CSCI 6405 – Data Mining And Data Warehousing
Course Project Report :
Frequent Itemset Mining using H-Mine algorithm

Cover page (1 Point):

Group Id: 38

Group Members:

1. Aman Jaiswal (B00857194)
2. Mohamed Muzamil H (B00838531)

Research Paper Implemented:

Paper Title : H-Mine: Hyper-Structure Mining of Frequent Patterns in Large Databases

Authors : Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, Dongqing Yang

Publication Venue : IEEE International Conference on Data Mining, San Jose, California, USA. IEEE Computer Society 2001, ISBN 0-7695-1119-8

Publication Time : 29 November - 2 December 2001

Introduction (2 Point):

Data Analytics is an important aspect of the decision-making process which enables us to leverage available data. The Discovery of frequent patterns has emerged as a crucial sub-domain due to its application in a variety of fields ranging from market basket analysis, association mining, sequential mining, clustering to classification. The problem of frequent pattern mining can be defined as finding the complete set of frequent patterns in a given transaction database with respect to a given support threshold.

The key value insights gained from frequent pattern mining helps in growing revenue and improved competitive advantage. However, The increasing data deluge has made the discovery of hidden patterns challenging in terms of computational power and time required to mine such patterns. Therefore, efficient frequent pattern mining algorithms that require less time and less memory for computation are needed. Efficient ways of frequent pattern mining have been studied extensively by many researchers and it is a hot topic of research even today.

There are two classical algorithms for itemset mining namely Apriori and FP-growth. The former is a candidate-generation-and-test approach while the latter is a depth-first approach. Both Apriori and FP-growth algorithms have their drawbacks for different cases. Huge space is required to store a large number of candidate sets in the Apriori algorithm when there exists a large number of frequent patterns in data. Similarly, In the case of large sparse datasets, the FP-tree becomes large and space requirements for the FP-growth algorithm become challenging. In large datasets, the size of candidate sets in the Apriori algorithm and the size of FP-tree in case of the FP-growth algorithm becomes too big to fit in the main memory for processing.

The Hyper-mining research paper proposes a novel hyper-linked data structure called H-struct and a new data mining algorithm called H-mine(Mem) for finding frequent itemsets in a dataset that can fit in the main memory. It can also be scaled to very large databases by database partitioning. Furthermore, the paper also proposes H-mine to consolidate the results of each partition thus providing a scalable approach. H-mine can further be integrated with FP-growth for dense data sets.

The project aims to compare and implement an efficient version of the original H-mine(Mem) algorithm and apriori algorithm covered in the course. In this project, we'll limit our implementation to a memory-based efficient pattern-growth algorithm called H-Mine(Mem) given in Section 2 of the paper for relatively smaller datasets and compare the performance of this approach with that of Apriori algorithm.

Algorithms (4 Points):

Selected Algorithm: H-mine(Mem) (memory-based hyper- structure mining of frequent patterns)

The algorithm is fairly complex and it is hard to condense the complete working of the process in this short report. A deeper understanding requires a thorough reading of the research paper.

The algorithm initializes by obtaining a frequent item projection of the input database.

The complete set of frequent itemsets is mined in just two scans of the database.

Trans ID	Items	Frequent-item projection
100	c, d, e, f, g, i	c, d, e, g
200	a, c, d, e, m	a, c, d, e
300	a, b, d, e, g, k	a, d, e, g
400	a, c, d, h	a, c, d

Fig 1: Example projection of database having support_count =2

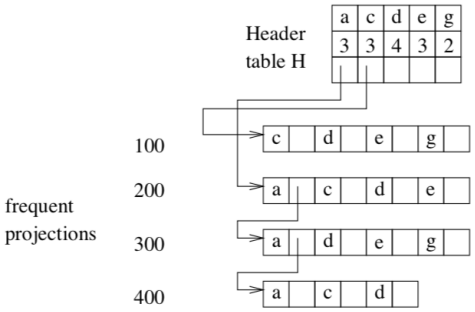

Fig 2: H-struct

Input: A transaction database TDB and a support threshold min_sup.

Output: The complete set of frequent patterns.

Step 1: Scan *TDB* once, find and output *L*, the set of frequent items. Let *F-list*: "*x*1-. . . -*xn*" (*n* = |*L*|) be a list of frequent items.

The first scan of the database is used to find frequent itemsets of length one called the F-list. The F-list is then sorted in lexicographic order. This sorting will help in the mining process. In the above example, (Fig 1) the frequent items are {a : 3, c : 3, d : 4, e : 3, g : 2}.

Step 2: Scan *TDB* again, construct H-struct, with header table *H*, and with each *xi* -queue linked to the corresponding entry in *H*.

The second scan of the dataset is used to create frequent item projections (Fig.1) of the database, which is the occurrence of each frequent item in a transaction(transaction refers to a single line in the database) in the order of their occurrence in the F-list. The complete set of frequent itemsets are generated by mining these frequent projections. A novel data structure called H-struct(hyper-structure) shown in (Fig.2) is created for the mining process.H-struct contains a header table for each level of the mining process. Each row in a header table consists of the frequent item, it's corresponding support count and a hyperlink that links every frequent projection beginning from the corresponding itemset.

Step 3: For *i*=1 to *n* do

(a) Call H-mine({*xi*},*H*,*F-list*).

(b) traverse the *xi*-queue in the header table *H*, for each frequent-item projection *X*, link *X* to the *xj*-queue in the header table *H*, where *xj* is the item in *X* following *xi* immediately.

The main procedure hmine of the algorithm is called in a loop that runs for every element X (a,c,d,e,g) in the initial header table. The details of the hmine procedure are discussed below. The X-Queue generated from the hyperlinks is traversed for each frequent item (a,c,d,e,g) in the header table. The important step here is to link the queue of the item occurring on the left side of the F-list to the right side of the F-list. This dynamic adjustment to the links is required to get the complete queue. Since the projections are sorted in lexicographic order, there might be some items that did not get counted due to their occurrence order in the transaction. The link step ensures every item is counted completely without repetitions.

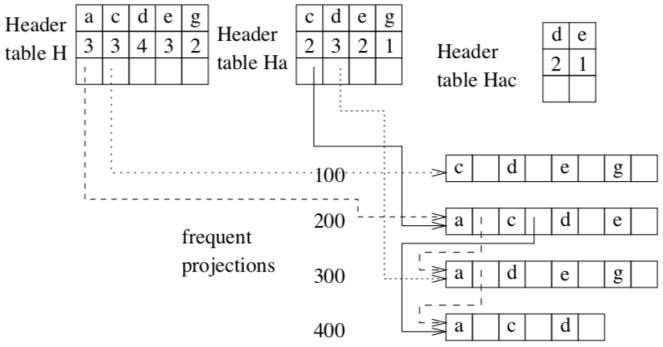

Fig 3: Header Hac.

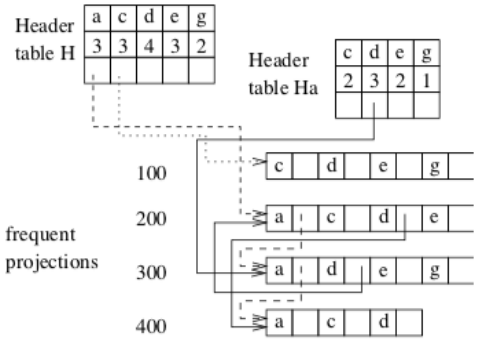

Fig 4: Header and ad pointer.

Procedure: Hime(P, H, F-list)

Step 1: Traverse the *P*-queue once, find and output its locally frequent items and derive *F-list P*: "*xj*1-...-*xjn′* ".

The P-queue (a-queue in the example), which is all the frequent projection beginning with P is traversed with the help of links in h-struct.The local absolute support count of each unique item in p-queue is computed . This allows us to find locally frequent items in the p-queue. The set of the local frequent items must also be frequent in the whole database, therefore this set of frequent items are outputted with P as the prefix. The F-list(P) is derived, which is the list of all the elements occurring on the right side of the P in the F-list(c,d,e,g in case of a-queue). The F-list(P) is used in the next step for making a temporary header table for the mining of longer frequent itemsets.

Step 2: Construct header table *HP*, scan the *P*-projected database, and for each frequent-item projection *X* in the projected database, use the hyperlink of *xji* ($1 \le i \le n'$) in *X* to link *X* to the *Pxji*-queue in the header table *HP*, where *xji* is the first locally frequent item in *X* according to the *F-list(P)*.

The F-list(P) is used to form another header table (HP) in the h-struct which contains the local support count of items in F-list(P) along with a hyperlink which links all the projections that begin with the Prefix X in F-list(P). The Ha header can be seen in (Fig.3) The X-queue in the header table is additionally linked to the some links in (X-1) queue where X occurs. This dynamic adjustment is a necessity to ensure completeness in support count.

Step 3: For*i*=1 to *n′* d
(a) Call H-mine(*P*∪{*xji*},*HP*,*F-listP*).
(b) Traverse *Pxji* -queue in the header table *HP*, for each frequent-item projection *X*, link *X* to the *xjk* -queue ($i < k \le n'$) in the header table *HP*, where *xjk* is the item in *X* following *xji* immediately

according to *F-list*.

Hmine is recursively called with $P \cup \{xji\}$ as the new frequent pattern(ac is passed as the new frequent pattern in Fig.3), which can be imagined as the prefix of a frequent itemset which becomes longer with every recursion.The header table generated in step (2) is passed as the new header table and F-list(P) as the new F-List. The recursive call will output a longer frequent itemset in step 1. And again a new temporary header will be created. Only a single header table is required in level 3 of the mining process.If there are no items that are frequent the algorithm backtracks(Fig 4. The program backtracks and creates a header for 'ad' and also dynamically updates the links of ad-queue with links in ac-queue which consists of 'd') The mining process progresses in a depth-first manner, mining deeper levels of a prefix before moving on to the next prefix.

Other Algorithm: Apriori Algorithm

Apriori algorithm uses a level-wise iterative approach based on candidate generation.It is a classical algorithm in frequent itemset mining.The database is scanned for every level of mining.

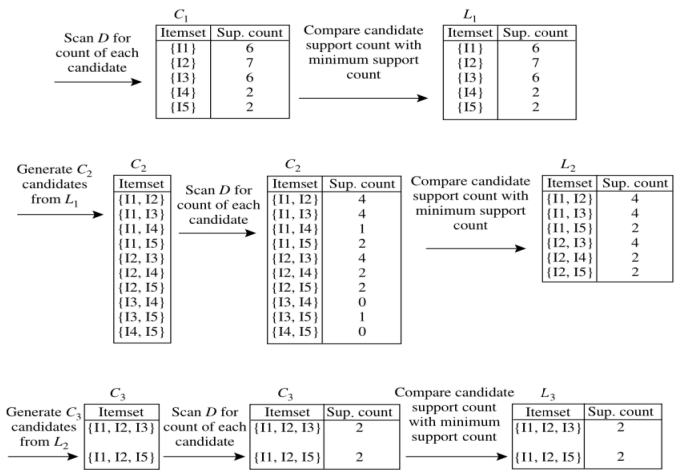

Fig 5: Generation of the candidate itemsets and frequent itemsets, where the minimum support count is 2.

Input: A transaction database TDB and a support threshold min_sup.
Output: L (candidates), frequent itemsets in D (database)
Method:

```
(1)   L1 = find_frequent_1-itemsets(D);
(2)   for (k = 2; Lk−1 ≠ φ; k++) {
(3)       Ck = apriori_gen(Lk−1);
(4)       for each transaction t ∈ D { // scan D for counts
(5)           Ct = subset(Ck, t); // get the subsets of t that are candidates
(6)           for each candidate c ∈ Ct
(7)               c.count++;
(8)       }
(9)       Lk = {c ∈ Ck | c.count ≥ min_sup}
(10)  }
(11)  return L = ∪k Lk;
```

The 1-frequent itemsets are found by a single scan of the database these are denoted by L1.

The candidates in every level are checked for minimum support.
The iterative approach runs in a loop until the generated Lk is null.Apriori_gen function is used to get the candidates for the next level(Fig 5). For every candidate , the support is incremented if the candidate is a subset of the transaction read. Finally Lk, for the next level is found.

```
procedure apriori_gen(L_{k-1}:frequent (k − 1)-itemsets)
(1)   for each itemset l_1 ∈ L_{k-1}
(2)     for each itemset l_2 ∈ L_{k-1}
(3)       if (l_1[1] = l_2[1]) ∧ (l_1[2] = l_2[2])
              ∧... ∧ (l_1[k − 2] = l_2[k − 2]) ∧ (l_1[k − 1] < l_2[k − 1]) then {
(4)         c = l_1 ⋈ l_2; // join step: generate candidates
(5)         if has_infrequent_subset(c, L_{k-1}) then
(6)           delete c; // prune step: remove unfruitful candidate
(7)         else add c to C_k;
(8)       }
(9)   return C_k;
```

The apriori_gen function tries to join every set in the Lk if their first (K-2) are the same and prunes candidates using the has_infrequent_subset function and returns the Ck from L(K-1).

```
procedure has_infrequent_subset(c: candidate k-itemset;
          L_{k-1}: frequent (k − 1)-itemsets); // use prior knowledge
(1)   for each (k − 1)-subset s of c
(2)     if s ∉ L_{k-1} then
(3)       return TRUE;
(4)   return FALSE;
```

Has_infrequent _subset is required for the pruning step in the apriori algorithm.The function is used to find if the candidates generated have any subset (size = k-1) which is infrequent, a candidate having an infrequent subset is removed from the candidate list(Pruning).

Apriori implementation:

We took the traditional approach to implement the apriori algorithm. The candidates for every level are generated using the apriori_gen function. The frequent itemsets are manipulated using the python built-in data type ‘set’. The get_l1 function is used to get the frequent itemsets of size 1 i.e.L1. The candidates for level K are generated using L(k-1) denoted by Ck, where k refers to the level.The join function sorts the input and then joins the sets if their first (K-2) elements are the same. The apriori_gen function tries to join every set in the Lk using the join function and prunes candidates using the has_infrequent_subset function and returns the Ck from L(K-1). The main apriori algorithm is implemented in an iterative manner. K is initialized as 2 and Lk is initialized as L1.

while the generated Lk is not empty, the algorithm generates the C(K) using the apriori_gen function. A Ck_count list is initiated to keep the support count of every candidate. The database is scanned for every level. For each transaction, the subsets of the transaction that are also candidates are found and their count is accumulated. Finally, the Lk is generated from all the candidates that are frequent and stored in a list for further processing.

Data Preparation and Algorithm Implementation (4 Points):

Hmine implementation:

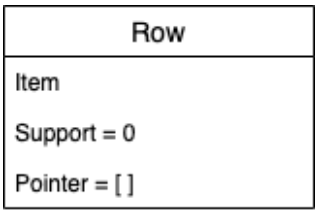

Fig 6. Row class

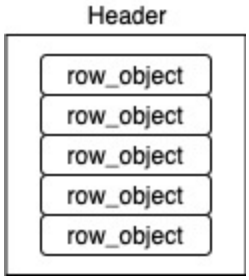

Fig 7. Header or Rowlist

We took a slightly different approach than the original algorithm for better efficiency. The database is scanned once to find the support count of every unique item and the items whose support count is greater than or equal to the minimum support are used to create the initial header. The header is a list of 'row objects(Fig.7). The row class has 3 attributes item, support and pointer where the pointer is a list of indexes (Fig.6). The database is scanned again to find the projected database. The projected database is stored in a single large python list with every projection separated by -1. For example,

Projected database :
Cell =[transaction -1 transaction -1]

```
['34', '62', '7', '36', '60', '40', '29', '52', '58', -1, '34', '62', '7', '36', '60', '40', '29', '52', '58', -1]
```

We have sorted the items on the projection according to their support count rather than the lexicographic order.This allows us to mine the most relevant frequent itemsets first.The pointer of row objects in the initial header is populated by traversing the frequent projection once, This is done by adding the indexes of the items in frequent projections to their respective pointer list.For example, given a projected database:

cell=[a,b,c,d,-1,c,d,e,-1]

The item pointers will be as follows:

a.pointer =[**0**]
b.pointer = [**1**]
c.pointer = [**2, 5**]
d.pointer= [**3, 6**]
e.pointer= [**7**]

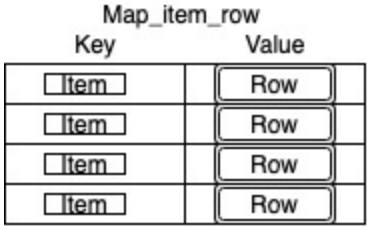

Fig 8. Map_item_row

This representation of the cells gives us an elegant way to implement dynamic adjustments of links without making the process computation heavy.The hmine function is called with three parameters namely Prefix(prefix of the frequent itemset, it is initially empty), prefix_length(Initially zero) and the initial header. The hmine function traverses each item in the header one by one. For each item in the header, it creates a new header that is initially empty and the variable called map_item_row(it maps the item to its row Fig.8) is cleared. For every item in the header, it traverses its pointer (which is the indexes of that item in the frequent projection called 'cell' in our program). It then increments the index by one and then goes into a while loop until it reaches the -1 i.e. the end of a frequent projection of a transaction.

Inside the while loop, It gets the item at that pointer and creates a row object of it in the map_item_row (Fig.8), if the item does not exist in it already, otherwise it increases its support count and appends the pointer list. After the pointer list of item in the row_list is traversed, their support and pointers are updated in map_item_row. Then, For every item in the map_item_row, the algorithm checks if their support is sufficient to be a frequent item. If it is a frequent item then their row object is appended in a new header. A buffer mechanism combined with prefix_length is used to output the frequent itemsets. Before recursively calling the hmine function, Again the new header is sorted on the basis of their support. The prefix , prefix_length +1 and the new header are passed as arguments for the recursive call to hmine. The algorithm works in a depth-first approach outputting the longest frequent itemsets for each prefix and then backtracking to move to the next prefix.

Data Preparation:

We used a combination of open source data obtained from 'Frequent Itemset Mining Dataset Repository', Market Basket Optimisation dataset from 'kaggle' and artificial data made with the help of python random function. This ensures that frequent items are generated for datasets with varied characteristics.

Dataset used:

Chess.txt from FIMI dataset repository
Total Unique Items: **75** Total Num of Transactions: **3196**

1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74
1 3 5 7 9 12 13 15 17 19 21 23 25 27 29 31 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74
1 3 5 7 9 12 13 16 17 19 21 23 25 27 29 31 34 36 38 40 42 44 46 48 50 52 54 56 58 60 62 64 66 68 70 72 74
1 3 5 7 9 11 13 15 17 20 21 23 25 27 29 31 34 36 38 40 42 44 47 48 50 52 54 56 58 60 62 64 66 68 70 72 74
….

market_basket _optimization dataset from kaggle
The only preprocessing required was to convert csv into .txt to ensure consistency in test datasets.
Number of unique items: **118** Total Num of Transactions: **7501**

shrimp almonds avocado vegetables_mix green_grapes whole_weat_flour yams cottage_cheese energy_drink
tomato_juice low_fat_yogurt green_tea honey salad mineral_water salmon antioxydant_juice frozen_smoothie
spinach olive_oil
burgers meatballs eggs
chutney
turkey avocado
….
Random_Test.txt
Generated using python random function.
Total Unique Items: **10** Total Num of Transactions:**200**

4 6 1 8 10
3 6 5 4 4
7 3 9 6 10
4 6 4 6 10...

Experimental results (4 Points):

The major comparison criteria are memory usage and time. These criteria are compared for both the algorithms in datasets having different characteristics.

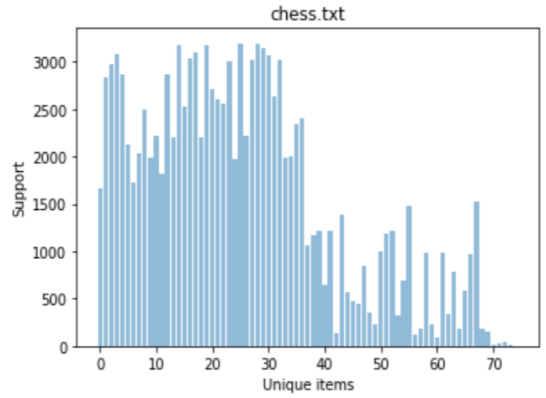

Fig. 9(a)Support count and unique items in chess.txt

Fig.10(a) Execution time vs support for chess.txt

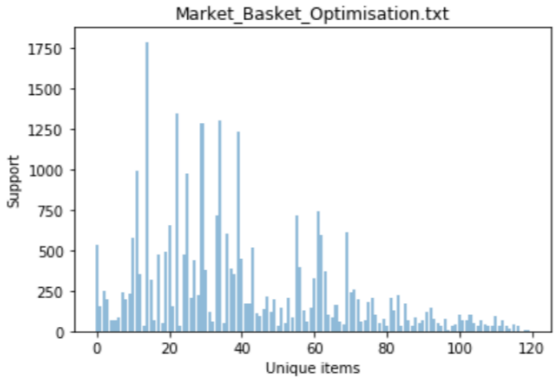

Fig. 9(b)Support count and unique items in Market_basket_optimization.txt

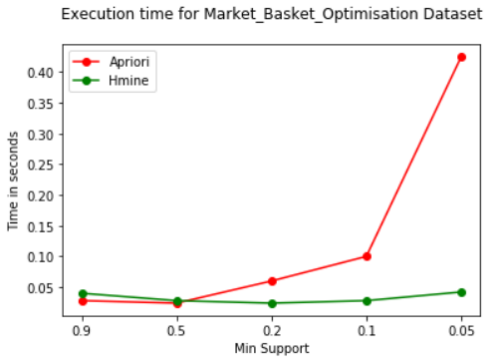

Fig.10(b) Execution time vs support for Market_basket_analysis.txt

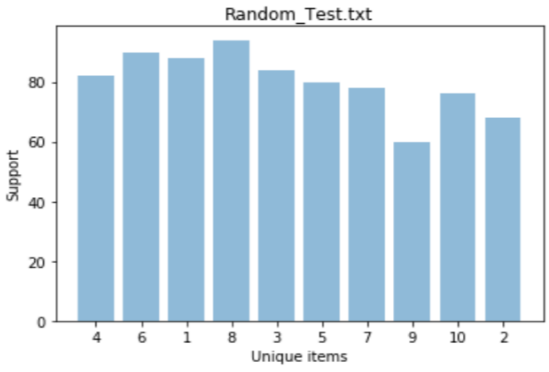

Fig 9(c)Support count and unique items in Random_test.txt

Fig.10(c) Execution time vs support for random_test.txt

Table(1). Data characteristics and execution time vs support graph for Hmine and Apriori.

The tests were performed in a computer with Intel Core i7-9750 CPU with 16 GB memory running Windows 10. All the reports of runtime take into account the time taken to generate Hstruct and frequent projections, CPU time and I/O time. We used the 'matplotlib' library in python to visualize the bar-chart, line graph and 'time' module to get the runtime. The Fig.9(a), 9(b) and 9(c) describes the absolute support and unique items in the datasets. It allows us to see the characteristics of the dataset and how the number of rows and number of frequent itemsets can affect the execution of algorithms. There is not much difference in the execution time when the number of complete frequent itemsets is low (High minimum support). However, when the number of frequent itemsets becomes large (Low minimum support) Hmine outperforms Apriori. Hmine is twice as fast in case of 10(c) and factors ahead in case of 10(b) when the dataset is large. For chess.txt which is quite dense and large, apriori fails to compute frequent itemsets in a reasonable amount of time. Whereas, hmine algorithm runs in a sufficient amount of time considering the large number of frequent Itemsets that are mined.

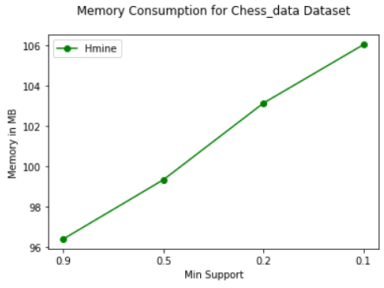

Fig.11(a) Memory(Mb) vs support for chess.txt

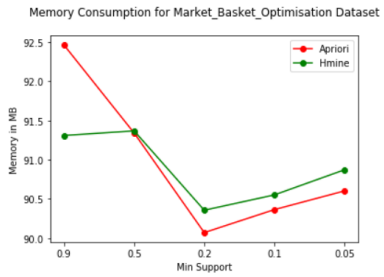

Fig.11(b) Memory(Mb) vs support for Market_Basket_optimization.txt

Fig.11(c) Memory(MB) vs support for random_test.txt

Table 2. Memory vs Support for all data sets

For computing the memory usage we used a built-it python 'psutil' module. The memory usage for small datasets like our artificial random dataset(200 transaction), the memory usage is equivalent and hence the graph in 11(c)Overlaps for both the algorithms. In market basket optimization dataset (7501 Transaction) The memory usage of Hmine algorithm is relatively less than the Apriori algorithm. In chess.txt,The memory usage is not reported for the apriori algorithm since, it failed to produce frequent itemsets in a reasonable amount of time.In the case of Hmine(mem) algorithm, The memory usage increases with decreasing support.The important point to be noted is that the deviation is only of 10 Mb even though frequent Projections of dataset becomes large and also the number of frequent itemsets increases. For sparse datasets,apriori works well since the most of the itemsets become frequent. Whereas, for dense dataset Hmine(mem) algorithm performs much better than apriori.

The aim of Hmine(mem) algorithm was to to find frequent itemsets of a partition of a huge dataset which can fit in main memory.According to the table above, although Hmine performs slightly better than apriori when both algorithms (unlike apriori for chess.txt) are able to produce results. The difference is marginal. The real benefit of Hmine is for large datasets where the number of itemsets meeting the threshold criteria is huge.

Conclusion (1 Point):

In this project, the version of Hmine called Hmine(mem) that is used to mine frequent itemsets of datasets which can fit in main memory is implemented. Hmine(Mem) is an interesting and challenging algorithm to learn and understand deeply. It proposes a new data structure called Hstruct to mine itemsets in just two scans of the database. It utilises Hstruct for it's advantage and dynamically adjusts links in the mining process.After various iteration, we finalised on the current implementation of hstruct and frequent projection which allows for automatic adjustment of links.According to the performance report, Hmine Works well on various kinds of data with different characteristics.The space overhead is close to stable and often predictable.It outperforms apriori in various settings. It is found to be very scalable, Another implementation is mentioned in the research paper which scales Hmine(Mem) to very large datasets.Our implementation can be made more space efficient by using a hash table and storing the keys in the frequent projections rather than the items themselves.

List of References (1 Point):

1. *SPMF: A Java Open-Source Data Mining Library*. Retrieved from http://www.philippe-fournier-viger.com/spmf/

2. Frequent Itemset Mining Implementations Repository. *Frequent Itemset Mining Implementations Repository*. Retrieved from http://fimi.uantwerpen.be/

3. Jiawei Han, Micheline Kamber, and Jian Pei. 2012. *Data mining: concepts and techniques*. Elsevier/Morgan Kaufmann.

4. Jian Pei, Jiawei Han, Hongjun Lu, Shojiro Nishio, Shiwei Tang, and Dongqing Yang. H-mine: hyper-structure mining of frequent patterns in large databases. *Proceedings 2001 IEEE International Conference on Data Mining*. http://doi.org/10.1109/icdm.2001.989550

5. Jian Pei, Jiawei Han, Hongjun Lu†, Shojiro Nishio, Shiwei Tang & Dongqing Yang (2007) H-Mine: Fast and space-preserving frequent pattern mining in large databases, IIE Transactions, 39:6, 593-605, DOI: 10.1080/07408170600897460

6. Xvivancos. 2020. Market Basket Analysis. *Kaggle*. Retrieved from https://www.kaggle.com/xvivancos/market-basket-analysis

7. Aman Jaiswal, 3 April 2020. Data Mining Assignment 4: Association rule mining using Apriori algorithm.