Proceedings of the Federated Conference on Computer Science and Information Systems pp. 403–406 DOI: 10.15439/2017F166 ISSN 2300-5963 ACSIS, Vol. 11

Correlation clustering: a parallel approach?

László ASZALÓS*, Mária BAKÓ†
* University of Debrecen
Faculty of Informatics
26 Kassai str., H4028 Debrecen, Hungary
Email: aszalos.laszlo@inf.unideb.hu
† University of Debrecen
Faculty of Economics
138 Böszörményi str., H4032 Debrecen, Hungary
Email: bakom@unideb.hu

***Abstract*—Correlation clustering is a NP-hard problem, and for large graphs finding even just a good approximation of the optimal solution is a hard task. In previous articles we have suggested a contraction method and its divide and conquer variant. In this article we present several improvements of this method (preprocessing, quasi-parallelism, etc.) and prepare it for parallelism. Based on speed tests we show where it helps the concurrent execution, and where it pulls us back.**

I. Introduction

CLUSTERING is an important tool of unsupervised learning. Its task is to group objects in such a way, that the objects in one group (cluster) are similar, and the objects from different groups are dissimilar. It generates an equivalence relation: the objects being in the same cluster. The similarity of objects are mostly determined by their distances, and the clustering methods are based on distance.

Correlation clustering is an exception, it uses a tolerance (reflexive and symmetric) relation. Moreover it assigns a cost to each partition (equivalence relation), i.e. number of pairs of similar objects that are in different clusters plus number of pairs of dissimilar objects that are in the same cluster. Our task is to find the partition with the minimal cost. Zahn proposed this problem in 1965, but using a very different approach [1]. The main question was the following: *which equivalence relation is the closest to a given tolerance (reflexive and symmetric) relation?* Many years later Bansal et al. published a paper, proving several of its properties, and gave a fast, but not quite optimal algorithm to solve this problem [2]. Bansal have shown, that this is an NP-hard problem.

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.

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 [3].

In a former article we have shown the clustering algorithm based on the divide&conquer method, which was more effective than our previous methods. But our measurements have pointed out, that this method is not scalable. Hence for large graphs the method will be very slow. Therefore we would like to speed up the method. The *simplest* way to do it is to distribute the calculations between the cores of the processor. Unfortunately, theory and practice often differs.

The structure of the paper is the following: in Section 2 correlation clustering is defined mathematically, Section 3 presents the contraction method and some variants. Next, the best combination of local improvements is selected, and in Section 5 the former divide and conquer method is improved. Later the technical details of the concurrency is discussed

II. Correlation clustering

In the paper the following notations are used: V denotes the set of the objects, and $T \subset V \times V$ the tolerance relation defined on V. A partition is handled as a function $p : V \to \{1, \dots, n\}$.

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 and partition iff value of $c_T^p(x, y) = 1$ in (1).

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

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

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

The task of correlation clustering is to determine the value of $\min_p c_T(p)$, and a 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 only get approximative, near optimal solutions.

Correlation clustering can be defined as a problem of statistical physics [4], where the authors use analogies from physics to solve the problem for small graphs. Here we do something similar. We can define the attraction between two objects: if they are similar then the attraction between them is 1; if they are dissimilar then the attraction between them is -1 (they repulse each other); otherwise—which can occur at a partial tolerance relation—the attraction is 0.

$$a(x,y) \leftarrow \begin{cases} 1, & \text{if } (x,y) \in T \\ -1, & \text{if } (x,y) \notin T \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

(3) can be generalized for object x and for clusters g and h:

$$a'(x,g) = \sum_{y\in g} a(x,y) \text{ and } \hat{a}(g,h) = \sum_{y\in h} a'(y,g).$$

We leave it to the reader to check that if these sums are positive and we join these element and clusters—by getting a partition p' containing the clusters $g \cup \{x\}$ or $g \cup h$—then $c_T(p) \geq c_T(p')$. This means that by joining attractive clusters, the cost decreases.

III. Contraction method

The contraction method [5] is based on two operation: the name *contraction* means that we join two attractive clusters. We can treat a cluster as *stable*, if for each of its elements the best position is inside this cluster, because the superposition of the forces (attraction or repulsion between an element and other elements is attraction for each element in the cluster; and there does not exist another cluster which is more attractive for any element in the cluster. But it is possible that the joining of two stable clusters produces a non-stable cluster: the new elements are mostly repulsive for a given element. In this case to get less conflicts this element needs to be *moved* into another cluster. Specially, if one object is repulsed by all clusters, a singleton containing this element needs to be constructed. The process includes calculation of the attractive forces of one object x for all clusters, and moving nodes based on the maximal attraction we called *movement*.

[5] contains the forces that are needed to recalculated after a movement or a contraction. These recalculations can be applied for any kind of tolerance relation. If the graph of the tolerance relation is dense—by using the matrices of forces between objects, forces between objects and clusters (for the movements) and forces between clusters (for the contractions)—then the contraction method can be run in an efficient way by adding and subtracting rows and columns of these matrices.

If the graph of the tolerance relation is sparse, then it is a waste to use full matrices for storing the actual forces. (If the tolerance relation contains small amounts of dissimilarity, then the optimal partition consist of only some clusters, so a small matrix is enough to store the forces between clusters.) In our former articles the algorithms were implemented in Python, and we used associative arrays (dict) and associative arrays of associative arrays to store the non-zero objects. But the deletion is problematic for this data type, therefore the implementations based on hash apply only logical deletion. Working

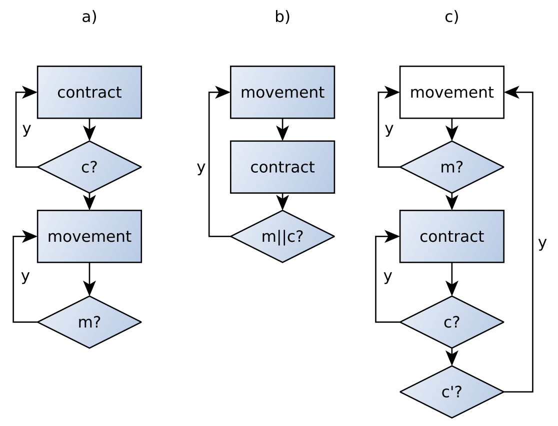

Fig. 1. Different combinations of steps of the local search.

with big tolerance relations, the limit of the implementation is noticeable.

Our new implementation approaches the problem from a new direction. By working with a sparse graph, most of its nodes (the objects) only have a few neighbours. If the forces on a specific node are needed to be calculated (for the movement), only its neighbours need to be checked, not all the objects. Therefore instead of searching for the neighbours of a given object again and again, we store them and the signs of their edges. Of course this means each edge is stored twice i.e. at both of its endpoints, but at a sparse graph ($|E| = O(|V|)$) this is not a serious problem.

To calculate the forces between clusters in the case of a dense graph all edges need to be visited, so the complexity is $O(n^2)$. But at sparse graphs the number of edges is proportional to the number of vertices, so the complexity of the calculation of forces is $O(n)$.

Correlation clustering can be treated as an optimization problem, where the aim is to minimize the number of conflicts. The steps of contraction and movement can be treated as a local search step. Nevertheless, simple variants of the hill climbing method are not effective for this problem. We have tested this in case of a graph with 13 nodes and from almost 3 million partitions only 2 were global optimum and around ten thousand were local minimum. For bigger graphs the ratio of number of global and local optima will be even smaller, so to find a global optima or a near optimal local optima is a truly difficult task.

The interesting question is how to combine the steps of contraction and movement. Fig. 1/a shows the method we implemented in the former article [6]. A contraction could be a dramatic change, even when two big clusters are joined. This means that with contraction many object get their final positions at the same time. In this variant this contraction step is repeated until it is successful (the number of conflicts decreases). Next, from the unstable clusters some objects are moved to better positions, and this movement is repeated until it is profitable.

This algorithm produced a rather fast method. By rewriting the source we compared this algorithm with some other variants. At first we tested what is the effect of changing the order of contraction and movement. Fig. 1/b shows a variant with a different order, where we execute a contraction after each movement. It is obvious, that the movements only produce local changes, so it takes many cycles to move the objects into their final cluster. Finally we created a variant which moves the objects until it is profitable, then joins the clusters until is it profitable, and if there was a contraction, then it starts a new turn, as Fig. 1/c shows. There is an unbreakable conflict between the speed and efficiency/accuracy: the number of conflicts at method c were 13 percent less that at method a.

IV. Quasi-parallel variant

Formerly we have discussed the quasi-parallel variant of the algorithms [5]. The most naive variant of the *contraction* step calculates all the forces between clusters, and next joins the two most attractive clusters and drops the other calculations. A bit cleverer variant reuses the calculated forces to calculate the forces according to the new (contracted) clusters.

The most costive variant wants to use all the calculations (without any recalculations). Hence it sorts all the calculated forces in decreasing order, and if that value is positive and valid, it joins the suitable clusters. (When can a calculated force be invalid? If some of the clusters it belongs to do not exist any more, because we have already merged them with another, a third cluster.) We named this last variant as *quasi-parallel*, because we practically join the clusters parallel, although not independently.

Of course we can implement similar variants of the *movement* step too. We did, and compared their speed and efficiency. Obviously, the latter variants are faster than the previous ones. The efficiency of the first two variants is the same: we calculated the same values, but the speedier variant needed more technical implementation. Which was surprising is that the last two variants differ in efficiency for contraction and movement.

In case of contraction, the sequential (the first two) variants were better than the quasi-parallel. Maybe the contraction step was so dramatic, that if we join two weakly attractive clusters before we realize that these clusters are more attracted by other clusters, then we cannot redo this action later. Here, based on our tests, the best strategy is to join only the most attractive clusters, and in the next round consider this new joined cluster as well.

In case of movement, the opposite holds. Here the costive (quasi-parallel) version is the best. We can interpret a movement of one object as an engagement. If we allow for a cluster that is getting increasingly stronger to keep on gathering new and new objects, this cluster becomes huge and will not release its objects. With quasi-parallel execution several small clusters grow parallel, and movements can be redone, if there is be a more attractive cluster for an object.

TABLE I
Remaining edges of the subgraphs (in percent)

N	no. of subgraphs			
	2	5	10	20
	BA(3,2)			
500	62.8/50.6	45.1/18.9	43.4/9.8	39.1/5.7
1000	63.2/48.9	50.3/21.6	42.8/9.8	40.4/5.2
5000	62.5/50.0	49.3/20.9	45.3/10.0	42.6/4.5
10000	62.6/50.3	49.2/20.0	44.8/10.6	42.8/5.0
20000	62.7/50.2	49.5/20.4	44.7/10.1	42.8/4.9
	ER (p=0.015)			
500	58.6/51.2	39.0/19.7	31.1/9.4	28.2/5.5
1000	53.4/49.9	28.1/20.6	20.8/10.2	16.2/4.8
5000	51.2/50.0	21.7/20.0	11.9/10.0	7.3/4.9
10000	50.5/49.9	20.9/19.9	11.0/10.0	6.2/5.0
20000	50.2/50.0	20.4/20.0	10.5/10.0	5.6/5.0

In the following measurements we will use the algorithm of Fig. 1/c with quasi-parallel movement and sequential contraction.

V. Divide and conquer reloaded

In a former article we have examined whether the divide and conquer approach is useful for correlation clustering [6]. As a reminder, the divide and conquer solution consist of three *simple* steps:

- divide the problem into subproblems,
- solve the subproblems (in a recursive way)
- construct the solution of the original problem from the solutions of the subproblems.

In some cases some of the steps could be left out or are very trivial. In our former article the construction of the subproblems was simple: we divided the graph into same size sub-graphs by the IDs of the objects. It can be checked easily, that with this construction most of the edges are left out from our calculations.

In case of Erdős-Rényi random graphs (ER) the edges are distributed uniformly at random. As the matrices of subgraphs cover only n/n^2 part of the matrix of original graph, only $1/n$ of the edges are left to work with. We construct slightly better sub-graphs with a little effort i.e. with complexity of $O(n)$. The breadth first traversal of the graph is used, and the nodes are taken in that order in which they are deleted from the fringe, i.e. when they get closed. The effectiveness of this trick is shown in Table I. It is not a surprise that in case of ER graphs, where the edges are independent from each other, this trick has no real effect. But at Barabási-Albert type random graphs (BA)—where the construction guaranties that the edges are not independent—the trick works well: when the former method left 5 percent of the edges, this leaves us 40 percent of them.

The other steps of the divide and conquer approach remained the same. The sub-problems were solved by recursion if they were big enough, otherwise a direct solution was used: starting from singleton clusters, the algorithm of Fig. 1/c for the graph of the sub-problem was followed. Finally all the clusters from the solutions of the sub-problems were collected and put together (as an initial clustering of the whole graph), and we executed the algorithm of Fig. 1/c again. It is surprising, but *solving the original problem, the sub-problems, the subsub-problems, etc. is faster than solving the original problem alone.* This is not a paradox, the key question is the initial clustering of the original problem.

VI. Technical details: concurrency

The last sentences of the previous chapter are very promising. Moreover at the reimplementation of our software we have taken care of parallelism.

We implemented our software in Python.[1] For calculation intensive tasks this language offers a `multiprocessing` package. At first we used the instruction `map` for each object, which could be familiar to the reader from Google's MapReduce concept. We recall, that *movement* in the algorithm of Fig. 1/c (at top right, emphasized by colour) is inside a double cycle. One task (to calculate the forces on an object) is extremely simple, hence the overhead is huge, it run thousands time slower than the original. Next we created a `pool`, and the set of nodes were divided into four, and each core of the processor received one subset, and the role to calculate the forces on nodes that are in that subset. At graphs with hundred nodes the parallel version was 300 times slower than the original. As the number of nodes of the graph increased the running time ratio became smaller and smaller, but at graphs with 20,000 nodes the parallel version was twice slower.

Our framework—constructed for divide and conquer method (D&C)—enables us to break the original problem into sub-problems, and solve them in parallel using the possibilities of a multi-core processor.

One categorisation of tolerance graphs is based on the rate of positive edges. As the edges of BA graphs are dependent, two graphs with the same rate could be very different, but using big samples can help us to discover tendencies. Based on the measurements, the preprocessing for D&C (the trick in the previous section) is useless when this rate is small, and very profitable if this rate is near to 1.

The biggest divergence in number of conflicts was at rate 0.71—where even the number of conflicts was maximal—so we executed speed tests for $3/2$ type BA graphs with this rate.

Based on the measurements, the running time the algorithm of Fig. 1/c is near quadratic—a problem with $20,000$ nodes was solved within a minute on an i5-6500 processor—and the aim is to solve problems with million of nodes in reasonable time.

We tested the D&C method which gave about 8 percent worse results than solving the problem at once. Does the running time compensate for this penalty? If we only have a few objects, the overhead of solving sub-problems gives a longer running time. At 3000-5000 objects this overhead disappears. But at problematic cases the hardness of solving the sub-problems brings this overhead back. We examined the running time of the subproblems, and we found, that for big graphs the combination of subsolutions (repeat the contraction method for the whole graph) could take up 98% of the running.

[1]The source files are available at https://github.com/aszalosl/DC-CC2.

VII. Future plans

Although we have a fast algorithm to solve the problems for large graphs, and some hints about how to choose between them, the research is not over. When solving big graph problems, most of the time only one thread is running, hence we have possibilities to use the concurrency. It is worth to try a manager and a pool of worker processes defined not inside cycles, but at the upper levels. The overhead of the communication between processes could be problematic, but only tests could decide on usefulness of this approach.

The fastest computation is *no* computation. Therefore we need to examine which calculations are necessary, and which can be omitted.

Of course these tricks do not change the quadratic complexity of the algorithm, but we believe, that we can reduce the the constant part, which will be very important in practice.

VIII. Conclusion

We introduced a correlation clustering problem, and we presented the contraction method to solve it. We improved our former algorithm in several ways, and we created several variants to it. Some of them used the elements of concurrent execution of the Python code with a small success.

To our knowledge, these are the state of the art algorithms in correlation clustering.

We made several measurements and the results gave hints on how to select amongst them to solve a particular problems. By these measurements our method has quadratic complexity. Finally, we presented the bottleneck of the algorithms. Our next step is to eliminate this, hopefully by using concurrency in a different way.

References

[1] 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

[2] 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

[3] L. Aszalós and M. Bakó, "Advanced search methods (in Hungarian)," http://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0103_13_fejlett_keresoalgoritmusok, 2012.

[4] 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

[5] L. Aszalós and T. Mihálydeák, "Correlation clustering by contraction, a more effective method," in *Recent Advances in Computational Optimization*. Springer, 2016, vol. 655, pp. 81–95. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-40132-4_6

[6] L. Aszalós and M. Bakó, "Correlation clustering: divide and conquer," in *Position Papers of the 2016 Federated Conference on Computer Science and Information Systems*, ser. Annals of Computer Science and Information Systems, M. Ganzha, L. Maciaszek, and M. Paprzycki, Eds., vol. 9. PTI, 2016. doi: 10.15439/2016F168 pp. 73–78. [Online]. Available: http://dx.doi.org/10.15439/2016F168