Finding maximal frequent subgraphs

Wagemans, J.M.M.

Award date:
2008

UNIVERSITY OF TECHNOLOGY EINDHOVEN
Department of Mathematics and Computer Science

Finding Maximal Frequent Subgraphs

By
ing. J.M.M. Wagemans

Supervisors:
prof. dr. Hans Zantema (TU/e, W&I)
dr. Dragan Bošnački (TU/e, BMT)

Eindhoven, April 2008

Abstract

We describe several improvements for the detection of maximal frequent subgraphs (MFS) amongst a number of graphs. Some improvements are (minor) modifications of the current state of the art frequent subgraph mining algorithm. Some other improvements have their basis in the reduction of the MFS problem to the problem of finding maximal frequent item sets (MFI). The latter includes the translation of the MFI problem to Pseudo-boolean constraints (PBC) and the use of very efficient MFI and PBC tools. Our experimental results will show that the use of a MFI tool, including appropriate postprocessing, is much more efficient than the current state of the art MFS tool.

1 Introduction

The problem of finding maximal frequent subgraphs (MFS) amongst a number of graphs is used in bioinformatics and systems biology to tackle one of their main challenges, the understanding of the structure of biological networks. Insight into the network organization and behavior helps to understand the biological processes inside the cell. With the increase of network related data, effective methods and models to analyze this data are needed.

A natural way to model biological networks is by means of graphs. The problem of finding frequent subgraphs in a collection of graphs representing biological networks was introduced by Koyutürk, Grama and Szpankowski [5]. In a sense, the problem is analogous to finding common subsequences in a collection of biological sequences that usually represent genes. Specialized tools for such sequence alignment, like CLUSTAL and BLAST, have been used routinely by researchers. The similarity between gene sequences detected in this way can be used to derive different kinds of structural, evolutionary, and functional information.

Similarly, detecting common parts in biological networks within one organism or across different organisms can provide insight into common motifs of cellular interactions, organization of functional modules, relationships and interactions between sequences, and patterns of gene regulation.

These graphs have enzymes as nodes, while the edges represent functional relations between the enzymes. In particular, there is an edge between two enzymes if the corresponding reactions that they catalyze are related. This means that the output of one reaction is an input for the other. Although some information is inevitably lost by such a representation, the benefit is that the graph mining problem is simplified significantly. At the same time, in practice, the enzyme graphs can be related back to the metabolic pathways in a straightforward way.

The choice to use enzyme graphs derived from KEGG [7] metabolic pathways is mainly motivated by the fact that we want to compare our approach with the existing state of the art method from [5]. This being said, our algorithms/methods are much more general and can be applied to other kinds of biological networks as well. Most can even be applied to similar problems from other research areas.

The problem of finding all maximal frequent subgraphs can be formulated as follows: given a collection of graphs and a natural number n, called threshold, find all subgraphs that are contained in at least n graphs of the collection. An additional condition is that the subgraphs are connected, i.e., between each two nodes a, b of such a subgraph there is a path either from a to b or vice versa. Moreover, we require that the graphs are maximal in the sense that they are not contained in any other frequent subgraph. The MFS problem and the MFI problem will be formalized in the preliminaries.

For our experiments we used real data (biological networks) extracted from the KEGG database [7]. Also we applied our improved algorithms to graphs extracted from metabolic networks as it was done in [5]. By doing so we established good basis for reliably comparing our algorithms with the current state of the art algorithm in [5]. The biomedical information and information about the KEGG database is taken from [5] and [7]. Although we use only KEGG data, other data can be used as well for the same purpose of finding maximal frequent subgraphs.

We will be using three different KEGG database versions, each consists of the data sets Alanine & Aspartate, Glutamate and Pyrimidine. We will be using the same KEGG database version of the data sets as were used in [5]. The author kindly provided us with these exact data sets. Since such version of the data sets become larger and more complex over time we will also be using a KEGG database version from 2004 and another version of January 2008.

Thesis layout. In Section 2 we will formally define both maximal frequent subgraphs (MFS) and maximal frequent item sets (MFI). Section 3 explains and compares the algorithm used in [5] and the algorithm used in [3]. Section 4 proves that we can transform all maximal frequent item sets (MFI) into all maximal frequent subgraphs (MFS) using the observation that we only need to extract all connected components from MFI and use the definition of MFS in Section 2.2. Section 5 discusses KEGG versions, implementations and run times.

Section 6 takes a closer look at the edges and items which remain after preprocessing the input data. In particular how these edges or items are represented in our respective implementations. Section 7 investigates the use of some form of probability, inspired by the fact that finding just one maximal frequent subgraph or maximal frequent item set is extremely fast. Section 8 improves the MFS algorithm by applying some methods from the MFI algorithm. Section 9 will show that we can add multiple edges (or items) at the same time as opposed to adding these one by one as the current algorithms do.

Section 10 describes a translation of the MFI problem into a Pseudo-boolean constrains (PBC) problem and uses the PBC tool "minisatplus" ([1]) obtained from [2] to solve our MFS problem. Section 11 will describe our ultimate solution, the usage of the external MFI tool "Afopt" from ([8]) in a black box way. The usage of an existing efficient MFI tool and appropriate postprocessing proves to be the best, fastest and by far most efficient solution for mining maximal frequent subgraphs with regards to the current state-of-the-art tool.

2 Preliminaries

2.1 Maximal Frequent Item sets

The problem of finding maximal frequent item sets can be described as follows: Let be given

- I is a finite set, its elements are called items.
- T is a set of subsets of I, $T \subseteq \mathcal{P}(I)$.
- A natural number m which is called the minimum support or threshold, $m \in \mathbb{N}$.

An item set $V \subseteq I$ is called a frequent item set if and only if

- $|\{X \in T | V \subseteq X\}| \geq m$.

An item set $V \subseteq I$ is called maximal frequent if and only if:

- V is frequent
- $(\forall W : W \subseteq I \wedge V \subseteq W \wedge V \neq W : W$ is not frequent)

We want all maximal frequent item sets (MFI) from T.

2.2 Maximal Frequent Subgraphs

The problem of finding maximal frequent subgraphs can be described as follows: Let be given

- E is a finite set, its elements are called edges.
- G is a set of subsets of E, $G \subseteq \mathcal{P}(E)$.
- $C(e, f)$ boolean value indicating whether or not e and f are connected, $e, f \in E$.
- A natural number m which is called the minimum support or threshold, $m \in \mathbb{N}$.

A set $V \subseteq E$ is called a connected subgraph or connected if and only if:

- $\emptyset, \{e\}$ are connected ($e \in E$).
- if X is connected and $X \subseteq E$ and $e \in X$ and $f \in E$ and $C(e, f) =$ true then $X \cup f$ is also connected.

A set $V \subseteq E$ is called a frequent subgraph if and only if

- V is connected.
- $|\{X \in G | V \subseteq X\}| \geq m$.

A set $V \subseteq E$ is called a maximal frequent subgraph if and only if:

- V is frequent (and connected)
- $(\forall W : W \subseteq E \wedge V \subseteq W \wedge V \neq W :$ W is not frequent)

We want all maximal frequent subgraphs (MFS) from G.

2.3 Usage of definitions

We will be using these definitions, note the details of the notation we use to denote the difference between one or all maximal frequent subgraphs or maximal frequent item sets:

- With "mfi" we denote *one* maximal frequent item set as defined in 2.1.
- With "MFI" we denote *all* maximal frequent item sets as defined in 2.1.
- With "mfs" we denote *one* maximal frequent subgraph as defined in 2.2.
- With "MFS" we denote *all* maximal frequent subgraphs as defined in 2.2.

2.4 A running example

Our running example will be used to explain a problem or running an algorithm throughout this thesis. Each graph may have the following nodes: a,b,c,d,e. An edge (or item) will be described by the source node name and destination node name. An edge from node 'a' to node 'b' or vice versa will be called edge 'ab'. For our examples we use the undirected graphs in Figure 1.

Graph1 has the following edges: $\{ab, ac, de\}$
Graph2 has the following edges: $\{ab, ac, bc, de\}$
Graph3 has the following edges: $\{ac, bc\}$
Graph4 has the following edges: $\{ac, de\}$

Figure 1: A graph collection example.

3 The algorithms

In order to actually find maximal frequent item sets or maximal frequent subgraphs one could implement the MFI algorithm from [3] or the MFS algorithm from [5]. First we give the respective algorithms, we discuss the differences between these algorithms.

3.1 The MFI-algorithm

In the algorithm for finding all mfi from [3], MFI is the set of so far found maximal frequent item sets. We assume some order on the items.

Figure 2: The MFI algorithm and FICombine by [3]

Algorithm *MFI-backtrack*(I_l, C_l, l)
1. **for each** $x \in C_l$
2. $I_{l+1} := I_l \cup x$
3. $P_{l+1} := \{y : y \in C_l \wedge y > x\}$
4. **if** $I_{l+1} \cup P_{l+1}$ has a superset in MFI
5. **return** //all subsequent branches pruned!
6. $C_{l+1} := FICombine(I_{l+1}, P_{l+1})$
7. **if** C_{l+1} is empty
8. **if** I_{l+1} has no superset in MFI
9. **MFI** := **MFI** $\cup I_{l+1}$
10. **else** MFI-backtrack$(I_{l+1}, C_{l+1}, l+1)$

Algorithm *FICombine*(I_{l+1}, P_{l+1})
1. $C := \emptyset$
2. **for each** $y \in P_{l+1}$
3. **if** $I_{l+1} \cup \{y\}$ is frequent
4. $C := C \cup \{y\}$
5. **return** C

The set I is the frequent item set that has been found so far. Initially I is empty. The set C contains the candidates or items that may end up in the set I. All items $x \in C$ are frequent by themselves. Initially all frequent items are in the set C_0. The integer l indicates the current level of 'exploration'. Initially $l = 0$. So MFI-backtrack is initially invoked with MFI-backtrack$(\emptyset, C_0, 0)$.

The set P contains all items from C, which have not been processed yet. We can skip all branches that can only lead to one or more (maximal) frequent item set(s) we have already found. So if $I_{l+1} \cup P_{l+1}$ has a superset in MFI, we can prune this branch. If not, we continue our search on this branch.

FICombine checks which items from P_{l+1} can be added to I_{l+1} such that the result is still frequent. All such items are then returned and form the new set C, i.e. $C_{l+1} = \{p \mid p \in P_{l+1} \wedge p \cup I_{l+1}$ is frequent$\}$. Each item which when added to I_{l+1} results in an infrequent item set is no longer relevant, since this item will always lead to an infrequent set on this branch forward.

An example. To illustrate this example even better, we will run the main algorithm on our example graphs from Figure 1. Note that we consider each graph to be a set and its edges as items.

From this example we have the following frequent items: $\{ab, ac, bc, de\}$. Initially I_0 is empty, C_0 contains all the items, $l := 0$. We will find maximal frequent item sets $\{ab, ac, de\}$ and $\{ac, bc\}$ for our running example.

Initial invocation should be MFI-Backtrack($\{\}, \{ab, ac, bc, de\}, 0$). Then the first item, 'ab' is added to I_0 resulting in I_1 in line 2. Then in line 3, P_1 is assigned the items beyond 'ab' in the set (assume the order of appearance in the set as the ordering): $P_1 := \{ac, bc, de\}$. The union of I_1 and P_1 (yields C_0 in this case) has no superset in the so far found $mfi's$ at line 4. Now we skip line 5 and continue with line 6 (FICombine). Running FICombine on I_1 and P_1 results in $C_1 := \{ac, de\}$. The empty test for C_1 in line 7 yields false, so we continue with the recursive call MFI-Backtrack($\{ab\}, \{ac, de\}, 1$) in line 10.

After going through lines 1 through 6 in this recursive call we end up with $I_2 = \{ab, ac\}, P_2 = \{de\}, C_2 = \{de\}$. Again C_2 is not empty at line 7, again we have a recursive call in line 10: MFI-Backtrack($\{ab, ac\}, \{de\}, 2$). Lines 1 through 6 yield $I_3 = \{ab, ac, de\}, P_3 = \emptyset, C_3 = \emptyset$. The test in line 7 yields true this time. The superset check for I_3 yields false, I_3 has no superset in $\emptyset$. Now we are allowed to add I_3 to MFI, our result under construction. Note that there are no more items to iterate through we go back to level 1.

On level 1, we continue with the next iteration of the loop. After line 3 we have $I_2 = \{ab, de\}, P_2 = \emptyset$. Now the test in line 4 yields true ($\{ab, de\} \subseteq \{ab, ac, de\}$) and we return to level 0, our initial level. So far we have found one mfi, namely $\{ab, ac, de\}$, we still need to iterate through $\{ac, bc, de\}$. We have discussed all relevant aspects of this algorithm we believe the reader can complete our example run. If you are finished you can check if you came up with the same run as in Appendix A.

3.2 The MFS-algorithm

An algorithm to find maximal frequent subgraphs is described in [5]. Here MFS is the set of all maximal frequent subgraphs found so far. E_k is the frequent subgraph found so far. This set may be expanded by edges from the set C_k which are not yet processed. Edges which already have been processed are in the set D. Also the edge currently being processed is in the set D.

Initially MinePathways is invoked using just one frequent edge. To that end we first filter out all non frequent edges from all the edges available. We do this to avoid unnecessary calculations, note that an infrequent edge can never contribute to some mfs. After running this algorithm on an edge we have all maximal frequent subgraphs which contain this particular edge.

Figure 3: The MFS algorithm by [5]

Algorithm *MinePathways*(MFS, E_k, C_k, D)
1. ▷ MFS: Set of maximal frequent subgraphs
2. ▷ E_k: Frequent subgraph with k edges
3. ▷ C_k: Set of candidate edges
4. ▷ D: Set of already visited edges
5. $ismaximal \leftarrow$ **true**
6. **for** all edges $e_i \in C_k$ **do**
7. $D \leftarrow D \cup \{e_i\}$
8. $E_{k+1} \leftarrow E_k \cup \{e_i\}$
9. **if** E_{k+1} is frequent **then**
10. $ismaximal \leftarrow$ **false**
11. $C_{k+1} \leftarrow (C_k \cup N(e_i)) \backslash D$
12. MinePathways$(MFS, E_{k+1}, C_{k+1}, D)$
13. **if** $ismaximal$ **then**
14. **if** E_k has no superset in MFS **then**
15. $MFS \leftarrow MFS \cup E_k$

In order to find all maximal frequent subgraphs, MinePathways must be run for all frequent edges. Each run MFS is updated as required. Also The function N simply returns the neighbors of an edge, or $N(e) = \{f | C(e, f)\}$. So initially the invocation of MinePathways would be MinePathways$(\emptyset, e_0, N(e_0), \{e_0\})$. The invocation of the next edge (e_1) would be MinePathways$(MFS, e_1, N(e_1), \{e_0, e_1\})$.

Note that the variable MFS is shared between all such runs of MinePathways. So actually one would need an additional loop which iterates through the edges, which can be represented using an array. Each iteration of this "super loop" takes care of calling MinePathways with the next edge. Also MFS is updated as required. We leave such super loop to the reader to implement if required.

An example. Again we use our running example from Figure 1. In this example we have the following edges: $\{ab, ac, bc, de\}$. Initially MFS is empty, E_0 has only one frequent edge, C_0 contains the neighbors of the edge in E_0 and D contains only the edge in E_0. We will find three maximal frequent subgraphs for our running example: $\{ab, ac\}$, $\{ac, bc\}$ and $\{de\}$.

First we run MinePathways on frequent edge 'ab', $N(ab) := \{ac, bc\}$ and $D := \{ab\}$. Initial invocation should be MinePathways$(\emptyset, \{ab\}, \{ac, bc\}, \{ab\})$. Line 6 selects the first candidate to be 'ac', we use the order of the elements in the set {ac,bc} for the ordering. Line 7 adds the candidate to consider to D, $D := \{ab, ac\}$. Then in line 8 the candidate is added to E_1, $E_1 = E_0 \cup \{ac\} = \emptyset \cup \{ac\} = \{ac\}$. Line 9 checks whether or not E_1 is frequent, it is frequent so we go to line 10 (the 'then clause'). Here we set $ismaximal := false$. Then in line 11 we add the neighbors of candidate ac to the new set of candidates minus the edges in D, $C_1 := (C_0 \cup N(ac)) \backslash D = (\{ac, bc\} \cup \{ab, bc\}) \backslash \{ab, ac\} = \{bc\}$.

Line 12 invokes MinePathways($\emptyset, \{ab, ac\}, \{bc\}, \{ab, ac\}$), thus going into recursion. Lines 6,7 and 8 then produce $D = \{ab, ac, bc\}$, $E_2 = \{ab, ac, bc\}$. In line 9 we discover that this set is not frequent. There are no more elements to iterate through so we exit the loop (we are still in the recursive step). Line 13 $ismaximal$ is $true$ so go to line 14 where we see that E_1 has no superset in $\emptyset$. Now line 15 is allowed to add E_1 to our temporary MFS-result MFS.

We get back from recursion to our initial invocation, lines 6, 7 and 8 yield $D = \{ab, ac, bc\}$, $E_1 = \{ab, bc\}$. Line 9, E_1 is not frequent. Also there are no more edges to iterate through so we go to line 13. Variable $ismaximal$ is $false$ so we exit MinePathways. Now we need to run MinePathways three more times:

- MinePathways($\emptyset, \{ac\}, \{bc\}, \{ab, ac\}$)
- MinePathways($\emptyset, \{bc\}, \emptyset, \{ab, ac, bc\}$)
- MinePathways($\emptyset, \{de\}, \emptyset, \{ab, ac, bc, de\}$)

We leave these invocations to the reader. In Appendix B you will find a more detailed and complete run of this MFS algorithm on our running example.

As you may have noticed the output from the MFI algorithm and the MFS algorithm on our running example differs. This will be explained in more detail in the next section and in Section 4 we will show that we can convert any MFI to MFS. In other words we can obtain all mfs from all mfi by applying some very fast post processing. You might want to try to obtain MFS from MFI for our example. You will see in Section 11 that this leads to a very fast and efficient algorithm to solve the MFS problem (at least for KEGG data sets).

3.3 The differences

We have now described two different algorithms with (possibly) different outputs with identical input data. As you may know the MFS algorithm is based upon the MFI problem in general as described in [5]. For this reason we will first compare both algorithms as described earlier. As you will see in Section 4 it does not matter that the MFI algorithm produces different output than the MFS algorithm with identical input data. However we do need appropriate post processing to get MFS from the MFI.

To illustrate the differences better we will rewrite both algorithms into similar pseudo code for a better comparison. We assume some order on the edges. Such order can be, for example, the order of the edges/items in an array or the order of the edges/items in a list, i.e. due to the data structure. For the pseudo code in this section we assume such data structure for the edges or items.

There are two major differences between the two algorithms. One is the connectedness constraint of the MFS algorithm. The MFI algorithm does not care about connectedness and runs immediately on all items as candidates. The MFS algorithm does care about connectedness by only allowing neighbors of a freshly added edge to also become candidates. So this connectedness constraint is of great influence of the result. If we would remove this constraint we would have the same result.

Algorithm *MFI-backtrack*(I_l, C_l)
1. ▷ MFI: Set of maximal frequent item sets
2. ▷ I_l: Frequent item set
3. ▷ C_l: Set of candidate items
4. ▷ P_l: Set of items not yet processed
5. **for each** $x \in C_l$
6. $\quad I_{l+1} := I \cup x$
7. $\quad P_{l+1} := \{y : y \in C_l \wedge y > x\}$
8. $\quad$ **if** $I_{l+1} \cup P_{l+1}$ has a superset in MFI
9. $\quad\quad$ **return** //all subsequent branches pruned!
10. $\quad C_{l+1} := FICombine(I_{l+1}, P_{l+1})$
11. $\quad$ **if** C_{l+1} is empty
12. $\quad\quad$ **if** I_{l+1} has no superset in MFI
13. $\quad\quad\quad$ **MFI** := **MFI** $\cup I_{l+1}$
14. $\quad$ **else** MFI-backtrack(I_{l+1}, C_{l+1})

Algorithm *MinePathways*(E_k, C_k)
1. ▷ MFS: Set of maximal frequent subgraphs
2. ▷ E_k: Frequent subgraph
3. ▷ C_k: Set of candidate edges
4. ▷ D: Set of already visited edges
5. $ismaximal$:= **true**
6. **for each** $x \in C_l$
7. $\quad E_{k+1} := E_k \cup \{x\}$
8. $\quad D := D \cup \{x\}$
9. $\quad$ **if** E_{k+1} is frequent **then**
10. $\quad\quad ismaximal$:= **false**
11. $\quad\quad C_{k+1} := (C_k \cup N(x)) \backslash D$
12. $\quad\quad$ MinePathways(E_{k+1}, C_{k+1})
13. **if** $ismaximal$ **then**
14. $\quad$ **if** E_k has no superset in MFS **then**
15. $\quad\quad MFS := MFS \cup E_k$

The second major difference is the pruning of branches which will not be resulting in a new maximal frequent item set in the MFI algorithm but in the MFS algorithm this is not possible. This difference is of great influence on the runtime, but not on the respective results. In Section 4, *MFS from MFI* we will show that we can obtain MFS from MFI, therefor enabling us to use MFI algorithms with appropriate postprocessing.

In the MFI algorithm we start with all items as candidates, therefore no new items are ever added to the candidates set. So in any step of the MFI algorithm no "new" candidates can be introduced which may or may not contribute to a maximal frequent item set. This information enables us to determine whether or not the union of the current frequent item set, I_i, and the set of remaining candidates P_i may or may not lead to *maximal* frequent item set. If it does not, we can prune the entire branch.

We cannot do this in the MFS algorithm because the (new) set of candidates may contain freshly introduced edges at any point in the MFS algorithm. In other words in the MFS algorithm we may have the situation that we can expand I_k with an edge, being I_{k+1} and that the union of I_{k+1} and C_k does not lead to any maximal frequent subgraph. However the neighbors of the edge just added to I_k are added to the set C_k (minus those in the set D), being C_{k+1}.

Obviously this may add an edge to the candidates which has not yet been in any candidates set. This freshly introduced candidate may even expand I_{k+1} further. If we would have pruned the branch, this particular candidate may never be added to I_{k+1}, thus possibly resulting in some faulty MFS. Since we may come across such fresh candidate anytime, we can add an edge to I_k, we cannot just prune branches based on this information in this MFS algorithm.

There are also some differences between the MFS algorithm and the actual tool, *pwmine* (also [5]), implementing this MFS algorithm. The MFS algorithm does not only output MFS but also in which graphs each mfs occurs as a subset. On the other hand the MFI algorithm does not provide the names of the sets in which each mfi occurs as a subset either, however for good comparison we did implemented this additional output. The MFI tool we used from [8] also does not output this additional information.

However adjusting both algorithms to also output the information in which graphs or sets each mfs or mfi occurs respectively is neither hard nor costly. It is actually quite efficient, since this information is already internally available, we just need to output these.

The information in which graphs/sets a subgraph/itemset occurs is be already present during the algorithm due to the frequency testing using the input sets names or graphs names. For each item/edge we know in which sets/graphs it occurs (using the names of these sets/graphs). Now if we have a itemset/subgraph we simply take the intersection of all these names and we know in which sets/graphs this itemset/subgraph occurs. By accumulating this information, this can be done efficiently.

4 MFS from MFI

The difference between the algorithms in Section 2.1, *MFI* and in Section 2.2, *MFS* is actually only the connectedness constraint with regards to the final result. So the MFI and the MFS algorithms are quite similar. We will see that we can use MFI algorithms, tools and programs which result in all maximal frequent item sets (MFI) order to find all maximal frequent subgraphs (MFS).

To use MFI algorithms we need a link between the edges/graphs and items/sets. We can consider the edges to be items and the graphs to be sets. More formally we have $I := E \wedge T := G$ in the MFI problem definition of Section 2.1 and E, G are from the MFS problem definition, Section 2.2. Now we can easily see from the MFS and MFI definitions in Section 2 that each frequent subgraph is also a frequent item set with the above substitutions.

In order to prove that we can transform such MFI to MFS, we will introduce the notion of connected component:

- A set $V \subseteq mfi$ is called a connected component if and only if:

  V is connected.

  $(\forall W : W \subseteq mfi \wedge V \subseteq W \wedge V \neq W :$ W is not connected$)$

It is not hard to see that any maximal frequent subgraph is also a connected component. A mfs is by definition connected, fulfilling the first requirement of our connected component definition. The second requirement can be fulfilled as well by substituting:

- W in the connected component definition by W from the mfs definition,
- V in the connected component definition by V from the mfs definition,
- mfi in the connected component definition by E from the mfs definition.

Now we can continue with proving the following theorem.

Theorem 1. *Each maximal frequent subgraph is a component of a maximal frequent item set.*

Proof. Let X be an arbitrary maximal frequent subgraph (mfs) in some set containing all maximal frequent subgraphs (MFS). Let the set of all maximal frequent item sets (MFI) be generated from the same input data as the MFS. Assume that no maximal frequent item set in this set MFI has X as a component.

As we already saw, each mfs is a connected component. By definition, each mfs is frequent. Also each mfs can also be a frequent item set. Furthermore, by definition, each frequent item set is either maximal or a subset of a maximal frequent item set. We have that X must also be a frequent item set and must therefor be maximal or be a subset of some maximal frequent item set. Hence the contradiction to our assumption, therefor each maximal frequent subgraph is indeed a component of a maximal frequent item set.

□

Theorem 2. *Let FS be an arbitrary set of frequent subgraphs, such that every maximal frequent subgraph is in FS. Then the set of maximal frequent subgraphs is equal to*

$$\{X \in FS \mid \text{ there is no } Y \in FS \text{ distinct from } X \text{ satisfying } X \subseteq Y\}.$$

Proof. Let X be a maximal frequent subgraph, then it is in FS. By definition there is no strictly greater frequent subgraph Y than X in FS.

Conversely let X be in the given set. Since it is in FS it is a frequent subgraph. Now construct Y from X by adding edges as long as Y is a frequent subgraph. Now by construction Y is a maximal frequent subgraph. So $Y \in FS$. Since X is in the given set, this is only possible if $Y = X$. Hence X is a maximal frequent subgraph. □

Since each mfs is a connected component of some mfi, we can obtain MFS from MFI by:

- Split each mfi in its connected components, we have then FS.
- Checking for each connected component whether or not it is a mfs.

In addition to the MFI algorithm described in Section 3.1 *The MFI-algorithm*, we would need the following additional algorithms to obtain MFS from MFI:

Algorithm *ObtainMFS*(MFI)
1. $MFS := \emptyset$
2. **for each** $mfi \in MFI$ **do**
3. **for each** edge $e \in mfi$ **do**
4. $mfi := mfi \backslash e$
5. $mfs := \{e\}$
6. $mfs := extractConnectedComponent(mfi, mfs)$
7. **for each** $e' \in mfs$ **do**
8. $mfi := mfi \backslash e'$
9. **for each** $mfs' \in MFS$ **do**
10. $supersetFound := false$
11. **if** $mfs' \subseteq mfs$ **do**
12. $MFS := MFS \backslash mfs'$
13. **if** $mfs \subseteq mfs'$ **do**
14. $supersetFound := true$
15. **if** $\neg supersetFound$ **do**
16. $MFS := MFS \cup mfs$
17. **return** MFS

Note that $e' \in mfi$ holds for each and every e' in line 8. Therefore these (e') will not be visited in any subsequent iteration of the loop in line 3.

Note that mfi might never become an empty set. This happens if mfi consists of a connected component and one or more edges which have no connection with this connected component. We choose to use the size of the extracted neighbors to guarantee termination. Eventually it won't be possible to extract neighbors anymore due to the simple fact that there are no neighbors left to extract.

Algorithm *extractConnectedComponent*(mfi, mfs)
1. $neighborLength := 0$
2. **for each** $e \in mfs$ **do**
3. $\quad neighbors := \emptyset$
4. $\quad$ **for each** $e' \in mfi$ **do**
5. $\quad\quad$ **if** $C(e, e')$ **do**
6. $\quad\quad\quad neighbors := neighbors \cup e'$
7. $\quad\quad\quad mfi := mfi \backslash e'$
8. $\quad$ **if** $|neighbors| \neq 0$ **do**
9. $\quad\quad mfs := mfs \cup neighbors$
10. $\quad\quad mfs := extractConnectedComponent(mfi, mfs)$
11. $\quad$ **return** mfs

We know that all maximal frequent subgraphs can be found using any item set algorithm and ObtainMFS. You can find and compare run times in Section 5, including for this implementation (Table 3).

Although our own implementation of this already greatly reduces run times, it is not the fastest possible way to mine MFS. However the fact that we can obtain all mfs from all mfi will prove to be even more useful in Section 11. There we reduce run times to a fraction of what we had before, i.e. with pwmine.

5 Implementations, KEGG versions & run times

First we will say a few things about the data we used, then we discuss our implementations and finally we compare run times for *pwmine* and our implementations.

5.1 KEGG data versions

Our real biological data is, like [5], taken from the Kyoto Encyclopedia of Genes and Genomes (KEGG) [7] database. Since more and more biological information is uncovered and added to KEGG, the data in this database is expanded accordingly, thus resulting in more complete and more complex data.

From this KEGG database one can freely obtain data about metabolic pathways, which we will be dealing with. We will be using data about the following metabolic pathways: Alanine & Aspartate, Glutamate and Pyrimidine. This we will call data sets or pathway. These three data sets together we will call a version of KEGG data.

Throughout this thesis we will be using three versions of KEGG data. Note that we are dealing with nine different data sets, three versions of Alanine & Aspartate, three versions of Glutamate and three versions of Pyrimidine. Each data set consists of a number of organisms for this type of pathway. As you will see later more data about more and more organisms is added over time to these data sets. Also each organisms will be represented by a graph.

The first version of KEGG data is the exact KEGG data version as was used by [5] in January 2004 which was kindly provided by Koyutürk. In this KEGG data version Alanine & Aspartate consisted of 156 organisms, Glutamate consisted of 155 organisms and Pyrimidine consisted of 156 organisms. The second KEGG data version is from later 2004 and the data sets of this KEGG data version were already expanded significantly. All three data sets of this KEGG data version consisted of 228 organisms each. The latest KEGG data version at 1st of January 2008 consisted of even much bigger data sets. Alanine & Aspartate and Glutamate each consisted of a staggering 698 organisms, Pyrimidine consisted of 697 organisms.

The information about each organism in each data set of any KEGG data version is stored in XML file format. Also these XML files contain much more information than we need to find maximal frequent subgraphs. So like p2g by Koyutürk et al in [5] we will also transform these XML to a new format with just the information we need to find maximal frequent subgraphs.

5.2 Our implementations

For easy comparison purposes throughout this thesis, we will be using the same file format as p2g uses since it is the only file format accepted by *pwmine*. The tool p2g converts the XML files for a data set to a new file format with extension ".gr", these we will be calling GR files or graph files. See [6] for more information on the tool *p2g* and *pwmine*, including input formats.

Although a lot of information is discarded, this is not a problem. An organism is represented by a XML file which is then reduced (converted) to a graph (GR) file. Since we use the same filename for the graph file as is used for the XML file, we can easily retrieve the full information of this organism. Note that only the filename extension is different (.gr instead of .xml). So if we have found a mfs then we can use the names of the graphs in which the mfs occurs to map these to their original XML files.

Like [5] this transformation of XML files to graph files needs to be done only once for each data set. In our case we needed to do this only nine times. Such conversions take only some seconds to complete from a few for the oldest KEGG data version to about 25 seconds for the latest KEGG data version. Therefor we will not consider these time costs any further in this thesis. Time costs are for calculating $MFS \backslash MFI$ (and some $pre \backslash post$ processing of graph files) only.

Our tool createfiles takes care of the conversion from XML files to graph (GR) files. This tool is available for usage. Another tool, comparefiles which we just mention, can compare two individual MFS files generated by *pwmine* or our implementations or two directories each containing such output files. In the latter case comparefiles each time compares two files from each directory with the same filename. Comparefiles compares both the mfs and whether or not a mfs occurs in the same graph in both files.

For our experiments we implemented both the MFS algorithm and MFI algorithm in Java. As we know from Section 4 we can use MFI algorithms and appropriate postprocessing to obtain MFS. Our MFI algorithm implementation does include this appropriate post processing. Both implementations take a set of graph files as their input, a threshold value (chosen by the user) and an output directory. Both implementations output MFS, given correct input, in a similar way as *pwmine* does.

We assume that our implementations are correct if and only if our implementations produce exactly the same MFS as the existing tool *pwmine* does for all data sets, threshold values and KEGG data version combinations. As we already described *pwmine* (from [5]), also outputs some extra data, namely the names of the graphs in which each $mfs \in MFS$ occurs.

With an additional parameter, *pwmine* has the ability to limit the size of each individual mfs. Now let us maximize this size to at most one, then in our running example a MFS with four elements $MFS = \{ab\}, \{ac\}, \{bc\}, \{de\}\}$ will be the result. Now assume we have for some MFS a mfs of size 16, we call A. If we would limit mfs size to 15 edges, the output MFS would contain all possible subsets of A of size 15 as a mfs. This is quite a lot mfs. Hence the size of MFS grows significantly by this.

Since all of such possible subsets must be found, it takes much longer to find MFS, which will not a correct MFS. This is due to the backtracking part of the MFS algorithm. For these reasons we did not implement this "feature" of *pwmine* in our implementations. When we run *pwmine* we make sure we provide a big enough number m for this parameter such that always largest $|mfs| < m$.

With our implementations and the data sets that were used in [5] which were kindly provided by the author, we have found exactly the same MFS for each data set and threshold as in [5] (for that KEGG data version). Also for the later 2004 and 2008 KEGG data versions our implementations found exactly the same MFS for each data set and threshold as *pwmine* did. From this we assume that our implementations are correct.

As mentioned before, the data sets for the three KEGG data versions (January 2004, later 2004 and January 2008) were increased as more information became uncovered about organisms already present in the data sets and information about organisms not yet in the data sets was added. From January 2004 to January 2008 the number of organisms for which information was available was increased from 155/156 to 697/698 respectively. Due to this increase *pwmine* takes a long time complete MFS calculation for lower threshold values.

5.3 Run times

In the tables below you will find run times (seconds) of different tools for all data sets of all KEGG data versions. The first column states the relative thresholds used. Then for each data set the run times for each KEGG data version is shown. The KEGG data versions are labeled "J2004" for January 2004, "l2004" for later 2004 and "2008" for the January 2008 version of KEGG data.

The run times for *pwmine* by [5] based on their algorithm, for such data can be found in Table 1. The run times for our implementation of the MFS algorithm by [5] are listed in Table 2 for all such data. Finally the run times for our implementation of the MFI algorithm by [3] for such data is shown in Table 3.

All time measurements were done on a Linux system with 4 cpu cores, of type "Dual Core AMD Opteron(tm) Processor 270" running at 1994.375 MHz with 16GB memory. For our measurements, we used the Linux "time" command. We have the total time by adding the user time (u) and the system time (s).

Table 1: Run times of *pwmine* on all data sets of all KEGG data versions

	Alanine & Aspartate			Glutamate			Pyrimidine		
T (%)	J2004	l2004	2008	J2004	l2004	2008	J2004	l2004	2008
5.0	>2000	>2000	>2000	>2000	>2000	>2000	>2000	>2000	>2000
7.50	11.46	>2000	>2000	1.488	>2000	>2000	10.33	>2000	>2000
10.00	2.494	>2000	>2000	0.384	>2000	>2000	0.306	>2000	>2000
12.50	0.715	>2000	>2000	0.127	>2000	>2000	0.09	>2000	>2000
15.00	0.115	>2000	>2000	0.028	>2000	>2000	0.026	>2000	>2000
17.50	0.061	>2000	>2000	0.016	>2000	>2000	0.013	>2000	>2000
20.00	0.024	>2000	>2000	0.01	>2000	>2000	0.007	>2000	>2000
30.00	0.008	>2000	498.6	0.008	>2000	>2000	0.007	>2000	>2000
40.00	0.007	>2000	13.35	0.008	>2000	>2000	0.007	>2000	1203
50.00	0.006	73.43	0.491	0.007	>2000	10.88	0.007	1893	12.22
60.00	0.007	1.324	0.033	0.009	38.37	0.124	0.007	15.67	0.348
70.00	0.006	0.044	0.027	0.008	1.122	0.028	0.006	0.28	0.039
80.00	0.012	0.008	0.015	0.008	0.018	0.027	0.007	0.024	0.032

Table 2: Run times of *normalMFS* on all data sets of all KEGG data versions

	Alanine & Aspartate			Glutamate			Pyrimidine		
T (%)	J2004	l2004	2008	J2004	l2004	2008	J2004	l2004	2008
5.0	>2000	>2000	>2000	>2000	>2000	>2000	>2000	>2000	>2000
7.5	>2000	>2000	>2000	247.0	>2000	>2000	>2000	>2000	>2000
10.0	213.7	>2000	>2000	37.16	>2000	>2000	73.8	>2000	>2000
12.5	45.26	>2000	>2000	10.49	>2000	>2000	19.52	>2000	>2000
15.0	4.488	>2000	>2000	2.26	>2000	>2000	3.512	>2000	>2000
17.5	2.829	>2000	>2000	1.813	>2000	>2000	1.595	>2000	>2000
20.0	2.245	>2000	>2000	1.393	>2000	>2000	1.011	>2000	>2000
30.0	0.822	>2000	>2001	0.91	>2000	>2000	0.697	>2000	>2000
40.0	0.672	>2000	1506	0.961	>2000	>2000	0.726	>2000	>2000
50.0	0.738	>2000	23.36	0.843	>2000	1544	0.656	>2000	>2000
60.0	0.689	317.9	1.916	0.954	>2000	6.591	0.666	>2000	30.3
70.0	0.742	5.312	1.44	0.798	418.5	1.733	0.6	178.9	2.243
80.0	0.664	0.924	1.041	0.821	1.135	1.047	0.665	1.093	1.323

Table 3: Run times of *fasterMFS* on all data sets of all KEGG data versions

	Alanine & Aspartate			Glutamate			Pyrimidine		
T (%)	J2004	l2004	2008	J2004	l2004	2008	J2004	l2004	2008
5.0	75.47	>2000	>2000	349.1	>2000	>2000	>2000	>2000	>2000
7.5	5.284	>2000	>2000	3.64	>2000	>2000	256.3	>2000	>2000
10.0	1.535	>2000	>2000	1.633	>2000	>2000	7.928	>2000	>2000
12.5	1.146	>2000	>2000	1.356	>2000	>2000	2.335	>2000	>2000
15.0	1.301	>2000	>2000	1.203	>2000	>2000	1.741	>2000	>2000
17.5	0.935	>2000	>2000	1.051	>2000	>2000	1.161	>2000	>2000
20.0	0.894	>2000	>2000	1.036	>2000	>2000	0.899	>2000	>2000
30.0	0.81	>2000	312.1	0.917	>2000	>2000	0.759	>2000	>2000
40.0	0.668	357.1	11.69	0.81	1054	298.5	0.656	>2000	>2000
50.0	0.712	15.35	3.626	0.921	1.203	8.294	0.663	>2000	323.3
60.0	0.667	1.564	2.373	0.845	3.002	2.304	0.67	837.9	9.895
70.0	0.708	1.016	1.919	0.812	6.354	2.144	0.651	17.44	2.538
80.0	0.704	0.882	1.183	0.797	1.054	1.121	0.678	1.071	1.582

From Table 1 it appears that *pwmine* is much faster at threshold 30% for the "2008" version of Alanine & Aspartate than it was for the smaller set "l2004". This happened because of the big difference in number of organisms between these two versions of Alanine & Aspartate. If we take 30% of 698 we have absolute value 209. Taking 30% of 228 results in 68 absolute value. Resulting in 43 frequent edges and 55 frequent edges respectively. The more edges to process the more time it takes, generally. This also holds for such differences in the other tables.

As you can see, our own simple implementation of the MFS algorithm *normalMFS* is slower than *pwmine*. One reason for this is very likely the programming language. We programmed in Java while *pwmine* was programmed in C. Even so, our (Java) implementation of the MFI algorithm *fasterMFS* is already (significantly) faster for lower threshold values than *pwmine*, for many runs.

6 Sorting the edges/items

We implemented both the MFS and the MFI algorithms in order to understand the problem better and more importantly to find ways to speed them up. As we already saw, *fasterMFS* (our implementation of the MFI algorithm) is already more promising than our MFS algorithm implementation, *normalMFS*.

Internally, for both our implementations, we represent the set of edges and the set of items in an array. The order we assumed earlier in Section 3 is now the order the edges or items in this array. We will see that this order does matter with regard to the run time of our implementations. Obviously this does not affect the result, the MFS. The result remains correct in the sense that it is identical to the result of *pwmine* where *pwmine* could produce one in a reasonable amount of time.

Both our implementations process the input graph (GR) files such that we have an array of unique edges or unique items. Then the number of graphs/sets each unique edge or unique item occurs in, must be greater or equal to the threshold value. If it is then we keep the edge or item, if it is not then we throw away such edge or item. In this way only unique *frequent* edges or items remain. This is called frequency testing.

Frequency testing can also be performed for a set of edges or items. We simply take the intersection of the set of graph names in which each edge of this set occurs in. Then this new set of graph names must be greater or equal to the threshold value. If it is, then the set of edges is frequent. If it is not, then the set edges is not frequent. This is how we have implemented frequency testing.

As said before by accumulating the intersected set of graph names, this can be done efficiently. We only need to intersect the current accumulated set of graph names in which the current frequent subgraph occurs and the set of graph names of the newly added edge. then compare the new set of graph names with the threshold value. Similarly this can be done for items as well. After this preprocessing we end up with an array of unique frequent edges or items. Then we continue with either *normalMFS* or *fasterMFS* respectively.

6.1 MFS program

The most computation time (nearly all) is taken up by the function intersectGraphnames, which is the intersection part of our frequency testing. This function is responsible for nearly all computation time, so it would make sense to analyze this function in more detail. The resulting set is then tested if is greater or equal to the threshold value, the other part of frequency testing.

Our function intersectGraphnames takes an accumulation list of graph names in which the current subgraph occurs (for the current subgraph we have: accGN) and a list of graph names belonging to the added edge (gn). It returns a new accumulation list (newGN) which is the intersection of the two input lists. I.e. $newGN := accGN \cap gn$.

Then in our code, a simple check whether or not this new subgraph is frequent is performed. If ($\#newGN >= threshold$) then we continue with the added edge (this is in the pseudo code the line "... is frequent" $\equiv$ true) else we continue without the added edge or backtrack (in the pseudo code "... is frequent" $\equiv$ false).

To understand better what is happening we will run the program 1000 times, each time we create a random order of the edges in the all unique frequent edges array. For this test we take threshold value 168 for data set pyrimidine of the later 2004 version of KEGG data. We choose this because our implementation takes a reasonable amount of time to complete and still results in multiple maximal frequent subgraphs of reasonable suited sizes.

After the preprocessing, only 35 unique frequent edges remain. We want to establish a relation between the input (#edges) and the total runtime. The total runtime of these 1000 times random order of edges was 16 hours and 45 minutes on a Intel Pentium 4, dual 2.6 GHz system with 512 MB memory, running Windows XP SP2.

The results revealed a relation between the order of the edges in the edges array and #calls to intersectGraphnames (and thus the runtime). Obviously the fewer calls to intersectGraphnames the faster the program. This test supports that. Here are the results:

- The least number of times the function was called was 131,145 times.
- The average number of times the function was called was 683,093 times.
- The largest number of times the function was called was 3,029,219 times.
- For 676 random orders, the number of times the function was called was less than 750,000 times.
- For 180 random orders, the number of times the function was called was less than 350,000 times.
- For 29 random orders, the number of times the function was called was less than 200,000 times.

Since the order is quite relevant for the runtime we will try to find a way to sort the edges such that the number of times the function intersectGraphnames will be called is minimized. To accomplish this in a fast and cheap way we identify three attributes that are easy and cheap to obtain:

- The number of neighbors of an edge.
- The number of graphs in which an edge occurs.
- The product of #neighbors and #graphs.

Now we sort the edges according to the three attributes described above. We both sort ascending and descending. We run this for the same data set on the same machine as our 1000 times random order test. The results are shown in Table 4.

Table 4: Sort results for #Calls to intersectGraphnames

Ascending	#Calls to intersectGraphnames:
#Neighbors	169,904
#Graphs	343,358
#Neighbors * #Graphs	108,721
Descending	#Calls to intersectGraphnames:
#Neighbors	668,659
#Graphs	538,945
#Neighbors * #Graphs	1,295,170

As you can see in the Table 4, the product of #neighbors and #graphs results in few calls to the function intersectGraphnames. This is even better than the least number of calls after the random 1000 orders of edges in array test. All other results, result in far less calls than the largest one from the random 1000 orders of edges in array test.

Table 5 shows the sort methods used on other threshold values. Let n denote $\#neighbors$, g denotes $\#graphs$, "asc" is short for ascending and "desc" is short for descending. The values in the table are #calls to intersectGraphnames.

Table 5: Later 2004 KEGG data, Pyrimidine

Threshold	asc n*g	asc n	asc g	desc n*g	desc n	desc g
165	678,536	1,020,629	3,476,781	10,896,303	10,720,088	5,746,211
168	108,721	169,904	343,358	1,295,170	668,659	538,945
170	5,057	5,043	6,806	9,975	14,758	8,927
175	1,831	1,500	1,987	5,649	7,534	3.623
180	2,231	2,252	2,811	4,123	4,048	2,771
185	1,396	1,682	2,271	3,234	3,107	2,031
190	107	109	128	142	139	161

As you can see most sort methods' results are worse than sorting by $n * g$. The next best sorting method is sorting ascending on n. This sort method is in most cases comparable to sorting ascending on $n * g$. In some cases, for sorting ascending on n, the #calls to intersectGraphnames is less than the #calls to intersectGraphnames for sorting ascending on $n * g$. But still sorting ascending on $n * g$ is in most cases either comparable or better than sorting ascending on n.

Although not shown, something similar holds for the other data sets, for the same KEGG data version as well as other KEGG data versions. Generally, sorting ascending on n is sometimes a bit better than sorting ascending on $n * g$. But in those cases sorting ascending on $n * g$ is not much slower.

So even if this easy and cheap sorting method does not lead to the least possible number of calls to intersectGraphnames, it is very effective to minimize the number of calls to intersectGraphnames. And therefore keeping the time cost low. In our example this method is 6 times faster than the average of 1000 runs on a random order of the edges!

Here's an overview how we can make the program run 6 times faster than the average of the 1000 runs test:

- We know or determine the number of neighbors of each edge: $\#neighbors$.
- We know or determine the number of graphs the edge occurs in: $\#graphs$.
- We calculate for each edge: $x := \#neighbors * \#graphs$.
- We sort the edges ascending on x.

It can happen that we encounter for some edge $x := \#neighbors * \#graphs$ and for some other edge $y := \#neighbors * \#graphs$ with $x = y$, i.e. the number to sort on is equal for two edges. In that case it can matter for the run time which edge comes first in the unique frequent edges array. However we were unable to find a sorting method for these cases to guarantee the lowest possible times the function intersectGraphnames is called. But by sorting these edges ascending lexicographically on the source vertice name and if this is equal too, then sort lexicographically ascending on the destination vertices, seems to result in lower counts of times the function intersectGraphnames is called.

6.2 MFI program

After investigating this we wonder whether or not the order of the items array also matters for *fasterMFS*. Given the above results and the "slight" difference in the algorithms we would think so. Since we work with graphs and edges to be considered sets and items respectively, we can use the same attributes as for *normalMFS* in this case.

We use the same intersectGraphnames function the same programming language, the same machine etc etc. Instead of *normalMFS* we now use *fasterMFS*. The order of the items in the items array is randomly changed using the same function as for the test with *normalMFS*. Again we let the program run fully for 1000 times, each time we use a randomly chosen order of the items array. We use the same threshold and data set as before. It took only 2 hours and 39 minutes to complete. Here are the results:

- The least number of times the function intersectGraphnames was called was 12,171 times.
- The average number of times the function intersectGraphnames was called was 96,926 times.
- The largest number of times the function intersectGraphnames was called was 314,547 times.
- For 775 random orders, the number of times the function intersectGraphnames was called was less than 125,000 times.
- For 150 random orders, the number of times the function intersectGraphnames was called was less than 50,000 times.
- For 16 random orders, the number of times the function intersectGraphnames was called was less than 25,000 times.

Table 6: Sort results for #Calls to intersectGraphnames

Ascending	#Calls to intersectGraphnames:
#Neighbors	20,599
#Graphs	39,396
#Neighbors * #Graphs	15,908
Descending	#Calls to intersectGraphnames:
#Neighbors	136,867
#Graphs	224,720
#Neighbors * #Graphs	121,589

Now we sort the items as we did for *normalMFS*. We both sort ascending and descending in Table 6. Remember that we can do this here, because we are really working with edges and graphs represented as items and sets respectively.

As you can see there is a possible faster way to get this done. Since *fasterMFS* does not care about connectedness and thus neighbors, it is not so strange that there might be a better order of the items array.

One possible order of the items array can be obtained by calculating for each item an array of all other items for which the combination of the two items is NOT frequent, i.e. for each item x we want $\{y|y \text{ is item} \wedge \{x, y\} \text{ is not frequent}\}$. Then sort descending and ascending on this number in the Table below.

Ascending	#Calls to intersectGraphnames:
#infrequents	287,631
Descending	#Calls to intersectGraphnames:
#infrequents	7,154

No matter how promising this looks, when we try threshold 160 it takes 42,905 seconds and 470,100 calls to intersectGraphnames. When we sort as we do for edges it takes only 8,234 seconds and only 94,667 calls to intersectGraphnames. Unfortunately this is the case for most lower thresholds and therefor useless. Sorting ascending is also useless, since every tested case takes far more time and calls to intersectGraphnames than the MFS-sorting method. Sorting descending on $\#infrequents * \#graphs$ is faster than sorting descending on only $\#infrequents$ but still much slower than what we already had.

Unfortunately as far as we currently know, there is no other easy straightforward way to speed *fasterMFS* up by sorting the items. Our method, sorting the items, obviously won't work for "real" items. However there are some indications that it can be done much faster, we just need to find it. So that leaves us, for now, with the same sorting method as we do for the *normalMFS* to speed it up.

However by using existing MFI programs, which are very fast, the sort order is probably not much of a speed up. Some existing MFI programs are part of a contest so they have to be very fast and efficient, much more so than a standard implementation in Java. We will investigate the use of existing MFI tools in Section 11. It will prove to be very useful.

7 Random runs

Another attempt to speed the program(s) up is inspired by the observation that finding just one mfs or mfi is always very fast. Even in the respective (slow) Java implementations it takes only a few milliseconds to find just one mfs or one mfi. So what if we can find all mfs by running the program 1000 times and stop whenever we have found just one mfs? So let us find out.

7.1 Until one mfs is found

Again we run *normalMFS* 1000 times on the same data set as in Section 6, this we call "one run". We don't run the program fully, but we stop when we have found just one mfs. Again we randomly change the order of the edges, just like we did for the sorting test.

This time we will use threshold 150, not 168. All tests so far show that until now all mfs are found with this method on threshold 168, using about 30 seconds average. Using threshold value 150 results each tested time in failure with regard to finding all mfs. Each time, after a run, one particular mfs is not found.

The total average time of three runs was about 36 seconds. Each run 1000 mfs were found. Naturally this is what we would expect. Running this algorithm, *normalMFS*, once but fully takes 470.184 seconds, but we do find all mfs. When we run *fasterMFS* once but fully it takes just 21.155 seconds. Naturally we also find all mfs.

Also some mfs are found very often while others are found only few times each run. See Table 7 for the full overview of some tests. Some mfs have the same number of edges, but the edges are not identical. "Edge" denotes the edges which make the difference between same sized mfs. The next three columns state how many times a mfs has been found for each of the three runs. Finally the last column states the average number of times each mfs has been found for these three runs.

Although *fasterMFS* is already faster than the 1000 times random permutation of the all unique frequent edges array, this does show that this probalistic method does have advantages when compared to running *normalMFS* fully. When one is not necessarily interested in **all** mfs, but just many this method could be applied. However is is faster to use *fasterMFS* for the obvious reasons.

7.2 Until one mfi is found

We can also run *fasterMFS*, our implementation of the MFI algorithm, on 1000 random permutations (runs) of the items array. Each time we have found one mfi we stop and continue with the next permutation or run. We will be using the same data set as before (pyrimidine of the later 2004 KEGG data version) and threshold value 150. We saw earlier that each mfs is contained at least once in the set of **all** mfi. Since we stop when we have found just one mfi, we wonder how many mfs these mfi contain, if any at all. Do we find all mfs after 1000 runs?

A single mfi may consist of edges which are not connected. Therefor such mfi may contain just a set of edges without having any mfs as a subset. However, the connected components are, in such cases, strict subset(s) of some mfs, but **no** such connected component is a mfs. Obviously we can't just extract the biggest connected component from a single mfi and call it a mfs.

Our mfs-check includes not only extracting connected components and dismissing all (strict) subsets of these like we would do otherwise. It also consists of trying to add a neighboring (connected!) edge to each such resulting connected component. Then if such an edge can be added to the connected component, it is obviously not maximal thus it cannot be a mfs. When no such edge can be added, then the connected component is indeed maximal and therefor also mfs.

Running the program on 1000 permutations of the items array for the same data set on threshold 168 takes only 9.048 seconds. All mfs are found easily and fast. When we try threshold 150, the average runtime is only 12.374 seconds and all mfs are found in all three runs. As said before running *normalMFS*, once but fully takes 470.184 seconds. Running *fasterMFS* once but fully it takes just 21.155 seconds. In both cases we find all mfs.

In Table 8 you can see how many times which mfs is found. Again, "Edge" denotes the edges which make the difference between same sized mfs. The next three columns state how many times a mfs has been found for each of the three runs. Finally the last column states the average number of times each mfs has been found for these three runs.

Table 7: #times each mfs is found in one run using probalisticMFS

mfs with # edges	Edge	Run 01	Run 02	Run 03	Average
19		227	220	235	227.33
17		14	5	9	9.33
16	13 to 18	271	294	285	283.33
16	55 to 57	23	27	30	26.67
14		58	63	71	64.00
11	14 to 20	11	5	4	6.67
11	12 to 102	0	0	0	0.00
10		15	9	24	9.33
7		75	101	66	80.67
4		57	42	48	49.00
3	75 to 80	67	46	61	61.33
3	65 to 67	67	66	48	60.33
2	76 to 77	44	38	38	40.00
2	64 to 67	29	37	38	34.67
2	25 to 100	18	22	19	19.67
1		24	24	24	24.00

For this data set and threshold value, running *fasterMFS* on just 100 permutations of the items array gives us sometimes all mfs, taking only 2.786 seconds to complete. This method seems to be better and faster than running *fasterMFS* fully. But it also does not and cannot guarantee to find all mfs. As you will see in Section 11, there is a much faster way to find all mfs. This method is even so fast that there will not be any need for probalistic algorithms like the ones in this Section.

Table 8: #times each mfs is found in one run using probalisticMFI

mfs with # edges	Edge	Run 01	Run 02	Run 03	Average
19		167	126	138	144
17		21	28	28	26
16	13 to 18	69	67	75	70
16	55 to 57	155	166	172	164
14		31	20	33	28
11	14 to 20	30	22	19	24
11	12 to 102	18	12	13	14
10		10	4	9	8
7		19	26	20	22
4		24	18	12	21
3	75 to 80	12	8	12	11
3	65 to 67	228	238	253	240
2	76 to 77	242	258	283	261
2	64 to 67	41	28	26	32
2	25 to 100	15	23	21	20
1		39	32	17	29
	Total files:	792	749	764	768
	Total mfs:	1121	1076	1130	1109

8 Using FICombine for MFS algorithm

As you can see in the MFI algorithm in Subsection 3.1, there are some nice things that have a positive influence on the run time. Unfortunately we cannot try all of them in the MFS algorithm. One such example is pruning branches that do not result in some new mfs. Simply because for the MFI algorithm, the set of candidates is always fixed, while the set of candidates for the MFS algorithm is very dynamic, depending on the neighbors of some newly added edge.

Another method used in this MFI algorithm, we really can use for the MFS algorithm, is passing on only candidates which are frequent in combination with the so far found subgraph. In the MFI algorithm (Section 3.1), FICombine is used for this. We will improve the algorithm in Figure 3 by using our adopted version of FICombine: FSCombine.

Figure 4: Improved MFS algorithm and FSCombine

Algorithm *MinePathwaysFSC*(MFS, E_k, C_k, D)
1. ▷ MFS: Set of maximal frequent subgraphs
2. ▷ E_k: Frequent subgraph with k edges
3. ▷ C_k: Set of candidate edges, which are frequent in combination with E_k
4. ▷ D: Set of already visited edges
5. $ismaximal \leftarrow$ **true**
6. **for** all edges $e_i \in C_k$ **do**
7. $\quad D \leftarrow D \cup \{e_i\}$
8. $\quad E_{k+1} \leftarrow E_k \cup \{e_i\}$
9. $\quad ismaximal \leftarrow$ **false**
10. $\quad PC := (C_k \cup N(e_i)) \backslash D$
11. $\quad C_{k+1} := FSCombine(E_{k+1}, PC)$
12. $\quad$ MinePathways$(MFS, E_{k+1}, C_{k+1}, D)$
13. **if** *ismaximal* **then**
14. $\quad$ **if** E_k has no superset in MFS **then**
15. $\quad\quad MFS \leftarrow MFS \cup E_k$

Algorithm *FSCombine*(E_{k+1}, PC)
1. $C := \emptyset$
2. **for each** $y \in PC$
3. $\quad$ **if** $E_{k+1} \cup \{y\}$ is frequent
4. $\quad\quad C := C \cup \{y\}$
5. **return** C

Note again that this algorithm must be run for all edges with each edge for E_0 and their neighbors for C_0, until all edges are done. Now we still run the algorithm for all edges with each edge for E_0 but this time we only use its neighbors which are frequent in combination with E_0 for C_0. Also during the algorithm when we create the new candidate set, we also only allow candidates which are frequent in combination with E_k. The frequency check in line 9 of the original MFS algorithm (Section 3.2) is removed, since all candidates are now already frequent in combination with E_k.

In the tables below you will find an overview of the run times of this new program, *combineMFS*. All measurements were done on the later 2004 KEGG data version for consistency reasons. Also the tables show both the run times for *combineMFS* with sorting the unique frequent edges array as proposed in 6. But also for the same unique frequent edges array unsorted. The sorting is denoted in the columns. The column "*normalMFS*" denotes the run times for our original implementation of the MFS algorithm in 3, without any sorting for the same data set.

Furthermore the tables also show the number of times the function responsible for the intersection part of the frequency testing, intersectGraphnames, is called for both sort options. "T (%)" denotes the relative threshold value and T (abs) denotes the absolute threshold value. All run times are given in seconds.

Table 9: Run times on data set Alanine & Aspartate

		normalMFS	Not sorted		Sorted	
T (%)	T (abs)	run time	run time	#times ig	run time	#times ig
50.0	114	>2000	681.6	9,008,354	644.4	8,662,178
60.0	137	317.9	17.91	225,901	19.11	230,314
70.0	160	5.312	2.042	7,269	1.805	7,505
73.7	168	2.376	1.313	2,272	1.276	2,518
80.0	182	0.924	0.865	144	0.814	145

Table 10: Run times on data set Glutamate

		normalMFS	Not sorted		Sorted	
T (%)	T (abs)	run time	run time	#times ig	run time	#times ig
50.0	114	>2000	>2000	N/A	>2000	N/A
60.0	137	>2000	142.0	2,051,976	179.7	2,148,630
70.0	160	418.5	33.38	558,962	32.06	589,639
73.7	168	120.3	16.76	229,183	16.72	231,628
80.0	182	1.135	0.968	562	0.904	625

Table 11: Run times on data set Pyrimidine

		normalMFS	Not sorted		Sorted	
T (%)	T (abs)	run time	run time	#times ig	run time	#times ig
50.0	114	>2000	>2000	N/A	>2000	N/A
60.0	137	>2000	268.4	4,316,923	399.3	5,274,088
70.0	160	178.9	3.608	47,63	4.971	47,593
73.7	168	7.840	1.864	9,960	2.051	13,411
80.0	182	1.093	0.891	399	0.932	449

Although these improvements greatly shorten the runtime, this method is not the fastest method as you will see in Section 11. However this method is even successful for threshold values, *normalMFS* needs a lot of time for. Also sorting the edges does not really matter for this method.

9 Adding multiple edges/items at once

We have found another optimalization that significantly speeds up both algorithms, up to 75 times faster! The idea behind this optimalization is quite simple. Why only add one edge/item at a time when you can add more edges/items at once? This Section will show you that we do can add multiple edges or items at once instead of one by one.

We are allowed to add an edge or item if and only if the set of edges or items remains frequent after adding this edge or item. For the set of edges it must also hold that this set remains connected as well after adding the edge. Frequency testing is, in our implementations, done through the graph names or set names as described before. Also this frequency testing takes up almost all running time.

Each edge/item occurs in at least one graph/set. Suppose we have an item x that occurs in the sets A and B, an item y that occurs in the sets A, B and C and finally an item z which occurs in sets A, B, C and D. Now, if we can add item x to an item set, this will be because of x's occurrence in $A \vee B$ (obviously, for threshold values 1 or 2). So any other item that occurs in at least sets $A \wedge B$ can also be added, without having to check for frequency. Hence we can also add y and z to the same item set(s) we can add x to. Similarly this also holds for edges and subgraphs, provided that the new subgraph remains connected.

If we consider the graph/set names an edge/item occurs in as sets and a function $C(x)$ which returns this set of names. Then we could say that if we can add an edge/item x, we can also add $\{y | y \text{ is edge or item} \wedge C(x) \subseteq C(y)\}$. So instead of adding just one edge or item we can add a whole set of edges or items at once without the need to check for frequency!

Now how can we modify the program to add multiple edges/items at once without the need to check for frequency? First in the pre processing, we add to each edge/item a list of all other edges/items which occur in at least the graphs/sets this edge/item does. This is easy and cheap to do. Then during either program, when we can add an edge/item we simply also add all the edges/items in that list without having to check for frequency.

In our running example from Figure 1, we can also add multiple edges or items. Whenever we can add item ab, we can also add item ac. If we can add item bc, we can also add item ac. Similarly this holds for edges, because in this case the only edge to add without an additional frequency check is connected to the edge which was tested for frequency (ab or bc). Such edges or items added without an additional frequency check will be labeled "processed" from that point forward, thus saving a few recursive calls and some frequency testing.

This causes only little change in the MFI algorithms as one can see in Figure 5. The MFS algorithm however need to include connectedness checking. The new MFS algorithm is displayed in Figure 6. The functions getFreeItems and getFreeEdges return all items/edges we can add without the need to check for frequency for a certain item or edge.

Figure 5: MFI algorithm with adding multiple items at once

Algorithm *MFI-backtrack*(I_l, C_l)
1. ▷ MFI: Set of maximal frequent item sets
2. ▷ I_l: Frequent item set
3. ▷ C_l: Set of candidate items
4. ▷ P_l: Set of items not yet processed
5. **for each** $x \in C_l$
6. $\quad freeItems := getFreeItems(x)$
7. $\quad I_{l+1} := I \cup x \cup freeItems$
8. $\quad P_{l+1} := \{y : y \in C_l \wedge y > x\}$
9. $\quad$ **if** $I_{l+1} \cup P_{l+1}$ has a superset in MFI
10. $\quad\quad$ **return** //all subsequent branches pruned!
11. $\quad C_{l+1} := FICombine(I_{l+1}, P_{l+1})$
12. $\quad$ **if** C_{l+1} is empty
13. $\quad\quad$ **if** I_{l+1} has no superset in MFI
14. $\quad\quad\quad$ **MFI** := **MFI** $\cup I_{l+1}$
15. $\quad$ **else** MFI-backtrack(I_{l+1}, C_{l+1})

Figure 6: MFS algorithm with adding multiple edges at once

Algorithm *MinePathways*(E_k, C_k)
1. ▷ MFS: Set of maximal frequent subgraphs
2. ▷ E_k: Frequent subgraph
3. ▷ C_k: Set of candidate edges
4. ▷ D: Set of already visited edges
5. $ismaximal :=$ **true**
6. **for each** $x \in C_l$
7. $\quad E_{k+1} := E_k \cup \{x\} \cup freeEdges$
8. $\quad D := D \cup \{x\}$
9. $\quad$ **if** E_{k+1} is frequent **then**
10. $\quad\quad ismaximal :=$ **false**
11. $\quad\quad freeEdges := getFreeEdges(x)$
12. $\quad\quad$ **for each** $y \in freeEdges$ **do**
13. $\quad\quad\quad$ **if** $E_{k+1} \cup y$ is connected
14. $\quad\quad\quad\quad E_{k+1} := E_{k+1} \cup \{y\}$
15. $\quad\quad\quad\quad D := D \cup \{y\}$
16. $\quad\quad C_{k+1} := (C_k \cup N(x)) \backslash D$
17. $\quad\quad$ MinePathways(E_{k+1}, C_{k+1})
18. **if** $ismaximal$ **then**
19. $\quad$ **if** E_k has no superset in MFS **then**
20. $\quad\quad MFS := MFS \cup E_k$

We run these new algorithms on the KEGG data version of later 2004. In the tables below you will find the run times and the number of times frequency testing was required for the new MFS program. This new MFS program is based on the new algorithm in Figure 6. The run times are denoted "run time" and #frequency tests is denoted "#times ig". The relative threshold is denoted "T(%)" and the absolute threshold is denoted "T(abs)".

The column after "T(abs)" shows run times from Table 2 for the used data set. The next two columns show results when we do not sort the edges. The next two columns show results when the edges are sorted as suggested by Section 6.

Table 12: Run times on data set Alanine & Aspartate

		normalMFS	Not sorted		Sorted	
T (%)	T (abs)	run time	run time	#times ig	run time	#times ig
40.0	91	357.1	11.07	361,805	1.036	1,434
50.0	114	15.35	1.802	18,486	1.002	1,382
60.0	137	1.564	1.147	2,185	0.975	1,180
70.0	160	1.016	0.98	445	0.953	282
73.7	168	0.811	0.886	234	0.842	150
80.0	182	0.882	0.772	52	0.833	48

Table 13: Run times on data set Glutamate

		normalMFS	Not sorted		Sorted	
T (%)	T (abs)	run time	run time	#times ig	run time	#times ig
40.0	91	1054	45.08	1,585,310	71.24	2,681,165
50.0	114	1.203	1.3	2,289	1.316	5,806
60.0	137	3.002	1.724	17,769	1.297	6,149
70.0	160	6.354	2.443	38,386	4.337	98,268
73.7	168	1.302	1.178	3,703	1.286	4,593
80.0	182	1.054	0.974	511	1.076	672

Table 14: Run times on data set Pyrimidine

		normalMFS	Not sorted		Sorted	
T (%)	T (abs)	run time	run time	#times ig	run time	#times ig
40.0	91	>2000	31.77	685,524	63.15	1,577,769
50.0	114	>2000	20.31	506,173	20.28	555,687
60.0	137	837.9	2.492	28,735	3.5	59,172
70.0	160	17.44	1.491	6,127	1.566	6,753
73.7	168	2.519	1.23	3,337	1.356	2,310
80.0	182	1.071	0.992	562	0.914	399

As you can see this improves the run times greatly. For the new MFI program based on the algorithm in Figure 5 we have similar results, although generally even lower run times and fewer frequency tests are needed. This may much faster, but we do better as you will see in Section 11.

10 Pseudo boolean constraints

Although we got some nice results, we still wonder if we can find MFS even faster. We are going to try a completely new method to find MFS. This method is called "*Pseudo boolean constraints*". First we introduce some terminology, translate our problem into Pseudo boolean constraints and illustrate this with an example. Then we apply this to real data and evaluate run times.

SAT is the problem of assigning truth values to the variables of a boolean formula such that the formula yields true or determining that this is not possible. From the SAT point of view pseudo-boolean constraints (PBC) can be seen as a *generalization* of clauses. A PBC is an inequality on a linear combination of boolean variables: $C_0p_0 + C_1p_1 + ... + C_{n-1}p_{n-1} \geq C_n$, where the variables $p_i \in \{0, 1\}$ and C_i an integer value. If all constraints C_i are 1, then the PBC is equivalent to a standard SAT clause. PBC is also often referred to as *0-1 integer linear programming* (ILP).

In a PBC a true literal is interpreted as the value 1, a false literal as 0. Note that $\neg x = (1 - x)$. A constant C_i is *activated* under a (partial) assignment if its corresponding literal p_i is assigned $TRUE$. A PBC is *satisfied* under an assignment if the inequality holds with the sum of the activated values for the left-hand-side (LHS) of the inequality.

We will translate our problem into a PBC problem, to find a suitable assignment to the resulting formula, we will use *minisatplus* [1], an efficient tool for solving PBC problems. Actually this tool translates PBCs into SAT and then the SAT problem is solved. The ability to count (the constants C), gives us an opportunity to translate our problem into PBC.

In order to use *minisatplus*, we need to translate our problem to a PBC problem. We have already proven that we can use item set mining and some postprocessing to obtain MFS. Since item set mining is a simpler problem we will translate this to a PBC problem instead of translating subgraph mining to PBC problem.

When we take a good look at the item set mining problem we can observe the following:

- Each item set is a subset of all the items.
- Each item is in at least one item set.
- For any (maximal) frequent item set it must hold that this (maximal) frequent item set is a subset of at least n item sets, where n is the threshold value.

We want to find a maximal frequent item set V. For that we define some variables.

- Each set $X \in T$ will be assigned an unique variable, A_X such that $A_X \Leftrightarrow V \subseteq X$.
- Each item $y \in I$ will be assigned an unique variable, b_y such that $b_y \Leftrightarrow y \subseteq V$.

We consider an example with 4 sets and items {item1,item2,item3,item4}:

- Set 1: {item1}
- Set 2: {item2}
- Set 3: {item2,item3}
- Set 4: {item1,item4}

Given this example, we will have variable A_1 to denote whether or not some item set occurs in set 1. Variables A_2, A_3, A_4 denote this for set 2, set 3 and set 4 respectively. Variables b_1, b_2, b_3, b_4 denote whether or not item1, item2, item3 and item4 occur in item set V respectively.

A frequent item set occurs in at least n sets, where n is the threshold specified by the user. So the inequality will be of type "$\geq$", and C_nn is the threshold value. Now we count the sets in which an item set occurs by letting each such set add 1 to the total number of sets this item set occurs in. We let the literals for such sets yield true. If a set literal (C_i) yields true, this set should contribute 1. So we have $C_i = 1$, for all i, $0 < i < n$. The first PBC will look like $+1 * A_1 + 1 * A_2 + 1 * A_3 + 1 * A_4 \geq 2$ for sets A_1, A_2, A_3, A_4 and threshold value 2. This we will call rule1.

Now we get to the more difficult part: how to represent which items are in which sets. First we make some observations:

- If an item is in some set, all sets NOT containing this item cannot contribute to pass the threshold value.
- We have at least one item in our item set.
- An item either is in a contributing set or not.

In terms of PBC we have that either a set literal A_i yields true if this set is contributing OR at least one item literal(s) not in A_i yield true, but not both! We will be using the connective "xor" which is an exclusive or, indicating that precisely one operand can be true, to express this better.

So we allow any number of items not in A_i to be true xor A_i to be true. We translate this into PBC by assigning $C_i = m$ for the set A_i, with m = total number of items and $C_i = 1$ for all the items not in A_i. The total of the LHS must be equal or less than m. Obviously we need one PBC for each set, we will call this rule 2. This leads us to our next 4 PBC's:

- $+1 * b_2 + 1 * b_3 + 1 * b_4 + 4 * A_1 \leq 4$
- $+1 * b_1 + 1 * b_3 + 1 * b_4 + 4 * A_2 \leq 4$
- $+1 * b_1 + 1 * b_4 + 4 * A_3 \leq 4$
- $+1 * b_2 + 1 * b_3 + 4 * A_4 \leq 4$

Although by using the above we do obtain correct results, adding the following will speed it up significantly. Let D_i denote all items not in A_i and E is the set of all items. Obviously we have that $D_i \cup A_i$ yields E. Rule 2 guarantees at most one option can be true (either at least one item not in A_i yields true xor A_i yields true). Now we will say at least one option must be true. This will be rule 3.

We can easily translate this into PBC. The type of the inequality will be of type "$\geq$", with $C_n = 1$. Now we allow one or more item literals b_i not in A_i and/or A_i to yield true. We may add the following 4 lines to our formula:

- $+1 * b_2 + 1 * b_3 + 1 * b_4 + 1 * A_1 \geq 1$
- $+1 * b_1 + 1 * b_3 + 1 * b_4 + 1 * A_2 \geq 1$
- $+1 * b_1 + 1 * b_4 + 1 * A_3 \geq 1$
- $+1 * b_2 + 1 * b_3 + 1 * A_4 \geq 1$

Now we need to tell *minisatplus* what we want it to do. *Minisatplus* minimizes assignments over specified variables (literals). So if we minimize over "false" values for these literals we obtain the maximal "true" values for these variables. We want as much as possible items in our maximal frequent item set. So our initial part of the formula would be: "$min : -1 * b1\ -1 * b2\ -1 * b3\ -1 * b4;$". By adding the negation of each found mfi to the formula, *minisatplus* finds another (smaller!) mfi if it exists. The union of all found mfi yield MFI. Running "ObtainMFS" on the found MFI gives us our desired MFS.

Remember the definition of MFI, in Section 2.1. Any item set resulting from *minisatplus* is by the definition of our formula (rule 1) always frequent. Also by minimizing over negated item-variables, *minisatplus* tries to find an assignment such that the least number of item-variables yield $false$. Since these item-variables can only yield $\{true, false\}$, *minisatplus* assigns the most item-variables with $true$. In this way find a maximal frequent item set, each time. Adding to the formula that we do not want to find this set in the again, we obtain some other maximal frequent item set, if there are more mfi to find. Doing this until we cannot find more maximal frequent item sets, we have found all maximal frequent item sets or MFI.

With our ingredients we have the following formula for our example:
$min : -1 * b1\ -1 * b2\ -1 * b3\ -1 * b4;$
$+1 * A_1 + 1 * A_2 + 1 * A_3 + 1 * A_4 \geq 2$
$+1 * b_2 + 1 * b_3 + 1 * b_4 + 4 * A_1 \leq 4$
$+1 * b_1 + 1 * b_3 + 1 * b_4 + 4 * A_2 \leq 4$
$+1 * b_1 + 1 * b_4 + 4 * A_3 \leq 4$
$+1 * b_2 + 1 * b_3 + 1 * b_4 + 4 * A_4 \leq 4$
$+1 * b_2 + 1 * b_3 + 1 * b_4 + 1 * A_1 \geq 1$
$+1 * b_1 + 1 * b_3 + 1 * b_4 + 1 * A_2 \geq 1$
$+1 * b_1 + 1 * b_4 + 1 * A_3 \geq 1$
$+1 * b_2 + 1 * b_3 + 1 * A_4 \geq 1$

This formula yields the following part of *minisatplus* output after the first run:
$b_1\ -b_2\ -b_3\ -b_4\ A_1\ -A_2\ -A_3\ A_4$

This means that we have found a mfi containing only item b_1 which occurs in sets A_1 and A_4. Now we add $+1 * b_i$ for each item b_i occurring in this found item set to the LHS of a new PBC. The RHS will the the total number of items, h, in this item set minus 1, the inequality will be "$\leq$". Note that we found the biggest possible item set, another item set with exactly h items is possible but not with exactly the same items. We have $h = 1$, the RHS value is $h - 1 = 0$. So we add to the formula the following line: $+1 * b_1 \leq 0$.

The output after running this new formula indeed results in a new mfi, just $\{b_2\}$. The complete output is: $-b_1\ b_2\ -b_3\ -b_4\ -A_1\ A_2\ A_3\ -A_4$
When we add its negation $(+1 * b_2 \leq 0)$ to our formula, the next run yields the verdict "*UNSATISFIABLE*". Meaning that no more mfi exist. In other words we have already found all mfi and thus MFI.

A short overview of our ingredients:

- Minimize over negated item variables.
- For each item set A_i, $0 < i \leq k$, k is #sets, we add $+1 * A_i$ to the LHS. The RHS is the threshold value given by the user. The inequality is "$\geq$".
- For each set A_i we add $+1 * b_j$ for each item $b_j \notin A_i$ and $+m * A_i$ for the LHS, where m is the number of unique items. For the RHS we simply take m. The inequality type is then "$\leq$".
- For each set A_i we add $+1 * A_i$ and $+1 * b_j$ for each item $b_j \notin A_i$ for the LHS and just the value of l for the RHS. The inequality type is then "$\geq$".
- If an mfi was found: For each item $b_i \in mfi$ add $+1 * b_i$ to the LHS, let $h = |mfi|$ then the RHS is the value $h - 1$. The inequality is "$\leq$".

Table 15 shows some run times in seconds of both *pwmine* in the (*pwmine* column) and our tool, the "FindMFSMP" column. Tests are done for KEGG data version later 2004. The thresholds can be found in the first two columns, relative threshold value and absolute threshold value respectively.

Our tool not only processes the input graphs and creates the formula but it also runs *minisatplus* as a black box tool. Our tool then captures the *minisatplus* output. If necessary our tool also adds the negation of a found mfi to the formula and runs *minisatplus* again until all mfi are found. All the found mfi together is MFI. This MFI is then translated to MFS using ObtainMFS.

Table 15 shows some run times for data set pyrimidine of KEGG data version later 2004 at various threshold values in seconds. All tested threshold/dataset combinations resulted in the same output files as *pwmine* generates for the same threshold/dataset combinations. Also the formula used, is a straight forward translation of our MFI problem. Therefor we assume this technique to be correct. More threshold/dataset combinations were tested than shown in Table 15.

Table 15: Run times of minisatplus for Pyrimidine

T (%)	T (abs)	*pwmine*	FindMFSMP
70.0	160	1	> 2000
80.0	182	0	493.321
85.0	194	0	17.867
90.0	205	0	1.309

Unfortunately using *minisatplus* is not the fastest way. This technique is not very fast, also not for other data sets. Although this technique is not the winning one, it does show that this problem can also be solved by a completely other technique. We have shown a way to encode the MFI problem into a PBC problem which can be solved by *minisatplus*. In the next Section, the fastest and best solution is described.

11 Use blackbox MFI tool

We created our tool *FindMFS* and it turns out to be the fastest method currently available as far as we know. *FindMFS* computes MFS by using the tool "Afopt" by [8] in a black box way. In other words, we just use that tool. This method proved to be very successful such that a paper about this method is accepted at Bioinformatics Research and Development (BIRD) 2008 [4].

So far, we have two main methods: the MFS-algorithm from Figure 3 and the MFI-algorithm from Figure 2. Additionally we have found many variations to them. Using the MFI algorithm is very promising, as we already saw in Table 3. So let us try to use an existing MFI mining tool in a black box way. We choose to use "Afopt" [3] for this purpose, a tool resulted from a frequent item set mining workshop [4] at data mining conference [5].

Basically we do some pre processing to provide for the input for Afopt and we do the post processing on its result, i.e. converting the resulting MFI to MFS. Our tool *FindMFS* takes care of the whole process, it also calls *Afopt* and captures its output for the postprocessing.

The pre processing is quite simple. Our tool, *createFiles*, transforms the original XML files (from the KEGG database) or the GR files previously created by *p2g*, [5] into the right input format for *Afopt*. This input for *Afopt* will be written to a file called "transactions.txt". This needs to be done only once for each data set for each KEGG data version.

The input format for *Afopt* is quite simple. *Afopt* needs an input file with on each line of that file exactly one item set. On each such line, each item is represented by a non-negative number. These numbers must be in ascending order.

We can achieve this by representing each edge by its index number from the unique **all** edges array. Note that we DO NOT filter out infrequent edges during this pre processing. We create one line with ascending non-negative integers for each graph. First for each graph, we "replace" each edge by its index number of this all unique edges array. Then we sort these numbers in ascending order. We then write these lines to a file, called "transactions.txt". We write this edges array to a file for translating the numbers back to edges, when *Afopt* is done.

The converting from the KEGG XML files is done in a similar way as *p2g* converts KEGG XML to graph files. For each data set, a so called "map" file is available. Such "map" file is really a super graph which maps each vertice in each other XML file (of that data set) to its corresponding enzyme. For all our conversions we use this map file, like was done in [5].

Our tool, *FindMFS*, automatically calls *Afopt* with the required parameters. Then when *Afopt* has finished, *FindMFS* automatically takes the output from *Afopt* (which is MFI). Since each $mfi \in MFI$ consists of only non-negative integers,we first translate back each non-negative number to its corresponding edge using the file "edges.txt". Next the MFI is converted to MFS and written to file in the same directory as "transactions.txt" and "edges.txt" are.

The following parameters for *FindMFS* are required. First it needs the directory of "transactions.txt" and "edges.txt" (both must be in same directory). Secondly the Afopt executable file is required and finally the threshold value you wish to use is required. *FindMFS* writes MFS to a file called "fsg.n", where n is the threshold value you used.

In the tables below you will find the run times of both *pwmine* and *FindMFS* for some threshold values and all data sets of all KEGG data versions. The run times are measured, for both tools, from start to finish. So when either tools needs x seconds, this is from the input file to and including the output (MFS) file. The columns in the tables indicate the following:

- T (%) indicates the threshold value as percentages of organisms present in the data set.
- T (abs) indicates the absolute threshold value for that data set.
- #E/I indicates the number of distinct frequent edges.
- #MFS indicates how many maximal frequent subgraphs are found.
- Max E indicates how many edges the largest maximal frequent subgraph has.
- The *pwmine* column indicates how many seconds *pwmine* ([5]) needed.
- The *FindMFS* column indicates how many seconds *FindMFS* needed.

Table 16, Table 17 and Table 18 show the run times for both *pwmine* and *FindMFS* for the respective data sets of the KEGG data version of January 2008. Similarly tables 19, 20 and 21 show these run times for the data sets of KEGG data version January 2004. Finally tables 22, 23 and 24 show run times of the data sets of KEGG data version Later 2004.

As you can see in the tables of the KEGG data version of January 2004, *FindMFS* is in most cases a bit slower than *pwmine*. However for the 5% and the 7.5% thresholds for all three data sets of this version, *FindMFS* is indeed faster than *pwmine*. *FindMFS* even easily produces MFS where *pwmine* could not in a reasonable amount of time.

For the Later 2004 KEGG data version, the data sets were bigger and more complex. Although *pwmine* already had much more trouble in finding MFS in a reasonable amount of time, whereas *FindMFS* had no trouble at all. This is even more so true for our latest KEGG data version of January 2008.

Our tool *FindMFS* easily calculates MFS for all threshold values of any KEGG data version. As you may have noticed, *FindMFS* is not always the fastest solution. But when things get complicated, usually at lower threshold values with more than a few frequent edges, *FindMFS* finds MFS much, much faster than *pwmine*. Also the cases where *FindMFS* is slower than *pwmine*, *FindMFS* takes at most a few seconds. *FindMFS* is, to our knowledge, by far the fastest way to compute MFS.

Table 16: Alanine & Aspartate of KEGG data version January 2008

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	35	106	172	54	>2000	2.905
7.5	52	95	177	48	>2000	2.292
10.0	70	86	158	39	>2000	1.869
12.5	87	77	93	37	>2000	1.688
15.0	105	72	84	30	>2000	1.965
17.5	122	63	71	30	>2000	1.12
20.0	140	61	55	29	>2000	1.433
30.0	209	43	36	23	498.561	1.061
40.0	279	32	19	18	13.348	0.921
50.0	349	30	18	13	0.491	1.031
60.0	419	17	13	7	0.033	0.985
70.0	489	11	10	3	0.027	0.838
80.0	558	0	0	0	0.015	0.704

Table 17: Glutamate of KEGG data version January 2008

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	35	115	1260	76	>2000	11.602
7.5	52	113	959	70	>2000	6.361
10.0	70	107	658	62	>2000	5.158
12.5	87	100	498	49	>2000	3.716
15.0	105	90	376	42	>2000	3.197
17.5	122	87	275	38	>2000	2.196
20.0	140	82	222	37	>2000	2.249
30.0	209	57	135	29	>2000	1.794
40.0	279	42	36	25	>2000	1.084
50.0	349	28	29	18	10.88	1.132
60.0	419	18	14	11	0.124	1.022
70.0	489	13	13	3	0.028	0.89
80.0	558	0	0	0	0.027	0.661

Table 18: Pyrimidine of KEGG data version January 2008

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	35	135	2684	70	>2000	49.151
7.5	52	108	3126	57	>2000	61.632
10.0	70	96	3595	50	>2000	63.989
12.5	87	90	3263	48	>2000	43.869
15.0	105	88	2396	46	>2000	27.46
17.5	122	83	1800	42	>2000	17.241
20.0	139	75	1390	38	>2000	12.103
30.0	209	64	546	30	>2000	4.401
40.0	279	52	310	26	1202.748	2.769
50.0	349	38	148	17	12.224	2.151
60.0	418	32	43	12	0.348	1.468
70.0	488	21	13	8	0.039	0.998
80.0	558	9	6	3	0.032	0.86

Table 19: Alanine & Aspartate of KEGG data version January 2004

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	8	173	100	31	>2000	1.331
7.5	12	114	72	18	11.464	1.107
10.0	16	61	34	16	2.494	0.818
12.5	20	52	31	13	0.715	0.923
15.0	23	39	21	12	0.115	0.791
17.5	27	31	19	12	0.061	0.816
20.0	31	22	15	11	0.024	0.821
30.0	47	11	3	8	0.008	0.763
40.0	62	8	3	4	0.007	0.606
50.0	78	1	1	1	0.006	0.742
60.0	94	0	0	0	0.007	0.595
70.0	110	0	0	0	0.006	0.689
80.0	125	0	0	0	0.012	0.730

Table 20: Glutamate of KEGG data version January 2004

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	8	180	132	30	>2000	1.880
7.5	12	103	56	15	1.488	1.088
10.0	16	70	34	15	0.384	0.931
12.5	19	58	39	13	0.127	0.949
15.0	23	36	21	11	0.028	0.830
17.5	27	26	13	10	0.016	0.904
20.0	31	23	12	9	0.010	0.801
30.0	47	10	7	3	0.008	0.757
40.0	62	4	2	3	0.008	0.859
50.0	78	1	1	1	0.007	0.807
60.0	93	0	0	0	0.009	0.744
70.0	109	0	0	0	0.008	0.923
80.0	124	0	0	0	0.008	0.778

Table 21: Pyrimidine of KEGG data version January 2004

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	8	172	191	37	>2000	2.827
7.5	12	126	185	19	10.333	2.597
10.0	16	91	120	15	0.306	1.690
12.5	20	71	67	15	0.090	1.001
15.0	23	57	49	12	0.026	1.048
17.5	27	42	35	9	0.013	0.832
20.0	31	36	23	7	0.008	0.674
30.0	47	13	8	4	0.007	0.734
40.0	62	5	4	2	0.007	0.628
50.0	78	0	0	0	0.007	0.553
60.0	94	0	0	0	0.007	0.531
70.0	110	0	0	0	0.006	0.678
80.0	125	0	0	0	0.007	0.531

Table 22: Alanine & Aspartate of KEGG data version Later 2004

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	11	105	28	82	>2000	1.142
7.5	17	98	34	70	>2000	1.043
10.0	23	91	35	60	>2000	1.881
12.5	29	85	32	55	>2000	1.783
15.0	34	77	27	47	>2000	0.965
17.5	40	75	30	45	>2000	1.433
20.0	46	71	16	45	>2000	1.141
30.0	68	55	18	30	>2000	0.810
40.0	91	43	8	29	>2000	0.765
50.0	114	29	4	23	73.428	0.569
60.0	137	24	4	17	1.324	0.469
70.0	160	19	4	12	0.044	0.538
80.0	182	9	2	7	0.008	0.384

Table 23: Glutamate of KEGG data version Later 2004

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	11	123	104	89	>2000	3.105
7.5	17	116	106	81	>2000	3.679
10.0	23	111	87	80	>2000	2.874
12.5	29	107	108	63	>2000	3.424
15.0	34	104	92	55	>2000	2.664
17.5	40	99	68	52	>2000	2.912
20.0	46	90	47	52	>2000	2.039
30.0	68	75	19	40	>2000	0.822
40.0	91	48	24	30	>2000	1.052
50.0	114	36	4	29	>2000	0.604
60.0	137	33	10	21	38.368	0.732
70.0	160	21	7	18	1.122	0.790
80.0	182	21	14	6	0.018	0.812

Table 24: Pyrimidine of KEGG data version Later 2004

T (%)	T (abs)	#E/I	#MFS	Max E	*pwmine*	FindMFS
5.0	11	128	62	97	>2000	2.990
7.5	17	120	115	85	>2000	3.901
10.0	23	119	158	73	>2000	5.275
12.5	29	117	167	71	>2000	5.074
15.0	34	114	133	68	>2000	4.021
17.5	40	107	146	63	>2000	4.974
20.0	46	103	134	63	>2000	4.505
30.0	68	86	81	43	>2000	3.227
40.0	91	71	42	33	>2000	2.264
50.0	114	52	14	31	1892.604	1.172
60.0	137	41	11	23	15.669	0.995
70.0	160	38	11	16	0.28	0.951
80.0	182	26	7	9	0.024	0.672

12 Conclusions

We have discussed several improvements of the original state of the art algorithm from [5]. Obviously leaving out edges which are infrequent by themselves reduces the work to be done. One improvement is sorting the edges on $\#neighbors * \#graphs$ which also results in faster run times.

We also add multiple edges (or items) instead of adding just edges (or items) one by one. In this way we save on executing a time consuming method, speeding the program up significantly. Also implementing "FICombine" from the MFI algorithm in Figure 2 in the MFS algorithm (Figure 3), speeds the MFS program up significantly.

We ran the MFS program and the MFI program a large number of times, each time on a random permutation of the edges/items. And we stopped each time we found a mfs or mfi respectively. Then we will find a lot of mfs or mfi possibly all, but this is not guaranteed. This is however very fast for low(er) threshold values.

Our most important conclusion is that we can use tools and methods from the area of item set mining. Using the tool *Afopt* from [8] in combination with the proper postprocessing gives us MFS much faster than any other method or algorithm could achieve. Even for the latest KEGG data version (January 2008) this method provides us with MFS for threshold value of 5% in than one minute!

Acknowledgments. On a personal note I would like to thank my supervisors, prof. dr. Hans Zantema and dr. Dragan Bošnački, for their part during my master project, their work on writing and publishing the paper "*Finding Frequent Subgraphs in Biological Networks via Maximal Item Sets*", [4], based on my findings and their part in the final assessment committee. Also I would like to thank dr. Erik de Vink for his review of a draft version of the paper. Furthermore I would also like to thank the other members of the final assessment committee, dr. Erik de Vink and dr. Toon Calders for their part in the assessment committee and during my master project. I would like to thank Mehmet Koyutürk for providing the data used in [5].

References

[1] N. Eén, N. Sörensson, *Translating Pseudo-Boolean Constraints into SAT*, Journal on Satisfiability, Boolean Modeling and Computation 2, 2006, pp. 1-25, published by University of Delft in cooperation with IOS Press.

[2] Frequent Itemset Mining Implementations Repository, http://fimi.cs.helsinki.fi/

[3] K. Gouda, M.J. Zaki, *Efficiently Mining Maximal Frequent Itemsets*, IEEE International Conference on Data Mining ICDM '01, pp. 163-170, 2001.

[4] H. Zantema, S. Wagemans, D. Bošnački, *Finding Frequent Subgraphs in Biological Networks via Maximal Item Sets*, Proceedings of Bioinformatics Research and Development (BIRD) 2008, accepted to be published, Technical University of Vienna, Austria, July 2008.

[5] M. Koyutürk, A. Grama, W. Szpankowski, *An efficient algorithm for detecting frequent subgraphs in biological networks*, Bioinformatics, vol. 20 (suppl. 1), pp. i200-i207, Oxford University Press, 2004.

[6] M. Koyutürk, A. Grama, W. Szpankowski, *Description of the tools* p2g *and* pwmine, http://www.cs.purdue.edu/homes/koyuturk/pathway/

[7] Kyoto Encyclopedia of Genes and Genomes, http://www.kegg.com.

[8] G. Liu, H. Lu, J. X. Yu, W. Wang, X. Xiao, *AFOPT: An Efficient Implementation of Pattern Growth Approach* Proc. of the ICDM 2003 Workshop on Frequent Itemset Mining Implementations, FIMI '03, 19 December 2003, Melbourne, Florida, USA, 2003.

A Full MFI algorithm for running example

MFI-Backtrack({}, {ab,ac,bc,de}, 0)
for each $x \in \{ab, ac, bc, de\}$ do
<Iteration 0, $x := \{ab\}$ >
$I_1 := I_0 \cup x = \{\} \cup ab = ab$
$P_1 := \{y : y \in C_0 \wedge y > x\} = \{ac, bc, de\}$
$I_1 \cup P_1 = \{ab, ac, bc, de\}$ has no superset in {}
$C_1 := FICombine(I_1, P_1) = \{ac, de\}$
$C_1 = \{ac, de\}$ is not empty
<else part (going into recursion 1)>
MFI-Backtrack($\{ab\}, \{ac, de\}, 1$)
for each $x \in \{ac, de\}$ do
<Iteration 0, $x := \{ac\}$ >
$I_2 := \{ab, ac\}$
$P_2 := \{de\}$
$\{ab, ac, de\}$ has no superset in {}
$C_2 := FICombine(\{ab, ac\}, \{de\}) = \{de\}$
$\{de\}$ is not empty
<else part (going into recursion 1.1)>
MFI-Backtrack($\{ab, ac\}, \{de\}, 2$)
for each $x \in \{de\}$ do
<Iteration 0, $x := \{de\}$ >
$I_3 := \{ab, ac, de\}$
$P_3 := \{\}$
$\{ab, ac, de\}$ has no superset in {}
$C_3 := FICombine(\{ab, ac, de\}, \{\}) = \{\}$
{} is empty
<then part>
$\{ab, ac, de\}$ has no superset in {}
$MFI := MFI \cup I_3 = \{\} \cup \{ab, ac, de\} = \{\{ab, ac, de\}\}$
<return from recursion 1.1, continue recursion1>
<Iteration 1, $x := \{de\}$ >
$I_2 := \{ab, de\}$
$P_2 := \{\}$
$\{ab, de\}$ has superset in $\{\{ab, ac, de\}\}$
return
<return from recursion 1, continue initial call>
<Iteration 1, $x := \{ac\}$ >
$I_1 := \{ac\}$
$P_1 := \{bc, de\}$
$\{ac, bc, de\}$ has no superset in $\{\{ab, ac, de\}\}$
$C_1 := FICombine(\{ac\}, \{bc, de\}) = \{bc, de\}$
$\{bc, de\}$ is not empty
<else part, (going into recursion 2)>
MFI-Backtrack($\{ac\}, \{bc, de\}, 1$)
for each $x \in \{bc, de\}$ do
<Iteration 0, $x := \{bc\}$ >
$I_2 := \{ac, bc\}$
$P_2 := \{de\}$

$\{ac, bc, de\}$ has no superset in $\{\{ab, ac, de\}\}$
$C_2 := FICombine(\{ac, bc\}, \{de\}) = \{\}$
$\{\}$ is empty
<then part>
$\{ac, bc\}$ has no superset in $\{\{ab, ac, de\}\}$
$MFI := \{\{ab, ac, de\}\} \cup \{ac, bc\} = \{\{ab, ac, de\}, \{ac, bc\}\}$
< Iteration 1, $x := \{de\}$ >
$I_2 := \{ac, de\}$
$P_2 := \{\}$
$\{ac, de\}$ has superset in $\{\{ab, ac, de\}, \{ac, bc\}\}$
return
<return from recursion2>
< Iteration 2, $x := \{bc\}$ >
$I_1 := \{bc\}$
$P_1 := \{de\}$
$\{bc, de\}$ has no superset in $\{\{ab, ac, de\}, \{ac, bc\}\}$
$C_1 := FICombine(\{bc\}, \{de\}) = \{\}$
$\{\}$ is empty
<then part>
$\{bc\}$ has superset in $\{\{ab, ac, de\}, \{ac, bc\}\}$
< Iteration 3, $x := \{de\}$ >
$I_1 := \{de\}$
$P_1 := \{\}$
$\{de\}$ has superset in $\{\{ab, ac, de\}, \{ac, bc\}\}$
return
<exit MFI-Backtrack>

B Full MFS algorithm for running example

MinePathways($\emptyset, \{ab\}, \{ac, bc\}, \{ab\}$)
$ismaximal := true$
for all edges $i \in \{ac, bc\}$ do
<Iteration 0, $i := \{ac\}$ >
$D := \{ab\} \cup \{ac\} = \{ab, ac\}$
$E_1 := \{ab\} \cup \{ac\} = \{ab, ac\}$
$\{ab, ac\}$ is frequent
$ismaximal := false$
$C_1 := (C_0 \cup N(ac)) \backslash D = (\{ac, bc\} \cup \{ab, bc\}) \backslash \{ab, ac\} = \{bc\}$
<recursion>
MinePathways($\emptyset, \{ab, ac\}, \{bc\}, \{ab, ac\}$)
$ismaximal := true$
for all edges $i \in \{bc\}$ do
<Iteration 0, $i := \{bc\}$ >
$D := \{ab, ac, bc\}$
$E_2 := \{ab, ac, bc\}$
$\{ab, ac, bc\}$ is NOT frequent
<nomore candidates to add to $\{ab, ac\}$, exit loop>
$ismaximal$ yields $true$
$\{ab, ac\}$ has no superset in $\emptyset$
$MFS := \{\emptyset\} \cup \{ab, ac\} = \{\{ab, ac\}\}$
<comes back from recursion>
<Iteration 1, $i := \{bc\}$ >
$D := \{ab, ac, bc\}$
$E_1 := \{ab, bc\}$
$\{ab, bc\}$ is NOT frequent
<no more candidates to add to $\{ab\}$, exit loop>
ismaximal yields false
<exit MinePathways>

Now we run MinePathways for the next frequent edge ac:
MinePathways($\{\{ab, ac\}\}, \{ac\}, \{bc\}, \{ab, ac\}$)
$ismaximal := true$
for all edges i in bc do
< Iteration 0, $i := \{bc\}$ >
$D := \{ab, ac, bc\}$
$E_1 := \{ac, bc\}$
$\{ac, bc\}$ is frequent
$ismaximal := false$
$C_1 := C_0 \cup N(bc) \backslash D = bc \cup \{ab, ac\} \backslash \{ab, ac, bc\} = \emptyset$
<recursion>
MinePathways($\{\{ab, ac\}\}, \{ac, bc\}, \emptyset, \{ab, ac, bc\}$)
$ismaximal := true$
for all edges $i \in \emptyset$ do
<Nothing to loop, exit loop>
$ismaximal$ is $true$
$\{ac, bc\}$ has no superset in $\{\{ab, ac\}\}$

MFS:= $\{\{ab, ac\}\} \cup \{ac, bc\} = \{\{ab, ac\}, \{ac, bc\}\}$
<comes back from recursion>
<no more candidates to add to $\{ac\}$, exit loop>
$ismaximal$ is $false$
<exit MinePathways>

Frequent edge bc:
MinePathways($\{\{ab, ac\}, \{ac, bc\}\}, \{bc\}, \emptyset, \{ab, ac, bc\}$)
$ismaximal := true$
for all edges $i \in \emptyset$ do
<nothing to loop, exit loop>
$ismaximal$ is $true$
$\{bc\}$ has superset in $\{\{ab, ac\}, \{ac, bc\}\}$
<exit MinePathways>

frequent edge de remains, edge de has no neighbors:
MinePathways($\{\{ab, ac\}, \{ac, bc\}\}, \{de\}, \emptyset, \{ab, ac, bc, de\}$)
$ismaximal := true$
for all edges $i \in \emptyset$ do
<nothing to loop, exit loop>
$ismaximal$ is $true$
$\{de\}$ has no superset in $\{\{ab, ac\}, \{ac, bc\}\}$
$MFS := \{\{ab, ac\}, \{ac, bc\}, \{de\}\}$
<exit MinePathways>