UCLouvain-EPL

Mining Patterns in Data

Project 1: Implementing Apriori

Teacher:
Siegfried Nijssen
Course assistants:
Thomas Charles
Guillaume Derval

Authors:
Nima Farnoodian - 68372000
nima.farnoodian@student.uclouvain.be

César Liesens - 39161400
cesar.liesens@student.uclouvain.be

Group 53

1 Introduction

In this project, for the Frequent Itemset Mining task, we have been asked to implement two itemset miners one of which must be the Apriori algorithm, and the other one should be one of its improved versions or a DFS based-method such as ECLAT. The aim of this project is to compare the performance of the selected miners in terms of run-time. As our Itemset miners, we have chosen the original `Apriori algorithm` and the `Apriori Graph algorithm`[1] that is deemed to be faster than the original one according to the authors' claim. The Apriori Graph algorithm benefits from a $O(|V| * |E|)$ time complexity, where V and E are the set of nodes and links of the obtained graph, respectively. Before mentioning anything about these approaches, it is interesting to say that, in this project, we suggest the use of vertical representation proposed in ECLAT algorithm in both aforementioned algorithms to speed up the search and candidate selection processes. We show that, using said vertical representation, not only can these algorithms be significantly improved in terms of run-time, but also the original Apriori algorithm could outperform the Apriori Graph algorithm so that it can handle the largest data-set with a lower threshold within a reasonable time. Therefore, due to the page limit, we will mostly converse about our improvements and their positive impact on the performance.

In what follows, in section 2, we will discuss the chosen algorithms and the use of vertical representation. In section 3, we will present our experimental results in-depth. Finally, we will conclude this paper in section 4.

2 Algorithms: Apriori and Modified Apriori Graph

In this section, we will briefly discuss the original implementations of Apriori and Apriori Graph Algorithms. In 3.3, we will then explain our modifications to the algorithms using vertical representation and pinpoint the tiny but efficient changes that we applied to them.

2.1 Apriori: Original implementation

```
def Apriori(transaction dataset ds, threshold x):
    itemsetList[1] = find all unique items with supports >= x in ds
    k = 2
    # supp is referred to as support []
    while currentSet is not empty:
        candidateSet = Generate the candidates of size k
        candidateSet = Prune the candidateSet
        **currentSet = Compute supp and keep those candidates with supp >= x
        itemsetList[k] = currentSet
        k++
    for each level l:
        print(itemsetList[l])
```

Considering a support threshold x, the Apriori algorithm first finds all the unique items with supports above x in the transaction datasets and creates a list of the unique items called the itemset list, let's say *itemsetList*[1]—*itemsetList*[k] is referred to as the set of frequent itemsets with the size of k at level k. Using *itemsetList*[1], at the next level 2, it generates the candidates by joining and combining all the items in *itemsetList*[1]. It then prunes the candidate sets and keeps only the candidates whose subsets were deemed frequent at the previous level 1. Next, it examines all detected candidates at level 2 and adds the candidates with the supports above x to *itemsetList*[2]. The algorithm repeats all the processes above for all levels k until no more frequent itemsets can be obtained. Indeed, it stops at level l if no frequent itemsets at this level can be found — due to the Anti-monotonicity, no more candidates for the level $l + 1$ can be derived since all subsets at level l are infrequent.

Technically speaking, the Apriori algorithms consists of three key functional parts: i- Candidate Generation (Generating candidates of size k using itemsets of size of $k-1$), ii- Pruning the candidate set (Removing the candidates whose subsets are infrequent), and iii- Finding the eligible Itemsets for level k with supports above the threshold. The above pythonic pseudo-code represents the Apriori algorithm.

2.2 Modified Apriori Graph Algorithm

The following pythonic pseudo-codes show the Apriori-Graph-algorithm [1] and AprioriGraph. Note that the Apriori-Graph-algorithm is modified to support the frequent itemset mining.

```
def Apriori-Graph-algorithm(transaction dataset ds, threshold x):
    # supp is referred to as support
    L = find all unique (single) items with supp >= x in ds
    webaddr = Create a graph with |L| nodes that are frequent items (Matrix Rep.)

    # webaddr can be implemented by a dictionary
    for each transaction t in ds:
        ct = find all subsets (x,y) of t of length 2
        for each subset (x,y) in ct:
            webaddr[x][y]++
    Answers = [] # a list holding the frequent itemsets
    result_list = [] # result_list holds the candidate itemset

    for each u in L:
        empty result_list
        result_list.append(u)
        AprioriGraph(result_list, u, webaddr)
    print(Answers) # Answers holds all the frequent itemsets

def AprioriGraph(list result_list, node u, graph webaddr):
    for each connected node v in webaddr[u]:
        if ((webaddr[u][v] > x) and (v not in result_list)):
            result_list.append(v)
            AprioriGraph(result_list, v, webaddr)
    if result_list not in Answers:
        **supp = Compute the support of the result_list (itemset)
        if supp >= x:
            Answer.append(result_list)
    result_list.pop()
    return True
```

To extract the frequent patterns, this modified version of the Apriori algorithm[1] makes use of the concept of graphs and Apriori together, allowing for detecting the patterns in the graph instead of mining the original database at each time. Based on the authors' claim, this technique could confine the Input/Output operations to only two operations and reduce the huge number of combinations in the original algorithm (2^N) to E that is the number of links in the resulting graph.

Considering a support threshold x, this algorithm begins with finding the frequent items —single unique items— whose supports are above x. Then, it builds a weighted graph with a Matrix representation where the nodes are the frequent items, and there exists a weighted and directed link between two nodes if they appear in the same transactions. The weights in this graph are proportional to the support of the endpoints. Moreover, there is a directed link from item A to B if A appears before B in the transactions. Using this graph and starting from each node u, the Apriori Graph Algorithm tries to find possible frequent itemsets with prefix u at each time by traversing the reachable nodes and forming a path from u to the traversed nodes. Each path from u to any successors is considered as a potential candidate. To traverse the nodes, it uses a recursive algorithm called AprioriGraph. During the recursive calls started from u, a node v is traversed provided that v is not already added to the path from u to v and the weight from v's predecessor to v is above the threshold. As said earlier, each path from u to any reachable node v can form a candidate itemset $[u, ..., v]$. After each recursive call returns an itemset, the itemset is examined and its support is computed. If the support is above the threshold, the itemset is therefore added to the detected itemset lists. It is worth noting that, this algorithm was generally proposed for the maximal itemset mining task to discover usage patterns from web data, but we modified it to support frequent itemset mining. According to the authors' claim, the algorithm benefits from a time complexity of $O(|V| \cdot |E|)$, which is much better than $O(e^N)$ time complexity of the original Apriori algorithm[2]. "We should remark that this algorithm without Vertical Representation Improvement could initially solve all the instances on Ingenious".

2.3 Improvement using Vertical Representation

Both the original Apriori algorithm and the Apriori Graph Algorithm attempt to find the candidate itemsets that could be the frequent itemsets. In the original Apriori algorithm, the candidate sets are those itemsets whose subsets are frequent, and in the Apriori Graph Algorithm, the candidate sets are the paths whose links' weights are above the threshold. In both algorithms, after detecting candidate sets, the support for each candidate is calculated and if it is higher than the threshold, the candidate is considered frequent. Given that n is the number of transactions in the database, computing the support takes $\Omega(n)$ in the best case since we should go through all transactions linearly to find those transactions holding the candidate itemset. Therefore, considering a substantial number of candidates, it imposes a huge computational overhead, making both algorithms sluggish. To this end, we here suggest a practical method to compute the supports much more quickly. The idea is inspired by the Vertical Representation proposed in ECLAT.

```
def vertical_intersect(Vertical_Representation VP, Candidate CIT):
    first_item=pick the most left item from CIT
    result= Check if VP stores first_item , if not, return empty set {}
    for each item in CIT:
        result= intersect(VP[item],result)
        if length(result)==0:
            break the loop
    return result
```

In our vertical representation-based improvement, for every single item with support above the threshold, we first store the transaction IDs in which the single item appears. The cool fact behind the vertical representation is that the intersection of vertical representations of two or more items, let's say items A, B, and C, can lead to a set of transaction IDs in which the items appear together—in the frequent itemset problem terminology, these transaction IDs are the cover for itemset (ABC) and the length of the cover is the itemset support. Therefore, using this simple idea, we can compute the support for the candidate sets in the worst-cast run-time of $O(n)$. To explain this worst-case run-time, we should remark that an item may appear in all transactions in the worst-case. Given that all items in a candidate set of size of δ appear in all transactions, the required time to compute the intersects can be bounded above by $O(\delta n)$. Based on the observations and also the anti-monotonicity property, we know that the highly frequent itemsets are mostly small itemsets, whereby δ can be considered constant.

Although the use of the vertical representation gives us a $O(n)$-time support computation for the candidates, it can cause the total run-time to drop dramatically in practice. We know that the frequent itemsets miners get slower as we decrease the threshold. Thanks to anti-monotonicity, we can understand that the itemsets that the miners detect or select as the candidates after decreasing the threshold are mostly less frequent and large but not as large as the database. Thereupon, if the vertical representation is used for support computation, its run-time rarely hits our upper bound of $O(n)$ for the new candidates. So, the use of vertical representation allows for faster support computations for large and sometimes small candidates. In order to utilize the vertical representation improvement in the Apriori and the Apriori graph algorithms, function `vertical-intersect` should be used inside the algorithms, in the lines specified by **. In addition, the vertical representation should be initially computed after loading the database.

3 Performance Evaluation

In this section, we evaluate the performance of the algorithms including Apriori, Apriori Graph, and their improvements with Vertical Representation meant for support computations. The transaction datasets are listed in Table1 in ascending order of magnitude. All datasets used in our experiments are available in [3]. The implementations, measurements, and comparisons were carried out in Python 3.8.3 under Windows 10 on a Dell all-in-one PC powered by Intel Core i5-4590S CPU @ 3.00GHz and 8 GB of main memory.

	Number of Transactions	Number of Items
toy	8	5
chess	3196	75
mushroom	8124	119
pumsb	49046	2113
pumsb-star	49046	2088
connect	67557	129
retail	88162	16470
accidents	340183	468

Table 1: Sample Transaction Datasets

First and foremost, a cursory look at all figures reveals that the Apriori algorithm equipped with Vertical Representation (denoted by Vert. Apriori) outperforms all other algorithms in all datasets except our tiny dataset ”toy”. This observation indicates that the use of the vertical representation for computing supports could sufficiently ameliorate the performance of Apriori algorithm so that it could even handle the task on the largest dataset ”accidents” for the low threshold of 0.4 in around 27 minutes. Moreover, it is evident in Fig. 7 that the original Apriori algorithm could only detect the frequent itemsets on our second largest dataset ”retail” for the threshold of 0.65 while the Vert. Apriori could extract all frequent items in this dataset in less than 20 seconds for even the threshold of 0.01. The reason is that the retail dataset possesses many items, thereby lower chance of forming many itemsets of size$> x$ where x is a tiny number. So, since the itemsets are mostly small and the items in the candidates are less frequent, support computation using Vertical Representation for each candidate takes a small amount of time, which can be deemed $O(1)$, thereby a faster itemset detection process. In opposite, in the original Apriori algorithm, each support computation takes at least $\Omega(n)$, and due to the magnitude of the dataset, the candidate selection process becomes time-consuming. It is worth mentioning that both Apriori Graph and Vert. Apriori Graph algorithms perform well on the same dataset since the Apriori Graph algorithm, in turn, tries to reduce the number of candidates.

Generally speaking, the Vertical Representation idea could lead to improvements on both algorithms although it is more observable and sensible in the original Apriori Algorithm since it generates many candidates for which it requires to compute the supports. In spite of the authors’ claim regarding the Apriori Graph Algorithm in their paper[1], this algorithm does not seem to provide a faster frequent itemset mining—though it may be faster in the Maximal itemset problem. In conclusion, the Vert. Apriori Algorithm is clearly chosen as the winner approach here. However, its space complexity still remains its big drawback.

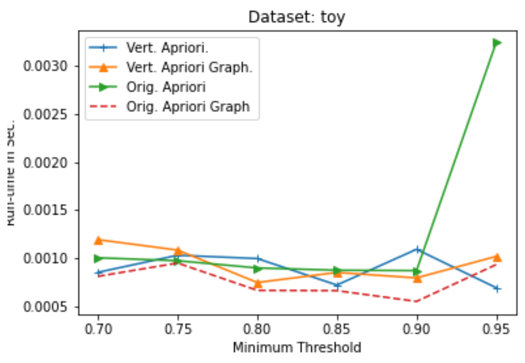

Figure 1: Algorithms' run-time on toy

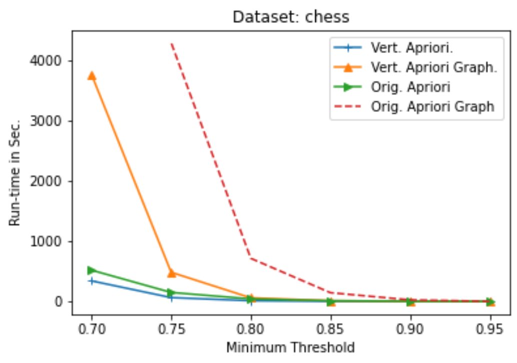

Figure 2: Algorithms' run-time on chess

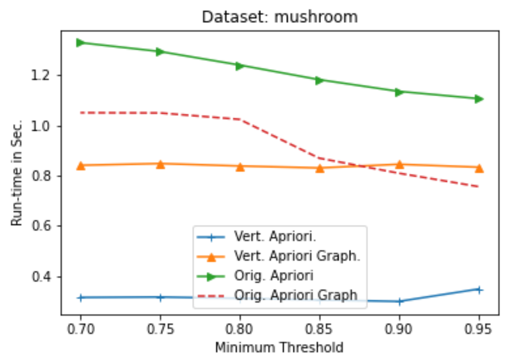

Figure 3: Algorithms' run-time on mushroom

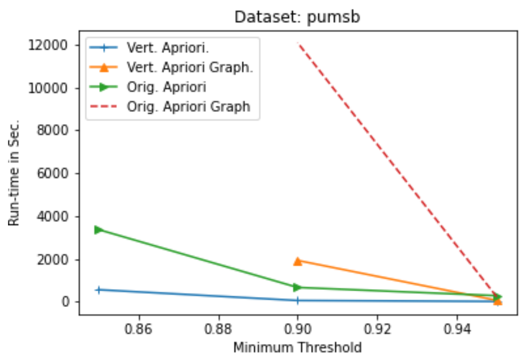

Figure 4: Algorithms' run-time on pumsb

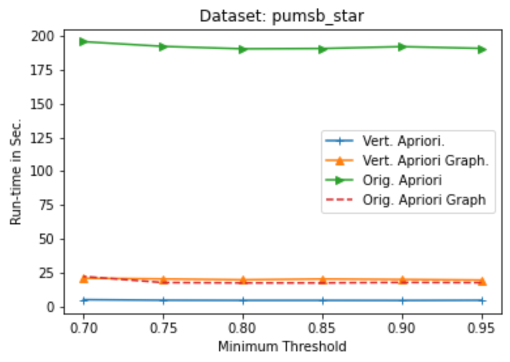

Figure 5: Algorithms' run-time on pumsb-star

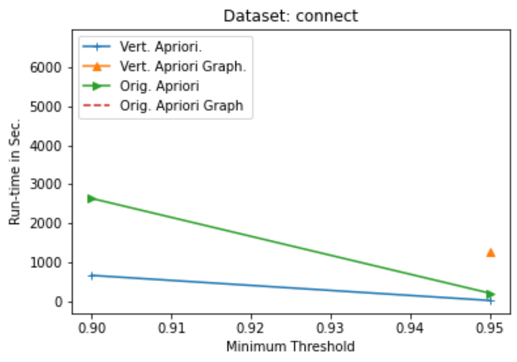

Figure 6: Algorithms' run-time on connect

4 Conclusion

In this project, we worked on a primary algorithm in the Itemset Mining problem called Apriori and one of its variations called the Apriori Graph Algorithm. We first challenged the Apriori Graph algorithm and showed that this algorithm could not be superior to the Apriori algorithm in terms of running-time. Furthermore, we recommended the use of the vertical representation in both algorithms with the hope of reducing the required time for computing the supports for candidate itemsets. The experimental result acknowledged the positive impact of the vertical representation, specifically on the original Apriori algorithm. It is remarkable that, at the time of writing this report, we do not know that if the Vertical Representation had been previously

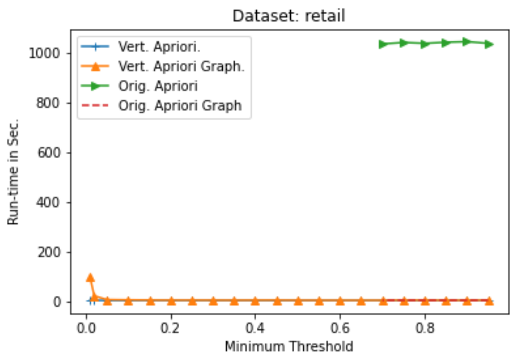

Figure 7: Algorithms' run-time on retail

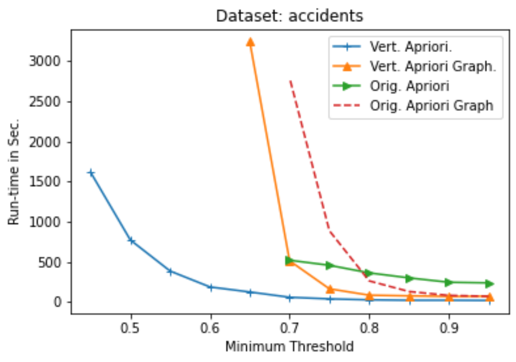

Figure 8: Algorithms' run-time on accidents

propounded for the Apriori algorithm or not. Therefore, we cannot attest its novelty nor its banality.

References

[1] Pritish Yuvraj and Suneetha K. R. *Modified Apriori Graph Algorithm for Frequent Pattern Mining*. International Conference on Innovations in information Embedded and Communication Systems, 2016.

[2] Suneetha K. R. and Krishnamoorthi R. *Advanced Version of Apriori Algorithm*. Integrated Intelligent Computing (ICIIC), 2010 First International Conference, 2010.

[3] *Frequent Itemset Mining Dataset Repository*. http://fimi.uantwerpen.be/data/