Attributed graph mining in the presence of automorphism

Claude Pasquier, Frédéric Flouvat, Jérémy Sanhes, Nazha Selmaoui-Folcher

Knowl Inf Syst
DOI 10.1007/s10115-016-0953-9

REGULAR PAPER

Attributed graph mining in the presence of automorphism

Claude Pasquier[1,2,3] · **Frédéric Flouvat**[3] · **Jérémy Sanhes**[3] · **Nazha Selmaoui-Folcher**[3]

Received: 24 June 2015 / Revised: 21 March 2016 / Accepted: 19 April 2016

Abstract Attributed directed graphs are directed graphs in which nodes are associated with sets of attributes. Many data from the real world can be naturally represented by this type of structure, but few algorithms are able to directly handle these complex graphs. Mining attributed graphs is a difficult task because it requires combining the exploration of the graph structure with the identification of frequent itemsets. In addition, due to the combinatorics on itemsets, subgraph isomorphisms (which have a significant impact on performances) are much more numerous than in labeled graphs. In this paper, we present a new data mining method that can extract frequent patterns from one or more directed attributed graphs. We show how to reduce the combinatorial explosion induced by subgraph isomorphisms thanks to an appropriate processing of automorphic patterns.

Keywords Attributed graph · Frequent pattern mining · Automorphism · Structure mining · Itemset mining

1 Introduction

Directed graphs are well suited to model complex structures present in the real world. Gene regulatory networks, for example, are directed graphs in which nodes represent genes and edges represent regulatory influences. The World Wide Web is well modeled by a directed graph in which nodes are pages and edges are hyperlinks. Email communications; social graphs in which an individual can follow the activities of other people; citation networks in which an article cites other articles are also ideally represented by directed graphs.

✉ Claude Pasquier
claude.pasquier@unice.fr

1 Univ. Nice Sophia Antipolis, I3S, UMR 7271, 06900 Sophia Antipolis, France

2 CNRS, I3S, UMR 7271, 06900 Sophia Antipolis, France

3 Multidisciplinary Research Team on Material and Environment (PPME), University of New Caledonia, 98851 Nouméa, New Caledonia

Because of these numerous applications, graphs are extensively studied in graph theory, and, more recently, in the context of data mining. Most researches focus either on unlabeled graphs or on graphs whose nodes are associated with a unique label. In a social graph, for example, labels can be the ID of individuals; in a gene regulatory network, they can represent genes' names; in a Web graph, nodes are often labeled with pages' URL, etc.

However, in many applications, objects (represented by nodes) can be associated with many features. Individuals in social graphs have many characteristics, as genes in regulatory networks. In citation networks, articles are associated with many data like keywords, authors, publication dates and patents. In these data, each characteristic associated with an object is an attribute of the corresponding node. Graphs in which nodes are annotated with sets of attributes (or itemsets) are named attributed graphs, and up to now, only few studies are devoted to their analysis.[1]

One of the classical tasks when analyzing graph data is the discovery of frequent subgraphs. The interest of such patterns is to highlight how node labels are often organized. Mining attributed graphs allows the identification of structural patterns, but also highlights the relationship between nodes' attributes.

Two main reasons make the mining of attributed graphs very difficult. First, it is necessary to combine the exploration of the graph structure with the identification of frequent itemsets associated with nodes. The second reason is that, as for labeled graphs, the performance of the mining process is highly affected by the cost of subgraph isomorphism tests [22]. As noted by [22], in sparse labeled graphs with diverse labels, the number of subgraph isomorphisms remains pretty small. However, it is no longer the case in attributed graphs where the combinatorics of attributes often implies a large number of subgraph isomorphism tests.

The number of subgraphs with automorphisms is also much higher. As highlighted by [15], the presence of automorphism is problematic for all labeled graph mining algorithms. Using algorithms based on pattern extension, each automorphism in a subgraph of size k can lead to a different candidate pattern of size $k + 1$. In the worst case, when the processed subgraph is an unlabeled clique (or a clique with the same label attached to all nodes), the number of automorphisms is $k!$.

However, up to now, little attention is given to this problem. Indeed, in labeled graphs, nodes are associated with unique labels, which are, in general, manifold. Therefore, it is not very frequent to have the same label associated with many nodes in a subgraph. This is different in attributed graphs because entities represented by nodes are characterized by multiple attributes, and some of them are very common. Thus, it greatly increases the number of possible automorphisms. In a social graph, for example, a given individual is often connected with a group of other individuals sharing several characteristics (age group, hobbies, social class, musical taste, etc.).

The key contributions of our work are the following: (1) We present the problem of mining frequent attributed subgraphs in one or more directed attributed graphs, and (2) we show how the identification of frequent patterns can be achieved by exploring the spanning trees of the graphs. (3) We define a new canonical form adapted to the exploration of the search space, and (4) we show how to extend such patterns to generate attributed subgraphs. (5) We analyze the particular case of automorphic patterns and show how the combinatorial explosion caused by subgraph isomorphisms can be significantly limited. Before concluding, (6) we present the

[1] Graphs with labeled edges can always be transformed into graphs with only labels on nodes (using, e.g., the method proposed by [11]). For this reason, we only consider attributed nodes in our study.

results obtained by analyzing several artificial and real datasets with the program AADAGE (Automorphism Aware Directed Attributed Graph Explorer) that we have developed.

2 Related works

Many algorithms have been proposed for mining labeled graphs (with one item associated with each node). Almost all proposed solutions are adaptations of frequent itemset mining. They explore the solution space by enlarging candidate subgraphs and eliminating infrequent patterns [1,12]. These methods differ in their way to explore the search space and in their pruning strategies to avoid the generation of redundant solutions. Indeed, unlike itemset mining where it is easy to generate all possible solutions in a unique way, in graph mining, it is very common to build the same pattern several times.

One of the privileged ways to avoid repeatedly exploring the same part of the search space is to define a systematic and orderly manner to add edges and nodes to candidate patterns based on a canonical representation of subgraphs. The best-known methods for generating all candidate subgraphs are depth-first search (gSpan [22], Closegraph [23]) and breadth-first search (MoSS/MoFa [5]).

The underlying idea behind the use of canonical forms is to extend only one solution in each isomorphism class of subgraphs. In graph theory, there are many ways to define the canonical form of a graph. A commonly used method is to represent a graph as an adjacency matrix, to swap rows and columns in order to generate all isomorphic graphs and to choose one of the matrices as the canonical form. In data mining, one avoids as much as possible to perform complete (and costly) isomorphism tests for each new candidate. The canonical representation is chosen such that it is easy (and effective) to test whether a pattern has already been generated. The most commonly used solution is to create a sequence of all edges contained in a subgraph in the order in which they were added [4,22].

The use of canonical forms, if it avoids exploring several times the same subgraph, does not obviate the need to identify all the possible ways to generate patterns. Thus, if the nodes are ordered in the lexicographical order of their labels, the pattern in Fig. 1b is canonical, and it appears 4 times in the initial graph presented in Fig. 1a. The patterns of Fig. 1c, d, which are also canonical, can be generated in 12 different ways (in the initial graph, there are 12 possible ways to choose 2 *b* nodes out of the 4 children labeled *b* or *a*). It is necessary to consider all these forms because the chosen configuration influences the future extensions of the pattern; for example, whether or not to extend each of the nodes labeled *b* with a node labeled *c*. The patterns with many subgraph isomorphisms in input data are challenging for all existing algorithms because of the problem of subgraph isomorphism, which is itself NP-complete. Patterns presented in Fig. 1c, d, which have automorphisms, generate the biggest problems and contribute significantly to performance degradation of the algorithms, because the number of subgraph isomorphisms with the initial graph is increased.

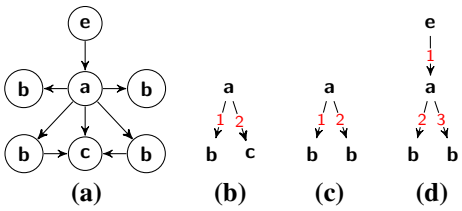

Fig. 1 Example of a directed labeled graph (**a**) and 3 patterns (**a**, **b**, **c** and **d**) present in the graph (the numbers displayed on the edges show their creation order)

In the case of sparse labeled graphs, this number remains relatively small. In attributed graphs, however, since many attributes are associated with nodes, the probability of observing subgraphs sharing one or more identical attributes is greatly increased. One way to circumvent it is to remove from the graph the most frequent items, but we are depriving ourselves of potentially interesting patterns.

Some studies deal specifically with the analysis of graphs in which the nodes are associated with itemsets as in the works of [8] or [18]. The issues addressed in these studies are different from ours in the sense that they deal with the analysis of undirected graphs and search for subgraphs sharing the same itemsets.

In our work, we seek to identify the recurrent presence of certain attributes associated with connected nodes. In its objectives, our work is closer to sequence mining [2,3] or attributed tree mining [19]. In our previous work [19], we proposed a method to mine substructures reflecting the evolution of itemsets. However, our algorithm was dedicated to the analysis of datasets that can be represented as attributed trees.

3 Basic concepts and problem statement

In this section, we give basic definitions and concepts, and then we introduce the problem of attributed graph mining.

3.1 Preliminaries

Let $\mathcal{I} = \{i_1, i_2, .., i_n\}$ be a set of items. An **itemset** is a subset of $\mathcal{I}$. The items belonging to an itemset are sorted by lexicographical order and are accessed by their index (e.g., $\mathcal{I}_2$ to access the second item of $\mathcal{I}$).

A **directed attributed graph** $G = (V, E, \lambda)$ on a set of items $\mathcal{I}$ consists of a set of vertices or nodes V, a set of edges $E \subseteq \{(u, v) \in V^2 | u \neq v\}$ and a labeling function $\lambda(v) : V \rightarrow \mathcal{P}(\mathcal{I})$ which associates a set of labels $i \in \mathcal{I}$ to each vertex $v \in V$. $\mathcal{P}(\mathcal{I})$ denotes the power set of $\mathcal{I}$. For an edge $(u, v) \in E$, u is the parent of v and v is the child of u. There is a **cycle** in the directed graph if a path can be found from a vertex to itself. A directed labeled graph is **rooted** if one can find a vertex v such that there exists a path between v and any other vertex of the graph.

Two attributed graphs $G_1 = (V_1, E_1, \lambda_1)$ and $G_2 = (V_2, E_2, \lambda_2)$ are **isomorphic** *iff* there exists a bijective function $\varphi : V_1 \rightarrow V_2$ such that:

1. $\forall v \in V_1 : \lambda_1(v) = \lambda_2(\varphi(v))$,
2. $\forall (u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$.

An **automorphism** is an isomorphism where $G_1 = G_2$.

Attributed graphs can be seen as itemsets organized in a graph structure. As such, the notion of **attributed subgraph** can be defined w.r.t. itemsets inclusion and structural inclusion. For itemset inclusion, we say that an attributed graph G_1 is contained in another attributed graph G if both attributed graphs have the same structure, and for each vertex of G_1, the associated itemset is contained in the itemset of the corresponding vertex in G.

Definition 1 *(Itemset/content inclusion)* $G_1 = (V_1, E_1, \lambda_1)$ is **contained in** $G = (V, E, \lambda)$ and is denoted by $G_1 \sqsubset_I G$, *iff* there exists a function $\varphi : V_1 \rightarrow V$ such that:

1. $\forall v \in V_1 : \lambda_1(v) \subseteq \lambda_2(\varphi(v))$,
2. $\forall (u, v) \in E_1 \Leftrightarrow (\varphi(u), \varphi(v)) \in E_2$.

Structural inclusion is represented by the classical concept of subgraph [10,14,24]. In labeled graph mining, the labels on vertices are preserved by the mapping function. In the case of attributed graphs, the mapping function preserves the itemsets associated with vertices.

Definition 2 *(Structural inclusion)* $G_1 = (V_1, E_1, \lambda_1)$ is **included in** $G = (V, E, \lambda)$ and is denoted by $G_1 \sqsubset_S G$, *iff* there exists a function $\varphi : V_1 \to V$ such that:

1. $\forall v \in V_1 : \varphi(v) \in V$,
2. $\forall v \in V_1 : \lambda_1(v) = \lambda(\varphi(v))$,
3. $\forall (u, v) \in E_1 : (\varphi(u), \varphi(v)) \in E$.

An attributed subgraph is defined in the following way to include both content and structural inclusion.

Definition 3 *(Attributed subgraph)* An attributed graph $G_1 = (V_1, E_1, \lambda_1)$ is an **attributed subgraph** of $G = (V, E, \lambda)$ and is denoted by $G_1 \sqsubset G$, *iff* there exists a function $\varphi : V_1 \to V$ such that:

1. $\forall v \in V_1 : \varphi(v) \in V$,
2. $\forall v \in V_1 : \lambda_1(v) \subseteq \lambda(\varphi(v))$, and
3. $\forall (u, v) \in E_1 : (\varphi(u), \varphi(v)) \in E$.

If G_1 is an attributed subgraph of G, we say that G is an **attributed supergraph** of G_1.

Each attributed graph G_1 that is isomorphic to G defines one **embedding** of G_1 in G. Two different embeddings may refer to the same nodes and edges, simply by mapping the nodes in a permuted way.

3.2 Problem statement

3.2.1 Frequent attributed subgraph mining

Our work considers two settings w.r.t. input data: the transactional setting and the single graph setting. In the first case, input data are organized as a transactional database $\mathcal{B}$ in which each transaction is a graph. In such case, it is possible to calculate, for each generated pattern P, a per-transaction frequency, which is given by the number of graphs in $\mathcal{B}$ for which P is a subgraph. In the second case, input data are only composed of a single graph (not necessarily connected). In such case, we use the frequency measure defined by [6]. Using this definition, the absolute frequency of a graph pattern P is equal to the minimum number of unique nodes in G that a node of P is mapped to. Based on this measure, we can calculate a relative frequency, which is given by the absolute frequency divided by the number of nodes in the graph.

These two methods of frequency calculation are used the same way. In this paper, we will refer to these measures by the generic term "frequency."

A pattern is frequent if its frequency is greater than or equal to a minimum threshold value that is called **support**. The problem consists in enumerating all frequent attributed subgraphs in a given dataset.

3.2.2 Mining closed attributed subgraphs

The problem with frequent attributed graph mining is that the number of frequent patterns is often huge. In real applications, generating all solutions can be very expensive or even

impossible. Moreover, many of these frequent attributed subgraphs contain redundant information.

Huge efforts have been made to design condensed representations that are able to summarize solutions in smaller sets (see, e.g., the works of [16] and [20]). The set of closed patterns is an example of such a condensed representation [20]. We say that an attributed graph G is a closed attributed graph if none of its proper attributed supergraphs has the same support as G. However, mining frequent closed attributed graphs could also be very expensive. We propose to relax the closure property on attributed graphs and to mine only attributed graphs which are closed w.r.t. their content (i.e., itemsets associated with vertices). This condensed representation, called the set of c-closed attributed graphs (content closed), is a superset of the set of closed attributed graphs (and a subset of the set of attributed graphs). We say that an attributed graph G is a c-closed attributed graph if there is no pattern with the same structure, the same support and supersets in the corresponding vertices.

4 Canonical form of directed attributed graphs

We propose a canonical form based on the spanning tree of the pattern (a subgraph). Our canonical form is the sequence of nodes obtained by performing a depth-first traversal of the spanning tree (note that other approaches use a sequence of edges).

In order to unambiguously identify each spanning tree, it is sufficient to identify each node by its associated attributes and its depth. As we also need to represent reentrant links, we use a third attribute that contains the index of the destination node if it is already present in the pattern.

4.1 Ordering of itemsets

In our application, nodes are associated with itemsets and one should define a total order on itemsets. Given two itemsets $\mathcal{I}$ and $\mathcal{J}$ ($\mathcal{I} \neq \mathcal{J}$), we say that $\mathcal{I} < \mathcal{J}$ iff $\exists k \in [1, \min(|\mathcal{I}|, |\mathcal{J}|)]$ such that (1) $\forall i < k : \mathcal{I}_i = \mathcal{J}_i$ and (2) $\mathcal{I}_k < \mathcal{J}_k$ or $k = |\mathcal{J}|$.

4.2 Ordering of nodes

The code of a node is a triple $(d, \mathcal{Q}, p)$ where d is the depth of the node in the spanning tree, $\mathcal{Q}$ is the set of items associated with the node, and p is 0 if the node is used for the first time or p is equal to the position of a node in the sequence (starting position is 0) if the destination node is already in the sequence. To compare two nodes, it is sufficient to compare their codes as triples.

Given two triples $T_1 = (d_1, \mathcal{Q}_1, p_1)$ and $T_2 = (d_2, \mathcal{Q}_2, p_2)$, we say that $T_1 < T_2$ iff one of the following assertions is true:

1. $d_1 > d_2$,
2. $d_1 = d_2$ and $\mathcal{Q}_1 < \mathcal{Q}_2$,
3. $d_1 = d_2$ and $\mathcal{Q}_1 = \mathcal{Q}_2$ and $p_1 < p_2$.

In Fig. 2, for example, the three children of node a in pattern $\mathcal{P}_1$ are represented by the following triples: $(1, cd, 0)$, $(1, cd, 3)$ and $(1, cd, 0)$. All nodes are located at a depth of 1, they are all associated with the same itemset cd, but the second node references the fourth node present in the sequence (whose index is 3).

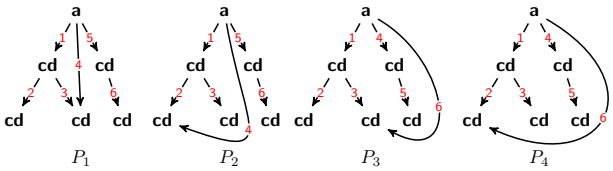

Fig. 2 Example of four isomorphic patterns (the numbers displayed on the edges show their creation order)

4.3 Ordering of attributed subgraphs

Let P be a pattern with a root node $r(P)$, we note $code(r(P))$ the code of this pattern. It is constructed by concatenating the codes of the nodes of the corresponding spanning tree explored in a depth-first fashion. The smallest code represents the canonical form.

Given two patterns P_1 and P_2, respectively, encoded by $C_1 =< c_1^1, ..., c_1^m >$ and $C_2 =< c_2^1, ..., c_2^n >$, we say that $C_1 < C_2$ iff one of the following assertions is true:

1. $\exists k \in [1, min(m, n)]$ such that $\forall i < k : c_1^i = c_2^i$ and $c_1^k < c_2^k$
2. $\forall i < min(m, n) : c_1^i = c_2^i$ and $m > n$.

Figure 2 shows four isomorphic patterns of the same directed attributed graph. Their codes are represented by the following sequences:

1. $code(r(P_1)) =< (0, a, 0)(1, cd, 0)(2, cd, 0)(2, cd, 0)(1, cd, 3)(1, cd, 0)(2, cd, 0) >$
2. $code(r(P_2)) =< (0, a, 0)(1, cd, 0)(2, cd, 0)(2, cd, 0)(1, cd, 2)(1, cd, 0)(2, cd, 0) >$
3. $code(r(P_3)) =< (0, a, 0)(1, cd, 0)(2, cd, 0)(2, cd, 0)(1, cd, 0)(2, cd, 0)(1, cd, 3) >$
4. $code(r(P_4)) =< (0, a, 0)(1, cd, 0)(2, cd, 0)(2, cd, 0)(1, cd, 0)(2, cd, 0)(1, cd, 2) >$

By using the ordering of triples previously defined, we found $code(r(P_4)) < code(r(P_3)) < code(r(P_2)) < code(r(P_1))$; then, the pattern P_4 is the canonical form.

5 Enumeration of patterns

Subgraph patterns are generated by performing a depth-first traversal of the search space based on their spanning tree. All items present in the input graph(s) constitute the initial patterns that will be progressively extended. Each pattern extension is built using the canonical form sequence generated in the previous step.

The use of the strategy defined by [19] allows enumerating all spanning trees. The generation of new patterns is performed by using the rightmost path extension method [7]. Two types of extensions are possible: The **itemset extension** adds a new item to the itemset associated with the rightmost node of the spanning tree, and the **structural extension** adds a new child to one of the nodes composing the right path of the spanning tree. The method is complete but might generate redundant patterns in the form of isomorphic graphs. Duplicate candidates are detected and discarded before the candidate extension process by performing a canonical check. The limitation of this approach is that it only allows finding rooted patterns.

5.1 Identification of canonical and automorphic patterns

The code for the graphs defined above has, like most of the codes used by graph mining algorithms, the following property: Each prefix of a canonical code is itself canonical [9,22]. The extension of a pattern does not change the order of the triples in the case of the generation

of a new canonical pattern. Thus, each canonical pattern can only be obtained by adding a new triple or adding an item to the itemset associated with the last triple.

By using the rightmost path expansion strategy, it is possible to check the canonicity of a pattern by testing only the nodes belonging to the right path. Let l and p be two functions that return the last and penultimate child of a node, respectively. It is possible to determine the canonicity of a pattern in the following way: For all the nodes n belonging to the right path, if $\exists p(n)$ and $code(p(n)) \leq code(l(n))$, then the pattern is in canonical form. It is, in the same way, fairly easy to identify the extensions that produce patterns with automorphisms. If at least one of the nodes belonging to the right path has more than one child and for all the nodes n belonging to the right path, if $\exists p(n)$ and $code(p(n)) = code(l(n))$, then the pattern has automorphisms. On such a pattern, we call **split node** the node from the right path that is closest to the root and that has several children with the same label. Thus, the patterns shown in Fig. 1c, d have automorphisms. On these two figures, the split node is the node a.

5.2 Handling reentrant links

From the enumeration of spanning trees described above, it is possible to define a method for enumerating all rooted subgraphs (since a rooted subgraph is a tree with reentrant links). For this, we need to identify when a structural extension adds a node that is already present in the candidate pattern. When this happens, the node is added to the pattern as if it is a normal structural extension, but we indicate that it is a reuse of a node already present by storing the index of the pointed node. The exploration of the search space is stopped from this node. Indeed, because using this exploration strategy, we are sure that all the nodes belonging to the right path are extended in the current generation phase. All structures that are not part of the rightmost path have already been fully explored. The reuse of a node is only reflected in the code of the graph by adding a triple that refers to another triple. This operation generates no change in the properties of the code, the completeness of the method and the way to eliminate noncanonical patterns.

5.3 Handling cycles

The extension of a pattern can lead to the creation of a cycle if and only if the new added node points to one of the nodes already present in the rightmost path. When a cycle is created, we proceed in the same manner as other reentrant links: A new node is added with the index of the reused node, and the extensions from this node are stopped. The completeness of the method is not affected by this operation. However, if the reused node is the root, we can generate redundant solutions. In such case, many isomorphic patterns are generated by starting the exploration from the other nodes belonging to the cycle.

We solve this problem by only extending one of these patterns. First, we generate all the isomorphic patterns s.t. their root is one of the nodes belonging to the right path. Then, we keep the pattern with the smallest canonical form and continue the exploration only for this pattern. In Fig. 3, P_1 represents a pattern generated from the graph G. The node labeled with d is the last node added to the pattern. This extension creates a cycle composed of 3 nodes belonging to the right path (d, c and bc). Thus, we have 2 other isomorphic patterns (one for each other node of the cycle). From the node bc, we obtain the pattern P_2. From the node c, we obtain the pattern P_3. The pattern that has the smallest code is the one rooted in bc, so we stop here the exploration of the pattern P_1.

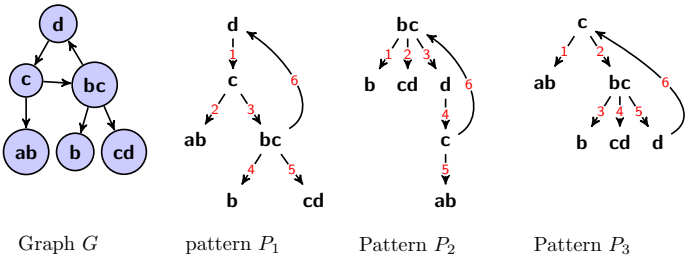

Fig. 3 Illustration of redundant patterns related to cycle

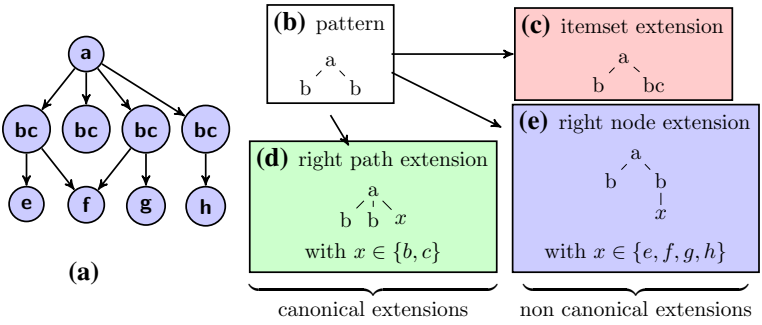

Fig. 4 Illustration of the possible extensions of a pattern with automorphisms

5.4 Handling automorphic patterns

The use of canonical forms avoids the generation of redundant solutions. It is used in our algorithm to prune the search space. However, this does not solve the problem posed by the presence of automorphic patterns.

For example, let us consider the directed attributed graph shown in Fig. 4a. This graph contains several automorphic substructures that make less efficient the pruning based on canonical forms. When the pattern shown in Fig. 4b is analyzed, it is accepted and extended since it is in canonical form. However, this pattern can be obtained in twelve different ways. The naive algorithm consists, for each of these forms, to extend the pattern by using, as defined above, the itemset extension or the structural extension.

Itemset extension generates the pattern shown in Fig. 4c. This pattern will be rejected because it is not in canonical form. Structural extensions from node b generate one of the patterns shown in Fig. 4e. In all cases, the resulting pattern is not in canonical form and, as a consequence, it will be discarded. The only operation that can generate a canonical pattern is a structural extension from the split node a (Fig. 4d).

When we identify a new pattern with automorphisms, we know that this pattern cannot be extended on the last node or on all the nodes located after the split node. In fact, for a split node n, we have $code(p(n)) = code(l(n))$, then any extension done on the pattern of root $l(n)$ will lead to a pattern with a code lower than $p(n)$, so this pattern will not be in canonical form. We can use this fact to avoid generating unnecessary patterns. This is a first optimization, but its impact is limited.

The second optimization is to remove the mappings of a pattern that are not able to generate new canonical patterns. In Fig. 1, for example, the pattern of Fig. 4d, which is obtained in twelve different ways, can be only extended by adding a third child to the split node annotated a. Whatever way the pattern of Fig. 4d has been obtained, the possibilities of expansion are unchanged. It is therefore possible, in this case, without omitting solution, to keep only one mapping of the pattern.

With the graph in Fig. 4a, the situation is more complicated because it is possible, from this pattern, to add a child c to the node labeled a. In this case, it will be possible in the next generation, to apply any type of extension for all the nodes belonging to the right path. In the example, we could, in particular, add e, f, g or h as a child of node c. In order that all possible patterns could be generated, we must keep the possibility of extending the pattern with 2 bc nodes from the 3 bc nodes with a child. This represents $\binom{3}{2}$ different mappings.

In a real graph, the situation is often less simpler and determining what mappings it is possible to remove is not an easy task. To avoid performing calculations on the graph structure that would affect the performance of the algorithm, we choose to keep only one mapping among those using the same nodes. This decreases the number of mappings from a number of permutations to a number of combinations. This solution is not optimal, but it is sufficient to limit the combinatorial explosion.

In a star graph in which a node is associated with n nodes with the same label, there exists $\sum_{i=1}^{n} {}^{n}P_i$ different ways of generating all the included patterns. After the optimization, this number drops to $\sum_{i=0}^{n-1} (n-i)\binom{n}{i}$.

6 Mining algorithm

Mining all frequent rooted attributed subgraphs generates a huge number of patterns. In real applications, generating all solutions can be very expensive or even impossible. In addition, many of these frequent attributed subgraphs contain redundant information. On the other hand, mining frequent closed attributed graphs is very expensive in terms of processing time.

Based on the experiments we conducted on attributed tree mining [19], we developed an algorithm called AADAGE (Automorphism Aware Directed Attributed Graph Explorer) that is designed to mine content closed attributed graphs. It has been shown (Pasquier et al. [19]) that mining content closed patterns is a good compromise between non redundancy of solutions and execution time.

Figure 5 shows the high-level structure of the AADAGE algorithm. The algorithm takes in parameter a database ($\mathcal{B}$) and a minimum support ($minSup$). First (line 1), a set $\mathcal{C}$ containing all attributed subgraphs of size 1 is built by scanning $\mathcal{B}$. A subgraph of size 1 is composed of one node associated with one item. Therefore, $\mathcal{C}$ is the set of nodes associated with every item presents in $\mathcal{B}$. In line 2, the set of solutions $\mathcal{S}$ is initialized with the empty set. Between line 3 and line 22, a loop allows to process every candidate in the set.

The function $GetFirst$ (line 4) returns the smallest candidate in the set according to the order of attributed graphs defined in Sect. 4.3. In line 5, the algorithm performs a canonical test (function $isCanonical$, implementing the method described in Sect. 5.1) and a frequency test (function $frequency$) and discards, from further processing, the candidates that do not pass these tests. In line 6, candidates for which the previous extension step leads to the creation of a cycle (detected with the function $hasCycle$) are checked for canonicity with the method described in Sect. 5.3. Candidates that have a cycle and that are not canonical are discarded. In line 7, the algorithm performs a test for the presence of automorphisms (function

```
AADAGE(B, minSup)
 1: C ← {all attributed subgraphs of size 1 in B}
 2: S ← ∅
 3: while C ≠ ∅ do
 4:    G ← getFirst(C)
 5:    if isCanonical(G) and frequency(G) ≥ minSup then
 6:       if not hasCycle(G) or (hasCycle(G) and isCycleCanonical(G)) then
 7:          if hasAutomorphisms(G) then
 8:             G ← optimizeMappings(G)
 9:             X ← extendBefore(G, getSplitNode(G))
10:          else
11:             X ← extend(G)
12:          end if
13:          if ∄G' ∈ S : G ⊏_I G' and omFrequency(G') = omFrequency(G)
             then
14:             C ← C ∪ X
15:          end if
16:          if ∄G' ∈ S : frequency(G') = frequency(G) then
17:             S ← S ∪ {G}
18:          end if
19:       end if
20:    end if
21:    C ← C \ {G}
22: end while
23: printSolutions(S)
```

Fig. 5 AADAGE Algorithm

hasAutomorphisms, implementing the method described in Sect. 5.1). Candidates with automorphisms are processed in line 8 to 9. In line 8, the mappings of the patterns that are not able to generate new canonical patterns are removed (see Sect. 5.4). In line 9, extension of the patterns (stored in the set $\mathcal{X}$) are only performed for nodes located before the split node (function *extendBefore* that takes as parameters the candidate node G and the splitNode determined by the function *getSplitNode*. see Sect. 5.4). In line 11, candidates that do not have automorphisms are extended by the regular method *extend*.

In lines 13 to 15, the set of extensions is added to the set of candidates only if the current candidate G is not contained in a pattern belonging to the set of solutions with exactly the same occurrences. We use for the test, the method *omFrequency* that computes the total number of occurrences of a pattern in $\mathcal{B}$ (the occurrence-match frequency).

In line 16 to 18, the candidate is added to the set of solutions only if it is content closed; i.e., G is not contained in a pattern belonging to the set of solutions with the same frequency.

The processing of a candidate finishes by removing it from the candidates' list (line 21).

7 Experimental results

The method described in this paper has been implemented in C++ using STL. Experiments were performed on a computer running Ubuntu 13.04 and based on a Intel©Core™i5-2400 @ 3.10 GHz with 12 GB main memory. The hard disk used for the experiment is a 500 GB capacity with a SATA 3 Gb/s interface, 32 MB cache and 5400RPM spin speed. All timings are based on total execution time, including all preprocessing and results output.

7.1 Synthetic datasets

A set of 10,000 directed graphs, each comprising 1,000 nodes and 5000 edges, were generated with the program of [13]. This set of graphs was used as the basis for the creation of five datasets of directed attributed graphs named $A1$ to $A5$ in which nodes were associated with itemsets of size ranging from 1 to 5. To simulate the fact that, as in many real datasets, most attributes are rare, but a few are very common, we assigned each item a random integer value distributed according to a power law (more specifically, we have used a Pareto distribution with parameter $\alpha = 0.05$).

7.2 Google+ and Twitter datasets

We used the social graphs of Google+ and Twitter built by [17]. In each dataset, a node represents an individual and the associated attributes are the characteristics of this person. There is an edge between two persons when one of them is following the activity of the other. The datasets are substantial (107,614 nodes and 13,673,453 edges for Google+, 81,306 nodes and 1,768,149 edges for Twitter).

7.3 PubMed Central citation network

PubMed Central is a bibliographic database of scientific articles in the field of life sciences. This database contains 322,526 Open Source papers but only 58,728 of them cite another paper which is itself Open Source. We built a citation graph where each node represents a paper, each edge, a citation link, and the attributes associated with nodes identify the papers' keywords. The graph is very sparse as it contains only 60,012 edges; however, some attributes are frequently associated with subsets of connected nodes.

7.4 Performances evaluation

The results of our algorithm are shown in Fig. 6. The comparison with existing algorithms is only possible with labeled graphs. The only dataset in which each node is associated with a single item is dataset $A1$ presented in Fig. 6a. For this dataset, the performance obtained with the implementations of Gspan carried out in the project ParSeMiS [21] (an implementation that allows the mining of directed graphs) is also presented. Gspan and AADAGE, which are set to mine directed, rooted and connected subgraphs, generate exactly the same results. For the smallest values of support, our strategy of mapping filtering allows making a difference with Gspan.

Figure 6b shows the impact of itemset size (for the same graph) on algorithm performance. The execution times increase exponentially when the number of itemsets in each node increases. This highlights the difficulty of mining attributed graphs in comparison with labeled graphs. Indeed, when mining attributed graphs, we have to deal with the combinatorics of graphs and itemsets at the same time.

The Google+ and Twitter datasets contain many nodes, many attributes and especially some very common attributes (e.g., the attribute "$gender = male$," which is present in 52 % of the nodes) whose presence generates so many patterns that the mining fails. In Fig. 6c, the two gender attributes were removed from Google+ dataset. Twitter dataset has not been changed. Despite its size, Google+ dataset was successfully mined by setting a minimum support of 3 %. Twitter dataset, although smaller, has been treated up to a support of 5 %. The common feature of these two datasets is that they are treated fairly rapidly up to a

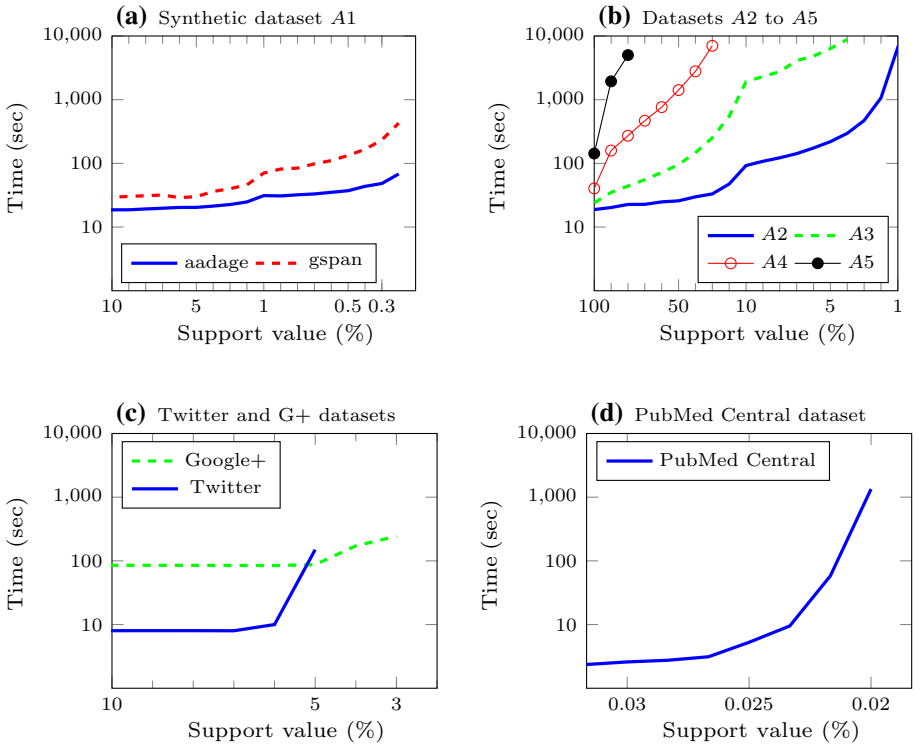

Fig. 6 Performances of the algorithm on several datasets

specific threshold corresponding to the inclusion of a new frequent attribute that generates an enormous amount of patterns.

With PubMed Central dataset, it was possible to use a very low support (Fig. 6d). This allowed identifying some interesting patterns. A pattern, for example, concerns a set of papers annotated with "$p53$" which is the name of an oncogene. These papers are cited by other papers annotated "$rapamycin$" (an immunosuppressant drug designed in 1975) that are themselves cited by papers annotated "$aging$" (the influences of rapamycin on the deceleration of cellular senescence were discovered in 2009) and "$mTOR$" (a gene that is inhibited by rapamycin). This pattern represents a highly summarized picture of the progress of some research on aging.

8 Conclusion and perspectives

We have presented in this paper a method to mine directed attributed graphs and shown its effectiveness in the analysis of multiple datasets. We focused specifically on graphs in which, as in real data, some attributes are shared by a large number of nodes. We have shown that, depending on the structure of the pattern being analyzed, it is not always necessary to extend all possible mappings of a pattern. We treated specifically the case of automorphic patterns, but there exists other patterns for which some optimizations can be performed. For example, when the pattern in Fig. 1b, which does not have automorphisms, is found 12

times on the graph of Fig. 4a, we can see quite easily that some mappings can be discarded. We are convinced that the detailed analysis of the patterns and their mappings may reveal other configurations where a reduction of the search space can be performed, which could make possible the mining of larger or denser datasets. It is a future work that we aim to explore.

Acknowledgments This work was supported by the ANR Grant "FOSTER" ANR-2010-COSI-012-01.

References

1. Agrawal R, Imieliński T, Swami A (1993) Mining association rules between sets of items in large databases. SIGMOD Rec 22(2):207–216
2. Agrawal R, Srikant R (1995) Mining sequential patterns. In: ICDE'95, pp 3–14
3. Ayres J, Flannick J, Gehrke J, Yiu T (2002) Sequential pattern mining using a bitmap representation. In: KDD'02, pp 429–435
4. Borgelt C (2007) Canonical forms for frequent graph mining. In: Decker R, Lenz H-J (eds) Advances in data analysis. Springer, Berlin, pp 337–349
5. Borgelt C, Berthold M (2002) Mining molecular fragments: finding relevant substructures of molecules. In: ICDM'02, pp 51–58
6. Bringmann B, Nijssen S (2008) What is frequent in a single graph?. In: PAKDD'08, pp 858–863
7. Chi Y, Yang Y, Xia Y, Muntz RR (2004) Cmtreeminer: mining both closed and maximal frequent subtrees. In: PAKDD'04, pp 63–73
8. Fukuzaki M, Seki M, Kashima H, Sese J (2010) Finding itemset-sharing patterns in a large itemset-associated graph. In: PAKDD'10, pp 147–159
9. Huan J, Wang W, Prins J (2003) Efficient mining of frequent subgraphs in the presence of isomorphism. In: ICDM'05, pp 549–552
10. Inokuchi A, Washio T, Motoda H (2000) An apriori-based algorithm for mining frequent substructures from graph data. In: PKDD'00, pp 13–23
11. Inokuchi A, Washio T, Motoda H (2003) Complete mining of frequent patterns from graphs: mining graph data. Mach Learn 50(3):321–354
12. Jiang C, Coenen F, Zito M (2013) A survey of frequent subgraph mining algorithms. Knowl Eng Rev 28:75–105
13. Johnsonbaugh R, Kalin M (1991) A graph generation software package. SIGCSE Bull 23(1):151–154
14. Kuramochi M, Karypis G (2001) Frequent subgraph discovery. In: ICDM'01, pp 313–320
15. Kuramochi M, Karypis G (2004) An efficient algorithm for discovering frequent subgraphs. IEEE Trans Knowl Data Eng 16(9):1038–1051
16. Mannila H, Toivonen H (2005) Multiple uses of frequent sets and condensed representations. In: KDD'05, pp 189–194
17. McAuley J, Leskovec J (2012) Learning to discover social circles in ego networks. Neural Inf Process Syst 25:548–556
18. Miyoshi Y, Ozaki T, Ohkawa T (2009) Frequent pattern discovery from a single graph with quantitative itemsets. In: ICDMW'09, pp 527–532
19. Pasquier C, Sanhes J, Flouvat F, Selmaoui-Folcher N (2015) Frequent pattern mining in attributed trees: algorithms and applications. Knowl Inf Syst 46(3):491–514
20. Pasquier N, Bastide Y, Taouil R, Lakhal L (1999) Discovering frequent closed itemsets for association rules. In: ICDT'99, pp 398–416
21. Wörlein M, Meinl T, Fischer I, Philippsen M (2005) A quantitative comparison of the subgraph miners mofa, gspan, ffsm, and gaston. In: PKDD'05, pp 392–403
22. Yan X, Han J (2002) gspan: graph-based substructure pattern mining. In: ICDM'02, pp 721–724
23. Yan X, Han J (2003) CloseGraph: mining closed frequent graph patterns. In: KDD'03, pp 286–295
24. Yan X, Yu PS, Han J (2004) Graph indexing: a frequent structure-based approach. In: SIGMOD conference, pp 335–346

Claude Pasquier received a Ph.D. degree in Computer Science from the University of Nice Sophia Antipolis, France, in 1994. During his thesis, he explored the use of software engineering paradigms in the field of structured document manipulation. Subsequently, he was a postdoctoral researcher at the Biophysics and Bioinformatics Laboratory of the University of Athens, Greece, where he conducted research on protein structure prediction. He held positions at the French National Institute for Research in Computer Science and Control (INRIA) and with Schlumberger, Smart Cards & Terminals division (now Gemalto) where he worked on language-based systems. He is presently a researcher at the French National Center for Scientific Research (CNRS). His current research interests include data mining, bioinformatics and Knowledge-based systems.

Frédéric Flouvat is an Associate Professor at the University of New Caledonia (Nouméa, New Caledonia), where he is teaching Algorithmic and Databases. He is also a member of a multidisciplinary research team on material and environment. This laboratory brings together geologists, physicists and computer scientists to address both fundamental and applied questions related to the concepts of risk and sustainable development. His research interests are spatio-temporal data mining and its application to environmental sciences.

Jérémy Sanhes is a Ph.D. student at the University of New Caledonia and INSA Lyon (France). His thesis deals with spatio-temporal data mining issues.

Nazha Selmaoui-Folcher is an Associate Professor (HDR) at the University of New Caledonia since 1998. She is the leader of a multidisciplinary laboratory on material and environment since 2012 (PPME EA 3325). She is teaching computer sciences and mathematics. She is the coordinator for the FOSTER national projects dedicated to Knowledge Discovery on Spatio-temporal Databases and Application to Soil Erosion. She received her Ph.D. degree in 1992 from the "Institut National des Sciences Appliques" at Lyon (France) and her Habilitation degree in 2012 from the University of Lyon I. Her current research interests are data mining of time series of satellite images analysis and spatio-temporal data and its application to environmental sciences. She is involved in the program committees of many data mining conferences.