataset.

The *USFlight* dataset [22] contains data about flights (edges) between US airports (vertices) in 2005. The graph contains 4,030 edges, 280 vertices and 24 attributes that represent trends about the states of each airport in terms of factors such as the number of arrivals/departures, average arrival/departure delay and the number of canceled flights.

The *Pokec* dataset[1] indicates friendship relationships (edges) between persons (vertices) on the Pokec social network. It contains 1,632,803 edges, 30,622,564 vertices and 10 attributes for describing users such as their music interests.

6.1.1. Effect of the Optimization

The first experiments were carried out to evaluate the benefits of using the optimized CSPM-Batch algorithm. The two versions of CSPM (CSPM-Basic and CSPM-Batch) were run on the inverted database of each dataset. Moreover, SLIM [38] was also included in this comparison.

(1) *Runtime*. In the first experiment, the influence of the proposed optimization on the runtime was evaluated by comparing the two CSPM versions.

Table 2: Dataset statistics and runtime comparison

	General statistics				Runtime (s)		
Dataset	$\|V\|$	$\|E\|$	$\|AV\|$	$\|S_c^M\|$	SLIM	CSPM-Basic	CSPM-Batch
DBLP	2,723	3,464	27	127	4.69	43.13	**2.50**
DBLP-Trend	2,723	3,464	132	271	48.69	956.61	**24.40**
USFlight	280	4,030	24	70	1.25	10.16	**0.60**
Pokec	1,632,803	30,622,564	10	914	166,678.3	–	**14,285.18**

The results are presented in Table. 2. While the left part of that table gives general statistics about the datasets, the right part indicates the runtime of the different algorithms. The columns $|V|$, $|E|$, and $|AV|$ give the number of nodes, edges, and attribute values per dataset, respectively. The column $|S_c^M|$ indicates the number of coresets in the inverted database. For runtimes, the CSPM-Basic algorithm has no value on Pokec (denoted as $-$) because it failed to terminate within 48 hours. Some important observations can be made by analyzing these results:

First, it can be seen that the baseline CSPM-Basic is less efficient than SLIM. CSPM-Basic took 10 times more time than SLIM. This result is reasonable since CSPM-Basic solves a problem that has two variables (coresets and leafsets) and it considers the graph topology, which is more complex. Moreover, CSPM-Basic merges and updates candidate pairs without using any optimization.

Second, it was found that the optimized algorithm is considerably faster than the baseline (CSPM-Basic). This is especially true for the large dataset (Pokec), where the optimization reduces the number of iterations significantly, and thus makes the algorithm more efficient.

A third observation is that the runtime of CSPM-Basic increases as the number of coresets increases. This is observed by comparing the runtime on the DBLP and DBLP-Trend datasets. In fact, CSPM-Batch optimizes the

[1] Obtained from http://snap.stanford.edu/data/web-Stanford.html

performance by generating several new leafsets during each iteration. However, more gains of pairs need to be calculated for each iteration which increases the runtime. Globally, the CSPM-Batch algorithm achieves a high performance in terms of runtime especially for large datasets.

(2) *Pattern distribution*. The CSPM-Basic and CSPM-Batch algorithms do not find exactly the same patterns as they use different merge orders. To gain insights into that difference, their pattern distributions were compared. A frequency histogram was drawn for each algorithm and dataset to visualize the number of a-stars for different code lengths. Fig. 19 shows these frequency histograms for the CSPM-Basic and CSPM-Batch algorithms on the DBLP, DBLP-Trend and USFlight datasets. The X axis indicates the code lengths of patterns, where patterns are grouped into several bins. The Y axis indicates the count rate of each bin, which is the number of patterns in the bin, divided by the total number of patterns. In an histogram, a taller bar indicates a larger number of patterns in the corresponding bin. The code length of an a-star is the sum of the code lengths of its coreset and leafset (Equation (4)). According to the MDL principle, a-stars with shorter code lengths better describe the properties of attribute values.

By observing the histograms line by line in Fig. 19 (dataset by dataset), it is found that the pattern distributions of the two algorithms are similar but not exactly the same. There are differences due to the different merge orders, but also several similarities which indicate that results from CSPM-Batch and CSPM-Basic are similar. Thus, the CSPM-Batch algorithm not only optimizes the CSPM-Basic algorithm in terms of runtime but at the same time provides a similar code length distribution for results. Overall, the optimized version of CSPM is found to provide a good trade-off between efficiency and pattern quality, when compared to CSPM-Basic.

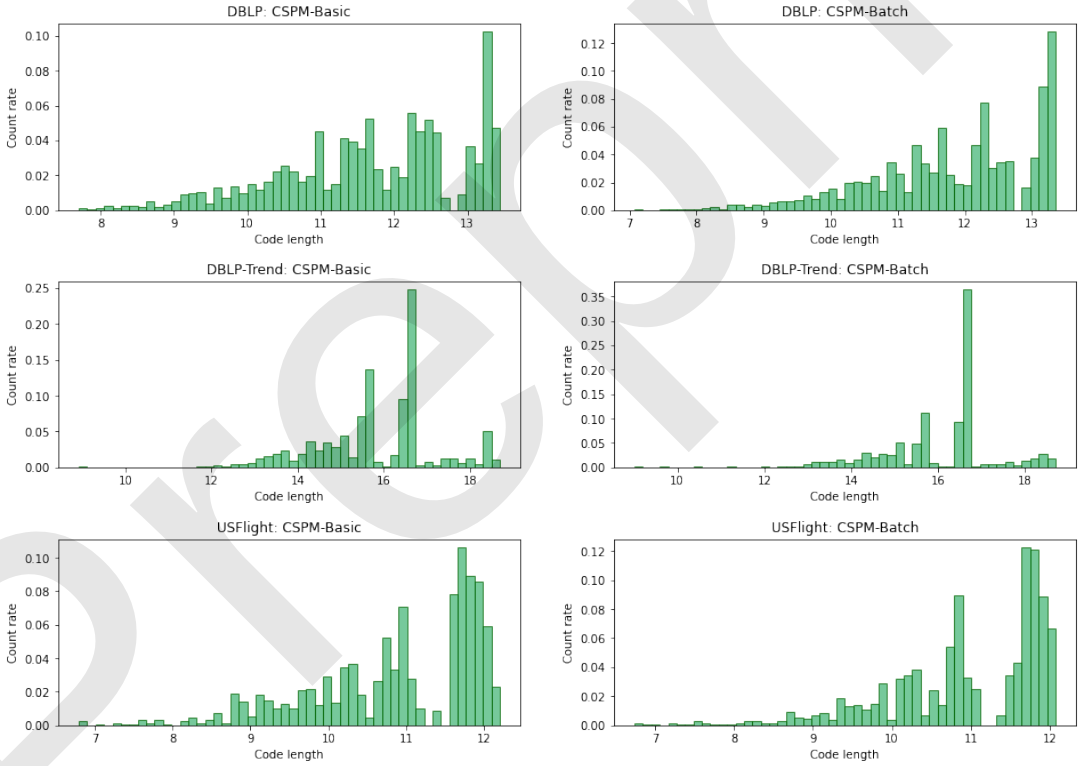


Figure 19: Distributions of patterns extracted by CSPM-Basic and CSPM-Batch on different datasets

Besides, an interesting observation is that there are few patterns having short code lengths in the histograms. This indicates that a small amount of long patterns are able to cover most of the information about attribute values, indicating that there are strong relationships between attribute values and their co-occurrence properties.

6.1.2. Compression

In a subsequent experiment, the relative compression rates obtained by different algorithms were analyzed. The notation $L\%$ is used to denote the relative compressed size calculated as $\frac{L(I,M)}{L(I)} \times 100\%$. As shown in Fig. 20, the algorithms can reach high compression rates (from about 20% to 58% on three datasets). Note that the $L\%$ measure is based on the Shannon entropy for the SLIM algorithm while it is based on a more complex metric with the conditional entropy for the CSPM algorithms. A lower compression rate implies a stronger descriptive capability of the resulting patterns.

Overall, it can be observed that there is a slight difference among the three compared algorithms for different datasets except for Pokec. For this dataset, the compression rates of CSPM-Batch is much higher than for SLIM (lower is better). Although the results obtained by SLIM seem preferable, the advantage of CSPM and its optimization is that they can find patterns that can be more interesting and meaningful as the topology is taken into account. This can be observed by analyzing the discovered patterns. The Pokec dataset has only 10 attribute values while it contains over one million vertices. Because of this, almost all possible combinations of attribute values may be seen as frequent and compressed. During the first step, since there are 10 distinct attribute values, the number of possible combinations of attribute values S_c is $2^{10} - 1 = 1{,}023$. The SLIM algorithm outputs 914 of those combinations as compressing patterns. This shows that although SLIM achieves better compression rates compared with CSPM, the results obtained by SLIM can be viewed as not very meaningful because the topology is not taken into account. The CSPM algorithms have higher relative compression rates because they take more constraints into account to mine patterns representing strong relationships between attribute values.

Comparing CSPM-Basic and CSPM-Batch, we can see that the two CSPM versions have similar relative compression rates with a slight advantages for CSPM-Basic which always achieves the lowest compression rate (lower is better). However, it is clear that the optimized CSPM-Batch algorithm is faster than CSPM-Basic. Globally, we can thus conclude that CSPM-Batch offers a good trade-off between runtime and compression.

On overall, CSPM can mine different patterns from SLIM because CSPM does not only focus on frequency but also considers the conditional entropy and topological structure. Moreover, CSPM-Batch can considerably decrease the runtime but with a slight compression loss.

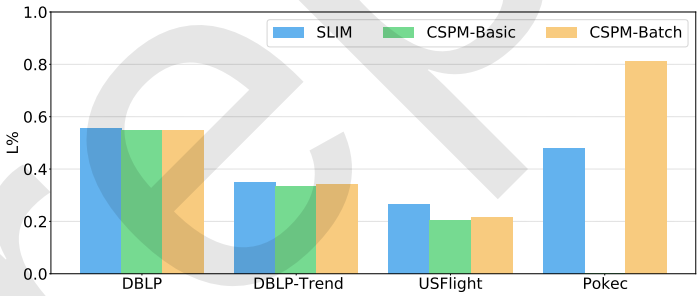


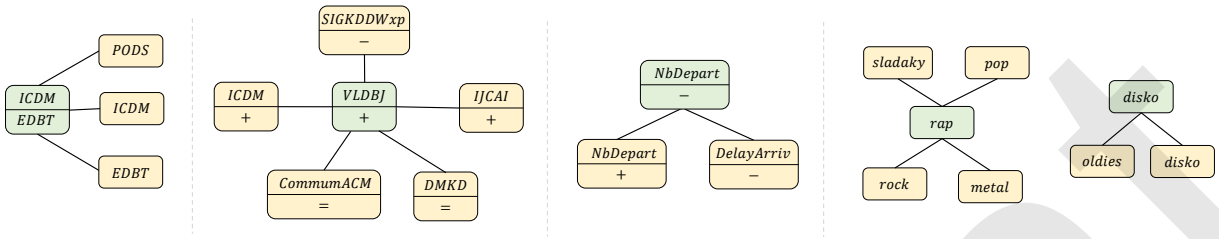
Figure 20: Comparison of the relative compression rate $L\%$. For each dataset, the colored bars from left to right denote the compression rates of SLIM, CSPM-Basic, and CSPM-Batch, respectively (smaller is better).

6.1.3. Pattern Analysis

To evaluate the quality of patterns discovered by the proposed algorithms, a manual inspection of the discovered patterns was performed for each dataset. Patterns with small code lengths are more frequent and indicate a stronger relationship between their core-values and leaf-values.

(1) *DBLP and DBLP-Trend*. Some interesting a-stars found in DBLP and DBLP-Trend are depicted in Fig. 21(a) and Fig. 21(b), respectively. It is found that there are many occurrences of the first pattern, which indicates that a researcher publishing a paper in *ICDM* and *EDBT* during a year has co-authored papers with researcher(s) who have published in *PODS*, *ICDM* and *EDBT* during the same year. This is reasonable as *PODS*, *ICDM* and *EDBT* all belong to the same research area which is database and data-mining, and generally co-authors of a paper are usually interested in the same research areas. A similar remark can be done about Fig. 21(b) for the pattern found in the DBLP-Trend dataset. The main difference is that patterns found in DBLP-Trend contain information about trends.

These patterns indicate if the number of publications in a venue is increasing (+), decreasing (-), or the same (=) for two consecutive years.



(a) DBLP (b) DBLP-Trend (c) USFlight (d) Pokec

Figure 21: Example patterns found in DBLP and DBLP-Trend

(2) *USFlight*. An example of pattern discovered in the USFlight dataset is: ({*NbDepart*−} , {*NbDepart*+ , *DelayArriv*−}). It is depicted in Fig. 21(c). This pattern indicates that if the number of departing flights from an airport is reduced, there is a high chance that some connected airports will have an increase in the number of departing flights and a decrease in the number of delayed flights.

(3) *Pokec*. Some interesting patterns were also found in the Pokec dataset about the music preferences of people from different communities, which are shown in Fig. 21(d). One of the patterns discovered by CSPM is: ({*rap*}, {*rock*, *metal*, *pop*, *sladaky*}). This pattern has a short encoding length and a high frequency which means that it greatly contributes to reducing the conditional entropy. This pattern appears to be reasonable because some music types such as rock, metal, pop, sladaky, and rap are preferred by young people. Another discovered pattern appears to show the music preferences of older people: ({*disko*}, {*oldies*, *disko*}).

6.2. Graph completion of missing attribute values

The task of graph attribute completion, which was introduced in subsection 5.2, provides an independent way of assessing the quality of the proposed a-star patterns. In this subsection, an experiment is done and results are reported for several datasets.

6.2.1. Baselines and datasets

The proposed framework for graph attribute completion was evaluated on several standard datasets, by combining CSPM with several state-of-the-art graph completion models. In this experiment, to achieve better performance, the CSPM$_w$ variant is selected to score the attributes, and the optimization of CSPM-Batch is also applied. The scores obtained by the CSPM scoring module are combined with the existing graph completing network models.

Baseline algorithms. Several models to solve the graph attribute completion task were selected. They can be categorized into four types. NeighAggre is an algorithm based on aggregation. VAE relies on auto-encoding. GCN, GraphSage, and GAT apply Graph neural networks (GNN) to focus on representative learning. Lastly, SAT uses the concept of shared-latent space. All models except NeighAggre are neural network-based.

- NeighAggre. NeighAggre [37] uses mean pooling to aggregate attributes around test nodes. It is a classical profiling algorithm which is easy to apply as it does not require any learning process. Note that one-hop neighbors are regarded as the attribute-missing node's neighbors in this experiment.

- VAE. Variational Auto-Encoder [24] is a generative method that aims at finding a latent space that could generate new data with good properties. Thus, it regularizes the prior encoding distribution to ensure the quality of restored data during training. In our case, the output dimension of the encoder in VAE is equal to the attribute values dimension. And the latent codes of the test nodes are predicted through their neighbors because of the loss of attribute values.

- GCN, GraphSage and GAT. GCN [25], GraphSage [17] and GAT [40] are representative GNN models which pay attention to graph data. To achieve our goal, the output dimensions are also the same as the attribute values'

dimensions. For attribute-missing nodes, GNN algorithms can take advantage of structural information and therefore perform well on the task.

- SAT. The structure-attribute transformer (SAT) [4] is a complex model that combines GNN and distribution matching techniques. Its starting point is that the attributes and structures are heterogeneous spaces. With the shared-latent assumption, SAT can match the distributions of attribute values and structures together. And for the graph completion task, the attribute values of the test nodes can be predicted by their structure information. Note that there are multiple ways of generating latent codes for structures based on GNNs, and two versions are selected, namely SAT(GCN) and SAT(GAT) as the baselines.

Datasets. This experiment was done on three standard graph attribute completion datasets. *Cora* [35] is a citation graph whose nodes represent papers and edges are citation links. There are a total of 2708 papers and 5278 edges. Each attribute value in the graph corresponds to a specific work token. There are 1433 attribute values and the average per vertex is 18.17. *Citeseer* [36] is also a citation network, which contains 3327 papers (vertices), 4228 citation links (edges), and 3703 words (attributes). The vertices in Citeseer have an average of 31.6 attribute values. *DBLP* was introduced in subsection 6.1. It has 4.97 attribute values on average per vertex.

6.2.2. Results for the node completion task

Because the real number of attribute values of a node is unknown, attribute values are predicted without using this information. To evaluate predictions, it was decided to choose the top K attribute values having the highest final scores as predictions (called topK), where K is a constant that is varied in the experiments. Two evaluation metrics commonly used for graph attribute completion were used, namely the Recall@K and NDCG@K metrics [4]. In information retrieval, the Recall is the fraction of the relevant documents that are successfully retrieved. For graph attribute completion, the documents are the attributes values. Thus, Recall@K is the ratio of the number of correctly predicted attribute values in topK to the number of real attribute values. The NDCG (Normalized Discounted Cumulative Gain) is an evaluation metric that takes the significance of ranking positions into account and normalizes the results using the ideal discounted cumulative gain [9]. In this study, NDCG@K is useful to assess the average ranking quality of the resulting vector for the test nodes.

For K values, the top 10, 20, 50 attribute values were used to evaluate the performance on the Cora and Citeseer datasets, while the top 3, 5, and 10 attribute values were used for the DBLP dataset. The reason for smaller K values for DBLP is that it contains much less attribute values per node on average than the other two datasets.

To prepare the experiment, 40% of the vertices with known attributes were randomly selected for training and 10% for validation. The remaining 50% of the vertices are used as test set. It was also ensured that the baselines and CSPM model are all trained/generated using the same vertices set and attribute values.

For all the learning-based methods, the Adam optimizer was applied to refine the model parameters, and the number of latent dimensions was set to 64. All GNNs use a two-layer graph convolution, where GAT uses the one-head attention. VAE uses two MLPs (multilayer preceptrons) as encoder and decoder, respectively. Besides, the learning rate, dropout rate, and the maximum iteration number are set to 0.005, 0.5 and 1000, respectively. Specifically, for the SAT algorithm, the additional hyper-parameter λ_c is set as suggested in a prior study [4]. What's more, the γ in CSPM scoring module is set to 0.4 on the Cora, Citeseer and DBLP datasets.

Table 3 shows the results for graph attribute completion. Note that there are some isolated nodes that cannot be predicted, and hence were ignored. This is the reason why some results of SAT in the experiment are not the same as in the original paper. By analyzing the results, some conclusions can be drawn. First, the SAT algorithm performs the best among all the baselines, while NeighAggre and VAE have relatively poor performance. The CSPM algorithm can boost the performance of the weaker baselines. Secondly, CSPM can increase the performance of all the baselines on the three datasets. Especially on the DBLP dataset, the average gain of using CSPM reaches up to 15.39% in Recall and 15.00% for the NDCG metric. This indicates that the designed CSPM algorithm can mine some interesting patterns that can help the baselines in correcting their predictions.

Significance test of CSPM. To gain further insights, a significance test was done. The experiments were run 10 times with random seeds ranging from 72 to 81. The one-sided Smirnov two-sample test [18] was applied, which is widely used to check whether 2 samples are drawn from the same continuous distribution, to check the significance

Table 3: Profiling evaluation for node attribute completion

	Method \ Metric	Recall@10	Recall@20	Recall@50	NDCG@10	NDCG@20	NDCG@50
Cora	NeighAggre	0.0895	0.1396	0.1944	0.1203	0.1534	0.1832
	CSPM+NeighAggre	**0.1170**	**0.1717**	**0.2851**	**0.1601**	**0.1968**	**0.2563**
	VAE	0.0875	0.1215	0.2069	0.1219	0.1451	0.1901
	CSPM+VAE	**0.0979**	**0.1469**	**0.2537**	**0.1338**	**0.1667**	**0.2227**
	GCN	0.1261	0.1782	0.2930	0.1727	0.2075	0.2680
	CSPM+GCN	**0.1329**	**0.1945**	**0.3256**	**0.1800**	**0.2212**	**0.2900**
	GraphSage	0.1224	0.1711	0.2790	0.1689	0.2018	0.2586
	CSPM+GraphSage	**0.1298**	**0.1882**	**0.3114**	**0.1759**	**0.2150**	**0.2801**
	GAT	0.1281	0.1811	0.2982	0.1747	0.2108	0.2722
	CSPM+GAT	**0.1333**	**0.1938**	**0.3225**	**0.1807**	**0.2218**	**0.2895**
	SAT(GCN)	0.1461	0.2132	0.3375	0.2069	0.2514	0.3177
	CSPM+SAT(GCN)	**0.1568**	**0.2276**	**0.3540**	**0.2173**	**0.2646**	**0.3311**
	SAT(GAT)	0.1602	0.2302	0.3591	0.2239	0.2707	0.3384
	CSPM+SAT(GAT)	**0.1647**	**0.2335**	**0.3609**	**0.2281**	**0.2740**	**0.3408**
	Average improvement(%)	+9.75	+11.18	+15.08	+8.79	+9.72	+12.12
Citeseer	NeighAggre	0.0410	0.0737	0.1242	0.0658	0.0932	0.1274
	CSPM+NeighAggre	**0.0510**	**0.0870**	**0.1678**	**0.0792**	**0.1096**	**0.1627**
	VAE	0.0369	0.0662	0.1292	0.0580	0.0825	0.1288
	CSPM+VAE	**0.0406**	**0.0736**	**0.1435**	**0.0644**	**0.0920**	**0.1378**
	GCN	0.0613	0.1060	0.2000	0.1000	0.1370	0.1990
	CSPM+GCN	**0.0641**	**0.1104**	**0.2050**	**0.1039**	**0.1426**	**0.2045**
	GraphSage	0.0561	0.1008	0.1939	0.0879	0.1250	0.1858
	CSPM+GraphSage	**0.0592**	**0.1053**	**0.2011**	**0.0922**	**0.1357**	**0.1932**
	GAT	0.0535	0.0988	0.1958	0.0860	0.1238	0.1871
	CSPM+GAT	**0.0602**	**0.1039**	**0.2024**	**0.0954**	**0.1319**	**0.1962**
	SAT(GCN)	0.0657	0.1125	0.2103	0.1110	0.1501	0.2142
	CSPM+SAT(GCN)	**0.0699**	**0.1161**	**0.2118**	**0.1165**	**0.1552**	**0.2180**
	SAT(GAT)	0.0710	0.1195	0.2183	0.1186	0.1591	0.2237
	CSPM+SAT(GAT)	**0.0714**	**0.1200**	**0.2188**	**0.1193**	**0.1599**	**0.2246**
	Average improvement(%)	+9.14	+6.66	+8.10	+8.10	+7.46	+6.93
	Method \ Metric	Recall@3	Recall@5	Recall@10	NDCG@3	NDCG@5	NDCG@10
DBLP	NeighAggre	0.2022	0.2648	0.3755	0.2538	0.3000	0.3631
	CSPM+NeighAggre	**0.3170**	**0.3981**	**0.5452**	**0.3955**	**0.4511**	**0.5320**
	VAE	0.2511	0.3119	0.4728	0.3071	0.3494	0.4370
	CSPM+VAE	**0.3418**	**0.4269**	**0.6048**	**0.4201**	**0.4784**	**0.5742**
	GCN	0.3994	0.4832	0.6657	0.4943	0.5523	0.6510
	CSPM+GCN	**0.4035**	**0.4912**	**0.6764**	**0.4998**	**0.5600**	**0.6603**
	GraphSage	0.3445	0.4258	0.6069	0.4227	0.4793	0.5760
	CSPM+GraphSage	**0.3843**	**0.4797**	**0.6725**	**0.4743**	**0.5340**	**0.6430**
	GAT	0.3964	0.4910	0.6574	0.4902	0.5539	0.6419
	CSPM+GAT	**0.3970**	**0.4940**	**0.6659**	**0.4912**	**0.5566**	**0.6483**
	SAT(GCN)	0.4890	0.6418	0.8227	0.5708	0.6694	0.7629
	CSPM+SAT(GCN)	**0.4969**	**0.6470**	**0.8248**	**0.5777**	**0.6746**	**0.7671**
	SAT(GAT)	0.5269	0.6697	0.8530	0.6038	0.6974	0.7919
	CSPM+SAT(GAT)	**0.5296**	**0.6723**	**0.8560**	**0.6061**	**0.6995**	**0.7941**
	Average improvement(%)	+15.39	+14.76	+12.49	+15.00	+14.52	+13.26

Table 4: Significance analysis of CSPM on Citeseer

Method \ Metric	Recall@10	Recall@20	Recall@50	NDCG@10	NDCG@20	NDCG@50
SAT	0.0641 ± 0.10	0.1090 ± 0.13	0.2076 ± 0.16	0.1083 ± 0.19	0.1458 ± 0.19	0.2104 ± 0.20
CSPM+SAT	0.0671 ± 0.10	0.1134 ± 0.16	0.2135 ± 0.17	0.1130 ± 0.17	0.1516 ± 0.19	0.2173 ± 0.19
p-value	3.12e-5	3.12e-5	3.12e-5	3.12e-5	3.12e-5	1.42e-6

* The unit of standard deviation is e-2

between the proposal (i.e. CSPM+SAT) and the second best model (i.e. SAT). The results on Citeseer[2], are summarized in Table 4. It is found that the p-values are less than 0.05 for all metrics, which indicates that the resulting distributions of CSPM + SAT are significantly larger than those of SAT. Therefore, a conclusion can be drawn that the improvements provided by CSPM over the best baseline are statistically significant.

In a nutshell, the CSPM method is found to improve multiple methods for the graph attribute completion task, which shows that the patterns mined by CSPM are meaningful and good. It is also interesting that CSPM can be applied in parallel to the training of the baselines. Besides, the property that CSPM is mainly parameter-free makes it more robust and easy to apply on various datasets.

6.2.3. *Hyper-parameter sensitivity analysis*

Another experiment was done to assess the impact of the hyper-parameter γ introduced in subsection 5.2. Recall that γ is the hyper-parameter that determines the relative importance of the code length of a coreset and leafset, i.e. $L(S_{code}) = \gamma \times L(Code_L) + (1-\gamma) \times L(Code_c)$. Here, the code length of coreset $L(Code_c)$ indicates the occurrence frequency of the a-star pattern, and $L(Code_L)$ represents the relationship between attribute values in the coreset and leafset. A shorter code length is better which means that the pattern summarizes more information of the original database. We applied CSPM to the three baselines (VAE, NeighAggre and SAT) by changing the value of γ from 0 to 1. Fig. 22 shows the recall@K and NDCG@K for different γ values.

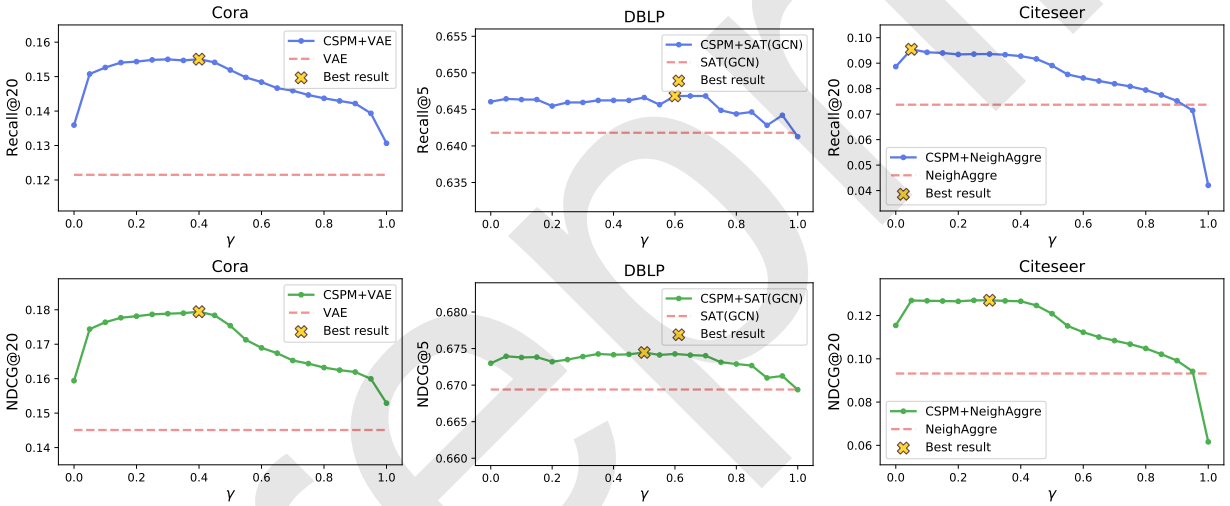


Figure 22: Sensitivity analysis of hyper-parameter γ

It is found that γ has a slight impact on results for both the recall and NDCG metrics. Overall, changes resulting from varying γ for the two metrics are similar in that almost all the results for different γ values help to improve the performance of the different attribute completion models. Besides, the results are relatively lower when only considering the coreset ($\gamma = 0$) or leafset ($\gamma = 1$). Especially for Citeseer, the results even get much worse than baselines when merely the code length of leafsets (i.e. relationship) is considered. Thus, both the information of coresets and leafsets positively contribute to the final results. For the value selection of γ, the value corresponding to the optimal results depends on the properties of the datasets. However, it can be observed that the results are always close to the optimal (shown in yellow) when the value is around 0.5.

6.2.4. *Scalability of CSPM*

This subsection increases the training set size for the three datasets to understand how CSPM scales. Specifically, we use 40%, 60% and 80% nodes of each input attributed graph as the training set. Other settings are the same with those for a 40% training set which have been introduced in subsection 6.2.2. Fig. 23 depicts some representative

[2]For brevity, we only list the results of Citeseer as it shows the smallest average improvement.

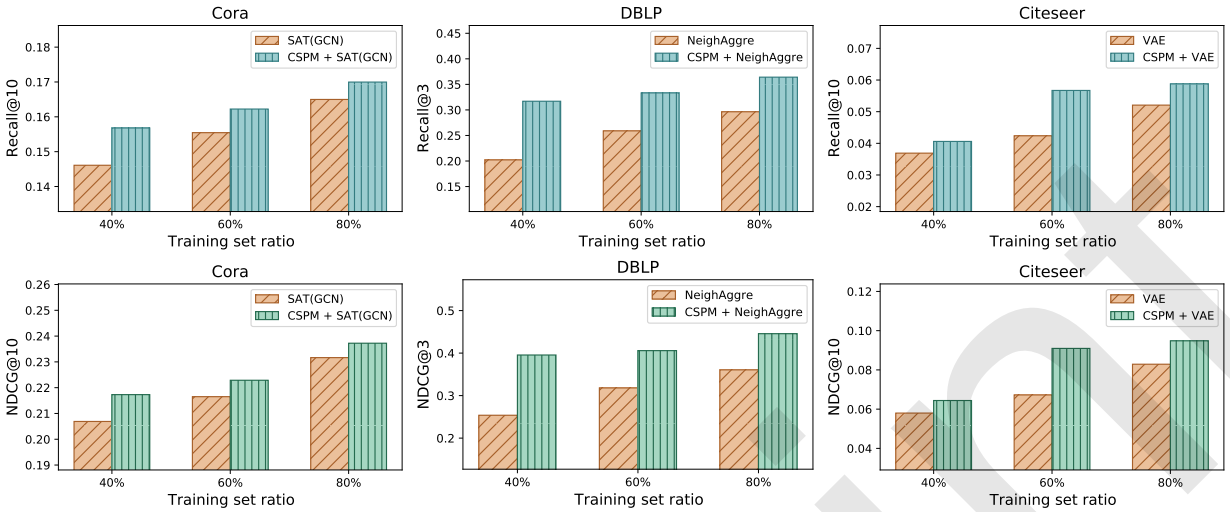


Figure 23: Scalability analysis of CSPM

results. It can be observed that regardless of the training set size, CSPM always boosts the results of the baselines. The second observation is that all the results get better as the training set size increases. Interestingly, CSPM+NeighAggre with 40% training set on DBLP even outperforms NeighAggre with a 80% training set. This is also observed for CSPM+VAE (60%) and VAE (80%) for the Citeseer dataset.

6.3. Alarm correlation analysis

This subsection further verifies the effectiveness of the CSPM algorithm by applying it to a practical industrial application of alarm analysis, mentioned in Subsection 5.3.

Dataset. The dataset is collected from the telecommunication network of a metropolitan city, which contains triggered alarms from the 12^{th} to the 16^{th} April, 2019. In total, there are more than 6 million alarms, categorized into 300 types. The topology of the network has not changed in the given time period.

Compared algorithms. AABD [42] and ACOR [16] are two algorithms designed to address the alarm correlation problem of telecommunication networks. AABD mines patterns in alarm logs using sequential pattern mining [39]. Then the patterns are used to generate rules by relying on a knowledge base which was provided by domain experts. As parameter-free pattern mining algorithms are not utilized to generate patterns in AABD, the quality of results is sensitive to parameter values. What's more, AABD ignores the actual connections between the devices where alarms are triggered. But the topology is important for the propagation of information such as alarms on a network. To solve the above problems, ACOR models alarm data as a dynamic attributed graph, and then extracts alarm pairs having a high correlation using a specially designed measure that is easy to calculate.

The AABD and ACOR algorithms can evaluate and rank all possible pairwise alarm rules, and they are chosen as the baselines to evaluate CSPM$_t$ (Subsection 5.3). A pairwise rule contains two parts: a parent alarm and children alarms where the parent alarm is likely to be the root cause of the children alarms.

Comparison of coverage ratio. For the given 300 types of alarms, AABD finds a total of 121 valid pair-rules [42] verified by domain experts. The comparison experiment was done by checking whether the rules found by AABD could be rediscovered by ACOR and CSPM$_t$. The coverage ratio [16] is used as the metric to evaluate the correlation rules, which was used for the ACOR algorithm. It is defined as $coverage = |A \cap B| \,/\, |B|$ where A is the set of valid rules and B is the set of rules found by CSPM or ACOR. A high coverage ratio indicates that many rules of the proposed framework are valid. Hereafter, the coverage ratio for topK means the ratio of valid rules in the K rules that have the highest correlation scores found by CSPM or ACOR.

Note that the alarm patterns extracted by CSPM$_t$ are star-shaped, while the compared algorithm extracts patterns that have a rule format. Thus, to compare the results, the a-stars mined by CSPM$_t$ are split as pairs. All the core-values play the roles of the parent alarms in rules, and leaf-values represent the children alarms. The comparison of

the coverage ratio is shown in Fig. 24. The code length of each pair is equal to the shortest code length of its a-star where smaller code lengths mean that alarm pairs have a higher correlation.

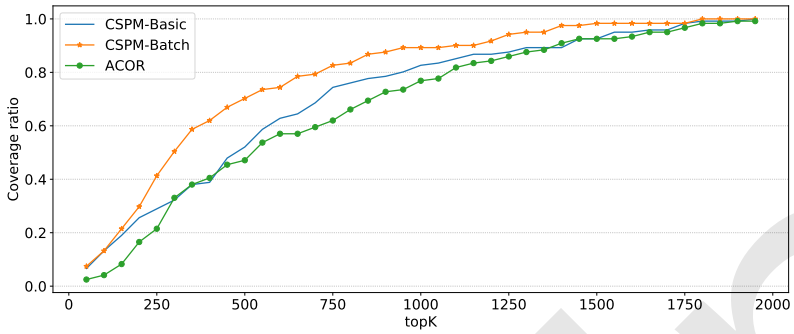


Figure 24: The coverage ratio of ACOR and CSPM$_t$ for alarm correlation analysis

It can be observed that the coverage ratio increases as more rules are selected, and finally all the rules labelled by human experts are found. Compared to the ACOR algorithm, as it is expected, the valid rules mined by either CSPM-Basic or CSPM-Batch are ranked higher as CSPM evaluates all the rules simultaneously by the MDL principle. Interestingly, due to the different merge orders, the valid rules in the selected topK of CSPM-Basic and CSPM-Batch are different, while CSPM-Batch is more effective for uncovering alarm correlations.

7. Conclusion

To discover patterns in attributed graph that can reveal complex structures between attribute values, this paper has proposed a novel compression-based algorithm, named CSPM. It identifies all star-shaped attribute patterns that best compress the data according to the conditional entropy and minimum description length. This has the advantage of being parameter-free and of providing a set of patterns that capture strong relationships between node attribute values. Furthermore, an optimized version of CSPM called CSPM-Batch was presented to reduce the runtime. CSPM is flexible and can be applied to many application scenarios. For example, this paper has introduced two variants of CSPM by making slight adjustments, called CSPM$_w$ and CSPM$_t$. While CSPM$_w$ adds weights to the inverted database to mine more precise patterns, CSPM$_t$ extends CSPM to deal with dynamic attributed graph data.

Experiments on large graphs have shown that CSPM can not only identify interesting patterns but also has excellent performance. In particular, it was found that CSPM-Batch can considerably decrease runtime for a slight compression loss. The analysis of the pattern distribution also revealed that CSPM-Batch can provide a good trade-off between pattern quality and runtime. Two other experiments based on real world application scenarios were done to evaluate the quality of the patterns obtained by CSPM. Results showed that it can help improve the performance of graph attribute completion models and find valid alarm rules with higher rank in data from a large telecommunication network, when compared to the state-of-art industrial methods. In summary, CSPM is a parameter-free, efficient and robust algorithm to find patterns of attributes values, and can be used in multiple applications.

This research opens several possibilities for future work. In particular, we plan to extend the proposed algorithm to mine compressing patterns in graphs with multiple attributes per edge. Another important research direction is to develop a parallel and distributed version of the algorithm to process very large databases.

References

[1] Martin Atzmueller, Henry Soldano, Guillaume Santini, and Dominique Bouthinon. Minerlsd: Efficient local pattern mining on attributed graphs. pages 219–228, 2018.

[2] Francesco Bariatti, Peggy Cellier, and Sébastien Ferré. Graphmdl: Graph pattern selection based on minimum description length. In *International Symposium on Intelligent Data Analysis*, pages 54–66. Springer, 2020.

[3] Roel Bertens, Jilles Vreeken, and Arno Siebes. Keeping it short and simple: Summarising complex event sequences with multivariate patterns. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 735–744, 2016.

[4] Xu Chen, Siheng Chen, Jiangchao Yao, Huangjie Zheng, Ya Zhang, and Ivor W Tsang. Learning on attribute-missing graphs. *arXiv e-prints*, pages arXiv–2011, 2020.

[5] Zhi Cheng, Frédéric Flouvat, and Nazha Selmaoui-Folcher. Mining recurrent patterns in a dynamic attributed graph. In *Proc. of 21st Pacific-Asia Conf. on Knowledge Discovery and Data Mining*, pages 631–643. Springer, 2017.

[6] Thomas M Cover. *Elements of information theory*. John Wiley & Sons, 1999.

[7] J. Cuppers and J. Vreeken. Just wait for it... mining sequential patterns with reliable prediction delays. In *2020 IEEE International Conference on Data Mining (ICDM)*, 2020.

[8] Elise Desmier, Marc Plantevit, Céline Robardet, and Jean-François Boulicaut. Cohesive co-evolution patterns in dynamic attributed graphs. In *International Conference on Discovery Science*, pages 110–124. Springer, 2012.

[9] Consistent Distinguishability. A theoretical analysis of normalized discounted cumulative gain (ndcg) ranking measures. 2013.

[10] Batjargal Dolgorsuren, Kifayat Ullah Khan, Mostofa Kamal Rasel, and Youngkoo Lee. Starzip: Streaming graph compression technique for data archiving. *IEEE Access*, 7:38020–38034, 2019.

[11] Raïssa Yapan Dougnon, Philippe Fournier-Viger, Jerry Chun-Wei Lin, and Roger Nkambou. Inferring social network user profiles using a partial social graph. *Journal of intelligent information systems*, 47(2):313–344, 2016.

[12] P. Fournier-Viger, G. He, C. Cheng, J. Li, M. Zhou, Jerry Chun-Wei Lin, and U. Yun. A survey of pattern mining in dynamic graphs. *WIREs Data Mining and Knowledge Discovery*, 2020.

[13] Philippe Fournier-Viger, Chao Cheng, Zhi Cheng, Jerry Chun-Wei Lin, and Nazha Selmaoui-Folcher. Mining significant trend sequences in dynamic attributed graphs. *Knowledge-Based Systems*, 182:104797, 2019.

[14] Philippe Fournier-Viger, Chao Cheng, Jerry Chun-Wei Lin, Unil Yun, and R Uday Kiran. Tkg: Efficient mining of top-k frequent subgraphs. In *Proc. 7th International Conference on Big Data Analytics*, pages 209–226. Springer, 2019.

[15] Philippe Fournier-Viger, Ganghuan He, Jerry Chun-Wei Lin, and Heitor Murilo Gomes. Mining attribute evolution rules in dynamic attributed graphs. In *International Conference on Big Data Analytics and Knowledge Discovery*, pages 167–182. Springer, 2020.

[16] Philippe Fournier-Viger, Ganghuan He, Min Zhou, Mourad Nouioua, and Jiahong Liu. Discovering alarm correlation rules for network fault management. In *International Conference on Service-Oriented Computing*, pages 228–239. Springer, 2020.

[17] William L Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs. *arXiv preprint arXiv:1706.02216*, 2017.

[18] John L Hodges. The significance probability of the smirnov two-sample test. *Arkiv för Matematik*, 3(5):469–486, 1958.

[19] Chuntao Jiang, Frans Coenen, and Michele Zito. A survey of frequent subgraph mining algorithms. *Knowledge Engineering Review*, 28:75–105, 2013.

[20] Aída Jiménez, Fernando Berzal, and Juan-Carlos Cubero. Frequent tree pattern mining: A survey. *Intelligent Data Analysis*, 14(6):603–622, 2010.

[21] Mehdi Kargar, Morteza Zihayat, and Jaroslaw Szlichta. Mining and exploration of attributed graphs: theory and applications. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, pages 397–398, 2019.

[22] Mehdi Kaytoue, Yoann Pitarch, Marc Plantevit, and Céline Robardet. Triggering patterns of topology changes in dynamic graphs. In *2014 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM 2014)*, pages 158–165. IEEE, 2014.

[23] Kifayat Ullah Khan, Waqas Nawaz, and Youngkoo Lee. Set-based unified approach for attributed graph summarization. pages 378–385, 2014.

[24] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[25] Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

[26] Danai Koutra, U Kang, Jilles Vreeken, and Christos Faloutsos. Vog: Summarizing and understanding large graphs. pages 91–99, 2014.

[27] Hoang Thanh Lam, Fabian Mörchen, Dmitriy Fradkin, and Toon Calders. Mining compressing sequential patterns. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 7(1):34–52, 2014.

[28] Ming Li, Paul Vitányi, et al. *An introduction to Kolmogorov complexity and its applications*, volume 3. Springer, 2008.

[29] Y. Lim, U. Kang, and C. Faloutsos. Slashburn: Graph compression and mining beyond caveman communities. *IEEE Transactions on Knowledge and Data Engineering*, 26(12):3077–3089, 2014.

[30] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM Computing Surveys*, 51(3):62, 2018.

[31] Zhaoyu Lou, Jiaxuan You, Chengtao Wen, Arquimedes Canedo, Jure Leskovec, et al. Neural subgraph matching. *arXiv preprint arXiv:2007.03092*, 2020.

[32] Wuman Luo, Haoyu Tan, Lei Chen, and Lionel M Ni. Finding time period-based most frequent path in big trajectory data. In *Proceedings of the 2013 ACM SIGMOD international conference on management of data*, pages 713–724, 2013.

[33] Claude Pasquier, Jérémy Sanhes, Frédéric Flouvat, and Nazha Selmaoui-Folcher. Frequent pattern mining in attributed trees: algorithms and applications. *Knowledge and Information Systems*, 46(3):491–514, 2016.

[34] Leonardo Pellegrina and Fabio Vandin. Efficient mining of the most significant patterns with permutation testing. *Data Mining and Knowledge Discovery*, 34:1201–1234, 2020.

[35] W. M. Pottenger. Social networking on a website with topic-based data sharing. 2009.

[36] S. Prithviraj, N. Galileo, B. Mustafa, G. Lise, G. Brian, and E. R. Tina. Collective classification of network data. *Ai Magazine*, 29(3):93–, 2008.
[37] Özgür Şimşek and David Jensen. Navigating networks by using homophily and degree. *Proceedings of the National Academy of Sciences*, 105(35):12758–12762, 2008.
[38] Koen Smets and Jilles Vreeken. Slim: Directly mining descriptive patterns. In *Proceedings of the 2012 SIAM international conference on data mining*, pages 236–247. SIAM, 2012.
[39] Tin Truong, Hai Duong, Bac Le, and Philippe Fournier-Viger. Fmaxclohusm: An efficient algorithm for mining frequent closed and maximal high utility sequences. *Engineering Applications of Artificial Intelligence*, 85:1–20, 2019.
[40] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.
[41] Jilles Vreeken, Matthijs Van Leeuwen, and Arno Siebes. Krimp: mining itemsets that compress. *Data Mining and Knowledge Discovery*, 23(1):169–214, 2011.
[42] Jiantao Wang, Caifeng He, Yijun Liu, Guangjian Tian, Ivy Peng, Jia Xing, Xiangbing Ruan, Haoran Xie, and Fu Lee Wang. Efficient alarm behavior analytics for telecom networks. *Information Sciences*, 402:1–14, 2017.
[43] Xifeng Yan and Jiawei Han. gspan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.*, pages 721–724, 2002.