RWTH AACHEN
Diese Arbeit wurde vorgelegt am Lehrstuhl für Operations Research

Bachelorarbeit

Clustering algorithms to find special structures in matrices

vorgelegt von
Igor Pesic
Matrikelnummer: 329807

Erstgutachter:	Univ.-Prof. Dr. Bastian Leibe
Zweitgutachter:	Univ.-Prof. Dr. Marco Lübbecke
Abgabe:	**10. March 2016**

Eidesstattliche Versicherung

Name, Vorname

Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum

Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum

Unterschrift

Contents

List of Figures

1 Introduction

An integer program is a mathematical problem that is widely used in many fields such as economics, management and computer science. Because of its importance, scientists have continuously been developing and extending the algorithms and methods that can solve integer programs efficiently. The motivation comes from the fact that solving integer programs is an NP-hard problem. This means that, in theory, we need exponential time to solve an integer program. As a result of the previous research we can, in practice, solve many problems very quickly. However, solving integer programs is still a very complex and time consuming task in general.

One of the methods used to solve an integer or a mixed-integer program is the *Branch-price-and-cut* algorithm developed by Gamrath (2010). In order to run the algorithm, we need to divide the problem in a set of sub-problems. In order to divide a problem, we must find special structures in the constraint matrix of the integer program. Finding a good structure in the matrix can significantly ease the problem and therefore reduce the solving time. In this study, we try a novel method to discover good structure(s) in the matrix with the help of clustering techniques. Particularly, we can use graph clustering methods, as the constraint matrix can be implicitly mapped to a graph.

1.1 Motivation

As mentioned earlier, solving an integer program is a very complex task (see Schrijver (2003)). This means that, in practice, the solving time for an integer program can be very long. For that reason, many heuristics that try to reduce the solving time have been proposed. There are many different approaches on how to reduce the runtime (e.g. the pre-solving steps), and they are often combined together to reduce the runtime as much as possible. Finding a good decomposition (i.e. division of a problem into more sub-problems) of an integer problem is another approach that tries to minimise the solving time. In this study we propose a method to find a good special structure in the matrix with the help of clustering techniques.

Even though there are already a few methods to automatically find special structures in the matrices, according to our best knowledge, there are yet no methods to find special structures with the help of clustering algorithms. With the proposed method we have obtained satisfactory results, which were often as good as or even better than those obtained with methods implemented before. However we still cannot give a proof that the solving time for all integer problems will be significantly reduced or reduced at all. For example, our method did not work on bin packing and equivalent problem instances. Nevertheless our method showed some remarkable improvements that may be very useful in the practical use.

1.2 Overview

In this Section we will take a look at the steps of the method described in the study. As our input we take a linear program (LP). From the LP, we extract the constraints and pack them into a sparse matrix. After that we can divide our algorithm into three main parts. The first part of the algorithm, which is discussed in Section 3.1, has the task to convert the constraint matrix to a graph, i.e. to define a similarity (or a distance respectively) measure between any two rows in the constraint matrix. This enables us to use the clustering algorithms in a meaningful way. The second part of the algorithm, described in Section 3.2, is focused on the clustering of the matrix. There we try a couple of different clustering algorithms that run on the constraint matrix with the similarities (or distances) defined in the first part of the study. The third part of the algorithm has the task to process the clusters and bring them in a suitable form. This means that the clustered constraints obtained at the end of this process must be in an acceptable form required by the Dantzig-Wolfe decomposition. This step is described in more detail in the Section 3.3.

1.3 Related work

There has been very little work done on the topic of the automated special structure detection. It is considered that finding a special structure has to be done based on the specific problem knowledge i.e. the experts who state a problem can exploit their knowledge of the problem to define most suitable structure. Therefore, automatic structure detection is omitted from the most state of the art integer program solvers. However, there is a method described by Bergner et al. (2015) that shows how the automatic

structure detection can be implemented and scored. Based on these scores, we can, at least up to a certain probability, predict how good the discovered structure is.

The methods used by Bergner et al. (2015) to find special structures are based on the (hyper-)graph partitioning where no clustering algorithms were mentioned. Also, the special structure detectors implemented in the Generic Column Generation[1] (GCG) that was developed by Gamrath (2010) and Gamrath & Lübbecke (2010) rely on the graph partitioning. The graph partitioning procedures are mostly based on finding min-cuts of the graphs or hyper-graphs. Also some methods implemented in the GCG do not use graph partitioning, but only the breath-first-search methods.

As, according to our knowledge, no previous works have applied clustering techniques to the problem, we take the linking of this two concepts main contribution of this thesis.

1.4 Outline

- **Chapter 2** describes the main terms, concepts and algorithms used in this study. Here we will briefly explain what is an integer program, and how it is solved. In addition, we will give a short insight into clustering in general, and the main ideas and concepts used in clustering algorithms in general.
- **Chapter 3** describes in detail the work done as a part of this study. This includes the pre-processing step, the clustering step and the post-processing. We will take a deeper look into each of the clustering algorithms used in the study, and give the motivation for choosing each one of them.
- **Chapter 4** provides an evaluation of each of the methods (i.e. clustering techniques and pre-processing methods) used in the study. Here we will also present the improvements achieved in this study.
- **Chapter 5** summarizes the main idea of the study and proposes the possible future work.

[1] www.or.rwth-aachen.de/gcg/

2 Basics

The thesis combines at least two different fields of study: data mining and mathematical optimization. From data mining we use the clustering techniques and apply them to structure the mathematical optimization problems such as integer programs. In order to better understand the problem and the solutions discussed in Chapter 3 we shall first introduce some basic definitions, algorithms and concepts used in the thesis. We will start by explaining what an integer program is. After that we will briefly look at the algorithm used to solve an integer program so that we can understand why we need to find special structures at the first place. Lastly, we will explain what clustering is and describe different approaches used by the clustering algorithms.

2.1 Integer program

A **linear program (LP)** is a mathematical optimization and feasibility problem. It consists of two parts. The first part is an objective function. This is the function that we want to maximize or minimize, which essentially does not make any difference. The second part of the linear program are the constraint functions. The objective function is a subject to the constraint functions. Our goal is to find the optimal solution for our objective function under the constraints that are given. Sometimes we can find the optimal solution to our problem, sometimes we can find more than one optimal solution, but it can also happen that no solution exists. In that case we have shown that the problem is infeasible. Solving a linear program can be done in polynomial time by the, for example, ellipsoid method described by Khachiyan (1979) and the interior point methods described by Karmarkar (1984) and Renegar (1988). However, the most widely used method for solving the LPs is the simplex method developed by Dantzig et al. (1955) which is very efficient in practice.

An **integer program (IP)** is a special case of a linear program but with one significant additional constraint: all variables are integer numbers. An integer program in

its canonical form looks like this:

$$\begin{aligned} \text{maximize} \quad & c^T x \\ \text{subject to} \quad & Ax \leqslant b \\ & x \in \mathbb{Z}_+^n \end{aligned}$$

where A is the constraint matrix, b the vector of the right-hand-side values and c the vector of coefficients in the objective function. The complexity of solving IP increases due to its integral constraint and because of that it belongs to the NP-hard complexity class as proven in Papadimitriou (1981).

The integer programs have a wide field of application. We can use them to formulate a travelling salesman problem, a bin packing problem, a set cover problem, a p-median, a generalized assignment problem and many many others. Integer programs are used in many areas. We can use them in the production planning, logistics, scheduling, telecommunication networks, computer science etc...

One type of an integer program is the **Mixed Integer Program (MIP)**. This program is very similar to the IP but here we allow some of the variables to have a fractional component. Because of integral constraints that are still present in the MIP, MIP has the same complexity as the IP and it is solved by the same algorithm: **Branch-price-and-cut algorithm**. We will briefly explain this algorithm in the following section.

2.1.1 Example: Generalized Assignment Problem

One typical example of an integer program is the *Generalized Assignment Problem (GAP)*. The objective of the GAP is to find the maximum profit assignment of **n jobs** to **m agents** such that each job is assigned to exactly one agent. Besides that, each agent has a restricted capacity. Formally, we can state the problem as follows:

$$\begin{aligned} \max \quad & \textstyle\sum_{j=1}^{m} \sum_{i=1}^{n} p_{ij} x_{ij} \\ \text{s.t.} \quad & \textstyle\sum_{j=1}^{m} x_{ij} = 1 && \forall i = 1..n \\ & \textstyle\sum_{i=1}^{n} w_{ij} x_{ij} \leq c_j && \forall j = 1..m \\ & x_{ij} \in \{0,1\} && i = 1..n, j = 1..m \end{aligned}$$

where:

- p_{ij} is the profit of assigning job i to agent j,

- w_{ij} is the claim of the capacity of agent j by job i,
- c_j is the capacity of agent j and
- x_{ij} equals 1 if job i is assigned to agent j, and 0 otherwise.

The first constraint guarantees that each job is assigned to exactly one agent. The second constraint ensures that each agent can get no more jobs than his capacity allows. The last constraint makes sure that we get an integral solution.

2.2 Dantzig-Wolfe Decomposition

In this Section, we will briefly look at the Dantzig-Wolfe decomposition method (as described by Desrosiers & Luebbecke (2010), Gamrath (2010) and Desrosiers & Luebbecke (2005)) and the input it needs. Furthermore, we will examine why we have to discover special structure in the constraint matrix as a motivation for the further research done in this thesis. Also, we will examine what types of special structures are possible for Dantzig-Wolfe decomposition.

Dantzig-Wolfe decomposition is a method a for solving the MIPs. It exploits the special structure in the constraint matrix. The structure that our algorithm will output, and that can be well exploited by Dantzig-Wolfe decomposition is defined as follows:

$$\min \sum_{k\in[K]} c_k^T x^k \tag{2.1}$$

$$\text{s.t.} \sum_{k\in[K]} A^k x^k \geq b \tag{2.2}$$

$$D^k x^k \geq d^k \qquad \forall k \in [K] \tag{2.3}$$

$$x^k \geq 0 \qquad \forall k \in [K] \tag{2.4}$$

$$x_i^k \in \mathbb{Z} \qquad \forall k \in [K], i \in [n_k^*] \tag{2.5}$$

This is a reformulation of MIP which is called *bordered block diagonal structure*. The structure is illustrated in Figure 2.1. Sometimes it is possible to reformulate the MIP as shown above, but without the constraint 2.2. In that case, we call it *pure block diagonal structure*. As we will discuss later, the pure block diagonal structure is favourable in comparison to the bordered one. Beside these two structures, the third, *staircase structure* is also allowed, but it will not be part of this study, and thus we will not discuss it

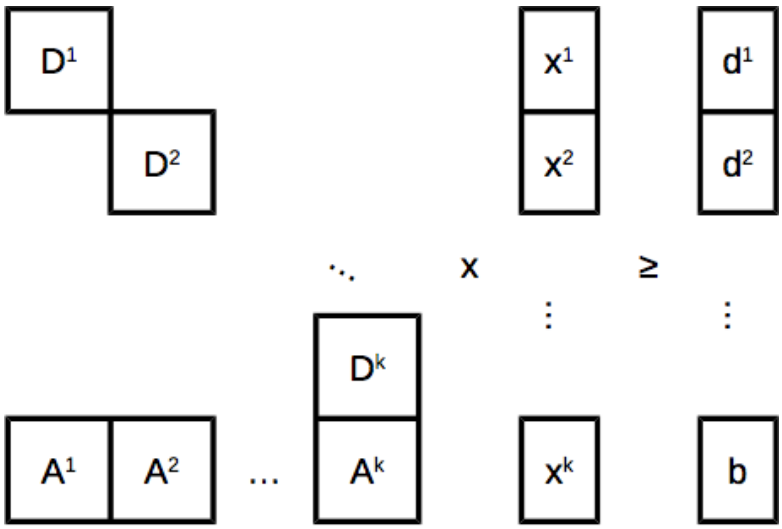


Figure 2.1: Bordered block diagonal structure

any further.

In this model, with $k \in [K]$ blocks where $K \in \mathbb{Z}_{>0}$, we have two types of constraints: linkage constraints 2.2 and structural constraints 2.3. The linkage constrains are represented by $A^k \in \mathbb{Q}^{m_A \times n_k}$ matrices and the right-hand-side b. These constraints represent *master constraints*. Furthermore, the structural constraints are represented by $D^k \in \mathbb{Q}^{m_k \times n_k}$ and d_k on the right-hand-side. These constraints are "separated" in K blocks and constraints in block k enforce restrictions for vector x^k. This structure is also allowed to have only one block, but typically it has more than one blocks (i.e. $k > 1$).

In order to obtain diagonal structure of the matrix, we must rearrange its columns and rows. The rearrangement will not change anything in the linear program, because the order of columns (i.e. variables) and rows (i.e. constraints) does not matter and cannot affect the solution. We arrange the rows so that constraints belonging to one block k are listed one after the other (i.e. they should not mix with constraints of the other blocks). At the bottom of the matrix we list all the remaining (i.e. master) constraints. We must also rearrange the columns of the matrix in the same manner. Now, the only "problem" is how to decide which rows and columns belong to which block. This is done with the help of clustering algorithms and more detail on that is given in Chapter 3.

Furthermore, we can define a set X_k for each block K as follows:

$$X_k := \{x^k \in \mathbb{Z}_+^{n_k^*} \times \mathbb{Q}_+^{n_k - n_k^*} | D^k x^k \geq d^k\} \tag{2.6}$$

This set represents a set of feasible solutions for block K, i.e. it satisfies all the constrains that are a part of D^k. It also satisfies corresponding integrality and non-negativity constraints. This means that in this definition we have implicitly included constraints 2.3, 2.4 and 2.5. Therefore, we can write the MIP in *compact form*:

$$\min \sum_{k \in [K]} c_k^T x^k \tag{2.7}$$

$$\text{s.t.} \sum_{k \in [K]} A^k x^k \geq b \tag{2.8}$$

$$x^k \in X_k \qquad \forall k \in [K] \tag{2.9}$$

After applying the convexification (i.e. representing each vector $x_k \in X_k$ by a convex combination of extreme points plus a conical combination of extreme rays, see Dantzig & Wolfe (1960)) or the discretisation (see Vanderbeck (2000)) of X_n, which we will not explain in this study (see Desrosiers & Luebbecke (2010), Gamrath (2010)), we can obtain the following equivalent *master problem* (MP):

$$\begin{aligned} \min \quad & \textstyle\sum_{p \in P} c_p \lambda_p + \sum_{r \in R} c_r \lambda_r \\ \text{s.t.} \quad & \textstyle\sum_{p \in P} a_p \lambda_p + \sum_{r \in R} a_r \lambda_r \leq b \\ & \textstyle\sum_{p \in P} \lambda_p = 1 \\ & \lambda \geq 0 \end{aligned}$$

This master problem is equivalent to the original problem and the optimal solution value to the master problem is also a lower bound for the optimal solution value to the original problem. Optimal solutions to the master problem can be transformed to the optimal solution (possibly fractional) candidates of the original problem (i.e. we can find the candidates for the optimal values of x).

The disadvantage of this reformulation is that the number of variables increases exponentially. Now for each polyhedron k (one polyhedron corresponds to one block) we have one new variable λ_p^k for each extreme point $p \in P_k$ and one variable λ_r^k for each extreme ray $r \in R_k$. However, this huge number of variables that suddenly appears can

be managed by the *column generation* approach (see Barnhart et al. (1998)). Besides that, the main reason for reformulating the original MIP in the master problem is the fact that the master problem yields much lower integrality gap than a simple relaxation of the MIP. Integrality gap is the difference between the optimal solution of the original MIP and the optimal solution of the same problem but without any integrality constraints. Lower integrality gap has a huge effect on the complexity of the problem, i.e. it significantly reduces the solving time.

In conclusion, due to its effect on reducing the integrality gap, Dantzig-Wolfe decomposition has proven to be very a efficient way of solving MIPs. For that reason it has become a standard part of the MIP solvers. This made us think, how we could best generate the input for the solver i.e. how we could automatically find good special structures in the matrices. With that motivation, the idea has raised, to try to discover the blocks in the matrix with the help of the clustering algorithms, where each cluster will represent one block, and where non-clustered points will represent the maser constraints.

2.3 Clustering

In this section we will describe what clustering is and what connects the clustering and the reformulation needed for the Dantzig-Wolfe decomposition.

Clustering is a technique for grouping similar data points. Data points which are similar to each other should belong to the same cluster, while the points that do not have much in common should belong to different clusters. Some clustering algorithms also allow some data points to be out of the clusters. These points are labelled as *non-clustered* points. These are the "noisy" points, which means that they do not fulfil necessary criteria to be assigned to any of the clusters.

The similarity between each two points in the feature space is usually defined as a negative squared Euclidean distance. On the other hand, most of the algorithms allow similarity measure to be defined as desired (for example Hamming distance or Cosine distance) if it is necessary. By defining a specific similarity for each two data points, we implicitly build a (complete) graph where the vertices are the original data points and the edges contain the similarity value of the two adjacent vertices. If the similarity between some two points is equal to zero, we can omit the corresponding edge. Defining

a specific similarity allows us also to use *graph clustering* algorithms on our data.

There are several clustering methods as described by Berkhin (2002), for example "density-based clustering" (e.g. DBSCAN), "graph clustering" (e.g. Markov Chain Clustering), "hierarchical clustering", partitioning relocation clustering (e.g. K-means), "distribution-based clustering" (e.g. EM), etc.

In this study, clustering algorithms had to fulfil one of the following two prerequisites:

- the algorithm is able to deal with the user-defined similarity (or distance) measure
- the algorithm can cluster binary data in a meaningful way

As the second prerequisite might seem a bit odd, the reason for that is the following: the input for a clustering algorithm is the binary version of the constraint matrix. Namely, we wanted to find a special structure in the matrix, i.e. we wanted to divide constraints into blocks. Two constraints belong to the same block if they have a certain amount of common variables, but two constraints that belong to different blocks are **not** allowed to have **any** common variables. For that reason, we were **only** interested in the information what variables are included in what constraints. Thus we were able to convert the original constraint matrix A to its binary version A' as follows:

$$a'_{ij} = \begin{cases} 1 & \text{for } a_{ij} \neq 0 \\ 0 & \text{for } a_{ij} = 0 \end{cases} \tag{2.10}$$

If implemented efficiently, this reformulation can save a lot of memory: instead of storing a floating-point number which is 32 or 64 bits long, we can now save each coefficient as a corresponding binary number which has the length of 1 bit.

In the context of discovering special structures in the constraint matrix of a LP, clustering has the task to partition the constraints in such a manner that constraints that share more variables have a higher probability to appear in the same cluster, while the constraints that share less or none variables are less likely to be part of the same cluster. With that in mind, we have examined different similarity (or distance) measures that were tailored for this task.

After the clustering had been done, we wanted to evaluate the obtained result. There are

many measures that can be applied to evaluate the quality of the clustering. Basically the evaluation metrics can be divided in two parts: *internal* and *external* as described in the book by Manning et al. (2008). The internal metrics express the goal of obtaining high intra-cluster similarity and low inter-cluster similarity. The external metrics are measured based on the data that was not clustered and the metrics may vary heavily depending on the problem. For example, an external metric can be based on the true labels or some external benchmarks. The way we are going to evaluate the results is tailored for our application and it will be discussed in the Section 3.3.

In conclusion, the clustering has been applied to many different problems, and very often, the input for the algorithms and their evaluation has to be tailored to the purpose of their use. Here we have used clustering techniques for a completely new objective and thus we had to make some adjustments in order to get good results.

3 Own work

In this Chapter we will describe the core work that has been done in the study. The biggest part of the study was choosing, implementing and testing different clustering algorithms and similarity measures in order to find the special structures in the matrices. Also we have done some post-processing. These three steps are divided as follows:

Defining the similarity measure In this part of the work we have defined different similarity (or distance) measures for each pair of rows in the constraint matrix of the given MIP. More detail on this is given in the Section 3.1.

Clustering the data Here we have run different clustering algorithms on the given similarity measures. The algorithms used in the study and more detail on each one of them is given in the Section 3.2.

Post-processing Last part of the work was divided into two sub-parts. Firstly, we adapted the clustering labels to make a suitable matrix decomposition from the given clustering. Secondly, as all of our clustering algorithms ran many times (i.e. we ran each of them with different parameters), we had to decide, which clustering we want to pick as the most suitable one. A brief explanation of these steps is given in Section 3.3.

Beside the three main points mentioned above, the work consisted of two more parts. The first part was parsing the input LP file from which we obtained the constraint matrix. The input file types supported in this study were *.lp* and *.mps*. Furthermore, in order to test the obtained decompositions, the structured matrix was saved in decomposition file (*.dec*) which specified for each row in the matrix if it was part of a block or it was a master constraint, and if it belonged to a block, it specified to which one. After saving the output, the GCG solver ran on each obtained decomposition. As these operations are not relevant part of the research, but rather trivial steps needed in the implementation, they will not be discussed any further.

The whole source code needed for the thesis was written in Python[1]. Two main reasons

[1] www.python.org

for this choice are the fact that Python is an open-source language and that it supports the numerical operations, linear algebra operations and clustering algorithms very well. This means that most of the basic functionalities and clustering algorithms were already implemented in some of the numerous open-source libraries for this language (e.g. scikit-learn[2], numpy[3], scipy[4]). Beside this fact, Python is also very easy and fast to code, which made the work much faster and gave more time and space to focus on the research itself. On the other side, Python has one disadvantage. Namely, it is a script language, and thus some basic concepts (e.g. loops) are very slow in comparison to the other languages. For this reason Python is suitable for prototyping, but should not be used in the final version.

3.1 Defining similarity measures

Running the clustering algorithms in order to find special structures in the constraint matrices is a very specific problem. Here we consider the rows of the matrix to be the data points and the columns the features. Herewith we can notice the matrices have very different feature spaces. As our goal is to find diagonal blocks in the matrix, where entries outside the blocks must be zero, we have discovered that distinguishing only between zero and non-zero entries is enough and that only that way we can produce some decent results. So, we have defined a more abstract and more simple feature space. In such a feature space we had to define the similarity (or distance) measures on our own as the usual Euclidean distance would not be applicable any more.

Some clustering algorithms deal with similarity measures while the others need distance measure between data points. In order to be able to run all algorithms, we need a way to convert similarity measure into comparable distance measure and vice versa easily. For that purpose, we define all similarity values to be in the range [0,1], where 1 is the highest and 0 is the lowest similarity between any two rows in the matrix. With that in mind, we can now use the following equations to convert between distance and similarity:

$$d(x, y) = 1 - s(x, y),\ s(x, y) : [0, 1]^M \times [0, 1]^M \mapsto [0, 1] \tag{3.1}$$

[2] www.scikit-learn.org
[3] www.numpy.org
[4] www.scipy.org

where $d(x, y)$ and $s(x, y)$ are distance and similarity functions respectively between rows x and y of length M.

If we want to define the distance measure, it must be a *true* distance measure, i.e. it must fulfil the following requirements according to Giancarlo et al. (2010):

1. Non-negativity: $d(x, y) \geq 0, \forall x, y$
2. Identity: $d(x, y) = 0$ if and only if $x == y$
3. Symmetry: $d(x, y) = d(y, x)$

In the following sections, we will show and discuss the similarities (i.e. distances) used in the study. There are much more known measures that could be used for this problem as stated by Choi et al. (2010). However, we have implemented and evaluated only a handful of them because of the time limit and the complexity that would arise from big number of similarity measures.

We had two basic criteria when choosing these similarity measures. Firstly, similarity between two rows had to be different than zero if and only if two rows had at least one common variable. This significantly reduces the running time of the algorithms and also enables us to discover the *pure diagonal structure* (see Section 2.2) with some algorithms. Second, the increase in the similarity had to correlate with the increase in the number of common variables.

In order to better describe some similarity/distance measures, we will first define some expressions as in paper by Choi et al. (2010):

$$a = \sum_{n=0}^{M-1} i_n * j_n$$
$$b = \sum_{n=0}^{M-1} \neg i_n * j_n$$
$$c = \sum_{n=0}^{M-1} i_n * \neg j_n$$
$$d = \sum_{n=0}^{M-1} \neg i_n * \neg j_n$$

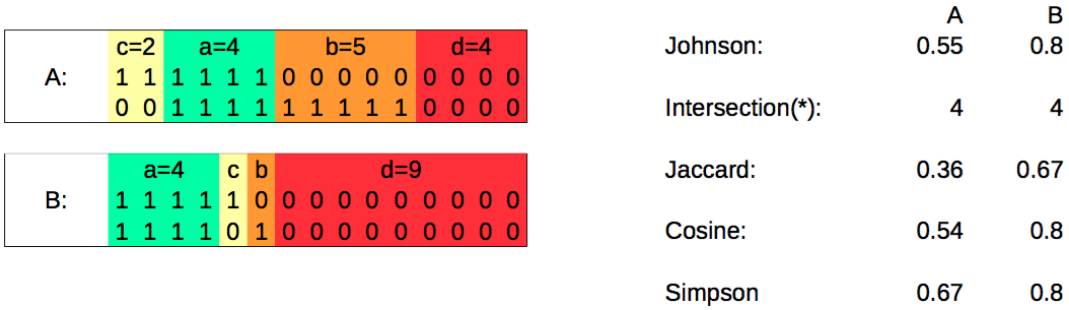


Figure 3.1: Example for the similarity values calculated for two different pairs of matrix rows. (*): the values obtained by intersection similarity have to be normalized later based on the biggest similarity.

where i and j are binary rows of length M. We have obtained binary values by converting each non-zero value to 1.

We will also define the distance or similarity matrix $A \in \mathbb{R}^{NxN}$ with $a_{i,j} = d(i,j)$, or $a_{i,j} = s(i,j)$ respectively.

3.1.1 Johnson similarity

Johnson similarity measure is defined as:

$$s_{Johnson}(i,j) = \frac{a}{a+b} + \frac{a}{a+c} \tag{3.2}$$

In order to fulfil our requirement that all similarity measures are in range [0,1], we have to divide $s_{Johnson}(i,j)$ by 2. Here is similarity the fraction of the common variables in each of the two rows. This way we give smaller similarity in the cases where one row can have decent number of common variables, but still has a huge number of the other variables. In that case we might prefer this row to be part of the master constraints.

3.1.2 Intersection similarity

Intersection similarity measure is defined as:

$$s_{Intersection}(i,j) = a \tag{3.3}$$

This is very simple similarity measure, but seemed to bring very good results. This measure is a number of common non-zero entries. In order to scale it properly, we divide each entry in the similarity matrix by the maximum similarity found and then

multiply it with $1 - \varepsilon$ in order to make it less than 1. After that we manually add $s(i, j) = 1$, *for* $i = j$. We have chosen this similarity because it was the most intuitive way to connect the rows with common variables.

3.1.3 Jaccard similarity

Jaccard similarity measure is defined as:

$$s_{Jaccard}(i, j) = \frac{a}{a + b + c} \tag{3.4}$$

Jaccard similarity is also known as the ratio between the intersection and the union of two sets. Here we calculate the fraction of common variables in the total number of variables present in the both rows. Similarly as in Johnson similarity, we try to reduce the similarity for the rows with big number of non-common variables. This similarity penalises heavily the case where number of non-common variables in any of the two rows is high relative to the number of common variables.

3.1.4 Cosine similarity

Cosine similarity is defined as:

$$s_{Cosine}(i, j) = \frac{a}{\sqrt{a + b} * \sqrt{a + c}} \tag{3.5}$$

This represents a $\cos\Theta$, with Θ being an angle between binary vectors i and j. This should represent the natural distance of two binary vectors. If their distance is smaller, it means that they have more common variables and less of the other variables.

3.1.5 Simpson similarity

Simpson similarity is defined as:

$$s_{Simpson}(i, j) = \frac{a}{\min(a + b, a + c)} \tag{3.6}$$

This similarity measure correlates with the number of common variables, but also penalizes the case where both rows have lots of uncommon variables. In contrast to Johnson, Jaccard and the Cosine similarities, this one gives higher similarity in the case where one row has almost all variables as the other, no matter how many uncommon variables the other row has.

3.1.6 Further similarity measures

At the end we will describe couple more similarity measures that did not work well in order to motivate our basic criteria for choosing the similarities which described at the beginning of this section.

- $s_3(i,j) = d$, with this similarity measure we were not able to distinguish clearly among the clusters because very weak correlation existed with the number of common variables. For example DBSCAN was not able to detect any cluster or only 1 cluster that contained all the rows in the matrix. This is due to the fact that the constraint matrices are sparse matrices with lots of columns. This similarity measure had typically very high value for any pair of rows and the obtained values could not been distinguished well.

- $s_4(i,j) = d + a * (a + \frac{b+c}{2}) * \frac{1}{N}$, this similarity measure is by default not in the range [0,1] and thus has to be scaled in the same way as we scaled $s_{Intersection}$. With this similarity measure we have achieved similar results as with s_3 and it wont be subject of further discussion.

- $s_7(i,j) = a + d * (a + \frac{b+c}{2}) * \frac{1}{N}$, this similarity measure contradicts our criteria that only pairs of rows with common variables should have similarity > 0. This has significantly affected the running times of MCL and R-MCL algorithms and additionally did not bring any results better than the other similarities.

3.2 Clustering

In this section we will describe the algorithms used in this study, theory behind them and their advantages and disadvantages. The algorithms are listed below:

- DBSCAN
- Minimum spanning tree clustering
- Markov Chain Clustering
- Regularized Markov Chain Clustering
- Expectation Maximisation
- Agglomerative clustering

Motivation for choosing each of these algorithms defer and will be explained more detailed in the following subsections. Even though all of these algorithms were used and compared at some point of the study, only the first four algorithms from the list had sustainably good performance. For that reason, the final evaluation will focus only on these algorithms.

3.2.1 DBSCAN

DBSCAN algorithm was the first clustering algorithm used in this study. Main reasons for that are that it supports the pre-computed distance measure and it allows certain points to be defined as "noise".

The DBSCAN algorithm belongs to the group of "density-based" clustering methods and it was developed and described by Martin Ester & Xu (1996). Main advantages of the algorithm are:

- it is very efficient and scalable
- we need to set only one input parameter (we can easily find the appropriate value for it and the second parameter may be fixed)
- it discovers clusters of arbitrary shapes
- it supports any user-defined distance function

DBSCAN considers clusters as the areas of high density that are separated by low density areas. It divides the points in three different types as showed in figure 3.2:

- **Core points:** these are the points that have at least minimal required number of points (*MinPts*) within the radius *Epsilon* including the point itself.
- **Border points:** these are the points that have at least one core point within the Epsilon radius.
- **Outliers**: these points meet the requirements for neither core nor border points.

The core points are those located in high density regions, border points are on the edges of the high-density regions, while the outliers are the points in the low density areas.

[5]Adapted from Wikipedia

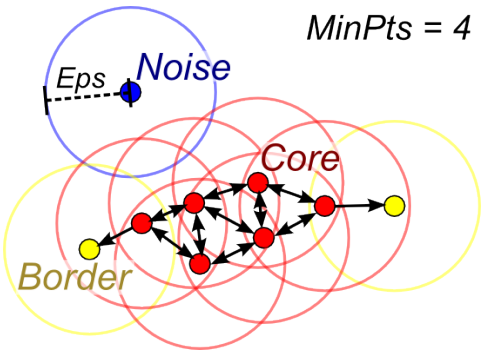


Figure 3.2: Core, border and noisy points [5]

We can build the clusters recursively by choosing an arbitrary core point, adding it to the cluster, finding all of its neighbours that are core points, adding them to the cluster and repeat the process for the chosen neighbours until no core points are neighbours of the cluster members. At the end we can also add the border points as our cluster members. The running time of this algorithm is almost linear and it depends on nearest neighbour search. By definition, every cluster must have at least *MinPts* number of points and the core points are deterministically assigned to the clusters. On the other hand, this clustering method is not deterministic because the border points might belong to different clusters.

This kind of approach enables us to detect clusters of arbitrary shapes and sizes and it also allows us to cluster the points without predefining the number of clusters. The only two parameters we can adjust are the Epsilon and minPts. In the study done by Martin Ester & Xu (1996) it was empirically shown that for $k > 4$ no significant improvement in clustering exists, but only the computation time increases. Because of that we have fixed minPts to 4 in this study. On the other side we had to determinate Epsilon parameter for each data set (in our case constraint matrix). In the paper Martin Ester & Xu (1996) there was a solution proposed that could help determinate Epsilon, but it was not implemented in this study because of certain limitations of Python (e.g. time-expensive function calls). For that reason, we ran DBSCAN with different Epsilon values that were obtained from symmetric geometric array on length 49 and the range $[mid - 0.5, mid + 0.5]$, where mid was calculated as the percentile of non-zero values in the distance matrix with $q = 10$. Of course, the values in the array that were smaller or equal to 0 and bigger or equal to 1 were omitted. Also, if our clustering consisted of

only one cluster that contained all the points, we did not run DBSCAN with any higher Epsilon values.

Overall, DBSCAN is very simple and efficient, but also very powerful algorithm. As shown later, it has brought some very promising results.

3.2.2 Minimum Spanning Tree Clustering

The Minimum Spanning Tree (MST) Clustering is a graph clustering procedure which is based on minimum spanning trees of the graphs. It was developed by Zahn (1971). Also it was shown by Xu et al. (2002), Morris et al. (1986) to have very good results in various applications. Besides that, there is no need to adjust many parameters (e.g. number of clusters) and this algorithm can also do a good job in finding the clusters of various sizes and "shapes". Also, because this is a graph clustering technique, it fits well to our problem as our similarity (or distance) matrix represents a graph implicitly. These were the main reasons for choosing this algorithm as part of our study. The source code used here was written by Vanderplas (2015).

Even though this algorithm has a completely different approach than DBSCAN (i.e. DBSCAN uses a density-based approach and MST is a graph clustering technique), it has brought very similar and often the same results as DBSCAN. However, this algorithm has a small advantage over DBSCAN because it can detect a *pure diagonal structure* in the constraint matrix.

As said before the algorithm is based on finding the minimum spanning tree of the graph. Input is the distance matrix, were every distance is "length" of the corresponding edge. In addition we enter two parameters: *cutoff* and *min cluster size*. *Cutoff* denotes the threshold for the length of the edges, while *min cluster size* is responsible for labelling too small clusters as noise. After choosing these two parameters, we find a minimum spanning tree. A MST is an acyclic sub-graph such that the sum of the length of its edges is minimal. We can find a MST e.g. with the algorithm proposed by Kruskal (1956), which has the time complexity of $O(n \log n)$, where n is the number of edges. After finding the MST we remove all edges of the MST which have length bigger than cutoff parameter. At the end we obtain a graph which consists of multiple components, where each component represents a cluster. If a component has less vertices than min cluster size, we label its vertices as "non-clustered".

Because of similar meaning of the input parameters as in DBSCAN, we choose them in the same way as we did for DBSCAN.

In conclusion, the algorithm has very good scalability because its running time is dominated by finding an MST. Also, its well-behaving has contributed to choosing it as a part of this study and eventually as part of the final evaluation.

3.2.3 Markov Chain Clustering

Markov Chain Clustering (MCL) is an example of a graph-clustering method and it is based on Markov chains. It was developed as a part of the Ph.D. thesis by van Dongen (2000). The algorithm was also used in research done by A. J. Enright & Ouzounis (2002).

The algorithm is based on the assumption that there are more and stronger links between the members of the same cluster and fewer and weaker links among the vertices of different clusters. This is an assumption of the random walk which states that if we start at some node we are more likely to stay within the cluster than to change the cluster. Based on these random walks, we can see where the flow tends to congregate. MCL calculates the random walks using Markov chains.

MCL takes a similarity matrix (one described in Section 3.1) as an input. Besides that, MCL takes two input parameters: *expand factor* and *inflate factor*. The first step of the algorithm is to normalize the input matrix column-wise so we can recognize it as a Markov chain transition matrix. The second step is called expansion. This is basically moving to the next generation of Markov chain (i.e. increasing the length of the random walk). This is simply multiplication of the matrix by itself expand-factor-times. This step is responsible for pushing the flow to different regions of the graph. The third step is called inflation. In this step we try to minimize the effect that, on the long run, the flow tends to be equal across the graph. For that purpose we intentionally increase the edge weights that are within the cluster and decrease the weights of the edges that connect different clusters. This is simply done by raising each column in the matrix to a non-negative power (inflate factor), and then re-normalising each column. The forth step of the algorithm is pruning. We do this step only for the purpose of reducing the computation time. In this step we convert very small values to 0 as we assume that

they would reach zero anyway. After this step we repeat the whole procedure until the convergence or until the maximum number of iterations is reached. In this study the maximum number of iterations was set to 20. Even though there is no proof given in the van Dongen (2000), it was experimentally shown that the algorithm converges, namely in 10 to 100 iterations.

The idea behind this approach is very intuitive and also correct as shown in many studies (e.g. A. J. Enright & Ouzounis (2002), Satuluri & Parthasarathy (2009), "Engineering Graph Clustering: Models and Experimental Evaluation" (2008)). This approach lets the cluster structure of the graph show itself and if no structure exists, it will return one cluster. We also do not have to specify the number of clusters, which is very good, because this information is not available to us. In addition this algorithm is suitable for our study, as the similarity measures implicitly represent a graph.

The bad side of the algorithm is its complexity. The most expensive step is the expansion and it has the complexity of $O(n^3)$, where n is the number of rows in the constraint matrix. Inflation step is done in $O(n_{nz})$ time, where n_{nz} is the number of non zero values in the matrix. The good thing is that our input matrix is sparse and also after very few steps the number of non-zero values in the matrix decreases significantly. If implemented efficiently, (e.g. with CSR matrix from the scipy[6] Python module) multiplication time gets reduced significantly, so that only first couple of iterations are time consuming. In order to reduce the computation time even further, we can set very small values in the input matrix directly to zero as we assume they would become zero eventually.

As mentioned above, we need to set two input parameters: expand factor and inflate factor. As A. J. Enright & Ouzounis (2002) suggest we have fixed the expand factor to 2, which is equal moving one step in a random walk per 1 iteration of the algorithm. Furthermore we have chosen our inflate factor to have the values [1.1, 2) with the step of 0.05. The reason for that is the following: choosing the inflate factor to be 1 would mean that we annihilate the inflation step. Also choosing inflate factor to be too high produces lots of smaller clusters, which is not desirable (explained in the Section 3.3). In the Figure 3.3 you can clearly see that the number of clusters climbs rapidly.

In conclusion, the MCL clustering is very elegant and intuitive method for clustering

[6] www.scipy.org

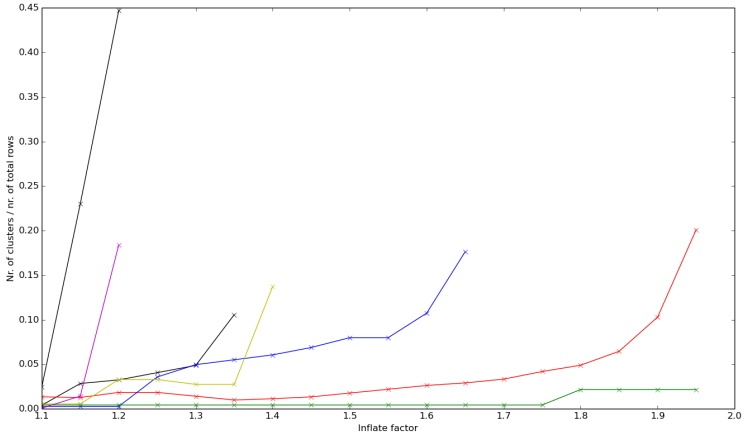


Figure 3.3: Number of clusters increases with the increase of the inflate factor. Stop criterion is set to 0.4 of nr. of clusters/nr. of total rows ratio.

the graphs and according to A. J. Enright & Ouzounis (2002), Satuluri & Parthasarathy (2009), "Engineering Graph Clustering: Models and Experimental Evaluation" (2008) it returns good results. On the other side, the theoretical complexity of the algorithm is $O(n^3)$ which is extremely expensive and MCL **sometimes** tends to produce the singleton clusters as stated in Satuluri & Parthasarathy (2009) and some very big clusters. Overall the idea behind the algorithm seems to be promising and that was the main reason for choosing this algorithm as part of this study.

3.2.4 Regularized Markov Chain Clustering

Regularized Markov Chain Clustering (R-MCL) is a variant of the MCL algorithm with one significant difference. In the expansion step of MCL we have multiplied the **current flow** graph with itself expand-factor-times. In the R-MCL, instead of multiplying the current flow graph with itself, we multiply the the current graph with the initial input graph.

R-MCL is a response to the problem of over-fitting of the MCL. As stated in Satuluri & Parthasarathy (2009), MCL usually tends to produce lots of small clusters, rather then few bigger ones. Reason for that is that MCL often allows the edges between neighbouring nodes to raise significantly and therewith reduce the weights of the other surrounding edges. This is due to the fact that the input graph structure is taken in consideration only in the first iteration, while in the later runs of the algorithm, the

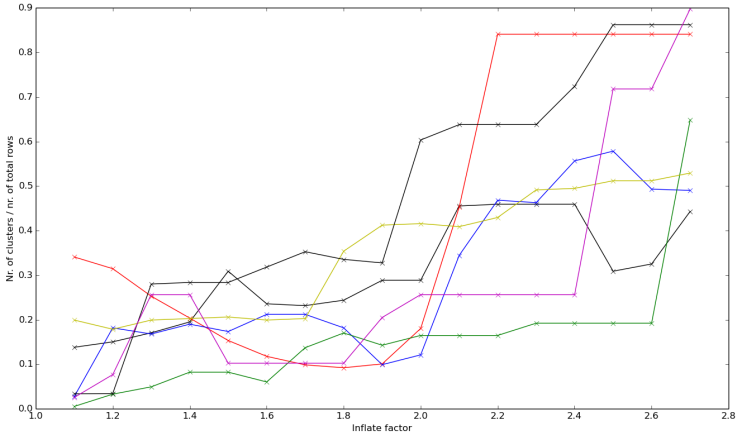


Figure 3.4: Number of clusters increases with the increase of inflate factor.

initial graph structure is being neglected.

As for the MCL, we must also choose the suitable value of the inflate factor. In this study, we have used the values of the inflate factor in the range $[1.1, 2)$, with the step of 0.1. The value of 1 or below would mean, same as in MCL, that we annihilate the inflation step and for values above 2, as shown in Figure 3.4, the number of clusters increases significantly.

3.2.5 Expectation Maximisation

Expectation maximisation (EM) is an algorithm that softly assigns data points to each of the mixture components. It was developed by Hartley (1958) and later described by Dempster et al. (1977). If we consider the points in a cluster as the points generated from a single distribution, we can then assume that all data points can be described by a mixture model. In this study we have assumed that every component is defined with Bernoulli distribution because our data is of binary type, but other probability distributions are also supported by the algorithm. Bernoulli distribution is defined as

$$p(\mathbf{x}|\mu) = \prod_{i=1}^{D} \mu_i^{x_i}(1-\mu_i)^{(1-x_i)} \tag{3.7}$$

with parameter μ which represents a mean of the distribution, i.e. a "centroid" for each cluster. One advantage of this method is that we do not have to calculate any similarity measure before running the algorithm, but only to initialize the means of the clusters.

Sadly, the initialization of the centroids makes the algorithm non-deterministic, and the results may drastically vary in each run.

The benefit of the algorithm is that we only have to determinate one parameter, namely the number of clusters. We will call this parameter K, and we will label each cluster with $k \in [K]$. In addition we define $\mathbf{x}_n$ to be the n-th row the input constraint matrix. After setting parameter K, algorithm runs as follows:

1. Initialize the clustering with K randomly chosen centroids (cluster means).
2. **E-step:** softly assign samples to mixture components:

$$\gamma(z_{n,k}) = \frac{\pi_k p(\mathbf{x}_n|\mu_k)}{\sum_{j=1}^{K} \pi_j p(\mathbf{x}_n|\mu_j)} \tag{3.8}$$

   This is the responsibility of component k for point x_n. Here $z_{n,k}$ represents a latent variable corresponding to data point n and cluster k and $\gamma(z_{n,k})$ is its expected value.

3. **M-step:** re-estimate the parameters for all the components:

$$N_k = \sum_{n=1}^{N} \gamma(z_{n,k}), \text{ is a soft number of points labeld with k} \tag{3.9}$$

$$\mu_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{n,k}) \mathbf{x}_n, \text{ is centroid of component k} \tag{3.10}$$

$$\pi_k = \frac{N_k}{N}, \text{ is a weight for component k} \tag{3.11}$$

4. Go to step 2 until convergence

We have tried out this algorithm $\forall K \in \{2, 3, ..., 10\}$ but the results that we obtained after the post-processing showed that points were clearly clustered in no more than 3 or 4 clusters, while the other algorithms were able to detect more and better clusters for the same instances. Because of that and the huge time complexity of the algorithm, $O(N * K * D)$ (with N the number of constraints, K number of clusters and D the number of variables), we have decided to discard it from the further study.

In conclusion, even thought this algorithm is good in many application fields, like image processing, it did not show very well for our problem.

3.2.6 Hierarchical/Agglomerative clustering

Hierarchical/Agglomerative clustering approach is very well known and common technique for clustering. It was used in many different applications and studies (Cheng et al. (2006), Dasgupta & Long (2005), King (1967)) where it has shown to have very strong results what made us also include it in our study. The algorithm was described in many studies, among which the earliest central study was done by Sneath & Sokal (1962).

This algorithm basically consists of two parts. First we declare each point as a separate cluster. In further iterations of this step we merge the clusters with the smallest distance. Distance between two clusters u and v can be defined in many ways, but we have mostly tested following two distance measures:

- Single method: $d(u, v) = \min(d(u[i], v[j])), \forall i \in u, \forall j \in v$
- Complete method: $d(u, v) = \max(d(u[i], v[j])), \forall i \in u, \forall j \in v$

We iterate as long as we reach the state where all points belong to the same cluster. We also save the clusterings obtained in each iteration in a *linkage table*.

The second step is hierarchical clustering. Here we start with one cluster containing all the points and divide it according to the linkage table saved in previous step. We divide the clusters as long as we reach the terminating criteria. We have used k, the number of clusters, as the terminating criteria. We have obtained k as the highest value of the second derivative of cluster distances obtained from the linkage table. This way we have chosen the k for which we had the biggest change in the inter-cluster distances.

Sadly, this algorithm did not bring good enough results to be tested further. Clusters obtained from the algorithm were often very unequally sized e.g. many clusters with 1 point and few or 1 very big cluster. Because of that we will not include this algorithm in the final evaluation.

3.3 Post-processing

In this section we will talk about the steps that we made just after we ran the clustering algorithms mentioned earlier. We can divide the procedure in two parts:

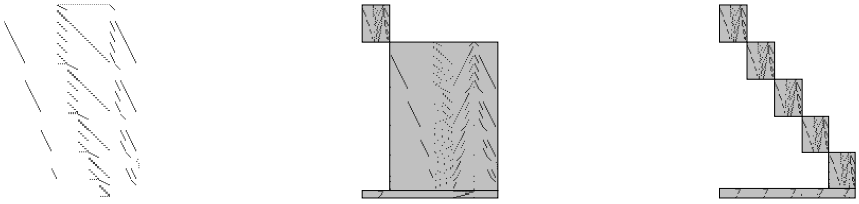
- Adopt the clustering to the acceptable form for Dantzig-Wolfe decomposition.
- Decide which iteration(s) of clustering was the best.

In the first part, we sort matrix rows according to the cluster labels in the ascending order and then we find those rows in a cluster, which had the common variables in the constraint(s) that belonged to some other cluster. When we find such a row, we remove it from the cluster i.e. we label it as non-clustered row. After repeating the procedure for all clusters we have satisfied the requirement that one variable cannot belong to the constraints of two different blocks. At the end we sort the columns so that all variables belonging to a cluster are grouped together, and these groups are then also sorted in the ascending order. This way we obtain the matrix which has the form shown in Figure 2.1.

During this study, we have discovered that it is better that clustering algorithm returns less bigger clusters than more smaller clusters. This is because after complying to the above mentioned requirement, we often have to remove several rows from each cluster, so at the end we finish up with removing lots of rows from the clusters. This means that clustering was not very successful. According to that, we can now evaluate the output of the clustering algorithms in the way that we define some result good, if the number of non-clustered points did not increase significantly after removing non-compliant rows. If the number of non-clustered rows increased dramatically we can freely assume that clustering was not successful enough.

The second part of post-processing, which builds on top of the first part, is meant to choose the best iteration of clustering algorithm. As mentioned earlier, we run each algorithm multiple times with different parameters. Because of that we get multiple clusterings for one algorithm and one problem instance. It would be almost impossible, but certainly unnecessary to compare all these clusterings at the end. As a result we had to decide for only few of them which we want to include in the later comparison of the algorithms. We have choose to pick up to two clusterings for each algorithm. The selection criteria are defined as follows:

- Choose the clustering with fewest non-clustered points.



(a) Original matrix (b) Smallest lower border (c) Score $s = 0$

Figure 3.5: Original matrix and two decompositions chosen according the two criteria.

- Choose the clustering with lowest score s. We defined s as:

$$s = \frac{std}{mean} * n_{-1}^2 \tag{3.12}$$

where *std* and *mean* are standard deviation and mean of sizes of the clusters and n_{-1} is the number of non-clustered points.

Setting these two criteria is based on the research done by Bergner et al. (2015). They suggest a couple of factors that can influence the quality of a decompositions. The strongest correlation observed in the study was the one between the size of border (i.e. number of non-clustered points in our case) and solving time. Namely, it was found that solving time for an MIP decreases with the decrease of the size of the border. Furthermore, it was discovered that the solving time is high in cases with extremely high or extremely low border size.

Having that in mind, we wanted to pick those clusterings with least number of non-clustered points. This is precisely what the first criterion is about. The second criterion also takes this into consideration, but it also rewards more equally distributed points in clusters. This way we want to penalize cases where, for example, we have two very big clusters (with size of 90% of all points) and many clusters with only one or two points. In the Figure 3.5 we have illustrated the case where we have chosen decomposition with smallest lower border and decomposition where all clusters have the same size, which means that our score $s = 0$.

Beside these two criteria, we have one more superior criterion: if we get zero non-clustered points, and more than one cluster, we consider such clustering as the best possible clustering and we immediately break the loop and forward the result to the GCG. This is the case when we have discovered the pure block diagonal structure. This

decision is based on the suggestions given by Bergner et al. (2015) and empirical analysis which is described more detailed in Chapter 4.

4 Evaluation

In this Chapter we will discuss the results obtained by the clustering algorithms. We will first describe the measurement used in evaluation and then we will discuss how did the algorithms perform in comparison to those implemented in the GCG. We will also discuss the special cases and some unexpected results that appeared during the testing.

4.1 Measurements for evaluation

The main goal of the thesis was to reduce the solving time for MIPs by finding the appropriate decompositions for the constraint matrices. Therefore, the primary measurement for the evaluation was the solving time. GCG is an MIP solver which already contains couple of methods for the automatic decomposition detection which were described in the Section 1.3. We have used the GCG as the benchmark for the evaluation. After generating couple of different decompositions, GCG chooses the one, that is considered to be the best. In this study, we also compare *all* of its decompositions with the chosen one and with *all* of our decompositions. In addition we use the GCG to solve MIPs so the resulting runtime for each instance was obtained from that solver. We limit the computation time for the solver to one hour. We have run the solver on the machine with the Intel i7 CPU and 16GB RAM. If the solver cannot find a solution within one hour, it terminates.

4.2 Test data

For evaluating the performance of the clustering algorithms, we have chosen problem instances of several types. Most of the instances used in the testing come from the MIPLIB2003 by Achterberg et al. (2006) and MIPLIB2010 by by Koch et al. (2011) data sets. These data sets contain various instances of mixed integer programs. Beside these, we also use instances from the following data sets: airland (MIPs), various bin packing problem data instances, cpmp (capacitated p-median problem) and setcover. Instances from each of these four sets yield very similar results, so in the Appendix we include at most one representative from each of the four sets.

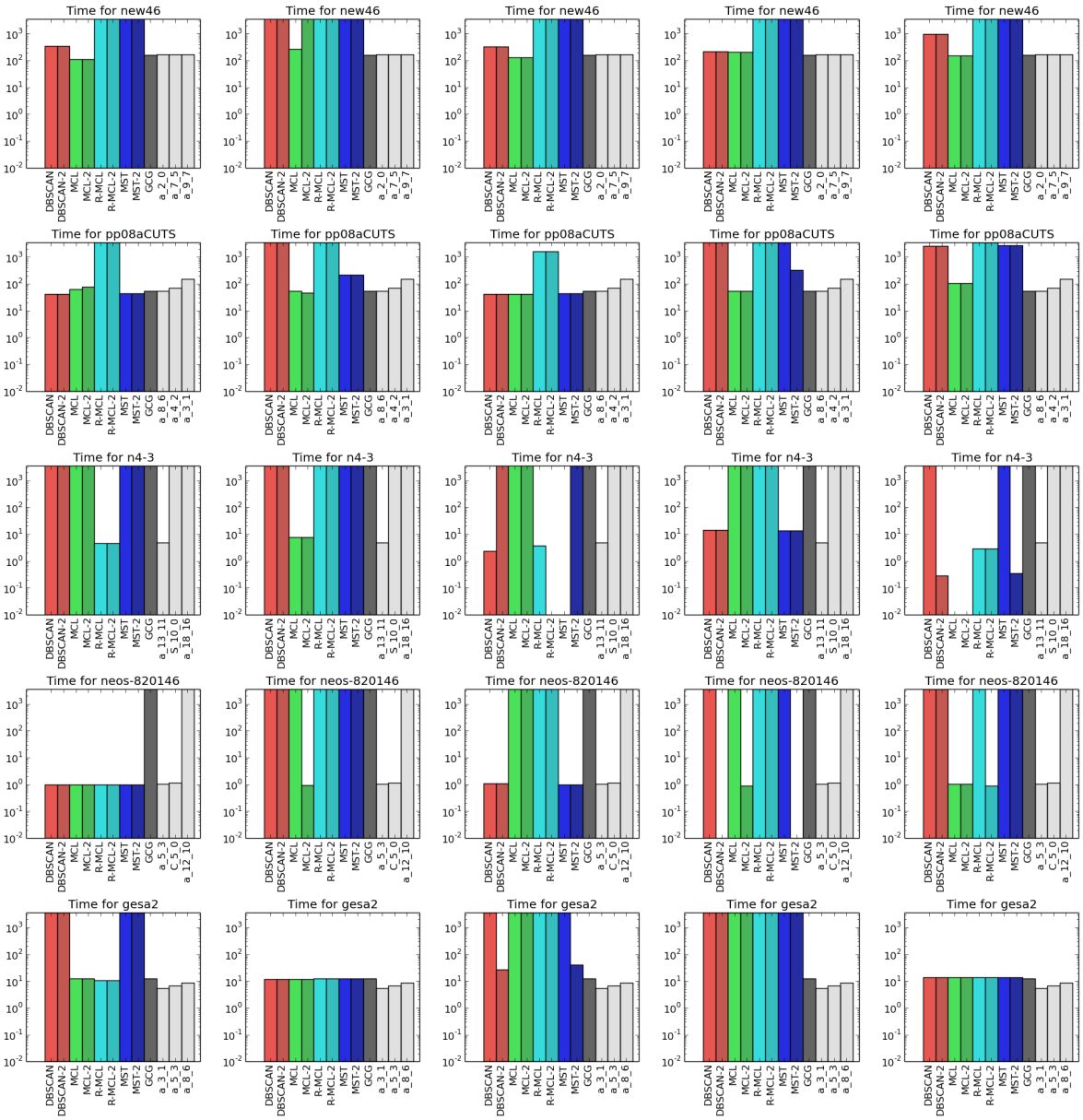
After choosing the instances from these data sets, we group them in different test sets. In total we have made six test sets with approximately four to five instances in each one of them. This makes twenty seven instances and for each instance we had eight clusterings used in the comparison for each of the similarity measures defined in the Section 3.1 excluding measures 3,4 and 7. Combined together, this gives $27 * 8 * 5 = 1080$ decompositions that we have obtained. Thus we will not discuss all of the decompositions obtained, but rather try to pull out some conclusions, or to give some interesting observations from the obtained results.

4.3 General performance

Here we present a sample from our test set and we discuss the results in general. In Figures 4.1 and 4.2 you can see the runtime comparison for ten different instances. Each row corresponds to one instance and each column corresponds to one similarity measure.

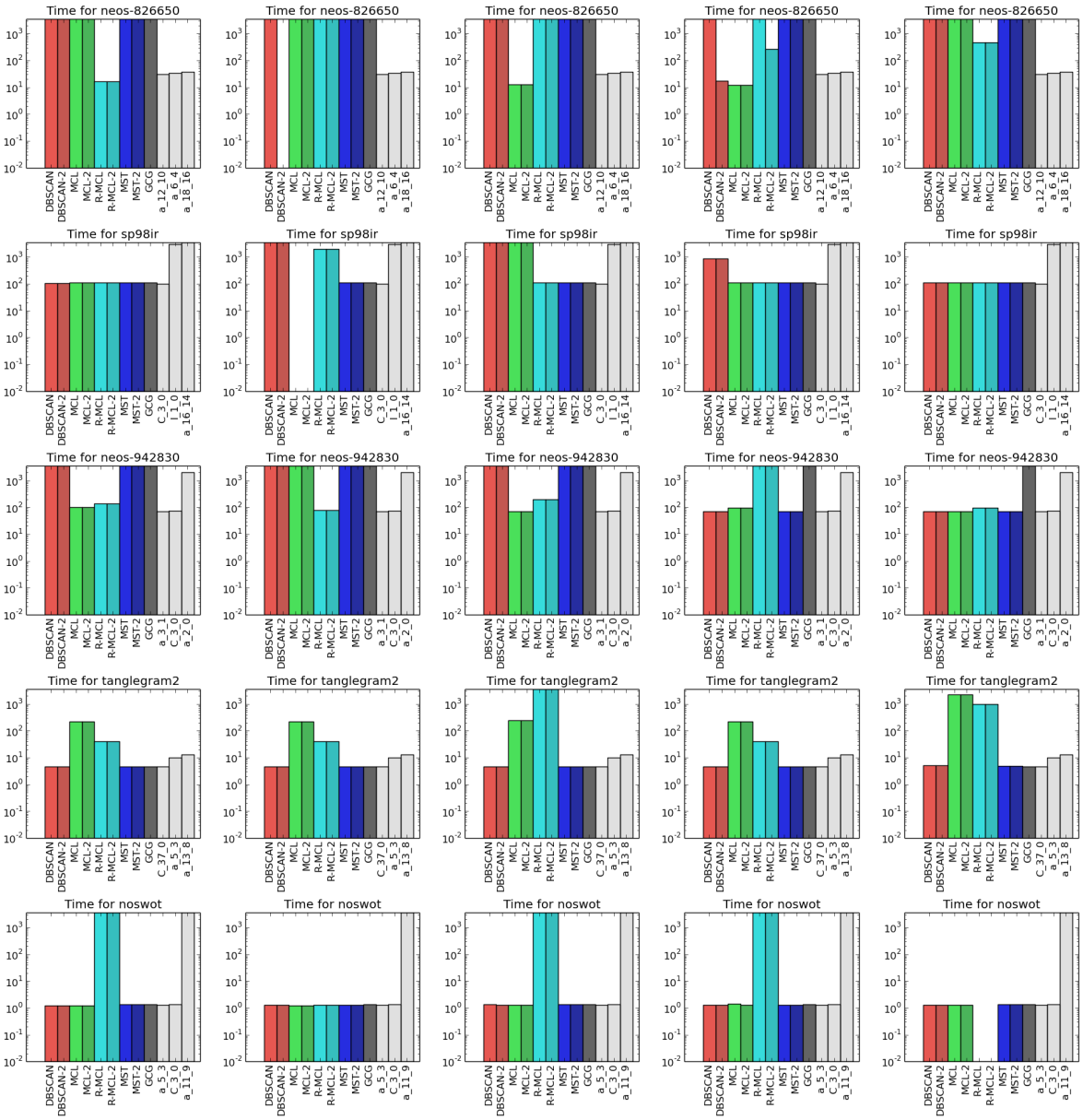
In the figures we observe that our clustering methods, when combined together, did behave well in most of the cases. In almost all cases, at least one decomposition obtained by the clustering algorithms was equally good or better than those obtained by the GCG. Also we can see that it is very hard to pull some general conclusion for a specific clustering algorithm or a specific similarity measure. For example, Figure .17 shows that very small variations in the clusterings can result in large differences in the solving time. With that said, we cannot predict what clustering is optimal for a certain problem instance, and thus we cannot generalize performance of a single algorithm or a single similarity measure.

However, when used together, these algorithms can be very useful. For example, in the Figures 4.1 and 4.2 you can see that we had some cases where R-MCL was the only one to produce good results, while often it did not contribute at all. In addition, we have observed that DBSCAN has one small advantage. Namely, if some structure exits in the matrix it is likely to be discovered by DBSCAN and also, when no suitable structure exists, DBSCAN returns one big cluster, which in some cases (e.g. in fiber, neos-1224597, air04) was shown to be better solution. We also notice that DBSCAN and MST produce very similar result (this can be concluded from the Figures given in the



(a) Johnson (b) Intersection (c) Jaccard (d) Cosine (e) Simpson

Figure 4.1: Time comparison for different clusterings and different similarity measures.



(a) Johnson (b) Intersection (c) Jaccard (d) Cosine (e) Simpson

Figure 4.2: Time comparison for different clusterings and different similarity measures.

Appendix 5.2) with an additional feature that MST **always** discovers the pure diagonal structure.

From the Table 4.1 we can see statistics, which give us a rough estimate of the algorithms and similarity measures performance. From the table we can see that the best results were obtained with the MST and MCL clustering algorithms. Also, we can notice that DBSCAN had good results, while R-MCL has shown the worst results among these four algorithms.

	DBSCAN			MCL			R-MCL			MST			Total Per Sim.		
	b	w	fail	b	w	fail	b	w	fail	b	w	fail	b	w	fail
Johnson	18	7	23	19	9	21	17	4	27	21	4	24	75	24	95
Intersection	13	5	21	20	6	23	16	5	26	20	9	16	69	25	86
Jaccard	18	7	21	15	8	27	13	6	28	26	1	15	72	22	91
Cosine	21	4	18	22	4	24	8	5	35	22	3	20	73	16	97
Simpson	20	4	23	23	8	17	14	6	28	22	7	21	79	25	89
Total per Alg.	90	27	106	99	35	112	68	26	144	111	24	96			

Table 4.1: Final statistics. Meaning of the fields: b - number of cases where runtime with clustering found decompositions was better or equal than GCG found decs, w - number of cases where GCG found decompositions were better, fail - number of cases where clustering found decompositions failed after the time limit

From the table we can observe that there was no similarity measure that is dominant over the others, or that there is one similarity that is remarkably worse than the rest. All of the similarities shown in the evaluation have relatively good results. From the table 4.1 we can see that the Simpson similarity is just slightly better than the others, while the Intersection similarity is a bit worse.

Finally, we can notice a small correlation between "good clustering" (i.e. small lower bound and similarly sized clusters) and reduced solving times. Furthermore, as stated by Bergner et al. (2015), we notice that the pure diagonal structure is **usually** better than the one with the lower bound. But also here we find a counter example. If Figure

.11 we discover that in the case of the instance "neos-826650" R-MCL has given better results than MCL or MST, which have indeed discovered the pure diagonal structure.

4.3.1 Clustering running time

In the Table 4.2 we provide a runtime comparison for the used clustering algorithms. For this comparison we use problem instances of different sizes and densities. In this study we run DBSCAN and MST up to 49 times (for different *eps*), MCL up to 15 times (for different *inflate factor*) and R-MCL up to 8 times (for different *inflate factor*). Also, in each run of the MCL or R-MCL we have limited the number of loops to 20.

Instance	Size	Density	DBSCAN	MCL	R-MCL	MST
noswot	182	9.7%	0.02	0.7	1.9	~ 0
fiber	363	2.4%	0.04	2.4	3.2	~ 0
neos-820146	830	2.9%	0.06	9.9	19.3	~ 0
n4-3	1236	2.1%	0.08	14.8	57.1	~ 0
wachplan	1553	20%	0.25	177.4	190	0.006
neos-948126	7272	0.3%	0.2	289	117.8	0.002
rmatr100-p5	8685	1.07%	0.45	83.9	106	0.01
neos18	11402	0.67%	1.02	183	733	0.01

Table 4.2: Running time comparison for clustering algorithms. In column "Size" is the number of constraints and in column "Density" is the percentage of non-zero values in the similarity matrix for Johnson similarity measure.

We observe that DBSCAN and MST are very fast in all of the cases, while MCL and R-MCL are efficient only in the cases up to certain size and density. It is also to notice that the runtimes of MCL and R-MCL heavily depended on the matrix density and not only on the size of the matrix.

4.4 Special case: bin packing problem

In this section we talk about a special case that has occurred during testing. Namely, we could not use clustering algorithms to find special structures in the constraint matrices of the bin packing problem and any other equivalent problem. In this group, we also include the GAP, the p-median, the cpmp problems. The reason for that is very

simple. According to Valerio de Carvalho (n.d.), the best decomposition for these kind of problems is the following:

- For each bin make a block of size one with the corresponding capacity constraint
- Put the rest of the constraints (allocation constraints) in the lower border.

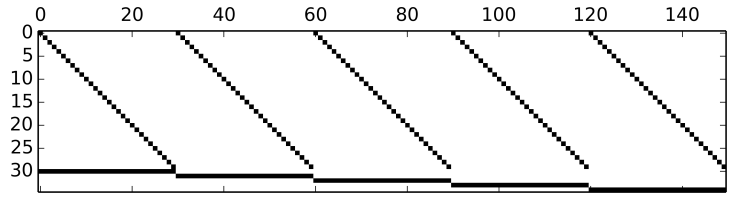


Figure 4.3: Example of BPP-like constraint matrix with 4 bins.

In order to understand why clustering techniques are useless here, we have visualized a BPP constraint matrix in the Figure 4.3. In this example, we have four bins, which are represented with four lower rows, and thirty allocation constraints which are in the upper part of the matrix. As you can notice in the figure, no matter what similarity measure we use to describe this matrix, we will always have the same similarity between any two allocation constraints as we will have between any two capacity constraints. Namely, no pair of capacity nor pair of allocation constraints has any common variables. This makes it impossible for the clustering algorithms to distinguish between capacity and allocation constraints. In conclusion, we are unable to recognize one cluster per capacity constraint, while labelling the allocation constraints as non-clustered.

5 Summary and future work

5.1 Future work

Markov chain clustering

As mentioned in the Section 3.2.3, Markov chain clustering is not very efficient approach. Thus we could find some more efficient variations of MCL. For example, we could implement Multi-level Regularized MCL (MLR-MCL) method described by Satuluri & Parthasarathy (2009). This method has shown to produce results similar or better to those of MCL, and the method is scalable in the contrast to the MCL.

Post-processing

In this study we have used only two criteria for finding the best clusterings for each algorithm. In the future, we could think of some other scoring functions in order to determinate what clustering should be chosen. As our results have shown, the smallest lower border or the smallest score s was not always the best choice and thus a further research has to be done on how to evaluate the given decompositions. Furthermore, in this study we have used a very simple method for removing the rows with colliding variables, and in some cases we have noticed that in the post-processing step more rows were removed from the blocks than actually needed. For that reason, the optimal solution for this problem should be found.

Implementation

Currently our code is implemented in Python as a separate program, independent of GCG. Because of the limitations of Python, some parts of the code were very slow. For example the pre-processing and post-processing steps took very long time to finish, even though the same steps implemented in the GCG took significantly less time. For that reason, it would be better to implement this code in C/C++, or maybe even as a part of the GCG itself. If it was part of the GCG, than we could have certainly got better results, because here we did not include the pre-solving step of the solver which can simplify the problem instance and thus reduce the solving time. We were not able to do

that because our program was a separate entity, and the pre-solving step is executed by the solver before we load our decomposition.

5.2 Summary

This thesis was one step further in the goal to try to automatically find suitable Dantzig-Wolfe decompositions. The particular goal of the thesis was to examine a completely different approach for decomposition detection, namely the clustering techniques. We have compared four algorithms with each other and also with existing decomposition detectors in the GCG. In addition, we have compared different similarity measures used in the clustering step.

Chosen clustering algorithms and similarity measures have shown that they were able to detect the special structures in most of the cases and often very efficiently indeed. Also the resulting IP solving time was sustainable, and at least one, but often more decompositions found by clustering algorithms have shown to have better or equally good performance in comparison to the decompositions obtained by already existing detectors.

In conclusion, we believe that this novel approach was a good choice for automatic decomposition detection and we believe that it could be improved even more with the steps proposed in the Future work section.

References

Achterberg, T., Koch, T., & Martin, A. (2006). MIPLIB 2003. *Operations Research Letters*, *34*(4), 361–372. Retrieved from http://www.zib.de/Publications/abstracts/ZR-05-28/ doi: 10.1016/j.orl.2005.07.009

A. J. Enright, S. V. D., & Ouzounis, C. A. (2002). An efficient algorithm for large-scale detection of protein families. , *30*(7), 1575–1584.

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., & Vance, P. H. (1998, March). Branch-and-price: Column generation for solving huge integer programs. *Oper. Res.*, *46*(3), 316–329. Retrieved from http://dx.doi.org/10.1287/opre.46.3.316 doi: 10.1287/opre.46.3.316

Bergner, M., Caprara, A., Ceselli, A., Furini, F., Luebbecke, M., Malaguti, E., & Traversi, E. (2015, January). Automatic dantzig-wolfe reformulation of mixed integer programs. *Math. Prog.*, *149*(1-2), 391-424. Retrieved from http://www.or.rwth-aachen.de/research/publications/s10107-014-0761-5.pdf

Berkhin, P. (2002). *Survey of clustering data mining techniques* (Tech. Rep.). Accrue Software, Inc.

Cheng, D., Kannan, R., Vempala, S., & Wang, G. (2006, December). A divide-and-merge methodology for clustering. *ACM Trans. Database Syst.*, *31*(4), 1499–1525. Retrieved from http://doi.acm.org/10.1145/1189769.1189779 doi: 10.1145/1189769.1189779

Choi, S.-s., Cha, S.-h., & Tappert, C. C. (2010). A Survey of Binary Similarity and Distance Measures. *Journal on Systemics, Cybernetics and Informatics*, *0*(1), 43–48.

Dantzig, G. B., Orden, A., & Wolfe, P. (1955). The generalized simplex method for minimizing a linear form under linear inequality restraints. *Pacific Journal of Mathematics*(5), 183–195.

Dantzig, G. B., & Wolfe, P. (1960). Decomposition principle for linear programs. *Operations Research*, *8*(1), 101-111. doi: 10.1287/opre.8.1.101

Dasgupta, S., & Long, P. M. (2005, June). Performance guarantees for hierarchical clustering. *J. Comput. Syst. Sci.*, *70*(4), 555–569. Retrieved from `http://dx.doi.org/10.1016/j.jcss.2004.10.006` doi: 10.1016/j.jcss.2004.10.006

Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data via the EM Algorithm. *Journal of the Royal Statistical Society. Series B (Methodological)*, *39*(1), 1–38. Retrieved from `http://web.mit.edu/6.435/www/Dempster77.pdf` doi: 10.2307/2984875

Desrosiers, J., & Luebbecke, M. E. (2005, January). *A primer in column generation.* New York: Springer.

Desrosiers, J., & Luebbecke, M. E. (2010). Branch-price-and-cut algorithms.

Engineering graph clustering: Models and experimental evaluation. (2008). *J. Exp. Algorithmics*, *12*, 1.1:1–1.1:26. Retrieved from `http://doi.acm.org/10.1145/1227161.1227162` doi: 10.1145/1227161.1227162

Gamrath, G. (2010). Generic branch-cut-and-price.

Gamrath, G., & Lübbecke, M. E. (2010). Experiments with a Generic Dantzig-Wolfe Decomposition for Integer Programs. In P. Festa (Ed.), *Proceedings of the 9th international symposium on experimental algorithms (sea)* (Vol. 6049, pp. 239–252). Berlin: Springer-Verlag. doi: http://dx.doi.org/10.1007/978-3-642-13193-6_21

Giancarlo, R., Lo Bosco, G., & Pinello, L. (2010). Distance functions, clustering algorithms and microarray data analysis. In C. Blum & R. Battiti (Eds.), *Learning and intelligent optimization* (Vol. 6073, p. 125-138). Springer Berlin Heidelberg. Retrieved from `http://dx.doi.org/10.1007/978-3-642-13800-3_10` doi: 10.1007/978-3-642-13800-3_10

Hartley, H. O. (1958). Maximum Likelihood Estimation from Incomplete Data. *Biometrics*, *14*(2), 174–194. Retrieved from `http://dx.doi.org/10.2307/2527783` doi: 10.2307/2527783

Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual acm symposium on theory of computing* (pp. 302–311). New York, NY, USA: ACM. doi: 10.1145/800057.808695

Khachiyan, L. G. (1979). A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, *244*, 1093–1096.

King, B. (1967). Step-wise clustering procedures. *Journal of the American Statistical Association*, *62*(317), 86-101. doi: 10.1080/01621459.1967.10482890

Koch, T., Achterberg, T., Andersen, E., Bastert, O., Berthold, T., Bixby, R. E., ... Wolter, K. (2011). MIPLIB 2010. *Mathematical Programming Computation*, *3*(2), 103–163. Retrieved from `http://mpc.zib.de/index.php/MPC/article/view/56/28` doi: 10.1007/s12532-011-0025-9

Kruskal, J. B. (1956). On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, *7*(1), 48-50. Retrieved from `http://www.jstor.org/stable/2033241`

Manning, C. D., Raghavan, P., & Schütze, H. (2008). *Introduction to information retrieval*. New York, NY, USA: Cambridge University Press.

Martin Ester, J. S., Hans-Peter Kriegel, & Xu, X. (1996). A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96)*.

Morris, O., Lee, M. J., & Constantinides, A. (1986, April). Graph theory for image analysis: an approach based on the shortest spanning tree. *Communications, Radar and Signal Processing, IEE Proceedings F*, *133*(2), 146-152. doi: 10.1049/ip-f-1.1986.0025

Papadimitriou, C. H. (1981, October). On the complexity of integer programming. *J. ACM*, *28*(4), 765–768. Retrieved from `http://doi.acm.org/10.1145/322276.322287` doi: 10.1145/322276.322287

Renegar, J. (1988). A polynomial-time algorithm, based on newton's method, for linear programming. *Math. Program.*, *40*(1), 59–93. Retrieved from `http://dx.doi.org/10.1007/BF01580724` doi: 10.1007/BF01580724

Satuluri, V., & Parthasarathy, S. (2009). Scalable graph clustering using stochastic flows: Applications to community discovery. In *Proceedings of the 15th acm sigkdd international conference on knowledge discovery and data mining* (pp. 737–746). New York, NY, USA: ACM. Retrieved from `http://doi.acm.org/10.1145/1557019.1557101` doi: 10.1145/1557019.1557101

Schrijver, A. (2003). *Combinatorial optimization: Polyhedra and efficiency.* Springer.

Sneath, P. H. A., & Sokal, R. R. (1962, March). Numerical Taxonomy. *Nature*, *193*(4818), 855–860. Retrieved from http://mfkp.org/INRMM/article/13078534 doi: 10.1038/193855a0

Valerio de Carvalho, J. (n.d.). Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, *86*(0), 629–659. Retrieved from http://dx.doi.org/10.1023/A:1018952112615 doi: 10.1023/A:1018952112615

Vanderbeck, F. (2000). On dantzig-wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research*, *48*(1), 111–128. Retrieved from http://dx.doi.org/10.1287/opre.48.1.111.12453 doi: 10.1287/opre.48.1.111.12453

Vanderplas, J. (2015). *Minimum spanning tree clustering.* Retrieved 2016-02-10, from https://github.com/jakevdp/mst_clustering

van Dongen, S. (2000). *Graph clustering by flow simulation* (Ph.D Thesis). Centre for Mathematics and Computer Science (CWI) in Amsterdam.

Xu, Y., Olman, V., & Xu, D. (2002). Clustering gene expression data using a graph-theoretic approach: an application of minimum spanning trees. *Bioinformatics*, *18*(4), 536-545. Retrieved from http://bioinformatics.oxfordjournals.org/content/18/4/536.abstract doi: 10.1093/bioinformatics/18.4.536

Zahn, C. (1971, Jan). Graph-theoretical methods for detecting and describing gestalt clusters. *Computers, IEEE Transactions on*, *C-20*(1), 68-86. doi: 10.1109/T-C.1971.223083

Appendices

In the figures shown in the appendix each row of plots represents one problem instance. The plot on the left represents the comparison of solving time by GCG with each of the following clusterings: DBSCAN with smallest lower border, DBSCAN with lowest score s (as described in 3.3) (i.e. DBSCAN-STD), MCL with smallest border, MCL with smallest score s (i.e. MCL-STD), R-MCL with smallest border, R-MCL with smallest score s (i.e. R-MCL-STD), MST with smallest border, MST with smallest score s (i.e. MST-STD). Running times obtained by these clusterings are coloured, while the running times of already existing GCG decomposition detectors are grey (with dark grey we have represented the automatic chosen decomposition). It is also to notice that the time axis is scaled logarithmically. Beside the time plots are the visualizations of the constraint matrices for each clustering as shown in 2.1.

Also, you may notice that some bars are missing from the time bars charts. If the corresponding matrix visualizations are also missing, this means that clustering algorithm has failed to produce any results. If, on the other hand, image is there while the time bar is missing, this is due to an error in GCG, where GCG has failed to output any result (e.g. solved, aborted or not feasible) for the given decomposition.

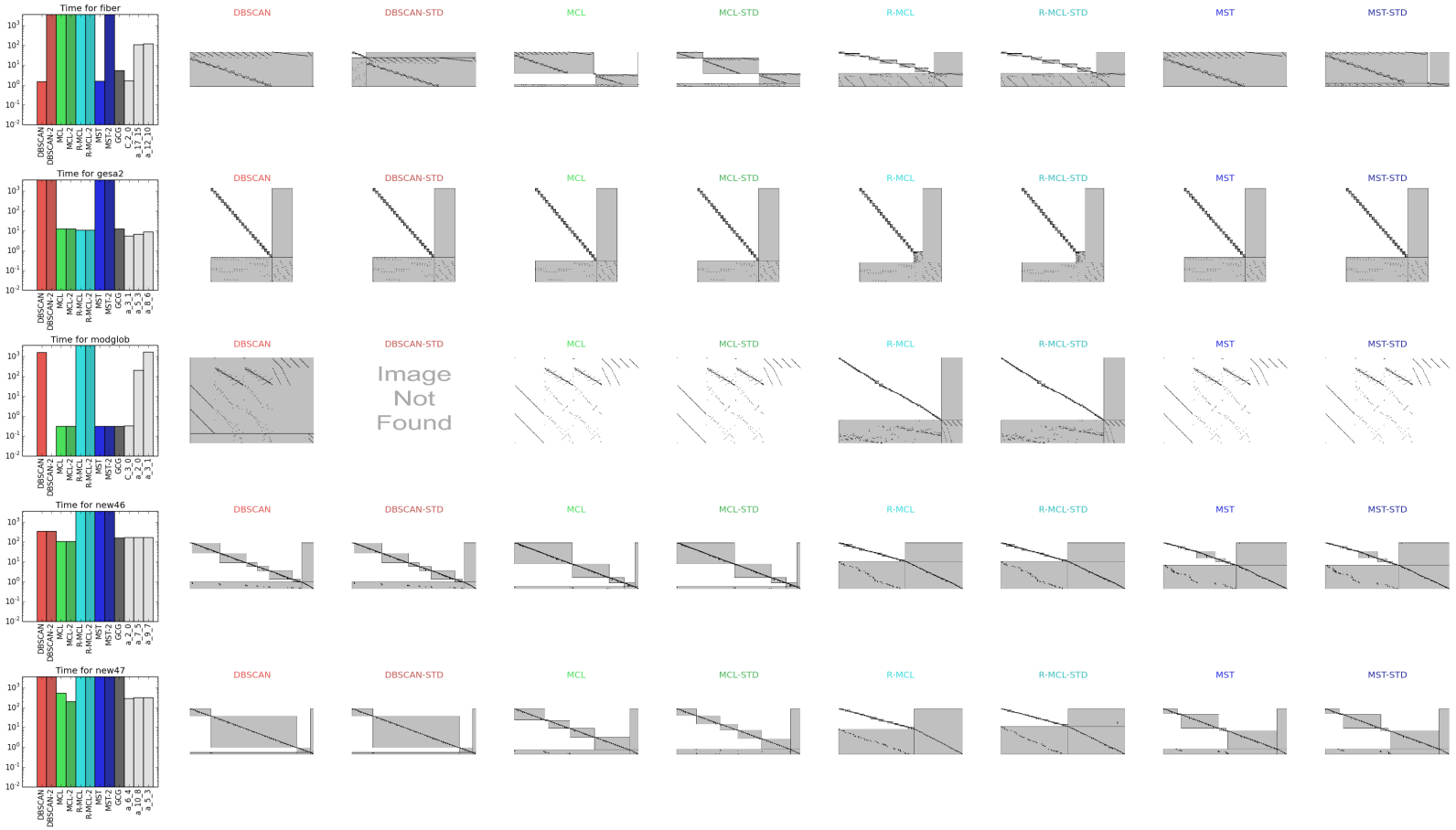


Figure .1: Test set 0, Johnson similarity

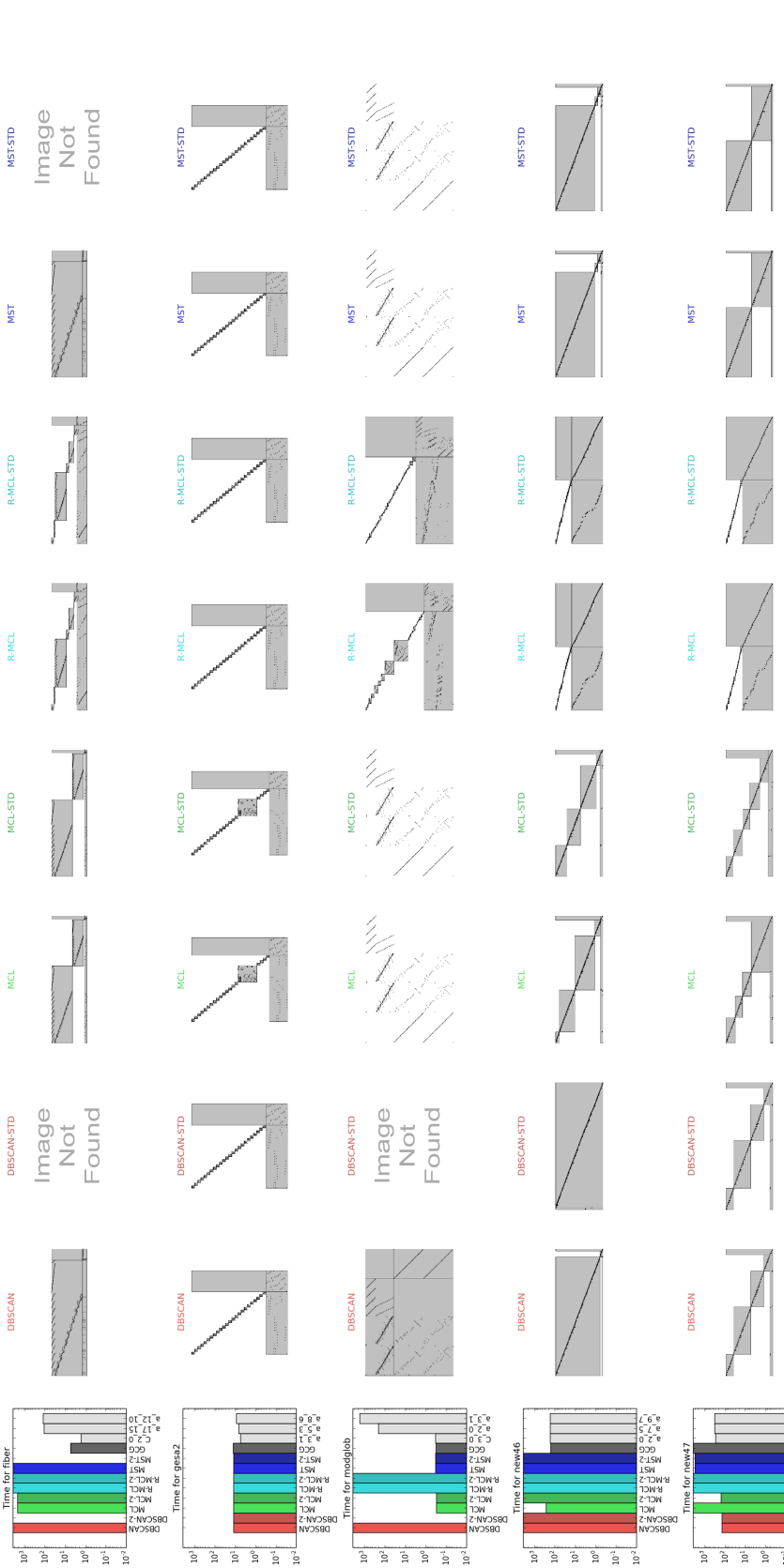


Figure .2: Test set 0, Intersection similarity

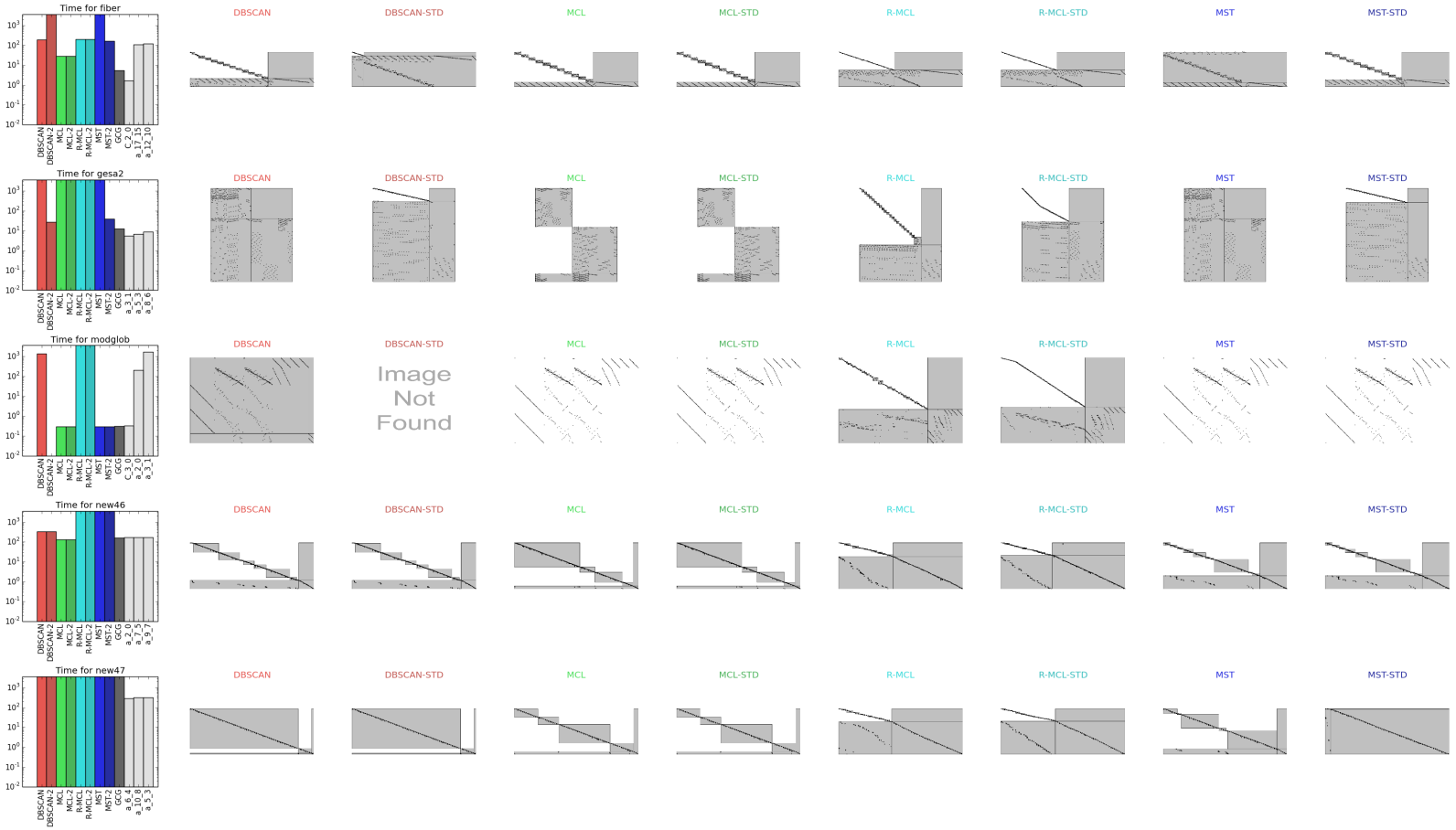


Figure .3: Test set 0, Jaccard similarity

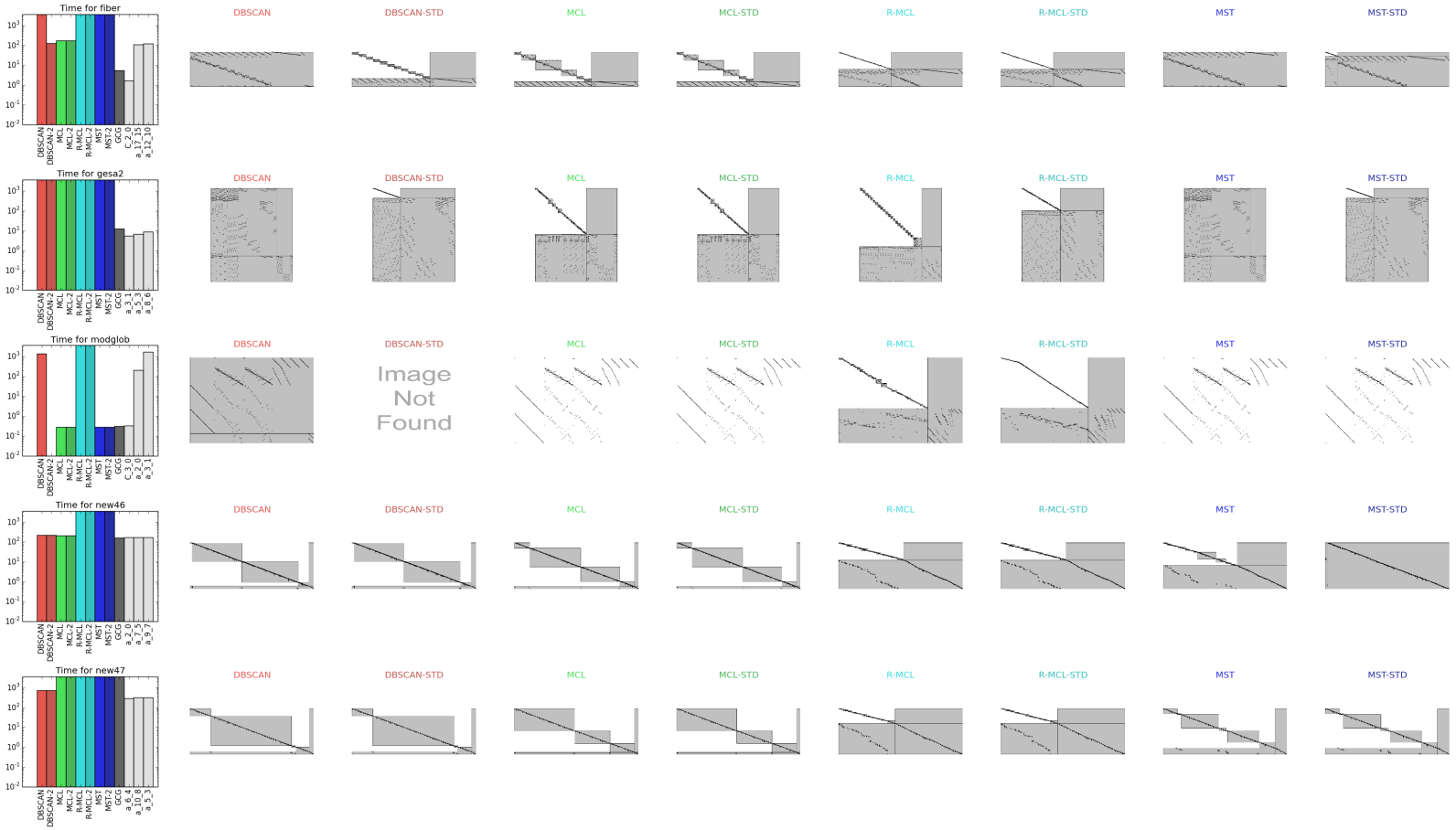


Figure .4: Test set 0, Cosine similarity

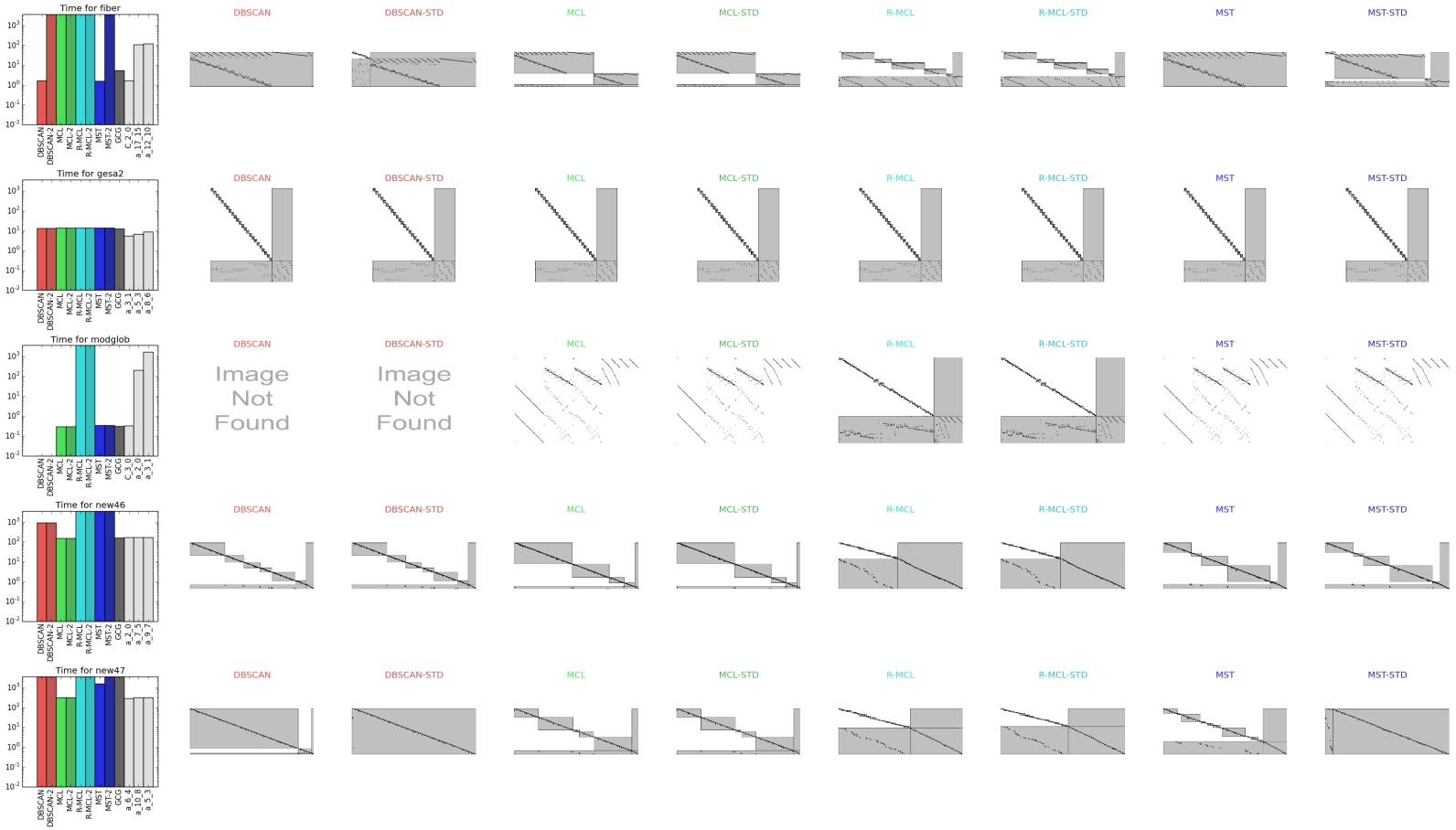


Figure .5: Test set 0, Simpson similarity

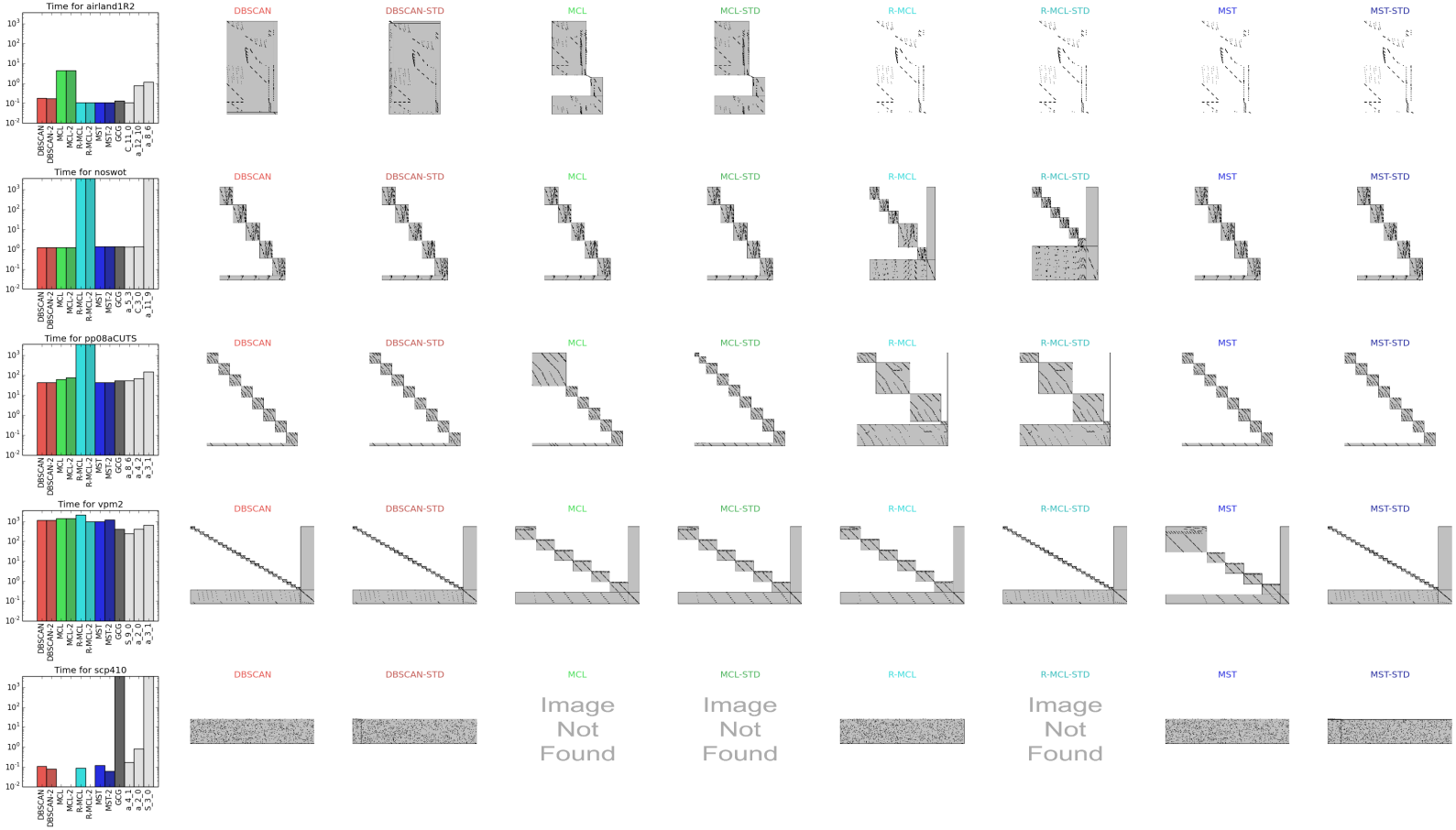


Figure .6: Test set 1, Johnson similarity

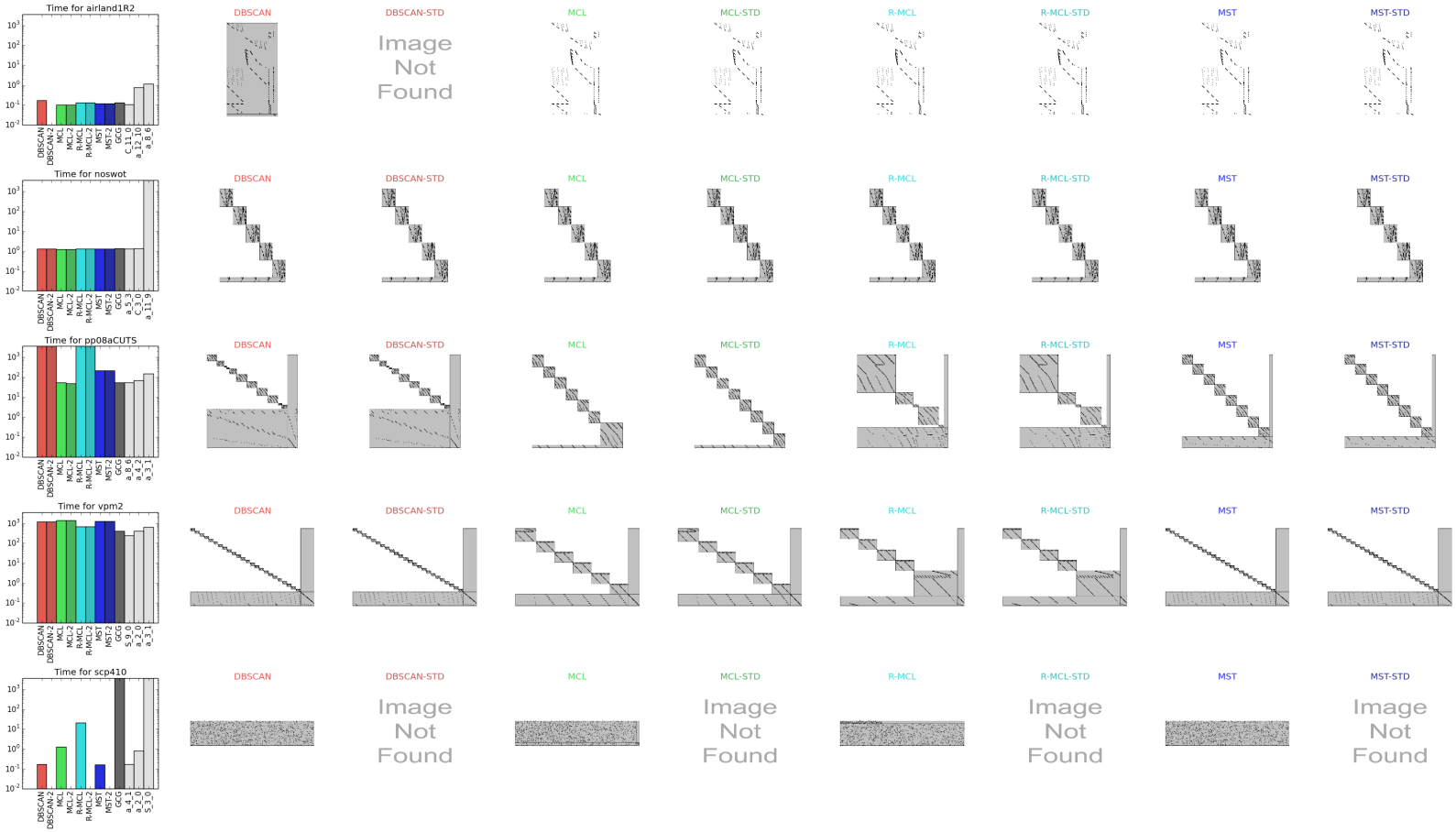


Figure .7: Test set 1, Intersection similarity

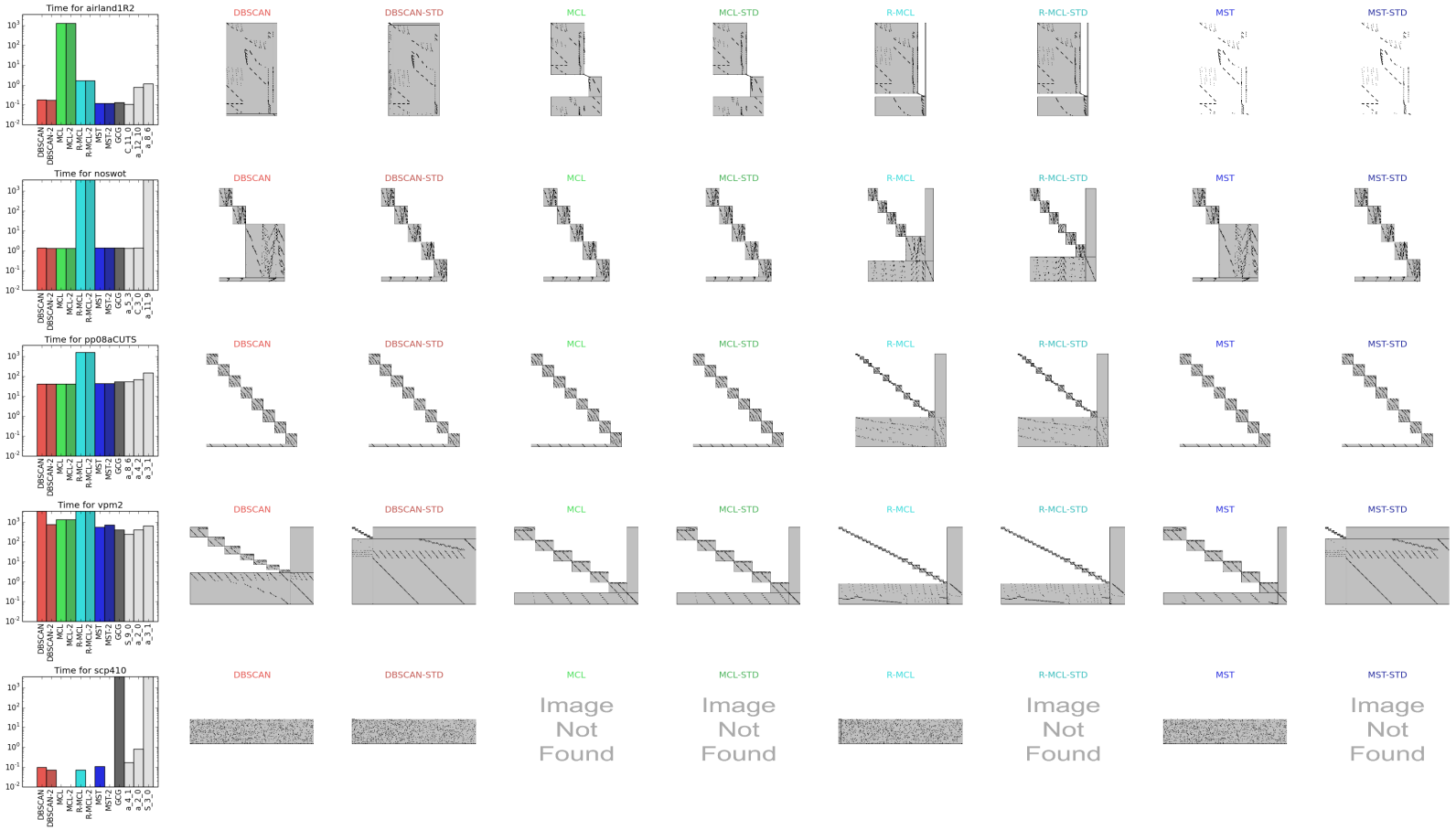


Figure .8: Test set 1, Jaccard similarity

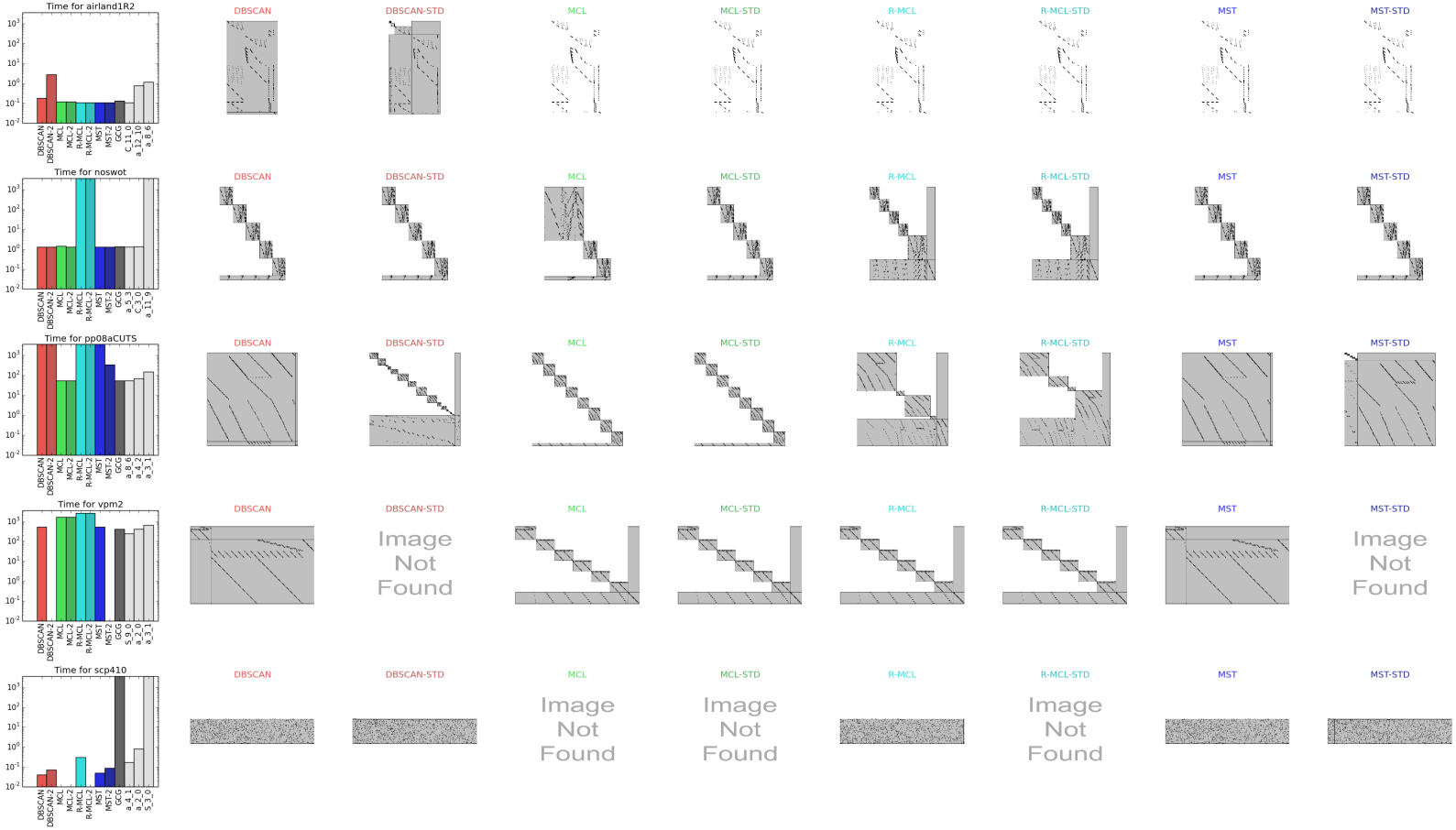


Figure .9: Test set 1, Cosine similarity

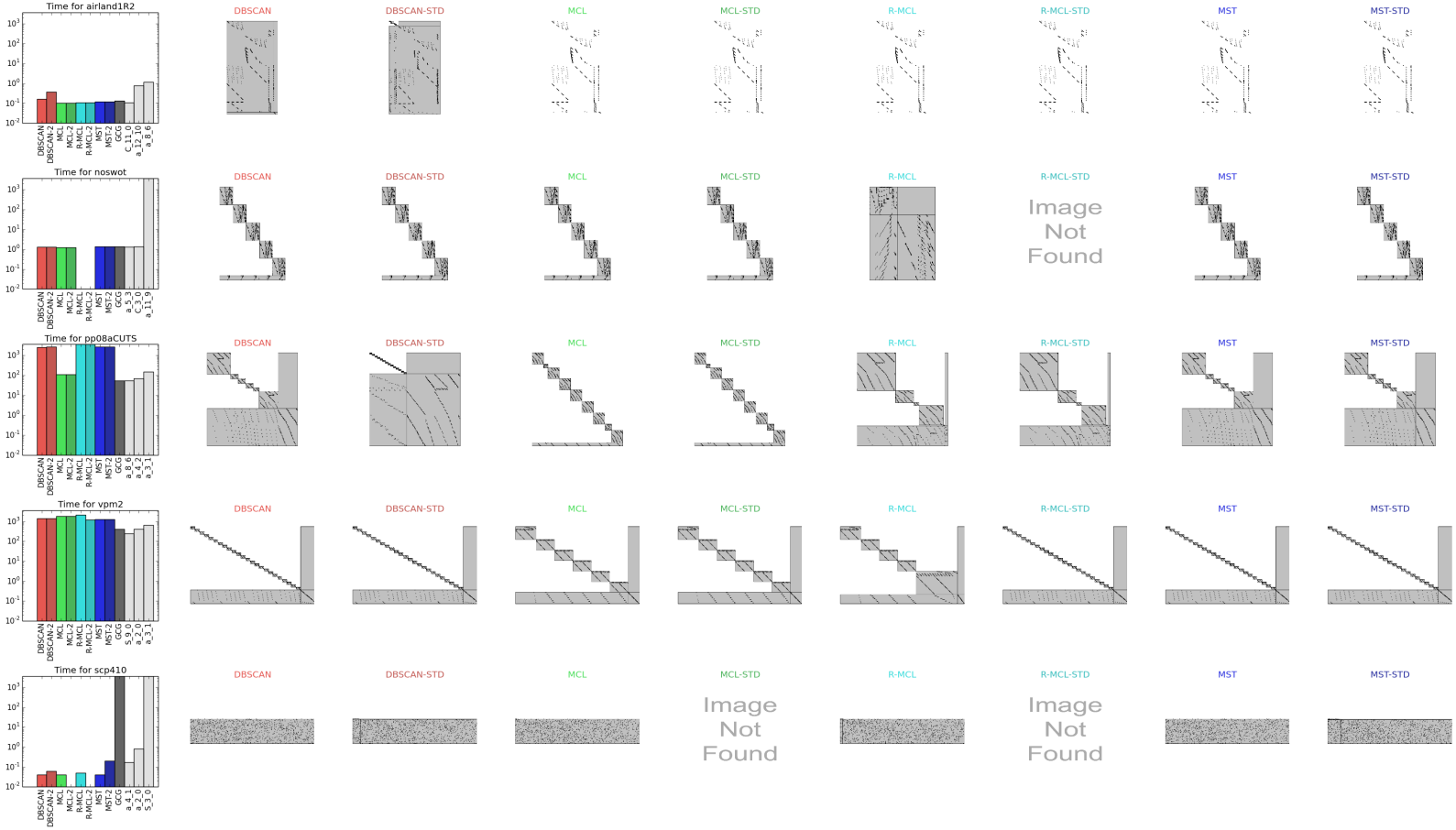


Figure .10: Test set 1, Simpson similarity

Figure .11: Test set 2, Johnson similarity

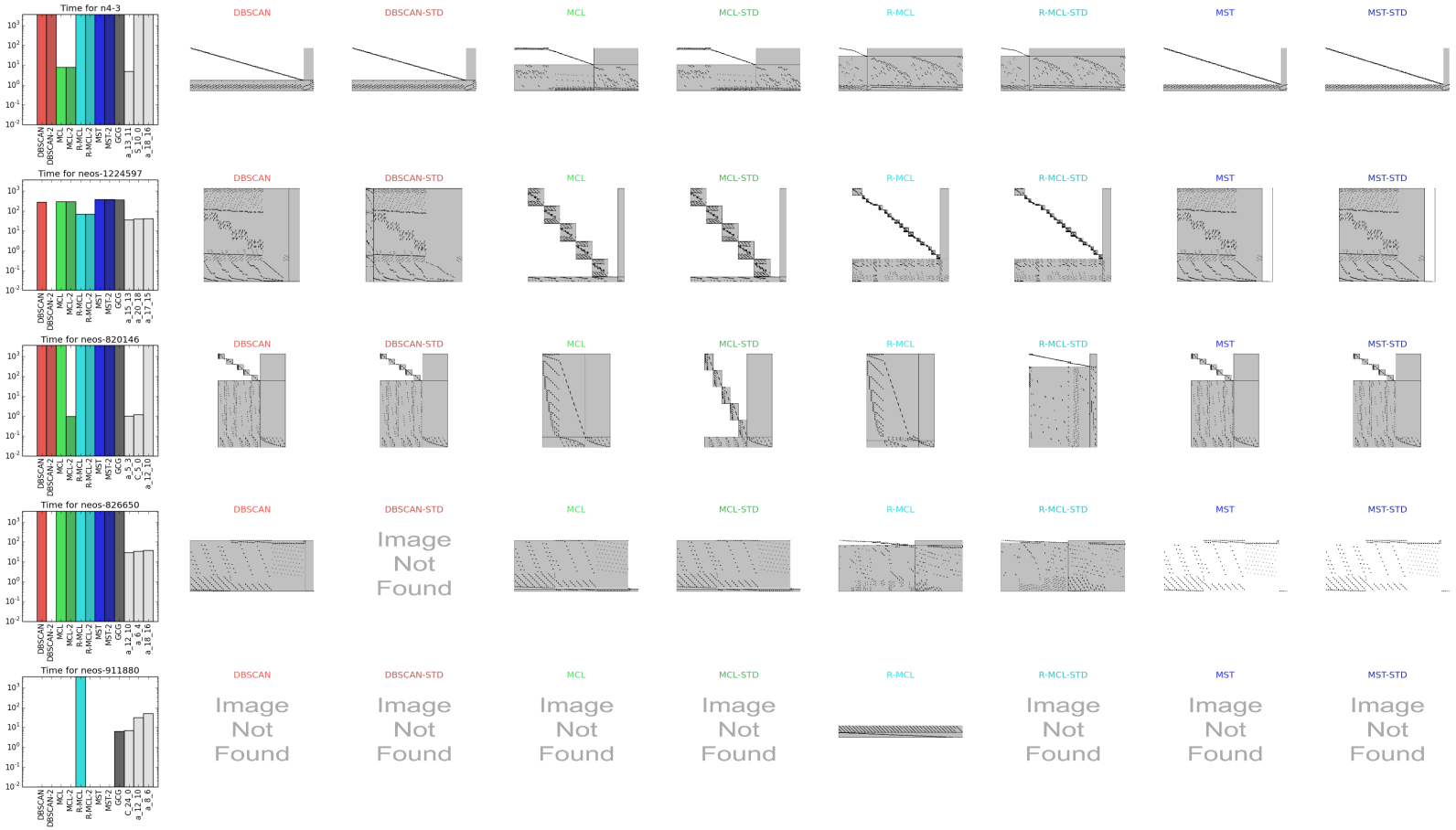


Figure .12: Test set 2, Intersection similarity

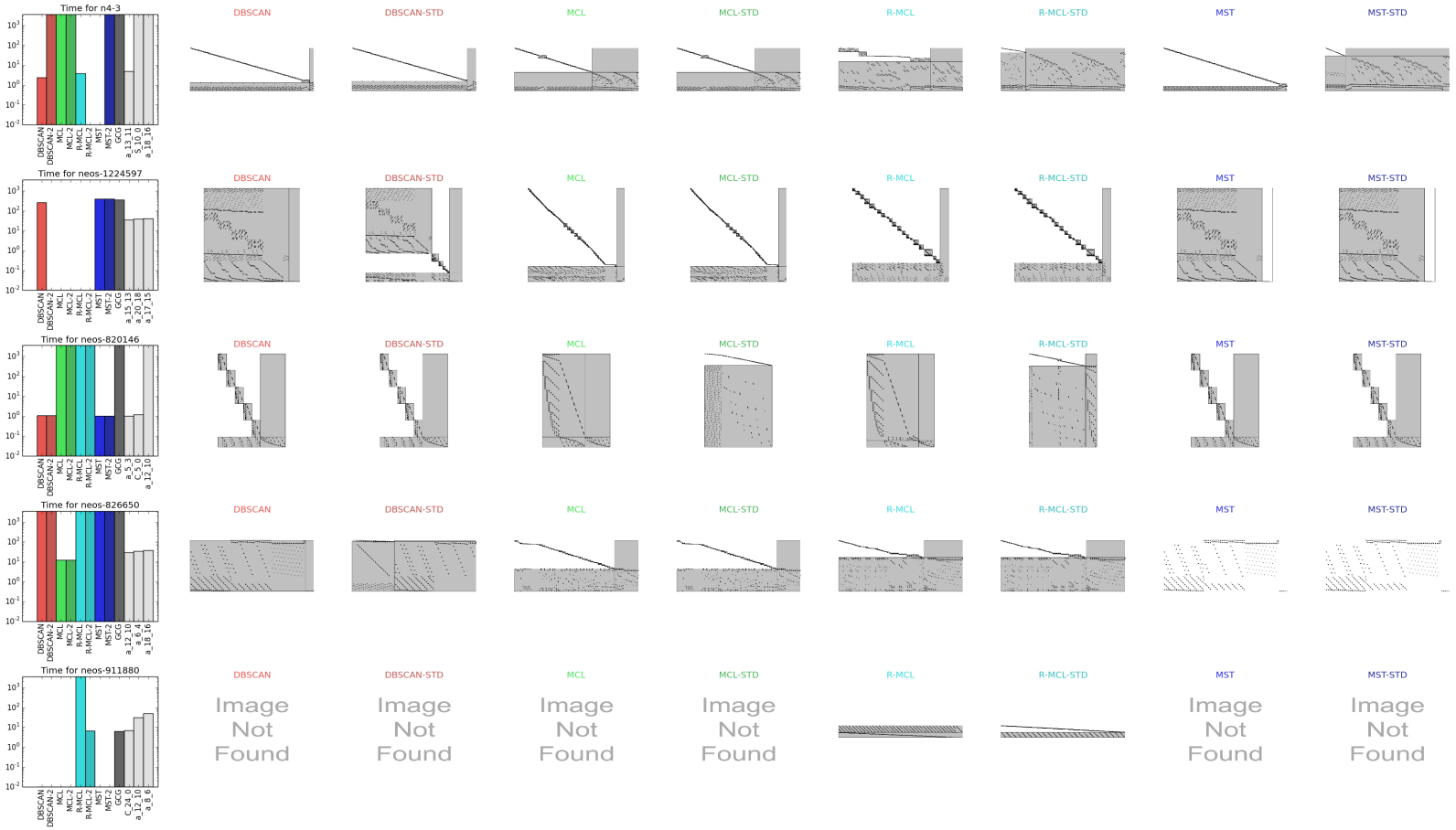


Figure .13: Test set 2, Jaccard similarity

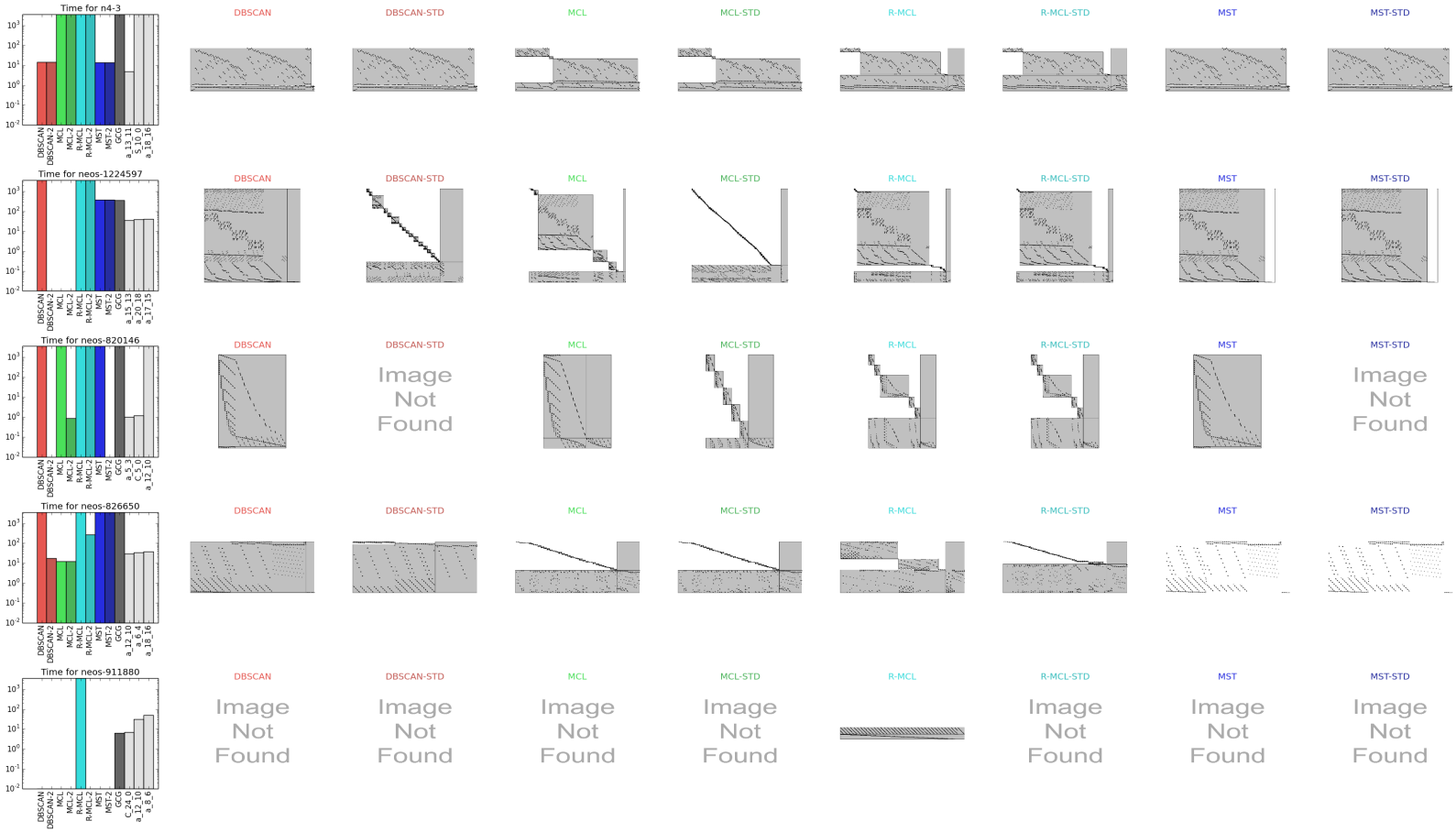


Figure .14: Test set 2, Cosine similarity

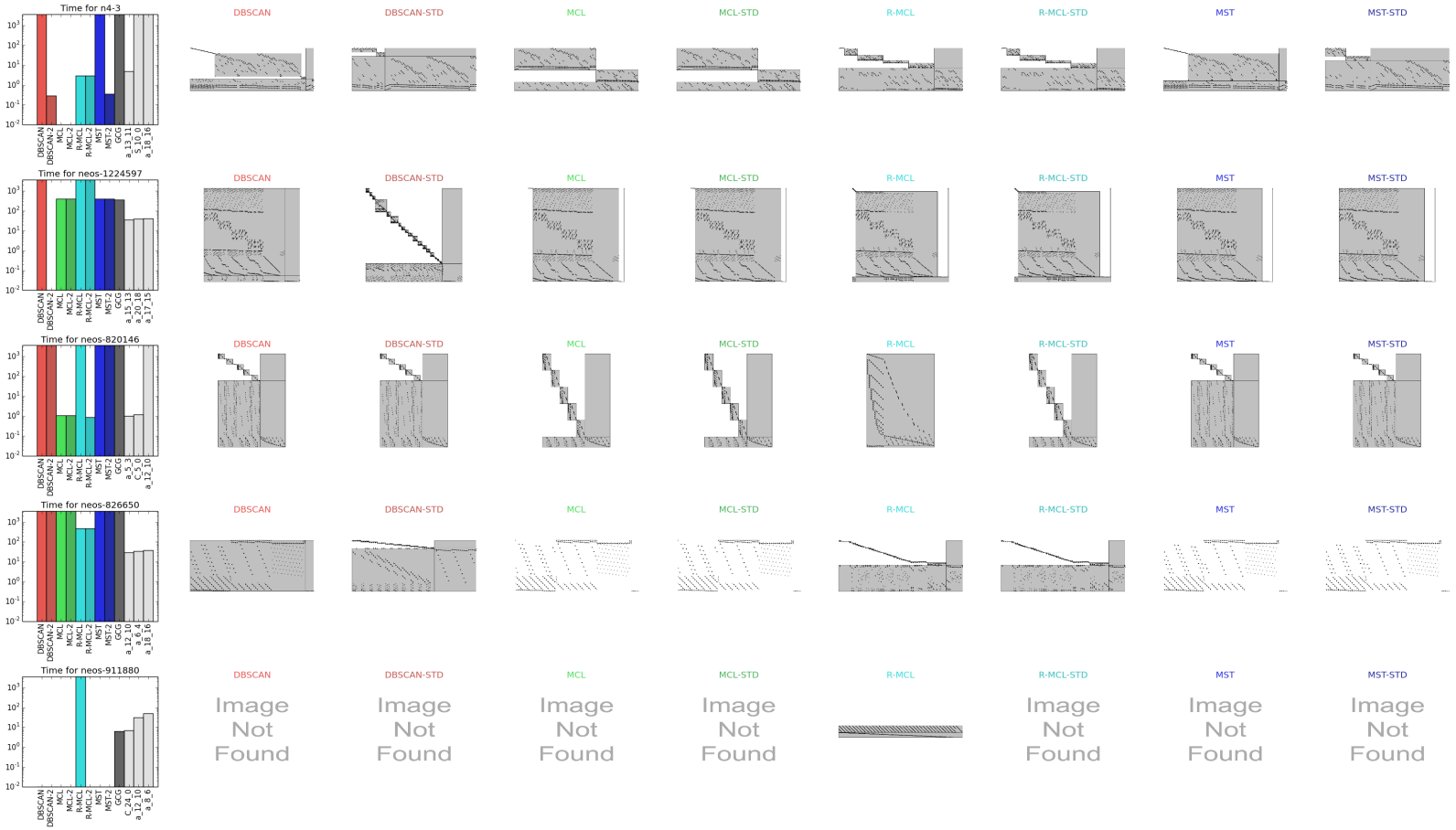


Figure .15: Test set 2, Simpson similarity

Figure .16: Test set 3, Johnson similarity

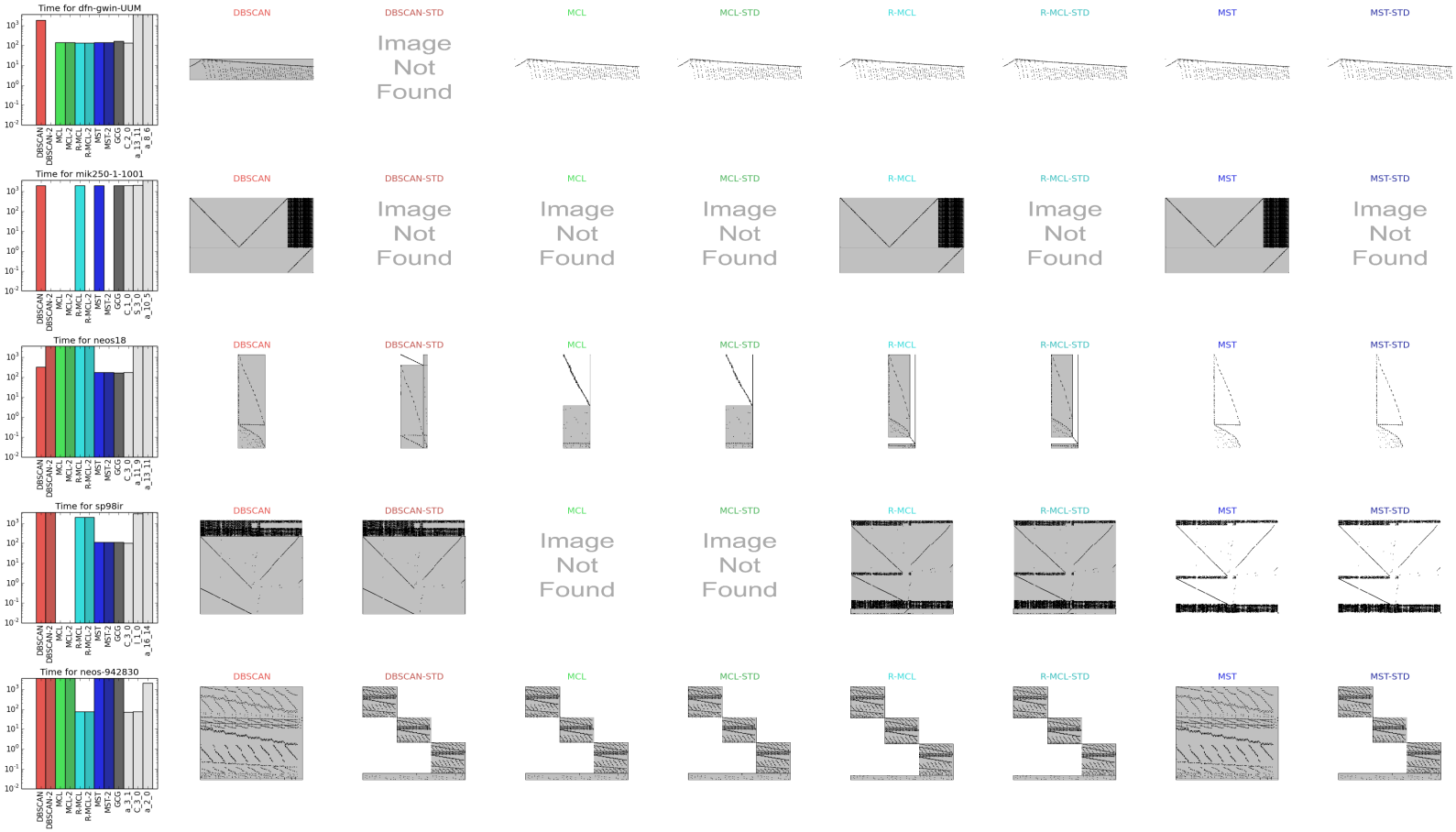


Figure .17: Test set 3, Intersection similarity

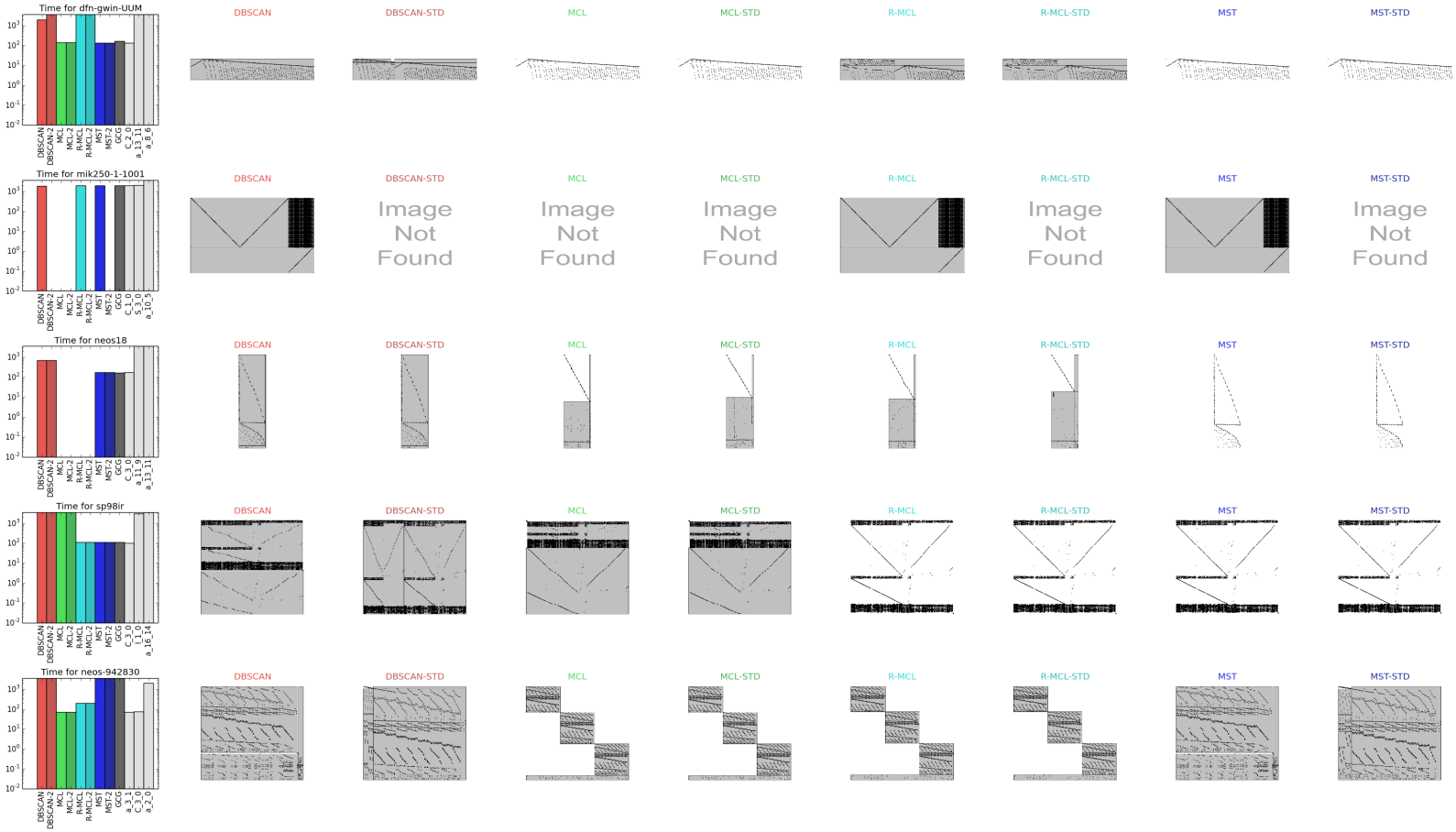


Figure .18: Test set 3, Jaccard similarity

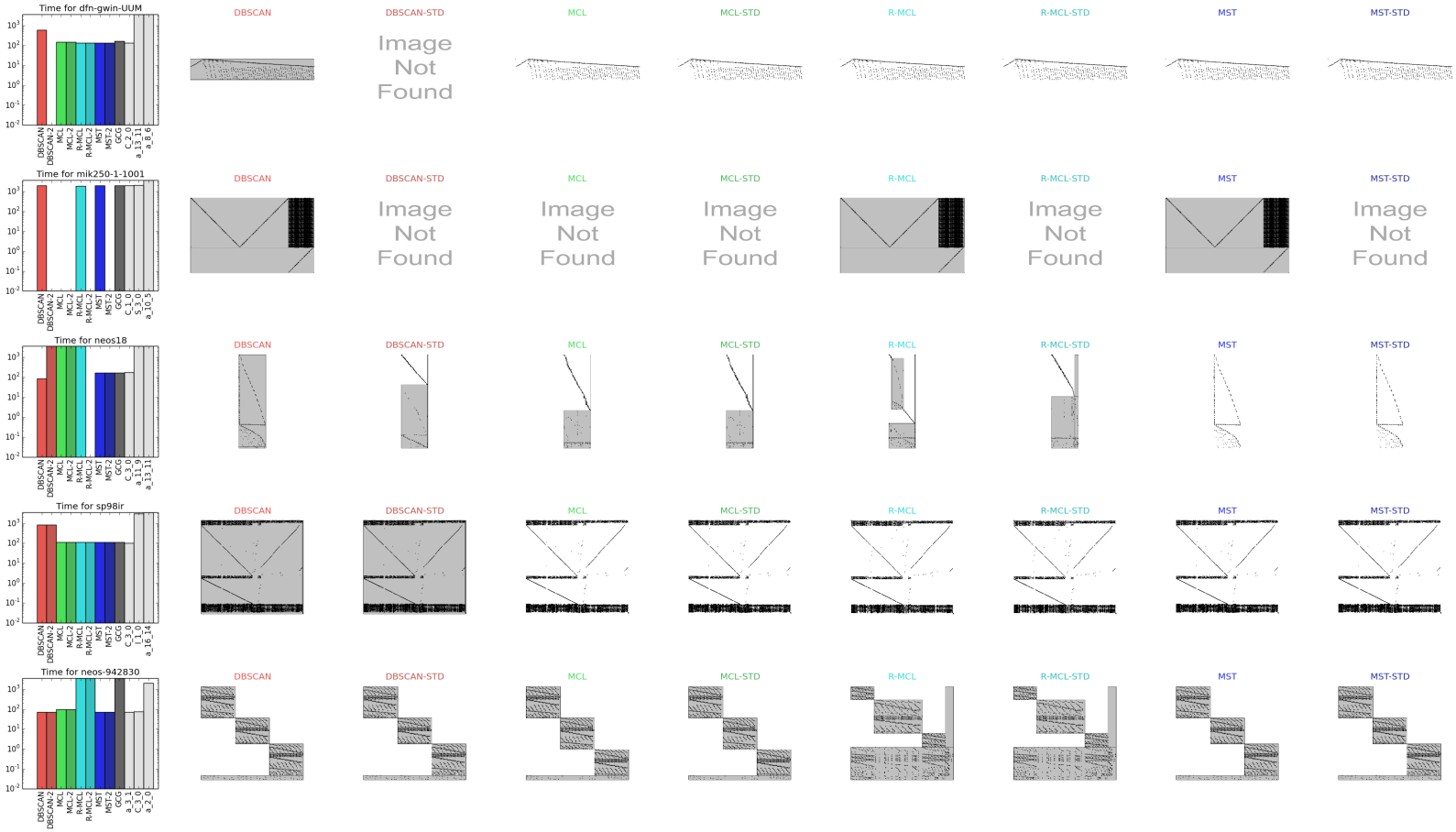


Figure .19: Test set 3, Cosine similarity

Figure .20: Test set 3, Simpson similarity

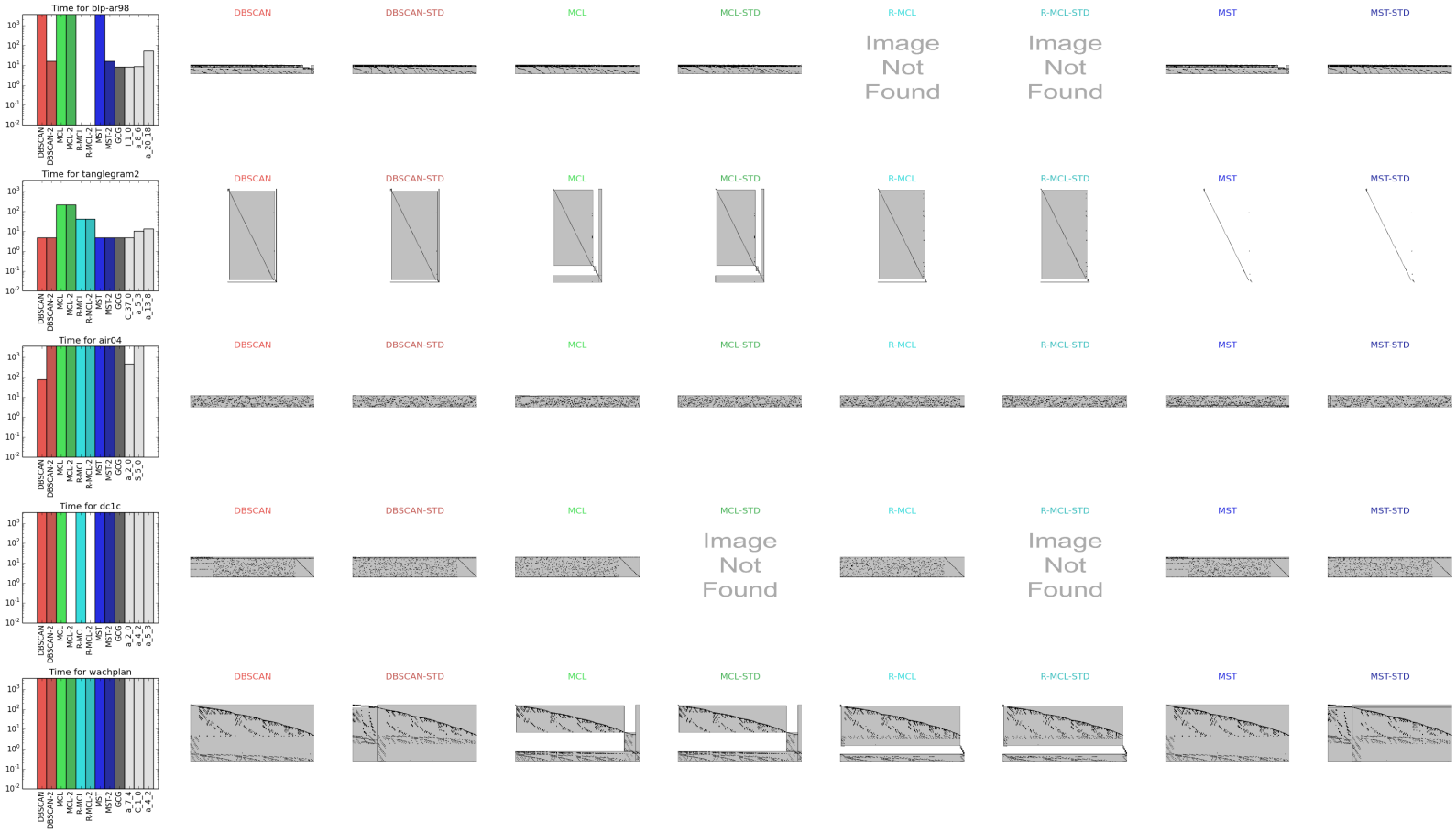


Figure .21: Test set 4, Johnson similarity

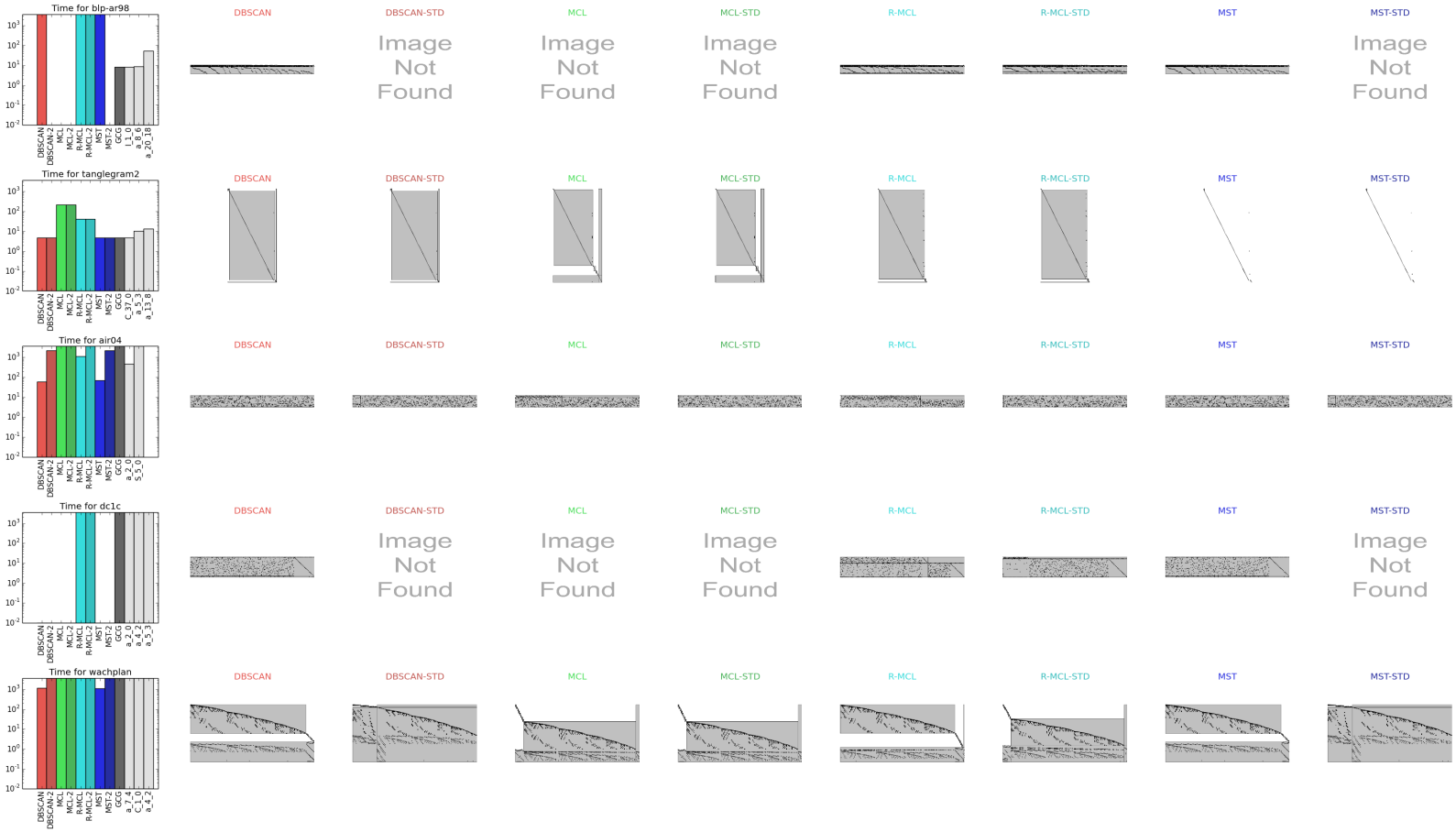


Figure .22: Test set 4, Intersection similarity

Figure .23: Test set 4, Jaccard similarity

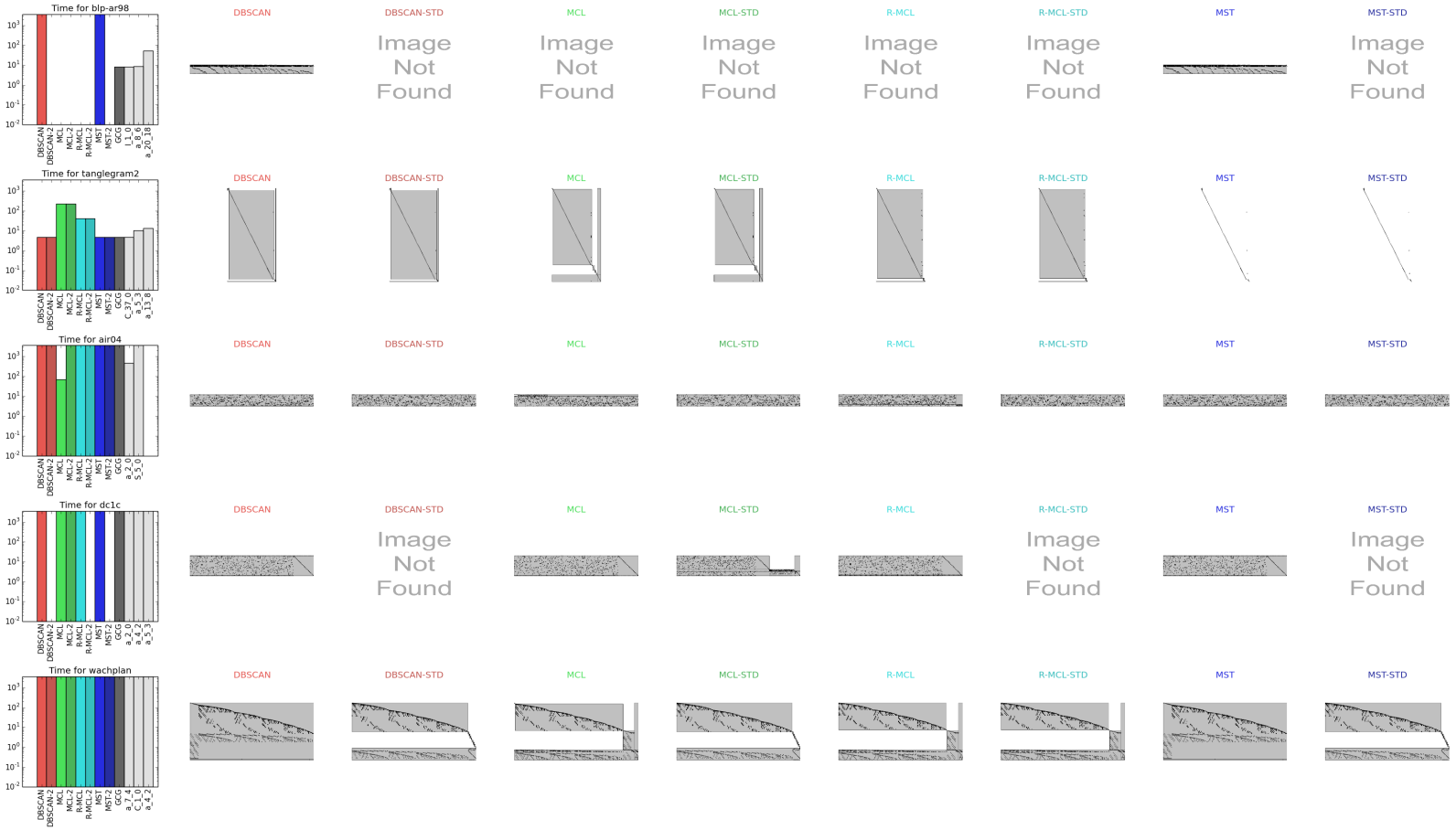


Figure .24: Test set 4, Cosine similarity

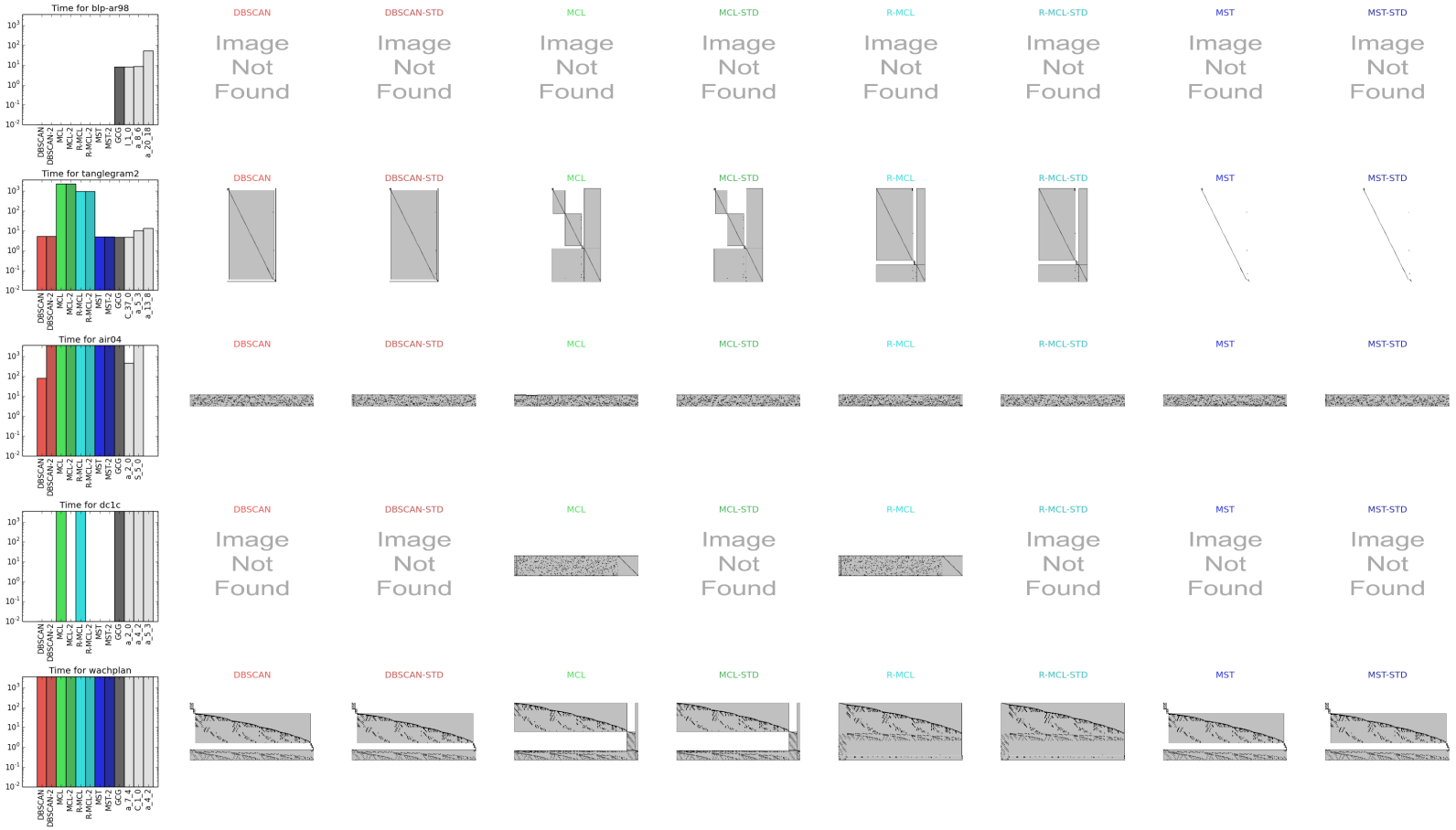
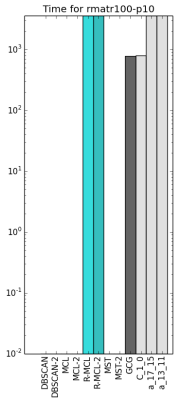
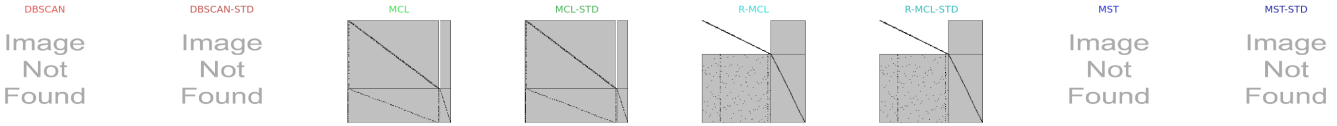
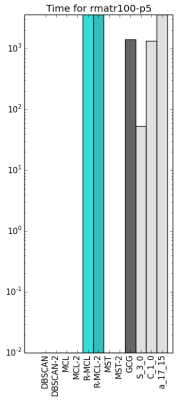


Figure .25: Test set 4, Simpson similarity







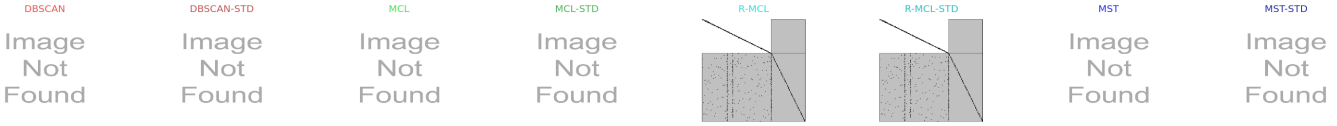


Figure .26: Test set 5, Johnson similarity

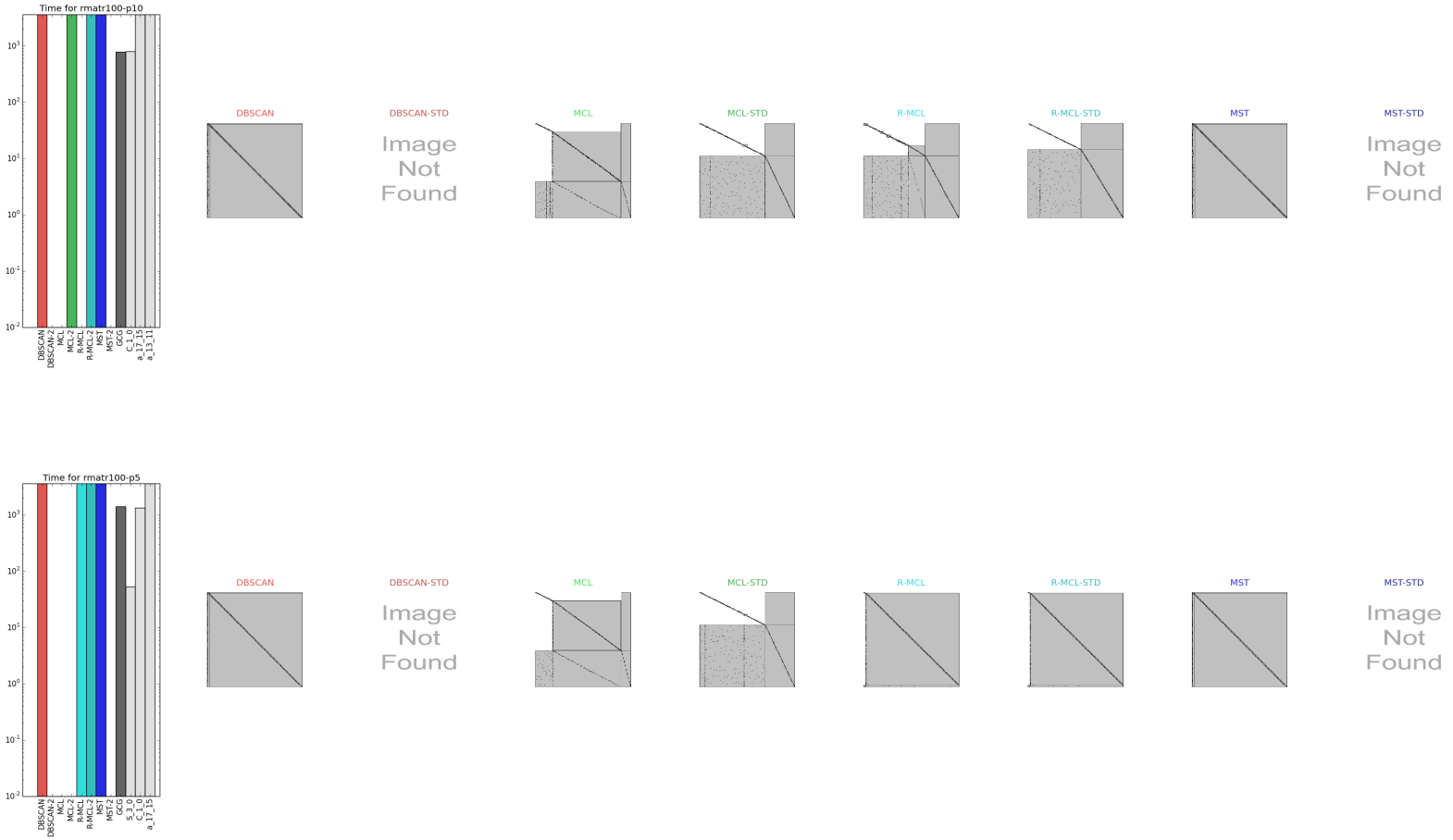


Figure .27: Test set 5, Intersection similarity

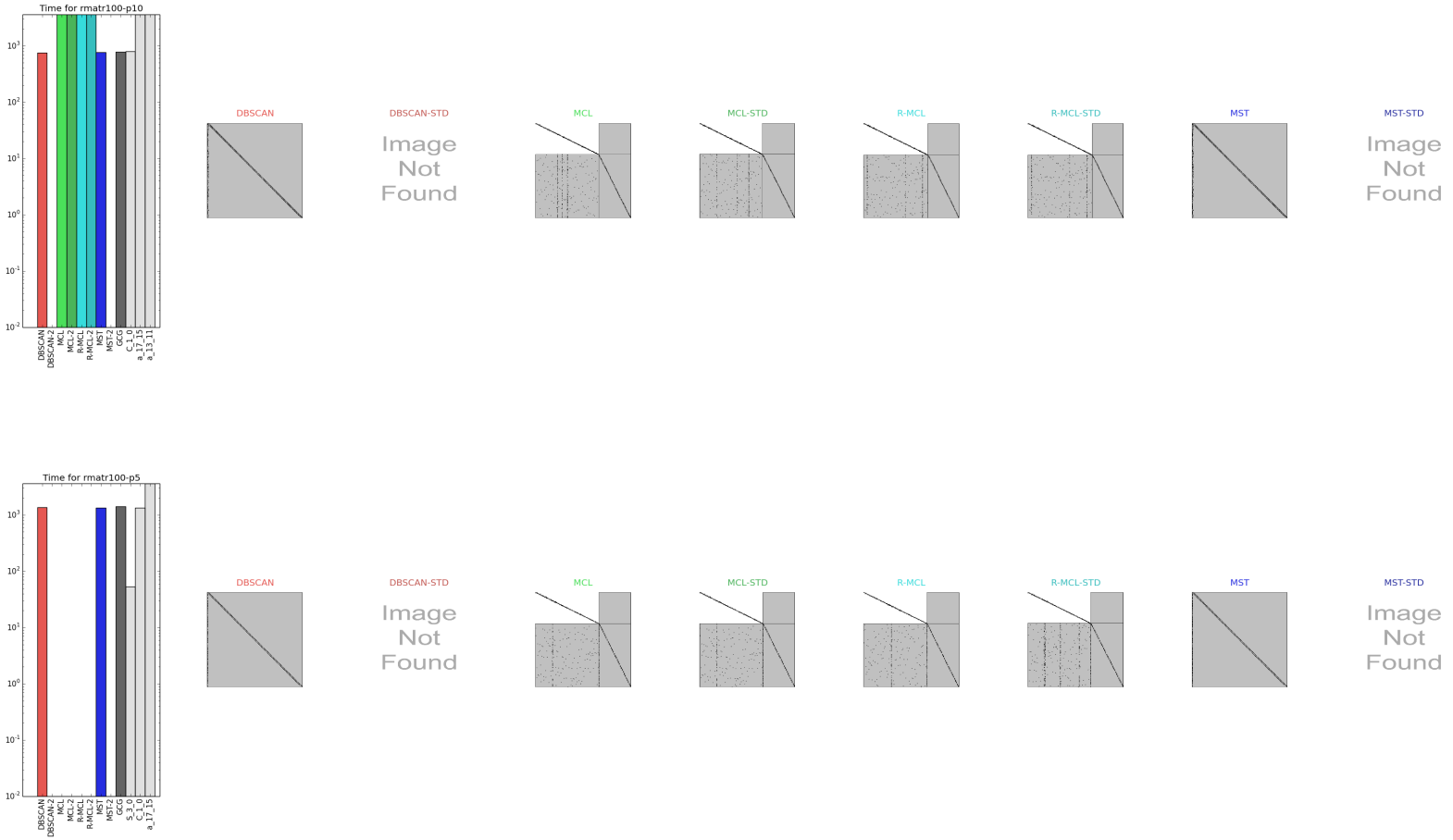


Figure .28: Test set 5, Jaccard similarity

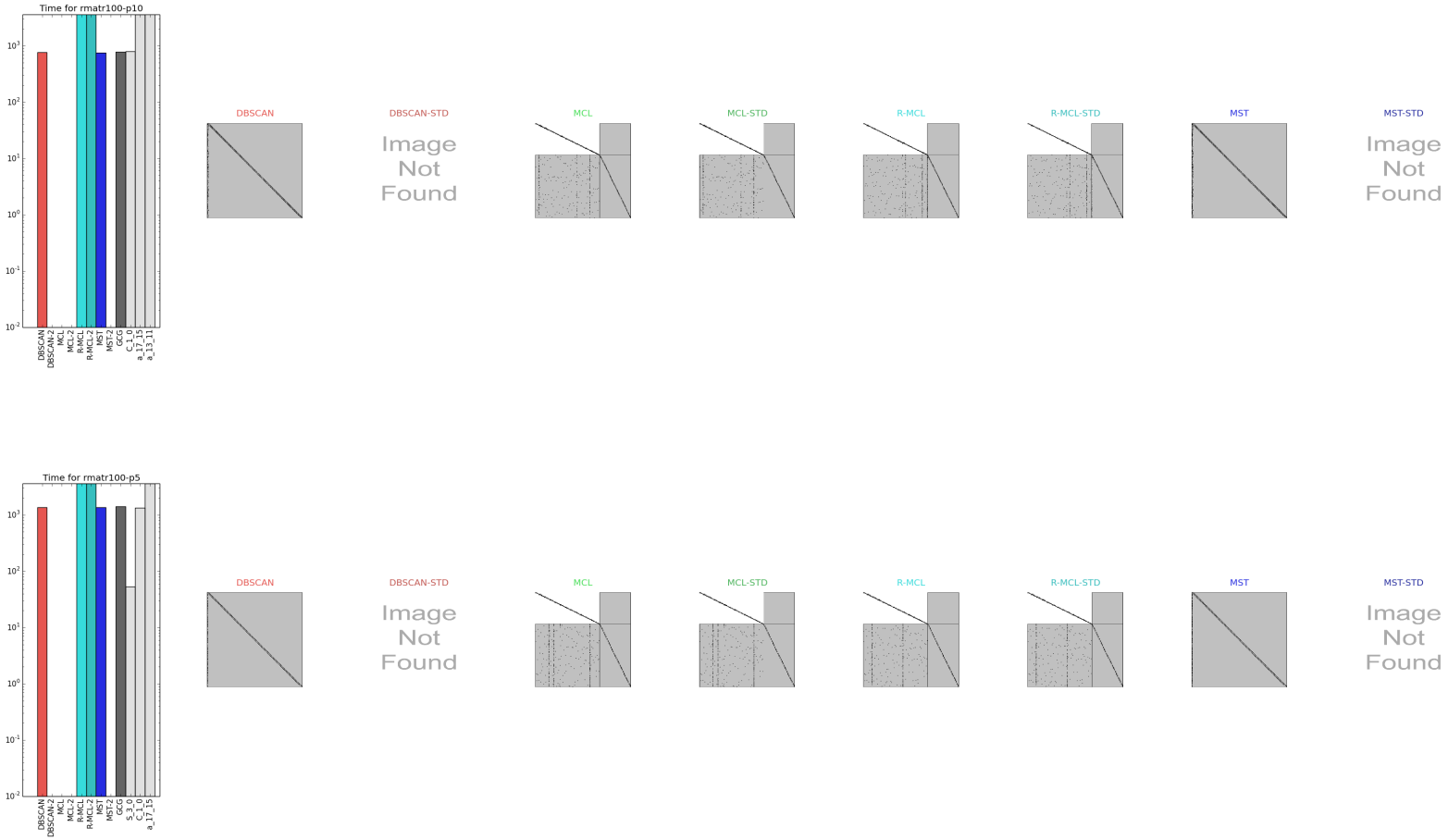


Figure .29: Test set 5, Cosine similarity

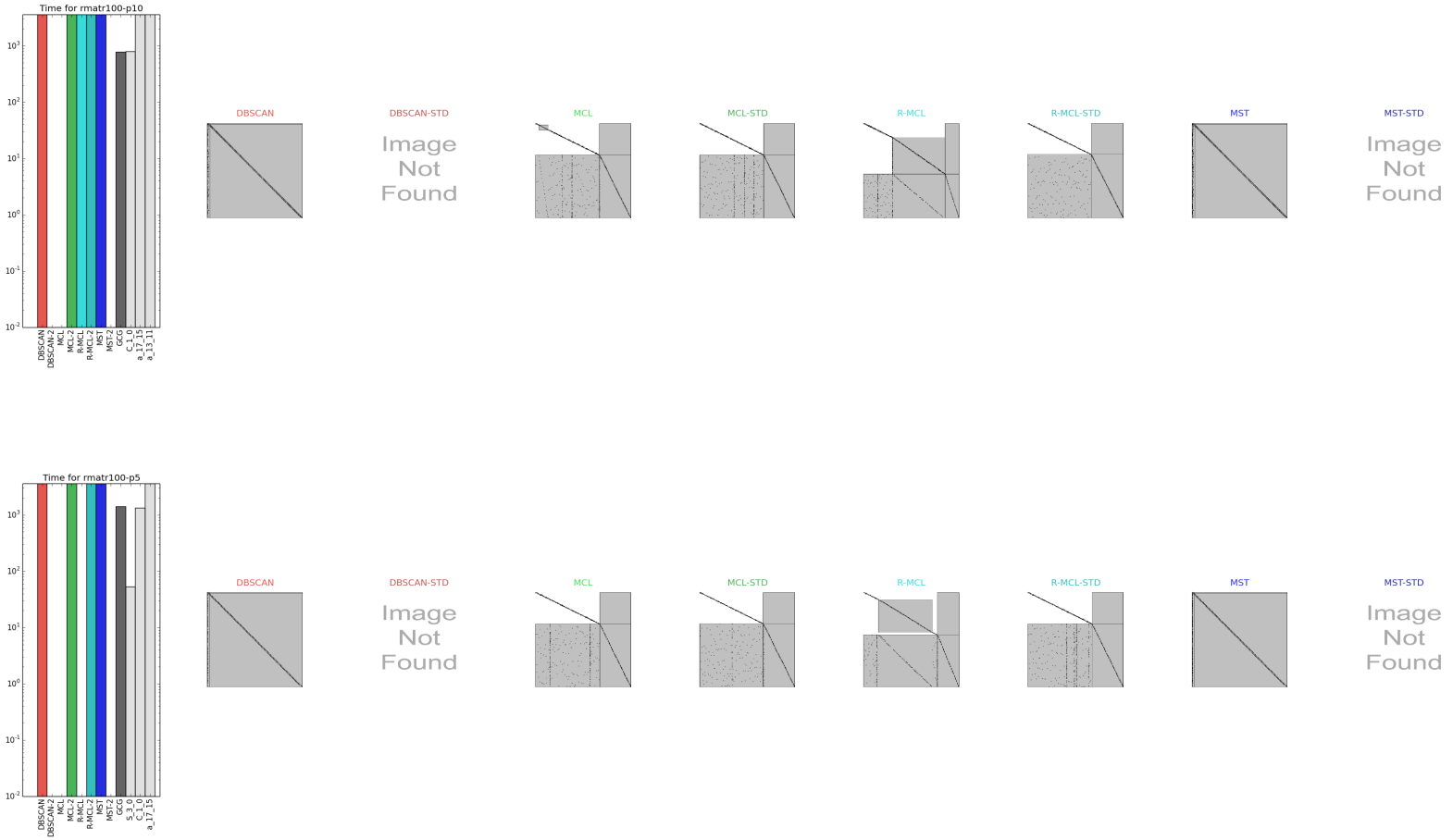


Figure .30: Test set 5, Simpson similarity