Journal of
the Brazilian Computer Society
a SpringerOpen Journal

RESEARCH Open Access

PQR sort: using PQR trees for binary matrix reorganization

Celmar Guimarães da Silva[1*], Marivaldo Felipe de Melo[2,3], Felipi de Paula e Silva[2,4] and João Meidanis[5]

Abstract

Background: Reorganization of rows and columns of a matrix does not modify data but may ease or impair visual analysis of data similarities in this structure, according to Gestalt spatial proximity laws. However, there are a factorial number of permutations of rows and columns. Matrix reordering algorithms, such as 2D sort and Sugiyama-based reordering, permute matrix rows and columns in order to highlight hidden patterns.

Methods: We present PQR sort, a matrix reordering algorithm based on a recent data structure called PQR tree, and compare it with the previous ones in terms of time complexity and quality of reordering, according to predefined evaluation criteria.

Results: We found that PQR sort is an interesting method for minimizing minimal span loss functions based on Jaccard or simple matching coefficients, specially for a given pattern called Rectnoise with a noise ratio of 0.01 or 0.02 and a matrix size of 100 × 100 or 1,000 × 1,000.

Conclusion: We concluded that "PQR sort" is a valid alternative method for matrix reordering, which may also be extended for other visual structures.

Keywords: Visualization; Seriation; Matrix reordering; Matrix visualization; Data visualization; PQR tree

Background

Defining spatial positioning of graphical elements is the most effective procedure for representing any kind of data [1]. This statement emphasizes the importance of using spatial axes for representing any type of variable, which may be classified in three distinct types [2]: nominal, ordinal and quantitative.

When one represents values of ordinal or quantitative variables in an axis, the predefined order usually yields optimal interpretation. Nominal variables, however, do not have a predefined order for displaying their values in spatial axes. One may populate an axis with a list of country names in any order, e. g., [Brazil, Argentina, Uruguay, Paraguay] or [Argentina, Brazil, Paraguay, Uruguay], without data gain or loss. Consequently, little (or none at all) attention is given to how easily users will interpret nominal data.

*Correspondence: celmar@ft.unicamp.br
[1]School of Technology, University of Campinas, Rua Paschoal Marmo, 1888, Jardim Nova Itália, Limeira, São Paulo 13484-332, Brazil
Full list of author information is available at the end of the article

However, Figure 1 shows how spatial distribution of axis values related to nominal variables may be relevant for highlighting a pattern present in data. In Figure 1a, an initial dataset is presented, organized by X-axis (columns) and Y-axis (rows); both axes are alphanumerically ordered. In Figure 1b, after some row permutations, a visual pattern begins to appear. Column permutations are then applied to the matrix in Figure 1b, generating the configuration presented in Figure 1c, which uncovers a visual data pattern whose identification was previously much harder to perceive.

A matrix to which row and column permutations may be applied is called 'reorderable matrix' by Bertin [3]. He highlights that permuting matrices' columns and rows enables people to discover overall relationships present in the data. Spence [4] adds that 'a mere rearrangement of how the data is displayed can lead to a surprising degree of additional insight into that data'.

Given the relevance of row and column reorganization, it is important to provide mechanisms for supporting it on visual structures. A possible approach for this is enabling users to reorganize rows and columns by themselves,

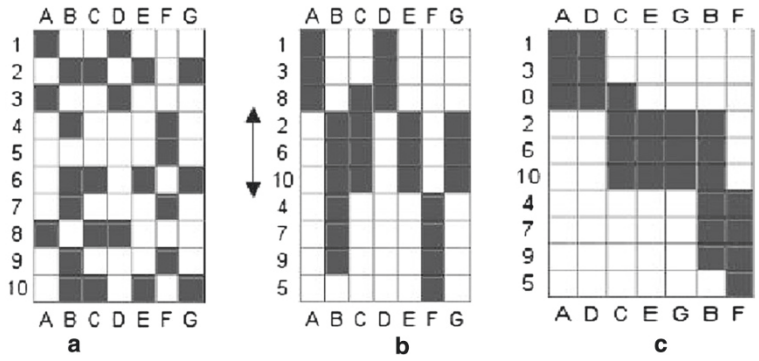

Figure 1 Reordering a binary matrix. (a) Original matrix. **(b)** The same matrix, vertically reordered. **(c)** The previous matrix, now horizontally reordered. (Adapted from [4] (p. 15)).

until they find a relevant pattern. However, the number of available permutations is of factorial order. Therefore, algorithms should be provided to visually reveal hidden visual patterns on the data, reduce cognitive effort for data understanding, and avoid manual reordering of data. Some of these algorithms are based on heuristic approaches, such as 2D sort and Sugiyama-based methods [5].

This paper presents an interesting alternative approach based on PQR tree, a data structure which represents dataset permutations that agree with a set of imposed permutation restrictions. The proposed approach uses this data structure for reordering rows and columns, according to their similarities. Also, a PQR tree may be created in an almost linear time, a fact that encouraged us to create this approach and to compare it to previous ones.

The paper is organized as follows. The 'Related research' subsection presents related research about matrix reordering methods. 'Theoretical background' subsection presents some concepts related to the problem, including Gestalt laws of pattern visual perception, the consecutive ones problem, and PQR trees. 'Methods' section suggests a way of reordering binary matrices using two PQR trees. The 'Results and discussion' section presents examples of matrices reordered by our approach and analyzes its performance through experiments and complexity analysis. Finally, the 'Conclusions' section concludes the paper and proposes ideas for future work.

Related research

Liiv [6] presents a historical overview of some matrix reordering methods. He also defines the concept of seriation as an 'exploratory data analysis technique to reorder objects into a sequence along a one-dimensional continuum so that it best reveals regularity and patterning among the whole series'. Therefore, a matrix reordering algorithm is directly related to this concept. Liiv [6] depicts the application of seriation and matrix reordering in disciplines such as archeology, cartography, graphics, information visualization, and bioinformatics.

Wu et al. [7] use the expression 'matrix visualization' to unify terms such as reorderable matrix, heatmap, and color histogram. They also present relevant concepts in this context, such as similarity (or proximity) matrices and measures. One may calculate similarity measures, such as Jaccard and simple matching coefficients, for each pair of rows or columns of a data matrix; after that, one may use these measures for constructing similarity matrices (for row and column similarities). The authors also refer to anti-Robinson and minimal span loss functions as available criteria for evaluating row and column permutations of these similarity matrices and, consequently, for evaluating the permutation of the data matrix itself. It is worth considering that there are other alternatives for evaluating the permutation of data matrices which are not based on similarity (or even dissimilarity) matrices; indeed, calculating measures of effectiveness [8] and stress [9] uses only the data matrix itself.

Mäkinen and Siirtola [5,10] propose the use of two methods for automatic matrix reordering: the 2D sort method and an adapted version of Sugiyama's algorithm. The latter was originally developed for edge crossing minimization in bipartite graphs, but Mäkinen and Siirtola applied it to matrices, considered as adjacency matrices of graphs. Both methods run over non-binary matrices, but they reduce the problem to binary matrices, defining that a cell is black (1) or white (0) if its value is lower than a defined threshold or greater than it. Both methods are based on heuristics which try to construct areas with black cells in the top-left and right-bottom parts of the matrix, and white cells elsewhere. These methods also only work well with relatively small matrices [10]. Based on some experiments, Mäkinen and Siirtola indicate that the heuristic nature of the Sugiyama's reordering algorithm produces unexpected results for some users, since

it generates distinct results for algorithm executions that have identical data and circumstances.

Guo and Gahegan [11] present other possible approaches for seriation. Among them, multidimensional scaling (MDS) and hierarchical clustering algorithms seem to be directly related to the reorganization problem addressed in this paper. MDS algorithms try to preserve the original distances between points in an n-dimensional space when these points are projected into a m-dimensional space, $m < n$. Setting $m = 1$ gives points in a one-dimensional space, and consequently, an ordered set of points. Hierarchical clustering algorithms also use similarity measures to define an ordered set of elements. Guo and Gahegan [11] present also a simple ordering algorithm based on clustering. Given a set of clusters (initially consisting of single elements), an interactive step groups two clusters, creating a bigger one. Given two clusters $a, ..., b$ and $x, ..., y$, it is necessary to choose one of four fusion possibilities: $a, ..., b, x, ..., y$; $a, ..., b, y, ..., x$; $b, ..., a, x, ..., y$; and $b, ..., a, y, ..., x$. The fusion with higher similarity between the elements at the clusters' endpoints is selected, and this choice defines element reordering. When all clusters are fused, the new order of the elements is defined.

Wilkinson [12] points out that not only MDS and hierarchical clustering may be used for matrix reordering, but also singular values decomposition (SVD), nonlinear dimension reduction, and entropy minimization. Besides, he defines five matrix types ('canonical data patterns') with quantitative cells, which may serve as a good initial set of matrices for testing permutation methods. Our 'Rectnoise' experiment to be presented in Section 'Experiments' is in some sense similar to Wilkinson's 'block' pattern.

Expanding the original problem, Qeli et al. [13] present a situation containing n matrices with the same dimensions. In this situation, only one matrix is visualized at any given moment. If each matrix is reordered according to its own data only, the cells in row 1 of matrix A may be, say, in row 7 of matrix B and in row 5 of matrix C. This would confuse users, which would see undesirable row and column exchanges. In this sense, Qeli et al. propose a heuristic algorithm for finding an optimal permutation of these rows and columns, in such a way that this permutation may be applied to all n matrices, avoiding row and column exchange. This algorithm chooses, among the n matrices, the one that is most similar to the other $n - 1$ matrices, according to a defined similarity measure.

Although the focus of this paper is the automatic reorganization of matrices, the importance of human-computer interaction must not be dismissed, given that users must have the final word about which ordering fits better some task. In this direction, different works, such as Siirtola and Mäkinen [10], present not just automatic reordering capabilities, but also interactive ones. This interaction may be done when user manually transposes matrix columns and rows, or when he asks software to sort individual columns and/or rows according to their values (which impacts the overall matrix disposition).

Theoretical background

This section presents the theoretical background of this work, composed by Gestalt laws, consecutive ones problem, and PQR trees. The proposed reordering is related to the Gestalt laws of pattern visual perception [14], and more specifically to the spatial proximity law. This law states that close elements are perceived as a group. The objective of the reorganization proposed by this paper is to enable users to understand a set of similar elements as a group at the visual structure level, which therefore may reduce the cognitive overload of the visual analysis.

In order to demonstrate this perception of grouped elements, Figure 2 highlights some cells of Figure 1 with gray circles. These circles make evident that rows 4, 7, and 9 in Figure 2a are similar, but these rows are not close to each other. Therefore, a quick look at Figure 1a would not reveal this similarity. However, if these rows are positioned consecutively, their similarity becomes evident, because the black cells of a given column are visually perceived as two vertical groups of cells, as presented in Figure 2b. A subsequent column reordering (Figure 2c) also places similar cells together, configuring bigger cell groups. The relationships highlighted in Figure 2c include not just the similarity among rows 4, 5, 7, and 9, but also the one between the two columns containing black cells.

Applying the proposed reordering is a task related to the so-called consecutive ones problem. A binary matrix has the consecutive ones property for its columns when there is a permutation of its rows that makes all 1-cells (cells that have value 1, or the black cells in the previous figures) consecutive in each column [15]. Discovering this permutation (if it exists) is the consecutive ones problem for columns. One may define a similar problem for rows. More formally, the consecutive ones problem may be stated as follows: Given a collection of m subsets $S_1, S_2, ..., S_m$, called here *restrictions*, of a set U, the consecutive ones problem consists in answering whether there is a valid permutation of the elements in U, that is, a permutation that keeps the elements of each S_i consecutive [16].

Meidanis et al. [15] proposed a solution for the consecutive ones problem by using so-called PQR trees to represent all valid permutations and also situations in which the consecutive ones property does not hold. The purpose of a PQR tree is to reduce substantially the factorial permutation possibilities of the elements of U to a smaller subset, according to restrictions imposed on these permutations in order to consider them valid. These restrictions are

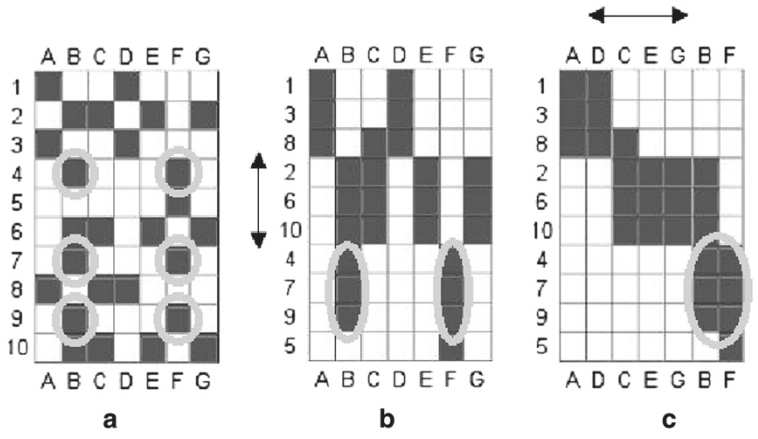

Figure 2 Matrices of Figure 1 with some highlighted cells. (a) These cells are not close to each other. **(b)** After row reordering, the similarity of rows 4, 7, and 9 becomes evident. **(c)** After column reordering, the similarity of columns B and F also becomes evident.

the subsets $S_1, S_2, ..., S_m$ from the consecutive ones problem, whose elements, if possible, are consecutive in the PQR-tree generated permutations.

Given a universe set U of elements to be permuted and a set of restrictions to be applied to U, a PQR tree is created to represent permutations of U. If the input has the consecutive ones property, then this tree represents only valid permutations. Otherwise, all the permutations represented by the resulting tree are non-valid, because conflicting restrictions cannot be fulfilled. However, in this case, the PQR tree can often pinpoint the elements and restrictions involved in the conflicts.

In a PQR tree, as defined by Meidanis and Munuera [17], each element of U becomes a leaf. Each non-leaf node defines that the leaves of all its descendants must be consecutive in the permutations represented by the tree. Also, the type of a non-leaf node defines how its children may be reordered among themselves. The children of a P-node may be arbitrarily reordered. The children of a Q-node, in turn, only support two permutations: the current left-to-right order in which they were drawn and the inverse order. R-nodes are similar to P-nodes because they also allow arbitrary permutations of their children; however, the children of an R-node are elements in which the consecutive ones property fails [16]. That is, R-node's children are elements whose restrictions forbid the existence of the consecutive ones property in U, because these restrictions are in conflict. So, a PQR tree containing R-nodes indicates that U does not have the consecutive ones property. Figure 3 summarizes the algorithm's inputs and outputs.

As an example, consider an instance of a consecutive ones problem in which $U = \{a, b, c, d, e, f, g\}$ and the restrictions are $S_1 = \{a, c, e\}$, $S_2 = \{b, f, g\}$, and $S_3 = \{a, c, d\}$; that is, the restriction set is $C = \{\{a, c, e\}, \{b, f, g\}, \{a, c, d\}\}$. A PQR tree that represents this restriction set is presented in Figure 4a. In Figures 4 and 5, a P-node is represented by a circle and a Q-node is represented by a rectangle. Therefore, this tree represents 48 (that is, 2! 2! 2! 3!) possible permutations of U, some of which are $\{d, a, c, e, g, f, b\}$, $\{d, c, a, e, b, g, f\}$, and $\{f, g, b, e, c, a, d\}$. If one permutes P-, Q-, and R-nodes' children appropriately (i.e., obeying the permutation properties of these nodes), one will obtain a list of all permutations represented by a PQR tree. This example tree has no R-nodes; therefore, any permutation represented by the tree obeys all given restrictions, and the instance has the consecutive ones property.

As a second example, consider the restriction set $C = \{\{a, c, e\}, \{b, f\}, \{b, g\}, \{f, g\}, \{a, c, d\}\}$. Figure 4b shows a PQR tree that represents these restrictions. It is very similar to the previous tree, but it has an R-node represented by a circle with an 'R' inside it. This means that nodes b, g, and f are related to conflicting restrictions - in this case, $\{b, f\}$, $\{b, g\}$, and $\{f, g\}$ - that cannot be satisfied simultaneously. Therefore, this instance does not have the

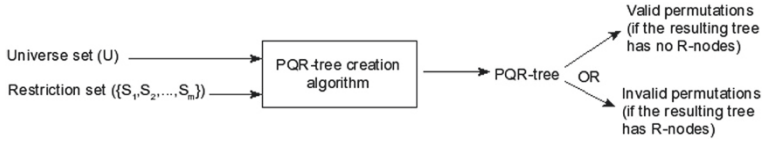

Figure 3 Schema of the PQR tree creation algorithm's inputs and outputs.

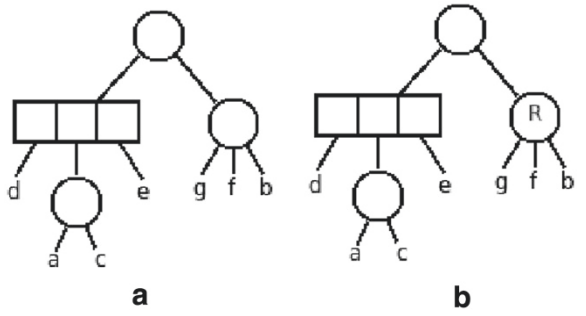

Figure 4 Examples of PQR trees. (a) Without R-nodes and **(b)** with one R-node.

consecutive ones property. Permutations represented by this tree will comply with all defined restrictions, except those identified as conflicting.

Another important concept related to PQR trees is *frontier*, which is a list of the leaves of a PQR tree, read from left to right. When the tree has no R-nodes, this list represents one of the valid permutations represented by the tree, and therefore, it is a possible and easy to obtain solution for the PQR tree-related problem. We do not give an in-depth explanation about the theory behind PQR trees. Interested readers should consult more specific works [15-17].

Currently, there are only two algorithms to create PQR trees [16,17]. The algorithm by Telles and Meidanis [16] is an almost linear time, incremental algorithm. It generates a PQR tree in time $O(\alpha(r)(n + m + r))$, where n is the number of elements in the universe set, m is the number of restrictions, r is a sum of the sizes of all restrictions, and α is a function that grows very slowly, known as the inverse of Ackermann's function. This algorithm starts with a PQR tree composed by a P-node containing all elements of U as its children. While the restriction set is not empty, its main loop removes a restriction from this set and applies it to the current tree, generating a new tree [16].

The order in which a given set of restrictions is inserted into a PQR tree and the order of the elements of U may generate distinct, but equivalent PQR trees. Therefore, their frontiers may be different, depending on these factors. However, two executions of the algorithm with exactly the same arguments will provide exactly the same result.

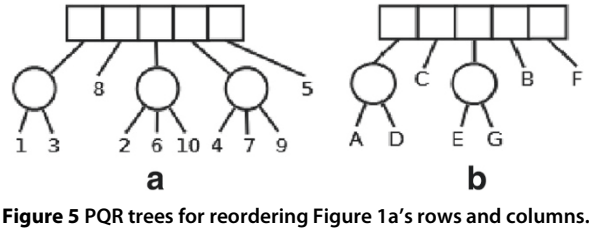

Figure 5 PQR trees for reordering Figure 1a's rows and columns.

Methods

This paper proposes a method for visually grouping similar columns and similar rows in a binary matrix, in order to provide a better visual perception of these groups. The proposed reordering has two independent phases: permuting columns (aiming to group columns with similar elements in each row) and permuting rows (aiming to group rows with similar elements in each column). Given that these phases are independent, they may be executed in this order or in the inverse one.

The problem takes the form of the consecutive ones problem, which in turn suggests PQR trees as an appropriate tool for its solution. Our method, called PQR sort, constructs two PQR trees: one for representing row permutations and another for column permutations. For each matrix column, the set of rows whose value is 1 in this column becomes a restriction to be inserted in the row-related tree. Analogously, for each matrix row, the set of columns whose value is 1 in this row becomes a restriction to be inserted in the column-related tree.

For example, consider the matrix in Figure 1a. Looking at column A of this matrix, one will group all rows that have value 1 in this column (rows 1, 3, and 8) and then create a restriction $\{1, 3, 8\}$. Using the same logic for the other columns, the following restrictions will be also created: $\{2, 4, 6, 7, 9, 10\}$, $\{2, 6, 8, 10\}$, $\{1, 3, 8\}$, $\{2, 6, 10\}$, $\{4, 5, 7, 9\}$, and $\{2, 6, 10\}$. When the same algorithm is executed to create restrictions for column permutations, the following restrictions are defined: $\{A, D\}$, $\{B, C, E, G\}$, $\{A, D\}$, $\{B, F\}$, $\{F\}$, $\{B, C, E, G\}$, $\{B, F\}$, $\{A, C, D\}$, $\{B, F\}$, and $\{B, C, E, G\}$.

Figure 5 depicts the PQR trees that summarize these restrictions. Observe that the frontiers of these trees represent the orders of Figure 1c's rows and columns: $\{1, 3, 8, 2, 6, 10, 4, 7, 9, 5\}$ and $\{A, D, C, E, G, B, F\}$, respectively. Next, we discuss how to choose a permutation among the found ones and whether the existence of R-nodes on PQR trees would affect this choice.

Figure 5 shows that no R-node was generated in either of the two trees. In such cases, the problem is solved by the adoption of one of the permutations represented by the PQR tree created. There is no reason to use any other permutation than the one available in the PQR tree frontier.

However, there may be conflicting restrictions, and therefore the R-nodes in the resulting PQR tree (as in the situation presented in Figure 4b). In these situations, it is necessary to decide how to order R-node's children. An R-node represents a set of rows (or columns) that are in some sense inter-related, with possible inner similarities. So, placing these rows (or columns) together may

be interesting, even knowing that some of the restrictions will not be obeyed. In this sense, the R-node has the role of spatially grouping these rows or columns. Therefore, our first approach to this problem is to keep R-node's children in the same order they are in the frontier.

It is possible that grouping R-node's children have a negative impact on the results. However, this impact should be minimal for R-nodes with few children. Two comments must be made before we pass to the test section. First, users do not need to inform PQR sort about the restrictions to be considered. Instead, the algorithm calculates them from the input matrix. Second, for two equal input sets, the algorithm creates two equal PQR trees, which represent the same set of permutations and whose frontiers have the same elements in the same order. Therefore, we may state that PQR sort also presents the same output for the same input. If distinct matrix organizations were provided to a user each time he/she runs the reordering algorithm (for example, due to algorithm random factors), users probably would have difficulties related to their mental models about the matrix data. Therefore, the algorithm's consistent behavior helps avoid this problem.

Results and discussion

This section compares the PQR sort to other approaches for matrix reordering. We present a complexity analysis of the algorithms and evaluate their results for reorganizing synthetic and real-world matrices.

Complexity analysis

The asymptotic time complexities of PQR sort, 2D sort, and Sugiyama-based algorithms in the case of $n \times n$ matrix reordering (square matrices were used for convenience only) are the following:

- *PQR sort*: For reordering binary matrices, PQR sort has two restriction defining steps, each using $O(n^2)$ time, and two PQR tree creation steps, each having time complexity $O(\alpha(r)(n+m+r))$ [16] (see section 'Theoretical background'). Given that $m = n$ and $r \leq n^2$ in our algorithm, the complete algorithm is in $O(n^2\alpha(n^2))$.
- *2D sort* [5]: Mäkinen and Siirtola did not analyze the time complexity of 2D sort. However, a simple analysis reveals that this algorithm is a conditional loop of two steps; each step calculates weighted row (or column) sums ($O(n^2)$) and then sort rows (or columns) according to these sums ($O(n \log n)$). This loop repeats until no row or column exchanges occur. Then, the time needed is in $O(n^2 t)$, where t represents the number of loop iterations.
- *Sugiyama-based (or barycenter heuristic-based) matrix reordering* [5]: The complexity analysis is similar to that of 2D sort. The difference is that the Sugiyama-based method calculates the average of 1-cells' positions in each row (or column) instead of calculating row (or column) sums. Therefore, the time complexity is $O(n^2 t)$.

According to these analyses, it is not clear which algorithm has the lowest complexity, because the number of iterations of barycenter heuristic and 2D sort conditional loop is hard to foresee.

Experiments

We designed two experiments for evaluating the performance of 2D sort, Sugiyama-based reordering (hereafter called Sugiyama) and PQR sort in terms of execution time and of 'quality' of the reordered matrices. These experiments were executed by a evaluation software that creates sets of binary matrices, reorganizes them using the three reordering algorithms and then evaluates the results according to predefined criteria, as depicted in Figure 6.

In the first experiment (which we called 'Block'), we analyzed the performance of the reordering algorithms when applied to matrices that have the Block pattern [12]. The following matrix exemplifies this pattern:

$$X = \begin{pmatrix} 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&1&1&1 \\ 0&0&0&0&0&0&1&1&1 \\ 0&0&0&1&1&1&0&0&0 \\ 0&0&0&1&1&1&0&0&0 \\ 0&0&0&1&1&1&1&1&1 \\ 0&0&0&1&1&1&1&1&1 \\ 1&1&1&0&0&0&0&0&0 \\ 1&1&1&0&0&0&0&0&0 \\ 1&1&1&0&0&0&1&1&1 \\ 1&1&1&0&0&0&1&1&1 \\ 1&1&1&1&1&1&0&0&0 \\ 1&1&1&1&1&1&0&0&0 \\ 1&1&1&1&1&1&1&1&1 \\ 1&1&1&1&1&1&1&1&1 \end{pmatrix}.$$

This example presents a 3-bit pattern [000, 001, 010, 011, 100, 101, 110, 111] expanded along matrix dimensions. Other values could be used for creating other k-bit patterns. Therefore, given a $n \times p$ data matrix X and an integer positive variable k, we may define $x_{i,j}$, $0 \leq i < n$, and $0 \leq j < p$ as follow. First, we define the dimensions of each block in the matrix as (*blockHeight*, *blockWidth*).

$$\text{blockHeight} = n/2^k$$

$$\text{blockWidth} = p/k$$

After that, we define the bit pattern of each row i as *rowPattern(i)*.

$$\text{rowPattern}(i) = \lfloor i/\text{blockHeight} \rfloor$$

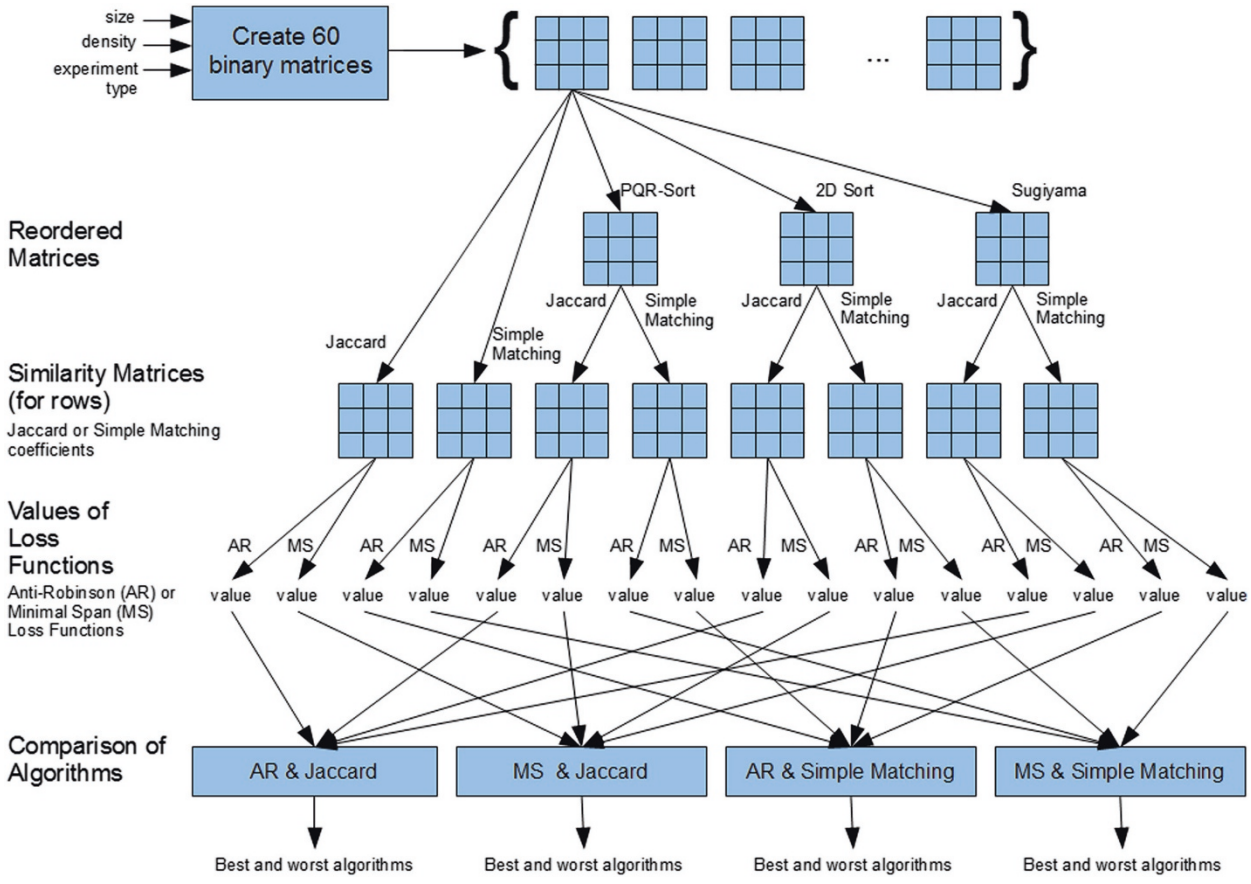

Figure 6 Steps for executing part of our experiments. The whole experiment will set 'experiment type' as 'Block' or 'Rectnoise' and will define some values for size and density.

The position of the bit of a row pattern that defines the value of a given column j of a row is given by *bitPosition(j)*.

$$\text{bitPosition}(j) = \lfloor j/\text{blockWidth} \rfloor.$$

The following expression defines the value of any $x_{i,j}$ cell that obeys the Block pattern. In this expression, '&' is the binary AND operator.

$$x_{i,j} = \begin{cases} 1 & \text{if } (\text{rowPattern}(i) \,\&\, 2^{\text{bitPosition}(j)}) > 0 \\ 0 & \text{otherwise} \end{cases}$$

In our experiment, we used $k = 3$. The independent variables of this experiment were matrix size and noise ratio (where noise is, in fact, the inversion of the values of some cells). Given a set of noise ratios $D = \{0.01, 0.02, 0.05, 0.10\}$ and a set of matrix sizes $S = \{100 \times 100, 1000 \times 1000\}$, we defined eight matrix types $\{d, s\}$, $d \in D$, and $s \in S$. For each matrix type, we created a set of 60 matrices to be reordered. The evaluation software fills each matrix according to the Block pattern and the predefined noise ratio. After that, the software shuffles its columns and rows. Our purpose here was to create matrices in which there are known similarities between some lines and some columns. Noise ratios were not set to values greater than 0.10 because we perceived empirically that greater ratios could destroy the original visual pattern even before any shuffling procedure.

Given these matrices, it was necessary to choose how to evaluate them. As a first approach, we choose to evaluate them based on similarity matrices. For each matrix, our evaluation software calculates column and row similarity matrices. These matrices depend on what similarity coefficient will be considered. Cox and Cox [18] and Wu et al. [7] present 17 similarity coefficients for binary data. Neither list have Euclidean distance and correlation coefficients; Wu et al. [7] argue that these coefficients cannot be applied directly to binary data sets. Cox and Cox [18] state that choosing a coefficient depends on the situation and that using more than one can help reach robustness against choice.

Wu et al. [7] classify binary variables according to symmetry: 'a variable is symmetric if both of its states are equally valuable', that is, given two possible states, neither is more relevant than the other. Otherwise, the variable is called asymmetric, and therefore, one of its

states is rarer and more important than the other. Moreover, they argue that asymmetric variables are usually sparse in nature. Wu et al. [7] also state that it is a common practice to use Jaccard coefficient for sparse data instead of simple matching coefficient. We conclude that one can use Jaccard coefficient for evaluating asymmetric variables and simple matching for evaluating symmetric ones. Therefore, we selected these two similarity coefficients for our experiments, in order to cover both types of variables. The evaluation software constructs then two sets of similarity matrices for each matrix, one for each of these coefficients. Each set has similarity matrices for columns and rows related to its coefficient.

In the next step, the evaluation software uses evaluation functions whose resulting values summarize the 'quality' of each similarity matrix (and, therefore, the 'quality' of their respective original binary matrices). Wu et al. ([7], pp. 688–689) describe two types of evaluation functions. Anti-Robinson loss functions are global criteria for quantifying how far a similarity matrix is from a Robinson matrix and therefore how 'smooth and pleasant' the visual effect of the original matrix is not. Minimal span loss function is a sum of the coefficients of neighbor columns (or rows); lower values of this function represent a good matrix permutation in terms of local structures. We choose to work with these two types of functions in our experiments, in order to cover both global and local structures. Given this, the evaluation software calculates the minimal span loss function and the anti-Robinson loss function $\mathrm{AR}(i)$ of the similarity matrices for evaluating the reordered matrix according to the selected coefficients.

The evaluation software outputs a set of files, where each file presents values related to a given matrix type. For each non-reordered matrix and for the reordered versions provided by the reordering algorithms, each file presents loss function values of each coefficient for row and column similarities. The file also contains mean and standard deviation for these values. We expected that the PQR sort could provide good permutations of matrices that have this pattern, given that creating blocks of cells with value 1 is one of the objectives of the consecutive ones problem, which is directly related to our solution.

The second experiment (which we called 'Rectnoise') was very similar to the first one, but with different input matrices. In this experiment, the evaluation software fills each matrix of size $w_{\mathrm{size}} \times h_{\mathrm{size}}$ with 10 rectangles filled with 1-cells. The rectangles' dimension $w_{\mathrm{rect}} \times h_{\mathrm{rect}}$ is random, but is limited in such a way that $1 \leq w_{\mathrm{rect}} \leq w_{\mathrm{size}}/4$ and $1 \leq h_{\mathrm{rect}} \leq h_{\mathrm{size}}/4$. Also, each rectangle starts at a random position (x, y), $0 \leq x < w_{\mathrm{size}} - w_{\mathrm{rect}}$ and $0 \leq y < h_{\mathrm{size}} - h_{\mathrm{rect}}$. After that, the software adds noise and shuffles its columns and rows. The software creates a set of 60 matrices to be reordered, according to the same set of noise ratios and matrix sizes of the first experiment. Due to the direct relationship between this experiment and consecutive ones problem, we also expected good performances of our method.

Results

In this section, we analyze the experiments' results provided by our evaluation software. These results came from a set of hypothesis tests, in which we tried to understand two ideas. First, we wanted to know if the three reordering algorithms have any positive effects (i.e., if they decrease the values of the loss functions). Second, we wanted to understand which reordering algorithm gives us the best values of the loss functions in a time that is appropriate for providing a fast feedback for users. We believe that these results are general enough to be extended to non-square matrices as well.

Given that we have 60 matrices and that we can only estimate the measures' variance, we will analyze the experiments' results based on dependent t tests for paired samples. Given two different reordering algorithms, A and B, the null hypothesis of each test is that the loss function values based on a given similarity coefficient are equal for their resulting matrices. This hypothesis is defined as $D = 0$, where $D = X - Y$, X and Y are the values of a loss function for algorithms A and B, respectively. The alternative hypothesis is $D \neq 0$, which implies that $X \neq Y$. In this last case, the better algorithm is defined by D's sign. Alternatively, X or Y may be the loss function value for the original matrix. Without loss of generality, we only analyzed the results of row similarity matrices instead of analyzing the two similarity matrices.

As part of the t test, we calculated the mean values of D for each experiment, matrix type (size and noise ratio), and possible pairs of A and B algorithms. Therefore, we defined the winner algorithm for each pair {A,B}, and based on these results, we defined which algorithm has the best (or the worst) values among the three algorithms, for each experiment and matrix type. Most of the tests have statistically significant results (significance level of 0.01).

Tables 1 and 2 summarize the results of our experiments and present the reordering algorithms that returned the best, medium, and worst values for each situation, in terms of quality and execution time. In some situations, there is a clear winner in both criteria, for example, the PQR sort provides the best values for the minimal span function with simple matching coefficient in 100×100 matrices of the Rectnoise experiment, in the shortest execution time. In other situations, there is a trade-off between quality and execution time.

Table 1 Summary of the best (●), medium (○) and worst (×) reordering algorithms for the first experiment (Block)

Experiment: block			Matrix sizes and noise ratios							
			100 × 100				1,000 × 1,000			
Evaluator	**Coefficient**	**Algorithm**	**0.01**	**0.02**	**0.05**	**0.1**	**0.01**	**0.02**	**0.05**	**0.1**
Anti-Robinson	Jaccard	2D sort	×	×	×	×	×	×	×	×
		Sugiyama	●	●	●	●	●	●	●	●
		PQR sort	○	○	○	×	○	○	○	○
	Simple matching	2D sort	●	●	●	●	●	●	●	●
		Sugiyama	×	○	○	○	○	○	○	○
		PQR sort	○	×	×	×	×	×	×	×
Minimal span	Jaccard	2D sort	○	●	●	●	●	●	●	●
		Sugiyama	×	×	×	×	×	×	×	×
		PQR sort	●	○	○	●	○	○	○	○
	Simple matching	2D sort	○	●	●	●	●	●	●	●
		Sugiyama	×	×	×	×	×	×	×	×
		PQR sort	●	○	○	○	○	○	○	○
Execution time		2D sort	●	●	●	●	●	○	○	○
		Sugiyama	○	○	×	●	×	×	×	×
		PQR sort	×	×	×	●	○	●	●	●

However, regarding execution time, we must consider that we want to provide responsive interaction, and therefore, we expect fast execution time (preferably in the order of 100 ms). Feedbacks in the order of 1 s are also acceptable, given that users' flow of thought stay uninterrupted and no extra feedback is necessary. Figure 7 presents mean execution times for the reordering algorithms in both experiments with both matrix sizes. Figure 7a,c shows that in most cases, the algorithms spent less than 10 ms for reordering 100 × 100 matrices. For reordering $1,000 \times 1,000$ matrices, Figure 7b,d indicates that 2D sort and PQR sort execution times are lower than 1 s

Table 2 Summary of the best (●), medium (○), and worst (×) reordering algorithms for the second experiment (Rectnoise)

Experiment: Rectnoise			Matrix sizes and noise ratios							
			100 × 100				1,000 × 1,000			
Evaluator	**Coefficient**	**Algorithm**	**0.01**	**0.02**	**0.05**	**0.1**	**0.01**	**0.02**	**0.05**	**0.1**
Anti-Robinson	Jaccard	2D sort	×	○	○	●	●	●	●	●
		Sugiyama	●	●	●	●	×	○	×	×
		PQR sort	×	×	×	×	○	×	○	○
	Simple matching	2D sort	●	●	●	●	○	●	○	○
		Sugiyama	×	○	○	●	●	●	●	●
		PQR sort	×	×	×	×	×	×	×	×
Minimal Span	Jaccard	2D sort	×	×	×	×	○	●	●	○
		Sugiyama	○	×	×	×	×	×	×	●
		PQR sort	●	●	●	●	●	●	×	×
	Simple matching	2D sort	○	○	○	●	○	●	●	●
		Sugiyama	×	×	×	×	×	×	×	●
		PQR sort	●	●	●	●	●	●	○	×
Execution time		2D sort	●	●	●	○	○	○	○	○
		Sugiyama	×	×	×	×	×	×	×	×
		PQR sort	●	●	●	●	●	●	●	●

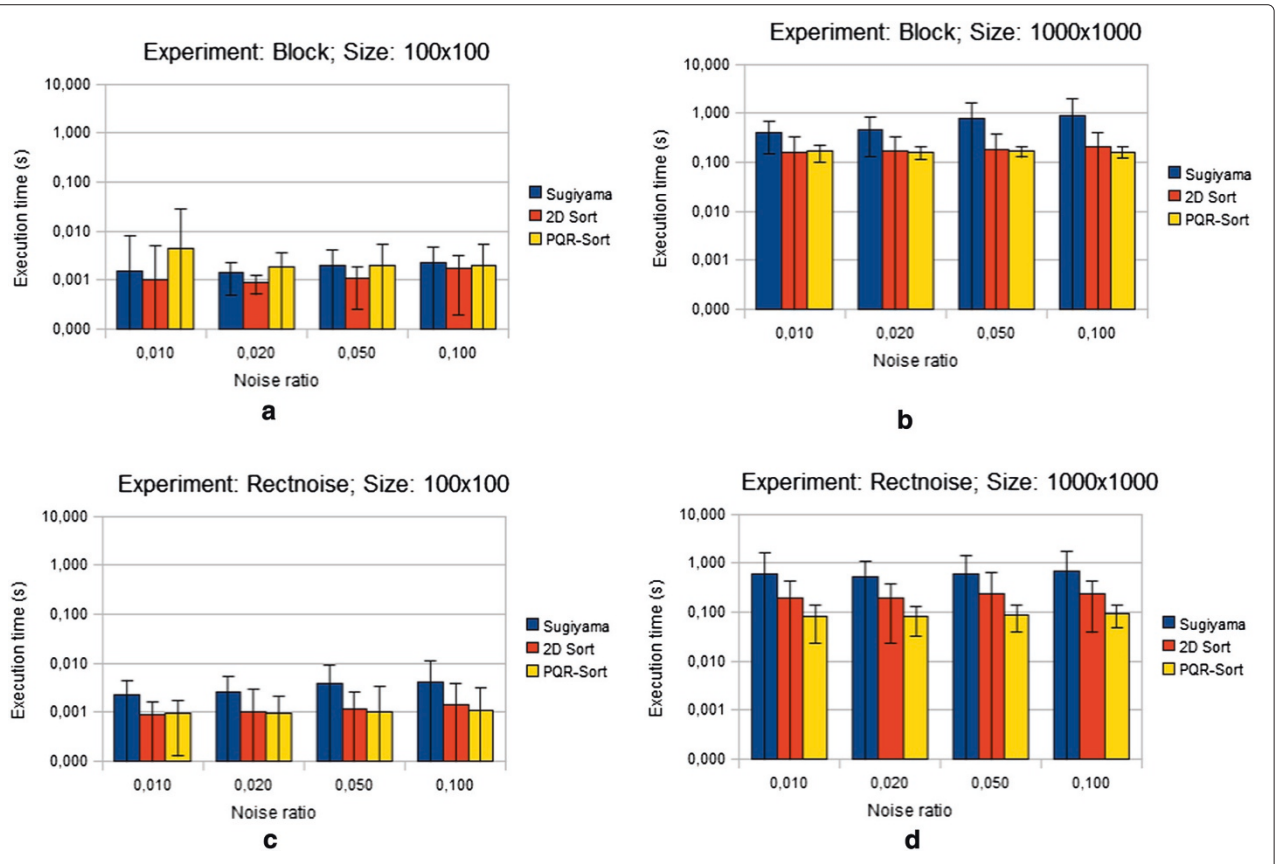

Figure 7 Execution time (in milliseconds) for Sugiyama, 2D sort, and PQR sort. (a) Block experiment, matrix size 100 × 100. **(b)** Block experiment, matrix size 1000 × 1000. **(c)** Rectnoise experiment, matrix size 100 × 100. **(d)** Rectnoise experiment, matrix size 1000 × 1000. Time axis in logarithmic scale. The error bars stand for a confidence interval of 99.7%.

and that Sugiyama may surpass this limit (indeed, it may reach approximately 3 s). Due to these conclusions, we opted for not considering time in our analysis, given that the time spent for reordering matrices with the tested characteristics would not impact user interaction in terms of feedback (except for the Sugiyama case).

Block experiment's results

We considered three situations in Block experiment's results:

- If the objective is to optimize the anti-Robinson function with Jaccard coefficient, Sugiyama returns the best reordering for both matrices.
- If we aim to optimize the anti-Robinson function with simple matching coefficient, 2D sort returns the best results for both matrices.
- If the objective is to optimize minimal span function (with simple matching or Jaccard coefficient), we identified three sub-cases:
  - For 100×100 matrices with noise ratio 0.1, the PQR sort returns the best results.
  - For the remaining 100×100 cases, the 2D sort returns the best results.
  - For $1,000 \times 1,000$ matrices, the 2D sort also wins.

Rectnoise experiment's results

We perceived three cases in the Rectnoise experiment's results:

- In order to optimize the anti-Robinson function with Jaccard coefficient:
  - For 100×100 matrices, Sugiyama returns the best results. One may prefer to run 2D sort for noise ratio 0.1 (similar result quality at possibly lower time than Sugiyama).
  - For $1,000 \times 1,000$ matrices, 2D sort produces the best results.
- If we desire to optimize the anti-Robinson function with simple matching coefficient, the choices are almost the inverse of the previous situation:

  - For 100×100 matrices, the 2D sort returns the best results (and in lower time than Sugiyama).
  - For $1,000 \times 1,000$ matrices, Sugiyama produces the best results. One may prefer to run 2D sort for noise ratio 0.02 (similar result quality at possibly lower time than Sugiyama).

- If we aim to optimize the minimal span function (with any of the studied coefficients):
  - PQR sort provides the fastest and best results for 100×100 matrices;
  - PQR sort also returns the fastest and best results for $1,000 \times 1,000$ matrices whose noise ratio is 0.01 or 0.02. For other ratios, 2D sort and Sugiyama provide the best results.

Summary of PQR sort contributions

The previous analysis indicates weaknesses and strengths of the three methods. We highlight the good performance of PQR sort for minimizing minimal span function for both the studied coefficients. In this sense, its best results were in 100×100 matrices; also, for $1,000 \times 1,000$ matrices, it returned the best results for noise ratios 0.01 and 0.02. This result points out that the PQR sort has a good potential to reveal local structures that the shuffling process hide from us. PQR sort weakness in matrices with greater noise ratios may be related to the insertion of big R-nodes in the PQR trees it uses, which would hamper the construction of useful trees for those situations.

On the other hand, due to its poor performance on minimizing anti-Robinson loss functions, PQR sort seems not to be useful for providing insights related to global structures hidden in shuffled matrices. For this functions, 2D sort and Sugiyama still provide the best results.

We also believe that the strategy for creating restrictions in our PQR trees was responsible for the failure of PQR sort in the Block experiment. Given that the step of creating restrictions does not take in consideration similarity measures between columns (or rows), again some R-nodes may mix blocks from the matrix which should be distinct. Therefore, future adaptations of PQR sort should consider similarity measures in order to improve the restriction set.

Examples

We also provide some examples of real-world binary matrices. The first example is a symmetric matrix of character coappearance in Victor Hugo's novel *Les Misérables*. These coappearance data were first compiled by Knuth [19] and further transformed into a CSV file [20], from which we constructed our adjacency matrix.

Figure 8 has a set of matrices representing this adjacency matrix. In these matrices, blocks of 1-cells (black cells) represent sets of characters that interact each other, and continuous or almost continuous rows (and respective columns) of 1-cells may indicate main characters. Figure 8a presents the original adjacency matrix of the novel, in the order of characters provided by the CSV file. We may see five blocks and two ticked rows (and respective columns), where the biggest row refers to the main character, and the other row seems to be an important character that interacts with 1/4 of his colleagues. 2D sort (Figure 8b) produces a reordered matrix in which only two clear blocks of characters may be perceived, and a third and very incomplete block at the bottom right corner. The last row refers to the main character. Sugiyama (Figure 8c) algorithm permutation presents three blocks and an almost central 'cloud' of sparse 1-cells, which is difficult to analyze; a 'cross' indicates the main character. The matrix reorganized by PQR sort (Figure 8d) is the only one that has a continuous row (and column) representing the main character, which seems to interact with approximately 50% of the novel characters (the other matrices may not help conclude this fact). Also, it is possible to see five blocks of characters along the main diagonal, as in the original matrix.

Even though there is not a clear winner solution in this example, we believe that the original matrix and the reordered matrix produced by PQR sort revealed more useful information than the other two matrices. The second real example is based on the online dictionary of library and information science (ODLIS) [21], from which a graph of terms was extracted [22]. It consists on a directed network with 2909 vertexes and 18,419 arcs, which we transformed into a $2,909 \times 2,909$ adjacency matrix. Each vertex represents a term in the dictionary; there is an edge (r,s) if and only if the dictionary uses the term related to s in order to explain the term related to r.

Figure 9a presents the original ODLIS adjacency matrix (rows for edge source and columns for edge destination). 2D sort (Figure 9b) gradually increased the density of 1-cells near to the right border; it also created two curved lines, which seem to be meaningless in this example. The matrix reordered by Sugiyama (Figure 9c) seems to reveal three groups of 1-cells, which in this case, may be related to three distinct subjects. In a more detailed analysis, if we consider this matrix as a torus, left border group would be the continuation of the right border one, and therefore, the number of groups is reduced to two. PQR sort (Figure 9d) created a matrix with a single dense region of dots whose width is approximately 1/8 of the matrix width and whose height is similar to the matrix one. The format of this region points out that the majority of the ODLIS' terms may be explained using almost 1/8 of the terms used to describe items in the dictionary.

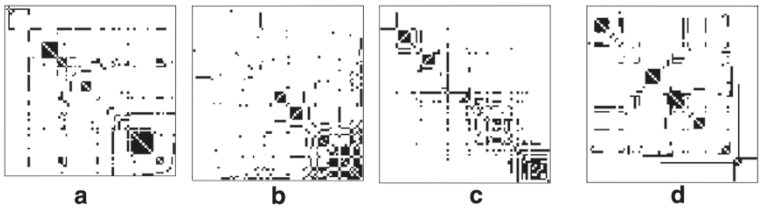

Figure 8 Adjacency matrix of character coappearance in *Les Misérables*. (a) Original adjacency matrix of character coappearance, with 77 × 77 cells. **(b-d)** The same matrix after being reordered by 2D sort, Sugiyama, and PQR sort. Characters' names were omitted.

In this example, Sugiyama presented an interesting global pattern hidden in the data, while PQR sort highlighted local structures. They seem to be more insightful than the result of 2D sort and the original matrix.

Conclusions

This work presented PQR sort as an alternative approach for matrix reordering, based on the use of PQR trees. Our experiments were executed in matrices of size 100×100 and $1,000 \times 1,000$, and revealed the usefulness of PQR sort for some situations. We highlight that PQR sort provides good results if we want to minimize the minimal span loss function (and, therefore, to reveal local structures) calculated over similarity matrices whose coefficient is Jaccard or simple matching, specially for the Rectnoise pattern with noise ratio 0.01 or 0.02, as summarized in Subsection 'Summary of PQR sort contributions'. In most situations, time performance was not relevant, given that execution time of the compared algorithms was lower than 1 s, which may be considered a fast feedback; in particular, PQR sort mean execution time was about 0.1 s. Besides, real-world examples pointed out how the permutations provided by PQR sort may help observe hidden visual patterns on the data, even for medium-size matrices (about $3,000 \times 3,000$).

Some future directions are under consideration for this work:

- *Improving PQR tree restrictions*: We believe that it is possible to improve the definition of PQR tree restrictions in our method, in order to better incorporate the concept of similarity among rows (or columns). Also, ongoing work points out that a correct ordering of restrictions at the creation of PQR trees may provide better results.
- *Reordering algorithms and metrics*: Many algorithms can be applied to matrix reordering. In this paper, we compared our method to only two previous ones. In addition, there are many metrics for assessing the quality of a reordering. Comparison of other reordering algorithms based on a broad set of metrics is under execution.
- *Matrix data*: In this paper, we manipulated only binary matrices. However, we believe that PQR sort may be extended for dealing with quantitative and nominal cells.
- *Visual structures*: Last but not least, it is worth to consider that PQR tree-based reorganization may improve other visual structures and techniques, such as 3D matrices, parallel coordinates, and even

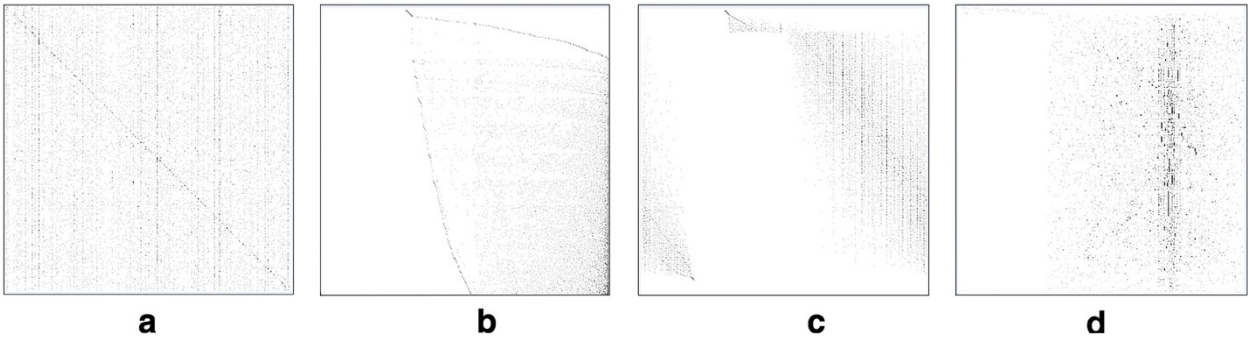

Figure 9 ODLIS adjacency matrix. (a) Original adjacency matrix with $2,909 \times 2,909$ terms. Row terms are explained by column terms according to the adjacency. **(b-d)** The same matrix after being reordered by 2D sort, Sugiyama, and PQR sort. Original images have one pixel per cell; they were resized and sharpened for this paper.

high-dimensional approaches. Indeed, a recent result published by our research group points out success on hybridizing PQR sort and barycentric heuristic algorithms for reordering directed acyclic graphs [23].

Competing interests
The authors declare that they have no competing interests.

Authors' contributions
CG da Silva executed and analyzed all experiments and wrote the manuscript. MF de Melo, FP e Silva and CG da Silva developed the evaluation software cited in Section 'Results and discussion'. J Meidanis provided all knowledge and guidance about PQR trees. All authors read and approved the final manuscript.

Acknowledgements
This research was financially supported by FAEPEX/PRP/Unicamp and CNPq. We also want to thank Prof. Maria Cristina Ferreira de Oliveira and Prof. Rosane Minghim by their useful advices, and Prof. Luis Augusto Angelotti Meira by helping us find interesting examples of binary matrices.

Author details
[1]School of Technology, University of Campinas, Rua Paschoal Marmo, 1888, Jardim Nova Itália, Limeira, São Paulo 13484-332, Brazil. [2]School of Technology, University of Campinas, Rua Paschoal Marmo, 1888, Jardim Nova Itália, Limeira, São Paulo 85070-200, Brazil. [3]Present address: Prefeitura Municipal de Limeira, São Paulo 13481-900, Brazil. [4]Present address: TOTVS S/A, Av. Braz Leme 1717, Casa Verde, São Paulo 02511-000, Brazil. [5]Institute of Computing, University of Campinas, Campinas, São Paulo 13083-852, Brazil.

Received: 17 April 2013 Accepted: 7 November 2013
Published: 23 January 2014

References

1. Mackinlay J (1986) Automating the design of graphical presentations of relational information. ACM Trans Graph 5(2): 110–141
2. Card S, Mackinlay J, Shneiderman B (1999) Readings in information visualization: using vision to think. Morgan Kaufmann Publishers, San Francisco
3. Bertin J (1981) Graphics and graphic information processing. Walter De Gruyter, Berlin
4. Spence R (2001) Information visualization. Addison-Wesley, Boston
5. Mäkinen E, Siirtola H (2000) Reordering the reorderable matrix as an algorithmic problem. In: Anderson M, Cheng P, Haarslev V (eds) Theory and Application of Diagrams. Springer, Berlin, pp 453–468
6. Liiv I (2010) Seriation and matrix reordering methods: an historical overview. Stat Anal Data Mining 3(2): 70–91
7. Wu HM, Tzeng S, Chen Ch (2008) Matrix Visualization. In: Chen C, Härdle W, Unwin A (eds) Handbook of data visualization. Springer, Berlin, pp 681–708
8. McCormick WT, White TW (1972) Problem decomposition and data reorganization by a clustering technique. Oper Res 20: 993–1009
9. Niermann S (2005) Optimizing the ordering of tables with evolutionary computation. Am Stat 59: 41–46
10. Siirtola H, Mäkinen E (2005) Constructing and reconstructing the reorderable matrix. Inf Vis 4(1): 32–48
11. Guo D, Gahegan M (2006) Spatial ordering and encoding for geographic data mining and visualization. J Intell Inf Syst 27: 243–266
12. Wilkinson L (2005) The grammar of graphics. Springer, New York
13. Qeli E, Wiechert W, Freisleben B (2004) Visualizing time-varying matrices using multidimensional scaling and reorderable matrices In: Proceedings of the Eighth International Conference on Information Visualization, London, UK, 14–16 July 2004, pp 561–567
14. Ware C (2004) Information visualization: perception for design, 2nd edition. Morgan Kaufmann, San Francisco
15. Meidanis J, Porto O, Telles GP (1998) On the consecutive ones property. Discrete Appl Math 88(1–3): 325–354
16. Telles GP, Meidanis J (2005) Building PQR trees in almost-linear time. Electron Notes Discrete Math 19: 33–39
17. Meidanis J, Munuera E (1996) A theory for the consecutive ones property. In: Ziviani N, Baeza-Yates R, Guimarães K (eds) Proceedings of third South American workshop on string processing, vol 4. Carleton University Press, Ottawa, pp 194–202
18. Cox MAA, Cox TF (2008) Multidimensional scaling. In: Chen C, Härdle W, Unwin A (eds) Handbook of data visualization. Springer, Berlin, pp 315–347
19. Knuth DE (1993) The Stanford GraphBase: a platform for combinatorial computing. http://www-cs-faculty.stanford.edu/~uno/sgb.html. Accessed 12 Sept 2013
20. GitHub Inc (2013) Adaptation of Mike Bostock's force-directed graph of Les Mis characters using .csv dataset instead of json. https://gist.github.com/timelyportfolio/5049980/. Github repository. Accessed 12 Sept 2013
21. Reitz JM (2013) ODLIS—online dictionary for library and information science. http://www.abc-clio.com/ODLIS/odlis_A.aspx. Accessed 12 Sept 2013
22. Batagelj V, Mrvar A (2006) Pajek datasets. http://vlado.fmf.uni-lj.si/pub/networks/data/. Accessed 12 Sept 2013
23. Marchete Filho JR, Silva CG (2013) Using PQR-trees for reducing edge crossings in layered directed acyclic graphs. In: Frery AC, Musse S (eds) Workshop of works in progress (WIP) in SIBGRAPI 2013 (XXVI Conference on Graphics, Patterns and Images). Universidad Católica de San Pablo, Arequipa

doi:10.1186/1678-4804-20-3
Cite this article as: Silva *et al.*: PQR sort: using PQR trees for binary matrix reorganization. *Journal of the Brazilian Computer Society* 2014 **20**:3.