

Frequent Subgraph Miners: Runtimes Don't Say Everything

Siegfried Nijssen¹ and Joost N. Kok²

¹ Albert-Ludwigs-Universität, Georges-Köhler-Allee, Gebäude 097, D-79110, Freiburg im Breisgau, Germany.

² LIACS, Leiden University, Niels Bohrweg 1, 2333 CA, Leiden, The Netherlands.
snijssen@informatik.uni-freiburg.de

Abstract. In recent years several frequent subgraph miners were proposed. The authors of these new algorithms typically compared the runtimes of their implementations with those of previous implementations to confirm the efficiency of their methods. To get a better perspective on the mutual benefits of the algorithms, Wörlein et al. [9] performed an experimental evaluation of re-implementations of several depth-first graph miners, where also some statistics beyond runtimes were compared. In this paper we present results of an additional experimental comparison of several graph miners, which differs in the following aspects from this previous study: (1) we compare original implementations; (2) we compare these implementations on a larger set of measures than runtimes, thus providing further insight in the benefits of the algorithms; (3) we include breadth-first graph miners and free tree miners in the comparison.

1 Introduction

Given a database consisting of small graphs, for example, molecular graphs, the problem of mining frequent subgraphs is to find all subgraphs that are subgraph isomorphic with a large number of example graphs in the database. References to applications can be found in [2, 4, 8, 3]. The first frequent subgraph miner was Inokuchi et al.'s AGM algorithm (2000), for unconnected subgraphs. The algorithm was followed by the FSG algorithm of Kuramochi and Karypis [7] (2001), and an adaption of AGM, AcGM, for mining connected subgraphs [6] (2002). These initial algorithms performed the search breadth-first; the first depth-first graph miners were MoFA [2] and gSpan [10] (2002), followed by FFSM [5] (2003) and GASTON [8] (2004). In parallel, also free tree (cycleless graph) miners were developed: both the breadth-first FREETREEMINER (2003) and the depth-first HYBRIDTREEMINER (2004) by Chi et al. (see [3] for references). The development of new graph miners was motivated by the supposed inefficiency of earlier graph miners. In this paper, we perform a large set of experiments with the implementations of the graph and free tree miners that were used in most of these original publications.

We have several aims by doing these experiments. First, our study verifies earlier studies, both the studies of the original implementers, as the re-implementation study of [9]. This provides either additional evidence for the

results obtained in these studies, or will allow us to refute claims made in these studies, in both cases providing us additional insights in the effects of, for instance, implementation optimizations. Second, by computing additional measures, we wish to obtain a deeper understanding in the true benefits of the graph mining algorithms. It can be observed that most graph mining algorithms can be separated in separate components for ‘candidate generation’ and ‘candidate evaluation’. Until now, most algorithms provide a unique combination for each of these elements. By performing additional experiments, we can provide insight in the question if it is useful to try out different combinations of these components. Thus, this study is complementary to the earlier graph mining study of [9], and our earlier study on the efficiency of frequent tree miners [3].

The paper is organized as follows. First, we shortly review the problem of frequent subgraph mining and the frequent subgraph miners. The main part of the paper consists of a large number of experiments. Finally, we conclude.

2 Concepts and Algorithms

In this section we briefly recall the most important concepts and algorithms.

Concepts A *labeled undirected graph* is a quadruple (V, E, λ, Σ) , where V is a set of vertices, $E \subseteq \{e \subseteq V : |e| = 2\}$ is a set of edges, and $\lambda : V \cup E \rightarrow \Sigma$ is a function that assigns labels to the vertices and edges. In this paper, we only consider connected graphs, in which there is a path between every pair of nodes. Graph $G' = (V', E', \lambda', \Sigma')$ is a subgraph of graph G if $V' \subseteq V$, $E' \subseteq E$, for all $x \in V' \cup E' : \lambda'(x) = \lambda(x)$ and $\Sigma' \subseteq \Sigma$. Graph G and G' are isomorphic iff there is a bijective function $\phi : V \rightarrow V'$ such that (1) $\forall \{v_1, v_2\} \in E : \{\phi(v_1), \phi(v_2)\} \in E'$ and $\lambda(\{v_1, v_2\}) = \lambda'(\phi(\{v_1, v_2\}))$, (2) $\forall v \in V : \lambda(v) = \lambda'(\phi(v))$. Graph G is subgraph isomorphic with G' , denoted by $G' \succeq G$, iff G is isomorphic with a subgraph of G' . The corresponding function ϕ , which maps nodes from the subgraph to the supergraph, is called an embedding. Given a multiset of graphs D (also referred to as database graphs), the frequency (or support) of a graph G , denoted by $freq(G)$ is the cardinality of the set $\{G' \in D | G' \succeq G\}$. Assuming that each database graph has an identifier, this set could alternatively consist of identifiers; we refer to this set as the transaction identifier (TID) list of the subgraph. Given a threshold $minsup$, a (pattern) graph G is frequent iff $freq(G) \geq minsup$; frequent subgraph miners find all such subgraphs. An important property is the antimonotonicity property that states that if $G \succeq G'$, $freq(G) \leq freq(G')$. If we start searching from the smallest subgraphs, a consequence of this property is that we do not need to consider supergraphs of infrequent graphs; we can cut parts of the search space.

Two different subgraphs can be isomorphic with each other; let $V = \{v_1, v_2\}$, $E = \{\{v_1, v_2\}\}$ and $\Sigma = \{\mathbf{a}, \mathbf{b}\}$; then the graph $G_1 = (V, E, \lambda_1, \Sigma)$ with $\lambda_1(v_1) = \mathbf{a}$ and $\lambda_1(v_2) = \mathbf{b}$ is isomorphic with the graph $G_2 = (V, E, \lambda_2, \Sigma)$ having $\lambda_2(v_1) = \mathbf{b}$ and $\lambda_2(v_2) = \mathbf{a}$. Care has to be taken that frequent subgraph miners do not find such isomorphic subgraphs multiple times. All algorithms address this issue by determining a canonical string for a graph. The idea is that every

graph has a corresponding string representation, for example, $(v_1, v_2, \mathbf{a}, \mathbf{b})$ for G_1 and $(v_1, v_2, \mathbf{b}, \mathbf{a})$ for G_2 . By imposing a total order on the strings of isomorphic graphs, for example, lexicographically, we can decide that one representation is the highest. This string is considered to be the canonical representation of the isomorphic graphs. As no polynomial algorithm is known to compute graph isomorphism or subgraph isomorphism, all graph mining algorithms use exponential search algorithms to find the canonical label or embeddings; free tree mining algorithms, on the other hand, can exploit polynomial algorithms.

Breadth-First Algorithms The breadth-first algorithms have the same setup as the original APRIORI algorithm for mining frequent itemsets [1]. They iterate a process of generating candidate subgraphs and determining their support in the database. Candidates are generated by joining two subgraphs that were previously found to be frequent. After the joined subgraph is obtained, it is checked whether all its subgraphs are frequent; if not, the candidate is pruned.

To represent candidates, AcGM and FSG use a canonical string that is obtained from an adjacency matrix. FSG's code allows for the quick computation of the canonical string for any given graph. AcGM's representation is optimized to minimize the generation of non-canonical graphs. Chi et al.'s FREETREEMINER is similar to AcGM, but uses a polynomially computable canonical string.

The search for embeddings is improved by exploiting the observation that an embedding for a supergraph is also an embedding for its subgraphs. In FSG and the FREETREEMINER, a TID list is stored with every pattern graph. AcGM takes the idea a step further by also storing one embedding for each database graph in the TID list, and using it as the start of the exponential search.

Depth-First Algorithms The depth-first algorithms do not subdivide the search into strict candidate generation and candidate evaluation phases. Essentially, each of these algorithms scans all embeddings of a subgraph in the database, and collects from that scan the support of the refinements, i.e., the individual edges and nodes that can be connected to the subgraph. The search recurses immediately on each of the frequent refinements.

To avoid duplicate subgraphs in the output, all depth-first graph miners use a special kind of canonical string, which has the property that every prefix of the string is also canonical. A refinement corresponds to extending a canonical string. For each extended string, it is checked whether it is canonical; any non-canonical string can be removed immediately. Although for every refined string it has to be checked with an exponential algorithm if the resulting string is really canonical, an interesting property of most codes is that it is often easy to decide that a code is *not* canonical. As a small example, almost all codes sort sibling nodes in the order of their labels. This order limits which new siblings can be added; we will call this the 'label trick'. Such simple tricks limit which nodes in the data have to be scanned for extensions. gSpan, FFSM and GASTON differ in their canonical string, and consequently, also in the 'easy' rules that can be used to eliminate strings that are not canonical:

- gSpan uses a code that consists of a list of edges in the order of a depth-first traversal tree of the subgraph. Canonical refinements can only connect to a node on the rightmost path of this tree. The ‘label trick’ applies to gSpan.
- FFSM uses a code that consists of a concatenation of columns of an adjacency matrix. The last column of the adjacency matrix restricts to which nodes new nodes can be connected; ‘label tricks’ can also be applied in FFSM.
- GASTON uses a code that consists of a concatenation of a code for paths, trees and cycles. The code allows to determine in $O(1)$ time if a refinement leads to a non-canonical code for a tree. The ‘label trick’ can not be applied.
- The HYBRIDTREEMINER uses another code to represent free trees. The code is efficiently computable, and guarantees that trees or paths are never enumerated more than twice.

To evaluate the frequency of subgraphs, several alternatives have been proposed. In gSpan and GASTON (RE variant) a TID list is maintained with every subgraph. All embeddings are recomputed for graphs in the TID list. For each embedding, refinements that are not pruneable by easy rules, are counted. Non-canonical refinements are pruned later.

An alternative is to not to recompute embeddings, but to store them, as only embeddings of subgraphs can lead to embeddings of supergraphs. Although the frequency of refinements could be collected from the data, FFSM and GASTON (OS variant) use a more elaborate approach, in which the embeddings of some refinements are obtained by *joining* the embeddings of other subgraphs. The motivation is that this reduces the chances of building embedding lists for graphs that are not frequent, as at least two subgraphs need to be frequent before an embedding list is constructed. However, to reduce the number of subgraphs for which the embedding list has to be stored at the same time, it is necessary that joining is performed in a ‘local’ way in the search tree. To this purpose, all algorithms require an additional set of embedding lists for graphs that are not canonical.

3 Experiments

For our comparison, we obtained source code of the FREETREEMINER, the HYBRIDTREEMINER and GASTON (RE, OS), and binaries of gSpan, FFSM, FSG and AcGM. The source code was compiled using the GNU C compiler and the O3 compiler option. We used a range of 6 different datasets: (1) three tree datasets (A1, A3, A4) were generated using Zaki’s tree dataset generator [11]. A4 is equal to A1, except for the node labels. Most frequent trees in dataset A3 have diameter 2, and differ mainly in degree of the nodes; (2) two molecular datasets. The PTE dataset was used in [10, 8], and includes hydrogens in the graph encoding; the Cancer datasets is a similar, although larger, dataset obtained from the National Cancer Institute, and does not include hydrogen in the encoding³; (3) a protein secondary structure dataset of Huan et al. [4].

³ We performed further experiments with molecular datasets and real-world tree datasets. Results on these datasets are similar and omitted due to space constraints.

	A1	A3	A4	PTE	Cancer	Protein
Number of graphs	5000	10000	5000	340	32557	40
Number of nodes	62936	183743	62936	9189	857126	9502
Number of edges	57936	173743	57936	9317	922081	22016
Avg max number equally labeled nodes	2.7	3.9	12.6	11.9	19.2	25.6
Avg number of nodes labeled neighbors	12.6	18.4	12.6	27.0	26.3	237.6
Avg max number equally labeled neighbors	1.7	3.2	5.2	2.6	2.7	1.1
Number of node labels	10	10	1	1	66	67
Number of edge labels	1	1	1	4	3	1

Fig. 1. Characteristics of the datasets.

Algorithm	PTE				Cancer	
	2%	3%	4%	5%	4%	4%
GASTON (OS)	9.1MB	4.4MB	3.4MB	3.0MB	430MB	
GASTON (RE)	1.5MB	1.3MB	1.3MB	1.3MB	23MB	
FFSM	8.2MB	4.1MB	3.7MB	3.2MB	257MB	
gSpan	3.8MB	2.8MB	2.8MB	2.8MB	46MB	
AcGM	33.9MB	5.3MB	2.2MB	1.6MB	434MB	
FSG	123.5MB	25.8MB	25.8MB	25.8MB	107MB	

Fig. 2. Memory usage of the algorithms.

	A1				A3			
	30%	10%	8%	6%	7.5%	4%	3%	2.5%
Minimum support	838	18552	36945	93979	156	1826	5405	10451
Frequent graphs	103%	107%	108%	108%	107%	101%	101%	101%
gSpan — Suboptimality	310%	231%	213%	195%	254%	289%	254%	234%
FFSM — Suboptimality	99%	70%	62%	55%	100%	99%	91%	85%
Join efficiency								
GASTON — Suboptimality	207%	136%	125%	115%	212%	214%	161%	136%
Join efficiency	27%	41%	43%	47%	11%	29%	44%	52%
Join necessity	97%	95%	96%	97%	100%	98%	98%	99%
HYBRID TREEMINER — Suboptimality	160%	168%	173%		104%	101%	101%	
Join efficiency	21%	20%	21%		33%	39%	42%	
Join necessity	96%	96%	94%		100%	100%	100%	

Fig. 4. Characteristics of the gSpan, FFSM and GASTON algorithms on the A1, A3, PTE, Protein and Cancer datasets.

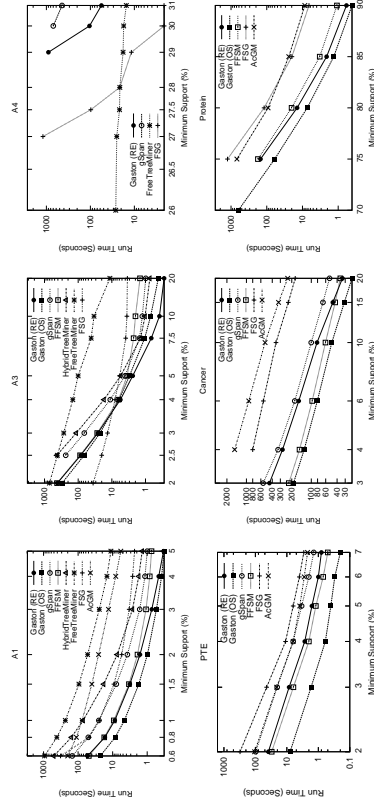


Fig. 3. Runtime experiments on all datasets.

	PTE				Protein				Cancer				
	20%	6%	3%	2%	80%	75%	70%	9%	5%	3%	9%	5%	3%
Minimum support	190	2326	22758	136949	26740	276297	2709331	1818	5663	15566			
Frequent graphs	139%	134%	137%	129%	-	-	-	161%	158%	159%			
gSpan — Suboptimality	192%	152%	133%	134%	166%	144%	132%	-	-	-			
FFSM — Suboptimality	61%	50%	45%	41%	38%	35%	35%	-	-	-			
Join efficiency													
GASTON — Overall suboptimality	176%	149%	151%	165%	180%	219%	241%	170%	157%	147%			
Free tree suboptimality	160%	125%	109%	103%	129%	111%	104%	156%	139%	126%			
Cyclic join efficiency	100%	100%	94%	89%	70%	75%	80%	39%	29%	28%			
Free tree join efficiency	100%	100%	94%	90%	82%	68%	61%	94%	93%	93%			
Free tree join necessity	52%	37%	64%	75%	38%	49%	55%	27%	27%	27%			
Free tree join efficiency	83%	73%	86%	90%	100%	99%	99%	96%	95%	95%			

Properties regarding the number of equally labeled nodes, edges and neighbors, are listed in Figure 1. These properties are important, as they yield a higher branching factor in algorithms that perform an exponential search for embeddings. The datasets represent a broad range of such properties.

To perform our experiments we used two computers: all cyclic graph datasets were mined on an AMD Athlon XP1600+ with 512MB main memory, running Mandrake Linux 10; all free tree datasets were mined on an Intel Pentium IV 2.8Ghz with 512MB main memory, running Red Hat Linux 7.3.

The results of runtime experiments are given in Figure 3. The experiments are measured using the Unix `time` command and are averaged over 3 runs.

The experiments show the importance of the branching factor of subgraph isomorphism algorithms. The depth first graph mining algorithms fail miserably on the A4 dataset, which has a high branching factor (they run too long, or consume too much memory); only the breadth-first `FREETREEMINER` performs well. Otherwise, however, the `FREETREEMINER` usually performs worse than `FSG`. The A3 dataset shows the difference between breadth-first algorithms that have to find one embedding per database graph (like `FSG`), and depth-first algorithms that have to find all embeddings. In some cases the breadth-first graph miners run out of memory, which is caused by extremely large sets of candidates.

The results on the cyclic datasets are similar to those on the tree datasets, and confirm the observations of previous publications.

The runtime experiments however do not show whether differences are due to a ‘better’ canonical form, as claimed by several authors, or due to more optimized implementations or memory-runtime trade-offs. Results of experiments to assess the quality of the canonical form are listed in Figure 4. The following statistics are listed in this table:

- frequent graphs: the number of frequent subgraphs resulting from the runs of all algorithms;
- suboptimality: the number of frequent subgraphs for which the canonical string test is computed, divided by the number of frequent subgraphs; the higher this number is, the less well does the code prevent isomorphic graphs using easy rules, and thus more redundant supports are computed;
- join efficiency: the number of joins that results in a frequent subgraph, divided by the total number of joins that is performed; the closer to 100% this value is, the better does the algorithm succeed in only computing embedding lists that are really frequent; in the case of `GASTON`, there is a distinction between the joins between embedding lists of trees and of cyclic graphs;
- join necessity: the number of joins that results in a subgraph with support higher than zero, divided by the total number of joins that is performed; if this number would not be close to 100%, joining of embedding lists would not make much sense, as we are performing many computations that an algorithm that collects extensions from data would not need to perform.

For `GASTON` and the `HYBRIDTREEMINER`, we obtained these measures by changing the source code. For `gSpan` and `FFSM`, we used the Valgrind tool available

Algorithm	A1 1%				PTE 2%				Protein 75%		
	1×	3×	5×	Regression	1×	2×	3×	Regression	1×	2×	3×
GASTON (OS)	4.6s	14.8s	26.1s	5.5x-1.4s	7.9s	15.2s	23.0s	7.5x+0.4s	60s	116s	183s
GASTON (RE)	8.7s	27.7s	48.0s	9.9x-1.8s	39.9s	76.3s	112.4s	36.2x3.7s	148s	398s	643s
FFSM					29.7s	55.7s	82.2s	26.3x+3.4s	177s	353s	554s
gSpan	15s	47s	81s	16.6x-2.2s	100s	186s	271s	85.4x+14.9s	(78s)	(166s)	(290s)
FSG	17s	35s	53s	9.1x+7.7s	316s	402s	489s	86.3x+229.8s	(148s)	(398s)	(643s)
AcGM					107s	170s	234s	63.5x+43.3s			
HYBRIDTREEMINER	46s	146s	248s	48.9x-1.1s							
FREE TREEMINER	243s	715s		238.9x+1.5s							

Fig. 5. Scale-up experiments on several datasets; experiments between brackets were obtained on an Intel Pentium IV.

at www.valgrind.org. Valgrind makes it possible to count numbers of function calls, even with sufficient reliability for binaries that have been compiled without debugging information. We counted the number of calls to the `isCanonicalForm` (FFSM) and `isDuplicate` (gSpan) functions, allowing us to determine the sub-optimality of the graph codes. The functionality of these functions was confirmed by the authors of the binaries.

The tables provide a large amount of information. Most interesting is the good performance of gSpan's code. In terms of optimality, this code performs best in almost all cases. This result suggests that if we combine the code of gSpan with the evaluation mechanisms of the other algorithms, we might achieve similar runtimes as for these other algorithms. Comparing GASTON to FFSM, GASTON is less optimal than FFSM on cyclic graphs, but joins more efficiently. Only considering trees in the molecular databases, GASTON's code is more optimal. The good result of the HYBRIDTREEMINER on the A3 dataset can be explained by the fact that most frequent subtrees in this database turn out to be (single) centred; HYBRIDTREEMINER's tree code is optimized for such trees.

The most interesting observation is that there is no strong relation between the runtime experiments and the efficiencies of the graph codes. Then what does determine the efficiency of the graph miners?

More insights can be obtained from the experiments in Figure 5 and Figure 2. In Figure 5, we repeat the same mining experiments for increasingly larger datasets that are obtained by concatenating the same dataset multiple times. In general, the algorithms scale linearly. The constant in the linear regression relates to the operations that are independent of the dataset size, such as candidate generation. The time to count candidates is comparable in both the breadth-first and depth-first graph miners that do not use embedding lists. The effort to find all embeddings is apparently negligibly higher than the effort to find only one. On the Protein dataset, however, the scale-up is not linear. If we perform this experiment on two different computers, we can conclude that the reason is the very small size of this dataset. Without using embedding lists, the data fit all within the cache of the CPU, and the computation is extremely

fast. On a computer with less cache (like the Athlon) the advantage of in-cache computations is lost more quickly as the dataset grows larger.

Finally, the memory usage experiments show that the runtime differences between FFSM and gSpan are the result of a memory-CPU trade-off. Please note that the differences in memory usage are strongly influenced by the choices made in the binaries for the amount of bits used to represent node labels, etc., and are therefore only indicative.

4 Conclusions

In this study, we have confirmed that there is not a strong relation between the canonical string that is used and the runtime behavior of the algorithms. Polynomially computable codes, such as used in frequent tree mining, are often less efficient from a practical point of view. Of the many canonical codes that have been proposed, the DFS code that was introduced in gSpan perform consistently most well. Given its conceptual simplicity, this code should be preferred.

The differences between breadth-first and depth-first graph mining are significant. In most cases, the depth-first miners are faster and require less memory.

Many of the runtime differences between graph miners can be attributed to two aspects. First, there are the typical high performance computing issues, such as caching behavior; second, there is the memory usage-runtime trade-off.

References

1. R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328, 1996.
2. C. Borgelt and M. R. Berthold. Mining molecular fragments: Finding relevant substructures of molecules. In *ICDM*, pages 51–58, 2002.
3. Y. Chi, S. Nijssen, R. R. Muntz, and J. N. Kok. Frequent subtree mining—An overview. In *Fundamenta Informaticae*, volume 66, pages 161–198, 2005.
4. J. Huan, W. Wang, D. Bandyopadhyay, J. Snoeyink, J. Prins, and A. Tropsha. Mining family specific residue packing patterns from protein structure graphs. In *RECOMB*, pages 308–315, 2004.
5. J. Huan, W. Wang, and J. Prins. Efficient mining of frequent subgraphs in the presence of isomorphism. In *ICDM*, pages 549–552, 2003.
6. A. Inokuchi, T. Washio, K. Nishimura, and H. Motoda. A fast algorithm for mining frequent connected subgraphs. Technical Report RT0448, IBM Research, 2002.
7. M. Kuramochi and G. Karypis. Frequent subgraph discovery. In *ICDM*, pages 313–320, 2001.
8. S. Nijssen and J. N. Kok. A quickstart in frequent structure mining can make a difference. In *KDD*, pages 647–652, 2004.
9. M. Wörlein, T. Meinl, I. Fischer, and M. Philippsen. A quantitative comparison of the subgraph miners MoFa, gSpan, FFSM, and Gaston. In *PKDD*, LNCS 3721, pages 392–403, 2005.
10. X. Yan and J. Han. gSpan: Graph-based substructure pattern mining. In *ICDM*, pages 721–724, 2002.
11. M. J. Zaki. Efficiently mining frequent trees in a forest. In *KDD*, pages 71–80, 2002.