Mining interesting frequent patterns in a single graph using inexact matching

Por

Marisol Flores Garrido

Tesis sometida como requisito parcial para obtener el grado de

DOCTOR EN CIENCIAS EN LA ESPECIALIDAD DE CIENCIAS COMPUTACIONALES

en el

Instituto Nacional de Astrofísica, Óptica y Electrónica.

Abril de 2015
Tonantzintla, Puebla

Supervisada por:
Dr. Jesús Ariel Carrasco Ochoa, INAOE

Instituto Nacional de Astrofísica, Óptica y Electrónica

Mining interesting frequent patterns in a single graph using inexact matching

by

Marisol Flores-Garrido

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Coordinación de Ciencias Computacionales

March 2015

"Where is the wisdom we have lost in knowledge?
Where is the knowledge we have lost in information?"

T. S. Eliot

Abstract

Frequent graph pattern mining is an important problem in diverse research fields but, until recently, algorithms that focus on the problem had only used graph isomorphism to identify occurrences of a given pattern. In the last years, however, a few works have focused on the case where a pattern could differ from its occurrences, which is a scenario that corresponds better to reality and that could be important, for example, to analyze noisy data. Although these algorithms allow differences in labels and structural differences in edges, none of them considers structural differences in vertices. How can we identify occurrences that differ by one (or several) nodes from the pattern they represent?

Our work approaches the problem of frequent graph pattern mining with three main characteristics. First, we use inexact matching, allowing structural differences in both edges and vertices. Second, we focus on the problem of mining patterns in a single graph, a problem that has been less explored than the case in which patterns are mined from a graph collection. Third, instead of mining the whole set of frequent patterns, we are interested in obtaining an *interesting* subset of patterns, thus lessening redundancies and facilitating the analysis of the output set.

We introduce the algorithm AGraP, which, by allowing patterns that can have structural differences, in vertices as well as in edges, respect to their occurrences, is able to find patterns missed by other state of the art algorithms. Then, we introduce the algorithms CloseAFG, MaxAFG and IntAFG that, by focusing on closed, maximal or interesting patterns, respectively, are able to reduce the amount of mined patterns by AGraP, while retaining new potentially useful information found by allowing structural differences in vertices. Finally, we show, through experimental results, that the "extra" patterns found by our proposed algorithms can help in tasks of classification and clustering, thus suggesting that useful information about the input data can be captured through the patterns mined by our algorithms.

Resumen

La minería de patrones frecuentes en grafos es un problema importante en diversas áreas pero, hasta hace apenas unos años, los algoritmos enfocados a resolver este problema se habían limitado a usar isomorfismo de grafos para identificar las ocurrencias de cada patrón. Recientemente, algunos trabajos han abordado el problema considerando el caso en el que un patrón pudiera tener algunas diferencias respecto a sus ocurrencias, un escenario que corresponde bien a la realidad y que pudiera ser importante, por ejemplo, cuando se analizan datos con ruido. Aunque dichos algoritmos permiten diferencias en las etiquetas de los vértices y diferencias estructurales en aristas, ninguno de ellos permite diferencias estructurales en vértices. ¿Cómo podríamos identificar ocurrencias que difieren en uno, o varios, vértices del patrón que representan?

En este trabajo abordamos el problema de minería de patrones frecuentes en grafos considerando tres características importantes. Primero, se emplea correspondencia inexacta al momento de comparar patrones, permitiendo diferencias estructurales tanto en vértices, como en aristas. Segundo, la búsqueda de patrones se realiza en un solo grafo, un problema que ha recibido menos atención que la búsqueda de patrones en una colección de grafos. Tercero, no se desea obtener el conjunto de todos los patrones frecuentes, sino un subconjunto *interesante* de patrones, que disminuya la redundancia del conjunto de salida y facilite su análisis posterior.

Con esto en cuenta, presentamos el algoritmo AGraP, que, al permitir patrones que pueden tener diferencias estructurales, en vértices y aristas, respecto a sus ocurrencias, encuentra patrones que pasan desapercibidos cuando se usan otros algoritmos del estado del arte. Posteriormente, presentamos los algoritmos CloseAFG, MaxAFG e IntAFG, que, respectivamente, se enfocan en patrones cerrados, maximales e interesantes, reduciendo la cantidad de patrones encontrados por AGraP y, al mismo tiempo, conservando información nueva, implícita en los patrones que se identificaron al permitir diferencias estructurales en vértices. Finalmente, mediante resultados experimentales, mostramos que los patrones "extra" que los algoritmos propuestos encuentran, pueden ayudar en tareas de clasificación y agrupamiento, sugiriendo que, efectivamente, existe informacin útil sobre los datos analizados que se puede capturar mediante los patrones que nuestros algoritmos encuentran.

Acknowledgements

Quiero expresar mi agradecimiento a las personas que de una forma u otra hicieron posible este trabajo. Ante todo, debo gratitud a mi supervisor, el Dr. Ariel Carrasco Ochoa, por su dirección a lo largo de estos años, por las conversaciones siempre disfrutables y porque su apoyo fue fundamental para llevar a cabo la investigación y enfrentar algunas crisis en el proceso. Gracias también al Dr. Francisco Trinidad Martínez por su colaboración, su amabilidad y por compartir generosamente su conocimiento.

Asimismo, agradezco a los miembros de mi comité doctoral, los doctores Pilar Gómez, Enrique Sucar, Eduardo Morales y Carlos Reyes, por sus observaciones y sugerencias en cada etapa del trabajo.

Doy las gracias a mi familia, Xavier, Lulú, Nats y Xavo, por su amor y apoyo constante sin importar qué suceda. Sólo soy capaz de emprender aventuras porque sé que cuento con ustedes.

Muchísimas gracias a Juan Aldebarán y a Adrián por su amistad y porque fueron, respectivamente, la primera y la última persona que me acompañó en esta etapa poblana, rodeada de maletas y estrés por el cambio. Gracias por la ayuda y por ese cariño que se hace tangible cuando realmente importa.

Gracias, también, a los amigos que estuvieron conmigo en esta etapa: Manuel, Saúl y Valdo (mi familia poblana inicial), y a Emmanuel, César, Miguel, David, Ana, Pastor, Rosales, Andrea, Gabo, el grupo de Trecegen, el grupito chido de astrofísicos, ópticos y electrónicos. En particular, gracias a Rigo por todo su apoyo para realizar los trámites al final de este trabajo. Gracias, también, a los que, aunque en otra ciudad, no dejan de ser cercanos: Diane, Lacho, Cpi, Lluvia. Y gracias, de manera muy especial, a Anaely, Emmaly, Gaby y Majandy porque, para mí, el último año fue uno de los más felices. Creo que, en este tiempo y lugar, todavía es difícil desafiar convenciones sociales y tener el valor para ser lo que honestamente se quiere ser; por eso las admiro y me ilusiona pensar en nuestra agenda de churros y cambios al mundo.

Finalmente, agradezco al Consejo Nacional de Ciencia y Tecnología de México por la beca recibida para llevar a cabo mis estudios de doctorado en el INAOE.

Contents

List of Figures

List of Tables

A Humberto, porque en verdad creo que alguien que enseña puede cambiar el mundo

Chapter 1

Introduction

The core idea of this research work is using inexact matching to find *new* patterns in a single graph, while trying to avoid an overwhelming amount of them.

We live in an age where data has become an essential resource in every field, from science to economy or even the social sphere. The advances both, in computing resources and data collecting and storage, have brought an increment to the data available to institutions and individuals, and, as the data becomes more complicated, new tools to uncover hidden patterns and extract potentially useful information are required; we propose one of those tools.

Our research problem is within the context of the, more general, frequent pattern mining problem. Let us recall that frequent patterns are itemsets, subsequences, or substructures that appear in a dataset with frequency no less than a user-specified threshold. Mining frequent patterns is an essential problem in Data Mining; once the patterns are identified, they can be used to analyze associations, correlations and other relationships among data, as well as to perform many important data mining tasks, like indexing, classification and clustering, among others.

While mining patterns is never a trivial problem, it becomes specially challenging when the patterns are mined from graphs. The inherent ability of graphs to represent structural information makes them a powerful modeling tool, used in areas ranging from chemoinformatics [Mahé et al., 2005] to web analysis [Schenker et al., 2004], but it also makes their analysis difficult. Graph mining algorithms face challenges concerning the combinatorial explosion in the number of subgraphs to be analyzed in order to identify patterns, as well as the high-cost of some of the

steps involved in the process, like solving the graph isomorphism problem, which is believed to be NP-complete [López-Presa, 2009].

Grosso modo, the frequent mining problem consists of three main steps: (1) generating pattern candidates, (2) identifying the occurrences of each candidate, and (3) establishing each candidate support to determine if it satisfies the given frequency threshold. Concerning the first step, we could divide the existing algorithms into apriori-based and pattern growth-based approaches; both of them have been explored in the literature. For the last step, there is a well established definition of support when working in a collection of graphs, and a broader, less-explored, notion for the single-graph case. However, it is in the second step where it can be noticed a significant gap in the amount of attention that different paradigms have received: although the use of inexact matching in real life problems has been greatly acknowledged, the great majority of the algorithms have based the identification of pattern occurrences on the graph isomorphism problem.

Beyond the computational cost, using the graph isomorphism problem in mining algorithms represents a conceptual constraint in the type of patterns mined. Although, for efficiency reasons, most algorithms try to reduce the number of times the problem is solved, the idea remains that a subgraph must match a pattern *exactly* in ordert o be considered an occurrence.

What if, in some context, two graphs could be considered to match despite having some differences? SUBDUE [Cook and Holder, 1994] was the first algorithm to allow mining approximate subgraphs, although its heuristic nature does not guarantee to find all the frequent patterns. More than a decade later, the question on using inexact match was addressed again in a handful of works. First, Chen et al. [2007] studied the case in which a pattern and its occurrences could differ by edges; this case, they explain, could be relevant when studying problems with inherent noise and data diversity. Later, Jia et al. [2011] described an scenario in which, due to noisy data, labels in graphs could be wrongly assigned and they proposed an algorithm to find patterns so that two graphs could match – be "approximately isomorphic" in their words – despite differences in labels. Finally, Acosta-Mendoza et al. [2012] generalized the work of Jia et al. to include edge mislabelling.

Although the later works use inexact matching in order to mine frequent graph patterns, they still require a bijection between the vertices for matching graphs. In fact, to the best of our knowledge, there is no algorithm reported in the literature

aiming to find a complete set of approximate graph patterns by allowing structural differences in vertices and edges. We argue that allowing structural differences, in vertices as well as in edges, in the mining process can be important in order to bring out new information on the analyzed data.

However, while we focus on finding patterns that can be unnoticed when using other algorithms, we are also concerned on the amount of discovered patterns, since having a large set of patterns can be overwhelming and render the user unable to take full advantage of the discovered information. Managing a big amount of output information is a challenge itself, thus, although we bring an increment on the amount of mined patterns by using inexact matching, we also propose a way to reduce the pattern set, in an effort to balance new information and usability.

Finally, we chose to mine patterns in a single graph, which, as explained ahead, offers different challenges than mining patterns in a graph collection.

The formal statement of our research problem is presented next.

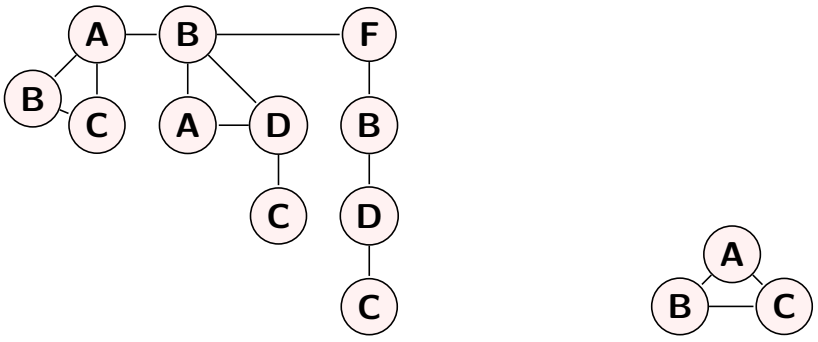
1.1 Research problem

The problem we approach in this thesis can be formally stated as:

> *Obtaining frequent patterns in a single graph using inexact matching, in such a way that the patterns are not only frequent but also interesting.*

This problem can be viewed as composed by three characteristics that define it and represent the research lines explored in our work.

First, we are interested in searching patterns in a single graph. As stated before, the problem of pattern mining in a collection of graphs has been widely studied and efficient algorithms have been developed to solve it [Gago-Alonso et al., 2008, Nijssen and Kok, 2004, Ranu and Singh, 2009, Yan and Han, 2002], but the problem of pattern mining in a single graph has been less explored. One of the main challenges in the single graph case is to define the support of a pattern; whereas in a graph collection the support of a pattern is defined as the number of graphs that contain a pattern occurrence in the collection, in a single graph, due to a possible overlapping, using as support the number of a pattern occurrences leads to the



(A) G (B) P

FIGURE 1.1: *Using a frequency threshold $\sigma = 2$, the pattern P cannot be identified in G using graph isomorphism, but it can when inexact matching is used instead.*

lost of the antimonotonicity property (Section 2.1.3) and, therefore, traditional approaches to prune the search space cannot be applied. In order to maintain the antimonotonicity property, several support definitions have been proposed in the literature [Bringmann and Nijssen, 2008, Chen et al., 2007, Fiedler and Borgelt, 2007, Kuramochi and Karypis, 2001] but it is clear that the problem poses challenges different from those in the graph collection case. Despite being less studied, mining frequent patterns in a single graph is important because it can be useful in different applications. Frequent subgraphs have been used, for instance, in graph indexing [Yan et al., 2004], in the discovery of symbols (and their relations) for classification and indexation of document images [Barbu et al., 2005], in the identificaton of peer influence groups in networks [Moody, 2001], in finding patterns in biological networks that can be later used to derive functional, structural and evolutionary information [Koyutürk et al., 2004], among others [Alguliev et al., 2011, Leskovec, 2009].

Second, we want to use inexact matching to identify the occurrences of a given graph pattern, i.e., we want to allow differences between a candidate graph and the subgraphs considered as its occurrences; moreover, we want to allow structural differences not only in edges, but also *in vertices*. An algorithm that allows differences is important when the examined data come from real world measurements and, therefore, could be noisy; but, beyond a potential contaminated source of data, allowing differences between matching graphs allows identifying patterns that, otherwise, would be missed (Fig. 1.1 shows an example).

Third, we have interest in mining only a subset of the whole set of frequent graph patterns; we want to find frequent patterns that are *interesting*. Algorithms like

Gaston [Nijssen and Kok, 2004] and gSpan [Yan and Han, 2002] are able to find the complete set of frequent patterns, but how useful is to have them *all*? The amount of patterns could be prohibitively large, generating a set so large that it is not really informative or useful for the analysis of the input data. Algorithms like SPIN [Huan et al., 2004], MARGIN [Thomas et al., 2010] and CloseGraph [Yan and Han, 2003] deal with this problem by focusing on the search of close and maximal patterns, but, although this approach does decrease the number of patterns, it could be desirable to reduce further the set in favor of practicality. For this reason, in recent years there has been a shift in the field of graph data mining, from focusing on getting the complete set of frequent patterns to getting a smaller set of patterns that can be considered representative of the whole set. Following this trend, we would like to get a set of frequent graphs that can be considered representative of the input graph and that, being smaller than the complete set of frequent graphs, can be easier to analyze and, therefore, more useful for getting an insight of the data.

1.2 Objectives

The general objective of this research work is to develop an algorithm to obtain interesting frequent patterns from a single graph, using inexact matching and allowing structural differences in edges and vertices.

The specific objectives of our work are:

1. Define a dissimilarity measure between graphs that allows structural differences in edges as well as in vertices, and that takes into account the size of the graphs being compared.

2. Propose a strategy that, using the dissimilarity measure previously defined, allows graph pattern mining with inexact matching.

3. Define a support measure to determine the frequency of a pattern in a graph, considering inexact matching between graphs.

4. Integrate and implement an algorithm to mine frequent subgraphs in a single graph, using the strategies and measures previously established.

5. Design and implement a strategy to get graphs that, besides being frequent, are interesting, at least within the context of a particular application.

6. Evaluate the quality of the mined patterns within the context of data mining tasks like supervised classification and clustering.

1.3 Contributions

In this work, we present an algorithm, AGraP, able to mine graph patterns using inexact matching, allowing structural differences in vertices and edges. Except SUBDUE [Cook and Holder, 1994], which is not focused on finding all frequent patterns, no other graph mining algorithm allows matching patterns and occurrences that do not hold a one-to-one correspondence between their vertices. Given a frequency and a dissimilarity threshold, AGraP uses a depth-first search together with an strategy to identify patterns occurrences that could have missed vertices (by keeping occurrences that are unable to grow, but satisfy the dissimilarity threshold), "extra vertices" (by replacing the constraint of having an edge between an occurrence and a new vertex by the requirement of having a path) or differences in edges. Also, label differences are allowed and managed through a dictionary, that specifies which label substitutions are valid. By using inexact matching, we are able to find patterns, otherwise missed, that capture potentially useful information about the examined data (as shown in our experiments).

We also propose two dissimilarity graph measures, one of them used in AGraP, that are based on the edit distance [Gao et al., 2010] and are compatible with structural differences between graphs.

Focusing on decreasing the amount of patterns, we propose MaxAFG and CloseAFG, two modifications of AGraP that find maximal and closed patterns, respectively, in a single graph, using inexact matching.

Finally, aiming at further reducing the amount of patterns in the output set, we propose the greedy algorithm IntAFG that, given a pattern set, gets a subset of *interesting* patterns. We obtain said subset based on the idea of having a subset of patterns covering the original pattern set, while lessening redundancy among selected patterns. We use IntAFG, in a post-mining step, to reduce the output set of AGraP, in order to obtain a set of interesting frequent approximate patterns.

Besides experiments where we explore the number and nature of patterns that our proposed algorithms discover, we also explore the use of the patterns mined by our algorithms in clustering and classification tasks, getting results that suggest that there is useful information captured by allowing structural differences in vertices.

1.4 Thesis organization

The following chapters are organized as follows: Chapter 2 contains the basic notions to understand the problem that we approach, and summarizes the related work in each of the areas that compose our problem. Chapter 3 presents our proposed ideas to mine patterns in a single graph allowing structural differences in both, edges and vertices, and introduces our proposed algorithm AGraP, showing experiments that confirm that, by using inexact matching, it is able to find new patterns. In Chapter 4, we deal with the issue of reducing the set of mined patterns, thus, we present the algorithms MaxAFG and CloseAFG, and a greedy algorithm, IntAFG, to select interesting patterns and we show the effectiveness of the proposed algorithms to reduce the number of patterns. In Chapter 5, we address the question of the usefulness of the patterns we found through the use of inexact matching and we show, by performing tasks of clustering and classification, that the patterns found though our approach are able to capture potentially useful information on the analyzed data. Finally, Chapter 6 presents some conclusions and future research directions.

Chapter 2

Related work

We begin this chapter by introducing some graph basic concepts and the notation used in further sections. Then, we present the work related to our research problem. As far as we know, there is no algorithm in literature that focuses on the same problem, thus we describe the related work by presenting algorithms that approach one of the three aspects that compose our problem: pattern mining in a single graph, pattern mining through inexact matching, and, finally, interesting graph patterns.

2.1 Basic concepts

2.1.1 Notation

Since we work with *labeled graphs*, we consider a graph to be a quadruple $G = (V, E, \mathcal{L}, \psi)$ where V is a set of vertices, E is a set of edges $E \subseteq \{\{u, v\} | u, v \in V, u \neq v\}$, $\mathcal{L}$ is a finite set of labels and ψ is a function assigning a label in $\mathcal{L}$ to each vertex and edge. We will use the notation $V(G)$, $E(G)$ and ψ_G to refer to the set of vertices, the set of edges and the labeling function of a graph G, respectively.

A graph H is said to be a *subgraph* of G, denoted by $H \subseteq G$, if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and $\forall x \in V(H) \cup E(H)$ we have $\psi_H(x) = \psi_G(x)$. Let $V' \subseteq V(G)$ be a subset of vertices of G; the subgraph of G induced by V' is the subgraph G' such that $V(G') = V'$ and for all $u, v \in V'$ we have $\{u, v\} \in E(G')$ iff $\{u, v\} \in E(G)$.

Finally, we use the symbol $\diamond$ to denote the concatenation of a subgraph and a new vertex, i.e., given a subgraph H of G and a vertex $v \in V(G)$, we denote by $H \diamond v$ the subgraph in G induced by $V(H) \cup \{v\}$.

2.1.2 Graph matching

Graph matching refers to the process of evaluating the similarity (or dissimilarity) of two graphs [Riesen et al., 2010]. *Grosso modo*, there are two types of graph matching: exact matching and inexact matching. In the first one, for two graphs to be considered to match, it is required that they are isomorphic.

Two graphs G_1 and G_2, are isomorphic, denoted $G_1 \simeq G_2$, if there exists a bijection $f : V(G_1) \mapsto V(G_2)$ that is structure preserving, i.e., $\{u, v\} \in E(G_1)$ if and only if the $\{f(u), f(v)\} \in E(G_2)$; in this case, the function f is called isomorphism. Similarly, the graphs are subisomorphic if there exists an isomorphism between G_1 and a subgraph of G_2.

In *inexact* matching the previous constraints are relaxed and, although there is not an specific definition of it (unlike exact matching), the focus shifts to assess the similarity between graphs. In this way, two nonidentical graphs can be considered to match (in an inexact way). How to compare graphs when using inexact matching is an extense topic that includes a variety of approaches, like solving an optimization problem, using spectral properties of graphs or computing the graph edit distance [Foggia et al., 2014, Riesen et al., 2010].

2.1.3 Frequent graph pattern mining

A subgraph g is frequent if its support is greater than some predefined threshold value, σ. The support of a subgraph refers to its occurrence counting, which, according to Jiang et al. [2013], can be computed using either transaction-based counting or occurrence-based counting. In the former, applicable only when the mining takes place in a collection of graphs, the support is defined by the number of graphs of the collection (transactions) in which g occurs, no matter how many times it appears in each graph. Thus, given a collection of graphs, $D = \{G_1, G_2, \ldots, G_n\}$, the support of g is defined as $supp(g) = |D_g|/|D|$, where D_g represents the support graph set of g defined as $D_g = \{G_i | g \subseteq G_i, G_i \in D\}$.

In the occurrence-based counting, a frequency measure must be established, which ideally maintains the antimonotonicity, or *Downward Closure Property*, which establishes that if a graph is frequent, then all of its subgraphs will also be frequent. Some of the support definitions following this approach are presented in Section 2.2.

In general, *frequent graph pattern mining* refers to the problem of, given a frequency threshold, finding the frequent subgraphs either in a single graph or in a collection of graphs.

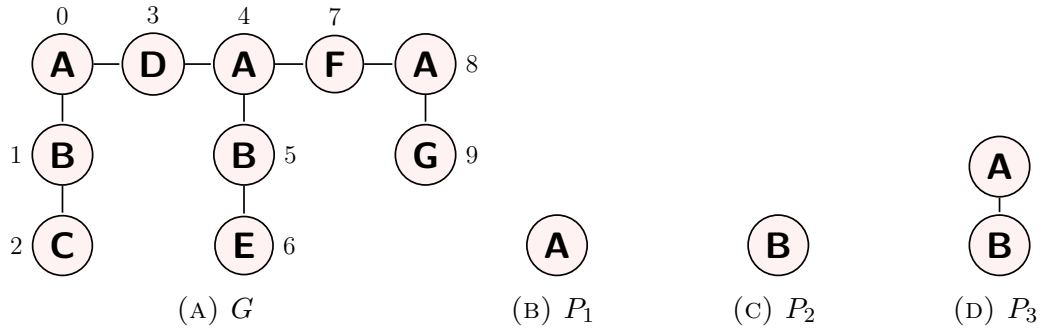
2.1.4 Maximal and closed graphs

Maximal and closed patterns have been used to condense information in frequent pattern mining. A definition of these kinds of patterns is offered next.

Closed subgraph. A subgraph g is a closed frequent subgraph in D (a collection of graphs or a single graph), if g is frequent in D and there does not exist a subgraph g' such that $g \subseteq g'$ and g' has the same support as g in D.

Maximal subgraph. A subgraph g is a maximal frequent subgraph in D (a collection of graphs or a single graph) if g is frequent in D and there does not exist a subgraph g' such that $g \subseteq g'$ and g' is frequent in D.

Fig. 2.1 shows an example of these kinds of patterns. Let us consider the graph G and a frequency threshold $\sigma = 2$, and assume that a pattern's support is given by simply counting the number of its non-overlapping exact occurrences. In that scenario, the set of frequent patterns would consist of P_1 ($supp(P_1) = 3$), P_2 ($supp(P_2) = 2$) and P_3 ($supp(P_3) = 2$). The pattern P_1 is a closed pattern, since it cannot grow without decreasing its support; although P_3 is a supergraph of P_1 and is still frequent, it has a lesser support. The pattern P_3 is also a closed pattern, however, P_2 is not closed, because it *can* grow into P_3 preserving its suppport. Thus, the set of closed patterns in G is $\{P_1, P_3\}$. Regarding maximal patterns, we have that P_1 and P_2 are not maximal, because they can grow into P_3 (which is frequent), whereas P_3 is maximal because it cannot grow into a pattern that still satisfies the frequency threshold. Thus, the set of maximal patterns in G is $\{P_3\}$.



(A) G (B) P_1 (C) P_2 (D) P_3

FIGURE 2.1: *Using a dissimilarity frequency $\sigma = 2$, both patterns, P_1 and P_3 are closed patterns but only P_3 is a maximal pattern.*

As it can be noticed, the set of maximal patterns is always a subset of the set of closed ones.

Closed (maximal) graph pattern mining refers to the problem of, given a frequency threshold, finding all the frequent subgraphs, either in a single graph or in a collection of graphs, that are also closed (maximal).

2.2 Pattern mining in a single graph

Algorithms for mining graph frequent patterns can be classified according to different criteria, like the search strategy (breadth or depth), the output pattern set (all frequent patterns or only some of them) or the analyzed data (a collection of graphs or a single one).

The last criterion establishes two clearly distinct scenarios for the problem, with different challenges. When analyzing a collection of graphs, a problem defined in the previous section, there are well known algorithms to mine frequent patterns, like SUBDUE [Holder, 1988], gSpan [Yan and Han, 2002], CloseGraph [Yan and Han, 2003], MoFa [Borgelt and Berthold, 2002], Gaston [Nijssen and Kok, 2004], gRed [Gago-Alonso et al., 2008] and GraphSig [Ranu and Singh, 2009]. A description of these algorithms, the differences among them and their advantages can be found in [Cheng et al., 2010] and [Krishna et al., 2011].

Although it is a problem that appears in several applications [Alguliev et al., 2011, Leskovec, 2009, Lima et al., 2012, Moody, 2001], mining patterns in a single graph has been less explored than mining patterns in a graph collection. Furthermore, it is known that algorithms that mine patterns in a single graph can be adapted,

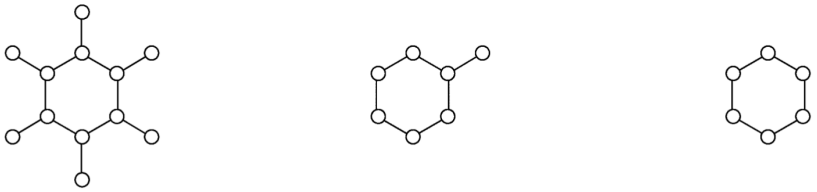
relatively easily, to the problem of mining a graph collection; the core idea to this end is that for each graph in the collection it would be possible to identify if a given pattern has at least one occurrence in the graph. On the other hand, algorithms that look for patterns in a collection of graphs cannot be directly applied to the single graph scenario [Kuramochi and Karypis, 2005].

The main difference between both scenarios lies in the way of counting the pattern support. In the single graph case, the pattern occurrences could be overlapped and, if it happens, it is necessary to establish how many occurrences should be taken into account, since an inadequate counting could bring difficulties to the traversal of the search space. The lack of an strategy to work with overlapping pattern occurrences is one of the reasons why the algorithms designed for graph collections do not work in the single graph case.

Ideally, the function used to calculate the pattern support in a single graph should satisfy the antimonotonicity property, which is important because it allows pruning the search space. The algorithms that mine frequent subgraphs start by considering graphs with size 1, i.e., vertices, and then grow the subgraphs. For each subgraph, the support is calculated; if it is less than a given threshold, the subgraph is discarded so that only frequent subgraphs are used to generate other (bigger) subgraphs. Pruning the search space in this way is possible because we know that throwing away non frequent subgraphs does not affect the search, since, because of the antimonotonicity property, there cannot be a discarded-graph's supergraph with bigger support.

Establishing a support measure for patterns in a single graph is not a trivial issue. If overlaps are ignored and we just count the occurrences of a pattern (subgraph), the antimonotonicity property is lost. This can be seen in Fig. 2.2, which shows an example provided by Kuramochi and Karypis [2005]. The structures G^7 and G^6 are both subgraphs of G; however, despite the fact that G^6 appears only once in G and is a subgraph of G^7, we find that G^7 has six occurrences in G.

Kuramochi and Karypis [2004a] proposed the algorithms Hsigram, Vsigram and GREW to mine frequent patterns in a single graph (a work extended in [Kuramochi and Karypis, 2005]). Hsigram and Vsigram differ only in the search strategy they use (breadth-first and depth-first, respectively), but both of them use vertex labeling to order the subgraphs and avoid repeating comparisons; in this way they achieve efficiency. The authors follow the formulation of the frequent subgraph



(A) G (B) G^7 (C) G^6

FIGURE 2.2: *These patterns would not satisfy the antimonotonicity property if an occurrence-based counting were used, without taking into account overlapping. The graph G^6 is a subgraph of G^7, yet it has a lesser frequency in G. This example was taken from [Kuramochi and Karypis, 2005].*

discovery problem given by Vanetik et al. [2002], where a pattern must have at least σ *edge-disjoint* embeddings in a graph, in order to be frequent; thus, to compute a pattern support, the algorithms construct the overlap graph of the pattern occurrences (a graph with a vertex for each occurrence and an edge joining each pair of occurrences that have overlap) and define the pattern support as the cardinality of the maximum independent set (MIS) of the overlap graph. In Graph Theory, the MIS of a graph is the largest set of vertices such that none of them are adjacent in the graph. This idea to compute the support, however, results in an expensive procedure, since finding the MIS of a set is an NP-hard problem [Robson, 1986].

Later on, the authors propose a new algorithm, GREW [Kuramochi and Karypis, 2004b], which uses a heuristic approach (greedy) and, although it does not find the complete set of frequent patterns in a graph (unlike Hsigram and Vsigram), it gains efficiency with respect to the other two algorithms.

Fiedler and Borgelt [2007] follow the idea of finding the MIS of an overlap graph, but suggest a definition that allows certain kinds of overlap, as long as they are not *harmful*, i.e., do not lead to the lost of antimonotonicity. Later, Bringmann and Nijssen [2008] suggest a support measure that does not require either the construction of the overlap graph or finding a MIS, and that, therefore, is less computationally expensive; given a graph G and a pattern $P = (V, E)$, the support of a pattern is defined as

$$supp(P, G) = \min_{v \in V(P)} |\{\varphi(v) : \varphi \text{ is a mapping between } P \text{ and one of its occurrences in } G\}| .$$

Recently, Hellal and Romdhane [2013] proposed a support measure based on the

Size of Maximum Non Overlapping Embedding Set (SMNOES). Given a pattern P and its occurrences in an input graph G, Hellal and Romdhane construct a new graph whose vertices represent P occurrences and whose edges join occurrences that do not overlap, i.e., do not share any vertex in G. Then, they look for the maximum clique of this new graph (this requires time in $O(2^n)$ for a graph with n vertices); the number of nodes of the maximum clique represents the support of P. The SMNOES support measure holds the antimonotonicity property, which the authors use in their proposed algorithm, NODAR, to mine patterns in a single labeled graph.

SUBDUE [Holder et al., 1994] uses the *Minimum Description Length* principle, a greedy algorithm and beam search, to mine patterns in a single graph, but its goal is not finding frequent patterns but those that better compress the input graph. The algorithm identifies the structures that allow the best compression of the original graph and replace them by single vertices to, then, repeat the procedure. Thus, SUBDUE does not guarantee to find all the frequent structures, since it does not focus on that problem.

All of the mentioned algorithms identify the occurrences of a given pattern by solving the graph isomorphism problem. This problem is clearly NP but, until now, it remains unknown if it is NP-complete [López-Presa, 2009] (although it is widely believed so); as a consequence, the algorithms use a data preprocessing stage and try to solve the isomorphism problem as few times as possible.

SUBDUE allows to find approximate patterns in a single graph, but it uses a constrained beam search method and, due to its nature, could miss some frequent patterns. On the other hand, the main difference between the rest of the described algorithms and the one we propose lies on the use of inexact matching, which makes it unnecessary to solve the isomorphism problem and allows us to find patterns missed when using exact matching. The only work in literature that mines all frequent patterns in a single graph using inexact matching, in a similar way to the one we propose, is gApprox [Chen et al., 2007], which is described in the next section.

2.3 Inexact patterns

Inexact matching has been recognized as a very important research topic in graph-based data mining, to the extent that, according to Foggia et al. [2014], "the recent work on graphs in the Pattern Recognition community has been mostly focused on inexact graph matching". All this attention to the topic is easily understood when considering that real-life problems involve complex, and often noisy, data, where it makes more sense to relax some of the constraints involved in exact matching, so that there is room for some error-tolerance [Riesen et al., 2010, Schuhmacher and Ponzetto, 2014].

Despite a common agreement on the importance and applicability of inexact matching, there has been little exploration on its use for mining graph patterns. In this section, we first present an overview of the approaches to compare graphs when the graph isomorphism is put aside. Then, we mention the algorithms that have been proposed to mine patterns using inexact matching.

2.3.1 Similarity measures for graphs

According to Xiao et al. [2008], graph similarity measures can be grouped in three categories:

- *Distance measures based on cost or edit distances* [Gao et al., 2010]. The similarity between graphs is defined as the minimum cost required to transform one graph into the other. Each operation to transform the graph (insertion, deletion or substitution of vertices or edges) has an assigned cost, provided by the user, that possibly incorporates specific information on the context of the problem. Although, given two graphs G_1 and G_2, there could be different operation sequences that transform G_1 into G_2, the edit distance is defined as the minimum cost associated to the transformation, and, thus, is unique. However, to find the minimum cost is not easy and the method chosen to calculate it depends on the type of the graphs being compared. Establishing the similarity between correspondent vertices and edges is an unsolved problem yet; in the case of labeled graphs, the different methods to calculate the edit distance use the labels in vertices and/or edges [Neuhaus and Bunke, 2004, 2005, 2006]. In the case of unlabeled graphs, usually graphs

are transformed into strings, using information of their vertices, edges and connectivity, and the edit distance between two graphs is calculated from the edit distance between the corresponding strings [Robles-Kelly and Hancock, 2003, 2004, Yu and Hancock, 2006].

- *Distance measures based on structure* [Bunke and Shearer, 1998, Hidovic and Pelillo, 2004]. The similarity between two graphs is measured by one or several substructures they have in common. An example of this type of measures is based on the maximum common subgraph [Bunke and Shearer, 1998]; if g is a common subgraph between G_1 and G_2, it is maximum if there is not another common subgraph between G_1 and G_2 with more vertices than g. If $mcs(G_1, G_2)$ denotes the maximum common subgraph between G_1 and G_2, Bunke and Shearer define the distance between the graphs as

$$d(G_1, G_2) = \frac{|V(mcs(G_1, G_2))|}{\max(|V(G_1)|, |V(G_2)|)} \tag{2.1}$$

  and prove that this measure satisfies the properties of a metric. Finding the maximum common subgraph between two graphs can be computationally expensive, but it has been shown [Bunke, 1997] that the computation of the maximum common subgraph between two graphs is equivalent to the computation of the edit distance between them, using certain cost function. As a consequence, some algorithms that calculate the edit distance can be adapted to calculate the distance based on the maximum common subgraph.

- *Distance measures based on characteristics* [Dehmer and Emmert-Streib, 2007]. Each graph is associated to a vector of characteristics and the distance between two graphs is defined as the distance between their associated vectors. Creating these vectors depends on the structural characteristics important for the particular problem being considered.

A different approach to measure the similarity between graphs is to use graph kernels. A kernel function is a symmetric function that maps pairs of patterns to real numbers [Neuhaus and Bunke, 2007]; in principle, a kernel K corresponds to the inner product in some feature space that, generally, is different from the representation space of the instances. Graph kernels were initially proposed by Kondor and Lafferty [2002] and extended by Smola and Kondor [2003] and Gartner [2002]. Since then, numerous functions have been proposed that allow the comparison of

graphs in specific problems. According to Bunke and Riesen [2011], these functions can be put into three main families: diffusion kernels [Kondor and Lafferty, 2002], kernels based on random walks [Gartner et al., 2003, Vishwanathan et al., 2008], and convolution kernels [Ramon and Gartner, 2003, Shervashidze et al., 2009]. Unfortunately, kernels could be expensive to evaluate, which makes them an inconvenient option when numerous graph comparisons are required. A detailed description of graph kernels, their properties, characteristics and use can be found in [Gärtner, 2008] or [Neuhaus and Bunke, 2007].

2.3.2 Inexact pattern mining

The use of inexact matching in pattern mining has been relatively little explored. There are some algorithms that, although do not properly look for frequent graphs using inexact matching, somehow diverge from the traditional frequent pattern mining problem. Monkey [Zhang et al., 2007] allows the detection of approximately frequent trees in structured data. MUSE [Zou et al., 2009] and UGRAP [Papapetrou et al., 2011] search for frequent patterns in uncertain graph data. SUBDUE [Holder et al., 1994, Holder, 1988] offers the possibility of providing a similarity threshold and a distortion function, which assigns an specific cost to each edit operation between graphs; in this way, two graphs are similar if the cost of transforming one into the other is below the provided threshold. GraMi [Saeedy and Kalnis, 2011] uses generalized isomorphism to find frequent patterns, allowing the replacement of some edges by paths of bounded length, but keeping the isomorphism in vertices and without considering inexact matching.

The previous algorithms acknowledge both, the existence of problems where it cannot be expected that two graphs representing the same pattern are identical, and the need of searching patterns that could go unnoticed when using algorithms designed for the "standard scenario". However, none of the mentioned algorithms focus on the problem of our interest. In fact, in the literature we find only a few algorithms whose main goal is to find frequent patterns using inexact matching: gApprox [Chen et al., 2007], APGM [Jia et al., 2011] and VEAM [Acosta-Mendoza et al., 2012].

The algorithm gApprox was proposed in 2007 by Chen et al. to mine approximate patterns in a single graph, being the first algorithm approaching this problem. Showing an example from a protein network and motivated by "the inherent noise

or data diversity", the authors allow matching graphs with label differences and structural differences in edges. With the purpose of managing label differences, the user is given the option of providing a list of admissible label replacements. Then, given an error threshold, two graphs are considered to match if their dissimilarity is below the threshold. To establish the dissimilarity between graphs, gApprox counts the total number of label differences (all of them admissible, according to the list provided by the user), as well as the differences between the edges of the compared graphs; thus the measure is a variant of the edit distance [Gao et al., 2010], counting the differences between two compared graphs.

The algorithm explores the search space following a pattern-growth approach. Pattern support is determined by following, in essence, the definition proposed by Kuramochi and Karypis [2005] avoiding to count overlapping occurrences. However, since the computation of this support measure is expensive, they calculate an upperbound, given by the maximum number of disjoint pattern occurrences, and, in this way, they gain efficiency.

In 2011 Jia et al. propose APGM [Jia et al., 2011], an algorithm to mine patterns in data that might be affected by noise in real-life applications. The authors focus on the particular problem of structures in proteins which could have noise and distortions from sources as changes in protein aminoacids or imperfect experimental measurements, among others, therefore, they point out, using exact matching would impose an unrealistic constraint that could lead to miss important patterns. To avoid the later situation, the authors propose to use a similarity graph measure based on a real square *compatibility matrix*, M, whose indices correspond to the vertex labels existent in the analyzed database. Each entry $M_{ij} \in [0, 1]$ indicates the probability of label i being mistakenly replaced by label j. Given a compatibility matrix and a similarity threshold τ, a graph G is defined as approximately isomorphic to a graph G' if there is an injective function $f : V(G) \mapsto V(G')$ so that the product, S_f, of the normalized similarity between each vertex pair (v, v') is greater than τ. In each case, the similarity between the two vertices is normalized by dividing the expected probability of v label being replaced by v' label by the probability of v label being correctly assigned (diagonal entry of the compatibility matrix). Then, the similarity S between two graphs is defined as the maximum value that S_f can take, considering all possible functions f between the two graphs. APGM is designed to mine patterns in a collection of graphs D, so, the support

of a pattern P is defined as

$$\text{supp}_P = \sum_{G' \in D_P} S(P, G')/|D|, \tag{2.2}$$

where D_P represents the subset of graphs in D that contain a subgraph approximately isomorph to P. Also, it is important to mention that, although the authors talk about the possibility of defining further compatibility matrices and allowing differences in edge labels, as well as "topology distortions" in edges (missing edges), they do not follow this idea.

Finally, Acosta-Mendoza et al. [2012] extended the mining process of APGM to allow approximation over the edge label set. In order to do this, the authors of VEAM present a definition of approximate sub-isomorphism in which they extend the definition of vertex approximate sub-isomorphism, allowing both probabilistic vertex label substitution and probabilistic substitution on the edge label set. VEAM allows variations of vertex and edge labels through substitution probability matrices, but it requires that two graphs that are approximately isomorphic have the same topology.

These relatively recent algorithms represent the first steps towards mining patterns in an important, and little explored, way. It should be highlighted that, although the algorithms use inexact matching, none of them allows structural differences in vertices, so, they require a bijective function between the vertex sets of matching graphs. In our work, we want to take further the use of inexact matching by relaxing the requirement of correspondent vertices; this constitutes the main difference between what we propose and previous algorithms.

2.4 Interesting patterns

A current trend in pattern mining is to focus in getting patterns that, besides being frequent, are interesting; this could lower the computational cost of analyzing the obtained patterns and increase their applicability. Reducing the number of patterns is not a new issue and, in fact, the need of work focusing on discovering a meaningful set of frequent subgraphs, instead of a complete set, has been pointed out by different authors [Cheng et al., 2010, Jiang et al., 2013].

Several approaches have been proposed to address the problem. First, we find works that have focused on mining maximal and closed graph patterns, since these kind of patterns "condense" information, reducing the size of the output set and, therefore, making it easier to be managed. Then, we find another approach that focus on choosing important graphs as a mean to reduce a given graph pattern set. We describe both approaches in this section.

2.4.1 Closed and maximal patterns

Although maximal patterns, in general, had received attention before, it was until 2004 that Huan et al. [2004] brought attention to the problem of mining maximal subgraphs as a way of making more efficient the overall data mining process, decreasing the amount of storage needed and the number of mined patterns. Their algorithm, SPIN, finds all the frequent trees in a collection of graphs, then expands the trees into frequent cyclic graphs, and, finally, constructs the set of maximal frequent subgraphs, using some pruning techniques to make the maximal subgraph mining more efficient.

Over the next years, other algorithms were proposed to find maximal frequent patterns in a collection of graphs. Thomas et al. [2010] proposed the algorithm Margin to find maximal patterns in a collection of graphs. For each graph in the input set, they use a graph lattice to represent the search space and identify frequent maximal graphs candidates. To this end, the authors first find a frequent connected subgraph and expand it until it is maximal; from that graph, represented (like every subgraph) by a point in the lattice, they traverse the lattice to identify other maximal candidates. Finally, in a post-processing step, the authors merge the candidates and select the maximal frequent patterns.

Chen et al. [2012] proposed a method to find maximal patterns in a collection of graphs by using a top-down mining strategy. First, the authors construct a tree structure for the larger graph in the collection; each level in the tree consists of subgraphs obtained by removing infrequent edges from each graph in the previous level. Relying on the antimonotonicity property, the algorithm removes edges until reaching a frequent graph, which is assumed to be maximal, since its parents are infrequent. Then, the algorithm continues by adding to the tree the remaining graphs in the collection, sorted in descending order respect to their size, finding for

each graph its corresponding level in the tree and using isomorphism to compare it to subgraphs in the same level.

Other algorithms that allow finding maximal patterns, even if it is not their main focus, are FP-GraphMiner [Vijayalakshmi et al., 2011] and wgMiner [Ozaki and Etoh, 2011]. FP-GraphMiner analyzes a collection of graphs by finding the frequent edges in all the graphs and creating a special undirected graph, the FP-Graph, that contains the frequency and structural information of the vertices and edges; from the FP-Graph, frequent graphs can be determined, including the maximal ones. The algorithm wgMiner is designed to find closed and maximal patterns in a graph collection where each graph has associated importance weights both internally, in its edges, and externally, with respect to the whole collection; the algorithm maximizes the importance or weight sum of the mined patterns, and performs a depth first search with traditional pruning in order to get maximal patterns.

For mining closed frequent subgraphs, we find the algorithm CloseGraph, proposed by Yan and Han [2003]. This algorithm extends graphs by edges; given a graph g, an edge e can be added to generate a new graph, denoted by $g\Diamond_x e$. The authors prune the search space by using two properties; first, they have an "early termination" condition, which represents a way to establish if two graphs have equivalent occurrences and, therefore, only one of them should be extended. Second, the authors prove a lemma establishing that, given two graphs g and g', if $g \subset g'$ and $support(g) = support(g')$, then there exists h such that $h \subseteq g'$, $h = g\Diamond_x e$ and $support(g) = support(h)$; this lemma translates into the fact that, in order to check if a frequent graph is closed, one could check the support of its supergraphs. Their algorithm, then, works in three main steps: (1) generation of a frequent graph, (2) application of the aforementioned lemma to check whether a graph is closed, and (3) verification of the early termination condition, to determine if a graph should be extended.

We could say that mining maximal and closed frequent subgraphs are the most basic ways of approaching the problem of reducing the amount of generated patterns. Although in both cases the number of mined patterns is reduced, there still is certain generality, since the patterns are chosen without taking into account the context of the problem. Other works focus on more specific contexts or subgraphs of interest and propose more sophisticated algorithms to find the patterns in question [Ranu and Singh, 2009, Yan et al., 2008].

Finally, it is important to mention that, unlike the proposed algorithms in this thesis, all the works presented in this section use exact matching between graphs.

2.4.2 Choosing a pattern subset

Without having specific criteria to select patterns, the task of choosing a pattern subset could be seen as a dubious task: how can we select patterns with no clear, beforehand knowledge of what we are looking for? However, several works have proposed general guidelines and measurements to help to identify patterns that could be considered interesting, despite the lack of an specific context [Geng and Hamilton, 2006, Spyropoulou et al., 2014]. For instance, Geng and Hamilton [2006], in the context of association rules, enumerate 9 criteria that help to determine whether or not a pattern is interesting: conciseness, peculiarity, diversity, novelty, surprisingness, generality/coverage, reliability, utility and actionability; Table 2.1 shows a summary of the authors' description of these characteristics. Although these works do not consider graphs, they suggest that it is possible to reduce a pattern set in such a way that the resulting subset possesses some general interestingness.

In the context of graphs, there is one work that somehow follows this line. Al-Hasan et al. [2007] proposed the algorithm ORIGAMI, that, given a graph collection D, finds a representative set R of patterns that satisfy being α-orthogonal and β-representative. Two patterns are defined to be α-orthogonal if their similarity is below a given threshold α. On the other hand, a set R is β-representative for the set $\Gamma(R, D) \subseteq D$ if for every graph G_Γ in $\Gamma(R, D)$ there is a graph G_R in R such that the similarity between G_Γ and G_R is above the threshold β. Thus, ORIGAMI builds a pattern subset whose elements are different among them, but represents certain subset Γ of D. Ideally, R should represent *all* graphs in D, but, as explained ahead, the algorithm is based mainly on selecting $\alpha-$orthogonal patterns and there exists a *residual* or set of uncovered patterns in D.

The algorithm has two main stages. First, ORIGAMI finds a sample, $\hat{M}$, of the set of maximal frequent subgraphs. The set $\hat{M}$ is obtained by using random walks to traverse the search space, together with a termination condition designed to avoid the generation of duplicated subgraphs. Then, in the second stage, ORIGAMI constructs the set $R \subseteq \hat{M}$ by building a graph in which vertices represent the elements of $\hat{M}$ and two vertices are connected iff they are α-orthogonal; then,

TABLE 2.1: *Summary of characteristics that, according to Geng and Hamilton [2006], a pattern, or a set of patterns, must possess in order to be considered interesting.*

Characteristic	Description
conciseness	A pattern set is concise if it contains relatively few patterns, therefore being easy to understand and remember.
peculiarity	A pattern is peculiar if it is far away from other discovered patterns, according to some distance measure.
diversity	A set of patterns is diverse if the patterns in the set differ significantly from each other.
novelty	A pattern is novel to a user if it was not known before and cannot be infered from other known patterns.
suprisigness	A pattern is surprising if it contradicts a person's existing knowledge or expectations, or if it is an exception to a more general pattern which has already been discovered.
generality/coverage	A pattern is general if it covers a relatively large subset of a dataset; if a pattern characterizes more information, it tends to be more interesting. Concise patterns tend to have a greater coverage.
reliability	A pattern is reliable if the relationship described by the pattern occurs in a high percentage of applicable cases.
utility	A pattern is of utility if its use contributes to reaching a goal. Different users may have different goals concerning the knowledge that can be extracted from a dataset.
actionability	A pattern is actionable in some domain if it enables decision making about future actions in this domain.

the maximum clique is found by approximating the solution and evaluating, every time, the number of members in $\hat{M}$ that are not represented in R, aiming to minimize the cardinality of $D \backslash \{R \cup \Gamma(R, \hat{M})\}$.

Clearly, ORIGAMI focuses in a different mining problem than we do, analyzing a collection of graphs and using a random approach to perform the mining task. The second stage of the algorithm resembles the idea of what we attempt in the part of our work concerned with finding interesting graphs, however, we are more interested in coverage than we are in minimizing redundancy. Since one of the main features of the algorithm we propose is to capture new information by finding patterns missed when structural differences in vertices are not allowed, therefore,

we attempt to preserve this information by avoiding to leave patterns without coverage. Moreover, we use a differet strategy to find the interesting subset of patterns.

2.5 Discussion

From the algorithms described in this chapter, the ones that are most closely related to some aspect of our research problem (Vsigram [Kuramochi and Karypis, 2005], gApprox [Chen et al., 2007], APGM [Jia et al., 2011], Origami [Al-Hasan et al., 2007]) represent novel works in the problems they approach and, because of the differences between the proposed algorithms and previously existing ones, the authors could not show a direct comparison. Instead, authors assess the quality of the proposed algorithms by comparing *some aspects* of them; Origami's authors, for instance, evaluate their random-walks approach to mine maximal patterns in terms of performance, although some other aspects, like the quality of their representative patterns, are not assessed. In the case of gApprox, the authors perform experiments on real and synthetic graph datasets, showing the number of patterns that their algorithm finds and the time required for it; they, however, do not mention any work closely related (in the sense of finding approximate patterns) and do not compare against other algorithms. In our work, rather than seeking an improvement with respect to these algorithms, we explore a problem that, by its combined characteristics, is different from the problems that the algorithms described in this chapter solve.

Given that inexact matching has been acknowledged as being closer to describe some real-life scenarios than exact matching, it is rather surprising that mining graph patterns using inexact matching has received relatively little attention. The algorithms in literature that address the problem represent initial, and very important, steps in the direction of exploring this problem, however, to the best of our knowledge, the problem of mining all frequent patterns without requiring a bijection between vertices has not been approached before.

It is clear that, as a consequence of relaxing the conditions in matching graphs, we will face challenges related to the increment in the number of mined patterns. Besides the computational cost of the mining process, the amount of generated patterns – which is already a problem that has come to the attention of different

authors when using exact matching – could be so big that the new knowledge, acquired by the use of inexact matching, ends up buried in an overwhelming amount of information. Because of this, it is also clear the need of reducing the amount of mined patterns, somehow counteracting the relaxation of the matching constraint by requiring some characteristics, besides frequency, in the graphs.

Although the algorithms mentioned in this chapter set a precedent in the different research lines that encompass our problem of interest, none of them approaches it. Thus, the questions remain on how the use of inexact matching, allowing structural differences, could be integrated into the frequent graph mininig problem and how a set of interesting patterns could be obtained from the mined patterns.

Chapter 3

Algorithm for mining inexact patterns

During this research, our first objective was to be able to mine patterns in a single graph, using inexact matching, suggesting that allowing structural differences in vertices could lead to the discovery of new and useful patterns. In this chapter we begin by defining two dissimilarity functions for graphs, taking into account structural differences. Then, we describe a strategy to identify pattern occurrences that may have structural differences in vertices and edges using one of the proposed dissimilarity measures to compare graphs. Afterwards, we present our proposed algorithm, AGraP (*Approximate Graph Patterns*), that combines the dissimilarity measures and the strategy described before in order to mine patterns in a single graph. Finally, we present some experiments showing that AGraP is, indeed, able to discover new patterns in graphs.

3.1 Dissimilarity functions

For mining patterns in a single graph, using inexact matching, we require a graph comparison function that allows structural differences, in vertices as well as in edges, between graphs. As mentioned in Section 2.3.1, the graph distance measures that have been proposed in the literature can be categorized as cost-based, structure-based and feature-based distance measures. Measures in the later category rely on representing graphs through vectors, which is not suitable for our purposes. From the other two categories, we find that a cost-based measure can be

better incorporated to the algorithm we propose in the next sections. We propose two dissimilarity measures, both based on the graph edit distance; one of them takes into account the size of the graphs being compared, while the other works with the "absolute" (unnormalized) value of the calculated dissimilarity. The two proposed dissimilarity measures are described next.

3.1.1 Function f_{dis}

Let G_1 and G_2 be two graphs that we want to compare. We propose to use the graph edit distance [Sanfeliu and Fu, 1983] as a base for our proposed dissimilarity measure, and we start by defining a comparison between the vertices of G_1 and G_2.

Let us assume that it has been established a one-to-one binary relation $m \subseteq V(G_1) \times V(G_2)$, i.e., a correspondence between a subset of $V(G_1)$ and a subset of $V(G_2)$. We define the dissimilarity between the vertices of G_1 and G_2 as

$$v_{edit} = \sum_{v \in V(G_1) \setminus R_{V1}} d_v(v, m(v)) + |R_{V1}| + |R_{V2}|, \tag{3.1}$$

where

$$\begin{aligned} R_{V1} &= \{v_1 \in V(G_1) | \nexists\ v_2 \in V(G_2) \text{ s.t. } m(v_1) = v_2\} \\ R_{V2} &= \{v_2 \in V(G_2) | \nexists\ v_1 \in V(G_1) \text{ s.t. } m(v_1) = v_2\} \\ d_v(v, m(v)) &= \text{cost of replacing } \psi_{G_1}(v) \text{ by } \psi_{G_2}(m(v)). \end{aligned}$$

Since m is not a bijective function but just a one-to-one binary relation, there could be some vertices in $V(G_1)$ not related to any vertex in $V(G_2)$ and some vertices in $V(G_2)$ that are not the image of any vertex in $V(G_1)$; this is the reason to include the terms $|R_{V1}|$ and $|R_{V2}|$ in v_{edit}, which represent the cost associated to those unpaired vertices. In this way, we take into account structural differences in vertices between the graphs being compared.

In the same fashion, we define the dissimilarity between edges as

$$e_{edit} = \sum_{(u,v) \in E(G_1) \setminus R_{E1}} d_e((u, v), (m(u), m(v))) + |R_{E1}| + |R_{E2}|, \tag{3.2}$$

where

$$R_{E1} = \{(u,v) \in E(G_1) | \nexists\, u', v' \in V(G_2) \text{ s. t. } m(u) = u', m(v) = v' \text{ and } (u',v') \in E(G_2)\}$$

$$R_{E2} = \{(u',v') \in E(G_2) | \nexists\, u, v \in V(G_1) \text{ s. t. } m(u) = u', m(v) = v' \text{ and } (u,v) \in E(G_1)\}$$

$d_e((u,v),(m(u),m(v))) =$ cost of replacing $\psi_{G_1}((u,v))$ for $\psi_{G_2}((m(u),m(v)))$.

As in the case of vertices, in e_{edit} there have been included the terms $|R_{E1}|$ and $|R_{E2}|$, representing the costs of unmatched edges.

Finally, the dissimilarity between G_1 and G_2 is given by

$$f_{dis}(G_1, G_2) = \kappa\, v_{edit} + (1-\kappa) e_{edit},$$

where, $0 \leq \kappa \leq 1$; thus, we can see f_{dis} as the convex linear combination of v_{edit} and e_{edit}. By using κ, we can weight differently the differences between vertices and edges if a certain application requires to make such differentiation. Since we are not dealing with any specific application, for all our experiments, we will use $\kappa = 0.5$, which means that differences between vertices and edges are equally important.

We expect the value of f_{dis} to be higher when there are more differences between G_1 and G_2, there lies the reason of referring to this function as a *dissimilarity*.

Given the relation m, the function f_{dis} is relatively efficient, as it only requires additions and counting elements in the defined sets for v_{edit} and e_{edit}. Finding m could be not at all easy but, in our case, we can combine the process of finding this relation with the process of growing pattern candidates during the mining process. Thus, one important benefit of using the function f_{dis} is that it can be calculated during the mining process, following a pattern-growth approach, without increasing the algorithm's complexity.

As an example, consider the graphs in Fig. 3.1 and suppose that no label sustitutions are allowed. If we know that $m = \{(0,3),(2,5)\}$, we have that $d_v(v, m(v)) = 0$ (since there are no label replacements), the set R_{V1} would contain only vertex 1 (which is not related to any vertex in G_2) whereas the set R_{V2} would consist of vertex 4 (which is not related to any vertex in G_1), thus, $|R_{V1}| = |R_{V2}| = 1$ and

(A) G_1 (B) G_2

FIGURE 3.1: *Graphs used to exemplify the dissimilarity function f_{dis}. For these graphs, $f_{dis}(G_1, G_2) = 3$.*

$v_{edit} = 2$. Analogously, for the edges, $|R_{E1}| = 2$ (edges $(0, 1)$ and $(1, 2)$), $|R_{E2}| = 2$ (edges $(3, 4)$ and $(4, 5)$) and $d_e((u, v), (m(u), m(v))) = 0$, thus $e_{edit} = 4$. Therefore, using $\kappa = 0.5$, we have $f_{dis}(G_1, G_2) = (0.5)(2) + (0.5)(4) = 3$.

3.1.2 Function f_{rel}

Although the function f_{dis} makes it possible to compare graphs and evaluate structural differences in vertices and edges, the dissimilarity measure we get through this function does not depend on the size of the graphs; thus, if two graphs differ by one vertex, this dissimilarity will be equally important if the graphs have 2 vertices as if they have 5000.

However, sometimes we would like that small differences between graphs were less important in large graphs than in small ones; therefore, we follow the ideas of Bunke and Shearer [1998] and divide the differences between graphs by the size of the graphs being compared. Thus, to compare a pattern P with one of its possible occurrences g, and establish whether g is similar enough to P to be considered as its occurrence, we define the comparison function as

$$f_{rel}(P, g) = \frac{\kappa\, v_{edit}}{|V(P)| + |V(g)|} + \frac{(1 - \kappa) e_{edit}}{|E(P)| + |E(g)|}, \tag{3.3}$$

with v_{edit} and e_{edit} defined as in Equations 3.1 and 3.2. Following a pattern-growth strategy, P initially represents a single vertex and its occurrences are identified by comparing its label against other vertices' labels; the function f_{rel} will be used to compare patterns where $|V(P)| \geq 2$ and $|E(P)| \geq 1$, and, therefore, denominators in the previous expression will never be zero.

Each term in the expression (3.3) represents the edit distance between P and g, considering only vertices or only edges, normalized by the total amount of vertices or edges in both graphs. In this way, the impact of the values v_{edit} or e_{edit} depends on the size and order of the considered pattern and occurrence. Furthermore, while f_{dis} is unbounded, the function f_{rel} is always between 0 and 1, and it could be easier to establish a dissimilarity threshold using f_{rel}, as a percentage of the dissimilarity of graphs, than using f_{dis}.

Unfortunately, using f_{rel} causes the lost of the antimonotonicity property (as shown in Section 3.4.1.1) and prevents using this property for pruning when traversing the search space. For this reason, in the successive development of our proposed graph mining algorithms, we will only use the function f_{dis}. Mining algorithms using f_{rel} are left as a future work direction.

3.2 Strategy for mining inexact patterns

One of the main characteristics of the algorithm we propose is its ability to find patterns that do not match their occurrences in an exact way. Thus, our algorithm uses a dissimilarity threshold Δ and the comparison function f_{dis}, previously described, to determine when a subgraph is similar enough to a pattern to be considered its occurrence.

The search space is explored through a pattern-growth approach. Once we have a pattern P and its occurrences, we add a new vertex v to form a pattern $P' = P \diamond v$ and we find the occurrences of P' by analyzing, and possibly growing, every occurrence of P. The strategy we use to allow differences of each type is described next.

Label differences. A table specifying equivalences between labels must be provided. In this way, we allow occurrences to grow through a new vertex whose label could be non-identical to v's label, provided that the label substitution is permitted according to the given equivalence information, preserving the dissimilarity threshold Δ. So, following this idea, we are able to find occurrences in which labels are not exactly the same.

Structural differences in vertices. We allow structural differences in vertices, which means that a pattern occurrence could have less or more vertices than the pattern. In the first case, if an occurrence of P cannot grow into an occurrence of P', we keep it in the occurrence set of P' and simply add a mark to indicate that there is a missing vertex.

In the second case, occurrences that have more vertices than their pattern are found by replacing the requirement of having an edge between an occurrence of P and a vertex matching v, by the requirement of having a *path* between them. A similar idea is used in [Saeedy and Kalnis, 2011], although the authors explicitly state that they do not use inexact matching. By finding a path instead of a single edge, we identify occurrences that could have more vertices than their pattern.

The allowed path length can be calculated by taking into account the current distance existing between the occurrence and its pattern, so that the dissimilarity threshold is still satisfied. To this end, when we expand an occurrence P, we estimate the maximum path length ℓ that can be allowed without surpassing the dissimilarity threshold Δ. Let us consider a pattern P and an occurrence g with a dissimilarity given by

$$f_{dis}(P, g) = \kappa v^*_{edit} + (1 - \kappa)e^*_{edit}.$$

If the pattern P is grown into $P' = P \diamond v$ and the occurrence g is grown into $g' = g \diamond v_g$, where v_g is a vertex (with compatible label to v) connected to g by an ℓ-length path (with $\ell - 1$ intermediate vertices), we would have

$$\begin{aligned} f_{dis}(P', g') = (\kappa v^*_{edit} + \kappa(\ell - 1) + \kappa d_v(v, v_g)) \\ + (1 - \kappa)e^*_{edit} + (1 - \kappa)\ell + (1 - \kappa)e_{new}, \end{aligned}$$

where e_{new} is the amount of edges that connect the pattern P with the vertex v. When considering an ℓ-length path, the edition cost relative to vertices is modified because, in order to match g' and P', $\ell - 1$ vertices must be removed, while, in the edge case, ℓ edges must be removed and e_{new} must be added in order to connect the vertex v_g with g.

Since, in order to consider g' an occurrence of P', we require $f_{dis} \leq \Delta$, we can use this inequality and the previous equation to obtain:

$$\begin{aligned}\ell &\leq \Delta - \kappa v^*_{edit} - \kappa\ell + \kappa - \kappa d_v(v, v_g) - (1-\kappa)e^*_{edit} + \kappa\ell - (1-\kappa)e_{new} \\ &= \Delta - \kappa v^*_{edit} + \kappa - \kappa d_v(v, v_g) - (1-\kappa)e^*_{edit} - (1-\kappa)e_{new} \\ &= \Delta - f_{dis}(P, g) + \kappa - \kappa d_v(v, v_g) - (1-\kappa)e_{new},\end{aligned}$$

which represents the path length allowed to connect g and v_g while still having a dissimilarity, respect to P', below the established threshold. By bounding the length of the path, we restrict the search of a matching vertex to the neighbors up to a certain degree.

Structural differences in edges. Finally, in order to allow occurrences with structural differences in edges, we allow a new vertex matching v to be connected to an occurrence of P in any way, in a similar fashion of what gApprox does [Chen et al., 2007].

In all the cases previously described, whether there are label or structural differences, we always keep track of the differences between the pattern and each of its occurrences. In this way, we can easily use the function f_{dis}, without additional computational cost, to determine the occurrences that satisfy the dissimilarity threshold and calculate the pattern support.

3.3 AGraP

We combine the strategy described in the previous section, the dissimilarity function f_{dis} and a depth-first search strategy to develop the algorithm AGraP (*Approximate Graph Patterns*). AGraP allows to mine frequent patterns in a single graph using inexact matching and allowing structural differences in edges and vertices. To this end, the algorithm requires a frequency threshold σ and a dissimilarity threshold Δ.

Label differences are allowed and managed through a dictionary D, that specifies which label substitutions are valid. We use a pattern-growth approach to generate pattern candidates; every time a pattern P is expanded to create a new pattern

P', we construct the occurrences of P' starting from the occurrences of P. Also, every time we expand an occurrence, we keep track of the edit cost respect to the considered pattern. Pattern support is calculated by using the function proposed by Bringmann and Nijssen [2008], described in Section 2.2.

Algorithm 3.1, which uses the functions Traverse, Expand and ExpandOccurrence, contains the pseudocode of AGraP. We can see that the algorithm uses a depth-first search, as we start with a vertex v in G and identify all the patterns that can be grown from v, before we even identify the whole set of frequent patterns that consist of a single vertex. To see this with more details, we can observe that in each iteration of the `for` loop (line 2), AGraP chooses a vertex v in G and finds all the connected graphs in G that contain v. Then, the vertex v is marked as "explored" and it is ignored during the further generation of patterns. The functions Expand and Traverse are used to generate patterns that can be build starting from v. First, Traverse identifies the vertices connected to v that have not been explored and creates the set V_{exp}. Then, Expand grows the pattern P (initially $P = \{v\}$) to each vertex in V_{exp} and explores the new patterns through recursive calls to Traverse (which, in turn, calls to Expand). Notice that P_E stores the patterns that result from growing P to the vertices in the current V_{exp}, while P_T stores those patterns that can be obtained by growing P to neighboring vertices that are *not* already included in V_{exp}.

The traverse of the search space via mutual, and nested, recursive calls starts with a vertex and explores its supergraphs iterating through the sets of neighbors, level by level. Let us take as an example the graph shown in Fig. 3.2 and assume that the exploration starts at the vertex 0, labeled A. The first time that the function Traverse is called, it identifies the vertices 1, 2 and 3 as the set V_{exp} (direct neighbors) and, then, the function Expand is called. The later function expands the pattern, vertex by vertex through recursive calls, to all the neighbors in V_{exp} and calls Traverse to identify the next level neighbors and proceed expanding the pattern until either it does not satisfy the frequency threshold or it cannot be expanded further, i.e., it reaches the whole graph. Therefore, in our example, the first candidate patterns generated would be A (initial vertex), AB, ABC, ABCG (first-level neighbors), ABCGD, ABCGDE, (second- level neighbors) and ABCGDEF (third-level neighbors). Once this base case is reached, it takes place the next iteration (function Expand, line 2) over the next-to-last explored neighbors, expanding again each of them. In our example that would mean the

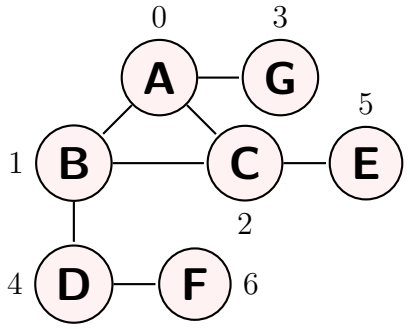


FIGURE 3.2: *A graph example whose subgraphs can be explored by mutual recursive calls of functions* Expand *and* Traverse.

next second-level neighbor (vertex 4, labeled D), so the next pattern generated would be ABCGDF. Following these ideas, the last pattern generated would be AG, since vertex 3 (labeled G) would correspond to the last iteration over the first-level neighbors an it cannot be expanded further. Then, the algorithm would proceed to explore patterns that can be grown starting from vertex 1. This process, for the vertex 0, is illustrated in the Fig. 3.3, where some vertices and edges have been marked in a different color to stress the fact that, at each level i, the corresponding ith-level neighbors are explored.

In the function ExpandOccurrence, occurrences of a given pattern are identified and it is in this function that we allow structural differences between graphs, thus, it is here where one of our main goals is reflected. When the given pattern P is grown to create the new pattern $P' = P \diamond v$, each occurrence O of P is analyzed and we try to add a new vertex u, corresponding to v (if it exists), while taking into account the dissimilarity constraint imposed by the threshold Δ; the vertices connected to O by paths are identified by gradually examining each vertex on the ith neighborhood of O, starting from $i = 1$. In this way, the set $M_{P'}$ of occurrences of P' is built upon the set M_P of occurrences of P. When $M_{P'}$ is being created, the ideas described in Section 3.2 are implemented; thus, we allow occurrences with more vertices than a pattern (lines 4-7), occurrences with less vertices than a pattern (line 3) and occurrences whose edges differ from those of the pattern (when calculating the edit distance), as long as the occurrences are similar enough to P to satisfy Δ.

As we expand each occurrence in M_P, we keep record of the edit distance between the expanded occurrence and the pattern P'. Differences between vertices and edges are kept in separated records, so, each expanded occurrence has two

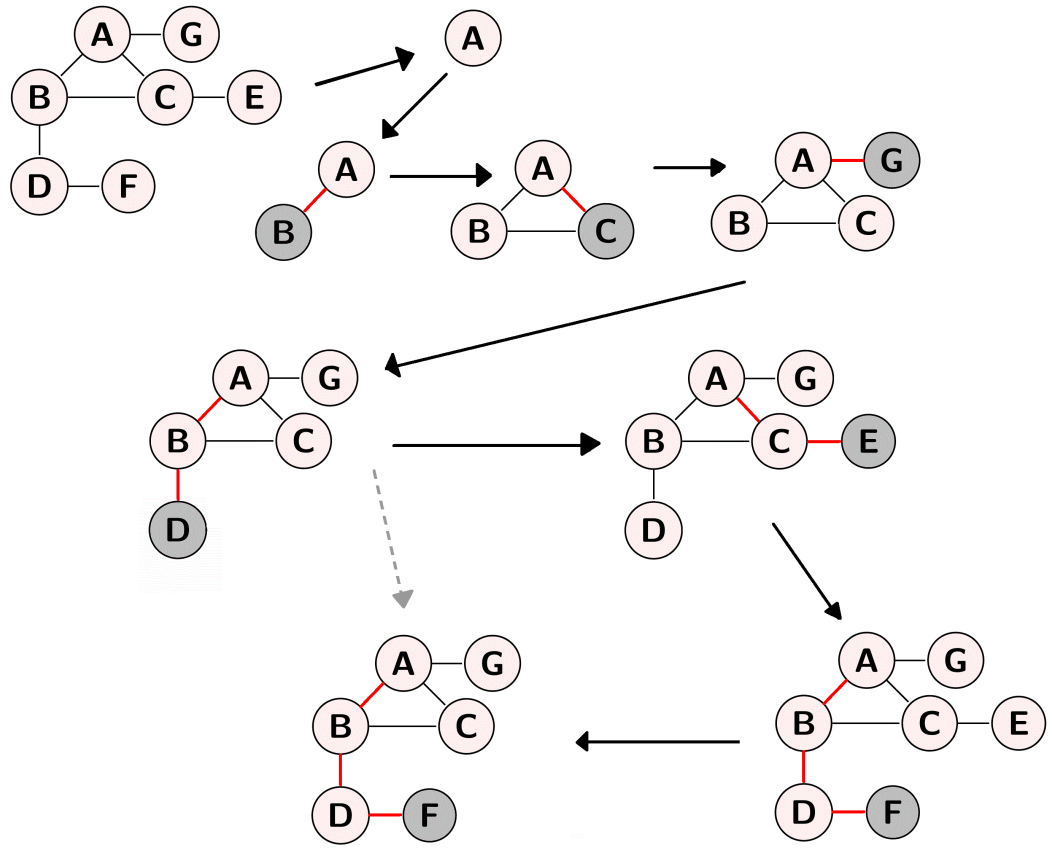


FIGURE 3.3: *Example of the traverse of the search space via mutual recursive calls, by our algorithm, in the graph shown in Fig. 3.2. The algorithm starts with a vertex (vertex 0 in this example) and explores its supergraphs iterating through the sets of neighbors, level by level.*

associated numbers, v_{edit} and e_{edit}, that count the differences in vertices and edges (respectively) between the occurrence and the pattern P'.

The support of each pattern is calculated through the definition given by Bringmann and Nijssen [2008]. As pointed out in Chapter 2, this support definition is based on the minimum number of mappings, in G, that a vertex in $V(P')$ has, considering all the occurrences of P'.

We can see (function Expand, line 7) that we rely on the antimonotonicity property when generating new patterns, i.e., we only expand patterns that satisfy the frequency threshold. Because of this, we work with the dissimilarity function f_{dis}, which preserves this downward closure property, while we left exploring the use of f_{rel} for future work.

Finally, in the line 8 of the AGraP algorithm (Algorithm 3.1), we can see that the function unique is invoked every time that frequent patterns grown from different

vertex are added to the frequent pattern set P. The function unique simply compares patterns already in P against newly found patterns in $P_{Traverse}$, using f_{dis}, in order to avoid repeated patterns in the output set.

Algorithm 3.1: AGraP Algorithm

Input: G: graph to be analyzed,
σ: frequency threshold,
Δ: dissimilarity threshold,
D: dictionary that specifies equivalences between labels *(optional).*
Output: P: set of frequent patterns in G.

```
1 P ← ∅;
2 for v ∈ G do
3     Mark v as "explored";
4     M_v ← List of vertices with the same or equivalent (according to the;
            given dictionary) label to v, and their edit cost respect to v;
5     if |M_v| ≥ σ then
6         Add {v} to P;
7         P_Traverse ← Traverse({v}, M_v) ;
8         P ← unique(P, P_Traverse);
```

Function Traverse(P, M_P)

Input: P: candidate pattern,
M_P: list of P occurrences and their edit cost.
Output: P_{Exp}: set of patterns that can be obtained growing P.

```
1 V_exp ←Vertices connected to P that have not been explored or marked;
2 for u ∈ V_exp do
3     marked(u) ← True;
4 P_Exp ← Expand(P, M_P, V_exp);
5 for u ∈ V_exp do
6     marked(u) ← False;
```

It must be highlighted that, following AGraP's search strategy, f_{dis} clearly preserves the antimonotonicity property. When a pattern grows, the occurrences of the new pattern are obtained from the occurrences of the previous one and, since f_{dis} counts differences between a pattern and its occurrences, the dissimilarity of the new pattern against each occurrence can be the same (if the occurrence grows exactly as the pattern) or higher (if the occurrence is unable to grow, grows through a path or grows to a compatible, but nonidentical, vertex). Notice that, if an occurrence o of the previous pattern can grow in more than one way, since overlapping is not allowed, the support of the new pattern can remain the same

Function Expand(P, M_P, V_{exp})

Input: P: candidate pattern,
M_P: list of P occurrences and their edit cost,
V_{exp}: list of unexplored vertices connected to P.
Output: P_P: set of patterns obtained by expanding P.

```
1  P_P, P_E, P_T ← ∅;
2  for v_exp ∈ V_exp do
3      P' ← P ⋄ v_exp;
4      M'_P ← ExpandOccurrence(P, M_P, v_exp);
5      Calculate the support of P'. If it satisfies σ, add P' to P_P;
6      V'_exp ← V_exp \ {v_exp};
7      if |M'_P| > σ then
8          P_E ← Expand(P', M'_P, V'_exp);
9          P_T ← Traverse(P', M'_P);
10     P_P ← P_P ∪ P_E ∪ P_T;
```

Function ExpandOccurrence($P, M_P, newVertex$)

Input: P: pattern candidate,
M_P: list of P occurrences and their associated edit cost,
$newVertex$: vertex connected to P.
Output: M'_P: list of occurrences of $P' = P \diamond newVertex$ and their associated edit cost.

```
1  Create an empty list M'_P;
2  for occurrence O in M_P do
3      Estimate the maximum path length, ℓ, that can exist between O and
         a new vertex without surpassing the dissimilarity threshold;
4      Get set Neigh of vertices connected to O by a path with length less or
         equal to ℓ;
5      Identify the subset V_e of vertices in Neigh that have a label equal or
         equivalent to the label of newVertex;
6      for v ∈ V_e do
7          Calculate the edit cost between the graph O' = O ⋄ v and P';
8          if the dissimilarity between O' and P' is below Δ then
9              Add O' to the list M'_P and store the edit cost between O' and P';
10     if O could not be expanded into an occurrence in M'_p then
11         Calculate the edit cost between O and P';
12         if the dissimilarity between O and P' is below Δ then
13             Add O to the M'_P list and mark the absence of a new vertex
                 by the symbol '-';
14             Keep record of the edit cost between O and P';
15 Return list M'_P.
```

Function unique(A, B)

Input: A, B: pattern lists.
Output: A: pattern list combining A and B elements, without repeated patterns.

```
1 for every pattern a in A and every pattern b in B do
2     if a and b have same order, size and label sets then
3         if f_dis(a, b) = 0 then
4             Delete b from B
5 Return list A ∪ B.
```

(if at least one of the occurrences derived from o satisfies the dissimilarity threshold) or become smaller (if either none of the occurrences that grow from o satisfy the dissimilarity threshold, or o is unable to grow and unable to satisfy itself the dissimilarity threshold for the new pattern).

Regarding the completeness of the proposed algorithm, it is important to point out that AGraP is able to identify a frequent inexact pattern as long as (at least) one of the vertices in the pattern has enough occurrences to satisfy the frequency threshold σ. Given a graph G and thresholds σ and Δ, if $g \subseteq G$ is a frequent graph satisfying the requirement of having a frequent vertex v, then, since each vertex in the analyzed graph G is examined by the algorithm (Algorithm 3.1, line 2), AGraP will eventually identify v as frequent and will explore candidates that can grow from it. Because we are assuming that g is a frequent graph, we know that it has enough occurrences in G to satisfy σ and that, each time one of its subpatterns grows, there are enough occurrences able to grow satisfying the dissimilarity threshold Δ, and, as a consequence, g can be identified as a frequent pattern. Thus, AGraP is able to find all frequent inexact patterns that have at least one frequent vertex.

Regarding the complexity of our algorithm, we must point out that the time and storage space required by AGraP depends, not only on the order and size of the input graph, but also on the degree of its vertices, the topology of the graph, the diversity and distribution of the vertex and edge labels, and the thresholds specified by the user.

Depending on the analyzed graph, the algorithm could be computationally very expensive. To see this, let us consider a specially difficult scenario: an input graph that has n vertices with the same label and is completely connected. In this case,

we have that for the first analyzed vertex in G, v_1 (first iteration in the `for` loop, line 2, algorithm AGraP), the function Traverse finds a set V_{exp} with $n-1$ vertices and calls the function *Expand*; notice that posterior calls to Traverse will output an empty set, since all the vertices in $G\backslash\{v_1\}$ are connected to v_1 and are, therefore, marked (Traverse, line 2). The function Expand has a loop executed $|V_{exp}| = n-1$ times (Expand, line 2); in the iteration i of this loop, the function recursively calls itself $n-i$ times (Expand, line 8); at each recursive call, the pattern P grows by one vertex, V_{exp} is decremented by one vertex, and the set M_P grows by a factor $n-k$, where k is the recursion level. Thus, by the end of the first iteration of the `for` loop (Expand, line 2), $(n-1)(n-1)!$ comparisons have been made inside ExpandOccurrence. Then, the total time in this loop (and therefore in the first call to Expand) can be bounded by $(n-1)^2(n-1)!$. For the second iteration of the `for` loop in AGraP (line 2), the function Traverse finds a set V_{exp} with $n-2$ vertices (v_1 is already "explored" and cannot be included in the set), but M_P has $n-1$ vertices (same as before). As a consequence, the `for` loop in Expand is executed only $n-2$ times, but the comparisons required in ExpandOccurrence is still $(n-1)(n-1)!$. Following this reasoning, the algorithm requires a total number of comparisons bounded by

$$\begin{aligned} T &= (n-1)^2(n-1)! + (n-2)(n-1)(n-1)! + \cdots + (n-1)(n-1)! \\ &= (n-1)(n-1)!\,[(n-1) + (n-2) + \cdots + 1] = (n-1)(n-1)!\frac{(n-1)n}{2} \\ &= \frac{1}{2}(n-1)^2 n!. \end{aligned}$$

Thus, for this scenario, the time complexity of AGraP would be $O(n^2 n!)$.

Let us analyze the same example with respect to space complexity. As stated before, since the graph is completely connected, all vertices are neighbors of the first analyzed vertex; thus, (1) the first vertex has $n-1$ occurrences, and (2) after the first call to the function Traverse all the vertices are marked and posterior calls to the function return an empty set. The function Expand recursively calls itself before calling the function Traverse, and, since the reached recursion level depends on the size of the set V_{exp}, the recursion depth for these calls is $n-1$, which, because of (2), is, in fact, the maximum recursion level reached in the algorithm for this example. However, the space required to store the set M_P greatly increases as the pattern P grows; when P grows from being one single vertex to being a pattern P' with two vertices, each of the $n-1$ occurrences of P can grow into

$n-2$ different occurrences of P'. Therefore, as Expand calls itself $n-1$ times, the amount of space S required by the pattern occurrences can be expressed as

$$\begin{aligned} S &= (n-1) + (n-1)(n-2) + \cdots + (n-1)! \\ &= \sum_{i=2}^{n-1} \frac{(n-1)!}{(n-i)!}, \end{aligned}$$

and, since this process is repeated for each vertex, we can conclude that the space complexity is $O(nn!)$.

We must say that the previous complexity (both, in time and space) is not what we expect in practice, as the analyzed graphs generally have several labels and are not fully connected. However, the previously analyzed scenario does highlight the fact that our algorithm is currently unable to handle large (in the order of thousands of vertices) highly connected graphs with low label diversity. Despite this drawback, AGraP is able to analyze some graphs that arise in real world applications [Acosta-Mendoza et al., 2012, PubChem-Database, 2004, Riesen and Bunke, 2008].

3.4 Experiments

The first experiment involved toy graphs built with the purpose of examining the patterns found by AGraP and show the accomplishments and drawbacks of our algorithm. In the second experiment we used slightly bigger graphs to see how the amount of output patterns varies with different frequency threshold values. In all the experiments we compared the patterns obtained by our algorithm against the patterns obtained by gApprox [Chen et al., 2007], which is the closest algorithm to our work and the only one we have found to mine patterns in a single graph using inexact matching. It is important to mention that, in order to make a comparison against gApprox, we used the function f_{dis} to compute the dissimilarity between graphs and gApprox's support upperbound to compute each pattern support.

Before presenting these experiments, it is important to point out the fact that AGraP finds all the patterns that gApprox does and, depending on the dissimilarity threshold used, it could find some additional ones. In other words, we claim that, given the frequency and dissimilarity thresholds σ and Δ (respectively), if

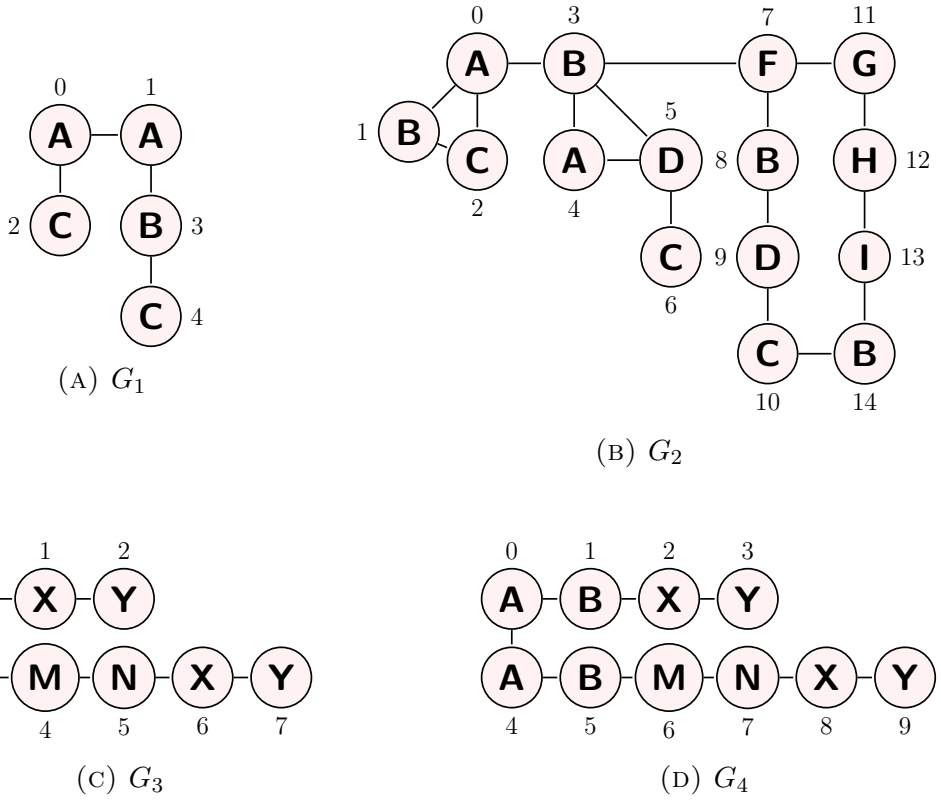
a pattern P is identified by gApprox as a frequent pattern in a graph G, then P will also be identified as a frequent pattern in G by AGraP.

To support this claim let us consider a frequent pattern P identified by gApprox. P is identical to a connected subgraph g in G that has enough occurrences to have a support greater than σ. Since AGraP analyzes each vertex in G, it will eventually analyze a vertex in g and it will be frequent; from there, it will grow the vertex and reach g at some point. By then, all the occurrences of P identified by gApprox, will be also identified by AGraP, since they are connected subgraphs in G that could differ from P by some labels or edges, but without surpassing the dissimilarity threshold Δ. Thus, AGraP will find all the occurrences that gApprox found for P and, perhaps, some occurrences with more or less vertices than P. Therefore, since AGraP will identify the same or more occurrences of P than gApprox, the support of a pattern P in AGraP will be equal or greater than the support of the same pattern in gApprox. Consequently, all frequent patterns found by gApprox will be also frequent in AGraP. Moreover, some non-frequent patterns in gApprox could be frequent in AGraP, due to the additional occurrences.

In this and subsequent chapters, the experiments were performed in a computer with an Intel Core i7 (3.4 Ghz) processor and 32GB in RAM, running the Ubuntu 11.10 distribution of the 64 bit GNU Linux operating system. The implementation of the proposed algorithm and gApprox were done in Python, using the NetworkX library [Hagberg et al., 2008].

3.4.1 Patterns found by AGraP

The toy graphs used in the first experiment are shown in Fig. 3.4; we tried to choose graphs that are small and simple yet big enough to show the difference that allowing structural differences in occurrences brings to the mining process. In Table 3.1 it is possible to observe the frequent patterns found in G_1 using AGraP and gApprox with $\sigma = 2$ and $\Delta = 2$, without allowing label substitution. The first column shows the found patterns; the second one lists the occurrences of each pattern in G_1, together with their edit costs for vertices and edges. In each vertex list representing an occurrence, we indicate a missing vertex through the symbol '-', while a negative sign preceding a vertex ID points out that the vertex does not have a match in the considered pattern, i.e., it is part of a path between two matched vertices. Finally, in the third and fourth columns, for gApprox and



(A) G_1 (B) G_2 (C) G_3 (D) G_4

FIGURE 3.4: *Graphs analyzed in order to compare the patterns found by gApprox (Chen et al, 2007) against those found by the proposed algorithm.*

AGraP respectively, the support of each pattern in G_1 is shown; the letters N.F. (Not Found) are used to indicate cases where gApprox couldn't find a pattern. It can be observed that gApprox found two patterns whereas AGraP found five: the same two patterns gApprox found, as we expected, and three additional ones.

Analyzing G_2 with $\sigma = 2$, $\Delta = 2$ and specifying that label `'A'` can be replaced by label `'F'`, resulted in 27 frequent patterns found by gApprox and 51 found by AGraP; again, AGraP found the 27 patterns found by gApprox and 24 additional ones. Table 3.2 shows two of the frequent patterns found by AGraP. In both cases, we can see that these patterns were not found by gApprox, since it could not find enough occurrences to have a support above the frequency threshold.

Through $G1$ and G_2 we corroborate that AGraP is able to find all the patterns found by gApprox, plus some additional patterns that are missed when structural differences in vertices are not allowed. However, AGraP does not always find the whole set of subgraphs that inexactly match a given pattern; due to the lack of antimonotonicity when using the function f_{rel} as dissimilarity measure, there could be some subgraphs that are similar enough (satisfying the threshold Δ given by

TABLE 3.1: *Frequent patterns found in the graph G_1 shown in Fig. 3.4, using $\sigma = 2$ and $\Delta = 2$. In the second column the symbol '-' indicates that a vertex is missing, while a negative sign indicates that a vertex does not have a match in the considered pattern. The letters* N.F. *in the third column indicate that the algorithm did not find the pattern shown in the first column of that row.*

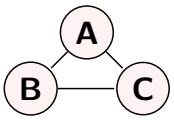
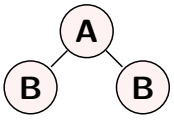
Pattern	**Identified occurrences** $\{Vertices\}$ / (v_{edit}, e_{edit})	**Support** gApprox	**Support** AGraP
A	$\{0\}$ / $(0,0)$ $\{1\}$ / $(0,0)$	2	2
C	$\{2\}$ / $(0,0)$ $\{4\}$ / $(0,0)$	2	2
A C	$\{0,2\}$ / $(0,0)$ *Only AGraP:* $\{1,-0,2\}$ / $(1,3)$ $\{1,-3,4\}$ / $(1,3)$ $\{1,-\}$ / $(1,1)$	1/N.F.	2
A B	$\{1,3\}$ / $(0,0)$ *Only AGraP:* $\{0,-1,3\}$ / $(1,3)$ $\{0,-\}$ / $(1,1)$	1/N.F.	2
A B C	$\{1,3,4\}$ / $(0,0)$ *Only AGraP:* $\{0,-1,3,4\}$ / $(1,3)$ $\{0,-,2\}$ / $(1,3)$ $\{0,-,-\}$ / $(2,2)$	1/N.F.	2

the user) to $P' = P \cup \{v\}$, but not to P, to be considered pattern occurrences. Since we create the occurrence set of the pattern P' based on the occurrence set of P, those patterns would be missed by the proposed algorithm. This is better understood through an example.

3.4.1.1 Patterns found using f_{rel}

Just for the sake of illustration, let us consider the patterns found by AGraP, with the dissimilarity function f_{rel}. To this end, we appropriately modified the algorithm; mainly, we modified the function Expand, lines 7-9, so that P_E and P_T are calculated even if P' is not frequent, since the antimonotonicity property does not hold anymore. For the same reason, in ExpandOccurrence, it is necessary to include all the occurrences in M_P in the set M'_P, as well as the pattern P itself followed by the indicator '-', even if they do not satisfy the dissimilarity

TABLE 3.2: *Two of the frequent patterns found in the graph G_2 shown in Fig. 3.4, using $\sigma = 2$, $\Delta = 2$ and specifying that* `'A'` *label can be replaced by* `'F'` *label. In the second column the symbol* `'-'` *indicates that a vertex is missing, while a negative sign indicates that a vertex does not have a match in the considered pattern. The letters* N.F. *in the third column indicate that the algorithm did not find the pattern shown in the first column of that row.*

Pattern	**Identified occurrences** $\{Vertices\}$ / (v_{edit}, e_{edit})	**Support**	
		gApprox	**AGraP**
	$\{0,1,2\}$ / $(0,0)$		
	*Only AGraP:* $\{4,3,-1,2\}$ / $(1,4)$ $\{4,3,-1,6\}$ / $(1,4)$ $\{4,3,-\}$ / $(1,1)$ $\{4,-,-\}$ / $(2,2)$ $\{7,8,-\}$ / $(2,1)$ $\{7,3,-\}$ / $(2,1)$	1/N. F.	3
	$\{0,1,3\}$ / $(0,0)$ $\{7,8,3\}$ / $(1,0)$ $\{7,3,8\}$ / $(1,0)$		
	*Only AGraP:* $\{4,3,-1,1\}$ / $(1,3)$ $\{4,3,-1,8\}$ / $(1,3)$ $\{4,3,-\}$ / $(1,1)$ $\{4,-,3\}$ / $(1,1)$ $\{4,-,-\}$ / $(2,2)$ $\{7,8,-\}$ / $(2,1)$ $\{7,3,-\}$ / $(2,1)$ $\{7,-,8\}$ / $(2,1)$ $\{7,-,3\}$ / $(2,1)$	1/N.F.	2

threshold Δ, since they could eventually grow into an occurrence that does satisfy the threshold. Additionally, the estimation of ℓ (Section 3.2) must be modified according to the definition of f_{rel}, resulting in the constraint

$$f_{rel}(P', g') = \frac{\kappa(v_{edit} + (\ell - 1))}{(|V(P)| + 1) + (|V(g)| + \ell)} + \frac{(1-\kappa)(e_{edit} + \ell + e_{new})}{(|E(P)| + e_{new}) + (|E(g)| + \ell)} \leq \Delta.$$

Solving directly this inequality for ℓ leads to a quadratic equation with many terms, so, it can be easier to solve it by using a bisection scheme.

When analyzing the graphs G_3 and G_4 shown in Fig. 3.4, using $\sigma = 2$, $\Delta = 0.5$ and none equivalence between labels, AGraP does not identify the pattern P_2, shown in the Table 3.3, as a frequent pattern in G_3, although we would expect

that $\{0, 1, 2\}$ and $\{3, -4, -5, 6, 7\}$ were considered as pattern occurrences (the second having vertices 4 and 5 as "extra" vertices) and that, therefore, the pattern were considered as frequent. Why AGraP does not identify P_2 as a frequent pattern? When using f_{rel}, due to the bound ℓ imposed to the amount of "surplus" vertices that an occurrence could have (in this case $\ell = 1$), $\{3, -4, -5, 6\}$ is not an occurrence of the pattern P_1. When P_1 grows into P_2, the dissimilarity between P_2 and the subgraph induced by vertices $\{3, 4, 5, 6, 7\}$ is below the dissimilarity threshold, but, since the set of occurrences of P_2 is created by extending the occurrences of P_1, it is not possible to find this occurrence of P_2 and the pattern is not identified as frequent. It is important to notice that the same would happen if vertices 2 and 7 were identical subgraphs of any size. By contrast, pattern P_3 is identified as frequent in G_4 (here the value of ℓ is 2 and, thus, $\{4, 5, -6, -7, 8\}$ is a pattern occurrence), so, it is also possible to identify the pattern P_4, which is obtained by growing P_3.

We can conclude that there could be patterns that, despite satisfying the given thresholds σ and Δ, will be missed by AGraP when using f_{rel}, due to the bound on the number of extra vertices than an occurrence can have with respect to the pattern it represents. Moreover, the necessary modifications to work with f_{rel} notably increase the cost of the algorithm. These examples show the kind of challenges that must be faced when f_{rel} is used; in this thesis we focus on developing algorithms based on f_{dis} and leave the development of algorithms based on f_{rel} for future research.

3.4.2 Experiments using different σ and Δ values

In the second experiment we analyzed the amount of patterns found by AGraP and gApprox [Chen et al., 2007] using different values for the frequency threshold σ and the dissimilarity threshold Δ. In this experiment, we used five graphs with 100, 150, 175, 200 and 250 vertices, respectively, which were generated using the graph generator developed by Kuramochi and Karypis [2001], also used by Chen *et al.* in the reported experiments with gApprox in [Chen et al., 2007].

We first fixed the value of the dissimilarity threshold ($\Delta = 4$) and examined several frequency threshold values that were big enough to avoid computationally expensive mining processes (due to the combinatorial explosion) and stopped at the value where frequent patterns were single nodes. Then, we fixed the value

TABLE 3.3: *Some of the frequent patterns found in the graphs G_3 and G_4 shown in Fig. 3.4, using $\sigma = 2$, $\Delta = 0.5$ and using the dissimilarity function f_{rel}. In the second column the symbol '-' indicates that a vertex is missing, while a negative sign indicates that a vertex does not have a match in the considered pattern. The letters* N.F. *in the third column indicate that the algorithm did not find the pattern shown in the first column of that row.*

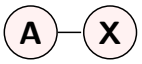
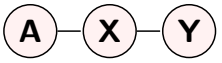
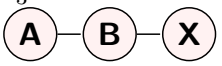
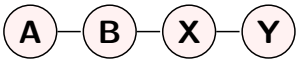
	Pattern	**Identified occurrences** $\{Vertices\}$ / (v_{edit}, e_{edit})	**Support AGraP**
In G_3:	P_1 	$\{0,1\}$ / $(0,0)$ $\{0,-\}$ / $(1,1)$ $\{3,-\}$ / $(1,1)$	1/N.F.
	P_2 	$\{0,1,-\}$ / $(1,1)$ $\{0,-,-\}$ / $(2,2)$ $\{3,-,-\}$ / $(2,2)$	1/N.F.
In G_4:	P_3 	$\{0,1,2\}$ / $(0,0)$ $\{0,1,-\}$ / $(1,1)$ $\{0,-,-\}$ / $(2,2)$ $\{4,5,-0,-1,2\}$ / $(2,4)$ $\{4,5,-6,-7,8\}$ / $(2,4)$ $\{4,5,-\}$ / $(1,1)$	2
	P_4 	$\{0,1,2,3\}$ / $(0,0)$ $\{0,1,2,-\}$ / $(1,1)$ $\{0,1,-,-2,3\}$ / $(2,4)$ $\{0,1,-,-\}$ / $(2,2)$ $\{0,-,-,-\}$ / $(3,3)$ $\{4,5,-0,-1,2,3\}$ / $(2,4)$ $\{4,5,-6,-7,8,9\}$ / $(2,4)$ $\{4,5,-,-\}$ / $(2,2)$	2

TABLE 3.4: *Amount of patterns found by AGraP and gApprox in the second experiment, using $\Delta = 4$ and different values of σ.*

	G_{100}		G_{150}		G_{175}		G_{200}		G_{250}	
σ	AGraP	gApprox	AGraP	gApprox	AGraP	gApprox	AGraP	gApprox	AGraP	gApprox
5	142	22	83	20	1296	88	681	29	2859	145
6	33	9	20	13	162	16	87	17	307	30
7	13	7	6	6	28	8	12	9	65	13
8	3	3	2	2	6	5	5	5	16	7
9	1	1	1	1	4	4	0	0	7	5
10	1	1	0	0	1	1	0	0	1	1

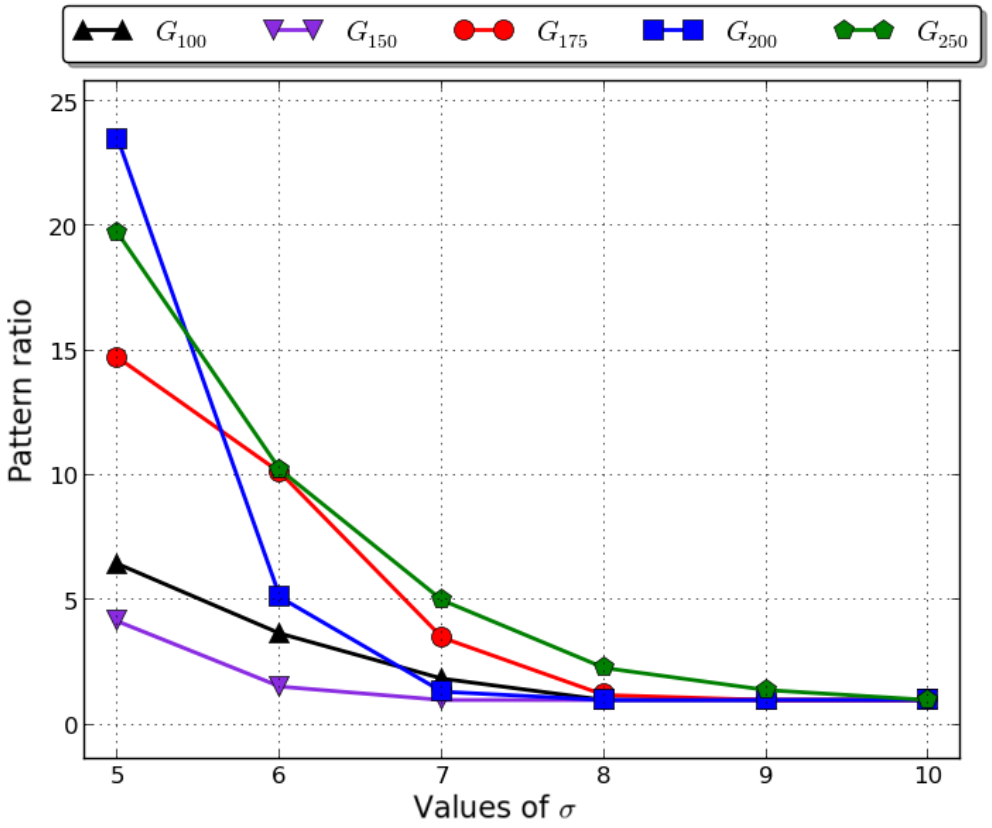


FIGURE 3.5: *Ratio of the number of patterns found by AGraP respect to the number of patterns found by gApprox, using* $\Delta = 4$ *and different values of* σ.

TABLE 3.5: *Amount of patterns found by AGraP and gApprox in the second experiment, using* $\sigma = 5$ *and different values of* Δ.

	G_{100}		G_{150}		G_{175}		G_{200}		G_{250}	
Δ	AGraP	gApprox	AGraP	gApprox	AGraP	gApprox	AGraP	gApprox	AGraP	gApprox
1	22	20	20	20	43	33	29	29	78	45
2	62	22	39	20	261	43	124	29	491	78
3	72	22	39	20	453	68	181	29	1000	121
4	142	22	83	20	1296	88	681	29	2859	145
5	203	22	101	20	2481	98	1279	29	5521	153
6	313	22	178	20	4897	105	2880	29	10859	157

of the frequency threshold ($\sigma = 5$) and chose the same amount of dissimilarity thresholds. Both fixed values, $\Delta = 4$ and $\sigma = 5$, were chosen with the only concern being the time required for the experiment (in both cases the most expensive performance required less than 10^4 seconds). We did not allow label substitution in this experiment, thus, the dissimilarity between graphs could be only structural.

Tables 3.4 and 3.5 show the amount of patterns found in this experiment for each graph and each value of σ and Δ; Figs. 3.5 and 3.6 show the ratio of the number of patterns found by AGraP and the number of patterns found by gApprox. The

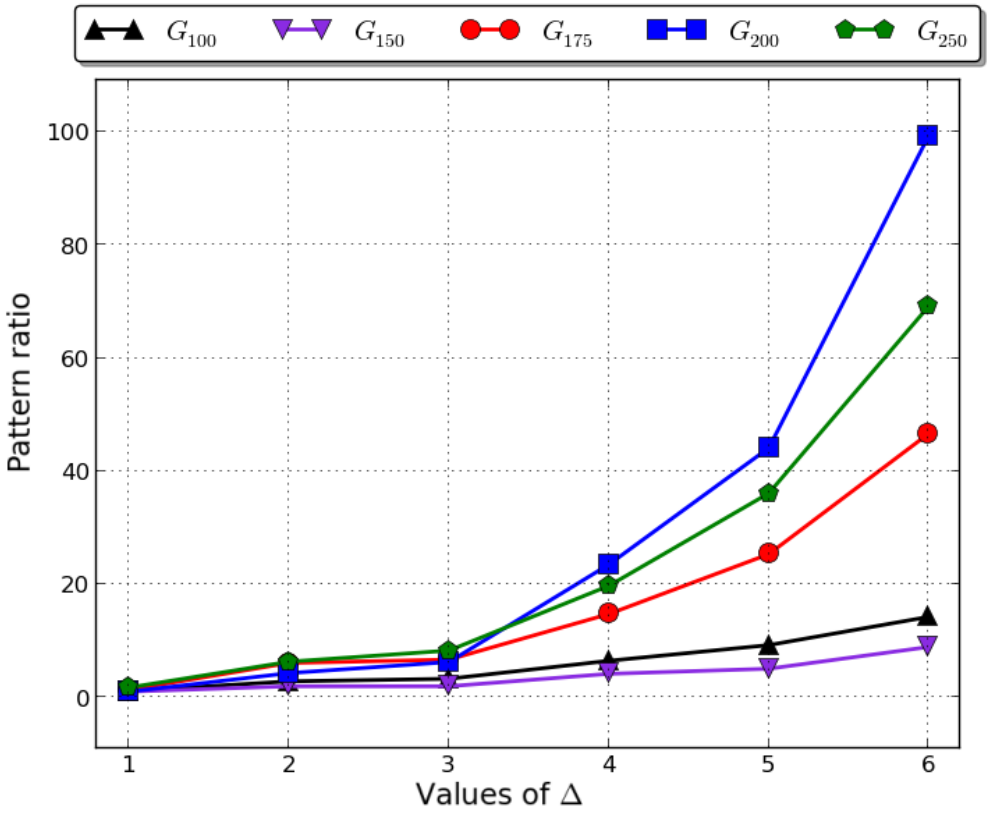


FIGURE 3.6:]
Ratio of the number of patterns found by AGraP respect to the number of patterns found by gApprox, using $\sigma = 5$ and different values of Δ.

ratio is in every case greater or equal to 1, as it was expected since AGraP finds all the patterns that gApprox does, plus some additional ones.

A smaller value of the frequency threshold increases the ratio of additional patterns found by AGraP and it can be as high as 23 (graph G_{200} with $\sigma = 5$). In a similar fashion, increasing the dissimilarity threshold increases the amount of additional patterns found by AGraP, which in this example was until 99 times the amount of patterns found by gApprox (graph G_{200} with $\Delta = 6$). We can see that as we increase the frequency threshold we eventually reach a ratio equal to 1, because high frequency thresholds lead to single-node patterns that can be found by both, gApprox and AGraP.

We can also see that increasing the dissimilarity threshold Δ does have an effect on the number of patterns found by gApprox in some graphs (G_{175}, Table 3.5), due to the fact that gApprox only allows structural differences in edges, however, in other graphs, it does not have an impact at all (G_{150} and G_{200}). Let us remember that, given a new pattern $P' = P \diamond v$, matching occurrences identified by gApprox

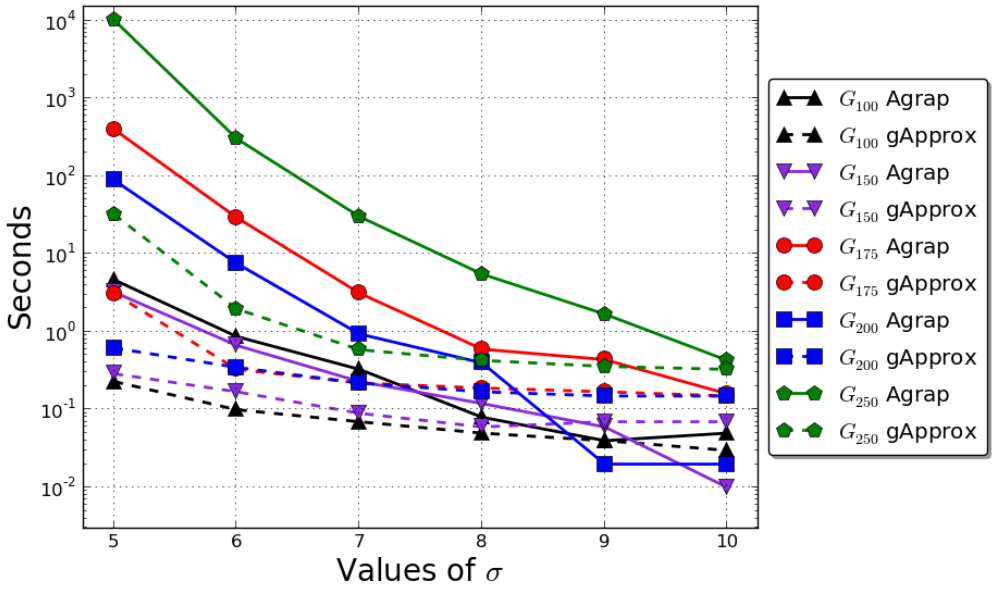


FIGURE 3.7: *Time required by AGraP and gApprox to find frequent patterns in the graphs of the second experiment, using* $\Delta = 4$ *and different values of* σ*. Time is shown in seconds and logarithmic scale.*

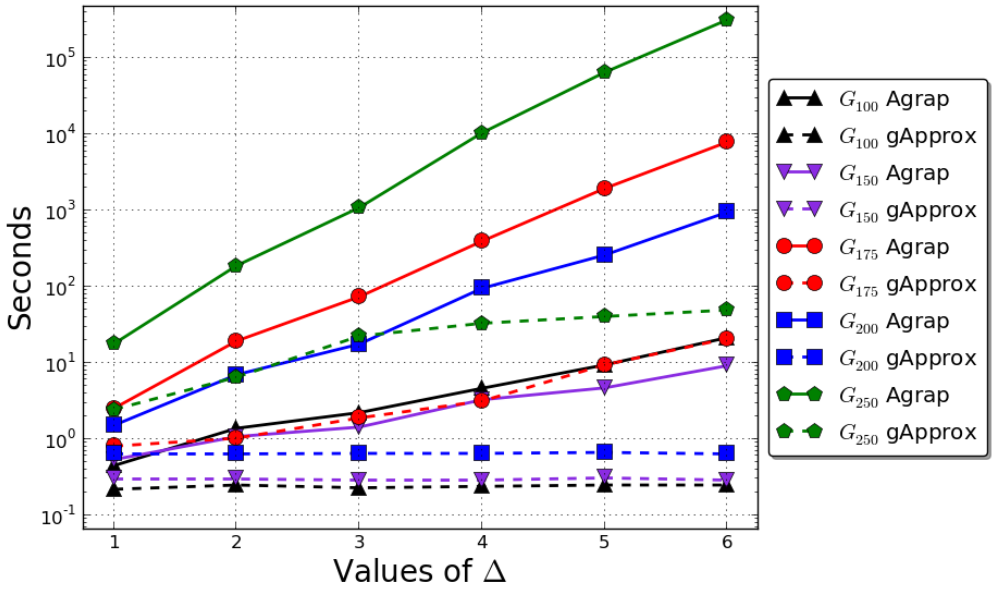


FIGURE 3.8: *Time required by AGraP and gApprox to find frequent patterns in the graphs of the second experiment, using* $\sigma = 5$ *and different values of* Δ*. Time is shown in seconds and using logarithmic scale.*

depend on a new vertex, matching the label of v, being (directly) connected to a P occurrence; if there is not such a vertex, no new occurrences of P' will be identified, no matter the value of Δ. By contrast, allowing structural differences in nodes makes, in every case, a huge difference in the number of identified frequent patterns.

Figs. 3.7 and 3.8 show the running time required by each algorithm. As we expected, AGraP requires more time than gApprox to find patterns, although the time difference lessens as we increase the frequency threshold (or decrease the dissimilarity one) for the mining task and the patterns found by both algorithms have smaller size.

3.5 Discussion

Our proposed algorithm is able to successfully use inexact matching during the mining process, although, as expected, it requires more time than its closest algorithm, gApprox. However, allowing structural differences, in both vertices and edges, leads to find some patterns that otherwise would be missed.

Since we know that AGraP identifies the patterns found by gApprox plus additional patterns, and the ratio between the amount of patterns by both algorithms is considerable, we can conclude that, by allowing structural differences in vertices, AGraP identifies a significant amount of patterns that would be otherwise missed; in this way the proposed algorithm accomplishes its goal.

It is important to inquire about the usefulness of those "extra" mined patterns but, before this question, it comes the issue of *the amount* of patterns found by our algorithm. In one of the cases shown in the previous section, we found over ten thousand patterns in a graph with only 250 vertices. Although we have interest precisely in the extra patterns found by allowing structural differences, we also acknowledge the need to reduce the amount of output patterns in order to gain practicality.

Chapter 4

Interesting patterns

We aim at finding a set of interesting patterns able to provide an insight into the analyzed graph and be useful in data mining tasks, like classification and clustering. Having a set of frequent patterns represents a good starting point; although there are typical drawbacks associated to these patterns, like redundancy, frequent patterns reflect strong associations and carry underlying information on the data that can be useful when trying to analyze it. Yan et al. [2008] point out that, after analyzing different objective functions to mine significant graphs, it is possible to observe that most significant patterns likely fall into the high-quantile of frequency, i.e., that if we rank all subgraph patterns according to their frequency, significant patterns often are within the first patterns enlisted (they call this phenomenon *the frequency association*).

Therefore, the objective we pursue in this chapter is to obtain a subset of patterns that are not only frequent, but *interesting*. By adding constraints to the type of patterns that we find, we expect to improve the quality of the output set, dealing with shortcomings typically associated to frequent patterns, like the *gargantuan* [Vreeken and Tatti, 2014] amount of discovered patterns, that could definitely diminish the usefulness of the set.

Attaining a balance between finding new information and finding a manageable amount of it is a main concern for us, since the use of inexact matching – particularly by allowing structural differences in vertices – brings a considerable increment in the amount of mined patterns and makes it highly desirable to reduce the size of the pattern set before even exploring the issue of its usefulness.

As stated in Chapter 2, in the literature there can be found discussions on what constitutes objective interestingness in patterns, without having context from an specific problem, and there are some properties mentioned as characteristics of interesting patterns. Remarkably, closed and maximal patterns are commonly mentioned as interesting sets due to their ability to condense information [Spyropoulou et al., 2014, Vreeken and Tatti, 2014].

In our search for interesting frequent graphs, and in this chapter, we start by exploring closed and maximal patterns, proposing the algorithms CloseAFG and MaxAFG respectively, and then proceed to consider other constraints in order to reduce pattern redundancy while covering the original pattern set, obtaining, in this way, a subset of frequent patterns that could be considered interesting. The later idea is implemented by proposing a greedy algorithm, IntAFG (Interesting Approximated Frequent Graphs). Finally, we show through experiments the efficacy of the described approaches in reducing the size of the pattern set obtained by AGraP.

4.1 Closed patterns

As mentioned earlier, a pattern is closed if it cannot be grown into patterns that preserve its support, i.e., all the patterns that can be grown from it are less frequent. Closed patterns are remarkable due to their ability to condense the information contained in the whole set of mined patterns without information loss. Once we know the set of closed patterns C, the whole set of frequent patterns can be recovered from the patterns in C. Furthermore, due to the very definition of closed patterns, we can use their support to infer the support of their subgraphs. Therefore, it comes as no surprise that closed patterns are commonly mentioned when there is interest in reducing the size of a pattern set.

Being able to preserve the information relative to the support of each pattern is not a minor advantage, since the frequency of a pattern can be related to its significance according to other measures. As pointed out at the beginning of this chapter, the frequency association discussed by Yan et al. [2008] states this relation between high frequency and significance. So, even if we have no other measure of significance, the frequency of a pattern can be a good measure of its relevance and

Algorithm 4.1: CloseAFG Algorithm

Input: G: graph to be analyzed,
σ: frequency threshold,
Δ: dissimilarity threshold,
D: dictionary that specifies equivalences between labels *(optional).*

Output: P: set of patterns in G.

```
1 P ← ∅;
2 for v ∈ G do
3     Mark v as "explored";
4     M_v ← List of vertices with the same or equivalent label to v;
5     if |M_v| ≥ σ then
6         Add {v} to P;
7         P_Traverse ← Traverse({v}, M_v) ;
8         P ← mergeClosed(P, P_Traverse);
```

having closed patterns we still can have access to the information concerning the particular frequency of each pattern.

Since we want to find closed patterns and we also want to use inexact matching, allowing structural differences in edges and vertices, during the mining process, we modified the algorithm AGraP in order to filter out all the non-closed patterns as they are computed by the algorithm. In particular, we modified the function ExpandClosed so that it stores a pattern P only when, after recursive calls to ExpandClosed and Traverse, we do not find any pattern with the same support as P.

Function Traverse(P, M_P)

Input: P: candidate pattern,
M_P: list of P occurrences and their edit cost.

Output: P_{Exp}: set of closed patterns that can be obtained growing P.

```
1 V_exp ←Vertices connected to P that have not been explored or marked ;
2 for u ∈ V_exp do
3     marked(u) ← True;
4 P_Exp ← ExpandClosed(P, M_P, V_exp);
5 for u ∈ V_exp do
6     marked(u) ← False;
```

The Algorithm 4.1 shows the algorithm CloseAFG. Details on some functions are omitted or simplified, since they are essentially as described in Chapter 3. In the function ExpandClosed, however, we can see that in lines 10 – 16 we examine the

Function ExpandClosed(P, M_P, V_{exp})

Input: P: candidate pattern,
M_P: list of P occurrences and their edit cost,
V_{exp}: list of unexplored vertices connected to P.
Output: P_P: set of closed patterns obtained by expanding P.

```
1  P_P, P_E, P_T ← ∅;
2  for v_exp ∈ V_exp do
3      P' ← P ∪ {v_exp};
4      M'_P ← ExpandOccurrence(P, M_P, v_exp);
5      V'_exp ← V_exp\{v_exp};
6      if support(P') ≥ σ then
7          P_E ← ExpandClosed(P', M'_P, V'_exp);
8          P_T ← Traverse(P', M'_P);
9      P_P ← P_P ∪ P_E ∪ P_T;
10     keepPattern ← True;
11     for newPattern in P_E ∪ P_T do
12         if support(newPattern) == support(P') then
13             keepPattern ← False ;
14             break;
15     if keepPattern then
16         P_P ← P_P ∪ P'
```

support of the patterns grown from the pattern P' in order to decide whether it is closed or not and, therefore, whether it should be stored; this test is made at every recursion level, so, by the end of the initial call to Expand, we have only the closed patterns that can be grown from P', including (probably) P' itself.

Notice that we still could get repeated or non-closed patterns, since the support comparison is only made within patterns derived from the same vertex and, as we explore patterns grown from different vertices, it is necessary to make a further filtering in order to ensure that all repeated and non-closed patterns are filtered out. Take, for example, the graph shown in Fig 4.1 and the frequency and dissimilarity thresholds $\sigma = 3$ and $\Delta = 1$, respectively. When CloseAFG starts exploring from vertex 0, the identified closed pattern is `A-B` with support 3 and occurrences in vertices $\{0, 1\}$, $\{4, 5\}$ and $\{7, 8\}$ (besides $\{4, -\}$ and $\{7, -\}$ which, in this example, does not affect the support). When the Algorithm explores patterns grown from vertex 1, it identifies the closed pattern `B` with support 3 (ocurrences in $\{1\}$, $\{5\}$ and $\{8\}$); later on, when the algorithm starts from vertex 4, the pattern `A-B-C`, with support 3, is identified (occurrences $\{4, 5, 6\}$, $\{7, 8, 9\}$ and $\{0, 1, -\}$). We

Function mergeClosed(A, B)

Input: A, B: lists of closed graph patterns and their support
Output: C: single list of closed graph patterns and their support

```
1  for (patA,patB) in A × B do
2      if (patA and patB have the same support) and (patB is equal or smaller in
       size than patA) then
3          if dissimilarity(patA,patB) is 0 then
4              Remove patB from B;
5              Mark patA;
6  for (patA,patB) in A × B do
7      if (patA and patB have same support) and (patA is smaller than patB) and
       (patA is NOT marked) then
8          if dissimilarity(patA,patB) is 0 then
9              Remove patA from A;
10 C ← A ∪ B;
```

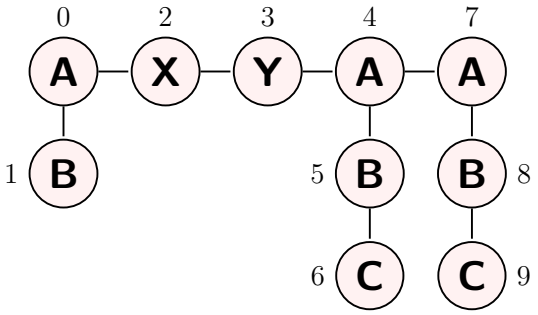


FIGURE 4.1: *Graph analyzed with CloseAFG, using $\sigma = 3$ and $\Delta = 1$. When starting from vertex 0, pattern* `A-B` *is identified as closed pattern, but when starting from vertex 4, the closed pattern* `A-B-C` *is identified, which subsumes the pattern* `A-B`*. The function* mergeClosed *is needed to deal with this situation.*

can see that, from these three patterns, only `A-B-C` is actually a closed pattern. Therefore, whenever we merge the set of already-found closed patterns with a new "branch" of closed patterns, we use the function mergeClosed (line 8, Algorithm 4.1).

In mergeClosed we compare pattern lists, A and B, of closed patterns. In this function, we compare the support of each pair of patterns (pat_A, pat_B) in $A \times B$. The support of each pattern is computed only once (in the function ExpandClosed) and stored, so it does not have to be recomputed. When two patterns have the same support, we try to determine if the smaller one is a subgraph of the other one, and we use the dissimilarity function f_{dis} for this purpose.

We can also see in the function mergeClosed that, seemingly unnecessary, we traverse the lists A and B twice (lines 1 and 6). We do that in order to avoid, as many times as possible, the calculation of the dissimilarity between pat_A and pat_B, which is the most expensive step in the function. Since A and B are lists of closed patterns, we know that $pat_B \not\subseteq pat_{B*}$ for any pat_B, pat_{B*} in B with the same support; therefore, if $pat_B \subseteq pat_A$, we know that pat_A cannot be "absorbed" by any pattern in B, i.e., it cannot be a subgraph of any pattern in B with the same support. Thus, traversing the lists A and B twice allow us to take advantage of the fact that they are lists of closed patterns to avoid computing f_{dis}.

4.2 Maximal patterns

Maximal patterns are those frequent patterns that cannot be grown into another frequent pattern, i.e., they cannot be grown without stop satisfying the frequency threshold given by the user. Since the constraint satisfied by maximal patterns is a particular case of the constraint imposed on closed patterns (all maximal patterns are also closed patterns, but there could be some closed patterns that are not maximal), there are fewer maximal patterns than closed ones, therefore, the output set is smaller.

The appeal of maximal patterns is precisely the fact that they generate an smaller set of frequent patterns and still allow to recover the whole set of frequent patterns. With maximal patterns, however, the information regarding patterns' support is lost. Therefore, the decision between using maximal patterns or closed ones lies on the trade-off between keeping all the information and reducing the output set further.

Since we are interested in reducing the size of the set of mined patterns, we propose a modification of AGraP that focuses on finding maximal patterns: the algorithm MaxAFG. In this algorithm, we use inexact matching in the same way we did in AGraP, but storing only frequent graphs that cannot grow further into a frequent pattern. The pseudocode of the algorithm is shown in Algorithm 4.2, and the idea, like in the case of AGraP and CloseAFG, is to use a depth-first search strategy through recursive calls to functions that grow patterns (ExpandMaximal and Traverse).

Algorithm 4.2: MaxAFG Algorithm

Input: G: graph to be analyzed,
σ: frequency threshold,
Δ: dissimilarity threshold,
D: dictionary that specifies equivalences between labels *(optional).*

Output: P: set of frequent patterns in G.

```
1 P ← ∅;
2 for v ∈ G do
3     Mark v as "explored";
4     M_v ← List of vertices with the same or equivalent label to v;
5     if |M_v| ≥ σ then
6         Add {v} to P;
7         P_Traverse ← Traverse({v}, M_v) ;
8         P ← mergeMaximal(P, P_Traverse);
```

The function Traverse is essentially the same function used by CloseAFG. On the other hand, in lines 10-13 of ExpandMaximal we can see that only maximal patterns are stored. The support of each pattern is calculated in ExpandMaximal (line 7) at every recursion level, thus, P_E and P_T contain only patterns that are maximal. Since P' is a frequent pattern and a subgraph of any pattern in $P_E \cup P_T$, we know that P' is maximal only when both, ExpandMaximal and Traverse, output empty sets, meaning that the pattern P' could not grow further without stop satisfying the frequency threshold.

We also use a function, mergeMaximal, to combine the maximal patterns found when exploring graphs that can be grown from different vertices. This function is based on the same ideas explained for mergeClosed, in the previous section, i.e., it uses the idea that a pattern in A that "absorbed" a smaller pattern in B in the first `for` loop cannot be absorbed by a bigger pattern in B (in the second `for` loop). The function mergeMaximal is invoked in line 8 of the MaxAFG algorithm.

Finally, it is worth mentioning that both algorithms, MaxAFG and CloseAFG, have the same complexity than AGraP, since they still have to examine the same set of pattern candidates than AGraP does, in order to identify the maximal and closed patterns, respectively. Both algorithms do some extra comparisons inside the Expand function, but they do not increase the overall complexity of the algorithm. Moreover, the time required by the functions mergeClosed and mergeMaximal is less than the time required by the function unique in AGraP,

Function ExpandMaximal(P, M_P, V_{exp})

Input: P: candidate pattern,
M_P: list of P occurrences and their edit cost,
V_{exp}: list of unexplored vertices connected to P.
Output: P_P: set of patterns obtained by expanding P.

```
1  P_P, P_E, P_T ← ∅;
2  for v_exp ∈ V_exp do
3      P' ← P ∪ {v_exp};
4      M'_P ← ExpandOccurrence(P, M_P, v_exp);
5      Calculate the support of P'. If it satisfies σ, add P' to P_P;
6      V'_exp ← V_exp\{v_exp};
7      if support(P') ≥ σ then
8          P_E ← ExpandMaximal(P', M'_P, V'_exp);
9          P_T ← Traverse(P', M'_P);
10     if P_E ∪ P_T = ∅ then
11         P_P ← P_P ∪ P'
12     else
13         P_P ← P_P ∪ P_E ∪ P_T
```

Function mergeMaximal(A, B)

Input: A, B: lists of maximal graph patterns
Output: C: single list of maximal graph patterns

```
1  for (patA,patB) in A × B do
2      if patB is equal or smaller in size than patA then
3          if dissimilarity(patA,patB) is 0 then
4              Remove patB from B;
5              Mark patA;
6  for (patA,patB) in A × B do
7      if (patA is smaller than patB) and (patA is NOT marked) then
8          if dissimilarity(patA,patB) is 0 then
9              Remove patA from A;
10 C ← A ∪ B;
```

since, at every call of these functions, the lists that are being compared are smaller when dealing with maximal and closed patterns.

4.3 Interesting set of patterns

Although CloseAFG and MaxAFG find fewer patterns than AGraP, the size of their output sets could still be impractically big. Therefore, it would be useful to reduce further the number of patterns. Moreover, we are also interested on exploring the way in which some of the characteristics that have been mentioned in the literature for interesting patterns can be used to choose a subset of frequent patterns that could be considered interesting.

Although the term *interesting* is clearly associated to the background of the problem considered, we do not want to use an idea of *interestingness* linked to some specific task. Instead, we want to obtain a subset of the patterns found by AGraP in such a way that the selection process is not related to a particular application. To this end, we explore an approach inspired by the definition of a *basis* of a given vector space.

In Linear Algebra, a basis is a set of vectors that combines both, lineal independence and the ability of, together, being able to describe the whole vector space. In the same fashion, we want to look for some independence in our patterns, lessening the redundancy in the set of frequent patterns, and, at the same time, we want a subset being able to retain information from the original pattern set.

This idea also matches what some authors have proposed as interesting. In particular, the nine criteria that Geng and Hamilton [2006] enumerate, in the context of association rules, to help to determine whether or not a pattern is interesting are: conciseness, peculiarity, diversity, novelty, surprising, coverage, reliability, utility and actionability (see Table 2.1). Broadly speaking, we could say that the former characteristics are related with the fact that interesting patterns are clearly distinct among them, while the later are related to the fact that interesting patterns are expected to capture useful information from the set they were extracted.

In order to obtain an interesting set, we propose a greedy algorithm to select a pattern subset in a post-mining stage, aiming at covering the original pattern set while lessening redundancy among selected patterns. The pseudocode of this selection algorithm is shown in Algorithm 4.3.

We begin by choosing the pattern G_0 that "covers" the highest amount of patterns (line 1), understanding that one pattern covers another if its dissimilarity is below a given threshold Δ^*; this pattern represents our initial interesting set $\mathcal{I}$.

Afterwards, we remove, from the set of frequent patterns F, the patterns covered by G_0 (line 3). Then, we proceed to select – and add to $\mathcal{I}$ – more graphs based on how distant they are from the patterns already in the interesting set (lines 4-7).

The stop condition used in the `while` loop (line 4) ensures that we keep adding graphs to $\mathcal{I}$ until we can guarantee that every pattern in the original set F is covered by at least one pattern in $\mathcal{I}$. On the other hand, once we choose the initial pattern G_0, patterns subsequently added to $\mathcal{I}$ are selected merely by their dissimilarity towards previously chosen patterns G_k, meaning that we attempt to privilege low redundancy over compression – which could be better achieved if we chose G_k with the same criterion that we followed to find G_0. As explained before, low redundancy is one of the two main ideas that we chose to follow in order to obtain an interesting set of patterns (the other idea is coverage, which is already guaranteed in the algorithm).

Algorithm 4.3: IntAFG

Input: F: set of patterns,
Δ^*: dissimilarity threshold.
Output: $\mathcal{I}$: interesting pattern set.

1 $G_0 \leftarrow \max_{g_i \in F} |\{g_j : g_j \in F, dissimilarity(g_i, g_j) \leq \Delta^*, i \neq j\}|$;
2 $\mathcal{I} \leftarrow \{G_0\}$;
3 $F \leftarrow F \backslash \{g_j : g_j \in F, dissimilarity(g_j, G_0) \leq \Delta^*\}$;
4 **while** F *is not empty* **do**
5 $\quad G_k \leftarrow \max_{g_j \in F} \sum_{G_i \in \mathcal{I}} dissimilarity(g_j, G_i)$;
6 $\quad \mathcal{I} \leftarrow \mathcal{I} \cup \{G_k\}$;
7 $\quad F \leftarrow F \backslash \{g_j : g_j \in F, dissimilarity(g_j, G_k) \leq \Delta^*\}$;

Although any dissimilarity measure can be used in the algorithm, choosing f_{dis} seems like the proper choice, since we want to form the interesting set from patterns mined by AGraP and it is reasonable that the same dissimilarity function that AGraP uses to decide whether a graph can be considered a pattern occurrence, is used at this stage to decide whether a pattern is covered by another one.

Finally, we want to highlight the fact that, through this algorithm, we find an interesting *set* of patterns and not a set of interesting patterns, since it is *the set* the one that captures our proposed idea of interestingness and not individual patterns. As pointed out by Vreeken and Tatti [2014] for pattern set mining algorithms: *"a pattern is only as good as its contribution to the set"*, contribution that, in our case, is measured by low redundancy and the number of covered patterns (in that order).

4.4 Experiments

We performed experiments roughly following those shown in Chapter 3, for AGraP. In the first one, we used toy graphs, like the ones presented in Section 3.4.1, to observe the way in which imposing different constraints (maximal, closed or interesting) on frequent patterns affects the number and type of mined patterns. In the second, we observed how the amount of patterns varies, following the different approaches, as different values of frequency and dissimilarity thresholds are used. In every case, we compared the patterns obtained against those obtained with AGraP, without further constraints, trying to observe the compression achieved by the algorithms proposed in this chapter. Also, like in the previous chapter, we compared the patterns against those obtained by gApprox [Chen et al., 2007]; to this end, we used again gApprox's support upperbound to compute the support of each pattern.

4.4.1 Interesting patterns

Let us consider the graph G_2, introduced in Section 3.4.1, and shown in Fig. 4.2, and suppose that it is acceptable to replace the label G by the label A. Using AGraP and the parameters $\sigma = 3$, $\Delta = 2$, we obtain the eleven patterns shown in Fig. 4.3. In contrast, using the same parameters and gApprox we obtain only the four frequent patterns that are shown in Fig. 4.4. Just as described in the previous chapter, we can observe that, by allowing structural differences in vertices, we obtain almost three times the amount of patterns mined by gApprox. Also, the patterns found by AGraP are larger, as a consequence of allowing to either miss or skip vertices in pattern occurrences; in fact, all the patterns mined by gApprox are subgraphs of those mined by AGraP.

Fig. 4.5 shows the patterns mined using closeAFG and maxAFG, respectively. We can observe that there are fewer maximal patterns than closed ones and that, as mentioned earlier, they are a subset of the closed patterns. We can also see that every pattern found by AGraP is a subgraph of both, closed and maximal patterns. Regarding the number of patterns, in this example, we see that closed patterns are about a third of AGraP patterns, while maximal patterns represent less than a fifth of those found by AGraP.

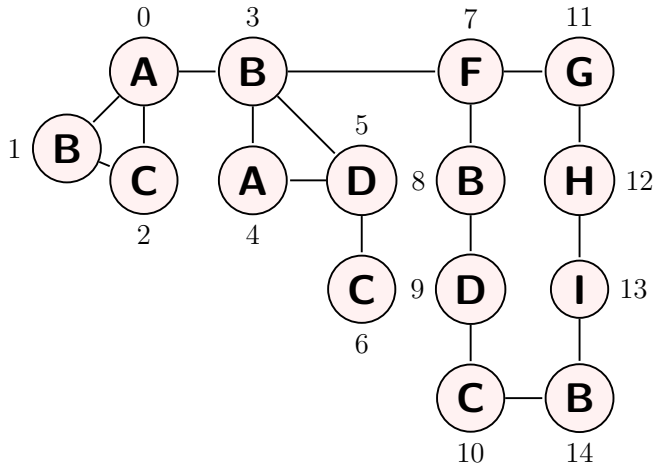


FIGURE 4.2: *Graph G_2, analyzed to compare interesting frequent patterns obtained by using different approaches.*

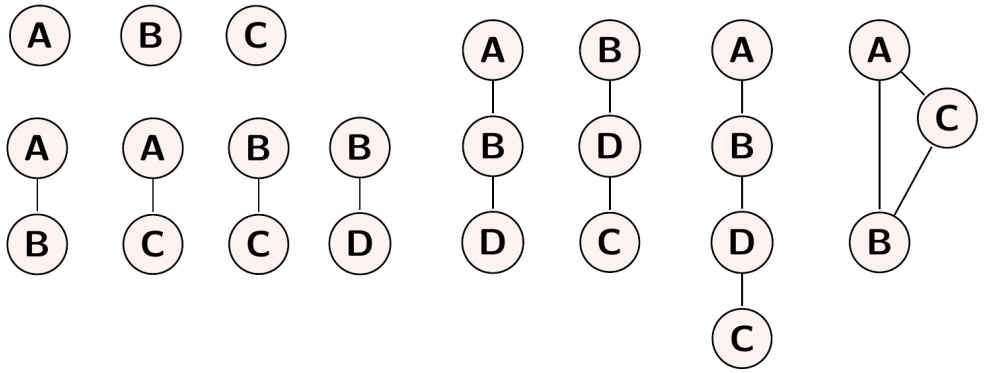


FIGURE 4.3: *Frequent patterns found, in G_2, by AGraP using $\sigma = 3$ and $\Delta = 2$.*

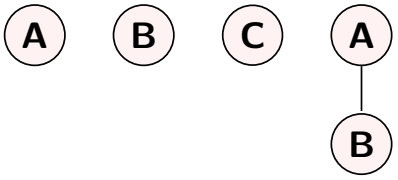
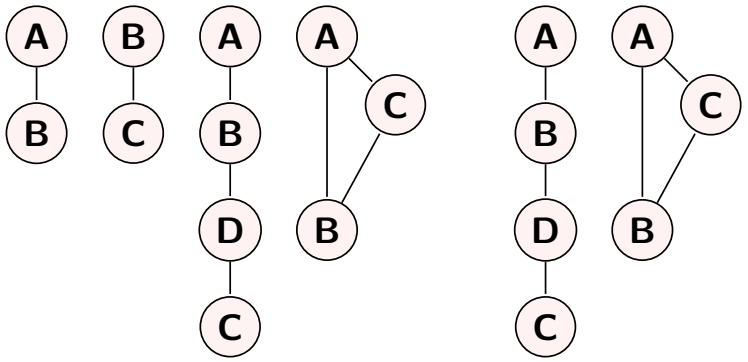


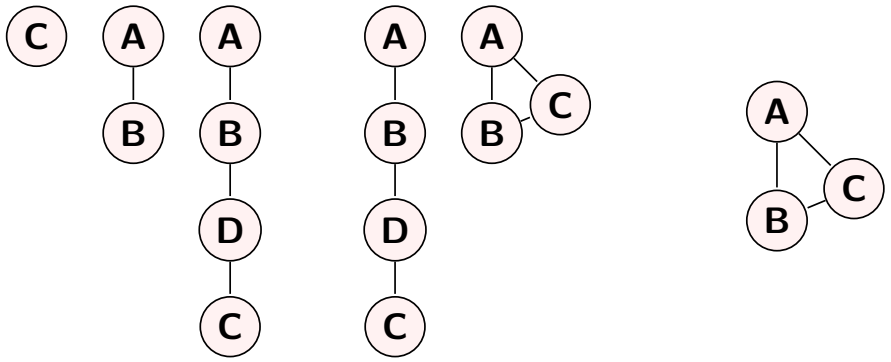
FIGURE 4.4: *Frequent patterns found, in G_2, by gApprox using $\sigma = 3$ and $\Delta = 2$.*

Finally, when we apply the algorithm IntAFG to the set of patterns found by AGraP, CloseAFG and MaxAFG, using $\Delta^* = \Delta = 2$, we obtain the sets shown in Fig. 4.6. Regarding the number of patterns, in every case, the size of the pattern set was reduced to (at least) half of its original size. On the other hand, when we observe the form of the patterns we can see that, although the interesting sets are smaller than the set of gApprox patterns, the nature of the patterns remains different: patterns in Fig. 4.6 are larger and still reflect the effect of allowing structural differences in vertices.



(A) *Closed patterns* (B) *Maximal patterns*

FIGURE 4.5: *Frequent patterns found, in G_2, by* (A) *CloseAFG and* (B) *MaxAFG, using $\sigma = 3$ and $\Delta = 2$.*



(A) *Interesting AGraP* (B) *Interesting CloseAFG* (C) *Interesting MaxAFG*

FIGURE 4.6: *Frequent patterns found, in G_2, by IntAFG using $\Delta^* = \Delta = 2$; after* (A) *AGraP,* (B) *CloseAFG and* (C) *MaxAFG, with $\sigma = 3$ and $\Delta = 2$.*

Although in a very small scale, this is the interestingness we aimed for: a downsized set of patterns preserving some of the insight gained by the use of inexact matching. In this way, though comparable in size to the output given by other algorithms, the set of interesting patterns contains information that would have been missed if structural differences in vertices were not allowed.

4.4.2 Ratio of patterns respect to AGraP

One of our goals when proposing the algorithms described in this chapter was to decrease the amount of mined patterns. In this experiment we wanted to observe the way in which the number of patterns can be reduced by focusing on closed and maximal graphs and by obtaining an interesting subset of the patterns found by AGraP.

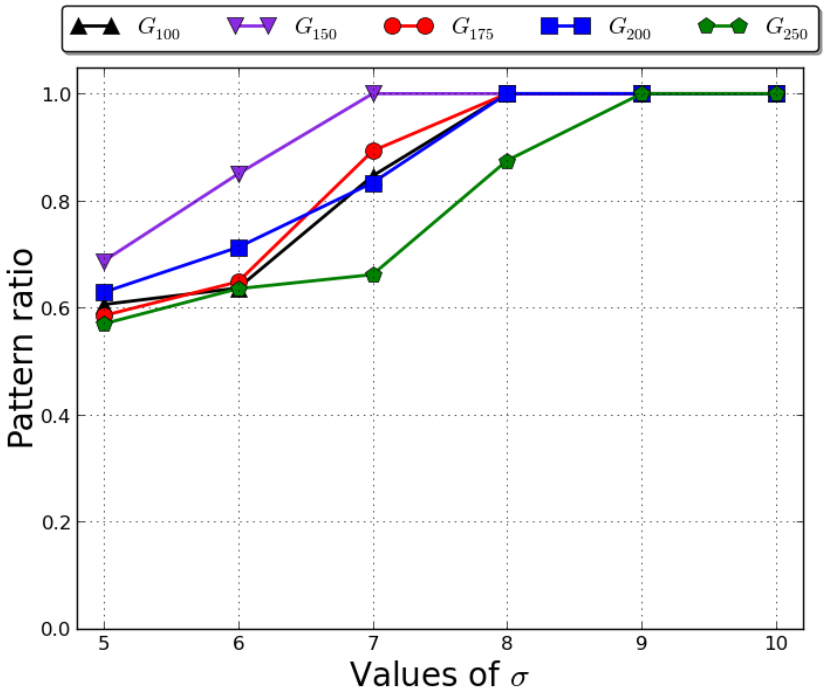
We repeated the experiment showed in Section 3.4.2, but this time we used the algorithms CloseAFG and MaxAFG to mine patterns. We analyzed the same graphs than in the aforementioned experiment, created by using the graph generator developed by Kuramochi and Karypis [2001], and we explored the amount of closed and maximal patterns found by CloseAFG and MaxAFG, respectively, as the values of the frequency and dissimilarity thresholds varied. First, we fixed the value of the dissimilarity threshold at $\Delta = 4$ and used a range of different σ values; then, we fixed $\sigma = 5$ and changed the value of Δ. In both cases, we chose the same ranges of σ and Δ values used in Chapter 3, so that we could observe the variation in the amount of mined patterns with respect to AGraP and make a comparison.

After exploring CloseAFG and MaxAfG, we used the algorithm IntAFG to select interesting patterns among those found by AGraP with the same values of σ and Δ that have been managed and using $\Delta^* = \Delta$. In this way, we were able to compare all the size-reducing approaches described in this Chapter.

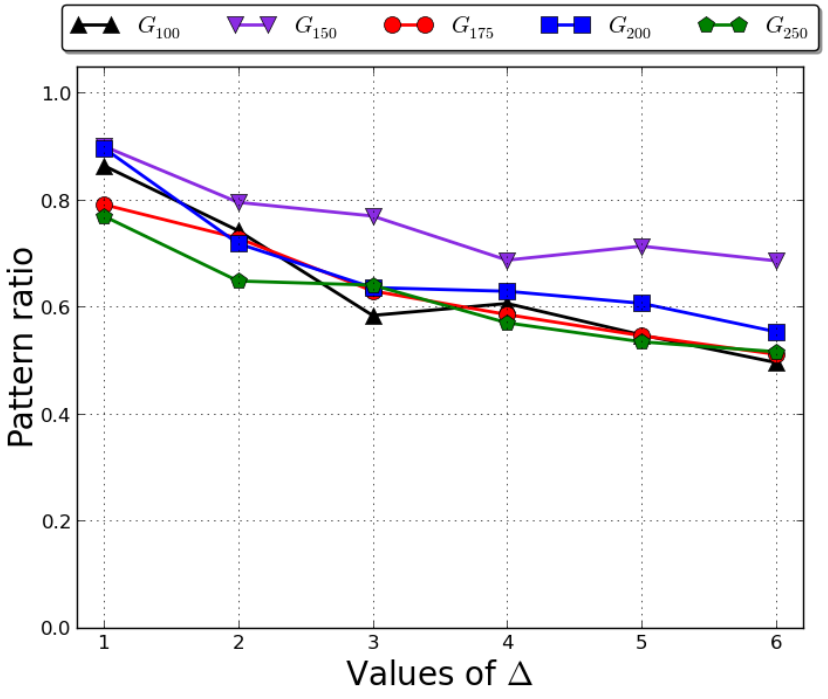
In Figs. 4.7–4.9 we can see the ratio between the number of patterns in the output sets of CloseAFG, MaxAFG and IntAFG and the number of patterns mined by AGraP.

We observe the general trends that were expected: the number of mined patterns increases whenever σ decreases or Δ increases, and the compression ratio is always larger when using MaxAFG than when using CloseAFG. For either small values of σ or large values of Δ, MaxAFG finds patterns that represents approximately 50% of those mined by AGraP, while, for the same parameter values, CloseAFG finds around 60 or 70% of the patterns. Choosing an interesting set of patterns clearly results in the best compression, since, for the aforementioned parameter values, they contain only around 20% of the patterns mined by AGraP.

The difference between the amount of patterns lessens as the values of σ increase (or the values of Δ decrease) and the patterns found by the algorithms – including AGraP – tend to be single vertices. Also, we can observe that the largest compression is achieved when the value of Δ grows. In the case of CloseAFG and MaxAFG, when more dissimilary is allowed, closed and maximal patterns tend to be larger and are able to "subsume" more patterns and, as a consequence, both, maximal and closed patterns, represent a smaller fraction of the whole set of patterns found by AGraP. In the case of IntAFG, we use $\Delta^* = \Delta$; a larger value of

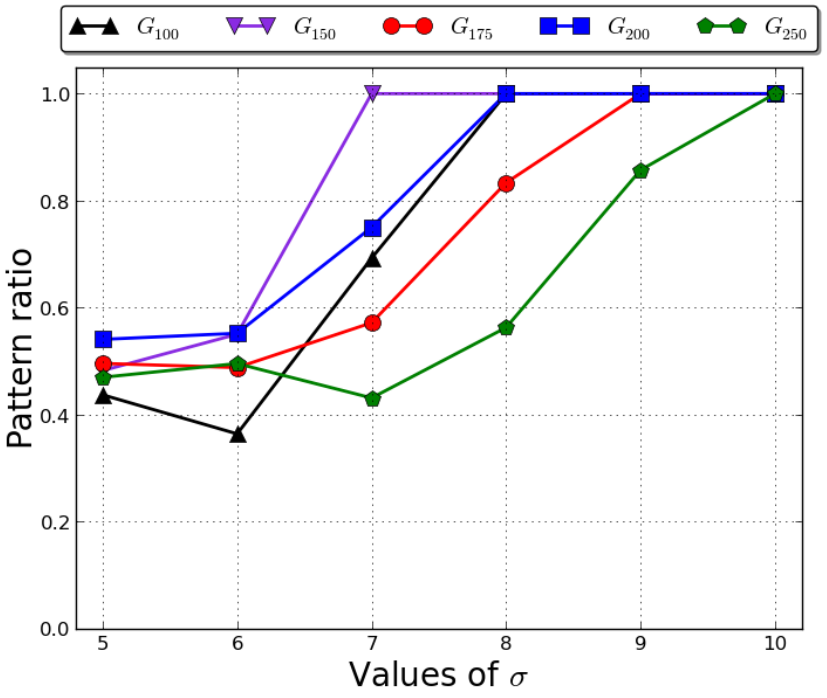


(A)

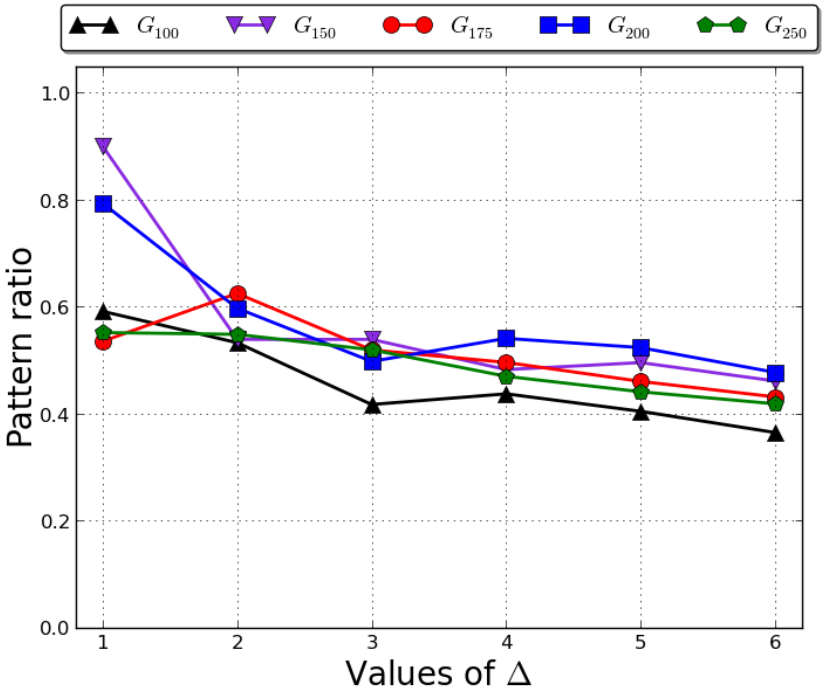


(B)

FIGURE 4.7: *Ratio of the number of patterns found by CloseAFG respect to the number of patterns found by AGraP.* (A) *shows the ratios obtained using* $\Delta = 4$ *and different values of* σ*, while* (B) *shows the ratios obtained with* $\sigma = 5$ *and different values of* Δ*.*

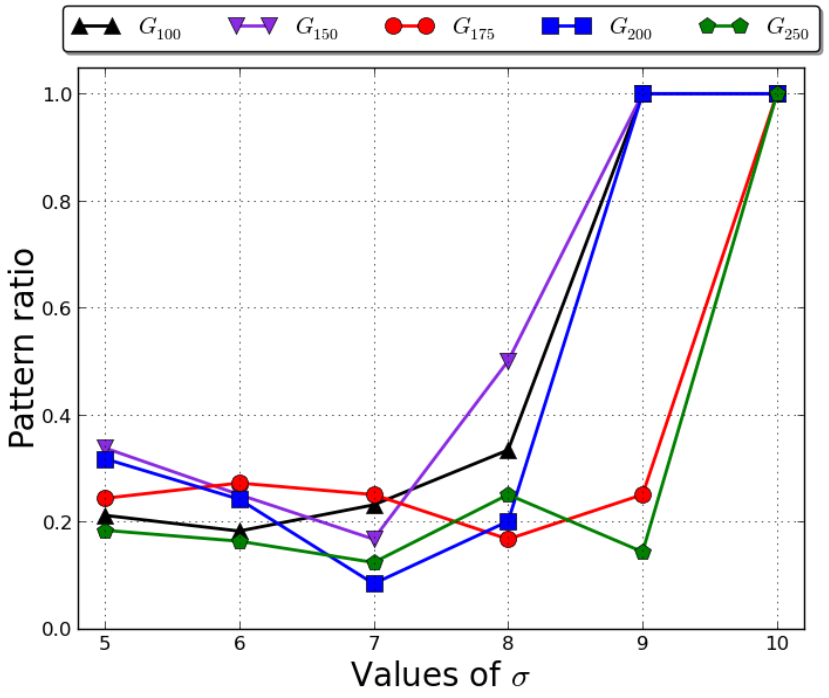


(A)

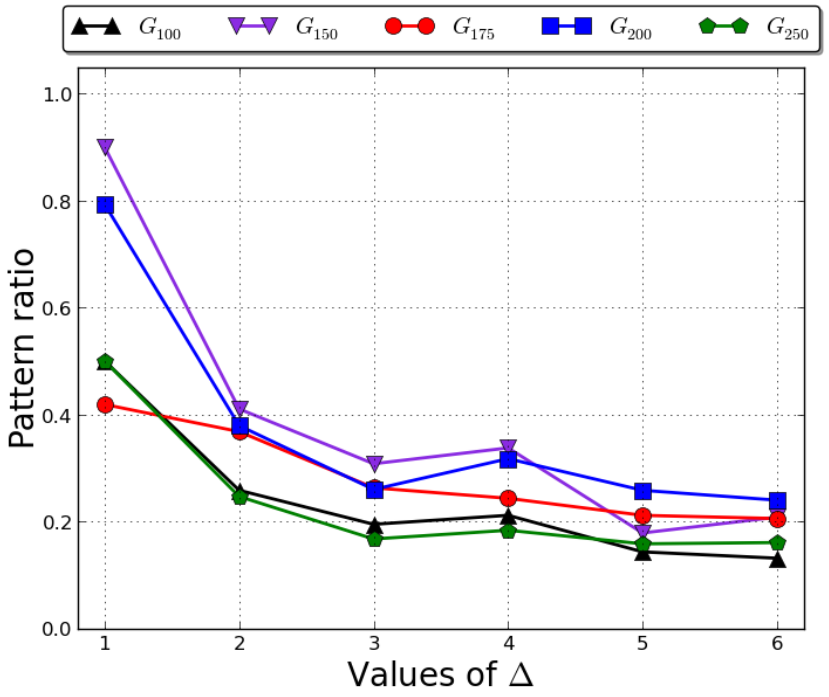


(B)

FIGURE 4.8: *Ratio of the number of patterns found by MaxAFG respect to the number of patterns found by AGraP.* (A) *shows the ratios obtained using* $\Delta = 4$ *and different values of* σ*, while* (B) *shows the ratios obtained with* $\sigma = 5$ *and different values of* Δ*.*



(A)



(B)

FIGURE 4.9: *Ratio of the number of patterns in the set obtained by IntAFG and the number of patterns found by AGraP. In* (A)*, the patterns were found using* $\Delta = 4$ *and different values of* σ*, while in* (B) *it was used* $\sigma = 5$ *and different values of* Δ*. In both cases we used* $\Delta^* = \Delta$ *to choose the interesting set.*

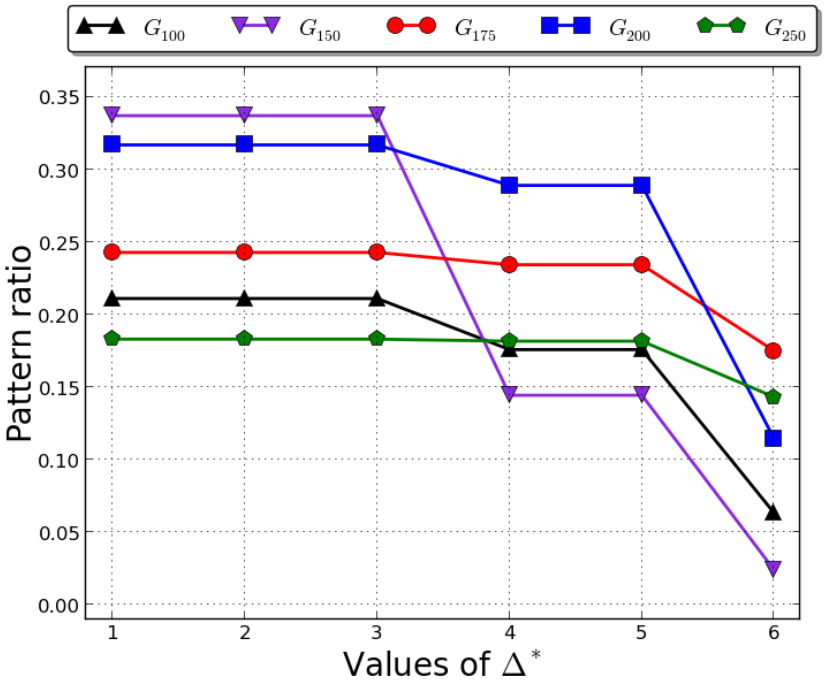


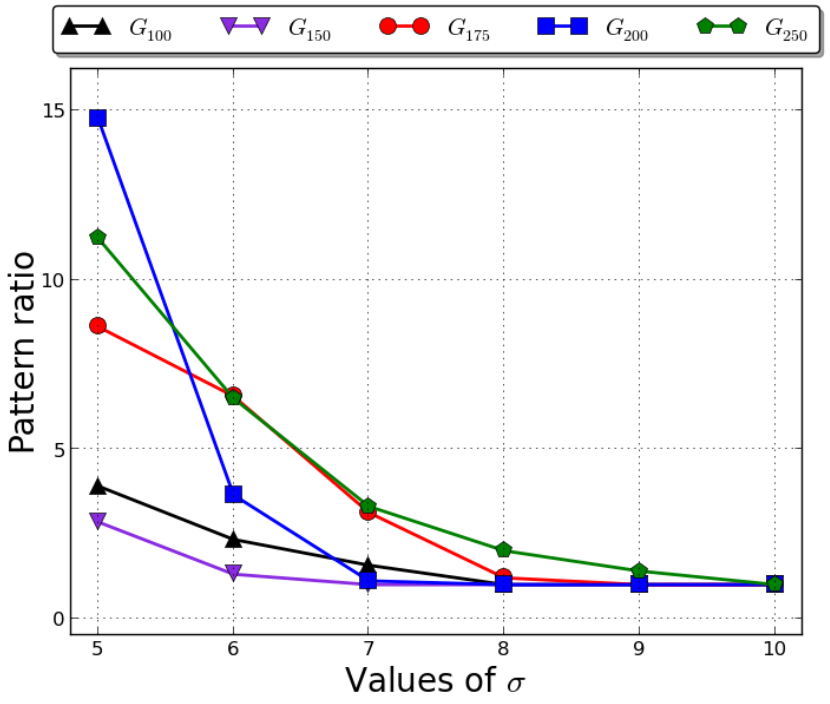
FIGURE 4.10: *Ratio of the number of patterns obtained by IntAFG, with different values of* Δ^**, respect to the number of patterns found by AGraP with* $\sigma = 5$ *and* $\Delta = 4$.

Δ^* translates into patterns with largest covering, which, in turn, means that the final set $\mathcal{I}$ requires less patterns to satisfy the constraint of covering the whole original pattern set. The relationship between the value of Δ^* and the number of interesting patterns obtained is shown in Fig. 4.10, where we used the patterns mined by AGraP with $\sigma = 5$ and $\Delta = 4$, and got different interesting sets by varying the value of Δ^*.

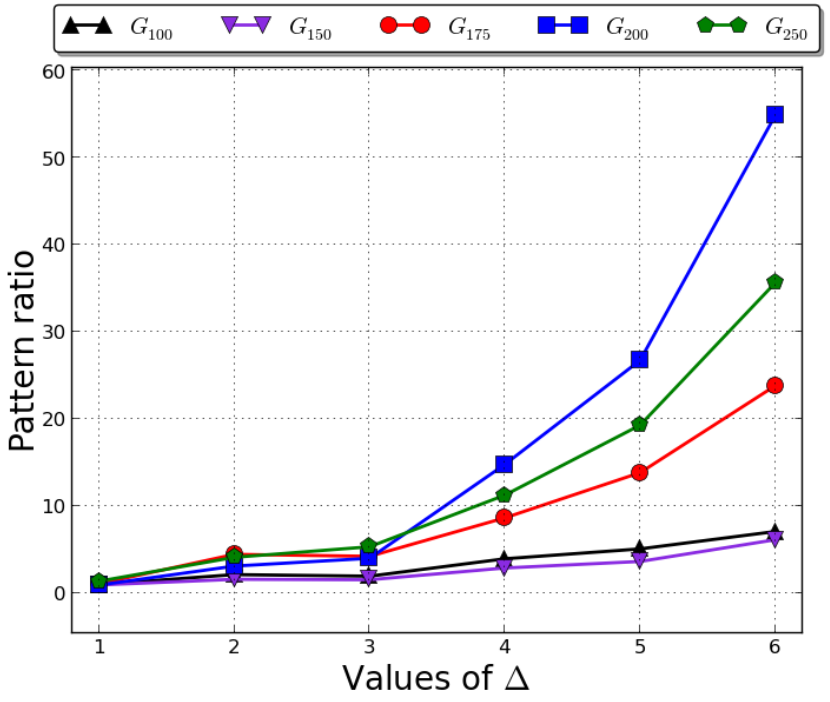
Figs. 4.11–4.13 show the ratio of the number of patterns found by the proposed algorithms with respect to the number of patterns found by gApprox. As we can see, although we succeeded at reducing the number of patterns, the three algorithms lead to an output set larger than that of gApprox.

Table 4.1 puts together the number of patterns obtained by the different algorithms using $\sigma = 5$ and $\Delta = 4$.

Finally, Fig. 4.14 shows the runtime, in logarithmic scale, required by the algorithm IntAFG to obtain the sets shown in Fig. 4.10. We can see that the value of Δ^* does not have any effect on the runtime required by the algorithm IntAFG. Fig. 4.15 shows, in logarithmic scale, the time required by the algorithm in function of the number of patterns in the original set.

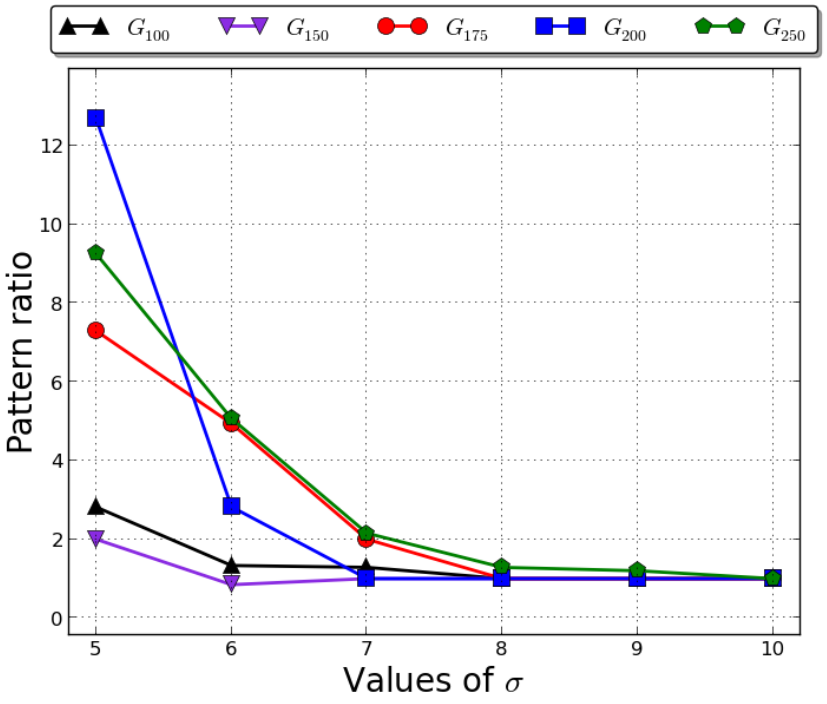


(A)

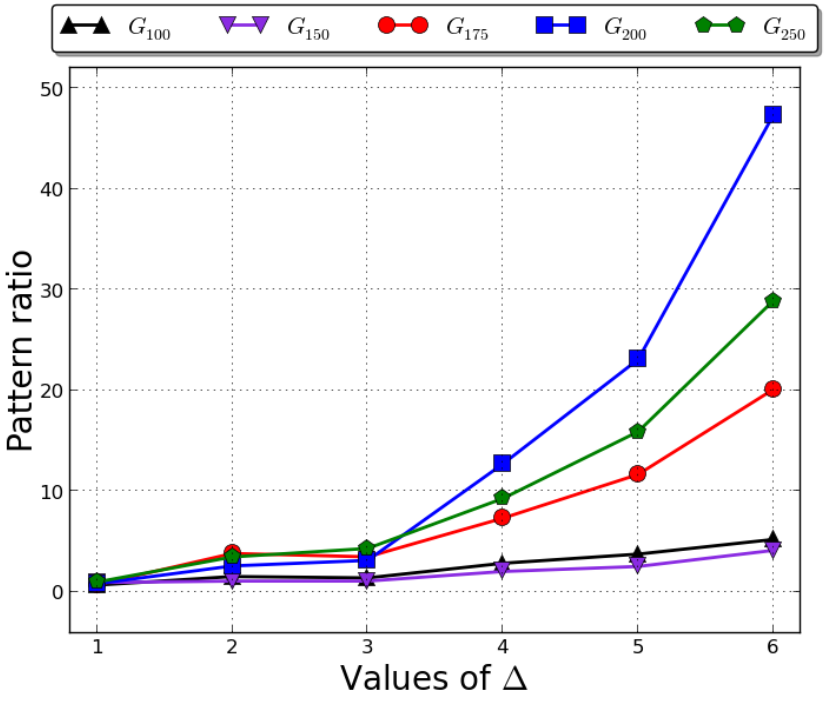


(B)

FIGURE 4.11: *Ratio of the number of patterns found by CloseAFG respect to the number of patterns found by gApprox.* (A) *shows the ratios obtained using* $\Delta = 4$ *and different values of* σ*, while* (B) *shows the ratios obtained with* $\sigma = 5$ *and different values of* Δ*.*

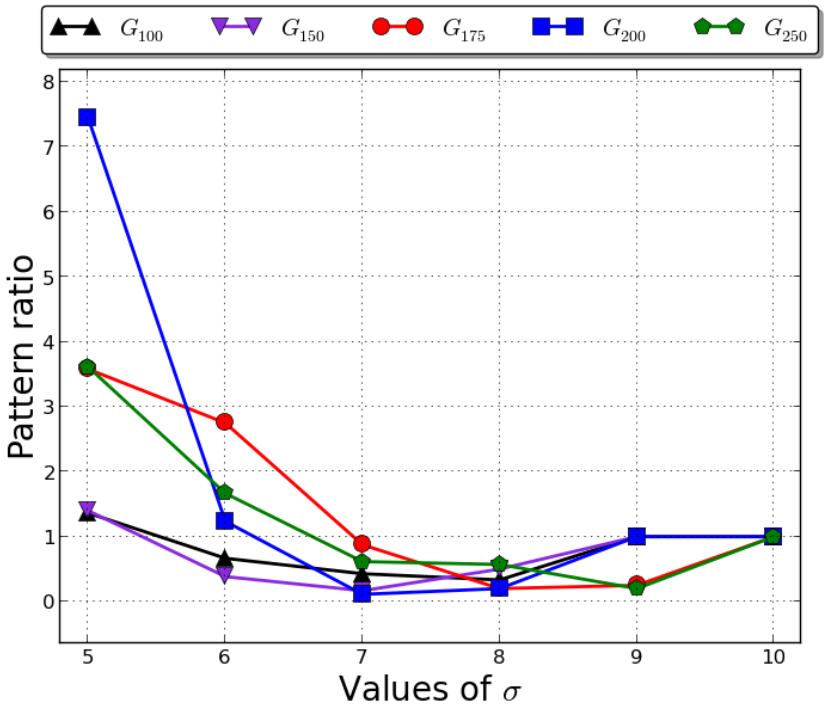


(A)

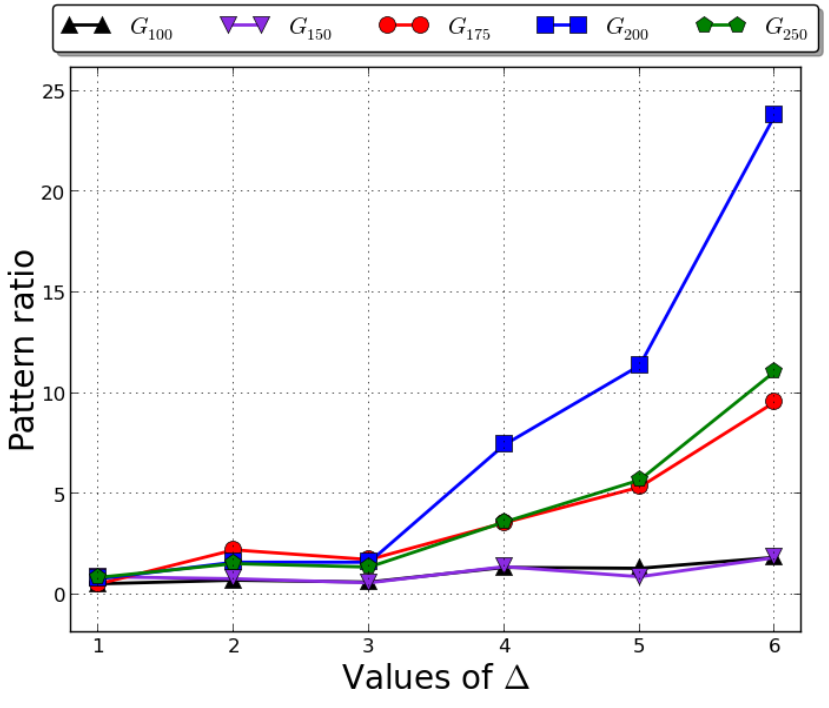


(B)

FIGURE 4.12: *Ratio of the number of patterns found by MaxAFG respect to the number of patterns found by gApprox.* (A) *shows the ratios obtained using* $\Delta = 4$ *and different values of* σ*, while* (B) *shows the ratios obtained with* $\sigma = 5$ *and different values of* Δ*.*



(A)



(B)

FIGURE 4.13: *Ratio of the number of patterns in the set obtained by IntAFG and the number of patterns found by gApprox. In* (A)*, the patterns were found using* $\Delta = 4$ *and different values of* σ*, while in* (B) *it was used* $\sigma = 5$ *and different values of* Δ*. In both cases we used* $\Delta^* = \Delta$ *to choose the interesting set.*

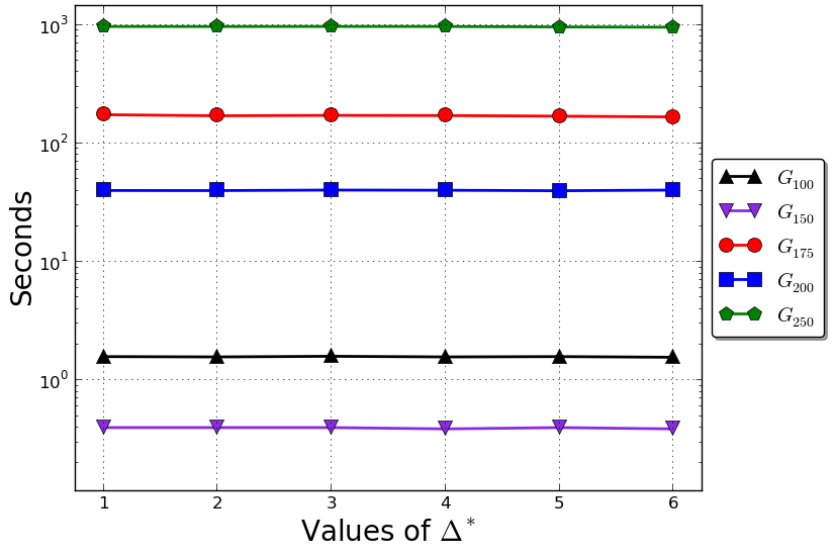


FIGURE 4.14: *Runtime required by IntAFG, with different values of* Δ^**, to obtain an interesting set from patterns found by AGraP with* $\sigma = 5$ *and* $\Delta = 4$*.*

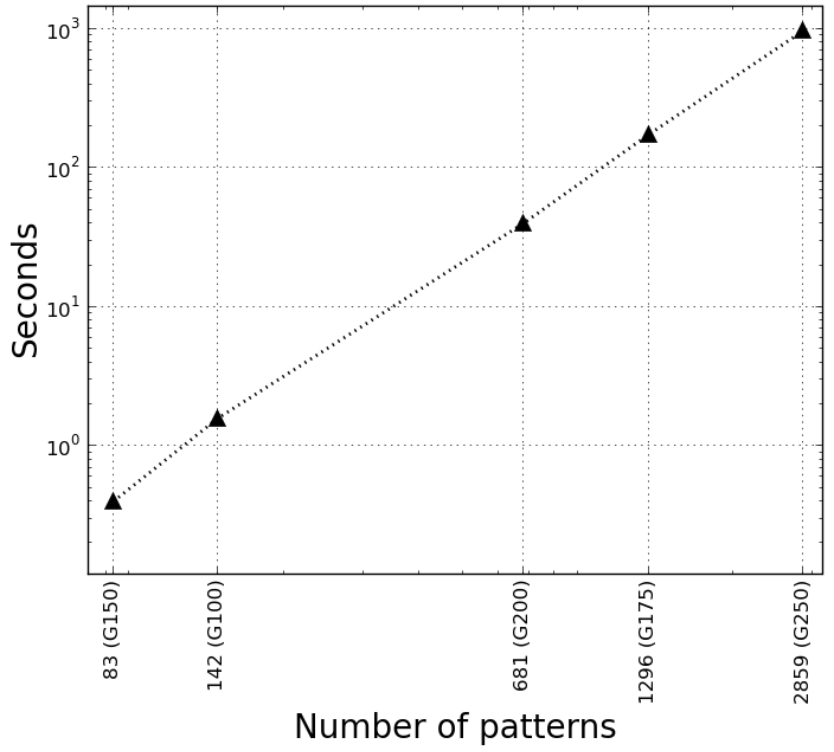


FIGURE 4.15: *Runtime required by IntAFG to obtain an interesting set from patterns found by AGraP with* $\sigma = 5$ *and* $\Delta = 4$*, as a function of the number of patterns in the original set.*

TABLE 4.1: *Number of patterns obtained by the different algorithms with* $\sigma = 5$ *and* $\Delta = 4$.

	G_{100}	G_{150}	G_{175}	G_{200}	G_{250}
gApprox	22	20	88	29	145
AGraP	142	83	1296	681	2859
closeAFG	86	57	758	428	1628
maxAFG	62	40	642	368	1342
Interesting, $\Delta^* = 1$	30	28	315	216	524
Interesting, $\Delta^* = 2$	30	28	315	216	524
Interesting, $\Delta^* = 3$	30	28	315	216	524
Interesting, $\Delta^* = 4$	25	12	304	197	520
Interesting, $\Delta^* = 5$	25	12	304	197	520
Interesting, $\Delta^* = 6$	9	2	227	78	410

4.5 Discussion

Focusing on closed and maximal patterns effectively reduces the amount of patterns found by AGraP, while retaining information captured by the use of inexact matching during the mining process – in all the cases the number of patterns found by CloseAFG and MaxAFG still exceeded the number of those patterns found by gApprox. As expected, focusing on maximal patterns achieves a greater reduction than focusing on closed ones, although there is an information loss related to the pattern support, which could be important depending on the subsequent use intended for the patterns.

On the other hand, the greedy algorithm, which implements the idea of obtaining a pattern subset aiming at covering the original pattern set while lessening redundancy among selected patterns, greatly reduces the amount of patterns and, indeed, it outperforms the previous algorithms at this goal. While CloseAFG and MaxAFG have a positive ratio compared to gApprox with respect to the number of mined patterns, the proposed selection algorithm obtains a set that, depending on the dissimilarity threshold Δ^*, could contain *fewer*, or almost equal, patterns than using gApprox. As in the case of maximal patterns, the level of reduction that is desirable depends on the particular interest of the user, as there is a trade-off between the number of patterns and information loss.

Regardless of the threshold used to select interesting patterns, IntAFG clearly reduces the number of patterns while being flexible at the selection of patterns

that cover the original set by allowing to provide a dissimilarity threshold and the dissimilarity function to be used for selection.

Having fewer patterns makes easier the management of the output information, but the question about the usefulness of the information that we obtain by using inexact matching, particularly by allowing structural differences in vertices, remains. This aspect is explored in the next chapter.

Chapter 5

Pattern usefulness

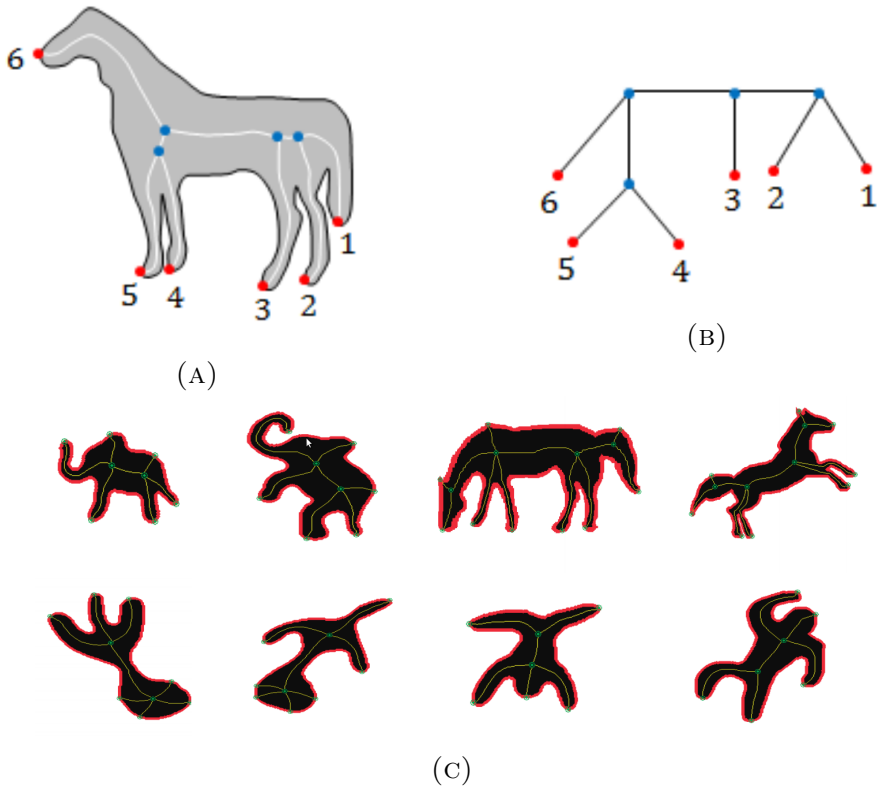
Allowing structural differences in both, vertices and edges, in the frequent graph mining problem, leads to the discovery of "extra" patterns that cannot be found by state of the art algorithms. In this chapter, we explore the usefulness of the pattern sets obtained, through the use of inexact matching, by the algorithms AGraP, CloseAFG and MaxAFG.

We perform tasks of clustering and classification on image graph datasets, aiming to test if the patterns mined by our proposed algorithms are indeed able to capture new potentially useful information about the analyzed data.

Furthermore, the experiments also are intended to show how the interesting subset of patterns obtained by the algorithm IntAFG is able to retain some of the aforementioned extra information and how the reduction on the number of patterns affects the usefulness of the obtained information in comparison with the original output set of AGraP.

5.1 Database description

The tasks were performed on the databases described next.



(A) (B)

(C)

FIGURE 5.1: Structural Image Skeletons *database, used in the experiments presented in this Chapter and created by Pinilla-Buitrago et al. [2013].* (A) *and* (B) *show stages in the graph construction;* (C) *shows some of the classes in the database.*

5.1.1 SIS

The first graph database that we used is known as SIS (Structural Image Skeletons database), composed by 36 graphs representing the skeleton of real-image silhouettes [Pinilla-Buitrago et al., 2013]. Pinilla *et al.* first calculate the skeleton of each image (Fig. 5.1A) in three stages: (1) segmenting the image contour by parts, (2) building the skeleton, and (3) extracting features from the skeleton and creating weighted graphs (Fig. 5.1B) so that images belonging to different classes are distinguishable. The database is divided in 9 classes: elephant, fork, heart, horse, human, L-star, star, tortoise and whale (Fig. 5.1C); each class has 4 graphs. In the graph database, vertices are labeled with body parts, while edges are labeled with the distance between the vertices they connect. The collection has 13 different vertex labels, 211 edge labels, an average graph order of 7 and an average graph size of 6.

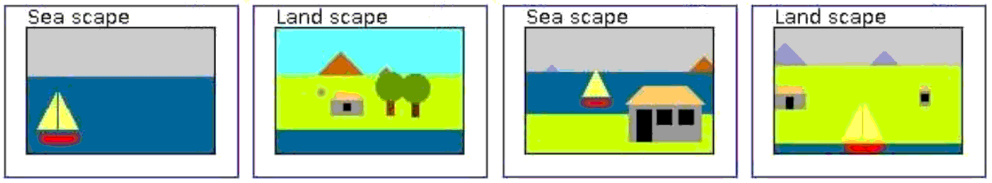


FIGURE 5.2: *Examples of images in the Database* Images*, used in the experiments presented in this chapter and created by Acosta-Mendoza et al. [2012]. The artificial images in this database are labeled as "landscape" and "seascape".*

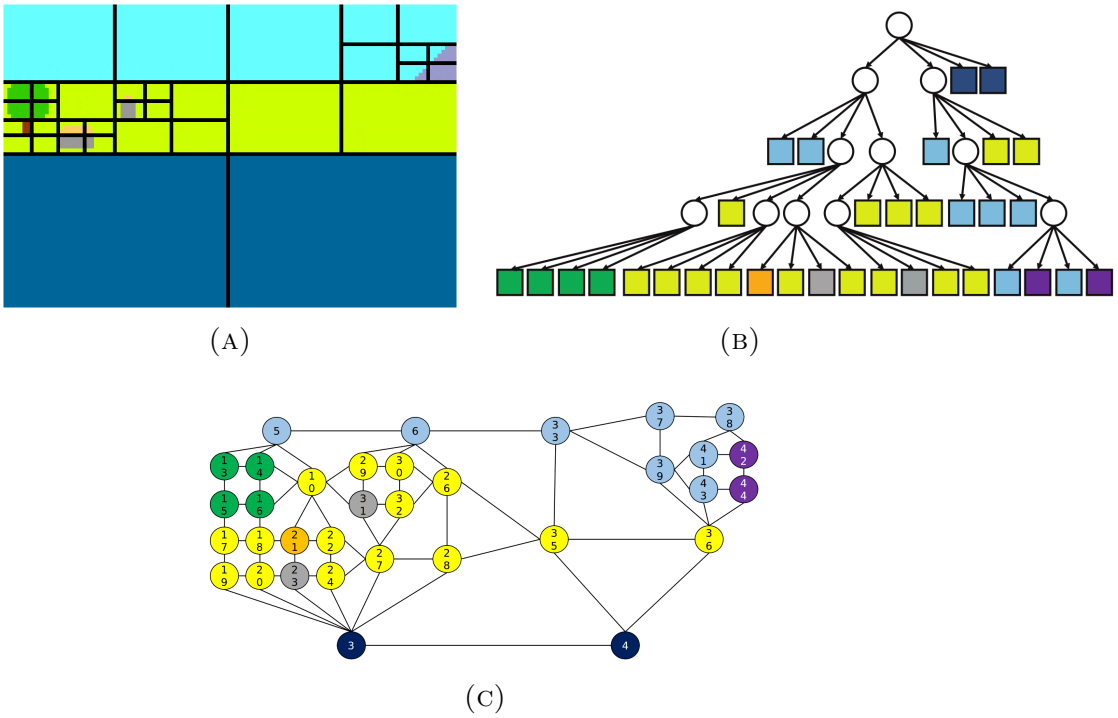
5.1.2 Images

The graph database *Images* was created by Acosta-Mendoza et al. [2012] and consists of 700 graphs built from (artificial) images (Fig. 5.2) obtained using the Coenen image generator [Coenen, 2006]. First, the authors recursively divide the image in quadrants, obtaining a quadtree where the predominant feature (color) of each quadrant is assigned as an attribute to its corresponding quad-tree leaf (Fig. 5.3B). Then, they generate graphs by representing through vertices the quad-tree leaves and their attributes, and connecting each vertex, through edges, with its neighbors in the north, south, east and west directions (Fig. 5.3C). Finally, edges are assigned a label given by an index that depends on the angle of the edge respect to the horizontal axis. Images, and therefore graphs, are labeled as "landscape" (l) and "seascape" (s). To perform our experiments, we chose the 200 graphs with larger label diversity, 100 from each class, from this database, with the average graph size and order being 20 and 35, respectively.

5.1.3 ETH-80

This graph database was created by Morales-González et al. [2014], from the ETH-80 Image Set [Leibe and Schiele, 2003], which contains color images of 80 objects from 8 categories and each object is represented by 41 different views, yielding a total of 3280 images. Some of the images in this database are shown in Fig. 5.4.

Morales-González et al. chose 6 categories to create the graph database: apples, cars, cows, cups, horses and tomatoes. Then, they constructed an irregular graph pyramid for each image, each level representing a Region Adjaceny Graph. The whole pyramid is built from bottom to top, being the base level (level 0) the entire image (i.e. each vertex of the base level represents one pixel in the image, and the edges are the 4-connections of each pixel). Then, each level l is built from its



(A) (B)

(C)

FIGURE 5.3: *Example of a graph generated from an image in the Database* Images, (A) *shows one of the initial images represented in the database,* (B) *shows the quad-tree created from the image and* (C) *shows the graph associated to the image and constructed from the quad-tree.*

FIGURE 5.4: *Some of the images in the* ETH-80 *Image Set [Leibe and Schiele, 2003].*

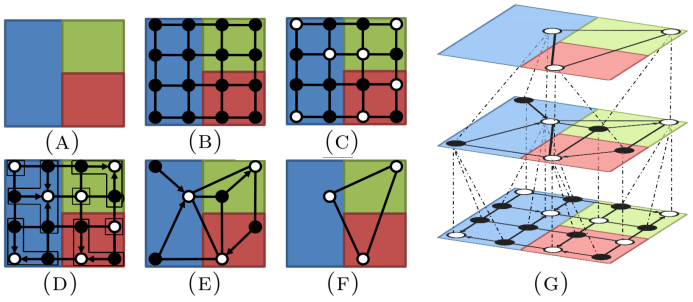


FIGURE 5.5: *Example of an irregular graph pyramid generated from an image in the ETH database [Morales-González and García-Reyes, 2013].* (A) *shows one of the images in the database;* (B)-(F) *show the Region Adjacency Graphs built at each level;* (G) *shows the level hierarchy from bottom* (B) *to top* (F).

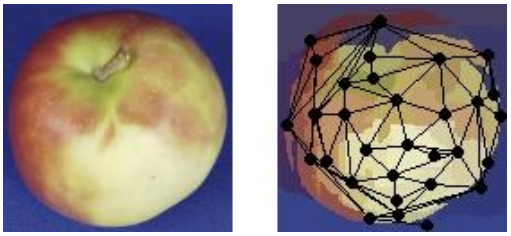


FIGURE 5.6: *Example of an image in the ETH database and its representing graph, which corresponds to the best segmentation level of the pyramid generated from the image.*

previous level $l-1$ by contracting several edges in the process. In a given level, a set of surviving vertices is chosen to be part of the new level, and the set of vertices to be merged into a surviving one is called its Contraction Kernel (CK). In the new level l, each surviving vertex will represent all the vertices from level $l-1$ in its contraction kernel, and will keep a connection to them (Fig. 5.5). Since the pyramid provides several graphs at different levels of resolution for a single image, Morales-González et al. used the B measure proposed in [Morales-González and García-Reyes, 2013] to evaluate the segmentation "quality" of each level and, finally, they represented each image by a single graph, which corresponds to the best segmentation level of each pyramid (Fig. 5.6).

For our experiment, we chose 40 graphs for each category, i.e., we used 240 graphs with an average size and order of 153 and 64, respectively.

5.2 Classification

Our goal is to test the information captured by the patterns mined using our proposed algorithms and, to that end, we propose a classification approach in which the predicted class for a given graph is based on the top k patterns that are similar enough to the graph, ranking the patterns by a discriminative score.

Given training and testing sets, we begin by obtaining the frequent patterns of graphs in the training set, using the frequency and dissimilarity thresholds σ and Δ, respectively. In order to show the usefulness of the mined patterns for supervised classification, we chose a simple and straightforward classifier. First, we eliminate patterns that belong to more than one class. Then, we order the remaining patterns in ascendant order according to their discriminative score, computed as the support of the pattern in the classes different from its own. To classify a graph g in the testing set, we compute the dissimilarity between g and each pattern in the sorted list, using f_{dis}, until we found k patterns similar enough to g (having dissimilarity below the dissimilarity threshold Δ); the majority class among the selected patterns is the class predicted for g.

By following this approach, we can classify graphs using the patterns mined by our algorithms, without going deep into the pattern-based classification problem, which is out of the scope of this thesis.

For all the databases, we used cross validation. For the SIS database, we used four folds, since each class has only four elements, and $\sigma = 2$, $\Delta = 2$ and $k = 1$. The value of σ is the minimum frequency threshold we can use to get patterns that could be different from the input graphs themselves, while the value of Δ allows structural differences in vertices of up to 3 vertices, which is half the average number of vertices in the graphs of this collection; $k = 1$ decides the class of a new graph g using only the best pattern that is similar enough to g. Also, it is important to mention that, after eliminating patterns that belong to more than one class, there were three classes (*L-star, star* and *tortoise*) left without representatives, since they shared the same patterns; thus, in order to carry on with the experiment, we defined a "superclass" containing these three classes. Therefore, we worked with only 7 different classes.

For the Image database, we used ten folds and $\sigma = 6$, $\Delta = 2$ and $k = 3$, whereas for the ETH database we used four folds and $\sigma = 5$, $\Delta = 2$ and $k = 3$. All these

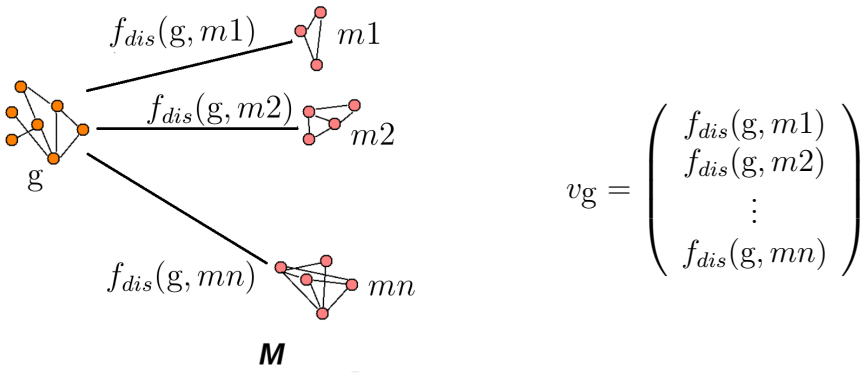


FIGURE 5.7: *Embedding for clustering. Graphs in the dataset are embedded via their dissimilarity (measured by f_{dis}) respect to every pattern found by the algorithm in question.*

values allow patterns with structural differences, while avoiding a too expensive mining stage.

5.3 Clustering

For the graph clustering task, we embedded the graphs in a vector space via their dissimilarity respect to the patterns in the set M, mined by the analyzed algorithm.

Usually, graphs are embedded in a vector space through a set of features, but, according to Pekalska and Duin [2005], in human recognition processes it is more natural to rely on similarities or dissimilarities between objects than to find explicit features of the objects that could be used for recognition (an idea that led to prototype-based embeddings [Foggia et al., 2014]).

We represent each graph g as a vector v_g in an n-dimensional space, where $n = |M|$ and the ith element of v_g is the dissimilarity between g and the ith pattern in M; $i = 1, \ldots, n$ (see Fig. 5.7). We expect that two graphs with similar structure have small (or large) dissimilarities respect to the same patterns, thus, embedding the graphs in this way somehow preserves the structural similarities among the graphs.

To measure the dissimilarity between graphs and patterns, we used f_{dis}. Once graphs have been embedded into vectors, traditional clustering algorithms can be applied. We used the well-known *k-means* algorithm, with random initialization; the value of k was chosen according to the number of classes in the databases.

Since we knew the graph labels, we used the true labels in order to validate our clustering results. In this step we used the General Rand Index [Rand, 1971], which, given two clusterings C and C' over a set S, is defined as

$$R(C, C') = \frac{2(|S_{11}| + |S_{00}|)}{|S|(|S| - 1)},$$

where S_{11} is the set of pairs of graphs that are in the same cluster under C and C', and S_{00} is the set of pairs of graphs that are in different clusters under C and C'.

5.4 Algorithms used for comparison

In both tasks, classification and clustering, we compared our results against those obtained when the patterns mined by gApprox [Chen et al., 2007] were used, since, as mentioned in previous chapters, gApprox is the closest algorithm we could find in the literature. Because we use maximal and closed patterns and gApprox does not focus on them, we mined the patterns using gApprox and, then, we selected the maximal, or closed, ones in a post-mining step.

For clustering, we used f_{dis} in the embedding step for both, our proposed algorithms and gApprox, since what we wanted to find out is if allowing structural differences in vertices leads to discover information, implicit in the mined patterns, that could be useful in clustering.

While comparing against gApprox helps to see the change that allowing structural differences in vertices brings in the overall information obtained from the analyzed data, we also wanted to observe the effectiveness of the algorithm IntAFG to select a subset of interesting patterns. To this end, we implemented the part of the Origami algorithm [Al-Hasan et al., 2007, Section 4.2] oriented to select a set of orthogonal representative patterns using a random clique approach. The algorithm 5.1 shows the pseudocode that we used, based on the description of the authors, but considering that we use a *dissimilarity* function rather than a similarity one (as the authors do).

It is worth noting that this algorithm is *not* Origami, but only the part of it proposed to select representative patterns by using a random clique approach. As explained in Chapter 2, Origami is an algorithm that finds, through a random

Algorithm 5.1: ORIGAMI, Orthogonal Representative Sets

Input: M: set of frequent graphs,
β: orthogonality threshold,
α: dissimilarity threshold,
$nClique$: number of cliques considered.
Output: $\mathcal{R}$: orthogonal representative set.

```
for i = 1 to nClique do
    M_T ← M;
    R_i ← Graph g_0 randomly selected from M;
    Remove g_0 from M;
    for every graph g_M in M do
        Compute dissimilarity between g_M and every graph in R_i;
        if f_dis(g_M, g_R) ≥ α, ∀ g_R ∈ R_i then
            R_i ← R_i ∪ {g_M};
            M ← M\{g_M};
    M_res ← {g_u | g_u ∈ M, f_dis(g_u, g_R) > β ∀ g_R ∈ R_i};
    Residual_i ← |M_res|;
    M ← M_T;
i* ← arg min_{i∈{1,...,nClique}} Residual_i ;
R ← R_{i*};
```

approach, maximal patterns in a collection of graphs and, then, chooses representative patterns. We chose this final part of Origami to compare against our proposed algorithm IntAFG because, as explained in Chapters 2 and 4, it is the closest algorithm in the literature that resembles the ideas in which we are interested when reducing a graph pattern set.

Origami uses the parameters α and β to measure the orthogonality and the similarity, respectively, of the representative patterns that are chosen. In our experiments, we considered β as equivalent to the parameter Δ^* that IntAFG uses. In contrast, the parameter α does not have a counterpart in our proposed algorithm. For each database in our experiment we used three values of α; the first one is the value that coincides with the biggest considered value of β, i.e., $\alpha_1 = 3$ for both databases. When $\alpha = \beta$, Origami's algorithm considers that each pair of graphs is either equivalent (one graph represents the other) or orthogonal; this idea would be close to our algorithm, in which there is not a threshold to satisfy for two graphs to be orthogonal. The next value of α that we used is the equivalent of the value that Hasan et al. report to use in some of their experiments, in which $\alpha = 0.2$, meaning that two orthogonal graphs have, at most, 20% of similarity (respect to their size); since we use a dissimilarity function, we used values representing 80%

of the size of the graphs considered, this is, $\alpha_3 = 5$ for SIS and $\alpha_3 = 16$ for Images. Finally, the last value of α is just the middle point between the two other α values, $\alpha_2 = 4$ for SIS and $\alpha_2 = 9$ for Images.

For the number of cliques in Algorithm 5.1, we used $nClique = 50$, due to time constraints. For every clique, it is necessary to compute the dissimilarity, using f_{dis}, of every pair of patterns in the considered M and $\mathcal{R}$ sets.

5.5 Results

Tables 5.1 and 5.2 show the results obtained in our experiments, while Table 5.3 shows the time required (in seconds) in each case. In the former tables, the best result for each algorithm, gApprox and AGraP, and for each database is highlighted. For clustering, we show the best result that we obtained after running the experiment 100 times, measuring the clustering quality with the general Rand Index.

First of all, we see an improvement in the results when using patterns mined by AGraP compared to the results when using gApprox's patterns. Table 5.4 shows the p-values obtained by performing a Student's t-test on the results obtained using all gApprox's and AGraP's patterns, since we wanted to explore the advantage (if any) of the additional patterns found by AGraP. Using a p-value of 0.05 as the threshold to accept or reject the null hypothesis, we have that, according to the values shown in the table, the results for the SIS database could be considered statistically significant; for the IMAGES database, only in the case of classification we can see a significant improvement. Finally, we do not get significantly better results for the ETH database.

However, although we do not get significantly better results in all the cases, we want to remark that the essential difference between gApprox and the algorithms we propose (AGraP, CloseAFG, MaxAFG) is that, while all of them allow label differences and structural differences in edges, gApprox does not allow structural differences in vertices; as argued in Chapter 3, all of the patterns found by gApprox, are a subset of the patterns found by AGraP. Therefore, the quality improvement that we see in AGraP's results could be directly attributed to those patterns identified when allowing structural differences in vertices.

TABLE 5.1: *Results of classification using patterns mined by gApprox, AGraP and our proposed interesting set of AGraP patterns, respectively. The column* Pat *shows the average amount of patterns used in every fold, while the column* Acc *shows the mean accuracy achieved. The values* $(\alpha_1, \alpha_2, \alpha_3)$ *used in Origami are (3,4,5) for SIS, (3,9,16) for Images and (3,27,51) for ETH.*

		SIS		**Images**		**ETH**	
		Acc	Pat	Acc	Pat	Acc	Pat
gApprox	All	66.6	18	73.0	296	**62.5**	**1222**
	Closed	66.6	13	73.0	197	61.8	711
	Maximal	66.6	10	68.5	139	59.6	501
	Interesting, $\Delta^* = 1$	66.6	9	**80.0**	**26**	56.3	212
	$\Delta^* = 2$	63.9	8	**80.0**	**25**	57.1	208
	$\Delta^* = 3$	63.9	5	**80.0**	**19**	57.1	178
	Origami, α_1, $\beta = 1$	58.3	9	73.5	88	55.8	425
	$\beta = 2$	72.2	9	73.0	87	57.1	411
	$\beta = 3$	72.2	10	75.0	84	58.8	417
	α_2, $\beta = 1$	**83.3**	**11**	64.5	98	53.3	135
	$\beta = 2$	69.4	9	68.5	93	58.3	128
	$\beta = 3$	72.2	10	65.5	91	59.6	124
	α_3, $\beta = 1$	63.9	9	69.0	51	55.0	139
	$\beta = 2$	69.4	8	71.0	50	53.3	125
	$\beta = 3$	61.1	6	72.5	52	61.3	119
AGraP	All	**100**	**385**	**84.5**	**3445**	**63.8**	**3174**
	CloseAFG	88.9	240	83.0	2288	62.5	2365
	MaxAFG	88.9	201	82.5	1469	62.5	1711
	Interesting, $\Delta^* = 1$	**100**	**13**	76.1	42	58.3	256
	$\Delta^* = 2$	77.8	12	76.1	42	58.3	251
	$\Delta^* = 3$	75.0	11	76.1	38	58.3	232
	Origami, α_1, $\beta = 1$	94.4	110	75.5	469	58.8	679
	$\beta = 2$	83.3	115	74.0	468	62.1	660
	$\beta = 3$	94.4	112	78.0	476	60.4	650
	α_2, $\beta = 1$	91.7	110	77.0	598	54.2	209
	$\beta = 2$	91.7	114	78.0	601	57.5	191
	$\beta = 3$	91.7	119	75.0	601	55.8	201
	α_3, $\beta = 1$	91.7	126	81.0	639	58.8	155
	$\beta = 2$	88.9	136	80.0	623	59.2	153
	$\beta = 3$	91.7	123	79.5	632	52.9	148

These results suggest that the patterns found by AGraP are indeed able to capture useful information about the input graph, missed when no stuctural differences in vertices are allowed, as in gApprox.

Using CloseAFG and MaxAFG brought a decrement in the quality of the results, but we still observe an improvement with respect to the results obtained by using gApprox.

With respect to the use of IntAFG, in every case we can see how choosing the *interesting* set of patterns drastically reduces the pattern set size; this is even

TABLE 5.2: *Results of clustering using patterns mined by gApprox, AGraP and our proposed interesting set of AGraP patterns, respectively. The column* Pat *shows the number of patterns used for embedding. The clustering quality is measured using the General Rand Index (G.R.I.). The values* $(\alpha_1, \alpha_2, \alpha_3)$ *used in Origami are (3,4,5) for SIS, (3,9,16) for Images and (3,27,51) for ETH.*

		SIS		**Images**		**ETH**	
		G.R.I.	Pat	G.R.I.	Pat	G.R.I.	Pat
gApprox	All	.802	25	.730	347	.778	1677
	Closed	.799	19	.737	225	.770	977
	Maximal	.815	14	.741	160	.769	677
	Interesting, $\Delta^* = 1$	**.827**	**12**	.737	36	**.783**	**306**
	$\Delta^* = 2$	.801	11	.716	36	.778	296
	$\Delta^* = 3$	.794	8	.710	26	.776	255
	Origami, α_1, $\beta = 1$	.813	15	.741	116	.777	577
	$\beta = 2$	.794	13	.737	92	.770	591
	$\beta = 3$	.809	15	**.748**	**106**	.779	530
	α_2, $\beta = 1$	.816	15	.737	123	.776	205
	$\beta = 2$	.806	15	.745	106	.770	175
	$\beta = 3$	.823	13	.737	106	**.783**	**175**
	α_3, $\beta = 1$	.818	14	.730	70	.782	179
	$\beta = 2$	.795	11	.744	61	.781	175
	$\beta = 3$	.788	13	.744	58	.781	176
AGraP	All	**.842**	**509**	**.758**	**4012**	.778	4301
	CloseAFG	.832	317	.751	2539	.774	3214
	MaxAFG	.829	266	.730	1655	.776	2311
	Interesting, $\Delta^* = 1$	.830	20	**.758**	**56**	.779	363
	$\Delta^* = 2$	.830	18	.730	56	.778	353
	$\Delta^* = 3$	.817	16	.730	51	.783	327
	Origami, α_1, $\beta = 1$	.825	156	.730	548	.774	950
	$\beta = 2$	.825	134	.751	590	.778	902
	$\beta = 3$	.821	142	.751	511	.776	879
	α_2, $\beta = 1$	.829	153	.730	669	.774	272
	$\beta = 2$	.830	151	.751	678	**.785**	**254**
	$\beta = 3$	.825	149	.751	682	.780	238
	α_3, $\beta = 1$	.837	167	.751	683	.775	207
	$\beta = 2$	.829	180	.737	678	.782	207
	$\beta = 3$	.821	174	**.758**	**713**	.780	199

more notorious in AGraP, where the interesting sets of patterns represent only 4% *or less* of the whole set. As we expected from the previous chapter, interesting sets of patterns are a more effective approach than focusing on closed or maximal patterns in terms of reducing the size of the output set. Moreover, the number of patterns obtained through IntAFG represents only around 10% of the number of patterns obtained when using Origami. In fact, the size of the interesting pattern set is comparable to the size of gApprox's patterns in the case of SIS and ETH, and smaller in the case of Images.

Regarding the quality, the best results were achieved when using the smallest Δ^*

TABLE 5.3: *Time (seconds) required to mine, classify and cluster graphs used in the experiments shown in this Chapter. For each database, the first column (*Min*) shows the time required to mine the patterns used later on the classificatoin and clustering tasks. The column* Cla *shows the time required to classify the graphs, whereas the column* Clu *shows the time required to cluster the graphs (including the embedding process). The values* $(\alpha_1, \alpha_2, \alpha_3)$ *used in Origami are (3,4,5) for SIS, (3,9,16) for Images and (3,27,51) for ETH.*

		SIS			**Images**			**ETH**		
		Min	Cla	Clu	Min	Cla	Clu	Min	Cla	Clu
gApprox	All	0.02	0.10	0.23	0.22	26	72	1.7	302	521
	Closed	0.06	0.07	0.18	2.3	80	48	17	167	297
	Maximal	0.11	0.05	0.12	2.9	125	37	22	116	206
	Interesting, $\Delta^* = 1$	0.02	0.04	0.10	2.1	1.7	21	20	43	94
	$\Delta^* = 2$	0.03	0.05	0.08	2.1	1.7	17	20	42	84
	$\Delta^* = 3$	0.01	0.03	0.06	2.1	1.3	16	20	36	70
	Origami, α_1, $\beta = 1$	0.03	0.05	0.12	17	2.2	37	7.1	88	169
	$\beta = 2$	0.01	0.05	0.10	13	2.1	39	7.1	85	172
	$\beta = 3$	0.01	0.04	0.12	16	2.2	40	6.1	87	155
	α_2, $\beta = 1$	0.04	0.05	0.11	16	2.4	37	10	27	57
	$\beta = 2$	0.01	0.05	0.10	15	2.2	37	10	26	48
	$\beta = 3$	0.01	0.05	0.10	14	2.3	37	9.6	24	49
	α_3, $\beta = 1$	0.05	0.05	0.12	14	1.4	20	9.8	27	49
	$\beta = 2$	0.01	0.04	0.08	17	1.3	17	9.5	25	49
	$\beta = 3$	0.02	0.05	0.05	14	1.2	28	8.9	23	48
AGraP	All	0.32	7.49	23.9	1686	3016	3680	496	1123	1436
	CloseAFG	0.25	4.09	10.1	1694	1115	2831	499	726	1065
	MaxAFG	0.25	3.46	7.92	1704	686	2164	498	504	789
	Interesting, $\Delta^* = 1$	2.11	0.08	0.23	2854	22	76	408	52	105
	$\Delta^* = 2$	2.13	0.07	0.19	2819	22	75	408	51	102
	$\Delta^* = 3$	2.12	0.06	0.19	2791	20	71	408	48	94
	Origami, α_1, $\beta = 1$	0.42	1.10	2.92	1118	14	222	52	159	292
	$\beta = 2$	0.45	0.97	3.34	1291	16	293	44	150	273
	$\beta = 3$	0.39	1.11	2.83	1306	16	263	54	149	265
	α_2, $\beta = 1$	0.42	1.09	3.01	1248	16	342	65	53	91
	$\beta = 2$	0.40	1.02	3.60	1298	17	316	43	46	83
	$\beta = 3$	0.46	0.91	2.99	1183	17	299	66	52	78
	α_3, $\beta = 1$	0.37	1.25	3.21	1185	17	555	43	31	59
	$\beta = 2$	0.52	1.19	3.65	1296	17	566	42	32	59
	$\beta = 3$	0.41	1.33	3.97	1166	17	487	43	30	57

TABLE 5.4: *p-values from performing a Student's t-test on the results obtained by using all gApprox's and AGraP's patterns.*

	SIS	IMAGES	ETH
Classification	.00	.05	.61
Clustering	.01	.18	.42

value, which, intuitively, corresponds to use a tighter cover on the original set. Nevertheless, with some exceptions (classification in the SIS database, clustering in the ETH database, clustering using gApprox patterns), we see a decrement in the quality of the classification and clustering results when using the set of interesting patterns.

The results corresponding to the use of Origami seem, in general, better than

the results obtained through the set of interesting patterns. However, considering how much smaller the set of interesting patterns is, with respect to the whole set, we could say that the obtained results are satisfactory and represent an acceptable trade-off between the size of the pattern set and the quality achieved in the classification and clustering tasks.

Finally, a clear drawback of the proposed algorithms, including IntAFG, is the time they require, as they all are computationally expensive.

5.6 Discussion

In this chapter, we explored the usefulness, in the context of graph classification and clustering, of patterns found by the proposed algorithms AGraP, CloseAFG and MaxAFG, that mine patterns using inexact matching and allow structural differences in vertices and edges. We were also able to observe the results obtained when using a subset of interesting patterns chosen by the algorithm IntAFG.

We found that using the patterns mined by AGraP resulted in a better quality in classification and clustering tasks. Our experiments do show that allowing structural differences in vertices can lead to useful knowledge, providing a more insightful analysis of graph data.

On the other hand, the subsets of interesting patterns obtained through the algorithm IntAFG were comparable in size to the sets of patterns mined by a state-of-the-art algorithm that does not allow structural differences in vertices. The reduction, respect to the pattern set mined by AGraP, was more than 95% and yet, depending on the value of Δ^* used, the interesting-pattern set achieved better results than patterns obtained without structural differences in vertices. The results with the interesting subset did show a decrement in quality compared to the original set of patterns mined by AGraP but, taking into account the size reduction achieved, we consider that the results are acceptable.

Chapter 6

Conclusions

Mining graph patterns is a problem with increasing importance, given that data is becoming more complex and graphs provide a modeling tool able to capture its more sophisticated structured nature. In recent years, it has been addressed the case in which patterns and their occurrences are not necessarily identical, acknowledging that the analyzed data could be noisy or that, in certain contexts, the use of inexact matching reflects better the nature of the problem.

In this thesis, we propose the algorithm AGraP that finds frequent patterns in a single graph allowing structural differences in vertices and edges, and that identifies patterns missed by other state-of-the-art algorithms. In order to define our algorithm, we proposed a comparison function for graphs that is based on the edit distance. We also proposed strategies to identify occurrences with structural differences in vertices, so that occurrences can have more or less vertices than the pattern they represent.

Furthermore, we propose the algorithms CloseAFG and MaxAFG, for mining closed and maximal patterns, respectively. Additionally, we propose a post-processing algorithm, IntAFG, aiming to reduce the amount of patterns in the output set. CloseAFG and MaxAFG follow the core ideas of AGraP, using inexact matching during the mining process, but they reduce the size of the output set by focusing on closed and maximal patterns, respectively. On the other hand, the algorithm IntAFG obtains an *interesting* subset of patterns in a post-mining step. Although "interesting" is an inherently subjective idea, we propose to look for a set of patterns that covers the original pattern set, in the sense that every pattern in the original set has a dissimilarity below a threshold Δ^* towards a pattern in

the interesting set, while, at the same time, interesting patterns are iteratively chosen aiming at having large dissimilarity among them.

Given that, by using inexact matching, our proposed algorithms are able to find "extra" patterns missed when using exact matching in the mining process, the important question about their usefulness arises. Therefore, we perform tasks of classification and clustering on three image graph databases and we show that those extra patterns are indeed able to capture information on the analyzed data that can be useful and provide a new insight in the data.

In addition, we show that the proposed algorithm IntAFG is able to effectively reduce the amount of patterns, finding a set comparable in size to the sets mined by other algorithms. Although using the interesting pattern set leads to a reduction of the quality in the results of classification and clustering tasks, regarding AGraP's results, the results still showed improvements, with respect to using patterns obtained without structural differences in vertices, that could be attributed to the use of inexact matching. Therefore, it can be considered that the interesting set obtained achieves an acceptable trade-off between size and quality of the mined information.

6.1 Contributions

The main contributions of the work are:

1. AGraP, an algorithm for frequent graph pattern mining that allows structural differences in vertices as well as in edges, which, to the best of our knowledge is an idea that has not been explored before.

2. Two dissimilarity measures based on the graph edit distance that are compatible with the idea of allowing structural differences in edges and vertices.

3. The algorithms CloseAFG and MaxAFG, which focus on closed and maximal patterns, respectively.

4. The algorithm IntAFG that obtains an interesting subset of patterns by combining the ideas of covering and maximizing distance among patterns.

Some of these contributions have been reported in the following publications derived from this thesis:

- Flores-Garrido M. *et al.*, AGraP: an algorithm for mining frequent patterns in a single graph using inexact matching. *Knowledge and Information Systems*, 2014. Advance online publication. DOI: 10.1007/s10115-014-0747-x.
- Flores-Garrido M. *et al.*, Mining maximal frequent patterns in a single graph using inexact matching. *Knowledge-Based Systems* 66:166–177, 2014, ISSN 0950-7051.
- Flores-Garrido M. *et al.*, Graph clustering via inexact patterns. In Proceedings of the 19th Iberoamerican Congress on Pattern Recognition (*CIARP*), Lecture Notes in Computer Science, Volume 8827, pp. 391-398, 2014.

6.2 Future work

The main drawback of our proposed algorithm is its computational cost. Scalability is, in general, one of the main challenges currently faced in the graph based data mining area; in our algorithms this issue is amplified by the use of inexact matching during the mining process. Although we are able to reduce the amount of patterns in the output set, either by focusing on closed and maximal patterns or by choosing a subset of interesting patterns, in all the cases the reduction of graphs is done during the mining process or in post-mining steps but the work done by the core algorithm remains expensive. Therefore, the future work consists in, above all, finding a way to improve the efficiency of the proposed algorithms. By improving the efficiency, larger graphs could be analyzed and that would benefit the extraction of information on different practical applications.

Further work is needed to analyze the possibility of merging the selection of interesting patterns in the mining process, so that the overall process can be more efficient by pruning some patterns that might not be considered interesting because of its redundancy and, in this way, reducing the time required by the algorithm. Also, further study can be done in defining the characteristics that a pattern set must have in order to be considered interesting, without focusing on an specific context.

Another line of future research could be to focus on using the function f_{rel} to mine frequent patterns. Taking into consideration the size of the patterns, in order to establish a number of admissible differences with respect to their occurrences, is an idea that matches our intuition, but, since it leads to the lose of the anti-monotonicity property, it requires a different approach to deal with the lack of traditional pruning during the mining process.

Finally, we explored only one of the possible strategies to find inexact patterns allowing structural differences and, implicitly, by the strategy, dissimilarity function and support definition, we defined the kind of patterns we were able to find. However, different patterns could be found by varying different parts in the algorithm. Given that applications often require inexact matching and that our work suggests that patterns found through the use of inexact matching are able to discover new information, we believe that the subject of mining inexact patterns - allowing structural differences - is worth further exploration.

Bibliography

Acosta-Mendoza, N., Gago-Alonso, A., and Medina-Pagola, J. E. (2012). Frequent approximate subgraphs as features for graph-based image classification. *Knowledge-Based Systems*, 27(0):381–392.

Al-Hasan, M., Chaoji, V., Salem, S., Besson, J., and Zaki, M. J. (2007). Origami: Mining representative orthogonal graph patterns. In *ICDM*, pages 153–162. IEEE Computer Society.

Alguliev, R., Aliguliyev, R., and Ganjaliyev, F. (2011). Extracting a Heterogeneous Social Network of Academic Researchers on the Web Based on Information Retrieved from Multiple Sources. *American Journal of Operations Research*, 1(2):33–38.

Barbu, E., Héroux, P., Adam, S., and Trupin, E. (2005). Frequent graph discovery: Application to line drawing document images. *Electronic Letters on Computer Vision and Image Analysis*, 5(2):47–57.

Borgelt, C. and Berthold, M. (2002). Mining molecular fragments: Finding relevant substructures of molecules. In *Proceedings. 2002 IEEE International Conference on Data Mining*, pages 51–58.

Bringmann, B. and Nijssen, S. (2008). What is frequent in a single graph? In Washio, T., Suzuki, E., Ting, K., and Inokuchi, A., editors, *Advances in Knowledge Discovery and Data Mining*, volume 5012 of *Lecture Notes in Computer Science*, pages 858–863. Springer Berlin / Heidelberg.

Bunke, H. (1997). On a relation between graph edit distance and maximum common subgraph. *Pattern Recogn. Lett.*, 18(9):689–694.

Bunke, H. and Riesen, K. (2011). Recent advances in graph-based pattern recognition with applications in document analysis. *Pattern Recogn.*, 44:1057–1067.

Bunke, H. and Shearer, K. (1998). A graph distance metric based on the maximal common subgraph. *Pattern Recogn. Lett.*, 19(3-4):255–259.

Chen, C., Yan, X., Zhu, F., and Han, J. (2007). gApprox: Mining frequent approximate patterns from a massive network. In *ICDM*, pages 445–450. IEEE Computer Society.

Chen, X., Zhang, C., Liu, F., and Guo, J. (2012). Algorithm research of top-down mining maximal frequent subgraph based on tree structure. In Snac, P., Ott, M., and Seneviratne, A., editors, *Wireless Communications and Applications*, volume 72 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 401–411. Springer Berlin Heidelberg.

Cheng, H., Yan, X., and Han, J. (2010). Mining graph patterns. In Aggarwal, C. and Wang, H., editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 365–392. Springer.

Coenen, F. (2006). The lucs-kdd random image generator: The ranimggen directory. Website, http://cgi.csc.liv.ac.uk/~frans/KDD/Software/ImageGenerator/imageGenerator.html. Last checked on November 2014.

Cook, D. J. and Holder, L. (1994). Substructure discovery using minimum description length and background knowledge. *Journal of Artificial Intelligence Research*, pages 231–255. No tengo el artculo.

Dehmer, M. and Emmert-Streib, F. (2007). Comparing large graphs efficiently by margins of feature vectors. *Applied Mathematics and Computation*, 188(2):1699–1710.

Fiedler, M. and Borgelt, C. (2007). Support computation for mining frequent subgraphs in a single graph. In *5th Int. Workshop on Mining and Learning with Graphs*, pages 25–30.

Foggia, P., Percannella, G., and Vento, M. (2014). Graph matching and learning in pattern recognition in the last 10 years. *International Journal of Pattern Recognition and Artificial Intelligence*, 28(01):1450001.

Gago-Alonso, A., Medina-Pagola, J., Carrasco-Ochoa, J., and Martínez-Trinidad, J. (2008). Mining frequent connected subgraphs reducing the number of candidates. In Daelemans, W., Goethals, B., and Morik, K., editors, *Machine*

Learning and Knowledge Discovery in Databases, volume 5211 of *Lecture Notes in Computer Science*, pages 365–376. Springer Berlin / Heidelberg.

Gao, X., Xiao, B., Tao, D., and Li, X. (2010). A survey of graph edit distance. *Pattern Anal. Appl.*, 13(1):113–129.

Gartner, T. (2002). Exponential an geometric kernels for graphs. In *NIPS*02 workshop on unreal data*, volume Principles of modeling nonvectorial data.

Gärtner, T. (2008). *Kernels For Structured Data*. Series in machine perception and artificial intelligence. World Scientific.

Gartner, T., Flach, P., and Wrobel, S. (2003). On graph kernels: Hardness results and efficient alternatives. In *Conference on Learning Theory*, pages 129–143.

Geng, L. and Hamilton, H. J. (2006). Interestingness measures for data mining: A survey. *ACM Comput. Surv.*, 38(3).

Hagberg, A. A., Schult, D. A., and Swart, P. J. (2008). Exploring network structure, dynamics, and function using NetworkX. In Varoquaux, G., Vaught, T., and Millman, J., editors, *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA.

Hellal, A. and Romdhane, L. B. (2013). Nodar: mining globally distributed substructures from a single labeled graph. *Journal of Intelligent Information Systems*, 40(1):1–15.

Hidovic, D. and Pelillo, M. (2004). Metrics for attributed graphs based on the maximal similarity common subgraph. *IJPRAI*, pages 299–313.

Holder, L., Cook, D., and Djoko, S. (1994). Substructure discovery in the subdue system. In *In Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, pages 169–180.

Holder, L. B. (1988). Substructure discovery in subdue. Technical Report UILU-ENG-88-2220, Department of Computer Science, University of Illinois, Urbana.

Huan, J., Wang, W., Prins, J., and Yang, J. (2004). Spin: mining maximal frequent subgraphs from graph databases. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 581–586. ACM.

Jia, Y., Zhang, J., and Huan, J. (2011). An efficient graph-mining method for complicated and noisy data with real-world applications. *Knowl. Inf. Syst.*, 28(2):423–447.

Jiang, C., Coenen, F., and Zito, M. (2013). A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 28:75–105.

Kondor, R. and Lafferty, J. (2002). Diffusion kernels on graphs and other discrete input spaces. *International Conference on Machine Learning (ICML)*.

Koyutürk, M., Grama, A., and Szpankowski, W. (2004). An efficient algorithm for detecting frequent subgraphs in biological networks. *Bioinformatics*, 20(suppl 1):i200–i207.

Krishna, V., Ranga Suri, N., and Athithan, G. (2011). A comparative survey of algorithms for frequent subgraph discovery. *Curren Science*, 100(2):190–198.

Kuramochi, M. and Karypis, G. (2001). Frequent subgraph discovery. In *Proceedings of the Int. Conf. Data Mining (ICDM01)*, pages 313–320.

Kuramochi, M. and Karypis, G. (2004a). Finding frequent patterns in a large sparse graph. In *Proceedings of the 2004 SIAM International Conference on Data Mining (SDM04)*.

Kuramochi, M. and Karypis, G. (2004b). Grew - a scalable frequent subgraph discovery algorithm. In *Proceedings of the Fourth IEEE International Conference on Data Mining*, pages 439–442.

Kuramochi, M. and Karypis, G. (2005). Finding frequent patterns in a large sparse graph. *Data Mininig Knowledge Discovery*, 11(3):243–271.

Leibe, B. and Schiele, B. (2003). Analyzing appearance and contour based methods for object categorization. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR03)*, pages 409–415.

Leskovec, J. (2009). Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/.

Lima, D. J., Giacomini, H., R.M., T., A.A., A., and L.M., B. (2012). Patterns of interactions of a large fish-parasite network in a tropical floodplain. *Anim Ecol.*, 81(4):905–913.

López-Presa, J. L. (2009). *Efficient Algorithms for Graph Isomorphism Testing*. PhD thesis, Universidad Rey Juan Carlos, Madrid, España.

Mahé, P., Ueda, N., Akutsu, T., and Perret, J. (2005). Graph kernels for molecular structure-activity relationship analysis with support vector machines. *Journal of Chemical Information and Modeling*, 45(4):939–951.

Moody, J. (2001). Peer influence groups: identifying dense clusters in large networks. *SOCIAL NETWORKS*, 23:261–283.

Morales-González, A., Acosta-Mendoza, N., Gago-Alonso, A., Garca-Reyes, E., and Medina-Pagola, J. (2014). A new proposal for graph-based image classification using frequent approximate subgraphs. *Pattern Recognition*, 47(1):169–177.

Morales-González, A. and García-Reyes, E. B. (2013). Simple object recognition based on spatial relations and visual features represented using irregular pyramids. *Multimedia tools and applications*, 63(3):875–897.

Neuhaus, M. and Bunke, H. (2004). A probabilistic approach to learning costs for graph edit distance. In *ICPR (3)*, pages 389–393.

Neuhaus, M. and Bunke, H. (2005). Self-organizing maps for learning the edit costs in graph matching. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 35(3):503–514.

Neuhaus, M. and Bunke, H. (2006). A convolution edit kernel for error-tolerant graph matching. In *ICPR (4)*, pages 220–223. IEEE Computer Society.

Neuhaus, M. and Bunke, H. (2007). *Bridging the Gap Between Graph Edit Distance and Kernel Machines*. World Scientific.

Nijssen, S. and Kok, J. N. (2004). A quickstart in frequent structure mining can make a difference. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '04, pages 647–652. ACM.

Ozaki, T. and Etoh, M. (2011). Closed and maximal subgraph mining in internally and externally weighted graph databases. In *Proceedings of the 2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications*, WAINA '11, pages 626–631. IEEE Computer Society.

Papapetrou, O., Ioannou, E., and Skoutas, D. (2011). Efficient discovery of frequent subgraph patterns in uncertain graph databases. In *Proceedings of the 14th International Conference on Extending Database Technology*, EDBT/ICDT '11, pages 355–366. ACM.

Pekalska, E. and Duin, R. (2005). *The Dissimilarity Representation for Pattern Recognition: Foundations and Applications*. Series in machine perception and artificial intelligence. World Scientific.

Pinilla-Buitrago, L., Martínez-Trinidad, J., and Carrasco-Ochoa, J. (2013). New penalty scheme for optimal subsequence bijection. In *Proceedings of the 18th Iberoamerican Congress on Pattern Recognition (CIARP)*, volume 8258 of *Lecture Notes in Computer Science*, pages 206–213. Springer Berlin/Heidelberg.

PubChem-Database (2004). National Center for Biotechnology Information.

Ramon, J. and Gartner, T. (2003). Expressivity versus efficiency of graph kernels. In *Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences*, pages 65–74.

Rand, W. M. (1971). Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):pp. 846–850.

Ranu, S. and Singh, A. (2009). Graphsig: A scalable approach to mining significant subgraphs in large graph databases. In *IEEE 25th International Conference on Data Engineering*, pages 844–855.

Riesen, K. and Bunke, H. (2008). IAM graph database repository for graph based pattern recognition and machine learning. In *Proceedings of the International Workshop on Structural Syntatctic and Statistical Pattern Recognition*, Lecture Notes in Computer Science, pages 287–297.

Riesen, K., Jiang, X., and Bunke, H. (2010). Exact and inexact graph matching: Methodology and applications. In Aggarwal, C. C. and Wang, H., editors, *Managing and Mining Graph Data*, volume 40 of *Advances in Database Systems*, pages 217–247. Springer US.

Robles-Kelly, A. and Hancock, E. R. (2003). Graph matching using spectral seriation and string edit distance. In *Proceedings of the 4th IAPR international*

conference on Graph based representations in pattern recognition, GbRPR'03, pages 154–165. Springer-Verlag.

Robles-Kelly, A. and Hancock, E. R. (2004). String edit distance, random walks and graph matching. *Int. Journal of Pattern Recognition and Artificial Intelligence,*, 18(3):315–327.

Robson, J. M. (1986). Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3):425–440.

Saeedy, M. E. and Kalnis, P. (2011). GraMi: generalized frequent pattern mining in a single large graph. Technical report, Division of Mathematical and Computer Sciences and Engineering, King Abdullah University of Science and Technology.

Sanfeliu, A. and Fu, K. S. (1983). A distance measure between attributed relational graphs for pattern recognition. *IEEE Transactions on Systems, Man and Cybernetics*, 13(3):353–363.

Schenker, A., Last, M., Bunke, H., and Kandel, A. (2004). Classification of web documents using graph matching. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(3):475–496.

Schuhmacher, M. and Ponzetto, S. P. (2014). Knowledge-based graph document modeling. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, WSDM '14, pages 543–552. ACM.

Shervashidze, N., Vishwanathan, S. V. N., Petri, T., Mehlhorn, K., and Borgwardt, K. (2009). Efficient graphlet kernels for large graph comparison. In *Proceedings of International Conference on Artificial Intelligence and Statistics*.

Smola, A. J. and Kondor, I. R. (2003). Kernels and regularization on graphs. In Schölkopf, B. and Warmuth, M. K., editors, *Proc. Annuall Conf. Computational Learning Theory*, pages 144–158.

Spyropoulou, E., De Bie, T., and Boley, M. (2014). Interesting pattern mining in multi-relational data. *Data Mining and Knowledge Discovery*, 28(3):808–849.

Thomas, L. T., Valluri, S. R., and Karlapalem, K. (2010). Margin: Maximal frequent subgraph mining. *ACM Trans. Knowl. Discov. Data*, 4(3):10:1–10:42.

Vanetik, N., Gudes, E., and Shimony, S. E. (2002). Computing frequent graph patterns from semistructured data. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 458–, Washington, DC, USA. IEEE Computer Society.

Vijayalakshmi, R., Nadarajan, R., Roddick, J. F., Thilaga, M., and Nirmala, P. (2011). FP-GraphMiner: A fast frequent pattern mining algorithm for network graphs. *Journal of Graph Algorithms and Applications*, 15(6):753–776.

Vishwanathan, S. V. N., Borgwardt, K., Kondor, I., and Schraudolph, N. (2008). Graph kernels. *Journal of Machinel Learning Research*, 9:1–41.

Vreeken, J. and Tatti, N. (2014). Interesting patterns. In Aggarwal, C. C. and Han, J., editors, *Frequent Pattern Mining*, pages 105–134. Springer International Publishing.

Xiao, Y., Dong, H., Wu, W., Xiong, M., Wang, W., and Shi, B. (2008). Structure-based graph distance measures of high degree of precision. *Pattern Recognition*, 41(12):3547–3561.

Yan, X., Cheng, H., Han, J., and Yu, P. (2008). Mining significant graph patterns by leap search. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 433–444. ACM.

Yan, X. and Han, J. (2002). gSpan: Graph-based substructure pattern mining. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, ICDM '02, pages 721–724. IEEE Computer Society.

Yan, X. and Han, J. (2003). CloseGraph: mining closed frequent graph patterns. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, pages 286–295. ACM.

Yan, X., Yu, P. S., and Han, J. (2004). Graph indexing: A frequent structure-based approach. In *SIGMOD Conference*, pages 335–346. ACM.

Yu, H. and Hancock, E. R. (2006). String kernels for matching seriated graphs. *Pattern Recognition, International Conference on*, 4:224–228.

Zhang, S., Yang, J., and Cheedella, V. (2007). Monkey: Approximate graph mining based on spanning trees. In *Data Engineering, 2007. ICDE 2007. IEEE 23rd International Conference on*, pages 1247 –1249.

Zou, Z., Li, J., Gao, H., and Zhang, S. (2009). Frequent subgraph pattern mining on uncertain graph data. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 583–592. ACM.