A Quantitative Comparison of the Subgraph Miners MoFa, gSpan, FFSM, and Gaston

Marc Wörlein, Thorsten Meinl, Ingrid Fischer, and Michael Philippsen

University of Erlangen-Nuremberg, Computer Science Department 2,
Martensstr. 3, 91058 Erlangen, Germany
simawoer@stud.informatik.uni-erlangen.de
{meinl, idfische, philippsen}@cs.fau.de

Abstract. Several new miners for frequent subgraphs have been published recently. Whereas new approaches are presented in detail, the quantitative evaluations are often of limited value: only the performance on a small set of graph databases is discussed and the new algorithm is often only compared to a single competitor based on an executable. It remains unclear, how the algorithms work on bigger/other graph databases and which of their distinctive features is best suited for which database. We have re-implemented the subgraph miners MoFa, gSpan, FFSM, and Gaston within a common code base and with the same level of programming expertise and optimization effort. This paper presents the results of a comparative benchmarking that ran the algorithms on a comprehensive set of graph databases.

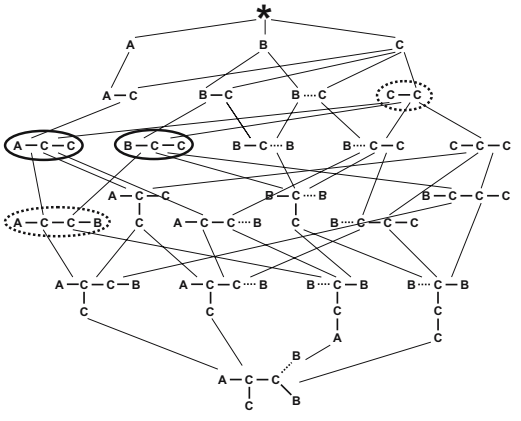
1 Introduction

Mining of frequent subgraphs in graph databases is an important challenge, especially in its most important application area "cheminformatics" where frequent molecular fragments help finding new drugs. Subgraph mining is more challenging than frequent itemset mining, since instead of bit vectors (i.e., frequent itemsets) arbitrary graph structures must be generated and matched. Since graph isomorphism testing is a hard problem [3], fragment miners are exponential in runtime and/or memory consumption. For a general overview see [1].

The naive fragment miner starts from the empty graph and recursively generates all possible refinements/fragment extensions by adding edges and nodes to already generated fragments. For each new possible fragment, it then performs a subgraph isomorphism test conceptually on each of the graphs in the graph database to determine if that fragment appears frequently (i.e., if it has enough *support*). Since a new refinement can only appear in those graphs that already hold the original fragment, the miner keeps appearance lists to restrict isomorphism testing to the graphs in these lists.

All possible graph fragments of a graph database form a lattice, see Fig. 1 for an example with just one graph. The empty graph $*$ is given at the top, the final graph at the bottom of the picture. During the search this lattice will be

pruned at infrequent fragments since their refinements will appear even more rarely.[1] Efficient fragment miners have to solve three main subproblems.



∗ is the empty fragment. Each graph is subgraph of all its descendants in the lattice. Subgraphs on one level have the same number of edges.

The dashed C-C-fragment is the common core of the two circled fragments. The new subgraph A-C-C-B can be generated by taking this core and adding the two edges A- and B- that only appear in one of the subgraphs.

Fig. 1. The complete subgraph lattice of the graph shown at the bottom

(A) Purposive refinement. Mining gets faster if instead of *all potential* refinements only those are created that might *appear* in the database. Two basic approaches exist: on the one hand two graphs can be joined from the previous level of the lattice that share a common core (Fig. 1). Although this may create some subgraphs that do not appear in the database, the appearance list of the refinement (i.e., the intersection of both preceeding appearance lists) is quickly checked. On the other hand, an existing subgraph can be extended by an edge and a node (or only an edge if cycles are closed). The node to be extended and the extension must be chosen carefully based on the appearance list.

(B) Efficient enumeration. Generated duplicates of the fragments have to be filtered out. One possibility are isomorphism tests on the database, which are costly. Hence, miners that generate less isomorphic refinements are faster. Using a canonical graph representation, some time is saved by detecting these duplicates before isomorphism testing on the database.

(C) Focused isomorphism testing. Known approaches either use efficient subgraph isomorphism tests, e.g. Nauty [3], or they trade time versus storage and keep embeddings. An embedding is a mapping of the nodes and edges of a fragment to the corresponding nodes and edges in the graph it occurs in. When counting the support of fragments, excessive isomorphism tests are necessary. It has to be clarified whether embedding lists lead to better results compared to isomorphism tests.

Early fragment miners generated refinements in a breadth first way, e.g., SUBDUE [6] (incomplete beam-search), AGM [7], and FSG [8]. Depth first search

[1] Similar to the *frequency antimonotone principle* in frequent itemset mining [4,5].

(dfs) approaches need less memory to store appearance lists because the number of lists that have to be stored in memory is proportional to the *depth* of the lattice (i.e. the size of the biggest graph) whereas it is proportional to its *width* (i.e. the maximal number of subgraphs in one level) in breadth first searches. The dfs-algorithms MoFa [9], gSpan [10], FFSM [11], and Gaston [12] attack the subproblems (A–C) quite differently. But since it is difficult to prove that a solution is better than another, the authors usually select a few databases and present benchmarks that demonstrate that their proposed solution works better than a competitor based on executables. Since different authors use different databases there is no general picture. It is unknown which of the solutions to (A–C) perform best under which conditions. To make things worse, sometimes only executables of the algorithms are available. Hence, measurements are skewed by use of different programming language and by exploitation of varying compiler optimization technology, etc.

In this paper we present an unbiased and detailed comparison of the four fragment miners MoFa, gSpan, FFSM and Gaston. We implemented them all from scratch using a common graph framework, i.e. all use the same graph data structures. In section 2 we briefly characterize how these algorithms solve subproblems (A–C). Section 3 contains the main body of this paper: the detailed experimental evaluation of the four contestants.

2 Distinctive Ideas of MoFa, gSpan, FFSM, and Gaston

All four fragment miners work on general, undirected graphs with labeled nodes and edges. They all are restricted to finding connected subgraphs and traverse the lattice as mentioned before in depth-first order.

MoFa (Molecule Fragment Miner, by Borgelt and Berthold in 2002 [9]) has been targeted towards molecular databases, but it can also be used for arbitrary graphs. MoFa stores all embeddings (both nodes and edges). Extension is restricted to those fragments, that actually appear in the database. Isomorphism tests in the database can cheaply be done by testing whether an embedding can be refined in the same way. MoFa uses a fragment-local numbering scheme to reduce the number of refinements generated from a fragment: MoFa counts the nodes of a fragment according to the sequence in which they have been added. When a fragment is extended at node n, later refinements may only occur at n or at nodes bigger than n. Moreover, all extensions that grow from the same node n are ordered according to increasing node and edge labels. Although this local ordering helps, MoFa still generates many isomorphic fragments and then uses standard isomorphism testing to prune duplicates.

gSpan (graph-based Substructure pattern, by Yan and Han in 2002 [10]) uses a canonical representation for graphs, called dfs-code. A dfs-traversal of a graph defines an order in which the edges are visited. The concatenation of edge representations in that order is the graph's dfs-code. Refinement generation is restricted by gSpan in two ways: First, fragments can only be extended at nodes that lie on the *rightmost path* of the dfs-tree. Secondly, fragment gen-

eration is guided by occurrence in the appearance lists. Since these two pruning rules cannot fully prevent isomorphic fragment generation, gSpan computes the canonical (lexicographically smallest) dfs-code for each refinement by means of a series of permutations. Refinements with non-minimal dfs-code can be pruned. Since instead of embeddings, gSpan only stores appearance lists for each fragment, explicit subgraph isomorphism testing must be done on all graphs in these appearance lists.

FFSM (Fast Frequent Subgraph Mining, by Huan, Wang, and Prins in 2003 [11]) represents graphs as triangle matrices (node labels on the diagonal, edge labels elsewhere). The matrix-code is the concatenation of all its entries, left to right and line by line. Based on lexicographic ordering, isomorphic graphs have the same canonical code (CAM – Canonical Adjacency Matrix). When FFSM joins two matrices of fragments to generate refinements, only at most two new structures result. FFSM also needs a restricted extension operation: a new edge-node pair may only be added to *the last node* of a CAM. After refinement generation, FFSM permutes matrix lines to check whether a generated matrix is in canonical form. If not, it can be pruned. FFSM stores embeddings to avoid explicit subgraph isomorphism testing. However, FFSM only stores the matching nodes, edges are ignored. This helps speeding up the join and extension operations since the embedding lists of new fragments can be calculated by set operations on the nodes.

Gaston (GrAph/Sequence/Tree extractiON, by Nijssen and Kok 2004 [12]) stores all embeddings, to generate only refinements that actually appear and to achieve fast isomorphism testing. The main insight is that there are efficient ways to enumerate paths and (non-cyclic) trees. By considering fragments that are paths or trees first, and by only proceeding to general graphs with cycles at the end, a large fraction of the work can be done efficiently. Only in that last phase, Gaston faces the NP-completeness of the subgraph isomorphism problem. Gaston defines a global order on cycle-closing edges and only generates those cycles that are "larger" than the last one. Duplicate detection is done in two phases: hashing to pre-sort and a graph isomorphism test for final duplicate detection.

For gSPan and MoFa several extensions exist that are described in section 3.5.

3 The Comparison

In the following sections we compare the four algorithms based on an analysis of the main computational parts on detailed experiments and on some special features of the algorithms.

3.1 Setup of Experiments

The tests were all done on 64bit Linux systems because of the huge memory requirements of some algorithms on the bigger datasets. Because of the lengthy tests we used several machines: Most experiments were run on a Dual-Itanium 2 PC running at 1.3 GHz with 10GB of RAM. Here we used IBM's Java Virtual

Machine (JVM) 1.4.2 because it produced the best runtime results for all algorithms.[2] The maximal heap space available to the JVM was set to 8GB to avoid swapping influences. For the memory tests we used the SUN JVM[3] as the IBM JVM showed garbage collector artifacts. The test on varying database sizes was carried out on an SGI Altix 3700 system[4] with Itanium 2 processors at 1.3 GHz. There only BEA Weblogic's JVM 1.4.2 was available.[5] The maximum heap was set to 14GB. Except for the database size experiments we aborted tests that ran longer than four hours.

We chose Java as programming language because this kept the implementation work at a bearable level. This may of course not lead to astonishingly fast execution times but the relative performance of the algorithms should not be affected significantly. Also the algorithms could be run on 64bit systems with no changes at all, which was important for some experiments.

Because the main application area of frequent subgraphs miners are molecular datasets, experiments were done on the databases described in Fig. 2. The IC93 dataset [13] is used to find out how the algorithms behave if the number of found fragments and the fragments itself get large. At a minimum support value of 4% the largest frequent fragment has 22 bonds, the number of fragments is 37,727. Typically all molecules of the HIV assay from 1999[6] are used for performance evaluations. The complete NCI database[7] is used to determine how the algorithms scale with increasing database size. The found fragments will very likely have no chemical meaning because the molecules in the dataset are very diverse.

Dataset	# molecules	average size # edges	largest molecule # edges	# node labels
IC93	1,283	28	81	10
HIV	42,689	27	234	58
NCI	237,771	22	276	78
PTE	337	26	213	66
CAN2DA99	32,557	28	236	69
HIV CA	423	42	196	21
HIV CM	1,083	34	234	27

Fig. 2. The molecular datasets used for testing and their sizes. There are always four edge labels in molecules.

Only to retest performance comparisons from [14,15] the PTE database[8], the DTP Human Tumor Cell Line Screen (dataset CAN2DA99)[9] and parts of the HIV dataset containing only the confirmend moderately active molecules (HIV CM) and the confirmed active molecules (HIV CA) were used. Except for the

[2] http://www-128.ibm.com/developerworks/java/jdk/index.html
[3] http://java.sun.com/
[4] http://www.sgi.com/products/servers/altix/index.html
[5] http://www.bea.com/framework.jsp/content/products/jrockit/
[6] http://dtp.nci.nih.gov/docs/aids/aids_data.html
[7] http://cactus.nci.nih.gov/ncidb2/download.html
[8] See [16] and http://web.comlab.ox.uk/oucl/research/areas/machlearn/PTE/. The dataset we used was provided by Siegfried Nijssen.
[9] http://dtp.nci.nih.gov/docs/cancer/cancer_data.html

CAN2DA99 these datasets are rather small compared to the complete HIV or the NCI dataset.

3.2 Hotspots

Section 2 has summarized how the four algorithms solve subproblems (A-C) and which tasks need to be done. Hence, we first show the runtime distribution by percentage for each task and each algorithm, a measurement, that was not done before in the literature. We used Quest's JProbe on a Profiler[10] on the PC with the SUN JVM for monitoring a run on the IC93 dataset with a minimum support of 5%, see Fig. 3. Using a profiler slows down the runtime a lot, so we took the biggest databases, that are manageable for this experiment: IC93 and HIV CA + HIV CM.

	IC93				**HIV CA+CM**			
	MoFa	gSpan	FFSM	Gaston	MoFa	gSpan	FFSM	Gaston
Duplicate filtering/pruning	11.3%	3.1%	0.1%	1.8%	12.3%	1.4%	0.2%	1.0%
Support computation	9.3%	62.9%	3.7%		9.6%	70.7%	3.3%	
Embedding list calculations	19.1%	-	60.4%	87.8%	18.1%	-	62.7%	95.9%
Extending of subgraphs	29.9%	17.3%	10.2%		31.1%	14.9%	8.1%	
Joining of subgraphs	-	-	0.1%	-	-	-	0.1%	-

Fig. 3. The table shows the main parts of the subgraph mining process and how much time (relative to the total runtime) each of the four algorithms spends for them

Filtering/pruning duplicates plays only a minor role in the whole subgraph mining process (0.1% - 12.3% of the total runtime). For MoFa, the time contains both the graph isomorphism tests for already generated graphs, and the deletion of extensions that do not comply with the structural pruning rules.

Support Computation or Embedding list calculation is where the algorithms spend most of their time. Using embedding lists (MoFa and FFSM) leads to low numbers in support computation, but calculating them is expensive. Although MoFa's 19.1% for IC93 seem faster than FFSM's 60.4% for IC93, both algorithms have spent about the same number of seconds in this task. If no embedding lists are used (gSpan), expensive subgraph isomorphism tests are necessary. For Gaston, it is impossible to separate runtimes for support computation, embedding list calculation and the extension of fragments. The 87.8% for IC93 includes Gaston's ability to uniquely generate paths and trees.

Extending or joining subgraphs takes about the same time in MoFa, gSpan and FFSM. Joining is only done by FFSM and is very cheap compared to the extension process.

The number for HIV CA + CM do not differ much from the numbers measured for IC93.

[10] `http://www.quest.com/jprobe/`

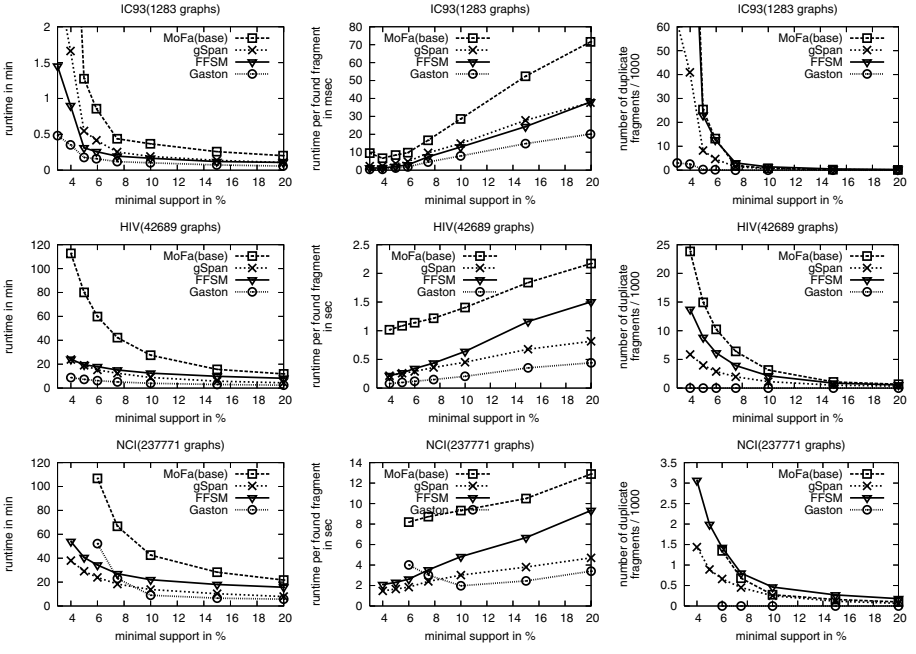


Fig. 4. Total runtime, runtime per found fragment, and the number of found duplicates for the three datasets IC93, HIV, and NCI measured for varying minimum support

3.3 Tests on Molecular Databases

First we retested successfully the results published in [14,15] (PC, IBM JVM) to prove that our implementations can compete with the original implementations and provide qualitatively the same results as given in the literature.

Second we recorded for each algorithm the total time needed at varying support values, the time needed per found frequent fragment and the number of found duplicates (which have to be filtered out by the algorithm in some way) for the IC93, HIV and NCI dataset. A comparison of MoFa[11], gSpan, FFSM and Gaston based on these databases was never published before. Figure 4 shows the results. The first obvious conclusion is the exponential rise in runtime with lower support values (left column). This is not very surprising as the number of fragments found also increases exponentially. Therefore, the runtime per found fragment (second column) is more interesting. For all datasets it shrinks with lower support values which can be explained by the cheaper frequency determination and calculation of embedding lists. The runtime per graph rises for Gaston on the NCI dataset for low support values. This is a memory problem as NCI is

[11] As for MoFa several extensions (closed fragments, ring mining, fuzzy chains) exist we did not use in our experiments, this algorithm is marked as *MoFa base* in the pictures, see section 3.5.

the largest database and Gaston needs the most memory of all algorithms, see Fig. 5 on the left.

There is a more or less clear runtime ranking among the four algorithms: MoFa is always the slowest. On the big datasets, FFSM is the second slowest algorithm, only on IC93 it is faster than gSpan. The result of this IC93 test equals the test result in [15]. The likely reason why gSpan is so slow on the IC93 dataset is the growing number of subgraph isomorphism tests gSpan has to do at these low support value (more than 37,000). All other algorithms use embedding list which speed up these tests especially for large fragments. On the large datasets, however, gSpan is faster than FFSM. Gaston is the fastest of all algorithms except at lower support values on the complete NCI dataset. A reason for this may be the amount of bookkeeping because of the large number of embeddings. Also a slowdown because of more frequent garbage collections may be the cause. The fragments found are however rather small so that gSpan gets by with cheap tests.

The number of found duplicates (right column) gives an insight into the power of the fragment refinement mechanisms. Also different pruning techniques minimize the number of duplicates, but as shown in section 3.2 they are not as relevant. Gaston wins as it does not produce duplicates for non-cyclic graphs. On the other hand FFSM's and MoFa's extension methods and pruning rules seem to be the weakest. For FFSM a look at relative time spend in filtering out these duplicates (see table 3), which is only 0.1% like Gaston, indicates that the canonical representation is very efficient.

Next the memory consumption at varying support values was recorded based on the SUN JVM. We frequently called the garbage collector and recorded the maximum heap size. This does not necessarily give the exact value of the memory consumption, but is a very good approximation. Because it slows down the runtime dramatically only the values for the HIV dataset were recorded. As can be seen in Fig. 5, gSpan needs the least memory as it does not use embedding lists. Although MoFa stores both edges and nodes in the embedding lists whereas FFSM only stores the nodes, MoFa still needs less memory. This is because MoFa only needs to store in each node of the search tree the embeddings of one subgraph, while in FFSM a search tree node consists of many subgraphs together with their embeddings. Gaston needs the most memory because embedding lists for a new fragment are built based on the embedding lists of the parent. Extensions to the parent's embedding list are stored with the children. Therefore, the size of the embedding lists does also depend on the number of children a fragment has. This results in the rise of the curve for low support values.

Finally the scalability of the algorithms for increasing database size was tested (Bea JVM, Altix), see Fig. 5, right. The complete NCI database was split into 119 pieces of 2,000 randomly selected molecules. For 5% support we have tested the performance for various subsets of the NCI database, each subset consisting of a growing number of these pieces. An obvious conclusion is, that all algorithms scale linearly with the database size, but with different factors. The surprising result is, that in this test Gaston is always slower than gSpan

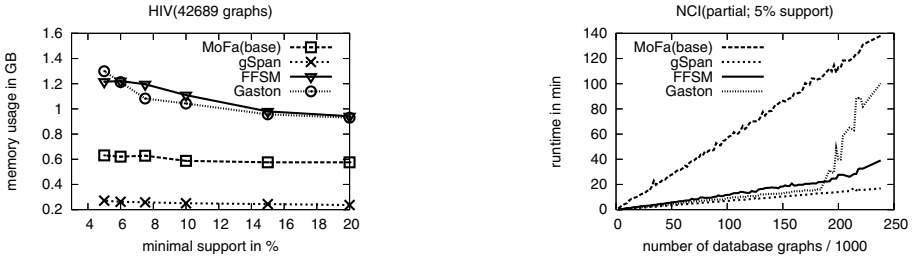


Fig. 5. Memory usage on the HIV database and the runtime in dependence of the database size on the complete NCI database

which was not the case in all other tests. We have performed some tests to be convinced that this is not an artefact of the different JVMs. Instead it seems that the uncommon memory architecture of the SGI Altix system penalizes memory intensive algorithms like Gaston. This also explains the raise in runtime for Gaston for larger databases. Testing Gaston with the IBM JVM on the Itanium on the same subsets of the NCI database did not result in this steep rise of the runtime curve.

3.4 Tests on Artificial Graph Databases

Real-world datasets are never "random". For example typical characteristics of molecular databases are certain distribution of labels, distinct cycles, and low node degrees. Although artificial generated graph databases seem to be a way to do general graph comparisons, there are several obstacles. The main problem is, that even with some fixed parameters randomly generated graph databases can be very different from each other and cause a wide spectrum of runtimes. By considering only the average or median of these results, no valid conclusion can be drawn.

Nevertheless we did some experiments with synthetical databases by our own graph generator. The most interesting test was done with graphs of varying edge densities as molecules mostly have a low edge density. We took a fixed number of 2000 graphs with an average of 50 nodes (ranging from 1 to 100) and 10 uniformly distributed different nodes and edge labels. Then we increased the number of edges in the graphs, starting at an edge density of 10% up to 40% (which means that the graphs contain e.g. $\frac{0.1 \cdot (\#nodes)^2}{2}$ edges). The minimum support was set to 10%. Figure 6 shows the runtime per graph in the left diagram and the total number of discovered fragments in the right one. Except MoFa, all other algorithms show a slight increase in the runtime which seems not to be strongly correlated to the number of found graphs. MoFa however shows a steep increase in the runtime. One reason is the number of discovered fragments: each new fragment has to be checked against all others to find out if it has already been found. The other algorithms rely on canonical representations and the test for duplicates is independent of the number of already discovered structures.

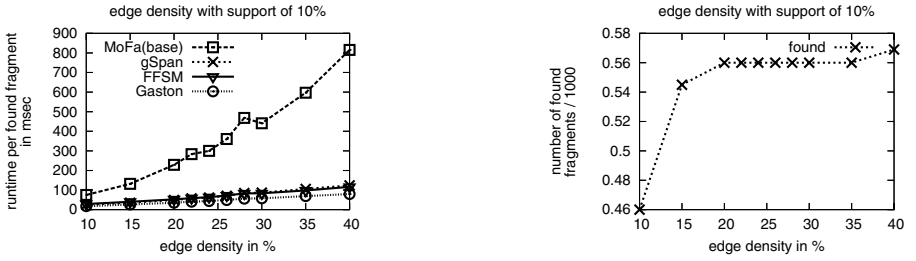


Fig. 6. Runtime and number of frequent subgraphs on synthetic datasets with varying edge density

3.5 Special Features and Possible Extensions

Some of the presented algorithms have special extensions not taken into account for this comparison, but which might improve the performance of the algorithms. One example are *closed subgraphs*. A subgraph is said to be closed if there is no bigger supergraph containing it that occurs in the same transactions of the database. Unclosed subgraphs can easily be filtered out after the search (and partly during the search), but for gSpan and MoFa there exist special extensions that prune branches of the search tree if only closed subgraphs are to be found [17,18]. This speeds up the search considerably (on some datasets for gSpan a speedup of a factor of 10 is reported, for MoFa the runtime is almost halved).

Another issue is the search in directed graphs. FFSM strongly relies on the triangle matrices, that cannot be used for directed graphs. Gaston's rules for uniquely constructing all paths and trees cannot be used for directed graphs without major changes. It is e.g. unclear how a spanning tree can be constructed in a directed graph. MoFa is capable of finding directed frequent subgraphs and also for gSpan only minor changes should be necessary.

Another topic of interest is the search for unconnected subgraphs. An example are molecules in which a certain part of the fragment must be present but the rest of the fragment is not known yet. MoFa can start the search with an unconnected *seed* instead of the empty graph. It is unclear how seeds can be combined with any of the other three algorithms.

For MoFa there also exists an extension for molecular databases that treats rings as single entities [19]. This not only dramatically reduces the number of search tree nodes but also avoids the reporting of fragments with open ring systems that normally make no sense for the biochemists. Another addition enables MoFa to find fragments with carbon chains of varying lengths [20], because this length is not important for biochemical reactions.

4 Conclusions

After re-implementing and testing four famous subgraph mining algorithms, the following conclusions can be drawn:

- Contrary to common belief embedding lists do not considerably speed up the search for frequent fragments. Even though gSpan does not use them, it is competitive to Gaston and FFSM. Only if the fragments become large (like in the IC93 dataset), gSpan falls off. On the other hand, embedding lists can cause problems if not enough memory is available or if the memory throughput is not high enough.
- The power of the pruning strategies to avoid duplicates is not the most important factor. The generation of candidates and support/embedding lists computations are much more critical.
- Using canonical representations for detecting duplicates is more efficient than doing explicit graph isomorphism test. Even better is the complete avoidance of duplicate fragment generation like Gaston does (at least for non-cyclic fragments).
- All algorithms scale linearly with the database size though with different factors.
- Depending on the used Java Virtual Machine results can sometimes differ. This problem can not be solved by the algorithms themself.
- Pure performance is not everything. Although MoFa is the slowest algorithm in all tests it offers much more functionality than the other miners for molecular databases and biochemical questions.

It is not yet clear, where the development of frequent subgraph mining will lead in the future. Possible directions are distributed or parallel search to overcome memory and performance limits. Exploring new application areas is expected to lead to new insights.

References

1. Fischer, I., Meinl, T.: Subgraph Mining. In Wang, J., ed.: Encyclopedia of Data Warehousing and Mining. Idea Group Reference, Hershey, PA, USA (2005)
2. Washio, T., Motoda, H.: State of the Art of Graph–based Data Mining. SIGKDD Explorations Newsletter **5** (2003) 59–68
3. McKay, B.: Practical graph isomorphism. Congressus Numerantium **30** (1981)
4. Agrawal, R., Imielinski, T., Swami, A.N.: Mining Association Rules between Sets of Items in Large Databases. In Buneman, P., Jajodia, S., eds.: Proc. 1993 ACM SIGMOD Int'l Conf. on Management of Data, Washington, D.C., USA, ACM Press (1993) 207–216
5. Zaki, M.J., Parthasarathy, S., Ogihara, M., Li, W.: New Algorithms for Fast Discovery of Association Rules. In Heckerman, D., Mannila, H., Pregibon, D., Uthurusamy, R., Park, M., eds.: In 3rd Int'l Conf. on Knowledge Discovery and Data Mining, AAAI Press (1997) 283–296
6. Cook, D.J., Holder, L.B.: Substructure Discovery Using Minimum Description Length and Background Knowledge. J. of Artificial Intelligence Research **1** (1994) 231–255
7. Inokuchi, A., Washio, T., Motoda, H.: An apriori-based algorithm for mining frequent substructures from graph data. In: PKDD '00: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery, London, UK, Springer (2000) 13–23

8. Kuramochi, M., Karypis, G.: Frequent subgraph discovery. In: Proceedings of the IEEE Intl. Conf. on Data Mining ICDM, Piscataway, NJ, USA, IEEE Press (2001) 313–320
9. Borgelt, C., Berthold, M.R.: Mining Molecular Fragments: Finding Relevant Substructures of Molecules. In: Proc. IEEE Int'l Conf. on Data Mining ICDM, Maebashi City, Japan (2002) 51–58
10. Yan, X., Han, J.: gSpan: Graph–Based Substructure Pattern Mining. In: Proc. IEEE Int'l Conf. on Data Mining ICDM, Maebashi City, Japan (2002) 721–723
11. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraphs in the presence of isomorphism. In: Proceedings of the 3rd IEEE Intl. Conf. on Data Mining ICDM, Piscataway, NJ, USA, IEEE Press (2003) 549–552
12. Nijssen, S., Kok, J.N.: Frequent Graph Mining and its Application to Molecular Databases. In Thissen, W., Wieringa, P., Pantic, M., Ludema, M., eds.: Proc. of the 2004 IEEE Conf. on Systems, Man and Cybernetics, SMC 2004, Den Haag, The Netherlands (2004) 4571 – 4577
13. Institute of Scientific Information, Inc. (ISI): Index chemicus - subset from 1993 (1993)
14. Nijssen, S., Kok, J.N.: A quickstart in frequent structure mining can make a difference. Technical report, Leiden Institute of Advanced Computer Science, Leiden University (2004)
15. Huan, J., Wang, W., Prins, J.: Efficient mining of frequent subgraphs in the presence of isomorphism. Technical report, Department of Computer Science at the University of North Carolina, Chapel Hill (2003)
16. Srinivasan, A., King, R.D., Muggleton, S.H., Sternberg, M.: The predictive toxicology evaluation challenge. In: Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97). Morgan-Kaufmann (1997) 1–6
17. Yan, X., Han, J.: Closegraph: Mining Closed Frequent Graph Patterns. In: Proc. of the 9th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining, Washington, DC, USA, ACM Press (2003) 286–295
18. Meinl, T., Borgelt, C., Berthold, M.R.: Discriminative Closed Fragment Mining and Pefect Extensions in MoFa. In Onaindia, E., Staab, S., eds.: STAIRS 2004 - Proc. of the Second Starting AI Researchers' Symp. Volume 109 of Frontiers in Artificial Intelligence and Applications., Valencia, Spain, IOS Press (2004) 3–14
19. Hofer, H., Borgelt, C., Berthold, M.R.: Large Scale Mining of Molecular Fragments with Wildcards. In: Advances in Intelligent Data Analysis. Number 2810 in Lecture Notes in Computer Science, Springer (2003) 380–389
20. Meinl, T., Borgelt, C., Berthold, M.R.: Mining Fragments with Fuzzy Chains in Molecular Databases. In Kok, J.N., Washio, T., eds.: Proc. of the Workshop W7 on Mining Graphs, Trees and Sequences (MGTS '04), Pisa, Italy (2004) 49–60