Order by Similarity

KWABENA W. AGYEMAN, Carnegie Mellon University
ALEX ZIRBEL, Carnegie Mellon University

Order by Similarity is a clustering and ordering problem similar to the Traveling Salesman Problem. The challenge is this: given a set of points and a way to determine the distances between a pair of points, create an ordered list of these points such that points closer together by distance are also closer together in the list. Whereas Traveling Salesman uses for its score only the distances of adjacent points, Order by Similarity cares about all distances between points, weighted by each pair's closeness in the list. We use two techniques to solve this problem: a complete branch-and-bound solver, and a k-means cluster solver.

1. INTRODUCTION

The Order by Similarity problem has applications wherever a set of items needs to be arranged in a sensibly-ordered list. It seems to be used most often in biology or chemistry; for example, the Bowling Green State University (BGSU) RNA lab uses an Order by Similarity algorithm to sort sets of RNA structures into a logical order. Though the problem bears some similarity to Traveling Salesman and to the topic of Hierarchical Clustering, very little literature exists on it.

One solution to this problem (and the inspiration for our work) is used by Bowling Green State University (BGSU) researchers to sort RNA structures, and is in use at http://rna.bgsu.edu/main/. We used the BGSU performance as our baseline, and also used their method of scoring the resulting list. As a part of our analysis, we also used their "matrix of discrepancies" approach, which they describe as follows:

> With n objects, one can compute an n by n matrix of discrepancies. It will be zero down the diagonal and symmetric. This is the fundamental data on which a cluster analysis is performed. Rather than try to identify clusters, the idea is to simply re-order the objects and look at the resulting distance matrix. Ordering the objects carefully will show the relations between them even more clearly than a cluster analysis.

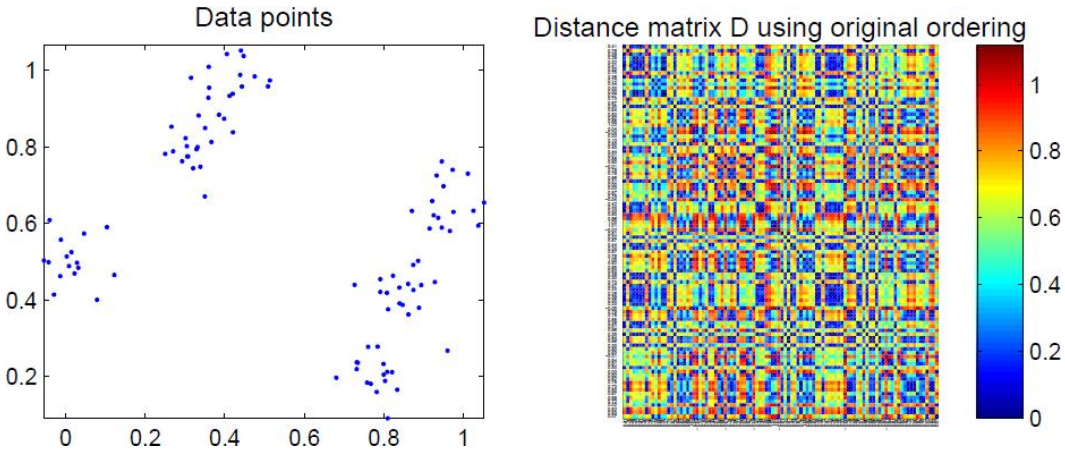

Figure 1 - Example Problem

The above diagrams show an example of 2D points and the distances between each pair in a matrix. Our goal is to order these points such that as much blue is near the diagonal is possible; this and a formal (more technical) scoring system is used in our evaluation.

To score the final set of points, we use the following scoring function:

$$score = \Sigma_{i,j}(d_{i,j} * \frac{1}{\sqrt{abs(i-j+1)}})$$

We found that this scoring function gave us a good tradeoff between closeness of a point's immediate and distant neighbors, and is used throughout the rest of our work.

2. BACKGROUND RESEARCH

Our background search confirms that prior work in this area is relatively scarce, but we discovered one other instance of direct prior work in the paper "Fast optimal leaf ordering for hierarchical clustering" [1]. Fast Optimal Leaf Ordering (FOLO) is a variant of the "dendogram correction" algorithm we proposed, which formulates the list to be sorted as a binary tree, then orders the branches in a hierarchical fashion so as to bring similarities in neighboring branches closer together. The algorithm is optimal under their scoring function and runs in $O(n^4)$ time, with $O(n^2)$ space.

During our research into FOLO, we discovered the field of Hierarchical Clustering, which attempts to sort points into clusters, then cluster those clusters, and so on. This problem is related to ours, but not identical, because it omits the problem of translating from a series of clusters to an one-dimensional list. FOLO solves this problem, but there may be other ways of solving it. Known solutions for Hierarchical Clustering run in $O(n^3)$ time.

Papers [2], [3], and [4] discuss ways to cluster points with a known distance metric, or create and tweak dendograms. These have helped us reconsider our original approaches. It seems that a lot of work has been done in creating and reordering dendograms ("dendogram seriation"). By contrast, we see almost no work in randomly generating and correcting orderings.

To fill the gaps in the prior work, our project takes two parts. We first show the results of a branch and bound solver, which works for small numbers of points and is a complete solver. We next show our cluster solver, which attempts to give a reasonable (though non-optimal) solution to a problem, and scales to huge numbers of points.

3. BRANCH AND BOUND SOLVER

We decided to implement a branch and bound solver for the Order by Similarity problem. We briefly looked at using a linear optimization solver but concluded that it would be extremely hard to formulate the Order by Similarity problem as a linear programming problem.

For our branch and bound solver we defined the following data structures:

Number of Points N

The number of points to order by similarity

Point List L[N]

A list of all the points to sort (this list is what gets ordered by similarity)

Distance Matrix D[N][N]

A matrix of distances between the positions of two points in the Point List

Weight Matrix W[N][N]

A matrix of weights for the matrix of distances

Point

Horizontal Position x	The x position of the point
Vertical Position y	The y position of the point

The distance between one point and another point is:

$\text{sqrt}((x1 - x2)^2 + (y1 - y2)^2)$

On startup we populate the distance matrix with the distances between all points. Note that this also means that the central diagonal of our matrix is zero because it represents the distances between each point and itself. For our weight matrix, we populate it on startup with our scoring function that is centered on the main diagonal in each row.

Our branch and bound solver's goal is to minimize the element by element multiplication of the weight and distance matrix. Through minimization we accomplish the goal of ordering the points in our problem by similarity.

The solver searches for the optimal solution to the problem using the following algorithm:

1. Choose the worst placed point in the list of points. This is simply the point whose row in the distance matrix multiplied by its row in the weight matrix is the greatest.

2. Move that point in the Point List (L) to a place in the L where that point would be the best placed. This is simply the position in the list where the distance between the point we are moving and its two neighbors (in a linear list) is minimized.

3. Now, we lock that point's position in the list and compute an upper and lower bound estimate for our problem which are:

Lower Bound:

The worst possible D .* W value of unlocked points which is sorted_rising(unlocked_D) .* sorted_falling(unlocked_W) plus D .* W for all locked points.

This is basically the lowest score you could have.

Upper Bound:

The best possible D .* W value of unlocked points which is sorted_rising(unlocked_D) .* sorted_rising(unlocked_W) plus D .* W for all locked points.

This is basically the highest score you could have.

If we find that the current search tree node is not valuable given the lower and upper bound then we terminate the search on that node and recurse upwards to try out the next most out of place value. Otherwise, we continue to our search with the point (above) locked in the list.

Once we have exhausted the entire search space (making our algorithm complete) we return the best possible result achieved so far while searching which is the optimal answer to the problem.

4. BRANCH AND BOUND SOLVER PREFORMANCE RESULTS

Below are two results of our solver shown in figure 2 and figure 3 for 12 points and 15 points respectively. We were not able to test more points than this because the runtime was too great. By the nature of the problem, the run time is O(n!) which has a search tree space of 479,001,600 nodes for 12 points and 1,307,674,368,000 nodes for 15 points – making both very, very, large. Our naive goal was to find the optimal solution to a problem with 100 points, but the size of the search tree renders this basically impossible.

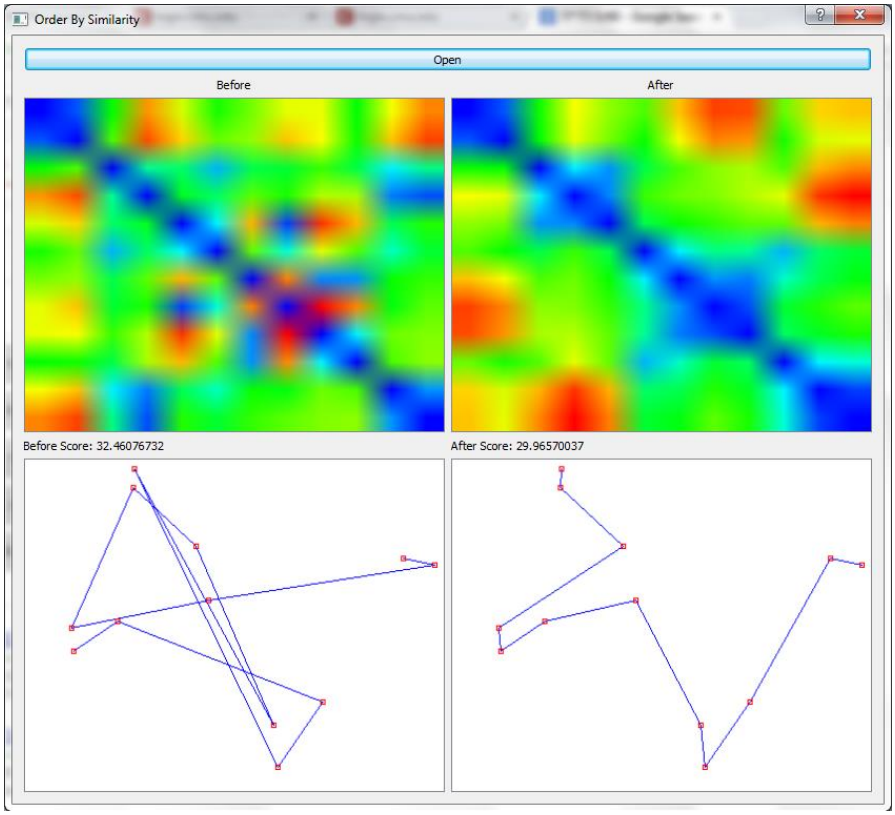

Figure 2 - 12 Points

In figure 2 above, is the output of our branch and bound solver for 12 nodes. On the left hand side is the original ordering and on the right hand side is the final ordering. The distance matrix is shown on the top and the ordering of points is shown on the bottom. The line drawn through points represents the order in which the points are placed in a linear array.

As you can see in the distance matrix, the optimal solution to the problem moves all the red colors (representing a larger distance) to the edge of the main diagonal while all blue colors (representing shorter distances) are moved to the center of the main diagonal.

In figure 3 below, we have the results for 15 points. Like the 12 point result before, the branch and bound solver moves cooler colors to the center and warmer colors to the edges.

An interesting observation of the search results suggest that a tighter lower bound could be obtained by clustering the points in 2D space first using a non-optimal or complete algorithm. For example, looking at figure 3 below, we can see that points closer to each other are ordered closer to each other. This would suggest that there are fewer orderings to really search for than (n!). By using this knowledge we could compute a much tighter lower bound to prune the search tree space much more quickly, making it possible to solve 100 point problems that have a structure which allows a the computation of a tight lower bound.

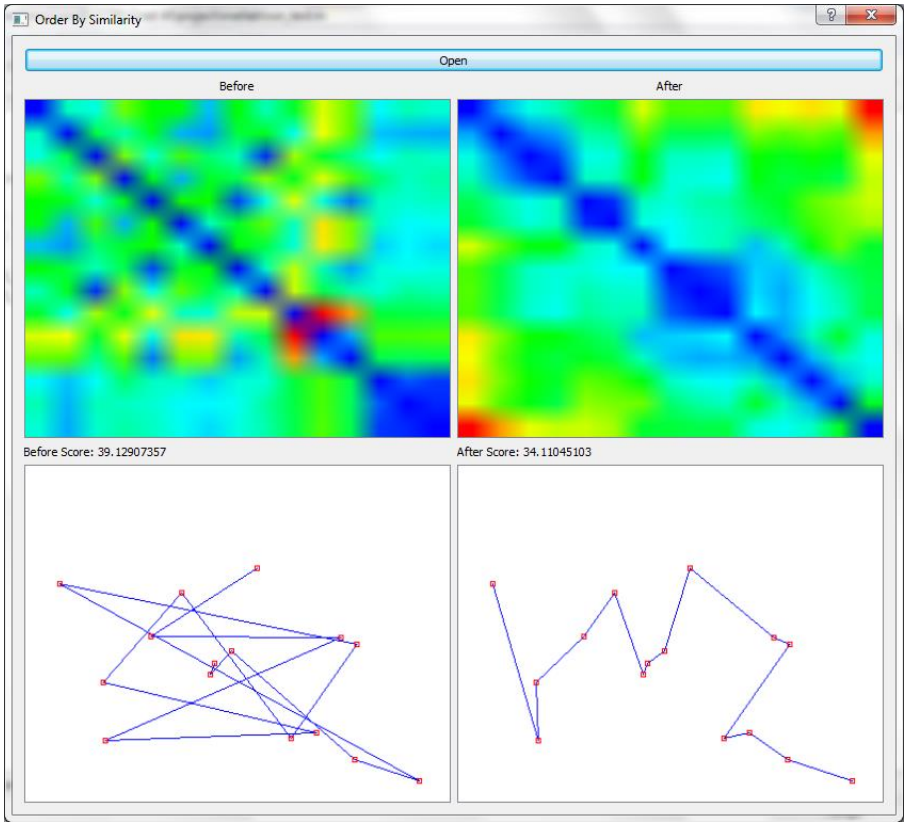

Figure 3 - 15 Points

To explore the previous observation about the effect of clustering in our search results we developed a recursive cluster solver capable of finding an improved ordering of points given a random (bad) ordering of points. The cluster solver is able to handle many more points than our branch and bound solver, making it a great candidate for lower bound generation. However, this is left to future work.

5. RECURSIVE CLUSTER SOLVER

The cluster solver is designed to order the points greedily to produce a reasonable, if not optimal, ordering. This fills a gap in existing work, where previous approaches (BGSU and SPIN) didn't run quickly on datasets of over ~1000 points.

The cluster algorithm works by splitting the points into two clusters using K-means, then drawing a bridge between them and running recursively on each of the resulting clusters. Our theory behind this approach is to find the maximum cut in the graph, and only cross it once.

To have this work reasonably well, two small improvements are needed. The first is to be sure to draw the bridge between the two closest points in each resulting cluster, and to exclude those points from the recursive clustering. For example, 10 points might be clustered into cluster A (4 points) and B (6 points), after which we would find the minimum bridge between them and run recursively on A without the bridge (3 points) and B without the bridge (5 points). The second improvement is to order clusters A and B correctly when joining them together: whether A-bridge-B or B-bridge-A is the better ordering depends on where the merged points need to connect to the rest of the points. We do this using a comparison of the cluster centroids of A and B to this connection point: if A's centroid is closer to the rest of the points, we merge with the ordering A-bridge-B. If B's centroid is closer, we use B-bridge-A.

6. RECURSIVE CLUSTER SOLVER PREFORMANCE RESULTS

We use the same method to evaluate the cluster solver as we used for the branch-and-bound solver above. One graph shows the distances between every pair of points, where the goal is to have blue or green (shorter distances) near the main diagonal and red (longer distances) toward the outside. The second graph shows a path through the points, which is a visualization of the linear ordering of the points.

Below, we show the output of the algorithm on 15 points, 200 points in favorable configurations, and 500 and 5000 points in random configurations.

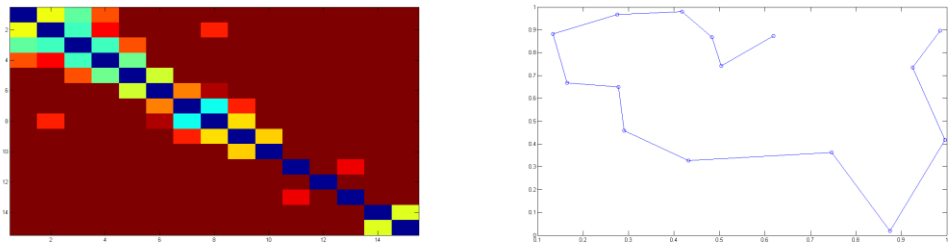

Figure 4 – 15 Points, randomly distributed

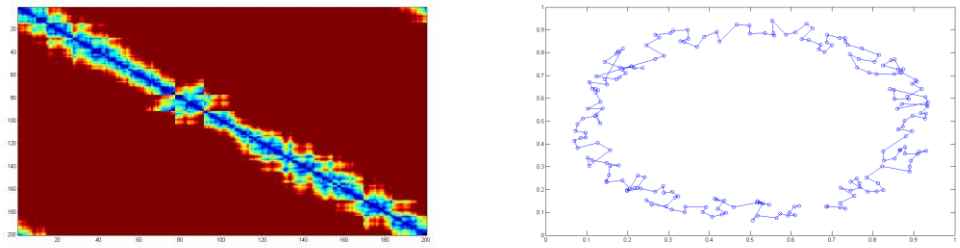

Figure 5 – 200 Points, distributed near a circle

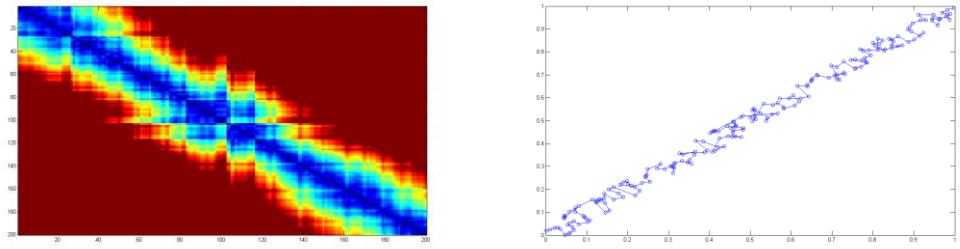

Figure 6 – 200 Points, distributed near a line

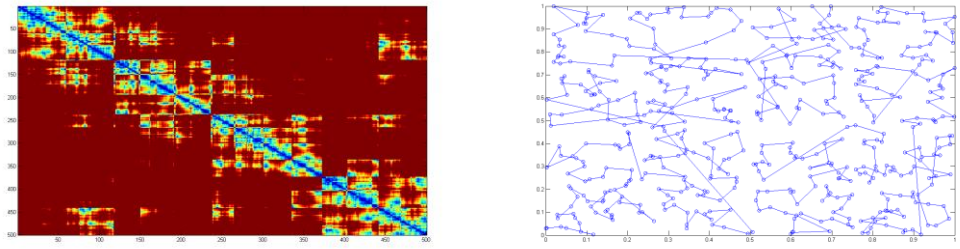

Figure 7 – 500 Points, randomly distributed

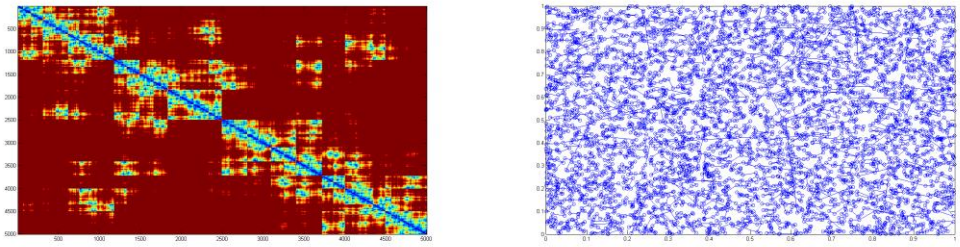

Figure 8 – 5000 Points, randomly distributed

As we can see, the cluster solver is able to handle these huge sets of points reasonably well, if not beautifully.

7. ALGORITHM COMPARISON

Another reasonable test is whether our branch and bound solver really improves performance at a small scale – at 10 or 12 points, is it even worth it to do a complete search? To answer this question, we ran both algorithms on a small dataset of the same points, and compared the outputs of each algorithm.

Due to constraints on the runtime of the branch and bound solver, we used a dataset of 20 randomly distributed sets of 10 points, and 5 randomly distributed sets of 12 points. Both algorithms found the same solution for 5 out of these 25 sets, and (as expected) the optimal branch and bound solver always scored equal to or better than the cluster solver. Below the average scores of each solver (using the scoring function from part 1):

Dataset Type	Original	Branch and Bound	Cluster
Random, 10 Pts	22.957	21.098	21.287
Random, 12 Pts	33.226	30.206	30.509

Table 1: Average output of both solvers on random datasets

This shows that the Branch and Bound solver is a fair bit better, but the scoring function does not always give a good intuitive feel for how well each algorithm worked. Below are shown two typical results of the algorithms, taken from the data above, to give more of an intuitive feel for their results.

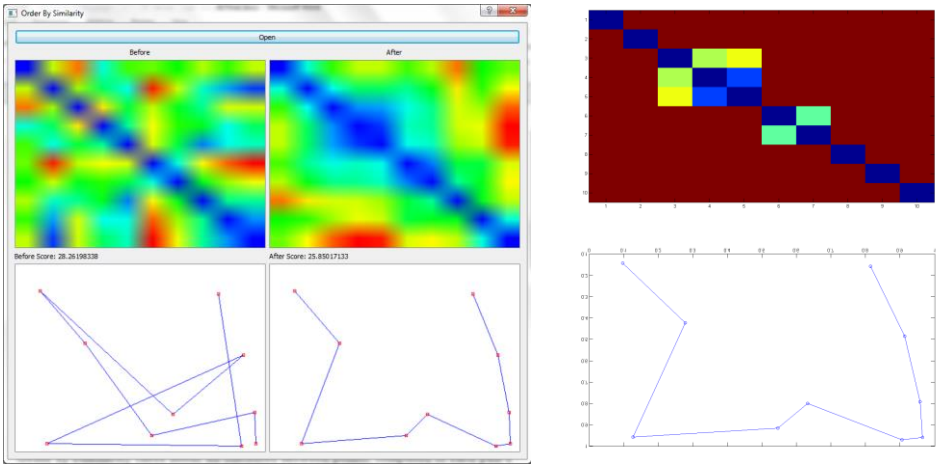

Figure 9 – Sample Comparison. Branch and Bound on the left.

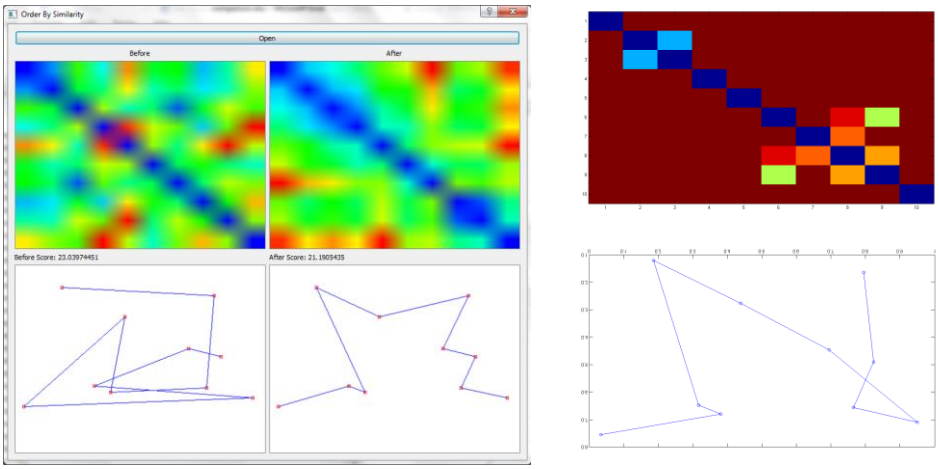

Figure 10 – Sample Comparison. Branch and Bound on the left.

These diagrams convey the basic tradeoff between the two approaches: sometimes the cluster solver will get lucky and find a good result, but most the time it will get a few things wrong on a small scale, and the resulting path won't do as well as the optimal solution.

8. CONCLUSION

Through working on the Order by Similarity problem, we explored the tradeoff between speed and optimality, and made some improvements to an optimal solution in order to scale it to larger datasets. Our resulting algorithms solve the problem well for very small numbers of points and for huge numbers of points. We believe that with more work, it could be possible to bridge these extremes and offer an optimal (or very close to optimal) solution for the more common case of 100-200 points.

Future work in this area would involve first integrating the two solvers, to investigate whether an approximation from the cluster solver might be useful as a lower bound for the branch and bound solver. The next step would be to integrate the resulting algorithm with BGSU's application, to see how well it fares in the real world.

REFERENCES

1. http://bioinformatics.oxfordjournals.org/content/17/suppl_1/S22.full.pdf
   Bar-Joseph, Ziv, David K. Gifford, and Tommi S. Jaakkola. "Fast optimal leaf ordering for hierarchical clustering." Bioinformatics 17.suppl 1 (2001): S22-S29.

2. http://compbio.fmph.uniba.sk/~tvinar/papers/01exprtr.pdf
   Biedl, Therese, et al. "Optimal arrangement of leaves in the tree representing hierarchical clustering of gene expression data." Dept. Computer Sci., Univ. Waterloo, Tech. Rep 14 (2001): 2001.

3. http://www.sciencedirect.com/science/article/pii/S0370157309002841
   Fortunato, Santo. "Community detection in graphs." Physics Reports 486.3 (2010): 75-174.

4. http://www.pnas.org/content/104/39/15224.full.pdf
   Sales-Pardo, M., et al. “Extracting the hierarchical organization of complex systems.” Proceedings of the National Academy of Sciences of the United States of America 104.10 (2007): 15224-15229

5. http://www.weizmann.ac.il/home/fedomany/spin_bioinfo_final.pdf
   Tsafrir, D., et al. "Sorting points into neighborhoods (SPIN): data analysis and visualization by ordering distance matrices." Bioinformatics 21.10 (2005): 2301-2308.