Proceedings of the Federated Conference on
Computer Science and Information Systems pp. 425–434

DOI: 10.15439/2015F137
ACSIS, Vol. 5

Correlation clustering by contraction

László ASZALÓS, Tamás MIHÁLYDEÁK
University of Debrecen
Faculty of Informatics
26 Kassai str., H4028 Debrecen, Hungary
Email: {aszalos.laszlo, mihalydeak.tamas}@inf.unideb.hu

***Abstract*—We suggest an effective method for solving the problem of correlation clustering. This method is based on an extension of a partial tolerance relation to clusters. We present several implementation of this method using different data structures, and we show a method to speed up the execution by a quasi-parallelism.**

I. Introduction

THE principle of minimum total potential energy (MTPE) is a fundamental concept used in physics, chemistry, biology and engineering. It asserts that a structure or body shall deform or displace to a position that minimizes the total potential energy. This concept could be used at other fields, too. In this paper, we show its application in clustering. The clustering is an important tool of unsupervised learning. Its task is to group the objects in such a way, that the objects in one group (cluster) are similar, and the objects from different groups are dissimilar, so it generates an equivalence relation: the objects being in the same cluster. If we want to apply the principle MTPE, then we can say that the objects aim to achive a situation in which they are in a cluster containing minimal number of dissimilar, and maximal number of similar objects. In the last fifty years, many different clustering methods were invented based on different demands.

Correlation clustering is a new method, Bansal at al. published a paper in 2004, proving several of its properties, and gave a fast, but not quite optimal algorithm to solve the problem [1]. Naturally, correlation clustering has a predecessor. Zahn proposed this problem in 1965, but using a very different approach [2]. The main question is the following: which equivalence relation is the closest to a given tolerance (reflexive and symmetric) relation? Bansal et al. have shown, that this is an NP-hard problem [1]. The number of equivalence relations of n objects, i.e. the number of partitions of a set containing n elements is given by Bell numbers B_n, where $B_1 = 1$, $B_n = \sum_{i=1}^{n-1} \binom{n-1}{k} B_k$. It can be easily checked that the Bell numbers grow exponentially. Therefore if $n > 15$, in a general case we cannot achieve the optimal partition by exhaustive search, thus we need to use some optimization methods, which do not give optimal solutions, but help us achieve a near-optimal one.

This kind of clustering has many applications: image segmentation [3], identification of biologically relevant groups of genes [4], examination of social coalitions [5], improvement of recommendation systems [6] reduction of energy consumption [7], modelling physical processes [8], (soft) classification [9], [10], etc.

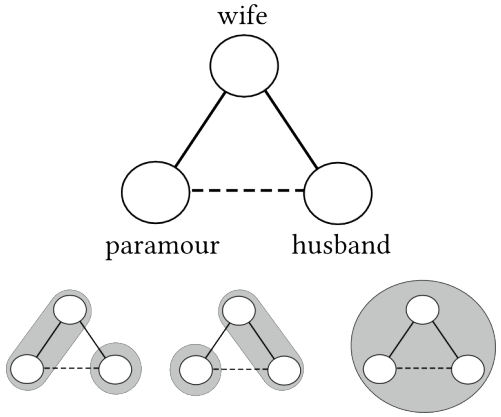

Fig. 1. Minimal frustrated graph and its optimal partitions

At correlation clustering, those cases where two dissimilar objects are in the same cluster, or two similar objects are in different clusters are treated as a conflict. Thus the main question could be rewritten as: which partition generates the minimal number of conflicts? It can be shown, that in the general case—where the transitivity does not hold for the tolerance relation—the number of conflicts is a positive number for all partitions of a given set of objects. Let us take a graph on Figure 1, where the similarity is denoted by solid, and dissimilarity by dashed lines. In case of persons, the similarity and dissimilarity are treated as liking or disliking each other, respectively. Mathematically, the similarity is described as a weighted graph, where 1 denotes similarity, and -1 denotes dissimilarity. As the absolute value of these weights are the same, thus it is enough to only use the signs of the weights. Hence the graph is called a *signed graph* in the literature. The lower part of this figure shows the three optimal partitions from the five, each of them containing only one conflict. As all partitions of the graph on Figure 1 have at least one conflict, it is called a *frustrated graph*, and this one is the smallest such graph.

If the correlation clustering is expressed as an optimization problem, the traditional optimization methods (hill-climbing, genetic algorithm, simulated annealing, etc.) could be used in order to solve it. We have implemented and compared the results in [11]. With these methods the authors were able to determine a near optimal partition of signed graphs with 500 nodes.

978-83-60810-66-8/$25.00

In this paper we introduce a new method which was invented directly to solve the problem of correlation clustering, and can use the specialities of the problem. The main idea is extremely simple, but it needs several witty concepts, to get a fast and effective algorithm.

Before going into details, let us see the hierarchical clustering—where the solution is usually received as a result of a series of contractions. The user needs to choose a dissimilarity metric of clusters. These metrics are based on the distance of the members (objects) of the clusters. Next, the hierarchical clustering constructs a hierarchy, which starts with singleton clusters, and each higher level is earned by joining two clusters of the previous level. The last level consists of one cluster which contains all the objects. This clustering is an automatic process, where the user selects one level of this hierarchy (and the equivalence relation generated by it), and uses accordingly.

In case of correlation clustering the user has limited options. If the hierarchy, according to the tolerance relation is generated, the cost function (i.e. the number of conflicts) can be calculated for each level (for each partition) of the hierarchy, and we take the minimal. This does not mean, that we reach the optimal partition for each tolerance relation. This hierarchy has n levels (n partition) from the B_n possible ones, e.g. if n is 500, then we have 500 levels, and $1.6 \cdot 10^{844}$ different partitions. Therefore we need to choose carefully which partitions to add to the hierarchy.

There are two ways to construct the hierarchy: bottom-up and top-down. In the latter, we split one cluster into two. If the original cluster contains k objects, there are $2^k - 2$ ways to split it, thus the full search is not feasible. There are other methods to find a good split, but the hierarchy construction by splitting is not so common. The bottom-up way is based on joining clusters. There are $k(k-1)/2$ ways of joining clusters, and after the contraction of two clusters we update the pre-calculated distances of clusters accordingly, as if we had k clusters before contraction.

In this paper we will use Python programs to present the algorithms. The Python programming language is perfect tool for prototyping:

- Python code listings are shorter than pseudo codes, because we can use program-libraries without long explanations or repetition of well-known algorithms.
- Python has high level data structures (list of set, set of lists, etc.) which simplify the programs.
- Although Python programs are slow, we are not interested in exact running times, but in the time rates: the effect of replacing an algorithm with a better one?

We tested programs on one core of a 2.3 GHz double core processor under Linux and Python 3.4.0. The times are given in seconds.

The first implementation uses the most trivial data structure to store a graph: the adjacency matrix. The adjacency matrix of a tolerance relation contains elements -1 and 1 only, so its graph is total. The asymptotic behaviour of the correlation clustering of tolerance relation is known from [8]. If we allow zero values in the adjacency matrix, i.e. we have a partial tolerance relation, the behaviour of the clustering changes. The more zeros are in the adjacency matrix, the bigger the difference in behaviour. These behaviours are not yet described or explained mathematically, computer experiments for many objects are needed to discover the exact tendencies. Hence for us the most interesting graphs are the sparse graphs. In general, it is easy to implement the generation of Erdős–Rényi type random graphs [12], but it is hard to ensure that it generates a connected graph for small probability parameter p. Therefore in our experiments we used Barabási-Albert type random graphs [12] (we refer to them as BA graphs in the following) where the connectivity follows from the algorithm of the generation.

The structure of the paper is the following: Section 2 explains the correlation clustering and shows the result of joining two clusters. In Section 3 we present a naive contraction method. Next we specialize the method for sparse graphs. In Section 5 we give the quasi-parallel version of the specialized algorithms. Finally we discuss our plans and conclude the results.

II. Correlation clustering

In the paper we use the following notations: V denotes the set of the objects, and $T \subset V \times V$ the tolerance relation defined on V. We handle a partition as a function $p : V \to \{1, \ldots, n\}$. The objects x and y are in a common cluster, if $p(x) = p(y)$. We say that objects x and y are in conflict at given tolerance relation T and partition p iff value of $c_T^p(x, y) = 1$ in (1), i.e. if they are similar and are in different clusters, or if they are dissimilar and in the same cluster.

$$c_T^p(x,y) \leftarrow \begin{cases} 1 & \text{if } (x,y) \in T \text{ and } p(x) \neq p(y) \\ 1 & \text{if } (x,y) \notin T \text{ and } p(x) = p(y) \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

We are ready to define the cost function of relation T according to partition p:

$$c_T(p) \leftarrow \frac{1}{2} \sum c_T^p(x,y) = \sum_{x<y} c_T^p(x,y) \tag{2}$$

As relation T is symmetric, we sum c_T^p twice for each pair, or restrict the summing in order to use each pair only once. The objects are usually represented with numbers, hence we can calculate the cost function as the last part of (2) shows.

Our task is to determine the value of $\min_p c_T(p)$, and one partition p for which $c_T(p)$ is minimal. Unfortunately this exact value cannot be determined in practical cases, except for some very special tolerance relations. Hence we can get only approximative, near optimal solutions.

The statistical software R has 6 different distance functions for determining distance of objects. In our case the tolerance relation replaces these metrics. The same software has 8 different cluster distance functions. None of them is suitable for our needs.

Let us see, what is the result of joining two clusters according to the cost function. Let A and B be these clusters,

moreover let us denote the partition before and after joining by p and q, respectively.

1) If $\{x, y\} \cap (A \cup B) = \emptyset$, then $c_T^p(x, y) = c_T^q(x, y)$ holds.
2) If $x \in (A \cup B)$ and $y \notin (A \cup B)$ (or in reverse), then $c_T^p(x, y) = c_T^q(x, y)$ holds.
3) If $x \in A$ and $y \in B$, then $c_T^p(x, y) = 1 - c_T^q(x, y)$ holds. As objects x and y are from different clusters, $p(x) \neq p(y)$. So if $c_T^p(x, y) = 1$, then $(x, y) \in T$, but after the joining $q(x) = q(y)$, hence $c_T^q(x, y) = 0$. Similarly if $c_T^p(x, y) = 0$, then $(x, y) \notin T$, so $c_T^q(x, y) = 1$.
4) Finally, if $x, y \in A$ (or $x, y \in B$), then $c_T^p(x, y) = c_T^q(x, y)$.

To determine $c_T(q)$ if we know $c_T(p)$, it is enough to calculate the difference $c_T(q) - c_T(p)$. From the previous list it is obvious that we only need to take into account the third item. Before the contraction between clusters A and B there were $\#\{(x, y) | x \in A, y \in B, (x, y) \in T\}$ conflict, and after the contraction these conflicts disappear. But $\#\{(x, y) | x \in A, y \in B, (x, y) \notin T\}$ new conflicts are created by the contraction. The change is the difference of these numbers.

If we treat the conflict of two objects as a distance, then the aggregating function is the difference of two sums. At hierarchical clustering the closest clusters are joined. For us, the contraction method is a greedy algorithm, and we promote the joining of those possible ones which produce maximal profit, which mostly decreases the cost function. These two ideas are confluent, because the difference mentioned before becomes negative, and we join those clusters where the absolute value of this difference is maximal, so where the difference is minimal. Unfortunately, according to its negativity we cannot call the difference as *distance*.

We note that our programs use notations of Bansal, and if the tolerance relation holds for two objects, the number denoting it is positive. From this, it naturally follows that the program uses the differences $c_T(p) - c_T(q)$, i.e. the profits of joining. We only execute the contraction if this difference is positive.

The upper part of the Figure 2 shows a relation. For the sake of simplicity of the picture, this relation is a partial tolerance relation. The middle part of the figure displays the "distances" of the singleton clusters earned from the relation. We formally define it in the next chapter. Finally the lower part of the figure shows the updated distances when some clusters are contracted. These distances are the superposition of the predecessor clusters.

III. Contraction method

We can now define the Contraction method for correlation clustering. Algorithm 1 is implemented in Python. In the following formulae the brackets refer to the lines of the code. The implementations use the routines of https://www.ics.uci.edu/~eppstein/PADS/UnionFind.py for handling disjoint sets: it contracts two clusters in line 8. We do not present the preprocessing of neither the tolerance relation, nor of the result, but the clustering phase is emphasized.

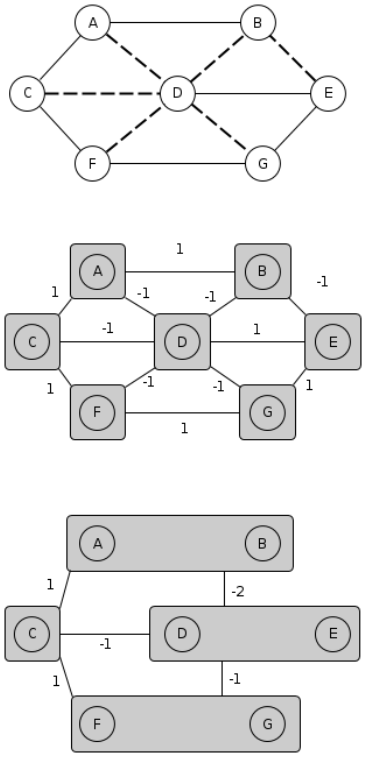

Fig. 2. The partial tolerance relation, the generated "distances" at the beginning and after several contraction steps.

1) Construct a "distance" matrix D based on a tolerance relation T:

$$d(i, j) \leftarrow \begin{cases} 1 & \text{if } (i, j) \in T, \\ -1 & \text{if } (i, j) \notin T, \end{cases} \tag{3}$$

   Although $d(i, i)$ would be 1 by definition, we set up 0, in order to avoid contracting any clusters by themselves. We denote the preprocessing T and setup of D with dots. [line $3 - 4$]
2) Now, each object—as a singleton cluster—in one-one bijection with rows and columns of D. Select one maximal element of matrix D, and then its row and column coordinates will refer to the clusters to join. Let these be x and y [line 6].
3) If the maximal element is not positive, the algorithm ends [line 7].
4) Otherwise add to column x the column y (contraction), next delete column y, i.e. fill it with zero [lines $9 - 12$]. Then repeat this for the rows.
5) Continue from Step 2. [line 13]

The statistics describing the running time of Algorithm 1 is given in Table I. Here the columns denote graphs with different sizes, and the rows tagged by the value of q. This parameter gives the ratio of fulfilment of tolerance relation T among objects. If this number is small, only a few clusters can be joined, so the Contraction method finishes soon and

Algorithm 1 Naive contraction with matrix

```
import numpy; import UnionFind
N=2000; q=0.4
d = numpy.zeros( (N,N), dtype = numpy.int)
...
uf = UnionFind.UnionFind()
x,y = numpy.unravel_index(d.argmax(),d.shape)
while d[x,y] > 0:
    uf.union(x,y)
    for z in range(N):
        d[x,z] += d[y,z] ; d[z,x] += d[z,y]
    d[...,y] = 0; d[y,...] = 0
    d[x,x] = 0
    x,y = numpy.unravel_index(d.argmax(),
                d.shape)
```

TABLE I
RUNNING TIMES FOR COMPLETE SIGNED GRAPHS.

	100	500	1000	2000	5000
$q = 0.1$	0.017	0.598	3.015	16.080	174.511
$q = 0.5$	0.027	0.773	3.736	19.152	198.269
$q = 0.9$	0.029	0.785	3.789	19.226	197.325

the solution contains many but quite small clusters. But if this ratio is large, there are lot of possibilities to join clusters, and the Contraction method runs for a long time and gives only a few, but big clusters. It can even occur, that we get only one cluster, containing all the elements.

In our previous article we examined the behaviour of contraction of tolerance relation on n objects [13]. For this, we took 101 different values for q from 0 to 1, and for each q we tested the clustering for 10 different relations (graphs), and used their average. By using this implementation to calculate everything for 2000 objects, took more than 5 hours. These are independent calculations, so they can be run in parallel. We note that this program uses the low-level routines of Numpy extensions, which speeds up the execution according to the conventional Python implementation.

This 8-line-long naive implementation does not take into account such details, which are obvious for the reader. As we contract rows and columns, and delete one row and one column (fill up with zeros), in the future the program will not put any non-zero number into these rows and columns, thus it is unnecessary to include these elements of the matrix in the calculations. We could tag them, and later skip them in the cycles. It could be a better solution, however, to change the deletion of a row and a column by swapping them with the last row and column, respectively. Next, we can logically reduce the size of the matrix D, and set the upper limit in cycles to the size of the matrix.

Up to here, for any pair of objects the tolerance relation either holds or does not, i.e. the objects are either similar or dissimilar. However, in some cases we have no knowledge about either, thus the relation is partial. (A well-known partial relation is the ordering of n-tuples, where the relation holds only if one tuple Pareto dominates the other.) We can use (1) for any partial tolerance relation. By definition, if two objects are neither similar nor dissimilar, there is no conflict between them. The partial tolerance relations are visualized by signed graphs: if two objects are comparable then there is an edge between them, otherwise there is not, like in Figure 2. If the relation holds, the weight of the edge is 1, if not, its weight is -1. If the relation is not defined between the objects, the weight of the non-existent edge could be defined as 0. From these weights we can construct the "distance matrix" D for any partial relation.

TABLE II
RUNNING TIME OF ALGORITHM 1 ON RANDOM ER GRAPHS WITH PARAMETERS $(2000, p_i)$, WHERE THE RATE OF THE POSITIVE EDGES IS q

	$q = 0.1$	$q = 0.5$	$q = 0.9$
$p_1 = 0.9$	16.870	19.294	19.418
$p_2 = 0.5$	18.040	19.240	19.372
$p_3 = 0.1$	18.148	19.112	19.371

It is not suprising that Algorithm 1 could be used for partial tolerance relations without any modification. We generated several partial relations with 2000 objects, to test our algorithm. The graphs of the partial tolerance relations were Erdős–Rényi type random graphs, where any two nodes are connected with probability p_i. Next, the weight of this edge (if exists) would be 1 with probability q and -1 with probability $1 - q$.

The algorithm is the same, it takes the same steps, therefore we can assume, that we get very similar results for clustering. But the data in Table II refutes this assumption. If many edges are missing from the complete graph (p_i is small) and most of the edges are negative (q is small), then the algorithm does not stops after a few steps, as did it for a bigger p_i. Similarities force the contractions. The more 1s in D, the more contractions are executed by the algorithm, which takes time. The more -1s in D—the contractions are obstructed—the less contractions are executed by the algorithm, and the process stops early. The zeros (i.e. missing edges) here decrease the effects of 1 and -1: the number of contractions fall between the two extreme cases (when all zeros are replaced with 1 and -1, respectively).

As we wrote before, Néda at al. described the asymptotic behaviour of correlation clustering for complete graphs [8], thus the computer simulations in this case are not challenging. In case of any signed graph (including the partial tolerance relations) we have conjectures only about the asymptotic behaviour. Néda at al. simulated correlation clustering of BA graphs with 140 nodes in 2009, which we extended to 500 nodes in 2014 [13], and even though the results are impressive, they are not quite sufficient to see the tendencies.

The BA graphs are sparse graphs, they have $O(|V|)$ edges. It is superfluous to reserve $O(|V|^2)$ memory cells in a matrix to store these edges. In the next section we show a memory efficient algorithm to solve the correlation clustering problem for sparse graphs.

IV. Refining the method

The previous Python implementation shows the crucial questions of the method:

- Q1 For constructing a greedy algorithm we need the most profitable contraction, i.e. from the stored and updated (recalculated) values we need to select the biggest one, or one of the biggest ones.
- Q2 We need to be able to reach the element $d_{x,y}$ of the matrix D, to read it, to update it, or to delete it.

As the previous implementation stores matrix D as a two-dimensional array (matrix), the solution of Q2 is trivial, however to solve Q1, we need a full search in D.

In BA graph with ten thousands of nodes, a node is only part of a few edges. This means that in its row of "distance" matrix D there are only a few non-zero elements (which are interesting at calculations). Therefore it is worth to represent the matrix D as a sparse matrix. There are several ways to store a sparse matrix in the memory. The simplest way is to store the list of $(row,\ column,\ value)$ triplets. This storage type is called a *coordinate list*.

In case of using a coordinate list, the solution to Q1 does not change substantially, because a full search is needed in triplets. (Now we have no element without valuable information.) Yet, the solution of Q2 becomes more complicated. Until now, $d_{x,y}$ was reachable in constant time (with pointer-arithmetic). Now the triplets of D are in a list, so in a worst case scenario, a full-search is needed. If this list is ordered, a binary search is enough, but it is very hard to preserve this ordering during contractions:

- If $d_{x,z} = -d_{y,z}$, then after contraction $d_{x,z}$ becomes 0, but we do not want to store this element, so we need to delete it.
- If $d_{x,z} = 0$ and $d_{y,z} \neq 0$, then after the contraction, one new item is created and needs to inserted into the list, while an old one is to deleted.

The hash table could help us solve Q2 effectively, because its speed is near to $O(1)$ at insertion. The hash table is a possible (and fast) implementation of the data structure *associative arrays* (dictionary in Python). Algorithm 2 is the reimplementation of the Contraction method with an associative array.

As before, we only indicate the phase of preprocessing [line 23]. This code is much longer than the previous one, we need to care about more details. While in the previous program, we retrieved the indices of the biggest element of D with one complicated instruction, we needed to take it into pieces here: from the values of the associative array the algorithm selects the biggest one [line 27], and collects the keys belonging to this maximal value [lines 29–30]. From these keys it chooses a pair. We refer the members of this pair as i and j. We wish to reuse the part of this code, which describes the contraction, therefore we implemented the contraction step as a function [lines 2–20]. For similar reasons, the pair (i, j) is stored in an associative array, called s. Our construction guaranties, that $i < j$, and at contraction

Algorithm 2 Contraction by associative array

```
import UnionFind                                                    1
def contraction(d, s):                                              2
    d2 = {}                                                         3
    for pair, value in d.items():                                   4
        x,y = pair                                                  5
        if x in s:                                                  6
            x = s[x]                                                7
        if y in s:                                                  8
            y = s[y]                                                9
        if x == y:                                                  10
            continue                                                11
        if x > y:                                                   12
            x,y = y,x                                               13
        if (x,y) in d2:                                             14
            d2[(x,y)] += value                                      15
            if d2[(x,y)] == 0:                                      16
                del d2[(x,y)]                                       17
        else:                                                       18
            d2[(x,y)] = value                                       19
    return d2                                                       20
                                                                    21
N=1000; q=0.2                                                       22
d = ...                                                             23
uf = UnionFind.UnionFind()                                          24
if len(d) < 2:                                                      25
    return                                                          26
max_d = max(d.values())                                             27
while max_d > 0:                                                    28
    pairs = [pair for pair,value in                                 29
        d.items() if value ==max_d]                                 30
    i,j = pairs[0]; s = {j:i}; uf.union(i,j)                        31
    d2 = contraction(d,s)                                           32
    if len(d2) < 2:                                                 33
        break                                                       34
    d = d2.copy()                                                   35
    max_d = max(d.values())                                         36
```

we keep the smaller one. This means, that upon meeting a key (which is a coordinate-pair), we need to check whether one of it is equal to j (to the key in the associative array s) or not. If it is, then we need to replace it with i (with the value according the key) [lines 6–9]. In the diagonal of D, only zero values are allowed, but we do not store them, so if something gets into this diagonal it is ignored [lines 10–11]. We store the indices by ordering [lines 12–13]. If a known pair occurs, then we update its value, whereas unknown pairs are stored [lines 14–19]. Sometimes one cluster is connected with two different clusters, and their effects are opposite. Remember on Figure 1 the husband who loves her wife (+1) but hates her paramour (-1). If the wife left him with her paramour, these numbers add up, and we get 0. So the best if he forget about this new couple. We can speed up our method by not storing zero values, so we omit this one, too [lines 16–17].

Unfortunately the associative arrays do not allow us to pick items by specific needs, hence, to get all the keys in form $(j, \cdot)$ and $(\cdot, j)$, we need traverse the whole associative array, and check whether the actual item is concerned in the contraction or not. In the BA random graphs, the nodes i and j have only

a few nearby node (connected to it by an edge), hence this traverse needs extra effort (in comparison to direct access of the edges of a node).

The statistic demonstrating Algorithm 2 is in the first data column—denoted with D—in Table IV. The numbers in this table are not seconds or number of conflicts, but rates of benchmarks. Figure 3 and Table III shows the real running times of the bases of the benchmarks. The real running times of Algorithm 2 are much smaller than the running times of Algorithm 1. But if we think about it, we can realize, that these numbers are incomparable. The naive algorithm used total graphs, and a total graph with two thousands nodes has about two millions edges, but a BA random graph has only four thousands. To find the one with the maximal value (Q1) is not the same job, and even the solution of Q2 is easier in the case of associative arrays. We have about five-hundredths many edge, and the running time only one third. To tell the truth Algorithm 2 only uses the conventional Python here—does not use any extension compiled to fast machine code—i.e. overall the execution of the code is much slower.

This programming language enables us to combine different data structures. Hence we can construct one associative array about profits related to a given cluster, and organize these associative arrays into an associative array. This *two level dictionary* is very common in the Python literature. To speed up the solution of Q2, we store one "distance" at two places: in each node's associative array of the according edge. This helps us collect all the edges belonging to the clusters needing to join. This double storage complicates our programs. Moreover, the empty associative arrays generate errors, so we need to check whether the associative arrays become empty, or all the objects are in one cluster.

Here the values of the profits of contractions are distributed among the associative arrays. Hence to find the maximal value we need to search for the maximal value in each associative array, and to select the maximal one among these maximal values (line 5). Lines 7–9 is a list comprehension, in which we traverse all associative arrays, and check, whether there is a key, for which this maximal value is assigned. If yes, we record the pair of the key i of the associative array, and the key j of the maximal value. If all the maximal values have been found, we select one (more precisely the first) key from the keys of the maximal values. This key is a pair, and we refer to its elements by i and j in the following (line 10). We delete the edge belonging to the selected pair (line 11), and we store the keys in the associative array belonging to i (line 12). It is an important step, Python does not allow to modify a data structure while traversing it. (Do not cut off the tree you are standing on!) Similarly, we docket the associative array belonging to j (line 14), and by traversing this associative array we update the weights of the edges in contact to the contracted cluster (lines 16–19). If it is not possible (there is no corresponding (i, z) edge to the edge (j, z)) then we construct the missing edge. When all edges $(j, \cdot)$ are processed, we can delete the associative array belonging to j (line 25), and the all references to this cluster (i.e. $(\cdot, j)$ type edges), too (line 22). If the contracted cluster becomes empty (any strange, sometimes it can happen), we need to delete it (lines 26–27). If only one cluster remains, we can stop the method (lines 28–29).

Algorithm 3 Contraction by associative array of associative arrays.

```
import UnionFind                                                    1
N=1000; q=0.2                                                       2
d = ...                                                             3
uf = UnionFind.UnionFind()                                          4
max_d = max([max(di.values()) for di in d])                         5
while max_d > 0:                                                    6
    pairs = [(i,j) for i,di in d.items()                            7
        for j,value in di.items()                                   8
        if value == max_d]                                          9
    i,j = pairs[0]                                                  10
    del d[i][j]; del d[j][i]                                        11
    di_k = list(d[i].keys())                                        12
    uf.union(i,j)                                                   13
    dj = list(d[j].items())                                         14
    for z,value in dj:                                              15
        if z in di_k:                                               16
            d[i][z] += value; d[z][i] += value                      17
            if d[i][z] == 0:                                        18
                del d[i][z]; del d[z][i]                            19
        else:                                                       20
            d[i][z] = value; d[z][i] = value                        21
        del d[z][j]                                                 22
        if d[z] == {}:                                              23
            del d[z]                                                24
    del d[j]                                                        25
    if d[i] == {}:                                                  26
        del d[i]                                                    27
    if len(d) == 1:                                                 28
        break                                                       29
    max_d = max([max(di.values())                                   30
             for i,di in d.items()])                                31
```

TABLE III
RUNNING TIMES FOR $\overline{D_r^2}$ IN SECONDS

N	0.1	0.5	0.9
100	0.004–0.009	0.004–0.012	0.008–0.015
500	0.076–0.227	0.075–0.227	0.165–0.327
1000	0.483–1.126	0.488–1.267	0.863–1.581
2000	2.370–5.867	2.397–5.927	3.985–8.506
5000	0.230–3.310	3.576–25.155	15.386–36.270
10000	6.205–23.424	15.298–104.581	122.131–265.172

The statistics about Algorithm 3 can be found in the third column—denoted with D^2— of Table IV. From this table it is obvious, that the two level dictionary is more effective than the ordinal dictionary.

In Python the associative arrays are implemented as hash tables. It is well known, that here the deletion is a costly operation, and thus we delete the whole structure in small steps. By examining the handling of the hash table in Pythonic way, we rewrote the code in Algorithm 3 in a such way, that at contraction we do not delete directly from the joined cluster, but we create a new associative array for the joined cluster, and fill it with its predecessors. As deletion from dictionary is solved by replacing data with dummy items, the recreation of

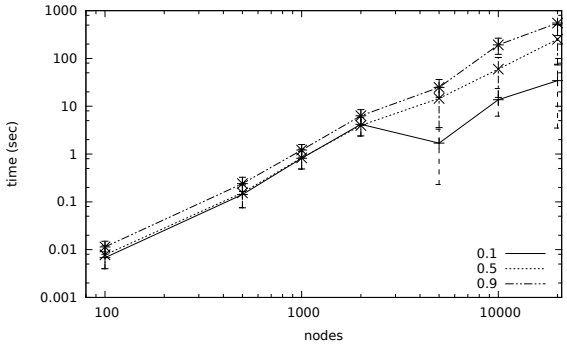

Fig. 3. Running time of implementation $\overline{D_r^2}$ for graphs with different N and q

the joined cluster can free up memory and speed up problem Q2. The deletion from the nearby clusters remains the same. It would be very time consuming to recreate them, too. By the fifth column—denoted with D_r^2— of Table IV this extra effort has no evident profit: the accuracy is almost the same, but the running time is slightly longer.

V. Quasi-parallel variant

The hierarchical clustering sometimes has very natural interpretations. For a given distance function $d : V \times V \to \mathbb{R}$, where $d(x, x) = 0$ for any $x \in V$ Sibson defined a *dendogram* function $c : [0, \infty) \to E(V)$ [14], where $E(V)$ is the set of equivalence relations on V. This function c fulfils the following criteria:

- If $h \leq h'$ then $c(h) \subseteq c(h')$,
- The final value of $c(h)$ [i.e. $c(\infty)$] is $V \times V$ and
- $c(h + \delta) = c(h)$ for any small $\delta > 0$.

In case of a minimal distance (the distance of clusters defined by the minimum of distances of their elements), let $E = \{(x, y)|x, y \in V, d(x, y) < h\}$, i.e. add to the set of nodes V all the edges shorter than h. This is a symmetric relation. Next, take the transitive closure of this relation, which becomes an equivalence relation. Taking these equivalence relations for all $h \geq 0$, we get the dendogram c.

Can we use this idea? Unfortunately not. If we have a chain of objects connected by edges, then their optimal clustering produces pairs and maybe one singleton cluster [15]. In the case of a star topology we got a pair and singletons. Therefore the transitive closure of a (partial) tolerance relation is not suitable for us. But we can try to contract independent pairs. As they are independent, the contraction can be done in parallel! This means that we lose the purely greediness of the contraction method. Moreover the dynamics of the method is changing. If we started by contracting two singleton clusters i and j, for which there was a third cluster k such that $d_{i,k} > 0$ and $d_{j,k} > 0$, then the next contraction used clusters $i \cup j$ and k at algorithms before. Moreover if there was a fourth cluster l, for which $d_{i,l} > 0$, $d_{j,l} > 0$ and $d_{k,l} > 0$, then the following contraction used clusters $i \cup j \cup k$ and l, and so on. In other words the initial combined cluster grows in each steps until it is surrounded with clusters only, for which the profit of the contraction with this giant cluster is negative. If we execute the contractions in parallel, then we have more centres (not so giant clusters).

Algorithm 4 Determining the independent edges

```
def independent(pairs):                                  1
    s = {}                                               2
    random.shuffle(pairs)                                3
    for i,j in pairs:                                    4
        if len({i, j} &                                  5
           (s.keys() | s.values())) == 0:                6
            s[j] = i                                     7
    return s                                             8
```

We do not need to rewrite the whole Algorithm 2 as most of the code is reusable. For example, to get the independent pairs we have the list of best pairs. Previously we used the first pair from the list, now we will use more. Algorithm 4 selects the independent ones. This algorithm gets all best pairs, and constructs an associative array from the independent edges. The algorithm traverses the best edges, and an edge is independent from the stored ones, if its nodes do not occur in the associative array, neither as a key, nor as a value. The original ordering limits the set of independent edges, so we shuffle the starting list, in order for the subsequent runs to give a different results, therefore the clustering becomes indeterministic, and by repeating the whole process it should be possible to choose the best of them.

The Algorithm 5 is a slight modification of Algorithm 2 so we only provide the difference: you need to replace lines 25–36 in Algorithm 2 with the listing of Algorithm 5. The main difference is that Algorithm 2 which allowed only one pair in the associative array, now allows any number of pairs, which need to be independent.

In this case, the question arises: it is worth to complicate things? Based on the first idea, it could speed up the process, because we can omit the superfluous steps, as we do not need to traverse the associative array several times, it is enough only once, to get the same number of contractions. The second

Algorithm 5 Quasi-parallel contraction, associative array

```
if len(d) < 2:                                           1
    return                                               2
max_d = max(d.values())                                  3
while max_d > 0:                                         4
    pairs = [pair for pair,value                         5
        in d.items() if value ==max_d]                   6
    s = independent(pairs)                               7
    for i,j in s.items():                                8
        uf.union(i,j)                                    9
    d2 = contraction(d, s)                               10
    if len(d2) < 2:                                      11
        break                                            12
    d = d2.copy()                                        13
    max_d = max(d.values())                              14
```

TABLE IV
COMPARISON OF RUNNING TIMES AND ACCURACY ON 3/2 BA GRAPHS

	D	$\overline{D}$	D^2	$\overline{D^2}$	D_r^2	$\overline{D_r^2}$
			q=0.1			
100	2.44/1.01	0.88/1.02	1.41/1.04	1.03/0.99	1.47/1.04	1.00/1.00
500	3.68/0.98	1.22/0.99	1.68/1.01	0.96/1.00	1.74/1.01	1.00/1.00
1000	4.12/0.98	1.64/1.00	1.50/1.00	0.99/1.00	1.54/1.00	1.00/1.00
2000	4.48/0.98	1.67/1.00	1.59/1.00	1.02/1.00	1.58/1.00	1.00/1.00
5000	61.79/1.08	0.96/1.00	16.84/1.32	1.03/0.99	17.43/1.32	1.00/1.00
10000	56.66/1.03	0.79/1.00	13.93/1.16	1.07/1.00	14.02/1.16	1.00/1.00
			q=0.5			
100	2.46/0.95	0.79/0.98	1.28/1.02	0.97/0.99	1.36/1.03	1.00/1.00
500	3.77/0.98	1.30/1.00	1.63/1.01	1.01/1.00	1.66/1.01	1.00/1.00
1000	4.37/0.97	1.61/1.00	1.58/1.00	1.01/1.00	1.70/1.00	1.00/1.00
2000	4.26/0.98	1.67/1.00	1.59/1.00	1.02/1.00	1.61/1.00	1.00/1.00
5000	5.43/0.98	1.67/1.00	2.07/1.02	1.04/1.00	2.18/1.02	1.00/1.00
10000	4.68/0.99	1.49/1.01	1.87/1.03	1.05/1.01	1.99/1.03	1.00/1.00
			q=0.9			
100	1.35/1.05	0.88/1.04	0.84/1.00	1.00/1.00	0.88/1.00	1.00/1.00
500	1.61/1.00	0.95/1.01	0.86/1.00	0.99/1.00	0.92/1.00	1.00/1.00
1000	1.85/1.00	1.12/1.00	0.79/1.00	1.01/1.00	0.82/1.00	1.00/1.00
2000	1.66/1.00	1.05/1.00	0.69/1.00	1.00/1.00	0.71/1.00	1.00/1.00
5000	2.29/1.00	1.18/1.00	1.01/1.00	0.97/1.00	1.08/1.00	1.00/1.00
10000	1.53/1.00	1.05/1.00	0.66/1.00	1.09/1.00	0.68/1.00	1.00/1.00

column—denoted by $\overline{D}$—of Table IV shows the numbers of the quasi-parallel variant. It is obvious that this parallel variant is better than the original according to the running time. But if we learned that there is no free lunch, maybe the accuracy of method were worse. Let us see the numbers. By Table IV Algorithm 2 produces less conflict, thus gets closer to the optimum. We measured the whole process at three different values of q. We wish to repeat this investigation in more details. We executed the original (Algorithm 2) and the parallel (Algorithm 5) version of the Contraction method on the same signed graphs. The original graph was 3/2 BA graph with thousands of nodes. The weight of the edges were given randomly according the value of q for 101 different q. After the contraction method we calculated the cost-functions. Figure 4 shows the result. Moreover we summed the values of the cost functions (calculated the "integral" of the curves), and at the parallel version the sum was 94.8–96.0 percent of the original version (these numbers denote conflicts, so the smaller is better here). Five experiments show similar results, so we believe that this is the tendency. This is not a big difference, but disproves our hypothesis. This fact undermines our beliefs in greedy algorithm, so a careful examination is needed. We think, that the giant cluster is the result of an early decision at non-parallel algorithms. This decision could be perfect or bad. Maybe the latter occurs more often. The parallel version postpones the decision, it executes several contractions in parallel, gets smaller clusters, and maybe causes less vital mistakes.

By examining the Table IV, we can see, that the parallel version really is faster than the original. Surprisingly, at different values of q the speed rate is different. For small q we see big differences, while at big q the parallel version will not be twice as fast as the original.

Let us compare this parallel version ($\overline{D}$) with the variant using an associative array of associative arrays (D^2), because until now this was the fastest implementation. If q is small,

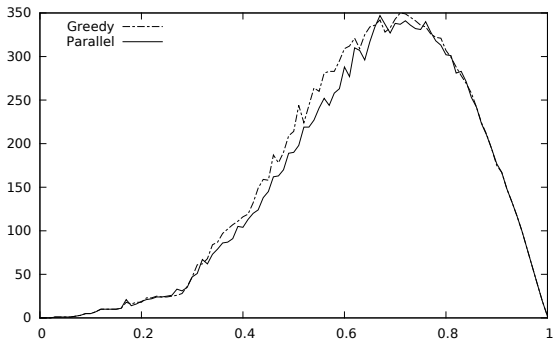

Fig. 4. Cost functions earned by the original and parallel algorithms. x-coordinate denotes the rate q of positive edges and y-coordinate denotes the value of the cost function.

the parallel version is remarkably faster for big N. If q is big, the variant with double dictionary is the clear winner.

Of course, the following question arises: It is worth to construct the parallel version of Algorithm 3, or not? The modification is minimal:

- We need to replace line 10 with a cycle for the independent pairs from `pairs`.
- We need to indent lines 11–29, to treat this lines as the core of the cycle in line 10.

We leave these modifications to the reader, and do not present this as a next code listing. Table IV shows the statistics of this parallel variant—denoted by $\overline{D^2}$. If q is small—we only need a small number of contractions to get the near optimal solution—then the second parallel variant ($\overline{D^2}$) is much faster then its origin (D^2), but for big Ns it is slower than the first parallel implementation ($\overline{D}$). Surprisingly if we need a lot of contraction (e.g. q is big), then the parallel version ($\overline{D^2}$) is much slower than its origin (D). By observing the accuracy

of the last implementation, the tendency is more evident: 84.3–87.7 percent of conflicts of the non-parallel version.

We could continue the comparison of greedy and parallel variants, but we compared the accuracy of the different parallel implementations. The rates of the integrals were 99.9–101.0 percent.

The running times of the three parallel versions are comparable by Table IV. Hence we run the different "parallel" implementations on BA graphs with 20,000 nodes to find the differences. Each implementation solved 15 problems in one hour and a half. The implementation $\overline{D}$ was the slowest, about 8 percent slower than the others. But its accuracy was the best: the difference was less than half percent.

We tried the skip-list data structure for Contraction, where the complexity of the insertion, deletion and search is $O(\log n)$, and not constant. By our statistics this implementation was five times slower than the implementation with double dictionary. The results based on these prototypes suggest for us that the industrial implementation will based on a (double) dictionary, too.

VI. Future plans

There are tools to find the weakness of codes presented in this paper, and their execution could be optimized. Moreover by choosing a different programming language it gives (maybe several magnitudes) faster implementation. We refer to a quote of D. E. Knuth: *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%*. We would like to invent new algorithms and not to patch this ones.

Our main aim was to introduce a simple idea, and its little improvements. We have fulfilled our plans and broke through our former limits of 500 nodes (with significantly higher results), although previously we used faster languages. To solve the correlation clustering problem for bigger and bigger set of objects is challenging for us, so we will continue this path. We believe to go futher we need to use other kind of parallelism: divide and empire.

The zero "distances" of clusters were left along the whole article. We did not examine whether they have any effect to contract clusters where the corresponding distance is zero. This contraction does not decrease and does not increase the number of conflicts *immediately*. It is worth to examine whether this kind of contractions has any future effects, or not. We do not know any examples that could assist this question from neither the natural nor the social sciences.

We can imagine three clusters i, j and k, where $d_{i,j} = 0$ $d_{i,k} = c$ and $d_{j,k} = -c$. It is obvious, that contracting i and j the value of $d_{i\cup j,k}$ becomes zero, so the number of conflicts does not lessen. While by contracting i and k the number of conflicts decreases by c, if $c > 0$. But it is not clear when we have thousands of clusters, the situation is the same, or not. If we have BA random graphs, the zero "distance" clusters, i.e. independent clusters are very common.

VII. Conclusion

We introduced a correlation clustering problem, and extended the (partial) tolerance relation to clusters. Using this concept we have shown the Contraction method, and its several implementation in Python. Despite of the weakness of this programming language these implementations gave fast results for big sets, although the problem is NP-hard. By our knowledge these are the state of the art algorithms in correlation clustering. We found a near-optimal solution for a problem where the upper bound of the number of possible partitions is $10^{64,079}$ [16].

Our previous measurements show, that the accuracy of this method is among the best optimization methods [11]. These two properties enable the usage of this method in real-life applications.

References

[1] N. Bansal, A. Blum, and S. Chawla, "Correlation clustering," *Machine Learning*, vol. 56, no. 1-3, pp. 89–113, 2004. doi: 10.1023/B:MACH.0000033116.57574.95. [Online]. Available: http://dx.doi.org/10.1023/B:MACH.0000033116.57574.95

[2] C. Zahn, Jr, "Approximating symmetric relations by equivalence relations," *Journal of the Society for Industrial & Applied Mathematics*, vol. 12, no. 4, pp. 840–847, 1964. doi: 10.1137/0112071. [Online]. Available: http://dx.doi.org/10.1137/0112071

[3] S. Kim, S. Nowozin, P. Kohli, and C. D. Yoo, "Higher-order correlation clustering for image segmentation," in *Advances in Neural Information Processing Systems*, 2011. doi: 10.1.1.229.4144 pp. 1530–1538.

[4] A. Bhattacharya and R. K. De, "Divisive correlation clustering algorithm (dcca) for grouping of genes: detecting varying patterns in expression profiles," *bioinformatics*, vol. 24, no. 11, pp. 1359–1366, 2008. doi: 10.1093/bioinformatics/btn133. [Online]. Available: dx.doi.org/10.1093/bioinformatics/btn133

[5] B. Yang, W. K. Cheung, and J. Liu, "Community mining from signed social networks," *Knowledge and Data Engineering, IEEE Transactions on*, vol. 19, no. 10, pp. 1333–1348, 2007.

[6] T. DuBois, J. Golbeck, J. Kleint, and A. Srinivasan, "Improving recommendation accuracy by clustering social networks with trust," *Recommender Systems & the Social Web*, vol. 532, pp. 1–8, 2009. doi: 10.1145/2661829.2662085. [Online]. Available: http://dx.doi.org/10.1145/2661829.2662085

[7] Z. Chen, S. Yang, L. Li, and Z. Xie, "A clustering approximation mechanism based on data spatial correlation in wireless sensor networks," in *Wireless Telecommunications Symposium (WTS), 2010*. IEEE, 2010. doi: 10.1109/WTS.2010.5479626 pp. 1–7. [Online]. Available: http://dx.doi.org/10.1109/WTS.2010.5479626

[8] Z. Néda, R. Florian, M. Ravasz, A. Libál, and G. Györgyi, "Phase transition in an optimal clusterization model," *Physica A: Statistical Mechanics and its Applications*, vol. 362, no. 2, pp. 357–368, 2006. doi: 10.1016/j.physa.2005.08.008. [Online]. Available: http://dx.doi.org/10.1016/j.physa.2005.08.008

[9] L. Aszalós and T. Mihálydeák, "Rough clustering generated by correlation clustering," in *Rough Sets, Fuzzy Sets, Data Mining, and Granular Computing*. Springer Berlin Heidelberg, 2013, pp. 315–324. [Online]. Available: http://dx.doi.org/10.1109/TKDE.2007.1061

[10] ——, "Rough classification based on correlation clustering," in *Rough Sets and Knowledge Technology*. Springer, 2014, pp. 399–410. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11740-9_37

[11] L. Aszalós and M. Bakó, "Advanced search methods (in Hungarian)," http://morse.inf.unideb.hu/~aszalos/diak/fka, 2012.

[12] R. Durrett, R. Durrett, and R. Durrett, *Random graph dynamics*. Cambridge university press Cambridge, 2007, vol. 200, no. 7.

[13] L. Aszalós, J. Kormos, and D. Nagy, "Conjectures on phase transition at correlation clustering of random graphs," *Annales Univ. Sci. Budapest., Sect. Comp*, no. 42, pp. 37–54, 2014.

[14] R. Sibson, "Slink: an optimally efficient algorithm for the single-link cluster method," *The Computer Journal*, vol. 16, no. 1, pp. 30–34, 1973. doi: 10.1093/comjnl/16.1.30. [Online]. Available: http://dx.doi.org/10.1093/comjnl/16.1.30

[15] D. Nagy, "Correlation clustering of trees," Master's thesis, University of Debrecen, Faculty of Informatics, Hungary, 2015. Available: http://hdl.handle.net/2437/211878

[16] D. Berend and T. Tassa, "Improved bounds on bell numbers and on moments of sums of random variables," *Probability and Mathematical Statistics*, vol. 30, no. 2, pp. 185–205, 2010.