Faculty of Electronics and Information Technology

**Warsaw University of Technology**

**EDAMI Project**

**Subject**

Implementation of “*dEclat” algorithm in Python language*

|  |  |
| --- | --- |
| **Authors** | **Repository** |
| Matysiak Piotr  Tureczek-Zakrzewska Maja | <https://github.com/pmatysiakq/edami-dEclat> |

**Table of contexts**

[**Introduction** 3](#_Toc93284133)

[Aim of the project 3](#_Toc93284134)

[Project structure 3](#_Toc93284135)

[**Phase 1 – Basic dEclat implementation** 4](#_Toc93284136)

[1. Selection of implemented algorithm 4](#_Toc93284137)

[2. Description of selected algorithm 4](#_Toc93284138)

[3. Pros and cons analysis 4](#_Toc93284139)

[4. Pseudo-code of the algorithm 5](#_Toc93284140)

[5. Implementation of first version of an algorithm 6](#_Toc93284141)

[6. Testing 9](#_Toc93284142)

[6.1 Experiment #1 9](#_Toc93284143)

[6.2 Experiment #2 11](#_Toc93284144)

[**Phase 2 - Improved dEclat implementation** 16](#_Toc93284145)

[1. Introduction 16](#_Toc93284146)

[2. Implementation 16](#_Toc93284147)

[3. Algorithm performance 18](#_Toc93284148)

[**Phase 3 – Association Rules discovery** 20](#_Toc93284149)

[**Conclusions** 20](#_Toc93284150)

[References 21](#_Toc93284151)

# **Introduction**

## Aim of the project

Our goal within this project was to discover and do frequent itemset mining by implementing, improving, and testing dEclat algorithm in Python on Twitter data.

We used two different sources of data

1. **Live Twitter data** downloaded using Twitter Developer
2. **“All Trump’s Twitter insults (2015-2021)”** database downloaded from Kaggle[3]

Live Twitter data were used in first experiment in phase 1, and we used them to test our implementation of dEclat algorithm and ensure our implementation works correctly (by comparing our results to Java dEclat implementation)

All Trump’s Twitter insults (2015-2021) dataset was used in second experiment, where we aimed to preprocess that data, prepare transactional database, make few frequent itemsets mining iterations - providing different minimal support, collect some parameters and visualize results

To visualize data, we used **“matplotlib**” and “**network”** packages. We chose **hasse-like diagrams** for visualizing frequent item-sets among min support.

All data was presented as **raw tweets**, which, by using python built-in methods and regex were cleaned up and divided into **transactions**. Each transaction consists at least two words, and words duplication within one transaction is not allowed.

Then we have created a transactional database which contains the transactions.

This **transactional database instance**, along with **minimal support** (min-supp) parameter, which indicates a threshold of acceptance are the input parameters towards our algorithm

While improving the algorithm we have focused on **reducing memory usage and run time** of our algorithm.

## Project structure

Project was divided into 3 phases:

1. **Phase 1**- the longest one- here we started from gaining theoretical knowledge – we got familiar with the definition of algorithm, we have performed its pros and cons analysis, proposed the algorithm’s pseudo-code, and finally implemented basic version of algorithm, based on Philippe Fournier-Viger Java implementation [1].
2. **Phase 2-** in this phase we have improved the algorithm, based on "An improvement for dEclat algorithm” article [4]. Our goal was to improve the algorithm in the way, that it would run faster and use less memory. We decided to use following method: “When about a half of newly created tidsets met a changing condition, then in next iteration diffsets will be used”
3. **Phase 3-** in this phase we focused on finding best association rules

All three phases will be explained in detail in next chapters of this document.

# **Phase 1 – Basic dEclat implementation**

## Selection of implemented algorithm

In the first phase we had to implement a simple version of the dEclat algorithm. Our implementation should be based on the Philippe Fournier-Viger Java implementation [1] and paper written by Mohammed Zaki and Karam Gouda [2].

## Description of selected algorithm

dEclat algorithm was introduced by Mohammed Zaki in 2001 in “Fast Vertical Mining Using Diffsets” paper. Being one of vertical mining algorithm and using the diffset format (the difference of two sets), dEclat algorithm has drastically reduced the running time and memory usage of the Eclat algorithm, thus Eclat algorithm using diffset format is called dEclat and it is considered to be more efficient variation of the Eclat algorithm.

Both mentioned algorithms are used for discovering frequent item-sets in a transactional database. The main difference, between them, is a structure which is implemented on top of them - dEclat uses "difflists" to represent differential lists of transaction identifiers, which means that it stores information about transactions which are not present in the set.

This algorithm requires two input parameters:

* data in the form of "transactions"
* minimal support (min-supp) parameter, which indicates a threshold of acceptance.

In simplified terms, we can say that this algorithm is all about computing the support of the item-sets and comparing with min-supp parameter. As output, we expect a group of items, which are called "frequents item-sets".

A group of items is frequent when they are present together in at least "min-supp + 1" transactions, which means that their support is greater than min-supp parameter.

## Pros and cons analysis

The limitations of the algorithm are that the items in transactions are assumed to be sorted by lexicographical order and each transaction in a set is unique - there are no duplicate transactions in a single itemset.

It is neither the only, nor the best algorithm to find frequent item-sets, but it is definitely one of the best. The advantage of dEclat is due to vertical representation of the data - it uses “depth first search” for discovering frequent item-sets.

The main advantage of the vertical format is support for fast frequency counting via intersection operations on transaction ids (tids) and automatic pruning of irrelevant data.

The main problem with these approaches is when intermediate results of vertical tidlists become too large for memory, thus affecting the algorithm scalability

The most important advantage of dEclat algorithm is that it's faster than many of other algorithms and at the same time needs less memory comparing with e.g. Apriori or Eclat algorithms.

Looking at disadvantages, we may point out, that dEclat algorithm is suitable for a dense database, but degrades with spare database, and it needs to switch between tidset and diffset for a sparse database

## Pseudo-code of the algorithm

dEclat is a simple algorithm, thus its pseudo-code is also short. Pseudo code was copied from [2]. In the code we define a few parameters:

|  |  |
| --- | --- |
|  | Initial data-set |
|  | Element present in |
|  | Union of and candidate item-sets |
|  | Difflist of element |
|  | Support of element |
|  | New data-set composed of elements which are frequent |
|  | Minimal support indicating if candidate is frequent or not |

Pseudo-code of dEclat algorithm [2]:

|  |
| --- |
|  |

## Implementation of first version of an algorithm

Initially we have implemented dEclat algorithm only based on the article, but this implementation wasn’t efficient enough and it could be hard for us to improve that implementation in Phase 2 of the project. That’s why we have reimplemented our algorithm. Now it’s mainly based on Philippe Fournier-Viger Java implementation. This new implementation can be found in git repository:

|  |
| --- |
| https://github.com/pmatysiakq/edami-dEclat/tree/master/dEclatV2 |

Our implementation consists of few python files:

* parser/**ParseTTData.py**

This file implements class **Parser**. The goal of this class is to create input file for the class **TransactionalDatabase** which is described later.

Before converting raw tweets into transactional database, we had to collect some datasets from Twitter. We have used two different sources of data:

1. **Live Twitter data downloaded using Twitter Developer API** – we have implemented methods which allow us to fetch tweets by users or tags and save as text files. Example raw tweets looks like shown below - one line represents a single tweet.

|  |
| --- |
| BREAKING NEWS: Australia announces that it may soon begin charging “irresponsible” unvaccinated residents who believe “rubbish” on the internet more for COVID hospital care in order to stop sticking responsible taxpayers with their hospital bills. RT IF YOU SUPPORT THIS!  They lied. I’m so glad I did not get to the “vaccine”, which does NOTHING to stop the spread of Covid. The rest of you are clowns for allowing them to get away with such blatant lies. https://t.co/16AGxzvbWW |

1. **Dataset from Kaggle website** – we downloaded “All Trump’s Twitter insults (2015-2021)” database. Then we extracted the text of the tweets and saved in raw format.

To make described data sources possible, we have implemented few static methods. It may be treated as prerequisites for moving on to data Parsing. When we have collected necessary data in raw format, we can move on to Parsing.

The instance of class **Parser** takes a file with raw tweets as an input. Then it makes use of python built-in methods and regex to clean up data and divide raw tweets into transactions. Each transaction consists of at least 2 words, we don’t allow duplicate words in the single transaction.

For implementation details see GitHub repository, below we present 4 steps which were performed to generate input data files for dEclat algorithm. Steps 1-3 allow us to create files with raw tweets and step 4 creates file which will be provided for **TransactionDatabase** instance.

|  |
| --- |
| **if \_\_name\_\_ == "\_\_main\_\_":**  **# STEP 1**  ***# Use tweepy to fetch 200 newest "elonmusk" tweets and save* as *tweets\_raw/tweets-elonmusk.txt***  Parser.get\_200\_tweets\_by\_user("elonmusk")  **# STEP 2**  ***# Use tweepy to fetch some tweeter data and save as tweets\_raw/db-covid.txt***  words = ["covid", "vaccine", "covid-19", "quarantine", "restrictions", "phizer", "moderna", "astrazeneca",  "fake covid", "wuhan", "coronavirus", "health", "pandemic", "virus", "corona", "stayhome", "lockdown",  "unvaccinated", "omicron", "sars-cov-2", "death", "antibodies", "plandemic"]  Parser.search\_tweets\_by\_tags(words, "covid")  **# STEP 3**  ***# Parse trump tweets downloaded from Kaggle and save as tweets\_raw/trump-tweets.txt***  Parser.parse\_trump\_dataset()  **# STEP 4**  ***# From raw tweets create databases which will be provided as input for dEclat algorithm***  input\_path = r"tweets\_raw/db-covid.txt"  output\_filename = "../input\_data/covid-transactions.txt"  Parser(input\_file=input\_path, output\_file=output\_filename)  input\_path = r"tweets\_raw/tweets-elonmusk.txt"  output\_filename = "../input\_data/elonmusk-transactions.txt"  Parser(input\_file=input\_path, output\_file=output\_filename)  input\_path = r"tweets\_raw/trump-tweets.txt"  output\_filename = "../input\_data/trump-transactions.txt"  Parser(input\_file=input\_path, output\_file=output\_filename) |

* db/**TransactionDatabase.py**

This file implements class **TransactionDatabase**. An instance of that database is provided as input for dEclat algorithm, along with **min\_sup** parameter.

We define a few methods for ***class TransactionDatabase -*** these two are the most important:

def parse\_data\_from\_file(self, file\_path):

# As input we provide text file which consists of 2 parts

# 1st part is transactional database

# 2nd part represents dictionary which will be used to translate integer #

# items into real words

Excerpt from input file is presented below. One line represents single transaction. This input file is generated using **class Parser** which was described earlier.

|  |
| --- |
| [omitted]  2752 170 915 6140  1031 6166 23 6167 6168 6170 6169 157 289 1191 3248 2356 309 182 439 1465 827 2752 986 1375  2752 1057 4967 424 903 6175 26 6171 6172 6173 6174 159  ==== DICTIONARY ====  {"1": "announces", "2": "australia", "3": "begin", "4": "believe", "5": "bills",…omitted} |

def translate\_integers\_into\_words(self, file\_path):

# Used to translate frequent item-sets represented by integers into frequent

# item-sets which consist of real words. Input file is generated by dEclat

# algorithm

Above method allow us to translate found frequent itemsets just like shown in table below:

|  |  |  |
| --- | --- | --- |
| 14 #SUP: 1142  21 #SUP: 961  29 #SUP: 1667  62 #SUP: 953  699 #SUP: 1159  2922 #SUP: 1251  699 2922 #SUP: 1042 |  | they #SUP: 1142  have #SUP: 961  that #SUP: 1667  with #SUP: 953  news #SUP: 1159  fake #SUP: 1251  news fake #SUP: 1042 |

* algorithms/DEclat.py

As said before, our dEclat implementation is based on Java example [1]. As a root of an algorithm we can consider two methods:

def run\_algorithm(self, output: str, database: TransactionDatabase, minsupp: float):

# This may be called the first phase of an algorithm. Here we initialize all

# necessary variables, we create dictionary of tidsets, calculate frequent

# itemsets of length 1 and 2. When processing items of length 1, conversion

# from tidsets to diffsets is done.

# In this method we call method “process\_equivalence\_class”, which is then

# called recursively until we find all frequent itemsets which supports

# are greater than “minsupp” parameter.

def process\_equivalence\_class(self, prefix: [], prefix\_length: int, support\_prefix: int, equivalence\_class\_items: [], equivalence\_class\_tidsets: []):

# This is the second phase of the declat algorithm. Here we process newly

# created equivalence class and we create new equivalence classes which are

# put as input for recursive call. It happens until all frequent itemsets

# are found. When all recursive calls are done, it returns to 1st phase

# where time and memory usage are recorded.

But other methods are also crucial. In particular, there are three methods which differs Eclat and dEclat implementation:

def calculate\_support(self, length\_of\_X: int, support\_prefix: int, tidsetX: set):

# Calculates support of element “Xij” based on support of elements

# “Xi” and “Xj”

def perform\_AND\_first\_time(self, tidset\_I: set, support\_I: int, tidset\_J: set, support\_J: int):

# In our implementation at the beginning we calculate tidsets and then

# we convert tidsets to diffsets while considering items “Xi” and “Xj”

# of length 1

def perform\_AND(self, tidset\_I: set, support\_I: int, tidset\_J: set, support\_J: int):

# For frequent itemsets greater than 1, we calculate new diffsets base on

# diffsets of items “Xi” and “Xj” rather than tidsets

DEclat class also implements methods to save output to the file , print statistics if needed and perform experiment, which will be described below. For more details please check out GitHub repository.

* data\_visualization/**Ploter.py**

This file implements methods which allow us to make use of data we generated and visualize that data using “matplotlib” and “network” packages.

## Testing

In Phase 1 of this project, we would like to perform 2 experiments.

I. The goal of first experiment is to test out implementation of dEclat algorithm on simple input data. We will make use of transactional database, which we will generate on our own. To make sure that our implementation is working properly, we will use already mentioned Java dEclat implementation and we will compare the results.

II. The second experiment will be more sophisticated. We will use dataset mentioned in section 5 - “All Trump’s Twitter insults (2015-2021)” which is available on Kaggle website. The goal is to preprocess that data, prepare transactional database, make few frequent itemsets mining iterations - providing different minimal support, collect some parameters and visualize results. If possible, we will try to infer something directly from frequent itemsets.

In the Phase 2 we will repeat the experiment #2 using improved dEclat implementation and then we will write conclusions.

## Experiment #1

As already said, we want to make sure that our dEclat implementation is working properly. To test algorithm, we have used ***./dEclatV2/input\_data/elonmusk-transactions.txt*** input file – which can be found in GitHub repository. Excerpt from that file:

|  |
| --- |
| 1 2 3 4 5 6  7 8 9 10 11  12 13 14 15 16 17 18  19 20 21 22 23 24 25 26 27 28 29 30  32 33 31  34 35 36 37 38 39 40 41  24 41 42 43  8 44 45 46 47 48 49 50 51 52 53  […] |

Here we describe how we performed 1st experiment. It is a listing from dEclat.py file.

|  |
| --- |
| if \_\_name\_\_ == "\_\_main\_\_":  ***# We define path to the input file***  input = "../input\_data/elonmusk-transactions.txt"  ***# We define path to the output file***  output = "../output/output-experiment-1.txt"  ***# We set-up min-supp parameter - 5%***  min\_supp = 0.05  ***# We create TransactionDatabase instance using input file***  database = TransactionDatabase(file\_path=input)  ***# We create dEclat instance and run algorithm for specified parameters***  declat = DEclat()  declat.run\_algorithm(output=output, database=database, minsupp=min\_supp)  ***# We print out statistics***  declat.print\_stats()  ***# Additionally, we translate found frequent item-sets to the actual words***  database.translate\_integers\_into\_words(file\_path=output) |

As output we get two text files – found FIs with items as integers and FIs with actual words.

|  |  |
| --- | --- |
| 10 #SUP: 11  27 #SUP: 13  30 #SUP: 13  47 #SUP: 8  67 #SUP: 7  72 #SUP: 14  90 #SUP: 9  252 #SUP: 8 | with #SUP: 11  that #SUP: 13  will #SUP: 13  like #SUP: 8  still #SUP: 7  this #SUP: 14  tesla #SUP: 9  just #SUP: 8 |

We also printed-out message which describes the run of the algorithm:

|  |
| --- |
| ============= dECLAT Based on SPMF Java implemetation - STATS =============  Transactions count from database: 135  Frequent itemsets count: 8  Total time ~ 0.09980201721191406 ms  Maximum memory usage: 0.159989 mb  =========================================================================== |

When we compare our results with Java implementation, we can see that our implementation is working properly. We can say that Experiment #1 PASSED the test.

|  |  |
| --- | --- |
| 10 #SUP: 11  27 #SUP: 13  30 #SUP: 13  47 #SUP: 8  67 #SUP: 7  72 #SUP: 14  90 #SUP: 9  252 #SUP: 8 | Algorithm is running... (08:52:47 PM)  ============= dECLAT v0.96r18 - STATS =============  Transactions count from database : 135  Frequent itemsets count : 8  Total time ~ 4 ms  Maximum memory usage : 7.291709899902344 mb  =================================================== |

## Experiment #2

As we described above, to perform second experiment, we have used dataset mentioned in section 5 - ***“All Trump’s Twitter insults (2015-2021)”*** which is available on Kaggle website [3].

For better understanding what was done, we provide how data preprocessing was done. Initially dataset was a \*.csv file, excerpt:

|  |
| --- |
| "","date","target","insult","tweet"  "1","2014-10-09","thomas-frieden","fool","Can you believe this fool, Dr. Thomas Frieden of CDC, just stated, ""anyone with fever should be asked if they have been in West Africa"" DOPE"  "2","2014-10-09","thomas-frieden","DOPE","Can you believe this fool, Dr. Thomas Frieden of CDC, just stated, ""anyone with fever should be asked if they have been in West Africa"" DOPE"  [omitted] |

This file consists of over 10 000 rows. To make used of that data, we took “tweet” text and saved them to file, which ten was processed using Parser instance, excerpt of newly created file:

|  |
| --- |
| Can you believe this fool, Dr. Thomas Frieden of CDC, just stated, ""anyone with fever should be asked if they have been in West Africa"" DOPE"  Can you believe this fool, Dr. Thomas Frieden of CDC, just stated, ""anyone with fever should be asked if they have been in West Africa"" DOPE"  Big time in U.S. today - MAKE AMERICA GREAT AGAIN! Politicians are all talk and no action - they can never bring us back."  [omitted] |

This “raw tweets” are further preprocessed and as input we get transactional database along with dictionary, which allow us to make translation between items representation – integers and words. Here we present input file for dEclat algorithm, which is a result of data preprocessing.

|  |
| --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18  14 19 20 21 22 23 24 25 26 27 28 29 30  [omitted]  385 1667 2181 701 810 42 396 2861 976 18 8787 470 1751 662 3514 1757  ==== DICTIONARY ====  {"1": "africa", "2": "anyone", "3": "asked", "4": "been", "5": "believe", "6": "dope", "7": "fever", "8": "fool", "9": "frieden", "10": "have", "11": "just", "12": "should",… omitted} |

As experiment we want to perform few iterations of declat algorithm run. Each run will be for different min\_sup parameter (supp values). For each iteration we will collect parameters, namely: min\_sup, total\_time, peak\_memory\_usage, database\_size, fis\_count.

For this purpose, we created special method “perform\_experiment()”. This method is described below:

|  |
| --- |
| def perform\_experiment(self):  # Define different min\_sup values  sup\_values = [0.1, 0.08, 0.06, 0.04, 0.03, 0.02, 0.01, 0.009, 0.008, 0.007, 0.006, 0.005]  # Create file to save results  results = open("../results/trump-experiment.csv", "w", encoding="utf-8")  # Indicate input database file and create new database instance  input = "../input\_data/trump-transactions.txt"  database = TransactionDatabase(file\_path=input)  # describe values in csv file  results.write("min\_sup,total\_time,peak\_memory,db\_size,fis\_count")  # For each support value, run declat algorithm and save results to the scv file  for sup in sup\_values:  output = f"../output/output-declat-{sup}.txt"  start\_time, end\_time, memory, db\_size, fi\_count = self.run\_algorithm(output=output, database=database, minsupp=sup)  total\_time = end\_time - start\_time  results.write(f"\n{sup},{total\_time},{memory},{db\_size},{fi\_count}")  database.translate\_integers\_into\_words(file\_path=output)  results.close() |

The method is called inside DEclat.py file, as shown below.

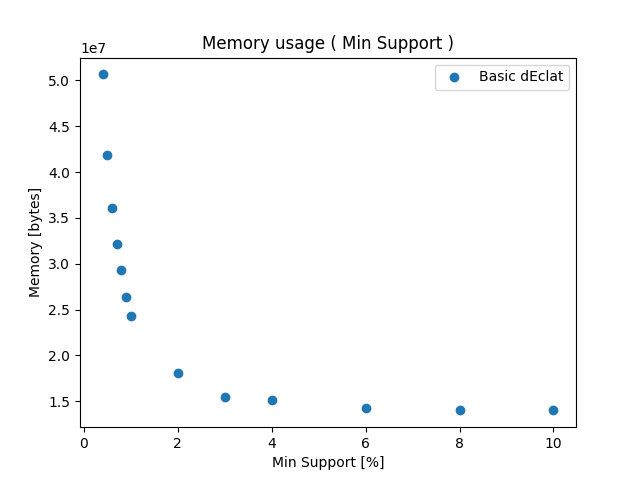
|  |
| --- |
| **if \_\_name\_\_ == "\_\_main\_\_":**  declat = DEclat()  declat.perform\_experiment() |

We saved experiment results to “**trump-experiment.csv**” file. We will use that file to draw plots.

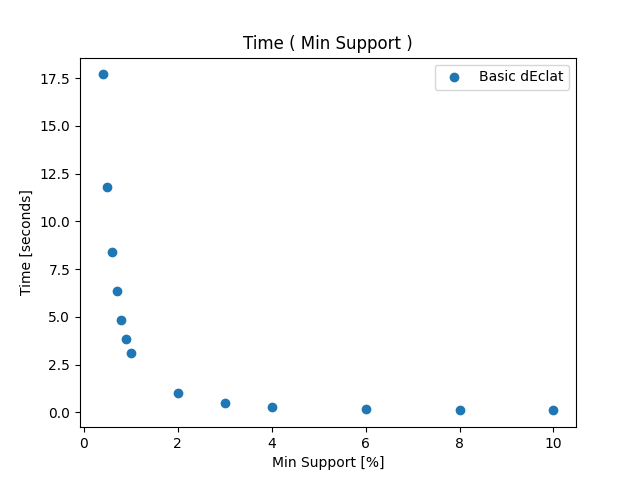
|  |
| --- |
| **min\_sup,total\_time,peak\_memory,db\_size,fis\_count**  0.1,0.10302877426147461,13996236,10336,15  0.08,0.12502837181091309,13995124,10336,25  0.06,0.16403818130493164,14301459,10336,39  0.04,0.2780630588531494,15182210,10336,68  0.03,0.4758181571960449,15450210,10336,107  0.02,1.1078088283538818,18088847,10336,184  0.01,3.2975175380706787,24301738,10336,370  0.009,3.996622085571289,26352385,10336,417  0.008,5.034431457519531,29282705,10336,466  0.007,6.353771448135376,32092638,10336,533  0.006,8.54745626449585,36089233,10336,620  0.005,11.883255004882812,41852078,10336,704 |

We decided to draw three plots based at data obtained during Experiment #2:

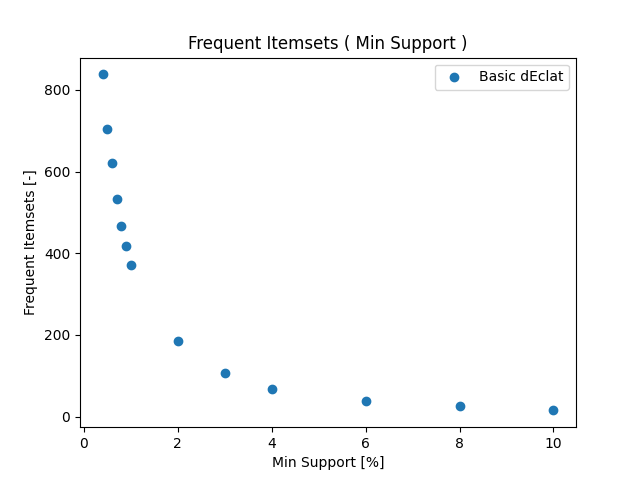
1. Memory consumption as a function of minimum support



1. Experiment duration as a function of minimum support

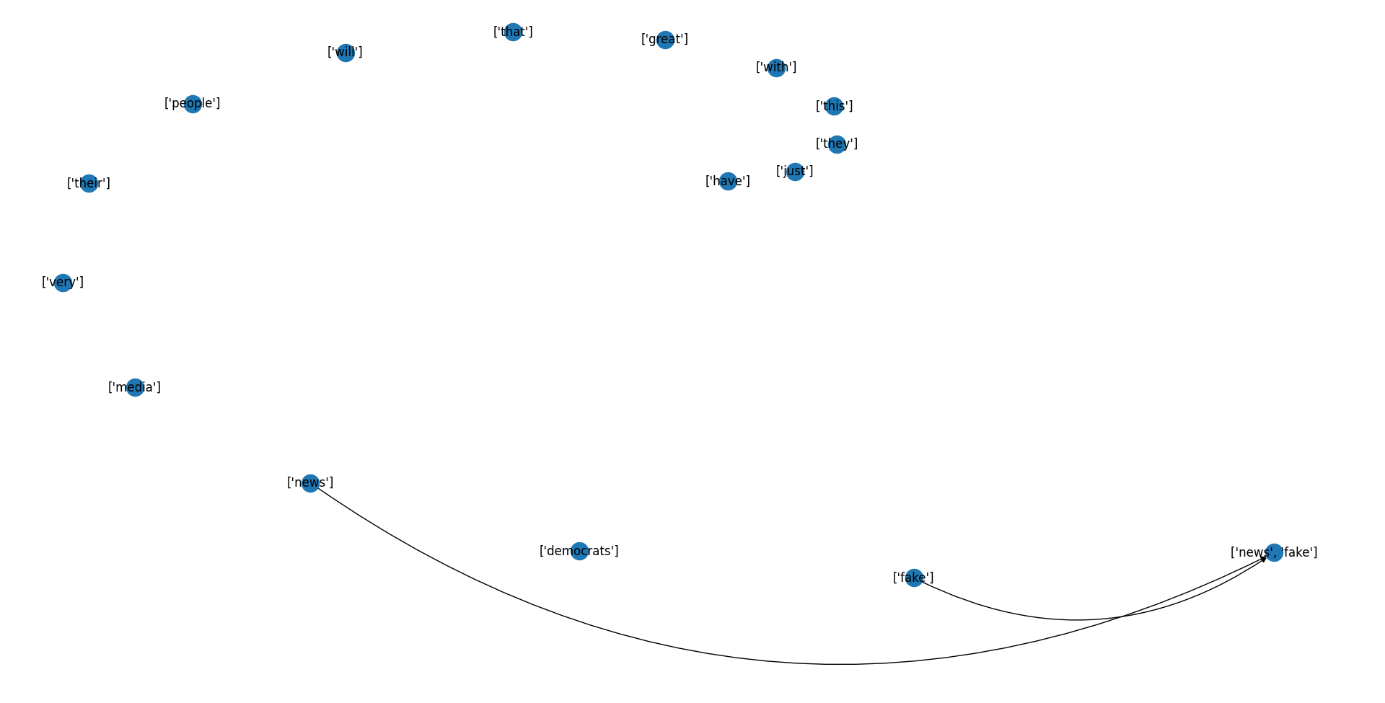


1. Frequent Itemsets count as a function of minimum support

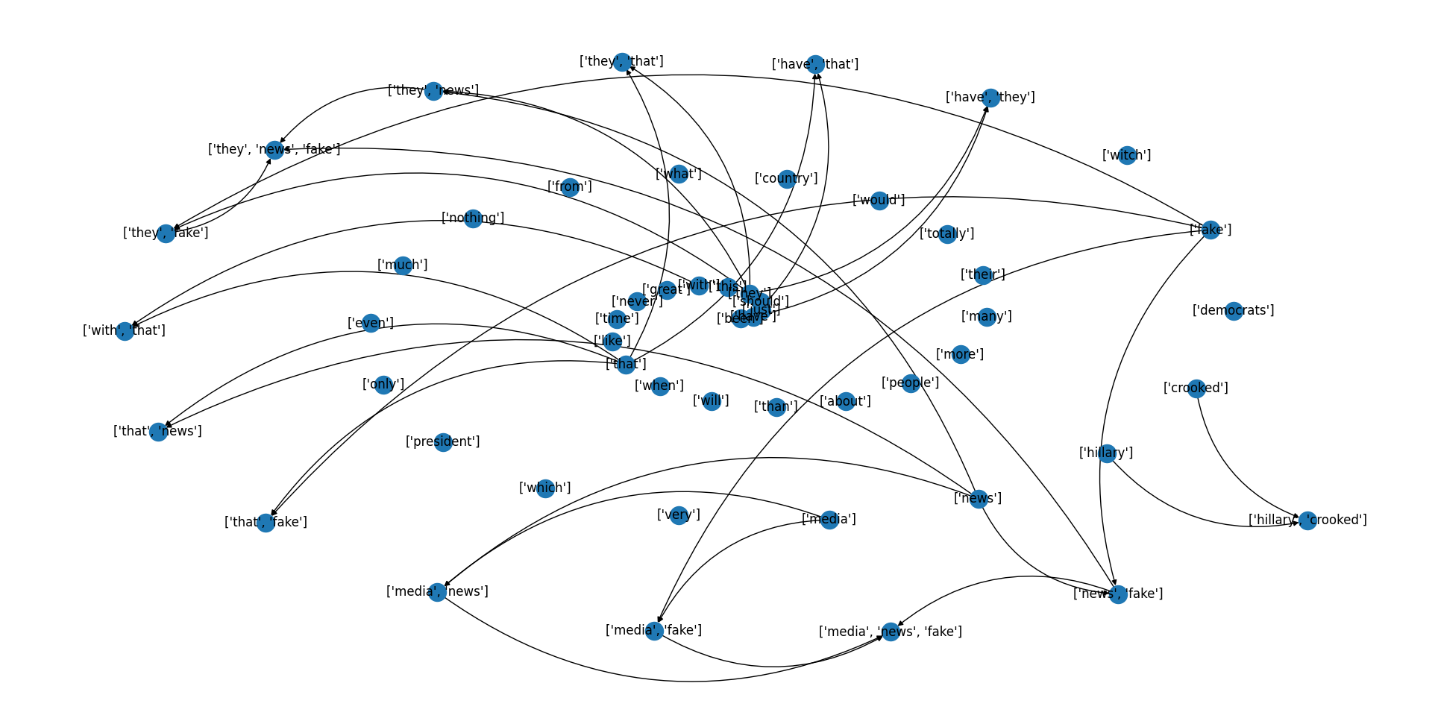


We also prepared hasse-like diagrams for frequent item-sets when min support was:

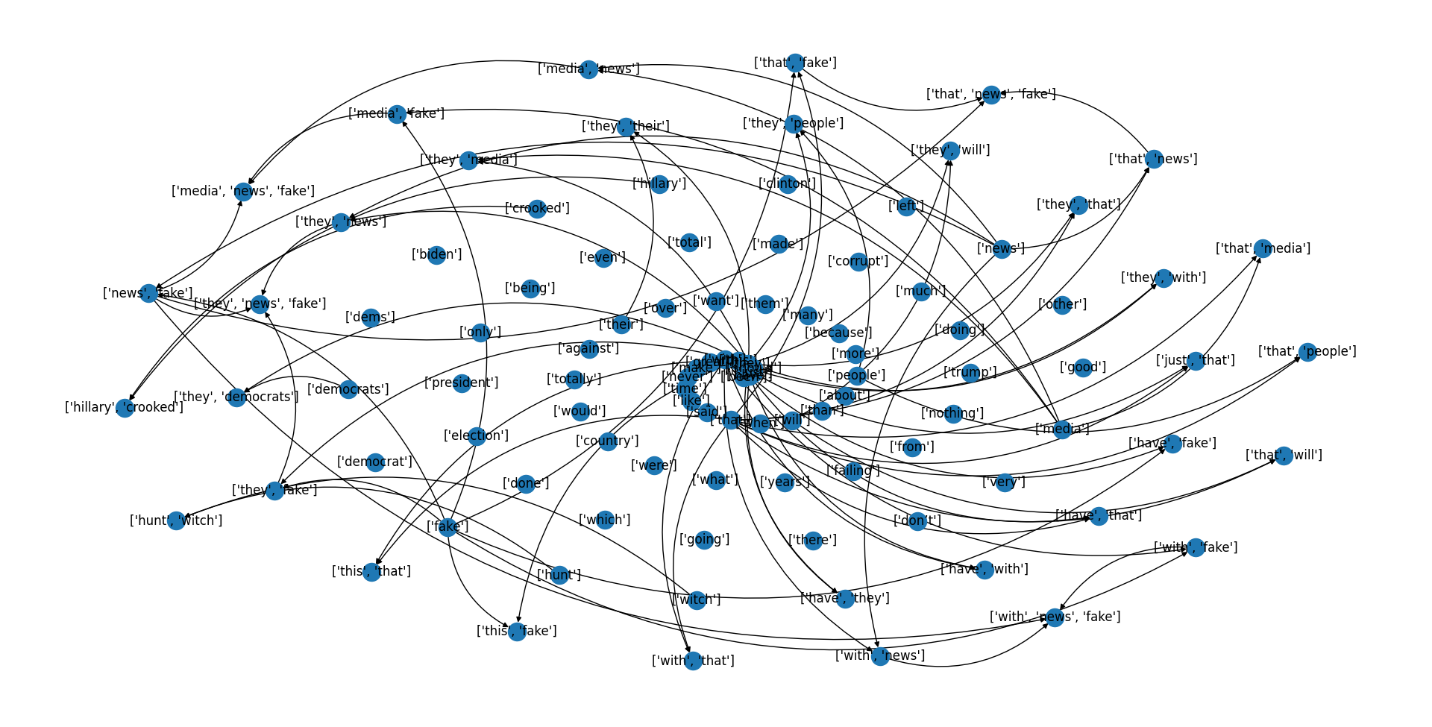
1. min\_sup = 10%



1. min\_sup = 6%



1. min\_sup = 4%



We are aware that this hasse-like diagrams are not exactly what was expected, but it’s very clearly visible what is going on that diagrams.

# **Phase 2 - Improved dEclat implementation**

## Introduction

In the second phase of the project our goal was to make an improvement in the basic version of dEclat algorithm in a way, that it would run faster and use less memory. We decided to implement enhancement discussed in [4]. Authors are aware that diffset format has drastically reduced the running time and memory usage of the Eclat algorithm, which is using tidsets format. But it’s not always the case – in some sparse datasets, diffset format loses its advantage over tidset format and in this case it is suggested to use tidset format at starting and then switch to diffset format later.

In the mentioned paper authors name several different approaches, when we should switch from tidsets to diffsets:

1. When about a half of newly created tidsets met a changing condition, then in next iteration diffsets will be used
2. For each tidset we change from tidsets to diffsets separately when changing condition is met for that single item.

Authors implements option B) - despite the sorting requirements which increases computational complexity, because in this case we are certain that all tidsets met the changing condition, which should maximize efficiency. We decided to implement option A) to find out how much this simple approach can improve efficiency of our basic dEclat algorithm.

In paper [4] is discussed how we can implement change of tidsets to diffsets.

## Implementation

Improved version of dEclat can be found in the git repository along site dEclat implementation:

|  |
| --- |
| https://github.com/pmatysiakq/edami-dEclat/blob/master/dEclatV2/algorithms/impv-dEclat.py |

This shouldn’t be considered as new implementation, it’s rather few improvement in basic version of algorithm. As we said before, while discussing dEclat class, there are few methods that differs dEclat and Eclat implementation – these to calculate support of itemset and these to perform “AND” operation. That’s why we have to adjust mentioned methods and implement checking changing condition.

Methods used when until we are using tidsets are:

def calculate\_support\_eclat(self, length\_of\_X: int, support\_prefix: int, tidsetI: set):  
 return len(tidsetI)

def perform\_AND\_eclat(self, tidset\_I: set, support\_I: int, tidset\_J: set, support\_J: int):  
 tidset\_IJ = set()  
  
 if support\_I > support\_J:  
 for tid in tidset\_J:  
 if tid in tidset\_I:  
 tidset\_IJ.add(tid)  
 else:  
 for tid in tidset\_I:  
 if tid in tidset\_J:  
 tidset\_IJ.add(tid)  
  
 return tidset\_IJ

If changing condition is met, then in next recursive call of **process\_equivalence\_class** method we will change tidsets to diffsets using method shown below. As input, we put tidsets of items “Xi” and “Xj”, but as output we get diffset of “Xij”:

def perform\_AND\_first\_time\_declat(self, tidset\_I: set, support\_I: int, tidset\_J: set, support\_J: int):  
 diffset\_IJ = set()  
  
 for tid in tidset\_I:  
 if tid not in tidset\_J:  
 diffset\_IJ.add(tid)  
 return diffset\_IJ

If condition is met just once, we won’t check it again. Until this moment, to the very end we will use diffset format and methods shown below:

def calculate\_support\_declat(self, length\_of\_X: int, support\_prefix: int, tidsetX: set):  
 if length\_of\_X == 1:  
 return len(tidsetX)  
 else:  
 return support\_prefix - len(tidsetX)

def perform\_AND\_declat(self, diffset\_I: set, support\_I: int, diffset\_J: set, support\_J: int):  
 diffsetIJ = set()  
  
 for tid in diffset\_J:  
 if tid not in diffset\_I:  
 diffsetIJ.add(tid)  
 return diffsetIJ

To check if condition is met, for single comparison, we use method presented below. As input we provide support of item PXY and item PX, when P is prefix and X, Y are elements “I” and “J”.

def check\_condition\_single\_tidset(self, supPXY, supPX):  
 if supPXY >= supPX:  
 return True  
 else:  
 return False

For each elements X and Y we check if condition is met – if yes we increment counter. Then if counter is grater than 0.5 x len(equivalence\_class\_items) we assume that global condition is met, and we provide change in the next iteration.

if declat:  
 pass  
elif condition\_met >= 0.5 \* len(equivalence\_class\_items):  
 is\_declat = True  
 change\_to\_declat = True

Variables “is\_declat” and “change\_to\_declat” are provided as arguments for reimplemented “process\_equivalence\_class” method. If **declat\_first\_time=True,** then in next iteration we will make change from tidsets to diffsets. If **declat=True** it means that support calculation and AND operation will be performed using diffsets.

def process\_equivalence\_class(self, prefix: [], prefix\_length: int, support\_prefix: int, equivalence\_class\_items: [], equivalence\_class\_tidsets: [], declat: bool, declat\_first\_time: bool):

This requires define conditions in the few places, one of them is presented below.

# This is an excerpt from process\_equivalence\_class() method

if declat\_first\_time:  
 tidset\_IJ = self.perform\_AND\_first\_time\_declat(tidset\_I, support\_I, tidset\_J, support\_J)  
 support\_IJ = self.calculate\_support\_declat(length, support\_I, tidset\_IJ)  
elif declat:  
 tidset\_IJ = self.perform\_AND\_declat(tidset\_I, support\_I, tidset\_J, support\_J)  
 support\_IJ = self.calculate\_support\_declat(length, support\_I, tidset\_IJ)  
else:  
 tidset\_IJ = self.perform\_AND\_eclat(tidset\_I, support\_I, tidset\_J, support\_J)  
 support\_IJ = self.calculate\_support\_eclat(length, support\_I, tidset\_IJ)  
  
 # If changing condition is met for processed tidset increment `condition\_met`

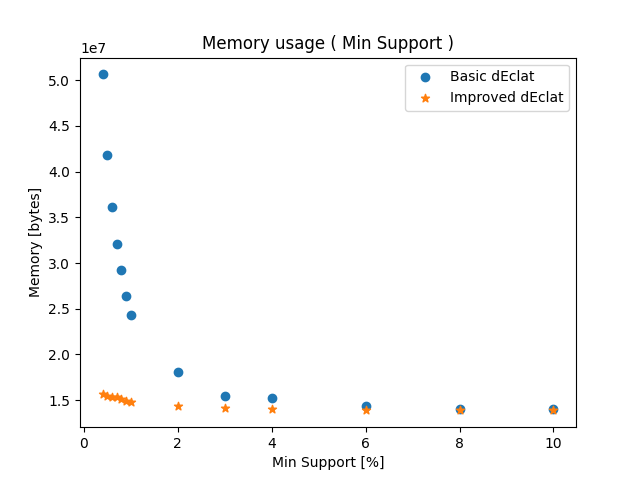
# variable. Do only if diffsets are not already active.  
 if self.check\_condition\_single\_tidset(supPXY=support\_IJ, supPX=support\_J) and not declat:  
 condition\_met += 1

## Algorithm performance

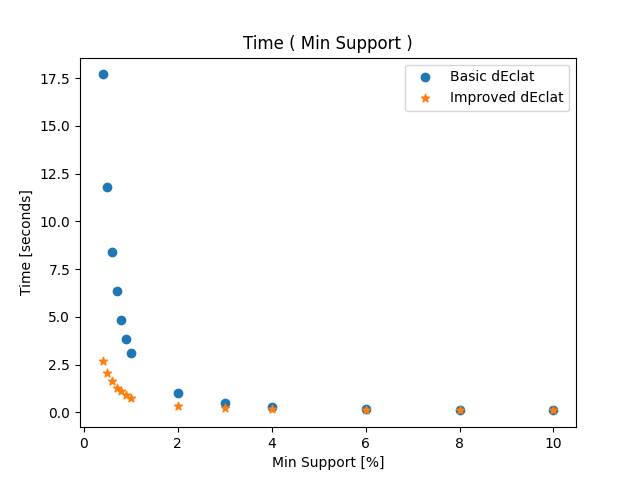
To measure performance of our improved implementation of dEclat algorithm, we have performed the same experiment as for basic version of the algorithm – the same: dataset, environment, min\_sup parameters etc.

As we can see on the charts below, we were able to decrease time usage multiple times, so as memory usage. This version is much more efficient that the initial for the datasets where many items, and many frequent sets are present. The advantage is getting bigger the bigger the **min\_supp** parameter is.

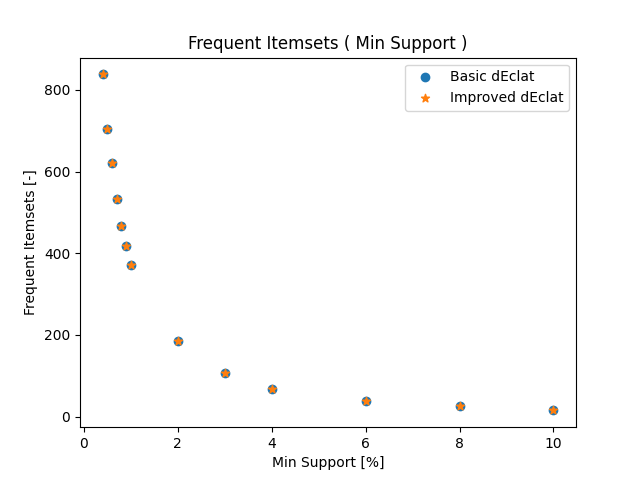
1. Memory consumption as a function of minimum support



1. Experiment duration as a function of minimum support



1. Frequent Itemsets count as a function of minimum support



# **Phase 3 – Association Rules discovery**

## Introduction

In this phase of the project, we aimed to discover association rules by searching data for frequent itemsets. Hence, to discover patterns by using a certain criterion under Support and Confidence to define what the most important relationships are. Our goal was to find association rules with the best confidence.

## Implementation

We decided to continue checking on ***“All Trump’s Twitter insults (2015-2021)”*** dataset.

Way of discovering rules can be found in the git repository along site dEclat implementation:

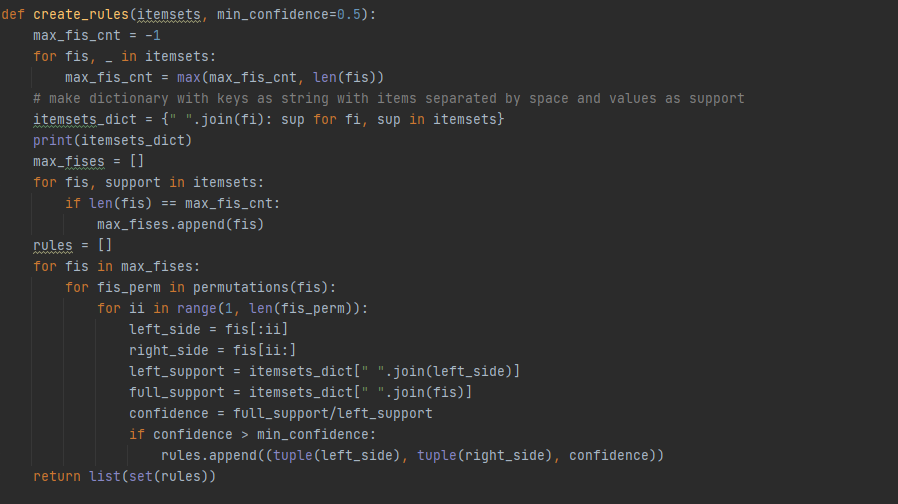
|  |
| --- |
| https://github.com/pmatysiakq/edami-dEclat/blob/master/association\_rules/rules.py |

This, however, is a direct continuation of Phase 2 experiments. As input files we have used already obtained in phase 2 output files (can be found here):

|  |
| --- |
| https://github.com/pmatysiakq/edami-dEclat/tree/master/dEclatV2/output |

from which we were able to use frequent itemsets, and their support.

Method of finding best rules is described on the picture below:



At first, we were checking all association rules, but as we were sure the method works correctly, we decided to value only rules with confidence level above 50%.

Firstly, we have made a dictionary with keys as string with items and values as support.

Then,

## Testing

# **Conclusions**

There are many different algorithms, which allows frequent item-sets discovery - dEclat is one of the most efficient, but of course there exists various improvements of the basic implementation. It’s an algorithm based on Eclat. These two use different data structures to represent database. Eclat uses tidsets, and dEclat diffsets. In general, diffsets are much better that tidsets, but in many sparse datasets it’s not a rule.

Our implementation of dEclat shows that we can find pretty interesting correlations in different datasets.

First of all, our first test has showed, that our implementation is working properly and in an efficient way. Our results were even better than these achieved by using Java implementation, which may indicate that Python is more efficient in memory allocation.

The second experiment have shown that in almost random datasets – like somebody’s tweets, we can find interesting correlations.

Moreover, we can see the dEclat limitations. When we decreased minimum support parameter, then the memory and time required to complete computations increased exponentially. To be more precise, increase of discovered frequent item-sets caused this exponential demand for resources.

The second phase, of the project, have showed that combination of dEclat database representation (diffsets) and Eclat database representation (tidsets) drastically improve efficiency.

In our experiments, we have implemented basic improvement but memory usage and running time decreased multiple times, when compared to basic dEclat implementation. It’s possible to further optimize an algorithm, and get better results, but is no the case for our project.

We ha

This project improved our awareness of data analysis possibilities and have showed that it’s not so hard to perform the first data analysis. We have also improved our programming skills and learned to collect live data from Twitter.

# References

[1] <http://www.philippe-fournier-viger.com/spmf/Eclat_dEclat.php>

[2] <https://www.philippe-fournier-viger.com/spmf/dEclat_dCharm.pdf>

[3] https://www.kaggle.com/ayushggarg/all-trumps-twitter-insults-20152021

[4] Trieu, Tuan A., and Yoshitoshi Kunieda. "An improvement for dEclat algorithm." *Proceedings of the 6th International Conference on Ubiquitous Information Management and Communication*. 2012.