Les Cahiers du GERAD ISSN: 0711–2440

Using Heuristics to Speed Up Frequent Pattern Mining

C. Desrosiers, Ph. Galinier,
P. Hansen, A. Hertz

G–2008–13

February 2008

Using Heuristics to Speed Up Frequent Pattern Mining

Christian Desrosiers*
Philippe Galinier
Alain Hertz*

École Polytechnique de Montréal
C.P. 6079, Succ. Centre-ville
Montréal (Québec) Canada, H3C 3A7
{christian.desrosiers,philippe.galinier,alain.hertz}@polymtl.ca

** and GERAD*

Pierre Hansen

GERAD and HEC Montréal
3000, chemin de la Côte-Sainte-Catherine
Montréal (Québec) Canada, H3T 2A7
pierre.hansen@gerad.ca

February 2008

Les Cahiers du GERAD
G–2008–13

Abstract

In this paper we present a simple technique that uses background information to improve mining the frequent patterns of structured data. This technique uses a heuristic function that remaps the search space in a way the greatly reduces the number of costly subgraph isomorphism tests, without using space-expensive data structures. We illustrate our approach on a popular structured data mining problem, called the frequent subgraph mining problem, and show, through experiments on synthetic and real-life data, that this simple approach has advantages over other frequent pattern mining algorithms.

Key Words: Structured data mining, frequent pattern, graph.

Résumé

Nous présentons, dans cet article, une technique simple qui utilise de l'information de fond pour améliorer l'exploration des patrons fréquents dans des données structurées. Cette technique emploie une fonction heuristique qui définie l'espace de recherche de manière à réduire le nombre de tests coûteux d'isomorphisme de sous-graphe, sans avoir recours à des structures de données complexes. Nous illustrons notre approche sur un problème populaire de l'exploration des données structurées, connu sous le nom d'exploration des sous-graphes fréquents, et montrons, à l'aide de tests numériques sur des données générées et provenant d'applications réelles, les avantages de notre approche simple sur des méthodes plus complexes pour le même problème.

1 Introduction

Recently, applications dealing with structured information have appeared in various fields. This enhanced information has a richer content that enables a more precise representation of the environment to model. Structured data mining is a discipline that plays a key role in important fields such as bioinformatics, in particular drug design, [12, 1, 11] and Web technologies [8, 2], and which aims at extracting useful knowledge from a great amount of structured information. At the center of this discipline lies the problem of finding the frequent patterns of a database, which can be defined as follows. Let $\mathcal{D}$ be a database and denote $sup(X)$ the support of the pattern X in $\mathcal{D}$, i.e. the number of patterns of $\mathcal{D}$ that have X as a subpattern. Given an integer s_{min}, called the minimum support threshold, we say that X is frequent if $sup(X) \geq s_{min}$. The frequent pattern mining problem, for a given database $\mathcal{D}$, consists in finding the patterns that are frequent in $\mathcal{D}$. A well-known specialization of this problem is the frequent subgraph mining problem, where the database contains graphs and the goal is to find the graphs the are isomorphic to a subgraph of at least s_{min} graphs of the database.

The frequent pattern mining problem can be decomposed in two tasks: 1) uniquely enumerate all possible patterns and 2) calculate the support of these patterns in the database. In most cases, computing the support of a pattern in the database is a complex operation, and the second task accounts for most of the time required to find the frequent patterns. Thus, in the case of frequent subgraph mining, testing if a graph is contained in another graph is known as the subgraph isomorphism problem which is *NP*-hard. Frequent pattern mining algorithms are distinguished by their enumeration strategy, which can be horizontal or vertical. Horizontal mining algorithms, such as AGM developed by Inokuchi et al. [6] and FSG proposed by Kuramochi and Karypis [7], traverse the pattern space level-by-level, where the level k contains the patterns of size k, called k-patterns. Most often, this traversal is done in a bottom-up fashion, i.e. starting with the smallest patterns and successively enumerating patterns of increasing size. This approach allows to prune infrequent patterns that do not satisfy the downward closure property: since the support of a pattern is anti-monotone, a k-pattern is frequent only if all its (k-1)-patterns are frequent. On the other hand, vertical algorithms, like gSpan proposed by Han and Yan [13], FFSM by Huan et al. [5], and Gaston by Nijssen and Kok [9], explore the pattern space depth-first, recursively extending a k-pattern before visiting another pattern of the same size. This approach implicitly prunes infrequent patterns by only extending frequent ones, and has the advantage of requiring much less memory than the horizontal mining approach.

In order to have efficient frequent pattern mining algorithms, it is necessary to devise strategies to improve support computation. The strategy employed by most vertical mining algorithms consists in storing the embeddings of a pattern in the database, and updating these embeddings when the pattern is extended. Although this accelerates support computation on smaller databases, it is not efficient on large databases or when patterns can be embedded in many different ways due to pattern isomorphism. Another strategy is to check the downward closure of patterns before calculating their support. This can be done either by using horizontal mining algorithms or with a special traversal order that ensures that every pattern is explored after its subpatterns, see, e.g., [3]. Again, this strategy has some disadvantages. First, horizontal mining algorithms are known to be much less efficient than vertical mining algorithms. Also, it may not be possible to find an efficient depth-first traversal order that ensures visiting a pattern after its subpatterns. Finally, checking the downward closure

requires to store all the frequent patterns and to perform an isomorphism test on all the (k-1)-subpatterns of a k pattern, which can be very time consumming.

In this paper, we propose a simple strategy that can greatly improve frequent pattern mining by avoiding some costly support computations. This strategy, which can be used alone or in combination with another one, such as storing embeddings, uses background information on the frequent patterns to avoid exploring infrequent ones. The rest of the paper is as follows. In Section 2, we present our approach. Then, in Section 3, we evaluate our approach on the problem of frequent subgraph mining. Finally, we close this paper with a short summary of our contributions and results.

2 A novel approach

As it is the case for graphs, the pattern space can usually be structured in the form of a lattice. However, since a k-pattern can have many (k-1)-subpatterns, a depth-first traversal of the lattice might visit the same pattern more than once. To visit each pattern once, we need to transform the lattice into a rooted tree with a function p that assigns to each pattern X a unique parent $p(X)$. For frequent subgraph mining, the parent $p(G)$ of a graph G is typically a graph produced by removing a single vertex or edge from G. The only requirement for p is that it is insensitive to isomorphism: let X and Y be two isomorphic patterns, written $X \simeq Y$, we must have $p(X) \simeq p(Y)$. Once the pattern space is transformed into a rooted tree, we can then uniquely enumerate each pattern by traversing this tree, using either a depth-first or breadth-first traversal.

Proposition 1. Let p be a parent function such that $p(X) \simeq p(Y)$ if $X \simeq Y$, and suppose that each (k+1)-extension of a k-pattern is explored once. The traversal of the rooted tree defined by function p explores every pattern exactly once.

Proof. We prove this by recursion. Consider any pattern X and suppose that the parent $p(X)$ of X is explored exactly once, i.e. either $p(X)$ or a pattern isomorphic to $p(X)$ is explored. We will show that X is also explored once. Since every extension of $p(X)$ is explored, we know that X is explored if $p(X)$ is. Furthermore, consider a pattern Y isomorphic to X. By definition, we have that $p(X) \simeq p(Y)$. However, since either $p(X)$ or $p(Y)$ is explored and since every possible extension of $p(X)$ and $p(Y)$ is considered exactly once, either X or Y will be explored. Finally, because the root of the search tree is an ancestor of every other pattern in the tree, and since the root is explored exactly once, by recursion, every pattern is explored only once. □

The main idea of our approach is to select the parent function in a way that minimizes the number of infrequent patterns explored in the search. This is done as follows. Consider any k-pattern X and denote $S(X)$ the (k-1)-subpatterns of X. If X satisfies the downward closure, then it can be either frequent or not, and we need to compute its support. Otherwise, we know that it is not frequent. Since we only need to extend frequent patterns in the traversal, X will not be visited if its parent is infrequent. Thus, to make the visit of an infrequent k-pattern X as unlikely as possible, we must select the parent of X as one of the (k-1)-subpatterns of X that are least likely to be frequent. This idea is formalized in the following propositions.

Proposition 2. Let p and p' be two parent functions such that $sup(p(Z)) \leq sup(p'(Z))$, for all patterns Z. Every pattern X explored in the traversal of the search space defined by p is also explored in the search space defined by p'.

Proof. Since a pattern is explored if and only if its parent is frequent, we have that $sup(p(X)) \geq s_{min}$. Furthermore, since $sup(p(Z)) \leq sup(p'(Z))$ for every pattern Z, we have, in particular, $sup(p'(X)) \geq sup(p(X)) \geq s_{min}$. Therefore, X is also explored in the search space defined by p'. □

Proposition 3. Denote p^* the parent function that maps, for any k-pattern X, a (k-1)-subpattern of X with the lowest support, i.e. $sup(p^*(X)) = \min_{Y \in S(X)} sup(Y)$. A k-pattern is explored in the search space defined by p^* if and only if it is closed, i.e. if all its (k-1)-subpatterns are frequent.

Proof. Let X be any k-pattern explored in the search. We know that $sup(p^*(X)) \geq s_{min}$. Furthermore, since $p^*(X)$ is a (k-1)-subpattern of X with the lowest support, we have, for every (k-1)-subpattern $Y \in S(X)$, $sup(Y) \geq sup(p^*(X)) \geq s_{min}$. Thus, all ($k$-1)-subpatterns of X are frequent and X is closed. □

By selecting as the parent of a k-pattern X the (k-1)-subpattern with the lowest support, we can thus minimize the number of explored patterns and, consequently, the number of support computations. Finding the optimal parent function p^*, however, is as difficult as finding the frequent patterns. Yet, in many cases, the intrinsic nature of the data makes some patterns more likely to be frequent than others. Our approach consists in using this background information to select p, as follows. Suppose we have a function h, called heuristic, that evaluates the likeliness of a pattern X to be frequent, i.e. if $h(X) > h(Y)$ then X is more likely to be frequent than Y. This function can be determined using some general knowledge on the type of data used, by extracting information from the database in a preprocessing step, or learned by any machine learning algorithm on a training dataset. The only requirements are that h should be easy to compute, and should evaluate to the same value for isomorphic patterns, i.e. $X \simeq Y \Rightarrow h(X) = h(Y)$, otherwise isomorphic patterns could be explored redundantly from different parents. Furthermore, to avoid exploring patterns that are not closed, we must then choose the parent of X as a (k-1)-subpattern which minimizes h. However, since two non-isomorphic (k-1)-subpatterns can have the same value of h, this heuristic may not be powerful enough to guarantee that isomorphic patterns have the same parent. To insure this, we also need to define a total order on the pattern space, given by a precedence operator $\prec_P$. The parent of a k-pattern X can then be uniquely defined as the (k-1)-subpattern, among those with minimal value of h, that is also minimal with respect to $\prec_P$.

Figure 1 summarizes our approach for the depth-first traversal of the pattern space. Starting with the root pattern $\perp$, the algorithm launches the depth-first traversal by calling the recursive procedure *explore*. This procedure takes as input a database $\mathcal{D}$, a minimum support threshold s_{min} and the current k-pattern X, and returns a set $\mathcal{F}$ containing the frequent patterns of the sub-space rooted at X. The procedure first calculates the support of X. If its support is less than s_{min}, the empty set is returned. Otherwise, X is added to $\mathcal{F}$ and is extended as follows. For each possible (k+1)-extensions Y of X, the procedure computes the parent of Y as the first k-subpattern Z of Y with minimal value of h, following order $\prec_p$. If Z is isomorphic to X then X is the parent of Y. In this case, the recursive procedure *explore* is called on Y, and its returned set of frequent patterns is added to $\mathcal{F}$. When all extensions have been tested, the procedure returns $\mathcal{F}$.

Depth-first frequent pattern mining algorithm

Input: A database $\mathcal{D}$ and a minimum support threshold s_{min}.
Output: The frequent patterns $\mathcal{F}$ of $\mathcal{D}$.
$\mathcal{F} :=$ `explore(`$\mathcal{D}$, s_{min}, $\perp$`)` ;
return $\mathcal{F}$;

Procedure `explore(`$\mathcal{D}$, s_{min}, X`)`

Input: A database $\mathcal{D}$, a support threshold s_{min} and a k-pattern X.
Output: The frequent patterns $\mathcal{F}$ of the sub-space rooted at X.
if $sup(X, \mathcal{D}) < s_{min}$ **then return** $\mathcal{F} = \emptyset$;
$\mathcal{F} := \{X\}$;
foreach *(k+1)-extension* Y *of* X **do**
  $P := \perp$;
  foreach *k-subpattern* Z *of* Y, *following* $\prec_P$ **do**
    if $P = \perp$ *or* $h(Z) < h(P)$ **then** $P := Z$;
  if $P \simeq X$ **then** $\mathcal{F} := \mathcal{F} \cup$ `explore(`$\mathcal{D}$, s_{min}, Y`)` ;
return $\mathcal{F}$;

Figure 1: Our approach to depth-first frequent pattern mining.

3 Experimentation

In this section, we evaluate our approach on the frequent subgraph mining problem, using two different types of data: synthetic and real-life.

3.1 Synthetic data

In the first experiment, we considered the task of finding the frequent connected[1] subgraphs of synthetic datasets whose labels have different distributions. To generate this data, we first produced 6 label probability distributions D_i, $i = 1, \ldots, 6$, as follows. For each distribution D_i, we created 8 classes C_j, $j = 1, \ldots, 8$, to which we randomly assigned a unique label from the set $\{1, 2, 3, 4, 5, 6, 7, 8\}$. Then, for each class C_j, we obtained the probability p_{ij} of a vertex in D_i having the label of C_j, using the following formula:

$$p_{ij} = \frac{\left(1 + \frac{i-1}{5}\right)^{j-1}}{\sum_{k=1}^{8} \left(1 + \frac{i-1}{5}\right)^{k-1}}$$

As shown in Figure 2, the probability distributions obtained in this way are increasingly skewed. Thus, for the first distribution, all labels are equally probable. However, in the last distribution, the label of class C_8 has a 50% chance of being on a vertex, while the probability of having a vertex with the label of class C_1 is only 0.4%. For each of these 6 distributions, we then generated 3 datasets using the random generator developed by Karypis and Kuramochi [7], each dataset containing 10000 unlabeled graphs. As parameters, we used an average size of the database graphs of 15, an average number of frequent subgraphs of 25, and an average

[1] A graph G is connected is there is a path connecting any two vertices of G.

Class	Distribution					
	D_1	D_2	D_3	D_4	D_5	D_6
C_1	12.5	6.1	2.9	1.4	0.7	0.4
C_2	12.5	7.3	4.1	2.3	1.3	0.8
C_3	12.5	8.7	5.7	3.7	2.4	1.6
C_4	12.5	10.5	8.0	5.9	4.3	3.1
C_5	12.5	12.6	11.2	9.4	7.7	6.3
C_6	12.5	15.1	15.6	15.0	13.8	12.5
C_7	12.5	18.1	21.9	24.0	24.9	25.1
C_8	12.5	21.7	30.6	38.4	44.9	50.2

Figure 2: Probability (%) of having a vertex with the label of classes C_j, $j = 1, \ldots, 8$, for distributions D_i, $i = 1, \ldots, 6$.

size of the frequent graphs of 15. These values are fairly standard for benchmarking, and ensure that most of the CPU time is spent on support calculation. Finally, we labelled the vertices of the generated datasets using their respective distribution. The graph edges were all given the same label.

We then tested four frequent subgraph mining algorithms on these datasets. The first algorithm, referred to as *heuristic* in the results, is based on an algorithm called SyGMA, that we developed to find the frequent connected subgraphs of datasets having a limited number of labels [4]. Like SyGMA, the *heuristic* algorithm transforms the search space into a rooted tree using a parent function p, and explores this tree depth-first. However, *heuristic* differs from SyGMA in the fact that it uses background knowledge to define p. In this case, the background knowledge used is the number of edges, in the database, with incident vertices of given labels. This information is obtained, with little added cost, while reading the database. Let G be a graph explored in the search and let φ be a permutation of the vertices of G. The code of G, under φ, is the string obtained by considering the elements of the adjacency matrix of a graph G, following the order of the vertices in φ. Likewise, the canonical code of G is the lexicographically minimal code, obtained under any permutation. This code can be obtained efficiently using, for instance, McKay's Nauty algorithm. Let φ^* be a permutation leading to the canonical code, and denote $\varphi^*(v)$ the position of a vertex v in φ^*. We define a total order on the edges of G using the precedence operator $\prec_E$, defined as follows. Let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ be two edges such that $\varphi^*(u_1) < \varphi^*(v_1)$ and $\varphi^*(u_2) < \varphi(v_2)^*$, we have $e_1 \prec_E e_2$ iff either one of these two cases is true:

1. $\varphi^*(u_1) < \varphi^*(u_2)$
2. $u_1 = u_2$ and $\varphi^*(v_1) < \varphi^*(v_2)$.

This order has the property that equivalent edges in two isomorphic graphs are ordered in the same fashion. Since our depth-first search only explores connected graphs, a graph is explored only if its parent is connected. Denote $G - \{e\}$ the graph obtained by removing from G an edge e and all the vertices that become isolated after this removal, i.e. the vertices that are only incident to edge e. We say that e is a disconnecting edge if $G - \{e\}$ is unconnected. We then define the parent of G, in the *heuristic* algorithm, as the graph produced by removing from G the first non-disconnecting edge, following $\prec_E$, whose incident vertex labels are most common in the database, i.e. for which the number of edges in the database having the same pair of labels is the greatest. Let e_1 and e_2 be any two edges of G and denote N_1 and N_2, respectively, the number of edges of the database that have the same label pair as e_1 and e_2.

The heuristic function h, in this case, is such that $h(G - \{e_1\}) < h(G - \{e_2\})$ if $N_1 > N_2$. If e_1 and e_2 are topologically equivalent edges, i.e. if there is an automorphism mapping the vertices of one edge to those of the other, then $G - \{e_1\} \simeq G - \{e_2\}$. Since e_1 and e_2 necessarily have the same label pair, we have $h(G-\{e_1\}) = h(G-\{e_2\})$. Thus, h satisfies the requirement to evaluate to the same value for isomorphic graphs.

The second algorithm, called *random*, is another implementation of our approach for which the parent of a graph is selected randomly. In order that isomorphic graphs have the same parent, we re-initialize the random number generator for every visited graph G, using the canonical code of G. We then enumerate the non-disconnecting edges of G following $\prec_E$, and generate, for each of these edges, a random number. The parent of G is then obtained by removing from G the first enumerated edge for which the random number was highest.

The third algorithm, named *prefix*, is our own implementation of the depth-first search (DFS) coding scheme of gSpan [13]. Briefly, this scheme assigns to each graph G a code obtained by concatenating the vertex indexes and labels of the edges of G, following the order in which these edges where added to G. A depth-first search exploration of the lattice is then made, in such a way that the graphs are visited in ascending code values. Thus, when visiting a graph G, if the code of G is not minimal, i.e. there exists a pre-order traversal of G yielding a lesser code, then a graph isomorphic to G has already been visited in the search and G is pruned. The *prefix* algorithm uses the same support computation procedures as *heuristic* and *random*.

Finally, the last algorithm, called *closure*, is a variation of *prefix* that explores the search space so that the subgraphs of a graph G are explored before G, thus allowing to check the downward closure of G. The search strategy employed by *closure*, called reverse prefix search [3], differs from the one used by *prefix* in the fact that the extensions of a graph are explored in descending code values, and that the frequent graphs found in the search are stored. Again, *closure* uses the same procedures to compute the support of a graph as the other three algorithms.

Figures 3 and 4 report, for each of the six distributions, the average results over their three datasets. On the left side are shown the percentage of visited graphs that are closed and that are frequent, as found by the *closure* algorithm for decreasing support thresholds.[2] The smaller the ratio of closed graphs is, the more graphs can be pruned by checking downward closure. This ratio also gives us an idea of the potential benefits of our *heuristic* algorithm. Moreover, a smaller gap between the ratios of closed and frequent graphs translates into less useless support calculations. The right side of these figures gives the runtimes, in seconds, of the four tested algorithms, also for decreasing support values. From these results, we can make several observations. First, we notice that the percentage of visited graphs that are frequent remains fairly constant (values range between 7% and 13%), for all distributions and support thresholds. On the other hand, the ratio of visited graphs that are closed decreases as the label distribution becomes more skewed, and as the support threshold is lowered. As a consequence, the efficiency of the *heuristic* and *closure* algorithm, compared to the *random* and *prefix* algorithms, increases in the same way. Thus, for the equiprobable distribution, D_1, the runtimes of all four algorithms are roughly the same. However, for the most skewed distribution, D_6, both *heuristic* and *closure* algorithms present a two-fold speedup over the other two algorithms. We also observe that the *random* algorithm is somewhat faster than the *prefix* algorithm (5% to 17% faster), due to the fact that the fixed lattice exploration order of

[2]The threshold values are given as a percentage of the dataset graphs.

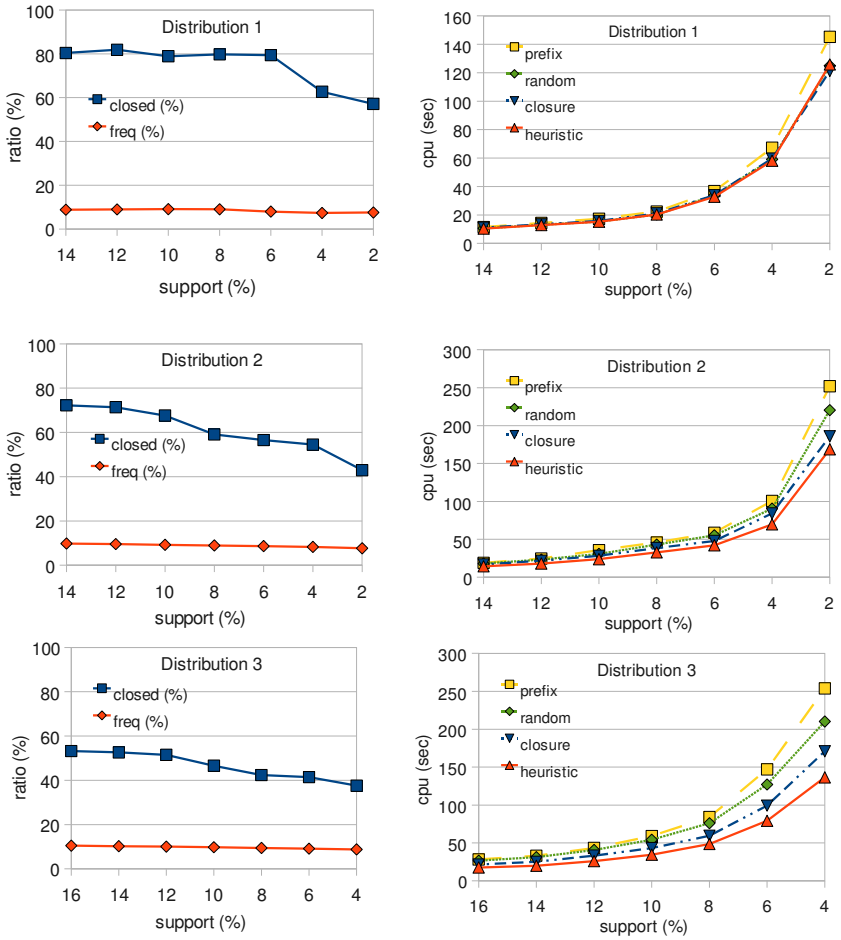


Figure 3: Ratio of closed and frequent visited graphs (*left*) and runtimes of the tested algorithms (*right*), for synthetic datasets using label distributions D_1, D_2, D_3.

prefix is not well suited for the data. We finally notice that our *heuristic* approach is faster (up to 22% faster) than the *closure* algorithm, in all but one case (distribution D_1, support threshold 2%). Although one might think that checking the downward closure is what slows down the second algorithm, this is not the case since this operation is negligible compared to support computation. In fact, the *heuristic* algorithm performs less subgraph isomorphisms tests than *closure*, eventhough *closure* is the algorithm that computes the support of the least number of graphs. This surprising result can be explained as follows. Since a graph G is contained by a database graph only if its parent is, the number of subgraph isomorphism tests required to compute the support of G is bounded by the support of its parent. By selecting the parent of G as the graph least likely to be frequent, the *heuristic* algorithm thus tends to reduce the number of subgraph isomorphism tests required for G.

3.2 Real-life data

In the second experiment, we evaluated our approach on a real-life dataset from the field of chemoinformatics. This dataset, which contains a set of 340 chemical compounds modelled as labeled graphs, was devised as a benchmark for the Predictive Toxicology Evaluation

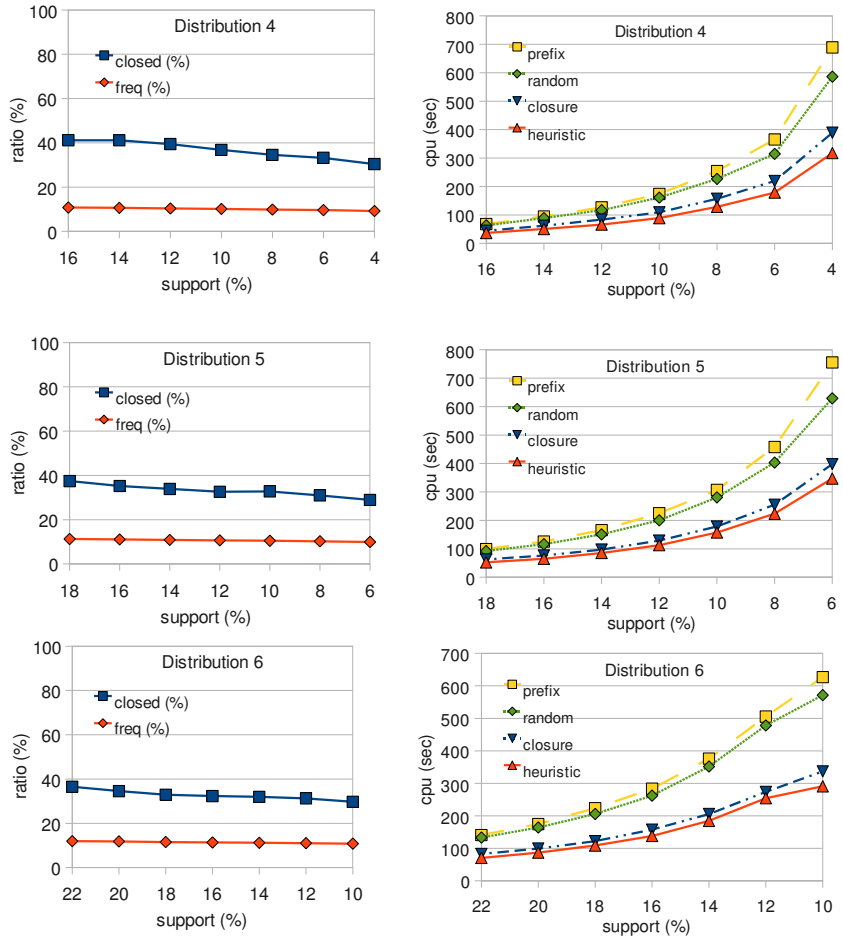


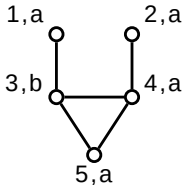
Figure 4: Ratio of closed and frequent visited graphs (*left*) and runtimes of the tested algorithms (*right*), for synthetic datasets using label distributions D_4, D_5, D_6.

(PTE) challenge [10]. As we have done in the previous experiment, we tested four algorithms, *heuristic*, *random*, *prefix* and *closure*, on the task of finding the frequent connected subgraphs of this dataset.[3]

For this experiment, we have modified our *heuristic* algorithm to exploit some common characteristics of this type of data: the frequent subgraphs are mostly cycle-free connected graphs, whose vertices have a low degree.[4] We illustrate how this information was used in our algorithm with a small example. Consider the graph shown on the left side of Figure 5, that we will denote G. The values shown beside each vertex of G are the index (1,2,3,4 or 5) and the label (a or b) of this vertex. Since the edges of G are non-disconnecting, they must either be part of a cycle (edges $(3,4)$, $(3,5)$, $(4,5)$ in this example) or have a vertex incident to no other edge (here $(1,3)$, $(2,4)$). If we remove an edge contained in a cycle, such as $(3,4)$, this cycle will not be present in the resulting graph. However, if we remove an edge of the second type, e.g. $(1,3)$, the graph we obtain will keep all its cycles. Because cycle-free graphs are more likely to be frequent, and since we want as parent $p(G)$ of G one of its least

[3]The edge labels of the dataset were discarded for this experiment.

[4]The degree of a vertex is the number of edges incident to this vertex.



Label pair	Occurrences
(a, a)	300
(a, b)	200
(b, b)	150

Figure 5: A labeled graph (*left*) and the number of database edges having vertices with the given labels (*right*).

frequent subgraphs, $p(G)$ should thus be obtained by removing from G an edge incident to a vertex of lowest degree. This approach also has the benefit that the parent graphs will have vertices of higher degree, and thus, will have less chance of being frequent. In this example, vertices 1 and 2 are the only ones with the lowest degree of 1. Thus the parent of G should be obtained by removing an edge incident to one of these vertices, i.e. either $(1, 3)$ or $(2, 4)$. To make our *heuristic* algorithm even more efficient, we also used background information on the vertex labels. Let $e_1 = (u_1, v_1)$ and $e_2 = (u_2, v_2)$ be two non-disconnecting edges of G (suppose, without loss of generality, that $deg(u_1) \leq deg(v_1)$ and $deg(u_2) \leq deg(v_2)$), and denote N_1, N_2, once more, the number of edges of the dataset that have incident vertices with the same labels as e_1 and e_2. The heuristic function h used in our algorithm is such that $h(G - \{e_1\}) < h(G - \{e_2\})$ if either one of the following cases is true:

1. $deg(u_1) < deg(u_2)$
2. $deg(u_1) = deg(u_2)$ and $N_1 > N_2$
3. $deg(u_1) = deg(u_2)$ and $N_1 = N_2$ and $deg(v_1) < deg(v_2)$.

Suppose that, in our example, the number of database edges whose incident vertices have the corresponding pair of labels is as shown on the right-side of Figure 5. In this case, since the label pair (a, a) of edge $(2, 4)$ is more frequent than the label pair (a, b) of edge $(1, 3)$ (300 occurrences versus 200) the parent of G will be the graph $G - \{(2, 4)\}$.

Figure 6 summarizes the results of this experiment. On the top left are shown the percentages of visited graphs that are frequent and closed, as found by *closure*, for decreasing values of support threshold. On the top right are presented the percentage of graphs, visited by all four algorithms, that are cycle-free. The graphic located at the bottom left gives the runtimes of the algorithms, again for decreasing support threshold values. Finally, at the bottom right are shown the number of subgraph isomorphism tests performed by all four algorithms. As in the first experiment, we notice that, while the ratio of frequent graphs is fairly constant for all support threshold values, the ratio of closed graphs decreases as the support threshold lowers. However, due to the particular nature of the data, the ratio of closed graphs is much lower than it was for the synthetic datasets. Because of this, we can thus expect a greater performance increase for *closure* and *heuristic*, compared to what we observed for synthetic data. Also, from the ratio of visited graphs that are cycle-free, we observe that both the *prefix* and *closure* algorithms are well adapted to limit the search to this type of graph (ratios ranging from 88% to 85%). Moreover, comparing the *random* and *heuristic* algorithms, we can see that the heuristic function helps in avoiding graphs with cycles. Thus, while the *random* and *heuristic* algorithms have similar ratios for a support threshold of 24% (respectively 85% and 88%), the *heuristic* algorithm does a much better job at avoiding such graphs (ratio of 92% versus 65% for *random*), for a support threshold of 3%. Furthermore, as we expected, *closure* and *heuristic* clearly outperform *prefix* and *random*. Thus, *heuristic* is 8 to 15 times faster

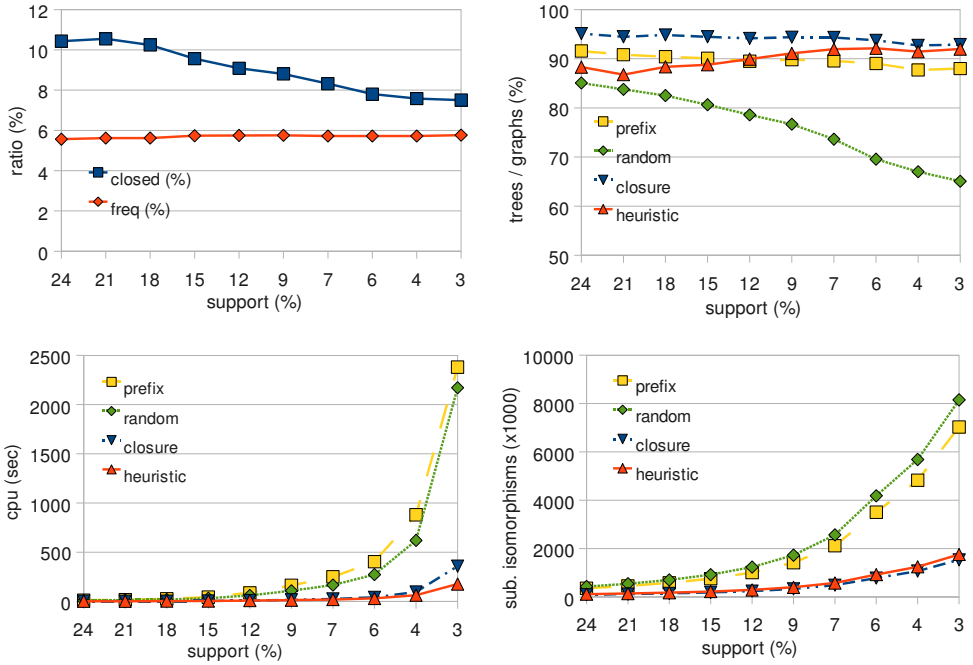


Figure 6: Ratio of closed and frequent visited graphs (*top left*), ratio of visited graphs that are trees (*top right*), runtimes (*bottom left*) and number of subgraphs isomorphism tests performed (*bottom right*), for the tested algorithms on the Predictive Toxicology Evaluation (PTE) dataset.

than *prefix* for all support threshold values. Finally, as we did for the first experiment, we notice that *heuristic* is somewhat faster than *closure* (up to twice faster for a support threshold of 3%), although, this time, the number of subgraphs isomorphism tests performed by the two algorithms is very comparable. Since the dataset only contains 340 graphs (versus 10000 for the synthetic datasets), the downward closure check of the *closure* algorithm accounts for a greater portion of the algorithm's runtime, and could, therefore, explain this performance gap.

4 Conclusion

We have presented, in this paper, a simple and general strategy that improves the task of finding the frequent patterns of a database containing structured data. This novel approach uses background information on the frequent patterns, in the form of a heuristic function that transforms the search space into a rooted tree such that the parent of a pattern is as unlikely as possible to be frequent. This allows to avoid exploring a great number of infrequent patterns and, consequently, to reduce the number of costly support computations. To evaluate our approach, we have tested it on a well-known specialization of the frequent pattern mining problem, the frequent subgraph mining problem, where the task is to find the connected graphs for which the support in the database is greater than a given threshold. These tests were carried out on two types of data: synthetic datasets that have a skewed distribution of vertex labels, and a real-life dataset from the Predictive Toxicology Evaluation (PTE) challenge. The results obtained for these tests have shown our approach to be more efficient

than a specialized technique using depth-first search (DFS) coding, and to be as powerful as the more complex strategy of testing downward closure.

References

[1] C. Borgelt, M. R. Berthold, and D. E. Patterson. Molecular fragment mining for drug discovery. Number 3571 in Lecture Notes in AI, pages 1002–1013. Springer Verlag, 2005.

[2] S. Chakrabarti. *Mining the Web: Discovering Knowledge from Hypertext Data*. Morgan-Kauffman, 2002.

[3] M. Cohen and E. Gudes. Diagonally subgraphs pattern mining. In *DMKD '04: Proceedings of the 9th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 51–58. ACM, 2004.

[4] C. Desrosiers, P. Galinier, P. Hansen, and A. Hertz. Sygma: Reducing symmetry in graph mining. Technical Report G-2007-12, Les Cahiers du GERAD, December 2007.

[5] J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraph in the presence of isomorphism. In *Proceedings of the 3rd IEEE International Conference on Data Mining (ICDM)*, pages 549–552, 2003.

[6] A. Inokuchi, T. Washio, and H. Motoda. An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery*, pages 13–23. Springer-Verlag, 2000.

[7] M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *Proceedings of the First IEEE Conference on Data Mining*, pages 313–320, 2001.

[8] B. Liu. *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Data-Centric Systems and Applications. Springer, 2007.

[9] S. Nijssen and J. N. Kok. The gaston tool for frequent subgraph mining. In *Proceedings of the International Workshop on Graph-Based Tools (Grabats 2004)*, pages 281–285. Elsevier, October 2004.

[10] A. Srinivasan, R. D. King, S. H. Muggleton, and M. Sternberg. The predictive toxicology evaluation challenge. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1–6. Morgan-Kaufmann, 1997.

[11] M. J. E. Sternberg, R. D. King, A. Srinivasan, and S. Muggleton. Drug design by machine learning. In *Machine Intelligence 15*, pages 328–338, 1995.

[12] J. Wang, M. Zaki, H. Toivonen, and D. Shasha, editors. *Data Mining in Bioinformatics*. Springer, 2005.

[13] X. Yan and H. Jiawei. gspan: Graph-based substructure pattern mining. In *ICDM '02: Proceedings of the 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 721–724, Washington, DC, USA, 2002. IEEE Computer Society.