Journal of Big Data

RESEARCH Open Access

A scalable association rule learning heuristic for large datasets

Haosong Li and Phillip C.-Y. Sheu*

*Correspondence:
psheu@uci.edu
Department of Electrical Engineering and Computer Science, University of California, Irvine, USA

Abstract

Many algorithms have proposed to solve the association rule learning problem. However, most of these algorithms suffer from the problem of scalability either because of tremendous time complexity or memory usage, especially when the dataset is large and the minimum support (*minsup*) is set to a lower number. This paper introduces a heuristic approach based on divide-and-conquer which may exponentially reduce both the time complexity and memory usage to obtain approximate results that are close to the accurate results. It is shown from comparative experiments that the proposed heuristic approach can achieve significant speedup over existing algorithms.

Keywords: Association rule learning, Frequent itemset mining, Scalability, Graph partitioning, Apriori algorithm, FP-Growth algorithm

Introduction

The association rule learning problem has played a significant role in data mining for the past few decades. Association rules are widely used in many fields, including market basket analysis [1] and bioinformatics [2]. However, the problem has an NP-hard nature, meaning it is challenging to find the results within a reasonable period of time.

The invention of the Apriori Algorithm [3] made this problem computationally feasible for most computers on regular-sized datasets. Since then, researchers have continued to develop more scalable algorithms. Among others, FP-Growth [4] and Eclat [5] are two algorithms developed that improve the scalability of the Apriori algorithm.

The increasing popularity of the Internet in recent decades has made big data available to many research institutions and companies. Their sizes are so large that traditional algorithms may not be able to handle them efficiently. We consider "big data" to be datasets that, at least, are too large to fit into the memory and take a long time (hours or even days) for traditional algorithms to process. The term big data is thus relevant to the machine. A dataset considered to be big data on a PC may be a small dataset on a powerful high-performance computer (or computer cluster). This imposes a challenge to the association rule learning problem as well. Most of the previously designed algorithms, including the Apriori algorithm, the FP-Growth algorithm, and the Eclat algorithm, suffer from the problem of scalability for big data. Still, these algorithms take an unacceptable amount of time to terminate (will be discussed in "Experiments and results" section). In addition, the FP-Tree of the FP-Growth algorithm, and the TID list of the Eclat algorithm may not fit in the memory.

This paper introduces an approach that makes it possible to mine association rules and frequent itemsets for large datasets. The approach, called the Scalable Association Rule Learning (SARL) heuristic, follows the divide-and-conquer paradigm and it vertically divides a dataset into almost equivalent partitions using a graph representation and the k-way graph partitioning algorithm [6]. The total time complexity of the SARL heuristic, including the overhead of partitioning a dataset, is up to 2^d faster than that of the Apriori algorithm, where d is the number of unique items in the dataset. The memory usage is also lower than those of the current algorithms. Because of the speedup, our heuristic may be applied to real-time data analysis that can benefit many scientific [7] and military applications [8, 9].

The rest of the paper is organized as follows. In "Related work" section, we survey existing association rule learning algorithms and graph partitioning algorithms. In "Our solution" section, we present the SARL heuristic with examples, formal descriptions, theorems, and proofs. The experiments and results are presented in "Experiments and results" section, followed by conclusions and future work.

Contributions

The contributions of this paper include the provision of association graphs that represent an efficient estimation of potential frequent itemsets and the use of the MLkP algorithm to divide the items into partitions while minimizing the loss of information.

The novelty of this paper lies in two parts of our solutions (discussed later). Firstly, we propose the verticle(item-wise) partition of datasets while most divide and conquer algorithms focus on horizontal (transaction-wise) divide and conquer methods. Secondly, the transformation of frequent two-itemsets into graph representation while applying the efficient MLkP algorithm is a novel and efficient approach in solving the association rule learning problem.

Related work

Association rule learning/frequent itemset mining has been an active research area. Among others, three approaches are considered the most popular and possibly the most efficient: the Apriori algorithm, the FP-Growth algorithm, and the Eclat algorithm.

The Apriori algorithm

The Apriori algorithm [3], introduced by Agrawal and Srikant, was the first efficient association rule learning algorithm. It incorporates various techniques to speed up the process as well as to reduce the use of memory. For example, the $L_{k-1} \times L_{k-1}$ method used in the candidate generation process can reduce the number of candidates generated, and the pruning process can significantly reduce the number of possible candidates at each level.

One of the most important mechanisms in the Apriori algorithm is the use of the hash tree data structure. It uses this data structure in the candidate support counting phase to reduce the time complexity from O(kmn) to O($kmT + n$), where k is the average size of the candidate itemset, m represents the number of candidates, n represents the number of items in the whole dataset, and T is the number of transactions.

The major advantage of the Apriori algorithm comes from its memory usage because only the $k - 1$ frequent itemsets, L_{k-1}, and the candidates in level k, C_k, need to be stored in the memory. It generates the minimum number of candidates based on the $L_{k-1} \times L_{k-1}$ (described in [3]) and the pruning method, and it stores them in the compact hash tree structure. In case the candidates fill up the memory from the dataset and a low *minsup* setting, the Apriori algorithm does not generate all the candidates to overload the memory. Instead, it generates as many candidates as the memory can hold.

The FP-growth algorithm

The Frequent Pattern Growth algorithm was proposed by Han et al. in 2000 [4]. It uses a tree-like structure (called Frequent Pattern Tree) instead of the candidate generation method used in the Apriori algorithm to find the frequent itemsets. The candidate generation method finds the candidates of the frequent itemsets before reducing them to the actual frequent itemsets through support counting.

The algorithm first scans a dataset and finds the frequent one itemsets. Then, a frequent pattern tree is constructed by scanning the dataset again. The items are added to the tree in the order of their support. Once the tree is completed, the tree is traversed from the bottom, and a conditional FP-Tree is generated. Finally, the algorithm generates the frequent itemsets from the conditional FP-Tree.

The FP-Growth algorithm is more scalable than the Apriori algorithm in most cases since it makes fewer passes and does not require candidate generation. However, it suffers from memory limitations since the FP-Tree is fairly complex and may not fit in the memory. Traversing the complexed FP-Tree may also be time-expensive if the tree is not compact enough.

The Eclat algorithm

Different from the Apriori algorithm and the FP-Growth algorithm that work on horizontal datasets (e.g., T001: {1, 3} T002:{1, 4}), the Eclat (Equivalence Class Clustering and bottom-up Lattice Traversal) algorithm [5] uses a vertical dataset (e.g. Item1: {T001, T002}, Item3: {T001}, Item4:{T002}). The Eclat algorithm only scans the dataset once. It finds the frequent itemsets by taking the intersections of the transaction sets.

The Eclat algorithm takes advantage of scanning the dataset only once. However, when the dataset is large, and the *minsup* is set to a low value, the TID associated with each itemset may become very long. In fact, the results can be larger than the original dataset; therefore, they may not fit into the memory.

Other association rule learning algorithms

There are three categories of association rule mining/frequent itemset mining algorithms [10]: Apriori-based algorithms, tree-based algorithms, and pattern growth algorithms. The Apriori algorithm, the Eclat algorithm, and the FP-Growth algorithm are the most popular algorithms for the three categories, respectively.

In the Apriori-based algorithm category, proposed by Agrawal and Srikant in [3] the AprioriTID algorithm is similar to Apriori, except that it generates C_k-bar and it mines the frequent itemsets from there instead of the dataset. The Apriori Hybrid algorithm [3] is a combination of the Apriori algorithm and the AprioriTID algorithm. The DHP (direct hashing and pruning) algorithm [11] uses a hash function to distribute the itemsets into buckets. If a bucket has the support lower than the *minsup*, then the bucket is discarded. The MR-Apriori [12] and HP-Apriori [13] algorithms are distributed versions of the Apriori algorithm. The MR-Apriori uses the MapReduce model on the Hadoop platform. They enable parallel execution of the Apriori algorithm.

The tree-based algorithms, represented by the Eclat algorithm, find the frequent itemset by constructing a lexicographic tree. The AIS algorithm [6] and the SETM algorithm [14] are the two earliest association rule mining algorithms in this category. Reference [3] shows that the Apriori algorithm beats them in running time. The Tree-Projection algorithm [15] counts the supports of the frequent itemsets and uses the nodes of a lexicographic tree as the representation of these support numbers. The TM algorithm [16] maps the TID of each transaction to transaction intervals before performing intersections between these intervals.

Lastly, the algorithms in the pattern growth category focus on frequent patterns. The P-Mine algorithm [17] is a parallel computing algorithm that utilizes the VLDBMine data structure to store the dataset and speed up the distribution of data, while the LP-Growth algorithm [18] makes use of an array-based linear prefix tree to improve the memory efficiency. The Can-Mining algorithm [19] finds the frequent itemsets from a canonical-order tree, which speeds up the tree traversal process when the number of frequent itemsets is low. Finally, the EXTRACT algorithm [20] uses the theory of Galois lattice to derive association rules.

The algorithms discussed above, unfortunately, have scalability problems. The Apriori-based algorithms, represented by the Apriori algorithm, have to go through the expensive candidate generation and support counting process. This causes a disadvantage in running time. The tree-based and the pattern-growth type algorithms often suffer from excessive usage of memory. For example, the FP-Growth algorithm could build a complex FP-Tree which does not fit into the memory.

We show the scalability problems of the Apriori algorithm and the FP-Growth algorithm in the experiment part of this paper. Both of the algorithms take too long to finish for most of the tested datasets. The need for faster, frequent itemset mining is urgent due to the vastly available data today. Companies and institutions have allocated many resources in data mining, and they need a time-saving, resource-saving solution. In addition, real-time data analysis plays an important role in government [21], scientific [7], and military [8, 9] applications. The experiments part of this paper shows that the current algorithms represented by the Apriori algorithm and the FP-Growth algorithm are not fast enough to complete real-time data analysis. The scalability problems of most existing association rule mining algorithms have also been addressed in [22] that is focused on paralleled computing of association rules whereas this paper presents a scalable algorithm that is suitable for a single machine also.

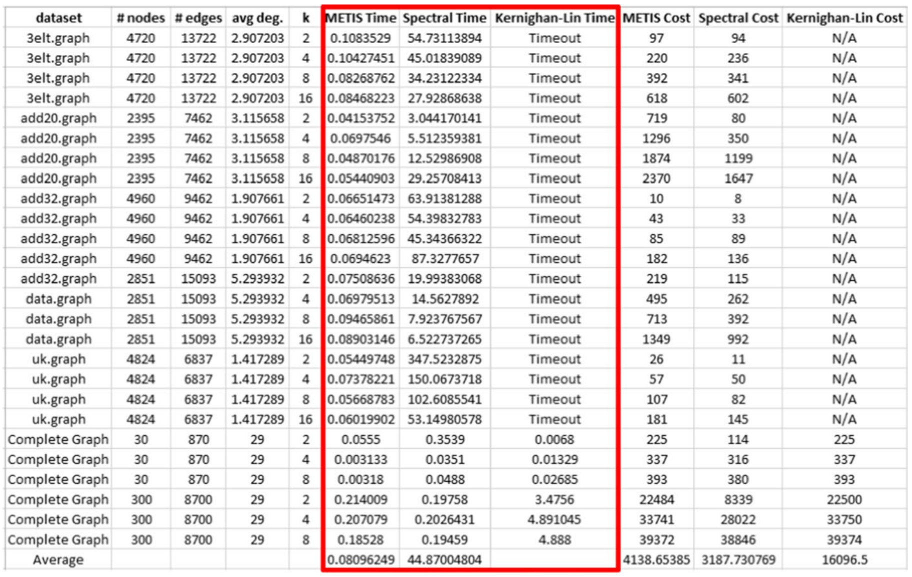
Graph partitioning algorithms

One of the key steps in the SARL heuristic that we will introduce shortly is to partition the IAG (item association graph, "Our solution" section, part 7) into *k* balanced partitions. An efficient graph partitioning algorithm is crucial since the balanced graph partitioning problem is NP-complete [23]. We have implemented three algorithms and compared them for the partitioning costs and running times. They are the recursive version of the Kernighan-Lin Algorithm [24], the Multilevel k-way Partitioning Algorithm (MLkP) [25], and the recursive version of the Spectral Partitioning Algorithm [26]. Other graph partitioning algorithms include the Tabu search-based MAGP algorithm [27] and the flow-based KaFFPa algorithm [28].

The Kernighan-Lin algorithm swaps the nodes assigned to both partitions and finds the largest decrease in the total cut size. The Multilevel k-way Partitioning algorithm (MLkP) uses coarsening-partitioning-uncoarsening/refining steps to shrink a graph into a much smaller graph. After partitioning, the graph is rebuilt to restore the original graph. A single global priority queue is used for all types of moves. The Spectral Partitioning Algorithm finds splitting of the values such that the vertices in a graph can be partitioned with respect to the evaluation of the Fiedler vector.

Experiments are conducted by us to compare the three algorithms. The datasets provided by Christopher Walshaw at the University of Greenwich [29] are used. The datasets are as large as possible while the partitioning algorithms can finish in a

Table 1 Results of the experiment that compare MLkP, Kernighan-Lin, and Spectral Partitioning algorithms



dataset	# nodes	# edges	avg deg.	k	METIS Time	Spectral Time	Kernighan-Lin Time	METIS Cost	Spectral Cost	Kernighan-Lin Cost
3elt.graph	4720	13722	2.907203	2	0.1083529	54.73113894	Timeout	97	94	N/A
3elt.graph	4720	13722	2.907203	4	0.10427451	45.01839089	Timeout	220	236	N/A
3elt.graph	4720	13722	2.907203	8	0.08268762	34.23122334	Timeout	392	341	N/A
3elt.graph	4720	13722	2.907203	16	0.08468223	27.92868638	Timeout	618	602	N/A
add20.graph	2395	7462	3.115658	2	0.04153752	3.044170141	Timeout	719	80	N/A
add20.graph	2395	7462	3.115658	4	0.0697546	5.512359381	Timeout	1296	350	N/A
add20.graph	2395	7462	3.115658	8	0.04870176	12.52986908	Timeout	1874	1199	N/A
add20.graph	2395	7462	3.115658	16	0.05440903	29.25708413	Timeout	2370	1647	N/A
add32.graph	4960	9462	1.907661	2	0.06651473	63.91381288	Timeout	10	8	N/A
add32.graph	4960	9462	1.907661	4	0.06460238	54.39832783	Timeout	43	33	N/A
add32.graph	4960	9462	1.907661	8	0.06812596	45.34366322	Timeout	85	89	N/A
add32.graph	4960	9462	1.907661	16	0.0694623	87.3277657	Timeout	182	136	N/A
add32.graph	2851	15093	5.293932	2	0.07508636	19.99383068	Timeout	219	115	N/A
data.graph	2851	15093	5.293932	4	0.06979513	14.5627892	Timeout	495	262	N/A
data.graph	2851	15093	5.293932	8	0.09465861	7.923767567	Timeout	713	392	N/A
data.graph	2851	15093	5.293932	16	0.08903146	6.522737265	Timeout	1349	992	N/A
uk.graph	4824	6837	1.417289	2	0.05449748	347.5232875	Timeout	26	11	N/A
uk.graph	4824	6837	1.417289	4	0.07378221	150.0673718	Timeout	57	50	N/A
uk.graph	4824	6837	1.417289	8	0.05668783	102.6085541	Timeout	107	82	N/A
uk.graph	4824	6837	1.417289	16	0.06019902	53.14980578	Timeout	181	145	N/A
Complete Graph	30	870	29	2	0.0555	0.3539	0.0068	225	114	225
Complete Graph	30	870	29	4	0.003133	0.0351	0.01329	337	316	337
Complete Graph	30	870	29	8	0.00318	0.0488	0.02685	393	380	393
Complete Graph	300	8700	29	2	0.214009	0.19758	3.4756	22484	8339	22500
Complete Graph	300	8700	29	4	0.207079	0.2026431	4.891045	33741	28022	33750
Complete Graph	300	8700	29	8	0.18528	0.19459	4.888	39372	38846	39374
Average					0.08096249	44.87004804		4138.65385	3187.730769	16096.5

reasonable time on the tested machine. We also run experiments on complete graphs with 30 and 300 nodes. Each dataset is tested four rounds with the number of partitions (k) being 2, 4, 8, and 16.

As shown in Table 1, the running times are highlighted in the red box. We can tell from average running time(the last row) that the MLkP algorithm has the highest speed in general. It is 560 times faster than the spectral partitioning algorithm and even faster than the recursive Kernighan-Lin algorithm. The spectral partitioning algorithm has, in general, the best partition quality. It is 1.3 times better than MLkP and much better than the recursive Kernighan-Lin algorithm. The recursive Kernighan-Lin algorithm takes too long to complete all five datasets. It also shows serious scalability issues for complete graphs.

Considering the MLkP algorithm has the best overall performance, we choose to use this algorithm for graph partitioning in our algorithm.

Our solution

Definitions

Below are some definitions that we will use in our algorithm:

1. K-itemset: an itemset with k items
2. Support: the occurrence of an item in the dataset
3. Minsup: the minimum requirement of support. The user usually provides this. Itemsets with support < *minsup* are eliminated.
4. Confidence: the indication of robustness of a rule in terms of percentage.

$$\text{Confidence}(\text{X} \rightarrow \text{Y}) = \text{support}(X \cup Y)\big/\text{support}(\text{X})$$

5. Minconf: the minimum requirement of confidence. The user usually provides this. Rules with confidence < minconf are eliminated.
6. Item-Association Graph: a graph structure that stores the frequent associations between pairs of items.
7. Balanced K-way Graph Partitioning Problem: Divide the nodes of a graph into k parts such that each part has almost the same number of nodes while minimizing the number of edges/sum of edge weights cut off.

A scalable heuristic algorithm—SARL-heuristic

The following is an outline of our scalable heuristic:

Step 1: Find frequent one and two itemsets using the Apriori algorithm (when *minsup* is high) or the direct generation method (when *minsup* is low).
Step 2: Construct the item association graph (IAG) from the result of step 1.
Step 3: Partition the IAG using the multilevel k-way partitioning algorithm (MLkP).
Step 4: Partition the dataset according to the result of step 3.

Table 2 Example dataset 1

TID	Items
T000	1, 2
T001	1, 2, 3
T002	4, 5
T003	1, 4, 5
T004	2, 3
T005	1, 2, 3
T006	1, 4, 5

Table 3 Frequent one itemsets

Frequent itemsets	Support
{1}	5
{2}	4
{3}	3
{4}	3
{5}	3

Table 4 Frequent two itemsets

Frequent itemsets	Support
{1, 2}	3
{1, 3}	2
{1, 4}	2
{1, 5}	2
{2, 3}	3
{4, 5}	2

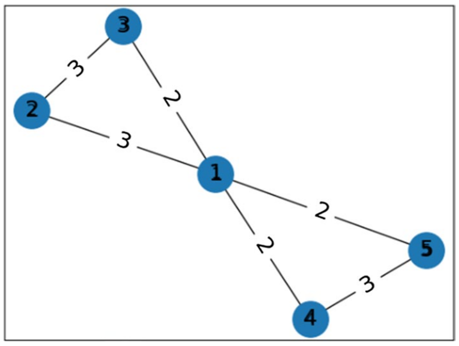


Fig. 1 An item association graph

Step 5: Call the modified Apriori algorithm or the FP-Growth algorithm to mine frequent itemsets on each transaction partition.
Step 6: Find the union of the results found from each partition.
Step 7: Generate association rules by running the Apriori-ap-genrules on the frequent itemsets found from step 6.

An example

Suppose the dataset shown in Table 2 is given and *minsup* is set to 0.1 (or 10%, or $7 * 0.1 \approx 1$ occurrence), and *minconf* is set to 0.7 (or 70%):

First, we use the Apriori algorithm to find the frequent two itemsets. As an intermediate step, the Apriori algorithm finds the frequent one-itemset first (shown in Table 3):

The frequent two-itemsets are found afterward (shown in Table 4):

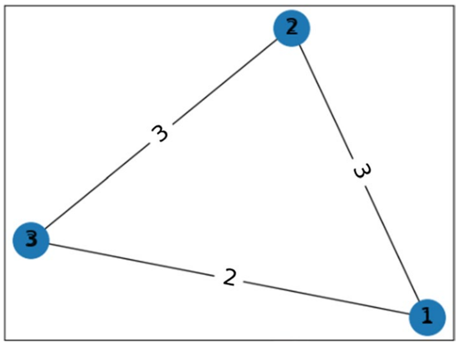


Fig. 2 Item association graph partition 1

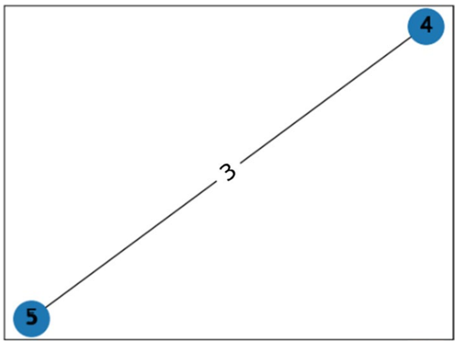


Fig. 3 Item association graph partition 2

Table 5 Transaction partition 1

TID	Items
T001	1, 2, 3
T005	1, 2, 3

Table 6 Transaction partition 2

TID	Items
None	None

Table 7 Frequent itemsets from transaction partition 1

Frequent itemsets	Support
{1, 2, 3}	2

Table 8 Frequent itemsets from transaction partition 2

Frequent itemsets	Support
None	N/A

Table 9 Frequent itemset final results

Frequent itemsets	Support
{1}	5
{2}	4
{3}	3
{4}	3
{5}	3
{1, 2}	3
{1, 3}	2
{1, 4}	2
{1, 5}	2
{2, 3}	3
{4, 5}	3
{1, 2, 3}	2

Next, we transform the above frequent two-itemsets into an item association graph (IAG), shown in Fig. 1:

To construct the graph, we first take the itemset {1, 2} with support 3. For this, we create node 1 and node 2 corresponding to the two items in the itemset. The edge between node 1 and node 2 has weight 3, representing the support of the itemset. The process is repeated for every frequent two-itemset found in the previous step.

Next, we use the multilevel k-way partitioning algorithm (MLkP) to partition the IAG. In this case, the number of nodes is small, so we only bisect the graph by setting k = 2. The result is shown in Figs. 2 and 3.

The MLkP algorithm divides the IAG into two equal or almost equal sets in linear time while the sum of the weights of edges that are cut off is the minimum.

Table 10 Association rules generated

Rules	Confidence
{2} {1}	0.75
{3} {2}	1
{5} {1}	1
{2} {3}	0.75
{5} {4}	1
{4} {5}	1
{1, 3} {2}	1

Next, we partition the dataset according to the partitions of the IAG, as shown in Tables 5 and 6. Each transaction partition has all the items from the corresponding IAG partition. However, since the algorithm has already found all frequent one and two itemsets, a transaction is not added to a transaction partition if the transaction has less than three items. For example, T000: {1, 2} is not added to the transaction partition 1, since it only has two items. Some items in the original dataset may not appear in any of the transaction partitions, because the infrequent one/two-itemsets are dropped in the IAG. This simplifies the subsequent computations. In this example, however, all the items are kept in the IAG because the IAG is a relatively dense graph. Tables 5 and 6 show the transaction partitions:

The next step is to pick the best algorithm and use it to find the frequent k-itemsets with k > 2. For this example, we choose the modified Apriori algorithm because it is faster for mining small datasets as it avoids the process of finding the one and two-itemsets again. The results from partition 1 are shown in Table 7:

Since the modified Apriori algorithm starts with three-itemsets, there are no additional frequent itemsets in the first partition. Table 8 shows the results found in transaction partition 2:

The final results (shown in Table 9) of frequent itemsets are simply the union of Tables 3, 4, 7, and 8:

After running the Apriori-ap-genrules algorithm, the association rules can be found in Table 10.

All frequent itemsets generated by the SARL heuristic are sound, meaning each frequent itemset generated indeed is correct, and the support number is accurate. However, it is possible that some frequent itemsets cannot be found by the SARL heuristic, as will be discussed shortly. In this example, the SARL heuristic loses one frequent itemset {1, 4, 5} and two related rules generated from {1, 4, 5}.

Formal description of the SARL heuristic

SARL(dataset, minsup, minconf, k, threshold):

```
SARL(dataset, minsup, minconf, k, threshold):
# if dataset is smaller than the threshold(explained later), then use
direct_gen algorithm to get frequent one and two itemsets
if size(dataset) <= threshold:
        results, two_itemsets = direct_gen(dataset)

# otherwise, use modified Apriori
else:
        results, two_itemsets = mod1-Apriori(dataset)

# build IAG from two itemsets
graph = build_IAG(two_itemsets)

# partition IAG with METIS that implements MLkP algorithm
partitions = METIS.partition(k, graph)

# partition dataset into smaller parts
parts = partition-dataset(partitions)

# repartition if some partitions cannot fit into the memory
while partition size > memory size:
        k+=1
        partitions = METIS.partition(k, graph)
        parts = partition-dataset(partitions)

# compute 3+ frequent itemsets
for part in parts:
        # when part is small, use Apriori
        if size(part) < threshold/k:
                results += mod2-Apriori(part, minsup,
two_itemsets)
        # when part is large, use FP-Growth(defined in [13])
        else:
                results += FP-Growth(part, minsup)

# generate rules with Apriori-gen(defined in [1])
rules = Apriori-gen(results, minconf)

direct_gen(dataset, minsup):
# count the support of all one itemsets and store in C1
C1 = {}
for transaction in dataset:
        for item in transaction:
                if item not in C1:
                        add item to C1
                        item.counter = 1
                else:
                        item.counter += 1

# eliminate those itemsets with support < minsup and store in L1
L1 = {}
for candidate in C1:
        if candidate.counter >= minsup:
                add candidate to L1

# compute support of all two itemsets
C2 = {}
for trans in dataset:
```

```
          # for each two-item pair in a transaction
          for comb in combinations(trans, 2):
                    if comb not in C2:
                              add comb to C2
                              comb.counter = 1
                    else:
                              comb.counter += 1

# eliminate those two itemsets with support < minsup
L2 = {}
for candidate in C2:
          if candidate.counter >= minsup:
                    add candidate to L1


mod1-Apriori(dataset, minsup):
# count the support of all one itemsets and store in C1
C1 = {}
for transaction in dataset:
          for item in transaction:
                    if item not in C1:
                              add item to C1
                              item.counter = 1
                    else:
                              item.counter += 1
# eliminate those itemsets with support < minsup and store in L1
L1 = {}
for candidate in C1:
          if candidate.counter >= minsup:
                    add candidate to L1

# generate two-itemset candidates from L1 and store in C2
C2 = {}
for itemset1 in L1:
          for itemset2 in L1:
                    if itemset1 != itemset2:
                              add itemset1 U itemset2 to C2
for transaction in dateset:
          for candidate in C2:
                    if candidate.issubset(transaction):
                              candidate.counter += 1

# eliminate those two-itemsets with support < minsup and store in L2
L2 = {}
for candidate in C2:
          if candidate.counter >= minsup:
                    add candidate to L2
return L1, L2

build_IAG(itemsets):
for itemset in itemsets:
          graph.add_node(itemset[0])
    graph.add_node(itemset[0])
    graph.add_edge(itemset[0], itemset[1], weight += 1)
return graph

partition-dataset(partitions, dataset):
for transaction in dataset:
          for partition in partitions:
                    intersect = parition intersect transaction
                    if len(inersect) > 2:
            add intersect to dataset_partition_i
return transaction partition names
```

```
mod2-Apriori(dataset, minsup, two_itemsets):
# generate frequent three-itemsets based on two_itemsets
results = []
Lk = {}
Ck = apriori-gen(two_itemsets) # apriori-gen is defined in [1]
for transaction in dataset:
        Ct = subset(Ck, t)
        for c in Ct:
                c.count += 1
for c in Ck:
        if c.count >= minsup:
                Lk.add(c)
result.append(Lk)

# generate frequent 3+ itemsets
while Lk-1 != {}:
        Lk = {}
        Ck = apriori-gen(two_itemsets)
        for transaction in dataset:
                Ct = subset(Ck, t)
                for c in Ct:
                        c.count += 1
        for c in Ck:
                if c.count >= minsup:
                        Lk.add(c)
        result.append(Lk)
return results
```

Finding frequent 2 itemsets using the Apriori algorithm or dirct_gen algorithm

The first step of the SARL heuristic is to find the frequent 2 itemsets efficiently.

Although the Apriori algorithm has scalability issues for very large datasets, it provides a fast and convenient feature to extract intermediate results and a tolerable speed for the first two passes.

The Apriori algorithm finds frequent itemset L_k for each k, and each L_k is stored separately. We run the Apriori algorithm until it finds L_2, the frequent two-itemset. It first tries to find the frequent one itemsets by traversing the dataset and count the occurrence of each unique item. If the number of occurrences of an item is less than the *minsup* provided by the user, that item is eliminated from the list of frequent one-itemset. The frequent two itemsets are discovered based on the frequent one itemsets. The algorithm generates C_2, the candidate sets for the frequent two itemsets, using $L_{k-1} \times L_{k-1}$:

$$insertintoC_k$$

$$selectp.item_1, p.item_2, \ldots, p.item_{k-1}, q.item_{k-1}$$

$$fromL_{k-1}p, L_{k-1}q$$

$$wherep.item_1 = q.item_1, \ldots, p.item_{k-2} = q.item_{k-2}, p.item_{k-1} < q.item_{k-1};$$

This method generates a minimum number of candidates from the frequent one itemsets so that we can have fewer candidates to consider in the support counting phase. The Apriori algorithm also predicts and eliminates some infrequent itemsets before support counting by implementing the Apriori principle in the pruning step. If an item in C2 is not in L1, which means that the item is infrequent, so all the two itemsets that include this item are dropped. We modify the Apriori algorithm, so it terminates after L2 is found.

Another method to find frequent one and two itemsets are through direct counting and generation. The algorithm to find frequent one itemsets is the same as the Apriori algorithm. To find frequent two itemsets, we can simply find all two-item pairs in each transaction and count the occurrence of them. The advantage of this algorithm is that it does not require candidate generation from L1, and avoids much unnecessary membership testing during support counting. However, this method is not efficient on large datasets since it does not use pruning and saves all two itemsets.

In the SARL heuristic, we ask the user for a threshold of the dataset size. If the dataset is larger than the threshold, the SARL heuristic will use the modified Apriori algorithm. Otherwise, it will use the direct_gen algorithm to compute the frequent one and two itemsets.

Construction of the item association graph

The item association graph G is constructed based on the two itemsets generated by the Apriori algorithm. G is an undirected, weighted graph. A node Vi is created for each unique item i in the two itemsets T with the maximum item number being n.

$$\{V\} = \left\{ \bigcup_{i=0}^{n} V_i | i \in |T| \right\} \tag{1}$$

The edges E in graph G are formed for each itemset in T:

$$\{E\} = \left\{ \bigcup_{i=0,j=0}^{n} E_{ij} | \{i,j\} \in T \right\} \tag{2}$$

The weight of each edge E_{ij} is equal to the support of itemset {i, j} in T:

$$W\left(E_{ij}\right) = Support\left(\{i,j\}\right) | \{i,j\} \in T \tag{3}$$

Partition the IAG using the multilevel k-way partitioning algorithm (MLkP)

The Multilevel k-way partitioning (MLkP) algorithm [25] is an efficient graph partitioning algorithm. The time complexity is *O(E)*, where E is the number of edges in the graph, and the maximum load imbalance is limited to 3%.

The general idea of MLkP is to shrink (coarsen) the original graph into a smaller graph, then partition the smaller graph using an improved version of the KL/FM algorithm. Lastly, it restores (uncoarsen) the partitioned graph to a larger, partitioned graph.

METIS is a software developed by Karypis at the University of Minnesota [30]. It includes an implementation of the MLkP algorithm that takes a graph as the input and outputs groups of nodes separated after the partition.

Transaction partitioning

Based on the results of the MLkP algorithm that divide the items into groups P_1, P_2,… ,P_m, we can partition the transactions into the same number of groups, where each group D_i contains only the items in partition P_i. For a transaction tobe included in D_i, it must have all the items from partition P_i. If a transaction includes more items than the items from partition P_i, only the items in P_i that are included in the transaction are added to D_i. That is, only a part of the transaction is added to D_i. As a result, each transaction in a transaction partition must be a subset of the corresponding transaction in the original dataset. If a transaction has less than three items, the transaction is not added. This is because we have already mined the one and two itemsets, and are only interested in itemsets that have 3 or more items. This optimization helps to reduce the size of transaction partitions.

$$D_i = \left\{ \bigcup_{j=1}^{n} T_j | (T_j \rightarrow \mathrm{S_j} \cap P_i | S_j \in D) \right\} \tag{4}$$

In the above, D_i is transaction partition i, T_j is the transactions to be added to partition i, S_j is the jth transaction in the original dataset, P_i is the item partition i, and D is the original dataset.

Since the number of unique items in each partition is less than or equal to $\frac{numberofnodesinIAG}{k}$ rather than $\frac{totalnumberofuniqueitems}{k}$, the size of each partition should be small compared to the original dataset. In rare cases, if the size of a transaction partition is greater than the memory size, the SARL heuristic can partition the IAG and the transactions again with k incremented by 1. This guarantees that each partition fits into the memory.

Selecting an algorithm on transaction partitions

One of the benefits that come with our solution is that the association rule learning on each transaction partition can be optimized by using an algorithm that best fits the partition.

During the association rule learning on the partitioned datasets, we have three candidates that are considered efficient: the Apriori algorithm, the FP-Growth algorithm, and the Eclat algorithm.

Since the modified Apriori algorithm has already computed the one itemsets and two itemsets during the preparation phase, the candidate generation feature of the Apriori algorithm is handy in this case. We modify the Apriori algorithm to skip the frequent one/two itemsets finding stages and start with the frequent three itemsets from the transaction partitions. This modification is particularly helpful when the minsup is set to a high value so that the expected number of itemsets is limited after the two itemsets are found.

We can estimate the expected number of itemsets from the average transaction length of each transaction partition. A higher average transaction length indicates a higher possibility of the presence of a long "tail" in the result. Results with long tails have itemsets with considerable maximum lengths, while results with short tails only contain itemsets with small maximum lengths. A dataset with an expected long tail means the association rule learning algorithm does not terminate soon after the two itemsets are found.

The average transaction length provides a fast and straightforward reference for selecting the best algorithm for each transaction partition. If the average transaction length is low, the Apriori algorithm can be the right choice, as the modified Apriori algorithm continues from the two itemsets that the preparation phase has already calculated. If the average transaction length is high, we can take advantage of the scalability of the FP-Growth algorithm. We omit the Eclat algorithm because the FP-Growth and the Eclat algorithms do not have the same advantage provided by the modified Apriori algorithm, of which the algorithm can start with the two itemsets. In addition, studies [31] show that the Eclat algorithm is slightly less scalable than the FP-Growth algorithm.

Next, the selected algorithm is used to find the frequent local itemsets from the given transaction partition. After the algorithm terminates, a simple union is performed on the frequent itemsets found from each partition. Finally, *Apriori-ap-genrule* is used to derive the rules from the frequent itemsets. This step is relatively simple.

Time complexity and space complexity

The theoretical time and space complexity of the Apriori algorithm is $O(2^d)$ where d is the number of unique items in the dataset.

Time complexity

The theoretical time complexity of the SARL heuristic consists of the complexity of several parts:

2-itemsets generation Finding frequent 2-itemsets requires finding 1-itemsets first. This step is simply $O(n)$ as the algorithm traverses the dataset once. Next, the candidate generation for 2-itemsets takes $O(d^2)$ where d is the number of unique items in the dataset. Finally, the support checking requires $O(n + d^2T)$ where T is the number of transactions in the dataset. Therefore, the time complexity of this step is $O(d^2T + n)$.

IAG construction Since each edge in the IAG is a representation of a frequent two-itemset, and the maximum number of two-itemsets is $\frac{d^2+d}{2}$, the maximum number of edges in IAG is also $\frac{d^2+d}{2}$. Therefore, constructing the IAG takes $O(d + \frac{d^2+d}{2})$ or $O\left(d^2\right)$.

IAG partition The time complexity of the IAG partition process is equal to the time complexity of the MLkP algorithm, which is $O(E)$ or $O(d^2)$.

Transaction partition The dataset is traversed once to assign items into different partitions. Hence the time complexity is $O(n)$.

Running a selected algorithm The algorithm selection requires the calculation of the average transaction width of each transaction partition. The time complexity of this is $O(kn)$, where k is the number of partitions.

If the modified Apriori algorithm is selected, the theoretical time complexity for each partition is $O(2^{1.03d/k})$ where the coefficient 1.03 comes from the 3% maximum imbalance of the partitions caused by the MLkP algorithm. The total running time for all the partitions is $O\left(k * 2^{\frac{1.03d}{k}}\right) \rightarrow O(2^{\frac{1.03d}{k}})$, and the total time complexity of the SARL algorithm, when the modified Apriori algorithm is selected, is $O\left(d^2T + n + d^2 + d^2 + n + 2^{\frac{1.03d}{k}}\right) \rightarrow O(d^2T + n + 2^{\frac{1.03d}{k}})$. Assume $n \gg d$, and $2^{\frac{1.03d}{k}} \gg n$, the time complexity can be simplified to $O(2^{\frac{1.03d}{k}})$. Compared with the time complexity of the Apriori algorithm, the SARL is $O\left(\frac{2^d}{2^{\frac{1.03d}{k}}}\right) \rightarrow O(2^{\frac{k-1.03}{k}d})$ times faster than the Apriori algorithm. The exponential speedup comes from the smaller number of unique items in each transaction partition. The algorithm that is chosen to mine frequent itemsets from the transaction partitions only needs to consider a portion of all the items for each partition.

Space complexity

Like time complexity, the space complexity of the SARL heuristic consists of the complexity of several parts:

2-itemsets generation Finding the frequent two itemsets requires finding the one itemsets first. This step is $O(d)$, where d is the number of unique items in the dataset, as we need to keep at most d items in the memory. Next, the candidate generation step for the 2-itemsets takes $O(d^2)$ space for at most $\frac{d(d-1)}{2}$ frequent 2-itemsets as candidates. Finally, the support checking requires another $O(d^2)$ space to store the support numbers. Hence, this step requires $O(d^2)$ space.

IAG construction Since each edge in the IAG is a representation of a frequent two-itemset, and the maximum size of the two-itemsets is $\frac{d^2+d}{2}$, the maximum number of edges in IAG is also $\frac{d^2+d}{2}$. Therefore, storing the IAG takes $O\left(d^2\right)$ space. This d^2 space occupation only occurs when every unique item in the dataset is included frequent two-itemsets with every other unique item in the dataset. In most cases, the actual space required to store IAG is smaller than the memory size.

In rare cases, if the IAG cannot fit into the memory, then the Apriori algorithm and FP-Growth algorithm must have memory issues, too. For the Apriori algorithm, all frequent two-itemsets must be stored in the memory to generate the candidates in the next level, and the size of frequent two-itemsets is similar to the IAG. FP-Tree must be stored in the memory for the FP-Growth algorithm. The space complexity of the FP-Tree is also $O\left(d^2\right)$, however, all unique items need to be stored in the tree while only the unique items in the frequent two-itemsets need to be stored in the IAG. Therefore, IAG has a lower space complexity than the FP-Tree.

IAG partition The space complexity of the IAG partition is equal to the space complexity of the MLkP algorithm, which is $O(E)$ or $O(d^2)$.

Transaction partition The dataset is traversed once to assign items into different partitions. We can assume each partition can fit into the memory. Therefore, the space complexity is $O(\frac{n}{k})$.

Selecting and running the selected algorithm The algorithm selection requires the calculation of the average transaction width of each transaction partition. The space complexity of this is O(k) = O(1), where *k* is the number of partitions.

If the modified Apriori algorithm is selected, the theoretical space complexity for each partition is $O\left(2^{\frac{1.03d}{k}}\right)$, where the coefficient 1.03 comes from the default 3% maximum imbalance of partitions caused by the MLkP algorithm. The total space complexity for all partitions is therefore $O\left(k * 2^{\frac{1.03d}{k}}\right) \rightarrow O(2^{\frac{1.03d}{k}})$, and the total space complexity of the SARL heuristic, when the modified Apriori algorithm is selected, is $O\left((3-1) * d^2 + \frac{n}{k} + 2^{\frac{1.03d}{k}}\right) \rightarrow O(d^2 + \frac{n}{k} + 2^{\frac{1.03d}{k}})$. Assume $\frac{n}{k} \gg d$, and $2^{\frac{1.03d}{k}} \gg \frac{n}{k}$, the space complexity can be simplified to $O(2^{\frac{1.03d}{k}})$. Compared with the space complexity of the Apriori algorithm, SARL uses only $O\left(\frac{2^{\frac{1.03d}{k}}}{2^d}\right) \rightarrow O\left(2^{\frac{1.03-k}{k}d}\right) \rightarrow o(\frac{1}{2^{\frac{k-1.03}{k}d}})$ space comparing to the Apriori algorithm. The exponential reduction of space usage comes from the smaller number of unique items in each transaction partition. If the modified Apriori is chosen to mine frequent itemsets from the transaction partitions, it only generates a smaller number of candidates for each transaction partition, since it does not consider items in other partitions.

Error bound

The SARL heuristic sacrifices some precision to obtain the speed up. However, every frequent itemset found by the algorithm is correct, and the support associated with each frequent itemset is also correct. The heuristic may miss some trivial frequent itemsets, i.e., the itemsets with low support. During the IAG partition phase, the MLkP algorithm makes cuts on the IAG to minimize the sum of the weights of the edges that are cut off. This feature helps to prevent large weights from cut off, while some trivial, small-weight (support) edges may be lost.

In the most (extreme) case, when every transaction has all the items and *minsup* is set to 0, we can calculate the error bound. In this case, the IAG is a complete graph, and the fraction of the edges cut off by the MLkP algorithm is $\frac{n*\left(n-\frac{n}{k}\right)}{E} = \frac{(k-1)n}{k(n-1)}$. When n is very large, the fraction is approximately$\frac{k-1}{k}$. In this case, we can set k as low as 2 to still maintain 50% coverage for the frequent three or more itemsets. The calculation of frequent one and two itemsets is always accurate because they are calculated using the Apriori algorithm or the direct-generate algorithm.

The error rate should be significantly lower in more practical cases. However, it is difficult to estimate such an error rate considering it is affected by many factors such as the closeness of groups of items (i.e., does an item appear with only a small number of other items?), the choice of *minsup*, and the max length of the frequent itemsets. We can make a rough estimation by introducing a parameter P_{out}, the ratio of the edges cut off in the IAG. $P_{out} = \frac{E_{cut}}{E_{total}}$. This parameter is determined by the characteristics of a dataset, the *minsup* choice, and the number of partitions we choose. P_{out} is also a rough estimation of the error rate for the frequent two or more itemsets. Assume the ratio of the frequent two or more itemsets found is P_m,

$$P_m = \frac{\#frequent2 + itemsets}{\#totalfrequentitemsets} \tag{5}$$

then the total error bound can be computed as

$$Error_{total} = P_m * P_{out} \tag{6}$$

Initial selection of number of partitions, k

The selection of k determines the speed and accuracy of the SARL heuristic. A larger k usually means faster speed and lower accuracy, and vice versa. Depending on the size of the dataset and the application, $k = 2$, 3, or 4 are some balanced choices. In rare cases, the heuristic will increase the k value if any transaction partition cannot fit into the memory based on the current setting of k.

Benefits of having datasets fit into the memory

According to "Our solution" section, Part 8, the transaction partitions are guaranteed to be small enough to fit into the memory. Therefore, any operations performed on these in-memory datasets should be faster than before. For example, the Apriori algorithm makes the number of passes on the dataset equal to the maximum length of frequent itemsets. Each of these passes requires reading the dataset from the disk. With our solution, the SARL heuristic makes at most two passes to the dataset. The first pass is to generate the frequent one and two itemsets, and in the second pass, the algorithm brings a fraction of the dataset into the memory. All further passes are made directly in the memory, resulting in speedup.

We do not analyze the communication cost between the main memory and the hard disk quantitatively in this paper. Due to the nature of our divide-and-conquer approach, we do not implement any additional swapping mechanism, so each partition is only brought into the memory once. Therefore, such cost should be no larger than the cost of the Apriori algorithm.

Theorems and proofs

Theorem 1 *Soundness—All frequent itemsets and association rules generated by the SARL heuristic are correct.*

Proof Assume the SARL heuristic generates an incorrect frequent itemset. We can assume the correctness of the Apriori algorithm and the FP-growth algorithm.

Therefore, there must be an error in transaction partitioning. There could be two possible types of error in transaction partitioning:

(Possibility 1) The support of some itemsets is higher or lower than it should be.

(Possibility 2) Some transactions include additional items or lose some items.

Assume the first possibility is true. We divide the dataset vertically (item-wise) during the transaction partitioning phase. Since every item in the original dataset D that belongs to P_i must be added to D_i, all unique items in a transaction partition must appear in the same number of transactions as the original dataset. Hence, the support of each itemset should be the same as the original dataset. This conflicts with the first possibility: the support of some itemsets is higher or lower than it should be.

Assume the second possibility is true. During the transaction partitioning phase, each transaction in the original dataset may be assigned to a transaction partition, or it may be split into different disjoint parts. Therefore, each transaction in a transaction partition must be a subset of the corresponding transaction in the original dataset, and this process cannot add any new items into any transactions. If some items are lost during the transaction partitioning phase, the results may have incorrect supports. However, we know that the union of the unique items in each transaction partition is equal to the unique items of the frequent two-itemsets, since the IAG partitioning cuts off some edges of IAG but not the nodes. According to the Apriori principle, a three-itemset can be frequent if and only if all its two-item subsets are frequent. This means that the unique items of three or more frequent itemsets must be a subset of the unique items of frequent two-itemsets. Hence, we have

$$\forall n \geq 3, I_n \subseteq I_2 = \bigcup_{j=1}^{m} P_j \tag{7}$$

where I_n is the unique items of frequent n-itemsets, P_j is the unique items of transaction partition j, and m is the number of transaction partitions. Therefore, all items needed by the frequent three (or higher) itemsets are present in the transaction partitions. Hence, we find a contradiction between our algorithm and the second possibility.

In summary, since both possibilities are proved to be false, the SARL heuristic is sound. $\square$

Theorem 2 *Computing the frequent two itemsets is considered relatively trivial compared to computing the frequent three or more itemsets.*

Proof If the computation of the frequent two itemsets takes more than half of the total computation time, we may say computing frequent two itemsets is not trivial.

To characterize the distribution of frequent itemsets is relatively difficult due to the challenges in modeling the data. We develop a mathematical model to simulate the

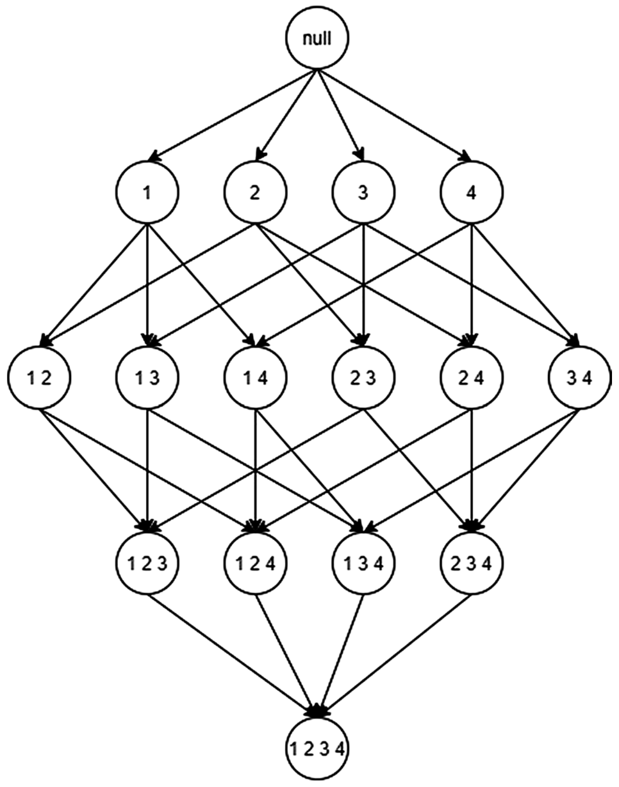


Fig. 4 An itemset lattice

characteristics of any dataset. The relationships of all the frequent itemsets can be depicted using an itemset lattice diagram shown below:

Figure 4 shows the case when every itemset has a support greater than *minsup*. However, in most cases, each layer will have some itemsets being removed due to either one of the two reasons: the anti-monotone property of the Apriori principle or the lack of support (i.e., support < *minsup*). To model the former, we apply the anti-monotone property to the itemset lattice. The anti-monotone property is as follows:

$$\forall X, Y \in J : (X \subset Y) \rightarrow f(Y) \leq f(X) \tag{8}$$

where if $J = 2^I$, I being a set of items, X is a subset of Y, then the measure f must be anti-monotone. Applying this property to the lattice, we can have the following explanation: if an itemset is infrequent, then all of its supersets must also be infrequent.

For example, in Fig. 5, if {*1, 3}* is infrequent, then *{1, 2, 3}*, *{1, 3, 4}*, and *{1, 2, 3, 4}* are all infrequent. To model this property, we can imagine that each infrequent itemset in the same layer causes some supersets in the next layer to be infrequent. The first infrequent itemset results in $n\text{-}k+1$ infrequent itemsets in the next layer, where n is the number of unique items in the dataset, and k is the current layer number or the number of items in each itemset of the current layer. We know that each layer has C_k^n itemsets if none of them is infrequent. Then the next layer will have C_{k+1}^n total itemsets. Since $n\text{-}k+1$ is the number of current infrequent itemsets in the next layer,

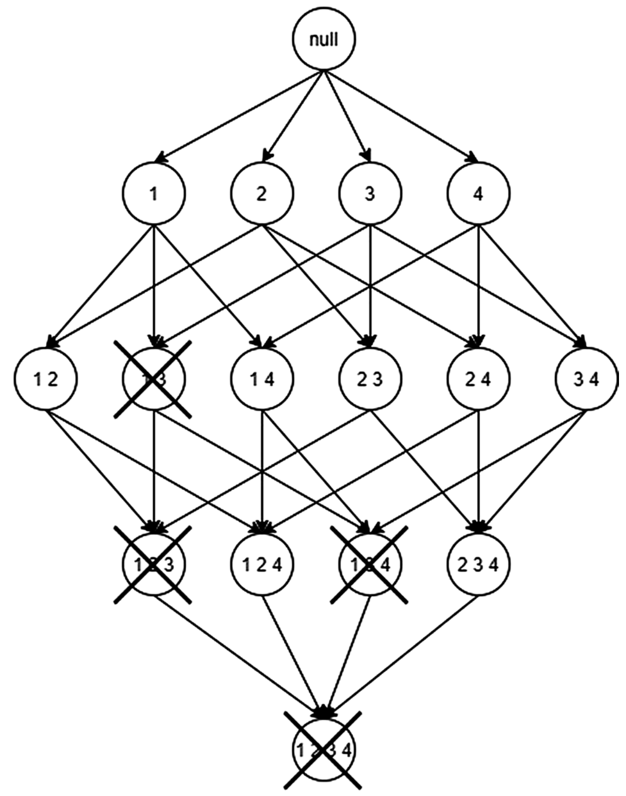


Fig. 5 An example of pruning

Table 11 Estimation of the number of itemsets

p	#two itemsets	#three itemsets	#four itemsets	2/(3 + 4)
0.8	12,736	228,346	972,761	0.010603552
0.6	7164	41,585	714,273	0.009477971
0.4	3184	6761	30,960	0.084409215
0.2	796	589	1898	0.320064335
0.1	199	67	118	1.075675676

$\frac{n-k+1}{C_{k+1}^{n}}$ is the current fraction of frequent itemsets over all the itemsets in the next layer. Therefore, $1-\frac{(n-k+1)}{C_{k+1}^{n}}$ is the probability of having a frequent itemset in the next layer if we randomly choose an itemset, and the second infrequent itemset should cause $\left(1-\frac{(n-k+1)}{C_{k+1}^{n}}\right)*(n-k+1)$ infrequent itemsets in the next layer. For the same reason, the third infrequent itemset in the current layer should cause.
$\left(1-\frac{(n-k+1)+\left(1-\frac{(n-k+1)}{C_{k}^{n}}\right)*(n-k+1)}{C_{k+1}^{n}}\right)*(n-k+1)$ infrequent itemsets in the next layer. We can now estimate the number of infrequent itemsets I in the next layer using the number of infrequent itemsets in the current layer:

Table 12 Estimation of the number of itemsets for larger datasets

p	#two itemsets	#three itemsets	2/3
0.8	1,279,360	231,482,728	0.005527
0.6	719,640	42,159,431	0.017069
0.4	319,840	6,855,578	0.046654
0.2	79,960	597,871	0.133741
0.1	19,990	68,301	0.292675

$$\begin{aligned} I_k =&(n-k+1)+\left(1-\frac{(n-k+1)}{C_{k+1}^n}\right)*(n-k+1) \\ &+\left(1-\frac{(n-k+1)+\left(1-\frac{(n-k+1)}{C_k^n}\right)*(n-k+1)}{C_{k+1}^n}\right)*(n-k+1)+ \end{aligned} \tag{9}$$

The remaining frequent itemsets in layer k, considering the above estimation of the influence of the Apriori principle, is $C_k^n - I_{k-1}$. Let us assume the probability p that an itemset to be frequent, assuming its parent is frequent. We can have the final estimated number of frequent itemsets for layer k:

$$f_k = \left(C_k^n - I_{k-1}\right) * p^k \tag{10}$$

For $n=200$, $p=0.8$, 0.6, 0.4, 0.2, 0.1, we can estimate the number of two, three, and more itemsets as shown in Table 11:

For $n=2000$, $p=0.8$, 0.6, 0.4, 0.2, 0.1, we can estimate the number of two, three, and more itemsets as shown in Table 12:

The above model with examples shows that the number of two itemsets is, on average, less than only 10% of the number of three or more itemsets. This means that only less than 10% of all computation power is consumed by the two itemsets. Thus, our algorithm speeds up the costly part, the part that mines three or more itemsets. $\square$

Theorem 3 *Consider a value of minsup such that the fraction of frequent one itemset over the total number of unique items, d, denoted by f, is less than (1—the maximum imbalance rate), where the maximum imbalance rate is usually set to* 3% *based on the MLkP algorithm. If the partition by MLkP is k-way, then each partition contains less than d/k unique items, where d is the total unique items in the original dataset. As a consequence, the complexity of each partition can be reduced.*

Proof Assume that given f < 100%—3% or f < 97%, and a transaction partition has $d_i \geq d/k$ unique items. According to our algorithm, since $d_i \geq d/k$, a partition in the IAG must have more than or equal to *d/k* nodes. As we assumed earlier, the maximum imbalance rate for the MLkP algorithm is set to 3%, then the number of nodes n in the IAG can be calculated as $\frac{d}{k} * 0.97 * k \leq n \leq \frac{d}{k} * 1.03 * k or 0.97d \leq n \leq 1.03d$. Since n cannot be more than the total number of unique items, $0.97d \leq n \leq d$. However, we know *f < 97%* or *f *d < 0.97d*, and $n \leq f * d$ since some frequent one itemsets

Table 13 Running times of different algorithms on the Bible dataset

	fp	sarl 2apF	sarl 2fpF	sarl 4apF	sarl 4fpF	ap
50	Timeout	8.69149	9.099829	8.689143	13.36314	17.57703
40	Timeout	8.169094	8.981468	9.30843	14.59894	17.60924
30	Timeout	9.39442	10.64038	11.12533	17.64162	21.45831
20	Timeout	10.68418	13.63021	10.22187	10.61251	27.32679
10	Timeout	20.87115	30.99064	27.77191	30.65479	51.8529

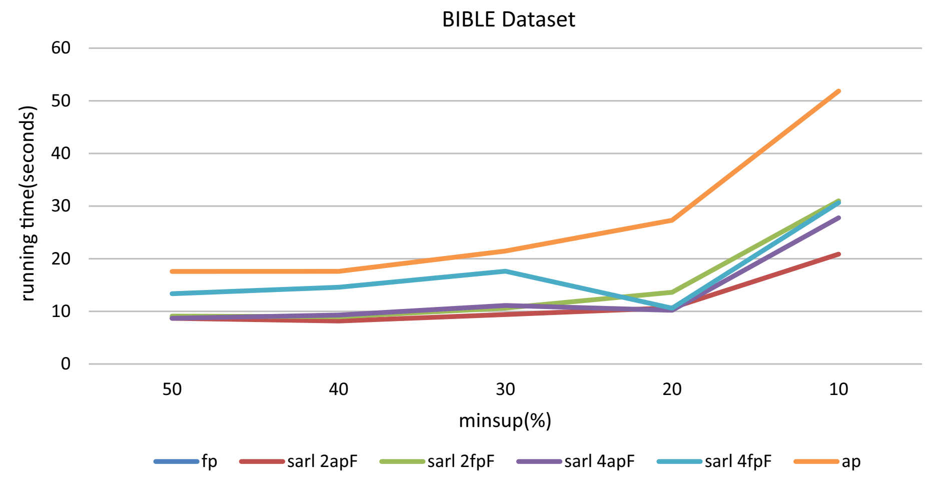


Fig. 6 Running times of different algorithms on the Bible dataset

may not appear in any frequent two itemsets, so $n \leq f * d < 0.97d$ and $n < 0.97d$. This contradicts $0.97d \leq n \leq d$. Therefore, the assumption $d_i \geq d/k$ is false, and the reverse, $d_i < \frac{d}{k}$, must be true. $\square$

Experiments and results

We design and conduct experiments on both small and large datasets to demonstrate the scalability of our algorithm. The experiments are performed on a computer with the following settings:

1. OS: Ubuntu 64-bit running on a virtual machine
2. CPU: Intel Core i7-4720HQ
3. Memory: 8192 MB allocated to the virtual machine
4. Disk: 5400RPM, 64 MB Cache, 6.0 Gb/s, SSHD, 8 GB flash memory
5. Programming Language: Python 3.7

The datasets [32] we use include Bible [33], T10I4D100K [32], and T40I10D100K [32]. The details of each dataset will be discussed later.

Table 14 Accuracy of the SARL algorithm on the Bible dataset

	sarl 2apF & sarl 2fpF (%)	sarl 4apF & sarl 4fpF (%)
50	73.91	100.00
40	60.00	100.00
30	51.72	100.00
20	44.17	40.83
10	39.80	31.37

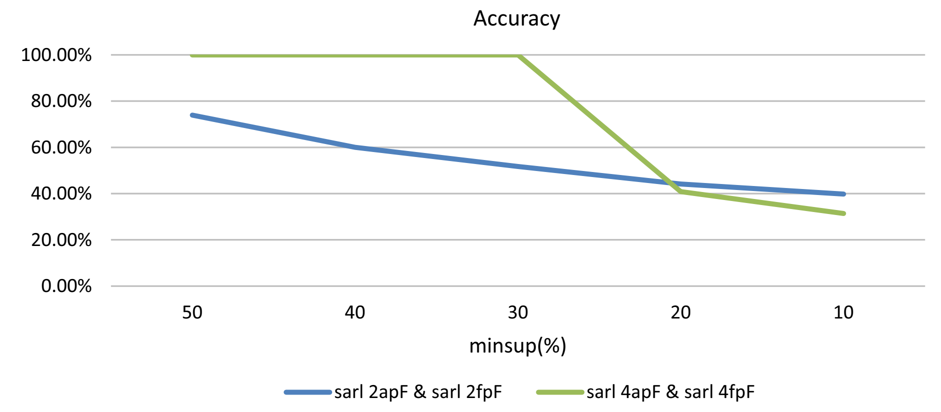


Fig. 7 Accuracy of different configurations of SARL heuristic on the Bible dataset

For each of these datasets, we test the SARL heuristic with various settings for the FP-Growth and the Apriori algorithms on different values of *minsup*. The various settings of the SARL heuristic are as follows:

2ap: $k = 2$, Apriori-based.
2fp: $k = 2$, FP-Growth-based.
4ap: $k = 4$, Apriori-based.
4fp: $k = 4$, FP-Growth-based.

The bible dataset

The Bible dataset has the following metrics:

1. Number of unique items: 13,905
2. Number of transactions: 36,396
3. Average transaction width: 21.6
4. File size: 5.4 MB

This is a small to medium-sized dataset. The experiments are done repeatedly for *minsup* of 50%, 40%, 30%, 20%, and 10%. The time limit for each experiment is set to 800 s for each of the experiments. The results are shown in Table 13:

Table 15 T10I4D100K running times

	fp	sarl 2apF	sarl 2fpF	sarl 4apF	sarl 4fpF	ap
10	2.835414	19.57188	19.11257	18.71221	19.44224	5.622139
4	4.288454	18.61193	18.25904	18.43984	18.39262	18.49691
1	277.5966	32.34425	21.23481	21.47209	20.94079	Timeout
0.7	288.9096	24.09774	23.6652	22.95777	23.90843	Timeout
0.4	Timeout	58.42791	199.8875	56.80203	207.7374	Timeout

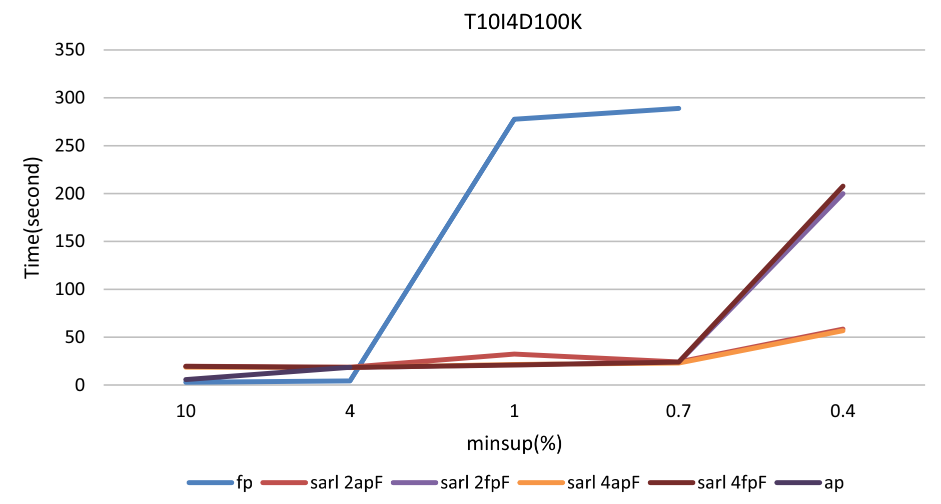


Fig. 8 Running times of different algorithms on the T10I4D100K dataset

According to Fig. 6, the two-partition, Apriori-based SARL heuristic scales the best for this dataset regardless of the *minsup* value. It is 2 to 2.5 times faster than the Apriori algorithm. The FP-Growth algorithm reaches the 800-s time limit for all test cases. It is possible that the number of unique items in this dataset is large; therefore the FP-tree cannot fit into the memory. As a result, the FP-growth algorithm does not perform well here. All other three settings of the SARL heuristic outperform the Apriori algorithm. Comparing to the FP-growth algorithm and the Apriori algorithm, the SARL heuristic is more scalable with all values set for *minsup*.

As we proved earlier, all the frequent itemsets found by the SARL heuristic are accurate, with the correct support. This is important because we need the accurate support to calculate the confidence of the rules. The SARL heuristic may miss some frequent itemsets with a lower support. Here, we calculate the accuracy $= \frac{\mathrm{numberoffrequentitemsetsfoundbySARL}}{numberoffrequentitemsetsfoundbyApriori}$. The accuracy of the SARL heuristic drops on the Bible dataset when the value of *minsup* is low. From Table 14 and Fig. 7, both settings of the four-partition SARL heuristic achieve 100% accuracy from the *minsup* range of 50% to 30%. This is because the MLkP algorithm is able to find a perfect or almost perfect cut on the IAG so that there are no inter-partition frequent itemsets for this range. When 100% accuracy is achieved, the SARL heuristic discovers not just the one and two frequent itemsets, but also the three or higher frequent itemsets. As

Table 16 Accuracy of SARL heuristic on T10I4D100K

	sarl 2apF (%)	sarl 2fpF (%)	sarl 4apF (%)	sarl 4fpF (%)
10	100	100	100	100
4	100	100	100	100
1	100	100	100	100
0.7	100	100	100	100
0.4	100	100	100	100

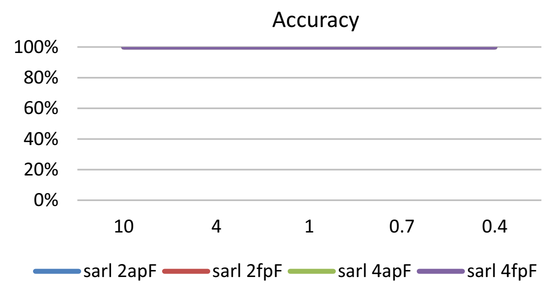


Fig. 9 Accuracy of SARL heuristic on T10I4D100K

Table 17 Running times of different algorithms on the T40I10D100K dataset

	fp	sarl 2apF	sarl 2fpF	sarl 4apF	sarl 4fpF	ap
20	8.736294	229.0365	235.9384	232.3693	231.474	23.49573
10	Timeout	228.2672	226.8315	239.0484	233.1036	158.8888
7	Timeout	236.5143	233.9087	235.3071	238.6583	Timeout
4	Timeout	241.4238	252.5584	242.9868	241.8205	Timeout

for the two-partition SARL heuristic settings, the accuracy starts at 73.91% at 50% *minsup* and drops to 39.8% at 10% *minsup*.

The T10I4D100K dataset

The second dataset we have tested is T10I4D100K. It has the following statistics:

1. Number of unique items: 870
2. Average size of transactions: 10
3. Number of transactions: 100,000
4. File size: 4 MB

The algorithms are tested on T10I4D100K for *minsup* of 10%, 4%, 1%, 0.7%, and 0.4%. This dataset has a medium size (for this environment), so the time limit is set to 300 s for each of the experiments.

Table 15 and Fig. 8 shows the results for T10I4D100K:

From Fig. 8, the Apriori algorithm has an average performance for the initial *minsup* of 10% and 4%. However, it quickly reaches the maximum running time after that

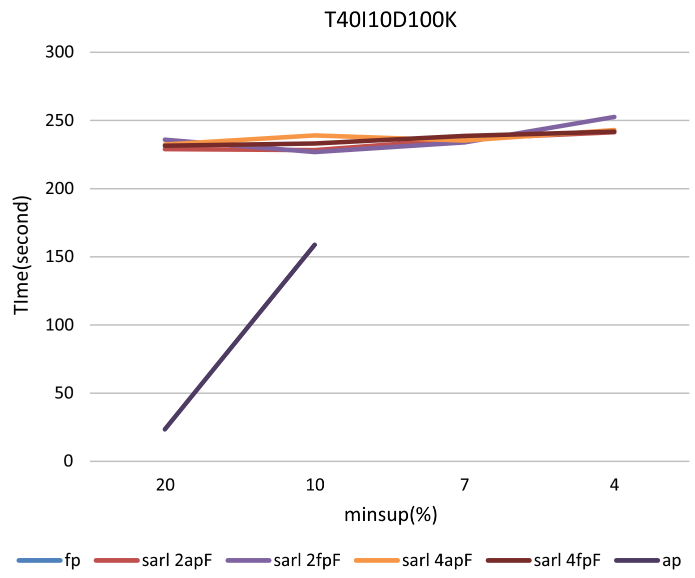


Fig. 10 Running times of different algorithms on the T40I10D100K dataset

and unable to finish the task in time for all subsequent settings of *minsup*. The FP-Growth algorithm has a better performance. It is the fastest for a higher value of *minsup* of 10% and 4%, but it jumps to almost 300 s for 1% and 0.7%, before timeout at 0.4%. All settings of the SARL heuristic outperform the Apriori and the FP-Growth algorithm for middle and low settings of *minsup*. It is 8.6 to 13.8 times faster than the FP-Growth algorithm on minsup = 1% and 0.7%. The SARL heuristic is slightly slower at a high *minsup* of 10%, and they are tied with the Apriori but slightly slower than FP-Growth at a *minsup* of 4%.

Table 18 Accuracy of SARL heuristic on T40I10D100K dataset

	sarl 2apF (%)	sarl 2fpF (%)	sarl 4apF (%)	sarl 4fpF (%)
20	100	100	100	100
10	100	100	100	100

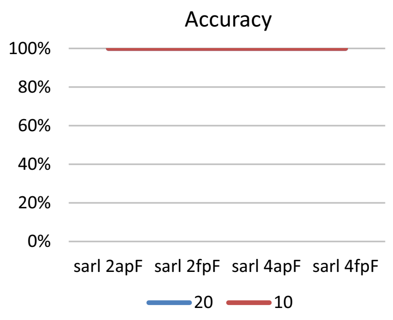


Fig. 11 Accuracy of SARL heuristic on T40I10D100K dataset

The accuracy of the SARL heuristic is high on the T10I4D100K dataset. As shown in Table 16 and Fig. 9, all four settings of the SARL heuristic achieve 100% accuracy for the values of *minsup* from 10% to 0.4%. This is because, for a high *minsup*, the number of frequent three or more itemsets for this dataset is small comparing to frequent two itemsets, and the mining of the one and two frequent itemsets is accurate. For low *minsup* values, the MLkP algorithm successfully finds a perfect or almost perfect cut on the IAG, so the results are accurate.

The T40I10D100K dataset

The dataset T40I10D100K has the following statistics:

1. Number of unique items: 942
2. Average size of transactions: 40
3. Average size of the maximal potentially large itemsets:10
4. Number of transactions:100,000
5. File size: about 15 MB

This relatively large-size dataset was tested on *minsup* values of 20%, 10%, 7%, and 4%. The maximum running time was set to 300 s each for the experiments.

Table 17 shows the results of the experiments:

The results of the experiments (shown in Table 17 and Fig. 10) show an obvious distinction between the scalability of different algorithms. All settings of the SARL heuristic demonstrate high scalability. Almost all settings of the SARL heuristic have stable running time throughout the entire range of *minsup*. Surprisingly, the Apriori algorithm performs better than the FP-Growth algorithm with a *minsup* between 20 and 7%. However, it is still unable to terminate within the time limit for $minsup = 4\%$. Lastly, the FP-Growth algorithm does not scale very well on this dataset. It fails to terminate within the given time for both 7% and 4% of *minsup*.

The accuracy of the SARL heuristic on the T40I10D100K dataset is the same as the T10I4D100K dataset. Table 18 and Fig. 11 show that the SARL heuristic has 100% accuracy based on similar reasons as we explained above in the analysis of the T10I4D100K experiment results.

Conclusions and future work

In this paper, we have proposed a scalable, highly parallelizable association rule mining heuristic (the SARL heuristic). The contributions include the use of the divide-and-conquer method to speed up complex computations, the use of an item association graph that provides an efficient estimation of potential frequent itemsets, and the use of the MLkP algorithm to divide the items into partitions while minimizing the loss of information. We have shown the scalability of the SARL heuristic through a series of experiments. The results indicate that the SARL heuristic has better scalability, with high accuracy, than both the Apriori and the FP-Growth algorithms in most cases.

As discussed, the proposed heuristic is limited by the space requirement that the memory should be large enough to accommodate the IAG (proportional to d^2 where *d* is the number of unique items in the transactions) which we think may be a reasonable assumption in practice.

In the future, we plan to extend our work with the following tasks:

- Develop a parallel version of the SARL heuristic and its implementation. The transaction partitions can be considered as independent datasets, and we can easily run the modified Apriori algorithm or FP-Growth algorithm on each of the transaction partition in parallel and then merge the results (frequent three or higher itemsets) together along with the frequent one and two itemsets to obtain the total frequent itemsets. Each parallel processor does not need to communicate with others during the computation since all the information needed is already included in the local dataset. This would result in maximum utilization of each processor.
- Study how different characteristics of the datasets influence the performance of the SARL heuristic. Although we know that the SARL heuristic has excellent performance for most datasets, the exact speed and accuracy of the SARL heuristic are still unpredictable. We think by applying some statistical measurements on the dataset, it is possible to estimate the accuracy and speed of the SARL heuristic roughly. This will help the user to determine if using the SARL heuristic is beneficial enough compared to other accurate algorithms.

Abbreviations

K-itemset: An itemset with *k* items; Minsup: The minimum requirement of support. The user usually provides this. Itemsets with support < *minsup* are eliminated.; Minconf: The minimum requirement of confidence. The user usually provides this. Rules with confidence < minconf are eliminated.; IAG: Item-association graph; MLkP: Multilevel k-way partitioning algorithm; SARL: Scalable association rule learning.

Acknowledgements

Not applicable

Authors' contributions

Both HL and PS are major contributors to writing the manuscript. HL and PS have made substantial contributions to the conception and design of work. HL has created the software used in this paper. Both HL and PS have approved the submitted version and have agreed on both to be personally accountable for the author's own contributions and to ensure that questions related to the accuracy or integrity of any part of the work, even ones in which the author was not personally involved, are appropriately investigated, resolved, and the resolution documented in the literature. Both authors read and approved the final manuscript.

Funding

Not applicable.

Availability of data and materials

The datasets analyzed during the current study are available in the SPMF Open-Source Data Mining Library repository[http://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php], The Graph Partitioning Archive[https://chriswalshaw.co.uk/partition/.], and Frequent Itemset Mining Dataset Repository[http://fimi.uantwerpen.be/data/].

Declarations

Ethics approval and consent to participate

Not applicable.

Consent for publication

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Received: 27 February 2021 Accepted: 22 May 2021
Published online: 09 June 2021

References

1. Kaur M, Kang S. Market basket analysis: identify the changing trends of market data using association rule mining. Procedia computer science. 2016;85:78–85.
2. Naulaerts S, Meysman P, Bittremieux W, Vu TN, Vanden Berghe W, Goethals B, Laukens K. A primer to frequent itemset mining for bioinformatics. Brief Bioinform. 2015;16(2):216–31.
3. Agrawal R, Srikant R. Fast algorithms for mining association rules. In: Proceedings of 20th international conference very large data bases, vol. 1215, VLDB. 1994, pp. 487–99.
4. Han J, Pei J, Yin Y. Mining frequent patterns without candidate generation. ACM SIGMOD Rec. 2000;29(2):1–12.
5. Zaki MJ. Scalable algorithms for association mining. IEEE Trans Knowl Data Eng. 2000;12(3):372–90.
6. Agrawal R, Imieliński T, Swami A. Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIGMOD international conference on Management of data. 1993, pp. 207–16.
7. Stone A, Shiffman S, Atienza A, Nebeling L. The science of real-time data capture: self-reports in health research. Oxford University Press; 2007.
8. Dubois E, Blättler C, Camachon C, Hurter C. Eye movements data processing for ab initio military pilot training. In: International conference on intelligent decision technologies. Springer, Cham, 2017. pp. 125–35.
9. Shiau Y, Liang S. Real-time network virtual military simulation system. In: 2007 11th international conference information visualization (IV '07), Zurich. 2007, pp. 807–12. doi: https://doi.org/10.1109/IV.2007.93.
10. Chee CH, Jaafar J, Aziz IA, Hasan MH, Yeoh W. Algorithms for frequent itemset mining: a literature review. Artif Intell Rev. 2019;52(4):2603–21.
11. Park JS, Chen MS, Yu PS. An effective hash-based algorithm for mining association rules. ACM SIGMOD Rec. 1995;24(2):175–86.
12. Lin X. Mr-apriori: Association rules algorithm based on mapreduce. In: 2014 IEEE 5th international conference on software engineering and service science. IEEE. 2014, pp. 141–144.
13. Nadimi-Shahraki MH, Mansouri M. Hp-Apriori: Horizontal parallel-apriori algorithm for frequent itemset mining from big data. In: 2017 IEEE 2nd international conference on big data analysis (ICBDA). IEEE. 2017, pp. 286–290.
14. Houtsma M, Swami A. Set-oriented mining for association rules. IBM Almaden research center. research report RJ 9567, San Jose. 1993.
15. Agarwal RC, Aggarwal CC, Prasad VVV. A tree projection algorithm for generation of frequent item sets. J Parallel Distrib Comput. 2001;61(3):350–71.
16. Song M, Rajasekaran S. A transaction mapping algorithm for frequent itemsets mining. IEEE Trans Knowl Data Eng. 2006;18(4):472–81.
17. Baralis E, Cerquitelli T, Chiusano S. Grand A. P-Mine: Parallel itemset mining on large datasets. In: 2013 IEEE 29th international conference on data engineering workshops (ICDEW). IEEE. 2013, pp. 266–271.
18. Pyun G, Yun U, Ryu KH. Efficient frequent pattern mining based on linear prefix tree. Knowl-Based Syst. 2014;55:125–39.
19. Hoseini MS, Shahraki MN, Neysiani BS. A new algorithm for mining frequent patterns in can tree. In: 2015 2nd international conference on knowledge-based engineering and innovation (KBEI). IEEE. 2015, pp. 843–846.
20. Feddaoui I, Felhi F, Akaichi J. EXTRACT: New extraction algorithm of association rules from frequent itemsets. In: 2016 IEEE/ACM international conference on advances in social networks analysis and mining (ASONAM). IEEE. 2016, pp. 752–6.
21. Croushore D. Frontiers of real-time data analysis. J Econ Liter. 2011;49(1):72–100.
22. Yang XY, Liu Z, Fu Y. MapReduce as a programming model for association rules algorithm on Hadoop. In: The 3rd international conference on information sciences and interaction sciences, Chengdu. 2010, pp. 99–102. Doi: https://doi.org/10.1109/ICICIS.2010.5534718.
23. Buluç A, Meyerhenke H, Safro I, Sanders P, Schulz C. Recent advances in graph partitioning. In: Algorithm engineering. Springer, Cham. 2016, pp. 117–58.
24. Kernighan BW, Lin S. An efficient heuristic procedure for partitioning graphs. Bell Syst Tech J. 1970;49(2):291–307.
25. Karypis G, Kumar V. Multilevelk-way partitioning scheme for irregular graphs. J Parallel Distrib Comput. 1998;48(1):96–129.
26. McSherry F. Spectral partitioning of random graphs. In: Proceedings 42nd IEEE symposium on foundations of computer science. IEEE. 2001, pp. 529–537.
27. Galinier P, Boujbel Z, Fernandes MC. An efficient memetic algorithm for the graph partitioning problem. Ann Oper Res. 2011;191(1):1–22.
28. Sanders P, Schulz C. Engineering multilevel graph partitioning algorithms. In: European symposium on algorithms. Springer, Berlin, Heidelberg. 2011, pp. 469–480.
29. Walshal C. The graph partitioning archive. 2020. https://chriswalshaw.co.uk/partition/.
30. Karypis G, Kumar V. A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM J Sci Comput. 1998;20(1):359–92.
31. Heaton J. Comparing dataset characteristics that favor the Apriori, Eclat or FP-Growth frequent itemset mining algorithms. SoutheastCon 2016, Norfolk, VA. 2016, pp. 1–7, doi: https://doi.org/10.1109/SECON.2016.7506659.
32. Goethals B. Frequent itemset mining dataset repository. 2020. http://fimi.uantwerpen.be/data/
33. Fournier-Viger P, Lin CW, Gomariz A, Gueniche T, Soltani A, Deng Z, Lam HT. The SPMF open-source data mining library version 2. In: Proceedings 19th European conference on principles of data mining and knowledge discovery (PKDD 2016) Part III, Springer LNCS 9853. 2016, pp. 36–40.

Publisher's Note