PLAGRAM : un algorithme de fouille de graphes plans efficace

Adriana Prado, Baptiste Jeudy, Elisa Fromont, Fabien Diot

Université de Lyon, Université de St-Etienne F-42000,
UMR CNRS 5516, Laboratoire Hubert-Curien, France.

Résumé : La recherche de sous-graphes fréquents dans une base de données de graphes est très coûteuse notamment à cause de l'utilisation massive de tests d'isomorphismes NP-complets. Pourtant, pour certains types de graphes, comme les graphes plans, la complexité de ces tests d'isomorphisme de sous-graphes peut être énormément réduite. Nous proposons un algorithme efficace de fouille de graphes plans 2-connectés qui exploite la propriété planaire des graphes en utilisant un test d'isomorphisme de complexité linéaire et en permettant de réduire drastiquement le nombre de candidats à envisager pendant la recherche. Nous montrons dans le cadre d'applications vidéo l'intérêt d'un tel algorithme non seulement en termes d'efficacité mais aussi en termes de pertinence des résultats obtenus dans le cadre de notre application. **Mots-clés** : Apprentissage de Graphes

1. Introduction

Graph mining is an important data mining process with many applications in, for example, molecular biology, analysis of (social) networks, etc. Typical frequent graph mining algorithms generate plausible *pattern* subgraphs and then compute their frequency (w.r.t. a user-defined frequency threshold also referred to as minimum support) while finding all their occurrences in a database of *target* graphs. Afterwards, the found frequent patterns are extended in a valid way such that bigger patterns can also be evaluated. All those subtasks are computationally expensive due to the numerous possibilities of extending a pattern, the isomorphism tests between all candidate common subgraphs and all target graphs, and, naturally, the huge number of possible frequent patterns that can be found in reasonably large databases.

The subgraph isomorphism problem, for example, is known to be NP-complete. Current graph mining approaches (e.g., Kuramochi & Karypis (2001); Yan & Han (2002); Nijssen & Kok (2004)) can deal with applications where the subgraph isomorphism test is not too costly or when there are not too many such tests. For example, when the graphs in the database are small, have low degrees, have many node (or edge) labels, have a low number of cycles, etc.

For some kinds of graphs, however, the complexity of such tests can be drastically improved. This is the case for plane graphs. A planar graph is a peculiar graph that can be drawn on the plane in such a way that its edges intersect only at their endpoints. A planar graph already drawn in the plane without edge intersections is called a plane graph or a planar embedding of the graph. Plane graphs are particularly interesting since it is known that the subgraph isomorphism test is polynomial for these type of graphs (see for example Damiand *et al.* (2009)).

Plane graphs can be found in several useful applications. For example, a video could be represented by a set of plane graphs (one for each frame) by means of, e.g., region adjacency relationships or interest point triangulation (Chang *et al.* (2004)). Using these representations, interesting objects in the video may be frequent subgraphs in the corresponding set of plane graphs and, thus, following an object in a video could be related to the frequent graph mining problem when sufficient information (for example, spatial information) is associated to the graphs.

Current general-purpose graph mining algorithms do not computationally benefit from the plane property of target graphs and therefore cannot tackle in a satisfactory way the aforementioned video application. In fact, as pointed out in our experimental section, the popular general-purpose graph mining algorithm GSPAN by Yan & Han (2002) could not finish its executions on our video datasets within 3 days of computation.

In this context, we propose an efficient graph mining algorithm, called PLAGRAM (**Pla**ne **Gra**ph **M**ining), which is dedicated to mining plane graphs and has the particularity of extending patterns using complete faces. This drastically reduces the complexity of the extension building phase (in which the patterns are extended) and, as we show in the experiments, it does not limit the quality of the found patterns.

PLAGRAM borrows many features from the algorithm GSPAN. GSPAN performs a depth-first search in a space of canonical codes, which are computed such that two isomorphic graphs are not evaluated twice. One of the most acknowledged bottleneck of this algorithm comes from the subgraph isomor-

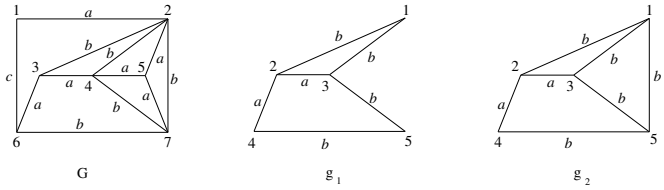


FIG. 1: Plane graphs. The edge labels are in $\{a, b, c\}$ and we assume that all node labels are equal to a.

phism tests. In particular, GSPAN is not well suited to mine graphs with many cycles as their presence increases exponentially the number of frequent subgraphs. However, the particular plane subgraphs considered in this paper are far less numerous than the subgraphs considered by GSPAN, which makes our approach not only faster, but also capable of dealing with more complex graphs (in terms of degrees and size) than GSPAN.

We have therefore divided the paper in the following way : next, we give some definitions. In Section 3., we introduce the (canonical) codes we use to represent plane graphs and then, in Section 4., we present the PLAGRAM algorithm. In Section 5., we report on some experiments on the efficiency of the proposed algorithm. We also discuss its usefulness in tackling video application problems. We conclude in Section 6..

2. Definitions

A graph is generally defined by its set of nodes and its set of edges. One way to extend this definition to plane graphs is to define them as a set of nodes where each node has a circular list of its neighbors in anticlockwise order. The infinite face of a plane graph is called the *outer face* of the graph. The other faces are called *internal faces*. The plane graphs that we consider can also have labels on their edges and/or nodes (denoted by functions L_e and L_n). A graph is *2-connected* if for every pair of nodes (x, y) there is a cycle that contains both.

Figure 1 presents three 2-connected plane graphs. The neighbor list of node 5 of graph G is $N(5) = \langle 2, 4, 7\rangle$ (or any circular permutation of it). Face $\langle 3, 6, 7, 4\rangle$ of G is an internal face and $\langle 1, 2, 7, 6\rangle$ is its outer face.

Graph G' is *plane subgraph isomorphic* to G (or G' is a *plane subgraph* of G), denoted $G' \subseteq G$, if there is an injective function f which maps the nodes of G' to nodes of G and which preserves the labels, the edges, and the internal

faces (if it also preserves the outer face, G' and G are plane isomorphic). The function f is *an occurrence* of G' in G.

In Figure 1, graph g_1 is a subgraph of G. The internal faces $\langle 1, 2, 3\rangle$ and $\langle 2, 4, 5, 3\rangle$ of g_1 correspond respectively to faces $\langle 2, 3, 4\rangle$ and $\langle 3, 6, 7, 4\rangle$ of G, with $f(1) = 2$, $f(2) = 3$, $f(3) = 4$, $f(4) = 6$ and $f(5) = 7$. Graph g_2 has three internal faces mutually adjacent, one with four edges and two with three edges. Since this configuration of faces does not exist in G, g_2 is not a plane subgraph of G.

The *frequency* (or support) of a plane graph g in a database $\mathcal{D} = \{G_1, ..., G_n\}$ of plane graphs is the number of these graphs which contain g as a plane subgraph, i.e., $|\{i \mid g \subseteq G_i\}|$.

Problem Definition : Given a frequency threshold σ, the problem we tackle in this paper is to compute the set of all 2-connected plane subgraphs with a frequency greater than σ in a database $\mathcal{D}$.

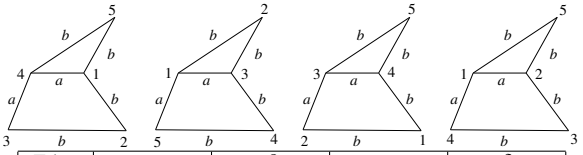
3. Graph Codes

As GSPAN, the algorithm PLAGRAM represents the pattern graphs by graph codes and actually explores a code search space to find the frequent ones. In this section, we define these new codes and we present important properties of the code search space.

Definition 1 (valid extension)
Given a plane graph g and two nodes u and v on the outer face of g, we can extend g by adding a new path $P = (u = x_1, x_2, \ldots, x_k = v)$ to g between u and v. This path must lie in the outer face of g. Nodes x_2,..., x_{k-1} are $(k-2) \geq 0$ new nodes with $N(x_i) = \langle x_{i-1}, x_{i+1}\rangle$. This new graph is denoted $g \cup P$. Given a plane graph G such that $g \subset G$, P is a valid extension *of g in G if $g \cup P \subseteq G$.*

In other words, this definition states that any pattern graph g constituted of aggregated faces can only be extended by the addition another complete face lying in the outer face of g. This restriction is related to that of GSPAN, where a graph is extended by the addition of a single edge only to nodes of the rightmost path of the depth-first search tree.

In Figure 1, there are three valid extensions of g_1 in the target graph G. These three extensions have 2 edges and thus a new node 6 must be added in the outer face of g_1. These extensions are : $P_1 = (1, 6, 3)$ and $P_2 = (3, 6, 5)$ with $f(6) = 5$, and $P_3 = (4, 6, 1)$ with $f(6) = 1$.



Edge	α	β	γ	δ
1	(1,2,a,b,a)	(1,2,a,b,a)	(1,2,a,b,a)	(1,2,a,a,a)
2	(2,3,a,b,a)	(2,3,a,b,a)	(2,3,a,a,a)	(2,3,a,b,a)
3	(3,4,a,a,a)	(3,1,a,a,a)	(3,4,a,a,a)	(3,4,a,b,a)
4	(4,1,a,a,a)	(3,4,a,b,a)	(4,1,a,b,a)	(4,1,a,a,a)
5	(4,5,a,b,a)	(4,5,a,b,a)	(3,5,a,b,a)	(1,5,a,b,a)
6	(5,1,a,b,a)	(5,1,a,a,a)	(5,4,a,b,a)	(5,2,a,b,a)

FIG. 2: Four codes α, β, γ and δ of graph g_1 of Figure 1.

A code for a plane graph g is a sequence of its edges. Each edge is represented by a 5-tuple $(i, j, L_n(i), L_e(i, j), L_n(j))$, where i and j are the indices of the nodes (from 1 to n, where n is the number of nodes in g). The nodes are numbered as they first appear in the code. The following definition describes the order in which the edges must appear for a code to be valid.

Definition 2 (valid code)
If $g = (V, N, L_n, L_e)$ is a plane graph with only one internal face $\langle v_0, ..., v_{n-1}\rangle$ (i.e., g is a cycle), then a valid code for g is $(1, 2, L_n(1), L_e(1, 2), L_n(2)).(2, 3, \ldots)$, $(3, 4, \ldots) \ldots$, $(n-1, n, \ldots).(n, 1, \ldots)$. We use a "dot" to denote the concatenation of each 5-tuple representing an edge of g. If $g = g' \cup P$ and P is a valid extension of g' in g, then a valid code for g is the concatenation of a valid code for g' and the code of P.

It is not obvious from this definition that every plane graph g has at least one valid code. This is actually a consequence of the fact that every $g' \subset g$ has at least one valid extension in g (proof omitted ; the 2-connectedness of g is needed). Therefore, it is possible to construct a valid code by choosing an internal face of g and then iteratively adding valid extensions to it.

Figure 2 shows four valid codes of graph g_1 in Figure 1 (among seven valid codes) and the corresponding node numbering on graph g_1 (recall that there is a different numbering of the nodes for each code). Codes α, γ, δ start with the 4-edge face and then a 2-edge extension is added to build the second face. Code β starts with the 3-edge face and then a 3-edge extension is added. In each column, the line separates the edges of the first face from the edges of the valid extension. A valid code for this graph can start with any of the six

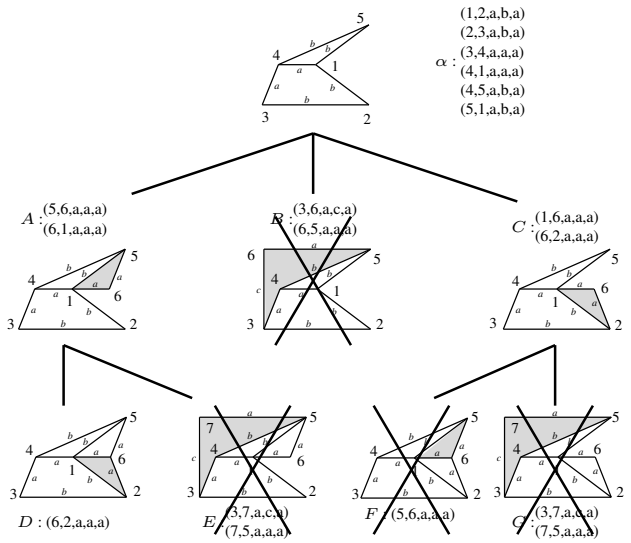


FIG. 3: Part of the code tree starting from code α of Figure 2. In each pattern, the gray face correspond to the last added extension. The extension codes are A, ..., G (the complete code of the last line leftmost pattern is thus $\alpha.A.D$). Non-canonical codes are crossed.

edges. For the edge that belongs to the two internal faces, the code can start with any of the two faces, hence the seven possible codes.

The set of valid codes is organized in a *code tree*. A code C is a child of C' if there is a valid extension P of C' such that C is the concatenation of C' with the code of P. The root of the code tree is the empty code. A part of this tree rooted at code α is represented in Figure 3. Notice that the codes at a given level of the tree represent graphs that have one more face than the codes of the level just above (in GSPAN, codes at one level have exactly one more edge than those at the level above).

In this code tree, each graph is represented by several codes (for instance, we already saw that graph g_1 has seven valid codes). In Figure 3 we also see that codes $\alpha.A.D$ and $\alpha.C.F$ represent the same graph. Of course, exploring several codes that represent the same graph is not efficient. We therefore define *canonical codes* such that each graph has exactly one canonical code.

We start by defining an order on the valid codes. We suppose that there exists an order on the labels. Then, we define an order on the edges by taking the lexicographic order derived from the natural order on indices and the order on labels. It means that $(i, j, L_n(i), L_e(i, j), L_n(j)) <$

$(x, y, L_n(x), L_e(x, y), L_n(y))$ if $i < x$ or ($i = x$ and $j < y$) or ($i = x$ and $j = y$ and $L_n(i) < L_n(x)$), and so on. Afterwards, we extend this order on edges to a lexicographic order on the codes. In the figures and examples, we assume the order on the labels is $a < b < c$. Therefore, in Figure 2, $\alpha > \beta$ because they have the same first two edges and the third edge of β is smaller than the third edge of α. Because of the second edge, $\beta > \gamma$ and finally $\gamma > \delta$ because the first edge of γ is bigger than the first edge of δ. Actually, code α is the biggest code that exists for graph g_1. We thus define the *canonical code* of a graph as the biggest code of this graph.

PLAGRAM does a depth-first exploration of this code tree. The next theorem states that, if C is a non-canonical code, then it is not necessary to explore the children of C ; the whole subtree rooted at C can safely be pruned.

Theorem 1
In the code tree, if a code is not canonical, then none of its descendants are.

For instance, in Figure 3, $\alpha.A.D$ and $\alpha.C.F$ represent the same graph. Since $\alpha.A.D > \alpha.C.F$, any extension of $\alpha.A.D$ will be bigger than any extension of $\alpha.C.F$. Thus, the latter code can be pruned.

4. PLAGRAM algorithm

The pseudo-code of PLAGRAM is shown in Figure 4. Its outline is similar to that of GSPAN. The main differences are the graph code used to represent a plane graph and the way extensions are generated. It is a depth-first recursive exploration of the code tree. Although the first level of this tree contains codes of graphs with one face, for efficiency reasons, PLAGRAM starts its exploration with frequent edges (Figure 4, line 1). Function *mine* explores the part of the code tree rooted at a code given by its parameter. If this code is canonical (line 4), then the algorithm computes its extensions on every target graph in $\mathcal{D}$ (lines 6-8) and makes a recursive call on the frequent ones (line 12).

In the next subsections, the sizes (expressed in number of edges) are denoted m for the pattern code P and m_i for each of the target graphs G_i. We use these notations to express the complexity of function *mine*.

4.1. Canonical test (line 4)

This test is done by comparing code P with the canonical code of the graph represented by P. Since two plane graphs are isomorphic if their ca-

```
Algorithm : PLAGRAM(D)
Input : a graph database D
Output : frequent plane subgraphs in D.

1   Find all frequent edge codes in D
2   for all frequent edge code E do
3       mine(E,D)
```

```
mine(P,D)
Input : the code of a pattern P and the graph database D.

4   if P is not canonical then return
5   LE = ∅   //list of extensions of P
6   for all graph G_i ∈ D do
7       for all occurrences f of P in G_i do
8           LE = LE ∪ build_extensions(P, G_i, f)
9   for all extensions E in LE do
10      if E is frequent then
11          print(P.E)
12          mine(P.E)
```

FIG. 4: PLAGRAM algorithm

nonical codes are equal, the complexity of this test is at least as high as an isomorphism test. The complexity of graph isomorphism, in the general case, is unknown, but for plane graphs, polynomial algorithms exist (see for instance Damiand *et al.* (2009) for a quadratic algorithm). Here is a sketch of our algorithm : it constructs the canonical code of a graph by first choosing a starting face and a starting edge (in this face) for the code. Since P has m edges and considering that each edge belongs to at most 2 faces, there are at most $2m$ such choices. Then, it iteratively extends the code by concatenating to it the code of the valid extension with the biggest code. Each of these steps has a complexity of $O(m)$ and must be repeated as many times as the number of faces in the graph (which is less than m). Therefore, the complexity of finding the canonical code of a graph is in the worst case $O(m^3)$. However, experiments show that this canonical test is not the bottleneck of PLAGRAM.

4.2. Pattern matching (line 7)

For each pattern P, the algorithm must find all its occurrences in every target graph G_i. Each of these operations involves a subgraph isomorphism test, which works as follows : for every edge e of G_i, it tries to match e with the first edge of P. Once this match is done, the complexity for matching the rest of P is $O(m)$. So, the total complexity is, in the worst case, $O(m.m_i)$.

PLAGRAM uses an optimization that makes this subgraph isomorphism test

linear : it stores with each pattern P a list of the matches of the first edge of P in every target graph G_i. This list is updated in line 8 when generating the extensions. Thus, when the algorithm needs to find the occurrences of P in G_i, it does not need to try every edge of G_i, but only those that are in this list. Therefore, for each occurrence, the cost of the matching is $O(m)$. The number of occurrences of a pattern in a target graph G_i cannot be larger than $2m_i$ (the first edge of P can match each edge of G_i in two "directions"). Therefore, the complexity of computing all occurrences of P in all target graphs is $O(m \sum m_i)$ (which we will bound later with $O(\sum m_i^2)$ in Theorem 2).

We show in the experimental section that this complexity improvement over GSPAN is visible in the measured matching times.

4.3. Extension building (line 8)

For every occurrence of P in a target graph G_i, the algorithm builds possible extensions. This is done by trying to find a valid extension starting from every node of the outer face of P. The complexity of this operation is linear in the total size of P plus the extensions. This is less than $2m_i$ since one edge of G_i is either in an occurrence of P or in at most two extensions. Since there are at most $2m_i$ occurrences of P in G_i, the complexity of building all extensions of all occurrences of P in all target graphs G_i is $O(\sum m_i^2)$.

Every time a new extension is added to the list LE, its frequency is updated. This enables the test in line 10. The LE list is implemented in a way such that the addition of a new extension (together with its frequency counting) is done with a logarithmic complexity (as a function of the number of edges of the extension). Thus, for a fixed pattern P, we can bound this complexity by the total size of all extensions, i.e, $O(\sum m_i^2)$.

According to the conducted experiments, the extension building part of the algorithm was found to be the most expensive one.

Theorem 2 (Complexity)
The total complexity of function mine$(P, \mathcal{D})$ *(excluding the complexity of recursive calls in line 12) is* $O(m^3 + \sum m_i^2)$*, where* m *is the size of the pattern* P *and* m_i *is the size of the target graph* G_i *(in number of edges).*

A consequence of this theorem is that, contrary to general graph mining algorithms as GSPAN, PLAGRAM has a polynomial output delay, i.e., the time between two outputs is polynomial in the size of the input $\sum m_i$.

Theorem 3 (Correctness)
PLAGRAM *finds and outputs exactly once all frequent 2-connected plane subgraphs in* $\mathcal{D}$.

5. Experiments

We now present the computational results obtained by PLAGRAM. Since, to the best of our knowledge, PLAGRAM is the first plane graph mining algorithm, we could not compare it with any other algorithm with the same purpose. Nevertheless, to check how efficient our dedicated plane graph mining algorithm is in comparison with a general-purpose graph mining algorithm, we report here a comparison between PLAGRAM and GSPAN.

The conducted experiments aimed to answer three main questions :

1. How do PLAGRAM and GSPAN scale on video data ?
2. How efficient is PLAGRAM in finding the patterns we are interested in, in comparison with GSPAN ?
3. Can PLAGRAM find meaningful patterns on video data ?

For GSPAN, we asked the authors of Bringmann & Nijssen (2008) for their C++ code. For PLAGRAM, we adapted the source code of GSPAN to implement its features and to allow a fair comparison between them.

The experiments were carried out on a 3.16GHz Xeon X5460 CPU with 16 GB of RAM memory under Debian GNU/Linux (2.6.26-2-amd64 x86_64) operating system.

5.1. Video datasets

The datasets we used for our experiments were created from a set of frames of a synthetic video. The choice of making a synthetic video was beneficial to our experiments, since we did not have to deal with common video artefacts that occasionally disturb the segmentation process. The video had 721 frames in total, with an object (an airplane) appearing in every frame (this helped us to evaluate whether the results of PLAGRAM could be used to track an object in the video, as reported at the end of this section).

After generating the video, we represented each frame as a plane graph. For this task, we used 2 different methods, which led to 2 different datasets of such graphs, as described below :

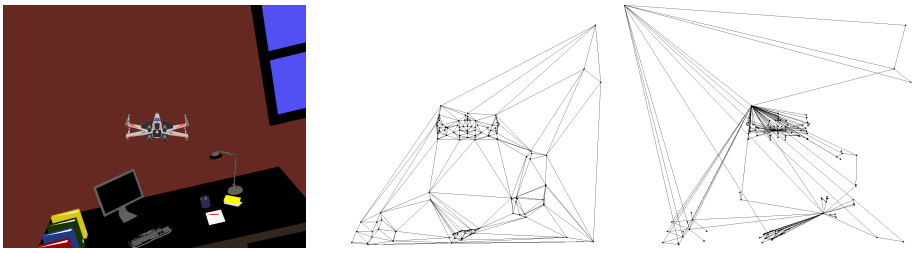


FIG. 5: Example video frame (left) along with its corresponding triangulated (middle) and RAG (right) representations. In the latter, the upper-left node represents the outer region.

– **Triangulation** : assuming that the video frames were already segmented by ther different pexel colors, for each video frame, the barycenters of the segmented regions became nodes and a Delaunay triangulation of this set of nodes was constructed[1]. The final graphs had, on average, 87.58 nodes with an average degree of 2.88. The labels of the nodes were generated based on the size of the regions (in number of pixels). Initially, there were in total 8,293 different sizes. Those were discretized into 10 equal bins, which led to 10 possible node labels. The final set of graphs formed the *Triangulated* dataset.
– **RAG** (Region Adjacency Graphs) : we also represented each frame as a RAG. More precisely, the nodes are the same as in the *Triangulated* dataset, except that there is one more node representing the outer region. An edge exists between 2 regions (or nodes) if these regions are adjacent in the frame. On average, each frame led to a graph with 88.58 nodes, with an average degree of 2.48, and the labels of the nodes were discretized in the same way as for the *Triangulated* dataset. Here, the final set of graphs formed the *RAG* dataset.

An example frame (left) along with its triangulated (middle) and RAG (right) representations is illustrated in Figure 5.

[1]For this task, we used the program available at `http://www.cs.cmu.edu/~quake/triangle.html`.

5.2. Efficiency

Here, we evaluate how efficient our dedicated algorithm is in comparison with the general-purpose one, gSPAN. Several factors may influence the efficiency of our algorithm. As PLAGRAM is dedicated to plane graphs, two patterns that are different for PLAGRAM (due to the order of their edges) may be only one pattern for gSPAN. In this way, our algorithm would find more patterns than gSPAN. However, since our extension building step is restricted to complete faces instead of single graph edges as in gSPAN, we would expect to generate fewer extensions as well as patterns. In any cases, the complexity of our isomorphism test is lower. Therefore, in order to understand the most important of these factors, we considered the following in our experiments :

- The total execution time.
- The number of output patterns.
- The number of generated extensions.

Since we detected that the number of patterns and extensions produced by the algorithms were indeed different, we also considered the following ratios in order to make a fair comparison between the pattern matching and the extension building steps of PLAGRAM and those of gSPAN :

- The ratio of the total pattern matching step time to the total size of the output patterns (in number of edges).
- The ratio of the total extension building step time to the total size of the generated extensions (in number of edges).

Figure 6 presents the results we obtained on the *Triangulated* dataset. In each graph, the x-axis represents absolute minimum supports, which were lowered in units of 10 until the computation times of PLAGRAM reached around 2 hours.

Graph (a) presents the total execution time of PLAGRAM. gSPAN could not finish its executions, even for the highest tested minimum support (in fact, it was interrupted after 3 days of computation). To understand its behaviour, however, we stopped it after 2 hours of execution and plotted here its intermidiate results. The 2-hour executions of gSPAN is referred to here as gSPAN2. Graphs (b) and (c) present, respectively, the number of extensions and the number of output patterns of PLAGRAM and gSPAN2.

Contrary to gSPAN, PLAGRAM finished its executions for every tested support. As presented in graphs (b) and (c), the total execution times increased along with the number of extensions and patterns, respectively. Considering gSPAN2, even for the highest minimum support, the number of extensions

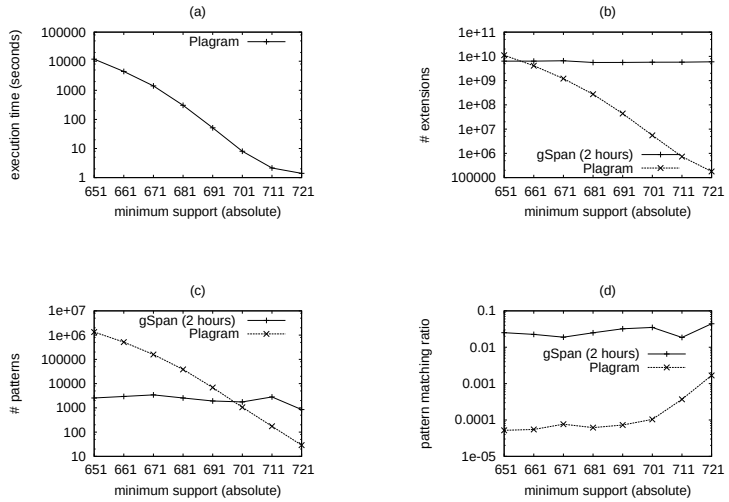


FIG. 6: Efficiency of PLAGRAM and the 2-hour executions of GSPAN on the *Triangulated* dataset.

was several orders of magnitude higher than that of PLAGRAM (because PLAGRAM only considers plane 2-connected graphs). In addition, for the minimum supports of, e.g., 661 and 651, the biggest pattern output by PLAGRAM had, respectively, 45 and 48 edges, while, in the same period of time (two hours), GSPAN2 output fewer patterns with at most 12 edges.

What is worth observing as well are the results given by graph (d). It presents the ratio of the total matching step time to the total size of the output patterns, in number of edges. The ratios were lower for PLAGRAM than for GSPAN2, for every tested minimum support, due to its linear isomorphism test (in the size of the patterns).

Regarding the ratio of the total extension step time to the total size of the generated extensions (in number of edges), PLAGRAM had slightly better results in comparison with GSPAN2.

Concerning the results on the *RAG* dataset, the behaviors of PLAGRAM and GSPAN2 were quite similar to those on the *Triangulated* one. The biggest pattern generated by PLAGRAM had 84 edges for the lowest tested minimum support (around 2 hours of execution), while for GSPAN2 it had only 12 edges.

5.3. Step Times

On our datasets, the extension building step of GSPAN2 was on average 90% of the total execution time, whereas the matching step was always less than 5%, and the canonical-test step was negligible. For PLAGRAM, the most expensive step was also the extension building step, which varied from 50% to 75% of the the execution time. The pattern matching step, in its turn, varied from 20% to 40%, while the canonical-test step was always less than 5%.

In conclusion, we believe that the main reason why PLAGRAM is more efficient than GSPAN is the lower number of extensions produced by PLAGRAM rather than only the faster pattern matching as one could expect.

5.4. Meaningfulness of Obtained Patterns

Computer vision researchers are struggling for more than 30 years with the problem of recognizing objects in images and more recently with the problem of tracking objects in videos (A. Yilmaz & Mubarak (2006)). In this context, to evaluate how meaningful the patterns found by PLAGRAM are, we study here whether they can be used to track an object in a video. We start by introducing two measures which assess how precise a pattern p corresponds to a given object o in the video frames. These measures are adaptations of the popular measures *precision* and *recall*, as described below :

- **precision** : fraction of the occurrences of p (in the target graphs) of which every node maps to o in the corresponding video frames. The intuition behind this measure is to evaluate the *purity* of p, that is, p has the maximum precision if it maps only to o and nothing else.
- **recall** : Let n be the number of frames in which o is present. The recall is defined as the fraction of n in which every node of p maps to o. Here, the intuition is to evaluate the *completeness* of p. The idea is to check whether p maps to all occurrences of o in the set of video frames.

Since PLAGRAM is an exhaustive algorithm, that is, it mines for all frequent patterns in the graph database, the mining result may consist of different patterns corresponding to different objects, or even to any specific one (w.r.t. to the proposed measures). Therefore, to follow a specific object in the video, the user should be able to select from the entire set of output patterns those that correspond to this object. A basic strategy for this task is the following :

1. First, the user selects a frame area where there exists an object he or she is interested in tracking. This is done in the first frame, referred to here

	Triangulated		*RAG*	
Support	precision (%)	recall (%)	precision (%)	recall (%)
721	96.2	99.8	97.1	99.9
711	97.6	98.9	97.2	99.8
701	99.3	97.7	96.5	99.0
691	99.7	96.3	93.9	95.6
681	99.8	95.0	92.8	93.9
671	99.8	93.7	92.5	93.5
661	99.9	92.5	92.5	93.5
651	99.9	91.0	91.8	92.6

TAB. 1: Average precision and recall (in percentage) computed for the patterns selected at step 3 of the proposed object tracking strategy.

as f, where this object occurs.

2. Afterwards, the user starts the graph mining process (by executing PLAGRAM with a given minimum support as input).
3. Next, the idea is to post-process the results by selecting the frequent patterns of which every node maps to the user-selected area in f.
4. Finally, all occurrences of the patterns selected in the previous step are mapped to the video frames after f, allowing the user to detected the position of the selected object through the video.

To evaluate this strategy, we checked whether it would be possible to follow the airplane in our video, by considering the patterns obtained in the experiments of Section 5.2. As might be expected, those patterns had different precision and recall w.r.t to the airplane. After executing step 3, we got the patterns whose average precision and recall (in percentage) are shown in Table 1.

Observe that the selected patterns had, on average, very good quality, making step 4 successful. Considering the *Triangulated* dataset, the average precision increased in inverse proportion to the minimum supports, while the average recall decreased with the minimum supports. Here, lower minimum supports led to bigger patterns with higher precision and lower recall. In the *RAG* dataset, the behavior was different : big patterns had nodes that did not map to the airplane. In addition, small patterns with low support did not have good precision nor recall. As a consequence, the average precision and recall decreased with the minimum support.

6. Conclusions

We presented PLAGRAM, an efficient frequent graph mining algorithm that is dedicated to mining plane 2-connected patterns. Conducted experiments

showed that PLAGRAM is able to efficiently run on graph-based video datasets, on which a general-purpose graph mining algorithm failed to finish its computations. The experiments also showed that besides providing an efficient algorithm, the 2-connectedness restriction does not limit the meaningfulness of the final patterns. On the contrary, we believe that PLAGRAM may be a useful tool to track objects in videos.

Having the example video applications in mind, we have identified three directions for further work : first, the idea is to use graphs where each node is associated with one or more labels (also referred to as attribute graphs). Two nodes will then be considered equivalent if at least a given fraction of their labels are equal. The same can be applied to edges. Second, to track an object in a video, we proposed a strategy in which one should post-process the entire set of frequent patterns to select those that map to the object of interest. An interesting way to enhance the efficiency of this strategy is to have these patterns as a constraint to the mining process. Finally, we currently represent a video as a set of graphs, not taking into account the order in which these graphs (frames) appear in a video. By considering this order, interesting constraints on the trajectory of objects or simply on their positions can be considered during the mining process.

Références

A. YILMAZ O. J. & MUBARAK S. (2006). Object tracking : A survey. *ACM Comput. Surv.*, **38**(4), 13+.

BRINGMANN B. & NIJSSEN S. (2008). What is frequent in a single graph ? In *PAKDD*, p. 858–863.

CHANG R.-F., CHEN C.-J. & LIAO C.-H. (2004). Region-based image retrieval using edgeflow segmentation and region adjacency graph. In *IEEE ICME*, p. 1883–1886.

DAMIAND G., DE LA HIGUERA C., JANODET J.-C., SAMUEL E. & SOLNON C. (2009). Polynomial algorithm for submap isomorphism : Application to searching patterns in images. In *Workshop on Graph-based Representation in Pattern Recognition (GBR)*, volume 5534, p. 102–112.

KURAMOCHI M. & KARYPIS G. (2001). Frequent subgraph discovery. In *IEEE ICDM*, p. 313–320.

NIJSSEN S. & KOK J. N. (2004). A quickstart in frequent structure mining can make a difference. In *ACM SIGKDD*, p. 647–652.

YAN X. & HAN J. (2002). gspan : Graph-based substructure pattern mining. In *IEEE ICDM*, p. 721–724.