Efficiently Mining Frequent Subpaths

Sumanta Guha[1]

[1] Computer Science & Information Management Program
Asian Institute of Technology
PO Box 4, Klong Luang, Pathumthani 12120, Thailand
Email: guha@ait.asia

Abstract

The problem considered is that of finding frequent subpaths of a database of paths in a fixed undirected graph. This problem arises in applications such as predicting congestion in network traffic. An algorithm based on Apriori, called AFS, is developed, but with significantly improved efficiency through exploiting the underlying graph structure, which makes AFS feasible for practical input path sizes. It is also proved that a natural generalization of the frequent subpaths problem is not amenable to any solution quicker than Apriori.

Keywords: AFS, Apriori, data mining, frequent subpath, frequent substructure, graph mining.

1 Introduction

Within the general problem of mining frequent patterns from a database of transactions, an area of some recent interest is where the transactions occur in a structured or semi-structured set. The structure considered often is that of a graph because objects under scrutiny in various applications can, in fact, be modeled as graphs, e.g., chemical compounds, web links, virtual communities, XML specifications and networks of different kinds. Finding frequent subgraphs of a database of graph transactions has been an area of particular activity. Apriori-based algorithms for this problem have been given, amongst others, by (Vanetik et al., 2002), (Inokuchi et al., 2000) and (Kuramochi and Karypis, 2001), while (Yan and Han, 2002) give an algorithm which uses a novel encoding scheme for graphs. See (Cook and Holder, 2006) for a survey of graph mining techniques.

The problem which we consider is a particular case of the problem of finding frequent subgraphs. In particular, in our case all transactions are paths in a fixed undirected graph, and we are interested in determining those paths in that graph which occur frequently as subpaths of the transaction paths. This is a natural problem to consider. For example, if each path in the database represents the route taken by an object such as a message or vehicle, then the frequent subpaths represent congested sections, or hot spots. Related work includes (Chen et al., 1998) and (Gudes and Pertsev, 2005), which both compute the *whole* paths themselves that are frequently traversed, rather then the frequently traversed shared parts which we consider (e.g., a set of paths may individually not be frequently traveled, but particular shared edges could well be congested).

Our algorithm is derived from Apriori (Agrawal and Srikant, 1994) as well. However, a simple-minded application of Apriori – say, by treating paths as itemsets of vertices – fails because the feasibility of Apriori depends on transactions being of small size. However, paths in graphs arising from practical applications are not necessarily short (e.g, consider vehicular traffic in a city), and a straight Apriori-type solution runs into exponential complexity. Instead, we exploit the graph structure for a significant gain in efficiency which leads to a generally applicable solution, which we call AFS (Apriori for Frequent Subpaths). In fact, we analyze and compare the complexities of Apriori and AFS to prove a theoretical gain in efficiency from exponential in input size to low polynomial.

Next, we show that, interestingly, there is no possibility of similarly leveraging the graph structure to improve Apriori for a solution to a natural generalization of the frequent subpaths problem – that of finding so-called frequent strings of subpaths – because the general problem is equivalent in complexity to that of finding frequent itemsets.

2 Problem and Algorithm

2.1 Problem Statement

Let $G = (V,E)$ be an undirected graph with vertex set V and edge set E.

Here are some definitions related to paths in graphs which we'll use. A *path* P in G of length k from a vertex u to u' is a sequence $(v_0, v_1, \ldots, v_k)$ of vertices such that $v_0 = u$ and $v_k = u'$ and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$. (We'll also allow the empty sequence $()$ to denote the empty path of undefined length.) A path Q in G is said to be a *subpath* of P, denoted $Q \lhd P$, if $Q = (w_0, w_1, \ldots, w_{k'})$, where $(w_0, w_1, \ldots, w_{k'})$ is a contiguous subsequence of $(v_0, v_1, \ldots, v_k)$, i.e., if, for some i such that $0 \leq i \leq i + k' \leq k$, we have $w_0 = v_i, w_1 = v_{i+1}, \ldots, w_{k'} = v_{i+k'}$. In this case, if $i = 0$, then Q is called a *prefix* subpath of P, and, if $i + k' = k$, then Q is called a *suffix* subpath of P. For a non-empty path $P = (v_0, v_1, \ldots, v_k)$, $front(P)$ denotes the first vertex v_0 and $tail(P)$ denotes the suffix subpath $(v_1, \ldots, v_k)$. A path (or, subpath) of length k will often be called a k-path (or, k-subpath).

Following are a few more definitions pertinent particularly to our problem. Let $\mathcal{P}$ be a given set of paths in G. A path Q in G is said to have *support* $\text{support}(Q) = |\{P \in \mathcal{P} : Q \lhd P\}|$, i.e., the number of paths in $\mathcal{P}$ of which Q is a subpath. Moreover, suppose a *minimum support* value *min_sup* is specified. If $\text{support}(Q) \geq min\_sup$, then Q is said to be a *frequent subpath*.

The statement of the problem is now straightforward: *Given a set $\mathcal{P}$ of paths in an undirected graph G, determine all frequent subpaths.*

See Figure 1 for an example of three paths in a grid graph.

Remark: In database terminology, $\mathcal{P}$ is a database of transactions, where each transaction P is a path in a fixed graph G.

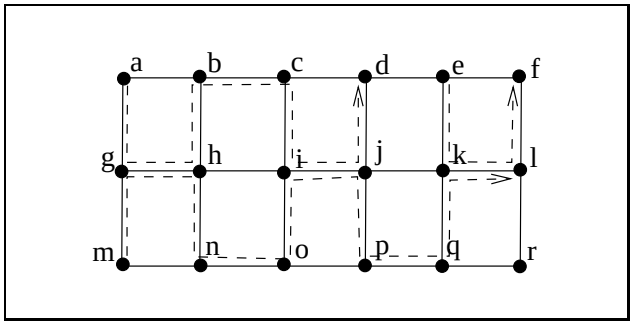

Figure 1: A grid graph with three paths indicated by directed broken lines. If $min\_sup = 2$ then the frequent subpaths are (g), (h), (i), (j), (k), (l), (g,h), (i,j) and (k,l).

2.2 Apriori Algorithm

As our algorithm to find frequent subpaths is derived from the Apriori algorithm, and as we'll be comparing the complexities of the two, we'll first describe Apriori in some detail.

Let $\mathcal{D}$ be a database of transactions, where each transaction $T \in \mathcal{D}$ is a subset of a set of all items $\mathcal{I}$. The support of an itemset $I \subset \mathcal{I}$ is $\mathsf{support}(I) = |\{T \in \mathcal{D} : I \subset T\}|$. If $\mathsf{support}(I) \geq min\_sup$, for a specified value min_sup, then I is frequent. Following is pseudo-code for the Apriori algorithm to determine all frequent itemsets (adapted from (Agrawal and Srikant, 1994)).

```
Apriori
L_1 = {frequent 1-itemsets};
for (k = 2; L_{k-1} ≠ ∅; k++)
    {
    C_k = join(L_{k-1}, L_{k-1}); // Generate candidates.
    C_k = prune(C_k); // Prune candidates.
    L_k = checkSupport(C_k); // Eliminate candidate
                             // if support too low.
    }
return ∪_k L_k; // Returns all frequents itemsets.
```

We discuss next the routines in the Apriori **for** loop and how all three are implemented using a function $\mathsf{subset}(X,T)$, where X is a set of itemsets and T is an itemset, which returns the subset Y of X consisting of those itemsets which are contained in T (we'll discuss implementing $\mathsf{subset}(X,T)$ itself later).

Firstly, $\mathsf{join}(L_{k-1},L_{k-1})$ generates all k-itemsets of the form $\{i_1,i_2,\ldots,i_k\}$, where both $\{i_1,i_2,\ldots,i_{k-1}\}$ and $\{i_1,i_2,\ldots,i_{k-2},i_k\}$ belong to L_{k-1} (note that itemsets are always assumed listed in lexicographic order), i.e., unions of pairs of itemsets in L_{k-1} both of whose members share the same first $k-2$ items. Secondly, $\mathsf{prune}(C_k)$ deletes all $I \in C_k$ such that some $(k-1)$-subset of I does not belong to L_{k-1}. It may be checked that *both* $\mathsf{join}(L_{k-1},L_{k-1})$ and $\mathsf{prune}(C_k)$ are implemented by the following routine which uses $\mathsf{subset}(L_{k-1},*)$:

```
pruneJoin
C_k = ∅;
for each itemset I = {i_1, i_2, ..., i_{k-1}} ∈ L_{k-1}
    for each item j ∈ J such that j > i_{k-1}
        {
        I' = {i_1, i_2, ..., i_{k-1}, j};
        for each (k-1)-subset A of I'
            if (subset(L_{k-1}, A) = ∅) goto reject;
            // Reject I' if it has a (k-1)-subset
            // not belonging to L_{k-1}.

        add I' to C_k;
        reject:
        }
return C_k; // Returns prune(join(L_{k-1}, L_{k-1})).
```

Finally, $\mathsf{checkSupport}(C_k)$ counts the support of each itemset currently in C_k to eliminate those which are not frequent. It is straightforwardly implemented with the help of $\mathsf{subset}(C_k,*)$:

```
checkSupport
L_k = ∅;
for each I ∈ C_k
    I.count = 0;
for each transaction T ∈ D
    {
    C_T = subset(C_k, T);
    for each I ∈ C_T
        I.count++;
    }
for each I ∈ C_k
    if (I.count ≥ min_sup) add I to L_k;
return L_k; // Returns members of C_k with support
            // at least min_sup.
```

Therefore, when implementing Apriori the calls to join and prune in the **for** loop are replaced by a single call to pruneJoin, while checkSupport is implemented as above.

The function $\mathsf{subset}(X,T)$ itself is implemented by first storing the itemsets of X in a trie (prefix tree) (Fredkin, 1960) $\mathcal{T}$ on the "alphabet" $\mathcal{I}$ of items ordered lexicographically, each itemset treated as an ordered string. (Agrawal and Srikant, 1994) actually use a particular implementation called a hash tree (Coffman Jr. and Eve, 1970), where pointers to children are stored in a hash table keyed on items at each internal node (the use of a hash tree in this case instead of a simple trie is justified by the typically large size of $\mathcal{I}$). See Figure 2 for an example.

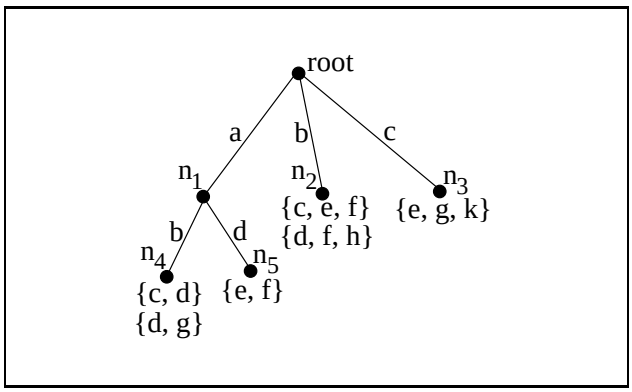

Figure 2: A hash tree $\mathcal{T}$ storing a set of six 4-itemsets $X = \{\{a,b,c,d\},\{a,b,d,g\},\{a,d,e,f\},\{b,c,e,f\},\{b,d,f,h\},\{c,e,g,k\}\}$, where each leaf can store at most two itemsets (only the suffix of an itemset following the prefix defined by the path to the leaf is stored).

The function $\mathsf{subset}(X,T)$ is then executed by calling $\mathsf{doSubset}(root(\mathcal{T}),T)$ using the recursive routine below:

```
doSubset(node, I)
{
Y = ∅;
if (node is leaf) add checkItemsets(node, I) to Y;
// Function checkItemsets(node, I) returns those
// itemsets stored at node that are contained in I.

else if (I = ∅); // Nothing is added to Y.
else for each (i ∈ I)
    if (node.ch(i) exists)
    add i * doSubset(node.ch(i), {j ∈ I : j > i}) to Y;
    // For each item i ∈ I recurse on the corresponding
    // child of node. We denote by i * Z the union of i
    // with each itemset in Z.

return Y;
```

}

For example, in Figure 2, doSubset$(root,\{a,b,c,d,e,f\})$ makes three recursive calls to doSubset with parameters $(n_1,\{b,c,d,e,f\})$, $(n_2,\{c,d,e,f\})$ and $(n_3,\{d,e,f\})$, respectively. The first of these in turn calls doSubset with parameters $(n_4,\{c,d,e,f\})$ and $(n_5,\{e,f\})$, while the second and third add $\{b,c,e,f\}$ and nothing, respectively, to the answer Y, etc.

Though various technical improvements in implementing Apriori have been suggested – see (Han and Kamber, 2005) for a discussion – we'll not consider them here, but use as our reference the basic implementation described above. This is in the interests of making an apples-to-apples comparison with AFS, whose basic implementation is described next.

2.3 Apriori for Frequent Subpaths

We present our algorithm AFS (Apriori for Frequent Subpaths) in a manner as similar as possible to that for Apriori in the previous section, so that it's easy to see exactly how the added structure in the setting of AFS helps make it more efficient.

```
AFS
L_0 = {frequent 0-subpaths};
for (k = 1; L_{k-1} ≠ ∅; k++)
    {
    C_k = AFSextend(L_{k-1}); // Generate candidates.
    C_k = AFSprune(C_k); // Prune candidates.
    L_k = AFScheckSupport(C_k);
    // Eliminate candidate if support too low.
    }
return ∪_k L_k; // Returns all frequents subpaths.
```

The gain from the graph structure is first seen in generating candidates: we obtain C_k by simply extending each path in L_{k-1} by every edge incident on its last vertex (instead of potentially "joining" every pair of paths in L_{k-1}). This is justified as it may be seen that the set of k-paths obtained by so extending paths in L_{k-1} indeed contains L_k. Pruning is simpler as well because, after extending a path P in L_{k-1} to a k-path P', the only $(k-1)$-subpath of P' whose membership in L_{k-1} need be checked is its suffix $k-1$-subpath. The reason is that P' has only two $k-1$-subpaths: one prefix (P itself) and the other suffix.

E.g., in Figure 1, $(g,h) \in L_1$ would generate four extensions for inclusion in C_2: (g,h,i), (g,h,b), (g,h,g) and (g,h,n). Moreover, in the prune step, e.g., for (g,h,i), only (h,i) has to be checked if it belongs to L_1.

Both AFSextend(L_{k-1}) and AFSprune(C_k) are implemented by the routine AFSpruneExtend below, which should be compared with the earlier pruneJoin routine for Apriori. AFSpruneExtend uses the function subpaths(X,P), where X is a set of paths and T is a path, which returns the subset Y of X consisting of those paths which are subpaths of T. Function subpaths(X,P), whose implementation we'll detail momentarily, is, of course, the counterpart of the earlier subset(X,T).

```
AFSpruneExtend
C_k = ∅;
for each path P = (v_0, v_1, ..., v_{k-1}) ∈ L_{k-1}
    for each vertex v ∈ V adjacent to v_{k-1}
        {
        P' = (v_0, v_1, ..., v_{k-1}, v);
        if (subpaths(L_{k-1}, (v_1, ..., v_{k-1}, v) = ∅)
         goto reject;
        // Reject P' if its suffix (k-1)-subpath
        // does not belong to L_{k-1}.

        add P' to C_k;
        reject:
        }
return C_k; // Returns ASFprune(ASFextend(L_{k-1})).
```

The routine AFScheckSupport is a near copy of its Apriori counterpart checkSupport.

```
AFScheckSupport
L_k = ∅;
for each Q ∈ C_k
    Q.count = 0;
for each path P ∈ 𝒫
    {
    C_P = subpaths(C_k, P);
    for each Q ∈ C_P
        Q.count++;
    }
for each Q ∈ C_k
    if (Q.count ≥ min_sup) add Q to L_k;
return L_k; // Returns members of C_k with support
            // at least min_sup.
```

Therefore, when implementing AFS the calls to AFSextend and AFSprune in the for loop are replaced by a single call to AFSpruneExtend, while AFScheckSupport is implemented as above.

It's in implementing subpaths(X,P) that we leverage the graph setting of AFS to huge gain over subset(X,T) (we'll see the actual calculations in the next section). Paths in X are stored in a hash tree $\mathcal{T}$ as well, exactly as for subset(X,T). It's straightforward to use this tree of paths to determine which are prefix subpaths of P. Therefore, noting that a path in X is a subpath of P if an only if it is a prefix subpath of some suffix subpath of P, subpaths(X,P) is implemented by calling doSubpaths$(root(\mathcal{T}),(w_0,w_1,\ldots,w_k))$, where $P=(w_0,w_1,\ldots,w_k)$.

```
doSubpaths(node, {w_0, w_1, ..., w_k})
{
Y = ∅;
for (i = 0; i ≤ k; i++)
    add doPrefixSubpaths(node, (w_i, w_{i+1}, ..., w_k))
    to Y;
    // Iteratively calls doPrefixSubpaths(node, Q)   // on
each suffix of Q of P = (w_0, w_1, ..., w_k).

return Y
}
```

Compare the following with doSubset.

```
doPrefixSubpaths(node, Q)
{
Y = ∅;
if (node is leaf) add checkPrefixPaths(node, Q) to Y;
// Function checkPrefixPaths(node, Q) returns those
// paths stored at node that are prefix subpaths of P.

else if (Q = ( ) ); // Nothing is added to Y.
else
    if (node.ch(first(Q)) exists)
    add first(Q) *
        doPrefixSubpaths(node.ch(first(Q)),
                         tail(Q)) to Y;
    // Descend from node along the path labeled by
    // successive vertices of Q. We denote by v*Z the
    // concatenation of v with each path in Z.

return Y;
}
```

For example, suppose the hash tree in Figure 2 represents a set of paths instead of itemsets. Then,

the call doSubpaths$(node,(a,b,c,d,e,f))$ spawns six iterations of the call doPrefixSubpaths with parameters $(node,(a,b,c,d,e,f)),(node,(b,c,d,e,f)),\ldots,$ $(node,(f))$, respectively. Each of the doPrefixSubpaths calls descends recursively from the root down a single path of $\mathcal{T}$. E.g., the one with parameters $(node,(a,b,c,d,e,f))$ descends to n_4 to finally call doPrefixSubpaths$(n_4,(c,d,e,f))$, which adds (a,b,c,d) to the answer Y.

2.4 Complexity: AFS vs. Apriori

Consider Apriori first. The recursion in doSubset$(node,I)$ yields a Fibonacci-type recurrence in running time of $t(k)=t(k-1)+t(k-2)+\ldots t(1)$, if $I=\{i_1,i_2,\ldots,i_k\}$, implying a time bound function of order exponential in the size of I, which we indicate by $O(\exp(|I|))$ (We ignore the cost of calls to checkItemsets$(node,I)$.) The size of the hash tree rooted at *node* is an obvious upper time bound as well on doSubset$(node,I)$.

Therefore, similar bounds apply to subset(X,T) as well. In particular, subset(C_k,T) and subset(L_k,T), used to implement Apriori, are bounded in running time by $O(\min(\exp(|T|),\text{size\_ht}(C_k)))$ and $O(\min(\exp(|T|),\text{size\_ht}(L_k)))$, respectively, where size_ht(X) denotes the size of the hash tree storing X.

It follows that the total time cost incurred by calls to pruneJoin from Apriori is

$$O(\ |\mathcal{J}|\sum_k k|L_k|\min(\ \exp(k),\text{size\_ht}(L_k)\ )\ )$$

(the expectation that on the average there will be $O(|\mathcal{J}|)$ items greater than the last one in an itemset justifies the $|\mathcal{J}|$ factor) and by those to checkSupport is

$$O(\ \sum_k(\ |C_k|+\sum_{T\in\mathcal{D}}\min(\exp(|T|),\text{size\_ht}(C_k)\ )\ )$$

Next, consider AFS. The routine doPrefixSubpaths$(node,Q)$ is bounded by time linear in $|Q|$ as the recursion descends from *node* along a path labeled by successive vertices of Q. The height *height* of the hash tree rooted at *node* is a bound as well. Consequently, doSubpaths$(node,P)$ takes time $O(\min(|P|,height)+\min(|P|-1,height)+\ldots+\min(1,height))=O(\ \min(|P|^2,|P|height)\ )$.

Therefore, subpaths(C_k,P) runs in time bounded by $O(\ \min(|P|^2,|P|\text{height\_ht}(C_k))\ )$, and subpaths$(L_k,P)$ in time bounded by $O(\ \min(|P|^2,|P|\text{height\_ht}(L_k))\ )$, where height_ht$(X)$ denotes the height of the hash tree storing X, which represents a gain in efficiency over the corresponding Apriori routine subset(X,T) from exponential to quadratic.

We have, therefore, that the total time cost incurred by calls to AFSextendJoin from AFS is

$$O(\ \sum_k|L_k|\min(\ k^2,\text{height\_ht}(L_k)\ )\ )$$

(we assume that on the average each vertex has $O(1)$ neighbors) and those to AFScheckSupport is

$$O(\ \sum_k(\ |C_k|+\sum_{P\in\mathcal{P}}\min(|P|^2,\text{height\_ht}(C_k)\ )\ )$$

Clearly, Apriori is vulnerable to exponential time worst-case behavior. In fact, it's evident from the complexity expressions for pruneJoin and checkSupport that the feasibility of applying Apriori lies in assuming that (a) the size of individual transactions in the database is $O(1)$, and (b) the size of C_k decreases rapidly with k. Fortunately, both assumptions are justified in various practical scenarios, e.g., market basket analysis.

In case of AFS though (a) is not a reasonable assumption: transactions in the database, i.e., paths in a graph, may not be short, or $O(1)$ in length. In practical applications, e.g., vehicles traveling in a network of roads, paths taken may even be of size comparable to that of the graph itself. However, we see from the last two expressions above that, even then, AFS has a worst-case behavior quadratic in the total length of the input paths, making it practically applicable.

Experimental Verification: The theoretical advantage of AFS can be tested in practical situations by using existing test data, or by generating random paths in large graphs, and then finding frequent subpaths using both Apriori (ignoring the graph structure and treating paths as itemsets of vertices) and AFS. We are currently in the process of setting up such experiments.

2.5 A Generalization and its Hardness

The intersection of a set of paths in an undirected graph G is not necessarily a path, but a union of paths. We'll call such an intersection a *string* of subpaths, or, simply, string. Therefore, a natural generalization of the frequent subpaths problem considered in the previous section is as follows: *Given a set $\mathcal{P}$ of paths in an undirected graph G, determine all frequent strings of subpaths*.

For example, in Figure 1, $(g,h)\cup(i,j)$ and (k,l) are the two maximal frequent strings. Observe that knowing all frequent strings evidently implies knowing all frequent subpaths. However, the converse is not true – e.g., it's not possible to deduce from the fact that (g,h), (i,j) and (k,l) are frequent subpaths in Figure 1, that $(g,h)\cup(i,j)$ is a frequent string. Therefore, the problem of finding frequent strings is at least as hard as that of finding frequent subpaths.

Surely, an Apriori-type algorithm may be implemented to find all frequent strings, but, interestingly, no improvement in efficiency over Apriori (as in AFS) can be expected because, as we'll see momentarily, the problem of finding frequent itemsets is equivalent to that of finding frequent strings. Firstly, we'll reduce the first problem to the second in time linear in the size of the input.

Let $\mathcal{D}$ be a database of transactions, each transaction T being a subset of the set of all items $\mathcal{J}$. Let G be the complete graph on the set of vertices $V=\mathcal{J}$. Represent each transaction $T\in\mathcal{D}$, where $T=\{i_1,i_2,\ldots,i_k\}$, by the path $P_T=(i_i,i_2,\ldots,i_k)$, the items in T being in lexicographic order. It may be seen that, given the set of paths $\mathcal{P}=\{P_T:T\in\mathcal{D}\}$, the set of frequent strings corresponds exactly to the set of frequent itemsets for the database $\mathcal{D}$, which completes the reduction claimed and proves that finding frequent strings is at least as hard as finding frequent itemsets.

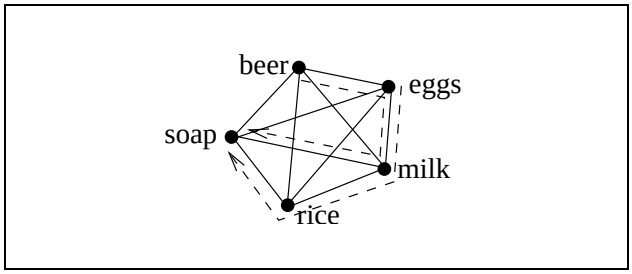

Figure 3: The database of two transactions {beer, eggs, milk, soap} and {eggs, milk, rice, soap} over the set of items $\mathcal{J}$ = {beer, eggs, milk, rice, soap} is represented by two corresponding paths in the complete graph on $\mathcal{J}$.

E.g., for the database of Figure 3, if $min\_sup=2$, then the one maximal frequent itemset is {eggs, milk, soap} and the corresponding one maximal frequent string is (eggs, milk) ∪ (soap).

We'll omit details here of the reduction in the opposite direction. The equivalence of the two problems means that there is no hope of leveraging the graph structure to find a more efficient variation of Apriori to determine frequent strings. However, this should not be an issue in practical applications where it is enough to simply identify the congested subpaths.

3 CONCLUSIONS

We have developed the AFS algorithm to find frequent subpaths which, though derived from Apriori, exploits the underlying graph structure for a gain in efficiency that makes it applicable to practical input sizes for this particular problem. We believe that similar improvements may be found for related problems, e.g., finding frequent subtrees of a collection of trees.

The development of a general framework in which to place the problem of finding frequent substructures of a collection of structures belonging to a family with certain given inheritance properties would be significant as well.

REFERENCES

Agrawal, R. and Srikant, R. (1994). Fast algorithms for mining association rules. In *VLDB'94, Proceedings of 20th International Conference on Very Large Data Bases*, pages 487–499.

Chen, M. S., Park, J. S., and Yu, P. S. (1998). Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10:209–221.

Coffman Jr., E. G. and Eve, J. (1970). File structures using hashing functions. *Communications of the ACM*, 13:427–432.

Cook, D. J. and Holder, L. B. (2006). *Mining Graph Data*. Wiley Interscience.

Fredkin, E. (1960). Trie memory. *Communications of the ACM*, 3:490–499.

Gudes, E. and Pertsev, A. (2005). Mining module for adaptive xml path indexing. In *Proceedings of the 16th International Workshop on Database and Expert Systems Applications*, pages 1015–1019.

Han, J. and Kamber, M. (2005). *Data Mining Concepts and Techniques, 2nd Ed.* Morgan Kaufmann.

Inokuchi, A., Washio, T., and Motoda, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery (Lecture Notes In Computer Science, Vol. 1910)*, pages 12–23.

Kuramochi, M. and Karypis, G. (2001). Frequent subgraph discovery. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 313–320.

Vanetik, N., Gudes, E., and Shimony, S. E. (2002). Computing frequent graph patterns from semistructured data. In *Proceedings of the 2002 IEEE International Conference on Data Mining*, pages 458–465.

Yan, X. and Han, J. (2002). gspan: Graph-based substructure pattern mining. In *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 721–724.