EDAMI - project documentation

Ewa Kowalska, Łukasz Lepak

23 January 2020

Contents

1 Objective

The aim of the project is to implement Charm and dCharm algorithms used to find all frequent closed itemsets [1] with proposed modifications and evaluate them experimentally.

An itemset X is considered frequent if:

$$sup(X) > minSup \tag{1}$$

where $minSup$ is defined threshold value. A closure of an itemset X is written as $\gamma(X)$ and defined as:

$$\gamma(X) = \cap\{T \in D \cup \{I\} | T \supseteq X\} \tag{2}$$

where T is a transaction, D is a list of all transactions, $\{I\}$ is a set of all items. A closure of an itemset can be understood as the greatest superset of X having the same support as X.

An itemset X is defined as closed when it is its own closure, that is the greatest superset of X with the same support is also X. All closures are closed itemsets. A set of frequent closed itemsets is a loseless representation of all frequent itemsets - if itemset X does not have any superset in frequent closed itemsets, then it is infrequent, else its support is equal to a maximum support of all its supersets in frequent closed itemsets.

Charm and dCharm algorithms are an efficient way to find frequent closed itemsets. They use tidlists and difflists respectively to merge itemsets and calculate supports. They use properties of these lists to traverse the IT tree (tree with itemsets and their lists in nodes) and discover closed frequent itemsets.

2 Assumptions

We use equation (1) to check if an itemset is frequent. However, in our program, user can supply a value of relative minimum support. To use absolute minimum support in algorithms, it is calculated as follows:

$$minSup = \lfloor rSup \cdot |D| \rfloor \tag{3}$$

where $rSup$ is user-defined relative minimum support threshold value and $|D|$ is a number of all transactions in a dataset.

All experiments were conducted on a single machine with Intel Core i7-7700HQ CPU (base clock 2.8GHz) on Linux Mint 19.1 operating system. Results regarding time of an execution of a program may vary depending on machine and its operating system. Also, in places where execution time is provided, it was calculated as a mean time of 10 runs with the same parameters. Standard deviations of these times were ommited, as algorithm works in a deterministic way and differences in execution time were very small, probably as the result of a current state of an operating system. Statistics regarding algorithm results (number of closed itemsets, properties call count) were taken from one run, as they always produce the same results due to algorithm's deterministic nature.

Data preprocessing was done using Python script in a following way - for every column, assign to each unique value first free integer and replace all ocurrences of this value with assigned integer. Both raw and preprocessed datasets are provided with source files of a project.

3 Input and output data

3.1 Datasets used for experiments

In our experiments, we used 3 datasets from UCI Machine Learning Repository [2]. We also created an "example" dataset based on an example used in article [1]. Statistics of datasets are presented in a table below:

dataset name	number of transactions	length of transaction	number of unique items
example	6	3-5	5
car	1728	7	25
mushroom	8124	23	119
nursery	12960	9	32

Table 1: Statistics of a dataset.

Datasets taken from UCI Machine Learning Repository (car, mushroom, nursery) were preprocessed to represent items (unique combinations of an attribute and its value) with integer numbers. "Example" dataset was prepared in an already processed way. Processed datasets are split into two files, data file ("*.data") contains a list of transactions, names file ("*.names") contains names of items in sorted order - index of a given name matches integer given to this item in transactions list.

Datasets vary in number of transactions and lengths of transactions. The smallest one is car dataset - it contains the least transactions and also transactions have the shortest lenght. In the case of nursery dataset, transactions are only a bit longer, but there are many transactions in a dataset. Finally, the mushroom dataset has many transactions which are significantly longer.

3.2 Input data format

Input data format is a list of transactions, where a transaction is a list of integers. Example input data:

```
0,1,3,4
1,2,4
0,1,3,4
0,1,2,4
0,1,2,3,4
1,2,3
```

Input data should be stored in a file with "data" extension. Optionally, names corresponding to integers should be stored in the same directory and with the same name, but "names" extension. Path to these files (without any extensions) can be given via a command line argument.

3.3 Output data format

In its default setting, algorithms' results are returned as a collection of closed itemsets with their absolute support, which are then printed to standard output (both relative and absulute support are printed) with items

represented by integers. This can be changed to display names of items instead of corresponding integers or statistics of execution, which include time (in milliseconds) of mining frequent itemsets, algorithm runtime and number of algorithms' property calls. These two options can be enabled with command line arguments, with default settings they are disabled. Also, printing of closed itemsets can be disabled via a command line argument. Sample output for small example dataset will be given later in the report, after describing command line options.

4 Class design and implementation

4.1 Classes

The methods and structures used by Charm and dCharm algorithms are represented by Charm and DCharm classes. Both Charm and dCharm algorithms are implemented based on a tree data structure. The nodes of the tree are represented respectively by CharmNode class and DCharmNode class. Methods for data preprocessing and structures for measurements are enclosed in Utils files.

4.1.1 CharmNode

Class CharmNode represents an IT-pair processed by Charm algorithm. Apart from structures holding itemset and tid lists, it contains methods operating on that structures, such as taking union of itemsets or intersection of tid lists. It also contains structure named *children*, which holds CharmNode objects representing unioned itemsets and intersected tid lists of the CharmNode object with other CharmNode objects (children of the given CharmNode in a tree).

We found interesting the way of implementing *children* structure. For this purpose we used std::multimap container [3] , which holds <key, value> pairs of std::list<int> container and the CharmNode object representing a child. Depending on the sort parameter (described in section 5, experiments in section 7.2) the std::list<int> container is either an one-element list containing support of the child or many-element list containing child's itemset. The advantage of this approach is that we could use the same container for convenient children sorting depending on their support or itemset for different different sort modes (by increasing support, decreasing support or in lexicographical order). We adjusted build-in function in multimap container, which sorts objects depending on their key (in our case it is child support or itemset), so that it behaves differently depending on chosen sort mode.

4.1.2 Charm

Charm class contains methods implementing Charm algorithm (charmExtend and charmProperty, subsumption checking) and a structure holding mined closed itemsets. It also contains prior mined frequent itemsets of size 2 for the purpose of experiment described in section 7.3.

4.1.3 DCharmNode

DCharmNode class represents an itemset - difflist pair processed by dCharm algorithm. The implementation of the class is in many cases analogical to CharmNode class, however there are some differences. DCharmNode class additionally stores the support of the node, since it can't be derived directly from diff list

as in case of CharmNode, and also holds value of hash used for subsumption checking. Additionally methods for counting support of itemset and hash value are implemented in another way.

4.1.4 DCharm

Class DCharm is implemented analogically to Charm class. The only difference is that it operates on difflists provided by DCharmNodes.

4.2 Utils files

Files utils.h and utils.cpp contain utility functions used by the program. It contains a function to parse input arguments, display help, mine frequent itemsets (of length one and two if needed), read transactions (and optionally names of items) from a file and print statistics of a program. All these functions do not belong to any class and are used outside of algorithms. In file utils.h there are also two defined structures: Parameter and Stats. The first one is responsible for gathering parameters controlling the flow of a program (changed by command line arguments), Stats structure collects statistics of execution.

5 User guide

Program is executed by launching an executable file charm.exe. To configure the behaviour of a program command line arguments were prepared and are as follows:
-h,-help - print help.
-d,-dcharm - use dCharm instead of Charm.
-dps,-disablePrintSets - disables printing of closed itemsets.
-ms,-minSup <support> - minimum relative support value for frequent itemsets. Defaults value is 0.1.
-p <path>,-path <path> - path to dataset (.data and .names files) without extensions. Available names: example, mushroom, nursery, car. Files must have the same first part name, i.e. mushroom.data and mushroom.names. Default is data/processed/example (example small dataset).
-pn,-printNames - print closed itemsets items' names instead of their integer mappings.
-ps,-printStats - print statistics.
-s <type>,-sort <type> - type of ordering used in algorithms, possible types: asc (ascending), desc (descending), lex (lexicographical). Default is lex.
-uts, -useTwoSets - use frequent two items ets for checking in algorithms.

The same information can be displayed when running the program with -help or -h flag [charm.exe -h] or when providing unsupported command line argument.

6 Results generated for a small input data

Provided example output shown on Figure 1 corresponds to the example from [1]. Apart from mined closed frequent itemsets and their support (relative and absolute), statistics on the prior mining frequent itemsets for algorithm initialization, the time elapsed for algorithm tree travelsal, total time, number of closed itemsets and number of property calls are displayed (when appropriate flags are set - in the case of an example flags used were: -ps -pn [to display statistics and names instead of integer mappings]).

```
C    support: 1 (6)
A C W    support: 0.666667 (4)
A C T W    support: 0.5 (3)
C D    support: 0.666667 (4)
C D W    support: 0.5 (3)
C T    support: 0.666667 (4)
C W    support: 0.833333 (5)
Frequent itemsets creation time (ms): 0
Algorithm tree traversal time (ms): 0
Total time (ms): 0
Number of closed itemsets: 7
Property 1 calls: 0
Property 2 calls: 2
Property 3 calls: 3
Property 4 calls: 3
```

Figure 1: Results generated for the example from [1]

7 Conducted experiments

In this section we present experiments conducted for the data described in section 3.1.

7.1 Comparison of Charm and dCharm performance depending on minimal support value

Charm algorithm uses a vertical data format, which makes it much more effective than algorithms based on horizontal data format. However, when the tidlist cardinality gets very large, the benefits of that approach starts to vanish due to large intersection time. dCharm algorithm is analogical to Charm, except that it operates on difflists instead of tidlists, which keep track only of differences in tidlists of candidate pattern from its parent frequent pattern. Therefore it is efficient even in dense domains since difflists are fraction of tidlists, so the time needed for the intersections is minimized.

In the experiments we inspect the impact of minimal relative support value on Charm and dCharm performance. The tests were run for mushroom, nursery and car dataset, for the usual lexicographical ordering function. The minimal relative support parameter was set from 0 (inclusive) to 1 (inclusive) with changes by 0.05. Table 3 shows gained results for mushroom dataset, table 4 - for nursery dataset and table 5 - for car dataset. In the column *time difference* there was calculated the difference between time elapsed for Charm and time elapsed for dCharm, i.e. t(Charm) - t(dCharm).

7.1.1 Mushroom dataset

In case of mushroom dataset, we can observe that in most cases (except for the minimal support value equal to 0) dCharm performs significantly faster than Charm. Figure 2 visualizes time differences between Charm and dCharm. The difference for minimal support value equal to 0 was not included in it, because it deranges the scale of the diagram.

7.1.2 Nursery and car datasets

In the case of nursery and car datasets, we can observe worse performance of dCharm algorithm compared to Charm algorithm in all cases (figure 3, 4). Worse results for dCharm in case of car and nursery datasets may result from the specifics of that data. We found out that most of the items in these datasets are used in less

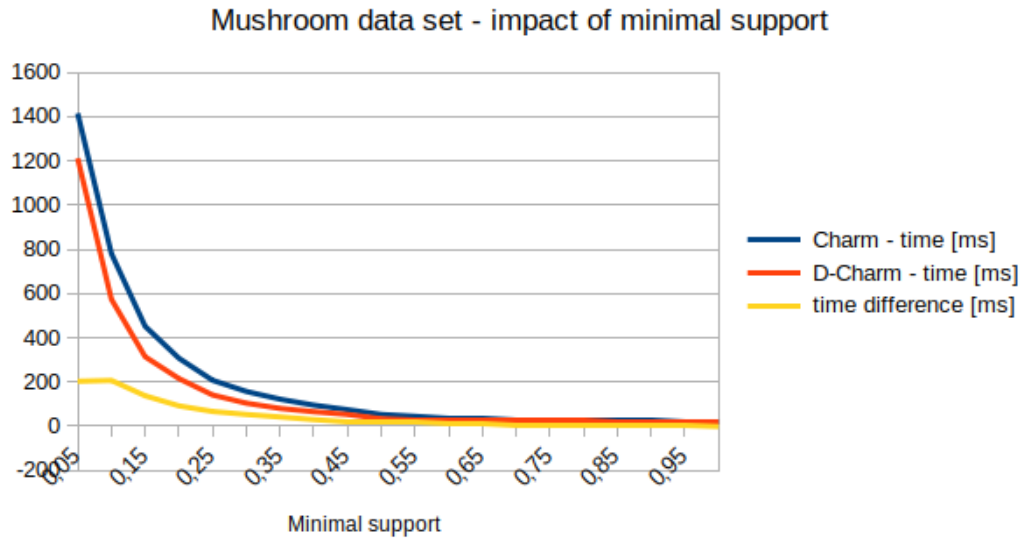

Figure 2: Impact of Minimal Support value on Charm and dCharm performance time for the mushroom dataset. Anomalous value for minimal support equal to 0 was not included.

than half of the transactions, which means that their initial difflists are longer than initial tidlists. Also, many combinations of items produce longer difflists than tidlists, so operations in dCharm take more time. These two datasets are an example that dCharm does not always beat Charm in terms of execution time.

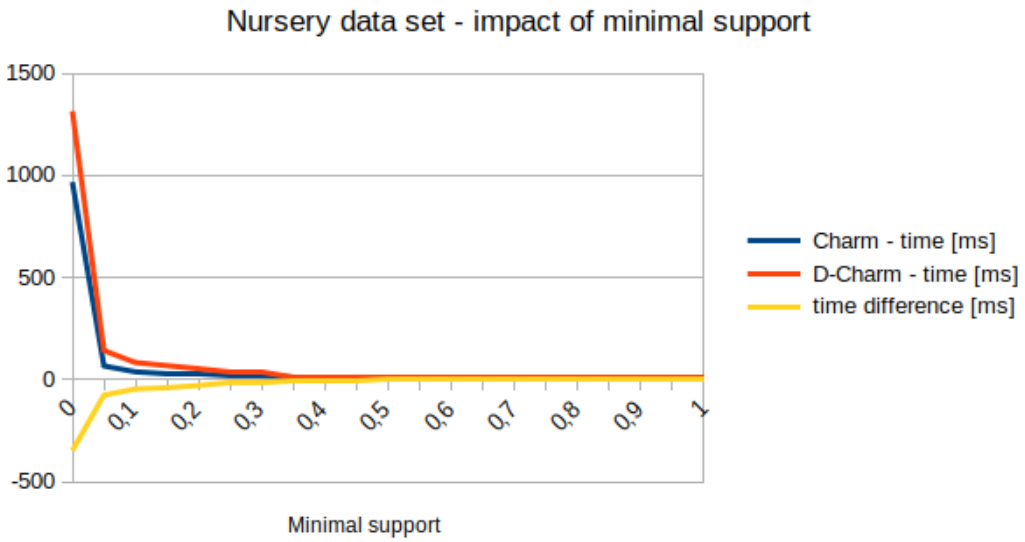

Figure 3: Impact of Minimal Support value on Charm and dCharm performance time for the nursery dataset.

7.1.3 Results for minimal relative support equal to 0

For minimal relative support equal to 0, in the case of all datasets, Charm algorithm got significantly better execution time than dCharm (Charm - 4764,7 ms, dCharm - 5451,8 ms). The reason may be that we are creating tidlists and difflists for each and every item and later, for all of their combinations. We consider all itemsets frequent, so even itemsets which occur very rarely are taken into account and itemsets with relative

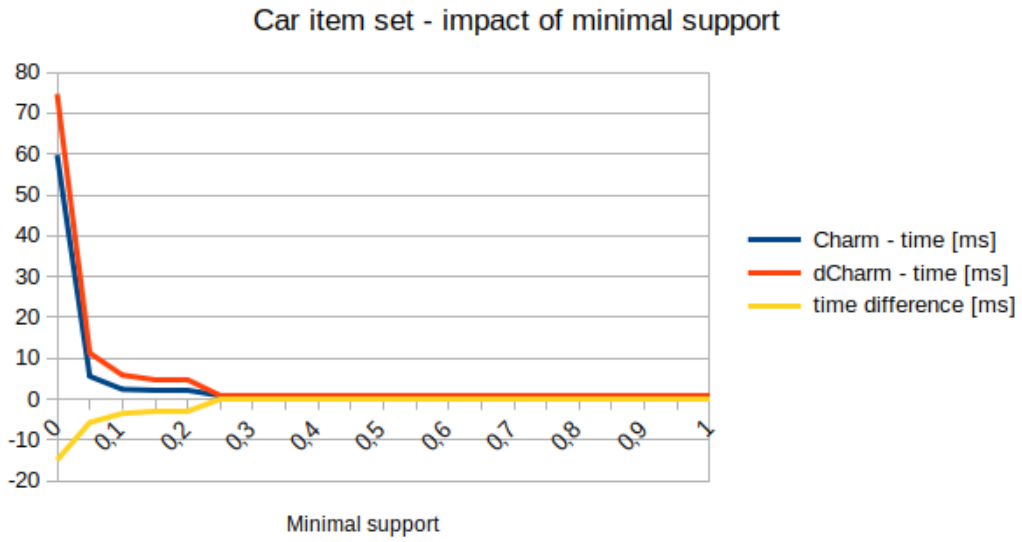

Figure 4: Impact of Minimal Support value on Charm and dCharm performance time for the car dataset.

support less than or equal to 0.05 are most often in every dataset. In these itemsets difflists may be significantly longer than tidlists and - similary as it was with nursery and car datasets - operations performed on difflists in dCharm take more time than opertions on tidlists in Charm.

7.2 Evaluation of impact of ordering function on IT-pairs

According to the article [1], the order of processing IT-pairs has crucial impact on the efficiency of Charm and dCharm algorithms performance. There are four properties (rules) that are applied while combining two IT-pairs for faster exploration of closed itemsets.

1. $If\ t(X_i) = t(X_j),\ then\ c(X_i) = c(X_j) = c(X_i \cup X_j)$
2. $If\ t(X_i) \subset t(X_j),\ then\ c(X_i) \neq c(X_j),\ but\ c(X_i) = c(X_i \cup X_j)$
3. $If\ t(X_i) \supset t(X_j),\ then\ c(X_i) \neq c(X_j),\ but\ c(X_j) = c(X_i \cup X_j)$
4. $If\ t(X_i) \neq t(X_j),\ then\ c(X_i) \neq c(X_j) \neq c(X_i \cup X_j)$

Properties 1 and 2 are preferable over 3 and 4, because they don't result in adding new IT-pairs to the processed tree, moreover, rule 1 discards some of them. As stated in the article, the most promising way to increase occurrences of rules 1 and 2 is to sort IT-pairs in increasing order of their support. In this experiment we evaluate that thesis by measuring the time elapsed for Charm and dCharm depending on the method of ordering. We compare three methods of ordering: increasing support of IT-pairs, decreasing support and lexicographical ordering.

Table 4 presents time elapsed for the datasets depending on chosen method of ordering. Also columns p1, p2, p3, p4 store information about the number of occurrences of each property (p 1 stands for property 1, etc.). Minimal support value was set to 0.1. Figures 5 - 10 visualize obtained data for mushroom, nursery and car dataset.

As we expected, for the ordering method based on ascending support property 3 never occurs, which can be deduced from the figures 5, 7 and 9. The fact worth noticing is, that for the ascending order in the three datasets property 4 has either the least number of occurrences or its occurrences are equal to property

dataset	ordering	Charm - time [ms]	dCharm - time [ms]	p 1	p 2	p 3	p 4
mushroom	support ascending	656.9	254.9	1153	2775	0	7385
mushroom	support descending	1251.5	1607.1	316	0	2132	11164
mushroom	lexicographical	780.7	573.6	394	1183	1849	8240
nursery	support ascending	37.2	78.0	10	0	0	125
nursery	support descending	42.3	103.6	12	0	0	127
nursery	lexicographical	38.7	84.0	12	0	0	127
car	support ascending	2.0	6.0	0	7	0	50
car	support descending	3.0	7.9	0	0	2	50
car	lexicographical	2.4	5.9	0	7	0	50

Table 2: Time elapsed depending on the chosen method of ordering and with relative minimum support set to 0.1.

4 occurrences for the remaining ordering methods. That information is very important, because occurrence of property 4 causes adding new elements to the tree and thus decreasing performance of the algorithm. Also for ordering based on descending support property 2 is never present. However for ordering method based on lexicographical method all 4 properties occurrences can be present. For all examined datasets and each of the three ordering methods property 4 is the most common.

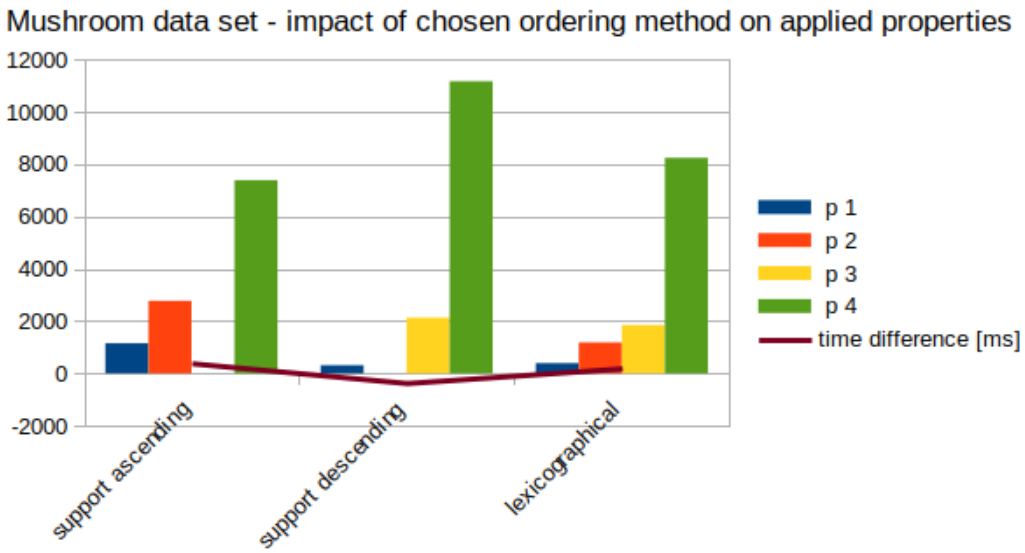

Figure 5: Relationship between chosen ordering function on mushroom dataset and the number of occurrences of particular properties.

In the mushroom dataset (figure 5) for the order based on ascending support, property 2 occurrences are almost 3 times rarer than occurrences of property 4 and property 1 occurrences - about 6 times rarer. For the descending support order - property 1 occurs even 3 times rarer than in ascending support order and property 4 occurrence is the highest among all ordering methods. For the lexicographical order property 3 occurrence is almost the same as for descending support, however property 4 is much rarer than for descending support.

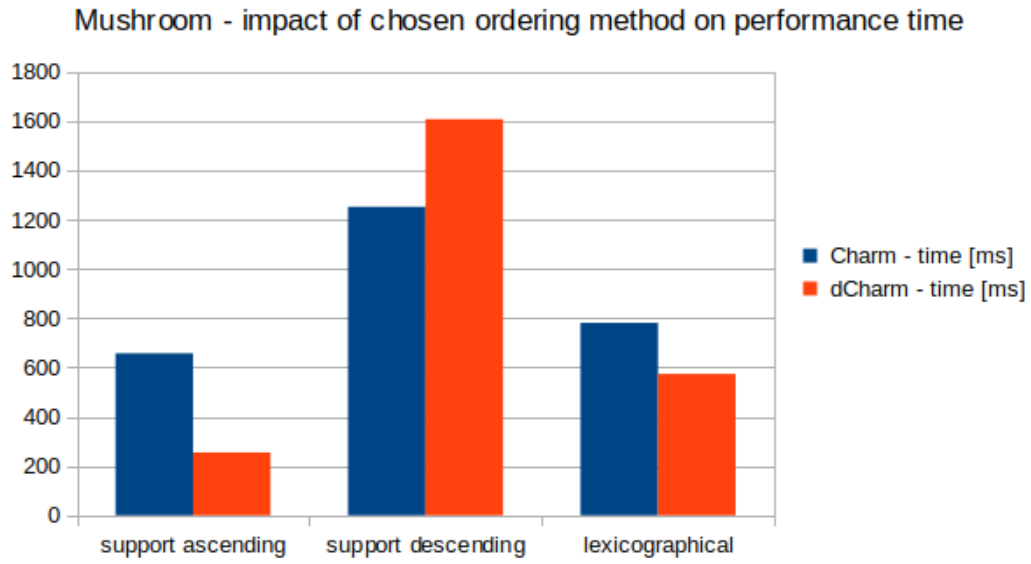

Figure 6: Relationship between chosen ordering function on mushroom data and time performance of Charm and dCharm.

Thus we can deduce, that for mushroom dataset order based on decreasing support is the worst. When we compare properties for ascending support and lexicographical order, we can notice that property 1 and 2 is more frequent for the ascending support, property 3 is not present and property 4 occurs less frequent. The conclusion is, the best ordering method for mushroom dataset is ascending support.

Figure 6 presents measurements of performance time depending on the ordering method, which supports that thesis. For the ascending support and lexicographical order dCharm has shorter elapsed time than Charm, moreover, the greatest difference in performance time is for the ascending support order. On the contrary, for descending support order dCharm performs worse than Charm. It is probably due to the fact, that initial items' combinations for that ordering are less frequent - they have shorter tidlists in Charm and longer difflists in DCharm, thus operations of taking difference of difflists in dCharm are longer than intersections of short tidlists in Charm.

For the nursery data set (figure 7) properties 2 and 3 are not present in any of the ordering methods. Occurrences of property 4 are practically equal among ordering methods and the same situations is in case property 1. Measurements of the time for nursery data set are shown in the figure 8. We can see that ascending support method and lexicographical order have very similar results. From the same reasons as for mushroom we observe longer performance time for dCharm with descending support order.

For the car data set there are no occurrences of property 1 (figure 9). Occurrences of property 4 are also the same among all ordering methods. However in case of ascending support and lexicographical order there are present occurrences of property 2, which frequency is also almost even within that two ordering methods. In the descending support order there are additionally occurrences of property 3, which implies the longest time of algorithms performance for this method. Figure 10 confirm that conclusion. Also measurements for ascending support and lexicographical order are very similar.

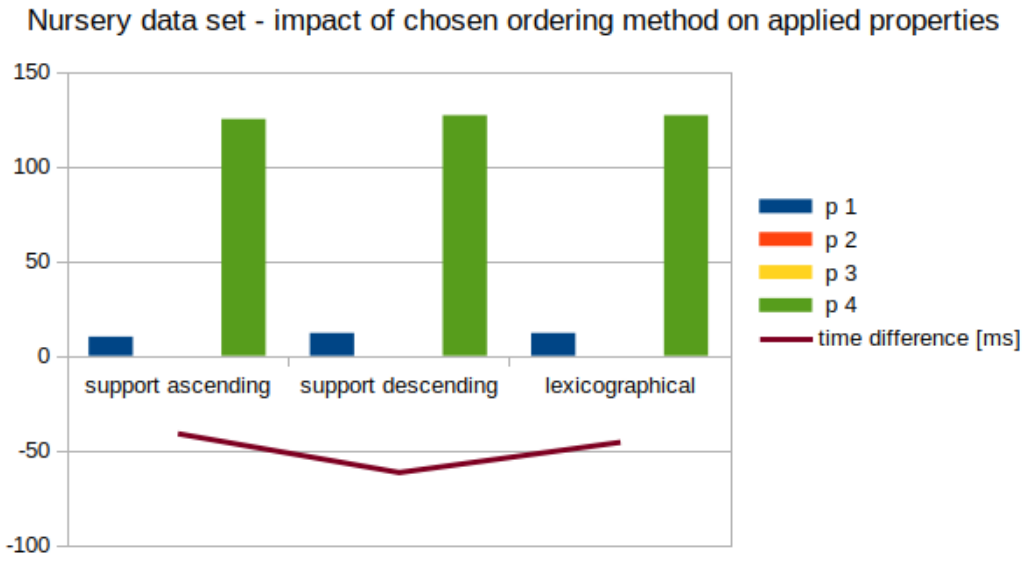

Figure 7: Relationship between chosen ordering function on nursery dataset and the number of occurrences of particular properties.

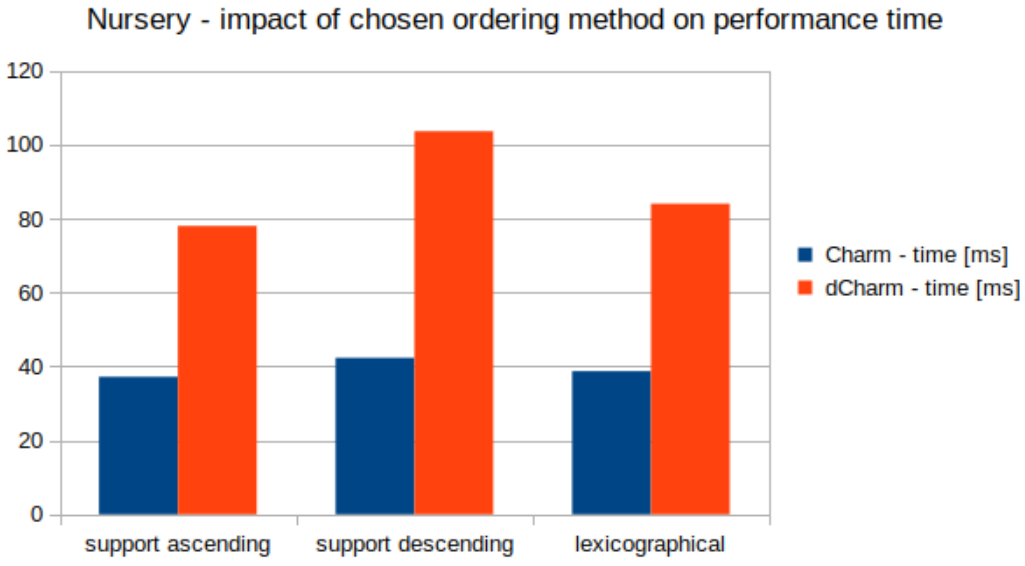

Figure 8: Relationship between chosen ordering function on nursery data and time performance of Charm and dCharm.

7.3 Evaluation of optimization connected with prior mining frequent itemsets of size 2

According to the [1], the initialization of the datasets can be optimized. Initialization with one-element frequent itemsets can cause in the worst case $n(n-1)/2$ difference operations where n is the number of one-element frequent itemsets. In case of infrequent items, their difflists may be longer than tidlists, so difference operations can be expensive. In order to solve that problem, in the [1] it is suggested to firstly compute the set of two-element frequent itemsets (with algorithm proposed in [4]) and instead of computing differences of difflists for every two-element candidates, check if they are among previously computed two-element frequent

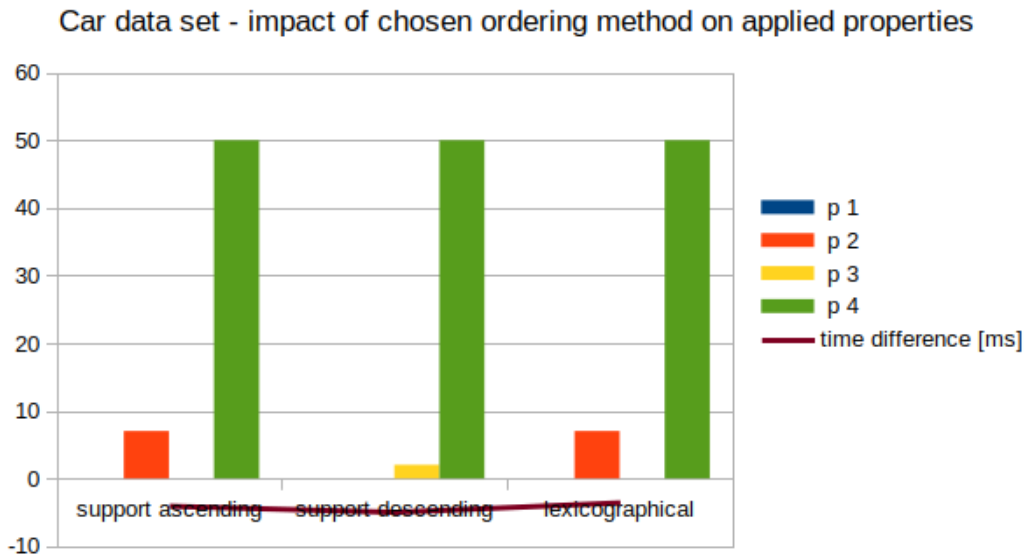

Figure 9: Relationship between chosen ordering function on car dataset and the number of occurrences of particular properties.

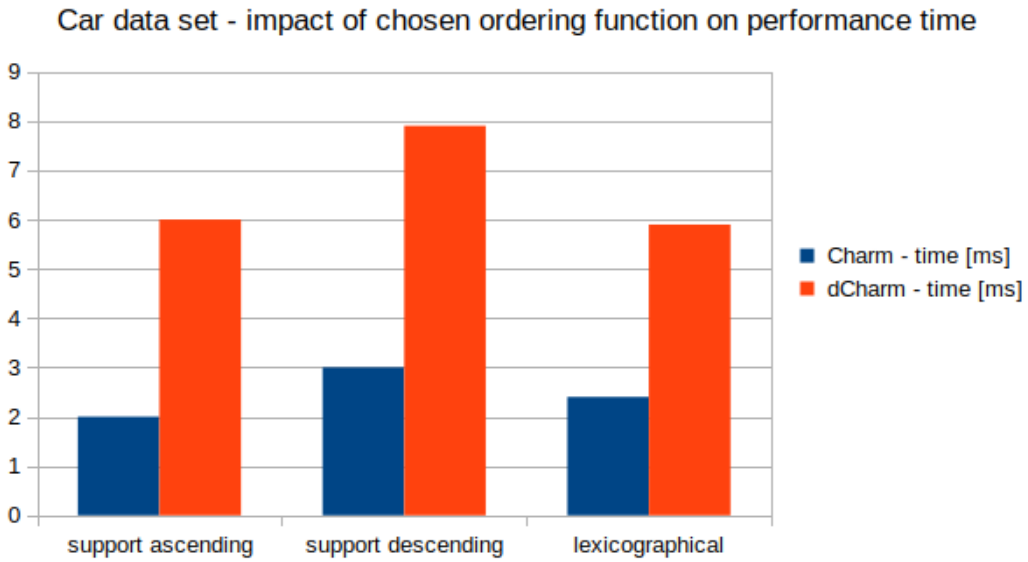

Figure 10: Relationship between chosen ordering function on car data and time performance of Charm and dCharm.

dataset	algorithm	two sets check	FI creation time [ms]	tree time [ms]	total time [ms]
mushroom	Charm	false	23.2	757.5	780.7
mushroom	Charm	true	354.1	806.9	1161.0
mushroom	dCharm	false	50.2	523.4	573.6
mushroom	dCharm	true	385.6	571.0	956.6
nursery	Charm	false	13.3	25.4	38.7
nursery	Charm	true	79.0	17.8	96.8
nursery	dCharm	false	32.7	51.3	84.0
nursery	dCharm	true	98.8	30.8	129.6
car	Charm	false	1.0	1.4	2.4
car	Charm	true	6.0	1.0	7.0
car	dCharm	false	2.9	3.0	5.9
car	dCharm	true	8.0	1.0	9.0

Table 3: Time elapsed on creating frequent itemsets, tree traversal and total time of execution depending on used dataset and usage of frequent two itemsets check with lexicographical order and relative minimum support set to 0.1

itemsets and calculate the difference only if a candidate itemset is there.

As was mentioned before, car and nursery datasets have many itemsets that are infrequent compared to mushroom dataset. Thus improvement is expected for car and nursery datasets. Additionally this way of initializing was checked also for Charm. Table 7 shows the results of measurements. Column *two sets check* determines if two-element frequent itemsets were prior mined to perform checking of candidates. Column FI creation time stands for 'Frequent itemsets creation time'. In case when *two sets check* is false, is shows the time elapsed for mining one-element frequent itemsets for initialization. In case of true - time elapsed for mining one- and two-element frequent itemsets. Figure 11, 12 and 13 are visualizing results for respectively mushroom, nursery and car datasets. Column *tree time* is the time of running the algorithm, without initialization. *Total time* is the sum of *tree time* and *total time*. Small differences in mining times for Charm and dCharm algorithms (in any variant) comes from the fact that in the case of dCharm we calculate difflists based on resulting tidlists, thus a small time overhead.

On the figure 11 we can see slight improvement of the performance of dCharm algorithm (*tree time*), but not very meaningful. The total time is however significantly longer because creation time of the frequent two-element itemsets is inadequately long comparing to the profits of *tree time*. Thus given the total time, that method in case of the mushroom dataset is exacerbating algorithm performance. If the costs of *FI creation time* could be reduced so that total time would be better than initial, it may be considered to use it in case mushroom dataset was bigger. For the Charm algorithm on the other hand the tree time performance is slightly worsen.

Figure 12 presents results for nursery dataset. For dCharm we can see clearer improvement in the *tree time*, comparing to mushroom dataset. However, *FI creation time* is still very long and the total time significantly exceed initial total time. There is also slight improvement for Charm algorithm.

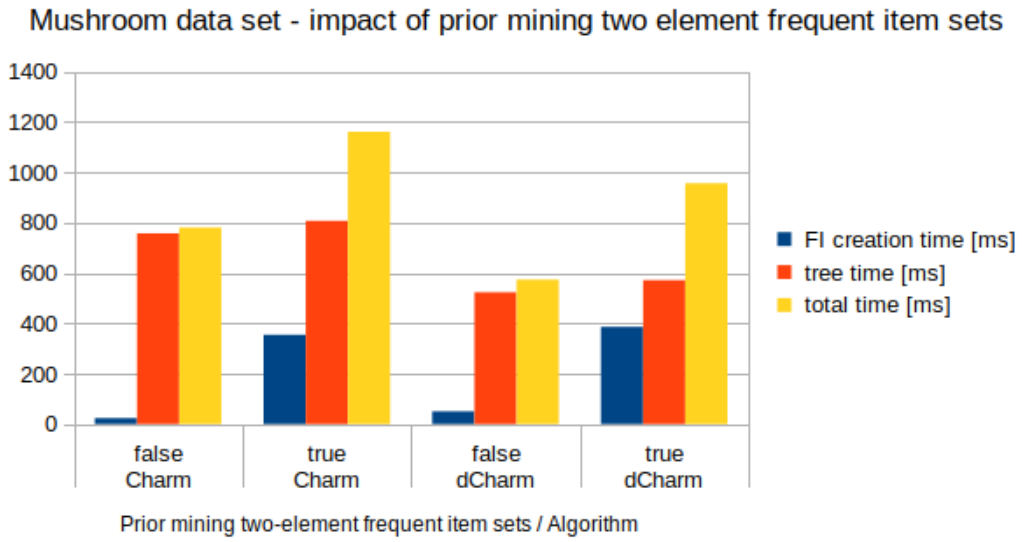

Figure 11: Comparison of performance Charm and dCharm algorithm on mushroom dataset depending on checking two-element frequent itemsets.

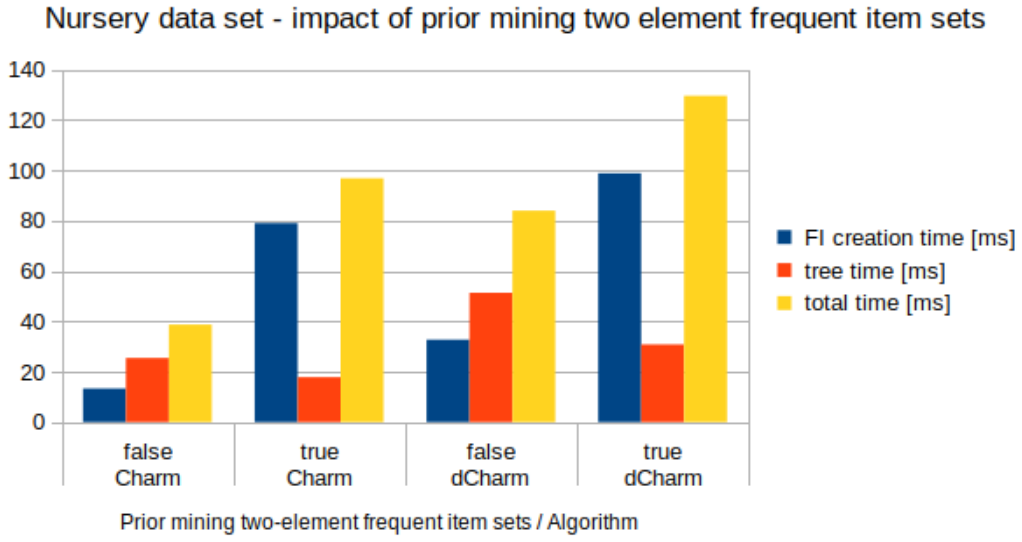

Figure 12: Comparison of performance Charm and dCharm algorithm on nursery dataset depending on checking two-element frequent itemsets.

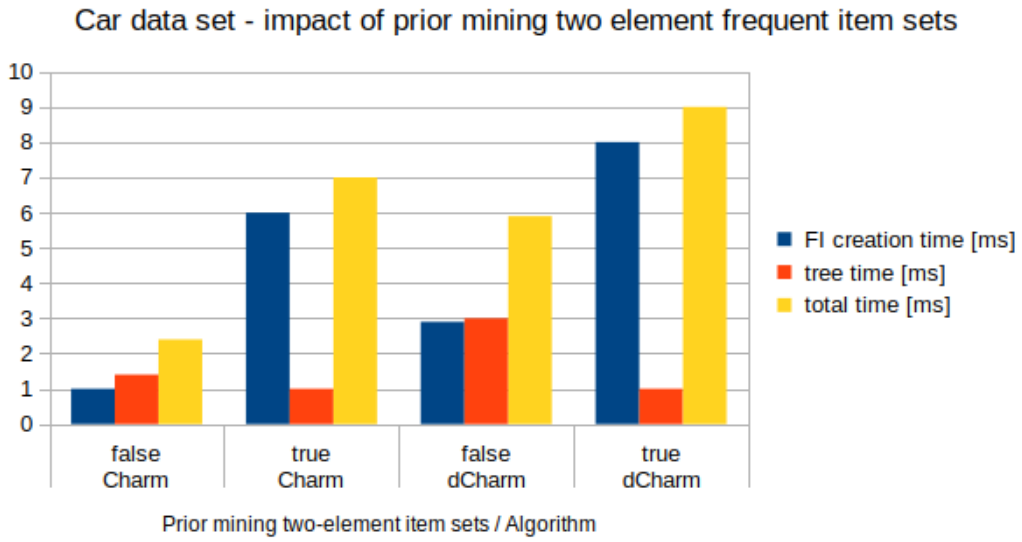

Figure 13: Comparison of performance Charm and dCharm algorithm on car dataset depending on checking two-element frequent itemsets.

For the car dataset on figure 13 we can detect the highest improvement of tree time from all the considered datasets. Also there is slight improvement of Charm performance. Same as in cases of the rest of datasets, total time is worsen.

To sum up, in case of datasets with rare items, examined method of initialization can improve tree traversal time, however it is required that the difference between mining frequent one itemsets and frequent two itemsets is less than difference between algorithm execution time with and without proposed check.

8 Conclusions

From the experiment in section 7.1 we can conclude, that for datasets with rare items performance of dCharm can be worse than performance of Charm. It's due to the fact, that initial tidlist in Charm are shorter for rare items than difflists for the same items in dCharm, so the intersection time is shorter in Charm. However in other cases performance of dCharm can be even two times better than Charm performance.

The most promising method of ordering candidates is ordering by ascending support. Occurrences of property 4, which are causing longer performance of both algorithms, are the rarest among other ordering methods for all datasets. The worst method of ordering is descending support. This method causes worse performance especially for dCharm. For some data sets (nursery, car) ascending support order and lexicographical order result in very similar distribution of properties occurrences.

The experiment in section 7.3 showed, that for datasets with rare items (car, nursery) the initialization with prior mining two-element frequent itemsets can improve performance of the tree traversal time, however it is useless to implement it, unless the cost of mining frequent two-element itemsets is reduced.

References

[1] Zaki, Mohammed & Hsiao, Ching-Jiu. (2002). CHARM: An efficient algorithm for Closed Itemset Mining. Proceedings of the 2002 SIAM international conference on data mining, pages 457-473

[2] Dua, Dheeru & Graff, Casey. (2017) UCI Machine Learning Repository [online] Available at: http://archive.ics.uci.edu/ml [Accessed 18 Jan. 2020]

[3] En.cppreference.com. std::multimap - cppreference.com. [online] Available at: https://en.cppreference.com/w/cpp/container/multimap [Accessed 18 Jan. 2020].

[4] Zaki, Mohammed. (2000). Scalable algorithms for association mining. IEEE transactions on knowledge and data engineering, volume 12, number 3, pages 372-390