FIMI'03: Workshop on Frequent Itemset Mining Implementations

FIMI Repository: http://fimi.cs.helsinki.fi/
FIMI Repository Mirror: http://www.cs.rpi.edu/~zaki/FIMI03/

Bart Goethals
HIIT Basic Research Unit
Department of Computer Science
University of Helsinki, Finland
bart.goethals@cs.helsinki.fi

Mohammed J. Zaki
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, New York, USA
zaki@cs.rpi.edu

1 Why Organize the FIMI Workshop?

Since the introduction of association rule mining in 1993 by Agrawal Imielinski and Swami [3], the frequent itemset mining (FIM) tasks have received a great deal of attention. Within the last decade, a phenomenal number of algorithms have been developed for mining all [3–5, 10, 18, 19, 21, 23, 26, 28, 31, 33], closed [6, 12, 22, 24, 25, 27, 29, 30, 32] and maximal frequent itemsets [1, 2, 7, 11, 15–17, 20, 35]. Every new paper claims to run faster than previously existing algorithms, based on their experimental testing, which is oftentimes quite limited in scope, since many of the original algorithms are not available due to intellectual property and copyright issues. Zheng, Kohavi and Mason [34] observed that the performance of several of these algorithms is not always as claimed by its authors, when tested on some different datasets. Also, from personal experience, we noticed that even different implementations of the same algorithm could behave quite differently for various datasets and parameters.

Given this proliferation of FIM algorithms, and sometimes contradictory claims, there is a pressing need to benchmark, characterize and understand the algorithmic performance space. We would like to understand why and under what conditions one algorithm outperforms another. This means testing the methods for a wide variety of parameters, and on different datasets spanning dense and sparse, real and synthetic, small and large, and so on.

Given the experimental, algorithmic nature of FIM (and most of data mining in general), it is crucial that other researchers be able to independently verify the claims made in a new paper. Unfortunately, the FIM community (with few exceptions) has a very poor track record in this regard. Many new algorithms are not available even as an executable, let alone the source code. How many times have we heard "this is proprietary software, and not available." This is not the way other sciences work. Independent verifiability is the hallmark of sciences like physics, chemistry, biology, and so on. One may argue, that the nature of research is different, they have detailed experimental procedure that can be replicated, while we have algorithms, and there is more than one way to code an algorithm. However, a good example to emulate is the bioinformatics community. They have espoused the open-source paradigm with more alacrity than we have. It is quite common for journals and conferences in bioinformatics to *require* that software be available. For example, here is a direct quote from the journal *Bioinformatics* (http://bioinformatics.oupjournals.org/):

> Authors please note that software should be available for a full 2 YEARS after publication of the manuscript.

We organized the FIMI workshop to address the three main deficiencies in the FIM community:

- Lack of publicly available implementations of FIM algorithms

- Lack of publicly available "real" datasets
- Lack of any serious performance benchmarking of algorithms

1.1 FIMI Repository

The goals of this workshop are to find out the main implementation aspects of the FIM problem for all, closed and maximal pattern mining tasks, and evaluating the behavior of the proposed algorithms with respect to different types of datasets and parameters. One of the most important aspects is that only open source code submissions are allowed and that all submissions will become freely available (for research purposes only) on the online FIMI repository along with several new datasets for benchmarking purposes. See the URL: `http://fimi.cs.helsinki.fi/`.

1.2 Some Recommendations

We strongly urge all new papers on FIM to provide access to source code or at least an executable immediately after publication. We request that researchers to contribute to the FIMI repository both in terms of algorithms and datasets. We also urge the data mining community to adopt the open-source strategy, which will serve to accelerate the advances in the field. Finally, we would like to alert reviewers that the FIMI repository now exists, and it contains state-of-the-art FIM algorithms, so there is no excuse for a new paper to not do an extensive comparison with methods in the FIMI repository. Such papers should, in our opinion, be rejected outright!

2 The Workshop

This is a truly unique workshop. It consisted of code submission as well as a paper submission describing the algorithm and a detailed performance study by the authors on publicly provided datasets, along with a detailed explanation on when and why their algorithm performs better than existing implementations. The submissions were tested independently by the co-chairs, and the papers were reviewed by members of the program committee. The algorithms were judged for three main tasks: all frequent itemsets mining, closed frequent itemset mining, and maximal frequent itemset mining.

The workshop proceedings contains 15 papers describing 18 different algorithms that solve the frequent itemset mining problems. The source code of the implementations of these algorithms is publicly available on the FIMI repository site.

The conditions for "acceptance" of a submission were as follows: i) a correct implementation for the given task, ii) an efficient implementation compared with other submissions in the same category or a submission that provides new insight into the FIM problem. The idea is to highlight both successful and unsuccessful but interesting ideas. One outcome of the workshop will be to outline the focus for research on new problems in the field.

In order to allow a fair comparison of these algorithms, we performed an extensive set of experiments on several real-life datasets, and a few synthetic ones. Among these are three new datasets, i.e. a supermarket basket dataset donated by Tom Brijs [9], a dataset containing click-stream data of a Hungarian on-line news portal donated by Ferenc Bodon [8], and a dataset containing Belgian traffic accident descriptions donated by Karolien Geurts [13].

2.1 Acknowledgments

We would like to thank the following program committee members for their useful suggestions and reviews:

- Roberto Bayardo, IBM Almaden Research Center, USA
- Johannes Gehrke, Cornell University, USA
- Jiawei Han, University of Illinois at Urbana-Champaign, USA
- Hannu Toivonen, University of Helsinki, Finland

We also thank Taneli Mielikäinen and Toon Calders for their help in reviewing the submissions.

We extend our thanks to all the participants who made submissions to the workshop, since their willingness to participate and contribute source-code in the public domain was essential in the creation of the FIMI Repository. For the same reason, thanks are due to Ferenc Bodon, Tom Brijs, and Karolien Geurts, who contributed new datasets, and to Zheng, Kohavi and Mason for the KDD Cup 2001 datasets.

3 The FIMI Tasks Defined

Let's assume we are given a set of items $\mathcal{I}$. An *itemset* $I \subseteq \mathcal{I}$ is some subset of items. A *transaction* is a couple $T = (tid, I)$ where tid is the transaction identifier and I is an itemset. A transaction $T = (tid, I)$ is said to *support* an itemset X, if $X \subseteq I$. A *transaction database* $\mathcal{D}$ is a set of transactions such that each transaction has a unique identifier. The *cover*

of an itemset X in $\mathcal{D}$ consists of the set of transaction identifiers of transactions in $\mathcal{D}$ that support X: $cover(X, \mathcal{D}) := \{tid \mid (tid, I) \in \mathcal{D}, X \subseteq I\}$. The *support* of an itemset X in $\mathcal{D}$ is the number of transactions in the cover of X in $\mathcal{D}$:

$$support(X, \mathcal{D}) := |cover(X, \mathcal{D})|.$$

An itemset is called *frequent* in $\mathcal{D}$ if its support in $\mathcal{D}$ exceeds a given minimal support threshold σ. $\mathcal{D}$ and σ are omitted when they are clear from the context. The goal is now to find all frequent itemsets, given a database and a minimal support threshold.

The search space of this problem, all subsets of $\mathcal{I}$, is clearly huge, and a frequent itemset of size k implies the presence of 2^k-2 other frequent itemsets as well, i.e., its nonempty subsets. In other words, if frequent itemsets are long, it simply becomes infeasible to mine the set of all frequent itemsets. In order to tackle this problem, several solutions have been proposed that only generate a representing subset of all frequent itemsets. Among these, the collections of all *closed* or *maximal* itemsets are the most popular.

A frequent itemset I is called *closed* if it has no frequent superset with the same support, i.e., if

$$I = \bigcap_{(tid, J) \in cover(I)} J$$

An frequent itemset is called *maximal* if it has no superset that is frequent.

Obviously, the collection of maximal frequent itemsets is a subset of the collection of closed frequent itemsets which is a subset of the collection of all frequent itemsets. Although all maximal itemsets characterize all frequent itemsets, the supports of all their subsets is not available, while this might be necessary for some applications such as association rules. On the other hand, the closed frequent itemsets form a lossless representation of all frequent itemsets since the support of those itemsets that are not closed is uniquely determined by the closed frequent itemsets. See [14] for a recent survey of the FIM algorithms.

4 Experimental Evaluation

We conducted an extensive set of experiments for different datasets, for all of the algorithms in the three categories (all, closed and maximal). Figure 1 shows the data characteristics.

Our target platform was a Pentium4, 3.2 GHz Processor, with 1GB of memory, using a WesternDigital IDE 7200rpms, 200GB, local disk. The operating system was Redhat Linux 2.4.22 and we used gcc 3.2.2 for the compilation. Other platforms were also tried, such as an older dual 400Mhz Pentium III processors with 256MB memory, but a faster SCSI 10,000rpms disk. Independent tests were also run on quad 500Mhz Pentium III processors, with 1GB memory. There were some minor differences, which have been reported on the workshop website. Here we refer to the target platform (3.2Ghz/1GB/7200rpms).

All times reported are *real* times, including system and user times, as obtained from the unix `time` command. All algorithms were run with the output flag turned on, which means that mined results were written to a file. We made this decision, since in the real world one wants to see the output, and the total wall clock time is the end-to-end delay that one will see. There was one unfortunate consequence of this, namely, we were not able to run algorithms for mining all frequent itemsets below a certain threshold, since the output file exceeded the 2GB file size limit on a 32bit platform. For each algorithm we also recorded its memory consumption using the `memusage` command. Results on memory usage are available on the FIMI website.

For the experiments, each algorithm was allocated a maximum of 10 minutes to finish execution, after which point it was killed. We had to do this to finish the evaluation in a reasonable amount of time. We had a total of 18 algorithms in the *all* category, 6 in the *closed* category, and 8 in the *maximal* category, for a grand total of 32 algorithms. Please note the algorithms eclat_zaki, eclat_goethals, charm and genmax, were not technically submitted to the workshop, however we included them in the comparison since their source code is publicly available. We used 14 datasets, with an average of 7 values of minimum support. With a 10 minute time limit per algorithm, the total time to finish *one round* of evaluation took 31360 minutes of running time, which translates to an upper-bound of 21 days! Since not all algorithms take a full 10 minute, the actual time for one round was roughly 7-10 days.

We should also mention that some algorithms had problems on certain datasets. For instance for mining all frequent itemsets, armor is not able to handle dense datasets very well (for low values of minimum support it crashed for chess, mushroom, pumsb); pie gives a segmentation fault for bms2, chess, retail and the synthetic datasets; cofi gets killed for bms1 and kosarak; and dftime/dfmem crash for accidents, bms1 and retail. For closed itemset mining, fpclose segmentfaults for bms1, bms2, bmspos and retail; borgelt_eclat also has problems with retail. Finally, for maximal set mining, apriori_borgelt crashes for bms1 for low value of support and so does eclat_borgelt for pumsb.

Database	#Items	Avg. Length	#Transactions
accidents	468	33.8	340,183
bms1	497	2.5	59,602
bms2	3341	5.6	77,512
bmspos	1658	7.5	515,597
chess	75	37	3,196
connect	129	43	67,557
kosarak	41,270	8.1	990,002
mushroom	119	23	8,124
pumsb*	2088	50.5	49,046
pumsb	2113	74	49,046
retail	16,469	10.3	88,162
T10I5N1KP5KC0.25D200K	956	10.3	200,000
T20I10N1KP5KC0.25D200K	979	20.1	200,000
T30I15N1KP5KC0.25D200K	987	29.7	200,000

Figure 1. Database Characteristics

4.1 Mining All Frequent Itemsets

Figures 5 and 6 show the timings for the algorithms for mining all frequent itemsets. Figure 2 shows the best and second-best algorithms for high and low values of support for each dataset.

There are several interesting trends that we observe:

1. In some cases, we observe a high initial running time of the highest value of support, and the time drops for the next value of minimum support. This is due to file caching. Each algorithm was run with multiple minimum support values before switching to another algorithm. Therefore the first time the database is accessed we observe higher times, but on subsequent runs the data is cached and the I/O time drops.

2. In some cases, we observe that there is a cross-over in the running times as one goes from high to low values of support. An algorithm may be the best for high values of support, but the same algorithm may not be the best for low values.

3. There is *no one best* algorithm either for high values or low values of support, but some algorithms are the best or runner-up more often than others.

Looking at Figure 2, we can conclude that for high values the best algorithms are either kdci or patricia, across all databases we tested. For low values, the picture is not as clear; the algorithms likely to perform well are patricia, fpgrowth* or lcm. For the runner-up in the low support category, we once again see patricia and kdci showing up.

4.2 Mining Closed Frequent Itemsets

Figures 7 and 8 show the timings for the algorithms for mining closed frequent itemsets. Figure 3 shows the best and second-best algorithms for high and low values of support for each dataset. For high support values, fpclose is best for 7 out of the 14 datasets, and lcm, afopt, and charm also perform well on some datasets. For low values of support the competition is between fpclose and lcm for the top spot. For the runner-up spot there is a mixed bag of algorithms: fpclose, afopt, lcm and charm. If one were to pick an overall best algorithm, it would arguably be fpclose, since it either performs the best or shows up in the runner-up spot, more times than any other algorithm. An interesting observation is that for the cases where fpclose doesn't appear in the table it gives a segmentation fault (for bms1, bms2, bmspos and retail).

4.3 Mining Maximal Frequent Itemsets

Figures 9 and 10 show the timings for the algorithms for mining maximal frequent itemsets. Figure 4 shows the best and second-best algorithms for high and low values of support for each dataset. For high values of support fpmax* is the dominant winner or runner-up. Genmax, mafia and afopt also are worth mentioning. For the low support category fpmax* again makes a strong show as the best in 7 out of 14 databases, and when it is not best, it appears as the runner-up 6 times. Thus fpmax* is the method of choice for maximal pattern mining.

4.4 Conclusions

We presented only some of the results in this report. We refer the reader to the FIMI repository for a more detailed experimental study. The study done by us was also somewhat limited, since we performed only timing and memory usage experiments for given datasets. Ideally, we would have liked to do a more detailed study of the scale-up of the algorithms, and for a variety of different parameters; our preliminary studies show that *none* of the algorithms is able to gracefully scale-up to very large datasets, with millions of transactions. One reason may be that most methods are optimized for in-memory datasets, which points to the area of *out-of-core* FIM algorithms an avenue for future research.

In the experiments reported above, there were no clear winners, but some methods did show up as the best or second best algorithms for both high and low values of support. Both patricia and kdci represent the state-of-the-art in all frequent itemset mining, whereas fpclose takes this spot for closed itemset mining, and finally fpmax* appears to be one of the best for maximal itemset mining. An interesting observation is that for the synthetic datasets, apriori_borgelt seems to perform quite well for all, closed and maximal itemset mining.

We refer the reader to the actual papers in these proceedings to find out the details on each of the algorithms in this study. The results presented here should be taken in the spirit of *experiments-in-progress*, since we do plan to diversify our testing to include more parameters. We are confident that the workshop will generate a very healthy and critical discussion on the state-of-affairs in frequent itemset mining implementations.

To conclude, we hope that the FIMI workshop will serve as a model for the data mining community to hold more such open-source benchmarking tests, and we hope that the FIMI repository will continue to grow with the addition of new algorithms and datasets, and once again to serve as a model for the rest of the data mining world.

References

[1] C. Aggarwal. Towards long pattern generation in dense databases. *SIGKDD Explorations*, 3(1):20–26, 2001.

[2] R. Agrawal, C. Aggarwal, and V. Prasad. Depth First Generation of Long Patterns. In *7th Int'l Conference on Knowledge Discovery and Data Mining*, Aug. 2000.

[3] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

[4] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In U. Fayyad and et al, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI Press, Menlo Park, CA, 1996.

[5] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *20th VLDB Conference*, Sept. 1994.

[6] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *SIGKDD Explorations*, 2(2), Dec. 2000.

[7] R. J. Bayardo. Efficiently mining long patterns from databases. In *ACM SIGMOD Conf. Management of Data*, June 1998.

[8] F. Bodon. A fast apriori implementation. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*, 2003.

[9] T. Brijs, G. Swinnen, K. Vanhoof, and G. Wets. Using association rules for product assortment decisions: A case study. In *Knowledge Discovery and Data Mining*, pages 254–260, 1999.

[10] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *ACM SIGMOD Conf. Management of Data*, May 1997.

[11] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: a maximal frequent itemset algorithm for transactional databases. In *Intl. Conf. on Data Engineering*, Apr. 2001.

[12] D. Cristofor, L. Cristofor, and D. Simovici. Galois connection and data mining. *Journal of Universal Computer Science*, 6(1):60–73, 2000.

[13] 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*, page 18, 2003.

[14] B. Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, transnational University of Limburg, Belgium, 2002.

[15] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *1st IEEE Int'l Conf. on Data Mining*, Nov. 2001.

[16] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *6th International Workshop on High Performance Data Mining*, May 2003.

[17] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all the most specific sentences by randomized algorithms. In *Intl. Conf. on Database Theory*, Jan. 1997.

[18] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM SIGMOD Conf. Management of Data*, May 2000.

[19] M. Houtsma and A. Swami. Set-oriented mining of association rules in relational databases. In *11th Intl. Conf. Data Engineering*, 1995.

[20] D.-I. Lin and Z. M. Kedem. Pincer-search: A new algorithm for discovering the maximum frequent set. In *6th Intl. Conf. Extending Database Technology*, Mar. 1998.

[21] J.-L. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *14th Intl. Conf. on Data Engineering*, Feb. 1998.

[22] F. Pan, G. Cong, A. Tung, J. Yang, and M. Zaki. CARPENTER: Finding closed patterns in long biological datasets. In *ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, Aug. 2003.

[23] J. S. Park, M. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *ACM SIGMOD Intl. Conf. Management of Data*, May 1995.

[24] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *7th Intl. Conf. on Database Theory*, Jan. 1999.

[25] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *SIGMOD Int'l Workshop on Data Mining and Knowledge Discovery*, May 2000.

[26] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *21st VLDB Conf.*, 1995.

[27] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *ACM SIGMOD Intl. Conf. Management of Data*, May 2000.

[28] H. Toivonen. Sampling large databases for association rules. In *22nd VLDB Conf.*, 1996.

[29] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining*, Aug. 2003.

[30] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372-390, May-June 2000.

[31] M. J. Zaki and K. Gouda. Fast vertical mining using Diffsets. In *9th ACM SIGKDD Int'l Conf. Knowledge Discovery and Data Mining*, Aug. 2003.

[32] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithm for closed itemset mining. In *2nd SIAM International Conference on Data Mining*, Apr. 2002.

[33] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *3rd Intl. Conf. on Knowledge Discovery and Data Mining*, Aug. 1997.

[34] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 401–406. ACM Press, 2001.

[35] Q. Zou, W. Chu, and B. Lu. Smartminer: a depth first algorithm guided by tail information for mining maximal frequent itemsets. In *2nd IEEE Int'l Conf. on Data Mining*, Nov. 2002.

	High Support		Low Support	
Database	1st	2nd	1st	2nd
accidents	kdci	eclat_zaki	fpgrowth*	patricia
bms1	patricia	lcm		
bms2	patricia	lcm	lcm	patricia
bmspos	kdci	patricia	fpgrowth*	patricia
chess	patricia	kdci	lcm	patricia
connect	kdci	aim	lcm	kdci
kosarak	kdci	patricia	patricia	afopt
mushroom	kdci	lcm	lcm	kdci
pumsb	patricia	fpgrowth*	mafia	lcm
pumsb*	kdci	aim/patricia	patricia	kdci
retail	patricia	afopt	lcm	patricia/afopt
T10I5N1KP5KC0.25D200K	patricia	fpgrowth*	fpgrowth*	patricia
T20I10N1KP5KC0.25D200K	kdci	apriori_borgelt	fpgrowth*	lcm
T30I15N1KP5KC0.25D200K	kdci	eclat_zaki/apriori_borgelt	apriori_borgelt	fpgrowth*

Figure 2. All FIM: Best (1st) and Runner-up (2nd) for High and Low Supports

	High Support		Low Support	
Database	1st	2nd	1st	2nd
accidents	charm	fpclose	fpclose	afopt
bms1	lcm	fpclose	lcm	fpclose
bms2	lcm	apriori_borgelt	lcm	charm
bmspos	apriori_borgelt	charm/afopt	lcm	charm
chess	lcm	fpclose	lcm	fpclose
connect	fpclose	afopt	lcm	fpclose
kosarak	fpclose	charm	fpclose	afopt
mushroom	fpclose	afopt	fpclose	lcm
pumsb	fpclose/charm	afopt	lcm	fpclose
pumsb*	fpclose	afopt/charm	fpclose	afopt
retail	afopt	lcm	lcm	apriori_borgelt
T10I5N1KP5KC0.25D200K	fpclose	afopt	fpclose	lcm
T20I10N1KP5KC0.25D200K	apriori_borgelt	charm	fpclose	lcm
T30I15N1KP5KC0.25D200K	fpclose	apriori_borgelt	apriori_borgelt	fpclose

Figure 3. Closed FIM: Best (1st) and Runner-up (2nd) for High and Low Supports

	High Support		Low Support	
Database	1st	2nd	1st	2nd
accidents	genmax	fpmax*	fpmax*	mafia/genmax
bms1	fpmax*	lcm	lcm	fpmax*
bms2	afopt	fpmax*	afopt	fpmax*
bmspos	fpmax*	genmax	fpmax*	afopt
chess	fpmax*	afopt	mafia	fpmax*
connect	fpmax*	afopt	fpmax*	afopt
kosarak	fpmax*	genmax	afopt	fpmax*
mushroom	fpmax*	mafia	fpmax*	mafia
pumsb	genmax	fpmax*	fpmax*	afopt
pumsb*	fpmax*	mafia	mafia	fpmax*
retail	afopt	lcm	afopt	lcm
T10I5N1KP5KC0.25D200K	fpmax*	afopt	fpmax*	afopt
T20I10N1KP5KC0.25D200K	apriori_borgelt	genmax	fpmax*	afopt
T30I15N1KP5KC0.25D200K	genmax	fpmax*	apriori_borgelt	fpmax*

Figure 4. Maximal FIM: Best (1st) and Runner-up (2nd) for High and Low Supports

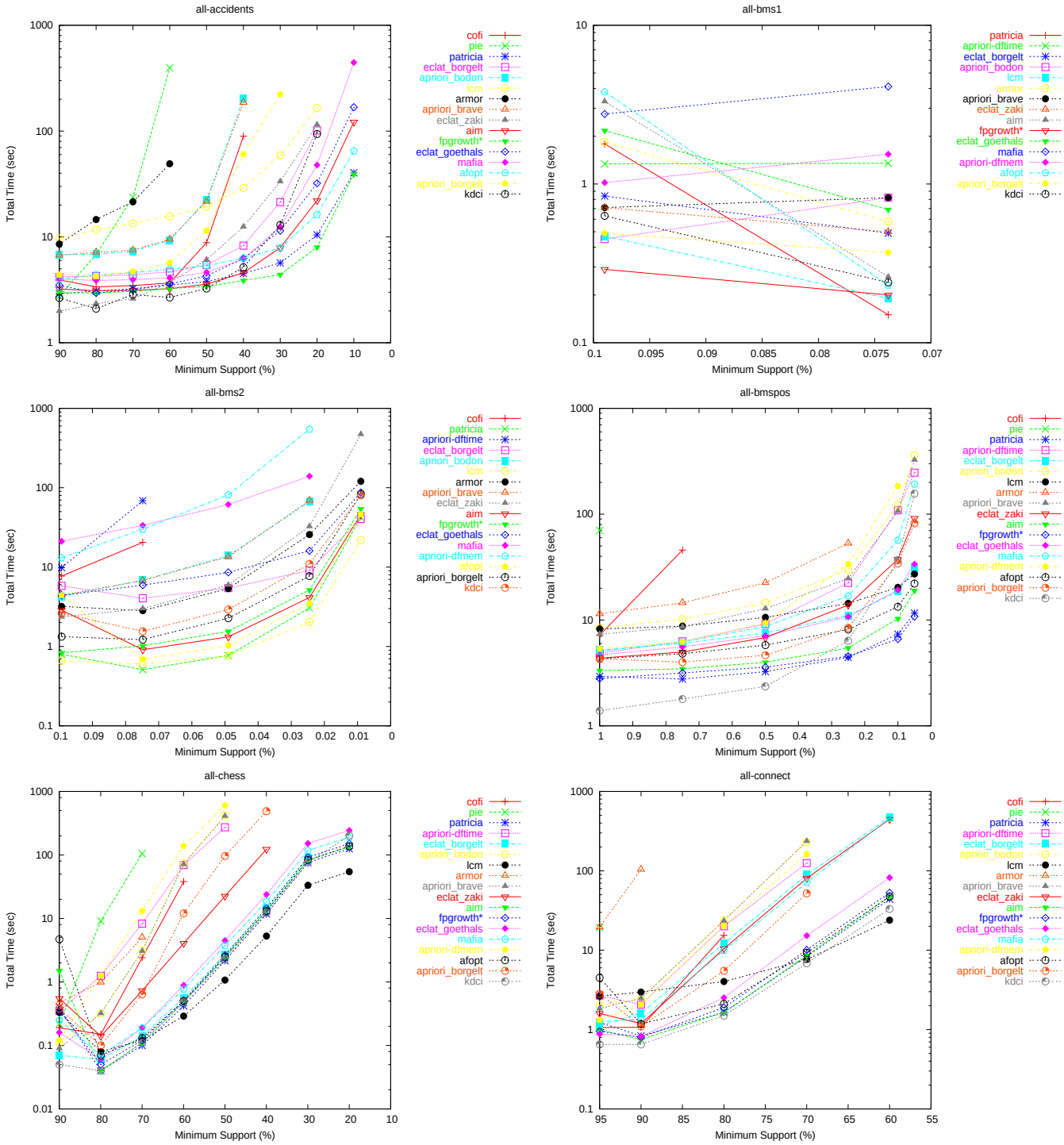


Figure 5. Comparative Performance: All

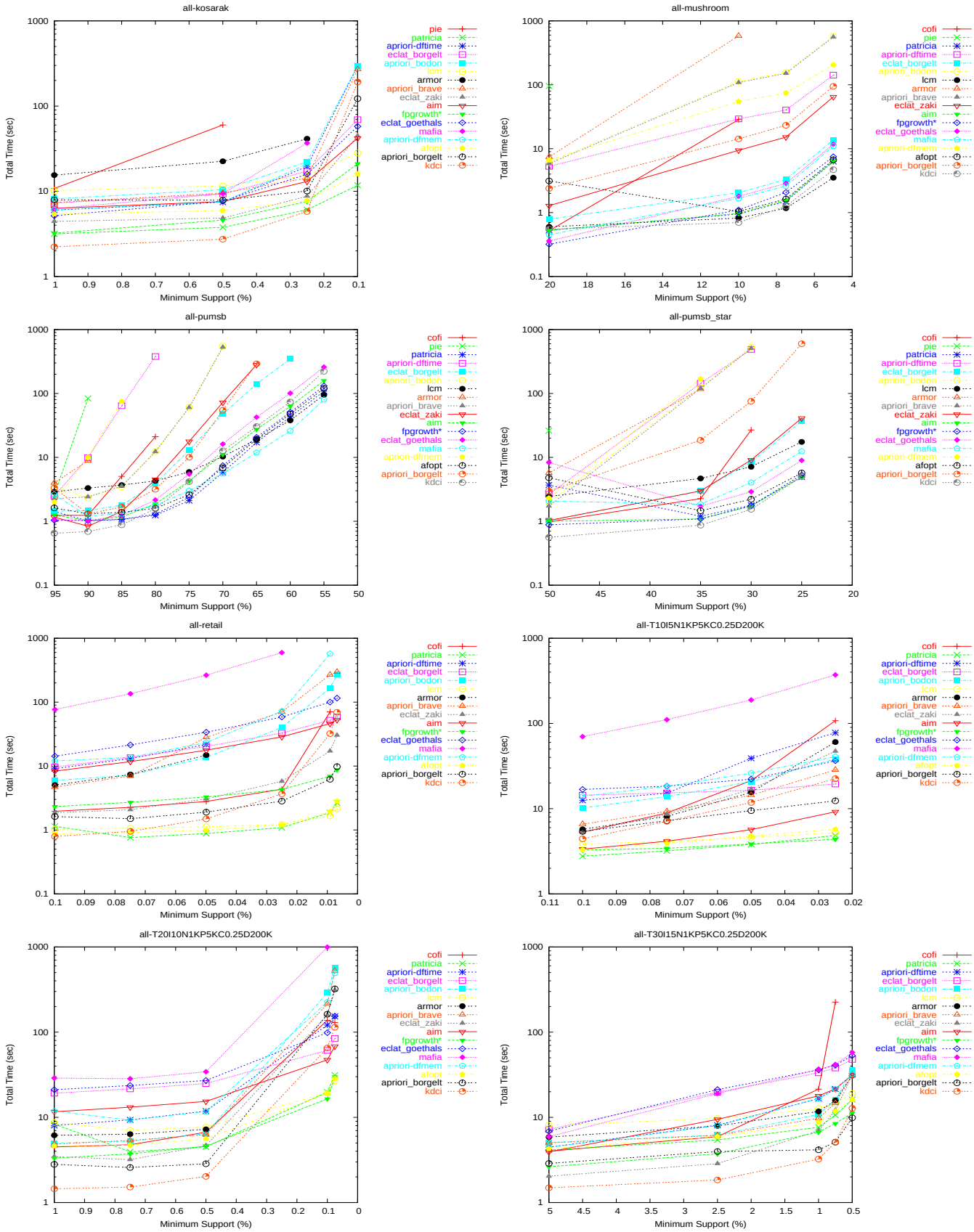


Figure 6. Comparative Performance: All

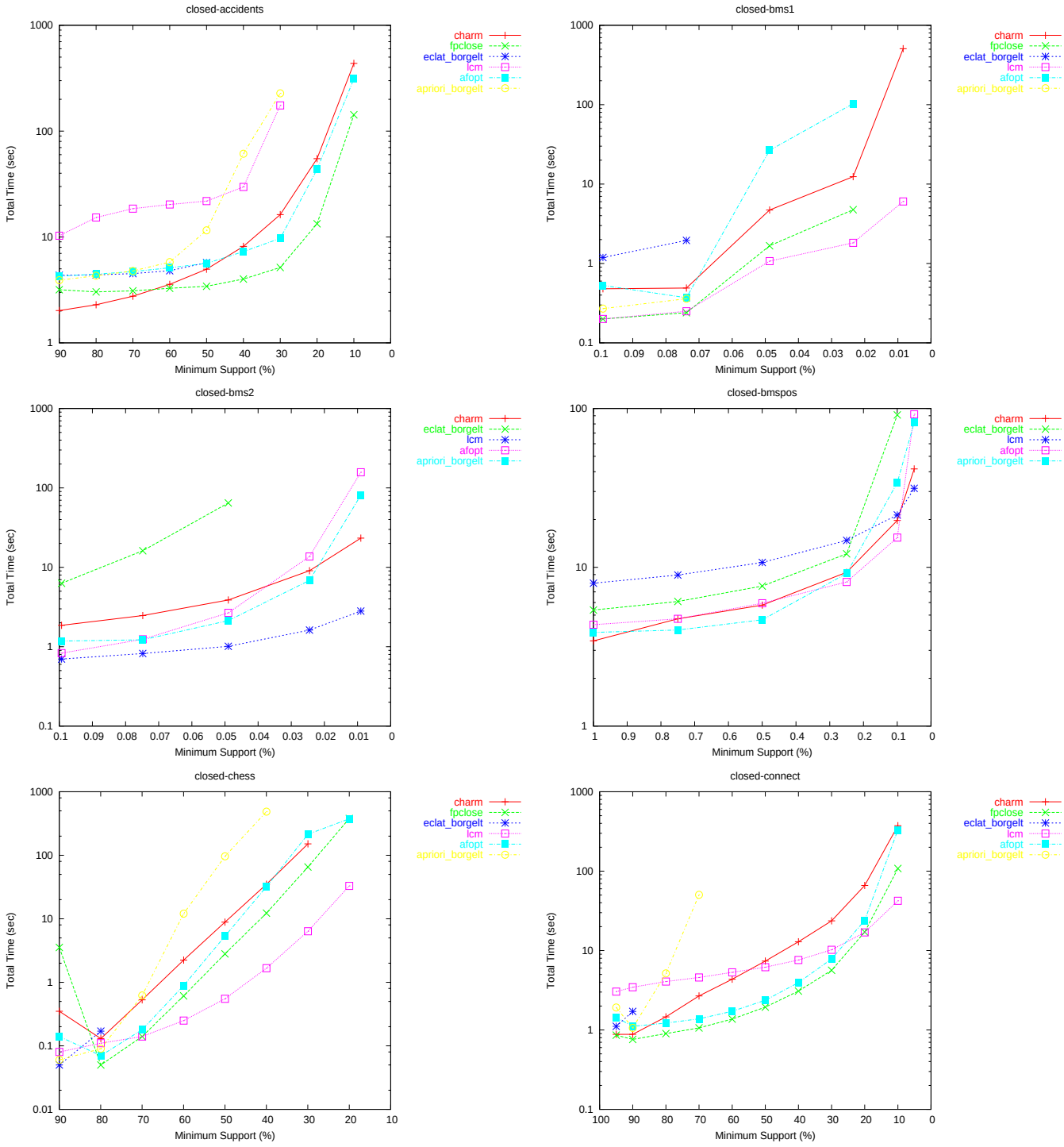


Figure 7. Comparative Performance: Closed

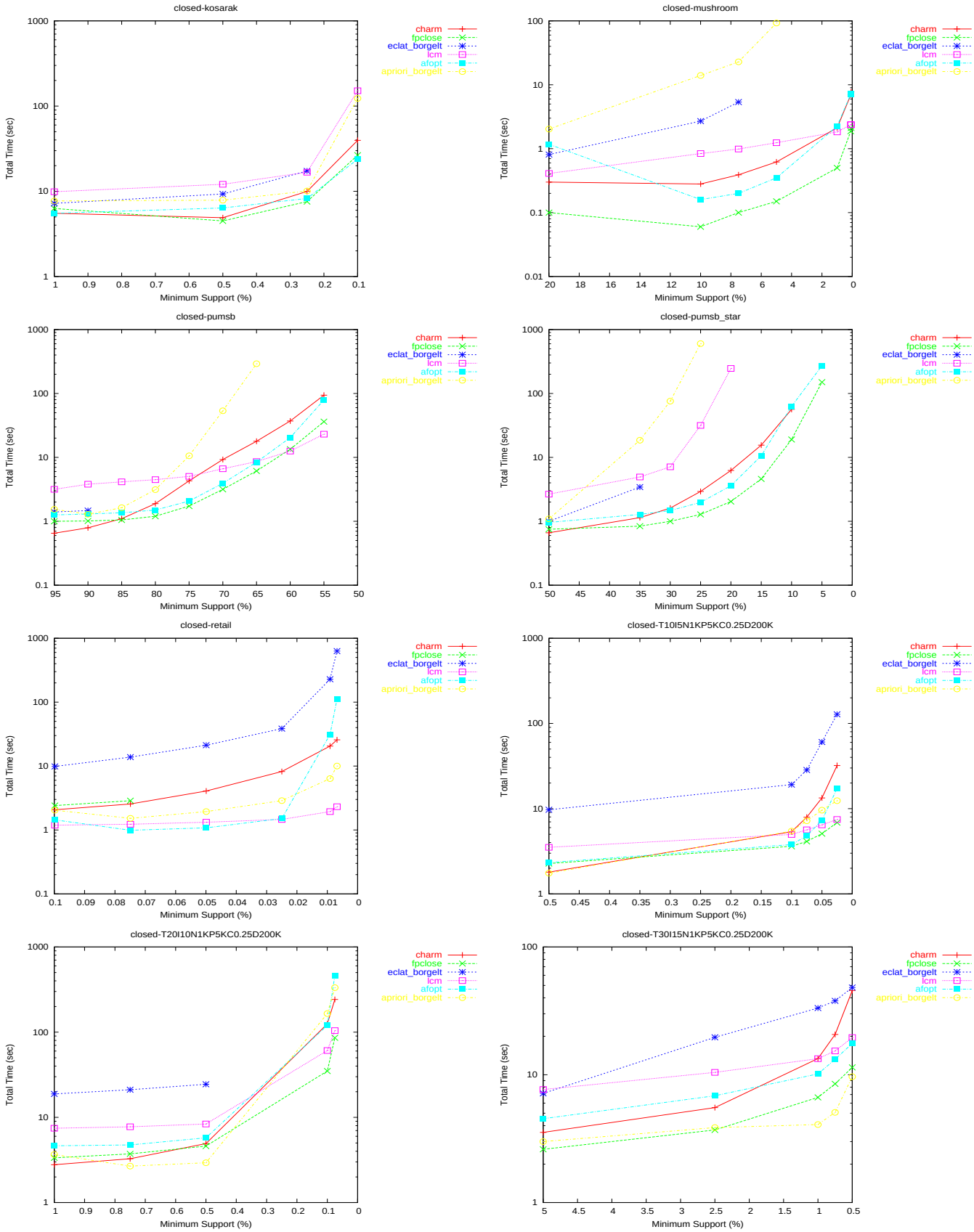


Figure 8. Comparative Performance: Closed

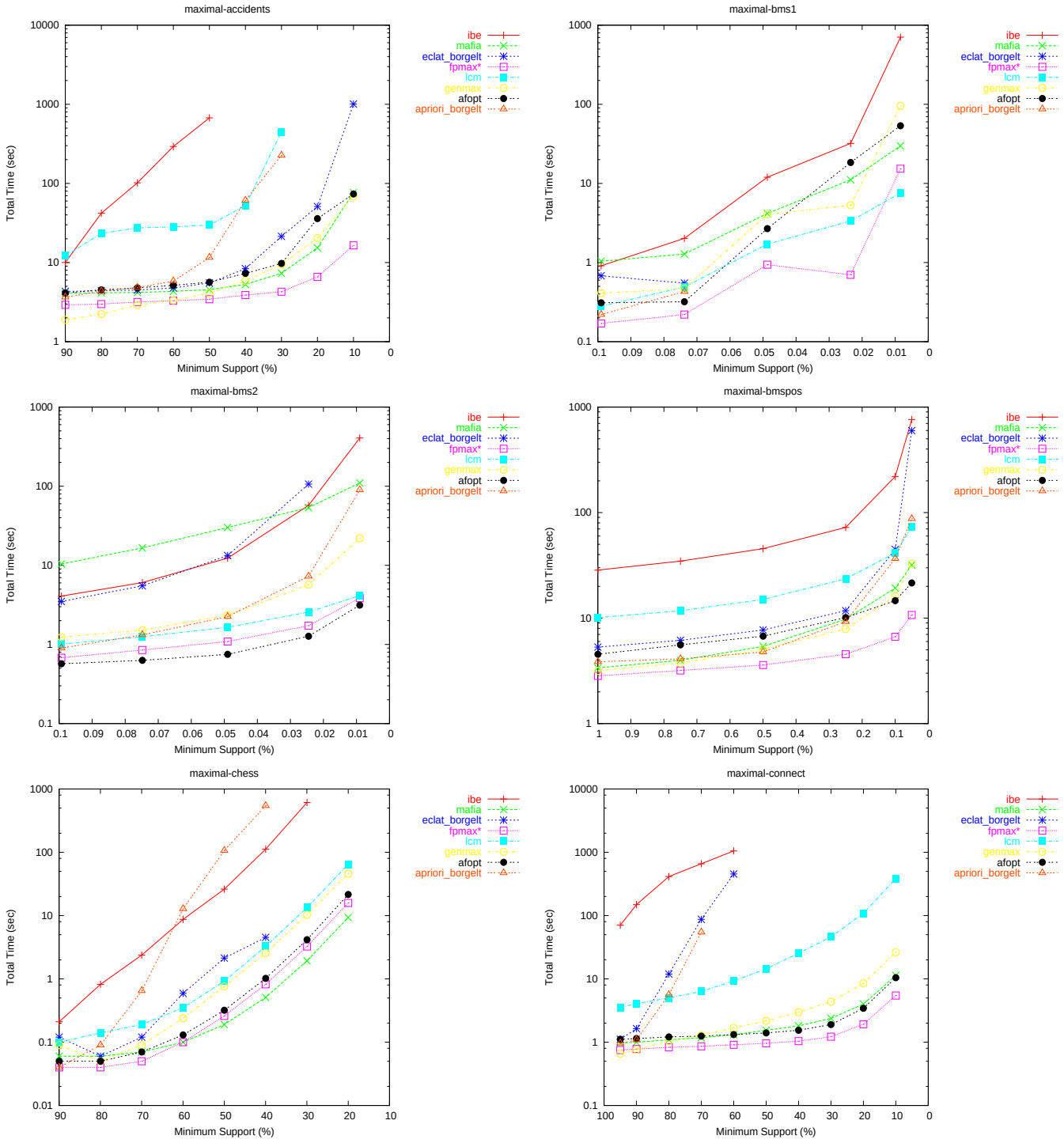


Figure 9. Comparative Performance: Maximal

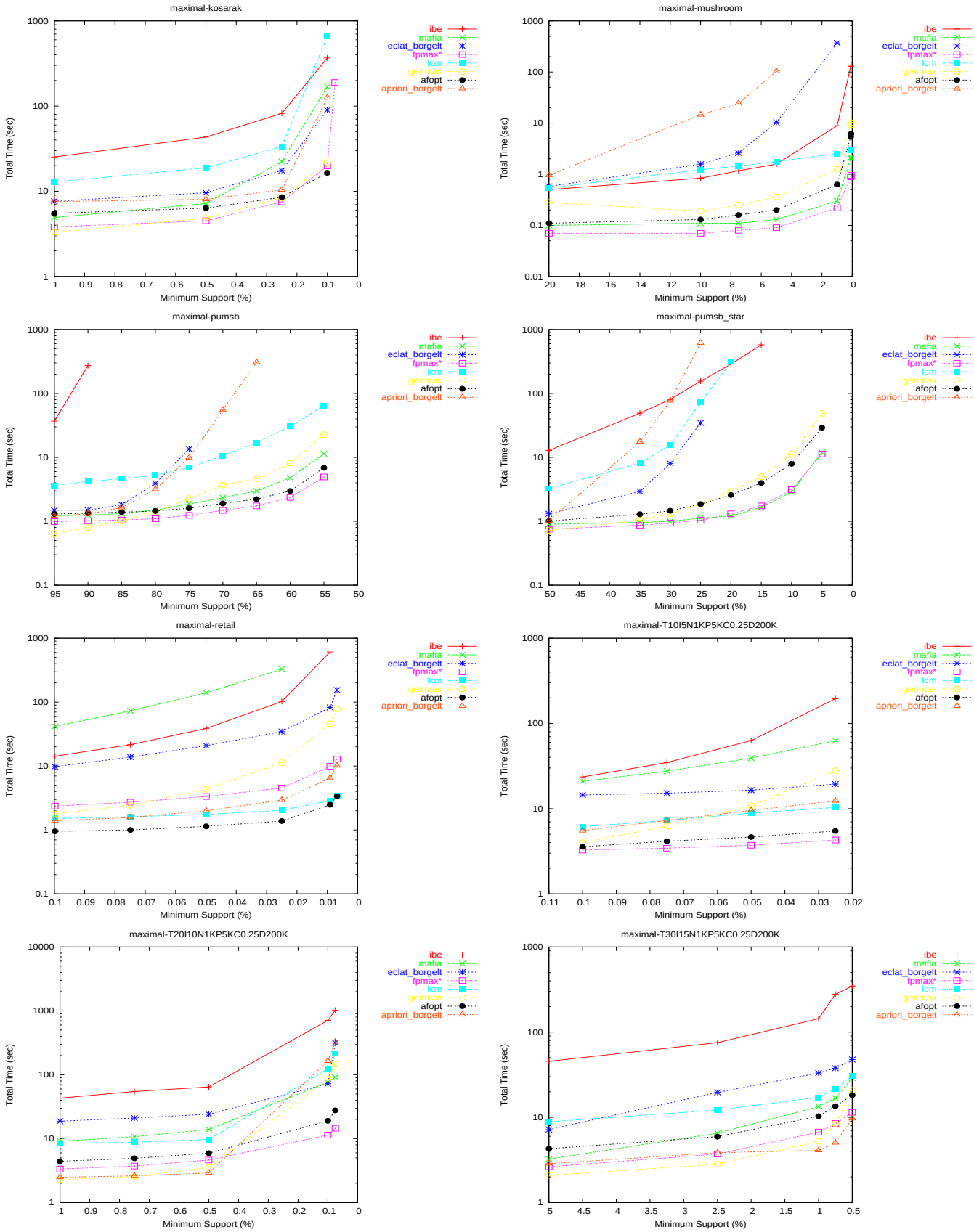


Figure 10. Comparative Performance: Maximal

A fast APRIORI implementation

Ferenc Bodon*
Informatics Laboratory, Computer and Automation Research Institute,
Hungarian Academy of Sciences
H-1111 Budapest, Lágymányosi u. 11, Hungary

Abstract

The efficiency of frequent itemset mining algorithms is determined mainly by three factors: the way candidates are generated, the data structure that is used and the implementation details. Most papers focus on the first factor, some describe the underlying data structures, but implementation details are almost always neglected. In this paper we show that the effect of implementation can be more important than the selection of the algorithm. Ideas that seem to be quite promising, may turn out to be ineffective if we descend to the implementation level.

We theoretically and experimentally analyze APRIORI which is the most established algorithm for frequent itemset mining. Several implementations of the algorithm have been put forward in the last decade. Although they are implementations of the very same algorithm, they display large differences in running time and memory need. In this paper we describe an implementation of APRIORI that outperforms all implementations known to us. We analyze, theoretically and experimentally, the principal data structure of our solution. This data structure is the main factor in the efficiency of our implementation. Moreover, we present a simple modification of APRIORI that appears to be faster than the original algorithm.

1 Introduction

Finding frequent itemsets is one of the most investigated fields of data mining. The problem was first presented in [1]. The subsequent paper [3] is considered as one of the most important contributions to the subject. Its main algorithm, APRIORI, not only influenced the association rule mining community, but it affected other data mining fields as well.

Association rule and frequent itemset mining became a widely researched area, and hence faster and faster algorithms have been presented. Numerous of them are APRIORI based algorithms or APRIORI modifications. Those who adapted APRIORI as a basic search strategy, tended to adapt the whole set of procedures and data structures as well [20][8][21][26]. Since the scheme of this important algorithm was not only used in basic association rules mining, but also in other data mining fields (hierarchical association rules [22][16][11], association rules maintenance [9][10][24][5], sequential pattern mining [4][23], episode mining [18] and functional dependency discovery [14] [15]), it seems appropriate to critically examine the algorithm and clarify its implementation details.

A central data structure of the algorithm is trie or hash-tree. Concerning speed, memory need and sensitivity of parameters, tries were proven to outperform hash-trees [7]. In this paper we will show a version of trie that gives the best result in frequent itemset mining. In addition to description, theoretical and experimental analysis, we provide implementation details as well.

The rest of the paper is organized as follows. In Section 1 the problem is presented, in Section 2 tries are described. Section 3 shows how the original trie can be modified to obtain a much faster algorithm. Implementation is detailed in Section 4. Experimental details are given in Section 5. In Section 7 we mention further improvement possibilities.

2. Problem Statement

Frequent itemset mining came from efforts to discover useful patterns in customers' transaction databases. A customers' transaction database is a sequence of transactions ($\mathcal{T} = \langle t_1, \ldots, t_n \rangle$), where each transaction is an itemset ($t_i \subseteq \mathcal{I}$). An itemset with k elements is called a k-itemset. In the rest of the paper we make the (realistic) assumption that the items are from an ordered set, and transactions are stored as sorted itemsets. The support of an itemset X in $\mathcal{T}$, denoted as $supp_{\mathcal{T}}(X)$, is the number of those transactions that contain X, i.e. $supp_{\mathcal{T}}(X) = |\{t_j : X \subseteq t_j\}|$. An itemset is *frequent* if its support is greater than a support threshold, originally denoted by min_supp. The frequent

*Research supported in part by OTKA grants T42706, T42481 and the EU-COE Grant of MTA SZTAKI.

itemset mining problem is to find all frequent itemset in a given transaction database.

The first, and maybe the most important solution for finding frequent itemsets, is the APRIORI algorithm [3]. Later faster and more sophisticated algorithms have been suggested, most of them being modifications of APRIORI [20][8][21][26]. Therefore if we improve the APRIORI algorithm then we improve a whole family of algorithms. We assume that the reader is familiar with APRIORI [2] and we turn our attention to its central data structure.

3. Determining Support with a Trie

The data structure *trie* was originally introduced to store and efficiently retrieve words of a dictionary (see for example [17]). A trie is a rooted, (downward) directed tree like a hash-tree. The root is defined to be at depth 0, and a node at depth d can point to nodes at depth $d+1$. A pointer is also called *edge* or *link*, which is labeled by a letter. There exists a special letter * which represents an "end" character. If node u points to node v, then we call u the parent of v, and v is a child node of u.

Every leaf ℓ represents a word which is the concatenation of the letters in the path from the root to ℓ. Note that if the first k letters are the same in two words, then the first k steps on their paths are the same as well.

Tries are suitable to store and retrieve not only words, but any finite ordered sets. In this setting a link is labeled by an element of the set, and the trie contains a set if there exists a path where the links are labeled by the elements of the set, in increasing order.

In our data mining context the alphabet is the (ordered) set of all items $\mathcal{I}$. A candidate k-itemset

$$C = \{i_1 < i_2 < \ldots < i_k\}$$

can be viewed as the word $i_1 i_2 \ldots i_k$ composed of letters from $\mathcal{I}$. We do not need the * symbol, because every inner node represent an important itemset (i.e. a meaningful word).

Figure 1 presents a trie that stores the candidates {A,C,D}, {A,E,G}, {A,E,L}, {A,E,M}, {K,M,N}. Numbers in the nodes serve as identifiers and will be used in the implementation of the trie. Building a trie is straightforward, we omit the details, which can be found in [17].

In support count method we take the transactions one-by-one. For a transaction t we take all ordered k-subsets X of t and search for them in the trie structure. If X is found (as a candidate), then we increase the support count of this candidate by one. Here, we do not generate all k-subsets of t, rather we perform early quits if possible. More precisely, if we are at a node at depth d by following the j^{th} item link, then we move forward on those links that have the labels $i \in t$ with index greater than j, but less than $|t| - k + d + 1$.

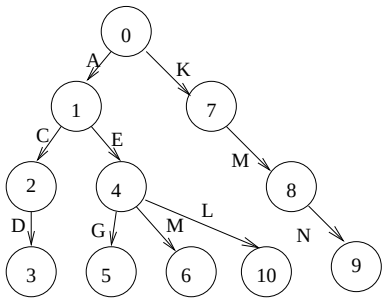


Figure 1. A trie containing 5 candidates

In our approach, tries store not only candidates, but frequent itemsets as well. This has the following advantages:

1. Candidate generation becomes easy and fast. We can generate candidates from pairs of nodes that have the same parents (which means, that except for the last item, the two sets are the same).

2. Association rules are produced much faster, since retrieving a support of an itemset is quicker (remember the trie was originally developed to quickly decide if a word is included in a dictionary).

3. Just one data structure has to be implemented, hence the code is simpler and easier to maintain.

4. We can immediately generate the so called *negative border*, which plays an important role in many APRIORI-based algorithm (online association rules [25], sampling based algorithms [26], etc.).

3.1 Support Count Methods with Trie

Support count is done, by reading transactions one-by-one and determine which candidates are contained in the actual transaction (denoted by t). Finding candidates in a given transaction determines the overall running time primarily. There are two simple recursive methods to solve this task, both starts from the root of the trie. The recursive step is the following (let us denote the number of edges of the actual node by m).

1. For each item in the transaction we determine whether there exists an edge whose label corresponds to the item. Since the edges are ordered according to the labels this means a search in an ordered set.

2. We maintain two indices, one for the items in the transaction, one for the edges of the node. Each index is initialized to the first element. Next we check whether the element pointed by the two indices equals. If they do,

we call our procedure recursively. Otherwise we increase the index that points to the smaller item. These steps are repeated until the end of the transaction or the last edge is reached.

In both methods if item i of the transaction leads to a new node, then item j is considered in the next step only if $j > i$ (more precisely item j is considered, if $j < |t| + \text{actual\_depth} - m + 1$).

Let us compare the running time of the methods. Since both methods are correct, the same branches of the trie will be visited. Running time difference is determined by the recursive step. The first method calls the subroutine that decides whether there exist an edge with a given label $|t|$ times. If binary search is evoked then it requires $\log_2 m$ steps. Also note that subroutine calling needs as many value assignments as many parameters the subroutine has. We can easily improve the first approach. If the number of edges is small (i.e. if $|t|^m < m^{|t|}$) we can do the inverse procedure, i.e. for all labels we check whether there exists a corresponding item. This way the overall running time is proportional to $\min\{|t| \log_2 m, m \log_2 |t|\}$.

The second method needs at least $min\{m, |t|\}$ and at most $m + |t|$ steps and there is no subroutine call.

Theoretically it can happen that the first method is the better solution (for example if $|t|$=1, m is large, and the label of the last edge corresponds to the only item in the transaction), however in general the second method is faster. Experiments showed that the second method finished 50% faster on the average.

Running time of support count can be further reduced if we modify the trie a little. These small modifications, tricks are described in the next subsections.

3.2 Storing the Length of Maximal Paths

Here we show how the time of finding supported candidates in a transaction can be significantly reduced by storing a little extra information. The point is that we often perform superfluous moves in trie search in the sense that there are no candidates in the direction we are about to explore. To illustrate this, consider the following example. Assume that after determining frequent 4-itemsets only candidate $\{A, B, C, D, E\}$ was generated, and Figure 2 shows the resulting trie.

If we search for 5-itemset candidates supported by the transaction $\{A, B, C, D, E, F, G, H, I\}$, then we must visit every node of the trie. This appears to be unnecessary since only one path leads to a node at depth 5, which means that only one path represents a 5-itemset candidate. Instead of visiting merely 6 nodes, we visit 32 of them. At each node, we also have to decide which link to follow, which can greatly affect running time if a node has many links.

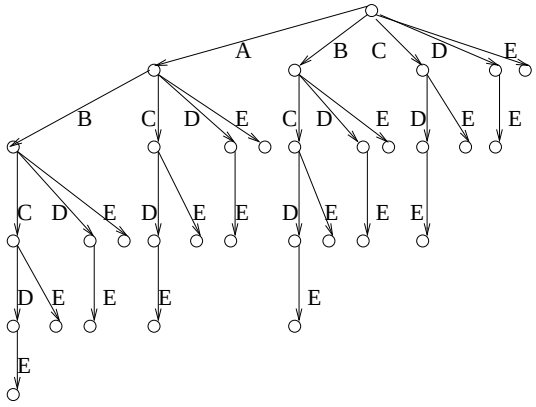


Figure 2. A trie with a single 5-itemset candidate

To avoid this superfluous traveling, at every node we store the length of the longest directed path that starts from there. When searching for k-itemset candidates at depth d, we move downward only if the maximal path length at this node is at least $k - d$. Storing counters needs memory, but as experiments proved, it can seriously reduce search time for large itemsets.

3.3 Frequency Codes

It often happens that we have to find the node that represents a given itemset. For example, during candidate generation, when the subsets of the generated candidate have to be checked, or if we want to obtain the association rules. Starting from the root we have to travel, and at depth d we have to find the edge whose label is the same as the d^{th} element of the itemset.

Theoretically, binary search would be the fastest way to find an item in an ordered list. But if we go down to the implementation level, we can easily see that if the list is small the linear search is faster than the binary (because an iteration step is faster). Hence the fastest solution is to apply linear search under a threshold and binary above it. The threshold does not depend on the characteristic of the data, but on the ratio of the elementary operations (value assignment, increase, division, ...)

In linear search, we read the first item, compare with the searched item. If it is smaller, then there is no edge with this item, if greater, we step forward, if they equal then the search is finished. If we have bad luck, the most frequent item has the highest order, and we have to march to the end of the line whenever this item is searched for.

On the whole, the search will be faster if the order of items corresponds to the frequency order of them. We know exactly the frequency order after the first read of the whole

database, thus everything is at hand to build the trie with the frequency codes instead of the original codes. The *frequency code* of an item i is $fc[i]$, if i is the $fc[i]^{th}$ most frequent item. Storing frequency codes and their inverses increases the memory need slightly, in return it increases the speed of retrieving the occurrence of the itemsets. A theoretical analysis of the improvement can be read in the Appendix.

Frequency codes also affect the structure of the trie, and consequently the running time of support count. To illustrate this, let us suppose that 2 candidates of size 3 are generated: $\{A, B, C\}, \{A, B, D\}$. Different tries are generated if the items have different code. The next figure present the tries generated by two different coding.

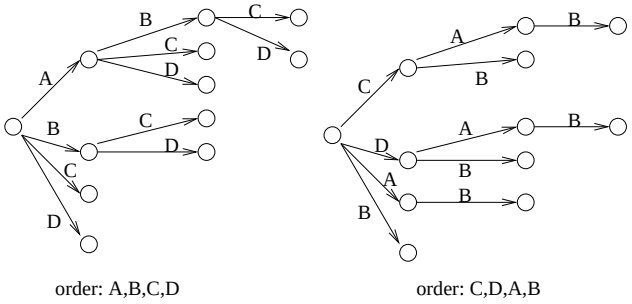


Figure 3. Different coding results in different tries

If we want to find which candidates are stored in a basket $\{A, B, C, D\}$ then 5 nodes are visited in the first case and 7 in the second case. That does not mean that we will find the candidates faster in the first case, because nodes are not so "fat" in the second case, which means, that they have fewer edges. Processing a node is faster in the second case, but more nodes have to be visited.

In general, if the codes of the item correspond to the frequency code, then the resulted trie will be unbalanced while in the other case the trie will be rather balanced. Neither is clearly more advantageous than the other. We choose to build unbalanced trie, because it travels through fewer nodes, which means fewer recursive steps which is a slow operation (subroutine call with at least five parameters in our implementation) compared to finding proper edges at a node.

In [12] it was showed that it is advantageous to recode frequent items according to ascending order of their frequencies (i.e.: the inverse of the frequency codes) because candidate generation will be faster. The first step of candidate generation is to find siblings and take the union of the itemset represented by them. It is easy to prove that there are less sibling relations in a balanced trie, therefore less unions are generated and the second step of the candidate generation is evoked fewer times. For example in our figure one union would be generated and then deleted in the first case and none would be generated in the second.

Altogether frequency codes have advantages and disadvantages. They accelerate retrieving the support of an itemset which can be useful in association rule generation or in on-line frequent itemset mining, but slows down candidate generation. Since candidate generation is by many order of magnitude faster that support count, the speed decrease is not noticeable.

3.4 Determining the support of 1- and 2-itemset candidates

We already mentioned that the support count of 1-element candidates can easily be done by a single vector. The same easy solution can be applied to 2-itemset candidates. Here we can use an array [19]. The next figure illustrates the solutions.

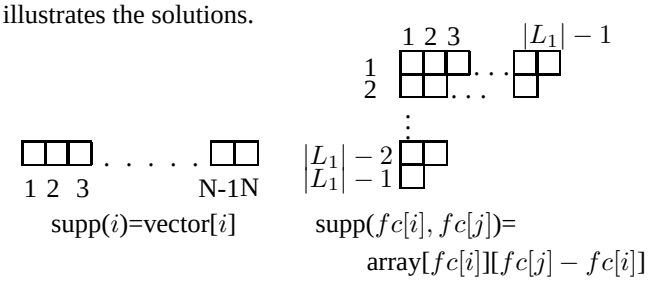


Figure 4. Data structure to determine the support of the items and candidate itempairs

Note that this solution is faster than the trie-based solution, since increasing a support takes one step. Its second advantage is that it needs less memory.

Memory need can be further reduced by applying on-the-fly candidate generation [12]. If the number of frequent items is $|L_1|$ then the number of 2-itemset candidates is $\binom{|L_1|}{2}$, out of which a lot will never occur. Thus instead of the array, we can use trie. A 2-itemset candidate is inserted only if it occurs in a basket.

3.5 Applying hashing techniques

Determining the support with a trie is relatively slow when we have to step forward from a node that has many edges. We can accelerate the search if hash-tables are employed. This way the cost of a step down is calculating one hash value, thus a recursive step takes exactly $|t|$ steps. We want to keep the property that a leaf represents exactly one itemset, hence we have to use perfect hashing. Frequency codes suit our needs again, because a trie stores only frequent items. Please note, that applying hashing techniques at tries does not result in a hash-tree proposed in [3].

It is wasteful to change all inner nodes to hash-table, since a hash-table needs much more memory than an ordered list of edges. We propose to alter only those inner nodes into a hash-table, which have more edges than a reasonable threshold (denoted by $leaf\_max\_size$). During trie construction when a new leaf is added, we have to check whether the number of its parent's edges exceeds $leaf\_max\_size$. If it does, the node has to be altered to hash-table. The inverse of this transaction may be needed when infrequent itemsets are deleted.

If the frequent itemsets are stored in the trie, then the number of edges can not grow as we go down the trie. In practice nodes, at higher level have many edges, and nodes at low level have only a few. The structure of the trie will be the following: nodes over a (not necessarily a horizontal) line will be hash-tables, while the others will be original nodes. Consequently, where it was slow, search will be faster, and where it was fast –because of the small number of edges– it will remain fast.

3.6 Brave Candidate Generation

It is typical, that in the last phases of APRIORI there are a small number of candidates. However, to determine their support the whole database is scanned, which is wasteful. This problem was also mentioned in the original paper of APRIORI [3], where algorithm APRIORI-HYBRID was proposed. If a certain heuristic holds, then APRIORI switches to APRIORI-TID, where for each candidate we store the transactions that contain it (and support is immediately obtained). This results in a much faster execution of the latter phases of APRIORI.

The hard point of this approach is to tell when to switch from APRIORI to APRIORI-TID. If the heuristics fails the algorithm may need too much memory and becomes very slow. Here we choose another approach that we call the brave candidate generation and the algorithm is denoted by APRIORI-BRAVE.

APRIORI-BRAVE keeps track of memory need, and stores the amount of the maximal memory need. After determining frequent k-itemsets it generates $(k+1)$-itemset candidates as APRIORI does. However, it does not carry on with the support count, but checks if the memory need exceeds the maximal memory need. If not, $(k+2)$-itemset candidates are generated, otherwise support count is evoked and maximal memory need counter is updated. We carry on with memory check and candidate generation till memory need does not reach maximal memory need.

This procedure will collect together the candidates of the latter phases and determine their support in one database read. For example, candidates in database T40I10D100K with $min\_supp = 0.01$ will be processed the following way: 1, 2, 3, 4-10, 11-14, which means 5 database scan instead of 14.

One may say that APRIORI-BRAVE does not consume more memory than APRIORI and it can be faster, because there is a possibility that some candidates at different sizes are collected and their support is determined in one read. However, accelerating in speed is not necessarily true. APRIORI-BRAVE may generate $(k+2)$-itemset candidates from frequent k-itemset, which can lead to more false candidates. Determining support of false candidates needs time. Consequently, we cannot guarantee that APRIORI-BRAVE is faster than APRIORI, however, test results prove that this heuristics works well in real life.

3.7 A Deadend Idea- Support Lower Bound

The false candidates problem in APRIORI-BRAVE can be avoided if only those candidates are generated that are surely frequent. Fortunately, we can give a lower bound to the support of a $(k+j)$-itemset candidate based on the support of k-itemsets ($j \geq 0$) [6]. Let $X = X' \cup Y \cup Z$. The following inequality holds:

$$supp(X) \geq supp(X' \cup Y) + supp(X' \cup Z) - supp(X').$$

And hence

$$supp(X) \geq \max_{Y,Z \in X} \{supp(X \setminus Y) + supp(X \setminus Z) - supp(X \setminus Y \setminus Z)\}.$$

If we want to give a lower bound to a support of $(k+j)$-itemset base on support of k-itemset, we can use the generalization of the above inequality ($X = X' \cup x_1 \cup \ldots \cup x_j$):

$$supp(X) \geq supp(X' \cup x_1) + \ldots + supp(X' \cup x_j) - (j-1)supp(X').$$

To avoid false candidate generation we generate only those candidates that are surely frequent. This way, we could say that neither memory need nor running time is worse than APRIORI's. Unfortunately, this is not true!

Test results proved that this method not only slower than original APRIORI-BRAVE, but also APRIORI outperforms it. The reason is simple. Determining the support threshold is a slow operation (we have to find the support of $\binom{k+j}{k-1}j$ itemsets) and has to be executed many times. It loses more time with determining support thresholds than we win by generating sooner some candidates.

The failure of the support-threshold candidate generation is a nice example when a promising idea turns out to be useless at the implementation level.

3.8 Storing input

Many papers in the frequent itemset mining subject focus on the number of the whole database scan. They say

that reading data from disc is much slower than operating in memory, thus the speed is mainly determined by this factor. However, in most cases the database is not so big and it fits into the memory. Behind the scenery the operating system swaps it in the memory and the algorithms read the disc only once. For example, a database that stores 10^7 transaction, and in each transaction there are 6 items on the average needs approximately 120Mbytes, which is a fraction of today's average memory capacity. Consequently, if we explicitly store the simple input data, the algorithm will not speed up, but will consume more memory (because of the double storing), which may result in swapping and slowing down. Again, if we descend to the elementary operation of an algorithm, we may conclude the opposite result.

Storing the input data is profitable if the same transactions are gathered up. If a transaction occurs ℓ times, the support count method is evoked once with counter increment ℓ, instead of calling the procedure ℓ times with counter increment 1. In Borgelt algorithm input is stored in a prefix tree. This is dangerous, because data file can be too large.

We've chosen to store only *reduced transactions*. A reduced transaction stores only the frequent items of the transaction. Storing reduced transactions have all information needed to discover frequent itemsets of larger sizes, but it is expected to need less memory (obviously it depends on min_supp). Reduced transactions are stored in a tree for the fast insertion (if reduced transactions are recode with frequency codes then we almost get an FP-tree [13]). Optionally, when the support of candidates of size k is determined we can delete those reduced transactions that do not contain candidates of size k.

4. Implementation details

APRIORI is implemented in an object-oriented manner in language C++. STL possibilities (`vector`, `set`, `map`) are heavily used. The algorithm (class `Apriori`) and the data structure (class `Trie`) are separated. We can change the data structure (for example to a hash-tree) without modifying the source code of the algorithm.

The baskets in a file are first stored in a `vector<...>`. If we choose to store input –which is the default– the reduced baskets are stored in a `map<vector<...>,unsigned long>`, where the second parameter is the number of times that reduced basket occurred. A better solution would be to apply trie, because `map` does not make use of the fact that two baskets can have the same prefixes. Hence insertion of a basket would be faster, and the memory need would be smaller, since the same prefixes would be stored just once. Because of the lack of time trie-based basket storing was not implemented and we do not delete a reduced basket from the `map` if it did not contain any candidate during some scan.

The class `Trie` can be programmed in many ways. We've chosen to implement it with vectors and arrays. It is simple, fast and minimal with respect to memory need. Each node is described by the same element of the vectors (or row of the arrays). The root belongs to the 0^{th} element of each vector. The following figure shows the way the trie is represented by vectors and arrays.

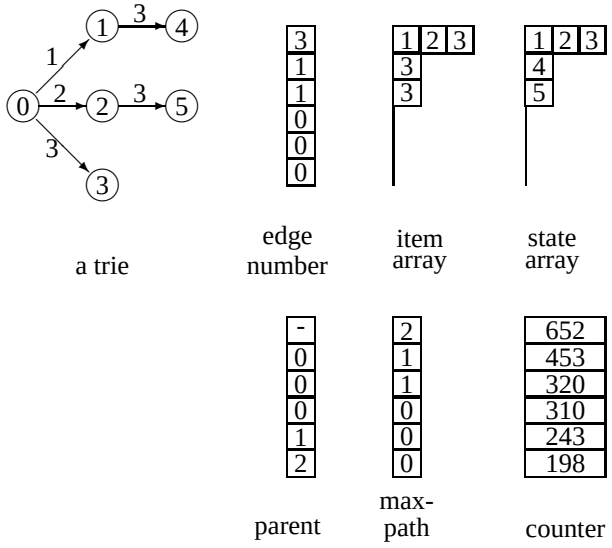


Figure 5. Implementation of a trie

The vector `edge_number` stores the number of edges of the nodes. The `itemarray[i]` stores the label of the edges, `statearray[i]` stores the end node of the edges of node i. Vectors `parent[i]` and `maxpath[i]` store the parents and the length of the longest path respectively. The occurrences of itemset represented by the nodes can be found in vector `counter`.

For vectors we use `vector` class offered by the STL, but arrays are stored in a traditional C way. Obviously, it is not a fixed-size array (which caused the ugly `calloc`, `realloc` commands in the code). Each row is as long as many edges the node has, and new rows are inserted as the trie grows (during candidate generation). A more elegant way would be if the arrays were implemented as a `vector` of `vectors`. The code would be easier to understand and shorter, because the algorithms of STL could also be used. However, the algorithm would be slower because determining a value of the array takes more time. Tests showed that sacrificing a bit from readability leads to 10-15% speed up.

In our implementation we do not adapt on-line 2-itemset candidate generation (see Section 3.4) but use a vector and an array (`temp_counter_array`) for determining the support of 1- and 2-itemset candidates efficiently.

The vector and array description of a trie makes it pos-

sible to give a fast implementation of the basic functions (like candidate generation, support count, ...). For example, deleting infrequent nodes and pulling the vectors together is achieved by a single scan of the vectors. For more details readers are referred to the html-based documentation.

5. Experimental results

Here we present the experimental results of our implementation of APRIORI and APRIORI-BRAVE compared to the two famous APRIORI implementations by Christian Borgelt (version 4.08)[1] and Bart Goethals (release date: 01/06/2003)[2]. 3 databases were used: the well-known T40I10D100K and T10I4D100K and a coded log of a click-stream data of a Hungarian on-line news portal (denoted by *kosarak*). This database contains 990002 transactions of size 8.1 on the average.

Test were run on a PC with 2.8 GHz dual Xeon processors and 3Gbyte RAM. The operating system was Debian Linux, running times were obtained by the `/usr/bin/time` command. The following 3 tables present the test results of the 3 different implementations of APRIORI and the APRIORI_BRAVE on the 3 databases. Each test was carried out 3 times; the tables contain the averages of the results. The two well-known implementations are denoted by the last name of the coders.

min_ supp	Bodon impl.	Borgelt impl.	Goethals impl.	APRIORI_ BRAVE
0.05	8.57	10.53	25.1	8.3
0.030	10.73	11.5	41.07	10.6
0.020	15.3	13.9	53.63	14.0
0.010	95.17	155.83	207.67	100.27
0.009	254.33	408.33	458.7	178.93
0.0085	309.6	573.4	521.0	188.0

Running time (sec.)

Table 1. *T40I10D100K* **database**

Tables 4-6 show result of Bodon's APRIORI implementation with hash techniques. The notation *leaf_max_size* stands for the threshold above a node applies perfect hashing technique.

Our APRIORI implementation beats Goethals implementation almost all the times, and beats Borgelt's implementation many times. It performs best at low support threshold. We can also see that in the case of these 3 databases APRIORI_BRAVE outperforms APRIORI at most support threshold.

Strange, but hashing technique not always resulted in faster execution. The reason for this might be that small vectors are cached in, where linear search is very fast. If we enlarge the size of the vector by altering it into a hashtable, then the vector may be moved into the memory, where read is a slower operation. Applying hashing technique is the other example when an accelerating technique does not result in improvement.

[1] http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html#assoc

[2] http://www.cs.helsinki.fi/u/goethals/software/index.html

min_ supp	Bodon impl.	Borgelt impl.	Goethals impl.	APRIORI_ BRAVE
0.0500	4.23	5.2	11.73	2.87
0.0100	10.27	14.03	30.5	6.6
0.0050	17.87	16.13	40.77	12.7
0.0030	34.07	18.8	53.43	23.97
0.0020	70.3	21.9	69.73	46.5
0.0015	85.23	25.1	86.37	87.63

Running time (sec.)

Table 2. *T10I4D100K* **database**

min_ supp	Bodon impl.	Borgelt impl.	Goethals impl.	APRIORI_ BRAVE
0.050	14.43	32.8	28.27	14.1
0.010	17.9	41.87	44.3	17.5
0.005	24.0	52.4	58.4	21.67
0.003	35.9	67.03	76.77	28.5
0.002	81.83	199.07	182.53	72.13
0.001	612.67	1488.0	1101.0	563.33

Running time (sec.)

Table 3. *kosarak* **database**

min_ supp	leaf_max_size 1	2	5	7	10	25	60	100
0.0500	8.1	8.1	8.1	8.1	8.1	8.1	8.1	8.4
0.0300	9.6	9.8	9.7	9.8	9.8	9.8	9.8	9.9
0.0200	13.8	14.4	13.6	13.9	13.6	13.9	13.9	14.1
0.0100	114.0	96.3	83.8	82.5	78.9	79.2	80.4	83.0
0.0090	469.8	339.6	271.8	258.1	253.0	253.0	251.0	253.8
0.0085	539.0	373.0	340.0	310.0	306.0	306.0	306.0	309.0

Runing time (sec.)

Table 4. T40I10D100K database

min_ supp	leaf_max_size 1	2	5	7	10	25	60	100
0.0500	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8
0.0100	7.8	7.3	6.9	6.9	6.9	6.9	7.0	7.1
0.0050	24.2	14.9	13.4	13.1	13	12.8	13.0	13.2
0.0030	55.3	34.6	30.3	25.9	25.2	25.2	25.3	25.8
0.0020	137.3	100.2	76.2	77.0	75.2	78.0	69.5	64.5
0.0015	235.0	176.0	125.0	130.0	125.0	132.0	115.0	103.0

Running time (sec.)

Table 5. T10I4D100K database

min_ supp	leaf_max_size 1	2	5	7	10	25	60	100
0.0500	14.6	14.3	14.3	14.2	14.2	14.2	14.2	14.2
0.0100	17.5	17.5	17.5	17.6	17.6	17.6	18	18.1
0.0050	21.0	21.0	22.0	21.0	22.0	22.0	22.8	22.8
0.0030	26.3	26.1	26.3	26.5	27.2	27.4	28.5	29.6
0.0020	98.8	77.5	62.3	60.0	59.7	61.0	61.0	63.4
0.0010	1630	1023.0	640.0	597.0	574.0	577.0	572.0	573.0

Running time (sec.)

Table 6. kosarak database

6. Further Improvement and Research Possibilities

Our APRIORI implementation can be further improved if trie is used to store reduced basket, and a reduced basket is removed if it does not contain any candidate.

We mentioned that there are two basic ways of finding the contained candidates in a given transaction. Further theoretical and experimental analysis may lead to the conclusion that a mixture of the two approaches would lead to the fastest execution.

Theoretically, hashing technique accelerates support count. However, experiments did not support this claim. Further investigations are needed to clear the possibilities of this technique.

7. Conclusion

Determining frequent objects (itemsets, episodes, sequential patterns) is one of the most important fields of data mining. It is well known that the way candidates are defined has great effect on running time and memory need, and this is the reason for the large number of algorithms. It is also clear that the applied data structure also influences efficiency parameters. However, the same algorithm that uses a certain data structure has a wide variety of implementation. In this paper, we showed that different implementation results in different running time, and the differences can exceed differences between algorithms. We presented an implementation that solved frequent itemset mining problem in most cases faster than other well-known implementations.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. *In Proc. of the ACM SIGMOD Conference on Management of Data*, pages 207–216, 1993.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328, 1996.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *The International Conference on Very Large Databases*, pages 487–499, 1994.

[4] R. Agrawal and R. Srikant. Mining sequential patterns. In P. S. Yu and A. L. P. Chen, editors, *Proc. 11th Int. Conf. Data Engineering, ICDE*, pages 3–14. IEEE Press, 6–10 1995.

[5] N. F. Ayan, A. U. Tansel, and M. E. Arkun. An efficient algorithm to update large itemsets with early pruning. In *Knowledge Discovery and Data Mining*, pages 287–291, 1999.

[6] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, pages 85–93. ACM Press, 1998.

[7] F. Bodon and L. Rónyai. Trie: an alternative data structure for data mining algorithms. *to appear in Computers and Mathematics with Applications*, 2003.

[8] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. *SIGMOD Record (ACM Special Interest Group on Management of Data),26(2):255*, 1997.

[9] D. W.-L. Cheung, J. Han, V. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *ICDE*, pages 106–114, 1996.

[10] D. W.-L. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Database Systems for Advanced Applications*, pages 185–194, 1997.

[11] Y. Fu. Discovery of multiple-level rules from large databases, 1996.

[12] B. Goethals. Survey on frequent pattern mining. Technical report, Helsinki Institute for Information Technology, 2003.

[13] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In W. Chen, J. Naughton, and P. A. Bernstein, editors, *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, 05 2000.

[14] Y. Huhtala, J. Kärkkäinen, P. Porkka, and H. Toivonen. TANE: An efficient algorithm for discovering functional

and approximate dependencies. *The Computer Journal*, 42(2):100–111, 1999.

[15] Y. Huhtala, J. Kinen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *ICDE*, pages 392–401, 1998.

[16] Y. F. Jiawei Han. Discovery of multiple-level association rules from large databases. *In Proc. of the 21st International Conference on Very Large Databases (VLDB), Zurich, Switzerland*, 1995.

[17] D. E. Knuth. *The Art of Computer Programming Vol. 3.* Addison-Wesley, 1968.

[18] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the First International Conference on Knowledge Discovery and Data Mining*, pages 210–215. AAAI Press, 1995.

[19] B. Ozden, S. Ramaswamy, and A. Silberschatz. Cyclic association rules. In *ICDE*, pages 412–421, 1998.

[20] J. S. Park, M.-S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 175–186, San Jose, California, 22–25 1995.

[21] A. Sarasere, E. Omiecinsky, and S. Navathe. An efficient algorithm for mining association rules in large databases. *In Proc. 21st International Conference on Very Large Databases (VLDB), Zurich, Switzerland, Also Gatech Technical Report No. GIT-CC-95-04.*, 1995.

[22] R. Srikant and R. Agrawal. Mining generalized association rules. *In Proc. of the 21st International Conference on Very Large Databases (VLDB), Zurich, Switzerland*, 1995.

[23] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. Technical report, IBM Almaden Research Center, San Jose, California, 1995.

[24] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. page 263.

[25] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Knowledge Discovery and Data Mining*, pages 263–266, 1997.

[26] H. Toivonen. Sampling large databases for association rules. In *The VLDB Journal*, pages 134–145, 1996.

8. Appendix

Let us analyze formally how frequency code accelerate the search. Suppose that the number of frequent items is m, and the j^{th} most frequent has to be searched for m_j times ($n_1 \geq n_2 \geq \ldots \geq n_m$) and $n = \sum_{i=1}^{m} n_i$. If an item is in the position j, then the cost of finding it is $c \cdot j$, where c is a constant. For the sake of simplicity c is omitted. The total cost of search based on frequency codes is $\sum_{j=1}^{m} j \cdot n_j$.

How much is the cost if the list is not ordered by frequencies? We cannot determine this precisely, because we don't know which item is in the first position, which item is in the second, etc. We can calculate the expected time of the total cost if we assume that each order occurs with the same probability. Then the probability of each permutation is $\frac{1}{m!}$. Thus

$$E[\text{total cost}] = \sum_{\pi} \frac{1}{m!} \cdot (\text{cost of } \pi)$$
$$= \frac{1}{m!} \sum_{\pi} \sum_{j=1}^{m} \pi(j) n_{\pi(j)}.$$

Here π runs through the permutations of $1, 2, \ldots, m$, and the j^{th} item of π is denoted by $\pi(j)$. Since each item gets to each position $(m-1)!$ times, we obtain that

$$E[\text{total cost}] = \frac{1}{m!}(m-1)! \sum_{j=1}^{m} n_j \sum_{k=1}^{m} k$$
$$= \frac{1}{m} \frac{(m+1)m}{2} \sum_{j=1}^{m} n_j = \frac{m+1}{2} n.$$

It is easy to prove that $E[\text{total cost}]$ is greater than or equal to the total cost of the search based on frequency codes (because of the condition $n_1 \geq n_2 \geq \ldots \geq n_m$). We want to know more, namely how small the ratio

$$\frac{\sum_{j=1}^{m} j \cdot n_j}{n \frac{m+1}{2}} \tag{1}$$

can be. In the worst case ($n_1 = n_2 = \ldots = n_m$) it is 1, in best case ($n_1 = n - m + 1, n_2 = n_3 = \ldots = n_m = 1$) it converges to 0, if $n \to \infty$.

We can not say anything more unless the probability distribution of frequent items is known. In many applications, there are some very frequent items, and the probability of rare items differs slightly. This is why we voted for an exponential decrease. In our model the probability of occurrence of the j^{th} most frequent item is $p_j = ae^{bj}$, where $a > 0, b < 0$ are two parameters, such that $ae^b \leq 1$ holds. Parameter b can be regarded as the gradient of the distribution, and parameter a determines the starting point[3].

[3]Note that $\sum p_j$ does not have to be equal with 1, since more than one item can occur in a basket.

We suppose, that the ratio of occurrences is the same as the ratio of the probabilities, hence $n_1 : n_2 : \ldots : n_m = p_1 : p_2 : \ldots : p_m$. From this and the condition $n = \sum_{j=1}^{m} n_j$, we infer that $n_j = n\frac{p_j}{\sum_{k=1}^{m} p_k}$. Using the formula for geometric series, and using the notation $x = e^{\,b}$ we obtain

$$n_j = n\frac{x^j(x-1)}{x^{m+1}-1} = n\frac{x-1}{x^{m+1}-1}\cdot x^j.$$

The total cost can be determined:

$$\sum_{j=1}^{m} j \cdot n_j = \frac{n(x-1)}{x^{m+1}-1}\sum_{j=1}^{m} j \cdot x^j.$$

Let us calculate $\sum_{j=1}^{m} j \cdot x^j$:

$$\sum_{j=1}^{m} j \cdot x^j = \sum_{j=1}^{m}(j+1)\cdot x^j - \sum_{j=1}^{m} x^j = \Big(\sum_{j=1}^{m} x^{j+1}\Big)' - \sum_{j=1}^{m} x^j$$
$$= \frac{mx^{m+2} - (m+1)x^{m+1} + x}{(x-1)^2}.$$

The ratio of total costs can be expressed in a closed formula:

$$\text{cost ratio} = \frac{2x(mx^{m+1} - (m+1)x^m + 1)}{(x^{m+1}-1)(x-1)(m+1)}, \qquad (2)$$

where $x = e^b$. We can see, that the speedup is independent of a. In Figure 6 3 different distributions can be seen. The first is gently sloping, the second has larger graduation and the last distribution is quite steep.

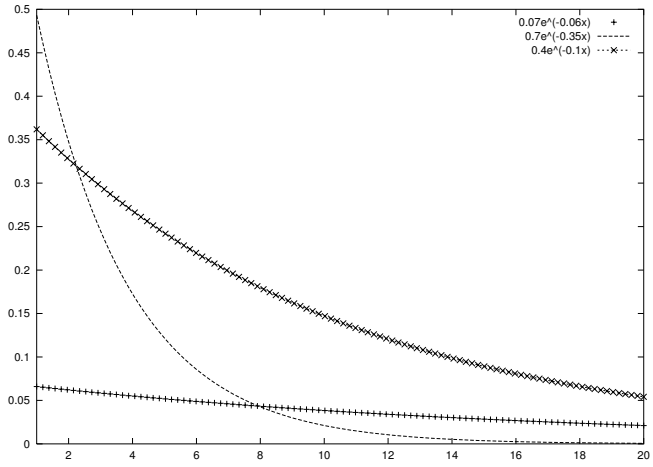


Figure 6. 3 different probability distribution of the items

If we substitute the parameters of the probability distribution to the formula (2) (with $m = 10$), then the result will be 0.39, 0.73 and 0.8, which meets our expectations: by adapting frequency codes the search time will drop sharply if the probabilities differ greatly from each other. We have to remember that frequency codes do not have any effect on nodes where binary search is applied.

Efficient Implementations of Apriori and Eclat

Christian Borgelt

Department of Knowledge Processing and Language Engineering
School of Computer Science, Otto-von-Guericke-University of Magdeburg
Universitätsplatz 2, 39106 Magdeburg, Germany
Email: borgelt@iws.cs.uni-magdeburg.de

Abstract

Apriori and Eclat are the best-known basic algorithms for mining frequent item sets in a set of transactions. In this paper I describe implementations of these two algorithms that use several optimizations to achieve maximum performance, w.r.t. both execution time and memory usage. The Apriori implementation is based on a prefix tree representation of the needed counters and uses a doubly recursive scheme to count the transactions. The Eclat implementation uses (sparse) bit matrices to represent transactions lists and to filter closed and maximal item sets.

1. Introduction

Finding frequent item sets in a set of transactions is a popular method for so-called market basket analysis, which aims at finding regularities in the shopping behavior of customers of supermarkets, mail-order companies, on-line shops etc. In particular, it is tried to identify sets of products that are frequently bought together.

The main problem of finding frequent item sets, i.e., item sets that are contained in a user-specified minimum number of transactions, is that there are so many possible sets, which renders naïve approaches infeasible due to their unacceptable execution time. Among the more sophisticated approaches two algorithms known under the names of Apriori [1, 2] and Eclat [8] are most popular. Both rely on a top-down search in the subset lattice of the items. An example of such a subset lattice for five items is shown in figure 1 (empty set omitted). The edges in this diagram indicate subset relations between the different item sets.

To structure the search, both algorithms organize the subset lattice as a prefix tree, which for five items is shown in Figure 2. In this tree those item sets are combined in a node which have the same prefix w.r.t. to some arbitrary, but fixed order of the items (in the five items example, this order is simply a, b, c, d, e). With this structure, the item sets contained in a node of the tree can be constructed easily in the following way: Take all the items with which the edges leading to the node are labeled (this is the common prefix) and add an item that succeeds, in the fixed order of the items, the last edge label on the path. Note that in this way we need only one item to distinguish between the item sets represented in one node, which is relevant for the implementation of both algorithms.

The main differences between Apriori and Eclat are how they traverse this prefix tree and how they determine the *support* of an item set, i.e., the number of transactions the item set is contained in. Apriori traverses the prefix tree in breadth first order, that is, it first checks item sets of size 1, then item sets of size 2 and so on. Apriori determines the support of item sets either by checking for each candidate item set which transactions it is contained in, or by traversing for a transaction all subsets of the currently processed size and incrementing the corresponding item set counters. The latter approach is usually preferable.

Eclat, on the other hand, traverses the prefix tree in depth first order. That is, it extends an item set prefix until it reaches the boundary between frequent and infrequent item sets and then backtracks to work on the next prefix (in lexicographic order w.r.t. the fixed order of the items). Eclat determines the support of an item set by constructing the list of identifiers of transactions that contain the item set. It does so by intersecting two lists of transaction identifiers of two item sets that differ only by one item and together form the item set currently processed.

2. Apriori Implementation

My Apriori implementation uses a data structure that directly represents a prefix tree as it is shown in figure 2. This tree is grown top-down level by level, pruning those branches that cannot contain a frequent item set [4].

2.1. Node Organization

There are different data structures that may be used for the nodes of the prefix tree. In the first place, we may use simple vectors of integer numbers to represent the counters for the item sets. The items (note that we only need one item to distinguish between the counters of a node, see above) are not explicitly stored in this case, but are implicit in the vector index. Alternatively, we may use vectors, each element of which consists of an item identifier (an integer number) and a counter, with the vector elements being sorted by the item identifier.

The first structure has the advantage that we do not need any memory to store the item identifiers and that we can very quickly find the counter for a given item (simply use the item identifier as an index), but it has the disadvantage that we may have to add "unnecessary" counters (i.e., counters for item sets, of which we know from the information gathered in previous steps that they must be infrequent), because the vector may not have "gaps". This problem can only partially be mitigated by enhancing the vector with an offset to the first element and a size, so that unnecessary counters at the margins of the vector can be discarded. The second structure has the advantage that we only have the counters we actually need, but it has the disadvantage that we need extra memory to store the item identifiers and that we have to carry out a binary search in order to find the counter corresponding to a given item.

A third alternative would be to use a hash table per node. However, although this reduces the time needed to access a counter, it increases the amount of memory needed, because for optimal performance a hash table must not be too full. In addition, it does not allow us to exploit easily the order of the items in the counting process (see below). Therefore I do not consider this alternative here.

Obviously, if we want to optimize speed, we should choose simple counter vectors, despite the gap problem. If we want to optimize memory usage, we can decide dynamically, which data structure is more efficient in terms of memory, accepting the higher counter access time due to the binary search if necessary.

It should also be noted that we need a set of child pointers per node, at least for all levels above the currently added one (in order to save memory, one should not create child pointers before one is sure that one needs them). For organizing these pointers there are basically the same options as for organizing the counters. However, if the counters have item identifiers attached, there is an additional possibility: We may draw on the organization of the counters, using the same order of the items and leaving child pointers nil if they are not needed. This can save memory, even though we may have unnecessary nil pointers, because we do not have to store item identifiers a second time.

2.2. Item Coding

It is clear that the way in which the items are coded (i.e., are assigned integer numbers as identifiers) can have a significant impact on the gap problem for pure counter vectors mentioned above. Depending on the coding we may need large vectors with a lot of gaps or we may need only short vectors with few gaps. A good heuristic approach to minimize the number and the size of gaps seems to be this: It is clear that frequent item sets contain items that are frequent individually. Therefore it is plausible that we have only few gaps if we sort the items w.r.t. their frequency, so that the individually frequent items receive similar identifiers if they have similar frequency (and, of course, infrequent items are discarded entirely). In this case it can be hoped that the offset/size representation of a counter vector can eliminate the greater part of the unnecessary counters, because these can be expected to cluster at the vector margins.

Extending this scheme, we may also consider to code the items w.r.t. the number of frequent pairs (or even triples etc.)

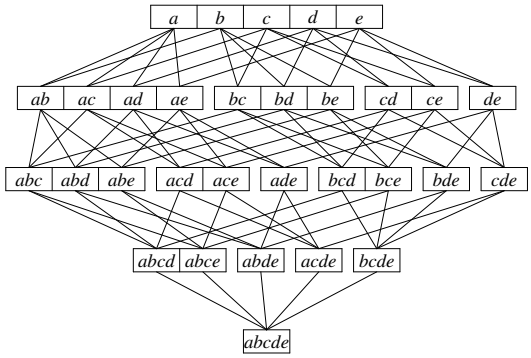


Figure 1. A subset lattice for five items (empty set omitted).

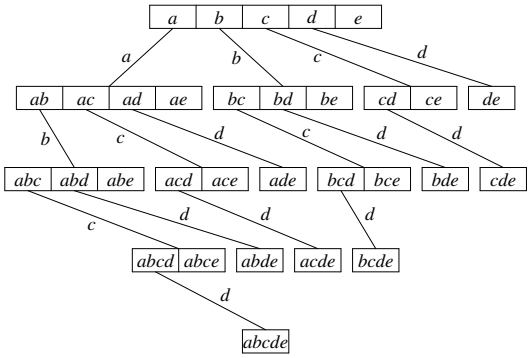


Figure 2. A prefix tree for five items (empty set omitted).

they are part of, thus using additional information from the second (or third etc.) level to improve the coding. This idea can most easily be implemented for item pairs by sorting the items w.r.t. the sum of the sizes of the transactions they are contained in (with infrequent items discarded from the transactions, so that this sum gives a value that is similar to the number of frequent pairs, which, as these are heuristics anyway, is sufficient).

2.3. Recursive Counting

The prefix tree is not only an efficient way to store the counters, it also makes processing the transactions very simple, especially if we sort the items in a transaction ascendingly w.r.t. their identifiers. Then processing a transaction is a simple doubly recursive procedure: To process a transaction for a node of the tree, (1) go to the child corresponding to the first item in the transaction and process the remainder of the transaction recursively for that child and (2) discard the first item of the transaction and process it recursively for the node itself (of course, the second recursion is more easily implemented as a simple loop through the transaction). In a node on the currently added level, however, we increment a counter instead of proceeding to a child node. In this way on the current level all counters for item sets that are part of a transaction are properly incremented.

By sorting the items in a transaction, we can also apply the following optimizations (this is a bit more difficult—or needs additional memory—if hash tables are used to organize the counters and thus explains why I am not considering hash tables): (1) We can directly skip all items before the first item for which there is a counter in the node, and (2) we can abort the recursion if the first item of (the remainder of) a transaction is beyond the last one represented in the node. Since we grow the tree level by level, we can even go a step further: We can terminate the recursion once (the remainder of) a transaction is too short to reach the level currently added to the tree.

2.4. Transaction Representation

The simplest way of processing the transactions is to handle them individually and to apply to each of them the recursive counting procedure described in the preceding section. However, the recursion is a very expensive procedure and therefore it is worthwhile to consider how it can be improved. One approach is based on the fact that often there are several similar transactions, which lead to a similar program flow when they are processed. By organizing the transactions into a prefix tree (an idea that has also been used in [6] in a different approach) transactions with the same prefix can be processed together. In this way the procedure for the prefix is carried out only once and thus considerable performance gains can result. Of course, the gains have to outweigh the additional costs of constructing such a transaction tree to lead to an overall gain.

2.5. Transaction Filtering

It is clear that in order to determine the counter values on the currently added level of the prefix tree, we only need the items that are contained in those item sets that are frequent on the preceding level. That is, to determine the support of item sets of size k, we only need those items that are contained in the frequent item sets of size $k-1$. All other items can be removed from the transactions. This has the advantage that the transactions become smaller and thus can be counted more quickly, because the size of a transaction is a decisive factor for the time needed by the recursive counting scheme described above.

However, this can only be put to work easily if the transactions are processed individually. If they are organized as a prefix tree, a possibly costly reconstruction of the tree is necessary. In this case one has to decide whether to continue with the old tree, accepting the higher counting costs resulting from unnecessary items, or whether rebuilding the tree is preferable, because the costs for the rebuild are outweighed by the savings resulting from the smaller and simpler tree. Good heuristics seem to be to rebuild the tree if

$$\frac{n_{\mathrm{new}}}{n_{\mathrm{curr}}} \frac{t_{\mathrm{tree}}}{t_{\mathrm{count}}} < 0.1,$$

where n_{curr} is the number of items in the current transaction tree, n_{new} is the number of items that will be contained in the new tree, t_{tree} is the time that was needed for building the current tree and t_{count} is the time that was needed for counting the transactions in the preceding step. The constant 0.1 was determined experimentally and on average seems to lead to good results (see also Section 4).

2.6. Filtering Closed and Maximal Item Sets

A frequent item set is called *closed* if there is no superset that has the same support (i.e., is contained in the same number of transactions). Closed item sets capture all information about the frequent item sets, because from them the support of any frequent item set can be determined.

A frequent item set is called *maximal* if there is no superset that is frequent. Maximal item sets define the boundary between frequent and infrequent sets in the subset lattice.

Any frequent item set is often also called a *free* item set to distinguish it from closed and maximal ones.

In order to find closed and maximal item sets with Apriori one may use a simple filtering approach on the prefix tree: The final tree is traversed top-down level by level (breadth first order). For each frequent item set all subsets

with one item less are traversed and marked as not to be reported if they have the same support (closed item sets) or unconditionally (maximal item sets).

3. Eclat Implementation

My Eclat implementation represents the set of transactions as a (sparse) bit matrix and intersects rows to determine the support of item sets. The search follows a depth first traversal of a prefix tree as it is shown in Figure 2.

3.1. Bit Matrices

A convenient way to represent the transactions for the Eclat algorithm is a bit matrix, in which each row corresponds to an item, each column to a transaction (or the other way round). A bit is set in this matrix if the item corresponding to the row is contained in the transaction corresponding to the column, otherwise it is cleared.

There are basically two ways in which such a bit matrix can be represented: Either as a true bit matrix, with one memory bit for each item and transaction, or using for each row a list of those columns in which bits are set. (Obviously the latter representation is equivalent to using a list of transaction identifiers for each item.) Which representation is preferable depends on the density of the dataset. On 32 bit machines the true bit matrix representation is more memory efficient if the ratio of set bits to cleared bits is greater than 1:31. However, it is not advisable to rely on this ratio in order to decide between a true and a sparse bit matrix representation, because in the search process, due to the intersections carried out, the number of set bits will decrease. Therefore a sparse representation should be used even if the ratio of set bits to cleared bits is greater than 1:31. In my current implementation a sparse representation is preferred if the ratio is greater than 1:7, but this behavior can be changed by a user.

A more sophisticated option would be to switch to the sparse representation of a bit matrix during the search once the ratio of set bits to cleared bits exceeds 1:31. However, such an automatic switch, which involves a rebuild of the bit matrix, is not implemented in the current version.

3.2. Search Tree Traversal

As already mentioned, Eclat searches a prefix tree like the one shown in Figure 2 in depth first order. The transition of a node to its first child consists in constructing a new bit matrix by intersecting the first row with all following rows. For the second child the second row is intersected with all following rows and so on. The item corresponding to the row that is intersected with the following rows thus is added to form the common prefix of the item sets processed in the corresponding child node. Of course, rows corresponding to infrequent item sets should be discarded from the constructed matrix, which can be done most conveniently if we store with each row the corresponding item identifier rather than relying on an implicit coding of this item identifier in the row index.

Intersecting two rows can be done by a simple logical *and* on a fixed length integer vector if we work with a true bit matrix. During this intersection the number of set bits in the intersection is determined by looking up the number of set bits for given word values (i.e., 2 bytes, 16 bits) in a precomputed table. For a sparse representation the column indices for the set bits should be sorted ascendingly for efficient processing. Then the intersection procedure is similar to the merge step of merge sort. In this case counting the set bits is straightforward.

3.3. Item Coding

As for Apriori the way in which items are coded has an impact on the execution time of the Eclat algorithm. The reason is that the item coding not only affects the number and the size of gaps in the counter vectors for Apriori, but also the structure of the *pruned* prefix tree and thus the structure of Eclat's search tree. Sorting the items usually leads to a better structure. For the sorting there are basically the same options as for Apriori (see Section 2.2).

3.4. Filtering Closed and Maximal Item Sets

Determining closed and maximal item sets with Eclat is slightly more difficult than with Apriori, because due to the backtrack Eclat "forgets" everything about a frequent item set once it is reported. In order to filter for closed and maximal item sets, one needs a structure that records these sets, and which allows to determine quickly whether in this structure there is an item set that is a superset of a newly found set (and whether this item set has the same support if closed item sets are to be found).

In my implementation I use the following approach to solve this problem: Frequent item sets are reported in a node of the search tree *after* all of its child nodes have been processed. In this way it is guaranteed that all possible supersets of an item set that is about to be reported have already been processed. Consequently, we can maintain a repository of already found (closed or maximal) item sets and only have to search this repository for a superset of the item set in question. The repository can only grow (we never have to remove an item set from it), because due to the report order a newly found item set cannot be a superset of an item set in the repository.

For the repository one may use a bit matrix in the same way as it is used to represent the transactions: Each row

corresponds to an item, each column to a found (closed or maximal) frequent item set. The superset test consists in intersecting those rows of this matrix that correspond to the items in the frequent item set in question. If the result is empty, there is no superset in the repository, otherwise there is (at least) one. (Of course, the intersection loop is terminated as soon as an intersection gets empty.)

To include the information about the support for closed item sets, an additional row of the matrix is constructed, which contains set bits in those columns that correspond to item sets having the same support as the one in question. With this additional row the intersection process is started.

It should be noted that the superset test can be avoided if any direct descendant (intersection product) of an item set has the same support (closed item sets) or is frequent (maximal item set).

In my implementation the repository bit matrix uses the same representation as the matrix that represents the transactions. That is, either both are true bit matrices or both are sparse bit matrices.

4. Experimental Results

I ran experiments with both programs on five data sets, which exhibit different characteristics, so that the advantages and disadvantages of the two approaches and the different optimizations can be observed. The data sets I used are: BMS-Webview-1 (a web click stream from a leg-care company that no longer exists, which has been used in the KDD cup 2000 [7, 9]), T10I4D100K (an artificial data set generated with IBM's data generator [10]), census (a data set derived from an extract of the US census bureau data of 1994, which was preprocessed by discretizing numeric attributes), chess (a data set listing chess end game positions for king vs. king and rook), and mushroom (a data set describing poisonous and edible mushrooms by different attributes). The last three data sets are available from the UCI machine learning repository [3]. The discretization of the numeric attributes in the census data set was done with a shell/gawk script that can be found on the WWW page mentioned below. For the experiments I used an AMD Athlon XP 2000+ machine with 756 MB main memory running S.u.S.E. Linux 8.2 and gcc version 3.3.

The results for these data sets are shown in Figures 3 to 7. Each figure consists of five diagrams, a to e, which are organized in the same way in each figure. Diagram a shows the decimal logarithm of the number of free (solid), closed (short dashes), and maximal item sets (long dashes) for different support values. From these diagrams it can already be seen that the data sets have clearly different characteristics. Only census and chess appear to be similar.

Diagrams b and c show the decimal logarithm of the execution time in seconds for different parameterizations of Apriori (diagram b) and Eclat (diagram c). To ease the comparison of the two diagrams, the default parameter curve for the other algorithm (the solid curve in its own diagram) is shown in grey in the background.

The curves in diagram b represent the following settings:

solid: Items sorted ascendingly w.r.t. the sum of the sizes of the transactions they are contained in; prefix tree to represent the transactions, which is rebuild every time the heuristic criterion described in section 2.5 is fulfilled.
short dashes: Like solid curve, prefix tree used to represent the transactions, but never rebuild.
long dashes: Like solid curve, but transactions are *not* organized as a prefix tree; items that are no longer needed are *not* removed from the transactions.
dense dots: Like long dash curve, but items sorted ascendingly w.r.t. their frequency in the transactions.

In diagram b it is not distinguished whether free, closed, or maximal item sets are to be found, because the time for filtering the item sets is negligible compared to the time needed for counting the transactions (only a small difference would be visible in the diagrams, which derives mainly from the fact that less time is needed to write the smaller number of closed or maximal item sets).

In diagram c the solid, short, and long dashes curve show the results for free, closed, and maximal item sets, respectively, with one representation of the bit matrix, the dense dots curve the results for free item sets for the other representation (cf. section 3.1). Whether the solid, short, and long dashes curve refer to a true bit matrix and the dense dots curve to a sparse one or the other way round depends on the data set and is indicated in the corresponding section below.

Diagrams d and e show the decimal logarithm of the memory in bytes used for different parameterizations of Apriori (diagram d) and Eclat (diagram e). Again the grey curve refers to the default parameter setting of the other algorithm (the solid curve in its own diagram).

The curves in diagram d represent the following settings:

solid: Items sorted ascendingly w.r.t. the sum of the sizes of the transaction they are contained in; transactions organized as a prefix tree; memory saving organization of the prefix tree nodes as described in section 2.1.
short dashes: Like solid, but *no* memory saving organization of the prefix tree nodes (always pure vectors).
long dashes: Like short dashes, but items sorted *descendingly* w.r.t. the sum of the sizes of the transaction they are contained in.
dense dots: Like long dashes, but items *not* sorted.

Again it is not distinguished whether free, closed, or maximal item sets are to be found, because this has no influence on the memory usage. The meaning of the line styles in diagram e is the same as in diagram c (see above).

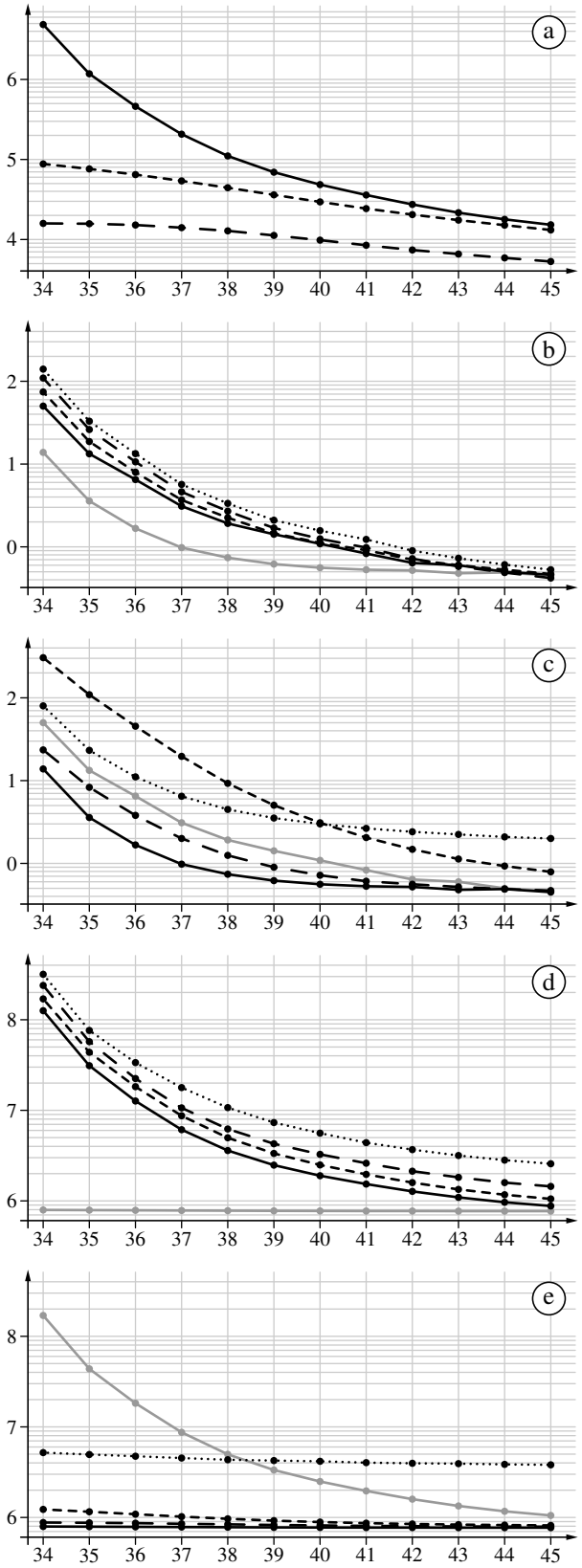


Figure 3. Results on BMS-Webview-1

BMS-Webview-1: Characteristic for this data set is the divergence of the number of free, closed, and maximal item sets for lower support values. W.r.t. the execution time of Apriori this data set shows perfectly the gains that can result from the different optimizations. Sorting the items w.r.t. the sum of transactions sizes (long dashes in diagram b) improves over sorting w.r.t. simple frequency (dense dots), organizing the transactions as a prefix tree (short dashes) improves further, removing no longer needed items yields another considerable speed-up (solid curve). However, for free and maximal item sets and a support less than 44 transactions Eclat with a sparse bit matrix representation (long dashes and solid curve in diagram c) is clearly better than Apriori, which also needs a lot more memory. Only for closed item sets Apriori is the method of choice (Eclat: short dashes in diagram c), which is due to the more expensive filtering with Eclat. Using a true bit matrix with Eclat is clearly not advisable as it performs worse than Apriori and down to a support of 39 transactions even needs more memory (dense dots in diagrams c and e).

T10I4D100K: The numbers of all three types of item sets sharply increase for lower support values; there is no divergence as for BMS-Webview-1. For this data set Apriori outperforms Eclat, although for a support of 5 transactions Eclat takes the lead for free item sets. For closed and maximal item sets Eclat cannot challenge Apriori. It is remarkable that for this data set rebuilding the prefix tree for the transactions in Apriori slightly degrades performance (solid vs. short dashes in diagram b, with the dashed curve almost covered by the solid one). For Eclat a sparse bit matrix representation (solid, short, and long dashes curve in diagrams c and e) is preferable to a true bit matrix (dense dots). (Remark: In diagram b the dense dots curve is almost identical to the long dashes curve and thus is covered.)

Census: This data set is characterized by an almost constant ratio of the numbers of free, closed, and maximal item sets, which increase not as sharply as for T10I4D100K. For free item sets Eclat with a sparse bit matrix representation (solid curve in diagram c) always outperforms Apriori, while it clearly loses against Apriori for closed and maximal item sets (long and short dashes curves in diagrams c and e, the latter of which is not visible, because it lies outside the diagram — the execution time is too large due to the high number of closed item sets). For higher support values, however, using a true bit matrix representation with Eclat to find maximal item sets (sparse dots curves in diagrams c and e) comes close to being competitive with Apriori. Again it is remarkable that rebuilding the prefix tree of transactions in Apriori slightly degrades performance.

Chess: W.r.t. the behavior of the number of free, closed, and maximal item sets this dataset is similar to census, although the curves are bend the other way. The main difference to the results for census are that for this data set a true bit ma-

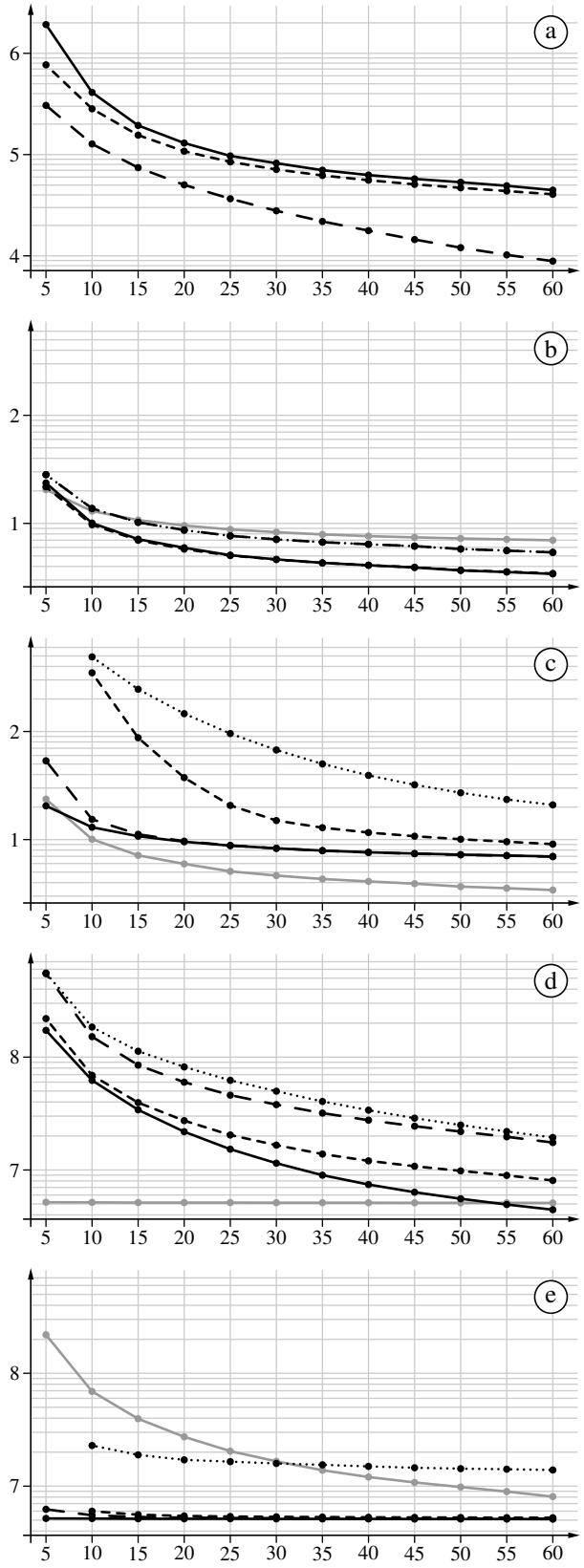


Figure 4. Results on T10I4D100K

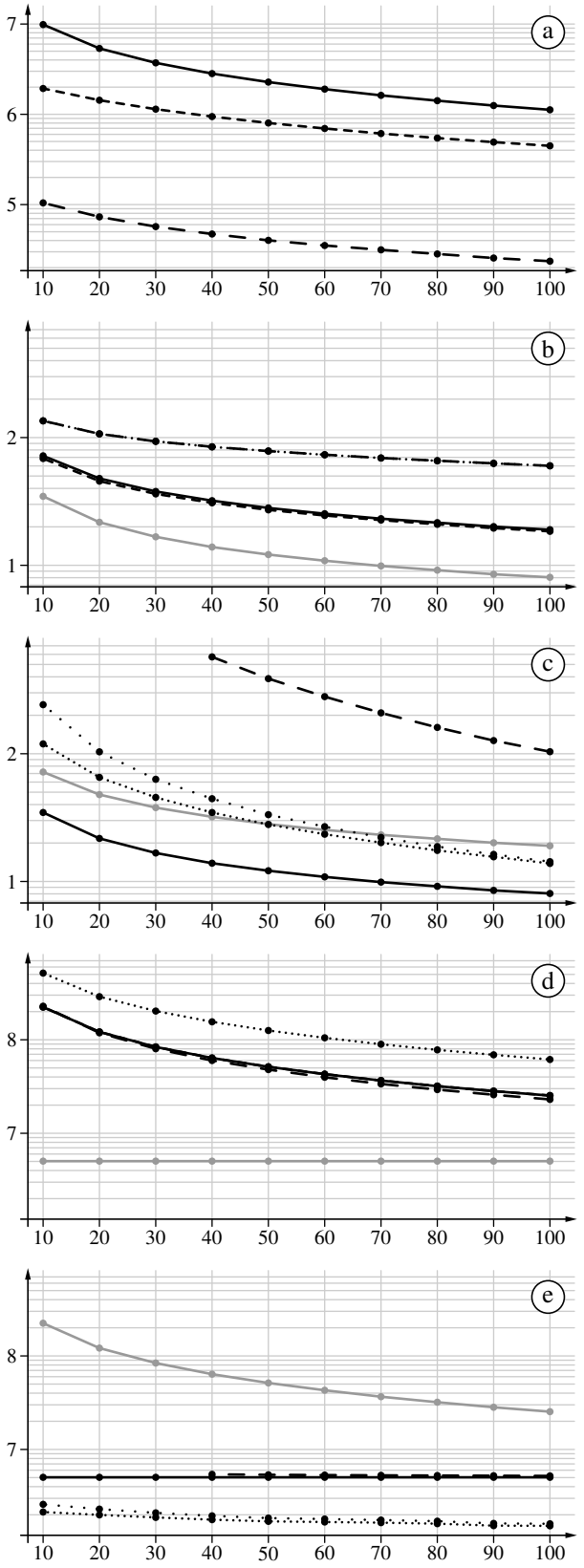


Figure 5. Results on census

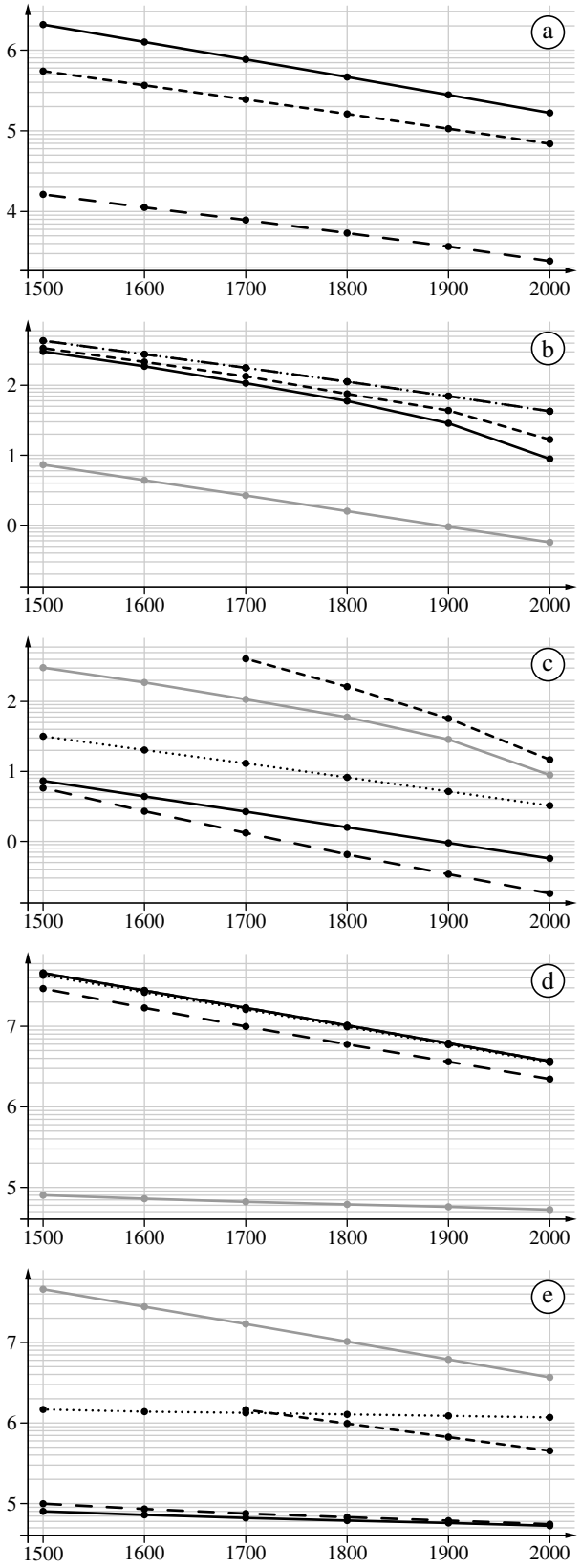


Figure 6. Results on chess

trix representation for Eclat (solid, short, and long dashes curves in diagrams c and e) is preferable to a sparse one (dense dots), while for census it is the other way round. The true bit matrix representation also needs less memory, indicating a very dense data set. Apriori can compete with Eclat only when it comes to closed item sets, where it performs better due to its more efficient filtering of the fairly high number of closed item sets.

Mushroom: This data set differs from the other four in the position of the number of closed data sets between the number of free and maximal item sets. Eclat with a true bit matrix representation (solid, short, and long dashes curves in diagrams c and e) outperforms Eclat with a sparse bit matrix representation (dense dots), which in turn outperforms Apriori. However, the sparse bit matrix (dense dots in diagram c) gains ground towards lower support values, making it likely to take the lead for a minimum support of 100 transactions. Even for closed and maximal item sets Eclat is clearly superior to Apriori, which is due to the small number of closed and maximal item sets, so that the filtering is not a costly factor. (Remark: In diagram b the dense dots curve is almost identical to the long dashes curve and thus is covered. In diagram d the short dashes curve, which lies over the dense dots curve, is covered the solid one.)

5. Conclusions

For free item sets Eclat wins the competition w.r.t. execution time on four of the five data sets and it always wins w.r.t. memory usage. On the only data set on which it loses the competition (T10I4D100K), it takes the lead for the lowest minimum support value tested, indicating that for lower minimum support values it is the method of choice, while for higher minimum support values its disadvantage is almost negligible (note that for this data set all execution times are less than 30s).

For closed item sets the more efficient filtering gives Apriori a clear edge w.r.t. execution time, making it win on all five data sets. For maximal item sets the picture is less clear. If the number of maximal item sets is high, Apriori wins due to its more efficient filtering, while Eclat wins for a lower number of maximal item sets due to its more efficient search.

6. Programs

The implementations of Apriori and Eclat described in this paper (Windows™ and Linux™ executables as well as the source code) can be downloaded free of charge at

http://fuzzy.cs.uni-magdeburg.de/~borgelt/software.html

The special program versions submitted to this workshop rely on the default parameter settings of these programs (solid curves in the diagrams b to e of Section 4).

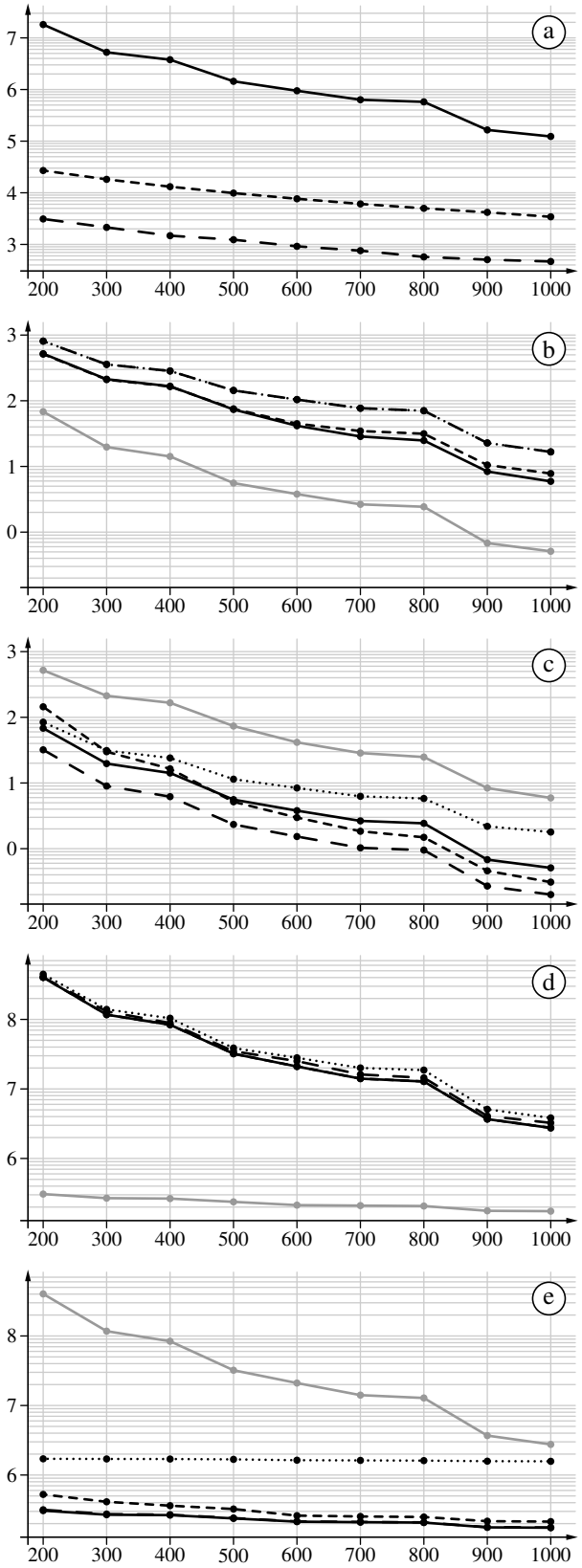


Figure 7. Results on mushroom

References

[1] R. Agrawal, T. Imielienski, and A. Swami. Mining Association Rules between Sets of Items in Large Databases. *Proc. Conf. on Management of Data*, 207–216. ACM Press, New York, NY, USA 1993

[2] A. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A. Verkamo. Fast Discovery of Association Rules. In: [5], 307–328

[3] C.L. Blake and C.J. Merz. *UCI Repository of Machine Learning Databases*. Dept. of Information and Computer Science, University of California at Irvine, CA, USA 1998
http://www.ics.uci.edu/ mlearn/MLRepository.html

[4] C. Borgelt and R. Kruse. Induction of Association Rules: Apriori Implementation. *Proc. 14th Conf. on Computational Statistics (COMPSTAT)*. Berlin, Germany 2002

[5] U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, eds. *Advances in Knowledge Discovery and Data Mining*. AAAI Press / MIT Press, Cambridge, CA, USA 1996

[6] J. Han, H. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In: *Proc. Conf. on the Management of Data (SIGMOD'00, Dallas, TX)*. ACM Press, New York, NY, USA 2000

[7] R. Kohavi, C.E. Bradley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 Organizers' Report: Peeling the Onion. SIGKDD Exploration 2(2):86–93. 2000.

[8] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New Algorithms for Fast Discovery of Association Rules. *Proc. 3rd Int. Conf. on Knowledge Discovery and Data Mining (KDD'97)*, 283–296. AAAI Press, Menlo Park, CA, USA 1997

[9] Z. Zheng, R. Kohavi, and L. Mason. Real World Performance of Association Rule Algorithms. In: *Proc. 7th Int. Conf. on Knowledge Discovery and Data Mining (SIGKDD'01)*. ACM Press, New York, NY, USA 2001

[10] Synthetic Data Generation Code for Associations and Sequential Patterns. http://www.almaden.ibm.com/software/quest/Resources/index.shtml Intelligent Information Systems, IBM Almaden Research Center

Detailed Description of an Algorithm for Enumeration of Maximal Frequent Sets with Irredundant Dualization

Takeaki Uno, Ken Satoh
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan
Email: uno, ksatoh@nii.ac.jp

Abstract

We describe an implementation of an algorithm for enumerating all maximal frequent sets using irredundant dualization, which is an improved version of that of Gunopulos et al. The algorithm of Gunopulos et al. solves many dualization problems, and takes long computation time. We interleaves dualization with the main algorithm, and reduce the computation time for dualization by as long as one dualization. This also reduces the space complexity. Moreover, we accelerate the computation by using sparseness.

1. Introduction

Let E be an item set and $\mathcal{T}$ be a set of transactions defined on E. For an item set $S \subseteq E$, we denote the set of transactions including S by $X(S)$. We define the *frequency* of S by $|X(S)|$. For a given constant α, if an item set S satisfies $|X(S)| \geq \alpha$, then S is said to be *frequent*. A frequent item set included in no other frequent item set is said to be *maximal*. An item set not frequent is called *infrequent*. An infrequent item set including no other infrequent item set is said to be *minimal*.

This paper describes an implementation of an algorithm for enumerating all maximal frequent sets using dualization in detail presented at [SatohUno03]. The algorithm is an improved version of that of Gunopulos et al. [Gunopulos97a, Gunopulos97b]. The algorithm computes maximal frequent sets based on **computing minimal transversals of a hypergraph**, **computing minimal hitting set**, or, in other words, **computing a dualization of a monotone function** [Fredman96]. The algorithm finds all minimal item sets not included in any current obtained maximal frequent set by dualization. If a frequent item set is in those minimal item sets, then the algorithm finds a new maximal frequent set including the frequent item set. In this way, the algorithm avoids checking all frequent item sets. However, this algorithm solves dualization problems many times, hence it is not fast for practical purpose. Moreover, the algorithm uses the dualization algorithm of Fredman and Khachiyan [Fredman96] which is said to be slow in practice.

We improved the algorithm in [SatohUno03] by using incremental dualization algorithms proposed by Kavvadias and Stavropoulos [Kavvadias99], and Uno [Uno02]. We developed an algorithm by interleaving dualization with finding maximal frequent sets. Roughly speaking, our algorithm solves one dualization problem with the size $|Bd^+|$, in which Bd^+ is the set of maximal frequent sets, while the algorithm of Gunopulos et al. solves $|Bd^+|$ dualization problems with sizes from 1 through $|Bd^+|$. This reduces the computation time by a factor of $1/|Bd^+|$.

To reduce the computation time more, we used Uno's dualization algorithm [Uno02]. The experimental computation time of Uno's algorithm is linear in the number of outputs, and $O(|E|)$ per output, while that of Kavvadias and Stavropoulos seems to be $O(|E|^2)$. This reduces the computation time by a factor of $1/|E|$. Moreover, we add an improvement based on sparseness of input. By this, the experimental computation time per output is reduced to $O(ave(Bd^+))$ where $ave(Bd^+)$ is the average size of maximal frequent sets. In summary, we reduced the computation time by a factor of $ave(Bd^+)$ / $(|Bd^+| \times |E|^2)$ by using the combination of the algorithm of Gunopulos et al. and the algorithm of Kavvadias and Stavropoulos.

In the following sections, we describe our algorithm and the computational result. Section 2 describes the algorithm of Gunopulos et al. and Section 3 describes our algorithm and Uno's algorithm. Section 4 explains our improvement using sparseness. Computational experiments for FIMI'03 instances are shown in Section 5, and we conclude the paper in Section 6.

Dualize and Advance[Gunopulos97a]
1 $Bd^+ := \{go\_up(\emptyset)\}$
2 Compute $MHS(\overline{Bd^+})$.
3 If no set in $MHS(\overline{Bd^+})$ is frequent, output $MHS(\overline{Bd^+})$.
4 If there exists a frequent set S in $MHS(\overline{Bd^+})$, $Bd^+ := Bd^+ \cup \{go\_up(S)\}$ and go to 2.

Figure 1: Dualize and Advance Algorithm

2. Enumerating maximal frequent sets by dualization

In this section, we describe the algorithm of Gunopulos et al. Explanations are also in [Gunopulos97a, Gunopulos97b, SatohUno03], however, those are written with general terms. In this section, we explain in terms of frequent set mining.

Let Bd^- be the set of minimal infrequent sets. For a subset family $\mathcal{H}$ of E, a **hitting set** HS of $\mathcal{H}$ is a set such that for every $S \in \mathcal{H}$, $S \cap HS \neq \emptyset$. If a hitting set includes no other hitting set, then it is said to be **minimal**. We denote the set of all minimal hitting sets of $\mathcal{H}$ by $MHS(\mathcal{H})$. We denote the complement of a subset S w.r.t. E by $\overline{S}$. For a subset family $\mathcal{H}$, we denote $\{\overline{S}|S \in \mathcal{H}\}$ by $\overline{\mathcal{H}}$.

There is a strong connection between the maximal frequent sets and the minimal infrequent sets by the minimal hitting set operation.

Proposition 1 *[Mannila96]* $Bd^- = MHS(\overline{Bd^+})$

Using the following proposition, Gunopulos et al. proposed an algorithm called *Dualize and Advance* shown in Fig. 1 to compute the maximal frequent sets [Gunopulos97a].

Proposition 2 *[Gunopulos97a] Let $Bd^+ \subseteq Bd^+$. Then, for every $S \in MHS(\overline{Bd^+})$, either $S \in Bd^-$ or S is frequent (but not both).*

In the above algorithm, $go\_up(S)$ for a subset S of E is a maximal frequent set which is computed as follows.

1. Select one element e from $\overline{S}$ and check the frequency of $S \cup \{e\}$.
2. If it is frequent, $S := S \cup \{e\}$ and go to 1.
3. Otherwise, if there is no element e in $\overline{S}$ such that $S \cup \{e\}$ is frequent, then return S.

Proposition 3 *[Gunopulos97a] The number of frequency checks in the "Dualize and Advance" algorithm to compute Bd^+ is at most $|Bd^+| \cdot |Bd^-| + |Bd^+| \cdot |E|^2$.*

Basically, the algorithm of Gunopulos et al. solves dualization problems with sizes from 1 through $|Bd^+|$. Although we can terminate dualization when we find a new maximal frequent set, we may check each minimal infrequent item set again and again. This is one of the reasons that the algorithm of Gunopulos et al. is not fast in practice. In the next section, we propose a new algorithm obtained by interleaving *gp_up* into a dualization algorithm. The algorithm basically solves one dualization problem of size $|Bd^+|$.

3. Description of our algorithm

The key lemma of our algorithm is the following.

Lemma 1 *[SatohUno03] Let Bd_1^+ and Bd_2^+ be subsets of Bd^+. If $Bd_1^+ \subseteq Bd_2^+$,*

$$MHS(\overline{Bd_1^+}) \cap Bd^- \subseteq MHS(\overline{Bd_2^+}) \cap Bd^-$$

Suppose that we have already found minimal hitting sets corresponding to $\overline{Bd^+}$ of a subset Bd^+ of the maximal frequent sets. The above lemma means that if we add a maximal frequent set to Bd^+, any minimal hitting set we found which corresponds to a minimal infrequent set is still a minimal infrequent set. Therefore, if we can use an algorithm to visit each minimal hitting set based on an incremental addition of maximal frequent sets one by one, we no longer have to check the same minimal hitting set again even if maximal frequent sets are newly found. The dualization algorithms proposed by Kavvadias and Stavropoulos [Kavvadias99] and Uno[Uno02] are such kinds of algorithms. Using these algorithms, we reduce the number of checks.

Let us show Uno's algorithm [Uno02]. This is an improved version of Kavvadias and Stavropoulos's algorithm [Kavvadias99]. Here we introduce some notation. A set $S \in \mathcal{H}$ is called *critical* for $e \in hs$, if $S \cap hs = \{e\}$. We denote a family of critical sets for e w.r.t. hs and $\mathcal{H}$ as $crit(e, hs)$. Note that mhs is a minimal hitting set of $\mathcal{H}$ if and only if for every $e \in mhs$, $crit(e, mhs)$ is not empty.

```
global S_0, ..., S_m;
compute_mhs(i, mhs) /* mhs is a minimal hitting set of S_0, ..., S_i */
begin
1   if i == m then output mhs and return;
2   else if S_{i+1} ∩ mhs ≠ ∅ then compute_mhs(i + 1, mhs);
    else
    begin
3   for every e ∈ S_{i+1} do
4     if for every e' ∈ mhs, there exists S_j ∈ crit(e', mhs), j ≤ i
        s.t. S_j does not contain e then
5       comupute_mhs(i + 1, mhs ∪ {e});
    end
    return;
end
```

Figure 2: Algorithm to Enumerate Minimal Hitting Sets

Suppose that $\mathcal{H} = \{S_1, ..., S_m\}$, and let MHS_i be $MHS(\{S_0, ..., S_i\})(1 \leq i \leq n)$. We simply denote $MHS(\mathcal{H})$ by MHS. A hitting set hs for $\{S_1, ..., S_i\}$ is minimal if and only if $crit(e, hs) \cap \{S_1, ..., S_i\} \neq \emptyset$ for any $e \in hs$.

Lemma 2 *[Uno02] For any $mhs \in MHS_i(1 \leq i \leq n)$, there exists just one minimal hitting set $mhs' \in MHS_{i-1}$ satisfying either of the following conditions (but not both),*

- $mhs' = mhs$.
- $mhs' = mhs \setminus \{e\}$ *where* $crit(e, mhs) \cap \{S_0, ..., S_i\} = \{S_i\}$.

We call mhs' the *parent* of mhs, and mhs a *child* of mhs'. Since the parent-child relationship is not cyclic, its graphic representation forms a forest in which each of its connected components is a tree rooted at a minimal hitting set of MHS_1. We consider the trees as traversal routes defined for all minimal hitting sets of all MHS_i. These transversal routes can be traced in a depth-first manner by generating children of the current visiting minimal hitting set, hence we can enumerate all minimal hitting sets of MHS in linear time of $\sum_i |MHS_i|$. Although $\sum_i |MHS_i|$ can be exponential to $|MHS|$, such cases are expected to be exceptional in practice. Experimentally, $\sum_i |MHS_i|$ is linear in $|MHS|$.

To find children of a minimal hitting set, we use the following proposition that immediately leads from the above lemma.

Proposition 4 *[Uno02]*
Any child mhs' of $mhs \in MHS_i$ satisfies one of the following conditions.
(1) $mhs' = mhs$
(2) $mhs' = mhs \cup \{e\}$
In particular, no mhs has a child satisfying (1) and a child satisfying (2).

If $mhs \cap S_{i+1} \neq \emptyset$ then $mhs \in MHS_{i+1}$, and (1) holds. If $mhs \cap S_{i+1} = \emptyset$, then $mhs \notin MHS_{i+1}$, and (2) can hold for some $e \in S_{i+1}$. If $mhs' = mhs \cup \{e\}$ is a child of mhs, then for any $e' \in mhs$, there is $S_j \in crit(e', mhs), j \leq i$ such that $e \notin S_j$. From these observations, we obtain the algorithm described in Fig. 2.

An iteration of the algorithm in Fig. 2 takes:

- O($|mhs|$) time for line 1.
- O($|S_{i+1} \cup mhs|$) time for line 2.
- O($(|E| - |mhs|) \times \sum_{e' \in mhs} |crit(e', mhs) \cap \{S_0, ..., S_i\}|$) time for lines 3 to 5, except for the computation of *crit*.

To compute *crit* quickly, we store $crit(e, mhs)$ in memory, and update them when we generate a recursive call. Note that this takes O(m) memory. Since $crit(e', mhs \cup \{e\})$ is obtained from $crit(e', mhs)$ by removing sets including e (i.e., $crit(e', mhs \cup \{e\}) = \{S | S \in crit(e', mhs), e' \notin S_{i+1}\}$), $crit(e', mhs \cup \{e\})$ for all e' can be computed in O(m) time. Hence the computation time of an iteration is bounded by O($|E| \times m$).

Based on this dualization algorithm, we developed a maximal frequent sets enumeration algorithm. First, the algorithm sets the input $\mathcal{H}$ of the dualization problem to the empty set. Then, the algorithm solves the dualization in the same way as the

Irredundant Border Enumerator
global integer $bdpnum$; sets $bd_0^+, bd_1^+ ....$;
main()
begin
  $bdpnum := 0$;
  $construct\_bdp(0, \emptyset)$;
  output all the $bd_j^+ (0 \leq j \leq bdpnum)$;
end

$construct\_bdp(i, mhs)$
begin
  if $i == bdpnum$ /* minimal hitting set for $\cup_{j:=0}^{bdpnum} \overline{bd_j^+}$ is found */
    then goto 1 **else goto** 2

1. **if** mhs is not frequent, **return**; /* new Bd^- element is found */
   $bd^+_{bdpnum} := go\_up2(mhs)$; /* new Bd^+ element is found */
   $bdpnum := bdpnum + 1$; /* proceed to 2 */

2. **if** $\overline{bd_i^+} \cap mhs \neq \emptyset$ **then** $construct\_bdp(i+1, mhs)$;
   else
   begin
   for every $e \in \overline{bd_i^+}$ **do**
     if $bd_i^+ \cup \{e\}$ is a minimal hitting set of $\{\overline{bd_0^+}, \overline{bd_1^+} ..., \overline{bd_{i-1}^+}\})$ **then** $construct\_bdp(i+1, mhs \cup \{e\})$;
   return;

end

Figure 3: Algorithm to Check Each Minimal Hitting Set Only Once

above algorithm. When a minimal hitting set mhs is found, the algorithm checks its frequency. If mhs is frequent, the algorithm finds a maximal frequent set S including it, and adds $\overline{S}$ to $\mathcal{H}$ as a new element of $\mathcal{H}$. Now mhs is not a minimal hitting set since $\overline{S} \cap mhs = \emptyset$. The algorithm continues generating a recursive call to find a minimal hitting set of the updated $\mathcal{H}$. In the case that mhs is not frequent, from Lemma 1, mhs continues to be a minimal hitting set even when $\mathcal{H}$ is updated. Hence, we backtrack and find other minimal hitting sets.

When the algorithm terminates, $\overline{\mathcal{H}}$ is the set of maximal frequent sets, and the set of all minimal hitting sets the algorithm found is the set of minimal infrequent sets. The recursive tree the algorithm generated is a subtree of the recursive tree obtained by Uno's dualization algorithm inputting $\overline{Bd^+}$, which is the set of the complement of maximal frequent sets.

This algorithm is described in Fig. 3. We call the algorithm *Irredundant Border Enumerator* (IBE algorithm, for short).

Theorem 1 *The computation time of IBE is* $O(Dual(\overline{Bd^+}) + |Bd^+|g)$, *where* $Dual(\overline{Bd^+})$ *is the computation time of Uno's algorithm for dualizing* $\overline{Bd^+}$, *and* g *is the computation time for go_up.*

Note also that, the space complexity of the IBE algorithm is $O(\Sigma_{S \in Bd^+} |S|)$ since all we need to memorize is Bd^+ and once a set in Bd^- is checked, it is no longer need to be recorded. On the other hand, Gunopulos et al. [Gunopulos97a] suggests a usage of Fredman and Khachiyan's algorithm [Fredman96] which needs a space of $O(\Sigma_{S \in (Bd^+ \cup Bd^-)} |S|)$ since the algorithm needs both Bd^+ and Bd^- at the last stage.

4. Using sparseness

In this section, we speed up the dualization phase of our algorithm by using a sparseness of $\mathcal{H}$. In real data, the sizes of maximal frequent sets are usually small. They are often bounded by a constant. We use this sparse structure for accelerating the algorithm.

```
global S_0, ..., S_m;
compute_mhs(i, mhs) /* mhs is a minimal hitting set of S_0, ..., S_i */
begin
1   if uncov(mhs) == ∅ then output mhs and return;
2   i := minimum index of uncov(mhs) ;
3   for every e ∈ mhs do
4     increase the counter of items in ∪_{S∈crit(mhs,e)} S̄ by one
    end
5   for every e' ∉ mhs s.t. counter is increased by |mhs| do /* items included in all ∪_{S∈crit(mhs,e)} S̄ */
6     compute_mhs(i + 1, mhs ∪ {e});
    return;
end
```

Figure 4: Improved Dualization Algorithm Using Sparseness

First, we consider a way to reduce the computation time of iterations. Let us see the algorithm described in Fig. 2. The bottle neck part of the computation of an iteration of the algorithm is lines 3 to 5, which check the existence of a critical set $S_j \in crit(mhs, e'), j < i$ such that $e \notin S_j$. To check this condition for an item $e \notin mhs$, we spend $O(\sum_{e' \in mhs} |crit(mhs, e')|)$ time, hence this check for all $e \notin mhs$ takes $O((|E| - |mhs|) \times \sum_{e' \in mhs} |crit(mhs, e')|)$ time.

Instead of this, we compute $\bigcup_{S \in crit(mhs,e)} \overline{S}$ for each $e \in mhs$. If and only if $e' \in \bigcup_{S \in crit(mhs,e)} \overline{S}$ for all $e \in mhs$, e' satisfies the condition of "if" at line 4. To compute $\bigcup_{S \in crit(mhs,e)} \overline{S}$ for all $e \in mhs$, we take $O(\sum_{e \in mhs} \sum_{S \in crit(mhs,e)} |\overline{S}|)$ time. In the case of IBE algorithm, $\overline{S}$ is a maximal frequent set, hence the average size of $|\overline{S}|$ is expected to be small. The sizes of minimal infrequent sets are not greater than the maximum size of maximal frequent sets, and they are usually smaller than the average size of the maximal frequent sets. Hence, $|mhs|$ is also expected to be small.

Second, we reduce the number of iterations. For $mhs \subseteq E$, we define $uncov(mhs)$ by the set of $S \in \mathcal{H}$ satisfying $S \cap mhs = \emptyset$. If $mhs \cap S_i \neq \emptyset$, the iteration inputting mhs and i does nothing but generates a recursive call with increasing i by one. This type of iterations should be skipped. Only iterations executing lines 3 to 5 are crucial. Hence, in each iteration, we set i to the minimum index among $uncov(mhs)$. As a result of this, we need not execute line 2, and the number of iterations is reduced from $\sum_i |MHS_i|$ to $|\bigcup_i MHS_i|$. We describe the improved algorithm in Fig. 4.

In our implementation, when we generate a recursive call, we allocate memory for each variable used in the recursive call. Hence, the memory required by the algorithm can be up to $O(|E| \times m)$. However, experimentally the required memory is always linear in the input size. Note that we can reduce the worst case memory complexity by some sophisticated algorithms.

5. Experiments

In this section, we show some results of the computational experiments of our algorithms. We implement our algorithm using C programming language, and examined instances of FIMI2003. For instances of KDD-cup 2000[KDDcup00], we compared the results to the computational experiments of CHARM[Zaki02], closed[Pei00], FP-growth[Han00], and Apriori[Agrawal96] shown in [Zheng01]. The experiments in [Zheng01] were done on a PC with a Duron 550MHz CPU and 1GB RAM memory. Our experiments were done on a PC with a Pentium III 500MHz CPU and 256MB RAM memory, which is little slower than a Duron 550MHz CPU. The results are shown in Figs. 4 – 14. Note that our algorithm uses at most 170MB for any following instance. We also show the number of frequent sets, frequent closed/maximal item sets, and minimal frequent sets.

In our experiments, IBE algorithm takes approximately $O(|Bd^-| \times ave(Bd^+))$ time, while the computation time of other algorithms deeply depends on the number of frequent sets, the number of frequent closed item sets, and the minimum support. We recall that $ave(Bd^+)$ is the average size of maximal frequent sets. In some instances, our IBE algorithm performs rather well compared to other algorithms. In these cases, the number of maximal frequent item

sets is smaller than number of frequent item sets. IBE algorithm seems to give a good performance for difficult problems such that the number of maximal frequent sets is very small rather than those of frequent item sets and frequent closed item sets.

6. Conclusion

In this paper, we describe the detailed implementation method of our algorithm proposed in [SatohUno03] and we give some experimental results on test data.

Acknowledgments
We are grateful to Heikki Mannila for participating useful discussions about this research.

References

[Agrawal96] Agrawal, R., Mannila, H., Srikant, R., Toivonen, H., and Verkamo, A. I., "Fast Discovery of Association Rules", U. M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, (eds), Advances in Knowledge Discovery and Data Mining, chapter 12, pp. 307–328 (1996).

[Fredman96] Fredman, M. L. and Khachiyan, L., "On the Complexity of Dualization of Monotone Disjunctive Normal Forms", Journal of Algorithms 21(3), pp. 618 – 628 (1996)

[Gunopulos97a] Gunopulos, D., Khardon, R., Mannila, H. and Toivonen, H., "Data mining, Hypergraph Transversals, and Machine Learning", *Proc. of PODS'97*, pp. 209 – 216 (1997).

[Gunopulos97b] Gunopulos, D., Mannila, H., and Saluja, S., "Discovering All Most Specific Sentences using Randomized Algorithms", *Proc. of ICDT'97*, pp. 215 – 229 (1997).

[Han00] Han, J., Pei, J., Yin, Y., "Mining Frequent Patterns without Candidate Generation," *SIGMOD Conference 2000*, pp. 1-12, 2000

[Kavvadias99] Kavvadias, D. J., and Stavropoulos, E. C., "Evaluation of an Algorithm for the Transversal Hypergraph Problem", *Algorithm Engineering*, pp 72 – 84 (1999).

[KDDcup00] Kohavi, R., Brodley, C. E., Frasca, B., Mason, L., and Zheng, Z., "KDD-Cup 2000 Organizers' Report: Peeling the Onion," *SIGKDD Explorations,* 2(2), pp. 86-98, 2000.

[Mannila96] Mannila, H. and Toivonen, T., "On an Algorithm for Finding All Interesting Sentences", *Cybernetics and Systems, Vol II, The Thirteen European Meeting on Cybernetics and Systems Research*, pp. 973 – 978 (1996).

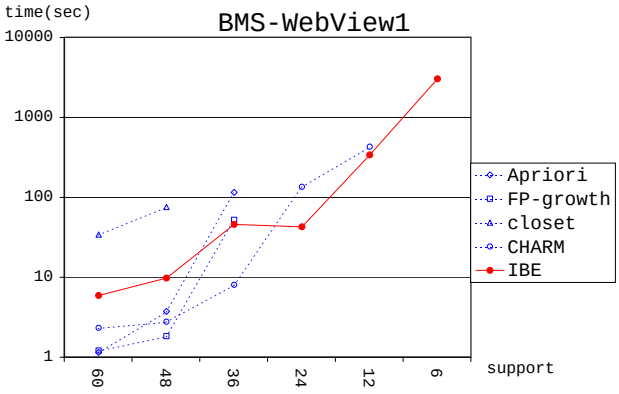
[Pei00] Pei, J., Han, J., Mao, R., "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets," *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery 2000*, pp. 21-30, 2000.

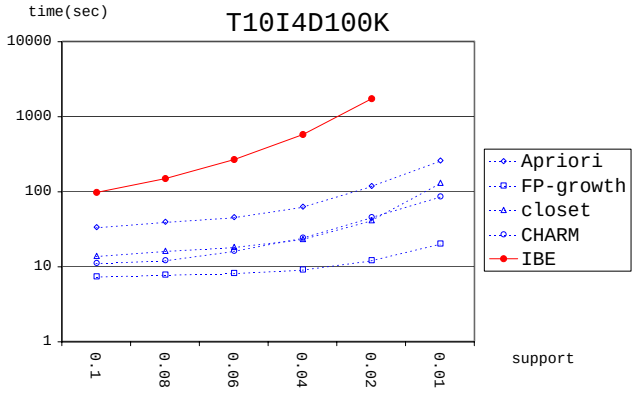
[SatohUno03] Satoh, K., Uno, T., "Enumerating Maximal Frequent Sets using Irredundant Dualization", *Lecture Notes in Artificial Intelligence* (Proc. of Discovery Science 2003), Springer-Varlag, pp. 192-201, 2003.

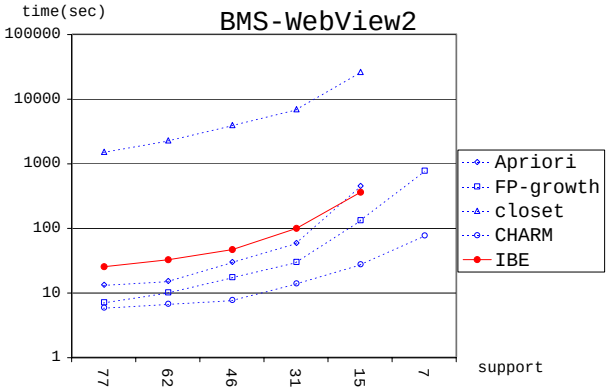
[Uno02] Uno, T., "A Practical Fast Algorithm for Enumerating Minimal Set Coverings", *SIGAL83*, Information Processing Society of Japan, pp. 9 – 16 (in Japanese) (2002).

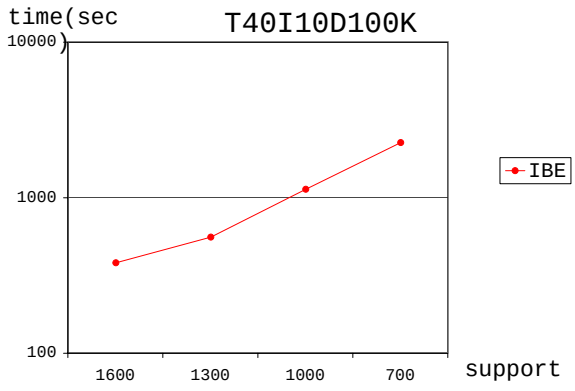
[Zaki02] Zaki, M. J., Hsiao, C., "CHARM: An Efficient Algorithm for Closed Itemset Mining," *2nd SIAM International Conference on Data Mining (SDM'02)*, pp. 457-473, 2002.

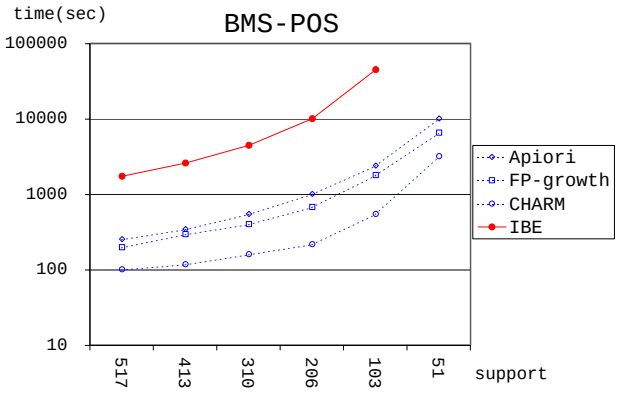
[Zheng01] Zheng, Z., Kohavi, R., and Mason, L., "Real World Performance of Association Rule Algorithms," *KDD 2001*, pp. 401-406, 2001.

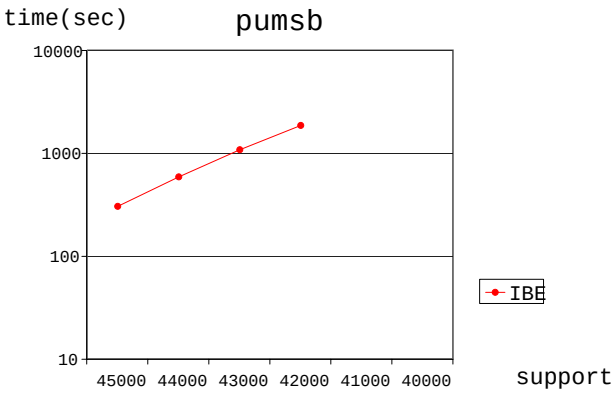

time(sec)
BMS-WebView1
10000
1000
100
10
1
60
48
36
24
12
6
support
Apriori
FP-growth
closet
CHARM
IBE


time(sec)
T10I4D100K
10000
1000
100
10
1
0.1
0.08
0.06
0.04
0.02
0.01
support
Apriori
FP-growth
closet
CHARM
IBE

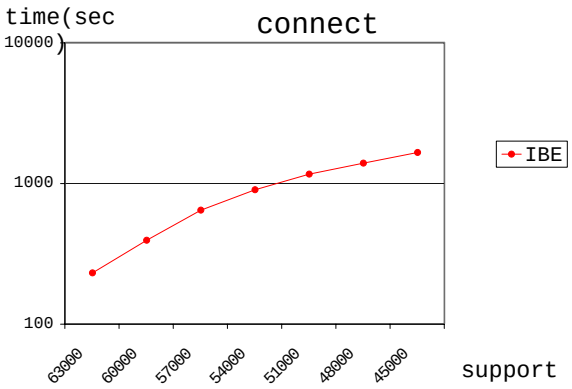

time(sec)
BMS-WebView2
100000
10000
1000
100
10
1
77
62
46
31
15
7
support
Apriori
FP-growth
closet
CHARM
IBE

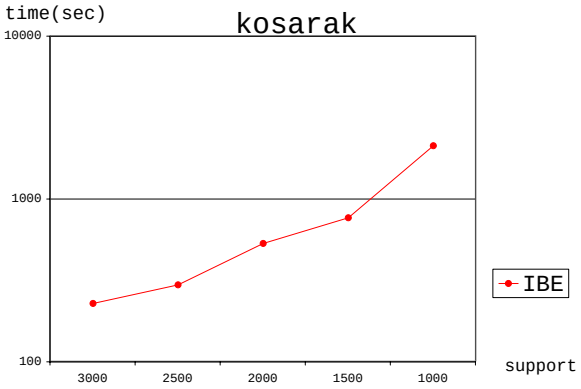

time(sec
T40I10D100K
10000)
1000
100
1600
1300
1000
700
support
IBE

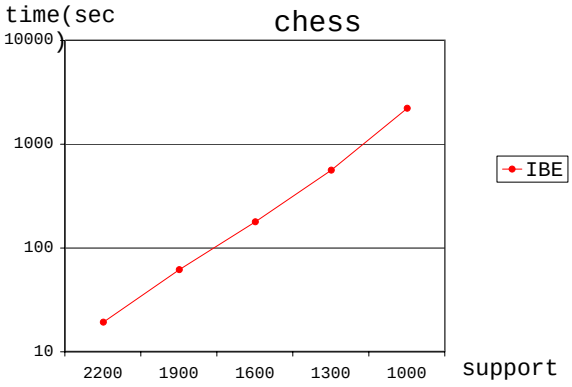

time(sec)
BMS-POS
100000
10000
1000
100
10
517
413
310
206
103
51
support
Apriori
FP-growth
CHARM
IBE

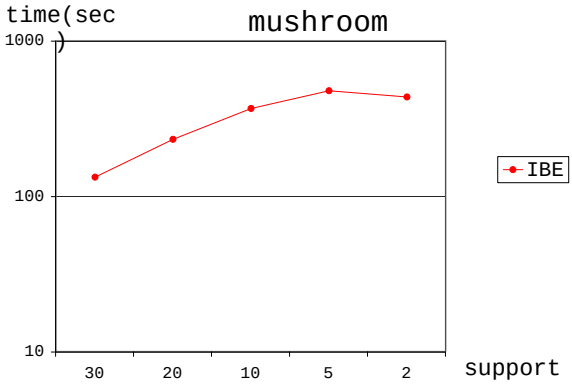

time(sec)
pumsb
10000
1000
100
10
45000
44000
43000
42000
41000
40000
support
IBE


time(sec)
pumsb_star
10000
1000
100
10
1
30000
25000
20000
15000
10000
support
IBE


time(sec
connect
10000)
1000
100
63000
60000
57000
54000
51000
48000
45000
support
IBE


time(sec)
kosarak
10000
1000
100
3000
2500
2000
1500
1000
support
IBE


time(sec
chess
10000)
1000
100
10
2200
1900
1600
1300
1000
support
IBE


time(sec
)
mushroom
1000
100
10
30
20
10
5
2
support
IBE

BMS-Web-View1: #item 497, #transactions, 59602, ave. size of transaction 2.51

support	60	48	36	24	12	6
Apriori	1.1	3.6	113	-	-	-
FP-growth	1.2	1.8	51	-	-	-
Closet	33	74	-	-	-	-
Charm	2.2	2.7	7.9	133	422	-
IBE	5.8	9.6	45	42	333	2982
#freq. sets	3992	10287	461522	-	-	-
#closed sets	3974	9391	64762	155651	422692	1240701
#max. freq. sets	2067	4028	15179	12956	84833	129754
#min. infreq. sets	66629	81393	150278	212073	579508	4320003

maximum use of memory: 45MB

BMS-Web-View2: #items 3340, #transactions 77512, ave. size of transaction 4.62

support	77	62	46	31	15	7
Apriori	13.1	15	29.6	58.2	444	-
Fp-growth	7.03	10	17.2	29.6	131	763
Closet	1500	2250	3890	6840	25800	-
Charm	5.82	6.66	7.63	13.8	27.2	76
IBE	25	32	46	98	355	1426
#closed sets	22976	37099	60352	116540	343818	754924
#freq. sets	24143	42762	84334	180386	1599210	9897303
#max. freq. sets	3901	5230	7841	16298	43837	118022
#min. infreq. sets	657461	958953	1440057	2222510	3674692	5506524

maximal use of memory: 100MB

BMS-POS: #items 1657, #transactions 517255, ave. size of transaction 6.5

support	517	413	310	206	103	51
Apriori	251	341	541	1000	2371	10000
Fp-growth	196	293	398	671	1778	6494
Closet	-	-	-	-	-	-
Charm	100	117	158	215	541	3162
IBE	1714	2564	4409	9951	44328	-
#closed sets	121879	200030	378217	840544	1742055	21885050
#freq. sets	121956	200595	382663	984531	5301939	33399782
#max. freq. sets	30564	48015	86175	201306	891763	4280416
#min. infreq. sets	236274	337309	530946	1047496	3518003	-

maximum use of memory: 110MB

T10I4D100K: #items 1000, #transactions 100000, ave. size of transaction 10

support	100	80	60	40	20	10
Apriori	33	39	45	62	117	256
Fp-growth	7.3	7.7	8.1	9.0	12	20
Closet	13	16	18	23	41	130
Charm	11	13	16	24	45	85
IBE	96	147	263	567	1705	-
#freq. sets	15010	28059	46646	84669	187679	335183
#closed sets	13774	22944	38437	67537	131342	229029
#max. freq. sets	7853	11311	16848	25937	50232	114114
#min. infreq. sets	392889	490203	736589	1462121	4776165	-

maximum use of memory: 60MB

T40I10D100K: #items 1000, #transactions 100000, ave. size of transaction 39.6

support	1600	1300	1000	700
IBE	378	552	1122	2238
#freq. sets	4591	10110	65236	550126
#closed sets	4591	10110	65236	548349
#max. freq. sets	4003	6944	21692	41473
#min. infreq. sets	245719	326716	521417	1079237

maximum memory use: 74MB

pumsb: #items 7117, #transactions 49046, ave. size of transaction 74

support	45000	44000	43000	42000
IBE	301	582	1069	1840
#freq. sets	1163	2993	7044	15757
#closed sets	685	1655	3582	7013
#max. freq. sets	144	288	541	932
#min. infreq. sets	7482	7737	8402	9468

maximum use of memory: 70MB

pumsb_star: #items 7117, #transactions 49046, ave. size of transaction 50

support	30000	25000	20000	15000	10000	5000
IBE	8	19	59	161	556	2947
#freq. sets	165	627	21334	356945	>2G	-
#closed sets	66	221	2314	14274	111849	-
#max. freq. sets	4	17	81	315	1666	15683
#min. infreq. sets	7143	7355	8020	9635	19087	98938

maximum use of memory: 44MB

kosarak: #items 41217, #transactions 990002, ave. size of transaction 8

support	3000	2500	2000	1500	1000
IBE	226	294	528	759	2101
#freq. sets	4894	8561	34483	219725	711424
#closed sets	4865	8503	31604	157393	496675
#max. freq. sets	792	1146	2858	4204	16231
#min. infreq. sets	87974	120591	200195	406287	875391

maximum use of memory: 170MB

mushroom: #items 120, #transactions 8124, ave. size of transaction 23

support	30	20	10	5	2
IBE	132	231	365	475	433
#freq. sets	505205917	781458545	1662769667	>2G	>2G
#closed sets	91122	109304	145482	181243	230585
#max. freq. sets	15232	21396	30809	34131	27299
#min. infreq. sets	66085	79481	81746	69945	31880

maximum use of memory 47MB

connect: #items 130, #transactions 67577, ave. size of transaction 43

support	63000	60000	57000	54000	51000	48000	45000
IBE	229	391	640	893	1154	1381	1643
#freq. sets	6327	41143	171239	541911	1436863	-	-
#closed sets	1566	4372	9041	15210	23329	-	-
#max. freq. sets	152	269	464	671	913	1166	1466
#min. infreq. sets	297	486	703	980	1291	1622	1969

maximum use of memory: 60MB

chess: #items 76, #transactions 3196, ave. size of transaction 37

support	2200	1900	1600	1300	1000
IBE	19	61	176	555	2191
#freq. sets	59181	278734	1261227	5764922	29442848
#closed sets	28358	106125	366529	1247700	4445373
#max. freq. sets	1047	3673	11209	35417	114382
#min. infreq. sets	1725	5202	14969	46727	152317

maximum use of memory: 50MB

Probabilistic Iterative Expansion of Candidates in Mining Frequent Itemsets

Attila Gyenesei and Jukka Teuhola
Turku Centre for Computer Science, Dept. of Inf. Technology, Univ. of Turku, Finland
Email: {gyenesei,teuhola}@it.utu.fi

Abstract

A simple new algorithm is suggested for frequent itemset mining, using item probabilities as the basis for generating candidates. The method first finds all the frequent items, and then generates an estimate of the frequent sets, assuming item independence. The candidates are stored in a trie where each path from the root to a node represents one candidate itemset. The method expands the trie iteratively, until all frequent itemsets are found. Expansion is based on scanning through the data set in each iteration cycle, and extending the subtries based on observed node frequencies. Trie probing can be restricted to only those nodes which possibly need extension. The number of candidates is usually quite moderate; for dense datasets 2-4 times the number of final frequent itemsets, for non-dense sets somewhat more. In practical experiments the method has been observed to make clearly fewer passes than the well-known Apriori method. As for speed, our non-optimised implementation is in some cases faster, in some others slower than the comparison methods.

1. Introduction

We study the well-known problem of finding *frequent itemsets* from a transaction database, see [2]. A *transaction* in this case means a set of so-called *items*. For example, a supermarket basket is represented as a transaction, where the purchased products represent the items. The database may contain millions of such transactions. The frequent itemset mining is a task, where we should find those subsets of items that occur at least in a given minimum number of transactions. This is an important basic task, applicable in solving more advanced data mining problems, for example discovering *association rules* [2]. What makes the task difficult is that the number of potential frequent itemsets is exponential in the number of distinct items.

In this paper, we follow the notations of Goethals [7]. The overall set of items is denoted by I. Any subset $X \subseteq I$ is called an itemset. If X has k items, it is called a k-itemset. A transaction is an itemset identified by a *tid*. A transaction with itemset Y is said to *support* itemset X, if $X \subseteq Y$. The *cover* of an itemset X in a database D is the set of transactions in D that support X. The *support* of itemset X is the size of its cover in D. The *relative frequency* (probability) of itemset X with respect to D is

$$P(X, D) = \frac{Support(X, D)}{|D|} \qquad (1)$$

An itemset X is frequent if its support is greater than or equal to a given threshold σ. We can also express the condition using a relative threshold for the frequency: $P(X, D) \geq \sigma_{rel}$, where $0 \leq \sigma_{rel} \leq 1$. There are variants of the basic 'all-frequent-itemsets' problem, namely the *maximal* and *closed* itemset mining problems, see [1, 4, 5, 8, 12]. However, here we restrict ourselves to the basic task.

A large number of algorithms have been suggested for frequent itemset mining during the last decade; for surveys, see [7, 10, 15]. Most of the algorithms share the same general approach: generate a set of *candidate itemsets*, count their frequencies in D, and use the obtained information in generating more candidates, until the complete set is found. The methods differ mainly in the order and extent of candidate generation. The most famous is probably the *Apriori* algorithm, developed independently by Agrawal et al. [3] and Mannila et al. [11]. It is a representative of *breadth-first* candidate generation: it first finds all frequent 1-itemsets, then all frequent 2-itemsets, etc. The core of the method is clever pruning of candidate k-itemsets, for which there exists a non-frequent k-1-subset. This is an application of the obvious *monotonicity* property: All subsets of a frequent itemset must also be frequent. Apriori is essentially based on this property.

The other main candidate generation approach is *depth-first* order, of which the best-known representatives are Eclat [14] and FP-growth [9] (though the 'candidate' concept in the context of FP-growth is disputable). These two are generally considered to be among the fastest algorithms for frequent itemset mining. However, we shall mainly use Apriori as a reference method, because it is technically closer to ours.

Most of the suggested methods are *analytical* in the sense that they are based on logical inductions to restrict the number of candidates to be checked. Our approach (called *PIE*) is *probabilistic*, based on relative item frequencies, using which we compute estimates for itemset frequencies in candidate generation. More precisely, we generate iteratively improving approximations (candidate itemsets) to the solution. Our general endeavour has been to develop a relatively simple method, with fast basic steps and few iteration cycles, at the cost of somewhat increased number of candidates. However, another goal is that the method should be robust, i.e. it should work reasonably fast for all kinds of datasets.

2. Method description

Our method can be characterized as a generate-and-test algorithm, such as Apriori. However, our candidate generation is based on probabilistic estimates of the supports of itemsets. The testing phase is rather similar to Apriori, but involves special book-keeping to lay a basis for the next generation phase.

We start with a general description of the main steps of the algorithm. The first thing to do is to determine the frequencies of all items in the dataset, and select the frequent ones for subsequent processing. If there are m frequent items, we internally identify them by numbers 0, …, m-1. For each item i, we use its probability (relative frequency) $P(i)$ in the generation of candidates for frequent itemsets.

The candidates are represented as a *trie* structure, which is normal in this context, see [7]. Each node is labelled by one item, and a path of labels from the root to a node represents an itemset. The root itself represents the empty itemset. The paths are sorted, so that a subtrie rooted by item i can contain only items > i. Note also that several nodes in the trie can have the same item label, but not on a single path. A complete trie, storing all subsets of the whole itemset, would have 2^m nodes and be structurally a *binomial tree* [13], where on level j there are $\binom{m}{j}$ nodes, see Fig. 1 for $m = 4$.

The trie is used for book-keeping purposes. However, it is important to avoid building the complete trie, but only some upper part of it, so that the nodes (i.e. their root paths) represent reasonable candidates for frequent sets. In our algorithm, the first approximation for candidate itemsets is obtained by computing estimates for their probabilities, assuming independence of item occurrences. It means that, for example, for an itemset $\{x, y, z\}$ the estimated probability is the product $P(x)P(y)P(z)$. Nodes are created in the trie from root down along all paths as long as the path-related probability is not less that the threshold σ_{rel}. Note that the probability values are monotonically non-increasing on the way down. Fig. 2

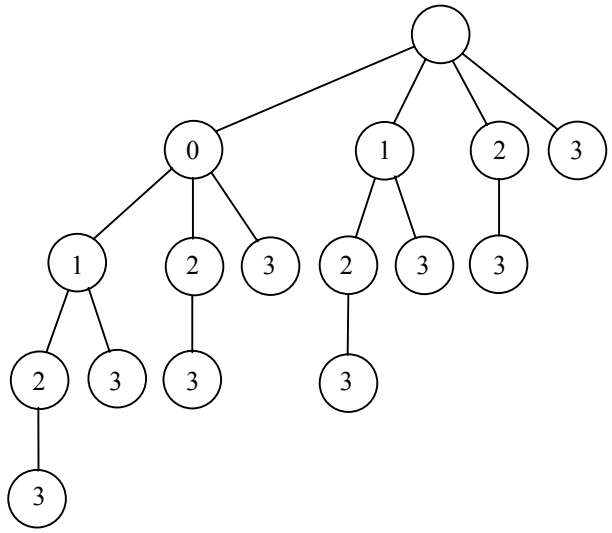


Figure 1. The complete trie for 4 items.

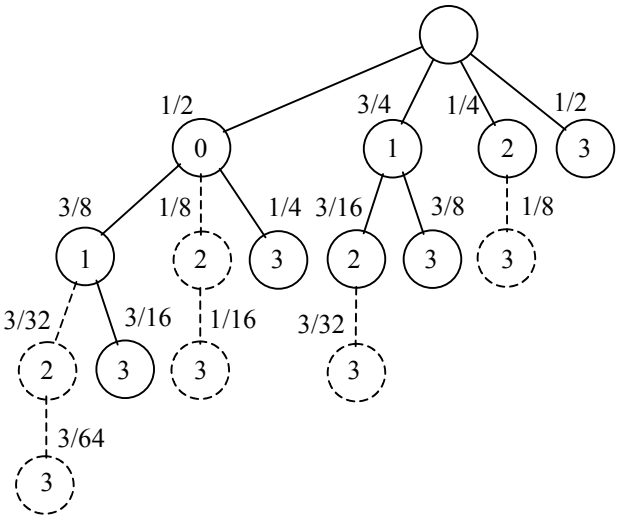


Figure 2. An initial trie for the transaction set {(0, 3), (1, 2), (0, 1, 3), (1)}, with minimum support threshold σ = 1/6. The virtual nodes with probabilities < 1/6 are shown using dashed lines.

shows an example of the initial trie for a given set of transactions (with m = 4). Those nodes of the complete trie (Fig. 1) that do not exist in the actual trie are called *virtual nodes*, and marked with dashed circles in Fig. 2.

The next step is to read the transactions and count the true number of occurrences for each node (i.e. the related path support) in the trie. Simultaneously, for each visited node, we maintain a counter called *pending support* (*PS*), being the number of transactions for which at least one

virtual child of the node would match. The pending support will be our criterion for the *expansion* of the node: If $PS(x) \geq \sigma$, then it is possible that a virtual child of node x is frequent, and the node must be expanded. If there are no such nodes, the algorithm is ready, and the result can be read from the trie: All nodes with support $\geq \sigma$ represent frequent itemsets.

Trie expansion starts the next cycle, and we iterate until the stopping condition holds. However, we must be very careful in the expansion: which virtual nodes should we materialize (and how deep, recursively), in order to avoid trie 'explosion', but yet approach the final solution? Here we apply item probabilities, again. In principle, we could take advantage of all information available in the current trie (frequencies of subsets, etc.), as is done in the Apriori algorithm and many others. However, we prefer simpler calculation, based on global probabilities of items.

Suppose that we have a node x with pending support $PS(x) \geq \sigma$. Assume that it has virtual child items $v_0, v_1, \ldots, v_{s-1}$ with global probabilities $P(v_0), P(v_1), \ldots, P(v_{s-1})$. Every transaction contributing to $PS(x)$ has a match with at least one of $v_0, v_1, \ldots, v_{s-1}$. The *local probability* (*LP*) for a match with v_i is computed as follows:

$$\begin{aligned} &LP(v_i) \\ &= P(v_i \text{ matches} \mid \text{One of } v_0, v_1, \ldots \text{ matches}) \\ &= \frac{P((v_i \text{ matches}) \wedge (\text{One of } v_0, v_1 \ldots \text{ matches}))}{P(\text{One of } v_0, v_1, \ldots \text{ matches})} \\ &= \frac{P(v_i \text{ matches})}{P(\text{One of } v_0, v_1, \ldots \text{ matches})} \\ &= \frac{P(v_i)}{1-(1-P(v_0))(1-P(v_1))\ldots(1-P(v_s))} \end{aligned} \quad (2)$$

Using this formula, we get an *estimated support* $ES(v_i)$:

$$ES(v_i) = LP(v_i) PS(Parent(V_i)) \quad (3)$$

If $ES(v_i) \geq \sigma$, then we conclude that v_i is expected to be frequent. However, in order to guarantee a finite number of iterations in the worst case, we have to relax this condition a bit. Since the true distribution may be very skewed, almost the whole pending support may belong to only one virtual child. To ensure convergence, we apply the following condition for child expansion in the k^{th} iteration,

$$ES(v_i) \geq \alpha^k \sigma \quad (4)$$

with some constant α between 0 and 1. In the worst case this will eventually (when k is high enough) result in expansion, to get rid of a *PS*-value $\geq \sigma$. In our tests, we used the heuristic value α = average probability of frequent items. The reasoning behind this choice is that it speeds up the local expansion growth by one level, on the average (k levels for α^k). This acceleration restricts the number of iterations efficiently. The largest extensions are applied only to the 'skewest' subtries, so that the total size of the trie remains tolerable. Another approach to choose α would be to do a statistical analysis to determine confidence bounds for *ES*. However, this is left for future work.

Fig. 3 shows an example of trie expansion, assuming that the minimum support threshold $\sigma = 80$, $\alpha = 0.8$, and $k = 1$. The item probabilities are assumed to be P(y) = 0.7, P(z) = 0.5, and P(v) = 0.8. Node t has a pending support of 100, related to its two virtual children, y and z. This means that 100 transactions contained the path from root to t, plus either *or both* of items y and z, so we have to test for expansion. Our formula gives y a local probability $LP(y) = 0.7 / (1-(1-0.7)(1-0.5)) \approx 0.82$, so the estimated support is $82 > \alpha \cdot \sigma = 64$, and we expand y. However, the local probability of z is only ≈ 0.59, so its estimated support is 59, and it will not be expanded.

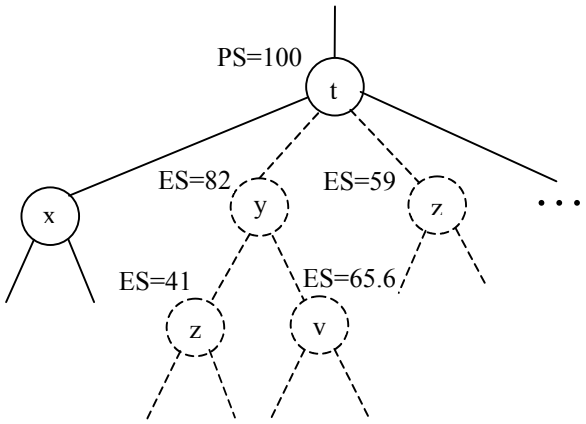


Figure 3. An example of expansion for probabilities P(y) = 0.7, P(z) = 0.5, and P(v) = 0.8.

When a virtual node (y) has been materialized, we immediately test also its expansion, based on its *ES*-value, recursively. However, in the recursive steps we cannot apply formula (2), because we have no evidence of the children of y. Instead, we apply the unconditional probabilities of z and v in estimation: $LP(z) = 82 \cdot 0.5 = 41 < \alpha \cdot \sigma = 64$, and $LP(v) = 82 \cdot 0.8 = 65.6 > 64$. Node v is materialized, but z is not. Expansion test continues down from v. Thus, both in initialization of the trie and in its expansion phases, we can create several new levels (i.e. longer candidates) at a time, contrary to e.g. the base version of Apriori. It is true that also Apriori can be modified to create several candidate levels at a time, but at the cost of increased number of candidates.

After the expansion phase the iteration continues with the counting phase, and new values for node supports and pending supports are determined. The two phases alternate

until all pending supports are less than σ. We have given our method the name 'PIE', reflecting this Probabilistic Iterative Expansion property.

3. Elaboration

The above described basic version does a lot of extra work. One observation is that as soon as the pending support of some node x is smaller than σ, we can often 'freeze' the whole subtrie, because it will not give us anything new; we call it 'ready'. The readiness of nodes can be checked easily with a recursive process: A node x is ready if $PS(x) < \sigma$ and all its real children are ready. The readiness can be utilized to reduce work both in counting and expansion phases. In counting, we process one transaction at a time and scan its item subsets down the trie, but only until the first ready node on each path. Also the expansion procedure is skipped for ready nodes. Finally, a simple stopping condition is when the root becomes ready.

Another tailoring, not yet implemented, relates to the observation that most of the frequent itemsets are found in the first few iterations, and a lot of I/O effort is spent to find the last few frequent sets. For those, not all transactions are needed in solving the frequency. In the counting phase, we can distinguish between relevant and irrelevant transactions. A transaction is irrelevant, if it does not increase the pending support value of any non-ready node. If the number of relevant transactions is small enough, we can store them separately (in main memory or temporary file) during the next scanning phase.

Our implementation of the trie is quite simple; saving memory is considered, but not as the first preference. The child linkage is implemented as an array of pointers, and the frequent items are renumbered to 0, …, m-1 (if there are m frequent items) to be able to use them as indices to the array. A minor improvement is that for item i, we need only m-i-1 pointers, corresponding to the possible children i+1, …, m-1.

The main restriction of the current implementation is the assumption that the trie fits in the main memory. Compression of nodes would help to some extent: Now we reserve a pointer for every possible child node, but most of them are null. Storing only non-null pointers saves memory, but makes the trie scanning slower. Also, we could release the ready nodes as soon as they are detected, in order to make room for expansions. Of course, before releasing, the related frequent itemsets should be reported. However, a fully general solution should work for any main memory and trie size. Some kind of external representation should be developed, but this is left for future work.

A high-level pseudocode of the current implementation is given in the following. The recursive parts are not coded explicitly, but should be rather obvious.

Algorithm PIE – Probabilistic iterative expansion of candidates in frequent itemset mining

---

Input: A transaction database D, the minimum support threshold σ.
Output: The complete set of frequent itemsets.

---

1. // Initial steps.
2. **scan** *D* and collect the set *F* of frequent items;
3. α := average probability of items in *F*;
4. *iter* := 0;

5. // The first generation of candidates, based on
   // item probabilities.
6. **create** a PIE-trie *P* so that it contains all such ordered subsets $S \subseteq F$ for which $\Pi(\text{Prob}(s \in S)) \cdot |D| \geq \sigma$; // Frequency test
7. **set** the status of all nodes of *P* to not-ready;

8. // The main loop: alternating count, test and
   // expand.
9. **loop**
10. // Scan the database and check readiness.
11. **scan** *D* and count the support and pending support values for non-ready nodes in *P*;
12. *iter* := *iter* + 1;
13. **for each** node $p \in P$ **do**
14. **if** *pending_support*(*p*) < σ **then**
15. **if** *p* is a leaf **then** set *p* ready
16. **else if** the children of *p* are ready **then**
17. **set** *p* ready;
18. **if** *root*(*P*) is ready **then exit loop**;

19. // Expansion phase: Creation of subtries on
    // the basis of observed pending supports.
20. **for each** non-ready node *p* **in** *P* **do**
21. **if** *pending_support*(*p*) ≥ σ **then**
22. **for each** virtual child *v* of *p* **do**
23. **compute** *local_prob*(*v*) by formula (2);
24. *estim_support*(*v*) := *local_prob*(*v*) · *pending_support*(*p*);
25. **if** *estim_support*(*v*) $\geq \alpha^{iter}\sigma$ **then**
26. **create** node *v* as the child of *p*;
27. **add** such ordered subsets $S \subseteq F \backslash \{1..v\}$ as descendant paths of *v*, for which $\Pi(\text{Prob}(s \in S)) \cdot \textit{estim\_support}(v) \geq \alpha^{iter}\sigma$;

28. // Gather up results from the trie
29. **return** the paths for nodes *p* in *P* such that *support*(*p*) ≥ σ;
30. **end**

---

4. Experimental results

For verifying the usability of our PIE algorithm, we used four of the test datasets made available to the Workshop on Frequent Itemset Mining Implementations (FIMI'03) [6]. The test datasets and some of their properties are described in Table 1. They represent rather different kinds of domains, and we wanted to include both dense and non-dense datasets, as well as various numbers of items.

Table 1. Test dataset description

Dataset	#Transactions	#Items
Chess	3 196	75
Mushroom	8 124	119
T40I10D100K	100 000	942
Kosarak	900 002	41 270

For the PIE method, the interesting statistics to be collected are the number of candidates, depth of the trie, and the number of iterations. These results are given in Table 2 for selected values of σ, for the 'Chess' dataset. We chose values of σ that keep the number of frequent itemsets reasonable (extremely high numbers are probably useless for any application). The table shows also the number of frequent items and frequent sets, to enable comparison with the number of candidates. For this dense dataset, the number of candidates varies between 2-4 times the number of frequent itemsets. For non-dense datasets the ratio is usually larger. Table 2 shows also the values of the 'security parameter' α, being the average probability of frequent items. Considering I/O performance, we can see that the number of iteration cycles (= number of file scans) is quite small, compared to the base version of the Apriori method, for which the largest frequent itemset dictates the number of iterations. This is roughly the same as the trie depth, as shown in Table 2.

The PIE method can also be characterized by describing the development of the trie during the iterations. The most interesting figures are the number of nodes and the number of ready nodes, given in Table 3. Especially the number of ready nodes implies that even though we have rather many candidates (= nodes in the trie), large parts of them are not touched in the later iterations.

Table 3. Development of the trie for dataset 'Chess', with three different values of σ.

σ	Iteration	#Frequent sets found	#Nodes	#Ready nodes
2600	1	4 720	4 766	2 021
	2	6 036	9 583	9 255
	3	6 134	10 296	10 173
	4	6 135	10 516	10 516
2400	1	15 601	15 760	5 219
	2	20 344	34 995	25 631
	3	20 580	47 203	46 952
	4	20 582	47 515	47 515
2200	1	44 022	44 800	1 210
	2	58 319	112 370	64 174
	3	59 176	206 292	196 782
	4	59 181	216 931	216 922
	5	59 181	216 943	216 943

For speed comparison, we chose the Apriori and FP-growth implementations, provided by Bart Goethals [6]. The results for the four test datasets and for different minimum support thresholds are shown in Table 4. The processor used in the experiments was a 1.5 GHz Pentium 4, with 512 MB main memory. We used a g++ compiler, using optimizing switch –O6. The PIE algorithm was coded in C.

Table 2. Statistics from the PIE algorithm for dataset 'Chess'.

σ	#Frequent items	#Frequent sets	Alpha	#Candidates	Trie depth	#Iterations	#Apriori's iterations
3 000	12	155	0.970	400	6	3	6
2 900	13	473	0.967	1 042	8	4	7
2 800	16	1 350	0.953	2 495	8	4	8
2 700	17	3 134	0.947	5 218	9	4	8
2 600	19	6 135	0.934	10 516	10	4	9
2 500	22	11 493	0.914	18 709	11	4	10
2 400	23	20 582	0.907	47 515	12	4	11
2 300	24	35 266	0.900	131 108	13	4	12
2 200	27	59 181	0.877	216 943	14	5	13

Table 4. Comparison of execution times (in seconds) of three frequent itemset mining programs for four test datasets.

(a) Chess

σ	#Freq. sets	Apriori	FP-growth	PIE
3 000	155	0.312	0.250	0.125
2 900	473	0.469	0.266	0.265
2 800	1 350	0.797	0.297	1.813
2 700	3 134	1.438	0.344	6.938
2 600	6 135	3.016	0.438	14.876
2 500	11 493	10.204	0.610	26.360
2 400	20 582	21.907	0.829	78.325
2 300	35 266	42.048	1.156	203.828
2 200	59 181	73.297	1.766	315.562

(b) Mushroom

σ	#Freq. sets	Apriori	FP-growth	PIE
5 000	41	0.375	0.391	0.062
4 500	97	0.437	0.406	0.094
4 000	167	0.578	0.438	0.141
3 500	369	0.797	0.500	0.297
3 000	931	1.062	0.546	1.157
2 500	2 365	1.781	0.610	6.046
2 000	6 613	3.719	0.750	27.047
1 500	56 693	55.110	1.124	153.187

(c) T40I10D100K

σ	#Freq. sets	Apriori	FP-growth	PIE
20 000	5	2.797	6.328	0.797
18 000	9	2.828	6.578	1.110
16 000	17	3.001	7.250	1.156
14 000	24	3.141	8.484	1.187
12 000	48	3.578	14.750	1.906
10 000	82	4.296	23.874	4.344
8 000	137	7.859	41.203	11.796
6 000	239	20.531	72.985	29.671
4 000	440	35.282	114.953	68.672

(c) Kosarak

σ	#Freq. sets	Apriori	FP-growth	PIE
20 000	121	27.970	30.141	5.203
18 000	141	28.438	31.296	6.110
16 000	167	29.016	32.765	7.969
14 000	202	29.061	33.516	9.688
12 000	267	29.766	34.875	12.032
10 000	376	34.906	37.657	18.016
8 000	575	35.891	41.657	30.453
6 000	1 110	39.656	51.922	70.376

We can see that in some situations the PIE algorithm is the fastest, in some others the slowest. This is probably a general observation: the performance of most frequent itemset mining algorithms is highly dependent on the data set and threshold. It seems that PIE is at its best for sparse datasets (such as T40I10D100K and Kosarak), but not so good for very dense datasets (such as 'Chess' and 'Mushroom'). Its speed for large thresholds probably results from the simplicity of the algorithm. For smaller thresholds, the trie gets large and the counting starts to consume more time, especially with a small main memory size.

One might guess that our method is at its best for *random* data sets, because those would correspond to our assumption about independent item occurrences. We tested this with a dataset of 100 000 transactions, each of which contained 20 random items out of 30 possible. The results were rather interesting: For all tested thresholds for minimum support, we found *all* the frequent itemsets in the first iteration. However, verification of the completeness required one or two additional iterations, with a clearly higher number of candidates, consuming a majority of the total time. Table 5 shows the time and number of candidates both after the first and after the final iteration. The stepwise growth of the values reveals the levelwise growth of the trie. Apriori worked well also for this dataset, being in most cases faster than PIE. Results for FP-growth (not shown) are naturally much slower, because randomness prevents a compact representation of the transactions.

We wish to point out that our implementation was an initial version, with no special tricks for speed-up. We are convinced that the code details can be improved to make the method still more competitive. For example, buffering of transactions (or temporary files) were not used to enhance the I/O performance.

5. Conclusions and future work

A probability-based approach was suggested for frequent itemset mining, as an alternative to the 'analytic' methods common today. It has been observed to be rather robust, working reasonably well for various kinds of datasets. The number of candidate itemsets does not 'explode', so that the data structure (trie) can be kept in the main memory in most practical cases.

The number of iterations is smallest for random datasets, because candidate generation is based on just that assumption. For skewed datasets, the number of iterations may somewhat grow. This is partly due to our simplifying premise that the items are independent. This point could be tackled by making use of the conditional probabilities obtainable from the trie. Initial tests did not show any significant advantage over the basic approach, but a more

Table 5. Statistics from the PIE algorithm for a random dataset.

σ	#Freq. sets	PIE						Apriori	
		After iteration 1.			After the last iteration (final)				
		#Freq. sets	Time (sec.)	#Cand.	#Cand.	#Iter-ations	Time (sec.)	#Iter-ations	Time (sec.)
50 000	30	30	0.500	30	464	2	2.234	2	3.953
44 000	42	42	2.016	465	509	3	2.704	3	5.173
43 800	124	124	1.875	465	1 247	3	10.579	3	6.015
43 700	214	214	1.876	465	1 792	3	20.250	3	7.235
43 600	331	331	1.891	465	2 775	3	37.375	3	9.657
43 500	413	413	1.860	465	3 530	3	48.953	3	11.876
40 000	465	465	1.844	465	4 443	2	62.000	3	13.875
28 400	522	522	60.265	4 525	4 900	3	64.235	4	15.016
28 300	724	724	61.422	4 525	5 989	3	82.140	4	15.531
28 200	1 270	1 270	61.469	4 525	8 697	3	115.250	4	19.265
28 100	2 223	2 223	61.734	4 525	13 608	3	167.047	4	31.266
28 000	3 357	3 357	60.969	4 525	19 909	3	219.578	4	69.797

sophisticated probabilistic analysis might imply some ways to restrict the number of candidates. The exploration of these elaborations, as well as tuning the buffering, data structure, and parameters, is left for future work.

References

[1] R. Agrawal, C. Aggarwal, and V.V.V. Prasad, "Depth First Generation of Long Patterns", In R. Ramakrishnan, S. Stolfo, R. Bayardo, and I. Parsa (eds.), *Proc. of the Int. Conf. on Knowledge Discovery and Data Mining*, ACM, Aug. 2000, pp. 108-118.

[2] R. Agrawal, T. Imielinski, and A. Swami, "Mining Association Rules Between Sets of Items in Large Databases", In P. Buneman and S. Jajodia (eds.), *Proc. of ACM SIGMOD Int. Conf. of Management of Data*, May 1993, pp. 207-216.

[3] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases", In J.B. Bocca, M. Jarke, and C. Zaniolo (eds.), *Proc. of the 20th VLDB Conf.*, Sept. 1994, pp. 487-499.

[4] R.J. Bayardo, "Efficiently Mining Long Patterns from Databases", In L.M. Haas and A. Tiwary (eds.), *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, June 1998, pp. 85-93.

[5] D. Burdick, M. Calimlim, and J. Gehrke, "MAFIA: a Maximal Frequent Itemset Algorithm for Transactional Databases", *Proc. of IEEE Int. Conf. on Data Engineering*, April 2001, pp. 443-552.

[6] Frequent Itemset Mining Implementations (FIMI'03) Workshop website, *http://fimi.cs.helsinki.fi*, 2003.

[7] B. Goethals, "Efficient Frequent Pattern Mining", *PhD thesis*, University of Limburg, Belgium, Dec. 2002.

[8] K. Gouda and M.J. Zaki, "Efficiently Mining Maximal Frequent Itemsets", In N. Cercone, T.Y. Lin, and X. Wu (eds.), *Proc. of 2001 IEEE International Conference on Data Mining*, Nov. 2001, pp. 163-170.

[9] J. Han, J. Pei, and Y. Yin, "Mining Frequent Patterns Without Candidate Generation", In W. Chen, J. Naughton, and P.A. Bernstein (eds.), *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, 2000, pp. 1-12.

[10] J. Hipp, U. Güntzer, and N. Nakhaeizadeh, "Algorithms for Association Rule Mining - a General Survey and Comparison", *ACM SIGKDD Explorations* 2, July 2000, pp. 58-65.

[11] H. Mannila, H. Toivonen, and A.I. Verkamo, "Efficient Algorithms for Discovering Association Rules", In U.M. Fayyad and R. Uthurusamy (eds.), *Proc. of the AAAI Workshop on Knowledge Discovery in Databases*, July 1994, pp. 181-192.

[12] J. Pei, J. Han, and R. Mao, "Closet: An Efficient Algorithm for Mining Frequent Closed Itemsets", *Proc. of ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery, May 2000*, pp. 21-30.

[13] J. Vuillemin, "A Data Structure for Manipulating Priority Queues", *Comm. of the ACM*, 21(4), 1978, pp. 309-314.

[14] M.J. Zaki, "Scalable Algorithms for Association Mining", *IEEE Transactions on Knowledge and Data Engineering* 12 (3), 2000, pp. 372-390.

[15] Z. Zheng, R. Kohavi, and L. Mason, "Real World Performance of Association Rule Algorithms", In F. Provost and R. Srikant (eds.), *Proc. of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2001, pp. 401-406.

Intersecting Data to Closed Sets with Constraints

Taneli Mielikäinen
HIIT Basic Research Unit
Department of Computer Science
University of Helsinki, Finland
Taneli.Mielikainen@cs.Helsinki.FI

Abstract

We describe a method for computing closed sets with data-dependent constraints. Especially, we show how the method can be adapted to find frequent closed sets in a given data set. The current preliminary implementation of the method is quite inefficient but more powerful pruning techniques could be used. Also, the method can be easily applied to wide variety of constraints. Regardless of the potential practical usefulness of the method, we hope that the sketched approach can shed some additional light to frequent closed set mining.

1 Introduction

Much of the research in data mining has concentrated on finding from some given (finite) set R all subsets that satisfy some condition. (For the rest of the paper we assume, w.l.o.g., that R is a finite subset of $\mathbb{N}$.)

The most prominent example of this task is probably the task of finding all subsets $X \subseteq R$ that are contained at least $minsupp$ times in the sets of a given sequence $d = d_1 \dots d_n$ of subsets $d_i \subseteq R$, i.e., to find the collection

$$\mathcal{F}(minsupp, d) = \{X \subseteq R : supp(X, d) \geq minsupp\}$$

where

$$supp(X, d) = |\{i : X \subseteq d_i, 1 \leq i \leq n\}|.$$

The collection $\mathcal{F}(minsupp, d)$ is known as the collection of frequent sets. (We could have defined the collection of frequent sets by by the frequency of sets which is a normalized version of supports: $fr(X, d) = supp(X, d)/n$.)

Recently one particular subclass of frequent sets, frequent closed sets, has received quite much attention. A set X is closed in d if $supp(X, d) > supp(Y, d)$ for all proper supersets Y of X. The collection of closed sets (in d) is denoted by

$$\begin{aligned}\mathcal{C}(d) &= \{X \subseteq R : Y \subseteq R, Y \supset X \\ &\qquad \Rightarrow supp(X, d) > supp(Y.d)\}\end{aligned}$$

The collection of frequent closed sets consists of the sets that are frequent and closed, i.e.,

$$\mathcal{FC}(minsupp, d) = \mathcal{F}(minsupp, d) \cap \mathcal{C}(d).$$

Most of the closed set mining algorithms [3, 12, 13, 14, 16, 19, 20] are based on backtracking [10]. In this paper we describe an alternative approach based on alternating between closed set generation by intersections and pruning heuristics. The method can be adapted to many kinds of constraints and needs only few passes over the data.

The paper is organized as follows. In Section 2 we sketch the method, in Section 3 we adapt the method for finding closed sets with frequency constraints, in Section 4 we describe some implementations details the method, and in Section 5 we experimentally study the properties of the method. Section 6 concludes the work and suggests some improvements to the work.

2 The Method

Let us assume that $R = \bigcup_{i \in \{1,\dots,n\}} d_i$ as sometimes R is not known explicitly. Furthermore, we shall use shorthand $d_{i,j}$ for the subsequence $d_i \dots d_j$, $1 \leq i \leq j \leq n$. The elements of R are sometimes called *items* and the sets d_i *transactions*.

As noted in the previous section, a set $X \subseteq R$ is closed in d if and only if $supp(X, d) > supp(Y, d)$ for all proper supersets Y of X. However, the closed sets can be defined also as intersection of the transactions (see e.g. [11]):

Definition 1 *A set $X \subseteq R$ is closed in d if and only if there is $I \subseteq \{1, \dots, n\}$ such that $X = \bigcap_{i \in I} d_i$. (By convention, $\bigcap_{i \in \emptyset} d_i = R$.)*

A straightforward implementation of Definition 1

$$\mathcal{C}(d) = \left\{ \bigcap_{i \in I} d_i : I \subseteq \{1, \ldots, n\} \right\}$$

leads to quite inefficient method for computing all closed sets:[1]

BRUTE-FORCE(d)
1 $R \leftarrow \bigcup_{i \in \{1,\ldots,n\}} d_i$
2 $supp(R) \leftarrow 0$
3 **for each** $I \subseteq \{1, \ldots, n\}, I \neq \emptyset$
4 **do** $X \leftarrow \bigcap_{i \in I} d_i$
5 **if** $supp(X) < |I|$
6 **then** $supp(X) \leftarrow |I|$
7 **return** $(supp : \mathcal{C} \rightarrow \mathbb{N})$

A more efficient solution can be found by the following recursive definition of closed sets:

$$\begin{aligned} \mathcal{C}(d_1) &= \{R, d_1\} \\ \mathcal{C}(d_{1,i+1}) &= \mathcal{C}(d_{1,i}) \cup \{X \cap d_{i+1} : X \in \mathcal{C}(d_{1,i})\} \end{aligned}$$

Thus the closed sets can be computed by initializing $\mathcal{C} = \left\{ R = \bigcup_{i \in \{1,\ldots,n\}} d_i \right\}$ (since R is always closed), initializing $supp$ to $R \mapsto 0$, and calling the following algorithm for each d_i ($1 \leq i \leq n$):

INTERSECT($supp : \mathcal{C} \rightarrow \mathbb{N}, d_i$)
1 **for each** $X \in \mathcal{C}$
2 **do** $\mathcal{C} \leftarrow \mathcal{C} \cup \{X \cap d_i\}$
3 **if** $supp(X \cap d_i) < supp(X) + 1$
4 **then** $supp(X \cap d_i) \leftarrow supp(X) + 1$
5 **return** $(supp : \mathcal{C} \rightarrow \mathbb{N})$

Using the above algorithm the sequence d does not have to be stored as each d_i is needed just for updating the current approximation of R and intersecting the current collection $\mathcal{C}$ of closed sets.

The closed sets can be very useful way to understand data sets that consist of only few different transactions and they have been studied in the field of Formal Concept Analysis [6]. However, many times all closed sets are not of interest but only frequent closed sets are needed. The simplest way to adapt the approach described above for finding the frequent closed sets is to first compute all closed sets $\mathcal{C}(d)$ and then remove the infrequent ones:

$$\mathcal{FC}(minsupp, d) = \{X \in \mathcal{C}(d) : supp(X, d) \geq minsupp\}$$

by removing all closed sets that are not frequent.

Unfortunately the collection of closed sets can be much larger than the collection of frequent closed sets. Thus the above approach can generate huge number of closed sets that do not have to be generated.

A better approach to find the frequent closed sets is to prune the closed sets that cannot satisfy the constraints – such as the minimum support constraint – as soon as possible. If the sequence is scanned only once and nothing is known about the sequence d in advance then no pruning of infrequent closed sets can be done: the rest of the sequence can always contain each closed set at least $minsupp$ times.

If more than one pass can be afforded or something is known about the data d in advance then the pruning of closed sets that do not satisfy the constraints can be done as follows:

INTERSECTOR(d)
1 $supp \leftarrow$ INIT-CONSTRAINTS (d)
2 **for each** d_i **in** d
3 **do** $supp \leftarrow$ INTERSECT $(supp, d_i)$
4 UPDATE-CONSTRAINTS $(supp, d_i)$
5 $supp \leftarrow$ PRUNE-BY-CONSTRAINTS $(supp, d_i)$
6 **return** $(supp : \mathcal{C} \rightarrow \mathbb{N})$

The function INTERSECTOR is based on three subroutines: function INIT-CONSTRAINTS initializes the data structures used in pruning and computes the initial collection of closed sets, e.g. the the collection $\mathcal{C} = \{R\}$, function UPDATE-CONSTRAINTS updates the data structures by one transaction at a time, and function PRUNE-BY-CONSTRAINTS prunes those current closed sets that cannot satisfy the constraints.

3 Adaptation to Frequency Constraints

The actual behaviors of the functions INIT-CONSTRAINTS, UPDATE-CONSTRAINTS and PRUNE-BY-CONSTRAINTS depend on the constraints used to determine the closed sets that are interesting. We shall concentrate on implementing the minimum and the maximum support constraints, i.e., finding the closed sets $X \in \mathcal{C}(d)$ such that $minsupp \leq supp(X, d) \leq maxsupp$.

The efficiency of pruning depends crucially on how much is known about the data. For example, if only the number of transactions in the sequence is known, then all possible pruning is essentially determined by Observation 1 and Observation 2.

Observation 1 *For all* $1 \leq i \leq n$ *holds:*

$$supp(X, d_{1,i}) + n - i < minsupp \Rightarrow supp(X, d) < minsupp$$

Observation 2 *For all* $1 \leq i \leq n$ *holds:*

$$supp(X, d_{1,i}) > maxsupp \Rightarrow supp(X, d) > maxsupp$$

[1] If $supp(X)$ is not defined then its value is interpreted to be 0.

Checking the constraints induced by Observation 1 and Observation 2 can be computed very efficiently. However, the pruning based on these observations might not be very effective: all closed sets in $d_{1,n-minsupp}$ can have frequency at least $minsupp$ and all closed sets in $d_{1,maxsupp}$ can have frequency at most $maxsupp$. Thus all closed sets in $d_{1,\min\{n-minsupp,maxsupp\}}$ are generated before the observations can be used to prune anything.

To be able to do more extreme pruning we need more information about the sequence d. If we are able to know the number of transactions in the sequence, it might be possible to count the supports of items. In that case Observation 3 can be exploited.

Observation 3 *If there exists $A \in X$ such that $supp(X, d_{1,i}) + supp(\{A\}, d_{i+1,n}) < minsupp$ then $supp(X, d) < minsupp$.*

Also, if we know the frequencies of some sets then we can make the following observation:

Observation 4 *If there exists $Y \subseteq X$ such that $supp(X, d_{1,i}) + supp(Y, d_{i+1,n}) < minsupp$ then $supp(X, d) < minsupp$.*

Note that these observations do not mean that we could remove the infrequent closed sets from the collection since an intersection of an infrequent closed set with some transaction might still be frequent in the sequence d. However, we can do some pruning based on the observations as shown in Proposition 1.

Proposition 1 *Let Z be the largest subset of $X \in \mathcal{C}$ such that for all $A \in Z$ hold $supp(X, d_{1,i}) + supp(\{A\}, d_{i+1,n}) \geq minsupp$. Then $X \in \mathcal{C}$ can be removed from $\mathcal{C}$ if there is a $W \subseteq Y \in \mathcal{C}, Z \subset W$, such that $supp(Y, d_{1,i}) \geq supp(X, d_{1,i})$ and for all $A \in W$ hold $supp(Y, d_{1,i}) + supp(\{A\}, d_{i+1,n}) \geq minsupp$, and replaced by Z otherwise.*

Proof. All frequent subsets of X are contained in Z. If there is a proper superset $W \subseteq Y \in \mathcal{C}$ of Z such that $supp(Y, d_{1,i}) + supp(\{A\}, d_{i+1,n}) \geq minsupp$ then all frequent subsets of X are contained in W and thus X can be removed. Otherwise Z is the largest subset of X that can be frequent and there is no superset of Z that could be frequent. If Z is not closed, then its support is equal to some of its proper supersets' supports. If Z is added to $\mathcal{C}$ then none of proper supersets is frequent and thus also Z is infrequent. □

This idea of replacing infrequent sets based on the supports items can be generalized to the case where we know supports for some collection $\mathcal{S}$ of sets.

Proposition 2 *Let $\mathcal{S}$ be the collection of sets such that $supp(Y, d_{1,i})$ and $supp(Y, d_{1,i})$ are known for all $Y \in \mathcal{S}$, and let $\mathcal{S}'$ consist of sets $Y \in \mathcal{S}, Y \subseteq X$, such that $supp(X, d_{1,i}) + supp(Y, d_{i+1,n}) < minsupp$. Then all frequent subsets of $X \in \mathcal{C}$ are the collection $\mathcal{S}''$ of subsets $Z \subseteq X$ such that $Z \not\subseteq Y$ for all $Y \in \mathcal{S}'$, no $W \subset Z \in \mathcal{S}''$ is contained in $\mathcal{S}''$.*

Proof. If $Z \subseteq X$ is frequent then there is a set in $\mathcal{S}''$ containing Z, or Z is contained in some set in $\mathcal{S}'$ but there is another set $Y \in \mathcal{C}$ such that $supp(Y, d_{1,i}) > supp(X, d_{1,i})$. □

Proposition 3 *Let $\mathcal{S}$, $\mathcal{S}'$ and $\mathcal{S}''$ be as in Proposition 2. Then $X \in \mathcal{C}$ can be replaced by the collection $\mathcal{S}'''$ consisting of sets in $\mathcal{S}''$ such that $supp(Y, d_{1,i}) + supp(W, d_{i+1,n}) < minsupp$ for some $W \subseteq Z \subseteq Y$ with $Y \in \mathcal{C}$ and $W \in \mathcal{S}$.*

Proof. If $Z \subseteq X$ is frequent then it is subset of some set in $\mathcal{S}''$ or there is $Y \in \mathcal{C}, Z \subseteq Y$, such that $supp(Y, d_{1,i}) + supp(W, d_{i+1,n}) < minsupp$ for all $W \in \mathcal{S}, W \subseteq Z$.

If $Z \in \mathcal{S}''$ is not closed then it is infrequent since none of its supersets is frequent. □

The efficiency of pruning depends crucially also on the ordering of the transactions. In Section 5 we experimentally evaluate some orderings with different data sets.

4 The Organization of the Implementation

A preliminary adaptation of the algorithm INTERSECTOR of Section 2 to minimum support and maximum support constraints is implemented as a program `intersector`. The main components of the implementation are classes `Itemarray`, `ItemarrayInput` and `ItemarrayMap`.

The class `Itemarray` is a straightforward implementation consisting of `int n` expressing the number of items in the set and `int* items` that is a length (at least) `n` array of items (that are assumed to be nonnegative integers) in ascending order. One of the reasons why this very simple representation of a set is used is that `Itemarray`s are used also in the data sources, and although some more sophisticated data structures would enable to do some operations more efficiently, we believe that `Itemarray` reflects better what an arbitrary source of transactions could give.

The class `ItemarrayInput` implements an interface to the data set d. The class handles the pruning of infrequent items from the input and maintaining the numbers of remaining occurrences of each item occurring in the data set. The data set d is accessed by a function `pair<Itemarray*,int>*`

`getItemarray()` which returns a pointer to next `pair<Itemarray*,int>`. The returned pointer is `NULL` if the previous pair were the last one in the data set d. The main difference to the reference implementation given at the home page of Workshop on Frequent Itemset Mining Implementations[2] is that `pair<Itemarray*,int>*` is returned instead of `Itemarray*`. This change were made partly to reflect the attempt to have the closure property of inductive databases [9] but also because in some cases the data set is readily available in that format (or can be easily transformed into that format). The interface `ItemarrayInput` is currently implemented in two classes `ItemarrayFileInput` and `ItemarrayMemoryInput`. Both of the classes read the data set d from a file consisting of rows of integers with possible count in brackets. Multiple occurrences of same item in one row are taken into account only once. For example, the input file

```
1 2 4 3 2 5 (54)
1 1 1 1
```

is transformed into pairs $\langle (1,2,3,4,5), 54\rangle$ and $\langle (1), 1\rangle$.

The class `ItemarrayFileInput` maintains in the main memory only the item statistics (such as the number of remaining occurrences of each item) thus possibly reading the data set several times. The class `ItemarrayMemoryInput` reads the whole data set d into main memory. The latter one can be much faster since it can also reorder the data set and replace all transactions $d_i, 1 \leq i \leq n$, with same frequent items by one pair with appropriate count. The implementations of these classes are currently quite slow which might be seen as imitating the performance of real databases quite faithfully.

The class `ItemarrayMap` represents a mapping from `Itemarray`s to supports. The class consists of a mapping `map<Itemarray*,int,CardLex> itemarrays` that maps the sets to supports, and a set `set<Itemarray*,CardLex> forbidden` consisting of the sets that are known to be infrequent or too frequent. The set `set<Itemarray*,CardLex> forbidden` is needed mainly because of the maximum frequency constraint. The class consists two methods:

- The method `intersect(const pair<Itemarray*,int>*)` intersects the current collection sets represented by the mapping `map<Itemarray*,int,CardLex> itemarrays` by the given set `pair<Itemarray*,int>*` and updates the supports appropriately.
- The method `prune(ItemarrayInput&)` prunes the sets that are already known to be infrequent or too frequent based on the statistics maintained by the implementation of the interface `ItemarrayInput`. (The class `CardLex` defines a total ordering of sets (of integers) by their cardinality and lexicographically within of each group with same sizes.) The pruning rules used in the current implementation of the method `prune(ItemarrayInput&)` are Observation 1, Observation 2, and Observation 3.

[2] `http://fimi.cs.helsinki.fi/`

5 The Experiments

name	# of rows	total # of items
`T10I4D100K`	100000	1010228
`T40I10D100K`	100000	3960507
`chess`	3196	118252
`connect`	67557	2904951
`internet`	10104	300985
`kosarak`	990002	8019015
`mushroom`	8124	186852
`pumsb`	49046	3629404
`pumsb*`	49046	2475947

Table 1. The data sets

We tested the efficiency and behavior of the implementation by the data sets listed in Table 1. All data sets except `internet` were provided by the Workshop on Frequent Itemset Mining Implementations. The data set `internet` is the Internet Usage data from UCI KDD Repository[3].

If the data sequence is read to main memory then it can be easily reordered. Also, even if this is not the case, there exist efficient external memory sorting algorithms that can be used to reorder the data [18]. The ordering of the data can affect the performance significantly.

We experimented especially with two orderings: ordering in ascending cardinality and ordering in descending cardinality. The results are shown in Figures 1–9. Each point $(|\mathcal{C}|, i)$ in the figures corresponds to the number $|\mathcal{C}|$ of closed sets in the sequence $d_{1,i}$ that could be frequent in the whole sequence d. Note that the reason why there is no point for each number i ($1 \leq i \leq n$) of seen transactions is that same set of items can occur several times in the sequence d.

There is no clear winner within the ascending and descending orderings: with data sets `T10I4D100K`, `T40I10D100K`, `internet`, `kosarak`, and `mushroom` the ascending order is better whereas the descending order seems to be better with data sets `chess`, `connect`, and `pumsb`. However, it is not clear whether this is due to the chosen minimum support thresholds.

[3] `http://kdd.ics.uci.edu/`

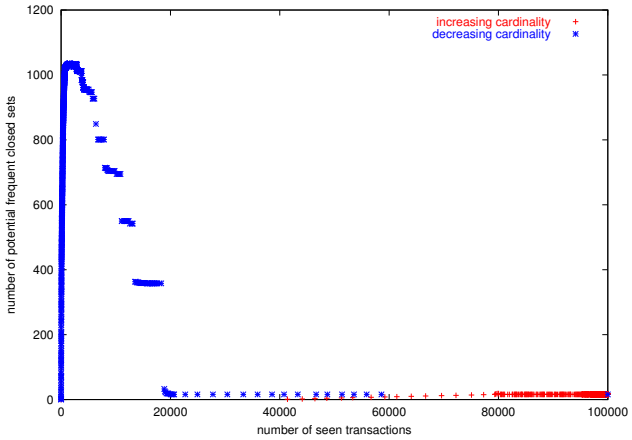


Figure 1. `T10I4D100K`, $minsupp = 4600$

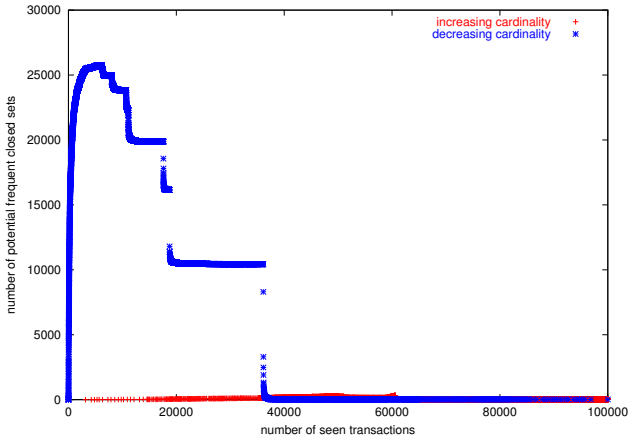


Figure 2. `T40I10D100K`, $minsupp = 16000$

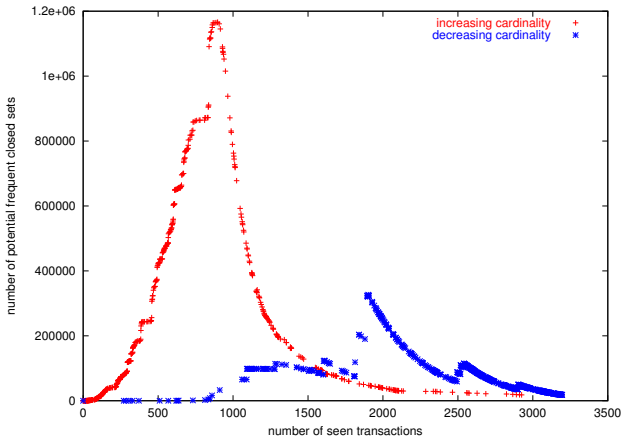


Figure 3. `chess`, $minsupp = 2300$

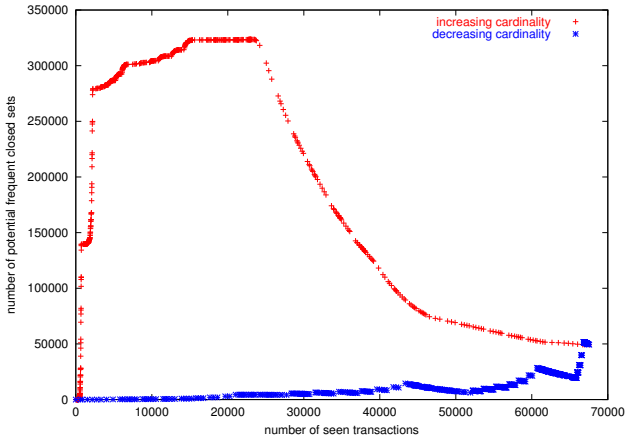


Figure 4. `connect`, $minsupp = 44000$

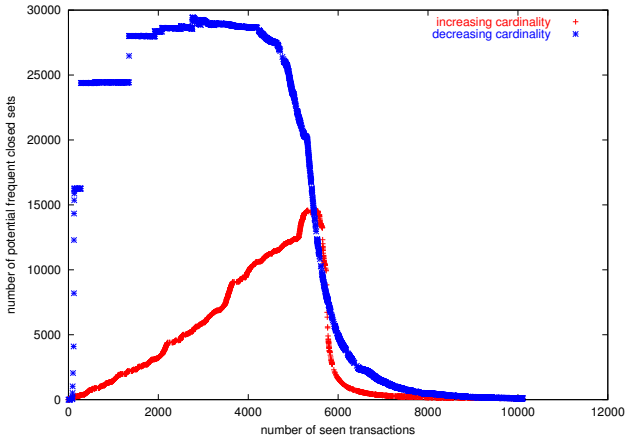


Figure 5. `internet`, $minsupp = 4200$

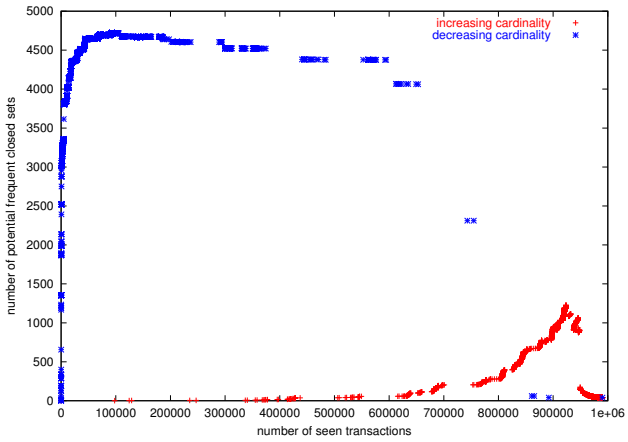


Figure 6. `kosarak`, $minsupp = 42000$

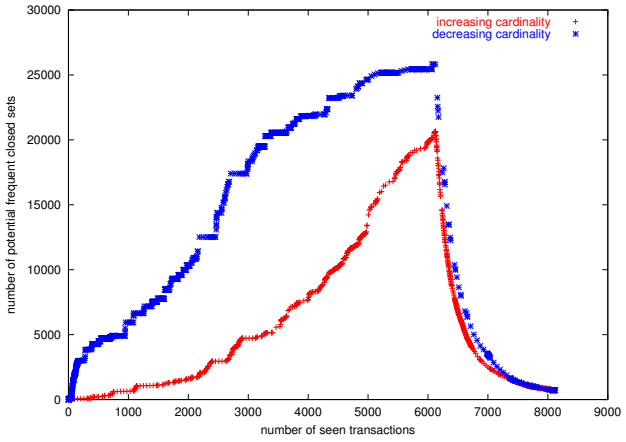


Figure 7. `mushroom`, $minsupp = 2000$

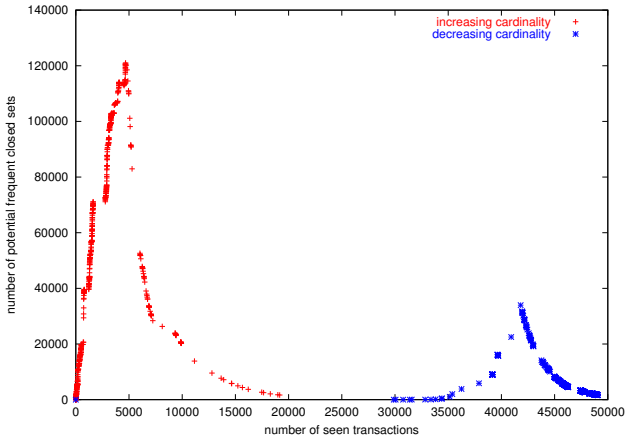


Figure 8. `pumsb`, $minsupp = 44000$

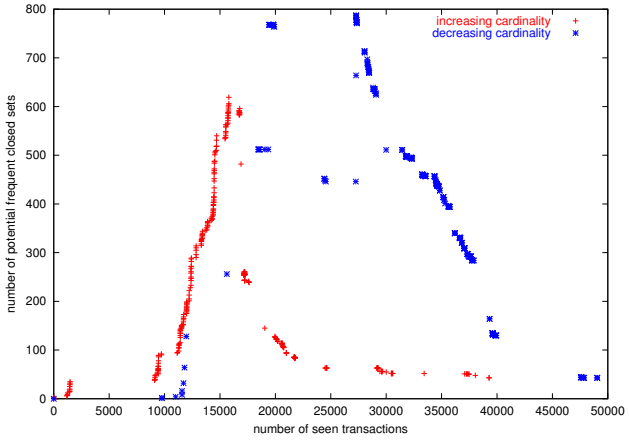


Figure 9. `pumsb*`, $minsupp = 32000$

One interpretation of the results is the following: small set d_i cannot increase the size of $\mathcal{C}$ dramatically since all new closed sets are subsets of d_i and d_i has at most $2^{|d_i|}$ closed subsets. However, the small sets do not decrease the remaining number of occurrences of items very much either. In the case of large sets d_j the situation is the opposite: each large set d_j decreases the supports $supp(\{A\}, d_{j+1,n})$ of each item $A \in d_j$ but on the other hand it can generate several new closed sets.

Also, we experimented with two data sets `internet` and `mushroom` to see how the behavior of the method changes when changing the minimum support threshold $minsupp$. The results are shown in Figure 10 and Figure 11.

The pruning seems to work satisfactory if the minimum support threshold $minsupp$ is high enough. However, it is not clear how much this is due to the pruning of infrequent items in the class `ItemarrayInput` and how much due to the pruning done by the class `ItemarrayMap`. Unfortunately, the performance rapidly collapses as the minimum support threshold decreases. It is possible that more aggressive pruning could help when the minimum support threshold $minsupp$ is low.

6 Conclusions and Future Work

In this paper we have sketched an approach for finding closed sets with some constraints from data with only few passes over the data. Also, we described a preliminary implementation of the method for finding frequent but not too frequent closed sets from data. The current version of the implementation is still quite inefficient but it can hopefully shed some light to the interplay of data and closed sets.

As the current implementation of the approach is still very preliminary, there is plenty of room for improvements, e.g., the following ones:

- The ordering of input seems to play crucial role in the efficiency of the method. Thus the favorable orderings should be detected and strategies for automatically finding them should be studied.

- The pruning heuristics described in this paper are still quite simplistic. Thus, more sophisticated pruning techniques such as inclusion-exclusion [5] should be tested. Also, pruning co-operation between closed sets generation and the data source management should be tighten.

- The pruning done by the data source management could be improved. For example, the data source management could recognize consecutive redundancy in the the data source.

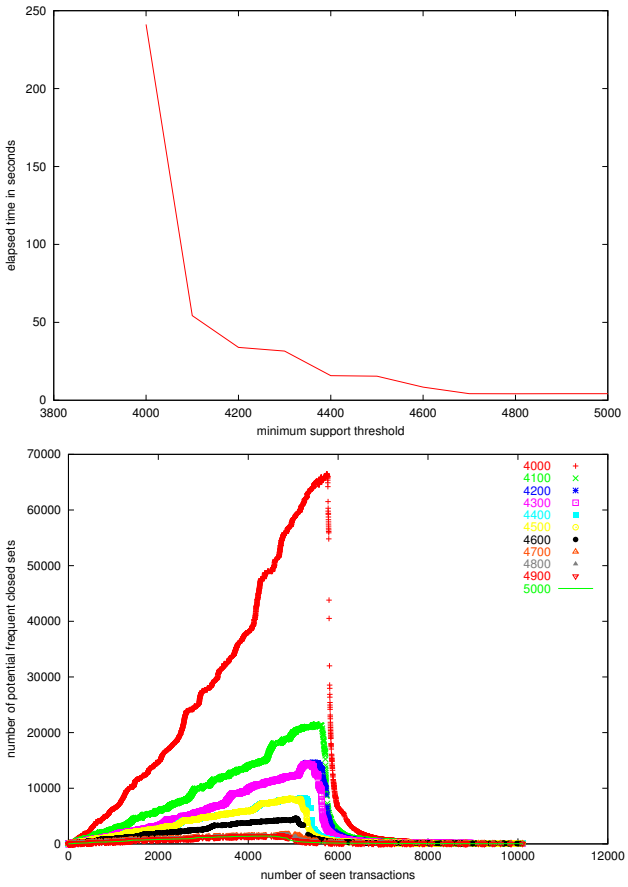


Figure 10. `internet`**, scalability**

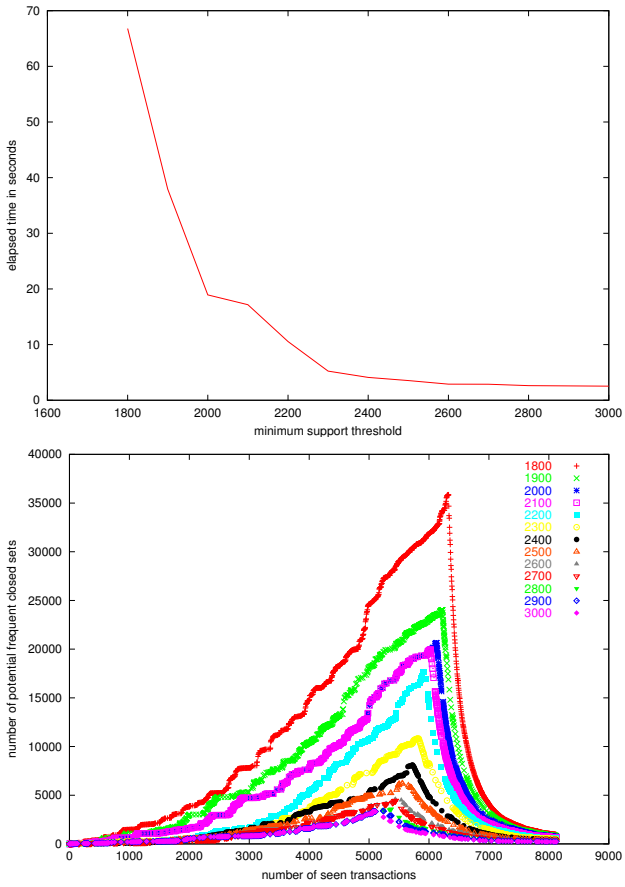


Figure 11. `mushroom`**, scalability**

- The intersection approach can be used to find all closed sets that are subsets of some given sets [11]. The method can be used to compute closed sets from the maximal sets in one pass over the data. As there exist very efficient methods for computing maximal sets [1, 2, 4, 7, 8, 15], it is possible that the performance of the combination could be quite competitive. Also, supersets of maximal frequent sets can be found with high probability from a small sample. Using these estimates one could compute supersets of frequent closed sets. This approach can be efficient if the supersets found from the sample are close enough to the actual maximal sets.

- After two passes over the data it is easy to do the third pass, or even more. Thus one could apply the intersections with several different minimum support thresholds to get refining collection of frequent closed sets in the data: the already found frequent closed sets with high frequencies could be used to prune less frequent closed sets more efficiently than e.g. the occurrence counters for frequent items.

- If it is not necessary to find the exact collection of closed sets with exact supports, then a sampling could be applied [17]. Also, if the data is generated by e.g. an i.i.d. source then one can sometimes obtain accurate bounds for the supports from relatively short prefixes $d_{1,i}$ of the sequence d.

- Other kinds of constraints than frequency thresholds should be implemented and experimented with.

References

[1] R. J. Bayardo Jr. Efficiently mining long patterns from databases. In A. T. Laura M. Haas, editor, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 85–93. ACM, 1998.

[2] E. Boros, V. Gurvich, L. Khachiyan, and K. Makino. On the complexity of generating maximal frequent and minimal infrequent sets. In H. Alt and A. Ferreira, editors, *STACS 2002*, volume 2285 of *Lecture Notes in Computer Science*, pages 133–141. Springer-Verlag, 2002.

[3] J.-F. Boulicaut and A. Bykowski. Frequent closures as a concise representation for binary data mining. In T. Terano,

H. Liu, and A. L. P. Chen, editors, *Knowledge Discovery and Data Mining*, volume 1805 of *Lecture Notes in Artificial Intelligence*, pages 62–73. Springer-Verlag, 2000.

[4] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of the 17th International Conference of Data Engineering (ICDE'01)*, pages 443–452, 2001.

[5] T. Calders and B. Goethals. Mining all non-derivable frequent itemsets. In T. Elomaa, H. Mannila, and H. Toivonen, editors, *Principles of Data Mining and Knowledge Discovery*, volume 2431 of *Lecture Notes in Artificial Intelligence*, pages 74–865. Springer-Verlag, 2002.

[6] B. Ganter and R. Wille. *Formal Concept Analysis: Mathematical Foundations*. Springer-Verlag, 1999.

[7] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In N. Cercone, T. Y. Lin, and X. Wu, editors, *Proceedings of the 2001 IEEE International Conference on Data Mining*, pages 163–170. IEEE Computer Society, 2001.

[8] D. Gunopulos, R. Khardon, H. Mannila, S. Saluja, H. Toivonen, and R. S. Sharma. Discovering all most specific sentences. *ACM Transactions on Database Systems*, 28(2):140–174, 2003.

[9] T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of The ACM*, 39(11):58–64, 1996.

[10] D. L. Kreher and D. R. Stinson. *Combinatorial Algorithms: Generation, Enumeration and Search*. CRC Press, 1999.

[11] T. Mielikäinen. Finding all occurring sets of interest. In J.-F. Boulicaut and S. Džeroski, editors, *2nd International Workshop on Knowledge Discovery in Inductive Databases*, pages 97–106, 2003.

[12] F. Pan, G. Cong, A. K. H. Tung, J. Yang, and M. J. Zaki. CARPENTER: Finding closed patterns in long biological datasets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2003.

[13] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In C. Beeri and P. Buneman, editors, *Database Theory - ICDT'99*, volume 1540 of *Lecture Notes in Computer Science*, pages 398–416. Springer-Verlag, 1999.

[14] J. Pei, J. Han, and T. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In D. Gunopulos and R. Rastogi, editors, *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[15] K. Satoh and T. Uno. Enumerating maximal frequent sets using irredundant dualization. In G. Grieser, Y. Tanaka, and A. Yamamoto, editors, *Discovery Science*, volume 2843 of *Lecture Notes in Artificial Intelligence*, pages 256–268. Springer-Verlag, 2003.

[16] G. Stumme, R. Taouil, Y. Bastide, N. Pasquier, and L. Lakhal. Computing iceberg concept lattices with TITANIC. *Data & Knowledge Engineering*, 42:189–222, 2002.

[17] H. Toivonen. Sampling large databases for association rules. In T. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *VLDB'96, Proceedings of 22nd International Conference on Very Large Data Bases*, pages 134–145. Morgan Kaufmann, 1996.

[18] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[19] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[20] M. J. Zaki and C.-J. Hsiao. CHARM: An efficient algorithms for closed itemset mining. In R. Grossman, J. Han, V. Kumar, H. Mannila, and R. Motwani, editors, *Proceedings of the Second SIAM International Conference on Data Mining*. SIAM, 2002.

ARMOR: Association Rule Mining based on ORacle

Vikram Pudi
Intl. Inst. of Information Technology
Hyderabad, India
vikram@iiit.net

Jayant R. Haritsa
Database Systems Lab, SERC
Indian Institute of Science
Bangalore, India
haritsa@dsl.serc.iisc.ernet.in

Abstract

In this paper, we first focus our attention on the question of how much space remains for performance improvement over current association rule mining algorithms. Our strategy is to compare their performance against an "Oracle algorithm" that knows in advance the identities of all frequent itemsets in the database and only needs to gather their actual supports to complete the mining process. Our experimental results show that current mining algorithms do not perform uniformly well with respect to the Oracle for all database characteristics and support thresholds. In many cases there is a substantial gap between the Oracle's performance and that of the current mining algorithms. Second, we present a new mining algorithm, called ARMOR, that is constructed by making minimal changes to the Oracle algorithm. ARMOR consistently performs within a factor of two of the Oracle on both real and synthetic datasets over practical ranges of support specifications.

1 Introduction

We focus our attention on the question of how much space remains for performance improvement over current association rule mining algorithms. Our approach is to compare their performance against an **"Oracle algorithm"** that knows *in advance* the identities of all frequent itemsets in the database and only needs to gather the actual supports of these itemsets to complete the mining process. Clearly, *any* practical algorithm will have to do at least this much work in order to generate mining rules. This "Oracle approach" permits us to clearly demarcate the maximal space available for performance improvement over the currently available algorithms. Further, it enables us to construct new mining algorithms from a completely different perspective, namely, as *minimally-altered derivatives* of the Oracle.

First, we show that while the notion of the Oracle is conceptually simple, its *construction* is not equally straightforward. In particular, it is critically dependent on the choice of data structures used during the counting process. We present a carefully engineered implementation of Oracle that makes the best choices for these design parameters at each stage of the counting process. Our experimental results show that there is a considerable gap in the performance between the Oracle and existing mining algorithms.

Second, we present a new mining algorithm, called **ARMOR** (Association Rule Mining based on ORacle), whose structure is derived by making minimal changes to the Oracle, and is guaranteed to complete in two passes over the database. Although ARMOR is derived from the Oracle, it may be seen to share the positive features of a variety of previous algorithms such as PARTITION [9], CARMA [5], AS-CPA [6], VIPER [10] and DELTA [7]. Our empirical study shows that ARMOR consistently performs within a factor of two of the Oracle, over both real (BMS-WebView-1 [14] from Blue Martini Software) and synthetic databases (from the IBM Almaden generator [2]) over practical ranges of support specifications.

The environment we consider is one where the pattern lengths in the database are small enough that the size of mining results is comparable to the available main memory. This holds when the mined data conforms to the sparse nature of market basket data for which association rule mining was originally intended. It is perhaps inappropriate to apply the problem of mining *all* frequent itemsets on dense datasets.

For ease of exposition, we will use the notation shown in Table 1 in the remainder of this paper.

Organization The remainder of this paper is organized as follows: The design of the Oracle algorithm is described in Section 2 and is used to evaluate the performance of current algorithms in Section 3. Our new ARMOR algorithm is presented in Section 4 and its main memory requirements are discussed in Section 5. The performance of ARMOR is evaluated in Section 6. Finally, in Section 7, we summarize the conclusions of our study.

$\mathcal{D}$	Database of customer purchase transactions
$minsup$	User-specified minimum rule support
F	Set of frequent itemsets in $\mathcal{D}$
N	Set of itemsets in the negative border of F
$P_1, ..., P_n$	Set of n disjoint partitions of $\mathcal{D}$
d	Transactions in partitions scanned so far during algorithm execution *excluding* the current partition
d^+	Transactions in partitions scanned so far during algorithm execution *including* the current partition
$\mathcal{G}$	DAG structure to store candidates

Table 1. Notation

2 The Oracle Algorithm

In this section we present the Oracle algorithm which, as mentioned in the Introduction, "magically" knows in advance the identities of all frequent itemsets in the database and only needs to gather the actual supports of these itemsets. Clearly, *any* practical algorithm will have to do at least this much work in order to generate mining rules. Oracle takes as input the database, $\mathcal{D}$ in item-list format (which is organized as a set of rows with each row storing an ordered list of item-identifiers (IID), representing the items purchased in the transaction), the set of frequent itemsets, F, and its corresponding negative border, N, and outputs the supports of these itemsets by making *one scan* over the database. We first describe the mechanics of the Oracle algorithm below and then move on to discuss the rationale behind its design choices in Section 2.2.

2.1 The Mechanics of Oracle

For ease of exposition, we first present the manner in which Oracle computes the supports of 1-itemsets and 2-itemsets and then move on to longer itemsets. Note, however, that the algorithm actually performs all these computations *concurrently* in one scan over the database.

2.1.1 Counting Singletons and Pairs

Data-Structure Description The counters of singletons (1-itemsets) are maintained in a 1-dimensional lookup array, $\mathcal{A}_1$, and that of pairs (2-itemsets), in a lower triangular 2-dimensional lookup array, $\mathcal{A}_2$ (Similar arrays are also used in Apriori [2, 11] for its first two passes.) The k^{th} entry in the array $\mathcal{A}_1$ contains two fields: (1) $count$, the counter for the itemset X corresponding to the k^{th} item, and (2) $index$, the number of frequent itemsets prior to X in $\mathcal{A}_1$, if X is frequent; **null**, otherwise.

```
ArrayCount (T, A1, A2)
Input: Transaction T, 1-itemsets Array A1, 2-itemsets Array A2
Output: Arrays A1 and A2 with their counts updated over T
1.    Itemset T^f = null;
        // to store frequent items from T in Item-List format
2.    for each item i in transaction T
3.        A1[i.id].count++;
4.        if A1[i.id].index ≠ null
5.            append i to T^f
6.    for j = 1 to |T^f| // enumerate 2-itemsets
7.        for k = j + 1 to |T^f|
8.            index1 = A1[T^f[j].id].index // row index
9.            index2 = A1[T^f[k].id].index // column index
10.           A2[index1, index2]++;
```

Figure 1. Counting Singletons and Pairs in Oracle

Algorithm Description The ArrayCount function shown in Figure 1 takes as inputs, a transaction T along with $\mathcal{A}_1$ and $\mathcal{A}_2$, and updates the counters of these arrays over T. In the ArrayCount function, the individual items in the transaction T are enumerated (lines 2–5) and for each item, its corresponding count in $\mathcal{A}_1$ is incremented (line 3). During this process, the frequent items in T are stored in a separate itemset T^f (line 5). We then enumerate all pairs of items contained in T^f (lines 6–10) and increment the counters of the corresponding 2-itemsets in $\mathcal{A}_2$ (lines 8–10).

2.1.2 Counting k-itemsets, $k > 2$

Data-Structure Description Itemsets in $F \cup N$ of length greater than 2 and their related information (counters, etc.) are stored in a DAG structure $\mathcal{G}$, which is pictorially shown in Figure 2 for a database with items {A, B, C, D}. Although singletons and pairs are stored in lookup arrays, as mentioned before, for expository ease, we assume that they too are stored in $\mathcal{G}$ in the remainder of this discussion.

Each itemset is stored in a separate node of $\mathcal{G}$ and is linked to the first two (in a lexicographic ordering) of its subsets. We use the terms "mother" and "father" of an itemset to refer to the (lexicographically) smaller subset and the (lexicographically) larger subset, respectively. E.g., {A, B} and {A, C} are the mother and father respectively of {A, B, C}. For each itemset X in $\mathcal{G}$, we also store with it links to those supersets of X for which X is a mother. We call this list of links as *childset*.

Since each itemset is stored in a separate node in the DAG, we use the terms "itemset" and "node" interchangeably in the remainder of this discussion. Also, we use $\mathcal{G}$ to denote the set of itemsets that are stored in the DAG structure $\mathcal{G}$.

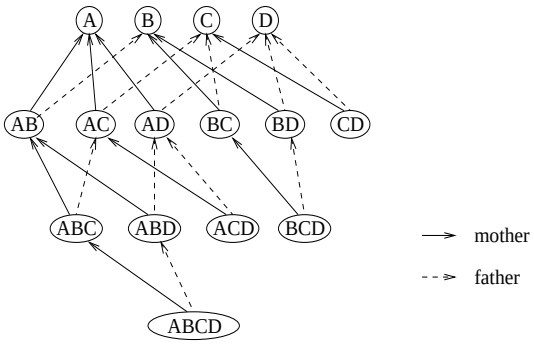


Figure 2. DAG Structure Containing Power Set of {A,B,C,D}

Algorithm Description We use a *partitioning* [9] scheme wherein the database is logically divided into n disjoint horizontal partitions $P_1, P_2, ..., P_n$. In this scheme, itemsets being counted are enumerated only at the *end of each partition* and not after every tuple. Each partition is as large as can fit in available main memory. For ease of exposition, we assume that the partitions are equi-sized. However, we hasten to add that the technique is easily extendible to arbitrary partition sizes.

The pseudo-code of Oracle is shown in Figure 3 and operates as follows: The ReadNextPartition function (line 3) reads tuples from the next partition and *simultaneously* creates tid-lists[1] (within that partition) of singleton itemsets in $\mathcal{G}$. Note that this conversion of the database to the tid-list (TL) format is an *on-the-fly* operation and does not change the complexity of Oracle by more than a (small) constant factor. Next, the Update function (line 5) is applied on each singleton in $\mathcal{G}$. This function takes a node M in $\mathcal{G}$ as input and updates the counts of all descendants of M to reflect their counts over the current partition. The count of any itemset within a partition is equal to the length of its corresponding tidlist (within that partition). The tidlist of an itemset can be obtained as the intersection of the tidlists of its mother and father and this process is started off using the tidlists of frequent 1-itemsets. The exact details of tidlist computation are discussed later.

We now describe the manner in which the itemsets in $\mathcal{G}$ are enumerated after reading in a new partition. The set of links, $\bigcup_{M \in \mathcal{G}} M.childset$, induce a spanning tree of $\mathcal{G}$ (e.g. consider only the solid edges in Figure 2). We perform a *depth first search* on this spanning tree to enumerate all its itemsets. When a node in the tree is visited, we compute the tidlists of all its children. This ensures that when an itemset is visited, the tidlists of its mother and father have already been computed.

[1] A tid-list of an itemset X is an ordered list of TIDs of transactions that contain X.

```
Oracle (D, G)
Input: Database D, Itemsets to be Counted G = F ∪ N
Output: Itemsets in G with Supports
1.    n = Number of Partitions
2.    for i = 1 to n
3.        ReadNextPartition(P_i, G);
4.        for each singleton X in G
5.            Update(X);
```

Figure 3. The Oracle Algorithm

```
Update (M)
Input: DAG Node M
Output: M and its Descendents with Counts Updated
1.    B = convert M.tidlist to Tid-vector format
        // B is statically allocated
2.    for each node X in M.childset
3.        X.tidlist = Intersect(B, X.father.tidlist);
4.        X.count += |X.tidlist|
5.    for each node X in M.childset
6.        Update(X);
```

Figure 4. Updating Counts

```
Intersect (B, T)
Input: Tid-vector B, Tid-list T
Output: B ∩ T
1.    Tid-list result = φ
2.    for each tid in T
3.        offset = tid + 1− (tid of first transaction in current
partition)
4.        if B[offset] = 1 then
5.            result = result ∪ tid
6.    return result
```

Figure 5. Intersection

The above processing is captured in the function Update whose pseudo-code is shown in Figure 4. Here, the tidlist of a given node M is first converted to the tid-vector (TV) format[2] (line 1). Then, tidlists of all children of M are computed (lines 2–4) after which the same children are visited in a depth first search (lines 5–6).

The mechanics of tidlist computation, as promised earlier, are given in Figure 5. The Intersect function shown here takes as input a tid-vector B and a tid-list T. Each tid in T is added to the result if $B[offset]$ is 1 (lines 2–5) where $offset$ is defined in line 3 and represents the position of the transaction T relative to the current partition.

[2] A tid-vector of an itemset X is a bit-vector of 1's and 0's to represent the presence or absence respectively, of X in the set of customer transactions.

2.2 Rationale for the Oracle Design

We show that Oracle is optimal in two respects: (1) It enumerates only those itemsets in $\mathcal{G}$ that need to be enumerated, and (2) The enumeration is performed in the most efficient way possible. These results are based on the following two theorems. Due to lack of space we have deferred the proofs of theorems to [8].

Theorem 2.1 *If the size of each partition is large enough that every itemset in $F \cup N$ of length greater than 2 is present at least once in it, then the only itemsets being enumerated in the Oracle algorithm are those whose counts need to be incremented in that partition.*

Theorem 2.2 *The cost of enumerating each itemset in Oracle is $\Theta(1)$ with a tight constant factor.*

While Oracle is optimal in these respects, we note that there may remain some scope for improvement in the details of *tidlist computation*. That is, the Intersect function (Figure 5) which computes the intersection of a tid-vector B and a tid-list T requires $\Theta(|T|)$ operations. B itself was originally constructed from a tid-list, although this cost is amortized over many calls to the Intersect function. We plan to investigate in our future work whether the intersection of two sets can, in general, be computed more efficiently – for example, using **diffsets**, a novel and interesting approach suggested in [13]. The diffset of an itemset X is the set-difference of the tid-list of X from that of its mother. Diffsets can be easily incorporated in Oracle – only the Update function in Figure 4 of Section 2 is to be changed to compute diffsets instead of tidlists by following the techniques suggested in [13].

Advantages of Partitioning Schemes Oracle, as discussed above, uses a partitioning scheme. An alternative commonly used in current association rule mining algorithms, especially in hashtree [2] based schemes, is to use a tuple-by-tuple approach. A problem with the tuple-by-tuple approach, however, is that there is considerable wasted enumeration of itemsets. The core operation in these algorithms is to determine all candidates that are subsets of the current transaction. Given that a frequent itemset X is present in the current transaction, we need to determine all candidates that are immediate supersets of X and are also present in the current transaction. In order to achieve this, it is often necessary to enumerate and check for the presence of many more candidates than those that are actually present in the current transaction.

3 Performance Study

In the previous section, we have described the Oracle algorithm. In order to assess the performance of current mining algorithms with respect to the Oracle algorithm, we have chosen VIPER [10] and FP-growth [4], among the latest in the suite of online mining algorithms. For completeness and as a reference point, we have also included the classical Apriori in our evaluation suite.

Our experiments cover a range of database and mining workloads, and include the typical and extreme cases considered in previous studies – the only difference is that we also consider database sizes that are *significantly larger* than the available main memory. The performance metric in all the experiments is the *total execution time* taken by the mining operation.

The databases used in our experiments were synthetically generated using the technique described in [2] and attempt to mimic the customer purchase behavior seen in retailing environments. The parameters used in the synthetic generator and their default values are described in Table 2. In particular, we consider databases with parameters T10.I4, T20.I12 and T40.I8 with 10 million tuples in each of them.

Parameter	Meaning	Default Values
N	Number of items	1000
T	Mean transaction length	10, 20, 40
L	Number of potentially frequent itemsets	2000
I	Mean length of potentially frequent itemsets	4, 8, 12
D	Number of transactions in the database	10M

Table 2. Parameter Table

We set the rule support threshold values to as low as was feasible with the available main memory. At these low support values the number of frequent itemsets exceeded twenty five thousand! Beyond this, we felt that the number of rules generated would be enormous and the purpose of mining – to find interesting patterns – would not be served. In particular, we set the rule support threshold values for the T10.I4, T20.I12 and T40.I8 databases to the ranges (0.1%–2%), (0.4%–2%) and (1.15%–5%), respectively.

Our experiments were conducted on a 700-MHz Pentium III workstation running Red Hat Linux 6.2, configured with a 512 MB main memory and a local 18 GB SCSI 10000 rpm disk. For the T10.I4, T20.I12 and T40.I8 databases, the associated database sizes were approximately 500MB, 900MB and 1.7 GB, respectively. All the algorithms in our evaluation suite are written in C++. We implemented a ba-

sic version of the FP-growth algorithm [3] wherein we assume that the entire FP-tree data structure fits in main memory. Finally, the partition size in Oracle was fixed to be 20K tuples.

3.1 Experimental Results for Current Mining Algorithms

We now report on our experimental results. We conducted two experiments to evaluate the performance of current mining algorithms with respect to the Oracle. Our first experiment was run on large (10M tuples) databases, while our second experiment was run on small (100K tuples) databases.

3.1.1 Experiment 1: Performance of Current Algorithms

In our first experiment, we evaluated the performance of Apriori, VIPER and Oracle algorithms for the T10.I4, T20.I12 and T40.I8 databases each containing 10M transactions and these results are shown in Figures 6a–c. The x-axis in these graphs represent the support threshold values while the y-axis represents the response times of the algorithms being evaluated.

In these graphs, we see that the response times of all algorithms increase exponentially as the support threshold is reduced. This is only to be expected since the number of itemsets in the output, $F \cup N$, increases exponentially with decrease in the support threshold.

We also see that there is a considerable gap in the performance of both Apriori and VIPER with respect to Oracle. For example, in Figure 6a, at a support threshold of 0.1%, the response time of VIPER is more than 6 times that of Oracle whereas the response time of Apriori is more than 26 times!

In this experiment, we could not evaluate the performance of FP-growth because it did not complete in any of our runs on large databases due to its heavy and database size dependent utilization of main memory. The reason for this is that FP-growth stores the database itself in a condensed representation in a data structure called FP-tree. In [4], the authors briefly discuss the issue of constructing disk-resident FP-trees. We however, did not take this into account in our implementation.

3.1.2 Experiment 2: Small Databases

Since, as mentioned above, it was not possible for us to evaluate the performance of FP-growth on large databases due to its heavy utilization of main memory, we evaluated the performance of FP-growth and other current algorithms on small databases consisting of 100K transactions. The results of this experiment are shown in Figures 7a–c, which correspond to the T10.I4, T20.I12 and T40.I8 databases, respectively.

In these graphs, we see there continues to be a considerable gap in the performance of current mining algorithms with respect to Oracle. For example, for the T40.I8 database, the response time of FP-growth is more than 8 times that of Oracle for the entire support threshold range.

[3] The original implementation by Han et al. was not available.

4 The ARMOR Algorithm

```
ARMOR (D, I, minsup)
Input: Database D, Set of Items I, Minimum Support minsup
Output: F ∪ N with Supports
1.    n = Number of Partitions

      //—— First Pass ——
2.    G = I        // candidate set (in a DAG)
3.    for i = 1 to n
4.        ReadNextPartition(P_i, G);
5.        for each singleton X in G
6.            X.count += |X.tidlist|
7.            Update1(X, minsup);

      //—— Second Pass ——
8.    RemoveSmall(G, minsup);
9.    OutputFinished(G, minsup);
10.   for i = 1 to n
11.       if (all candidates in G have been output)
12.           exit
13.       ReadNextPartition(P_i, G);
14.       for each singleton X in G
15.           Update2(X, minsup);
```

Figure 8. The ARMOR Algorithm

In the previous section, our experimental results have shown that there is a considerable gap in the performance between the Oracle and existing mining algorithms. We now move on to describe our new mining algorithm, ARMOR (Association Rule Mining based on ORacle). In this section, we overview the main features and the flow of execution of ARMOR – the details of candidate generation are deferred to [8] due to lack of space.

The guiding principle in our design of the ARMOR algorithm is that we consciously make an attempt to determine the *minimal amount of change* to Oracle required to result in an online algorithm. This is in marked contrast to the earlier approaches which designed new algorithms by trying to address the limitations of *previous* online algorithms. That is, we approach the association rule mining problem from a completely different perspective.

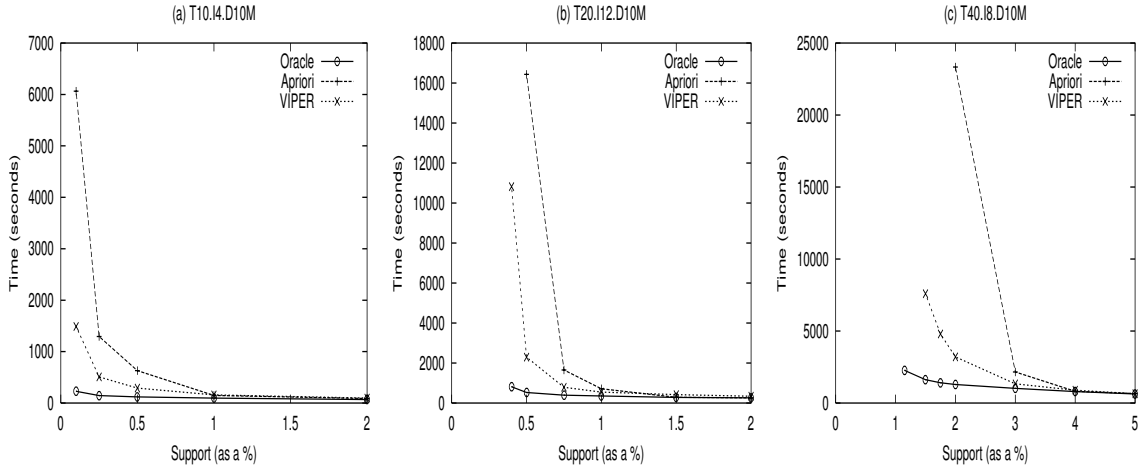


Figure 6. Performance of Current Algorithms (Large Databases)

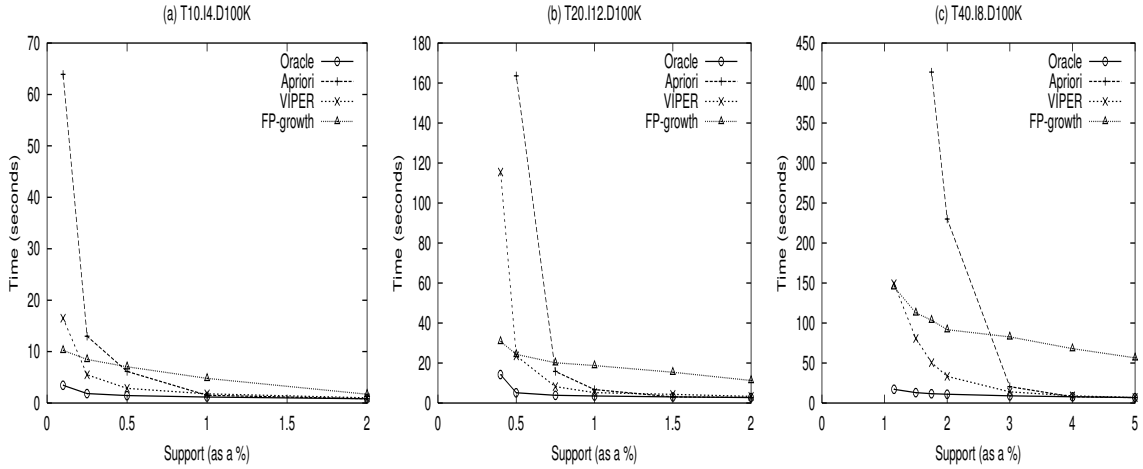


Figure 7. Performance of Current Algorithms (Small Databases)

In ARMOR, as in Oracle, the database is conceptually partitioned into n disjoint blocks $P_1, P_2, ..., P_n$. At most *two* passes are made over the database. In the first pass we form a set of candidate itemsets, $\mathcal{G}$, that is guaranteed to be a superset of the set of frequent itemsets. During the first pass, the counts of candidates in $\mathcal{G}$ are determined over each partition in exactly the same way as in Oracle by maintaining the candidates in a DAG structure. The 1-itemsets and 2-itemsets are stored in lookup arrays as in Oracle. But unlike in Oracle, candidates are inserted and removed from $\mathcal{G}$ at the end of each partition. Generation and removal of candidates is done *simultaneously* while computing counts. The details of candidate generation and removal during the first pass are described in [8] due to lack of space. For ease of exposition we assume in the remainder of this section that all candidates (including 1-itemsets and 2-itemsets) are stored in the DAG.

Along with each candidate X, we also store the following three integers as in the CARMA algorithm [5]: (1) $X.count$: the number of occurrences of X *since* X was last inserted in $\mathcal{G}$. (2) $X.firstPartition$: the index of the partition *at* which X was inserted in $\mathcal{G}$. (3) $X.maxMissed$: upper bound on the number of occurrences of X *before* X was inserted in $\mathcal{G}$. $X.firstPartition$ and $X.maxMissed$ are computed when X is inserted into $\mathcal{G}$ in a manner identical to CARMA.

While the CARMA algorithm works on a tuple-by-tuple basis, we have adapted the semantics of these fields to suit the partitioning approach. If the database scanned so far is d (refer Table 1), then the support of any candidate X in $\mathcal{G}$ will lie in the range $[X.count/|d|, (X.maxMissed + X.count)/|d|]$ [5]. These bounds are denoted by

minSupport(X) and maxSupport(X), respectively. We define an itemset X to be d-frequent if minSupport$(X) \geq minsup$. Unlike in the CARMA algorithm where only d-frequent itemsets are stored at any stage, the DAG structure in ARMOR contains other candidates, including the *negative border* of the d-frequent itemsets, to ensure efficient candidate generation. The details are given in [8].

At the end of the first pass, the candidate set $\mathcal{G}$ is pruned to include only d-frequent itemsets and their negative border. The counts of itemsets in $\mathcal{G}$ over the entire database are determined during the second pass. The counting process is again identical to that of Oracle. No new candidates are generated during the second pass. However, candidates may be removed. The details of candidate removal in the second pass is deferred to [8].

The pseudo-code of ARMOR is shown in Figure 8 and is explained below.

4.1 First Pass

At the beginning of the first pass, the set of candidate itemsets $\mathcal{G}$ is initialized to the set of singleton itemsets (line 2). The ReadNextPartition function (line 4) reads tuples from the next partition and simultaneously creates tid-lists of singleton itemsets in $\mathcal{G}$.

After reading in the entire partition, the Update1 function (details in [8]) is applied on each singleton in $\mathcal{G}$ (lines 5–7). It increments the counts of existing candidates by their corresponding counts in the current partition. It is also responsible for generation and removal of candidates.

At the end of the first pass, $\mathcal{G}$ contains a superset of the set of frequent itemsets. For a candidate in $\mathcal{G}$ that has been inserted at partition P_j, its count over the partitions $P_j, ..., P_n$ will be available.

4.2 Second Pass

At the beginning of the second pass, candidates in $\mathcal{G}$ that are neither d-frequent nor part of the current negative border are removed from $\mathcal{G}$ (line 8). For candidates that have been inserted in $\mathcal{G}$ at the first partition, their counts over the entire database will be available. These itemsets with their counts are output (line 9). The OutputFinished function also performs the following task: If it outputs an itemset X and X has no supersets left in $\mathcal{G}$, X is removed from $\mathcal{G}$.

During the second pass, the ReadNextPartition function (line 13) reads tuples from the next partition and creates tid-lists of singleton itemsets in $\mathcal{G}$. After reading in the entire partition, the Update2 function (details in [8]) is applied on each singleton in $\mathcal{G}$ (lines 14–15). Finally, before reading in the next partition we check to see if there are any more candidates. If not, the mining process terminates.

5 Memory Utilization in ARMOR

In the design and implementation of ARMOR, we have opted for speed in most decisions that involve a space-speed tradeoff. Therefore, the main memory utilization in ARMOR is certainly more as compared to algorithms such as Apriori. However, in the following discussion, we show that the memory usage of ARMOR is well within the reaches of current machine configurations. This is also experimentally confirmed in the next section.

The main memory consumption of ARMOR comes from the following sources: (1) The 1-d and 2-d arrays for storing counters of singletons and pairs, respectively; (2) The DAG structure for storing counters (and tidlists) of longer itemsets, and (3) The current partition.

The total number of entries in the 1-d and 2-d arrays and in the DAG structure corresponds to the number of candidates in ARMOR, which as we have discussed in [8], is only marginally more than $|F \cup N|$. For the moment, if we disregard the space occupied by tidlists of itemsets, then the amortized amount of space taken by each candidate is a small constant (about 10 integers for the dag and 1 integer for the arrays). E.g., if there are 1 million candidates in the dag and 10 million in the array, the space required is about 80MB. Since the environment we consider is one where the pattern lengths are small, the number of candidates will typically be well within the available main memory. [12] discusses alternative approaches when this assumption does not hold.

Regarding the space occupied by tidlists of itemsets, note that ARMOR only needs to store tidlists of d-frequent itemsets. The number of d-frequent itemsets is of the same order as the number of frequent itemsets, $|F|$. The total space occupied by tidlists while processing partition P_i is then bounded by $|F| \times |P_i|$ integers. E.g., if $|F| = 5K$ and $|P_i| = 20K$, then the space occupied by tidlists is bounded by about 400MB. We assume $|F|$ to be in the range of a few thousands at most because otherwise the total number of rules generated would be enormous and the purpose of mining would not be served. Note that the above bound is very pessimistic. Typically, the lengths of tidlists are much smaller than the partition size, especially as the itemset length increases.

Main memory consumed by the current partition is small compared to the above two factors. E.g., If each transaction occupies 1KB, a partition of size 20K would require only 20MB of memory. Even in these extreme examples, the total memory consumption of ARMOR is 500MB, which is acceptable on current machines.

Therefore, *in general we do not expect memory to be an issue* for mining market-basket databases using ARMOR. Further, even if it does happen to be an issue, it is easy to modify ARMOR to free space allocated to tidlists at the ex-

pense of time: $M.tidlist$ can be freed after line 3 in the Update function shown in Figure 4.

A final observation is that the main memory consumption of ARMOR is proportional to the size of the *output* and does not "explode" as the input problem size increases.

6 Experimental Results for ARMOR

We evaluated the performance of ARMOR with respect to Oracle on a variety of databases and support characteristics. We now report on our experimental results for the same performance model described in Section 3. Since Apriori, FP-growth and VIPER have already been compared against Oracle in Section 3.1, we do not repeat those observations here, but focus on the performance of ARMOR. This lends to the visual clarity of the graphs. We hasten to add that ARMOR does outperform the other algorithms.

6.1 Experiment 3: Performance of ARMOR

In this experiment, we evaluated the response time performance of the ARMOR and Oracle algorithms for the T10.I4, T20.I12 and T40.I8 databases each containing 10M transactions and these results are shown in Figures 9a–c.

In these graphs, we first see that ARMOR's performance is close to that of Oracle for high supports. This is because of the following reasons: The density of the frequent itemset distribution is sparse at high supports resulting in only a few frequent itemsets with supports "close" to $minsup$. Hence, frequent itemsets are likely to be locally frequent within most partitions. Even if they are not locally frequent in a few partitions, it is very likely that they are still d-frequent over these partitions. Hence, their counters are updated even over these partitions. Therefore, the complete counts of most candidates would be available at the end of the first pass resulting in a "light and short" second pass. Hence, it is expected that the performance of ARMOR will be close to that of Oracle for high supports.

Since the frequent itemset distribution becomes dense at low supports, the above argument does not hold in this support region. Hence we see that ARMOR's performance relative to Oracle decreases at low supports. But, what is far more important is that ARMOR consistently performs within a *factor of two* of Oracle. This is highlighted in Table 3 where we show the ratios of the performance of ARMOR to that of Oracle for the lowest support values considered for each of the databases.

6.2 Experiment 4: Memory Utilization in ARMOR

The previous experiments were conducted with the total number of items, N, being set to 1K. In this experiment we set the value of N to 20K items for the T10.I4 database – this environment represents an extremely stressful situation for ARMOR with regard to memory utilization due to the very large number of items. Figure 10 shows the memory utilization of ARMOR as a function of support for the N = 1K and N = 20K cases. We see that the main memory utilization of ARMOR scales well with the number of items. For example, at the 0.1% support threshold, the memory consumption of ARMOR for N = 1K items was 104MB while for N = 20K items, it was 143MB – an increase in less than 38% for a 20 times increase in the number of items! The reason for this is that the main memory utilization of ARMOR does not depend directly on the number of items, but only on the size of the output, $F \cup N$, as discussed in Section 5.

Database ($\|\mathcal{D}\|$=10M)	$minsup$ (%)	ARMOR (seconds)	Oracle (seconds)	ARMOR / Oracle
T10.I4	0.1	371.44	226.99	1.63
T20.I12	0.4	1153.42	814.01	1.41
T40.I8	1.15	2703.64	2267.26	1.19

Table 3. Worst-case Efficiency of ARMOR w.r.t Oracle

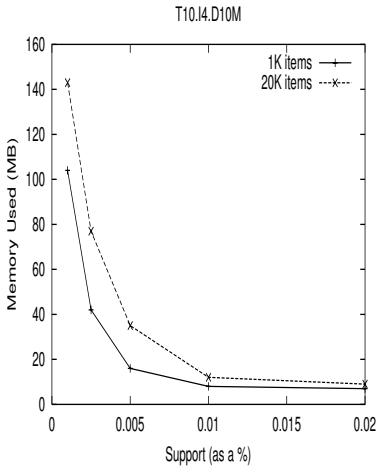


Figure 10. Memory Utilization in ARMOR

6.3 Experiment 5: Real Datasets

Despite repeated efforts, we were unable to obtain large real datasets that conform to the sparse nature of market basket data since such data is not publicly available due to proprietary reasons. The datasets in the UC Irvine public domain repository [3] which are commonly used in data mining studies were not suitable for our purpose since they

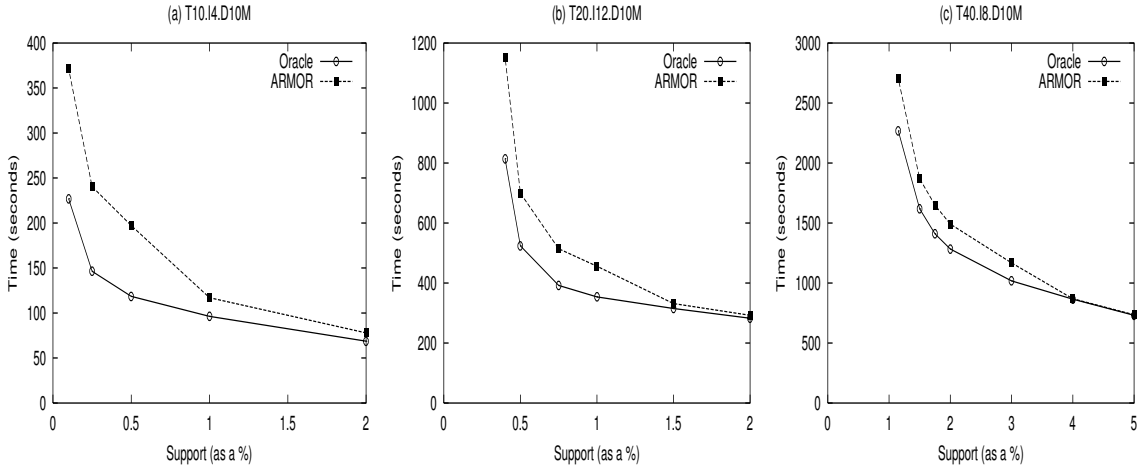


Figure 9. Performance of ARMOR (Synthetic Datasets)

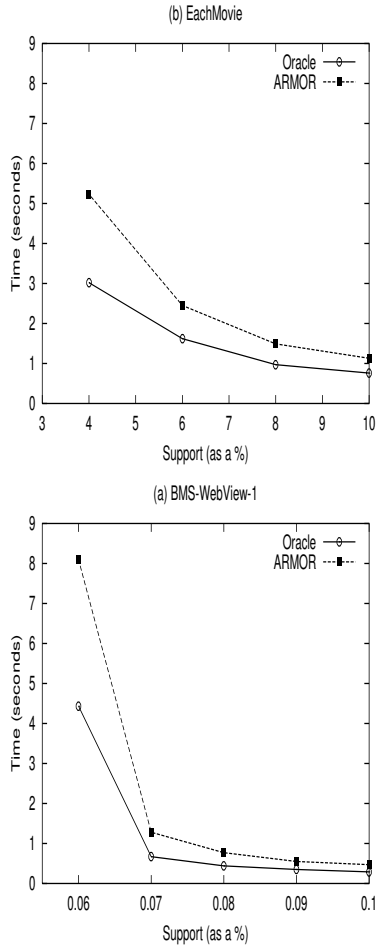


Figure 11. Performance of Armor (Real Datasets)

are dense and have long patterns. We could however obtain two datasets – BMS-WebView-1, a clickstream data from Blue Martini Software [14] and EachMovie, a movie database from Compaq Equipment Corporation [1], which we transformed to the format of boolean market basket data. The resulting databases had 59,602 and 61,202 transactions respectively with 870 and 1648 distinct items.

We set the rule support threshold values for the BMS-WebView-1 and EachMovie databases to the ranges (0.06%–0.1%) and (3%–10%), respectively. The results of these experiments are shown in Figures 11a–b. We see in these graphs that the performance of ARMOR continues to be within twice that of Oracle. The ratio of ARMOR's performance to that of Oracle at the lowest support value of 0.06% for the BMS-WebView-1 database was 1.83 whereas at the lowest support value of 3% for the EachMovie database it was 1.73.

6.4 Discussion of Experimental Results

We now explain the reasons as to why ARMOR should typically perform within a factor of two of Oracle. First, we notice that the only difference between the single pass of Oracle and the first pass of ARMOR is that ARMOR continuously generates and removes candidates. Since the generation and removal of candidates in ARMOR is dynamic and efficient, this does not result in a significant additional cost for ARMOR.

Since candidates in ARMOR that are neither d-frequent nor part of the current negative border are continuously removed, any itemset that is locally frequent within a partition, but not globally frequent in the entire database is likely to be removed from G during the course of the first pass (unless it belongs to the current negative border). Hence the resulting candidate set in ARMOR is a good approximation

of the required mining output. In fact, in our experiments, we found that in the worst case, the number of candidates counted in ARMOR was only about *ten percent* more than the required mining output.

The above two reasons indicate that the cost of the first pass of ARMOR is only slightly more than that of (the single pass in) Oracle.

Next, we notice that the only difference between the second pass of ARMOR and (the single pass in) Oracle is that in ARMOR, candidates are continuously removed. Hence the number of itemsets being counted in ARMOR during the second pass quickly reduces to much less than that of Oracle. Moreover, ARMOR does not necessarily perform a complete scan over the database during the second pass since the second pass ends when there are no more candidates. Due to these reasons, we would expect that the cost of the second pass of ARMOR is usually less than that of (the single pass in) Oracle.

Since the cost of the first pass of ARMOR is usually only slightly more than that of (the single pass in) Oracle and that of the second pass is usually less than that of (the single pass in) Oracle, it follows that ARMOR will typically perform within a factor of two of Oracle.

In summary, due to the above reasons, it appears unlikely for it to be possible to design algorithms that substantially reduce either the number of database passes or the number of candidates counted. These represent the primary bottlenecks in association rule mining. Further, since ARMOR utilizes the same itemset counting technique of Oracle, further overall improvement without domain knowledge seems extremely difficult. Finally, even though we have not proved optimality of Oracle with respect to tidlist intersection, we note that any smart intersection techniques that may be implemented in Oracle can also be used in ARMOR.

7 Conclusions

A variety of novel algorithms have been proposed in the recent past for the efficient mining of association rules, each in turn claiming to outperform its predecessors on a set of standard databases. In this paper, our approach was to quantify the algorithmic performance of association rule mining algorithms with regard to an idealized, but practically infeasible, "Oracle". The Oracle algorithm utilizes a partitioning strategy to determine the supports of itemsets in the required output. It uses direct lookup arrays for counting singletons and pairs and a DAG data-structure for counting longer itemsets. We have shown that these choices are optimal in that only required itemsets are enumerated and that the cost of enumerating each itemset is $\Theta(1)$. Our experimental results showed that there was a substantial gap between the performance of current mining algorithms and that of the Oracle.

We also presented a new online mining algorithm called ARMOR (Association Rule Mining based on ORacle), that was constructed with minimal changes to Oracle to result in an online algorithm. ARMOR utilizes a new method of candidate generation that is dynamic and incremental and is guaranteed to complete in two passes over the database. Our experimental results demonstrate that ARMOR performs within a *factor of two* of Oracle.

Acknowledgments This work was partially supported by a Swarnajayanti Fellowship from the Dept. of Science and Technology, Govt. of India.

References

[1] Eachmovie collaborative filtering data set. http://www.research.compaq.com/SRC/eachmovie/, 1997.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, Sept. 1994.

[3] C. L. Blake and C. J. Merz. UCI repository of machine learning databases. http://www.ics.uci.edu/∼mlearn/MLRepository.html, 1998.

[4] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[5] C. Hidber. Online association rule mining. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, June 1999.

[6] J. Lin and M. H. Dunham. Mining association rules: Anti-skew algorithms. In *Proc. of Intl. Conf. on Data Engineering (ICDE)*, 1998.

[7] V. Pudi and J. Haritsa. Quantifying the utility of the past in mining large databases. *Information Systems*, July 2000.

[8] V. Pudi and J. Haritsa. On the optimality of association-rule mining algorithms. Technical Report TR-2001-01, DSL, Indian Institute of Science, 2001.

[9] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, 1995.

[10] P. Shenoy, J. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *Proc. of ACM SIGMOD Intl. Conf. on Management of Data*, May 2000.

[11] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. of Intl. Conf. on Very Large Databases (VLDB)*, Sept. 1995.

[12] Y. Xiao and M. H. Dunham. Considering main memory in mining association rules. In *Proc. of Intl. Conf. on Data Warehousing and Knowledge Discovery (DAWAK)*, 1999.

[13] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, Rensselaer Polytechnic Institute, 2001.

[14] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In *Proc. of Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*, Aug. 2001.

AIM: Another Itemset Miner

Amos Fiat, Sagi Shporer
School of Computer Science
Tel-Aviv University
Tel Aviv, Israel
{fiat, shporer}@tau.ac.il

Abstract

We present a new algorithm for mining frequent itemsets. Past studies have proposed various algorithms and techniques for improving the efficiency of the mining task. We integrate a combination of these techniques into an algorithm which utilize those techniques dynamically according to the input dataset. The algorithm main features include depth first search with vertical compressed database, diffset, parent equivalence pruning, dynamic reordering and projection. Experimental testing suggests that our algorithm and implementation significantly outperform existing algorithms/implementations.

1. Introduction

Finding association rules is one of the driving applications in data mining, and much research has been done in this field [10, 7, 4, 6]. Using the support-confidence framework, proposed in the seminal paper of [1], the problem is split into two parts — (a) finding frequent itemsets, and (b) generating association rules.

Let I be a set of items. A subset $X \subseteq I$ is called an itemset. Let D be a transactional database, where each transaction $T \in D$ is a subset of $I : T \subseteq I$. For an itemset X, support(X) is defined to be the number of transactions T for which $X \subseteq T$. For a given parameter minsupport, an itemset X is call a *frequent itemset* if support$(X) \geq$ minsupport. The set of all frequent itemsets is denoted by $\mathcal{F}$.

The remainder of this paper is organized as follows. Section 2 contains a short of related work. In section 3 we describe the AIM-$\mathcal{F}$ algorithm. Section 4 contains experimental results. In Section 5 we conclude this short abstact with a discussion.

1.1. Contributions of this paper

We combine several pre-existing ideas in a fairly straightforward way and get a new frequent itemset mining algorithm. In particular, we combine the sparse vertical bit vector technique along with the difference sets technique of [14], thus reducing the computation time when compared with [14]. The various techniques were put in use dynamically according to the input dataset, thus utilizing the advantages and avoiding the drawbacks of each technique.

Experimental results suggest that for a given level of support, our algorithm/implementation is faster than the other algorithms with which we compare ourselves. This set includes the dEclat algorithm of [14] which seems to be the faster algorithm amongst all others.

2. Related Work

Since the introduction of the *Apriori* algorithm by [1, 2] many variants have been proposed to reduce time, I/O and memory.

Apriori uses breath-first search, bottom-up approach to generate frequent itemsets. (*I.e.*, constructs $i+1$ item frequent itemsets from i item frequent itemsets). The key observation behind Apriori is that all subsets of a frequent itemset must be frequent. This suggests a natural approach to generating frequent itemsets. The breakthrough with Apriori was that the number of itemsets explored was polynomial in the number of frequent itemsets. In fact, on a worst case basis, Apriori explores no more than n itemsets to output a frequent itemset, where n is the total number of items.

Subsequent to the publication of [1, 2], a great many variations and extensions were considered [3, 7, 13]. In [3] the number of passes over the database was reduced . [7] tried to reduce the search space by combining bottom-up and top-down search – if a set is infre-

quent than so are supersets, and one can prune away infrequent itemsets found during the top-down search. [13] uses equivalence classes to skip levels in the search space. A new mining technique, FP-Growth, proposed in [12], is based upon representing the dataset itself as a tree. [12] perform the mining from the tree representation.

We build upon several ideas appearing in previous work, a partial list of which is the following:

- Vertical Bit Vectors [10, 4] - The dataset is stored in vertical bit vectors. Experimentally, this has been shown to be very effective.
- Projection [4] - A technique to reduce the size of vertical bit vectors by trimming the bit vector to include only transaction relevant to the subtree currently being searched.
- Difference sets [14] - Instead of holding the entire tidset at any given time, Diffsets suggest that only changes in the tidsets are needed to compute the support.
- Dynamic Reordering [6] - A heuristic for reducing the search space - dynamically changing the order in which the search space is traversed. This attempts to rearrange the search space so that one can prune infrequent itemsets earlier rather than later.
- Parent Equivalence Pruning [4, 13] - Skipping levels in the search space, when a certain item added to the itemset contributes no new information.

To the best of our knowledge no previous implementation makes use of this combination of ideas, and some of these combinations are non-trivial to combine. For example, projection has never been previously used with difference sets and to do so requires some new observations as to how to combine these two elements.

We should add that there are a wide variety of other techniques introduced over time to find frequent itemsets, which we do not make use of. A very partial list of these other ideas is

- Sampling - [11] suggest searching over a sample of the dataset, and later validates the results using the entire dataset. This technique was shown to generate the vast majority of frequent itemsets.
- Adjusting support - [9] introduce SLPMiner, an algorithm which lowers the support as the itemsets grow larger during the search space. This attempts to avoid the problem of generating small itemsets which are unlikely to grow into large itemsets.

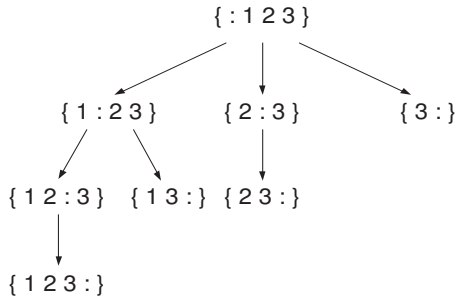


Figure 1. Full lexicographic tree of 3 items

3. The AIM-$\mathcal{F}$ algorithm

In this section we describe the building blocks that make up the AIM-$\mathcal{F}$ algorithm. High level pseudo code for the AIM-$\mathcal{F}$ algorithm appears in Figure 7.

3.1. Lexicographic Trees

Let $<$ be some lexicographic order of the items in I such that for every two items i and j, $i \neq j : i < j$ or $i > j$. Every node n of the lexicographic tree has two fields, n.head which is the itemset node n represent, and n.tail which is a list of items, possible extensions to n.head. A node of the lexicographic tree has a $l$evel. Itemsets for nodes at level k nodes contain k items. We will also say that such itemsets have length k. The root (level 0) node n.head is empty, and n.tail $= I$. Figure 1 is an example of lexicographic tree for 3 items.

The use of lexicographic trees for itemset generation was proposed by [8].

3.2. Depth First Search Traversal

In the course of the algorithm we traverse the lexicographic tree in a depth-first order. At node n, for every element α in the node's tail, a new node n' is generated such that n'.head $= n$.head $\bigcup \alpha$ and n'.tail $= n$.tail $- \alpha$. After the generation of n', α is removed from n.tail, as it will be no longer needed (see Figure 3).

Several pruning techniques, on which we elaborate later, are used in order to speed up this process.

3.3 Vertical Sparse Bit-Vectors

Comparison between horizontal and vertical database representations done in [10] shows that the representation of the database has high impact on the performance of the mining algorithm. In a vertical database the data is represented as a list of items,

```
Project(p : vector, v : vector )
/* p - vector to be projected upon
   v - vector being projected */
(1) t = Empty Vector
(2) i = 0
(3) for each nonzero bit in p, at offset j, in
    ascending order of offsets:
(4)     Set i'th bit of target vector t to be the
        j'th bit of v.
(5)     i = i + 1
(6) return t
```

Figure 2. Projection

```
DFS(n : node,)
(1) t = n.tail
(2) while t ≠ ∅
(3)     Let α be the first item in t
(4)     remove α from t
(5)     n'.head = n.head ⋃ α
(6)     n'.tail = t
(7)     DFS(n')
```

Figure 3. Simple DFS

where every item holds a list of transactions in which it appears.

The list of transactions held by every item can be represented in many ways. In [13] the list is a tid-list, while [10, 4] use vertical bit vectors. Because the data tends to be sparse, vertical bit vectors hold many "0" entries for every "1", thus wasting memory and CPU for processing the information. In [10] the vertical bit vector is compressed using an encoding called *skinning* which shrinks the size of the vector.

We choose to use a sparse vertical bit vector. Every such bit vector is built from two arrays - one for values, and one for indexes. The index array gives the position in the vertical bit vector, and the value array is the value of the position, see Figure 8. The index array is sorted to allow fast AND operations between two sparse bit vectors in a similar manner to the AND operation between the tid-lists. Empty values will be thrown away during the AND operation, save space and computation time.

3.3.1 Bit-vector projection

In [4], a technique called projection was introduced. Projection is a sparse bit vector compression technique specifically useful in the context of mining frequent

```
Apriori(n : node, minsupport : integer)
(1) t = n.tail
(2) while t ≠ ∅
(3)     Let α be the first item in t
(4)     remove α from t
(5)     n'.head = n.head ⋃ α
(6)     n'.tail = t
(7)     if (support(n'.head) ≥ minsupport)
(8)         Report n'.head as frequent itemset
(9)         Apriori(n')
```

Figure 4. Apriori

```
PEP(n : node, minsupport : integer)
(1) t = n.tail
(2) while t ≠ ∅
(3)     Let α be the first item in t
(4)     remove α from t
(5)     n'.head = n.head ⋃ α
(6)     n'.tail = t
(7)     if (support(n'.head) = support(n.head))
(8)         add α to the list of items removed by
            PEP
(9)     else if (support(n'.head) ≥ minsupport)
(10)        Report n'.head ⋃{All subsets of items
            removed by PEP} as frequent itemsets
(11)        PEP(n')
```

Figure 5. PEP

itemsets. The idea is to eliminate redundant zeros in the bit-vector - for itemset P, all the transactions which does not include P are removed, leaving a vertical bit vector containing only 1s. For every itemset generated from P (a superset of P), PX, all the transactions removed from P are also removed. This way all the extraneous zeros are eliminated.

The projection done directly from the vertical bit representation. At initialization a two dimensional matrix of 2^w by 2^w is created, where w is the word length or some smaller value that we choose to work with. Every entry (i,j) is calculated to be the projection of j on i (thus covering all possible projections of single word). For every row of the matrix, the number of bits being projected is constant (a row represents the word being projected upon).

Projection is done by traversing both the vector to project upon, p, and the vector to be projected, v. For every word index we compute the projection by table

DynamicReordering(n : node, minsupport : integer)
(1) $t = n$.tail
(2) for each α in t
(3) Compute $s_\alpha = \text{support}(n.\text{head} \bigcup \alpha)$
(4) Sort items α in t by s_α in ascending order.
(5) while $t \neq \emptyset$
(6) Let α be the first item in t
(7) remove α from t
(8) $n'.\text{head} = n.\text{head} \bigcup \alpha$
(9) $n'.tail = t$
(10) if (support($n'.head$) $\geq$ minsupport)
(11) Report n'.head as frequent itemset
(12) *DynamicReordering*(n')

Figure 6. Dynamic Reordering

lookup, the resulting bits are then concatenated together. Thus, computing the projection takes no longer than the AND operation between two compressed vertical bit lists.

In [4] projection is used whenever a *rebuilding threshold* was reached. Our tests show that because we're using sparse bit vectors anyway, the gain from projection is smaller, and the highest gains are when we use projection only when calculating the 2-itemsets from 1-itemsets. This is also because of the penalty of using projection with diffsets, as described later, for large k-itemsets. Even so, projection is used only if the sparse bit vector will shrunk significantly - as a threshold we set 10% - if the sparse bit vector contains less than 10% of '1's it will be projected.

3.3.2 Counting and support

To count the number of ones within a sparse bit vector, one can hold a translation table of 2^w values, where w is the word length. To count the number of ones in a word requires only one memory access to the translation table. This idea first appeared in the context of frequent itemsets in [4].

3.4 Diffsets

Difference sets (*Diffsets*), proposed in [14], are a technique to reduce the size of the intermediate information needed in the traversal using a vertical database. Using Diffsets, only the differences between the candidate and its generating itemsets is calculated and stored (if necessary). Using this method the intermediate vertical bit-vectors in every step of the DFS traversal are shorter, this results in faster intersections

AIM-$\mathcal{F}$(n : node, minsupport : integer)
/* Uses DFS traversal of lexicographic itemset tree
Fast computation of small frequent itemsets
for sparse datasets
Uses difference sets to compute support
Uses projection and bit vector compression
Makes use of parent equivalence pruning
Uses dynamic reordering */
(1) $t = n$.tail
(2) for each α in t
(3) Compute $s_\alpha = \text{support}(n.\text{head} \bigcup \alpha)$
(4) if ($s_\alpha = \text{support}(n.\text{head})$)
(5) add α to the list of items removed by PEP
(6) remove α from t
(7) else if ($s_\alpha <$ minsupport)
(8) remove α from t
(9) Sort items in t by s_α in ascending order.
(10) While $t \neq \emptyset$
(11) Let α be the first item in t
(12) remove α from t
(13) $n'.\text{head} = n.\text{head} \bigcup \alpha$
(14) $n'.\text{tail} = t$
(15) Report $n'.\text{head} \bigcup \{$All subsets of items removed by PEP$\}$ as frequent itemsets
(16) *AIM-$\mathcal{F}$*(n')

Figure 7. AIM-$\mathcal{F}$

between those vectors.

Let $t(P)$ be the tidset of P. The Diffset $d(PX)$ is the tidset of tids that are in $t(P)$ but not in $t(PX)$, formally : $d(PX) = t(P) - t(PX) = t(P) - t(X)$. By definition $\text{support}(PXY) = \text{support}(PX) - |d(PXY)|$, so only $d(PXY)$ should be calculated. However $d(PXY) = d(PY) - d(PX)$ so the Diffset for every candidate can be calculated from its generating itemsets.

Diffsets have one major drawback - in datasets, where the support drops rapidly between k-itemset to k+1-itemset then the size of $d(PX)$ can be larger than the size of $t(PX)$ (For example see figure 9). In such cases the usage of diffsets should be delayed (in the depth of the DFS traversal) to such k-itemset where the support stops the rapid drop. Theoretically the break even point is 50%: $\frac{t(PX)}{t(P)} = 0.5$, where the size of $d(PX)$ equals to $t(PX)$, however experiments shows small differences for any value between 10% to 50%. For this algorithm we used 50%.

Diffsets and Projection : As $d(PXY)$ in not a subset of $d(PX)$, Diffsets cannot be used directly for projection. Instead, we notice that $d(PXY) \subseteq$

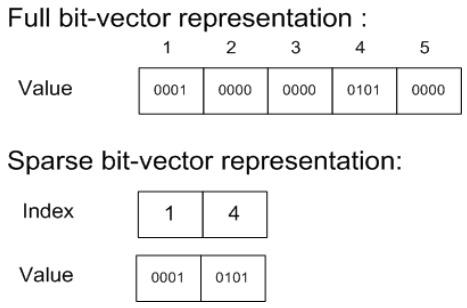


Figure 8. Sparse Bit-Vector data structure

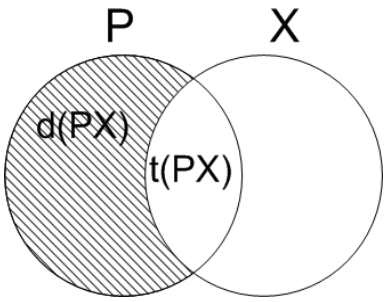


Figure 9. Diffset threshold

$t(PX)$ and $t(PX) = t(P) - d(PX)$. However $d(PX)$ is known, and $t(P)$ can be calculated in the same way. For example $t(ABCD) = t(ABC) - d(ABCD)$, $t(ABC) = t(AB) - d(ABC)$, $t(AB) = t(A) - d(AB)$ thus $t(ABCD) = t(A) - d(AB) - d(ABC) - d(ABCD)$. Using this formula the $t(PX)$ can be calculated using the intermediate data along the DFS trail. As the DFS goes deeper, the penalty of calculating the projection is higher.

3.5 Pruning Techniques

3.5.1 Apriori

Proposed by [2] the *Apriori* pruning technique is based on the monotonicity property of support: $\text{support}(P) \geq \text{support}(PX)$ as PX is contained in less transactions than P. Therefore if for an itemset P, $\text{support}(P) < \text{minsupport}$, the support of any extension of P will also be lower than minsupport, and the subtree rooted at P can be pruned from the lexicographic tree. See Figure 4 for pseudo code.

3.5.2 Parent Equivalence Pruning (PEP)

This is a pruning method based on the following property : If $\text{support}(n.\text{head}) = \textit{support}(n.\text{head} \bigcup \alpha)$ then all the transactions that contain $n.\text{head}$ also contain $n.\text{head} \bigcup \alpha$. Thus, X can be moved from the tail to the head, thus saving traversal of P and skipping to PX. This method was described by [4, 13]. Later when the frequent items are generated the items which were moved from head to tail should be taken into account when listing all frequent itemsets. For example, if k items were pruned using PEP during the DFS traversal of frequent itemset X then the all 2^k subsets of those k items can be added to X without reducing the support. This gives creating 2^k new frequent itemsets. See Figure 5 for pseudo code.

3.6 Dynamic Reordering

To increase the chance of early pruning, nodes are traversed, not in lexicographic order, but in order determined by support. This technique was introduced by [6].

Instead of lexicographic order we reorder the children of a node as follows. At node n, for all α in the tail, we compute $s_\alpha = \text{support}(t.\text{head} \bigcup \alpha)$, and the items are sorted in by s_α in increasing order. Items α in $n.\text{tail}$ for which $\text{support}(t.\text{head} \bigcup \alpha) < \text{minsupport}$ are trimmed away. This way, the rest of the sub-tree will benefit from a shortened tail. Items with smaller support, which are heuristically "likely" to be pruned earlier, will be traversed first. See Figure 6 for pseudo code.

3.7 Optimized Initialization

In sparse datasets computing frequent 2-itemsets can be done more efficiently than than by performing $\binom{n}{2}$ itemset intersections. We use a method similar to the one described in [13]: as a preprocessing step, for every transaction in the database, all 2-itemsets are counted and stored in an upper-matrix of dimensions $n \times n$. This step may take up to $O(n^2)$ operations per transaction. However, as this is done only for sparse datasets, experimentally one sees that the number of operations is small. After this initialization step, we are left with frequent 2 item itemsets from which we can start the DFS proceedure.

4. Experimental Results

The experiments were conducted on an Athlon 1.2Ghz with 256MB DDR RAM running Microsoft Windows XP Professional. All algorithms where compiled on VC 7. In the experiments described herein, we only count frequent itemsets, we don't create output.

We used five datasets to evaluate the algorithms performance. Those datasets where studied extensively in [13].

1. *connect* — A database of game states in the game connect 4.
2. *chess* — A database of game states in chess.
3. *mushroom* — A database with information about various mushroom species.
4. *pumsb** — This dataset was derived from the *pumsb* dataset and describes census data.
5. *T10I4D100K* - Synthetic dataset.

The first 3 datasets were taken from the UN Irvine ML Database Repository (http://www.ics.uci.edu/ mlearn/MLRepository). The synthetic dataset created by the IBM Almaden synthetic data generator (http://www.almaden.ibm.com/cs/quest/demos.html).

4.1 Comparing Data Representation

We compare the memory requirements of sparse vertical bit vector (with the projection described earlier) versus the standard tid-list. For every itemset length the total memory requirements of all tid-sets is given in figures 10, 11 and 12. We do not consider itemsets removed by PEP.

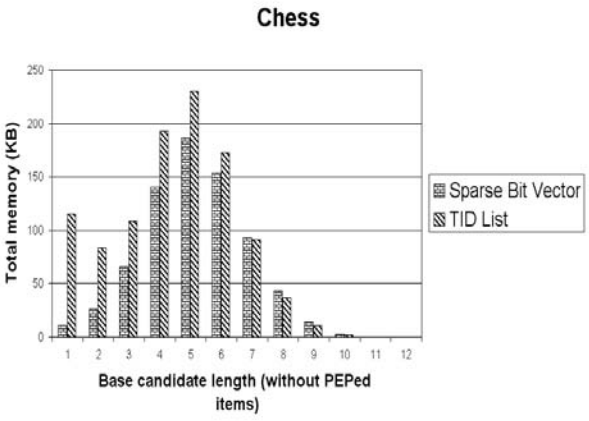


Figure 10. Chess - support 2000 (65%)

As follows from the figures, our sparse vertical bit vector representation requires less memory than tid-list for the dense datasets (chess, connect). However for the sparse dataset (T10I4D100K) the sparse vertical bit vector representation requires up to twice as much memory as tid-list. Tests to dynamically move from sparse vertical bit vector representation to tid-lists showed no significant improvement in performance, however, this should be carefully verified in further experiments.

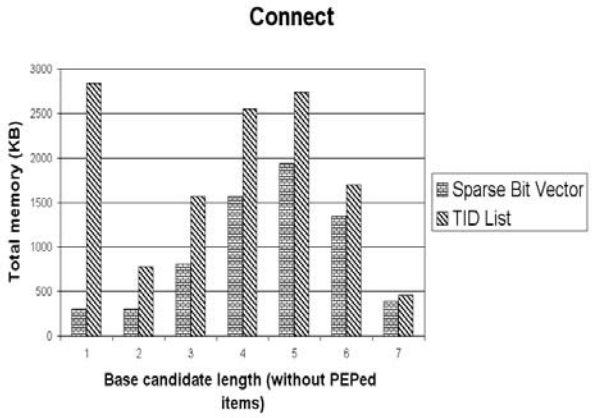


Figure 11. Connect - support 50000 (75%)

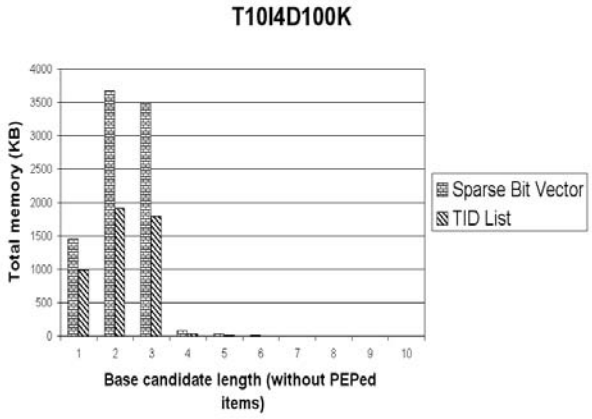


Figure 12. T10I4D100K - support 100 (0.1%)

4.2 Comparing The Various Optimizations

We analyze the influence of the various optimization techniques on the performance of the algorithm. First run is the final algorithm on a given dataset, then returning on the task, with a single change in the algorithm. Thus trying to isolate the influence of every optimization technique, as shown in figures 13 and 14.

As follows from the graphs, there is much difference in the behavior between the datasets. In the dense dataset, Connect, the various techniques had tremendous effect on the performance. PEP, dynamic reorder-

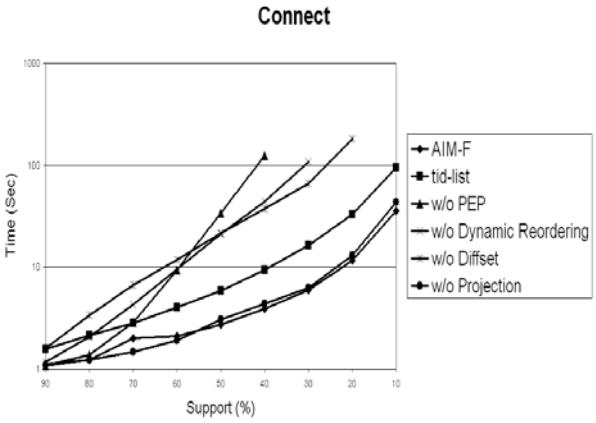


Figure 13. In¤uence of the various optimization on the Connect dataset mining

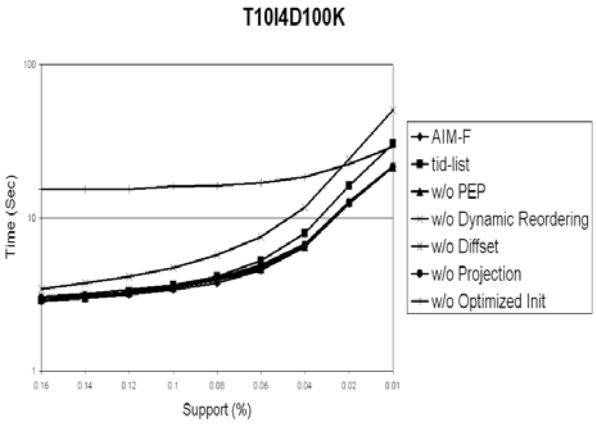


Figure 14. In¤uence of the various optimization on the T10I4D100K dataset mining

ing and diffsets behaved in a similar manner, and the performance improvement factor gained by of them increased as the support dropped. From the other hand the sparse bit vector gives a constant improvement factor over the tid-list for all the tested support values, and projection gives only a minor improvement.

In the second figure, for the sparse dataset T10I4D100K, the behavior is different. PEP gives no improvement, as can expected in sparse dataset, as every single item has a low support, and does not contain existing itemsets. There is drop in the support from k-itemset to k+1-itemset due to the low support therefore diffset also gives no impact, and the same goes for projection. A large gain in performance is made by optimized initialization, however the performance gain is constant, and not by a factor. Last is the dynamic reordering which contributes to early pruning much like in the dense dataset.

4.3 Comparing Mining Algorithms

For comparison, we used implementations of

1. Apriori [2] - horizontal database, BFS traversal of the candidates tree.
2. FPgrowth [5] - tree projected database, searching for frequent itemsets directly without candidate generation, and
3. dEclat [13] - vertical database, DFS traversal using diffsets.

All of the above algorithm implementations were provided by Bart Goethals (http://www.cs.helsinki/u/goethals/) and used for comparison with the AIM-$\mathcal{F}$ implementation.

Figures 15 to 19 gives experimental results on the various algorithms and datasets. Not surprising, Apriori [2] generally has the lowest performance amongst the algorithms compared, and in some cases the running time could not be computed as it did not finish even at the highest level of support checked. For these datasets and compared with the specific algorithms and implementations described above, our algorithm/implementation, AIM-$\mathcal{F}$, seemingly outperforms all others.

In general, for the dense datasets (Chess, Connect, Pumsb* and Mushroom, figures 15,16,17 and 18 respectively), the sparse bit vector gives AIM-$\mathcal{F}$ an order of magnitude improvement over dEclat. The diffsets gives dEclat and AIM-$\mathcal{F}$ another order of magnitude improvement over the rest of the algorithms.

For the sparse dataset T10I4D100K (Figure 19), the optimized initialization gives AIM-$\mathcal{F}$ head start, which

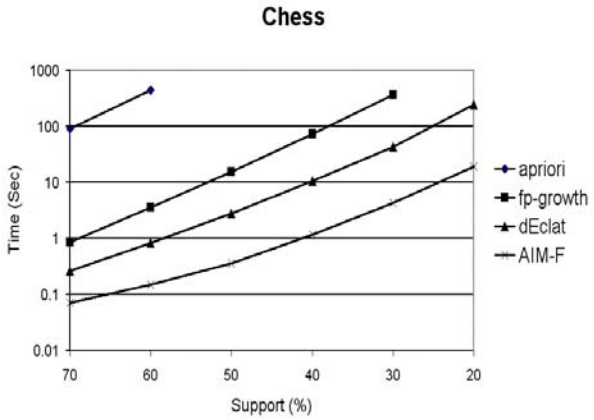


Figure 15. Chess dataset

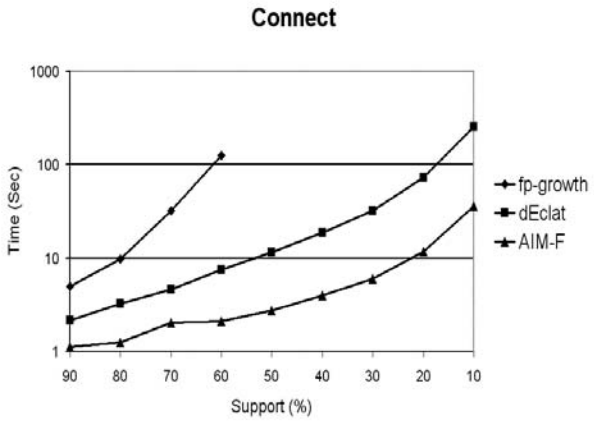


Figure 16. Connect dataset

is combined in the lower supports with the advantage of the sparse vertical bit vector (See details in figure 14)

5. Afterword

This paper presents a new frequent itemset mining algorithm, AIM-$\mathcal{F}$. This algorithm is based upon a mixture of previously used techniques combined dynamically. It seems to behave quite well experimentally.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, pages 207–216, 1993.

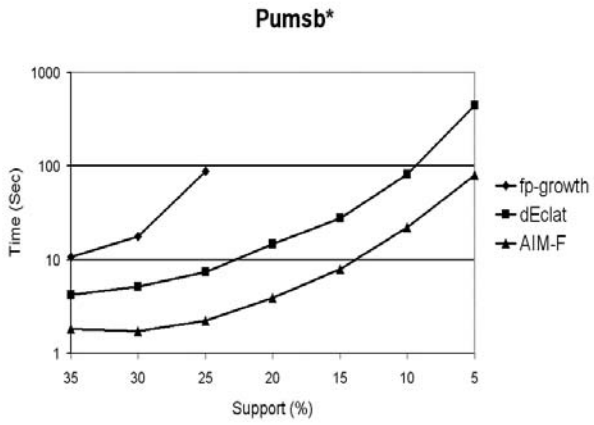


Figure 17. Pumsb* dataset

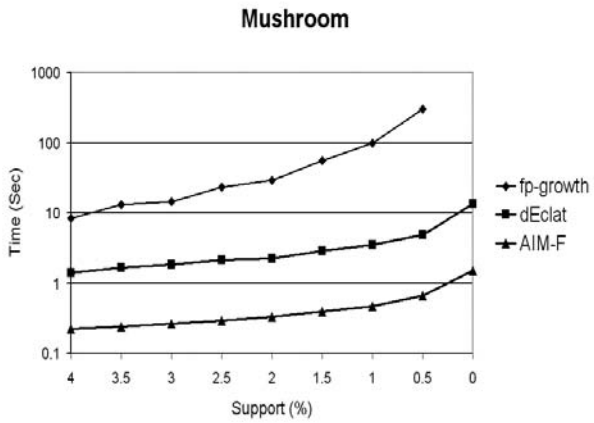


Figure 18. Mushroom dataset

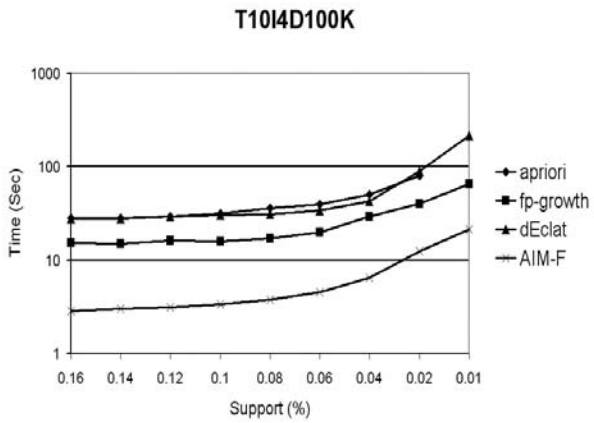


Figure 19. T10I4D100K dataset

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J. B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 12–15 1994.

[3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, pages 255–264, 1997.

[4] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: a maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

[6] R. J. B. Jr. Efficiently mining long patterns from databases. In *SIGMOD*, pages 85–93, 1998.

[7] D.-I. Lin and Z. M. Kedem. Pincer search: A new algorithm for discovering the maximum frequent set. In *EDBT'98*, volume 1377 of *Lecture Notes in Computer Science*, pages 105–119, 1998.

[8] R. Rymon. Search through systematic set enumeration. In *KR-92*, pages 539–550, 1992.

[9] M. Seno and G. Karypis. Slpminer: An algorithm for finding frequent sequential patterns using length decreasing support constraint. In *ICDE*, 2002.

[10] P. Shenoy, J. R. Haritsa, S. Sundarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *SIGMOD*, 2000.

[11] H. Toivonen. Sampling large databases for association rules. In *VLDB*, pages 134–145, 1996.

[12] S. Yen and A. Chen. An efficient approach to discovering knowledge from large databases. In *4th International Conference on Parallel and Distributed Information Systems*.

[13] M. J. Zaki. Scalable algorithms for association mining. *Knowledge and Data Engineering*, 12(2):372–390, 2000.

[14] M. J. Zaki and K. Gouda. Fast vertical mining using diffsets. Technical Report 01-1, RPI, 2001.

LCM: An Efficient Algorithm for Enumerating Frequent Closed Item Sets

Takeaki Uno[1], Tatsuya Asai[2], Yuzo Uchida[2], Hiroki Arimura[2]
[1] National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430, Japan
e-mail: uno@nii.jp
[2] Department of Informatics, Kyushu University, 6-10-1 Hakozaki, Fukuoka 812-0053, JAPAN
e-mail:{t-asai,y-uchida,arim}@i.kyushu-u.ac.jp

Abstract:

In this paper, we propose three algorithms LCM-freq, LCM, and LCMmax for mining all frequent sets, frequent closed item sets, and maximal frequent sets, respectively, from transaction databases. The main theoretical contribution is that we construct tree-shaped transversal routes composed of only frequent closed item sets, which is induced by a parent-child relationship defined on frequent closed item sets. By traversing the route in a depth-first manner, LCM finds all frequent closed item sets in polynomial time per item set, without storing previously obtained closed item sets in memory. Moreover, we introduce several algorithmic techniques using the sparse and dense structures of input data. Algorithms for enumerating all frequent item sets and maximal frequent item sets are obtained from LCM as its variants. By computational experiments on real world and synthetic databases to compare their performance to the previous algorithms, we found that our algorithms are fast on large real world datasets with natural distributions such as KDD-cup2000 datasets, and many other synthetic databases.

1. Introduction

Frequent item set mining is one of the fundamental problems in data mining and has many applications such as association rule mining [1], inductive databases [9], and query expansion [12].

Let E be the universe of *items*, consisting of items $1, ..., n$. A subset X of E is called an *item set*. $\mathcal{T}$ is a set of *transactions* over E, i.e., each $T \in \mathcal{T}$ is composed of items of E. For an item set X, let $\mathcal{T}(X) = \{ t \in \mathcal{T} \mid X \subseteq t \}$ be the set of transactions including X. Each transaction of $\mathcal{T}(X)$ is called an *occurrence* of X. For a given constant $\alpha \geq 0$, an item set X is called *frequent* if $|\mathcal{T}(X)| \geq \alpha$. If a frequent item set is included in no other frequent set, it is said to be *maximal*. For a transaction set $\mathcal{S} \subseteq \mathcal{T}$, let $I(\mathcal{S}) = \bigcap_{T \in \mathcal{S}} T$. If an item set X satisfies $I(\mathcal{T}(X)) = X$, then X is called a *closed item set*. We denote by $\mathcal{F}$ and $\mathcal{C}$ the sets of all frequent itemsets and all frequent closed item sets, respectively.

In this paper, we propose an efficient algorithm LCM for enumerating all frequent closed item sets. LCM is an abbreviation of *Linear time Closed item set Miner*. Existing algorithms for this task basically enumerate frequent item sets with cutting off unnecessary frequent item sets by pruning. However, the pruning is not complete, hence the algorithms operate unnecessary frequent item sets, and do something more. In LCM, we define a parent-child relationship between frequent closed item sets. The relationship induces tree-shaped transversal routes composed only of all the frequent closed item sets. Our algorithm traverses the routes, hence takes linear time of the number of frequent closed item sets. This algorithm is obtained from the algorithms for enumerating maximal bipartite cliques [14, 15], which is designed based on reverse search technique [3, 16].

In addition to the search tree technique for closed item sets, we use several techniques to speed-up the update of the occurrences of item sets. One technique is *occurrence deliver*, which simutaneously computes the occurrence sets of all the successors of the current item set during a single scan on the current occurrence set. The other is *diffsets* proposed in [18]. Since there is a trade-off between these two methods that the former is fast for sparse data while the latter is fast for dense data, we developed the *hybrid algorithm* combining them. In some iterations, we make

a decision based of the estimation of their computation time, hence our algorithm can use appropriate one for dense parts and sparse parts of the input.

We also consider the problems of enumerating all frequent sets, and maximal frequent sets, and derive two algorithms LCMfreq and LCMmax from LCM. LCMmax is obtained from LCM by adding the explicit check of maximality. LCMfreq is not merely a LCM without the check of closedness, but also achives substantial speed-up using closed itemset discovery techniques because it enumerates only the representatives of groups of frequent item sets, and generate other frequent item sets from the representatives.

From computer experiments on real and artificial datasets with the previous algorithms, we observed that our algorithms LCMfreq, LCM, and LCMmax significantly outperform the previous algorithms on real world datasets with natural distributions such as BMS-Web-View-1 and BMS-POS datasets in the KDD-CUP 2000 datasets as well as large synthesis datasets such as IBM T10K4D100K. The performance of our algorithms is similar to other algorithms for hard datasets such as Connect and Chess datasets from UCI-Machine Learning Repository, but less significant than MAFIA, however LCM works with small memory rather than other algorithms.

The organization of the paper is as follows. In Section 2, we explain our tree enumeration method for frequent closed item sets and our algorithm LCM. In Section 3, we describe several algorithmic techniques for speeding up and saving memory. Then, Section 4 and 5 give LCMmax and LCMfreq for maximal and all frequent item sets, respectively. Techniques for implementation is described in Section 6, and the results of computational experiments are reported in Section 7. Finally, we conclude in Section 8.

2. Enumerating Frequent Closed Item Sets

In this section, we introduce a parent-child relationship between frequent closed item sets in $\mathcal{C}$, and describe our algorithm LCM for enumeration them.

Recent efficient algorithms for frequent item sets, e.g.,[4, 17, 18], use a tree-shaped search structure for $\mathcal{F}$, called the *set enumeration tree* [4] defined as follows. Let $X = \{x_1, \ldots, x_n\}$ be an itemset as an ordered sequence such that $x_1 < \cdots < x_n$, where the *tail* of X is $tail(X) = x_n \in E$. Let X, Y be item sets. For an index i, $X(i) = X \cap \{1, \ldots, i\}$. X is a *prefix* of Y if $X = Y(i)$ holds for $i = tail(X)$. Then, the parent-child relation $\mathcal{P}$ for the set enumeration tree for $\mathcal{F}$ is define as $X = \mathcal{P}(Y)$ iff $Y = X \cup \{i\}$ for some $i > tail(X)$, or equivalently, $X = Y \backslash \{tail(Y)\}$. Then, the whole search space for $\mathcal{F}$ forms a *prefix tree* (or *trie*) with this edge relation $\mathcal{P}$.

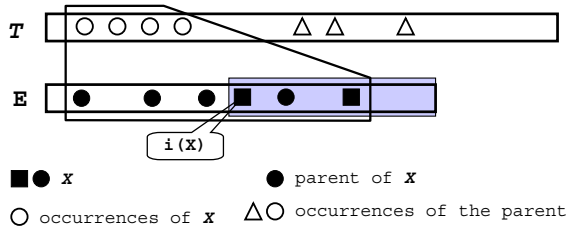


Figure 1: An example of the parent of X: The parent of X is obtained by deleting items larger than $i(X)$ (in the gray area) and take a closure.

Now, we define the parent-child relation $\mathcal{P}$ for closed item sets in $\mathcal{C}$ as follows. For $X \in \mathcal{C}$, we define the *parent* of X by $\mathcal{P}(X) = I(\mathcal{T}(X(i(X) - 1)))$, where $i(X)$ be the minimum item i such that $\mathcal{T}(X) = \mathcal{T}(X(i))$ but $\mathcal{T}(X) \neq \mathcal{T}(X(i-1))$. If Y is the parent of X, we say X is a *child* of Y. Let $\perp = I(\mathcal{T}(\emptyset))$ be the smallest item set in $\mathcal{C}$ called the *root*. For any $X \in \mathcal{C} \setminus \{\perp\}$, its parent $\mathcal{P}(X)$ is always defined and belongs to $\mathcal{C}$. An illustration is given in Fig. 1.

For any $X \in \mathcal{C}$ and its parent Y, the proper inclusion $Y \subset X$ holds since $\mathcal{T}(X(i(X) - 1)) \subset \mathcal{T}(X)$. Thus, the relation $\mathcal{P}$ is acyclic, and its graph representation forms a tree. By traversing the tree in a depth-first manner, we can enumerate all the frequent closed item sets in linear time in the size of the tree, which is equal to the number of the frequent closed item sets in $\mathcal{C}$. In addition, we need not store the tree in memory. Starting from the root $\perp$, we find a child X of the root, and go to X. In the same way, we go to a child of X. When we arrive at a leaf of the tree, we backtrack, and find another child. Repeating this process, we eventually find all frequent closed item set in $\mathcal{C}$.

To find the children of the current frequent closed item set, we use the following lemma. For an item set X and an index i, let $X[i] = X \cup H$ where H is the set of the items $j \in I(\mathcal{T}(X \cup \{i\}))$ satisfying $j \geq i$.

Lemma 1 *X' is a child of $X \in \mathcal{C}$ ($X' \in \mathcal{C}$ and the parent of X' is X) if and only if*
(cond1) $X' = X[i]$ for some $i > i(X)$,
(cond2) $X' = I(\mathcal{T}(X'))$ (X' is a closed item set)
(cond3) X' is frequent

Proof : Suppose that $X' = X[i]$ satisfies the conditions (cond1), (cond2) and (cond3). Then, $X' \in \mathcal{C}$. Since $\mathcal{T}(X(i-1)) = \mathcal{T}(X)$ and $\mathcal{T}(X(i-1) \cup \{i\}) = \mathcal{T}(X')$ holds thus $i(X') = i$ holds. Hence, X' is a child of X. Suppose that X' is a child of X. Then, (cond2) and (cond3) hold. From the definition of $i(X')$, $\mathcal{T}(X(i(X')) \cup \{i(X')\}) = \mathcal{T}(X')$ holds. Hence, $X' = X[i(X')]$ holds. We also have $i(X') > i(X)$ since $\mathcal{T}(X'(i(X')-1)) = \mathcal{T}(X)$. Hence (cond1) holds. ∎

Clearly, $\mathcal{T}(X[i]) = \mathcal{T}(X \cup \{i\})$ holds if (cond2) $I(\mathcal{T}(X[i])) = X[i]$ holds. Note that no child X' of X satisfies $X[i] = X[i'], i \neq i'$, since the minimum item of $X[i] \setminus X$ and $X[i'] \setminus X$ are i and i', respectively. Using this lemma, we construct the following algorithm scheme for, given a closed itemset X, enumerating all descendants in the search tree for closed itemsets.

Algorithm LCM (X : frequent closed item set)
1. **output** X
2. **For** each $i > i(X)$ **do**
3. **If** $X[i]$ is frequent **and** $X[i] = I(\mathcal{T}(X[i]))$ **then** **Call LCM**($X[i]$)
4. **End for**

Theorem 1 *Let $0 < \sigma < 1$ be a minimum support. Algorithm LCM enumerates, given the root closed item set $\perp = I(\mathcal{T}(\emptyset))$, all frequent closed item sets in linear time in the number of frequent closed item sets in $\mathcal{C}$.* ∎

The existing enumeration algorithm for frequent closed item sets are based on backtrack algorithm, which traverse a tree composed of all frequent item sets in $\mathcal{F}$, and skip some item sets by pruning the tree. Since the pruning is not complete, however, these algorithms generate unnecessary frequent item sets. On the other hand, the algorithm in [10] directly generates only closed item sets with the closure operation $I(\mathcal{T}(\cdot))$ as ours, but their method may generate duplicated closed item sets and needs expensive duplicate check.

On the other hand, our algorithm traverses a tree composed only of frequent closed item sets, and each iteration is not as heavy as the previous algorithms. Hence, our algorithm runs fast in practice. If we consider our algorithm as a modification of usual backtracking algorithm, each iteration of our algorithm re-orders the items larger than $i(X)$ such that the items not included in X follow the items included in X. Note that the parent X is not a prefix of $X[i]$ in a recursive call. The check of (cond2) can be considered as a pruning of non-closed item sets.

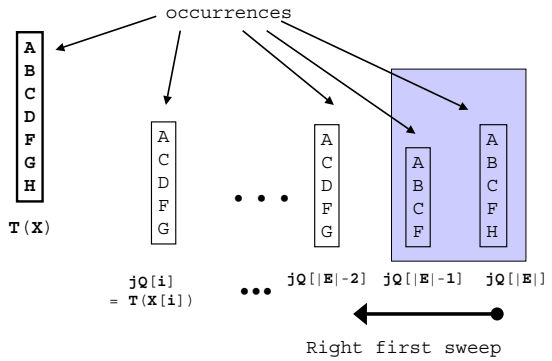


Figure 2: Occurrence deliver and right first sweep: In the figure, $\mathcal{J}[i]$ is written as $JQ[i]$. For each occurrence T of X, occurrence deliver inserts T to $\mathcal{J}[i]$ such that $i \in T$. When the algorithm generates a recursive call respect to $X[|E|-2]$, the recursive calls respect to $X[|E|-1]$ and $X[|E|]$ have been terminated, $\mathcal{J}[|E|-1]$ and $\mathcal{J}[|E|]$ are cleared. The recursive call of $X[|E|-2]$ uses only $\mathcal{J}[|E|-1]$ and $\mathcal{J}[|E|]$, and hence the algorithm re-uses them in the recursive call.

3. Reducing Computation Time

The computation time of LCM described in the previous section is linear in $|\mathcal{C}|$, with a factor depending on $\mathcal{T}(X)$ for each closed item set $X \in \mathcal{C}$. However, this still takes long time if it is implemented in a straightforward way. In this section, we introduce some techniques based on sparse and dense structures of the input data.

Occurrence Deliver. First, We introduce the technique called the *occurrence deliver* for reducing the construction time for $\mathcal{T}(X[i])$, which is needed to check (cond3). This technique is particularly efficient in the case that $|\mathcal{T}(X[i])|$ is much smaller than $|\mathcal{T}(X)|$. In a usual way, $\mathcal{T}(X[i])$ is obtained from $\mathcal{T}(X)$ in $\mathrm{O}(|\mathcal{T}(X)|)$ time by removing all transactions not including i based on the equiality $\mathcal{T}(X[i]) = \mathcal{T}(X \cup \{i\}) = \mathcal{T}(X) \cap \mathcal{T}(\{i\})$ (this method is known as *down-project*). Thus, the total computation for all children takes $|E|$ scans and $\mathrm{O}(|\mathcal{T}(X)| \cdot |E|)$ time.

Instead of this, we build for all $i = i(X), \ldots, |E|$ the occurrence lists $\mathcal{J}[i] \stackrel{\text{def}}{=} \mathcal{T}(X[i])$ simultaneously by scanning the transactions in $\mathcal{T}(X)$ at once as follows. We initialize $\mathcal{J}[i] = \emptyset$ for all $i = i(X), \ldots, |E|$. For each $T \in \mathcal{T}(X)$ and for each $i \in T$ $(i > i(X))$, we insert T to $\mathcal{J}[i]$. See Fig. 2 for explanation, where we write $jQ[i]$ for $\mathcal{J}[i]$. This correctly computes $\mathcal{J}[i]$ for all i in the total time $\mathrm{O}(|\mathcal{T}(X)|)$. Furthermore, we need not make recursive call of **LCM** for $X[i]$ if $\mathcal{T}(X[i]) = \emptyset$ (this is often called *lookahead* [4]). In

our experiments on BMS instances, the occurrence deliver reduces the computation time up to 1/10 in some cases.

Right-first sweep. The occurrence deliver method needs eager computation of the occurrence sets $\mathcal{J}[i] = \mathcal{T}(X[i])$ for all children before expanding one of them. A simple implementation of it may require much memory than the ordinary lazy computation of $\mathcal{T}(X[i])$ as in [17]. However, we can reduce the memory usage using a method called the *right-first sweep* as follows.

Given a parent X, we make the recursive call for $X[i]$ in the decreasing order for each $i = |E|, \ldots, i(X)$ (See Fig. 2). At each call of $X[i]$, we collect the memory allocated before for $\mathcal{J}[i+1], \ldots, \mathcal{J}[|E|]$ and then re-use it for $\mathcal{J}[i]$. After terminating the call for $X[i]$, the memory for $\mathcal{J}[i]$ is released for the future use in $\mathcal{J}[j]$ for $j < i$. Since $|\mathcal{J}[i]| = |\mathcal{T}(X[i])| \leq |\mathcal{T}(\{i\})|$ for any i and X, the total memory $\sum_i |\mathcal{J}[i]|$ is bounded by the input size $||\mathcal{T}|| = \sum_{T \in \mathcal{T}} |T|$, and thus, it is sufficient to allocate the memory for $\mathcal{J}$ at once as a global variable.

Diffsets. In the case that $|\mathcal{T}(X[i])|$ is nearly equal to $|\mathcal{T}(X)|$ we use the *diffset* technique proposed in [18]. The diffset for index i is $\mathcal{DJ}[i] = \mathcal{T}(X) \backslash \mathcal{T}(X[i])$, where $\mathcal{T}(X[i]) = \mathcal{T}(X \cup \{i\})$. Then, the frequency of $X[i]$ is obtained by $|\mathcal{T}(X[i])| = |\mathcal{T}(X)| - |\mathcal{DJ}[i]|$. When we generate a recursive call respect to $X[i]$, we update $\mathcal{DJ}[j], j > i$ by setting $\mathcal{DJ}[j]$ to be $\mathcal{DJ}[j] \setminus \mathcal{DJ}[i]$ in time $\mathrm{O}(\sum_{i > i(X), X[i] \in \mathcal{F}}((|\mathcal{T}(X)| - |\mathcal{T}(X[i])|)$. Diffsets are needed for only i such that $X[i]$ is frequent. By diffsets, the computation time for instances such as connect, chess, pumsb are reduced to 1/100, where $|\mathcal{T}(X[i])|$ is as large as $|\mathcal{T}(X)|$.

Hybrid Computation. As we saw in the preceding subsections, our occurrence deliver is fast when $|\mathcal{T}(X[i])|$ is much smaller than $|\mathcal{T}(X)|$ while the diffset of [18] is fast when $|\mathcal{T}(X[i])|$ is nearly close to $|\mathcal{T}(X)|$. Therefore, our LCM dinamically decides which of occurrence deliver and diffsets we will use. To do this, we compare two quantities on X:

$A(X) = \sum_i |\mathcal{T}(X \cup \{i\})|$ and

$B(X) = \sum_{i: X \cup \{i\} \in \mathcal{F}} (|\mathcal{T}(X)| - |\mathcal{T}(X \cup \{i\})|)$.

For some fixed constant $\alpha > 1$, we decide to use the occurrence deliver if $A(X) < \alpha B(X)$ and the diffset otherwise. We make this decision only at the child iterations of the root set $\perp$ since this decision takes much time. Empirically, restricting the range $i \in \{1, \ldots, |E|\}$ of the the index i in $A(X)$ and $B(X)$ to $i \in \{i(X)+1, \ldots, |E|\}$ results significant speed-up. By experiments on BMS instances, we observe that the hybrid technique reduces the computation time up to 1/3. The hybrid technique is also useful in reducing the memory space in diffset as follows. Although the memory $B(X)$ used by diffsets is not bounded by the input size $||\mathcal{T}||$ in the worst case, it is ensured in hybrid that $B(X)$ does not exceed $A(X) \leq ||\mathcal{T}||$ because the diffset is chosen only when $A(X) \geq \alpha B(X)$.

Checking the closedness in occrrence deliver. Another key is to efficiently check the closedness $X[i] = I(\mathcal{T}(X[i]))$ (cond 2). The straightforward computation of the closure $I(\mathcal{T}(X[i]))$ takes much time since it requires the access to the whole sets $\mathcal{T}(X[j]), j < i$ and i is usually as large as $|E|$.

By definition, (cond 2) is violated iff there exists some $j \in \{1, \ldots, i-1\}$ such that $j \in T$ for every $T \in \mathcal{T}(X \cup \{i\})$. We first choose a transaction $T^*(\cup\{i\}) \in \mathcal{T}(X \cup \{i\})$ of minimum size, and tests if $j \in T$ for increasing $j \in T^*(\cup\{i\})$. This results $\mathrm{O}(\sum_{j \in T^*(X \cup \{i\})} m(X[i], j))$ time algorithm, where $m(X', j)$ is the maximum index m such that all of the first m transactions of $\mathcal{T}(X')$ include j, which is much faster than the straightforward algorithm with $\mathrm{O}(\sum_{j<i} |\mathcal{T}(X \cup \{i\} \cup \{j\}])|)$ time.

In fact, the efficient check requires the adjacency matrix (sometime called *bitmap*) representing the inclusion relationship between items and transactions. However, the adjacency matrix requires $\mathrm{O}(|\mathcal{T}| \times |E|)$ memory, which is quite hard to store for large instances. Hence, we make columns of adjacency matrix for only transactions of size larger than $(\sum_{T \in \mathcal{T}} |T|)/\delta$. Here δ is a constant. This uses at most $\mathrm{O}(\delta \times \sum_{T \in \mathcal{T}} |T|)$, linear in the input size.

Checking the closedness in diffsets. In the case that $|\mathcal{T}(X[i])|$ is nearly equal to $|\mathcal{T}(X)|$, the above check is not done in short time. In this case, we keep diffset $\mathcal{DJ}[j]$ for all $j < i, i \notin X$ such that $X[i]$ is frequent. To maintain $\mathcal{DJ}$ for all i is a heavy task, thus we discard unnecessary $\mathcal{DJ}$'s as follows. If $\mathcal{T}(X \cup \{j\})$ includes an item included in no $\mathcal{T}(X[i']), i' > i(X)$, then for any descendant X' of X, $j \notin I(\mathcal{T}(X'[j']))$ for any $j' > i(X')$. Hence, we no longer have to keep $\mathcal{DJ}[j]$ for such j. Let $NC(X)$ be the set of items j such that $X[j]$ is frequent and any item of $\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})$ is included in some $\mathcal{T}(X[j']), j' > i(X)$. Then, the computation time for checking (cond2) is written as $\mathrm{O}(\sum_{j \in NC(X), j<i} |\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})|)$. By checking (cond2) in these ways, the computation time for checking (cond2) is reduced from 1/10 to 1/100.

Detailed Algorithm. We present below the description of the algorithm **LCM**, which recursively computes $(X, \mathcal{T}(X), i(X))$, simultaneously.

global: $\mathcal{J}, \mathcal{DJ}$ /* Global sets of lists */

Algorithm LCM()
1. $X := I(\mathcal{T}(\emptyset))$ /* The root $\perp$ */
2. **For** $i := 1$ **to** $|E|$
3. **If** $X[i]$ satisfies (cond2) and (cond3) **then**
   Call LCM_Iter($X[i], \mathcal{T}(X[i]), i$) or
   Call LCMd_Iter2($X[i], \mathcal{T}(X[i]), i, \mathcal{DJ}$)
   based on the decision criteria
4. **End for**

LCM_Iter($X, \mathcal{T}(X), i(X)$) /* occurrence deliver */
1. **output** X
2. **For each** $T \in \mathcal{T}(X)$
   For each $j \in T, j > i(X)$, insert t to $\mathcal{J}[j]$
4. **For** each $j, \mathcal{J}[j] \neq \emptyset$ in the decreasing order
5. **If** $|\mathcal{J}[j]| \geq \alpha$ and (cond2) holds **then**
   LCM_Iter($\mathcal{T}(\mathcal{J}[j]), \mathcal{J}[j], j$)
6. Delete $\mathcal{J}[j]$
7. **End for**

LCM_Iter2($X, \mathcal{T}(X), i(X), \mathcal{DJ}$) /* diffset */
1. **output** X
2. **For** each $i, X[i]$ is frequent
3. **If** $X[i]$ satisfies (cond2) **then**
4. **For** each $j, X[i] \cup \{j\}$ is frequent,
   $\mathcal{DJ}'[j] := \mathcal{DJ}[j] \setminus \mathcal{DJ}[i]$
5. **LCM_Iter2**($\mathcal{T}(\mathcal{J}[j]), \mathcal{J}[j], j, \mathcal{DJ}'$)
6. **End if**
7. **End for**

Theorem 2 *Algorithm LCM enumerates all frequent closed item sets in* $O(\sum_{j>i(X)} |\mathcal{T}(X[j])| + \sum_{j>i(X), X[j]\in\mathcal{F}} \sum_{j'\in T^*(X)} m(X[j], j'))$ *time, or* $O(\sum_{i>i(X), X[i]\in\mathcal{F}}((|\mathcal{T}(X)| - |\mathcal{T}(X[i])|) + \sum_{j\in NC(X), j<i} |\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})|))$ *time for each frequent closed item set* X*, with memory linear to the input size.* ∎

4. Enumerating Maximal Frequent Sets

In this section, we explain an enumeration algorithm of maximal frequent sets with the use of frequent closed item set enumeration. The main idea is very simple. Since any maximal frequent item set is a frequent closed item set, we enumerate frequent closed item sets and output only those being maximal frequent sets. For a frequent closed item set X, X is a maximal frequent set if and only if $X \cup \{i\}$ is infrequent for any $i \notin X$. By adding this check to LCM, we obtain LCMmax.

This modification does not increase the memory complexity but increase the computation time. In the case of occurrence deliver, we generate $\mathcal{T}(X \cup \{j\})$ for all j in the same way as the occurrence deliver, and check the maximality. This takes $O(\sum_{j<i(X)} |\mathcal{T}(X \cup \{j\})|)$ time. In the case of difference update, we do not discard diffsets unnecessary for closed item set enumeration. We keep diffsets $\mathcal{DJ}$ for all j such that $X \cup \{j\}$ is frequent. To update and maintain this, we spend $O(\sum_{j, X\cup\{j\}\in\mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X \cup \{j\})|)$ time. Note that we are not in need of check the maximality if X has a child.

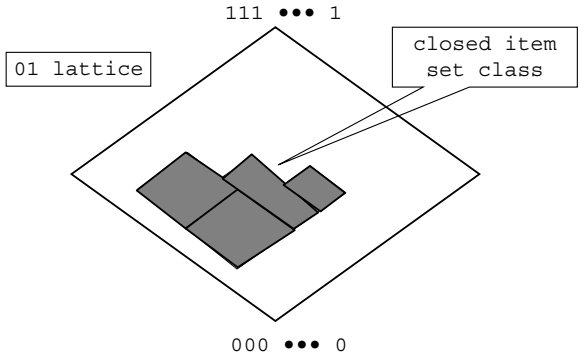


Figure 3: Hypercube decomposition: **LCMfreq** decomposes a closed item set class into several sublattices (gray rectangles).

Theorem 3 *Algorithm LCMmax enumerates all maximal frequent item sets in* $O(\sum_i |\mathcal{T}(X \cup \{i\})|)$ *time, or* $O(\sum_{i, X\cup\{i\}\in\mathcal{F}}((|\mathcal{T}(X)| - |\mathcal{T}(X\cup\{i\})|))$ *time for each frequent closed item set* X*, with memory linear in the input size.* ∎

5. Enumerating Frequent Sets

In this section, we describe an enumeration algorithm for frequent item sets. The key idea of our algorithm is that we classify the frequent item sets into groups and enumerate the representative of each group. Each group is composed of frequent item sets included in the class of a closed item set. This idea is based on the following lemma.

Lemma 2 *Suppose that frequent item sets* X *and* $S \supset X$ *satisfy* $\mathcal{T}(X) = \mathcal{T}(S)$*. Then, for any item set* X' *including* X*,* $\mathcal{T}(X') = \mathcal{T}(X' \cup S)$*.* ∎

Particularly, $\mathcal{T}(X') = \mathcal{T}(R)$ holds for any $X' \subseteq R \subseteq X' \cup S$, hence all R are included in the same class of a closed item set. Hence, any frequent item set X'

is generated from $X' \setminus (S \setminus X)$. We call $X' \setminus (S \setminus X)$ *representative*.

Let us consider a backtracking algorithm finding frequent item sets which adds items one by one in lexicographical order. Suppose that we currently have a frequent item set X, and find another frequent item set $X \cup \{i\}$. Let $S = X[i]$. Then, according to the above lemma, we can observe that for any frequent item set X' including X and not intersecting $S \setminus X$, any item set including X' and included in $X' \cup S$ is also frequent. Conversely, any frequent item set including X is generated from X' not intersecting $S \setminus X$. Hence, we enumerate only representatives including X and not intersecting $S \setminus X$, and generate other frequent item sets by adding each subset of $S \setminus X$. This method can be considered that we "decompose" classes of closed item sets into several sublattices (hypercubes) each of whose maximal and minimal elements are S and X', respectively (see Fig. 3). We call this technique *hypercube decomposition*.

Suppose that we are currently operating a representative X' including X, and going to generate a recursive call respect to $X' \cup \{j\}$. Then, if $(X'[i] \setminus X') \setminus S \neq \emptyset$, X' and $S \cup (X'[i] \setminus X')$ satisfies the condition of Lemma 2. Hence, we add $X'[i] \setminus X'$ to S.

We describe LCMfreq as follows.

Algorithm LCMfreq (X : representative,
S : item set, i : item)
1. **Output** all item sets $R, X \subseteq R \subseteq X \cup S$
2. **For each** $j > i, j \notin X \cup S$
3. **If** $X \cup \{j\}$ is frequent **then**
Call LCMfreq ($X \cup \{j\}, S \cup (X[j] \setminus (X \cup \{j\})), j$)
4. **End for**

For some synthetic instances such that frequent closed item sets are fewer than frequent item sets, the average size of S is up to 5. In these cases, the algorithm finds $2^{|S|} = 32$ frequent item sets at once, hence the computation time is reduced much by the improvement.

To check the frequency of all $X \cup \{j\}$, we can use occurrence deliver and diffsets used for LCM. LCMfreq does not require the check of (cond2), hence The computation time of each iteration is $O(\sum_{j>i(X)} |\mathcal{T}(X[j])|)$ time for occurrence deliver, and $O(\sum_{j>i(X), X[j] \in \mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|)$ for diffsets. Since the computation time change, we use another estimators for hybrid. In almost all cases, if once $\sum_{j>i(X), X[j] \in \mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|$ becomes smaller than $\sum_{j>i(X)} |\mathcal{T}(X[j])|$, the condition holds in any iteration generated by a recursive call. Hence, the algorithm first starts with occurrence deliver, and compares them in each iteration. If $\sum_{j>i(X), X[j] \in \mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|$ becomes smaller, then we change to diffsets. Note that these estimators can computed in short time by using the result of occurrence deliver.

Theorem 4 *LCMfreq enumerates all frequent sets of $\mathcal{F}$ in $O(\sum_{j>i(X)} |\mathcal{T}(X[j])|)$ time or $O(\sum_{j>i(X), X[j] \in \mathcal{F}} |\mathcal{T}(X) \setminus \mathcal{T}(X[j])|)$ time for each frequent set X, within $O(\sum_{T \in \mathcal{T}} |T|)$ space.* ∎

Particularly, LCMfreq requires one integer for each item of any transaction, which is required to store the input data. Other memory LCMfreq uses is bounded by $O(|\mathcal{T}| + |E|)$.

Experimentally, an iteration of LCMfreq inputting frequent set X takes $O(|\mathcal{T}(X)| + |X|)$ or O((size of diffset) + $|X|$) steps in average. In some sense, this is optimal since we have to take $O(|X|)$ time to output, and $O(|\mathcal{T}(X)|)$ time or O((size of diffset)) time to check the frequency of X.

6. Implementation

In this section, we explain our implementation. First, we explain the data structure of our algorithm. A transaction T of input data is stored by an array with length $|T|$. Each cell of the array stores the index of an item of T. For example, $t = \{4, 2, 7\}$ is stored in an array with 3 cells, $[2, 4, 7]$. We sort the elements of the array so that we can take $\{i, ..., |E|\} \cap T$ in linear time of $\{i, ..., |E|\} \cap T$. $\mathcal{J}$ is also stored in arrays in the same way. We are not in need of doubly linked lists or binary trees, which take much time to be operated.

To reduce the practical computation time, we sort the transactions by their sizes, and items by the number of transactions including them. Experimentally, this reduces $\sum_{j>i(X)} |\mathcal{T}(X \cup \{j\})|$. In some cases, the computation time has been reduced by a factor of 1/3.

7. Computational Experiments

To examine the practical efficiency of our algorithms, we run the experiments on the real and synthetic datasets, which are made available on FIMI'03 site. In the following, we will report the results of the experiments.

Table 1: The datasets. AvTrSz means the average transaction size

Dataset	#items	#Trans	AvTrSz	#FI	#FCI	#MFI	Minsup (%)
BMS-Web-View1	497	59,602	2.51	3.9K–NA	3.9K–1241K	2.1K–129.4K	0.1–0.01
BMS-Web-View2	3,340	77,512	4.62	24K–9897K	23K–755K	3.9K–118K	0.1–0.01
BMS-POS	1,657	517,255	6.5	122K–33400K	122K–21885K	30K–4280K	0.1–0.01
T10I4D100K	1,000	100,000	10.0	15K–335K	14K–229K	7.9K–114K	0.15–0.025
T40I10D100K	1,000	100,000	39.6	-	-	-	2–0.5
pumsb	7,117	49,046	74.0	-	-	-	95–60
pumsb_star	7,117	49,046	50.0	-	-	-	50–10
mushroom	120	8,124	23.0	-	-	-	20–0.1
connect	130	67,577	43.0	-	-	-	95–40
chess	76	3196	37.0	-	-	-	90–30

7.1 Datasets and Methods

We implemented our algorithms our three algorithms LCMfreq (LCMfreq), LCM (LCM), LCMmax (LCMmax) in C and compiled with gcc3.2.

The algorithms were tested on the datasets shown in Table 1. available from the FIMI'03 homepage[1], which include: T10I4D100K, T40I10D100K from IBM Almaden Quest research group; chess, connect, mushroom, pumsb, pumsb_star from UCI ML repository[2] and PUMSB; BMS-WebView-1, BMS-WebView-2, BMS-POS from KDD-CUP 2000[3].

We compare our algorithms LCMfreq, LCM, LCMmax with the following frequent item set mining algorithms: Implementations of Fp-growth [7], Eclat [17], Apriori [1, 2] by Bart Goethals [4]; We also compare the LCM algorithms with the implementation of Mafia [6], a fast maximal frequent pattern miner, by University of Cornell's Database group [5]. This versions of mafia with frequent item sets, frequent closed item sets, and maximal frequent item sets options are denoted by mafia-fi, mafia-fci, mafia-mfi, respectively. Although we have also planned to make the performance comparison with Charm, the state-of-the-art frequent closed item set miner, we gave up the comparison in this time due to the time constraint.

All experiments were run on a PC with the configuration of Pen4 2.8GHz, 1GB memory, and RPM 7200 hard disk of 180GB. In the experiments, LCMfreq, LCM and LCMmax use at most 123MB, 300MB, and 300MB of memory, resp. Note that LCM and LCMmax can save the memory use by decreasing δ.

[1] `http://fimi.cs.helsinki.fi/testdata.html`

[2] `http://www.ics.uci.edu/ mlearn/MLRepository.html`

[3] `http://www.ecn.purdue.edu/KDDCUP/`

[4] `http://www.cs.helsinki.fi/u/goethals/software/`

[5] University of Cornell Database group, Himalaya Data Mining Tools, `http://himalaya-tools.sourceforge.net/`

7.2 Results

Figures 6 through Figure 12 show the running time with varying minimum supports for the seven algorithms, namely LCMfreq, LCM, LCMmax, FP-growth, eclat, apriori, mafia-mfi on the nine datasets described in the previous subsection. In the following, we call all, maximal, closed frequent item set mining simply by all, maximal, closed.

Results on Synthetic Data

Figure 4 shows the running time with minimum support ranging from 0.15% to 0.025% on IBM-Artificial T10I4D100K datasets. From this plot, we see that most algorithms run within around a few 10 minutes and the behaviors are quite similar when minimum support increases. In Figure 4, All of LCMmax, LCM, and LCMfreq are twice faster than FP-growth on IBM T10I4D100K dataset. On the other hand, Mafia-mfi, Mafia-fci, and Mafia-fi are slower than every other algorithms. In Figure 5, Mafia-mfi is fastest for maximal, and LCMfreq is fastest for all, for minimum support less than 1% on IBM T10I4D100K dataset.

Results on KDD-CUP datasets

Figures 6 through Figure 8 show the running time with range of minimum supports from ranging from 0.1% to 0.01% on three real world datasets BMS-WebView-1, BMS-WebView-2, BMS-POS datasets. In the figure, we can observe that LCM algorithms outperform others in almost cases, especially for lower minimum support. In particular, LCM was best among seven algorithms in a wide range of minimum support from 0.1% to 0.01% on all datasets.

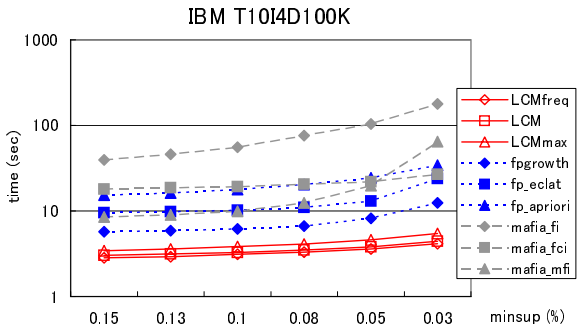


Figure 4: Running time of the algorithms on IBM-Artificial T10I40D100K

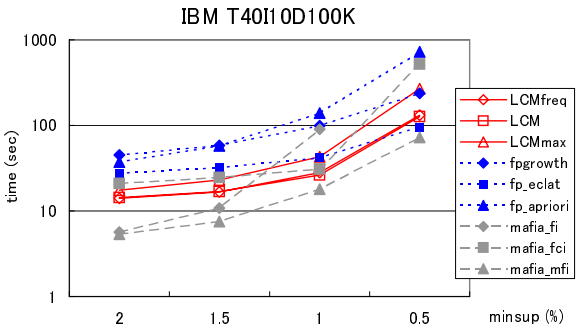


Figure 5: Running time of the algorithms on IBM-Artificial T40I10D100K

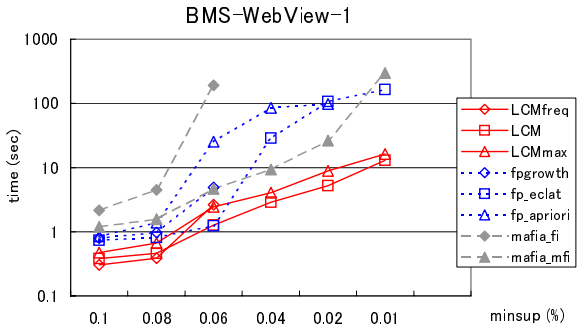


Figure 6: Running time of the algorithms on BMS-WebView-1

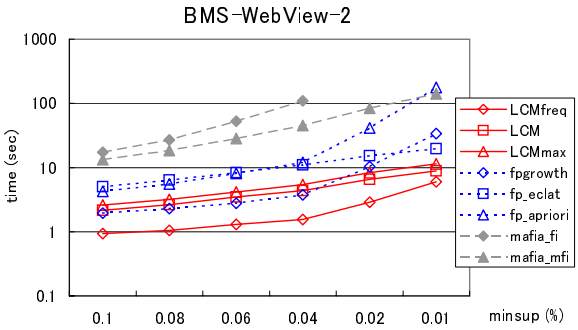


Figure 7: Running time of the algorithms on BMS-WebView-2

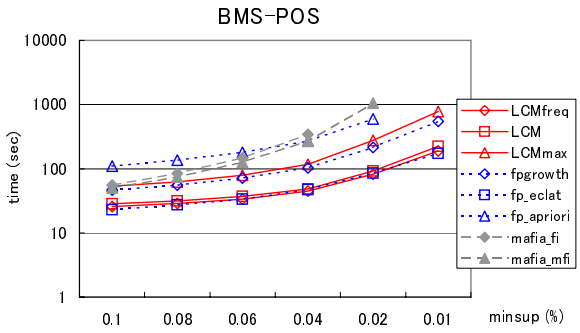


Figure 8: Running time of the algorithms on BMS-POS

For higher minimum support ranging from 0.1% to 0.06%, the performances of all algorithms are similar, and LCM families have slightly better performance. For lower minimum support ranging from 0.04% to 0.01%, Eclat and Apriori are much slower than every other algorithms. LCM outperforms others. Some frequent item set miners such as Mafia-fi, and Mafia-fci runs out of 1GB of main memory for these minimum supports on BMS-WebView-1, BMS-WebView-2, BMS-POS datasets. LCMfreq works quite well for higher minimum support, but takes more than 30 minutes for minimum support above 0.04% on BMS-Web-View-1. In these cases, the number of frequent item sets is quite large, over 100,000,000,000. Interestingly, Mafia-mfi's performance is stable in a wide range of minimum support from 0.1% to 0.01%.

In summary, LCM family algorithms significantly perform well on real world datasets BMS-WebView-1, BMS-WebView-2, BMS-POS datasets.

Results on UCI-ML repository and PUMSB datasets

Figures 9 through Figure 12 show the running time on middle sized data sets pumsb and pumsb*, kosarak and small sized datasets connect, chess, and mushroom. These datasets taken from machine learning domains are small but hard datasets for frequent pattern mining task since they have many frequent patterns even with high minimum supports, e.g., from 50% to 90%. These datasets are originally build for classification task and have slightly different characteristics than large business datasets such

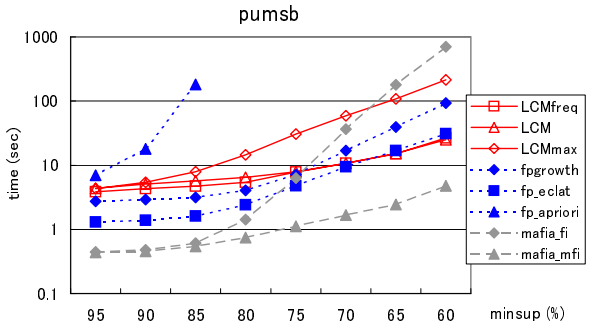


Figure 9: Running time of the algorithms on pumsb

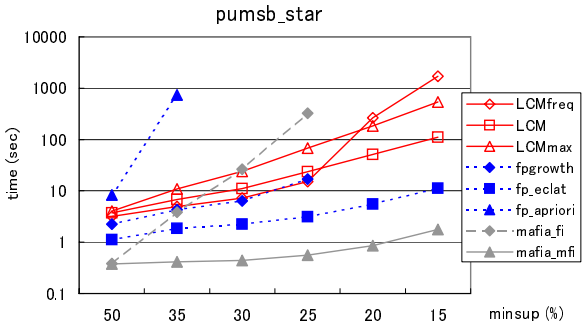


Figure 10: Running time of the algorithms on pumsb*

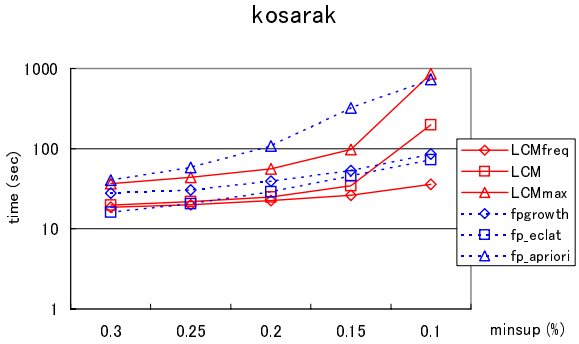


Figure 11: Running time of the algorithms on kosarak

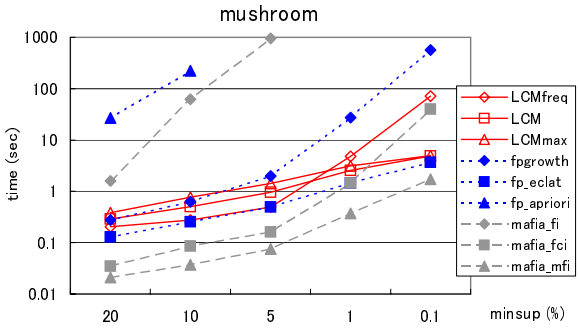


Figure 12: Running time of the algorithms on mushroom

as BMS-Web-View-1 or BMS-POS.

In these figures, we see that Mafia-mfi constantly outperforms every other maximal frequent item sets mining algorithms for wide range of minimum supports except on pumsb*. On the other hand, Apriori is much slower than other algorithm. On the mining of all frequent item sets, LCMfreq is faster than the others algorithms. On the mining of frequent closed item sets, there seems to be no consistent tendency on the performance results. However, LCM does not store the obtained solutions in the memory, while the other algorithms do. Thus, in the sense of memory-saving, LCM has an advantage.

8 Conclusion

In this paper, we present an efficient algorithm LCM for mining frequent closed item sets based on parent-child relationship defined on frequent closed item sets. This technique is taken from the algorithms for enumerating maximal bipartite cliques [14, 15] based on reverse search [3]. In theory, we demonstrate that LCM exactly enumerates the set of frequent closed item sets within polynomial time per closed item set in the total input size. In practice, we show by experiments that our algorithms run fast on several real world datasets such as BMS-WebView-1. We also showed variants LCMfreq and LCMmax of LCM for computing maximal and all frequent item sets. LCMfreq uses new schemes hybrid and hypercube decomposition, and the schemes work well for many problems.

Acknowledgement

We gratefully thank to Prof. Ken Satoh of National Institute of Informatics. This research had been supported by group research fund of National Institute of Informatics, JAPAN.

References

[1] R. Agrawal and R. Srikant, "Fast Algorithms for Mining Association Rules in Large Databases," In *Proc. VLDB '94*, pp. 487–499, 1994.

[2] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen and A. I. Verkamo, "Fast Discovery of Associa-

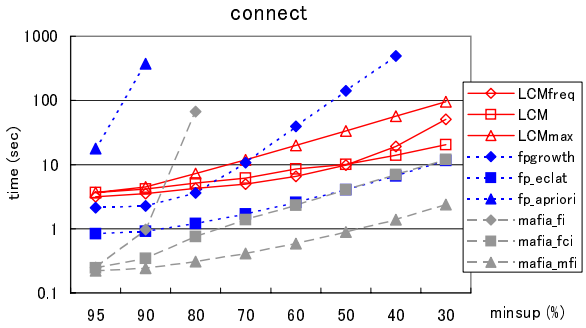


Figure 13: Running time of the algorithms on connect

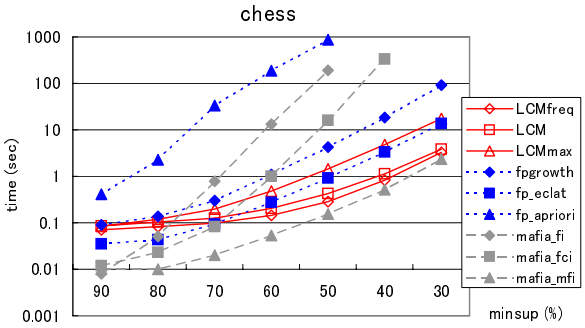


Figure 14: Running time of the algorithms on chess

tion Rules," In *Advances in Knowledge Discovery and Data Mining,* MIT Press, pp. 307–328, 1996.

[3] D. Avis and K. Fukuda, "Reverse Search for Enumeration," *Discrete Applied Mathematics,* Vol. 65, pp. 21–46, 1996.

[4] R. J. Bayardo Jr., *Efficiently Mining Long Patterns from Databases*, In Proc. SIGMOD'98, pp. 85–93, 1998.

[5] E. Boros, V. Gurvich, L. Khachiyan, and K. Makino, "On the Complexity of Generating Maximal Frequent and Minimal Infrequent Sets," In *Proc. STACS 2002*, pp. 133-141, 2002.

[6] D. Burdick, M. Calimlim, J. Gehrke, "MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases," In *Proc. ICDE 2001*, pp. 443-452, 2001.

[7] J. Han, J. Pei, Y. Yin, "Mining Frequent Patterns without Candidate Generation," In *Proc. SIGMOD'00*, pp. 1-12, 2000

[8] R. Kohavi, C. E. Brodley, B. Frasca, L. Mason and Z. Zheng, "KDD-Cup 2000 Organizers' Report: Peeling the Onion," *SIGKDD Explorations,* 2(2), pp. 86-98, 2000.

[9] H. Mannila, H. Toivonen, "Multiple Uses of Frequent Sets and Condensed Representations," In *Proc. KDD'96*, pp. 189–194, 1996.

[10] N. Pasquier, Y. Bastide, R. Taouil, L. Lakhal, Discovering frequent closed itemsets for association rules, In *Proc. ICDT'99*, pp. 398-416, 1999.

[11] J. Pei, J. Han, R. Mao, "CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets," *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery 2000*, pp. 21-30, 2000.

[12] B. Possas, N. Ziviani, W. Meira Jr., B. A. Ribeiro-Neto, "Set-based model: a new approach for information retrieval," In *Proc. SIGIR'02*, pp. 230-237, 2002.

[13] S. Tsukiyama, M. Ide, H. Ariyoshi and I. Shirakawa, "A New Algorithm for Generating All the Maximum Independent Sets," *SIAM Journal on Computing,* Vol. 6, pp. 505–517, 1977.

[14] Takeaki Uno, "A Practical Fast Algorithm for Enumerating Cliques in Huge Bipartite Graphs and Its Implementation," 89th Special Interest Group of Algorithms, Information Processing Society Japan, 2003,

[15] Takeaki Uno, "Fast Algorithms for Enumerating Cliques in Huge Graphs," Research Group of Computation, IEICE, Kyoto University, pp.55-62, 2003

[16] Takeaki Uno, "A New Approach for Speeding Up Enumeration Algorithms," In *Proc. ISAAC'98*, pp. 287–296, 1998.

[17] M. J. Zaki, "Scalable algorithms for association mining," *Knowledge and Data Engineering*, 12(2), pp. 372–390, 2000.

[18] M. J. Zaki, C. Hsiao, "CHARM: An Efficient Algorithm for Closed Itemset Mining," In *Proc. SDM'02*, SIAM, pp. 457-473, 2002.

[19] Z. Zheng, R. Kohavi and L. Mason, "Real World Performance of Association Rule Algorithms," In *Proc. SIGKDD-01*, pp. 401-406, 2001.

MAFIA: A Performance Study of Mining Maximal Frequent Itemsets

Doug Burdick *
University of Wisconsin-Madison
who0ps99@cs.wisc.edu

Manuel Calimlim
Cornell University
calimlim@cs.cornell.edu

Jason Flannick †
Stanford University
flannick@cs.stanford.edu

Johannes Gehrke
Cornell University
johannes@cs.cornell.edu

Tomi Yiu
Cornell University
ty42@cornell.edu

Abstract

We present a performance study of the MAFIA algorithm for mining maximal frequent itemsets from a transactional database. In a thorough experimental analysis, we isolate the effects of individual components of MAFIA, including search space pruning techniques and adaptive compression. We also compare our performance with previous work by running tests on very different types of datasets. Our experiments show that MAFIA performs best when mining long itemsets and outperforms other algorithms on dense data by a factor of three to thirty.

1 Introduction

MAFIA uses a vertical bitmap representation for support counting and effective pruning mechanisms for searching the itemset lattice [6]. The algorithm is designed to mine maximal frequent itemsets (*MFI*), but by changing some of the pruning tools, MAFIA can also generate all frequent itemsets (*FI*) and closed frequent itemsets (*FCI*).

MAFIA assumes that the entire database (and all data structures used for the algorithm) completely fit into main memory. Since all algorithms for finding association rules, including algorithms that work with disk-resident databases, are CPU-bound, we believe that our study sheds light on some important performance bottlenecks.

In a thorough experimental evaluation, we first quantify the effect of each individual pruning component on the performance of MAFIA. Because of our strong pruning mechanisms, MAFIA performs best on dense datasets where large subtrees can be removed from the search space. On shallow datasets, MAFIA is competitive though not always the fastest algorithm. On dense datasets, our results indicate that MAFIA outperforms other algorithms by a factor of three to thirty.

*Research for this paper done while at Cornell University

†Research for this paper done while at Cornell University

2 Search Space Pruning

MAFIA uses the lexicographic subset tree originally presented by Rymon [9] and adopted by both Agarwal [3] and Bayardo [4]. The itemset identifying each node will be referred to as the node's *head*, while possible extensions of the node are called the *tail*. In a pure depth-first traversal of the tree, the tail contains all items lexicographically larger than any element of the head. With a dynamic reordering scheme, the tail contains only the frequent extensions of the current node. Notice that all items that can appear in a subtree are contained in the subtree root's head union tail (HUT), a set formed by combining all elements of the head and tail.

In the simplest itemset traversal, we traverse the lexicographic tree in pure depth-first order. At each node n, each element in the node's tail is generated and counted as a 1-extension. If the support of $\{$n's head$\} \cup \{$1-extension$\}$ is less than $minSup$, then we can stop by the Apriori principle, since any itemset from that possible 1-extension would have an infrequent subset.

For each candidate itemset, we need to check if a superset of the candidate itemset is already in the **MFI**. If no superset exists, then we add the candidate itemset to the **MFI**. It is important to note that with the depth-first traversal, itemsets already inserted into the **MFI** will be lexicographically ordered earlier.

2.1 Parent Equivalence Pruning (PEP)

One method of pruning involves comparing the transaction sets of each parent/child pair. Let x be a node n's head and y be an element in n's tail. If $t(x) \subseteq t(y)$, then any transaction containing x also contains y. Since we only

want the maximal frequent itemsets, it is not necessary to count itemsets containing x and not y. Therefore, we can move item y from the node's tail to the node's head.

2.2 FHUT

Another type of pruning is superset pruning. We observe that at node n, the largest possible frequent itemset contained in the subtree rooted at n is n's HUT (head union tail) as observed by Bayardo [4]. If n's HUT is discovered to be frequent, we never have to explore any subsets of the HUT and thus can prune out the entire subtree rooted at node n. We refer to this method of pruning as *FHUT* (Frequent Head Union Tail) pruning.

2.3 HUTMFI

There are two methods for determining whether an itemset x is frequent: direct counting of the support of x, and checking if a superset of x has already been declared frequent; FHUT uses the former method. The latter approach determines if a superset of the HUT is in the *MFI*. If a superset does exist, then the HUT must be frequent and the subtree rooted at the node corresponding to x can be pruned away. We call this type of superset pruning *HUTMFI*.

2.4 Dynamic Reordering

The benefit of dynamically reordering the children of each node based on support instead of following the lexicographic order is significant. An algorithm that trims the tail to only frequent extensions at a higher level will save a lot of computation. The order of the tail elements is also an important consideration. Ordering the tail elements by increasing support will keep the search space as small as possible. This heuristic was first used by Bayardo [4].

In Section 5.3.1, we quantify the effects of the algorithmic components by analyzing different combinations of pruning mechanisms.

3 MAFIA Extensions

MAFIA is designed and optimized for mining maximal frequent itemsets, but the general framework can be used to mine all frequent itemsets and closed frequent itemsets.

The algorithm can easily be extended to mine all frequent itemsets. The main changes required are suppressing any pruning tools (PEP, FHUT, HUTMFI) and adding all frequent nodes in the itemset lattice to the set *FI* without any superset checking. Itemsets are counted using the same techniques as for the regular MAFIA algorithm.

MAFIA can also be used to mine closed frequent itemsets. An itemset is *closed* if there are no supersets with the same support. PEP is the only type of pruning used when mining for frequent closed itemsets (*FCI*). Recall from Section 2.1 that PEP moves all extensions with the same support from the tail to the head of each node. Any items remaining in the tail must have a lower support and thus are different closed itemsets. Note that we must still check for supersets in the previously discovered **FCI**.

4 Optimizations

4.1 Effective MFI Superset Checking

In order to enumerate the exact set of maximally frequent itemsets, before adding any itemset to the *MFI* we must check the entire *MFI* to ensure that no superset of the itemset has already been found. This check is done often, and significant performance improvements can be realized if it is done efficiently. To ensure this, we adopt the *progressive focusing* technique introduced by Gouda and Zaki [7].

The basic idea is that while the entire *MFI* may be large, at any given node only a fraction of the *MFI* are possible supersets of the itemset at the node. We therefore maintain for each node a *LMFI* (Local MFI), which is the subset of the *MFI* that contains supersets of the current node's itemset. For more details on the *LMFI* concept, please see the paper by Gouda and Zaki [7].

4.2 Support Counting and Bitmap Compression

MAFIA uses a vertical bitmap representation for the database [6]. In a vertical bitmap, there is one bit for each transaction in the database. If item i appears in transaction j, then bit j of the bitmap for item i is set to one; otherwise, the bit is set to zero. This naturally extends to itemsets. Generation of new itemset bitmaps involves bitwise-ANDing *bitmap(*X*)* with a bitmap for 1-itemset Y and storing the result in *bitmap (*$X \cup Y$*)*. For each byte in *bitmap(*$X \cup Y$*)*, the number of 1's in the byte is determined using a pre-computed table. Summing these lookups gives the support of $(X \cup Y)$.

4.3 Compression and Projected Bitmaps

The weakness of a vertical representation is the sparseness of the bitmaps especially at the lower support levels. Since every transaction has a bit in vertical bitmaps, there are many zeros because both the absence and presence of the itemset in a transaction need to be represented. However, note that we only need information about transactions containing the itemset X to count the support of the subtree rooted at node N. So, conceptually we can remove the bit for transaction T from X if T does not contain X. This is

Dataset	T	I	ATL
T10I4D100K	100,000	1,000	10
T40I10D100K	100,000	1,000	40
BMS-POS	515,597	1,657	6.53
BMS-WebView-1	59,602	497	2.51
BMS-WebView-2	3,340	161	4.62
chess	3196	76	37
connect4	67,557	130	43
pumsb	49,046	7,117	74
pumsb-star	49,046	7,117	50

T = Numbers of transactions
I = Numbers of items
ATL = Average transaction length

Figure 1. Dataset Statistics

a form of lossless compression on the vertical bitmaps to speed up calculations.

4.3.1 Adaptive Compression

Determining when to compress the bitmaps is not as simple as it first appears. Each 1-extension bitmap in the tail of the node N must be projected relative to the itemset X, and the cost for projection may outweigh the benefits of using the compressed bitmaps. The best approach is to compress only when we know that the savings from using the compressed bitmaps outweigh the cost of projection.

We use an adaptive approach to determine when to apply compression. At each node, we estimate both the cost of compression and the benefits of using the compressed bitmaps instead of the full bitmaps. When the benefits outweight the costs, compression is chosen for that node and the subtree rooted at that node.

5 Experimental Results

5.1 Datasets

To test MAFIA, we used three different types of data. The first group of datasets is sparse; the frequent itemset patterns are short and thus nodes in the itemset tree will have small tails and few branches. We first used artificial datasets that were created using the data generator from IBM Almaden [1]. Stats for these datasets can be found in Figure 1 under *T10I4D100K* and *T40I10D100K*. The distribution of maximal frequent itemsets is displayed in Figure 2. For all datasets, the minimum support was chosen to yield around 100,000 elements in the MFI. Note that both T10I4 and T40I10 have very high concentrations of itemsets around two and three items long with T40I10 having another smaller peak around eight to nine items.

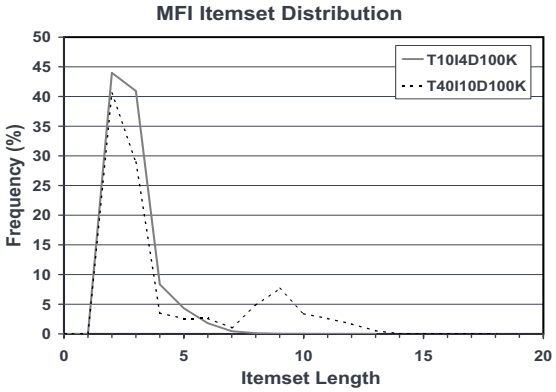


Figure 2. Itemset Lengths for shallow, artificial datasets

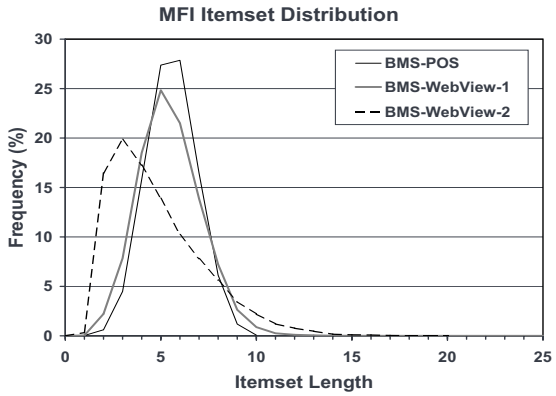


Figure 3. Itemset Lengths for shallow, real datasets

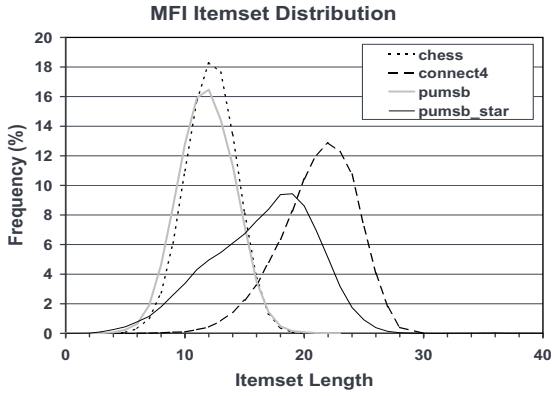


Figure 4. Itemset Lengths for dense, real datasets

The second dataset type is click stream data from two different e-commerce websites (*BMS-WebView-1* and *BMS-WebView-2*) where each transaction is a web session and each item is a product page view; this data was provided by Blue Martini [8]. *BMS-POS* contains point-of-sale data from an electronics retailer with the item-ids corresponding to product categories. Figure 3 shows that BMS-POS and BMS-WebView-1 have very similar normal curve itemset distributions with the average length of a maximal frequent itemset around five to six items long. On the other hand, BMS-WebView-2 has a right skewed distribution; there's a sharp incline until three items and then a more gradual decline on the right tail.

Finally, the last datasets used for analysis are the dense datasets. They are characterized by very long itemset patterns that peak around 10-25 items (see Figure 4). *Chess* and *Connect4* are gathered from game state information and are available from the UCI Machine Learning Repository [5]. The *Pumsb* dataset is census data from PUMS (Public Use Microdata Sample). *Pumsb-star* is the same dataset as *Pumsb* except all items of 80% support or more have been removed, making it less dense and easier to mine. Figure 4 shows that Chess and Pumsb have nearly identical itemset distributions that are normal around 10-12 items long. Connect4 and Pumsb-star are somewhat left-skewed with a slower incline that peaks around 20-23 items and then a sharp decline in the length of the frequent itemsets.

5.2 Other Algorithms

5.2.1 DepthProject

DepthProject demonstrated an order of magnitude improvement over previous algorithms for mining maximal frequent itemsets [2]. MAFIA was originally designed with DepthProject as the primary benchmark for comparison and we have implemented our own version of the DepthProject algorithm for testing.

The primary differences between MAFIA and DepthProject are the database representation (and consequently the support counting) and the application of pruning tools. DepthProject uses a horizontal database layout while MAFIA uses a vertical bitmap format, and supports of itemsets are counted very differently. Both algorithms use some form of compression when the bitmaps become sparse. However, DepthProject also utilizes a specialized counting technique called bucketing for the lower levels of the itemset lattice. When the tail of a node is small enough, bucketing will count the entire subtree with one pass over the data. Since bucketing counts all of the nodes in a subtree, many itemsets that MAFIA will prune out will be counted with DepthProject. For more details on the DepthProject algorithm, please refer to the paper by Agarwal and Aggarwal [2].

5.2.2 GenMax

GenMax is a new algorithm by Gouda and Zaki for finding maximal itemset patterns [7]. GenMax introduced a novel concept for finding supersets in the *MFI* called *progessive focusing*. The newest version of MAFIA has incorporated this technique with the *LMFI* update. GenMax also uses diffset propagation for fast support counting. Both algorithms use similar methods for itemset lattice exploration and pruning of the search space.

5.3 Experimental Analysis

We performed three types of experiments to analyze the performance of MAFIA. First, we analyze the effect of each pruning component of the MAFIA algorithm to demonstrate how the algorithm works to trim the search space of the itemset lattice. The second set of experiments examines the savings generated by using compression to speed support counting. Finally, we compare the performance of MAFIA against other current algorithms on all three types of data (see Section 5.1). In general, MAFIA works best on dense data with long itemsets, though the algorithm is still competitive on even very shallow data.

These experiments were conducted on a 1500 Mhz Pentium with 1GB of memory running Redhat Linux 9.0. All code was written in C++ and compiled using gcc version 3.2 with all optimizations enabled.

5.3.1 Algorithmic Component Analysis

First, we present a full analysis of each pruning component of the MAFIA algorithm (see Section 2 for algorithmic details). There are three types of pruning used to trim the tree: FHUT, HUTMFI, and PEP. FHUT and HUTMFI are both forms of superset pruning and thus will tend to "overlap" in their efficacy for reducing the search space. In addition, dynamic reordering can significantly reduce the size of the search space by removing infrequent items from each node's tail.

Figures 5 and 6 show the effects of each component of the MAFIA algorithm on the Connect4 dataset at 40% minimum support. The components of the algorithm are represented in a *cube* format with the running times (in seconds) and the number of itemsets counted during the MAFIA search. The top of the cube shows the time for a simple traversal where the full search space is explored, while the bottom of the cube corresponds to all three pruning methods being used. Two separate cubes (with and without dynamic reordering) rather than one giant cube are presented for readability.

Note that all of the pruning components yield great savings in running time compared to using no pruning. Applying a single pruning mechanism runs two to three orders of

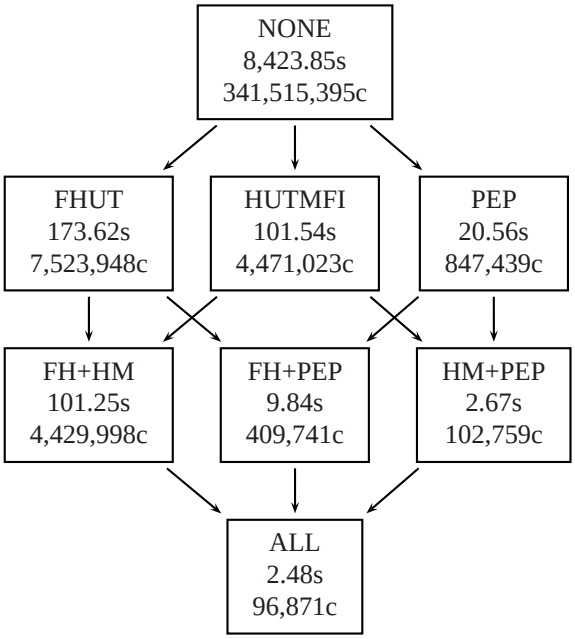


Figure 5. Pruning Components for Connect4 at 40% support *without reordering*

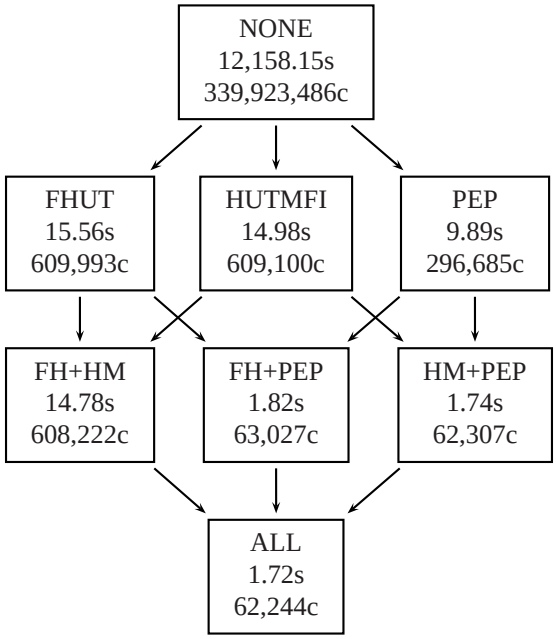


Figure 6. Pruning Components for Connect4 at 40% support *with reordering*

magnitude faster while using all of the pruning tools is four orders of magnitude faster than no pruning.

Several of the pruning components seem to overlap in trimming the search space. In particular, HUTMFI and FHUT yield very similar results, since they use the same type of superset pruning but with different methods of implementation. It is interesting to see that adding FHUT when HUTMFI is already performed yields very little savings, i.e. from HUTMFI to FH+HM or from HM+PEP to ALL, the running times do not significantly change. HUTMFI first checks for the frequency of a node's HUT by looking for a frequent superset in the MFI, while FHUT will explore the leftmost branch of the subtree rooted at that node. Apparently, there are very few cases where a superset of a node's HUT is not in the MFI, but the HUT is frequent.

PEP has the largest impact of the three pruning methods. Most of the running time of the algorithm occurs at the lower levels of the tree where the border between frequent and infrequent itemsets exists. Near this border, many of the itemsets have the same exact support right above the minimum support and thus, PEP is more likely to trim out large sections of the tree at the lower levels.

Dynamically reordering the tail also has dramatic savings (cf. Figure 5 with Figure 6). At the top of each cube, it is interesting to note that without any pruning mechanisms, dynamic reordering will actually run slower than static ordering. Fewer itemsets get counted, but the cost of reordering so many nodes outweighs the savings of counting fewer nodes.

However, once pruning is applied, dynamic reordering runs nearly an order of magnitude faster than the static ordering. PEP is more effective since the tail is trimmed as early in the tree as possible; all of the extensions with the same support are moved from the tail to the head in one step at the start of the subtree. Also, FHUT and HUTMFI have much more impact. With dynamic reordering, subtrees generated from the end of tail have the itemsets with the highest supports and thus the HUT is more likely to be frequent.

5.3.2 Effects of Compression in MAFIA

Adaptive compression uses cost estimation to determine when it is appropriate to compress the bitmaps. Since the cost estimate adapts to each dataset, adaptive compression is always better than using no compression. Results on different types of data show that adaptive compression is at least 25% faster as higher supports and at lower supports up to an order of magnitude faster.

Figures 7 and 8 display the effect of compression on sparse data. First, we analyze the sparse, artificial datasets T10I4 and T40I10 that are characterized by very short itemsets, where the average length of maximally frequent itemsets is only 2-6 items. Because these datasets are so sparse with small subtrees, at higher supports compression is not

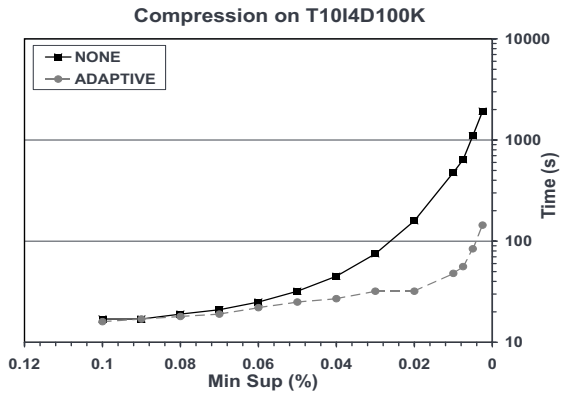


Figure 7. Compression on sparse datasets

often used and thus has a negligible effect. But as the support drops and the subtrees grow larger, the effect of compression is enhanced and the running times for adaptive compression increase to nearly 3-10 times faster.

Next are the results on the sparse, real datasets: BMS-POS, BMS-WebView-1, and BMS-WebView-2 in Figure 8. Note that for BMS-POS, adaptive compression follows the exact same pattern as the synthetic datasets with the difference growing from negligible to over 10 times better. BMS-WebView-1 follows the same general pattern except for an anomalous spike in the running times without compression around .05%. However, for BMS-WebView-2 compression has a very small impact and is only really effective at the lowest supports. Recall from Figure 3 that BMS-WebView-2 has a right-skewed distribution of frequent itemsets, which may help explain the different compression effect.

The final group of datasets is found in Figure 9 and shows the results of compression on dense, real data. The results on Chess and Pumsb indicate that very few compressed bitmaps were used; apparently, the adaptive compression algorithm determined compression to be too expensive. As a result, adaptive compression is only around 15-30% better than using no compression at all. On the other hand, the Connect4 and Pumsb-star datasets use a much higher ratio of compressed bitmaps and adaptive compression is more than three times faster than no compression.

It is interesting to note that Chess and Pumsb both have left-skewed distributions (see Figure 4) while Connect4 and Pumsb-star follow a more normal distribution of itemsets. The results indicate that when the data is skewed (left or right), adaptive compression is not as effective. Still, even in the worst case adaptive compression will use the cost estimate to determine that compression should not be chosen and thus is at least as fast as never compressing at all. In the best case, compression can significantly speed up support

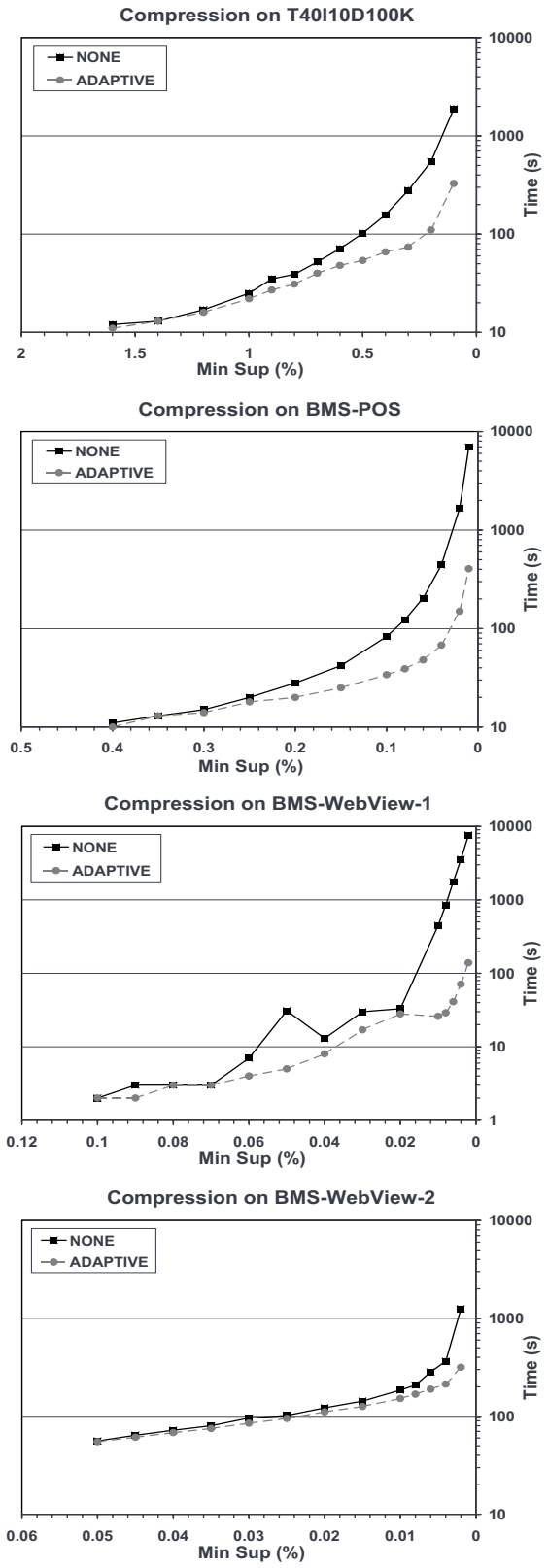


Figure 8. Compression on more sparse datasets

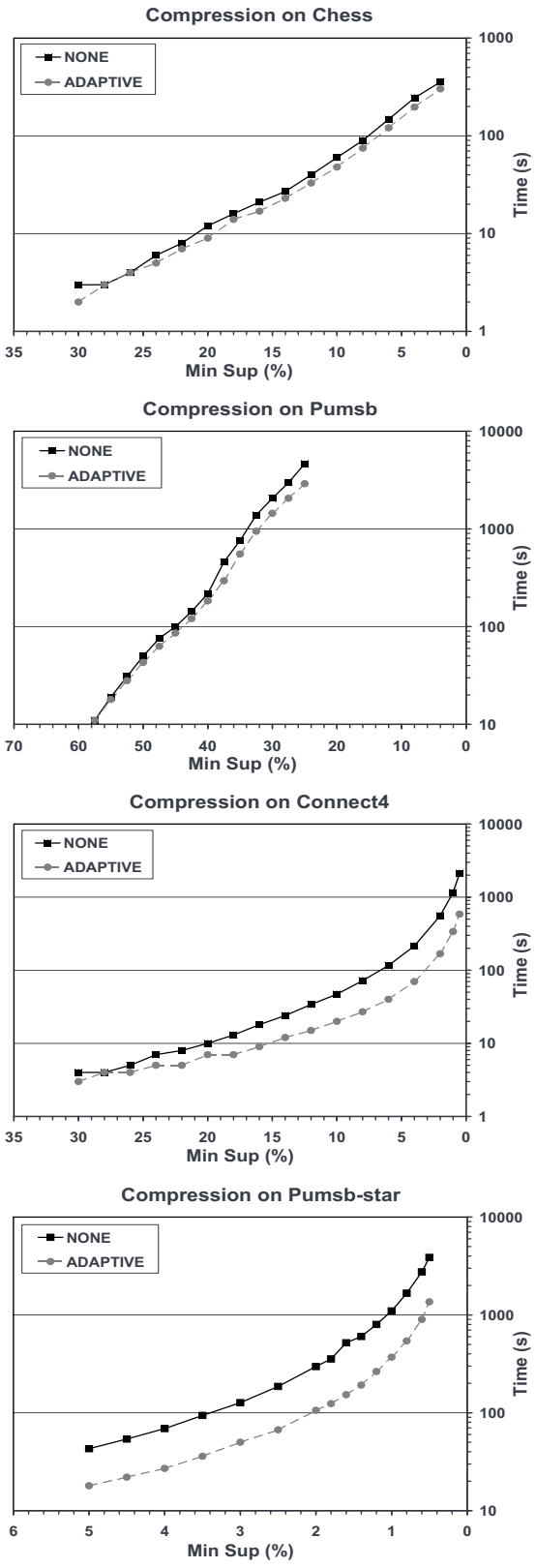


Figure 9. Compression on dense datasets

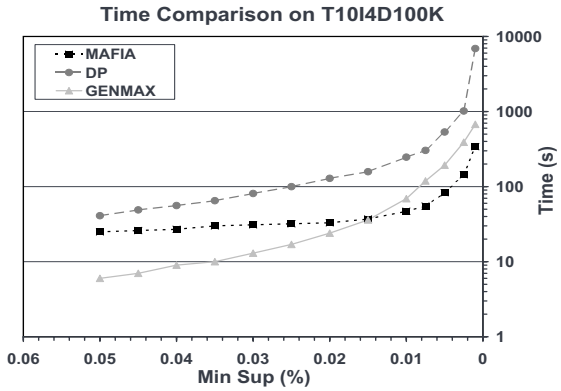


Figure 10. Performance on sparse datasets

counting by over an order of magnitude.

5.3.3 Performance Comparisons

Figures 10 and 11 show the results of comparing MAFIA with DepthProject and GenMax on sparse data. MAFIA is always faster than DepthProject and grows from twice as fast at the higher supports to more than 20 times faster at the lowest supports tested. GenMax demonstrates the best performance of the three algorithms for higher supports and is around two to three times faster than MAFIA. However, note that as the support drops and the itemsets become longer, MAFIA passes Genmax in performance to become the fastest algorithm.

The performances for sparse, real datasets are found in Figure 11. MAFIA has the worst performance on BMS-WebView-2 for higher supports, though it eventually passes DepthProject as the support lowers. BMS-POS and BMS-WebView-1 follow a similar pattern to the synthetic datasets where MAFIA is always better than DepthProject, and GenMax is better than MAFIA until the lower supports where they cross over. In fact, at the lowest supports for BMS-WebView-1, MAFIA is an order of magnitude better than GenMax and over 50 times faster than DepthProject. It is clear that MAFIA performs best when the itemsets are longer, though even for sparse data MAFIA is within two to three times the running times of DepthProject and GenMax.

The dense datasets in Figure 12 support the idea that MAFIA runs the fastest on longer itemsets. For all supports on the dense datasets, MAFIA has the best performance. MAFIA runs around two to five times faster than GenMax on Connect4, Pumsb, and Pumsb-star and over five to ten times faster on Chess. DepthProject is by far the slowest algorithm on all of the dense datasets and runs between ten to thirty times worse than MAFIA on all of the datasets across all supports.

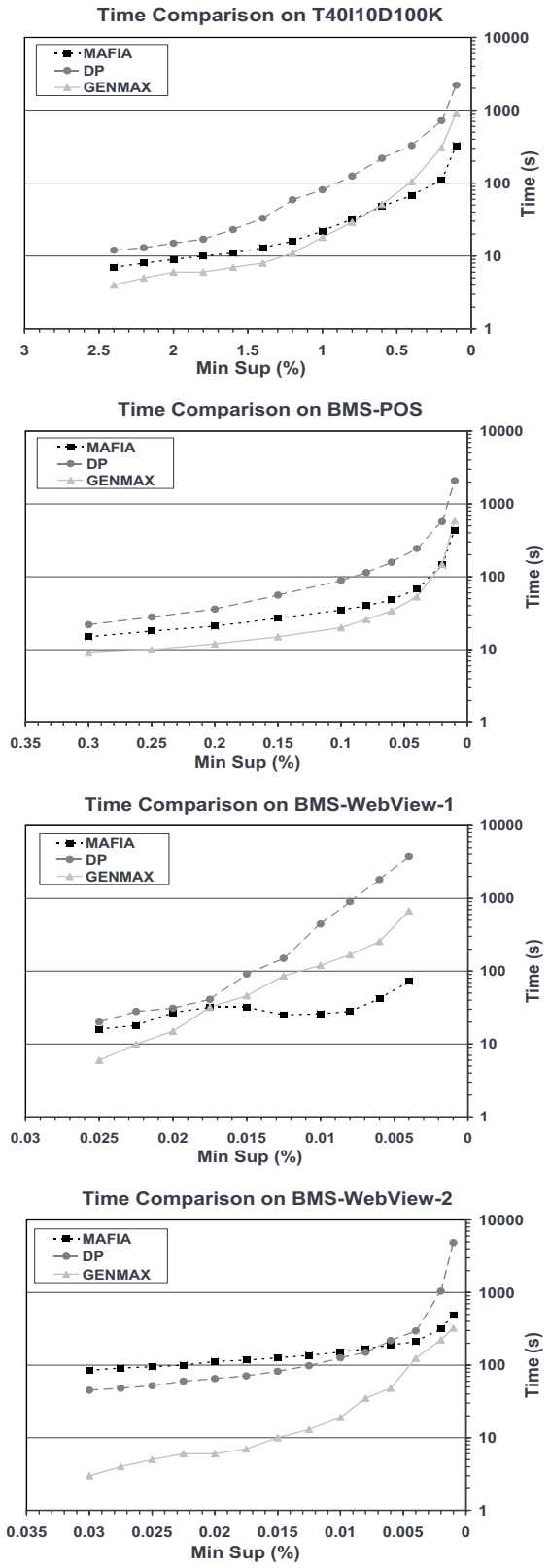


Figure 11. Performance on more sparse datasets

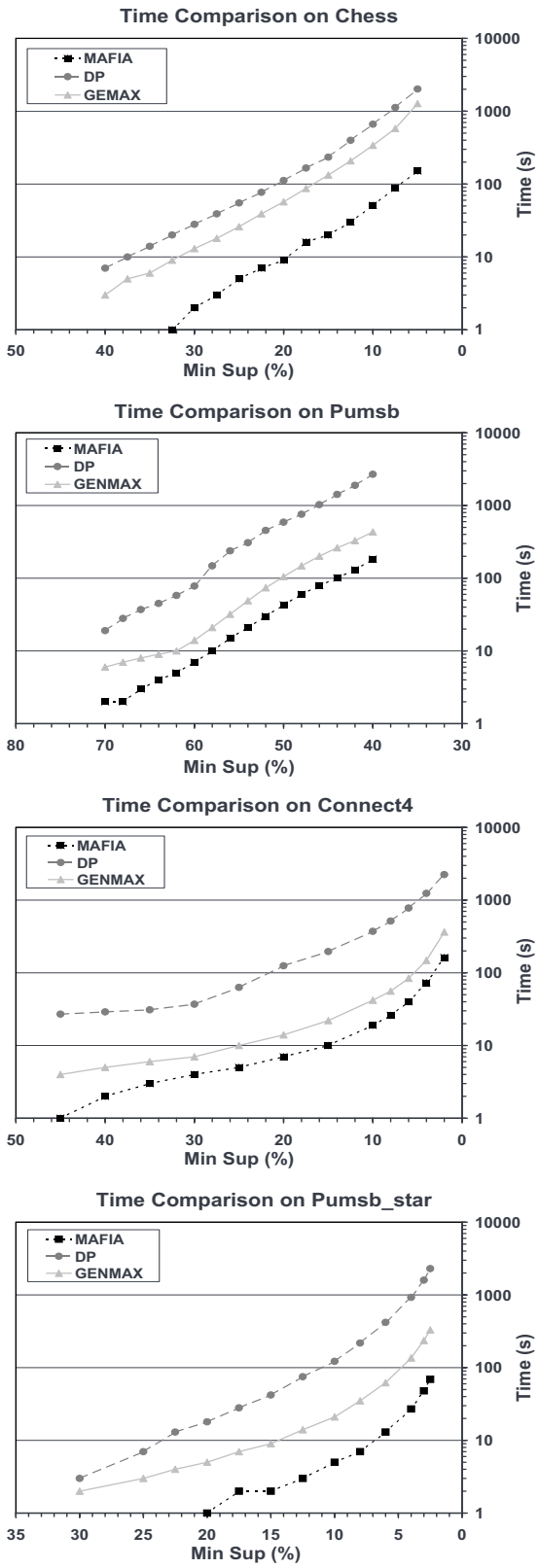


Figure 12. Performance on dense datasets

6 Conclusion

In this paper we present a detailed performance analysis of MAFIA. The breakdown of the algorithmic components show that powerful pruning techniques such as parent-equivalence pruning and superset checking are very beneficial in reducing the search space. We also show that adaptive compression/projection of the vertical bitmaps dramatically cuts the cost of counting supports of itemsets. Our experimental results demonstrate that MAFIA is highly optimized for mining long itemsets and on dense data consistently outperforms GenMax by two to ten and DepthProject by ten to thirty.

Acknowledgements: We would like to thank Ramesh Agarwal and Charu Aggarwal for discussing DepthProject and giving us advice on its implementation. We also thank Jayant Haritsa for his insightful comments on the MAFIA algorithm, Jiawei Han for helping in our understanding of CLOSET and providing us the executable of the FP-Tree algorithm, and Mohammed Zaki for making the source code of GenMax available.

References

[1] Data generator available at http://www.almaden.ibm.com/software/quest/Resources/.

[2] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. Depth first generation of long patterns. In *Knowledge Discovery and Data Mining*, pages 108–118, 2000.

[3] R. C. Agarwal, C. C. Aggarwal, and V. V. V. Prasad. A tree projection algorithm for generation of frequent item sets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[4] R. J. Bayardo. Efficiently mining long patterns from databases. In *SIGMOD*, pages 85–93, 1998.

[5] C. Blake and C. Merz. UCI repository of machine learning databases, 1998.

[6] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE 2001*, Heidelberg, Germany, 2001.

[7] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *ICDM*, pages 163–170, 2001.

[8] R. Kohavi, C. Brodley, B. Frasca, L. Mason, and Z. Zheng. KDD-Cup 2000 organizers' report: Peeling the onion. *SIGKDD Explorations*, 2(2):86–98, 2000. http://www.ecn.purdue.edu/KDDCUP.

[9] R.Rymon. Search through systematic set enumeration. In *International Conference on Principles of Knowledge Representation and Reasoning*, pages 539–550, 1992.

kDCI: a Multi-Strategy Algorithm for Mining Frequent Sets

Claudio Lucchese[1], Salvatore Orlando[1], Paolo Palmerini[1,2],
Raffaele Perego[2], Fabrizio Silvestri[2,3]

[1]Dipartimento di Informatica
Università Ca' Foscari di Venezia
Venezia, Italy
{orlando,clucches}@dsi.unive.it

[2]ISTI-CNR
Consiglio Nazionale delle
Ricerche
Pisa, Italy
{r.perego,p.palmerini}@isti.cnr.it

[3]Dipartimento di Informatica
Università di Pisa
Pisa, Italy
silvestri@di.unipi.it

Abstract

This paper presents the implementation of kDCI*, an enhancement of* DCI *[10], a scalable algorithm for discovering frequent sets in large databases.*

The main contribution of kDCI *resides on a novel counting inference strategy, inspired by previously known results by Basted* et al. *[3]. Moreover, multiple heuristics and efficient data structures are used in order to adapt the algorithm behavior to the features of the specific dataset mined and of the computing platform used.*

kDCI *turns out to be effective in mining both short and long patterns from a variety of datasets. We conducted a wide range of experiments on synthetic and real-world datasets, both in-core and out-of-core. The results obtained allow us to state that* kDCI *performances are not over-fitted to a special case, and its high performance is maintained on datasets with different characteristics.*

1 Introduction

Despite the considerable amount of algorithms proposed in the last decade for solving the problem of finding frequent patterns in transactional databases (among the many we mention [1] [11] [6] [13] [14] [4] [3] [7]), a single *best* approach still has to be found.

The Frequent Set Counting (*FSC*) problem consists in finding all the set of items (itemsets) which occur in at least $s\%$ (s is called support) of the transactions of a database D, where each transaction is a variable length collection of items from a set I. Itemsets which verify the minimum support threshold are said to be *frequent*.

The complexity of the FSC problem relies mainly in the potentially explosive growth of its full search space, whose dimension d is, in the worst case, $d = \sum_{k=1}^{|t_{max}|} \binom{|I|}{k}$, where t_{max} is the maximum transaction length. Taking into account the minimum support threshold, it is possible to reduce the search space, using the well known *downward closure relation*, which states that an itemset can only be frequent if all its subsets are frequent as well. The exploitation of this property, originally introduced in the *Apriori* algorithm [1], has transformed a potentially exponentially complex problem, into a more tractable one.

Nevertheless, the *Apriori* property alone is not sufficient to permit to solve the FSC problem in a reasonable time, in *all* cases, i.e. on all possible datasets and for all possible interesting values of s. Indeed, another source of complexity in the FSC problem resides in the dataset internal correlation and statistical properties, which remain unknown until the mining is completed. Such diversity in the dataset properties is reflected in measurable quantities, like the total number of transactions, or the total number of distinct items $|I|$ appearing in the database, but also in some other more fuzzy properties which, although commonly recognized as important, still lack a formal and univocal definition. It is the case, for example, of the notion of how *dense* a dataset is, i.e. how much its transactions tend to resemble among one another.

Several important results have been achieved for specific cases. Dense datasets are effectively mined with compressed data structure [14], explosion in the candidates can be avoided using effective projections of the dataset [7], the support of itemsets in compact datasets

can be inferred, without counting, using an equivalence class based partition of the dataset [3].

In order to take advantage of all these, and more specific results, hybrid approaches have been proposed [5]. Critical to this point is *when* and *how* to adopt a given solution instead of another. In lack of a complete theoretical understanding of the FSC problem, the only solution is to adopt a heuristic approach, where theoretical reasoning is supported by direct experience leading to a strategy that tries to cover a variety of cases as wide as possible.

Starting from the previous DCI (Direct Count & Intersect) algorithm [10] we propose here kDCI, an enhanced version of DCI that extends its adaptability to the dataset specific features and the hardware characteristics of the computing platform used for running the FSC algorithm. Moreover, in kDCI we introduce a *novel counting inference* strategy, based on a new result inspired by the work of Bastide *et al.* in [3].

kDCI is a multiple heuristics hybrid algorithm, able to adapt its behavior during the execution. Since it origins from the already published DCI algorithm, we only outline in this paper how kDCI differs from DCI. A detailed description of the DCI algorithm can be found in [10].

2 The kDCI algorithm

Several considerations concerning the features of real datasets, the characteristics of modern hw/sw system, as well as scalability issues of FSC algorithms have motivated the design of kDCI. As already pointed out, transactional databases may have different characteristics in terms of correlations among the items inside transactions and of transactions among themselves [9]. A desirable feature of an FSC algorithm should be the ability to adapt its behavior to these characteristics.

Modern hw/sw systems need high locality for exploiting memory hierarchies effectively and achieving high performance. Algorithms have to favor the exploitation of spatial and temporal locality in accessing in-core and out-core data.

Scalability is the main concern in designing algorithms that aim to mine large databases efficiently. Therefore, it is important to be able to handle datasets bigger than the available memory.

We designed and implemented our algorithm kDCI keeping in mind such performance issues. The pseudo code of kDCI is given in Algorithm 1.

kDCI inherits from DCI the level-wise behavior and the hybrid horizontal-vertical dataset representation. As computation is started, kDCI maintains the database in horizontal format and applies an effective pruning technique to remove infrequent items and short transactions. A temporary dataset is therefore written to disk at every iteration. The first steps of the algorithm are described in [8] and [10] and remain unchanged in kDCI. In kDCI we only improved memory management by exploiting compressed and optimized data structures (see Section 2.1 and 2.2).

The effectiveness of pruning is related to the possibility of storing the dataset in main memory in vertical format, due to the dataset size reduction. This normally occurs at the first iterations, depending on the dataset, the support threshold and the memory available on the machine, which is determined at run time.

Once the dataset can be stored in main memory, kDCI switches to the vertical representation, and applies several heuristics in order to determine the most effective strategy for frequent itemset counting.

The most important innovation introduced in kDCI regards a novel technique to determine the itemset supports, inspired by the work of Bastide *et al.* [3]. As we will discuss in Section 2.4, in some cases the support of candidate itemsets can be determined without actually

Algorithm 1 kDCI

```
Require: D, min_supp
  // During first scan get optimization figures
  F_1 = first_scan(D, min_supp)
  // second and following scans on a temporary db D'
  F_2 = second_scan(D', min_supp)
  k = 2
  while (D'.vertical_size() > memory_available()) do
    k++
    // count-based iteration
    F_k = DCP(D', min_supp, k)
  end while
  k++
  // count-based iteration + create vertical database VD
  F_k = DCP(D', VD, min_supp, k)
  dense = VD.is_dense())
  while (F_k ≠ ∅) do
    k++
    if (use_key_patterns()) then
      if (dense) then
        F_k = DCI_dense_keyp(VD, min_supp, k)
      else
        F_k = DCI_sparse_keyp(VD, min_supp, k)
      end if
    else
      if (dense) then
        F_k = DCI_dense(VD, min_supp, k)
      else
        F_k = DCI_sparse(VD, min_supp, k)
      end if
    end if
  end while
```

counting transactions, but by a faster inference reasoning.

Moreover, kDCI maintains the different strategies implemented in DCI for sparse and dense datasets. The result is a multiple strategy approach: during the execution kDCI collects statistical information on the dataset that allows to determine which is the best approach for the particular case.

In the following we detail such optimizations and improvements and the heuristics used to decide which optimization to use.

2.1 Dynamic data type selection

The first optimization is concerned with the amount of memory used to represent itemsets and their counters. Since such structures are extensively accessed during the execution of the algorithm, is it profitable to have such data occupying as little memory as possible. This not only allows to reduce the spatial complexity of the algorithm, but also permits low level processor optimizations to be effective at run time.

During the first scan of the dataset, global properties are collected like the total number of distinct frequent items (m_1), the maximum transaction size, and the support of the most frequent item.

Once this information is available, we remap the survived (frequent) items to contiguous integer identifiers. This allows us to decide the best data type to represent such identifiers and their counters. For example if the maximum support of any item is less than 65536, we can use an `unsigned short int` to represent the itemset counters. The same holds for the remapped identifiers of the items. The decision of which is the most appropriate type to use for items and counters is taken at run time, by means of a C++ template-based implementation of all the kDCI code.

Before remapping item identifiers, we also reorder them in increasingly support ordering: more frequent items are thus assigned larger identifiers. This also simplifies the intersection-based technique used for dense datasets (see Section 2.3).

2.2 Compressed data structures

Itemsets are often organized in *collections* in many FSC algorithms. Efficient representation of such collections can lead to important performance improvements. In [8] we pointed out the advantages of storing candidates in directly accessible data structures for the first passes of our algorithm. In kDCI we introduce a compressed representation of an itemset collection, used to store in the main memory collections of candidate and

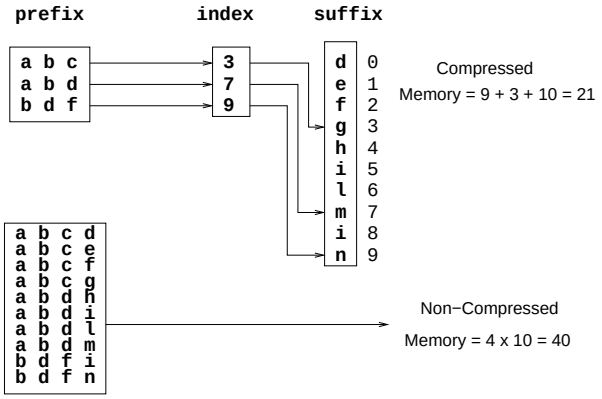


Figure 1. Compressed data structure used for itemset collection can reduce the amount of memory needed to store the itemsets.

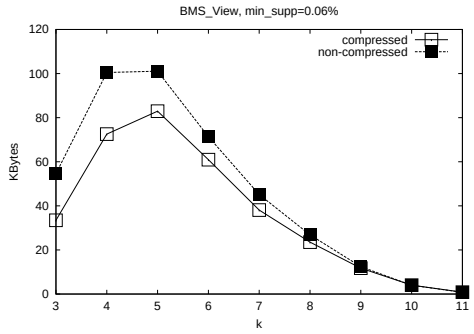
frequent itemsets. This representation take advantage of prefix sharing among the lexicographically ordered itemsets of the collection.

The compressed data structure is based on three arrays (Figure 1). At each iteration k, the first array (`prefix`) stores the different prefixes of length $k-1$. In the third array (`suffix`) all the length-1 suffixes are stored. Finally, in the element i of the second array (`index`), we store the position in the `suffix` array of the section of suffixes that share the same prefix. Therefore, when the itemsets in the collection have to be enumerated, we first access the `prefix` array. Then, from the corresponding entry in the `index` array we get the section of suffixes stored in `suffix`, needed to complete the itemsets.

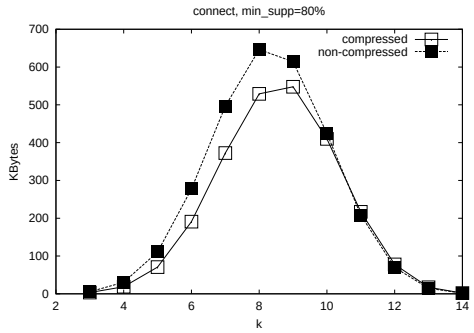
From our tests we can say that, in all the interesting cases – i.e., when the number of candidate (or frequent) iemsets explodes – this data structure works well and achieves up to 30% as compression ratio. For example, see the results reported in Figure 2.

2.3 Heuristics

One of the most important features of kDCI is its ability to adapt its behavior to the dataset specific characteristics. It is well known that being able to distinguish between sparse and dense datasets, for example, allows to adopt specific and effective optimizations. Moreover, as we will explain the Section 2.4, if the number of frequent itemsets is much greater than the number of closed itemsets, it is possible to apply a counting inference procedure that allows to dramatically reduce the time needed to determine itemset supports.



(a)



(b)

Figure 2. Memory usage with compressed itemsets collection representation for BMS with min_sup=0.06% (a) and connect with min_sup=80% (b)

In kDCI we devised two main heuristics that allow to distinguish between dense and sparse datasets and to decide whether to apply the counting inference procedure or not.

The first heuristic is simply based on the measure of the dataset density. Namely, we measure the correlation among the tidlists corresponding to the most frequent items. We require that the maximum number of frequent items for which such correlation is significant, weighted by the correlation degree itself, is above a given threshold.

As an example, consider the two dataset in Figure 3, where tidlists are placed horizontally, i.e. rows correspond to items and columns to transactions. Suppose to choose a density threshold $\delta = 0.2$. If we order the items according to their support, we have the most dense region of the dataset at the bottom of each figure. Starting from the bottom, we find the maximum number of items whose tidlists have a significant intersection. In the case of dataset (a), for example, a fraction $f = 1/4$ of the items share $p = 90\%$ of the transactions, leading

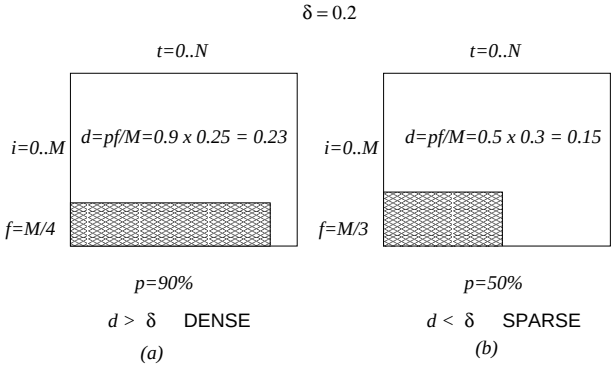


Figure 3. Heuristic to establish a dataset density or sparsity

to a density of $d = fp = 0.25 \times 0.9 = 0.23$ which is above the density threshold. For dataset (b) on the other hand, to a smaller intersection of $p = 50\%$ is common to $f = 1/3$ of the items. In this last case the density $d = fp = 0.3 \times 0.5 = 0.15$ is lower than the threshold and the dataset is considered as sparse. It is worth to notice that since this notion of density depends on the minimum support threshold, the same dataset can exhibits different behaviors when mined with different support thresholds.

Once the dataset density is determined, we adopted the same optimizations described in [10] for sparse and dense datasets. We review them briefly for completeness.

Sparse datasets. The main techniques used for sparse datasets can be summarized as follows:

- **projection**. Tidlists in sparse datasets are characterized by long runs of 0's. When intersecting the tidlists associated with the 2-prefix items belonging to a given candidate itemset, we keep track of such empty elements (words), in order to perform the following intersections faster. This can be considered as *a sort of raw projection* of the vertical dataset, since some transactions, i.e. those corresponding to zero words, are not considered at all during the following tidlist intersections.
- **pruning**. We remove infrequent items from the dataset. This can result in some transaction remaining empty or with too few items. We therefore remove such transactions (i.e. columns in the our bitvector vertical representation) from the dataset. Since this bitwise

pruning may be expensive, we only perform it when the benefits introduced are expected to balance its cost.

Dense datasets.

If the dataset is dense, we expect to deal with strong correlations among the most frequent items. This not only means that the tidlists associated with these *most frequent items* contain long runs of 1's, but also that they turn out to be very similar. The heuristic technique adopted by DCI and consequently by kDCI for dense dataset thus works as follows:

- we reorder the columns of the vertical dataset by moving identical segments of the tidlists associated with the most frequent items to the first consecutive positions;
- since each candidate is likely to include several of these most frequent items, we avoid repeated intersections of identical segments.

The heuristic for density evaluation is applied only once, as soon as the vertical dataset is built. After this decision is taken, we further check if the counting inference strategy (see Section 2.4) can be profitable or not. The effectiveness of the inference strategy depends on the ratio between the total number of frequent itemsets and how many of them are *key-patterns*. The closer to 1 this ratio is, the less advantage is introduced by the inference strategy. Since this ratio is not known until the computation is finished, we found that the same information can be derived from the average support of the frequent singletons (items), after the first scan. The idea behind this is that if the average support of the single items that survived the first scan is high enough, then longer patterns can be expected to be frequent and more likely the number of *key-patterns* itemsets will be lower than that of frequent itemsets. We experimentally verified that this simple heuristic gives the correct output for all datasets - both real and synthetic.

To resume the rationale behind kDCI multiple strategy approach, if the *key-patterns* optimization can be adopted, we use the counting inference method that allows to avoid many intersections. For the intersections that cannot be avoided and in the cases where the *key-patterns* inference method cannot be applied, we further distinguish between sparse and dense datasets, and apply the two strategies explained above.

2.4 Pattern Counting Inference

In this section we describe the count inference method, which constitute the most important innovation

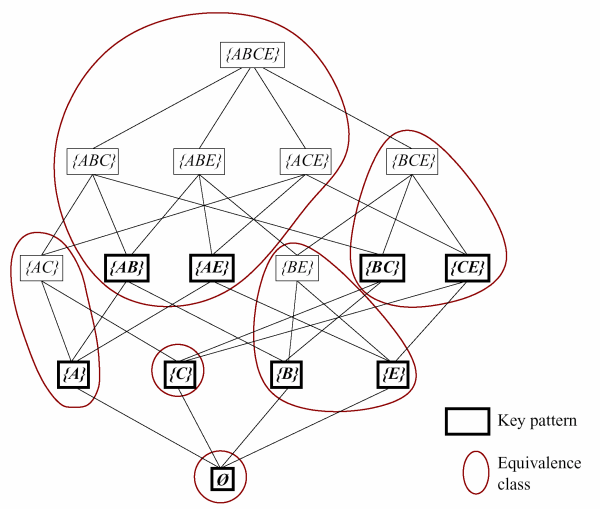


Figure 4. Example lattice of frequent items.

introduced in kDCI. We exploit a technique inspired by the theoretical results presented in [3], where the PASCAL algorithm was introduced. PASCAL is able to infers the support of an itemset without actually count its occurrences in the database. In this seminal work, the authors introduced the concept of *key pattern* (or *key itemset*). Given a generic pattern Q, it is possible to determine an equivalence class $[Q]$, which contains the set of patterns that have the same support and are included in the same set of database transactions. Moreover, if we define $min[P]$ as the set of the smallest itemsets in $[P]$, a pattern P is a *key pattern* if $P \in \min[P]$, i.e. no proper subset of P is in the same equivalence class. Note that we can have several key patterns for each equivalence class. Figure 4 shows an example of a lattice of frequent itemsets, taken from [3], where equivalence classes and key patterns are highlighted.

Given an equivalence class $[P]$, we can also define a corresponding *closed set* [12]: the closed set c of $[P]$ is equal to $\max[P]$, so that no proper supersets of c can belong to the same equivalence class $[P]$.

Among the results illustrated in [3] we have the following important theorems:

Theorem 1 *Q is a key pattern iff* $supp(Q) \neq \min_{p \in Q}(supp(Q \setminus \{p\}))$.

Theorem 2 *If P is not a key pattern, and $P \subseteq Q$, then Q is a non-key pattern as well.*

From Theorem 1 it is straightforward to observe that if Q is a non-key pattern, then:

$$supp(Q) = \min_{p \in Q}(supp(Q \setminus \{p\})). \quad (1)$$

Moreover, Theorem 1 says that we can check whether Q is a *key pattern* by comparing its support with the minimum support of its proper subsets, i.e. $\min_{p \in Q}(supp(Q \setminus \{p\}))$. We will show in the following how to use this property to make faster candidate support counting.

Theorems 1 and 2 give the theoretical foundations for the PASCAL algorithm, which finds the support of a *non-key* k-candidate Q by simply searching the minimum supports of all its $k-1$ subsets. Note that such search can be performed during the pruning phase of the Apriori candidate generation. DCI does not perform candidate pruning because its intersection technique is comparably faster. For this reason we will not adopt the PASCAL counting inference in kDCI.

The following theorem, partially inspired by the proof of Theorem 2, suggests a faster way to compute the support of a *non-key* k-candidate Q.

Before introducing the theorem, we need to define the function f, which assigns to each pattern P the set of all the transactions that include this pattern. We can define the support of a pattern in terms of f: $supp(P) = |f(P)|$. Note that f is a monotonically decreasing function, i.e. if $P_1 \subseteq P_2 \Rightarrow f(P_2) \subseteq f(P_1)$. This is obvious, because every transaction containing P_2 surely contains all the subsets of P_2.

Theorem 3 *If P is a non-key pattern and $P \subseteq Q$, the following holds:*

$$f(Q) = f(Q \setminus (P \setminus P')).$$

where $P' \subset P$, and P and P' belong to the same equivalence class, i.e. $P, P' \in [P]$.

PROOF. Note that, since P is a *non-key pattern*, it is surely possible to find a pattern P', $P' \subset P$, belonging to the same equivalence class $[P]$.

In order to demonstrate the Theorem we first show that $f(Q) \subseteq f(Q \setminus (P \setminus P'))$ and then that also $f(Q) \supseteq f(Q \setminus (P \setminus P'))$ holds, thus proving the Theorem hypotheses.

The first assertion $f(Q) \subseteq f(Q \setminus (P \setminus P'))$ holds because $(Q \setminus (P \setminus P')) \subseteq Q$, and f is a monotonically decreasing function.

To prove the second assertion, $f(Q) \supseteq f(Q \setminus (P \setminus P'))$, we can rewrite $f(Q)$ as $f(Q \setminus (P \setminus P') \cup (P \setminus P'))$, which is equivalent to $f(Q \setminus (P \setminus P')) \cap f(P \setminus P')$.

Since f is decreasing, $f(P) \subseteq f(P \setminus P')$. But, since $P, P' \in [P]$, then we can write $f(P) = f(P') \subseteq f(P \setminus P')$. Therefore $f(Q) = f(Q \setminus (P \setminus P')) \cap f(P \setminus P') \supseteq f(Q \setminus (P \setminus P')) \cap f(P')$. The last inequality is equivalent to $f(Q) \supseteq f(Q \setminus (P \setminus P') \cup P')$. Since $P' \subseteq (Q \setminus (P \setminus P'))$ clearly holds, it follows that $f(Q \setminus (P \setminus P') \cup P') = f(Q \setminus (P \setminus P'))$. So we can conclude that $f(Q) \supseteq f(Q \setminus (P \setminus P'))$, which completes the proof. □

The following corollary is trivial, since we defined $supp(Q) = |f(Q)|$.

Corollary 1 *If P is a non-key pattern, and $P \subseteq Q$, the support of Q can be computed as follows:*

$$supp(Q) = supp(Q \setminus (P \setminus P'))$$

where P' and P, $P' \subset P$, belong to the same equivalence class, i.e. $P, P' \in [P]$.

Finally, we can introduce Corollary 2, which is a particular case of the previous one.

Corollary 2 *If Q is k-candidate (i.e. $Q \in C_k$) and P, $P \subset Q$, is a frequent non-key* (k-1)*-pattern (i.e. $P \in F_{k-1}$), there must exist $P' \in F_{k-2}$, $P' \subset P$, such that P and P' belong to the same equivalence class, i.e. $P, P' \in [P]$ and P and P' differ for a single item: $\{p_{diff}\} = P \setminus P'$. The support of Q can thus be computed as:*

$$supp(Q) = supp(Q \setminus (P \setminus P')) = supp(Q \setminus \{p_{diff}\})$$

Corollary 2 says that to find the support of a *non-key candidate pattern* Q, we can simply check whether $Q \setminus \{p_{diff}\}$ belongs to F_{k-1}, or not. If $Q \setminus \{p_{diff}\} \in F_{k-1}$, then Q inherits the same support as $Q \setminus \{p_{diff}\}$ and is therefore frequent. Otherwise we can conclude that $Q \setminus \{p_{diff}\}$ is not frequent.

Using the theoretical result of Corollary 2, we adopted the following strategy in order to determine the support of a candidate Q at step k.

In kDCI, we store with each itemset $P \in F_{k-1}$ the following information:

- $supp(P)$;
- a flag indicating if P is a *key pattern* or not;
- if P is *non-key pattern*, also the item p_{diff} such that $P \setminus \{p_{diff}\} = P' \in [P]$.

Note that p_{diff} must be one of the items that we can remove from P to obtain a proper subset P' of P, belonging to the same equivalence class.

During the generation of a generic candidate $Q \in C_k$, as soon as kDCI discovers that one of the subsets of Q,

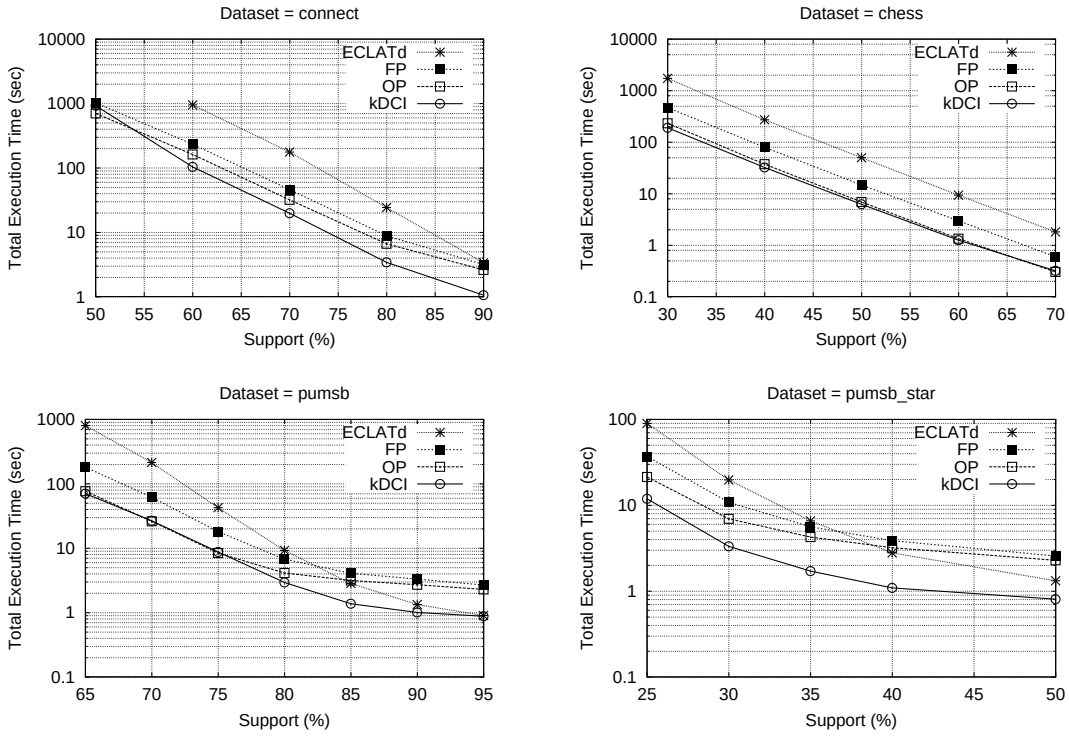


Figure 5. Total execution time of OP, FP, Eclatd, and kDCI on various datasets as a function of the support threshold.

say P, is a *non-key pattern*, kDCI searches in F_{k-1} the pattern $Q \setminus \{p_{diff}\}$, where p_{diff} is stored with P.

If $Q \setminus \{p_{diff}\}$ is found, then Q is a frequent *non-key pattern* (see Theorem 2), its support is $supp(Q \setminus \{p_{diff}\})$, and the item to store with Q is exactly p_{diff}. In fact, $Q' = Q \setminus \{p_{diff}\} \in [Q]$, i.e. p_{diff} is one of the items that we can remove from Q to obtain a subset Q' belonging to the same equivalence class.

The worst case is when all the subsets of Q in F_{k-1} are *key patterns* and the support of Q cannot be inferred from its subsets. In this case kDCI counts the support of Q as usual, and applies Theorem 1 to determine if Q is a *non-key-pattern*. If Q is a *non-key-pattern*, its support becomes $supp(Q) = \min_{p \in Q}(supp(Q \setminus \{p\}))$ (see Theorem 1), while the item to be stored with Q is p_{diff}, i.e. the item to be subtracted from Q to obtain the pattern with the minimum support.

The impact of this counting inference technique on the performance of an FSC algorithm becomes evident if you consider the Apriori-like candidate generation strategy adopted by kDCI. From the combination of every pair of itemsets P_a and $P_b \in F_{k-1}$, that share the same *(k-2)*-prefix (we called them *generators*), kDCI generates a candidate k-itemset Q. For very dense datasets, most of the frequent patterns belonging to F_{k-1} are *non-key patterns*. Therefore one or both patterns P_a and P_b used to generate $Q \in C_k$ are likely to be *non-key patterns*. In such cases, in order to find a *non-key pattern* and then apply Corollary 2, it is not necessary to check the existence of further subsets of Q. For most of the candidates, a single binary search in F_{k-1}, to look for the pattern $Q \setminus \{p_{diff}\}$, is thus sufficient to compute $supp(Q)$. Moreover, often $Q \setminus \{p_{diff}\}$ is exactly equal to one of the two *k-1*-itemsets belonging to the generating pair (P_a, P_b): in this case kDCI does not need to perform any search at all to compute $supp(Q)$.

We conclude this section with some examples of how the counting inference technique works. Let us consider Figure 4. Itemset $Q = \{A, B, E\}$ is a *non-key pattern* because $P = \{B, E\}$ is a *non-key pattern* as well. So, if $P' = \{B\}$, kDCI will store $p_{diff} = E$ with P. We have that the $supp(\{A, B, E\}) = supp(\{A, B, E\} \setminus (\{B, E\} \setminus \{B\})) = supp(\{A, B, E\} \setminus \{p_{diff}\} = supp(\{A, B\})$. From the Figure you can see that $\{A, B, E\}$ and $\{A, B\}$ both belong to the same

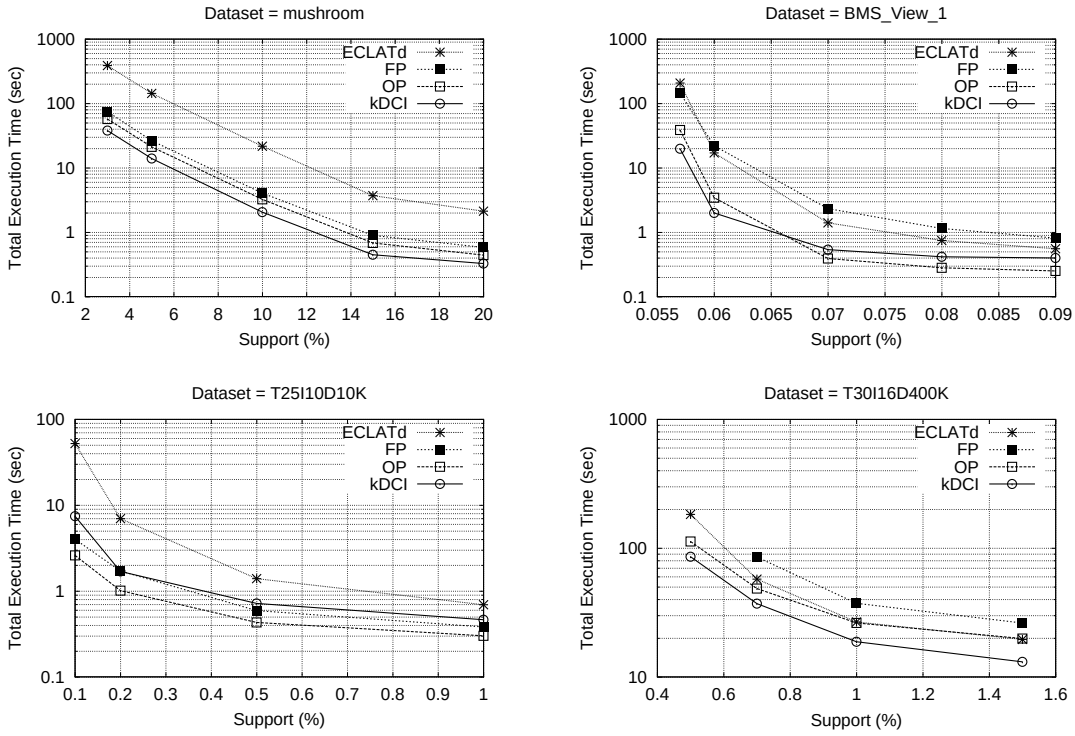


Figure 6. Total execution time of OP, FP, Eclatd, and kDCI on various datasets as a function of the support threshold.

equivalence class.

Another example is itemset $Q = \{A, B, C, E\}$, that is generated by the two *non-key patterns* $\{A, B, C\}$ and $\{A, B, E\}$. Suppose that $P = \{A, B, C\}$, i.e. the first generator, while $P' = \{A, B\}$. In this case kDCI will store $p_{diff} = C$ with P. We have that the $supp(\{A, B, C, E\}) = supp(\{A, B, C, E\} \setminus (\{A, B, C\} \setminus \{A, B\})) = supp(\{A, B, C, E\} \setminus \{p_{diff}\} = supp(\{A, B, E\}$, where $\{A, B, E\}$ is exactly the second generator. In this case, no search is necessary to find $\{A, B, E\}$. Looking at the Figure, it is possible to verify that $\{A, B, C, E\}$ and $\{A, B, E\}$ both belong to the same equivalence class.

3 Experimental Results

We experimentally evaluated kDCI performances by comparing its execution time with respect to the original implementations of state of the art FSC algorithms, namely FP-growth (FP) [6], Opportunistic Projection (OP) [7] and Eclat with diffsets (Eclatd) [14], provided by their respective authors.

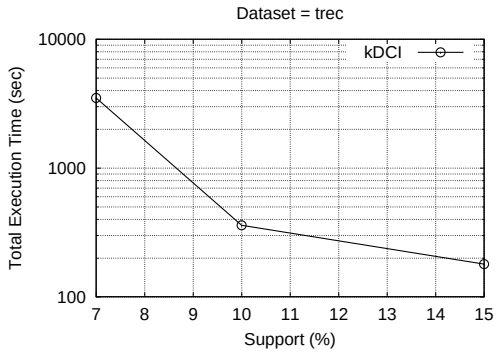
We used a MS-WindowsXP workstation equipped with a Pentium IV 1800 MHz processor, 368MB of RAM memory and an eide hard disk. For the tests, we used both synthetic and real-world datasets. All the synthetic datasets used were created with the IBM dataset generator [1], while all the real-world datasets but one were downloaded from the UCI KDD Archive (`http://kdd.ics.uci.edu/`). We also extracted a real-world dataset from the TREC WT10g corpus [2]. The original corpus contained about 1.69 millions of WEB documents. The dataset for our tests was built by considering the set of all the terms contained in each document as a transaction. Before generating the transactional dataset, the collection of documents was filtered by removing HTML tags and the most common words (*stopwords*), and by applying a *stemming* algorithm. The resulting `trec` dataset is huge. It is about 1.3GB, and contains 1.69 millions of short and long transactions, where the maximum length of a transaction is $71,473$ items.

kDCI performance and comparisons. Figure 5 and 6 report the total execution time obtained running FP, Eclatd, OP, and kDCI on various datasets as a func-

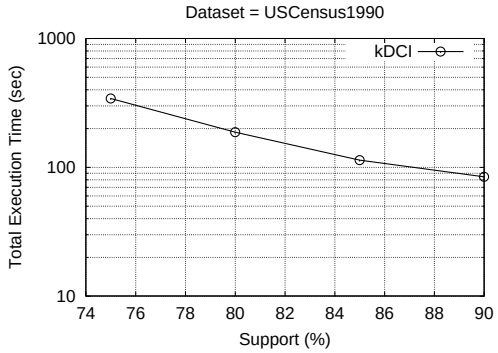
tion of the support threshold s. On all datasets in Figure 5, `connect`, `chess` `pumsb` and `pumsb_star`, kDCI runs faster than the others algorithms. On `pumsb` its execution time is very similar to the one of OP. For high support thresholds kDCI can drastically prune the dataset, and build a compact vertical dataset, whose tidlists presents large similarities. Such similarity of tidlists is effectively exploited by our strategy for compact datasets. For smaller supports, the benefits introduced by the counting inference strategy become more evident, particularly for the `pumsb_star` and `connect` datasets. In these cases the number of frequent itemsets is much higher than the number of *key-patterns*, thus allowing kDCI to drastically reduce the number of intersections needed to determine candidate supports.

On the datasets `mushroom` and `T30I16D400K` (see Figure 6), kDCI outperforms the other competitors, and this also holds on the real-world dataset `BMS_View_1` when mined with very small support thresholds (see Figure 6). On only a dataset, namely `T25I10D10K`, FP and OP are faster than kDCI for all the supports. The reason of this behavior is the size of the candidate set C_3, which for this dataset is much larger than F_3. While kDCI has to carry out a lot of useless work to determine the support of many candidate itemsets which are not frequent, FP-growth and OP take advantage of the fact that they do not require candidate generation.

Furthermore, differently from FP, Eclatd, and OP, kDCI can efficiently mine huge datasets such as `trec` and `USCensus1990`. Figure 7 reports the total execution time required by kDCI to mine these datasets with different support thresholds. The other algorithms failed in mining these datasets due to memory shortage, also when very large support thresholds were used. On the other hand, kDCI was able to mine such huge datasets since it adapts its behavior to both the size of the dataset and the main memory available.



(b)



(a)

Figure 7. Total execution time of kDCI: on datasets `trec` (a) and `USCensus1990` (b) as a function of the support.

4 Conclusions and Future Work

Due to the complexity of the problem, a good algorithm for FSC has to implement multiple strategies and some level of adaptiveness in order to be able to succesfully manage diverse and differently featured inputs.

kDCI uses different approaches for extracting frequent patterns: count-based during the first iterations and intersection-based for the following ones.

Moreover, a new counting inference strategy, together with, adaptiveness and resource awareness are the main innovative features of the algorithm.

On the basis of the characteristics of the mined dataset, kDCI chooses which optimization to adopt for reducing the cost of mining at run–time. Dataset pruning and effective out-of-core techniques are exploited during the count-based phase, while the intersection-based phase, which starts only when the pruned dataset can fit into the main memory, exploits a novel technique based on the notion of *key-pattern* that in many cases allows to infer the support of an itemset without any counting.

kDCI also adopts compressed data structures and dynamic type selection to adapt itself to the characteristics of the dataset being mined.

The experimental evaluation demonstrated that kDCI outperforms FP, OP, and Eclatd in most cases. Moreover, differently from the other FSC algorithms tested, kDCI can efficiently manage very large datasets, also on machines with limited physical memory.

Although the variety of datasets used and the large amount of tests conducted permit us to state that the performance of kDCI is not highly influenced by dataset

characteristics, and that our optimizations are very effective and general, some further optimizations and future work will reasonably improve kDCI performance. More optimized data structures could be used to store itemset collections in order to make faster searches in such collections. Note that such fast searches are very important in kDCI, which bases its count inference technique at level k on searching for frequent itemset in F_{k-1}. Finally, we can benefit from a higher level of adaptiveness to the available memory on the machine, either with fully memory mapped data structures or with out-of-core ones, depending on the data size. This should allow a better scalability and a wider applicability of the algorithm.

5 Acknowledgments

We acknowledge J. Han, Y. Pan, M.J. Zaki and J. Liu for kindly providing us the latest versions of their FSC software.

References

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proc. of the 20th VLDB Conf.*, pages 487–499, 1994.

[2] P. Bailey, N. Craswell, and D. Hawking. Engineering a multi-purpose test collection for Web retrieval experiments. *Information Processing and Management*. to appear.

[3] Y. Bastide, R. Taouil, N. Pasquier, G. Stumme, and L. Lakhal. Mining frequent patterns with counting inference. *ACM SIGKDD Explorations Newsletter*, 2(2):66–75, December 2000.

[4] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In J. Peckham, editor, *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. ACM Press, 05.

[5] B. Goethals. *Efficient Frequent Itemset Mining*. PhD thesis, Limburg University, Belgium, 2003.

[6] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, pages 1–12, Dallas, Texas, USA, 2000.

[7] J. Liu, Y. Pan, K. Wang, and J. Han. Mining Frequent Item Sets by Opportunistic Projection. In *Proc. 2002 Int. Conf. on Knowledge Discovery in Databases (KDD'02), Edmonton, Canada*, 2002.

[8] S. Orlando, P. Palmerini, and R. Perego. Enhancing the Apriori Algorithm for Frequent Set Counting. In *Proc. of 3rd Int. Conf. on Data Warehousing and Knowledge Discovery (DaWaK 01) - Munich, Germany*, volume 2114 of *LNCS*, pages 71–82. Springer, 2001.

[9] S. Orlando, P. Palmerini, and R. Perego. On Statistical Properties of Transactional Datasets. In *2004 ACM Symposium on Applied Computing (SAC 2004)*, 2004. To appear.

[10] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive and Resource-Aware Mining of Frequent Sets. In *Proc. The 2002 IEEE International Conference on Data Mining (ICDM'02)*, pages 338–345, 2002.

[11] J. S. Park, M.-S. Chen, and P. S. Yu. An Effective Hash Based Algorithm for Mining Association Rules. In *Proc. of the 1995 ACM SIGMOD Int. Conf. on Management of Data*, pages 175–186, 1995.

[12] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. *Lecture Notes in Computer Science*, 1540:398–416, 1999.

[13] J. Pei, J. Han, H. Lu, S. Nishio, and D. Tang, S. amd Yang. H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases. In *Proc. The 2001 IEEE International Conference on Data Mining (ICDM'01)*, San Jose, CA, USA, 2000.

[14] M. J. Zaki and K. Gouda. Fast Vertical Mining Using Diffsets. In *9th Int. Conf. on Knowledge Discovery and Data Mining, Washington, DC*, 2003.

APRIORI, A Depth First Implementation

Walter A. Kosters
Leiden Institute of Advanced Computer Science
Universiteit Leiden
P.O. Box 9512, 2300 RA Leiden
The Netherlands
kosters@liacs.nl

Wim Pijls
Department of Computer Science
Erasmus University
P.O. Box 1738, 3000 DR Rotterdam
The Netherlands
pijls@few.eur.nl

Abstract

We will discuss $\mathcal{DF}$, the depth first implementation of APRIORI *as devised in 1999 (see [8]). Given a database, this algorithm builds a trie in memory that contains* all *frequent itemsets, i.e., all sets that are contained in at least* minsup *transactions from the original database. Here* minsup *is a threshold value given in advance. In the trie, that is constructed by adding one item at a time, every path corresponds to a unique frequent itemset. We describe the algorithm in detail, derive theoretical formulas, and provide experiments.*

1 Introduction

In this paper we discuss the depth first ($\mathcal{DF}$, see [8]) implementation of APRIORI (see [1]), one of the fastest known data mining algorithms to find *all* frequent itemsets in a large database, i.e., all sets that are contained in at least *minsup* transactions from the original database. Here *minsup* is a threshold value given in advance. There exist many implementations of APRIORI (see, e.g., [6, 11]). We would like to focus on algorithms that assume that the whole database fits in main memory, this often being the state of affairs; among these, $\mathcal{DF}$ and $\mathcal{FP}$ (the FP-growth implementation of APRIORI, see [5]) are the fastest. In most papers so far little attention has been given to theoretical complexity. In [3, 7] a theoretical basis for the analysis of these two algorithms was presented.

The depth first algorithm is a simple algorithm that proceeds as follows. After some preprocessing, which involves reading the database and a sorting of the single items with respect to their support, $\mathcal{DF}$ builds a trie in memory, where every path from the root downwards corresponds to a unique frequent itemset; in consecutive steps items are added to this trie one at a time. Both the database and the trie are kept in main memory, which might cause memory problems: both are usually very large, and in particular the trie gets much larger as the support threshold decreases. Finally the algorithm outputs all paths in the trie, i.e., all frequent itemsets. Note that once completed, the trie allows for fast itemset retrieval in the case of online processing.

We formerly had two implementations of the algorithm, one being time efficient, the other being memory efficient (called `dftime.cc` and `dfmemory.cc`, respectively), where the time efficient version could not handle low support thresholds. The newest version (called `dffast.cc`) combines them into one even faster implementation, and runs for all support thresholds.

In this paper we first describe $\mathcal{DF}$, we then give some formal definitions and theoretical formulas, we discuss the program, provide experimental results, and conclude with some remarks.

2 The Algorithm

An appropriate data structure to store the frequent itemsets of a given database is a *trie*. As a running example in this section we use the dataset of Figure 1. Each line represents a transaction. The trie of frequent patterns is shown in Figure 2. The entries (or cells) in a node of a trie are usually called *buckets*, as is also the case for a hash-tree. Each bucket can be identified with its path to the root and hence with a unique frequent itemset. The example trie has 9 nodes and 18 buckets, representing 18 frequent itemsets. As an example, the frequent itemset $\{A, B, E, F\}$ can be seen as the leftmost path in the trie; and a set as $\{A, B, C\}$ is not present.

One of the oldest algorithms for finding frequent patterns is APRIORI, see [1]. This algorithm successively finds all frequent 1-itemsets, all frequent 2-itemsets, all frequent 3-itemsets, and so on. (A k-itemset has k items.) The frequent k-itemsets are used to generate candidate $(k+1)$-itemsets,

Dataset

transaction number	items
1	B C D
2	A B E F
3	A B E F
4	A B C F
5	A B C E F
6	C D E F

Frequent itemsets when *minsup* $= 3$

support	frequent itemsets
5	B, F
4	A, AB, AF, ABF, BF, C, E, EF
3	AE, ABE, ABEF, AEF, BC, BE, BEF, CF

Figure 1. An example of a dataset along with its frequent itemsets.

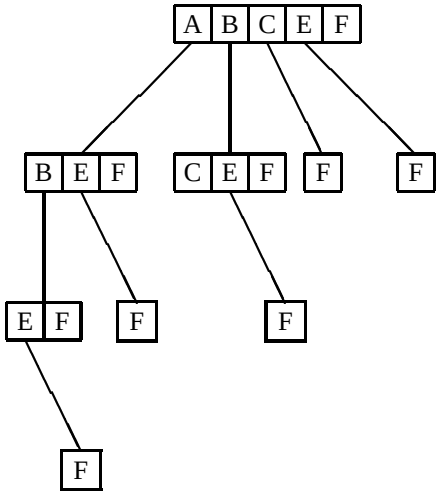


Figure 2. An example of a trie (without support counts).

where the candidates are only known to have two frequent subsets with k elements. After a pruning step, where candidates still having infrequent subsets are discarded, the support of the candidates is determined. The way APRIORI finds the frequent patterns implies that the trie is built layer by layer. First the nodes in the root (depth $= 0$) are constructed, next the trie nodes at depth 1 are constructed, and so on. So, APRIORI can be thought of as an algorithm that builds the pattern trie in a *breadth first* way. We propose an algorithm that builds the trie in a *depth first* way. We will explain the depth first construction of the trie using the dataset of Figure 1. Note that the trie grows from right to left.

The algorithm proceeds as follows. In a preprocessing step, the support of each single item is counted and the infrequent items are eliminated. Let the n frequent items be denoted by $i_1, i_2, \ldots, i_n$. Next, the code from Figure 3 is executed.

```
(1)  T := the trie including only bucket i_n;
(2)  for m := n − 1 downto 1 do
(3)    T' := T;
(4)    T := T' with i_m added to the left and
           a copy of T' appended to i_m;
(5)    S := T\T' (= the subtrie rooted in i_m);
(6)    count(S, i_m);
(7)    delete the infrequent itemsets from S;

(9)  procedure count(S, i_m) ::
(10) for every transaction t including item i_m do
(11)   for every itemset I in S do
(12)     if t supports I then I.support++;
```

Figure 3. The algorithm.

The procedure *count*(S, i_m) determines the support of each itemset (bucket) in the subtrie S. This is achieved by a database pass, in which each transaction including item i_m is considered. Any such transaction is one at a time "pushed" through S, where it only traverses a subtrie if it includes the root of this subtrie, meanwhile updating the support fields in the buckets. In the last paragraph from Section 4 a refinement of this part of the algorithm is presented. On termination of the algorithm, T exactly contains the frequent itemsets.

Figure 4 illustrates the consecutive steps of the algorithm applied to our example. The single items surpassing the minimum support threshold 3 are $i_1 = A, i_2 = B, i_3 = C, i_4 = E$ and $i_5 = F$. In the figure, the shape of T after each iteration of the **for** loop is shown. Also the infrequent itemsets to be deleted at the end of an iteration are mentioned. At the start of the iteration with index m, the root of trie T consists of the 1-itemsets $i_{m+1}, \ldots, i_n$. (We denote a 1-itemset by the name of its only item, omitting curly braces and commas as in Figure 1 and Figure 4.) By the statement in line *(3)* from Figure 3, this trie may also be referred to as T'. A new trie T is composed by adding bucket i_m to the root and by appending a copy of T' (the former value of T) to i_m. The newly added buckets are the new candidates and they make up a subtrie S. In Figure 4, the candidate set S is in the left part of each trie and is drawn in bold. Notice that

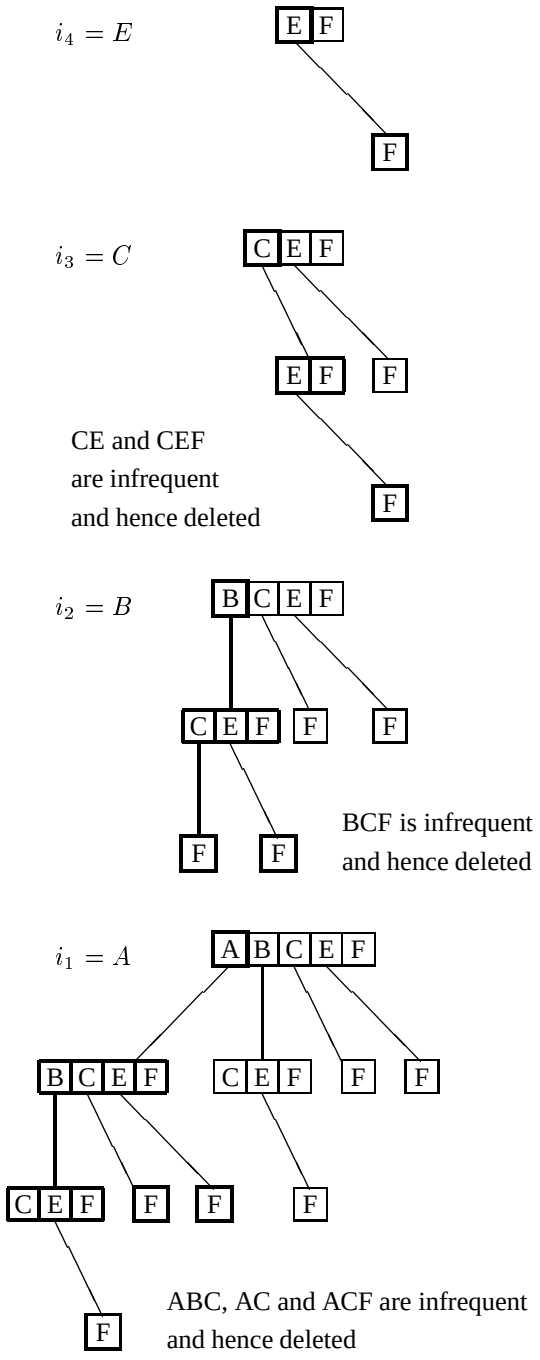


Figure 4. Illustrating the algorithm.

the final trie (after deleting infrequent itemsets) is identical to Figure 2.

The number of iterations in the **for** loop is one less than the number of frequent 1-itemsets. Consequently, the number of database passes is one less than the number of frequent 1-itemsets. This causes the algorithm to be tractable only if the database under consideration is memory-resident. Given the present-day memory sizes, this is not a real constraint any more.

As stated above, our algorithm has a preprocessing step which counts the support for each single item. After this preprocessing step, the items may be re-ordered. The most favorable execution time is achieved if we order the items by increasing frequency (see Section 3 for a more formal motivation). It is better to have low support at the top of the deeper side (to the left bottom) of the trie and hence, high support at the top of the shallow part (to the upper right) of the trie.

We may distinguish between "dense" data sets and "sparse" datasets. A dense dataset has many frequent patterns of large size and high support, as is the case for test sets such as `chess` and `mushroom` (see Section 5). In those datasets, many transactions are similar to each other. Datasets with mainly short patterns are called sparse. Longer patterns may exist, but with relatively small support. Real-world transaction databases of supermarkets mostly belong to this category. Also the synthetic datasets from Section 5 have similar properties: interesting support thresholds are much lower than in the dense case.

Algorithms for finding frequent patterns may be divided into two types: algorithms respectively with and without candidate generation Any APRIORI-like instance belongs to the first type. Eclat (see [9]) may also be considered as an instance of this type. The FP-growth algorithm $\mathcal{FP}$ from [5] is the best-known instance of the second type (though one can also defend the point of view that it does generate candidates). For dense datasets, $\mathcal{FP}$ performs better than candidate generating algorithms. $\mathcal{FP}$ stores the dataset in a way that is very efficient especially when the dataset has many similar transactions. In case of algorithms that do apply candidate generation, dense sets produce a large number of candidates. Since each new candidate has to be related to each transaction, the database passes take a lot of time. However, for sparse datasets, candidate generation is a very suitable method for finding frequent patterns. To our experience, the instances of the APRIORI family are very useful when searching transaction databases. According to the results in [7] the depth first algorithm $\mathcal{DF}$ outperforms FP-growth $\mathcal{FP}$ in the synthetic transaction sets (see Section 5 for a description of these sets).

Finally, note that technically speaking $\mathcal{DF}$ is not a full implementation of APRIORI, since every candidate itemset is known to have only one frequent subset (resulting from the part of the trie which has already been completed) instead of two. Apart from this, its underlying candidate generation mechanism strongly resembles the one from APRIORI.

3 Theoretical Complexity

Let m denote the number of transactions (also called customers), and let n denote the number of products (also called items). Usually m is much larger than n. For a non-empty itemset $A \subseteq \{1, 2, \ldots, n\}$ we define:

- $supp(A)$ is the *support* of A: the number of customers that buy all products from A (and possibly more), or equivalently the number of transactions that contain A;
- $sm(A)$ is the smallest number in A;
- $la(A)$ is the largest number in A.

In line with this we let $supp(\emptyset) = m$. We also put $la(\emptyset) = 0$ and $sm(\emptyset) = n + 1$. A set $A \subseteq \{1, 2, \ldots, n\}$ is called *frequent* if $supp(A) \geq minsup$, where the so-called *support threshold minsup* is a fixed number given in advance.

We assume every 1-itemset to be frequent; this can be effected by the first step of the algorithms we are looking at, which might be considered as preprocessing.

A "database query" is defined as a question of the form "Does customer C buy product P?" (or "Does transaction T has item I?"), posed to the original database. Note that we have mn database queries in the "preprocessing" phase in which the supports of the 1-itemsets are computed and ordered: every field of the database is inspected once. (By the way, the sorting, in which the items are assigned the numbers $1, 2, \ldots, n$, takes $O(n \log n)$ time.) The number of database queries for $\mathcal{DF}$ equals:

$$m(n-1) + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} supp(\{j\} \cup A \setminus \{la(A)\}) \quad . \quad (1)$$

For a proof, see [3]. It relies on the fact that in order for a node to occur in the trie the path to it (except for the root) should be frequent, and on the observation that this particular node is "questioned" every time a transaction follows this same path. In [3] a simple version of $\mathcal{FP}$ is described in a similar style, leading to

$$\sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{\substack{j=la(A)+1 \\ \{j\} \cup A \setminus \{la(A)\} \text{ frequent}}}^{n} supp(A) \quad (2)$$

database queries in "local databases" (FP-trees), except for the preprocessing phase. Note the extra condition on the inner summation (which is "good" for $\mathcal{FP}$: we have less summands there), while on the other hand the summands are larger (which is "good" for $\mathcal{DF}$: we have a smaller contribution there).

It makes also sense to look at the total number of nodes of the trie during its construction, which is connected to the effort of maintaining and using the datastructures. Counting each trie-node with the number of buckets it contains, the total is computed to be:

$$n + \sum_{\substack{A \neq \emptyset \\ A \text{ frequent}}} \sum_{j=1}^{sm(A)-1} 1 = \sum_{A \text{ frequent}} [sm(A) - 1] \quad . \quad (3)$$

When the trie is finally ready the number of remaining buckets equals the number of frequent sets, each item in a node being the end of the path that represents the corresponding itemset.

Notice that the complexity heavily depends on the sorting order of the items at the top level. It turns out that an increasing order of items is beneficial here. This is suggested by the contribution of the 1-itemsets in Equation (1):

$$\sum_{i=1}^{n} (n - i)\ supp(\{i\}) \quad , \quad (4)$$

which happens to be minimal in that case. This 1-itemset contribution turns out to be the same for both $\mathcal{DF}$ and $\mathcal{FP}$: see [3, 7], where also results for $\mathcal{FP}$ are presented in more detail.

4 Implementation Issues

In this section we discuss some implementation details of our program. As mentioned in Section 2, the database is traversed many times. It is therefore necessary that the database is memory-resident. Fortunately, only the occurrences of frequent items need to be stored. The database is represented by a two-dimensional boolean array. For efficiency reasons, one array entry corresponds to one bit. Since the function *count* in the algorithm considers the database transaction by transaction, a horizontal layout is chosen, cf. [4].

We have four preprocessing steps before the algorithm of Section 2 actually starts.

1 The range of the item values is determined. This is necessary, because some test sets, e.g., the BMS-WebView sets, have only values $> 10,000$.

2 This is an essential initial step. First, for each item the support is counted. Next, the frequent items are selected and sorted by frequency. This process is relevant, since the frequency order also prescribes the order in the root of the trie, as stated before. The sorted frequent items along with their supports are retained in an array.

3 If a transaction has zero or one frequent item, it needs not to be stored into the memory-resident representation of the database. The root of the trie is constructed

according to the information gathered in step 2. For constructing the other buckets, only transactions with at least two frequent items are relevant. In this step, we count the relevant transactions.

4 During this step the databases is stored into a two-dimensional array with horizontal layout. Each item is given a new number, according to its rank in the frequency order. The length of the array equals the result of step 3; the width is determined by the number of frequent items.

After this preparatory work, which in practice usually takes a few seconds, the code as described in Section 2 is executed. The cells of the root are constructed using the result of initial step 2.

In line *(12)* from Figure 3 in Section 2, backtracking is applied to inspect each path P of S. Inspecting a path P is aborted as soon as an item i with i outside the current transaction t is found. Obviously, processing one transaction during the *count* procedure is a relatively expensive task, which is unfortunately inevitable, whichever version of APRIORI is used.

As mentioned in the introduction, we used to have two implementations, one being time efficient, the other being memory efficient. These two have been used in the overall FIMI'03 comparisons. The newest implementation (called `dffast.cc`) combines these versions by using the following refinement. Instead of appending a copy T' of T to i_m (see Figure 3 in Section 2), first the counting is done in auxiliary fields in the original T, after which only the frequent buckets are copied underneath i_m. This makes the deletion of infrequent itemsets (line *(7)* from Figure 3) unnecessary and leads to better memory management. Another improvement might be achieved by using more auxiliary fields while adding two root items simultaneously in each iteration, thereby halving the number of database passes at the cost of more bookkeeping.

5 Experiments

Using the relatively small database `chess` (342 kB, with 3,196 transactions; available from the FIMI'03 website at `http://fimi.cs.helsinki.fi/testdata.html`), the database `mushroom` (570 kB, with 8,124 transactions; also available from the FIMI'03 website) and the well-known IBM-Almaden synthetic databases (see [2]) we shall examine the complexity of the algorithm. These databases have either few, but coherent records (`chess` and `mushroom`), or many records (the synthetic databases). The parameters for generating a synthetic database are the number of transactions D (in thousands), the average transaction size T and the average length I of so-called maximal potentially large itemsets. The number of items was set to $N = 1{,}000$, following the design in [2]. We use `T10I4D100K` (4.0 MB) and `T40I10D100K` (15.5 MB), both also available from the FIMI'03 website mentioned above; they both contain 100,000 transactions.

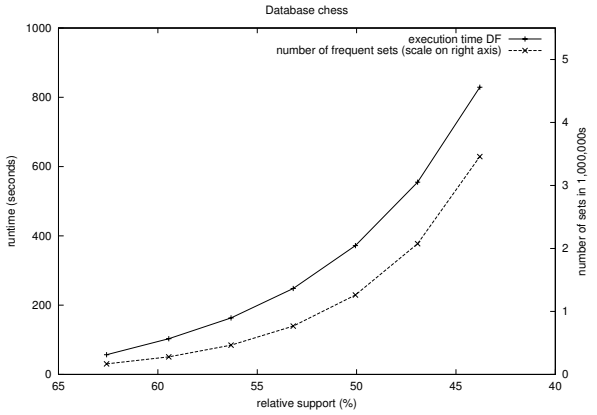


Figure 5. Experimental results for database `chess`.

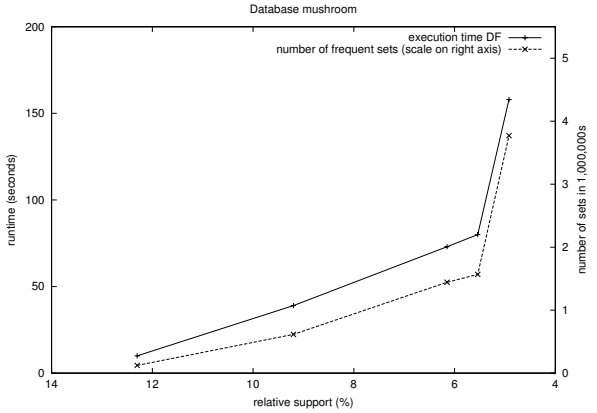


Figure 6. Experimental results for database `mushroom`.

The experiments were conducted at a Pentium-IV machine with 512 MB memory at 2.8 GHz, running Red Hat Linux 7.3. The program was developed under the GNU C ++ compiler, version 2.96.

The following statistics are plotted in the graphs: the execution time in seconds of the algorithm (see Section 4), and the total number of frequent itemsets: in all figures the corresponding axis is on the right hand side and scales 0–5,500,000 (0–8,000,000 for `T10I4D100K`). The execution time excludes preprocessing: in this phase the database is read three times in order to detect the frequent items (see

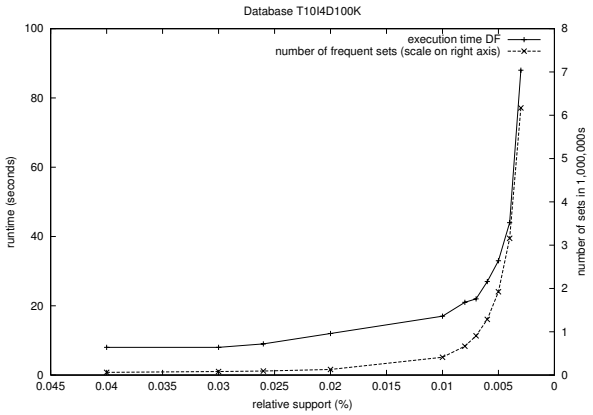


Figure 7. Experimental results for database `T10I4D100K`.

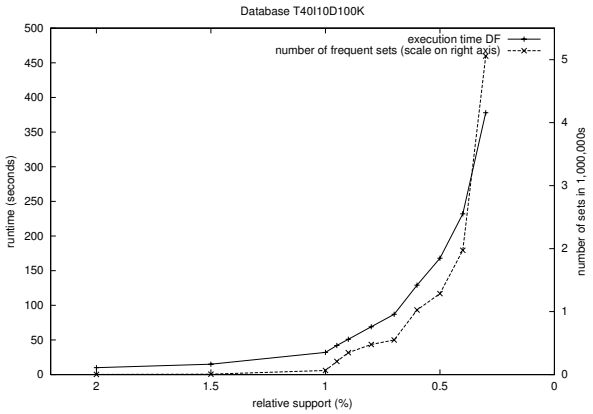


Figure 8. Experimental results for database `T40I10D100K`.

before); also excluded is the time needed to print the resulting itemsets. These actions together usually only take a few seconds. The number of frequent 1-itemsets (n from the previous sections, where we assumed all 1-itemsets to be frequent) has range 31–39 for the experiments on the database `chess`, 54–76 for `mushroom`, 844–869 for `T10I4D100K` and 610–862 for `T40I10D100K`. Note the very high support thresholds for `mushroom` (at least 5%) and `chess` (at least 44%); for `T10I4D100K` a support threshold as low as 0.003% was even feasible.

The largest output files produced are of size 110.6 MB (`chess`, *minsup* = 1,400, having 3,771,728 frequent sets with 13 frequent 17-itemsets), 121.5 MB (`mushroom`, *minsup* = 400, having 3,457,747 frequent sets with 24 frequent 17-itemsets), 131.1 MB (`T10I4D100K`, *minsup* = 3, having 6,169,854 frequent sets with 30 frequent 13-itemsets and 1 frequent 14-itemset) and 195.9 MB (`T40I10D100K`, *minsup* = 300, having 5,058,313 frequent sets, with 21 frequent 19-itemsets and 1 frequent 20-itemset). The final trie in the `T40I10D100K` case occupies approximately 65 MB of memory — the output file in this case being 3 times as large.

Note that the 3,457,747 sets for the `chess` database with *minsup* = 1,400 require 829 seconds to find, whereas the 3,771,728 frequent itemsets for `mushroom` with *minsup* = 400 take 158 seconds — differing approximately a factor 5 in time. This difference in runtime is probably caused by the difference in the absolute *minsup* value. Each cell corresponding to a frequent itemset is visited at least 1400 times in the former case against 400 times in the latter case. A similar phenomenon is observed when comparing `T40I10D100K` with absolute *minsup* value 300 and `T10I4D100K` with *minsup* = 3: this takes 378 versus 88 seconds. Although the outputs have the same orders of magnitude, the runtimes differ substantially. We see that, besides the number of frequent itemsets and the sizes of these sets, the absolute *minsup* value is a major factor determining the runtime.

6 Conclusions

In this paper, we addressed $\mathcal{DF}$, a depth first implementation of APRIORI. To our experience, $\mathcal{DF}$ competes with any other well-known algorithm, especially when applied to large databases with transactions.

Storing the database in the primary memory is no longer a problem. On the other hand, storing the candidates causes trouble in situations, where a dense database is considered with a small support threshold. This is the case for any algorithm using candidates. Therefore, it would be desirable to look for a method which stores candidates in secondary memory. This is an obvious topic for future research. To our knowledge, $\mathcal{FP}$ is the only algorithm that can cope with memory limitations. However, for real world retail databases this algorithm is surpassed by $\mathcal{DF}$, as we showed in [7]. Other optimizations might also be possible. Besides improving the C++ code, ideas from, e.g., [10] on diffsets with vertical layouts might be used.

Our conclusion is that $\mathcal{DF}$ is a simple, practical, straightforward and fast algorithm for finding all frequent itemsets.

References

[1] R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In U.M. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In J.B. Bocca, M. Jarke, and C. Zaniolo, editors, *Proceedings 20th International Conference on Very Large Data Bases, VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[3] J.M. de Graaf, W.A. Kosters, W. Pijls, and V. Popova. A theoretical and practical comparison of depth first and FP-growth implementations of Apriori. In H. Blockeel and M. Denecker, editors, *Proceedings of the Fourteenth Belgium-Netherlands Artificial Intelligence Conference (BNAIC 2002)*, pages 115–122, 2002.

[4] B. Goethals. Survey on frequent pattern mining. Helsinki, 2003. `http://www.cs.helsinki.fi/u/goethals/publications/survey.ps`.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD'00)*, pages 1–12, 2000.

[6] J. Hipp, U. Günther, and G. Nakhaeizadeh. Mining association rules: Deriving a superior algorithm by analyzing today's approaches. In D.A. Zighed, J. Komorowski, and J. Żytkov, editors, *Principles of Data Mining and Knowledge Discovery, Proceedings of the 4th European Conference (PKDD 2000)*, Springer Lecture Notes in Computer Science 1910, pages 159–168. Springer Verlag, 2000.

[7] W.A. Kosters, W. Pijls, and V. Popova. Complexity analysis of depth first and FP-growth implementations of Apriori. In P. Perner and A. Rosenfeld, editors, *Machine Learning and Data Mining in Pattern Recognition, Proceedings MLDM 2003*, Springer Lecture Notes in Artificial Intelligence 2734, pages 284–292. Springer Verlag, 2003.

[8] W. Pijls and J.C. Bioch. Mining frequent itemsets in memory-resident databases. In E. Postma and M. Gyssens, editors, *Proceedings of the Eleventh Belgium-Netherlands Conference on Artificial Intelligence (BNAIC1999)*, pages 75–82, 1999.

[9] M.J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12:372–390, 2000.

[10] M.J. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2003.

[11] Z. Zheng, R. Kohavi, and L. Mason. Real world performance of association rule algorithms. In F. Provost and R. Srikant, editors, *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2001)*, pages 401–406, 2001.

AFOPT: An Efficient Implementation of Pattern Growth Approach*

Guimei Liu Hongjun Lu
Department of Computer Science
Hong Kong University of
Science & Technology
Hong Kong, China
{cslgm, luhj}@cs.ust.hk

Jeffrey Xu Yu
Department of Systems Engineering and
Engineering Management
The Chinese University of Hong Kong
Hong Kong, China
{yu}@se.cuhk.edu.hk

Wei Wang Xiangye Xiao
Department of Computer Science
Hong Kong University of
Science & Technology
Hong Kong, China
{fervvac, xiaoxy}@cs.ust.hk

Abstract

In this paper, we revisit the frequent itemset mining (FIM) problem and focus on studying the pattern growth approach. Existing pattern growth algorithms differ in several dimensions: (1) item search order; (2) conditional database representation; (3) conditional database construction strategy; and (4) tree traversal strategy. They adopted different strategies on these dimensions. Several adaptive algorithms were proposed to try to find good strategies for general situations. In this paper, we described the implementation techniques of an adaptive pattern growth algorithm, called AFOPT, which demonstrated good performance on all tested datasets. We also extended the algorithm to mine closed and maximal frequent itemsets. Comprehensive experiments were conducted to demonstrate the efficiency of the proposed algorithms.

1 Introduction

Since the frequent itemset mining problem (FIM) was first addressed [2], a large number of FIM algorithms have been proposed. There is a pressing need to completely characterize and understand the algorithmic performance space of FIM problem so that we can choose and integrate the best strategies to achieve good performance in general cases.

Existing FIM algorithms can be classified into two categories: the candidate generate-and-test approach and the pattern growth approach. In each iteration of the candidate generate-and-test approach, pairs of frequent k-itemsets are joined to form candidate (k+1)-itemsets, then the database is scanned to verify their supports. The resultant frequent (k+1)-itemsets will be used as the input for next iteration. The drawbacks of this approach are: (1) it needs scan database multiple times, in worst case, equal to the maximal length of the frequent itemsets; (2) it needs generate lots of candidate itemsets, many of which are proved to be infrequent after scanning the database; and (3) subset checking is a cost operation, especially when itemsets are very long. The pattern growth approach avoids the cost of generating and testing a large number of candidate itemsets by growing a frequent itemset from its prefix. It constructs a conditional database for each frequent itemset t such that all the itemsets that have t as prefix can be mined only using the conditional database of t.

The basic operations in the pattern growth approach are *counting frequent items* and *new conditional databases construction*. Therefore, the number of conditional databases constructed during the mining process, and the mining cost of each individual conditional database have a direct effect on the performance of a pattern growth algorithm. The total number of conditional databases mainly depends on in what order the search space is explored. The traversal cost and construct cost of a conditional database depends on the size, the representation format (tree-based or array-based) and construction strategy (physical or pseudo) of the conditional database. If the conditional databases are represented by tree structure, the traversal strategy of the tree structure also matters. In this paper, we investigate various aspects of the pattern growth approach, and try to find out what are good strategies for a pattern growth algorithm.

The rest of the paper is organized as follows: Section 2 revisits the FIM problem and introduces some related works; In Section 3, we describe an efficient pattern growth algorithm—AFOPT; Section 4 and Section 5 extend the AFOPT algorithm to mine frequent closed itemsets and maximal frequent itemsets respectively; Section 6 shows experiment results; finally, Section 7 concludes this paper.

2 Problem Revisit and Related Work

In this section, we first briefly review FIM problem and the candidate generate-and-test approach, then focus on studying the algorithmic performance space of the pattern growth approach.

*This work was partly supported by the Research Grant Council of the Hong Kong SAR, China (Grant HKUST6175/03E, CUHK4229/01E).

2.1 Problem revisit

Given a transactional database D, let I be the set of items appearing in it. Any combination of the items in I can be frequent in D, and they form the search space of FIM problem. The search space can be represented using set enumeration tree [14, 1, 4, 5, 7]. For example, given a set of items $I = \{a, b, c, d, e\}$ sorted in lexicographic order, the search space can be represented by a tree as shown in Figure 1. The root of the search space tree represents the empty set, and each node at level l (the root is at level 0, and its children are at level 1, and so on) represents an l-itemset. The *candidate extensions* of an itemset p is defined as the set of items after the last item of p. For example, items d and e are candidate extensions of ac, while b is not a candidate extension of ac because b is before c. The frequent extensions of p are those candidate extensions of p that can be appended to p to form a longer frequent itemset. In the rest of this paper, we will use $cand\_exts(p)$ and $freq\_exts(p)$ to denote the set of candidate extensions and frequent extensions of p respectively.

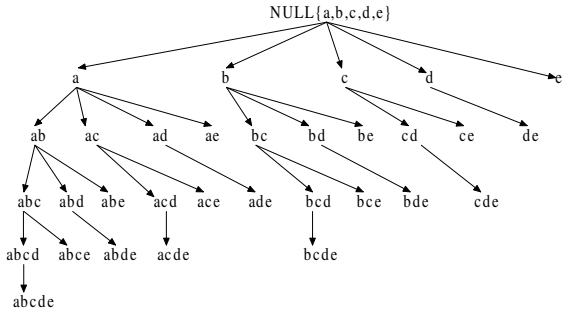


Figure 1. Search space tree

2.2 Candidate generate-and-test approach

Frequent itemset mining can be viewed as a set containment join between the transactional database and the search space of FIM. The candidate generate-and-test approach essentially uses block nested loop join, i.e. the search space is the inner relation and it is divided into blocks according to itemset length. Different from simple block nested loop join, in candidate generate-and-test approach the output of the previous pass is used as seeds to generate next block. For example, in the k-th pass of the Apriori algorithm, the transaction database and the candidate k-itemsets are joined to generate frequent k-itemsets. The frequent k-itemsets are then used to generate next block—candidate (k+1)-itemsets. Given the large amount of memory available nowadays, it is a waste of memory to put only a single length of itemsets into memory. It is desirable to fully utilize available memory by putting some longer and possibly frequent itemsets into memory in earlier stage to reduce the number of database scans. The first FIM algorithm AIS [2] tries to estimate the frequencies of longer itemsets using the output of current pass, and includes those itemsets that are estimated as frequent or themselves are not estimated as frequent but all of its subsets are frequent or estimated as frequent into next block. The problem with AIS algorithm is that it does not fully utilize the pruning power of the Apriori property, thus many unnecessary candidate itemsets are generated and tested. DIC algorithm [3] makes improvements based on Apriori algorithm. It starts counting the support of an itemset shortly after all the subsets of that itemset are determined to be frequent rather than wait until next pass. However, DIC algorithm cannot guarantee the full utilization of memory. The candidate generate-and-test approach faces a trade-off: on one hand, the memory is not fully utilized and it is desirable to put as many as possible candidate itemsets into memory to reduce the number of database scans; on the other hand, set containment test is a costly operation, putting itemsets into memory in earlier stage has the risk of counting support for many unnecessary candidate itemsets.

2.3 Pattern growth approach

The pattern growth approach adopts the divide-and-conquer methodology. The search space is divided into disjoint sub search spaces. For example, the search space shown in Figure 1 can be divided into 5 disjoint sub search spaces: (1) itemsets containing a; (2) itemsets containing b but no a; (3) itemsets containing c but no a, b; (4) itemsets containing d but no a, b and c; and (5) itemsets containing only e. Accordingly, the database is divided into 5 partitions, and each partition is called a conditional database. The conditional database of item i, denoted as D_i, includes all the transactions containing item i. All the items before i are eliminated from each transaction. All the frequent itemsets containing i can be mined from D_i without accessing other information. Each conditional database is divided recursively following the same procedure. The pattern growth approach not only reduces the number of database scans, but also avoids the costly set-containment-test operation.

Two basic operations in pattern growth approach are *counting frequent items* and *new conditional databases construction*. Therefore, the total number of conditional databases constructed and the mining cost of each individual conditional database are key factors that affect the performance of a pattern growth algorithm. The total number of conditional databases mainly depends on in what order the search space is explored. This order is called *item search order* in this paper. Some structures for representing conditional databases can also help reduce the total number of conditional databases. For example, if a conditional database is represented by tree-structure and there is only one branch, then all the frequent itemsets in the conditional database can be enumerated directly from the branch. There is no need to construct new conditional databases. The mining cost of a conditional database depends on the size, the

Datasets	Asc			Lex			Des		
	#cdb	time	max_mem	#cdb	time	max_mem	#cdb	time	max_mem
T10I4D100k (0.01%)	53688	4.52s	5199 kb	47799	4.89s	5471 kb	36725	5.32s	5675 kb
T40I10D100k (0.5%)	311999	30.42s	17206 kb	310295	33.83s	20011 kb	309895	43.37s	21980 kb
BMS-POS (0.05%)	115202	27.83s	17294 kb	53495	127.45s	38005 kb	39413	147.01s	40206 kb
BMS-WebView-1 (0.06%)	33186	0.69s	731 kb	65378	1.12s	901 kb	79571	2.16s	918 kb
chess (45%)	312202	2.68s	574 kb	617401	8.46s	1079 kb	405720	311.19s	2127 kb
connect-4 (75%)	12242	1.31s	38 kb	245663	2.65s	57 kb	266792	14.27s	113 kb
mushroom (5%)	9838	0.34s	1072 kb	258068	3.11s	676 kb	464903	272.30s	2304 kb
pumsb (70%)	272373	3.87s	383 kb	649096	12.22s	570 kb	469983	16.62s	1225 kb

Table 1. Comparison of Three Item Search Orders (Bucket Size=0)

representation and construction strategy of the conditional database. The traversal strategy also matters if the conditional database is represented using a tree-structure.

Item Search Order. When we divide the search space, all items are sorted in some order. This order is called *item search order*. The sub search space of an item contains all the items after it in *item search order* but no item before it. Two item search orders were proposed in literature: static lexicographic order and dynamic ascending frequency order. Static lexicographic order is to order the items lexicographically. It is a fixed order—all the sub search spaces use the same order. Tree projection algorithm [15] and H-Mine algorithm[12] adopted this order. Dynamic ascending frequency order reorders frequent items in every conditional database in ascending order of their frequencies. The most infrequent item is the first item, and all the other items are its candidate extensions. The most frequent item is the last item and it has no candidate extensions. FP-growth [6], AFOPT [9] and most of maximal frequent itemsets mining algorithms [7, 1, 4, 5] adopted this order.

The number of conditional databases constructed by an algorithm can differ greatly using different item search orders. Ascending frequency order is capable of minimizing the number and/or the size of conditional databases constructed in subsequent mining. Intuitively, an itemset with higher frequency will possibly have more frequent extensions than an itemset with lower frequency. We put the most infrequent item in front, though the candidate extension set is large, the frequent extension set cannot be very large. The frequencies of successive items increase, at the same time the size of candidate extension set decreases. Therefore we only need to build smaller and/or less conditional databases in subsequent mining. Table 1 shows the total number of conditional databases constructed (#cdb column), total running time and maximal memory usage when three orders are adopted in the framework of AFOPT algorithm described in this paper. The three item search orders compared are: dynamic ascending frequency order (Asc column), lexicographic order (Lex column) and dynamic descending frequency order (Des column). The minimum support threshold on each dataset is shown in the first column. On the first three datasets, ascending frequency order needs to build more conditional databases than the other two orders, but its total running time and maximal memory usage is less than the other two orders. It implies that the conditional databases constructed using ascending frequency order are smaller. On the remaining datasets, ascending frequency order requires to build less conditional databases and needs less running time and maximal memory usage, especially on dense datasets connect-4 and mushroom.

Agrawal et al proposed an efficient support counting technique, called *bucket counting*, to reduce the total number of conditional databases[1]. The basic idea is that if the number of items in a conditional database is small enough, we can maintain a counter for every combination of the items instead of constructing a conditional database for each frequent item. The bucket counting can be implemented very efficiently compared with conditional database construction and traversal operation.

Conditional Database Representation. The traversal and construction cost of a conditional database heavily depends on its representation. Different data structures have been proposed to store conditional databases, e.g. tree-based structures such as FP-tree [6] and AFOPT-tree [9], and array-based structure such as Hyper-structure [12]. Tree-based structures are capable of reducing traversal cost because duplicated transactions can be merged and different transactions can share the storage of their prefixes. But they incur high construction cost especially when the dataset is sparse and large. Array-based structures incur little construction cost but they need much more traversal cost because the traversal cost of different transactions cannot be shared. It is a trade-off in choosing tree-based structures or array-based structures. In general, tree-based structures are suitable for dense databases because there can be lots of prefix sharing among transactions, and array-based structures are suitable for sparse databases.

Conditional Database Construction Strategy Constructing every conditional database physically can be expensive especially when successive conditional databases do not shrink much. An alternative is to pseudo-construct them, i.e. using pointers pointing to transactions in upper

Algorithms	Item Search Order	CondDB Format	CondDB Construction	Tree Traversal
Tree-Projection [15]	static lexicographic	array	adaptive	-
FP-growth [6]	dynamic frequency	FP-tree	physical	bottom-up
H-mine [12]	static lexicographic	hyper-structure	pseudo	-
OP [10]	adaptive	adaptive	adaptive	bottom-up
PP-mine [17]	static lexicographic	PP-tree	pseudo	top-down
AFOPT [9]	dynamic frequency	adaptive	physical	top-down
CLOSET+ [16]	dynamic frequency	FP-tree	adaptive	adaptive

Table 2. Pattern Growth Algorithms

level conditional databases. However, pseudo-construction cannot reduce traversal cost as effectively as physical construction. The item ascending frequency search order can make the subsequent conditional databases shrink rapidly, consequently it is beneficial to use physical construction strategy with item ascending frequency order together.

Tree Traversal Strategy The traversal cost of a tree is minimal using top-down traversal strategy. FP-growth algorithm [6] uses ascending frequency order to explore the search space, while FP-tree is constructed according to descending frequency order. Hence FP-tree has to be traversed using bottom-up strategy. As a result, FP-tree has to maintain parent links and node links at each node for bottom-up traversal. which increases the construction cost of the tree. AFOPT algorithm [9] uses ascending frequency order both for search space exploration and prefix-tree construction, so it can use the top-down traversal strategy and do not need to maintain additional pointers at each node. The advantage of FP-tree is that it can be more compact than AFOPT-tree because descending frequency order increases the possibility of prefix sharing. The ascending frequency order adopted by AFOPT may lead to many single branches in the tree. This problem was alleviated by using arrays to store single branches in AFOPT-tree.

Existing pattern growth algorithms mainly differ in the several dimensions aforementioned. Table 2 lists existing pattern growth algorithms and their strategies on four dimensions. AFOPT [9] is an efficient FIM algorithm developed by our group. We will discuss its technical details in next three sections.

3 Mining All Frequent Itemsets

We discussed several trade-offs faced by a pattern growth algorithm in last section. Some implications from above discussions are: (1) Use tree structure on dense database and use array structure on sparse database. (2) Use dynamic ascending frequency order on dense databases and/or when minimum support threshold is low. It can dramatically reduce the number and/or the size of the successive conditional databases. (3) If dynamic ascending frequency order is adopted, then use physical construction strategy because the size of conditional databases will shrink quickly. In this section, we describe our algorithm AFOPT which takes the above three implications into consideration. The distinct features of our AFOPT algorithm include: (1) It uses three different structures to represent conditional databases: arrays for sparse conditional databases, AFOPT-tree for dense conditional databases, and buckets for counting frequent itemsets containing only top-k frequent items, where k is a parameter to control the number of buckets used. Several parameters are introduced to control when to use arrays or AFOPT-tree. (2) It adopts the dynamic ascending frequency order. (3) The conditional databases are constructed physically on all levels no matter whether the conditional databases are represented by AFOPT-tree or arrays.

3.1 Framework

Given a transactional database D and a minimum support threshold, AFOPT algorithm scans the original database twice to mine all frequent itemsets. In the first scan, all frequent items in D are counted and sorted in ascending order of their frequencies, denoted as $F = \{i_1, i_2, \cdots, i_m\}$. We perform another database scan to construct a *conditional database* for each $i_j \in F$, denoted as D_{i_j}. During the second scan, infrequent items in each transaction t are removed and the remaining items are sorted according to their orders in F. Transaction t is put into D_{i_j} if the first item of t after sorting is i_j. The remaining mining will be performed on conditional databases only. There is no need to access the original database.

We first perform mining on D_{i_1} to mine all the itemsets containing i_1. Mining on individual conditional database follows the same process as mining on the original database. After the mining on D_{i_1} is finished, D_{i_1} can be discarded. Because D_{i_1} also contains other items, the transactions in it will be inserted into the remaining conditional databases. Given a transaction t in D_{i_1}, suppose the next item after i_1 in t is i_j, then t will be inserted into D_{i_j}. This step is called push-right. Sorting the items in ascending order of their frequencies ensures that every time, a small conditional database is pushed right. The pseudo-code of AFOPT-all algorithm is shown in Algorithm 1.

3.2 Conditional database representation

Algorithm 1 is independent of the representation of conditional databases. We choose proper representations ac-

Algorithm 1 AFOPT-all Algorithm

Input:
p is a frequent itemset
D_p is the conditional database of p
min_sup is the minimum support threshold;

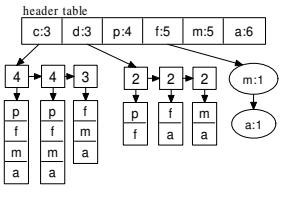
Description:
1: Scan D_p count frequent items, F=$\{i_1, i_2, \cdots, i_n\}$;
2: Sort items in F in ascending order of their frequencies;
3: **for all** item $i \in F$ **do**
4: $D_{p \bigcup \{i\}} = \phi$;
5: **for all** transaction $t \in D_p$ **do**
6: remove infrequent items from t, and sort remaining items according to their orders in F;
7: let i be the first item of t, insert t into $D_{p \bigcup \{i\}}$.
8: **for all** item $i \in F$ **do**
9: Output $s = p \bigcup \{i\}$;
10: AFOPT-all(s, D_s, min_sup);
11: PushRight(D_s);

TID	Transactions
1	a, b, c, f, m, p
2	a, d, e, f, g
3	a, b, f, m, n
4	a, c, e, f, m, p
5	d, f, n, p
6	a, c, h, m, p
7	a, d, m, s

(a) D

TID	Transactions
1	c, p, f, m, a
2	d, f, a
3	f, m, a
4	c, p, f, m, a
5	d, p, f
6	c, p, m, a
7	d, m, a

(b)



(c)

Figure 2. Conditional DB Representation

cording to the density of conditional databases. Three structures are used: (1) array, (2) AFOPT-tree, and (3) buckets. As aforementioned, these three structures are suitable for different situations. Bucket counting technique is proper and extremely efficient when the number of distinct frequent items is around 10. Tree structure is beneficial when conditional databases are dense. Array structure is favorable when conditional databases are sparse. We use four parameters to control when to use these three structures as follows: (1) frequent itemsets containing only top-$bucket\_size$ frequent items are counted using buckets; (2) if the minimum support threshold is greater than $tree\_min\_sup$ or average support of all frequent items is no less than $tree\_avg\_sup$, then all the rest conditional databases are represented using AFOPT-tree; otherwise (3) the conditional databases of the next $tree\_alphabet\_size$ most frequent items are represented using AFOPT-tree, and the rest conditional databases are represented using arrays.

Figure 2 shows a transactional database D and the initial conditional databases constructed with min_sup=40%. There are 6 frequent items $\{c$:3, d:3, p:4, f:5, m:5,a:6$\}$. Figure 2(b) shows the projected database after removing infrequent items and sorting. The values of the parameters for conditional database construction are set as follows: $bucket\_size$=2, $tree\_alphabet\_size$=2, $tree\_min\_sup$=50%, $tree\_avg\_sup$=60%. The frequent itemsets containing only m and a are counted using buckets of size 4 (=2^{bucket_size}). The conditional databases of f and p are represented by AFOPT-tree. The conditional databases of item c and d are represented using arrays. From our experience, the $bucket\_size$ parameter can choose a value around 10. A value between 20 and 200 will be safe for $tree\_alphabet\_size$ parameter. We set $tree\_min\_sup$ to 5% and $tree\_avg\_sup$ to 10% in our experiments.

Table 3 shows the size, construction time (build column) and push-right time if applicable, of the initial structure constructed from original database by AFOPT, H-Mine and FP-growth algorithms. We set $bucket\_size$ to 8 and $tree\_alphabet\_size$ to 20 for AFOPT algorithm. The initial structure of AFOPT includes all three structures. The array structure in AFOPT algorithm simply stores all items in a transaction. Each node in hyper-structure stores three pieces of information: an item, a pointer pointing to the next item in the same transaction and a pointer pointing to the same item in another transaction. Therefore the size of hyper-structure is approximately 3 times larger than the array structure used in AFOPT. A node in AFOPT-tree maintains only a child pointer and a sibling pointer, while a FP-tree node maintains two more pointers for bottom-up traversal: a parent pointer and a node link. AFOPT consumes the least amount of space on almost all tested datasets.

4 Mining Frequent Closed Itemsets

The complete set of frequent itemsets can be very large. It has been shown that it contains many redundant information [11, 18]. Some works [11, 18, 13, 16, 8] put efforts on mining frequent closed itemsets to reduce output size. *An itemset is closed if all of its supersets have a lower support than it*. The set of frequent closed itemsets is the minimum informative set of frequent itemsets. In this section, we describe how to extend Algorithm 1 to mine only frequent closed itemsets. For more details, please refer to [8].

4.1 Removing non-closed itemsets

Non-closed itemsets can be removed either in a postprocessing phase, or during mining process. The second strategy can help avoid unnecessary mining cost. Non-closed frequent itemsets are removed based on the following two lemmas (see [8] for proof of these two lemmas).

Lemma 1 *In Algorithm 1, an itemset p is closed if and only if two conditions hold: (1) no existing frequent itemsets is a superset of p and is as frequent as p; (2) all the items in D_p have a lower support than p.*

Lemma 2 *In Algorithm 1, if a frequent itemset p is not closed because condition (1) in Lemma 1 does not hold, then none of the itemsets mined from D_p can be closed.*

We check whether there exists q such that $p \subset q$ and $sup(p)$=$sup(q)$ before mining D_p. If such q exists, then there is no need to mine D_p based on Lemma 2 (line 10).

Datasets	AFOPT			H-Mine			FP-growth	
	size	build	pushright	Size	build	pushright	size	build
T10I4D100k (0.01%)	5116 kb	0.55s	0.37s	11838 kb	0.68s	0.19s	20403 kb	1.83s
T40I10D100k (0.5%)	16535 kb	1.85s	1.91s	46089 kb	2.10s	1.42s	104272 kb	6.16s
BMS-POS (0.05%)	17264 kb	2.11s	1.43s	38833 kb	2.58s	1.00s	47376 kb	6.64s
BMS-WebView-1 (0.06%)	711 kb	0.12s	0.01s	1736 kb	0.17s	0.01s	1682 kb	0.27s
chess (45%)	563 kb	0.04s	0.01s	1150 kb	0.05s	0.03s	1339 kb	0.12s
connect-4 (75%)	35 kb	0.73s	0.01s	22064 kb	1.15s	0.55s	92 kb	2.08s
mushroom (5%)	1067 kb	0.08s	0.04s	2120 kb	0.10s	0.03s	988 kb	0.17s
pumsb (70%)	375 kb	0.82s	0.02s	17374 kb	1.15s	0.43s	1456 kb	2.26s

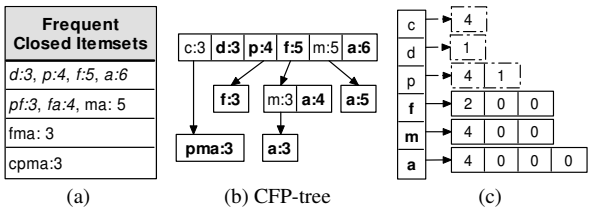
Table 3. Comparison of Initial Structures

Thus the identification of a non-closed itemsets not only reduces output size, but also avoids unnecessary mining cost. Based on pruning condition (2) in Lemma 1, we can check whether an item $i \in F$ appears in every transaction of D_p. If such i exists, then there is no need to consider the frequent itemsets that do not contain i when mining D_p. In other words, we can directly perform mining on $D_{p\bigcup\{i\}}$ instead of D_p (line 3-4). The efforts for mining $D_{p\bigcup\{j\}}$, $j \neq i$ are saved. The pseudo-code for mining frequent closed itemsets is shown in Algorithm 2.

Algorithm 2 AFOPT-close Algorithm

Input:
p is a frequent itemset
D_p is the conditional database of p
min_sup is the minimum support threshold;

Description:
1: Scan D_p count frequent items, F=$\{i_1, i_2, \cdots, i_n\}$;
2: Sort items in F in ascending order of their frequencies;
3: $I = \{i | i \in F \ and \ support(p \bigcup \{i\}) = support(p)\}$;
4: $F = F - I$; $p = p \bigcup I$;
5: **for all** transaction $t \in D_p$ **do**
6: remove infrequent items from t, and sort remaining items according to their orders in F;
7: let i be the first item of t, insert t into $D_{p\bigcup\{i\}}$.
8: **for all** item $i \in F$ **do**
9: $s = p \bigcup \{i\}$;
10: **if** s is closed **then**
11: Output s;
12: AFOPT-close(s, D_s, min_sup);
13: PushRight(D_s);



(a) (b) CFP-tree (c)

Figure 3. CFP-tree and Two-layer Hash Map

4.2 Closed itemset checking

During the mining process, we store all existing frequent closed itemsets in a tree structure, called Condensed Frequent Pattern tree or CFP-tree for short [8]. We use the CFP-tree to check whether an itemset is closed. An example of CFP-tree is shown in Figure 3(b) which stores all the frequent closed itemsets in Figure 3(a). They are mined from the database shown in Figure 2(a) with support 40%.

Each CFP-tree node is a variable-length array, and all the items in the same node are sorted in ascending order of their frequencies. A path in the tree starting from an entry in the root node represents a frequent itemset. The CFP-tree has two properties: *the left containment property* and *the Apriori property*. The *Apriori Property* is that the support of any child of a CFP-tree entry cannot be greater than the support of that entry. The *Left Containment Property* is that the item of an entry E can only appear in the subtrees pointed by entries before E or in E itself. The superset of an itemset p with support s can be efficiently searched in the CFP-tree based on these two properties. The apriori property can be exploited to prune subtrees pointed by entries with support less than s. The left containment property can be utilized to prune subtrees that do not contain all items in p. We also maintain a hash-bitmap in each entry to indicate whether an item appears in the subtree pointed by that entry to further reduce searching cost. The superset search algorithm is shown in Algorithm 3. BinarySearch($cnode$, s) returns the first entry in a CFP-tree node with support no less than s. Algorithm 3 do not require the whole CFP-tree to be in main memory because it is also very efficient on disk. Moreover, the CFP-tree structure is a compact representation of the frequent closed itemsets, so it has a higher chance to be held in memory than flat representation.

Although searching in CFP-tree is very efficient, it is still costly when CFP-tree is large. Inspired by the two-layer structure adopted by CLOSET+ algorithm[16] for subset checking, we use a two-layer hash map to check whether an itemset is closed before searching in CFP-tree. The two-layer hash map is shown in Figure 3(c). We maintain a hash map for each item. The hash map of item i is denoted by $i.hashmap$. The length of the hash map of an item i is set to $\min\{sup(i)\text{-}min\_sup, max\_hashmap\_len\}$, where $max\_hashmap\_len$ is a parameter to control the maximal size of the hash maps and min_sup=$\min\{sup(i) | i \ is \ frequent\}$. Given an itemset

Algorithm 3 Search_Superset Algorithm

Input:
l is a frequent itemset
$cnode$ the CFP+-tree node pointed by l
s is the minimum support threshold
I is a set of items to be contained in the superset
Description:
1: **if** $I = \phi$ **then**
2: **return** true;
3: $\bar{E}$ = the first entry of $cnode$ such that $\bar{E}.item \in I$;
4: E' = BinarySearch($cnode$, s);
5: **for all** entry $E \in cnode$, E between E' and $\bar{E}$ **do**
6: l'=$l \bigcup \{E.item\}$;
7: **if** $E.child \neq$ NULL AND all items in $I - \{E.item\}$ are in $E.subtree$ **then**
8: found = Search_Superset(l', $E.child$, s, $I - \{E.item\}$);
9: **if** found **then**
10: **return** true;
11: **else if** $I - \{E.item\} = \phi$ **then**
12: **return** true;
13: **return** false;

$p = \{i_1, i_2, \cdots, i_l\}$, p is mapped to $i_j.hashmap[(sup(p) - min\_sup)\%max\_hashmap\_len]$, $j = 1, 2, \cdots, l$. An entry in a hash map records the maximal length of the itemsets mapped to it. For example, itemset $\{c, p, m, a\}$ set the first entry of $c.hashmap$, $p.hashmap$, $m.hashmap$ and $a.hashmap$ to 4. Figure 3(c) shows the status of the two-layer hash map before mining D_f. An itemset p must be closed if any of the entry it mapped to contains a lower value than its length. In such cases there is no need to search in CFP-tree. The hash map of an item i can be released after all the frequent itemsets containing i are mined because they will not be used in later mining. For example, when mining D_f, the hash map of items c, d and p can be deleted.

5 Mining Maximal Frequent Itemsets

The problem of mining maximal frequent itemsets can be viewed as given a minimum support threshold min_sup, finding a border through the search space tree such that all the nodes below the border are infrequent and all the nodes above the border are frequent. The goal of maximal frequent itemsets mining is to find the border by counting support for as less as possible itemsets. Existing maximal algorithms [19, 7, 1, 4, 5] adopted various pruning techniques to reduce the search space to be explored.

5.1 Pruning techniques

The most effective techniques are based on the following two lemmas to prune a whole branch from search space tree.

Lemma 3 *Given a frequent itemset p, if $p \bigcup cand\_exts(p)$ is frequent but not maximal, then none of the frequent itemsets mined from D_p and from p's right sibling's conditional databases can be maximal because all of them are subsets of $p \bigcup cand\_exts(p)$.*

Lemma 4 *Given a frequent itemset p, if $p \bigcup freq\_exts(p)$ is frequent but not maximal, then none of the frequent itemsets mined from D_p can be maximal because all of them are subsets of $p \bigcup freq\_exts(p)$.*

Based on Lemma 3, before mining D_p, we can first check whether $p \bigcup cand\_exts(p)$ is frequent but not maximal. This can be done by two techniques.

Superset Pruning Technique: It is to check whether there exists some maximal frequent itemset such that it is a superset of $p \bigcup cand\_exts(p)$. Like frequent closed itemset mining, subset checking can be challenging when the number of maximal itemsets is large. We will discuss this issue in next subsection.

Lookahead Technique: It is to check whether $p \bigcup cand\_exts(p)$ is frequent when count frequent items in D_p. If D_p is represented by AFOPT-tree, the lookahead operation can be accomplished by simply looking at the left-most branch of AFOPT-tree. If $p \bigcup cand\_exts(p)$ is frequent, then the length of the left-most branch is equal to $|cand\_exts(p)|$, and the support of the leaf node of the left-most branch is no less than min_sup.

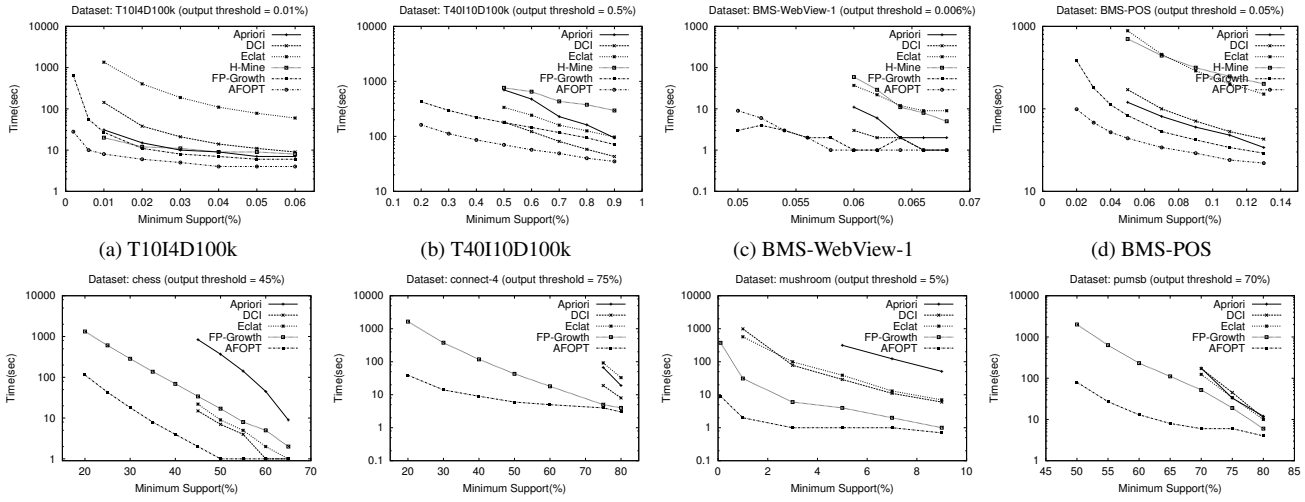
If the superset pruning technique and lookahead technique fail, then based on Lemma 4 we can use superset pruning technique to check whether $p \bigcup freq\_exts(p)$ is frequent but not maximal. Two other techniques are adopted in our algorithm.

Excluding items appearing in every transaction of D_p from subsequent mining: Like frequent closed itemset mining, if an item i appears in every transaction of D_p, then a frequent itemset q mined from D_p and not containing i cannot be maximal because $q \bigcup \{i\}$ is frequent.

Single Path Trimming: If D_p is represented by AFOPT-tree and it has only one child i, then we can append i to p and remove it from subsequent mining.

5.2 Subset checking

When do superset prunning, to check against all frequent maximal itemsets can be costly when the number of maximal itemsets is large. Zaki et. al proposed a progressive focusing technique for subset checking [5]. The observation behind the progressive focusing technique is that only the maximal frequent itemsets containing p can be a superset of $p \bigcup cand\_exts(p)$ or $p \bigcup freq\_exts(p)$. The set of maximal frequent itemsets containing p is called the local maximal frequent itemsets with respect to p, denoted as LMFI$_p$. When check whether $p \bigcup cand\_exts(p)$ or $p \bigcup freq\_exts(p)$ is a subset of some existing maximal frequent itemsets, we only need to check them against LMFI$_p$. The frequent itemsets in LMFI$_p$ can either come from p's parent's LMFI, or from p's left-siblings' LMFI. The construction of LMFIs is very similar to the construction of conditional databases. The construction consists of two steps: (1) projecting: after all frequent items F in D_p



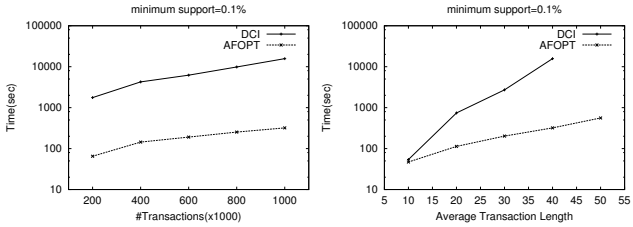
(a) T10I4D100k (b) T40I10D100k (c) BMS-WebView-1 (d) BMS-POS

(e) chess (f) connect-4 (g) mushroom (h) pumsb

Figure 4. Performance Comparison of FI Mining Algorithms

Data Sets	Size	#Trans	#Items	MaxTL	AvgTL
T10I4D100k (0.01%)	3.93M	100000	870	30	10.10
T40I10D100k (0.5%)	15.12M	100000	942	78	39.61
BMS-POS (0.05%)	11.62MB	515597	1657	165	6.53
BMS-WebView-1 (0.06%)	1.28M	59601	497	267	2.51
chess (45%)	0.34M	3196	75	37	37.00
connect-4 (75%)	9.11M	67557	129	43	43.00
mushroom (5%)	0.56M	8124	119	23	23.00
pumsb (70%)	16.30M	49046	2113	74	74.00

Table 4. Datasets



(a) #Transactions (b) AvgTransLen

Figure 5. Scalability Study

are counted, $\forall s \in \text{LMFI}_p$, s is put into $\text{LMFI}_{p \bigcup \{i\}}$, where i is the first item in F appears in s; (2) push-right: after all the maximal frequent itemsets containing p are mined, $\forall s \in \text{LMFI}_p$, s is put into LMFI_q if q is the first right sibling of p containing an item in s. In our implementation, we use pseudo projection technique to generate LMFIs, i.e. LMFI_p is a collection of pointers pointing to those maximal itemsets containing p.

6 Experimental results

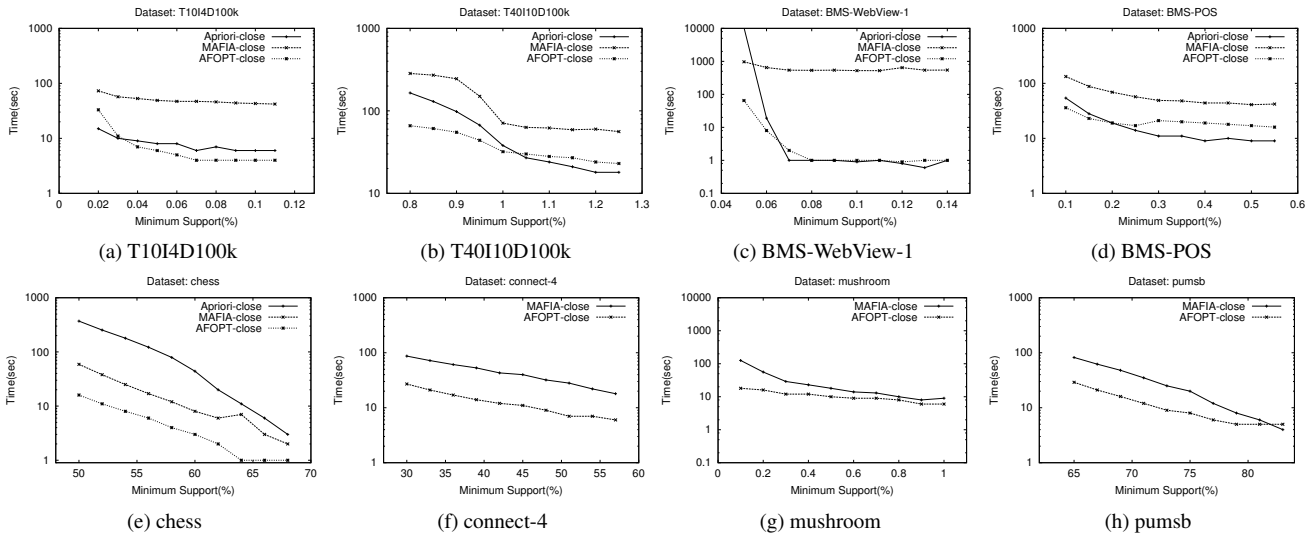
In this section, we compare the performance of our algorithms with other FIM algorithms. All the experiments were conducted on a 1Ghz Pentium III with 256MB memory running Mandrake Linux.

Table 4 shows some statistical information about the datasets used for performance study. All the datasets were downloaded from FIMI'03 workshop web site. The fifth and sixth columns are maximal and average transaction length. These statistics provide some rough description of the density of the datasets.

6.1 Mining all frequent itemsets

We compared the efficiency of AFOPT-all algorithm with Apriori, DCI, FP-growth, H-Mine and Eclat algorithms. The Apriori and Eclat algorithms we used are implemented by Christian Borgelt. DCI was downloaded from its web site. We obtained the source code of FP-growth from its authors. H-Mine was implemented by ourselves. We ran H-Mine only on several sparse datasets since it was designed for sparse datasets and it changes to use FP-tree on dense datasets. Figure 4 shows the running time of all algorithms over datasets shown in Table 4. When the minimum support threshold is very low, an intolerable number of frequent itemsets can be generated. So when minimum support threshold reached some very low value, we turned off the output. This minimum support value is called **output threshold**, and they are shown on top of each figure.

With high minimum support threshold, all algorithms showed comparable performance. When minimum support threshold was lowered, the gaps between algorithms increased. The two candidate generate-and-test approaches, Apriori and DCI, showed satisfactory performance on several sparse datasets, but took thousands of seconds to terminate on dense datasets due to high cost for generating and testing a large number of candidate itemsets. H-Mine demonstrated similar performance with FP-growth on dataset T10I4D100k, but it was slower than FP-growth on the other three sparse datasets. H-Mine uses pseudo con-



(a) T10I4D100k (b) T40I10D100k (c) BMS-WebView-1 (d) BMS-POS

(e) chess (f) connect-4 (g) mushroom (h) pumsb

Figure 6. Performance Comparison of FCI Mining Algorithms

struction strategy, which cannot reduce traversal cost as effective as physical construction strategy. Eclat uses vertical mining techniques. Support counting is performed efficiently by transaction id list join. But Eclat is not scale well with respect to the number of transactions in a database. The running time of AFOPT-all was rather stable over all tested datasets, and it outperformed other algorithms.

6.2 Scalability

We studied the scalability of our algorithm by perturbing the IBM synthetic data generator along two dimensions: the number of transactions was varied from 200k to 1000k and the average transaction length was varied from 10 to 50. The default values of these two parameters were set to 1000k and 40 respectively. We compared our algorithm with algorithm DCI. Other algorithms took long time to finish on large datasets, so we exclude them from comparison. Figure 5 shows the results when varying the two parameters.

6.3 Mining frequent closed itemsets

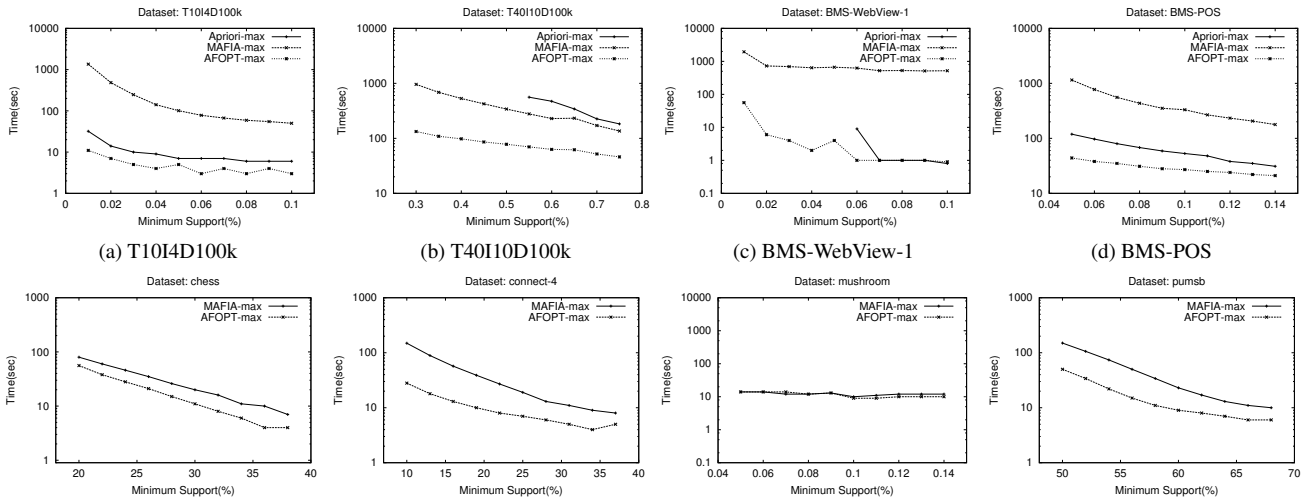
We compared AFOPT-close with MAFIA [4] and Apriori algorithms. Both algorithms have an option to generate only closed itemsets. We denoted these two algorithms as Apriori-close and MAFIA-close respectively in figures. MAFIA was downloaded from its web site. We compared with Apriori-close only on sparse datasets because Apriori-close requires a very long time to terminate on dense datasets. On several sparse datasets, AFOPT-close and Apriori-close showed comparable performance. Both of them were orders of magnitude faster than MAFIA-close. MAFIA-close uses vertical mining technique. It uses bitmaps to represent tid lists. AFOPT-close showed better performance on tested dense datasets due to its adaptive nature and the efficient subset checking techniques described in Section 4. On dense datasets, AFOPT-close uses tree structure to store conditional databases. The tree structure has apparent advantages on dense datasets because many transactions share their prefixes.

6.4 Mining maximal frequent itemsets

We compared AFOPT-max with MAFIA and Apriori algorithms. The Apriori algorithm also has an option to produce only maximal frequent itemsets. It is denoted as "Apriori-max" in figures. Again we only compare with it on sparse datasets. Apriori-max explores the search space in breadth-first order. It finds short frequent itemsets first. Maximal frequent itemsets are generated in a post-processing phase. Therefore Apriori-max is infeasible when the number of frequent itemsets is large even if it adopts some pruning techniques during the mining process. AFOPT-max and MAFIA generate frequent itemsets in depth-first order. Long frequent itemsets are mined first. All the subsets of a long maximal frequent itemsets can be pruned from further consideration by using the superset pruning and lookahead technique. AFOPT-max uses tree structure to represent dense conditional databases. The AFOPT-tree introduces more pruning capability than tid list or tid bitmap. For example, if a conditional database can be represented by a single branch in AFOPT-tree, then the single branch will be the only one possible maximal itemset in the conditional database. AFOPT-max also benefits from the progressive focusing technique for superset pruning. MAFIA was very efficient on small datasets, e.g chess and mushroom when the length of bitmap is short.

7 Conclusions

In this paper, we revisited the frequent itemset mining problem and focused on investigating the algorithmic performance space of the pattern growth approach. We identified four dimensions in which existing pattern growth al-



(a) T10I4D100k (b) T40I10D100k (c) BMS-WebView-1 (d) BMS-POS

(e) chess (f) connect-4 (g) mushroom (h) pumsb

Figure 7. Performance Comparison of MFI Mining Algorithms

gorithms differ: (1) item search order: static lexicographical order or ascending frequency order; (2) conditional database representation: tree-based structure or array-based structure; (3) conditional database construction strategy: physical construction or pseudo construction; and (4) tree traversal strategy: bottom-up or top-down. Existing algorithms adopted different strategies on these four dimensions in order to reduce the total number of conditional databases and the mining cost of each individual conditional database.

we described an efficient pattern growth algorithm AFOPT in the paper. It adaptively uses three different structures: arrays, AFOPT-tree and buckets, to represent conditional databases according to the density of a conditional database. Several parameters were introduced to control which structure should be used for a specific conditional database. We showed that the adaptive conditional database representation strategy requires less space than using array-based structure or tree-based structure solely. We also extended AFOPT algorithm to mine closed and maximal frequent itemsets, and described how to incorporate pruning techniques into AFOPT framework. Efficient subset checking techniques for both closed and maximal frequent itemsets mining were presented. A set of experiments were conducted to show the efficiency of the proposed algorithms.

References

[1] R. Agrawal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In *SIGKDD*, 2000.

[2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.

[3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, 1997.

[4] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.

[5] K. Gouda and M. J. Zaki. Genmax: Efficiently mining maximal frequent itemsets. In *ICDM*, 2001.

[6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.

[7] R.J. Bayardo. Jr. Efficiently mining long patterns from databases. In *SIGMOD*, 1998.

[8] G. Liu, H. Lu, W. Lou, and J. X. Yu. On computing, storing and querying frequent patterns. In *SIGKDD*, 2003.

[9] G. Liu, H. Lu, Y. Xu, and J. X. Yu. Ascending frequency ordered prefix-tree: Efficient mining of frequent patterns. In *DASFAA*, 2003.

[10] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *SIGKDD*, 2002.

[11] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, 1999.

[12] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, 2001.

[13] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *DMKD*, 2000.

[14] R. Raymon. Search through systematic set enumeration. In *Proc. of KR Conf.*, 1992.

[15] R.C.Agarwal, C.C.Aggarwal, and V.V.V.Prasad. A tree projection algorithm for finding frequent itemsets. *Journal on Parallel and Distributed Computing*, 61(3):350–371, 2001.

[16] J. Wang, J. Pei, and J. Han. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD*, 2003.

[17] Y. Xu, J. X. Yu, G. Liu, and H. Lu. From path tree to frequent patterns: A framework for mining frequent patterns. In *ICDM*, pages 514–521, 2002.

[18] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SDM*, 2002.

[19] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *SIGKDD*, 1997.

Efficiently Using Prefix-trees in Mining Frequent Itemsets

Gösta Grahne and Jianfei Zhu
Concordia University
Montreal, Canada
{grahne, j_zhu}@cs.concordia.ca

Abstract

Efficient algorithms for mining frequent itemsets are crucial for mining association rules. Methods for mining frequent itemsets and for iceberg data cube computation have been implemented using a prefix-tree structure, known as an FP-tree, for storing compressed information about frequent itemsets. Numerous experimental results have demonstrated that these algorithms perform extremely well. In this paper we present a novel array-based technique that greatly reduces the need to traverse FP-trees, thus obtaining significantly improved performance for FP-tree based algorithms. Our technique works especially well for sparse datasets.

Furthermore, we present new algorithms for a number of common data mining problems. Our algorithms use the FP-tree data structure in combination with our array technique efficiently, and incorporates various optimization techniques. We also present experimental results which show that our methods outperform not only the existing methods that use the FP-tree structure, but also all existing available algorithms in all the common data mining problems.

1. Introduction

A fundamental problem for mining association rules is to mine frequent itemsets (FI's). In a transaction database, if we know the support of all frequent itemsets, the association rules generation is straightforward. However, when a transaction database contains large number of large frequent itemsets, mining *all* frequent itemsets might not be a good idea. As an example, if there is a frequent itemset with size ℓ, then all 2^ℓ nonempty subsets of the itemset have to be generated. Thus, a lot of work is focused on discovering only all the *maximal frequent itemsets* (MFI's). Unfortunately, mining only MFI's has the following deficiency. From an MFI and its support s, we know that all its subsets are frequent and the support of any of its subset is not less than s, but we do not know the exact value of the support. To solve this problem, another type of a frequent itemset, the *Closed Frequent Itemset* (CFI), has been proposed. In most cases, though, the number of CFI's is greater than the number of MFI's, but still far less than the number of FI's.

In this work we mine FI's, MFI's and CFI's by efficiently using the FP-tree, the data structure that was first introduced in [6]. The FP-tree has been shown to be one of the most efficient data structures for mining frequent patterns and for "iceberg" data cube computations [6, 7, 9, 8].

The most important contribution of our work is a novel technique that uses an array to greatly improve the performance of the algorithms operating on FP-trees. We first demonstrate that the use of our array-based technique drastically speeds up the FP-growth method, since it now needs to scan each FP-tree only once for each recursive call emanating from it. We then use this technique and give a new algorithm FPmax*, which extends our previous algorithm FPmax, for mining maximal frequent itemsets. In FPmax*, we use a variant of the FP-tree structure for subset testing, and give number of optimizations that further reduce the running time. We also present an algorithm, FPclose, for mining closed frequent itemsets. FPclose uses yet another variation of the FP-tree structure for checking the closedness of frequent itemsets.

Finally, we present experimental results that demonstrate the fact that all of our FP-algorithms outperform previously known algorithms practically always.

The remaining of the paper is organized as follows. In Section 2, we briefly review the FP-growth method, and present our novel array technique that results in the greatly improved method FPgrowth*. Section 3 gives algorithm FPmax*, which is an extension of our previous algorithm FPmax, for mining MFI's. Here we also introduce our approach of subset testing needed in mining MFI's and CFI's. In Section 4 we give algorithm FPclose, for mining CFI's. Experimental results are given in Section 5. Section 6 concludes, and outlines directions of future research.

2. Discovering FI's

2.1. The FP-tree and FP-growth method

The FP-growth method by Han *et al.* [6] uses a data structure called the FP-tree (Frequent Pattern tree). The FP-tree is a compact representation of all relevant frequency information in a database. Every branch of the FP-tree represents a frequent itemset, and the nodes along the branches are stored in decreasing order of frequency of the corresponding items, with leaves representing the least frequent items. Compression is achieved by building the tree in such a way that overlapping itemsets share prefixes of the corresponding branches.

The FP-tree has a header table associated with it. Single items and their counts are stored in the header table in decreasing order of their frequency. The entry for an item also contains the head of a list that links all the corresponding nodes of the FP-tree.

Compared with Apriori [1] and its variants which need several database scans, the FP-growth method only needs two database scans when mining all frequent itemsets. The first scan counts the number of occurrences of each item. The second scan constructs the initial FP-tree which contains all frequency information of the original dataset. Mining the database then becomes mining the FP-tree.

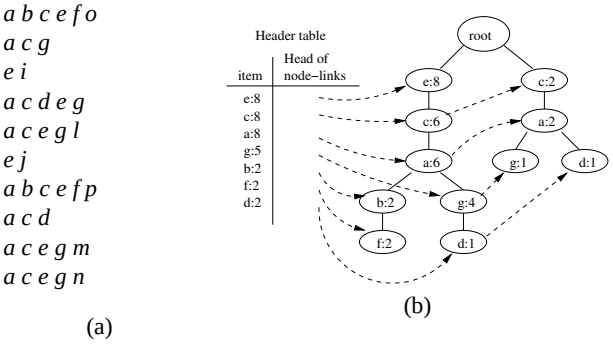


Figure 1. An Example FP-tree (minsup=20%)

To construct the FP-tree, first find all frequent items by an initial scan of the database. Then insert these items in the header table, in decreasing order of their count. In the next (and last) scan, as each transaction is scanned, the set of frequent items in it are inserted into the FP-tree as a branch. If an itemset shares a prefix with an itemset already in the tree, the new itemset will share a prefix of the branch representing that itemset. In addition, a counter is associated with each node in the tree. The counter stores the number of transactions containing the itemset represented by the path from the root to the node in question. This counter is updated during the second scan, when a transaction causes the insertion of a new branch. Figure 1 (a) shows an example of a database and Figure 1 (b) the FP-tree for that database. Note that there may be more than one node corresponding to an item in the FP-tree. The frequency of any one item i is the sum of the count associated with all nodes representing i, and the frequency of an itemset equals the sum of the counts of the least frequent item in it, restricted to those branches that contain the itemset. For instance, from Figure 1 (b) we can see that the frequency of the itemset $\{c, a, g\}$ is 5.

Thus the constructed FP-tree contains all frequency information of the database. Mining the database becomes mining the FP-tree. The FP-growth method relies on the following principle: if X and Y are two itemsets, the count of itemset $X \cup Y$ in the database is exactly that of Y in the restriction of the database to those transactions containing X. This restriction of the database is called the *conditional pattern base* of X, and the FP-tree constructed from the conditional pattern base is called X's *conditional FP-tree*, which we denote by T_X. We can view the FP-tree constructed from the initial database as $T_\emptyset$, the conditional FP-tree for $\emptyset$. Note that for any itemset Y that is frequent in the conditional pattern base of X, the set $X \cup Y$ is a frequent itemset for the original database.

Given an item i in the header table of an FP-tree T_X, by following the linked list starting at i in the header table of T_X, all branches that contain item i are visited. These branches form the conditional pattern base of $X \cup \{i\}$, so the traversal obtains all frequent items in this conditional pattern base. The FP-growth method then constructs the conditional FP-tree $T_{X \cup \{i\}}$, by first initializing its header table based on the found frequent items, and then visiting the branches of T_X along the linked list of i one more time and inserting the corresponding itemsets in $T_{X \cup \{i\}}$. Note that the order of items can be different in T_X and $T_{X \cup \{i\}}$. The above procedure is applied recursively, and it stops when the resulting new FP-tree contains only one single path. The complete set of frequent itemsets is generated from all single-path FP-trees.

2.2. An array technique

The main work done in the FP-growth method is traversing FP-trees and constructing new conditional FP-trees after the first FP-tree is constructed from the original database. From numerous experiments we found out that about 80% of the CPU time was used for traversing FP-trees. Thus, the question is, can we reduce the traversal time so that the method can be sped up?

The answer is yes, by using a simple additional data structure. Recall that for each item i in the header of a conditional FP-tree T_X, two traversals of T_X are needed for constructing the new conditional FP-tree $T_{X \cup \{i\}}$. The first traversal finds all frequent items in the conditional pattern base of $X \cup \{i\}$, and initializes the FP-tree $T_{X \cup \{i\}}$ by constructing its header table. The second traversal constructs

the new tree $T_{X\cup\{i\}}$. We can omit the first scan of T_X by constructing an array A_X while building T_X. The following example will explain the idea. In Figure 1 (a), supposing that the minimum support is 20%, after the first scan of the original database, we sort the frequent items as e:8, c:8, a:8, g:5, b:2, f:2, d:2. This order is also the order of items in the header table of $T_\emptyset$. During the second scan of the database we will construct $T_\emptyset$, and an array $A_\emptyset$. This array will store the counts of all 2-itemsets. All cells in the array are initialized as 0.

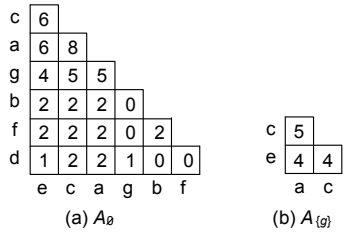


Figure 2. Two array examples

In $A_\emptyset$, each cell is a counter of a 2-itemset, cell $A_\emptyset[d,e]$ is the counter for itemset $\{d,e\}$, cell $A_\emptyset[d,c]$ is the counter for itemset $\{d,c\}$, and so forth. During the second scan for constructing $T_\emptyset$, for each transaction, first all frequent items in the transaction are extracted. Suppose these items form itemset I. To insert I into $T_\emptyset$, the items in I are sorted according to the order in header table of $T_\emptyset$. When we insert I into $T_\emptyset$, at the same time $A_\emptyset[i,j]$ is incremented by 1 if $\{i,j\}$ is contained in I. For example, for the first transaction, $\{a,b,c,e,f\}$ is extracted (item o is infrequent) and sorted as e,c,a,b,f. This itemset is inserted into $T_\emptyset$ as usual, and at the same time, $A_\emptyset[f,e]$, $A_\emptyset[f,c]$, $A_\emptyset[f,a]$, $A_\emptyset[f,b]$, $A_\emptyset[b,a]$, $A_\emptyset[b,c]$, $A_\emptyset[b,e]$, $A_\emptyset[a,e]$, $A_\emptyset[a,c]$, $A_\emptyset[c,e]$ are all incremented by 1. After the second scan, array $A_\emptyset$ keeps the counts of all pairs of frequent items, as shown in table (a) of Figure 2.

Next, the FP-growth method is recursively called to mine frequent itemsets for each item in header table of $T_\emptyset$. However, now for each item i, instead of traversing $T_\emptyset$ along the linked list starting at i to get all frequent items in i's conditional pattern base, $A_\emptyset$ gives all frequent items for i. For example, by checking the third line in the table for $A_\emptyset$, frequent items e,c,a for the conditional pattern base of g can be obtained. Sorting them according to their counts, we get a,c,e. Therefore, for each item i in $T_\emptyset$ the array $A_\emptyset$ makes the first traversal of $T_\emptyset$ unnecessary, and $T_{\{i\}}$ can be initialized directly from $A_\emptyset$.

For the same reason, from a conditional FP-tree T_X, when we construct a new conditional FP-tree for $X\cup\{i\}$, for an item i, a new array $A_{X\cup\{i\}}$ is calculated. During the construction of the new FP-tree $T_{X\cup\{i\}}$, the array $A_{X\cup\{i\}}$ is filled. For instance, in Figure 1, the cells of array $A_{\{g\}}$ is shown in table (b) of Figure 2. This array is constructed as follows. From the array $A_\emptyset$, we know that the frequent items in the conditional pattern base of $\{g\}$ are, in order, a,c,e. By following the linked list of g, from the first node we get $\{e,c,a\}:4$, so it is inserted as $(a:4,c:4,e:4)$ into the new FP-tree $T_{\{g\}}$. At the same time, $A_{\{g\}}[e,c]$, $A_{\{g\}}[e,a]$ and $A_{\{g\}}[c,a]$ are incremented by 4. From the second node in the linked list, $\{c,a\}:1$ is extracted, and it is inserted as $(a:1,c:1)$ into $T_{\{g\}}$. At the same time, $A_{\{g\}}[c,a]$ is incremented by 1. Since there are no other nodes in the linked list, the construction of $T_{\{g\}}$ is finished, and array $A_{\{g\}}$ is ready to be used for construction of FP-trees in next level of recursion. The construction of arrays and FP-trees continues until the FP-growth method terminates.

Based on above discussion, we define a variation of the FP-tree structure in which besides all attributes given in [6], an FP-tree also has an attribute, *array*, which contains the corresponding array.

Now let us analyze the size of an array. Suppose the number of frequent items in the first FP-tree is n. Then the size of the associated array is $\sum_{i=1}^{n-1} i = n(n-1)/2$. We can expect that FP-trees constructed from the first FP-tree have fewer frequent items, so the sizes of the associated arrays decrease. At any time, since an array is an attribute of an FP-tree, when the space for the FP-tree is freed, the space for the array is also freed.

2.3. Discussion

The array technique works very well especially when the dataset is sparse. The FP-tree for a sparse dataset and the recursively constructed FP-trees will be big and bushy, due to the fact that they do not have many shared common prefixes. The arrays save traversal time for all items and the next level FP-trees can be initialized directly. In this case, the time saved by omitting the first traversals is far greater than the time needed for accumulating counts in the associated array.

However, when a dataset is dense, the FP-trees are more compact. For each item in a compact FP-tree, the traversal is fairly rapid, while accumulating counts in the associated array may take more time. In this case, accumulating counts may not be a good idea.

Even for the FP-trees of sparse datasets, the first levels of recursively constructed FP-trees are always conditional FP-trees for *the most common prefixes*. We can therefore expect the traversal times for the first items in a header table to be fairly short, so the cells for these first items are unnecessary in the array. As an example, in Figure 2 table (a), since e,c, and a are the first 3 items in the header table, the first two lines do not have to be calculated, thus saving counting time.

Note that the datasets (the conditional pattern bases) change during the different depths of the recursion. In order to estimate whether a dataset is sparse or dense, during the construction of each FP-tree we count the number of nodes in each level of the tree. Based on experiments, we found that if the upper quarter of the tree contains less than 15% of the total number of nodes, we are most likely dealing with a dense dataset. Otherwise the dataset is likely to be sparse.

If the dataset appears to be dense, we do not calculate the array for the next level of the FP-tree. Otherwise, we calculate array for each FP-tree in the next level, but the cells for the first several (say 5) items in its header table are not set.

2.4. FPgrowth* : an improved FP-growth method

Figure 3 contains the pseudocode for our new method FPgrowth*. The procedure has an FP-tree T as parameter. The tree has attributes: *base*, *header* and *array*. $T.base$ contains the itemset X, for which T is a conditional FP-tree, the attribute *header* contains the head table, and $T.array$ contains the array A_X.

```
Procedure FPgrowth*(T)
Input:   A conditional FP-tree T
Output:  The complete set of FI's
         corresponding to T.
Method:
1. if T only contains a single path P
2. then for each subpath Y of P
3.   output pattern Y ∪ T.base with
     count = smallest count of nodes
     in Y
4. else for each i in T.header
5.   output Y = T.base ∪ {i} with i.count
6.   if T.array is not NULL
7.     construct a new header table
       for Y's FP-tree from T.array
8.   else construct a new header table
         from T;
9.     construct Y's conditional
       FP-tree T_Y and its array A_Y;
10.  if T_Y ≠ ∅
11.   call FPgrowth*(T_Y);
```

Figure 3. Algorithm FPgrowth*

In *FPgrowth**, line 6 tests if the array of the current FP-tree is NULL. If the FP-tree corresponds to a sparse dataset, its array is not NULL, and line 7 will be used to construct the header table of the new conditional FP-tree from the array directly. One FP-tree traversal is saved for this item compared with the FP-growth method in [6]. In line 9, during the construction, we also count the nodes in the different levels of the tree, in order to estimate whether we shall really calculate the array, or just set $T_Y.array = NULL$.

From our experimental results we found that an FP-tree could have millions of nodes, thus, allocating and deallocating those nodes takes plenty of time. In our implementation, we used our own main memory management for allocating and deallocating nodes. Since all memory for nodes in an FP-tree is deallocated after the current recursion ends, a chunk of memory is allocated for each FP-tree when we create the tree. The chunk size is changeable. After generating all frequent itemsets from the FP-tree, the chunk is discarded. Thus we successfully avoid freeing nodes in the FP-tree one by one, which is more time-consuming.

3. FPmax*: Mining MFI's

In [5] we developed FPmax, a variation of the FP-growth method, for mining maximal frequent itemsets. Since the array technique speeds up the FP-growth method for sparse datasets, we can expect that it will be useful in FPmax too. This gives us an improved method, FPmax*. Compared to FPmax, the improved method FPmax* also has a more efficient subset test, as well as some other optimizations. It turns out that FPmax* outperforms GenMax[4] and MAFIA [3] for all cases we discussed in [5].

3.1. The MFI-Tree

Since FPmax is a depth-first algorithm, a frequent itemset can be a subset only of an already discovered MFI. In FPmax we introduced a global data structure, the *Maximal Frequent Itemset tree* (MFI-tree), to keep the track of MFI's. A newly discovered frequent itemset is inserted into the MFI-tree, unless it is a subset of an itemset already in the tree. However, for large datasets, the MFI-tree will be quite large, and sometimes one itemset needs thousands of comparisons for subset testing. Inspired by the way subset checking is done in [4], in FPmax*, we still use the MFI-tree structure, but for each conditional FP-tree T_X, a small MFI-tree M_X is created. The tree M_X will contain all maximal itemsets in the conditional pattern base of X. To see if a local MFI Y generated from a conditional FP-tree T_X is maximal, we only need to compare Y with the itemsets in M_X. This achieves a significant speedup of FPmax.

Each MFI-tree is associated with a particular FP-tree. Children of the root of the MFI-tree are item prefix subtrees. In an MFI-tree, each node in the subtree has three fields: item-name, level and node-link. The level-field will be useful for subset testing. All nodes with same item-name are linked together, as in an FP-tree. The MFI-tree also has a header table. However, unlike the header table in an FP-tree, which is constructed from traversing the previous FP-tree or using the associated array, the header table of an

MFI-tree is constructed based on the item order in the table of the FP-tree it is associated with. Each entry in the header table consists of two fields, item-name and head of a linked list. The head points to the first node with the same item-name in the MFI-tree.

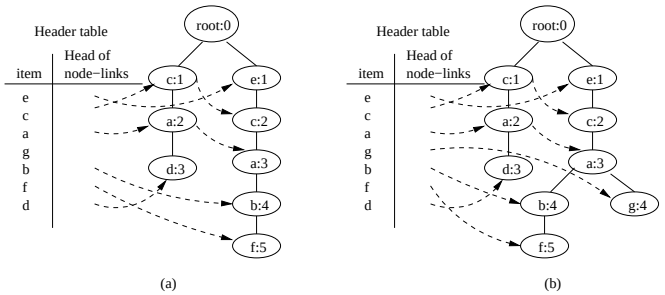


Figure 4. Construction of MFI-Tree

The insertion of an MFI into an MFI-tree is similar to the insertion of a frequent set into an FP-tree. Figure 4 shows the insertions of three MFI's into an MFI-tree associated with the FP-tree in Figure 1 (b). In Figure 4, a node $x : \ell$ means that the node is for item x and its level is ℓ. Figure 4 (a) shows the tree after (c, a, d) and (e, c, a, b, f) have been inserted. In Figure 4 (b), since new MFI (e, c, a, b, g) shares prefix (e, c, a) with (e, c, a, b, f), only one new node for g is inserted.

3.2. FPmax*

Figure 5 gives algorithm FPmax*. The first call will be for the FP-tree constructed from the original database, and it will have an empty MFI-tree. Before a recursive call *FPmax*(T,M)*, we already know from line 10 that the set containing $T.base$ and the items in the current FP-tree is not a subset of any existing MFI. During the recursion, if there is only one single path in T, this single path together with $T.base$ is an MFI of the database. In line 2, the MFI is inserted into M. If the FP-tree is not a single-path tree, then for each item i in the header table, we start preparing for the recursive call *FPmax*(T_Y, M_Y), for $Y = T.base \cup \{i\}$. The items in the header table of T are processed in increasing order of frequency, so that maximal frequent itemsets will be found before any of their frequent subsets. Lines 5 to 8 use the array technique, and line 10 calls function *subset_checking* to check if Y together with all frequent items in Y's conditional pattern base is a subset of any existing MFI in M (thus we do superset pruning here). If *subset_checking* return false, *FPmax** will be called recursively, with (T_Y, M_Y). The implementation of function *subset_checking* will be explained shortly.

Note that before and after calling *subset_checking*, if $Y \cup Tail$ is not subset of any MFI, we still do not know whether $Y \cup Tail$ is frequent. If, by constructing Y's conditional

```
Procedure FPmax*(T, M)
Input:   T, an FP-tree
         M, the MFI-tree for T.base
Output:  Updated M
Method:
1. if T only contains a single path P
2.  insert P into M
3. else for each i in T.header
4.  set Y = T.base ∪ {i};
5.  if T.array is not NULL
6.    Tail={frequent items for i in
       T.array}
7.  else
8.    Tail={frequent items in i's
       conditional pattern base}
9.  sort Tail in decreasing order of
       the items' counts
10. if not subset_checking(Y ∪ Tail, M)
11.   construct Y's conditional
        FP-tree T_Y and its array A_Y;
12.   initialize Y's conditional
        MFI-tree M_Y;
13.   call FPmax*(T_Y, M_Y);
14.   merge M_Y with M
```

Figure 5. Algorithm FPmax*

FP-tree T_Y, we find out that T_Y only has a single path, we can conclude that $Y \cup Tail$ is frequent. Since $Y \cup Tail$ was not a subset of any previously discovered MFI, it is a new MFI and will be inserted into M_Y.

3.3. Implementation of subset testing

The function *subset_checking* works as follows. Suppose *Tail* = $i_1 i_2, \ldots i_k$, in decreasing order of frequency according to the header table of M. By following the linked list of i, for each node n in the list, we test if *Tail* is a subset of the ancestors of n. Here, the level of n can be used for saving comparison time. First we test if the level of n is smaller than k. If it is, the comparison stops because there are not enough ancestors of n for matching the rest of *Tail*. This pruning technique is also applied as we move up the branch and towards the front of *Tail*.

Unlike an FP-tree, which is not changed during the execution of the algorithm, an MFI-tree is dynamic. At line 12, for each Y, a new MFI-tree M_Y is initialized from the predecessor MFI-tree M. Then after the recursive call, M is updated on line 14 to contain all newly found frequent itemsets. In the actual implementation, we however found that it was more efficient to update all MFI-trees along the recursive path, instead of merging only at the current level. In other words, we omitted line 14, and instead on line 2, P

is inserted into the current M, and also into all predecessor MFI-trees that the implementation of the recursion needs to keep in main memory in any case.

Since *FPmax** is a depth-first algorithm, it is straightforward to show that the above subset checking is correct. Based on the correctness of the FP-growth method, we can conclude that *FPmax** returns all and only the maximal frequent itemsets in a given dataset.

3.4. Optimizations

In the method FPmax*, one more optimization is used. Suppose, that at some level of the recursion, the header table of the current FP-tree is $i_1, i_2, \ldots, i_m$. Then starting from i_m, for each item in the header table, we may need to do the work from line 4 to line 14. If for any item, say i_k, where $k \leq m$, its maximal frequent itemset contains items $i_1, i_2, \ldots, i_{k-1}$, i.e., all the items that have not yet called *FPmax** recursively, these recursive calls can be omitted. This is because for those items, their tails must be subsets of $\{i_1, i_2, \ldots, i_{k-1}\}$, so $subset\_checking(Y \cup Tail)$ would always return true.

FPmax* also uses the memory management described in Section 2.4, for allocating and deallocating space for FP-trees and MFI-trees.

3.5. Discussion

One may wonder if the space required for all the MFI-trees of a recursive branch is too large. Actually, before the first call of *FPmax**, the first FP-tree has to fit in main memory. This is also required by the FP-growth method. The corresponding MFI-tree is initialized as empty. During recursive calls of *FPmax**, new conditional FP-trees are constructed from the first FP-tree or from an ancestors FP-tree. From the experience of [6], we know the recursively constructed FP-trees are relatively small. We can expect that the total size of these FP-trees is not greater than the final size of the MFI-tree for $\emptyset$. Similarly, the MFI-trees constructed from ancestors are also small. All MFI-trees grow gradually. Thus we can conclude that the total main memory requirement for running *FPmax** on a dataset is proportional to the sum of the size of the FP-tree and the MFI-tree for $\emptyset$.

4. FPclose: Mining CFI's

For mining frequent closed itemsets, FPclose works similarly to FPmax*. They both mine frequent patterns from FP-trees. Whereas FPmax* needs to check that a newly found frequent itemset is maximal, FPclose needs to verify that the new frequent itemset is closed. For this we use a CFI-tree, which is another variation of an FP-tree.

One of the first attempts to use FP-trees in CFI mining was the algorithm CLOSET+ [9]. This algorithm uses one global prefix-tree for keeping track of all closed itemsets. As we pointed out before, one global tree will be quite big, and thus slows down searches. In FPclose we will therefore use multiple, conditional CFI-trees for checking closedness of itemsets. We can thus expect that FPclose outperforms CLOSET+.

4.1. The CFI-tree and algorithm FPclose

Similar to an MFI-tree, a CFI-tree is related to an FP-tree and an itemset X, and we will denote the CFI-tree as C_X. The CFI-tree C_X always stores all already found CFI's containing itemset X, and their counts. A newly found frequent itemset Y that contains X only needs to be compared with the CFI's in C_X. If in C_X, there is no superset of Y with same count as Y, Y is closed.

In a CFI-tree, each node in the subtree has four fields: item-name, count, node-link and level. Here, the count field is needed because when comparing a Y with a set Z in the tree, we are trying to verify that it is not the case that $Y \subset Z$, and Y and Z have the same count. The order of the items in a CFI-tree's header table is same as the order of items in header table of its corresponding FP-tree.

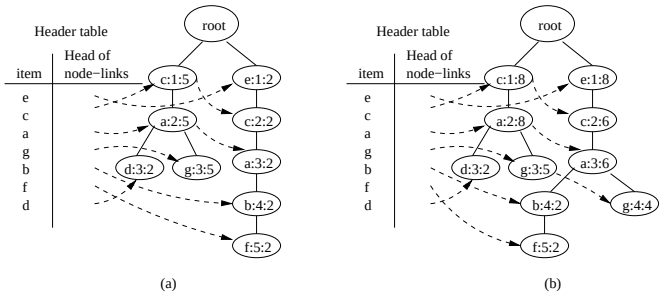


Figure 6. Construction of CFI-Tree

The insertion of a CFI into a CFI-tree is similar to the insertion of a transaction into an FP-tree, except now the count of a node is not incremented, it is always replaced by the maximal count up-to-date. Figure 6 shows some snapshots of the construction of a CFI-tree with respect to the FP-tree in Figure 1 (b). The item order in two trees are same because they are both for base $\emptyset$. Note that insertions of CFI's into the top level CFI-tree will occur only after recursive calls have been made. In the following example, the insertions would in actuality be performed during various stages of the execution, not in bulk as the example might suggest. In Figure 6, a node $x : \ell : c$ means that the node is for item x, its level is ℓ and its count is c. In Figure 6 (a), after inserting (c, a, d) and (e, c, a, b, f) with count 2, then we insert (c, a, g) with count 5. Since (c, a, g) shares the prefix (c, a) with (c, a, d), only node g is appended, and at the same time, the counts for nodes c and a are both changed to be 5. In part (b) of Figure 6, the CFI's $(e, c, a, g) : 4$,

$(c, a) : 8$, $(c, a, e) : 6$ and $(e) : 8$ are inserted. At this stage the tree contains all CFI's for the dataset in Figure 1 (a).

```
Procedure FPclose(T, C)
Input:   T, an FP-tree
         C, the CFI-tree for T.base
Output:  Updated C
Method:
1. if T only contains a single path P
2.   generate all CFI's from P
3.   for each CFI X generated
4.     if not closed_checking(X, C)
5.       insert X into C
6. else for each i in T.header
7.     set Y = T.base ∪ {i};
8.     if not closed_checking(Y, C)
9.       if T.array is not NULL
10.        Tail = {frequent items for
            i in T.array}
11.      else
12.        Tail={frequent items in i's
            conditional pattern base}
13.      sort Tail in decreasing order
          of items' counts
14.      construct the FP-tree T_Y and
          its array A_Y;
15.      initialize Y's conditional
          CFI-tree C_Y;
16.      call FPclose(T_Y, C_Y);
17.      merge C_Y with C
```

Figure 7. Algorithm *FPclose*

Figure 7 gives algorithm *FPclose*. Before calling *FPclose* with some (T, C), we already know from line 8 that there is no existing CFI X such that $T.base \subset X$, and $T.base$ and X have the same count. If there is only one single path in T, the nodes and their counts in this single path can be easily used to list the $T.base$-local closed frequent itemsets. These itemsets will be compared with the CFI's in C. If an itemset is closed, it is inserted into C. If the FP-tree T is not a single-path tree, we execute line 6. Lines 9 to 12 use the array technique. Lines 4 and 8 call function $closed\_checking(Y, C)$ to check if a frequent itemset Y is closed. If it is, the function returns true, otherwise, false is returned. Lines 14 and 15 construct Y's conditional FP-tree and CFI-tree. Then FPclose is called recursively for T_Y and C_Y.

Note that line 17 is not implemented as such. As in algorithm FPmax*, we found it more efficient to do the insertion of lines 3–5 into all CFI-trees currently in main memory.

CFI-trees are initialized similarly to MFI-trees, described in Section 3.3. The implementation of function *closed_checking* is almost the same as the implementation of function *subset_checking*, except now we also consider the count of an itemset. Given an itemset $Y = \{i_1, i_2, \ldots, i_k\}$ with count c, suppose the order of the items in header table of the current CFI-tree is $i_1, i_2, \ldots, i_k$. Following the linked list of i_k, for each node in the list, first we check if its count is equal to or greater than c. If it is, we then test if Y is a subset of the ancestors of that node. The function *closed_checking* returns true only when there is no existing CFI Z in the CFI-tree such that Z is a superset of Y and the count of Y is equal to or greater than the count of Z.

Memory management allocating and deallocating space for FP-trees and CFI-trees is similar to the memory management of *FPgrowth** and *FPmax**.

By a similar reasoning as in Section 3.5, we conclude that the total main memory requirement for running *FPclose* on a dataset is approximately sum of the size of the first FP-tree and its CFI-tree.

5. Experimental Evaluation

We now present a performance comparison of our FP-algorithms with algorithms dEclat, GenMax, CHARM and MAFIA. Algorithm dEclat is a depth-first search algorithm proposed by Zaki and Gouda in [10]. dEclat uses a linked list to organize frequent patterns, however, each itemset now corresponds to an array of transaction IDs (the "TID-array"). Each element in the array corresponds to a transaction that contains the itemset. Frequent itemset mining and candidate frequent itemset generation are done by TID-array intersections. A technique called *diffset*, is used for reducing the memory requirement of TID-arrays. The diffset technique only keeps track of differences in the TID's of a candidate itemsets when it is generating frequent itemsets. GenMax, also proposed by Gouda and Zaki [4], takes an approach called *progressive focusing* to do maximality testing. CHARM is proposed by Zaki and Hsiao [11] for CFI mining. In all three algorithms, the main operation is the intersection of TID-arrays. Each of them has been shown as one of the best algorithms for mining FI's, MFI's or CFI's. MAFIA is introduced in [3] by Burdick *et al*. for mining maximal frequent itemsets. It also has options for mining FI's and CFI's. We give the results of three different sets of experiments, one set for FI's, one for MFI's and one for CFI's.

The source codes for dEclat, CHARM, GenMax and MAFIA were provided by their authors. We ran all algorithms on many synthetic and real datasets. Due to the lack of space, only the results for two synthetic datasets and two real datasets are shown here. These datasets should be representative, as recent research papers [2, 3, 4, 11, 10, 8, 9], use these or similar datasets.

The two synthetic datasets, *T40I10D100K* and *T100I20D100K*, were generated from the application on the website of IBM [1]. They both use 100,000 transactions and 1000 items. The two real datasets, *pumsb** and *connect-4*, were also downloaded from the IBM website [2]. Dataset *connect-4* is compiled from game state information. Dataset *pumsb** is produced from census data of Public Use Microdata Sample (PUMS). These two real datasets are both quite dense, so a large number of frequent itemsets can be mined even for very high values of minimum support.

All experiments were performed on a 1Ghz Pentium III with 512 MB of memory running RedHat Linux 7.3. All times in the figures refer to CPU time.

5.1. FI Mining

In [6], the original FPgrowth method has been shown to be an efficient and scalable algorithm for mining frequent itemsets. FPgrowth is about an order of magnitude faster than the Apriori. Subsequently, it was shown in [10], that the algorithm dEclat outperforms FPgrowth on most datasets. Thus, in the first set of experiments, FP-growth* is compared with the original FP-growth method and with dEclat. The original FP-growth method is implemented on the basis of the paper [6]. In this set of experiments we also included with MAFIA [3], which has an option for mining all FI's. The results of the first set of experiments are shown in Figure 8.

Figure 8 (a) shows the CPU time of the four algorithms running on dataset *T40I10D100K*. We see that FPgrowth* is the best algorithm for this dataset. It outperforms dEclat and MAFIA at least by a factor of two. Main memory is used up by dEclat when the minimum support goes down to 0.25%, while FPgrowth* can still run for even smaller levels of minimum support. MAFIA is the slowest algorithm for this dataset and its CPU time increases rapidly.

Due to the use of the array technique, and the fact that *T40I10D100K* is a sparse dataset, FPgrowth* turns out to be faster than FPgrowth. However, when the minimum support is very low, we can expect the FP-tree to achieve a good compactification, starting at the initial recursion level. Thus the array technique does not offer a big gain. Consequently, as verified in Figure 8 (a), for very low levels minimum support, FPgrowth* and FPgrowth have almost the same running time.

Figure 8 (b) shows the CPU time for running the four algorithms on dataset *T100I20D100K*. The result is similar to the result in Figure 8 (a). FPgrowth* is again the best. Since the dataset *T100I20D100K* is sparser than *T40I10D100K*, the speedup from FPgrowth to FPgrowth* is increased.

From Figure 8 (c) and (d), we can see that the FP-methods are faster than dEclat by an order of magnitude in both experiments. Since *pumsb** and *connect-4* are both very dense datasets, FPgrowth* and FPgrowth have almost same running time, as the array technique does not achieve a significant speedup for dense datasets.

In Figure 8 (c), the CPU time increases drastically when the minimum support goes down below 25%. However, this is not a problem for FPgrowth and FPgrowth*, which still are able to produce results. The main reason for the nevertheless steeply increased CPU time is that a long time has to be spent listing frequent itemsets. Recall, that if there is a frequent "long" itemset of size ℓ, then we have to generate 2^ℓ frequent sets from it.

We also ran the four algorithms on many other datasets, and we found that FPgrowth* was always the fastest.

To see why FPgrowth* is the fastest, let us consider the main operations in the algorithms. As discussed before, FPgrowth* spends most of its time on constructing and traversing FP-trees. The main operation in dEclat is to generate new candidate FI's by TID-array intersections. In MAFIA, generating new candidate FI's by bitvector *and*-operations is the main work. Since FPgrowth* uses the compact FP-tree, further boosted by the array technique, the time it spends constructing and traversing the trees, is less than the time needed for TID-array intersections and bitvector *and*-operations. Moreover, the main memory space needed for storing FP-trees is far less than that for storing diffsets or bitvectors. Thus FPgrowth* runs faster than the other two algorithms, and it scales to very low levels of minimum support.

Figure 11 (a) shows the main memory consumption of three algorithms by running them on dataset *connect-4*. We can see that FP-growth* always use the least main memory. And even for very low minimum support, it still uses a small amount of main memory.

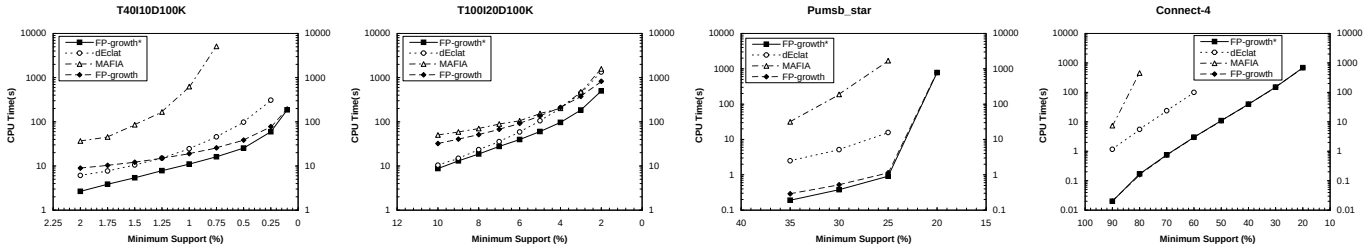
5.2. MFI Mining

In our paper [5], we analyzed and verified the performance of algorithm FPmax. We learned that FPmax outperformed GenMax and MAFIA in some, but not all cases. To see the impact of the new array technique and the new *subset_checking* function that we are using in FPmax*, in the second set of experiments, we compared FPmax* with FPmax, GenMax, and MAFIA.

Figure 9 (a) gives the result for running these algorithms on the sparse dataset *T40I10D100K*. We can see that FPmax is slower than GenMax for all levels of minimum support, while FPmax* outperforms GenMax by a factor of at least two. Figure 9 (b) shows the results for the very sparse dataset *T100I20D100K*, FPmax is the slowest algorithm, while FPmax* is the fastest algorithm. Figure 9 (c) shows that FPmax* is the fastest algorithm for the dense dataset *pumsb**, even though FPmax is the slowest algorithm on this dataset for very low levels of minimum support. In

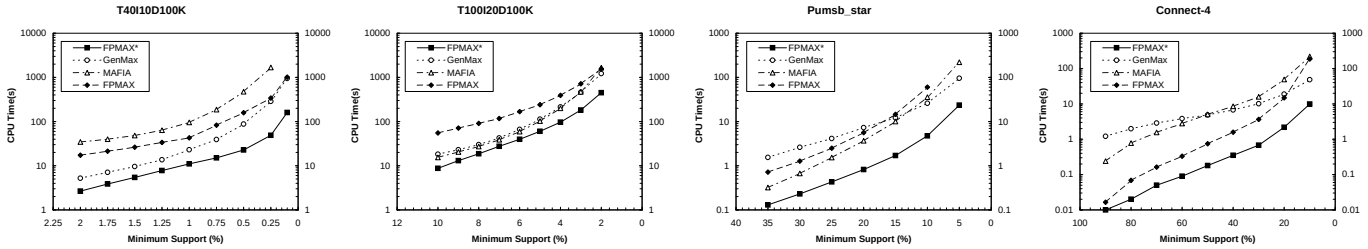
[1] http://www.almaden.ibm.com/cs/quest/syndata.html

[2] http://www.almaden.ibm.com/cs/people/bayardo/resources.html



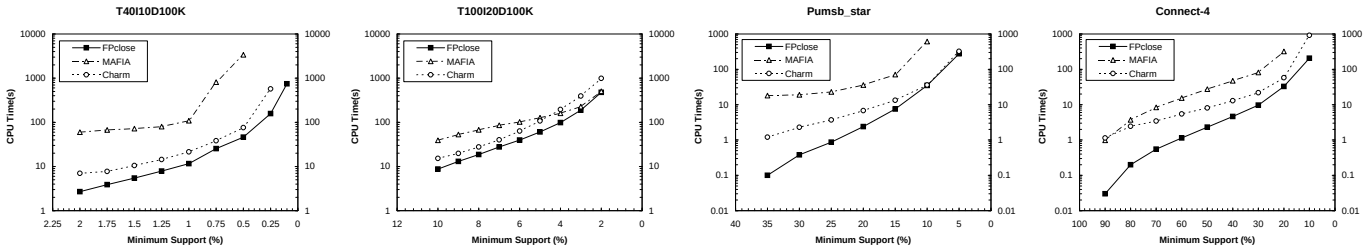
(a) (b) (c) (d)

Figure 8. Mining FI's



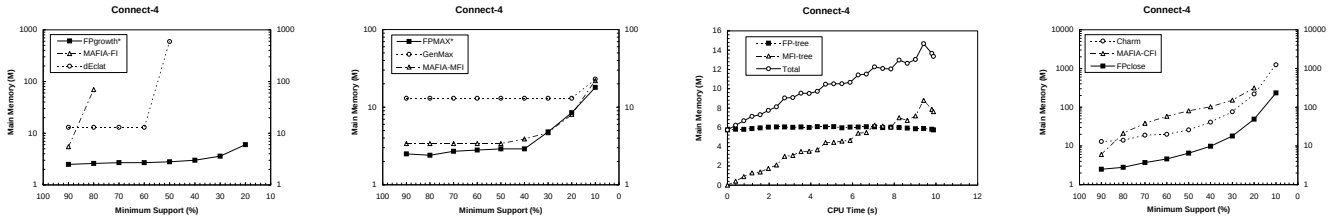
(a) (b) (c) (d)

Figure 9. Mining MFI's



(a) (b) (c) (d)

Figure 10. Mining CFI's



(a) (b) (c) (d)

Figure 11. Main Memory used by the algorithms

Figure 9 (d), FPmax outperforms GenMax and MAFIA for high levels of minimum support, but it is slow for very low levels. FPmax*, on the other hand is about one to two orders of magnitude faster than GenMax and MAFIA for all levels of minimum support.

All experiments in this second set show that the array technique and the new *subset_checking* function are indeed very effective. Figure 11 (b) shows the main memory used by three algorithms when running them on dataset *connect-4*. From the figure, we can see that FPmax* uses less main memory than the other algorithms. Figure 11 (c) shows the main memory used by FP-trees, MFI-trees and the whole algorithm when running FPmax* on dataset *connect-4*. The minimum support was set as 10%. In the figure, the last point of the line for FP-tree is for the main memory of the first FP-tree ($T_\emptyset$), since at this point the space for all condi-

tional FP-trees has been freed. The last point of the line for MFI-tree is for the main memory of the MFI-tree that contains whole set of MFI's, i.e., $M_\emptyset$. The figure confirms our analysis of main memory used by FPmax* in Section 3.5.

We also run these four algorithms on many other datasets, and we found that FPmax* always was the fastest algorithm.

5.3. CFI Mining

In the third set of experiments, the performances of FPclose, CHARM and MAFIA, with the option of mining closed frequent itemset, were compared.

Figure 10 shows the results of running FPclose, CHARM and MAFIA on datasets *T40I10D100K*, *T100I20D100K*, *pumsb** and *connect-4*. FPclose shows good performance on all datasets, due to the fact that it uses the compact FP-tree and the array technique. However, for very low levels of minimum support FPclose has performance similar to CHARM and MAFIA. By analyzing the three algorithms, we found that FPclose generates more non-closed frequent itemsets than the other algorithms. For each of the generated frequent itemsets, the function *closed_checking* must be called. Although the *closed_checking* function is very efficient, the increased number of calls to it means higher total running time. For high levels of minimum support, the time saved by using the compact FP-tree and the array technique compensates for the time FPclose spends on *closed_checking*. In all cases, FPclose uses less main memory for mining CFI's than CHARM and MAFIA. Figure 11 (d) shows the memory used by three algorithms by running them on dataset *connnect-4*. We can see that for very low levels of minimum support, CHARM and MAFIA were aborted because they ran out of memory, while FPclose was still able to run and produce output.

6. Conclusions

We have introduced a novel array-based technique that allows using FP-trees more efficiently when mining frequent itemsets. Our technique greatly reduces the time spent traversing FP-trees, and works especially well for sparse datasets. Furthermore, we presented new algorithms for mining maximal and closed frequent itemsets.

The FPgrowth* algorithm, which extends original FPgrowth method, also uses the novel array technique to mine all frequent itemsets.

For mining maximal frequent itemsets, we extended our earlier algorithm FPmax to FPmax*. FPmax* not only uses the array technique, but also a new subset-testing algorithm. For the subset testing, a variation of the FP-tree, an MFI-tree, is used for storing all already discovered MFI's. In FPmax*, a newly found FI is always compared with a small set of MFI's that are kept in an MFI-tree, thus making subset-testing much more efficient.

For mining closed frequent itemsets we give the FPclose algorithm. In the algorithm, a CFI-tree —another variation of a FP-tree— is used for testing the closedness of frequent itemsets.

For all of our algorithms we have presented several optimizations that further reduce their running time.

Our experimental results showed that FPgrowth* and FPmax* always outperforms existing algorithms. FPclose also demonstrates extremely good performance. All of the algorithms need less main memory because of the compact FP-trees, MFI-trees, and CFI-trees.

Though the experimental results given in this paper show the success of our algorithms, in the future we will test them on more applications to further study their performance. We are also planning to explore ways to improve the FPclose algorithm by reducing the number of closedness-tests needed.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of VLDB'94*, pages 487–499, 1994.

[2] R. J. Bayardo, Jr. Efficiently mining long patterns from databases. In *Proceedings of ACM SIGMOD'98*, pages 85–93, 1998.

[3] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *Proceedings of ICDE'01*, pages 443–452, Apr. 2001.

[4] K. Gouda and M. J. Zaki. Efficiently mining maximal frequent itemsets. In *Proceedings of ICDM'01*, San Jose, CA, Nov. 2001.

[5] G. Grahne and J. Zhu. High performance mining of maximal frequent itemsets. In *SIAM'03 Workshop on High Performance Data Mining: Pervasive and Data Stream Mining*, May 2003.

[6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proceedings of ACM SIGMOD'00*, pages 1–12, May 2000.

[7] J. Han, J. Wang, Y. Lu, and P. Tzvetkov. Mining top-k frequent closed patterns without minimum support. In *Proceedings of ICDM'02*, pages 211–218, Dec. 2002.

[8] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD'00 Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[9] J. Wang, J. Han, and J. Pei. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *Proceedings of ACM SIGKDD'03*, Washington, DC, 2003.

[10] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proceedings of ACM SIGKDD'03*, Washington, DC, Aug. 2003.

[11] M. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *Proceedings of SIAM'02*, Arlington, Apr. 2002.

COFI-tree Mining: A New Approach to Pattern Growth with Reduced Candidacy Generation

Mohammad El-Hajj
Department of Computing Science
University of Alberta Edmonton, AB, Canada
mohammad@cs.ualberta.ca

Osmar R. Zaïane
Department of Computing Science
University of Alberta Edmonton, AB, Canada
zaiane@cs.ualberta.ca

Abstract

Existing association rule mining algorithms suffer from many problems when mining massive transactional datasets. Some of these major problems are: (1) the repetitive I/O disk scans, (2) the huge computation involved during the candidacy generation, and (3) the high memory dependency. This paper presents the implementation of our frequent itemset mining algorithm, COFI, *which achieves its efficiency by applying four new ideas. First, it can mine using a compact memory based data structures. Second, for each frequent item assigned, a relatively small independent tree is built summarizing co-occurrences. Third, clever pruning reduces the search space drastically. Finally, a simple and non-recursive mining process reduces the memory requirements as minimum candidacy generation and counting is needed to generate all relevant frequent patterns.*

1 Introduction

Frequent pattern discovery has become a common topic of investigation in the data mining research area. Its main theme is to discover the sets of items that occur together more than a given threshold defined by the decision maker. A well-known application domain that counts on the frequent pattern discovery is the market basket analysis. In most cases when the support threshold is low and the number of frequent patterns "explodes", the discovery of these patterns becomes problematic for reasons such as: high memory dependencies, huge search space, and massive I/O required. However, recently new studies have been proposed to reduce the memory requirements [8], to decrease the I/O dependencies [7], still more promising issues need to be investigated such as pruning techniques to reduce the search space. In this paper we introduce a new method for frequent pattern discovery that is based on the Co-Occurrence Frequent Item tree concept [8, 9]. The new proposed method uses a pruning technique that dramatically saves the memory space. These relatively small trees are constructed based on a memory-based structure called FP-Trees [11]. This data structure is studied in detail in the following sections. In short, we introduced in [8] the COFI-tree stucture and an algorithm to mine it. In [7] we presented a disk based data structure, inverted matrix, that replaces the memory-based FP-tree and scales the interactive frequent pattern mining significantly. Our contributions in this paper are the introduction of a clever pruning technique based on an interesting property drawn from our top-down approach, and some implementation tricks and issues. We included the pruning in the algorithm of building the tree so that the pruning is done on the fly.

1.1 Problem Statement

The problem of mining association rules over market basket analysis was introduced in [2]. The problem consists of finding associations between items or itemsets in transactional data. The data could be retail sales in the form of customer transactions or even medical images [16]. Association rules have been shown to be useful for other applications such as recommender systems, diagnosis, decision support, telecommunication, and even supervised classification [5]. Formally, as defined in [3], the problem is stated as follows: Let $I = \{i_1, i_2, ...i_m\}$ be a set of literals, called items and m is considered the dimensionality of the problem. Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. A unique identifier *TID* is given to each transaction. A transaction T is said to contain X, a set of items in I, if $X \subseteq T$. An *association rule* is an implication of the form "$X \Rightarrow Y$", where $X \subseteq I$, $Y \subseteq I$, and $X \cap Y = \emptyset$. An itemset X is said to be *large* or *frequent* if its *support* s is greater or equal than a given minimum support threshold σ. An itemset X satisfies a constraint C if and only if C(X) is *true*. The rule $X \Rightarrow Y$ has a *support s* in the transaction set $\mathcal{D}$ if *s%* of the transactions in $\mathcal{D}$ contain $X \cup Y$. In other words, the support of the

rule is the probability that X and Y hold together among all the possible presented cases. It is said that the rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence c* if *c%* of transactions in $\mathcal{D}$ that contain X also contain Y. In other words, the confidence of the rule is the conditional probability that the consequent Y is true under the condition of the antecedent X. The problem of discovering all association rules from a set of transactions $\mathcal{D}$ consists of generating the rules that have a *support* and *confidence* greater than a given threshold. These rules are called *strong rules*. This association-mining task can be broken into two steps:
1. A step for finding all frequent k-itemsets known for its extreme I/O scan expense, and the massive computational costs;
2. A straightforward step for generating strong rules.

In this paper and our attached code, we focus exclusively on the first step: generating frequent itemsets.

1.2 Related Work

Several algorithms have been proposed in the literature to address the problem of mining association rules [12, 10]. One of the key algorithms, which seems to be the most popular in many applications for enumerating frequent itemsets, is the *apriori* algorithm [3]. This *apriori* algorithm also forms the foundation of most known algorithms. It uses an *anti-monotone* property stating that for a k-itemset to be frequent, all its (k-1)-itemsets have to be frequent. The use of this fundamental property reduces the computational cost of candidate frequent itemset generation. However, in the cases of extremely large input sets with big frequent 1-items set, the *Apriori* algorithm still suffers from two main problems of repeated I/O scanning and high computational cost. One major hurdle observed with most real datasets is the sheer size of the candidate frequent 2-itemsets and 3-itemsets.

TreeProjection is an efficient algorithm presented in [1]. This algorithm builds a lexicographic tree in which each node of this tree presents a frequent pattern. The authors report that their algorithm is one order of magnitude faster than the existing techniques in the literature. Another innovative approach of discovering frequent patterns in transactional databases, FP-Growth, was proposed by Han et al. in [11]. This algorithm creates a compact tree-structure, FP-Tree, representing frequent patterns, that alleviates the multi-scan problem and improves the candidate itemset generation. The algorithm requires only two full I/O scans of the dataset to build the prefix tree in main memory and then mines directly this structure. The authors of this algorithm report that their algorithm is faster than the *Apriori* and the TreeProjection algorithms. Mining the FP-tree structure is done recursively by building conditional trees that are of the same order of magnitude in number as the frequent patterns. This massive creation of conditional trees makes this algorithm not scalable to mine large datasets beyond few millions. In [14] the same authors propose a new algorithm, H-mine, that invokes FP-Tree to mine condensed data. This algorithm is still not scalable as reported by its authors in [13].

1.3 Preliminaries, Motivations and Contributions

The Co-Occurrence Frequent Item tree (or COFI-tree for short) and the *COFI* algorithm presented in this paper are based on our previous work in [7, 8]. The main motivation of our current research is the pruning technique that reduces the memory space needed by the COFI-trees. The presented algorithm is done in two phases in which phase 1 requires two full I/O scans of the transactional database to build the FP-Tree structure[11]. The second phase starts by building small Co-Occurrence Frequent trees for each frequent item. These trees are pruned first to eliminate any non-frequent items with respect to the COFI-tree based frequent item. Finally the mining process is executed.

The remainder of this paper is organized as follows: Section 2 describes the Frequent Pattern tree, design and construction. Section 3 illustrates the design, constructions, pruning, and mining of the Co-Occurrence Frequent Item trees. Section 4 presents the implementation procedure of this algorithm. Experimental results are given in Section 5. Finally, Section 6 concludes by discussing some issues and highlighting our future work.

2 Frequent Pattern Tree: Design and Construction

The COFI-tree approach we propose consists of two main stages. Stage one is the construction of a modified Frequent Pattern tree. Stage two is the repetitive building of small data structures, the actual mining for these data structures, and their release.

2.1 Construction of the Frequent Pattern Tree

The goal of this stage is to build the compact data structure called Frequent Pattern Tree [11]. This construction is done in two phases, where each phase requires a full I/O scan of the dataset. A first initial scan of the database identifies the frequent 1-itemsets. The goal is to generate an ordered list of frequent items that would be used when building the tree in the second phase.

This phase starts by enumerating the items appearing in the transactions. After enumeration these items (i.e. after reading the whole dataset), infrequent items with a support less than the support threshold are weeded out and the remaining frequent items are sorted by their frequency. This

Table 1. Transactional database

T.No.	Items				
T1	A	G	D	C	B
T2	B	C	H	E	D
T3	B	D	E	A	M
T4	C	E	F	A	N
T5	A	B	N	O	P
T6	A	C	Q	R	G
T7	A	C	H	I	G
T8	L	E	F	K	B
T9	A	F	M	N	O
T10	C	F	P	G	R
T11	A	D	B	H	I
T12	D	E	B	K	L
T13	M	D	C	G	O
T14	C	F	P	Q	J
T15	B	D	E	F	I
T16	J	E	B	A	D
T17	A	K	E	F	C
T18	C	D	L	B	A

list is organized in a table, called header table, where the items and their respective support are stored along with pointers to the first occurrence of the item in the frequent pattern tree. Phase 2 would construct a frequent pattern tree.

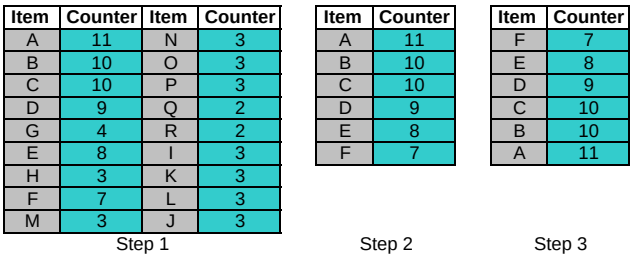


Figure 1. Steps of phase 1

Phase 2 of constructing the Frequent Pattern tree structure is the actual building of this compact tree. This phase requires a second complete I/O scan from the dataset. For each transaction read, only the set of frequent items present in the header table is collected and sorted in descending order according to their frequency. These sorted transaction items are used in constructing the FP-Trees as follows: for the first item on the sorted transactional dataset, check if it exists as one of the children of the root. If it exists then increment the support for this node. Otherwise, add a new node for this item as a child for the root node with 1 as support. Then, consider the current item node as the new temporary root and repeat the same procedure with the next item on the sorted transaction. During the process of adding any new item-node to the FP-Tree, a link is maintained between this item-node in the tree and its entry in the header table. The header table holds as one pointer per item that points to the first occurrences of this item in the FP-Tree structure.

2.2 Illustrative Example

For illustration, we use an example with the transactions shown in Table 1. Let the minimum support threshold be set to 4. Phase 1 starts by accumulating the support for all items that occur in the transactions. Step 2 of phase 1 removes all non-frequent items, in our example (G, H, I, J, K, L,M, N, O, P, Q and R), leaving only the frequent items (A, B, C, D, E, and F). Finally all frequent items are sorted according to their support to generate the sorted frequent 1-itemset. This last step ends phase 1 in Figure 1 of the COFI-tree algorithm and starts the second phase. In phase 2, the first transaction (A, G, D, C, B) is filtered to consider only the frequent items that occur in the header table (i.e. A, D, C and B). This frequent list is sorted according to the items' supports (A, B, C and D). This ordered transaction generates the first path of the FP-Tree with all item-node support initially equal to 1. A link is established between each item-node in the tree and its corresponding item entry in the header table. The same procedure is executed for the second transaction (B, C, H, E, and D), which yields a sorted frequent item list (B, C, D, E) that forms the second path of the FP-Tree. Transaction 3 (B, D, E, A, and M) yields the sorted frequent item list (A, B, D, E) that shares the same prefix (A, B) with an existing path on the tree. Item-nodes (A and B) support is incremented by 1 making the support of (A) and (B) equal to 2 and a new sub-path is created with the remaining items on the list (D, E) all with support equal to 1. The same process occurs for all transactions until we build the FP-Tree for the transactions given in Table 1. Figure 2 shows the result of the tree building process. Notice that in our tree structure, contrary to the original FP-tree [11], our links are bi-directional. This, and other differences presented later, are used by our mining algorithm.

3 Co-Occurrence Frequent-Item-trees: Construction, Pruning and Mining

Our approach for computing frequencies relies first on building independent, relatively small trees for each frequent item in the header table of the FP-Tree called COFI-trees. A pruning technique is applied to remove all non-frequent items with respect to the main frequent item of the tested COFI-tree. Then we mine separately each one of the trees as soon as they are built, minimizing the candidacy generation and without building conditional sub-trees recursively. The trees are discarded as soon as mined. At

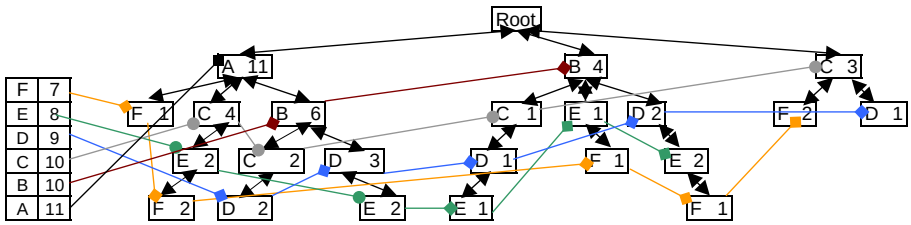


Figure 2. Frequent Pattern Tree.

any given time, only one COFI-tree is present in main memory. In our following examples we always assume that we are building the COFI-trees based on the modified FP-Tree data-structure presented above.

3.1 Pruning the COFI-trees

Pruning can be done after building a tree or, even better, while building it. We opted for pruning on the fly since the overhead is minimal but the consequences are drastic reduction in memory requirements. We will discuss the pruning idea, then present the building algorithm that considers the pruning on the fly.

In this section we are introducing a new *anti-monotone* property called global frequent/local non-frequent property. This property is similar to the *Apriori* one in the sense that it eliminates at the i^{th} level all non-frequent items that will not participate in the (i+1) level of candidate itemsets generation. The difference between the two properties is that we extended our property to eliminate also frequent items which are among the i-itemset and we are sure that they will not participate in the (i+1) candidate set. The *Apriori* property states that *all nonempty subsets of a frequent itemset must also be frequent*. An example is given later in this section to illustrate both properties. In our approach, we are trying to find all frequent patterns with respect to one frequent item, which is the base item of the tested COFI-tree. We already know that all items that participate in the creation of the COFI-tree are frequent with respect to the global transaction database, but that does not mean that they are also locally frequent with respect to the based item in the COFI-tree. The global frequent/local non-frequent property states that *all nonempty subsets of a frequent itemset with respect to the item A of the A-COFI-tree , must also be frequent with respect to item A*. For each frequent item A we traverse the FP-Tree to find all frequent items that occur with A in at least one transaction (or branch in the FP-Tree) with their number of occurrences. All items that are locally frequent with item A will participate in building the A-COFI-tree, other global frequent items, locally non-frequent items will not participate in the creation of the A-COFI-tree. In our example we can find that all items that participate in the creation of the F-COFI-tree are locally not frequent with respect to item F as the support for all these items are not greater than the support threshold σ which is equal to 4, Figure 3. From knowing this, there will be no need to mine the F-COFI-tree, we already know that no frequent patterns other than the item F will be generated. We can extend our knowledge at this stage to know that item F will not appear in any of the frequent patterns. The COFI-tree for item E indicates that only items D, and B are frequent with respect to item E, which means that there will be no need to test patterns as EC, and EA. The COFI-tree for item D indicates that item C will be eliminated, as it is not frequent with respect to item D. C-COFI-tree ignores item B for the same reason. To sum up the *Apriori* property states in our example of 6 1-frequent itemset that we need to generate 15 2-Candidate itemset which are (A,B), (A,C), (A,D), (A,E), (A,F), (B,C), (B,D), (B,E), (B,F), (C,D), (C,E), (C,F), (D,E), (D,F), (E,F), using our property we have eliminated (not generated or counted) 9 patterns which are (A,E), (A,F), (B,C), (B,F), (C,D), (C,E), (C,F), (D,F), (E,F) leaving only 6 patterns to test which are (A,B), (A,C), (A,D), (B,D), (B,E), (D,E).

3.2 Construction of the Co-Occurrence Frequent-Item-trees

The small COFI-trees we build are similar to the conditional FP-Trees [11] in general in the sense that they have a header with ordered frequent items and horizontal pointers pointing to a succession of nodes containing the same frequent item, and the prefix tree per se with paths representing sub-transactions. However, the COFI-trees have bi-directional links in the tree allowing bottom-up scanning as well, and the nodes contain not only the item label and a frequency counter, but also a participation counter as explained later in this section. The COFI-tree for a given frequent item x contains only nodes labeled with items that are more frequent or as frequent as x.

To illustrate the idea of the COFI-trees, we will explain step by step the process of creating COFI-trees for the FP-Tree of Figure 2. With our example, the first Co-Occurrence Frequent Item tree is built for item F as it is the least frequent item in the header table. In this tree for F, all frequent items, which are more frequent than F, and share transac-

tions with F, participate in building the tree. This can be found by following the chain of item F in the FP-Tree structure. The F-COFI-tree starts with the root node containing the item in question, then a scan of part of the FP-Tree is applied following he chain of the F item in the FP-Tree. The first branch FA has frequency of 1, as the frequency of the branch is the frequency of the test item, which is F. The goal of this traversal is to count the frequency of each frequent item with respect to item F. By doing so we can find that item E occurs 4 times, D occurs 2 times, C occurs 4 times, B 2 times, and A 3 times, by applying the *anti-monotone* constraint property we can predict that item F will never appear in any frequent pattern except itself. Consequently there will be no need to continue building the F-COFI-tree.

The next frequent item to test is E. The same process is done to compute the frequency of each frequent items with respect to item E. From this we can find that only two globally frequent items are also locally frequent which are (D:5 and B:6). For each sub-transaction or branch in the FP-Tree containing item E with other locally frequent items that are more frequent than E which are parent nodes of E, a branch is formed starting from the root node E. the support of this branch is equal to the support of the E node in its corresponding branch in FP-Tree. If multiple frequent items share the same prefix, they are merged into one branch and a counter for each node of the tree is adjusted accordingly. Figure 3 illustrates all COFI-trees for frequent items of Figure 2. In Figure 3, the rectangle nodes are nodes from the tree with an item label and two counters. The first counter is a *support-count* for that node while the second counter, called *participation-count*, is initialized to 0 and is used by the mining algorithm discussed later, a horizontal link which points to the next node that has the same *item-name* in the tree, and a bi-directional vertical link that links a child node with its parent and a parent with its child. The bi-directional pointers facilitate the mining process by making the traversal of the tree easier. The squares are actually cells from the header table as with the FP-Tree. This is a list made of all frequent items that participate in building the tree structure sorted in ascending order of their global support. Each entry in this list contains the *item-name*, *item-counter*, and a *pointer* to the first node in the tree that has the same *item-name*.

To explain the COFI-tree building process, we will highlight the building steps for the E-COFI-tree in Figure 3. Frequent item E is read from the header table and its first location in the FP-Tree is located using the pointer in the header table. The first location of item E indicate that it shares a branch with items CA, with support = 2, since none of these items are locally frequent then only the support of the E root node is incremented by 2. the second node of item E indicates that it shares items DBA with support equals to 2 for this branch as the support of the E-item is considered the

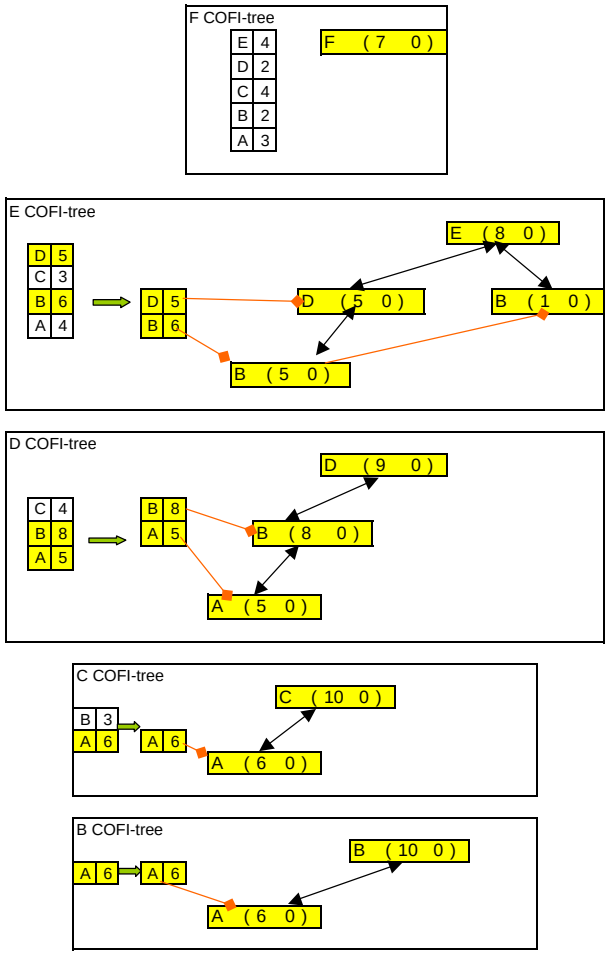


Figure 3. COFI-trees

support for this branch (following the upper links for this item). Two nodes are created, for items D and B with support equals to 2, D is a child node of B, and B is a child node of E. The third location of E indicate having EDB:1, which shares an existing branch in the E-COFI-tree, all counters are adjusted accordingly. A new branch of EB: 1 is created as the support of E=1 for the fourth occurrences of E. The final occurrence EDB: 2 uses an existing branch and only counters are adjusted. Like with FP-Trees, the header constitutes a list of all frequent items to maintain the location of first entry for each item in the COFI-tree. A link is also made for each node in the tree that points to the next location of the same item in the tree if it exists. The mining process is the last step done on the E-COFI-tree before removing it and creating the next COFI-tree for the next item in the header table.

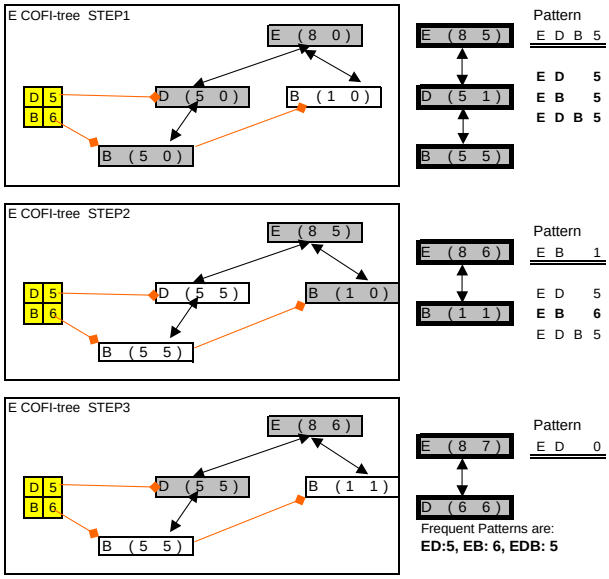


Figure 4. Steps needed to generate frequent patterns related to item E

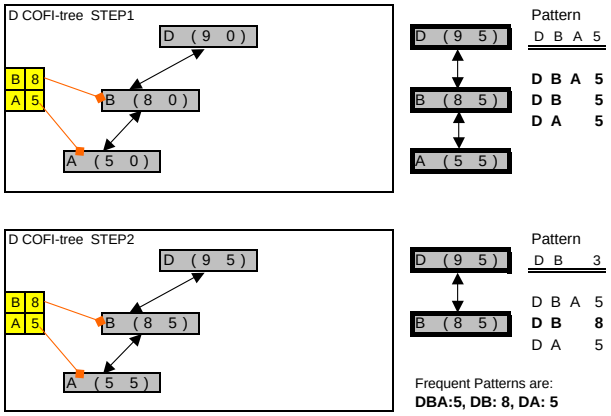


Figure 5. Steps needed to generate frequent patterns related to item D

3.3 Mining the COFI-trees

The COFI-trees of all frequent items are not constructed together. Each tree is built, mined, then discarded before the next COFI-tree is built. The mining process is done for each tree independently with the purpose of finding all frequent *k*-itemset patterns in which the item on the root of the tree participates.

Steps to produce frequent patterns related to the E item for example, as the F-COFI-tree will not be mined based on the pruning results we found on the previous step, are illustrated in Figure 4. From each branch of the tree, using the *support-count* and the *participation-count*, candidate frequent patterns are identified and stored temporarily in a list. The non-frequent ones are discarded at the end when all branches are processed. The mining process for the E-COFI-tree starts from the most locally frequent item in the header table of the tree, which is item B. Item B exists in two branches in the E-COFI-tree which are (B:5, D:5 and E:8), and (B:1, and E:8). The frequency of each branch is the frequency of the first item in the branch minus the participation value of the same node. Item B in the first branch has a frequency value of 5 and participation value of 0 which makes the first pattern EDB frequency equals to 5. The participation values for all nodes in this branch are incremented by 5, which is the frequency of this pattern. In the first pattern EDB: 5. We need to generate all sub-patterns that item E participates in, which are ED: 5, EB: 5, and EDB: 5. The second branch that has B generates the pattern EB: 1. EB already exists and its counter is adjusted to become 6. The COFI-tree of Item E can be removed at this time and another tree can be generated and tested to produce all the frequent patterns related to the root node. The same process is executed to generate the frequent patterns. The D-COFI-tree (Figure 5) is created after the E-COFI-tree. Mining this tree generates the following frequent patterns: DBA: 5, DA: 5, and DB:8. The same process occurs for the remaining trees that would produce AC: 6 for the C-COFI-tree and BA:6 for the B-COFI-tree.

The following is our algorithm for building and mining the COFI-trees with pruning.

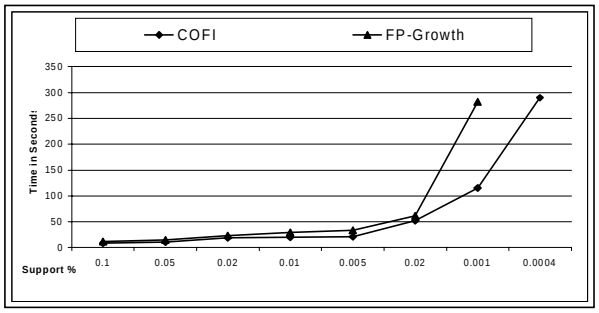
Algorithm COFI: Creating with pruning and Mining COFI-trees
Input: modified FP-Tree, a minimum support threshold σ
Output: Full set of frequent patterns
Method:
1. A = the least frequent item on the header table of FP-Tree
2. While (There are still frequent items) do
   2.1 count the frequency of all items that share item (A) a path. Frequency of all items that share the same path are the same as of the frequency of the (A) items
   2.2 Remove all non-locally frequent items for the frequent list of item (A)
   2.3 Create a root node for the (A)-COFI-tree with both *frequency-count* and *participation-count* = 0
      2.3.1 C is the path of locally frequent items in the path of item A to the root
      2.3.2 Items on C form a prefix of the (A)-COFI-tree.
      2.3.3 If the prefix is new then Set *frequency-count*= frequency of (A) node and *participation-count*= 0 for all nodes in the path
      Else

2.3.4 Adjust the *frequency-count* of the already exist part of the path.
2.3.5 Adjust the pointers of the *Header list* if needed
2.3.6 find the next node for item A in the FP-tree and go to 2.3.1
2.4 MineCOFI-tree (A)
2.5 Release (A) COFI-tree
2.6 A = next frequent item from the header table
3. Goto 2

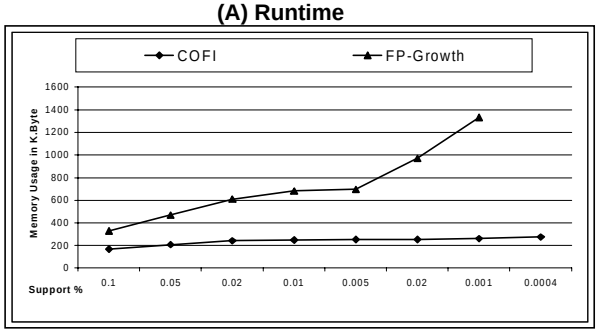
Function: MineCOFI-tree (A)
1. nodeA = select_next_node //Selection of nodes starts with the node of most locally frequent item and following its chain, then the next less frequent item with its chain, until we reach the least frequent item in the *Header list* of the (A)-COFI-tree
2. while there are still nodes do
2.1 D = set of nodes from nodeA to the root
2.2 F = nodeA.*frequency-count*-nodeA.*participation-count*
2.3 Generate all Candidate patterns X from items in D. Patterns that do not have A will be discarded.
2.4 Patterns in X that do not exist in the A-Candidate List will be added to it with frequency = F otherwise just increment their frequency with F
2.5 Increment the value of *participation-count* by F for all items in D
2.6 nodeA = select_next_node
3. Goto 2
4. Based on support threshold σ remove non-frequent patterns from A Candidate List.

4 Experimental Studies

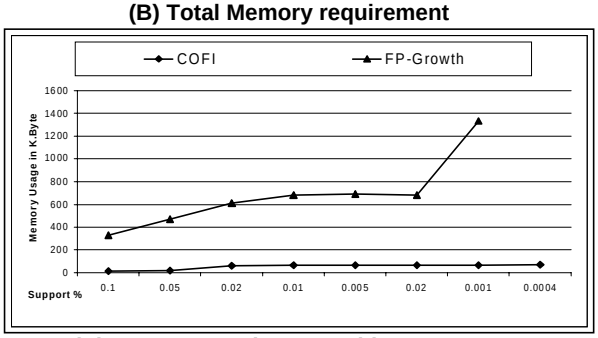
To study the COFI-tree mining strategies we have conducted several experiments on a variety of data sizes comparing our approach with the well-known FP-Growth [11] algorithm written by its original authors. The experiments were conducted on 2.6 GHz CPU machine with 2 Gbytes of memory using Win2000 operating system. Transactions were generated using IBM synthetic data generator [4]. We have conducted several types of experiments to test the effect of changing the support, transaction size, dimension, and transaction length. The first set of experiments were tested on a transaction database of 500K transactions, 10K the dimension, and the average transaction length was 12. We have varied the support from absolute value of 500 to 2 in which frequent patterns generated varied from 15K to 3400K patterns. FP-Growth could not mine the last experiment in this set as it used all available memory space. In all experiments the COFI-tree approach outperforms the FP-Growth approach. The major accomplishment of our approach is in the memory space saved. Our algorithm outperforms the FP-Growth by one order of magnitude in terms of memory space requirements. We have also tested the memory space used during the mining process only, (i.e, isolating the memory space used to create the FP-Tree by both FP-growth and COFI-tree FP-Tree based algorithms). We have found also that the COFI-tree approach outperforms the FP-tree by one order of magnitude in terms of memory space used by the COFI-tree compared with the conditional trees used by FP-Growth during the mining process. Figure 6A presents the time needed to mine 500K transactions using different support levels. Figure 6B depicts the memory needed during the mining process of the previous experiments. Figure 6C illustrates the memory needed by



(A) Runtime



(B) Total Memory requirement



(C) Memory requirement without FP-tree

No.of transactions = 500K, Dimension= 10K,
Average no. of items / transaction = 12

Figure 6. Mining dataset of 500K transactions

Table 2. Time and Memory Scalability with respect to support on the T10I4D100K dataset

	Time in Seconds		Memory in KB	
Support %	COFI	FP-Growth	COFI	FP-Growth
0.50	1.5	3.0	18	173
0.25	1.7	5.2	19	285
0.10	2.7	12.3	26	289
0.05	14.0	20.9	19	403

the COFI-trees and Conditional trees during the mining process. Other experiments were conducted to test the effect of changing the dimension, transaction size, transaction length using the same support which is 0.05%. Some of these experiments are represented in Figure 7. Figures 7A and 7B represent the time needed during the mining process. Figures 7C and 7D represent the memory space needed during the whole mining process. Figures 7E and 7F represent the memory space needed by the COFI-trees or conditional trees during the mining process. In these experiments we have varied the dimension, which is the number of distinct items from 5K to 10K, the average transaction length from 12 to 24 items in one transaction, and the number of transactions from 10K to 500K. All these experiments depicted the fact that our approach is one order of magnitude better than the FP-Growth approach in terms of memory usage.

We also run experiments using the public UCI datasets provided on the FIMI workshop website, which are Mushroom, Chess, Connect, Pumsb, T40I10D100K, and T10I4D100K. The COFI algorithm scales relatively well vis-à-vis the support threshold with these datasets. Results are not reported here for lack of space. Our approach revealed good results with high support value on all datasets. However, like with other approaches, in cases of low support value, where the number of frequent patterns increases significantly, our approach faces some difficulties. For such cases it is recommended to consider discovering closed itemsets or maximal patterns instead of just frequent itemsets. The sheer number of frequent itemsets becomes overwhelming, and some argue even useless. Closed itemsets and maximal itemsets represent all frequent patterns by eliminating the redundant ones. For illustration, Table 2 compares the CPU time and memory requirement for COFI and FP-Growth on the T10I4D100K dataset.

5 Implementations

The COFI-tree program submitted with this paper is a C++ code. The executable of this code runs with 3 parameters, which are: (1) the path to the input file name. (2) a positive integer that presents the absolute support. (3) An optional file name for the out patterns. This code generates ALL frequent patterns from the provided input file. The code scans the database twice. The goal of the first database scan is to find the frequency of each item in this transactional database. These frequencies are stored in a data structure called Candidate-Items. Each entry of this candidate items is a structure called ItemsStructure that is made of two long integers representing the item and its frequency. All frequent items are then stored in a special data structure called F1-Items. This data structure is sorted in descending order based on the frequency of each item. To access the location of each item we map it with a specific location using a new data structure called FindInHashTable. In brief, since we do not know the number of unique items at runtime, and thus can't create an array for counting the items, rather than having a linked list of items, we create blocks of p items. The number p could arbitrarily be 100 or 1000. Indeed, following links in a linked list each time to find and increment a counter could be expensive. Instead, blocs of items are easily indexed. In the worst case, we could lose the space of $p - 1$ unused items.

The second scan starts by eliminating all non frequent items from each transaction read and then sort this transaction based on the frequency of each frequent item. This process occurred in the Sort-Transaction method. The FP-tree is built based on the sub-transaction made of the frequent items. The FP-tree data structure is a tree of n children. The structure struct FPTTree { long Element; long counter; FPTTree* child; FPTTree* brother; FPTTree* father; FPTTree* next; } has been used to create each node of this tree, where a link is created between each node and its first child, and the brother link is maintained to create a linked list of all children of the same node. This linked list is built ordered based on the frequency of each item. The header list is maintained using the structure FrequentStruc { long Item; long Frequency; long COFIFrequency; long COFIFrequency1; FPTTree* first; COFITree* firstCOFI; }; After building the FP-tree we start building the first COFI-tree by selecting the item with least frequency from the frequent list. A scan is made of the FP-tree starting from the linked list of this item to find the frequency of other items with respect to this item. After that, the COFI-tree is created based on only the locally frequent items. Finally frequent patterns are generated and stored in the FrequentTree data structure. All nodes that have support greater or equal than the given support present a frequent pattern. The COFI-tree and the FrequentTree are removed from memory and the next COFI-tree is created until we mine all frequent trees.

One interesting implementation improvement is the fact that the participation counter was also added to the header table of the COFI-tree this counter cumulates the participation of the item in all paterns already discovered in the current COFI-tree. The difference between the participa-

tion in the node and the participation in the header is that the counter in the node counts the participation of the node item in all paths where the node appears, while the new counter in the COFI-tree header counts the participation of the item globally in the tree. This trick does not compromise the effectiveness and usefulness of the participation counting. One main advantage of this counter is that it looks ahead to see if all nodes of a specific item have already been traversed or not to reduce the unneeded scans of the COFI-tree.

6 Conclusion and future work

The COFI algorithm, based on our COFI-tree structure, we propose in this paper is one order of magnitude better than the FP-Growth algorithm in terms of memory usage, and sometimes in terms of speed. This Algorithm achieves this results thanks to: (1) the non recursive technique used during the mining process, in which with a simple traversal of the COFI-tree a full set of frequent patterns can be generated. (2) The pruning method that is used to remove all locally non frequent patterns, leaving the COFI-tree with only locally frequent items.

The major advantage of our algorithm *COFI* over FP-Growth is that it needs a significantly smaller memory footprint, and thus can mine larger transactional databases with smaller main memory available. The fundamental difference, is that COFI tries to find a compromise between a fully pattern growth approach, that FP-Growth adopts, and a total candidacy generation approach that apriori is known for. COFI grows targeted patterns but performs a reduced and focused generation of candidates during the mining. This is to avoid the recursion that FP-growth uses, and notorious to blow the stack with large datasets.

We have developed algorithms for closed itemset mining and maximal itemset mining based on our COFI-tree approach. However, their efficient implementations were not ready by the deadline of this workshop. These efficient algorithms and experimental results will be compared to existing algorithms such as CHARM[17], MAFIA[6] and CLOSET+[15], and will be reported in the future.

7 Acknowledgments

This research is partially supported by a Research Grant from NSERC, Canada.

References

[1] R. Agarwal, C.Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Parallel and distributed Computing*, 2000.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. 1993 ACM-SIGMOD Int. Conf. Management of Data*, pages 207–216, Washington, D.C., May 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. 1994 Int. Conf. Very Large Data Bases*, pages 487–499, Santiago, Chile, September 1994.

[4] I. Almaden. Quest synthetic data generation code. http://www.almaden.ibm.com/cs/quest/syndata.html.

[5] M.-L. Antonie and O. R. Zaïane. Text document categorization by term association. In *IEEE International Conference on Data Mining*, pages 19–26, December 2002.

[6] C. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A maximal frequent itemset algorithm for transactional databases. In *IEEE International Conference on Data Mining (ICDM 01)*, April 2001.

[7] M. El-Hajj and O. R. Zaïane. Inverted matrix: Efficient discovery of frequent items in large datasets in the context of interactive mining. In *In Proc. 2003 Int'l Conf. on Data Mining and Knowledge Discovery (ACM SIGKDD)*, August 2003.

[8] M. El-Hajj and O. R. Zaïane. Non recursive generation of frequent k-itemsets from frequent pattern tree representations. In *In Proc. of 5th International Conference on Data Warehousing and Knowledge Discovery (DaWak'2003)*, September 2003.

[9] M. El-Hajj and O. R. Zaïane. Parallel association rule mining with minimum inter-processor communication. In *Fifth International Workshop on Parallel and Distributed Databases (PaDD'2003) in conjunction with the 14th Int' Conf. on Database and Expert Systems Applications DEXA2003*, September 2003.

[10] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufman, San Francisco, CA, 2001.

[11] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *ACM-SIGMOD*, Dallas, 2000.

[12] J. Hipp, U. Guntzer, and G. Nakaeizadeh. Algorithms for association rule mining - a general survey and comparison. *ACM SIGKDD Explorations*, 2(1):58–64, June 2000.

[13] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by oppotunistic projection. In *Eight ACM SIGKDD Internationa Conf. on Knowledge Discovery and Data Mining*, pages 229–238, Edmonton, Alberta, August 2002.

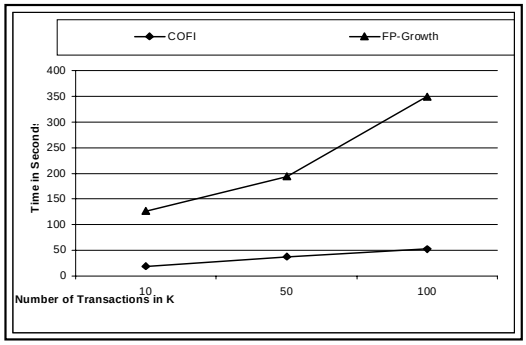
[14] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, pages 441–448, 2001.

[15] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the best strategies for mining frequent closed itemsets. In *9th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, July 2003.

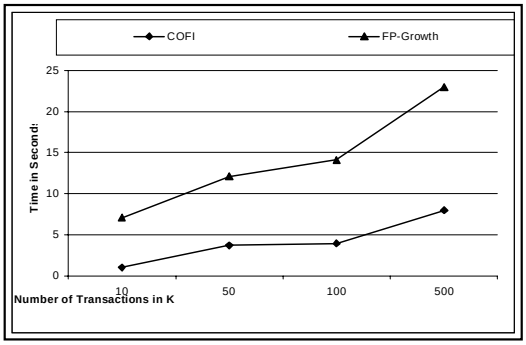
[16] O. R. Zaïane, J. Han, and H. Zhu. Mining recurrent items in multimedia with progressive resolution refinement. In *Int. Conf. on Data Engineering (ICDE'2000)*, pages 461–470, San Diego, CA, February 2000.

[17] M. Zaki and C.-J. Hsiao. ChARM: An efficient algorithm for closed itemset mining. In *2nd SIAM International Conference on Data MIning*, April 2002.

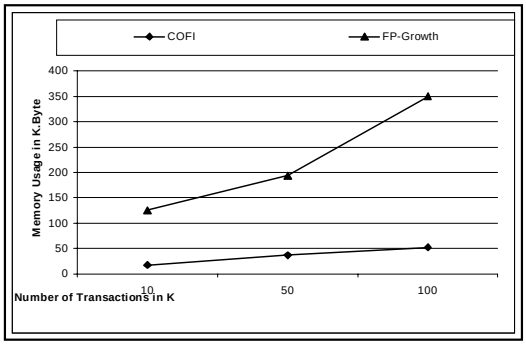
D = Dimension, L = Average number of items in one transaction
Support = 0.05%



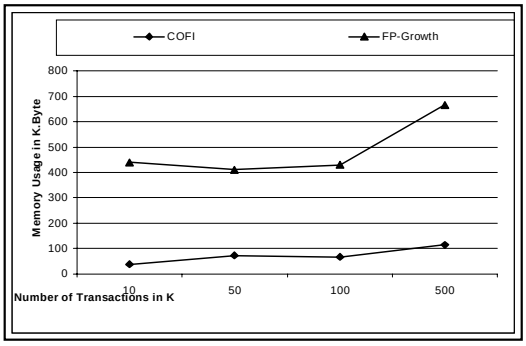
(A) D=5K, L=12



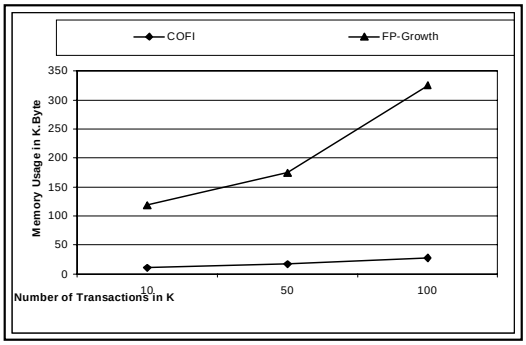
(B) D=10K, L=24



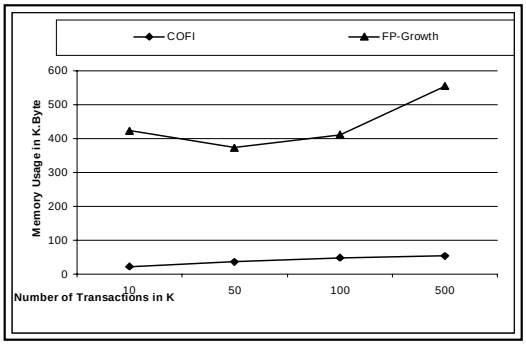
(C) D=5K, L=12



(D) D=10K, L=24



(E) D=5K, L=12



(F) D=10K, L=24

Figure 7. Mining dataset of different sizes

Mining Frequent Itemsets using Patricia Tries *

Andrea Pietracaprina and Dario Zandolin
Department of Information Engineering
University of Padova
andrea.pietracaprina@unipd.it, dzandol@tin.it

Abstract

We present a depth-first algorithm, PatriciaMine, that discovers all frequent itemsets in a dataset, for a given support threshold. The algorithm is main-memory based and employs a Patricia trie to represent the dataset, which is space efficient for both dense and sparse datasets, whereas alternative representations were adopted by previous algorithms for these two cases. A number of optimizations have been introduced in the implementation of the algorithm. The paper reports several experimental results on real and artificial datasets, which assess the effectiveness of the implementation and show the better performance attained by PatriciaMine with respect to other prominent algorithms.

1. Introduction

In this work, we focus on the problem of finding *all frequent itemsets* in a dataset $\mathcal{D}$ of transactions over a set of items $\mathcal{I}$, that is, all itemsets $X \subseteq \mathcal{I}$ contained in a number of transactions greater than or equal to a certain given threshold [2].

Several algorithms proposed in the literature to discover all frequent itemsets follow a depth-first approach by considering one item at a time and generating (recursively) all frequent itemsets which contain that item, before proceeding to the next item. A prominent member of this class of algorithms is FP-Growth proposed in [7]. It represents the dataset $\mathcal{D}$ through a standard trie (*FP-tree*) and, for each frequent itemset X, it materializes a projection $\mathcal{D}_X$ of the dataset on the transactions containing X, which is used to recursively discover all frequent supersets $Y \supset X$. This approach is very effective for dense datasets, where the trie achieves high compression, but becomes space inefficient when the dataset is sparse, and incurs high costs due to the frequent projections.

Improved variants of FP-Growth appeared in the literature, which avoid physical projections of the dataset (Top-Down FP-Growth [14]), or employ two alternative array-based and trie-based structures to cope, respectively, with sparse and dense datasets, switching adaptively from one to the other (H-mine [12]). The most successful ideas developed in these works have been gathered and further refined in OpportuneProject [9] which opportunistically selects the best strategy based on the characteristics of the dataset.

In this paper, we present an algorithm, PatriciaMine, which further improves upon the aforementioned algorithms stemmed from FP-Growth. Our main contribution is twofold:

- We use a compressed (Patricia) trie to store the dataset, which provides a space-efficient representation for both sparse and dense datasets, without resorting to two alternative structures, namely array-based and trie-based, as was suggested in [12, 9]. Indeed, by featuring a smaller number of nodes than the standard trie, the Patricia trie exhibits lower space requirements, especially in the case of sparse datasets, where it becomes comparable to the natural array-based representation, and reduces the amount of bookkeeping operations. Both theoretical and experimental evidence of these facts is given in the paper.
- A number of optimizations have been introduced in the implementation of PatriciaMine. In particular, a heuristic has been employed to limit the number of physical projections of the dataset during the course of execution, with the intent to avoid the time and space overhead incurred by projection, when not beneficial. Moreover, novel mechanisms have been developed for directly generating groups of itemsets supported by the same subset of transactions, and for visiting the trie without traversing individual nodes multiple times. The effectiveness of these optimizations is discussed in the paper.

We coded PatriciaMine in C, and compared its performance with that of a number of prominent algorithms,

*This research was supported in part by MIUR of Italy under project "ALINWEB: Algorithmics for Internet and the Web".

whose source/object code was made available to us, on several real and artificial datasets. The experiments provide clear evidence of the higher performance of PatriciaMine with respect to these other algorithms on both dense and sparse datasets. It must be remarked that our focus is on main-memory execution, in the sense PatriciaMine works under the assumption that the employed representation of the dataset fits in main memory. If this is not the case, techniques such as those suggested in [13, 9] could be employed, but this is beyond the scope of this work.

The rest of the paper is organized as follows. Section 2 introduces some notation and illustrates the datasets used in the experiments. Section 3 presents the main iterative strategy adopted by PatriciaMine, which can be regarded as a reformulation (with some modifications) of the recursive strategies adopted in [7, 12, 14, 9]. Sections 4 and 5 describe the most relevant features of the algorithm implementation, while the experimental results are reported and discussed in Section 6.

2. Preliminaries

Let $\mathcal{I}$ be a set of *items*, and $\mathcal{D}$ a set of *transactions*, where each transaction $t \in \mathcal{D}$ consists of a distinct identifier t_{id} and a subset of items $t_{\text{set}} \subseteq \mathcal{I}$. For an *itemset* $X \subseteq \mathcal{I}$, its *support* in $\mathcal{D}$, denoted by $\text{supp}_{\mathcal{D}}(X)$, is defined as the number of transactions $t \in \mathcal{D}$ such that $X \subseteq t_{\text{set}}$. Given an absolute support threshold *min_sup*, with $0 < \text{min\_sup} \leq |\mathcal{D}|$, an itemset $X \subseteq \mathcal{I}$ is *frequent w.r.t.* $\mathcal{D}$ and min_sup, if $\text{supp}_{\mathcal{D}}(X) \geq \text{min\_sup}$. With a slight abuse of notation, we call an item $i \in \mathcal{I}$ *frequent* if $\{i\}$ is frequent, and refer to $\text{supp}_{\mathcal{D}}(\{i\})$ as the support of i[1]. We study the problem of determining the set of all frequent itemsets for given $\mathcal{D}$ and min_sup, which we denote by $\mathcal{F}(\mathcal{D}, \text{min\_sup})$. For an itemset $X \subseteq \mathcal{I}$, we denote by $\mathcal{D}_X$ the subset of $\mathcal{D}$ projected on those transactions that contain X.

Let $\mathcal{I}' = \{i_1, i_2, \ldots\} \subseteq \mathcal{I}$ denote the subset of frequent items ordered by increasing support, and assume that the items in each frequent itemset are ordered accordingly. As observed in [1, 9], the set $\mathcal{F}(\mathcal{D}, \text{min\_sup})$ can be conveniently represented through a standard trie [8], called *Frequent ItemSet Tree* (FIST), whose nodes are in one-to-one correspondence with the frequent itemsets. Specifically, each node v is labelled with an item i and a support value σ_v, so that the itemset associated with v is given by the sequence of items labelling the path from the root to v, and has support σ_v. The root is associated with the empty itemset and is labelled with $(\cdot, |\mathcal{D}|)$. The children of every node are arranged right-to-left consistently with the ordering of their labelling items.

[1] When clear from the context, we will refer to frequent items or itemsets, omitting $\mathcal{D}$ and min_sup.

TID	Items
1	**A** **B** **D** E **F** **G** **H** I
2	**B** C E **L**
3	**A** **B** **D** **F** **H** **L**
4	**A** **B** C **D** **F** **G** **L**
5	**B** **G** **H** **L**
6	**A** **B** **D** **F** I

Figure 1. Sample dataset (items in bold are frequent for min_sup $= 3$)

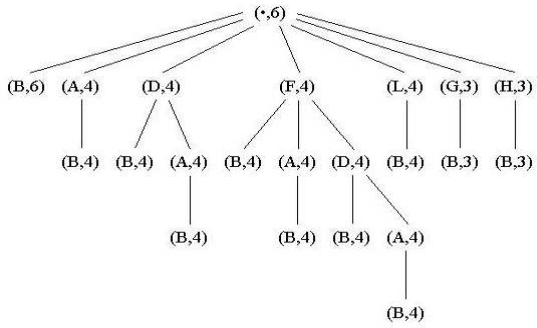


Figure 2. FIST for the sample dataset with min_sup $= 3$

A sample dataset and the corresponding FIST for min_sup $= 3$ are shown in Figures 1 and 2. Notice that a different initial ordering of the items in $\mathcal{I}'$ would produce a different FIST. Most of the algorithms that compute $\mathcal{F}(\mathcal{D}, \text{min\_sup})$ perform either a breadth-first or a depth-first exploration of some FIST. In particular, our algorithm performs a depth-first exploration of the FIST defined above.

2.1. Datasets used in the experiments

The experiments reported in this paper have been conducted on several real and artificially generated datasets, frequently used in previous works. We briefly describe them below and refer the reader to [4, 16] for more details (see also Table 1).

Pos: From Blue-Martini Software Inc., contains years worth of point-of-sale data form an electronics retailer.

WebView1, WebView2: From Blue-Martini Software Inc., contain several months of clickstream data from e-commerce web sites.

Pumsb, Pumsb*: derived by [4] from census data.

Mushroom: It contains characteristics of various species of mushrooms.

Connect-4, Chess: are relative to the respective games.

```
1. Determine I' and D';
2. Create IL and link it to D';
   X ← ∅; h ← 0; ℓ ← 0;
   while (ℓ < |IL|) do
3.    if (IL[ℓ].count < min_sup) then ℓ ← ℓ + 1;
      else
4.       if ((h > 0) AND (IL[ℓ].item = X[h − 1]))
5.       then ℓ ← ℓ + 1; h ← h − 1;
         else
6.          X[h] ← IL[ℓ].item;
7.          h ← h + 1;
8.          Generate itemset X;
9.          for i ← ℓ − 1 downto 0 do
               make IL[i].ptr point to head of t-list(i, D'_X);
               IL[i].count ← support of IL[i].item in D'_X;
            ℓ ← 0;
```

Figure 3. Main Strategy

IBM-Artificial: a class of artificial datasets obtained using the generator developed in [3]. A dataset in this class is denoted through the parameters used by the generator, namely as Dx.Ty.Iw.Lu.Nz, where x is the number of transactions, y the average transaction size, w the average size of maximal potentially large itemsets, u the number of maximal potentially large itemsets, and z the number of items.

Datasets from Blue-Martini Software Inc. and (usually) the artificial ones are regarded as sparse, while the other ones as dense.

3. The main strategy

The main strategy adopted by PatriciaMine is described by the pseudocode in Figure 3 and is based on a depth-first exploration of the FIST, similar to the one employed by the algorithms in [7, 12, 14, 9]. However, it must be remarked that while previous algorithms were expressed in a recursive fashion, PatriciaMine follows an iterative exploration strategy, which avoids the burden of managing recursion.

A first scan of the dataset $\mathcal{D}$ is performed to determine the set $\mathcal{I}'$ of frequent items, and a pruned instance $\mathcal{D}'$ of the original dataset where non-frequent items and empty transactions are removed (Step 1). Then, an *Item List* (IL) vector is created (Step 2), where each entry IL$[\ell]$ consists of three fields: IL$[\ell]$.item, IL$[\ell]$.count, and IL$[\ell]$.ptr, which store, respectively, a distinct item of $\mathcal{I}'$, its support and a pointer. The entries are sorted by decreasing value of the support field, hence the most frequent items are positioned to the top of the IL. The IL is linked to $\mathcal{D}'$ as follows. For each entry IL$[\ell]$, the pointer IL$[\ell]$.ptr points to a list that threads together all occurrences of IL$[\ell]$.item in $\mathcal{D}'$. We call such a list the *threaded list* for IL$[\ell]$.item with respect to $\mathcal{D}'$, and denote it by t-list$(\ell, \mathcal{D}')$. The initial IL for the sample dataset and the t-lists built on a natural representation of the dataset, are shown in Figure 4. (The actual data structure used to represent $\mathcal{D}'$ will be discussed in the next section.)

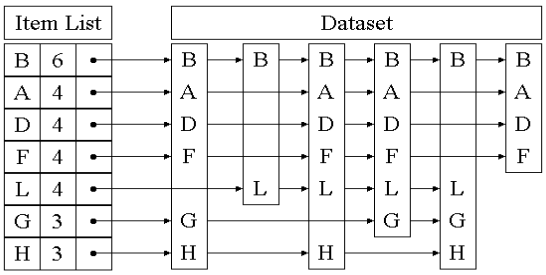


Figure 4. Initial IL and t-lists for the sample dataset

Then, a depth-first exploration of the FIST is started visiting the children of each node by decreasing support order (i.e., left-to-right with respect to Figure 2). This exploration is performed by the while-loop in the pseudocode. A vector X and an integer h are used to store, respectively, the itemset associated with the last visited node of the FIST and its length (initially, X is empty and $h = 0$, meaning that the root has just been visited).

Let us consider the beginning of a generic iteration of the while-loop and let v be the last visited node of the FIST, associated with itemset $X = (a_1, a_2, \ldots, a_h)$, where a_h is the item labelling v, and, for $j < h$, a_j is the item labelling the ancestor w_j of v at distance $h-j$ from it. For $1 \leq j \leq h$, let ℓ_j be the IL index such that IL$[\ell_j]$.item $= a_j$, and note that $\ell_h < \ell_{h-1} < \cdots < \ell_1$; also denote by X_j the prefix $(a_1, a_2, \ldots, a_j)$ of X, which is the itemset associated with w_j (clearly, $X = X_h$).

The following invariant holds at the beginning of the iteration. Let ℓ' be an arbitrary index of the IL, and suppose that $\ell_{j+1} < \ell' \leq \ell_j$, for some $0 \leq j \leq h$, setting for convenience $\ell_0 = |\text{IL}| - 1$ and $\ell_{h+1} = -1$. Then, IL$[\ell']$.count stores the support of item IL$[\ell']$.item in $\mathcal{D}'_{X_j}$, and IL$[\ell']$.ptr points to t-list$(\ell', \mathcal{D}'_{X_j})$, that threads together all occurrences of IL$[\ell']$.item in $\mathcal{D}'_{X_j}$ (we let $X_0 = \emptyset$ and $\mathcal{D}'_{X_0} = \mathcal{D}'$).

During the current iteration and, possibly, a number of subsequent iterations, the node u which is either the first child of v, if any, or the first unvisited child of one of v's ancestors is identified (Steps 3÷5). If no such node is found the algorithm terminates. It is easily seen that the item labelling u is the first item IL$[\ell]$.item found scanning the IL from the top, such that IL$[\ell]$.count $\geq$ min_sup and $\ell \neq \ell_j$ for every $1 \leq j \leq h$. If node u is found, its corresponding itemset is generated (Steps 6÷8). (Note that if u is the

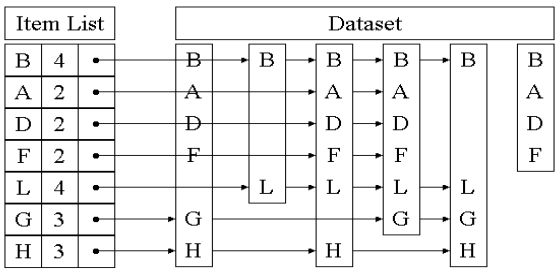


Figure 5. IL and t-lists after visiting (L,4)

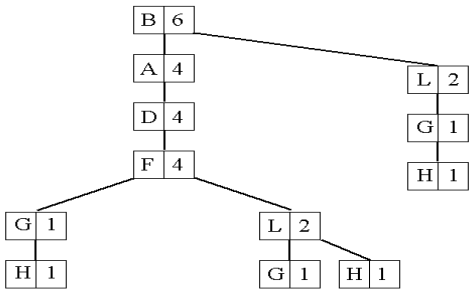


Figure 6. Standard trie for the sample dataset

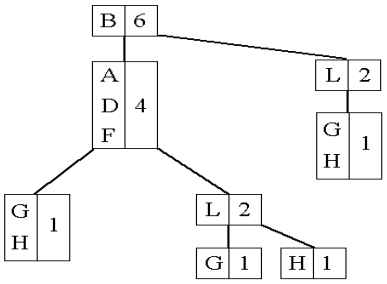


Figure 7. Patricia trie for the sample dataset

child of an ancestor w of v, we have that before Step 6 is executed $X[0 \ldots h-1]$ correctly stores the itemset associated with w.) Then, the first ℓ entries of the IL are updated so to enforce the invariant for the next iteration (for-loop of Step 9). Figure 5 shows the IL and t-lists for the sample dataset at the end of the while-loop iteration where node u=(L,4) is visited and itemset $X = (L)$ is generated. Observe that while the entries for items G and H (respectively, IL$[5]$ and IL$[6]$) are relative to the entire dataset, all other entries are relative to $\mathcal{D}'_X$.

The correctness of the whole strategy is easily established by noting that the invariant stated before holds with $h = 0$ at the beginning of the while-loop, i.e., at the end of the visit of the root of the FIST.

4. Representing the dataset as a Patricia trie

Crucial to the efficiency of the main strategy presented in the previous section is the choice of the data structure employed to represent the dataset $\mathcal{D}'$. Some previous works represented the dataset $\mathcal{D}'$ through a standard trie, called *FP-tree*, built on the set of transactions, with items sorted by *decreasing* support [7, 14]. The advantage of using the trie is substantial for dense datasets because of the compression achieved by merging common prefixes, but in the worst case, when the dataset is highly sparse, the number of nodes may be close to the size N of the original dataset (i.e., the sum of all transaction lengths). Since each node of the trie stores an item, a count value, which indicates the number of transactions sharing the prefix found along the path from the node to the root, plus other information needed for navigating the trie (e.g., pointers to the children and/or to the father), the overall space taken by the trie may turn out to be αN, where α is a constant greater than 1.

For these reasons, it has been suggested in [12, 9] that sparse datasets, for which the trie becomes space inefficient, be stored in a straightforward fashion as arrays of transactions. However, these works also encourage to switch to the trie representation during the course of execution, for portions of the dataset which are estimated to be sufficiently dense. However, an effective heuristic to decide when to switch from one structure to another is hard to find and may be costly to implement. Moreover, even if a good heuristic was found, the overhead incurred in the data movement may reduce the advantages brought by the compression gained.

To avoid the need for two alternative data structures to attain space efficiency, our algorithm resorts to a compressed trie, better known as *Patricia trie* [8]. The Patricia trie for a dataset $\mathcal{D}'$ is a modification of the standard trie: namely, each maximal chain of nodes $v_1 \to v_2 \to \cdots \to v_k$, where all v_i's have the same count value c and (except for v_k) exactly one child, is coalesced into a single node that inherits count value c, v_k's children, and stores the sequence of items previously stored in the v_i's. (A Patricia trie representation of a transaction dataset has been recently adopted by [6] in an dynamic setting where the dataset evolves with time, and on-line queries on frequencies of individual itemsets are supported.)

The standard and Patricia tries for the sample dataset are compared in Figure 6 and 7, respectively. As the figure shows, a Patricia trie may still retain some single-child nodes, however these nodes identify boundaries of transactions that are prefixes of other transactions. The following theorem provides an upper bound on the overall size of the

Patricia trie.

Theorem 1 *A dataset $\mathcal{D}'$ consisting of M transactions with aggregate size N can be represented through a Patricia trie of size at most $N + O(M)$.*

Proof. Consider the Patricia trie described before. The trie has less than $2M$ nodes since each node which has either zero or one child accounts for (one or more) distinct transactions, and, by standard properties of trees, all other nodes are at most one less than the number of leaves. The theorem follows by noting that the total number of items stored at the nodes is at most N. □

It is important to remark that even for sparse datasets, which exhibit a moderate sharing of prefixes among transactions, the total number of items stored in the trie may turn out much less than N, and if the number of transactions is $M \ll N$, as is often the case, the Patricia trie becomes very space efficient. To provide empirical evidence of this fact, Table 1 compares the space requirements of the representations based on arrays, standard trie, and Patricia trie, for the datasets introduced before, on some fixed support thresholds. For each dataset the table reports: the number of transactions, the average transaction size (AvTS), the chosen support threshold (in percentage), and the sizes in bytes of the various representations (data are relative to datasets pruned of non-frequent items). An item is assumed to fit in one word (4 bytes). For the array-based representation we considered an overhead of 1 word for each transaction, while for the standard and Patricia tries, we considered an overhead per node of 4 and 5 words, respectively, which are needed to store the count, the pointer to the father and other information used by our algorithm (the extra word in each Patricia trie node is used to store the number of items at the node).

The data reported in the table show the substantial compression achieved by the Patricia trie with respect to the standard trie, especially in the case of sparse datasets. Also, the space required by the Patricia trie is comparable to, and often much less than that of the simple array-based representation. In the few cases where the former is larger, indicated in bold in the table, the difference between the two is rather small (and can be further reduced through a more compact representation of the Patricia trie nodes). Furthermore, it must be observed that in the execution of the algorithm additional space is required to store the threaded lists connected to the IL. Initially, this space is proportional to the overall number of items appearing in the dataset representation, which is smaller for the Patricia trie due to the sharing of prefixes among transactions.

Construction of the Patricia trie Although the Patricia trie provides a space efficient data structure for representing $\mathcal{D}'$, its actual construction may be rather costly, thus influencing the overall performance of the algorithm especially if, as it will be discussed later, the dataset is projected a number of times during the course of the algorithm.

A natural construction strategy starts from an initial empty trie and inserts one transaction at a time into it. To insert a transaction t, the current trie is traversed downwards along the path that corresponds to the prefix shared by t with previously inserted transactions, suitably updating the count at each node, until either t is entirely covered, or a point in t is reached where the shared prefix ends. In the latter case, the remaining suffix is stored into a new node added as a child of the last node visited. In order to efficiently search the correct child of a node v during the downward traversal of the trie, we employ a hash table whose buckets store pointers to the children of v based on the first items they contain. (A similar idea was employed by the Apriori algorithm [3] in the hash tree.) The number of buckets in the hash table is chosen as a function of the number of children of the node, in order to strike a good trade-off between the space taken by the table and the search time. Moreover, since during the mining of the itemsets the trie is only traversed upwards, the space occupied by the hash table can be freed after the trie is build.

5. Optimizations

A number of optimizations have been introduced and tested in the implementation of the main strategy described in Section 3. In the following subsections, we will always make reference to a generic iteration of the while-loop of Figure 3 where a new frequent itemset X is generated in Step 8 after adding, in Step 6, item IL$[\ell]$.item. Also, we define as *locally frequent items* those items IL$[j]$.item, with $j < \ell$, such that their support in $\mathcal{D}'_X$ is at least min_sup.

5.1. Projection of the dataset

After frequent itemset X has been generated, the discovery of all frequent supersets $Y \supset X$ could proceed either on a physical projection of the dataset (i.e., a materialization of $\mathcal{D}'_X$) and on a new IL, both restricted to the locally frequent items, or on the original dataset $\mathcal{D}'$, with $\mathcal{D}'_X$ is identified by means of the updated t-lists in the IL (in this case, a new IL or the original one can be used).

The first approach, which was followed in FP-Growth [7], is effective if the new IL and $\mathcal{D}'_X$ shrink considerably. On the other hand, in the second approach, employed in Top-Down FP-Growth [14], no time and space overheads are incurred for building the projected datasets and maintaining in memory all of the projected datasets along a path of the FIST.

Dataset	Transactions	AvgTS	min_sup %	Array	Trie	Patricia
Chess	3,196	35.53	20	467,060	678,560	250,992
Connect-4	67,557	31.79	60	8,861,312	69,060	55,212
Mushroom	8,124	22.90	1	776,864	532,720	380,004
Pumsb	49,046	33.48	60	6,765,568	711,800	349,180
Pumsb*	49,046	37.26	20	7,506,220	5,399,120	2,177,044
T10.I4.D100k.N1k.L2k	100,000	10.10	0.002	4,440,908	14,294,760	**5,129,212**
T40.I10.D100k.N1k.L2k	100,000	39.54	0.25	16,217,064	71,134,380	**16,935,176**
T30.I16.D400k.N1k.L2k	397,487	29.30	0.5	48,175,824	163,079,980	41,023,616
POS	515,597	6.51	0.01	15,497,908	32,395,740	13,993,508
WebView1	59,601	2.48	0.054	831,156	1,110,960	618,292
WebView2	77,512	4.62	0.004	1,742,516	4,547,380	**1,998,316**

Table 1. Space requirements of array-based, standard trie, and Patricia trie representations

Ideally, one should implement a hybrid strategy allowing for physical projections only when they are beneficial. This was attempted in OpportuneProject [9] where physical projections are always performed when the dataset is represented as an array of transactions (and if sufficient memory is available), while they are inhibited when the dataset is represented through a trie, unless sufficient compression can be attained. However, in this latter case, no precise heuristic is provided to decide when physical projection must take place. In fact, the compression rate is rather hard to estimate without doing the actual projection, hence incurring high costs.

In our implementation, we experimented several heuristics for limiting the number of projections. Although no heuristic was found superior to all others in every experiment, a rather simple heuristic exhibited very good performance in most cases: namely, to allow for physical projection only at the top s levels of the FIST and when the locally frequent items are at least k (in the experiments, $s = 3$ and $k = 10$ seemed to work fairly well). The rationale behind this heuristic is that the cost of projection is justified if the mining of the projected dataset goes on for long enough to take full advantage of the compression it achieves. Moreover, the heuristic limits the memory blowup by requiring at most s projected datasets to coexist in memory. Experimental results regarding the effectiveness of the heuristic, will be presented and discussed in Section 6.1

5.2. Immediate generation of subtrees of the FIST

Suppose that at the end of the for-loop every locally frequent item IL$[j]$.item, with $j < \ell$, has support IL$[j]$.count $=$ IL$[\ell]$.count $= c$ in $\mathcal{D}'_X$. Let Z denote the set of the locally frequent items. Then, for every $Z' \subseteq Z$ we have that $X \cup Z'$ is frequent with support c. Therefore, we can immediately generate all of these itemsets and set $\ell = \ell + 1$ rather than resetting $\ell = 0$ after the for-loop.[2] Viewed on the FIST, this is equivalent to generate all nodes in the subtree rooted at the node associated with X, without actually exploring such a subtree.

A similar optimization was incorporated in previous implementations, but limited to the case when the t-list$(\ell, \mathcal{D}'_X)$, pointed by IL$[\ell]$.ptr, consists of a single node. Our condition is more general and encompasses also cases when t-list$(\ell, \mathcal{D}'_X)$ has more than one node.

5.3. Implementation of the for loop

Another important issue concerns the implementation of the for-loop (Step 9), which contributes a large fraction of the overall running time. By the invariant previously stated, we have that, before entering the for-loop, IL$[\ell]$.ptr points to head of t-list$(\ell, \mathcal{D}'_X)$, that is, it threads together all of the occurrences of IL$[\ell]$.item in nodes of the trie corresponding to transactions in $\mathcal{D}'_X$. Moreover, the algorithm must ensure that the count of each such node is relative to $\mathcal{D}'_X$ and not to the entire dataset. Let T_X denote the portion of the trie whose leaves are threaded together by t-list$(\ell, \mathcal{D}'_X)$.

The for-loop determines t-list$(j, \mathcal{D}'_X)$ for every $0 \leq j < \ell - 1$, and updates IL$[j]$.count to reflect the actual support of IL$[j]$.item in $\mathcal{D}'_X$. To do so, one could simply take each occurrence of IL$[\ell]$.item threaded by t-list$(\ell, \mathcal{D}'_X)$ and walk up the trie suitably updating the count of each node encountered, and the count and t-list of each item stored at the node. This is essentially, the strategy implemented by Top Down FP-growth [14] and OpportuneProject (under trie representation) [9]. However, it has the drawback of traversing every node $v \in T_X$ multiple times, once for each leaf in v's subtree. It is not difficult to show an example

[2] This optimization is inspired by the concept of *closed frequent itemset* [11] in the sense that only $X \cup Z$ is closed and would be generated when mining this type of itemsets.

where, with this approach, the number of node traversals is quadratic in the size of T_X.

In our implementation, we adopted an alternative strategy that, rather than traversing each individual leaf-root path in T_X, performs a global traversal from the leaves to the root guided by the entries of the IL which are being updated. In this fashion, each node in T_X is traversed only once. We refer to this strategy as the *item-guided traversal*. Specifically, the item-guided traversal starts by walking through the nodes threaded together in t-list$(\ell, \mathcal{D}'_X)$. For each such node v, the count and t-list of each item IL$[j]$.item stored in v, with $j < \ell$, are updated, and v is inserted in t-list$(j, \mathcal{D}'_X)$ marked as *visited*. Also, the count and t-list of the last item, say IL$[j']$.item, stored in v's father u are updated and u is inserted in t-list$(j', \mathcal{D}'_X)$ marked as *unvisited*. After all nodes in t-list$(\ell, \mathcal{D}'_X)$ have been dealt with, the largest index $j < \ell$ is found such that t-list$(j, \mathcal{D}'_X)$ contains some unvisited nodes (which can be conveniently positioned at the front of the list). Then, the item-guided traversal is iterated walking through the unvisited nodes in t-list$(j, \mathcal{D}'_X)$. It terminates when no threaded list is found that contains unvisited nodes (i.e., the top of the IL is reached). The following theorem is easily proved.

Theorem 2 *The item-guided traversal correctly visits all nodes in T_X. Moreover, each such node with k direct children is touched k times and fully traversed exactly once.*

6. Experimental results

This section presents the results of several experiments we performed on the datasets described in Section 2.1. Specifically, in Subsection 6.1 we assess the effectiveness of our implementation, while in Subsections 6.2 and 6.3 we compare the performance of PatriciaMine with that of other prominent algorithms. The experiments reported in the first two subsections have been conducted on an IBM RS/6000 SP multiprocessor, using a single 375Mhz POWER3-II processor, with 4GB main memory, and two 9.1 GB SCSI disks under the AIX 4.3.3 operating system. On this platform, running times as well as other relevant quantities (e.g., cache and TLB hits/misses) have been measured with hardware counters, accessed through the HPM performance monitor by [5]. Instead, since for OpportuneProject only the object code for a Windows platform was made available to us by the authors, the experiments in Subsection 6.3 have been performed on a 1.7Ghz Pentium IV PC, with 256MB RAM, and 100GB hard disk, under Windows 2000 Pro.

6.1. Effectiveness of the heuristic for conditional projection

A first set of experiments was run to verify whether allowing for physical projections of the dataset improves performance and if the heuristic we implemented to decide when to physically project the dataset is effective. The results of the experiments are reported in Figures 8 and 9 (running times do not include the output of the frequent itemsets). For each dataset, we compared the performance of PatriciaMine using the heuristic (line "WithProjection") with the performance of a version of PatriciaMine where physical projection is inhibited (line "WithoutProjection"), on four different values of support, indicated in percentage. It is seen that the heuristic yields performance improvements, often very substantial, at low support values (e.g., see Connect-4, Pumsb*, WebView1/2, T30.I16.D400k.N1k.L2k, and T40.I10.D100k.N1k.L2k) while it has often no effect or incurs a slight slowdown at higher supports. This can be explained by the fact that at high supports the FIST is shallow and the projection overhead cannot be easily hidden by the subsequent computation. Note that the case of Pos is anomalous. For this dataset the heuristic, and in fact all of the heuristics we tested, slowed down the execution, hence suggesting that physical projection is never beneficial. This case, however, will be further investigated.

We also tested the speed-up achieved by immediately generating all supersets of a certain frequent itemset X when the locally frequent items have the same support as X. In particular, we observed that the novelty introduced in our implementation, that is considering also those cases when the threading list t-list$(\ell, \mathcal{D}'_X)$ consists of more than one node, yielded a noticeable performance improvement (e.g., a factor 1.4 speed-up was achieved on WebView1 with support 0.054%, and a factor 1.6 speed-up was achieved on WebView2 with support 0.004%).

We finally compared the effectiveness of the implementation of the for-loop of Figure 3 based on the novel item-guided traversal, with respect to the straightforward one. Although the item-guided traversal is provably superior in an asymptotic worst-case sense (e.g., see Theorem 2 and the discussion in Section 5.3) , the experiments provided mixed results. For all dense datasets and for Pos, the item-guided traversal turned out faster than the straightforward one up to a factor 1.5 (e.g., for Mushroom with support 5%), while for sparse datasets it resulted actually slower by a factor at most 1.2. This can be partly explained by noting that if the tree to be traversed is skinny (as is probably the case for the sparse datasets, except for Pos) the item-guided traversal cannot provide a substantial improvement while it suffers a slight overhead for the scan of the IL. Moreover, for some sparse datasets, we observed that while the item-guided traversal performs a smaller number of instructions, it exhibits less locality (e.g., it incurs higher TLB misses) which causes the higher running time. We conjecture that a refined implementation could make the item-guided traversal competitive even for sparse datasets.

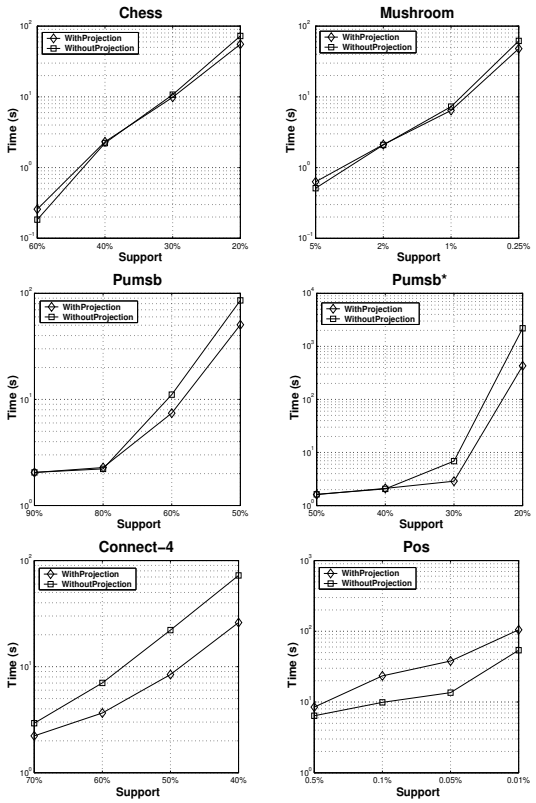


Figure 8. Comparison between PatriciaMine with and without projection on Chess, Mushroom, Pumsb, Pumsb*, Connect-4, Pos

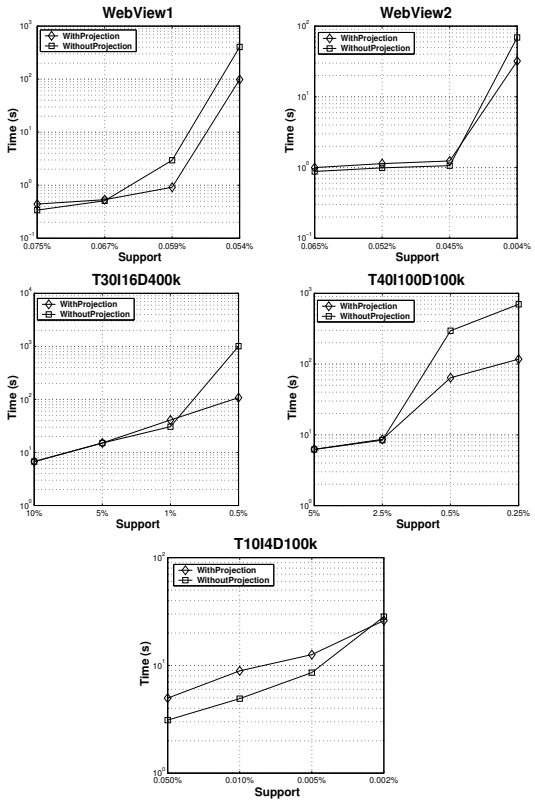


Figure 9. Comparison between PatriciaMine with and without projection on WebView1, WebView2, and some artificial datasets

6.2. Comparison with other algorithms

In this subsection, we compare PatriciaMine with other prominent algorithms whose source code was made available to us: namely FP-Growth [7], which has been mentioned before, DCI [10], and Eclat [15].

DCI (Direct Count & Intersect) performs a breadth-first exploration of the FIST, generating a set of candidate itemsets for each level, computing their support, and then determining the frequent ones. It employs two alternative representations for the dataset, a horizontal and a vertical one, and, respectively, a count-based and intersection-based method to compute the supports, switching adaptively from one to the other based on the characteristics of the dataset.

Eclat, instead is based on a depth-first exploration strategy (like FP-Growth and PatriciaMine). It employs a vertical representation of the dataset which stores with each item the list of transaction IDs (TID-list) where it occurs, and determines an itemset's support through TID-lists intersections. The counting mechanism was successively improved in dEclat [16] by using diffsets, that is, differences between TID-lists, in order to avoid managing very long TID-lists.

For FP-Growth and Eclat, we used the source code developed by Goethals[3], while for DCI we obtained the source code directly from the authors. The implementation of Eclat we employed includes the use of diffsets.

The experimental results are reported in Figures 10 and 11. For each dataset, a graph shows the running times achieved by the algorithms on four support values, indicated in percentages. (Here we included the output time since for DCI the writing on file of frequent itemsets is functional to the algorithm's operation.) It is easily seen that the performance of PatriciaMine is significantly superior to that of Eclat and FP-Growth on all datasets and supports. We also observed that Eclat features higher locality than FP-Growth, exhibiting in some cases a better running time, though performing a larger number of instructions.

Compared to DCI, PatriciaMine is consistently and often substantially faster at low values of support, while at higher supports, where execution time is in the order of a few sec-

[3] Available at `http://www.cs.helsinki.fi/u/goethals`

onds, the two algorithms exhibit similar performance and sometimes PatriciaMine is slightly slower, probably due to the trie construction overhead. However, it must be remarked that small differences between DCI and Patricia at low execution times could also be due to the different format required of the initial dataset, and different input/output functions employed by the two algorithms.

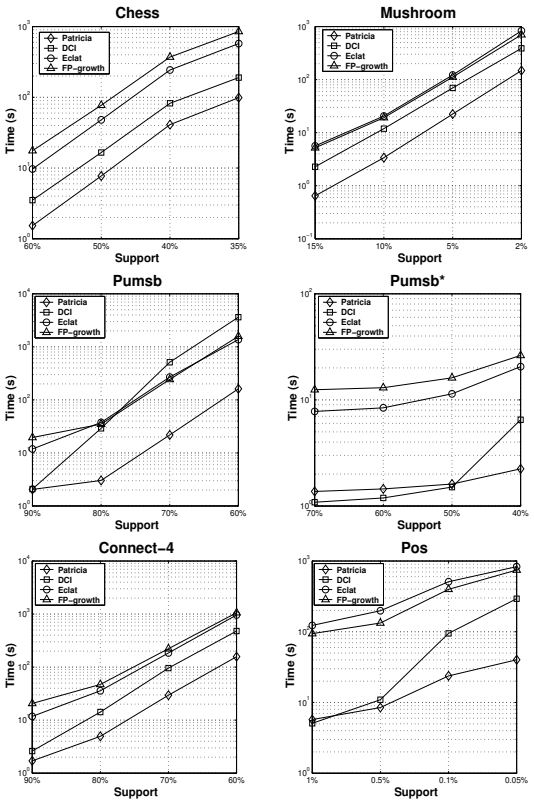


Figure 10. Comparison of PatriciaMine, DCI, Eclat and FP-Growth on Chess, Mushroom, Pumsb, Pumsb*, Connect-4, Pos

6.3. Comparison with OpportuneProject

Particularly relevant for our work is the comparison between PatriciaMine and OpportuneProject [9], which, to the best of our knowledge, represents the latest and most advanced algorithm in the family stemmed from FP-Growth. For lack of space, we postpone a detailed and critical discussion of the strengths and weaknesses of the two algorithms to the full version of the paper.

Figures 12 and 13, report the performances exhibited by PatriciaMine and OpportuneProject on the Pentium/Windows platform for a number of datasets and supports. It can be seen that, the performance of PatriciaMine is consistently superior, up to one order of magnitude (e.g., in Pumsb*). The only exception are Pos (see graph labelled "Pos with projection") and the artificial dataset T30.I16.D400k.N1k.L2k. For Pos, we have already observed that our heuristic for limiting the number of physical projections does not improve the running time. In fact, it is interesting to note that by inhibiting projections, PatriciaMine becomes faster than OpportuneProject (see graph labelled "Pos without projection"). This suggests that a better heuristic could eliminate this anomalous case.

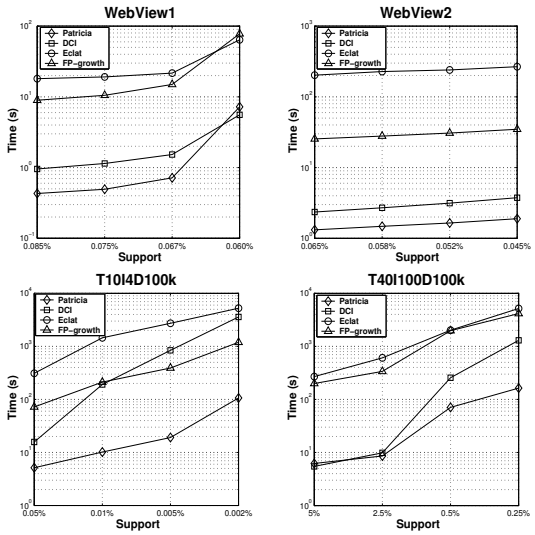


Figure 11. Comparison of PatriciaMine, DCI, Eclat and FP-Growth on WebView1, WebView2, and some artificial datasets

As for T30.I16.D400k.N1k.L2k, some measurements we performed revealed that the time taken by the initialization of the Patricia trie accounts for a significant fraction of the running time at high support thresholds, and such an initial overhead cannot be hidden by the subsequent mining activity. However, at lower support thresholds, where the computation of the frequent itemsets dominates over the trie construction, PatriciaMine becomes faster than OpportuneProject.

Finally we report that on WebView1 for absolute support 32 (about 0.054%), OpportuneProject ran out of memory while PatriciaMine successfully completed the execution.

References

[1] R. Agrawal, C. Aggarwal, and V. Prasad. A tree projection algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing*, 61(3):350–371, 2001.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc.*

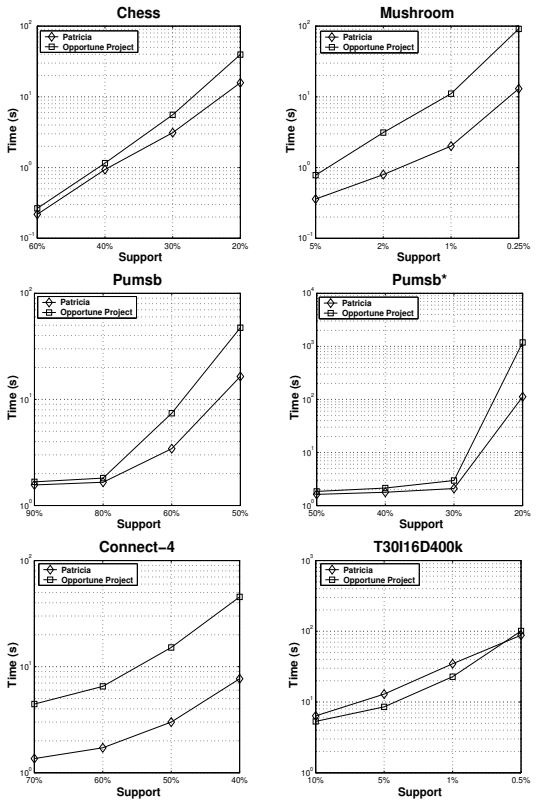


Figure 12. Comparison of PatriciaMine and OpportuneProject on Chess, Mushroom, Pumsb, Pumsb*, Connect-4, and T30I16D400k

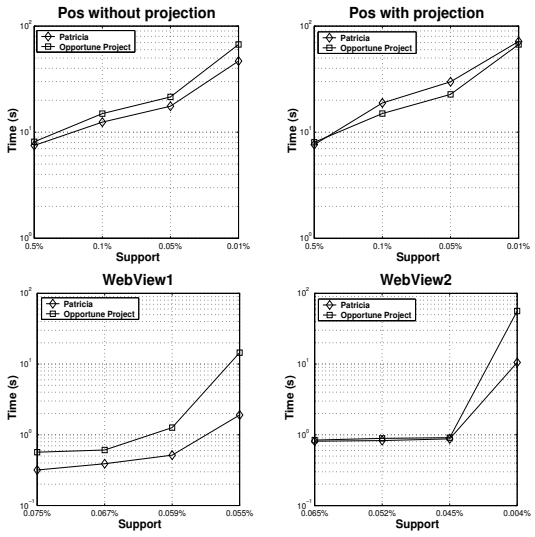


Figure 13. Comparison of PatriciaMine and OpportuneProject on Pos, WebView1, WebView2

of the ACM SIGMOD Intl. Conference on Management of Data, pages 207–216, 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. of the 20th Very Large Data Base Conference*, pages 487–499, 1994.

[4] R. Bayardo. Efficiently mining long patterns from databases. In *Proc. ot the ACM SIGMOD Intl. Conference on Management of Data*, pages 85–93, 1998.

[5] L. DeRose. Hardware Performance Monitor (HPM) toolkit. version 2.3.1. Technical report, Advanced Computer Technology Center, Nov. 2001.

[6] A. Hafez, J. Deogun, and V. Raghavan. The item-set tree: A data structure for data mining. In *Proc. of the 1st Int. Conference on Data Warehousing and Knowledge Discovery*, LNCS 1676, pages 183–192, 1999.

[7] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *Proc. of ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12, 2000.

[8] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison Wesley, Reading, MA, 1973.

[9] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *Proc. of the 8th ACM SIGKDD Intl. Conference on Knowledge Discovery and Data Mining*, pages 229–238, July 2002.

[10] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Adaptive resource-aware mining of frequent sets. In *Proc. of the IEEE Intl. Conference on Data Mining*, pages 338–345, Dec. 2002.

[11] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *Proc. of the 7th Int. Conference on Database Theory*, pages 398–416, Jan. 1999.

[12] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. H-mine: Hyper-structure mining of frequent patterns in large databases. In *Proc. of IEEE Intl. Conference on Data Mining*, pages 441–448, 2001.

[13] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21st Very Large Data Base Conference*, pages 432–444, Sept. 1995.

[14] K. Wang, L. Tang, J. Han, and J. Liu. Top down FP-Growth for association rule mining. In *Proc. of the 6th Pacific-Asia Conf. on Advances in Knowledge Discovery and Data Mining*, LNCS 2336, pages 334–340, May 2002.

[15] M. Zaki. Scalable algorithms for association mining. *IEEE Trans. on Knowledge and Data Engineering*, 12(3):372–390, May-June 2000.

[16] M. Zaki and K. Gouda. Fast vertical mining using diffsets. In *Proc. of the 9th ACM SIGKDD Intl. Conference on Knowledge Discovery and Data Mining*, Aug. 2003.