2010 IEEE International Conference on Data Mining

Max-Clique: A Top-down Graph-based Approach to Frequent Pattern Mining

Yan Xie
Department of Computer Science
University of Illinois at Chicago
Chicago, USA
Email: yxie@cs.uic.edu

Philip S. Yu
Department of Computer Science
University of Illinois at Chicago
Chicago, USA
Email: psyu@cs.uic.edu

***Abstract*—*Frequent pattern mining* is a fundamental problem in data mining research. We note that almost all state-of-the-art algorithms may not be able to mine very long patterns in a large database with a huge set of frequent patterns. In this paper, we point our research to solve this difficult problem from a different perspective: we focus on mining *top-k* long maximal frequent patterns because long patterns are in general more interesting ones. Different from traditional level-wise mining or tree-growth strategies, our method works in a *top-down* manner. We pull large maximal cliques from a *pattern graph* constructed after some fast initial processing, and directly use such large-sized maximal cliques as promising candidates for long frequent patterns. A separate *refinement* stage is needed to further transform these candidates into true maximal patterns.**

***Keywords*-frequent pattern mining; top-down; pattern graph.**

I. INTRODUCTION

Frequent pattern mining is considered as one of the fundamental research topics in data mining. So far, most of the existing algorithms mine the complete set of frequent patterns in a database. However, when the length of frequent patterns becomes large, none of these methods can produce results within a reasonable amount of time, mainly because all possible combinations of items are exponential in size, and also, to get the complete set, one must go through the complete set of candidates in a bottom-up mining method. The above dilemma is intrinsic to the frequent pattern mining problem, if one wants the complete result set. However, as we observe in many applications, mining tasks in practice usually attach greater importance to patterns with larger size.

Motivated by the above analysis, we try to design a new method that can mine those interesting long patterns without going through the painful process of growing an explosive candidate set. In particular, we are interested in solving the following problem: For a database, can we efficiently get its top-k long maximal frequent patterns?

Compared to the state-of-the-art bottom-up approaches, we explore the opposite direction and propose a *top-down* mining strategy in this paper. We note that items and their co-existence relationships can be summarized by a so-called *pattern graph*, where frequent patterns will show up as *cliques*. This fact makes cliques promising frequent pattern candidates. Now, we can transform the problem of finding long maximal frequent patterns into the detection of large maximal cliques: Starting from the pattern graph derived from the database, we try to find maximal cliques with large size. The essence of this step is to bypass the bottom-up pattern growth since we have directly jumped to the top of the search space. As will be shown in the rest of this paper, although it is not guaranteed that each maximal clique c will represent a true pattern p in this case, due to the filtering done by the pattern graph, it is likely that the set of items in c is only slightly different from those in p; to this end, we develop a separate *refinement* step to effectively transform c into p. Finally, the set of large cliques will give rise to a set of large frequent patterns, and we could follow this strategy to find the *top-k long maximal frequent patterns*.

Recapitulating the above discussions, we outline the contributions made in this paper as follows.

First, unlike existing bottom-up approaches, we try to solve the frequent pattern mining problem from a totally different perspective that is *top-down* in nature. In this way, we avoid the intrinsic complexity of generating the vast majority of frequent sub-patterns while still being able to reach those interesting long ones.

Second, an innovative two-pronged frequent pattern mining framework is introduced. We transform the transaction database into a pattern graph and perform mining on it instead. As the pattern graph size is only determined by the number of frequent items in the database, the time spent to work on these graphs is independent of either the database size (i.e., number of transactions) or the complete pattern set size (as in existing bottom-up approaches). This is very desirable in terms of computation time.

Third, we test our proposed algorithm by conducting extensive experiments on various real datasets. The results show that our top-down algorithm is able to accurately pinpoint top-k long maximal frequent patterns when state-of-the-art methods fail to generate any interesting results.

The paper is organized as follows. Section II gives the preliminaries, including definitions of the problem and the pattern graph we will be using. We introduce the overall top-down mining framework in Section III, followed by Section IV and Section V that focus on clique detection and candidate refinement, respectively. Section VI presents experimental results. Section VII concludes this study.

II. Preliminaries

A. Problem Definition

Let $I = \{i_1, i_2, \ldots, i_m\}$ be the set of all the m items in a given database, then a *pattern* or an *itemset* refers to a subset of *I*. Particularly, in our paper, a *pattern* including l items is called an l-item *pattern*. The database $D = \{t_1, t_2, \ldots, t_n\}$ is basically a collection of itemsets. The support set D_p of a pattern p is the set of transactions that contain the entire p, i.e., $D_p = \{t_i \mid p \subseteq t_i \text{ and } t_i \in D\}$, whose size is $|D_p|$.

Definition 1: **Frequent Pattern**
Given a transaction database D, a pattern p is frequent if and only if $|D_p| \geq \sigma$, where σ is a user-specified support level threshold that satisfies $0 \leq \sigma \leq |D|$.

As we have already mentioned, since mining of the complete set of frequent patterns might not always be doable and long patterns are in general more interesting, our discussions will be focusing on large-size frequent patterns. Specifically, our objective here is to mine top-k maximal frequent patterns, where top-k refers to the k patterns with biggest length, and a frequent pattern is *maximal* if there is no super-pattern of it which is also frequent. Also, without loss of generality, our method can be applied to find top-k maximal frequent patterns covering any particular item i or a collection of predetermined items in the database.

B. Pattern Graph Construction

Most of the existing pattern mining algorithms produce candidate patterns by growing one item at a time from scratch. This is undesirable and we went on to consider the possibility of adding several items together in one shot. Based on the downward closure property, these several added items must form a frequent pattern by themselves. This hints us to first mine all small frequent patterns P^s up to a predetermined size s, and then different patterns in P^s can be further combined to grow the target pattern in a much faster manner. To accomplish this, we will make use of a so-called *pattern graph* to register the mutual relationships among items in a transaction database, and it can guide us through the search for potential long frequent patterns.

We draw a pattern graph to understand when several small patterns can be combined and merged together. Essentially, a pattern graph is a compact representation of the complete set P^s of s-item frequent patterns. To start with, each item is represented by a node in the pattern graph, and an edge is drawn between two nodes if the corresponding items co-exist in at least one pattern of P^s. To measure how often two items appear together with each other, we also associate a weight with each edge that equals the number of times its two end items show up in the same pattern of P^s.

Table I and Figure 1 illustrate an example of constructed pattern graphs. The original transaction database is shown

Table I
Transaction database example

Transactions	2-item patterns ($\sigma = 1$)	3-item patterns ($\sigma = 1$)
$\{a, b, c\}$ $\{b, c, d\}$ $\{b, d, e\}$ $\{a, c, e, f\}$	*ab, ac, ae,* *af, bc, bd,* *be, cd, ce,* *cf, de, ef*	*abc, ace,* *acf, aef,* *bcd, bde,* *cef*

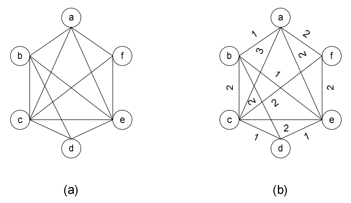


Figure 1. (a) 2-item Pattern Graph (b) 3-item Pattern Graph

in Table I, where 2-item and 3-item frequent patterns when $\sigma = 1$ are also given. With such a database, Figure 1 draws the pattern graphs derived from 2-item and 3-item frequent patterns, respectively. For example, *b* and *d* are both in a 3-item *bcd*, and thus there is an edge *b-d* in the 3-item pattern graph. Moreover, since *b* and *d* both exist in another 3-item *bde*, It means that the weight of edge *b-d* in the 3-item pattern graph is 2. As we can see, the size of a pattern graph is independent of the number of transactions as well as the choice of s. Thus, we can compress a huge transaction database into a compact representation without mining every single pattern targeted by traditional algorithms.

Taking Figure 1 as an example, we examine the condition under which several small s-item patterns can be merged into a large one. First, we have a frequent pattern *acef* in the transaction database; now, it is interesting to see that *acef* happens to correspond to a clique in both 2-item and 3-item pattern graphs. We formalize this observation into the following lemma, and it is useful in guiding us through the search for large-sized maximal frequent patterns.

Lemma 1: If a pattern $p = \{i_{q_1}, i_{q_2}, ..., i_{q_n}\}$ is frequent, then in the s-item pattern graph constructed, there should be a corresponding clique consisting of all items in p, plus that the weight of each edge in the clique is greater than or equal to $\binom{n-2}{s-2}$.

Following above discussions, in an s-item pattern graph, if a large clique meets the condition specified in Lemma 1, then it may quite likely lead to a potential large frequent pattern. Although this condition is not sufficient, it is a very good starting point and our empirical studies have confirmed the pruning power of this compact but surprisingly effective graph representation. With the pattern graph, instead of walking through the pattern growth procedure step by step, we can directly jump to the top of it as indicated by the clique constraint and start exploration right from there; this is a clear advantage offered by our top-down mining strategy.

Transactions:	All 3-item frequent patterns ($\sigma = 1$):
{a, b, c, d}	
{a, b, c, e}	abc, abd, abe, abf, acd, ace, ade,
{a, b, d, e, f}	adf, aef, bcd, bce, bde, bdf, bef,
{c, d, e, g}	cde, cdg, ceg, def, deg

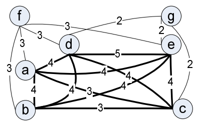


Figure 2. A False Positive Example

III. Max-Clique: A Top-Down Graph-based Frequent Pattern Mining Framework

The overall working flow of our mining algorithm has two stages: clique detection and candidate refinement.

Clique Detection. We first mine all frequent patterns of size s and construct the s-item pattern graph for the database. We shall further discuss the choice of s in Section IV. Now, the mining of long maximal frequent patterns in the database becomes the problem of finding large-sized maximal cliques as potential candidates. Here, the nicety is that pattern graph size is only determined by the number of frequent items in the database (which should be a moderate number), and thus finding maximal cliques is independent of either the database size (i.e., number of transactions) or the complete pattern set size. This is very desirable in terms of processing time. Note that, here we only target maximal cliques, since a clique with m vertices could generate 2^m cliques as its subparts and we cannot afford checking all including non-maximal ones.

Candidate Refinement. From Lemma 1, we know that a maximal frequent pattern p will surely induce a clique that satisfies certain constraints in the pattern graph. However, this clique might not be maximal: it is possible that there exists a set of other items $I(o)$ which seemingly pass the connectivity and weight constraints together with $I(p)$ as in Lemma 1, but actually cannot be merged with $I(p)$ to form a bigger frequent pattern. For example, in Figure 2, $I(p) = \{a, b, c, e\}$ is a frequent pattern, but the maximal clique candidate is *abcde* while *abcde* is not frequent and unrelated items $I(o) = \{d\}$ have been incorrectly merged. To cope with this possibility, we have to come up with a refinement step that can drop $I(o)$ out of the clique candidate c and recover the pattern p we want. Due to the effective pruning performed by pattern graph, it is very hard for a relatively small pattern p to combine with a substantial number of unrelated items and form a large-sized maximal clique. Thus, the item difference between the observed large clique candidate c and actual target long pattern p should not be too big. More details will be covered in Section V.

IV. Clique Detection

As we introduced above, the candidate generation step detects cliques in the pattern graph. Since our objective is to mine large patterns without going through the time-consuming pattern growth procedure, naturally, we want to find those maximal cliques with large size. Therefore, what we will be looking for are those largest maximal cliques in the pattern graph, say the top-k largest. This is a classic NP-hard problem. However, as we have already mentioned, the size of the pattern graph generated for a particular database is independent of the database size as well as the value of s when constructing the s-item pattern graph. To be more specific, it should be no greater than the total number of frequent items in the database. With a moderate graph size, it allows for fast deployment of clique detection routines.

Bearing this in mind, the algorithmic details of finding largest cliques is not a particular concern of this paper. For experiments, we will use the free software *igraph* introduced in [1]. Also, the weight constraint stated in Lemma 1 could serve as a further pruning condition to integrate into any existing clique detection method.

A. The choice of s

An important parameter we need to decide in our method is the value of s when constructing an s-item pattern graph. Naturally, one might expect that the pattern graph's pruning power would become stronger as s gets larger. For instance, in the example of Figure 1, the 2-item pattern graph on the left indicates that *abce* is a possible clique candidate; however, *abce* will be disqualified in the 3-item pattern graph on the right, since the weight on *ab* is 1, which is smaller than $\binom{4-2}{3-2} = 2$ and thus violates the condition specified in Lemma 1. This also applies to the clique *bcde*.

However, the construction of an s-item pattern graph requires us to mine from the database all frequent patterns up to length s, and this will for sure cost more computational resources when s gets larger. In this paper, we intend to use up to 3-item frequent patterns for pattern graph construction, as they can be computed quickly and experimental studies have also confirmed the strength of their pruning power. Of course, things always have to be balanced, and one can freely choose other values of s to trade off the construction time and pruning power of pattern graphs.

V. Candidate Refinement

After detection of potential candidates, in this section, we target the problem that a candidate found may not identically correspond to a real frequent pattern. Briefly mentioned in Section III, we know that any maximal clique c we observe in the pattern graph is the combination of a true frequent pattern p and some other unrelated itemset o, where o is the root cause of possible false positives. In such a relationship:

$$c = p \cup o,$$

two interesting questions to consider are:

1) How large is the true pattern part p.
2) There might be multiple ways of writing c as the combination of a pattern and other items, i.e., $c = p_1 \cup o_1 = p_2 \cup o_2 = \cdots$.

For question 1, because of the usage of pattern graphs, p should not be too small and o should not be too large; if $o = \phi$, then there is no false positive. Based on this assumption, our algorithm works as follows. We extract top-k' largest maximal cliques from the pattern graph, and try to recover the top-k longest maximal frequent patterns from them. Here, $k' \geq k$ is a parameter, and our belief is that longest maximal patterns should appear as largest maximal cliques, because otherwise a shorter pattern needs to absorb more unrelated items in order to make its clique as large as that of those longer patterns, and this is not very likely due to the filtering of pattern graphs.

For question 2, we need to keep in mind that our candidate refinement procedure should be able to recover multiple possible patterns from a single clique. To make it clearer, let us still look at the example in Figure 2. As we can see, patterns $\{a, b, d, e, f\}$ and $\{c, d, e, g\}$ have their own maximal cliques *abdef* and *cdeg*. However, *abcde* is a maximal clique of size 5 and the weight of each link is no less than 3; according to Lemma 1, it should be a qualified candidate, although $\{a, b, c, d, e\}$ is a false positive. We have two true frequent patterns embedded in the clique: $\{a, b, c, d\}$ and $\{a, b, c, e\}$ that need to be recovered, nonetheless.

Now let us take a closer look at how to actually implement refinement. For a clique candidate c, our major concern is to correctly identify the set of unrelated items o and remove them, so that we can find the target frequent pattern p. We propose a heuristic below to guide us through. Note that, although the heuristic is pretty effective in deciding the correct removal order as our experiments show, it is still possible that it will make mistakes and remove the wrong items; to this extent, we need a complementary phase to add these items back after the removal happens. Also, as an answer to the question 2 in above, we will add some amount of randomization into the whole process so that our search should not just try one particular direction. Two subsections are given next to go over more details about each phase in this candidate refinement step.

A. Item Removal

The open question in this first phase is about the order in which we remove different items. Definitely it is not ideal if one just randomly picks items and removes them one by one. Before moving to discussions on our strategy, we first introduce an alternative way of counting a candidate pattern c's occurrences in the input database, where c corresponds to a maximal clique detected before it is passed to the refinement step. Specifically, we will count not only transactions that contain all items in c, but also those including a substantial portion of c. The size of this portion is determined by a parameter t, i.e., a transaction would only be counted if it has more than t fraction of items that are present in the itemset of c. To formalize it, we call each of these transactions a t-valid supporting transaction of c with regard to the threshold t, which collectively form c's t-valid support set.

Definition 2: **t-valid support set**

Given a candidate pattern c, its t-valid support set includes all t-valid supporting transactions of c in the database, where t is a percentage parameter that is often set close to 1.

Given the t-valid support set of a candidate c, for each item i in the candidate, we count the number of times it appears in some t-valid supporting transaction. This counting indicates the frequency i appears in c's t-valid support set and is used to assign removal priority to different items.

The reasoning is as follows. Since our target p is a subpattern of the candidate c and p is usually not too different from c, a t-valid supporting transaction "approximately" covering c actually corresponds to one that may potentially support p after some items are removed. If an item appears in most of the transactions in the t-valid support set, then it is less likely that it does not appear in the supporting transactions of the target pattern we want to recover, and consequently, we may want to eliminate those items that appear less often in the t-valid set, which probably have been wrongly merged into the target pattern, giving rise to a false positive.

The overall removing procedure works as follows. First, given a candidate c, if the corresponding pattern is frequent, then it means it is already a valid long maximal frequent pattern. Otherwise, items are set to be removed from c based on the priority specified above. Here, in order to not have the removal follow just one direction and thus only recover one potential pattern, we integrate a randomizing factor $r \in [0, 1)$ as described below. When taking on an item in the ordered list pending removal, there is a chance of r that we bypass it and jump to the next item in the list. In this way, a diversification effect exists if we iterate the whole process several times, since now the removal can go into multiple directions while still preserving the nice property of prioritizing those more promising ones. The removing of items is stopped as soon as the new candidate c' becomes frequent. Because we are trying different ways of removing items from a candidate to achieve maximal frequent patterns, it is not meaningful to remove items from an already frequent pattern as this can only generate non-maximal patterns. In other words, items removed beyond this point will be recovered by the later extension phase anyway, and thus it is better to retain them in the first place.

Also, in order to accelerate the process, we do not have to strictly follow the above process by removing one item at a time. Specially, in each iteration, we intend to remove $m_{del} > 1$ items, and our experiments confirm that the resulting quality is almost as good. It is possible that removing multiple items in one round may not let us stop right at the boundary of frequent patterns, however, this is OK given the adding phase we will be discussing next, where items unnecessarily removed are finally recovered.

B. Candidate Extension

After the last phase of removing items, we get a new candidate c' that is frequent. The process of extending c' to make it maximal is much more straightforward. We project the original database against c' to drop all transactions that do not contain c', remove items that appear in c', and then mine maximal frequent patterns on the projected database. The result patterns are concatenated with c' to form maximal frequent patterns of the original database at last. Since we stop item removal at the frequent boundary (or, almost at the boundary when multiple items are removed in each round), it is reasonable to expect that the c'-projected database would be small and result maximal patterns mined from it would be short. To this extent, we will just utilize the existing frequent pattern mining method to work in this phase.

VI. Experimental Results

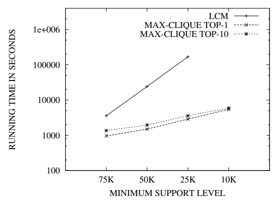
In this section, we test our proposed algorithm in terms of efficiency and effectiveness. For the purpose of evaluation, we use LCM [3], the winner of FIMI 2004, as our baseline. As one of the traditional bottom-up mining methods, LCM traverses the whole pattern space to mine the complete set and thus we can get top-k longest frequent patterns.

As we mentioned in Section II, our method is in fact general enough to specifically find patterns containing particular items. Here, to extensively experiment with our *Max-Clique* algorithm, we will sample a subset of items in the test datasets, mine the top-k maximal frequent patterns covering each of them, and then report the result quality and running time of *Max-Clique* on an average basis. Specifically, 20 items have been sampled from both test sets. All three steps, including pattern graph construction, clique detection and candidate refinement, are accounted for when recording the running time of *Max-Clique*.

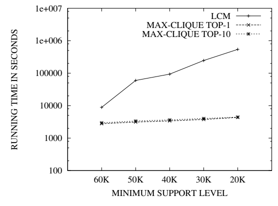
We use two real datasets: *chess* and *accidents*, which are both available from the Frequent Itemset Mining Dataset Repository[1]. The *chess* dataset contains 3,196 transactions, and each has a length of 37. There are 75 items in total. Since we are particularly interested in large data, we replicate everything in *chess* for 100 times. The *accidents* dataset [4] records 340,184 traffic accident records with an average size of 45, and it includes 572 distinct items.

All experiments are done on a Debian GNU/Linux server with two dual-core Xeon 3.0GHz CPUs and 16GB main memory. The program is written in C++. *Max-Clique* has a few parameters set as follows: the number k' of largest maximal cliques to extract is $2k$, i.e., twice the number of patterns we want to find; the threshold t in t-valid support set is 0.8; the randomizing factor r when recovering multiple patterns from a single clique is 0.3; and the number m_{del} of items removed in one round for acceleration purposes is defaulted to 5 except in the sensitivity analysis.

[1]http://fimi.cs.helsinki.fi/data/

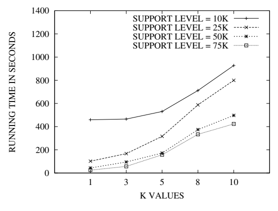


(a) Dataset *accidents*

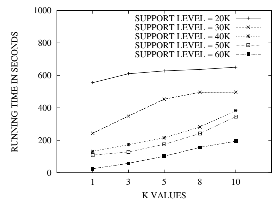


(b) Dataset *chess*

Figure 3. Efficiency on Different Support Levels



(a) Dataset *accidents*



(b) Dataset *chess*

Figure 4. Efficiency on Different k Values

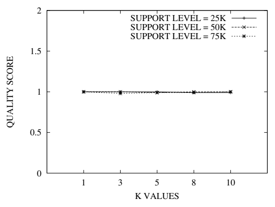
A. Efficiency Analysis

The running time of both *Max-Clique* and LCM, drawn in Figures 3(a) and 3(b) for *accidents* and *chess* respectively, increase with decreasing support level. k is set to 1 and 10. As we can see, the execution time of LCM has a dramatic upward trend toward the right hand side of the graph: When the support level goes to as low as 10k, we had to omit one point for LCM since the program could not finish within 10 days. In comparison, *Max-Clique* remains stable, due to the advantage of our top-down mining framework that only picks the "iceberg" of the pattern space, so it is orders of magnitude faster than LCM.

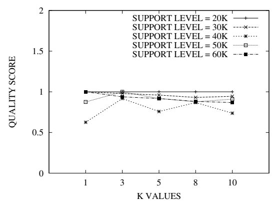
Unlike LCM, since the running time of *Max-Clique* depends on the choice of k, it is interesting to see how this trend looks like. The efficiency on two datasets are shown in Figure 4(a) and Figure 4(b). Note that, we have only included the execution time of clique detection and candidate refinement in these two graphs, because varying k has no impact on the execution time of pattern graph construction. As can be seen in the figures, in most cases we are able to get the top-10 target maximal frequent patterns within 10 minutes, even if the support level is very low. Meanwhile, as expected, the curves move like linearly with regard to k.

B. Quality Measurement

In order to quantify the effectiveness of our approach, we use a ranking based precision measure. First, each of the top-k maximal frequent patterns in the database is ranked in descending order of size and associated with a weight. Specifically, the i-th pattern in the list will have a weight of



(a) Dataset *accidents*



(b) Dataset *chess*

Figure 5. Quality Measure on Different k Values

$k - i + 1$. The score we use to measure the result quality is computed as follows,

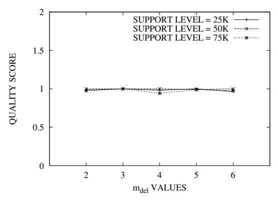
$$Q = \frac{\sum_i (I_i * Weight_i)}{\sum_i Weight_i},$$

where I_i is an indicator function denoting whether *Max-Clique* can successfully output the i-th pattern in the true top-k list. The idea behind this measure is to attach more importance to larger patterns, because presumably, more information will be lost if *Max-Clique* cannot catch them.

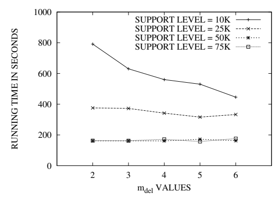
Figures 5(a) and 5(b) depict the quality scores over different k values for both datasets. Note that, in Figure 5(a), we do not have data for the support level of 10k, as LCM cannot finish within 10 days to provide the ground truth for measurement. Overall, *Max-Clique* is pretty effective in mining the top-k patterns, given the fact that it can run much faster than LCM. However, we also observe that the algorithm's performance is a little worse on *chess*, and this is due to the reason that patterns in *chess* overlap more often, which is the case we discussed in Section V that one has to recover multiple patterns from a single clique. Though we proposed the randomized scheme to alleviate this issue, it is hard to eliminate the problem completely, and we would like to investigate this further in future works.

C. Parameter Sensitivity Analysis

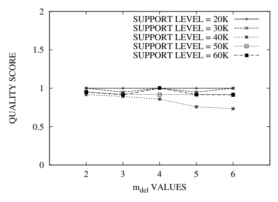
Due to space limit, we only focus on m_{del}, parameter to accelerate item removal, in this subsection. We focus on top-5 maximal frequent patterns in this experiment. Let us first look at the results on *accidents*: Figure 6(a) illustrates the quality scores over various values of m_{del}, and Figure 6(b) shows the corresponding running time. As seen from the graphs, our algorithm works nearly perfect at all tested support levels even when we try to remove $m_{del} = 6$ items at a time. In the meantime, the procedure becomes faster if m_{del} gets larger. The cutting of running time is more obvious when the support level is low, say 10k, when patterns are bigger and it takes more time to refine candidates. The fact that removing more items in one round may result in a longer candidate extension phase later also contributes to the less effectiveness of increasing m_{del} when support is higher. Nonetheless, the acceleration of item removal seems quite beneficial in overall. Similar results are also seen for the *chess* dataset in Figures 6(c) and 6(d).



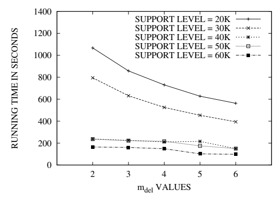
(a) Quality (*accidents*)



(b) Efficiency (*accidents*)



(c) Quality (*chess*)



(d) Efficiency (*chess*)

Figure 6. Quality Measure and Efficiency on Different m_{del}

VII. Conclusion

In this paper, we examine an important issue in frequent pattern mining: The difficulty to mine very long patterns in a large database. We propose a novel *top-down* mining framework which starts working from the top of the pattern lattice to discover the *top-k* longest maximal frequent patterns. This is achieved by mapping the database to a so-called *pattern graph* and performing *maximal clique detection* on it. We introduce a separate *refinement* stage to further transform these clique candidates into true maximal patterns.

Acknowledgment

This work is supported in part by NSF through grants IIS 0905215, DBI-0960443, OISE-0968341 and OIA-0963278. We thank Karolien Geurts for providing the dataset *accidents*.

References

[1] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal*, vol. Complex Systems, p. 1695, 2006. [Online]. Available: http://igraph.sf.net

[2] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules in large databases," in *VLDB*, 1994, pp. 487–499.

[3] T. Uno, M. Kiyomi, and H. Arimura, "Lcm ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets," in *FIMI*, 2004.

[4] K. Geurts, G. Wets, T. Brijs, and K. Vanhoof, "Profiling high frequency accident locations using association rules," in *Proceedings of the 82nd Annual Transportation Research Board, Washington DC. (USA), January 12-16*, 2003, p. 18.