TALLINN UNIVERSITY OF TECHNOLOGY
Faculty of Information Technology
Department of Informatics
Chair of Foundations of Informatics

Realization of equivalence class based clustering

Master's thesis

Student:	Meelis Pruks
Student code:	121866IAPM
Supervisors:	Grete Lind, Rein Kuusik

Tallinn
2014

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

-- --

(*kuupäev*) (*allkiri*)

Annotatsioon

Käesoleva magistritöö eesmärgiks on realiseerida ekvivalentsiklassidel põhinev klasterdamisalgoritm ning võrrelda seda hetkel parima klasterdamisalgoritmiga ECCC. Ekvivalentsiklasside leidmiseks kasutatakse Grete Lind'i poolt välja pakutud algoritmi, mida võrreldakse hetkel parima algoritmiga samas vallas, DPMiner'iga, mida kasutab ECCC.

Põhiline töös käsitletud probleem on leida kiirem ja efektiivsem ekvivalentsiklassidel põhinev klasterdamisalgoritm. Selleks tuleks analüüsida DPMiner'it ja ECCC'd ning võrrelda DPMinerit G. Lind'i poolt välja pakutud algoritmiga.

Tulemuseks on uus ekvivalentsiklassidel põhinev klasterdamisalgoritm koos prototüübiga. Lisaks näitab analüüs, et kuigi G. Lind'i poolt välja pakutud algoritm on kaks korda aeglasem algoritmist DPMiner, on sellega leitud ekvivalentsiklassidega tehtav klasterdus 25% täpsem kui DPMineri poolt leitavate ekvivalentsiklassidega, sest DPMiner leiab ainult δ-discriminatiivsed ekvivalentsiklassid. Töös on ka näidatud, et kui kõik lähteandmed sisaldavad sama palju attribuute, mida G. Lind'i algoritm eeldab, kuid DPMiner mitte , on võimalik klasterdamisel saavutada oodatavalt 10-kordne keskmise kiiruse kasv.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 73 leheküljel, 8 peatükki, 14 joonist, 9 tabelit.

Abstract

The purpose of this thesis is to prototype an equivalence class based clustering algorithm and compare it against the current best algorithm for clustering, ECCC. The equivalence classes should be found using Grete Lind's algorithm which is compared against the best algorithm in the same field, DPMiner, used by ECCC.

The main problem addressed in the thesis is to find a faster and more effective equivalence class based clustering algorithm. In order to achieve that an analysis of DPMiner and ECCC along with a comparison of DPMiner and G. Lind's algorithm is made.

The result is a new equivalence class based clustering algorithm and its prototype. In addition to that, the thesis shows that even though the algorithm proposed by G. Lind is two times slower than DPMiner, the clusters created with its equivalence classes are 25% more accurate than the clusters created using DPMiner's equivalence classes because DPMiner finds only δ-discriminative equivalence classes. Additionally, the thesis shows that if each row contains the same number of attributes, which G. Lind's algorithm assumes and DPMiner not, it is possible to gain a 10 time increase in average clustering speed.

The thesis is in English and contains 73 pages of text, 8 chapters, 14 figures, 9 tables., etc.

Terms and Abbreviations

CS/CP	*Closed Set/Closed pattern*
EC	*Equivalence Class*
FP-Tree	*Frequency Pattern Tree*
DPMiner	*Discriminative Pattern Miner*
ECCC	*Equivalence Class based Clustering Algorithm for Categorical data*
CC	*Candidate Cluster*
ms	*Minimal Support Threshold/Minimal Frequency*

List of Figures

Figure 1: Source table with frequency table 16
Figure 2: Sub table with frequency table for Atr2 value 1 16
Figure 3: FP-Tree with header table [2] 25
Figure 4: Header table and FP-Tree for G. Lind's algorithms example data 27
Figure 5: Conditional FP-Tree and header table for value 22 27
Figure 6: Conditional header table and FP-Tree for generator {11,22} 28
Figure 7: Condition header table and full FP-Tree for generator {11,22} 29
Figure 8: Header table with FP-Tree containing class label distribution[2] 29
Figure 9: Speed test results for EC finding using mushroom dataset 32
Figure 10: Original DPMiner speed test result for EC finding with mushroom dataset[2] 32
Figure 11: Speed test result for finding equivalence classes using connect-4 dataset 33
Figure 12: Original DPMiner speed test result for EC finding with connect-4 dataset [2] 34
Figure 13: New ECCC algorithm speed test results on mushroom dataset 43
Figure 14: Original ECCC algorithm speed test on mushroom dataset [1] 43

List of Tables

Table 1: Frequency table showing existing closed set 17
Table 2: The source data used to build the FP-Tree [2] 24
Table 3: ECCC candidate clusters 38
Table 4: ECCC candidate clusters after final cluster removal 38
Table 5: Execution time distribution of new ECCC algorithm on mushroom dataset 44
Table 6: Execution time of new ECCC algorithm on connect-4 dataset 44
Table 7: Original ECCC to New ECCC accuracy comparison 45
Table 8: ECCCs comparison with multiple frequencies result 45
Table 9: Clustering quality results for new ECCC on connect-4 dataset 46

Table of Contents

1. Introduction 10

1.1 Background and Problem 10

1.2 Goal of Thesis 10

1.3 Methodology 11

1.4 Overview of the Thesis 11

2. Definitions 13

3. Equivalence Class Finding Algorithm 14

3.1 Algorithm 15

3.2 Algorithm Optimizations 17

3.3 Implementation 20

4. Current Fastest Equivalence Class Finding Algorithm 23

4.1 FP-Tree 23

4.2 DPMiner Algorithm 25

4.3 DPMiner Equivalence Class Finding 26

4.4 Comparison of DPMiner and the New EC Finding Algorithm 30

4.4.1 Complexity Comparison of EC Finding Algorithms 30

4.4.2 Speed Comparison of EC Finding Algorithms 31

5. Equivalence Class Based Clustering Algorithm 35

5.1 ECCC Algorithm 35

5.1.1 ECCC Algorithm Base 35

5.1.2 ECCC Algorithm 37

5.2 ECCC Optimizations 38

5.2.1 Quality Index Optimization 38

5.2.2 Sort Clusters by Quality Index 40

5.2.3 Implementation Based Optimizations 41

5.3 Comparison of Clustering Algorithms 42

5.3.1 Speed Comparison of Clustering Algorithms 43

5.3.2 Accuracy Comparison of ECCC and the New ECCC 44

6. Use of the New Clustering Algorithm's Application 47

6.1 Application Input 47

6.2 Application Output 47
7. Conclusions 49
7.1 Results 49
7.2 Future Research 50
7.3 Lessons Learned 50
8. Summary 51
Kokkuvõte 53
Bibliography 55
Appendix 1. Application Source Code 57

1. Introduction

We live in a time of information. Usually the quantity of the information gathered is too much for people to be able to analyze it. This means that it has to be reduced to smaller groups. This thesis will focus on a method to reduce the amount of data by grouping similar elements to compact data clusters for further analysis.

1.1 Background and Problem

The authors' main motivation to choose this topic was to learn more about data analysis and pattern finding because the amount of data gathered in our society is growing rapidly and data analysis will be one of the key components for future businesses.

The main problems associated with the clustering of data is that either it takes a very long time or the resulting clusters are not describing the data properly meaning that anything derived from them is not perfectly accurate.

The data grouping method used in this thesis is based on the finding of equivalence classes. Currently the leading equivalence class finding algorithms use a frequency tree to find their equivalence classes. However, Grete Lind proposed an alternative method without the use of trees which this thesis will analyze by comparing it against the current leading algorithms in the field.

The purpose of this thesis is to analyze the best equivalence class based clustering algorithm ECCC [1] and develop a new one derived from it by using G. Lind's algorithm for equivalence class finding, and use it to prototype a new clustering algorithm.

1.2 Goal of Thesis

The following is a list of goals for the thesis:

- Develop a clustering algorithm's prototype which clusters data based on equivalence classes using G. Lind's algorithm for finding the equivalence classes.

- Analyze the current fastest equivalence class finding algorithm DPMiner[2] and compare it to G. Lind's algorithm.

- Analyze the best equivalence class based clustering algorithm ECCC and compare it with the new clustering algorithm.

1.3 Methodology

The algorithm will be created in C++ which is then compared against DPMiner using the mushroom and connect-4 datasets. Comparison is done on two levels. First the complexity is compared which is followed by a speed comparison of the applications.

After that a new equivalence class based clustering algorithm is created using G. Lind's algorithm for equivalence class finding. The basis for the clustering algorithm will be the best equivalence class finding algorithm ECCC which uses DPMiner as its equivalence class finding component. Then the new clustering algorithm is compared against the current clustering algorithm by using the mushroom dataset.

The mushroom dataset was chosen because it was used in the original ECCC article [1] and the DPMiner article [2], and since it was unable for the author to get the ECCC algorithms application and reproduce the DPMiner's result, that was the only way to compare the algorithms. The connect-4 dataset was chosen to show the behavior of the new clustering application on a larger dataset as well.

1.4 Overview of the Thesis

The first section (see section 2 Definitions) focuses on giving an overview of the equivalence class clustering terminology used in the thesis.

This is followed by a description of G. Lind's algorithm, its implementation and some improvements (see section 3 Equivalence Class Finding Algorithm). After which an analysis of the current fastest equivalence class finding algorithm DPMiner is made along with a comparison against G. Lind's algorithm (see section 4 Current Fastest Equivalence Class Finding Algorithm).

Then the thesis focuses on the clustering by first analyzing the current best equivalence class based clustering algorithm ECCC and suggesting an alternative solution (see section 5 Equivalence Class Based Clustering Algorithm). The section also compares the resulting clusters created from the equivalence classes found by DPMiner and G. Lind's algorithm.

The second to last section (see section 6 Use of the New Clustering Algorithm's) describes the usage of the new equivalence class finding algorithm's implementation. And the final section (see section 7 Conclusions) will analyze the results from the previous sections based on which it will propose suggestions for any future research. The section also contains lessons learned from the creation of the thesis.

2. Definitions

This section will cover the main definitions used in this thesis.

Cluster – "*a cluster is a set of transactions* $C \subseteq D$ *satisfying* $C = f_D(cp)$ *for some closed pattern* cp *in dataset* D" [1].

Closed pattern / Closed set – "*an itemset* X *is closed in dataset* D *if there exists no proper super-itemset* Y *such that* $X \subset Y$ *and* $support(X) = support(Y)$ *in* D. X *is a closed frequent pattern in* D *if it is both closed and frequent in* D" [3].

Generator – "*an itemset* Z *is a generator in* D *if there exists no proper sub-itemset* Z' *such that* $Z' \subseteq Z$ *and* $support(Z') = support(Z)$" [4].

Equivalence class - "*an equivalence class* EC *is a set of itemsets that always occur together in some transactions of dataset* D. *That is, for all* $X \in EC$ *and* $Y \in EC$, $f_D(X) = f_D(Y)$, *where* $f_D(Z) = \{T \in D | Z \subseteq T\}$. *Frequent equivalence classes can be uniquely represented by a set of generators* G *and their associated closed frequent patterns* C, *in the form of* $EC = |G, C|$ "[2].

3. Equivalence Class Finding Algorithm

In order to find the clusters it is necessary to first find all the closed patterns with their respective generators and this section covers the algorithm created by Grete Lind (see section 3.1 Algorithm) along with some improvements (see section 3.2 Algorithm Optimizations). And the final section will cover the implementation specific description (see section 3.3 Implementation).

3.1 Algorithm

The following is the pseudo code of the new equivalence class finding algorithm.

Input:

X_0 is the initial data table;

ms is the minimal support threshold;

Description:

1. find_all_ECs()
   - a. $t \leftarrow 0$ /*recursion level*/;
   - b. $gen_0 \leftarrow \{\}$ /*generator*/;
   - c. find FT_0 /*frequency table for X_0*/;
   - d. $V_0 \leftarrow$ max integer;
   - e. findFCs(t, V_0);
2. find_ECs(t, V_t)
   - a. FOR EACH element $h_f \in FT_t$ with frequency V_{t+1}=min $FT_t[h_f] \geq ms\ and\ V_{t+1} < V_t$ DO /* element is of value h of attribute f*/
   - b. make_extract($t+1$, h_f, V_{t+1});
   - c. $FT_t[h_f] \leftarrow 0$;
   - d. END FOR;
3. make_extract(t, h_f, V)
   - a. $gen_t \leftarrow gen_{t-1} \cup h_f$;
   - b. $CS \leftarrow gen_t$;
   - c. $CS_is_new \leftarrow true$;
   - d. separate submatrix $X_t \subset X_{t-1}$ such that $X_t = \{X_{ij} \in X_{t-1} | X.f = h_f\}$;
   - e. find FT_t;
   - f. FOR EACH empty position p in CS DO
   - g. IF exists element h_p such that $FT_t[h_p] = V$ THEN
   - h. $CS \leftarrow CS \cup h_p$;
   - i. IF $FT_{t-1}[h_p] = 0\ THEN$
   - j. $CS_is_new \leftarrow false$;
   - k. EXIT FOR-cycle;
   - l. END IF;
   - m. END IF;
   - n. END FOR;
   - o. IF CS_is_new THEN
   - p. new_EC(V, CS, gen_t); /*Creates new EC*/
   - q. ELSE
   - r. add_gen(V, gen_t, CS_t);/*Adds generator to existing EC*/
   - s. END IF;

   - t. /*Zeroes Down*/
   - u. FOR EACH element $h_u \in FT_t$ with frequency > 0 DO
   - v. IF $FT_{t-1}[h_u] = 0$ THEN $FT_t[h_u] \leftarrow 0$;
   - w. END FOR;

   - x. find_ECs(t, V); /*Recursion*/

The algorithm starts off by creating a frequency table for each attribute value (Line 1.c). Afterwards it selects the element with the smallest frequency which is above or equal to the minimal support threshold (Line 2.a) and below the previously found frequency. Then selected frequency is marked as used (Line 2.c) (see Figure 1).The minimal support threshold or ms is the smallest frequency that is still considered as a result. It is defined by the algorithm executor. The final constraint in the line which defines that each frequency must be smaller than the previously found frequency is necessary because of the anti-monotonous property of the generator. "*The idea is that an itemset G is a generator if and only if subset(G; X) < subset(K; X) for every immediate proper subset S of K. Therefore, G can be identified as a generator, or filtered out, by just comparing the support of G with that of G's (immediate) subsets*" [2].

Souce data table				
Obj-nr	Atr1	Atr2	Atr3	Atr4
1	*1*	*3*	*1*	*2*
2	*1*	*3*	*1*	*1*
3	*3*	*1*	*2*	*1*
4	*1*	*2*	*1*	*2*
5	*3*	*2*	*1*	*1*

Frequency table				
Value	Atr1	Atr2	Atr3	Atr4
1	3	1	4	3
2	0	2	1	2
3	2	2	0	0

Figure 1: Source table with frequency table

The smallest element is then added to the current generator (Line 3.a) {-, 1, -, -} and a new subset (Line 3.d) and frequency table (Line 3.e) are created for each row whose attribute value equals the selected smallest element (see Figure 2).

Sub matrix Atr2:1				
Obj-nr	Atr1	Atr2	Atr3	Atr4
3	*3*	*1*	*2*	*1*

Frequency table Atr2:1			
Value	Atr1	Atr3	Atr4
1	0	0	1
2	0	1	0
3	1	0	0

Figure 2: Sub table with frequency table for Atr2 value 1

Having found the new frequency table, the algorithm then extracts any elements whose frequency equals the previously found smallest frequency to the closed set(Line 3.h). In this example the closed set would be {3, 1, 2, 1}.

When extracting the closed set it checks if the closed set has already been found (Line 3.i) because any already used elements on the same recursion level were already marked (Line 2.c) and previous marked elements have been propagated (Line 3.u). For example if we

extract the next smallest frequency from the original table (see Figure 1), Atr3 value 2, we see that the closed set {3, 1, 2, 1} created from the new frequency table contains an already existing value, Atr2 value 1, (see Table 1).

Table 1: Frequency table showing existing closed set

Frequency table Atr3:2			
Value	**Atr1**	**Atr2**	**Atr4**
1	0	1	1
2	0	0	0
3	1	0	0

If the extracted closed set is new (Line 3.o) a new equivalence class is created containing the closed set, generator and frequency (Line 3.p), otherwise the current generator is added to the existing equivalence class.

3.2 Algorithm Optimizations

The following is a list of improvements made for the algorithm to make it slightly faster.

1. In order to avoid any unnecessary computation the zeroes down cycle (Line 3.u) is combined with the minimal frequency finding cycle (Line 2.a).

```
FOR EACH element h_f ∈ FT_t with frequency V_{t+1}=min
FT_t[h_f] ≥ ms and V_{t+1} < V_t DO
        IF FT_{t-1}[h_f] ≠ 0 THEN
                make_extract(t + 1, h_f, V_{t+1});
        END IF;
        FT_t[h_f] ← 0;
END FOR;
```

2. It is also unnecessary to continue if the current frequency V is equal to or lower than the minimal support threshold. Because when the minimal frequency V_t is equal to ms then the check $ms \leq V_{t+1} < V_t$ (Line 2.a) will be false in every case. But the algorithm would waste time going through every element to verify that all of them fail.

```
IF V > ms THEN find_ECs(t, V); /*Recursion*/
```

3. Changed the closed set CS to be propagated to other recursion levels because otherwise the already filled CS from the previous recursion level (Line 3.f) will be lost

in later levels and it is recalculated again which means going through every element in the frequency table. As a result CS is found like $CS_t \leftarrow CS_{t-1} \cup h_f$.

4. Since the closed set CS is propagated to lower recursion levels a check to search only empty attributes in the CS can be added to minimal frequency finding (Line 2.a):

```
FOR EACH element $h_f \in FT_t$with frequency $V_{t+1}$=min $FT_t[h_f] \geq$
$ms\ and\ V_{t+1} < V_t$ and $CS_t[f] = 0$ DO
...
```

The resulting new algorithm will look like:

Input:

X_0 is the initial data table;

ms is the minimal support threshold;

Description:

1. find_all_ECs()
   a. $t \leftarrow 0$ /*recursion level*/;
   b. $gen_0 \leftarrow \{\}$ /*generator*/;
   c. $CS_0 \leftarrow \{\}$;
   d. find FT_0 /*frequency table for X_0*/;
   e. $V_0 \leftarrow$ max integer;
   f. findFCs(t, V_0);
2. find_ECs(t, V_t)
   a. FOR EACH element $h_f \in FT_t$ with frequency V_{t+1}=min $FT_t[h_f] \geq ms$ *and* $V_{t+1} < V_t$ and $CS_t[f] = 0$ DO
   b. IF $FT_{t-1}[h_f] \neq 0$ THEN
   c. make_extract($t + 1$, h_f, V_{t+1});
   d. END IF;
   e. $FT_t[h_f] \leftarrow 0$;
   f. END FOR;
3. make_extract(t, h_f, V)
   a. $gen_t \leftarrow gen_{t-1} \cup h_f$;
   b. $CS_t \leftarrow CS_{t-1} \cup h_f$;
   c. $CS_is_new \leftarrow true$;
   d. separate submatrix $X_t \subset X_{t-1}$ such that $X_t = \{X_{ij} \in X_{t-1} | X.f = h_f\}$;
   e. find FT_t;
   f. FOR EACH empty position p in CS_t DO
   g. IF exists element h_p such that $FT_t[h_p] = V$ THEN
   h. $CS_t \leftarrow CS_t \cup h_p$;
   i. IF $FT_{t-1}[h_p] = 0$ THEN
   j. $CS_is_new \leftarrow false$;
   k. EXIT FOR-CYCLE;
   l. END IF;
   m. END IF;
   n. END FOR;
   o. IF CS_is_new THEN
   p. new_EC(V, $\mathrm{gen_t}$, CS_t); /*Creates new EC*/
   q. ELSE
   r. add_gen(V, $\mathrm{gen_t}$, $\mathrm{CS_t}$);/*Adds generator to existing EC*/
   s. END IF;
   t. IF $V > ms$ THEN find_ECs(t, V); /*Recursion*/

3.3 Implementation

The program is implemented in C++ because it allows for high level performance improvements and memory management. The other reason for choosing C++ was that the other equivalence class finding and clustering algorithms are also written in C++.

Since it is very easy to create memory leaks with arrays in C++, the choice for standard library vectors were made instead of arrays. This of course meant a little performance hit of about 5% of CPU cycles during array initialization [5]. But accessing elements is the same speed.

When using regular multidimensional arrays the compiler actually changes the multidimensional arrays to a single array to avoid the look up time of internal arrays [6]. However, because of the use of vectors this needed to be managed in the implementation. As a result accessing element K of row L in table T looks like T[L*M + K] where M is the number of elements in each row.

Since all multidimensional arrays are converted to a single dimension it is necessary to find the number of different values in each attribute to be able to access the frequency table properly since not every attribute has the same number of different values. So in addition to the initial table and minimal support threshold the algorithm requires an additional input parameter containing the number of different values each attribute can have.

Another speed improvement made is that instead of accessing the array the regular way using the [] operand iterators were used which allow traversing through memory instead of doing a look up like the [] operand does for every element accessed.

Minimal frequency element finding (Line 2.a in "3.1 Algorithm") is implemented by using an array of data structures which combine attribute, value and frequency into one object. "*A data structure is a group of data elements grouped together under one name*" [7]. Afterwards the vector is sorted using the standard sort method of the C++ library with a complexity of $O(N * \log_2 N)$.[8] Frequencies are first sorted by frequency number then attribute order and finally attribute value order in ascending order.

Since the most time consuming operation in the whole algorithm is traversing through every line $N \times M$ where N is the number of rows and M the number of attributes, the sub matrix

(Line 3.d in “3.1 Algorithm”) and frequency table calculation (Line 3.e in “3.1 Algorithm”) were combined into one cycle.

Already found equivalence classes are stored in a hash table where the matching is done as follows:

1. First during initialization, a coefficient is calculated and stored for each attribute.

```
koef_0 ← 1;
t ← 1;
FOR EACH attribute f ∈ NA DO /*NA is a list with the number of different
values that each attribute can have*/
        koef_t ← f * koef_{t−1};
        t ← t + 1;
END FOR;
```

2. When hashing an equivalence class the hash code is calculated by summing every attribute multiplied by its coefficient. The resulting hash is unique for each equivalence class. However, since the hash code is stored in a 32 bit integer, the maximal hash code value is 2^{32}, afterwards it will start back at 0. Which means that if $N \ mod \ 2^{32} = K \ mod \ 2^{32}$ then $N = K$. The other limit of the hash code is the size of the hash table itself, as it is highly unlikely that the number of equivalence classes reaches $2^{32} \sim 4 * 10^9$.

```
hash ← 0;
FOR EACH h_f ∈ CS DO
        hash ← hash + h_f * koef_f;
END FOR;
```

3. When the hash codes for two equivalence classes are equal or are mapped to the same spot also known as a hash collision [9]. Then the frequencies along with the closed sets for both equivalence classes are compared. Even though the closed set comparison is enough the frequency comparison is a very good filter since in a large data set it is highly unlikely that equivalence classes share the same hash code and frequency which in turn spares from accessing the array.

For example when the closed set is {2, 1, -, 2} and the first column contains three possible values and every other column two then the hash code for the closed set would be $2 * 1 + 1 * 3 + 0 * 5 + 2 * 7 = 19$.

The tests show that on average the number of collisions is smaller than the number of closed sets which means that equivalence class insertion will not affect the runtime of the application.

4. Current Fastest Equivalence Class Finding Algorithm

As of writing this thesis the fastest equivalence class finding algorithm is DPMiner [2]. DPMiner is based on FP-Trees [10] which in order to understand DPMiner have to be covered first (see section 4.1 FP-Tree). Then the next section will cover the DPMiner base algorithm itself (see section 4.2 DPMiner) which is then followed by a detailed explanation of how it finds the equivalence classes (see section 4.3 DPMiner Equivalence Class Finding). And the final section compares the previously found algorithm (see section 3 Equivalence Class Finding Algorithm) with DPMiner (see section 4.4 Comparison of DPMiner and the New EC Finding Algorithm).

4.1 FP-Tree

The following is the FP tree construction algorithm.

Input:

A transaction database DB and a minimal support threshold ms.

Output:

Its frequent pattern tree, FP-tree

Method:

The FP-tree is constructed in the following steps.

1. Scan the transaction database DB once. Collect the set of frequent items F and their supports. Sort F in support descending order as L, the list of frequent items.
2. Create the root of an FP-tree, T , and label it as "null".
3. For each transaction Trans in DB do the following.
    a. Select and sort the frequent items in Trans according to the order of L. Let the sorted frequent item list in Trans be [p|P], where p is the first element and P is the remaining list.
    b. Call insert_tree([p|P], T). The function insert tree([p|P], T) is performed as follows.
        i. If T has a child N such that N.item-name = p.item-name, then increment N's count by 1.
        ii. Else create a new node N, and let its count be 1, its parent link be linked to T , and its node-link be linked to the nodes with the same item-name via the node-link structure. If P is nonempty, call insert tree(P, N) recursively.

[10]

It starts off by scanning the rows, which are called transactions, and creating a frequency table for each value. Afterwards it sorts each attribute, called item, of every transaction according

to the incremental order of frequency in the previously found frequency table. In case the frequency of the item is below the minimal threshold it is ignored. An example with a minimal frequency of three can be seen in the following table (see Table 2) [10][11].

Table 2: The source data used to build the FP-Tree [2]

ID	Items	Ordered items
1	f, a, c, d, i, m, p	f, c, a, m, p
2	a, b, c, f, l, m, o	f, c, a, b, m
3	b, f, h, j, o	f, b
4	b, c, k, s, p	c, b, p
5	a, f, c, e, l, p, m, n	f, c, a, m, p

Then it creates a tree [12] of all the items in a way that when traversing the tree to its leaf we end up with a transaction. And this is how it maps every transaction to the tree. The mapped items, called nodes, also contain the number of times a transaction was mapped to them [10] [11].

In addition to the tree, a header table is created which contains all the nodes for each item. The reason for that is since one item might be in different places for different transactions and thus would have a different prefix. This means that the different items would belong to separate nodes. The result will look like the following figure (see Figure 3)[10] [11].

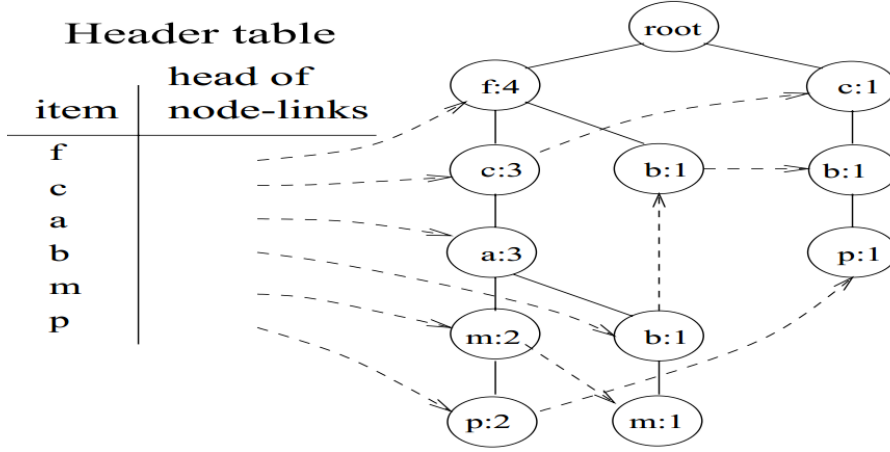


Figure 3: FP-Tree with header table [2]

4.2 DPMiner Algorithm

DPMiner stands for Discriminative Pattern Miner. It finds closed frequent patterns and frequent generators simultaneously to form equivalence classes. Given a dataset D, suppose D can be divided into various disjoint classes, denoted by $D = D_1 \cup D_2 \cup \ldots \cup D_n$. Let δ be the maximal threshold for each class of transactions (usually 1 or 2), ms be the minimal support threshold, and EC be a frequent equivalence class of D and C be a closed pattern of EC. An equivalence class EC is a δ-discriminative equivalence class provided that it's closed patterns C's support is greater than ms in D_i but less than δ in $D - D_i$ where $i \leq n$. Furthermore, EC is a non-redundant δ-discriminative equivalence class if and only if it is δ-discriminative and there exists no $\widehat{EC}$ such that $\hat{C} \subseteq C$, where $\hat{C}$ and C are the closed patterns of $\widehat{EC}$ and EC respectively [2].

"*DPMiner uses a modified FP-Tree structure. The normal FP-Tree stores all frequent items in descending frequency in the header table. DPMiner however excludes also those items whose frequency is equal to the number of transactions because they and those itemsets containing them are not generators due to the anti-monotone property of the generators. This modification reduces the header table size*" [2].

"*Given k non-empty classes of transactions —D_1, D_2, …, D_k, — , a minimal support threshold ms for each equivalence class and a maximal threshold δ for each class of transactions the method to discover equivalence classes for the k classes of transactions consists of the following 5 steps*" [2]:

1. Let $D = \bigcup_{i=1}^{k} D_i$
2. Construct a FP-tree based on dataset D and run a depth-first search on the tree to find frequent generators and closed patterns simultaneously. For each search path along the tree, the search terminates whenever a δ-discriminative equivalence class is reached.
3. For every frequent closed pattern X, determine the class label distribution. That is, find the class where a closed pattern has the highest support. This step is necessary because patterns are not mined separately for each $D_i (1 \leq i \leq k)$, but rather on the entire D.
4. Pair generators and closed frequent patterns to form δ-discriminative equivalence classes.
5. Output the non-redundant δ-discriminative equivalence classes where $EC = [G, C]$.

4.3 DPMiner Equivalence Class Finding

This section shows the equivalence class finding aspect of DPMiner as it is the core aspect of the algorithm. It is step 2 in the previous section.

DPMiner starts by first creating a FP-Tree (see section 4.1 FP-Tree). The following figure (see Figure 4) shows an header table with a FP-Tree created from the same data as G. Lind's algorithm example (see Figure 1) but since DPMiner does not allow duplicate values they are encoded to prefix plus the value of the attribute. For example, when the third attribute's value is one then it is encoded to 31 [2].

HeaderTable		
Order	Value	Frequency
1	31	4
2	41	3
3	11	3
4	42	2
5	23	2
6	22	2
7	13	2
8	32	1
9	21	1

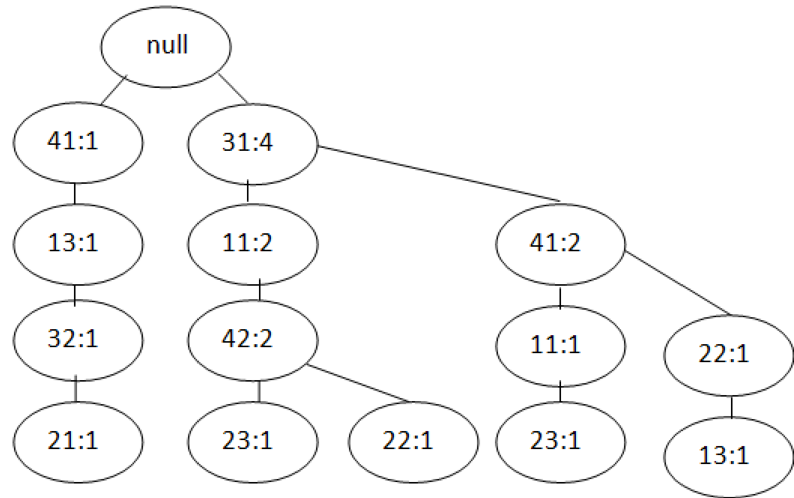


Figure 4: Header table and FP-Tree for G. Lind's algorithms example data

DPMiner then starts selecting elements one by one from the header table and traverses each of the nodes containing the value to the root. Every value found is added to a new conditional header table and also a new conditional FP-tree is created based on those values. The following figure (see Figure 5) demonstrates the conditional header table and FP-Tree the selected 22 with a frequency of 2 [2].

Conditional Header Table 22:2		
Order	Value	Frequency
1	31	2
2	11	1
3	41	1
4	42	1

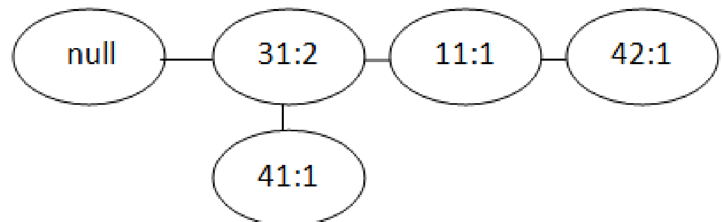


Figure 5: Conditional FP-Tree and header table for value 22

Whenever a value is selected from the header table it is added to the current generator. For example, if the value 11 is selected from this conditional header table then the generator for the next conditional header table would be {11, 22}. The current generator however consists of only the value 22 [2].

The frequency of the current equivalence class is equal to the frequency of the selected value which in this example is two [2].

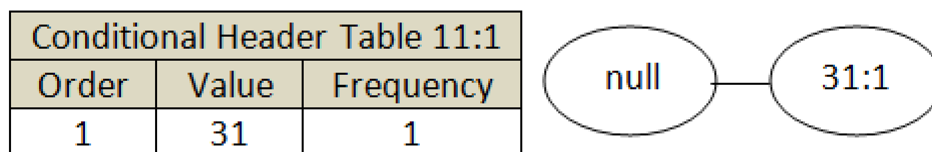
In order to find the closed set the header table is traversed and each element where the frequency is equal to the selected value's frequency is added to the closed set along with the generator. The order of the elements in the closed set is dictated by their order in the original

header table (see Figure 4). The elements added are also removed from the header table since they cannot be generators because of the anti-monotone property of generators. Values, whose frequency is lower than the minimal support threshold, are also removed from the header table [2].

The resulting closed set then hashed and a check is made whether it has already been found. If it is new then the equivalence class is inserted into the hash table which in this example consists of the generator {22}, the closed set {31, 22} and a frequency of two. If however the closed set has already been found then the new generator is added to the existing equivalence class [2].

The FP-Tree shown is actually only created when there exist values whose frequency is smaller than the selected element's frequency [2].

The current FP-Tree is correct, however this is usually not the case that a FP-Tree created from the conditional header table is complete. To demonstrate this we continue finding the next equivalence class by choosing the value 11 from 22's conditional header table (see Figure 5). The resulting conditional header table along with its FP-Tree can be seen in the following figure (see Figure 6).



Conditional Header Table 11:1		
Order	Value	Frequency
1	31	1

Figure 6: Conditional header table and FP-Tree for generator {11,22}

In order to get the correct closed set the FP-Tree needs to be appended with elements from the tail sub trees. The tail sub tree is the part of the previous FP-Tree which is after the selected element. In this example it does not matter, however not all the tail sub tree elements are selected but only elements whose frequency is equal to the specific branch frequency. The branch frequency is the frequency of the selected element in the branch. Using this example the branch {11, 31} frequency is one and the tail sub tree consists of the element 42 whose frequency is also one, which means it can be appended to the FP-Tree. The tail sub tree elements will not be included in the header table. The complete FP-Tree along with the header table can be seen in the following figure (see Figure 7) [2].

Conditional Header Table 11:1		
Order	Value	Frequency
1	31	1

Figure 7: Condition header table and full FP-Tree for generator {11,22}

The resulting equivalence class consists of a generator {22, 11}, a closed set {31, 11, 22, 42} and a frequency of one.

The complete algorithm however only finds δ-discriminative equivalence classes. An equivalence class is δ-discriminative when the value distribution among its sub classes is less than the maximal threshold for each class of transactions. And whenever a δ-discriminative equivalence class has been found the search terminates. Since in this example there is no class label distribution, meaning that every element is in the same class, the algorithm will not make any recursive calls like the previous 11 example. It will only find equivalence classes from the main header table like the 22 example. In order to illustrate the class label distribution we will use the following figure (see Figure 8) [2].

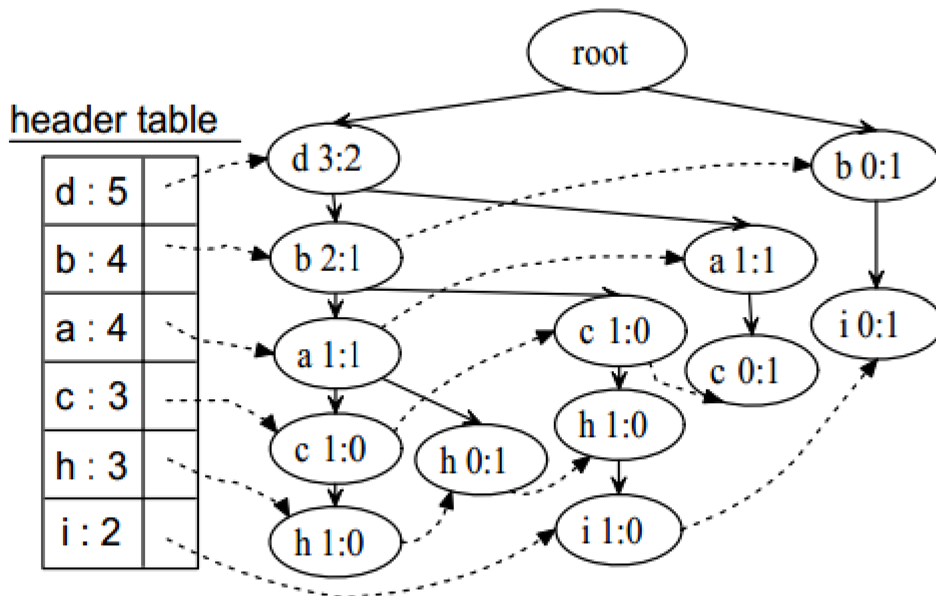


Figure 8: Header table with FP-Tree containing class label distribution[2]

In this example we can see that for example the element d has been distributed among two classes with a frequency of three elements in one class and two in the other.

The value distribution of an element can be calculated by subtracting the largest class frequency from the total frequency of the element. For example if the maximal threshold is

two and we have an element whose distribution among three classes is 1:1:6, then the search would not terminate because the distribution value $(1 + 1 + 6) - 6 = 2$ is not less than the maximal threshold of two [2].

4.4 Comparison of DPMiner and the New EC Finding Algorithm

This section compares the two algorithms by first comparing their complexity (see 4.4.1 Complexity Comparison) and then measuring the execution speed (see 4.4.2 Speed Comparison).

4.4.1 Complexity Comparison of EC Finding Algorithms

In computer science complexity comparison is done using the big O notation, which describes the worst case complexity for each step [13].

The two algorithms start with the same thing, which is scan the table and find the frequency of the values which is $O(N \times M)$ where N is the number of rows and M the number of attributes. Then they both sort the frequency table which is $O(V \times log_2 V)$ where the number of values in the frequency table is V.

The main difference between the two algorithms is in the data they are using. G. Lind's algorithm is meant for tables which have a fixed number of attributes in each row. DPMiner is mainly used for shopping cart like data. This means that the number of elements in each row is not fixed but is more of like either true or false depending on whether an attribute exists in the row or not. However, the fixed length rows can also be processed by DPMiner when the source data is presented in such a way that the same value does not appear in two different attributes. This means that identical values have to be encoded to be unique for each attribute. The result is that DPMiner operates on each attribute value separately and can eliminate values when their frequency is below the minimal support threshold.

When the initial analysis of the data is finished DPMiner requires only one more additional scan of the original data and afterwards it operates on the frequency values. G. Lind's algorithm however does not require this but it needs the initial data table to be scanned again for every equivalence class found which is $O(N_t \times M_t)$ where $N_t \subset N_{t-1}$ and $M_t \subset M_{t-1}$. $N_t \subset N_{t-1}$ and $M_t \subset M_{t-1}$ mean that in each step the number of rows and attributes is reduced based on the extracted attribute value. DPMiner on the other hand only requires the

scanning of each tree branch containing the element which is $O(K \times M_t)$ where $M_t \subset M_{t-1}$ is the number of attributes just like previously and K is the number of different branches that contain the current element. This in the worst case scenario is actually the number of rows containing the attribute in the case when every row in the original table is different and does not contain a similar prefix. And in the best case scenario it is M_t if the attribute is present only in one tree branch. On the other hand the worst case scenario for DPMiner is the complexity of G. Lind's algorithm. Another advantage that DPMiner has is that duplicate rows are absorbed in the table and will not cause additional data scanning.

When an equivalence class has been found both algorithms use a hash table to link already existing closed sets with generators. The difference here however lies in the fact that in G. Lind's algorithm it is already known if the closed set has been found and thus saves the time of checking the hash table if it is new or not.

In conclusion, DPMiner should be theoretically faster since for each equivalence class it does not need to traverse all the rows.

4.4.2 Speed Comparison of EC Finding Algorithms

The problem with comparing the two applications is that DPMiner terminates the search whenever a δ-discriminative equivalence class is found (step 2 in section 4.2). In addition to that, it only outputs δ-discriminative equivalence classes meaning that it is not possible to compare the resulting equivalence classes of the two algorithms. The result is that the two algorithms are not perfectly compatible.

Fortunately the DPMiner creators also created an alternative version called DPM-close of the same algorithm to mine only closed patterns in order to compare it to other algorithms and in the DPM-close algorithm they did not use the δ-discriminative constraint. The only downside is that the algorithm does not find the generators. As a result it does not need to do a look up in the equivalence class hash table in order to add a generator to an existing equivalence class. This is actually something where G. Lind's algorithm has an advantage since it does not need to do a lookup in the case where the closed pattern is new. But since the lookup operation is $O(1)$ for each equivalence class, it is not going to affect the result by a lot.

The following graph (see Figure 9) shows the execution speed of DPMiner-closed compared to the new EC finding algorithm.

The data used for the speed test is the well-known mushroom dataset which has 8124 rows and 22 attributes. The data describes the properties of the mushrooms and is used to determine which mushrooms are edible and which poisonous [14].

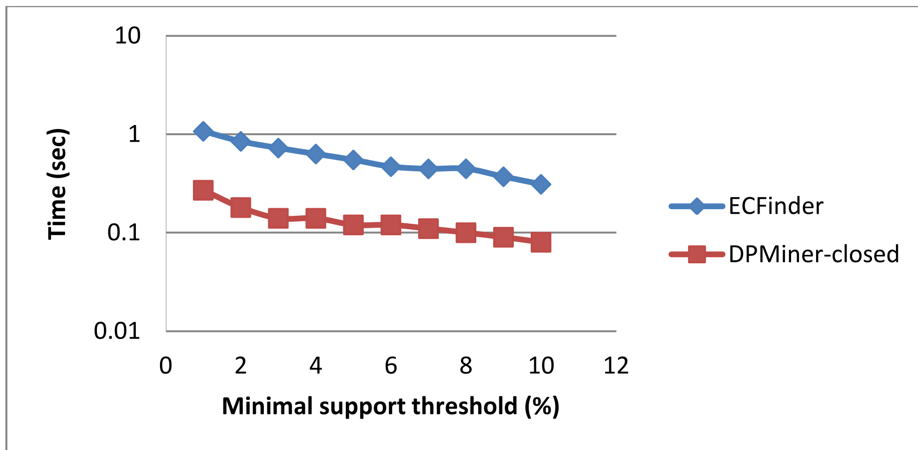


Figure 9: Speed test results for EC finding using mushroom dataset

In the figure we can clearly see that DPMiner is on average 70% faster as was predicted in the previous section (see 4.4.1 Complexity Comparison). This result however is not 100% accurate since DPMiner closed only finds the closed sets and not the generators.

Since the author was unable to repeat the test results of the main DPMiner program done in the original DPMiner speed test we'll use the original test results as a reference point (see Figure 10).

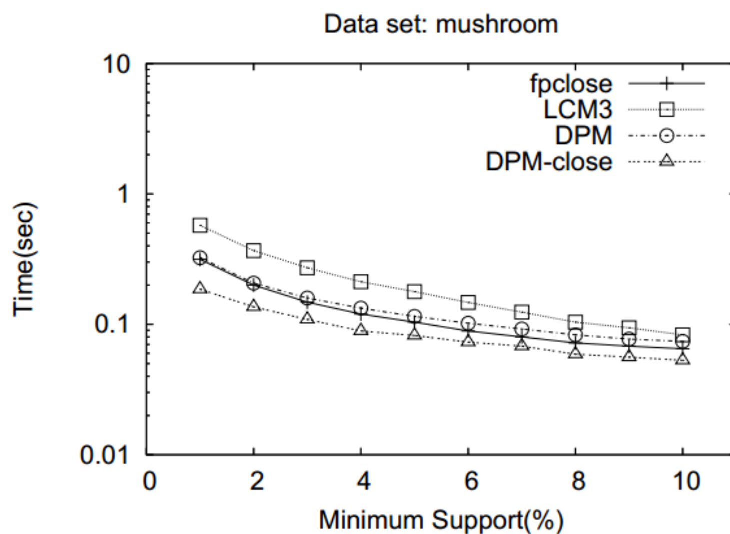


Figure 10: Original DPMiner speed test result for EC finding with mushroom dataset[2]

The DPM-close results from both graphs look roughly the same which means that we can derive that the main DPMiner algorithm DPM that also finds the generators is about two times faster than the new algorithm. It is however unclear what the exact parameters were for the DPM algorithm but judging from the execution times the result should be pretty accurate.

The next comparison was done using the connect-4 dataset [15] with 67558 rows and 42 attributes, which is a lot larger than the mushroom dataset (see Figure 11). This dataset contains all legal 8 positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced. So each attribute contains three possible values. Either it is player one's game piece, player two's game piece or blank.

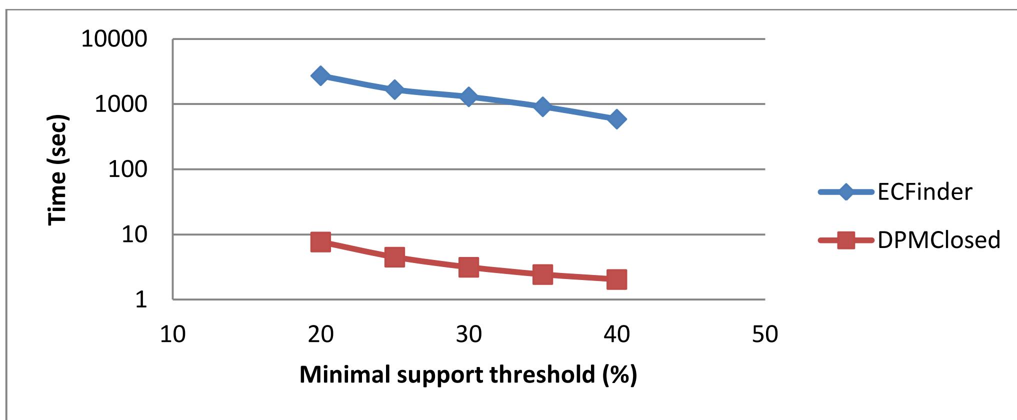


Figure 11: Speed test result for finding equivalence classes using connect-4 dataset

From here we can clearly see that when the amount of data increases the gap between the algorithms also increases. This is due to the fact that DPMiner does not have to traverse all the rows and can eliminate specific values which no longer fit the minimal support threshold and thus would not need to traverse any extra values.

And when comparing that to the original data from the DPMiner creators (see Figure 12) we can see that the main DPMiner algorithm is over 10 times faster.

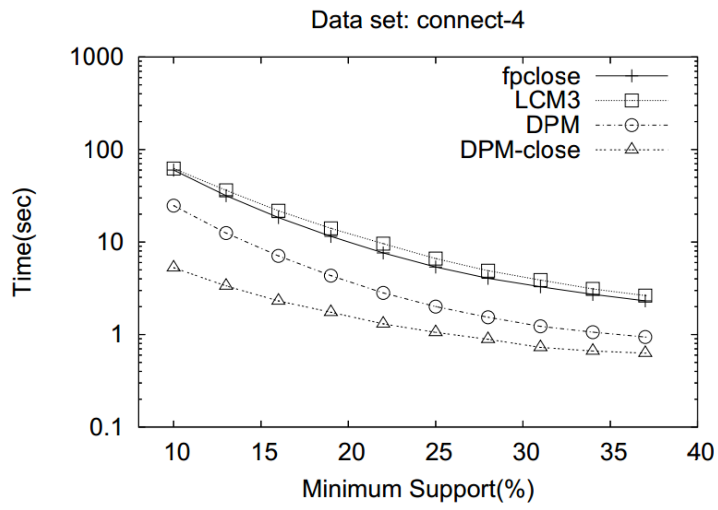


Figure 12: Original DPMiner speed test result for EC finding with connect-4 dataset [2]

5. Equivalence Class Based Clustering Algorithm

This section first describes the clustering algorithm ECCC which stands for Equivalence Class based Clustering Algorithm for Categorical data [1] (see section 5.1 ECCC Algorithm) which is currently the fastest clustering algorithm that uses equivalence classes. Afterwards a few optimizations are proposed in order to be able to combine the new EC finding algorithm with ECCC (see section 5.2 ECCC Optimizations). The final section compares the differences between the previous ECCC implementation and the new one (see section 5.3 Comparison of Clustering Algorithms).

5.1 ECCC Algorithm

This section first contains some base definitions in order to describe the ECCC algorithm (see section 5.1.1 ECCC Algorithm Base) and later describes the full algorithm (see section 5.1.2 ECCC Algorithm).

5.1.1 ECCC Algorithm Base

ECCC is currently the fastest clustering algorithm that uses equivalence classes. As a base it uses DPMiner to find the equivalence classes from which it then extracts the clusters [1].

First, each equivalence class is defined as a candidate cluster. "*There are often many candidate clusters, and only a few candidate clusters can become the final clusters. For example, with minimal support threshold of 5%, there are 9738 candidate clusters in the mushroom dataset.* [14] *So quality measures are needed to select the candidate clusters as final clusters. The algorithm uses factor's formula used by ECCLAT* [16] *and replaces the other factor's formula using a new one*" [1].

"*The Homogeneity Index* [16] *or HI of a candidate cluster* $CC(cs)$ *is defined by:*

$$HI_{CC}(cs) = \frac{|CC(cs)| * |cs|}{divergence(cs) + |CC(cs)| * |cs|}$$

Where $divergence(cs) = \sum_{t \in f_D} |t - cs|$ "[1].

For example if we have a closed set {3, 2, 4, -} which contains the rows {3, 2, 4, 1} and {3, 2, 4, 2} the homogeneity index would be $\frac{2\times3}{1+1+2\times3} = \frac{6}{8} = 0{,}75$

"*Homogeneity Index is used to measure the intra-cluster similarity. Larger values are better. Using this index, it prefers those candidate clusters whose closed patterns are very long. If a candidate cluster $CC(cs)$ has a very long closed pattern cp, then all the rows in $CC(cs)$ share all items in cs, implying that $CC(cs)$ is highly coherent. In this case $divergence(cp)$ is small and $HI_{CC}(cs)$ is large*" [1]. To demonstrate this we change the closed set from the previous example to {3, 2, -, -}. Then the homogeneity index would be $\frac{2\times2}{2+2+2\times2} = \frac{4}{8} = 0{,}5$ which is smaller than $0{,}75$.

For the inter-cluster diversity, they propose a measure called Discriminativeness Index or DI which for a candidate cluster $CC(cs)$ is defined as:

$$DI_{CC}(cs) = \prod_{g_i \in G(cs)} \left(1 + \frac{|cs - g_i|}{|cs|}\right)$$

Where $G(cs)$ is the set of generators in the equivalence class, $|cs|\ and\ |cs - g_i|$ are the number of attributes in the cs and $cs - g_i$ [1].

For example if we have an equivalence class consisting of a closed set {3, 2, 4, -} and the generators {3, -, -, -} and {-, 2, 4, -} then the Discriminativeness index would be $\left(1 + \frac{2}{3}\right) \times \left(1 + \frac{1}{3}\right) = \frac{20}{9} \approx 2{,}22$.

"*Larger $DI_{CC}(cs)$ values are better. Using this Discriminativeness Index, it prefers the candidate cluster which has a very long closed pattern and many short generator patterns*" [1].

Finally it combines the Homogeneity and Discriminativeness Index to form the Quality Index QI of the candidate cluster $CC(cs)$ which is defined as follows:

$$QI_{CC}(cs) = HI_{CC}(cs) \times DI_{CC}(cs)$$

This index is used to select candidate clusters with high Quality Index as the final clusters.[1]

5.1.2 ECCC Algorithm

The pseudo code of ECCC is given below:

Input:
X is a dataset to be clustered;
ms is the minimal support threshold;
Output:
CL is the set of Clusters;
$Trash$ is the set of trash transactions;
Description:
1. mine $CS \leftarrow \{cs_k | 1 \leq k \leq N\}, G(cs_k), CC(cs_k)$;
2. **FOR EACH** $cs \in CS$ **DO**
3. calculate $HI_{CC}(cs)$ and $DI_{CC}(cs)$;
4. $QI_{CC}(cs) \leftarrow HI_{CC}(cs) \times DI_{CC}(cs)$;
5. **END FOR;**
6. select $CC(cs^*)$, s.t $QI_{CC}(cs^*) = \max_{cs_k \in CS}\{QI_{CC}(cs_k)\}$;
7. $C(cs^*) \leftarrow CC(cs^*)$;
8. delete $CC(cs^*)$;
9. insert $C(cs^*)$ into CL;
10. **FOR EACH** $CC(cs)$ *and* $(CC(cs) \cap C(cs^*) \neq \emptyset)$ **DO**
11. $CC(cs) \leftarrow CC(cs) - C(cs^*)$;
12. **IF** $|CC(cs)| < ms$ **THEN**
13. delete $CC(cs)$;
14. **ELSE**
15. recalculate $HI_{CC}(cs)$, $DI_{CC}(cs)$ and $QI_{CC}(cs)$;
16. **END IF;**
17. **END FOR;**
18. repeat steps 6-17 until there are no candidate clusters;
19. classify the remaining transactions of X into the $Trash$;
20. return CL and $Trash$;

[1]

On dataset X, it first uses DPMiner algorithm [2] to mine the closed patterns and their generators simultaneously by using a minimal frequency threshold or ms. Then, it determines the candidate clusters of the frequent closed patterns which currently are equal to the equivalence classes, calculates the quality of each candidate cluster (see Table 3), and selects the candidate cluster $CC(cs^*)$ with highest quality as the final cluster $C(cs^*)$ which in the example is the cluster {3, 2, 1, 1}. When there are two highest quality candidate clusters, it prefers the candidate cluster with larger number of rows [1].

Table 3: ECCC candidate clusters

Frequency	Closed set	Generators	Rows	Quality index ($HI \times DI$)
1	{3, 1, 2, 1}	{-, 1, -, -} {-, -, 2, -}	{3, 1, 2, 1}	$1 \times 3{,}06 = 3{,}06$
2	{3, -, -, 1}	{3, -, -, -}	{3, 1, 2, 1} {3, 2, 1, 1}	$0{,}5 \times 1{,}5 = 0{,}75$
1	{3, 2, 1, 1}	{3, 2, -, -} {3, -, 1, -} {3, 2, -, 1}	{3, 2, 1, 1}	$1 \times 3{,}38 = 3{,}38$

For any remaining candidate cluster $CC(cs)$ such that $cs \neq cs^*$ and $CC(cs) \cap C(cs^*) \neq \emptyset$, it modifies the candidate cluster $CC(cs)$ as $CC(cs) = CC(cs) - C(cs^*)$. If $|CC(cs)| < ms$, it deletes the candidate cluster $CC(cs)$. Afterwards it recalculates $HI_{CC}(cs)$, $DI_{CC}(cs)$ and $QI_{CC}(cs)$ of the candidate clusters (see Table 4), and selects the cluster $CC(cs^*)$ with highest quality as the next final cluster which in this example is the closed set {3, 1, 2, 1} [1].

Table 4: ECCC candidate clusters after final cluster removal

Frequency	Closed set	Generators	Rows	Quality index ($HI \times DI$)
1	{3, 1, 2, 1}	{-, 1, -, -} {-, -, 2, -}	{3, 1, 2, 1}	$1 \times 3{,}06 = 3{,}06$
1	{3, -, -, 1}	{3, -, -, -}	{3, 1, 2, 1}	$0{,}5 \times 1{,}5 = 0{,}75$

The process above is repeated until there are no candidate clusters left. At the end, it classifies all remaining transactions into the trash set [1].

5.2 ECCC Optimizations

This section proves that it is possible to make the quality index calculation constant for a given candidate cluster (see section 5.2.1 Quality Index Optimization) which in turn allows reducing the number of candidate clusters traversed during clustering (see section 5.2.2 Sort Clusters by Quality Index and “5.2.3 Implementation Based Optimizations).

5.2.1 Quality Index Optimization

Because in G. Lind’s algorithm the data used has a fixed number of attributes, it is possible to reduce the complexity of ECCC by not having to recalculate the Quality Index of each candidate cluster whenever a final cluster is found.

First we take the initial divergence formula of the Homogeneity Index:

$$divergence(cs) = \sum_{t \in f_D} |t - cs|$$

Since each row has M attributes and the closed set length is $|cs|$ we can reduce the divergence formula to:

$$divergence(cs) = |CC(cs)| * (M - |cs|)$$

Where $|CC(cp)|$ is the number of rows in the candidate cluster CC and $M - |cs|$ the number of attributes missing from the closed set.

Now if we replace this in the original Homogeneity Index formula:

$$HI_{CC}(cs) = \frac{|CC(cs)| \times |cs|}{divergence(cs) + |CC(cs)| \times |cs|}$$

The result will be:

$$HI_{CC}(cs) = \frac{|CC(cs)| \times |cs|}{|CC(cs)| \times (M - |cs|) + |CC(cs)| \times |cs|}$$

Since the frequency of the candidate cluster $|CC(cs)|$ appears in each multiplication it cancels out and as a result the equation will be:

$$HI_{CC}(cs) = \frac{|cs|}{(M - |cs|) + |cs|}$$

In the denominator the number of attributes of the closed set $|cs|$ also cancels out which makes the final result look like:

$$HI_{CC}(cs) = \frac{|cs|}{M}$$

Where $|cs|$ is the number of attributes in the closed set and M the number of attributes in the data set X.

The resulting Homogeneity Index formula is constant to the size change of the candidate cluster. This means that it only needs to be calculated once for each cluster. And since the

Discriminativeness Index formula is also constant because the closed set and generators do not change after they have been found, we can derive that with this change of the Homogeneity Index the Quality Index will be constant to the change of the cluster size as well. This in turn means that it is not necessary to traverse the rows of each candidate cluster in order to determine the quality index which saves a lot of computational time since in the original version of the algorithm it had to be done every time a final cluster is found.

5.2.2 Sort Clusters by Quality Index

When the Quality index is constant rows 6-18 of the algorithm can be changed to:

6. sort CC-s based on QI in descending order;
7. **FOR EACH** $CC(cs_t)$ **DO**
8. select $CC(cs^*)$ where $|CC(cs^*)| = \max_{cs_k \in CS}\{|CC(cs_k)|\}$ and $QI_{CC}(cs^*) = QI_{CC}(cs_t)$;
9. $C(cs^*) \leftarrow CC(cs^*)$;
10. delete $CC(cs^*)$;
11. insert $C(cs^*)$ into CL;
12. **FOR EACH** $CC(cs)$ and $(CC(cs) \cap C(cs^*) \neq \emptyset)$ **DO**
13. $CC(cs) \leftarrow CC(cs) - C(cs^*)$;
14. **IF** $|CC(cs)| < ms$ **THEN**
15. delete $CC(cs)$;
16. **END IF;**
17. **END FOR;**
18. **END FOR;**

Instead of comparing each candidate cluster in order to find the maximal cluster which is $O(N \times C)$ where N is the number of candidate clusters and C the number of final clusters we only need to sort the candidate clusters by quality index which is $O(N * log(N))$. Then we can directly traverse through the candidate clusters which has a complexity of $O(N)$.

Now in order to get the final cluster a check needs to be made if any clusters with the same quality index might have a higher frequency but since the candidate cluster list was already sorted the possible final clusters must be one of the following candidate clusters. Of course the worst case scenario in which every cluster has the same Quality index means that each remaining cluster has to be traversed but this is highly unlikely as the quality index depends on the size of the closed set and its generators. Which in turn means it can be reduced to $O(1)$.

Another change that was made was the removal of the quality index calculation from the internal for cycle. This saves a lot of computational time because for each final cluster found

the algorithm would no longer have to traverse each row of all the connected clusters to find Quality Index.

5.2.3 Implementation Based Optimizations

The bottle neck in this algorithm is still the inner for cycle that was already trimmed in the previous section. The operation in question is reducing the frequency of the candidate clusters connected to the final cluster.

First of all, in order to get the connections, the rows contained in the closed set would need to be known. This requires the traversal of the original data set at some point since the equivalence class finding algorithms only return the closed set attributes. In addition to that, the knowledge of which other closed sets are connected to the same row is also needed.

However this is something where speed can be traded for memory. If there is enough space it is possible to store all inter closed set connections when finding them with the equivalence class finding algorithm and then this operation is very quick. But in a large enough dataset this is not possible.

With these changes in mind the algorithm from the previous section (see section 5.2.2 Sort Clusters by Quality Index) would change to:

6. sort CC-s based on QI in descending order;
7. **FOR EACH** $CC(cs_t)$ **DO**
8. calculate $CC(cs_t)$ new frequency based on still active rows in X;
9. **IF** $|CC(cs_t)| < ms$ **THEN**
10. delete $CC(cs_t)$;
11. continue for;
12. **END IF;**
13. calculate new frequency for $CC(cs_k)$ where $k > t$ and $QI_{CC}(cs_k) = QI_{CC}(cs_t)$
14. select $CC(cs^*)$ where $|CC(cs^*)| = \max_{cs_k \in CS}\{|CC(cs_k)|\}$ and $QI_{CC}(cs^*) = QI_{CC}(cs_t)$;
15. $C(cs^*) \leftarrow CC(cs^*)$;
16. delete $CC(cs^*)$;
17. insert $C(cs^*)$ into CL;
18. mark $C(cs^*) \cap X$ as used;
19. **END FOR;**

The changes made:

1. (Line 8) For each potential cluster the frequency is recalculated by checking how many rows have been used.

2. (Line 9-12) The algorithm still needs to check if after the new frequency has been recalculated it is over the minimal support threshold. Otherwise it removes the candidate cluster.

3. (Line 13) For each candidate cluster whose quality index is the same as the currently selected cluster a new frequency needs to be calculated in order for the frequency comparison in the next line (line 14) to be up to date.

4. (Line 18) Instead of updating every cluster connected to the current cluster, only the initial data set is updated by marking the extracted rows as used.

5. Actually as a side note it is not even necessary to delete the candidate clusters since the clusters are sorted and when a cluster has been passed it is no longer checked again. Deletion usually means shifting all the elements in the right one position to the left.

The result is that the updating of every cluster, with an average complexity of $O(N * \overline{CC})$ where N is the number of rows and $\overline{CC}$ the average number of candidate clusters connected to each row, has been replaced by traversing each row of the cluster, which has a complexity of $O(N * C)$ where N is the number of rows and C the number of unused candidate clusters with the same quality index.

5.3 Comparison of Clustering Algorithms

There is no need to compare the complexity of the two algorithms as that has already been done in the previous sections. As a result this section focuses on comparing the actual implementations. First in order to determine if the improvements have had an effect a comparison of the runtime of the two algorithms is made (see section 5.3.1 Speed Comparison). After that a quality comparison is made to check whether clusters created from the equivalence classes of G. Lind's algorithm will have a better result than the equivalence classes of the DPMiner algorithm (see section 5.3.2 Accuracy Comparison of ECCC and the New ECCC").

5.3.1 Speed Comparison of Clustering Algorithms

This section uses the mushroom dataset to compare the speed of the two algorithms.

The following figure (see Figure 13) shows the amount of time it takes the new ECCC algorithm to find all the clusters from the mushroom dataset. Since the original ECCC algorithm implementation could not be acquired we will use the graph from the original ECCC article (see Figure 14). In order for the two algorithms to be comparable the mushroom dataset has been divided into 1K to 8K rows containing datasets for the new ECCC speed test just like in the original test.

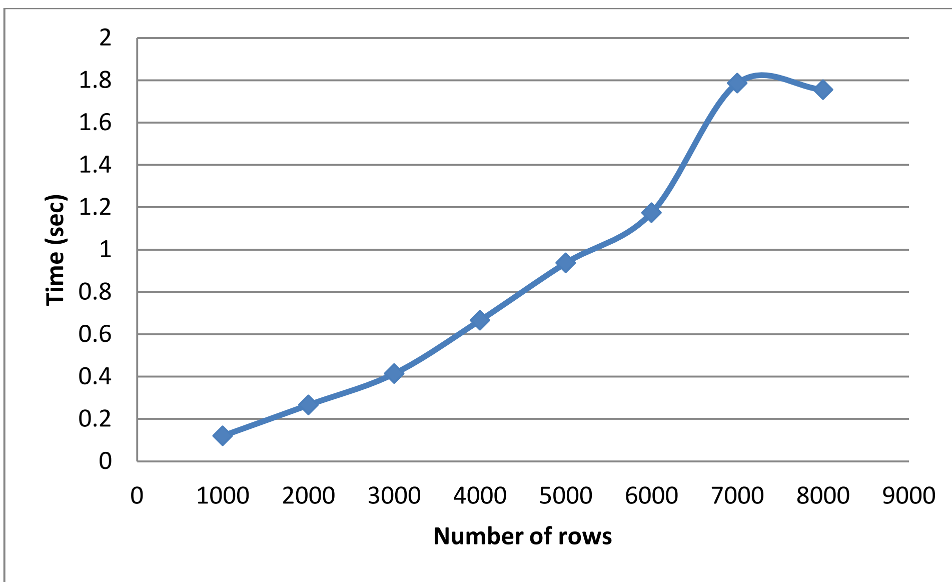


Figure 13: New ECCC algorithm speed test results on mushroom dataset

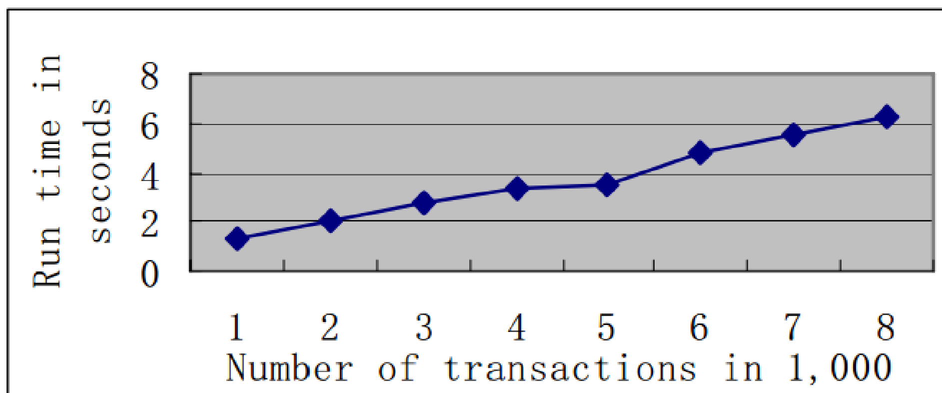


Figure 14: Original ECCC algorithm speed test on mushroom dataset [1]

The results show that even though the equivalence class finding algorithm is five times slower than DPMiner (see 4.4.2 Speed Comparison) the new ECCC algorithm is four times faster.

When looking at the execution time distribution (see Table 5) and the execution time difference of DPMiner to the new EC finding algorithm (see 4.4.2 Speed Comparison) we can conclude that the new ECCC is possibly 10 times faster than the old ECCC algorithm.

Table 5: Execution time distribution of new ECCC algorithm on mushroom dataset

#Rows	EC finding time (ms)	Clustering time (ms)
1000	113	7
2000	229	37
3000	363	51
4000	586	80
5000	828	109
6000	1062	112
7000	1645	141
8000	1611	144

The following table (see Table 6) will show the clustering time of the connect-4 dataset in order to demonstrate that the new algorithm is capable of processing larger amounts of data.

Table 6: Execution time of new ECCC algorithm on connect-4 dataset

Minimal frequency (%)	EC finding time (sec)	Clustering time (sec)	#ECs
40	523	231	175 340
35	642	133	234 829
30	1103	484	329 546
25	1581	232	565 934
20	2321	483	1 146 096

The table shows that the new clustering algorithm is capable of processing larger amounts of data with a stable execution time.

5.3.2 Accuracy Comparison of ECCC and the New ECCC

This section uses the mushroom data set to compare the algorithms quality against each other.

The following table (see Table 7) shows the clusters created using the mushroom dataset and a minimal support threshold or minimal frequency of 4%.

Table 7: Original ECCC to New ECCC accuracy comparison

Cluster No.	New ECCC		Original ECCC	
	#Poisonous	**#Edible**	**#Poisonous**	**#Edible**
1	0	768	0	576
2	0	864	432	0
3	864	0	0	384
4	0	432	0	384
5	0	432	864	0
6	648	0	576	0
7	648	0	576	0
8	432	0	576	0
9	0	512	0	400
10	448	0	0	400
11	256	**96**	272	**96**
12	288	**48**	**72**	528
13	0	480	**128**	384
14	200	**192**	**0**	384
15	**72**	272	**144**	384
16	0	0	240	**96**
Trash	**60**	112	**36**	192
Error	**468**		**572**	

The result is that the new ECCC which uses G. Lind's algorithm will find 15 clusters instead of 16 and it has 25% less errors. The number of errors is determined by the number of elements in the smaller subset highlighted in bold.

To further compare the two implementations we use the same 10 different frequencies from 1% to 10% used by the ECCC authors in order to determine if the previous result is correct [1]. The results are shown in the following table (see Table 7).

Table 8: ECCCs comparison with multiple frequencies result

minfr(%)	New ECCC		Original ECCC	
	#Clusters + trash cluster	#Errors	#Clusters + trash cluster	#Errors
1	28+1	72	28+1	252
2	22+1	236	24+1	252
3	18+1	332	20+1	172
4	15+1	468	16+1	572
5	13+1	476	12+1	890
6	11+1	596	11+1	890
7	10+1	620	11+1	890
8	9+1	620	6+1	890
9	8+1	876	6+1	890

minfr(%)	New ECCC		Original ECCC	
	#Clusters + trash cluster	#Errors	#Clusters + trash cluster	#Errors
10	8+1	876	5+1	890

The table shows that the error rates change with different frequencies. But the overall trend indicates that the result by the new algorithm is more accurate with the exception of a frequency of 3%. The best result was with a frequency of 1% where the error rate was 0.0088 whereas the best result for the original algorithm was only with an error rate of 0.0212. This is a two fold increase in accuracy. We can also see that the error rate and cluster number change is steady and monotonous for the new algorithm whereas in the previous algorithm when the frequency drops from 7% to 8% the number of clusters halves.

The average clustering error rates for both algorithms are 0.0637 for the new ECCC and 0.0811 for the old algorithm. This indicates that the new algorithm is about 25% more accurate as was also seen in the previous table (see Table 7).

Next up we will test if the error rates stay stable for larger amounts of data. We will use the same connect-4 setup as in the speed tests (see Table 6). The following table (see Table 9) shows the error rate, number of clusters plus trash cluster and number of equivalence classes found for each frequency percentage.

Table 9: Clustering quality results for new ECCC on connect-4 dataset

Minimal frequency (%)	#ECs	#Clusters	Error rate
40	175 340	2+1	0.0419
35	234 829	2+1	0.0649
30	329 546	3+1	0.0177
25	565 934	3+1	0.0471
20	1 146 096	4+1	0.0568

The table clearly indicates that the error rate stays stable even when the amount of equivalence classes changes. In fact the result is actually better than for the mushroom dataset with an average error rate of 0.0457 compared to 0.0637.

6. Use of the New Clustering Algorithm's Application

This section describes the use of the new ECCC algorithm's application by first describing its input (see section 6.1 Application Input) and then the output (see section 6.2 Application Output).

6.1 Application Input

The application input consists of the following parameters:

- Source file path
- Minimal support threshold - the minimal size of the cluster still considered as a result.
- Output file prefix - since there are multiple output files with different suffixes e.g. ".cluster", ".closed" etc.

In addition to that there are some optional parameters:

- c - if the source file contains a class label distribution
- A filter to select only specific attributes for analysis e.g. "1-3,14,15" will only use the attributes 1, 2, 3, 14, 15 for analysis. The parameter should be comma separated with no spaces.

The input file is a space separated table of integers. If the class label distribution parameter was provided then the first column in the table is considered the class label for the row. The class label can be a non-integer string; all the other elements should be integers.

6.2 Application Output

The output of the algorithm is divided into six files. Whenever an attribute value is missing from a column it is marked by a "-" sign:

- [prefix].closed – is used to display the closed sets. The file consists of the following columns: the number of attributes in the closed set, the attribute values, and the

frequency of the closed set. An example row from a closed set with a frequency of 1004 and two of the four possible attributes defined: “4 29 - 12 - 1004”

- [prefix].key – contains the generators. The file consists of the following columns: the number of attributes in the generator and the generator attributes values. An example row for a generator with two of the three values set would look like: “3 - 44 89”

- [prefix].pair – matches the generators to their closed sets. The first number is the index of the generator in the “.key” file and the second number is the index of the closed set in the “.closed” file.

- [prefix].fullpair – is an alternative method to combines all the previous files into one single file. The previous three files used to compare the results with DPMiner which has a similar output format. In this file columns of each row are: the frequency of the generator, the generator number “G#” e.g. G24, the generators non-empty values in the form of “V[attribute #].[value]”. In case the generator and closed set do not match “ZeroF” is added followed by the values that are included in the closed set but not the generator. The difference is in the same format as the generator values. An example row for an equivalence class with a frequency of four, a generator with the second attribute value of one and a closed set with the second attribute one and third attribute two, would look like “4 G1 V2.1 ZeroF V3.2”.

- [prefix].cluster – contains the final clusters. The file consists of the following columns: the number of attributes, the attributes of the cluster and the frequency of the cluster. On top of that if the input file contained a class label then a class label distribution for each cluster is also displayed. The last row will always be a trash cluster with no attributes. The trash cluster contains the remaining data that was not distributed to the clusters. An example row for a cluster with two of the three attributes defined and a frequency of 4000 distributed among two classes 1500 and 2500 looks like: “3 12 – 44 4000 1500 2500”

- [prefix]-date-time.rowClusterPair – contains the row cluster distribution. The file contains the data of the input file with the addition of the last column being the index of the cluster in the “.cluster” file. The date and time are added to allow this file to be used as an input file.

7. Conclusions

This section first analyses the results of the thesis (see 7.1 Results), then suggests possible future research (see 7.2 Future Research) and finally contains some lessons learned (see 7.3 Lessons Learned).

7.1 Results

First up the comparison of the new equivalence class finding algorithm to DPMiner (see section 4.4 Comparison of DPMiner and the New EC Finding Algorithm) shows that, even though it was not possible to directly compare the two algorithms, DPMiner is at least two times faster due to the fact that it only uses the frequency values instead of the initial rows for equivalence class finding.

The analysis of the clustering algorithm ECCC showed that it was possible to make the quality index calculation constant throughout the clustering process in case the number of attributes for each row is fixed (see section 5.2.1 Quality Index Optimization). This in turn allowed the algorithm to be optimized (see section 5.2 ECCC Optimizations) to the point where it was four times faster than the previous algorithm (see section 5.3.1 Speed Comparison of Clustering Algorithms).

Finally the quality comparison of the clusters showed that the clusters created with the new clustering algorithm were 25% more accurate than the ones created with the old algorithm (see section 5.3.2 Accuracy Comparison of ECCC and the New ECCC). The reason was that DPMiner restricts the found equivalence classes to δ-discriminative equivalence classes (see step 2 in section 4.2 DPMiner Algorithm). It is however unclear how much this affected the speed results as the author was unable to replicate the speed test results from the original DPMiner experiments (see Figure 10) using the main DPMiner implementation. The fact is that removing the δ-discriminative constraint will make the algorithm slower as it will need to traverse deeper into the FP-Tree as was seen in the DPMiner implementation example (see 4.3 DPMiner Equivalence Class Finding).

7.2 Future Research

Considering the fact that the equivalence class finding algorithm was at least two times slower than DPMiner in the mushroom dataset and the clustering algorithm four times faster it can be concluded that it is possible to create an algorithm that is close to 10 times faster by using the DPMiner algorithm without the δ-discriminative component combined with the new clustering algorithm for fixed attribute count datasets.

7.3 Lessons Learned

The algorithm implementation was created using a test driven development style [17] which helped a lot as it is otherwise quite easy to break some part of the algorithm without realizing. And when the bug would finally be discovered it would be very difficult to track down the origin of it. But the unit tests kept the amount of time wasted on bug finding at a minimum. Sadly due to time constraints and last minute changes some parts of the final application were not covered with tests.

The use of the Git version control system [18] when writing the prototype helped out a lot. As it was possible to easily try new approaches, switch between different implementations and quickly dismiss some methods with the use of different branches.

The full applications source code repository is publicly available at https://bitbucket.org/meelispr/magistritoo-c.git.

8. Summary

The goal of the thesis was to create an equivalence class based clustering algorithm, using Grete Lind's algorithm for equivalence class finding, and comparing the clustering algorithm against the current fastest algorithm in the field, ECCC, which uses DPMiner for equivalence class finding.

The following is a list of results of the thesis:

- A comparison of G. Lind's algorithm against the current best equivalence class finding algorithm DPMiner. The comparison shows that DPMiner is at least two times faster since it does not traverse through the data set but instead uses a frequency based tree structure, FP-Tree. The tree enables element based reduction which gives DPMiner a huge advantage in larger data tables.

- An analysis of ECCC which shows that, since DPMiner only returns δ-discriminative equivalence classes, the resulting clusters are 25% less accurate than the clusters created from the equivalence classes returned by G. Lind's algorithm.

- A new clustering algorithm for data where each row contains the same number of attributes. The algorithm is potentially 10 times faster than ECCC which is meant for varying number of attributes.

- The prototype of the new clustering algorithm which contains G. Lind's algorithm for equivalence class finding and is meant for equal size rows. The algorithm is four times faster and 25% more accurate from ECCC on the mushroom data set.

The main conclusions of the thesis are:

- If the δ-discriminative constraint is removed from DPMiner then the clusters created from its equivalence classes would be at least 25% more accurate. The downside is that the algorithm will be slower than the current DPMiner algorithm.

- If every row contains the same number of attributes then it is possible to achieve a 10 times increase in clustering speed with the proposed clustering algorithm. Assuming

that the modified DPMiner from the previous point is used for equivalence class finding.

The result however is not perfectly accurate because using the DPMiner's main algorithm it was not possible to create the same results as in the original article. However, using DPMiner's closed version it was possible to compare the algorithms indirectly. In addition to that it was not possible to get the implementation of the original ECCC algorithm from its creators. But since the graphs from the DPMiner test matched pretty accurately and the ECCC algorithm was created only a couple of years ago, we can conclude that the error is pretty small.

For future research the new clustering algorithm's prototype with the modified DPMiner should be created and tested if it actually is 10 times faster as the current data points to.

In conclusion, the goal of creating a better equivalence class based clustering algorithm was achieved. However the result is not optimal as it is possible to create a faster algorithm using the proposed modified DPMiner algorithm.

Kokkuvõte

Töö põhilisteks eesmärkideks oli realiseerida ekvivalentsiklassidel põhinev klasterdamisalgoritm, mis kasutab ekvivalentsiklasside leidmiseks Grete Lindi poolt välja pakutud algoritmi ning võrrelda saadud klasterdamisalgoritmi hetkel parima algoritmiga samas vallas, ECCC, mis kasutab DPMinerit ekvivalentsiklasside leidmiseks.

Põhiliste tulemuste loetelu:

- Võrdlus G. Lind'i algoritmi ja hetkel kiireima ekvivalentsiklasse leidva algoritmi, DPMineri, vahel. Võrdlus näitab, et DPMiner on oodatavalt vähemalt kaks korda kiirem kuna ei läbi iga ekvivalentsiklassi leidmiseks andmetabeli ridu, vaid selle asemel läbib rea elementidest koostatud sageduspuud, FP-Tree. Sageduspuu omakorda võimaldab väärtuse põhist kärpimist, mis annab DPMinerile veel suurema eelise mahukate andmetabelite puhul.

- ECCC analüüs, mis näitab et kuna DPMiner tagastab ainult δ-diskriminatiivseid ekvivalentsiklasse, on klastrite täpsus 25% madalam kui klastritel, mis on moodustatud G. Lind'i algoritmi poolt tagastatud ekvivalentsiklassidest.

- Uus klasterdamisalgoritm juhuks kui kõik read sisaldavad sama palju attribuute. Algoritm on potentsiaalselt 10 korda kiirem ECCC-st, mis on mõeldud varieeruva atribuutide arvuga ridade jaoks.

- Klasterdamisalgoritmi prototüüp, mis sisaldab G. Lind'i poolt välja pakutud algoritmi ekvivalentsiklasside leidmiseks ja fikseeritud atribuutide pikkusega klasterdamisalgoritmi. Algoritm on mushroom andmehulga puhul neli korda kiirem kui hetkel parim ekvivalentsiklassidel põhinev klasterdamisalgoritm ECCC.

Olulistemateks järeldusteks võib lugeda:

- Kui δ-diskriminatiivne kitsendus DPMinerilt ära võtta, siis tema ekvivalentsiklassidest moodustatud klastrid oleksid 25% täpsemad. Sellega muidugi kaasneb aga töötlusaja pikenemine.

- Kui eeldada, et iga rida sisaldab sama palju atribuute, siis on võimalik välja pakutud klasterdamisalgoritmiga saada 10 korda kiirem rakendus kui kasutada ekvivalentsiklasside leidmiseks eelmises punktis mainitud DPMinerit koos modifikatsiooniga.

Analüüsi tulemused pole kahjuks 100% täpsed kuna DPMineri põhialgoritmiga ei olnud võimalik replitseerida originaalartiklis mainitud tulemusi. Seega tuli kasutada DPMineri closed varianti ja selle kaudu kaudselt võrrelda DPMineri põhialgoritmiga. Lisaks ei õnnestunud leida ECCC algoritmi loojate poolt tehtud algoritmi rakendust. Seega võib kiirustega seotud tulemustes esineda väiksemaid kõrvalekaldeid. Kuid arvestades asjaoluga, et DPMineri kärbitud variandi kiiruste graafikud langesid suhteliselt täpselt kokku ja ECCC algoritm oli realiseeritud paar aastat tagasi, ei ole oodatav viga kuigi suur.

Tulevikus võiks kindlasti realiseerida uue prototüübi välja pakutud klasterdamisalgoritmile fikseeritud veergude jaoks, kasutades DPMineri uuendatud varianti ning analüüsida, kas tulemus on tõepoolest 10 korda kiirem nagu praegused andmed näitavad.

Kokkuvõtvalt võib väita et saavutati eesmärk leida kiirem klasterdamisalgoritm, kuid hetke lahendus pole optimaalne ja seega on võimalik luua tunduvalt kiirem algoritm kui kasutada ekvivalentsiklasside leidmiseks DPMineri muudetud varianti, kus leitakse kõik ekvivalentsiklassid.

Bibliography

[1] W. W. D. S. G. D. L. Qingbao, „An Equivalence Class Based Clustering Algorithm for Categorical Data," *The First International Conference on Advances in Information Mining and Management*, 2011.

[2] G. L. a. L. W. J. Li, "Mining Statistically Important Equivalence Classes and Delta-Discriminative Emerging Patterns," *KDD,* pp. 1-10,430-439, 2007.

[3] J. H. a. M. Kamber, Data Mining: Concepts and Techniques, Morgan Kaufmann, 2nd edition, 2006.

[4] J. L. a. L. W. Guimei Liu, „A New Concise Representation of Frequent," *Knowledge and Information Systems,* kd. 17, nr 1, pp. 35-56, 2008.

[5] D. Lemire, „Do not waste time with STL vectors," 20-06-2012. [WWW]. Available: http://lemire.me/blog/archives/2012/06/20/do-not-waste-time-with-stl-vectors/. [Visited 16-04-2014].

[6] The C++ Resources Network, „Arrays," The C++ Resources Network, [WWW]. Available: http://www.cplusplus.com/doc/tutorial/arrays/. [Visited 16-04-2014].

[7] The C++ Resources Network, „Structures," The C++ Resources Network, [WWW]. Available: http://www.cplusplus.com/doc/tutorial/structures/. [Visited 16-04 -2014].

[8] The C++ Resources Network, „Sort," The C++ Resources Network, [WWW]. Available: http://www.cplusplus.com/reference/algorithm/sort/. [Visited 16-04-2014].

[9] J. Floyd, „What do Hash Collisions Really Mean?," 18-07-2008. [WWW]. Available: http://permabit.wordpress.com/2008/07/18/what-do-hash-collisions-really-mean/. [Visited 17-04-2014].

[10] J. P. a. Y. Y. J. Han, „Mining frequent patterns," *SIGMOD,* pp. 1-12, 05-2000.

[11] H. Perera, „How to identify frequent patterns using FP tree algorithm," 06-2011. [WWW]. Available: http://hareenlaks.blogspot.com/2011/06/fp-tree-example-how-to-identify.html. [Visited 17-04-2014].

[12] Wikipedia, „Tree (graph theory)," Wikipedia, [WWW]. Available: http://en.wikipedia.org/wiki/Tree_(graph_theory). [Visited 17-04-2014].

[13] Wikipedia, „Big O notation," Wikipedia, [WWW]. Available: http://en.wikipedia.org/wiki/Big_O_notation. [Visited 21-04-2014].

[14] K. &. L. M. Bache, „UCI Machine Learning Repository Mushroom Data Set," University of California, Irvine, School of Information and Computer Sciences, 2013. [WWW]. Available: http://archive.ics.uci.edu/ml/datasets/Mushroom. [Visited 21-04-2014].

[15] K. &. L. M. Bache, „UCI Machine Learning Repository Connect-4 Data Set," University of California, Irvine, School of Information and Computer Sciences, 2013. [WWW]. Available: http://archive.ics.uci.edu/ml/datasets/Connect-4. [Visited 24-04-2014].

[16] N. D. a. B. Cr´emilleux, „ ECCLAT: a New Approach of Clusters Discovery in Categorical Data," *22nd SGAI International Conference on Knowledge Based Systems*, 2002.

[17] K. Beck, Test Driven Development: By Example, Addison Wesley, 2003.

[18] Software Freedom Conservancy, „Git home page," [WWW]. Available: http://git-scm.com/. [Visited 18-05-2014].

Appendix 1. Application Source Code

The source code of the main application is divided into 4 files. These files do not contain file reading and writing related code but only the algorithm implementation:

- ECFinder.h
- ECFinder.cpp – used to find equivalence classes
- ClusterFinder.h
- ClusterFinder.cpp – used to find clusters based on equivalence classes

ECFinder.h

```
#pragma once
#include <vector>
#include <unordered_map>
#include <memory>

struct FrequencyValue
{
	int frequency;
	int row;
	int col;

	int getOriginalTableCol() { return row; }
	int getOriginalTableColValue() { return col + 1; }

	FrequencyValue(int frequency, int row, int col) :frequency(frequency), row(row),
col(col){}
};
struct EquivalenceClass
{
	int frequency;
	std::vector<int> closedSet;
	std::vector<int> generators;

	double numberOfFilledColumnsInCS;
	double qualityIndex;
	double discriminativenessIndex;
	std::vector<int> rows;

	EquivalenceClass(int frequency, const std::vector<int> &closedSet, std::vector<int>
generators)
```

```
			:frequency(frequency), closedSet(closedSet), generators(generators){}
};
typedef std::shared_ptr<EquivalenceClass> EquivalenceClassPointer;

struct EquivalenceClassKey
{
	int frequency;
	std::vector<int> closedSet;

	EquivalenceClassKey(int frequency, const std::vector<int> &closedSet)
:frequency(frequency), closedSet(closedSet){}

	bool operator==(const EquivalenceClassKey &other) const
	{
		return frequency == other.frequency && closedSet == other.closedSet;
	}
};
struct EquivalenceClassKeyHasher
{
	static std::vector<size_t> colLengthHashed;
	size_t operator()(const EquivalenceClassKey& key) const
	{
		size_t hash = 0;
		for (size_t col = 0; col < key.closedSet.size(); col++)
		{
			hash += key.closedSet[col] * colLengthHashed[col];
		}
		return hash;
	}
};

class ECFinder
{
public:
```

```
    ECFinder(const std::vector<int> &table, const std::vector<size_t>&colLength, int
allowedMinFreq);
    ~ECFinder();
    inline void initializeFrequencytable(std::vector<int > &frequencyTable);
    inline void initializeSubMatrix(std::vector<int > &subMatrix);
    inline void orderFrequencyTableInAscendingOrderAndZeroesDown(std::vector< int>
&freqTable, const std::vector< int> &oldFreqTable, int maxFrequency,
std::vector<FrequencyValue> &orderedFrequencies, const std::vector< int> &closedSet);
    inline void populateNewSubMatrixAndFrequencyTable(const std::vector< int >
&oldSubMatrix, FrequencyValue &maxFrequency, std::vector< int > &newSubMatrix,
std::vector< int > &newFrequencyTable);
    inline bool populateClosedSetAndReturnIfExisting(const std::vector< int>
&oldFrequencyTable, const std::vector< int> &newFrequencyTable, int maxFrequency,
std::vector<int> &closedSet);
    inline bool addToResultAndReturnIfNew(const std::vector< int>
&oldFrequencyTable, const std::vector< int> &newFrequencyTable, int maxFrequency,
std::vector<int> &closedSet);
    inline void findEquivalenceClass(const std::vector< int > &oldSubmatrix, const
std::vector<int> &oldFrequencyTable, FrequencyValue &maxFrequency, const
std::vector<int> &oldClosedSet);
    void findAllEquivalenceClasses();
    void findFrequencyClasses(const std::vector<int> &submatrix, const std::vector< int >
&oldFrequencyTable, std::vector< int > &newFrequencyTable, int maxFrequency, const
std::vector< int > &closedSet);
    std::vector<int> currentGenerator;
    std::unordered_map<EquivalenceClassKey, EquivalenceClassPointer,
EquivalenceClassKeyHasher> resultMap;
private:
    std::vector<int> table;
    std::vector<size_t> colLength;
    size_t numberOfColumns;
    int allowedMinFreq;
};
```

ECFinder.cpp

```
#include "stdafx.h"
#include "ECFinder.h"

std::vector<size_t> EquivalenceClassKeyHasher::colLengthHashed;

ECFinder::ECFinder(const std::vector<int> &table, const std::vector<size_t>&colLength, int
allowedMinFreq)
:table(table), colLength(colLength), resultMap(200000), currentGenerator((colLength.size() -
1)),
allowedMinFreq(allowedMinFreq)
{
	numberOfColumns = (colLength.size() - 1);
	std::vector<size_t> colLengthHashed(numberOfColumns);
	colLengthHashed[0] = 1;
	for (size_t col = 1; col < colLengthHashed.size(); col++)
	{
		colLengthHashed[col] = (colLength[col] - colLength[col - 1]) *
colLengthHashed[col - 1];
	}
	EquivalenceClassKeyHasher::colLengthHashed = colLengthHashed;
}

ECFinder::~ECFinder(){}

void ECFinder::findAllEquivalenceClasses()
{
	std::vector< int > frequencyTable(colLength.back());
	std::vector< int > closedSet(numberOfColumns);
	std::vector<int > subMatrix(table.size() / numberOfColumns);
	initializeFrequencytable(frequencyTable);
```

```
        initializeSubMatrix(subMatrix);
        findFrequencyClasses(subMatrix, frequencyTable, frequencyTable,
std::numeric_limits<int>::max(), closedSet);
}

void ECFinder::findFrequencyClasses(const std::vector<int> &submatrix, const std::vector<
int > &oldFrequencyTable, std::vector< int > &newFrequencyTable, int maxFrequency, const
std::vector< int > &closedSet)
{
        std::vector< FrequencyValue > orderedFrequencies;
        orderFrequencyTableInAscendingOrderAndZeroesDown(newFrequencyTable,
oldFrequencyTable, maxFrequency, orderedFrequencies, closedSet);
        for (auto & minFrequency : orderedFrequencies)
        {
                currentGenerator[minFrequency.getOriginalTableCol()] =
minFrequency.getOriginalTableColValue();
                findEquivalenceClass(submatrix, newFrequencyTable, minFrequency,
closedSet);
                currentGenerator[minFrequency.getOriginalTableCol()] = 0;
                newFrequencyTable[colLength[minFrequency.row] + minFrequency.col] = 0;
        }
}

inline bool sortByFrequency(const FrequencyValue &leftFreq, const FrequencyValue
&rightFrequency)
{
        if (leftFreq.frequency == rightFrequency.frequency) {
                if (leftFreq.row == rightFrequency.row) {
                        return leftFreq.col < rightFrequency.col;
                }
                return leftFreq.row < rightFrequency.row;
        }
        return leftFreq.frequency < rightFrequency.frequency;
}
```

```
inline void ECFinder::orderFrequencyTableInAscendingOrderAndZeroesDown(std::vector<
int> &freqTable, const std::vector< int> &oldFreqTable, int maxFrequency,
std::vector<FrequencyValue> &orderedFrequencies, const std::vector< int> &closedSet)
{
	orderedFrequencies.reserve(freqTable.size());
	auto freqIt = freqTable.begin();
	auto oldfreqIt = oldFreqTable.begin();
	auto colIt = colLength.begin();
	auto closedSetIt = closedSet.begin();
	for (size_t row = 0; colIt < colLength.end() - 1; row++, colIt++, closedSetIt++){
		if (*closedSetIt == 0) {
			for (size_t col = 0; freqIt < freqTable.begin() + *(colIt + 1); col++,
freqIt++, oldfreqIt++)
			{
				if (*oldfreqIt == 0) {
					*freqIt = 0;
					continue;
				}
				if (*freqIt >= allowedMinFreq && *freqIt < maxFrequency){
					orderedFrequencies.emplace_back(*freqIt, row, col);
				}
			}
		}
		else {
			freqIt += *(colIt + 1) - *colIt;
			oldfreqIt += *(colIt + 1) - *colIt;
		}
	}
	std::sort(orderedFrequencies.begin(), orderedFrequencies.end(), sortByFrequency);
}
```

```
inline void ECFinder::findEquivalenceClass(const std::vector< int > &oldSubmatrix, const
std::vector<int> &oldFrequencyTable, FrequencyValue &maxFrequency, const
std::vector<int> &oldClosedSet)
{
	std::vector< int > newSubmatrix(maxFrequency.frequency);
	std::vector< int > newFrequencyTable(oldFrequencyTable.size());
	std::vector< int > newClosedSet(oldClosedSet);
	newClosedSet[maxFrequency.getOriginalTableCol()] =
maxFrequency.getOriginalTableColValue();
	populateNewSubMatrixAndFrequencyTable(oldSubmatrix, maxFrequency,
newSubmatrix, newFrequencyTable);
	bool isNew = addToResultAndReturnIfNew(oldFrequencyTable, newFrequencyTable,
maxFrequency.frequency, newClosedSet);
	if (maxFrequency.frequency > allowedMinFreq) {
		findFrequencyClasses(newSubmatrix, oldFrequencyTable,
newFrequencyTable, maxFrequency.frequency, newClosedSet);
	}
}

inline void ECFinder::populateNewSubMatrixAndFrequencyTable(const std::vector< int >
&oldSubMatrix, FrequencyValue &maxFrequency, std::vector< int > &newSubMatrix,
std::vector< int > &newFrequencyTable)
{
	auto it = newSubMatrix.begin();
	auto tableIt = table.begin();
	for (const int rowIdx : oldSubMatrix)
	{
		tableIt = table.begin() + rowIdx * numberOfColumns;
		if ( *(tableIt + maxFrequency.getOriginalTableCol()) ==
maxFrequency.getOriginalTableColValue()) {
			*it = rowIdx;
			++it;
			for (auto colIt = colLength.begin(); colIt < colLength.end() - 1; colIt++,
tableIt++)
```

```
                {
                        newFrequencyTable[*colIt + *tableIt - 1]++;
                }
            }
        }
}

inline bool ECFinder::addToResultAndReturnIfNew(const std::vector< int>
&oldFrequencyTable, const std::vector< int> &newFrequencyTable,
        int maxFrequency, std::vector<int> &closedSet )
{
        bool isExisting = populateClosedSetAndReturnIfExisting(oldFrequencyTable,
newFrequencyTable, maxFrequency, closedSet);
        if (isExisting) {
                EquivalenceClassPointer equivalenceClass = resultMap[{maxFrequency,
closedSet}];
                equivalenceClass->generators.reserve(equivalenceClass->generators.size() +
currentGenerator.size());
                equivalenceClass->generators.insert(equivalenceClass->generators.end(),
currentGenerator.begin(), currentGenerator.end());
        }
        else
        {
                EquivalenceClassPointer equivalenceClass(new
EquivalenceClass(maxFrequency, closedSet, currentGenerator));
                resultMap[{maxFrequency, closedSet}] = equivalenceClass;
        }
        return !isExisting;
}

inline bool ECFinder::populateClosedSetAndReturnIfExisting(const std::vector< int>
&oldFrequencyTable, const std::vector< int> &newFrequencyTable, int maxFrequency,
std::vector<int> &closedSet)
{
```

```
	int savedSize = 0;
	bool isExisting = false;
	auto freqIt = newFrequencyTable.begin();
	auto oldfreqIt = oldFrequencyTable.begin();
	auto colIt = colLength.begin();
	auto closedSetIt = closedSet.begin();
	for (size_t realCol = 0; colIt < colLength.end() - 1; colIt++, realCol++, closedSetIt++)
	{
		if (*closedSetIt == 0)
		{
			for (size_t col = 0; freqIt < newFrequencyTable.begin() + *(colIt + 1);
col++, freqIt++, oldfreqIt++)
			{
				if (*freqIt == maxFrequency) {
					*closedSetIt = col + 1;
					if (*oldfreqIt == 0) {
						isExisting = true;
					}
					freqIt = newFrequencyTable.begin() + *(colIt + 1);
					oldfreqIt = oldFrequencyTable.begin() + *(colIt + 1);
					break;
				}
			}
		}
		else {
			freqIt += *(colIt + 1) - *colIt;
			oldfreqIt += *(colIt + 1) - *colIt;
		}
	}
	return isExisting;
}

inline void ECFinder::initializeFrequencytable(std::vector<int > &frequencyTable)
{
```

```
	for (size_t count = 0; count < table.size(); count++)
	{
		frequencyTable[colLength[count % numberOfColumns] + (table[count] -
1)]++;
	}
}

inline void ECFinder::initializeSubMatrix(std::vector<int > &subMatrix)
{
	for (size_t count = 0; count < table.size() / numberOfColumns; count++)
	{
		subMatrix[count] = count;
	}
}
```

ClusterFinder.h

```
#pragma once
#include "ECFinder.h"

class ClusterFinder
{
public:
	ClusterFinder(size_t numberOfColumns, int minFrequency, std::vector<int> table);
	~ClusterFinder();
	inline void findDiscriminativenessIndex(EquivalenceClassPointer &candidateCluster);
	inline double findHomogeneityIndex(EquivalenceClassPointer &candidateCluster);
	inline void findQualityIndex(EquivalenceClassPointer &candidateCluster);
	EquivalenceClassPointer findFinalCluster(std::vector<EquivalenceClassPointer>
&candidateClusters);
	inline void removeExtractedRowsAndReduceFrequencies(EquivalenceClassPointer
&finalCluster, std::vector<EquivalenceClassPointer> &candidateClusters);
	inline EquivalenceClassPointer
findNextFinalCluster(std::vector<EquivalenceClassPointer> &candidateClusters);
	void findClusters(std::vector<EquivalenceClassPointer> &candidateClusters);
	std::vector<int> table;
	std::vector<int> availableRows;
	std::vector<int> tempAvailableRows;
	std::vector<int> potentialAvailableRows;
	std::vector<int> tempUsedRows;
	std::vector<int> potentialUsedRows;
	std::vector<EquivalenceClassPointer> result;
private:
	inline int calculateFrequency(const std::vector<int> &closedSet);
	size_t numberOfColumns;
	int minFrequency;
};
```

ClusterFinder.cpp

```
#include "stdafx.h"
#include "ClusterFinder.h"



ClusterFinder::ClusterFinder(size_t numberOfColumns, int minFrequency, std::vector<int>
table)
: numberOfColumns(numberOfColumns), minFrequency(minFrequency), table(table),
availableRows(table.size() / numberOfColumns, 1)
{
	for (size_t count = 0; count < availableRows.size(); count++)
	{
		availableRows[count] = count;
	}
	tempAvailableRows.reserve(availableRows.size());
	potentialAvailableRows.reserve(availableRows.size());
	tempUsedRows.reserve(availableRows.size());
	potentialUsedRows.reserve(availableRows.size());
}
ClusterFinder::~ClusterFinder()
{
}
std::vector<EquivalenceClassPointer> tmpCandidateClusters;

inline void ClusterFinder::findDiscriminativenessIndex(EquivalenceClassPointer
&candidateCluster)
{
	int genFilledSize = 0;
	candidateCluster->numberOfFilledColumnsInCS = 0;
	candidateCluster->discriminativenessIndex = 1;
	for(const int value : candidateCluster->closedSet) {
```

```
		if (value != 0) {
			candidateCluster->numberOfFilledColumnsInCS++;
		}
	}
	for (size_t i = 0; i < candidateCluster->generators.size(); i++)
	{
		if (candidateCluster->generators[i] != 0)
		{
			genFilledSize++;
		}
		if ((i + 1) % numberOfColumns == 0)
		{
			candidateCluster->discriminativenessIndex *= (1 + (candidateCluster->numberOfFilledColumnsInCS - genFilledSize) / candidateCluster->numberOfFilledColumnsInCS);
			genFilledSize = 0;
		}
	}
}

inline double ClusterFinder::findHomogeneityIndex(EquivalenceClassPointer &candidateCluster)
{
	return candidateCluster->numberOfFilledColumnsInCS / numberOfColumns;
}

inline void ClusterFinder::findQualityIndex(EquivalenceClassPointer &candidateCluster)
{
	findDiscriminativenessIndex(candidateCluster);
	candidateCluster->qualityIndex = candidateCluster->discriminativenessIndex *
		findHomogeneityIndex(candidateCluster);
}
```

```
inline bool sortByQualityIndex(const EquivalenceClassPointer &leftFreq, const
EquivalenceClassPointer &rightFrequency)
{
	if (leftFreq->qualityIndex == rightFrequency->qualityIndex) {
		return leftFreq->frequency > rightFrequency->frequency;
	}
	return leftFreq->qualityIndex > rightFrequency->qualityIndex;
}

void ClusterFinder::findClusters(std::vector<EquivalenceClassPointer> &candidateClusters)
{
	/*Find QIs*/
	for (auto & candidateCluster : candidateClusters)
	{
		findQualityIndex(candidateCluster);
	}
	/*sort*/
	std::sort(candidateClusters.begin(), candidateClusters.end(), sortByQualityIndex);
	/*find all clusters*/
	int frequency;
	int maxFrequency;
	EquivalenceClassPointer finalCluster;
	unsigned int unsignedMinFrequency = (unsigned int) minFrequency;
	for (auto cluster = candidateClusters.begin(); cluster < candidateClusters.end() &&
availableRows.size() >= unsignedMinFrequency; cluster++)
	{
		if ((*cluster)->frequency < minFrequency)
		{
			continue;
		}

		/* calculate new freq*/
		maxFrequency = calculateFrequency((*cluster)->closedSet);
		if (maxFrequency < minFrequency)
```

```
			{
				continue;
			}
			(*cluster)->frequency = maxFrequency;
			finalCluster = (*cluster);
			potentialAvailableRows.swap(tempAvailableRows);
			potentialUsedRows.swap(tempUsedRows);

			/*Check for same QI elements*/
			for (auto tempCluster = cluster + 1; tempCluster < candidateClusters.end() &&
!((*tempCluster)->qualityIndex < (*cluster)->qualityIndex); tempCluster++)
			{
				frequency = calculateFrequency((*tempCluster)->closedSet);
				(*tempCluster)->frequency = frequency;
				if (frequency > maxFrequency) {
					finalCluster = (*tempCluster);
					maxFrequency = frequency;
					potentialAvailableRows.swap(tempAvailableRows);
					potentialUsedRows.swap(tempUsedRows);
				}
			}
			if (finalCluster != (*cluster)) {
				cluster--; /* this step is necessary in order to recheck the current cluster
when it is not the final one*/
			}

			/*remove used rows*/
			availableRows.swap(potentialAvailableRows);

			finalCluster->rows = potentialUsedRows;
			result.push_back(finalCluster);
		}
}
```

```
inline int ClusterFinder::calculateFrequency(const std::vector<int> &closedSet){
	int frequency = 0;
	auto tableIt = table.begin();
	bool containedInRow;
	tempAvailableRows.clear();
	tempUsedRows.clear();
	for (const int rowIdx : availableRows){
		tableIt = table.begin() + rowIdx * numberOfColumns;
		containedInRow = true;
		for (auto closedSetIt = closedSet.begin(); closedSetIt < closedSet.end();
closedSetIt++, tableIt++) {
			if (*closedSetIt != 0 && *closedSetIt != *tableIt) {
				containedInRow = false;
				break;
			}
		}
		if (containedInRow) {
			frequency++;
			tempUsedRows.emplace_back(rowIdx);
		}
		else {
			tempAvailableRows.emplace_back(rowIdx);
		}
	}

	return frequency;
}
```