UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Faithful visualization of categorical data

Dissertation presented by
Thomas BOLLEN , Guillaume LEURQUIN

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Siegfried NIJSSEN

Reader(s)
Pierre SCHAUS, Vincent BRANDERS

Academic year 2016-2017

Abstract

Data is today collected in very large amounts from various kind of sources: shopping, genetic tests, etc. Given the amount of data collected, one might be interested in visualizing that data in a meaningful way.

Due to their size, producing an image of such datasets is not easy as the image has to fit on the screen. Compression of the image is needed in order to have fewer elements to display. However, the quality of this compression greatly depends on the structure of the data.

Given a binary matrix represented by white and black pixels, well separated groups of pixels (e.g. black and white points are respectively gathered together) gives a lower loss of information because summarizing that data concisely is easier. Moreover, reorganizing the rows and columns of a dataset can give interesting insights in structures hidden within the data.

In this thesis, we introduce an innovative algorithm to reorder binary and categorical matrices using convolution. We then use the results of the algorithm to display the datasets faithfully.

Table of contents

List of figures

List of tables

Nomenclature

Subscripts

c	Column index (0 indexed)
k_c	Column index of a kernel (0 indexed)
k_r	Row index of a kernel (0 indexed)
r	Row index (0 indexed)

Other Symbols

K	A kernel
k_m	Number of rows of a kernel
k_n	Number of columns of a kernel
M	A matrix
m	Number of rows of a matrix
$M_{r,c}$	Value at row r and column c of the matrix M
n	Number of columns of a matrix

Acronyms / Abbreviations

CGBS	Cross Gaussian Blur Smooth Kernel
GBE	Gaussian Blur Exponential Kernel
GBS	Gaussian Blur Smooth Kernel
LS	Local Search
NP	Non deterministic Polynomial time
P-LS	Parallelized Local Search
TSP	Traveling Salesman Problem

Chapter 1

Problem Statement

1.1 Introduction

Data is today collected in very large amounts from various kind of sources: shopping, genetic tests, etc. Given the amount of data collected, one might be interested in visualizing that data in a meaningful way. Given a matrix of binary data (black or white), reorganizing the rows and columns can give interesting insights in structures hidden within that data. Some researchers [14] already attempted to reorganize small sets of binary data in order to get useful information from the patterns contained within that data.

The goal of this thesis is to improve or create new algorithms to visualize larger datasets. The size of those datasets could be a few thousands rows/-columns. Moreover, the number of rows and columns might be very different (e.g. a thousand rows and only one hundred columns). Due to their size, producing an image of such datasets is not easy as the image has to fit on the screen. Compression of the image is needed in order to have fewer elements to display. However, the quality of this compression greatly depends on the structure of the data. Indeed, if the binary data is well separated (e.g. black and white points are respectively gathered together), the loss of information is smaller because summarizing that data concisely is easier. This gives a good incentive to develop algorithms that can find a good permutation of the rows and columns so that the data can be represented more easily.

Once the matrix has been reordered, looking for geometric forms inside the data that we will call patterns becomes easier (cfr. figure 1.1). Indeed, if the black points are grouped together, those forms are more likely to appear as

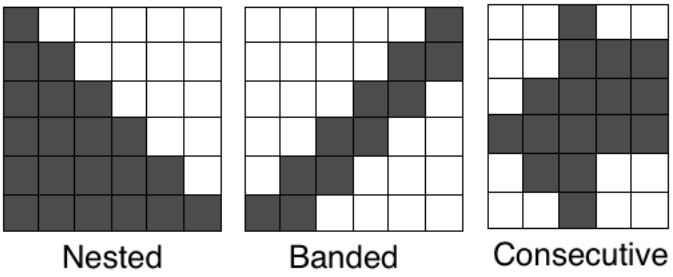


Fig. 1.1 Types of patterns

the data will look less random. Finding patterns is useful because they give information about the data. As an example, a nested matrix gives a hierarchy among the represented data.

The main goals of this thesis are to develop and assess such algorithms. The representation that we will obtain should also convey useful information and be interpretable by the user.

1.2 Challenges

In order to reorganize big datasets to form interesting patterns, multiple challenges will have to be solved.

Noise Most big datasets are not perfect, and incorporate some degree of noise in them. The developed algorithms should be able to deal with that fact.

Size The curse of dimensionality will also have to be taken into account. The developed algorithms must be as efficient as possible. On big datasets (a few thousands rows/columns), a trade-off between precision and computing time might be required.

Another issue is that most of the time the matrix does not have the same number of rows and columns (the matrix is not squared). If the number of rows (resp. columns) is much larger than the number of columns (resp. rows), the developed methods must take that into account.

Categorical data Not all datasets are binary, some are categorical. The algorithms should be able to deal with that kind of data as well.

Problems of the existing methods One of the limitations of the existing methods is that they do not allow the user to deal efficiently with big datasets. They focus more on finding patterns than on making the user able to visualize large matrices without losing too much information. This thesis will mainly focus on finding a representation of the data that can be shown. Hopefully that representation will also allow us to discover some hidden patterns inside the data and to interpret what we see. Another issue is that they only work for binary data, however these only represent a fraction of the available data. Finding a method that could be generalized to categorical data would allow us to tackle many more problems.

Density of the dataset In real life, some datasets are very sparse. In each row, there will only be a few 1's for many 0's (for example in a dataset representing customers who buy a few items in a shop that sells many things). Those datasets are more difficult to represent without losing too much information. Indeed, to be able to show all that data on a screen we will have to use some techniques to aggregate elements of the original matrix to provide a summary. If the number of 0's is much greater than the number of 1's, an aggregated matrix that is only composed of 0's is not useful.

We will limit ourselves to matrices that are not too sparse. If only some parts of the matrix are too sparse, we could cut the matrix into pieces.

1.3 Structure of the thesis

First, we present what has already be done in the literature in chapter 2. We then describe in chapter 3 more rigorously the problem we are interested in, that we call `ConvoMap`. We also define concepts that will be used later. Following that, we discuss the complexity of the search problem in chapter 4. First we talk about the size of the search space and the problems that come from that. Then we prove that `ConvoMap` is NP-Hard for a certain type of kernel. This is done by reducing the Hamiltonian Path problem to `ConvoMap`. Chapter 5 explains in detail the local search algorithm that we have implemented to solve the problem. We also discuss a number of difficulties that we encountered and their solutions. After that, we assess the performances of the algorithm on different datasets and we discuss the results in chapter 6. Finally, we conclude and talk about future works in chapter 7.

Chapter 2

Related work

2.1 Reordering binary matrices

The problem of finding permutations of the rows and columns of a binary matrix such that the reordered matrix brings out patterns has already been studied in the literature and is known as the *data seriation* problem. A historical overview of seriation techniques can be found in the work by Liiv [17]. The methods that aim to solve this problem can be distinguished into two classes, the ones that aim to order similar columns and rows together (*distance-base methods*) and the ones that aim to order rows and columns such that the resulting image most closely resembles a desirable structure (*structure-based methods*).

The PhD thesis [14] presents the different patterns that we have discussed in section 1.1 as well as several methods to reorder binary matrices. That document also proposes methods to assess how likely a matrix is to have each of the different patterns. This is done by computing the minimum number of elements that should be changed to obtain a matrix that contains each of those patterns.

The complexity of reordering a binary matrix is discussed in [18]. This is done by reducing that problem to several well known NP-Complete problems. The authors also propose two methods to reorder a binary matrix: a two dimensional sort and a variant of Sugiyama's graph layout.

We will now describe several existing methods for reordering binary datasets that we found in the literature and that we have implemented.

2.1.1 Distance based methods

The key difference between these methods and our method is that our aim is not only to consider similarities with direct horizontal and vertical neighbors, but also neighbors at a larger distance, including those along diagonals.

R seriation package A wide range of seriation techniques based on pairwise distances is available in the R seriation package [11].

Traveling salesman ordering This method is introduced in [12] and [21]. It consists in finding a permutation of a column store relation to minimize the number of transitions between attribute values. This is done by reducing this problem to an instance of the traveling salesman problem. This method can also work on binary matrices. Section 5.2.6 describes this process in details.

As mentioned before, those methods only work on binary data. However, the traveling salesman ordering can be generalized to categorical data (cfr. section 5.2.6).

Spectral ordering This way of reordering the rows and columns is presented in [1] and [14]. This method is based on spectral analysis on a similarity matrix over the columns of the matrix. It reorders the columns such that similar columns are close to each other. The general idea is to build a symmetric similarity matrix between the columns of the input matrix and then to construct an un-normalized Laplacian matrix L. The next step is to find the eigen vector associated with the second smallest eigenvalue of L (i.e. the Fiedler vector). As L is a symmetric real matrix, its eigen vectors contain real values. Therefore the Fiedler vector can be sorted to yield an ordering. Finally this ordering can be applied to the columns of the matrix. Several similarity measures can be defined such as the dot product or the *Pearson* correlation coefficient.

However, we observe that this method does not yield convincing results for the datasets that we study in this thesis. Therefore we have decided not to use this method in our experiments. Another issue is that the *Pearson* correlation coefficient is not defined for columns that have a unique value.

Heatmap Another popular way to obtain a graphical representation of data is the heat map. It is used to reveal hierarchical cluster structures on both the

rows and columns of a matrix. This ordering puts similar rows and columns next to each other based on some distance function. Standard implementations of heatmaps are available for a number of data analytics languages and systems, including R[1], Matlab[2], Python[3] and KNIME. In our experiments (see chapter 6), we use the `HeatMap3` package for R. We select the complete linkage clustering option and a `Manhattan` distance for that package. Although this method can be used both on categorical and binary datasets, the mapping of the colors is not changed, resulting in less visible patterns.

At the beginning of this clustering method, each row (resp. column) is in its own cluster. Then at every step, the two clusters separated by the shortest Euclidean distance are merged into a larger cluster. The distance between two clusters equals the distance between the two rows (resp. columns) (one in each cluster) that are farthest away from each other.

2.1.2 Structure based methods

Nested ordering This is the simplest method presented in [14] and [15] but it only works for datasets that have a nested structure. This method is based on the observation that nested datasets have their columns and rows sorted by the number of ones that they contain. The idea behind this method is therefore to count the number of ones in each row (resp. column) and then reorder the rows (resp. columns) according to those counts. They are both sorted by increasing number of ones.

Barycentric ordering This method is presented in [19] and [14]. The main idea is to find permutations such that ones are close to each other on each row and column. The paper defines a barycentric measure which is the average position of ones in this row/column. The barycenter of row r of matrix M is defined as:

$$barycenter(r) = \frac{\sum_{c=0}^{n-1} c * M_{r,c}}{\sum_{c=0}^{n-1} M_{r,c}} \tag{2.1}$$

Firstly, this algorithm computes the barycenters for all rows and then orders the row from the smallest to the largest barycenter; this process is repeated until convergence. At each iteration, the matrix is transposed such that both rows and columns will be reordered. This algorithm was designed to find a banded

[1] https://cran.r-project.org/web/packages/heatmap3/index.html
[2] https://www.mathworks.com/help/bioinfo/ref/clustergram.html
[3] http://seaborn.pydata.org/generated/seaborn.clustermap.html

structure in the data. Indeed, the rows and columns of banded structures are ordered by their barycenters

Bidirectional ordering This method is also described in [7] as the alternating algorithm, in which they observe that one can find a good row permutation on either M or M^T. Therefore, the paper suggests the following alternating method. At each iteration, use an algorithm called `Bidirectional Fixed permutation` to find a good row permutation given the current column permutation. Then transpose the matrix and apply the same procedure until convergence or until a maximum number of iterations is reached.

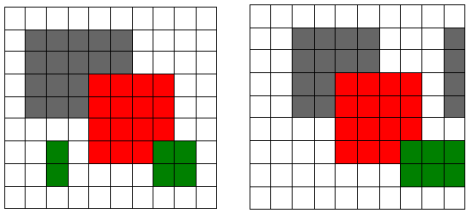
The `Bidirectional Fixed permutation` assumes that the ordering of the columns is fixed. First, given a matrix M, find the closest (by changing as few zeros or ones from M as possible) consecutive one matrix called C. C is a matrix that has the consecutive ones property, therefore every row contains at most one interval full of ones. Hence, every row of C can be defined by a (possibly empty) interval. The next step is to resolve the *Sperner* conflicts between rows. A matrix has *Sperner* conflicts if its rows do not form a *Sperner* family of intervals. The rows of a consecutive ones matrix are a *Sperner* family of intervals if for any two rows i and j, i (resp. j) is not properly included in j (resp. i). Finally the rows of C are sorted by increasing order of intervals. We obtain a permutation of the rows π that we apply to the rows of M.

A variant of `Bidirectional Fixed permutation` is to find the closest consecutive one matrix by changing only the zeros into ones.

2.2 Pattern mining

Frequent pattern mining is a well studied problem that consists in finding relationships among items in a database. Algorithms such as `Tile Mining` [9] can be used to find regions in the matrix consisting solely of ones. The outputs of these methods is a set of patterns that can be found in the matrix. However, they do not directly tackle the problem of visualization. Visualizing patterns is not always straightforward because several choices are possible, each of them displaying different patterns.

As an example, we consider the three rectangles of different colors on figure 2.1. Displaying an image where the three rectangles are immediately



(a) Permutation 1 (b) Permutation 2

Fig. 2.1 Dataset containing rectangles

visible is not possible, therefore a choice has to be made. We display two possible solutions: one where the gray and red rectangles are visible and one where the red and green rectangles are visible. For this small example, this is not very difficult but for larger datasets, finding the best permutations of the rows and columns such that as many patterns as possible are visible can be tricky. Our algorithm helps to tackle this problem.

Another problem that arises from these algorithms is that they usually do not deal with noise. They look for patterns (i.e. clusters of ones) that are perfect. If a tile contains a single zero, the algorithm will not find it. Therefore, our method should be able to display patterns that contain noise as they might still provide a lot of information. Moreover, most real datasets contain noise.

Chapter 3

Description of the problem

3.1 Properties of a faithful visualization

As explained earlier, the goal of this thesis is to find a permutation such that the error when the matrix is zoomed out is low and that similar rows and columns are close to each other. A possible criterion is to look for a matrix with few transitions between ones and zeros. Indeed, those transitions might induce errors when zoomed out because the ones and zeros will interact with each other. An element with value one surrounded by other elements with value one is likely to stay a one in the zoomed out version. On the other hand, an element with value one surrounded by some elements with value zero is likely to become gray and to induce some error.

In order to obtain a matrix with as few one-zero transitions as possible, we define an objective that will be minimized when such a matrix is found. In other words, when the number of one-zero transitions decreases, the value of the objective function should decrease too. We call the problem of minimizing this function `ConvoMap`, this will be our main focus.

An energy function has already been introduced in [7]. This function computes the minimal number of values that must be changed to make the matrix fully banded. However, this function is specific to the banded case and cannot be generalized to matrices with other patterns.

A possible solution is to use convolutions because they are used to apply a filter on an image. This technique can be used for blurring, sharpening, edge detection and more.

The blurring is what we are most interested in. Indeed, if a matrix (i.e. image) is such that the ones (i.e. black points) are not close to each other, blurring that matrix gives an image that is mainly gray. On the other hand, if the ones are close to each other, then the resulting image is more faithful to the original image. This is due to the fact that the points do not interact too much with each other, as explained before.

Before defining the objective function more formally, we describe in details the process of convolution.

3.2 Convolutions

Convention Unless otherwise specified, we consider by convention that the bottom left of the matrix is at coordinates $(0,0)$ and that every vector or matrix is 0-indexed.

The convolution uses two input matrices. The first one is the matrix we want to modify (M). The second one, called the kernel (K), is a matrix defining the filter. For the result of the convolution to yield values between 0 and 1, the kernel needs to be normalized (see later).

The filter represents the operation to be applied on each element of the first matrix M. Each element of M is replaced by the weighted sum of its neighbors and itself (the weights are defined by the kernel).

In other words, the kernel slides on M. Each element of the kernel is multiplied with the overlapping element of M. All those results are added up and this sum is the value of the output matrix at the point in M where the kernel is centered.

Note that the effect of the convolution depends on the kernel that is used.

Definition: Convolution Let M be a $m \times n$ matrix and K be a $k_m \times k_n$ kernel (with k_m and k_n odd, for symmetry). The convolution $M * K$ of the matrix coordinate (r,c) is defined as:

$$(M * K)_{r,c} = \sum_{k_r=0}^{k_m-1} \sum_{k_c=0}^{k_n-1} M_{r+k_r-\lfloor k_m/2 \rfloor, c+k_c-\lfloor k_n/2 \rfloor} * K_{k_r,k_c} \tag{3.1}$$

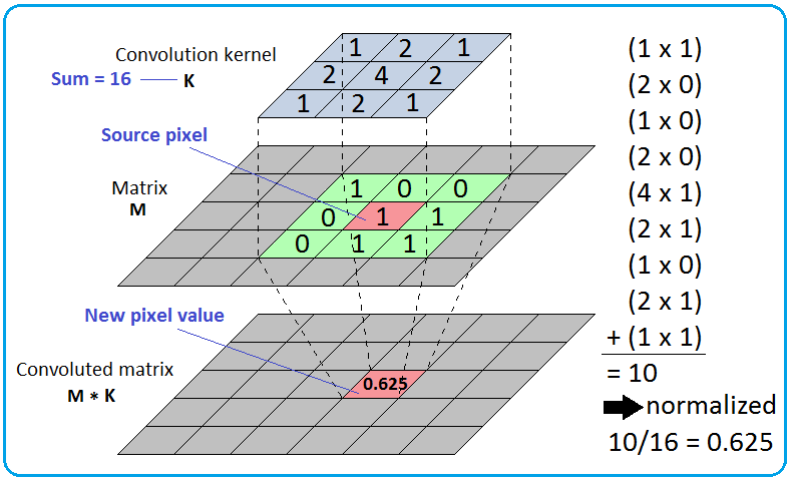


Fig. 3.1 Example of a convolution using a 3×3 Gaussian Blur Exponential kernel

3.2.1 Kernels

A kernel is a matrix. In this thesis, we only consider kernels that have an odd size for both rows and columns. We also consider kernels that only contain positive values. There are different types of kernels. A few are given below.

Uniform Kernel This kernel has 1's everywhere.

Gaussian Blur Exponential Kernel (GBE) Its 3×3 form is:

$$\begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

They are also defined for an arbitrary odd size $k_m \times k_n$, center of the rows $c_{row} = \lfloor k_m/2 \rfloor$, center of the columns $c_{col} = \lfloor k_n/2 \rfloor$ and center $c_{max} = max(c_{row}, c_{col})$ as follows:

$$K_{r,c} = \frac{2^{2c_{max}}}{2^{|r-c_{row}|+|c-c_{col}|}} \tag{3.2}$$

The effect of this kernel is to blur images.

Gaussian Blur Smooth Kernel (GBS) Its 3×3 form is:

$$\begin{pmatrix} 1 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

They are also defined for an arbitrary odd size $k_m \times k_n$, center of the rows $c_{row} = \lfloor k_m/2 \rfloor$, center of the columns $c_{col} = \lfloor k_n/2 \rfloor$ and center $c_{max} = max(c_{row}, c_{col})$ as follows:

$$K_{r,c} = c_{max} - max(|c_{row} - r|, [c_{col} - c|) + 1 \tag{3.3}$$

The effect of this kernel is also to blur images, except that more importance is given to the neighbors that are further. When Gaussian kernels are used, the rows and columns of the matrix cannot be reordered independently because swapping two rows (resp. columns) changes the errors of both rows and columns, because the kernel has non zero entries on its diagonals.

Cross Kernel Kernel type for which any value that is not on its middle row or middle column is equal to 0. Such a kernel allows to reorder the rows and columns independently. Indeed if we change the order of the rows (resp. columns) this only affects the number of transitions within every column (resp. row). Hence changing the order of the rows (resp. columns) does not have an impact on the optimal order the columns (resp. rows). Any kernel can be transformed into a cross kernel by setting to zero cells not in the cross (and renormalizing it).

Gaussian Blur Smooth Cross Kernel (CGBS) This is a GBS kernel for which all entries that are not on the middle row or middle column are equal to 0.

Properties of kernels

- *Square*: A kernel is square if its number of rows equals its number of columns
- *Row symmetric*: A $k_m \times k_n$ kernel is row symmetric if it is equal to itself with the order of the rows reversed:

$$\forall r \in \left\{0, \ldots, \frac{k_m}{2}\right\}, \forall c \in \{0, \ldots, k_n - 1\} : \quad kernel_{r,c} = kernel_{k_n - 1 - r, c} \tag{3.4}$$

- *Column symmetric*: A kernel is column symmetric if its transpose is row symmetric.
- *Equal to its transpose, or symmetric*: A kernel is symmetric if it is row and column symmetric and if it is square.
- *Normalized*: The sum of its values equals 1. To normalize a kernel, each of its values is divided by the sum of its values:

$$K_{normalized} = \frac{K}{\sum_r \sum_c K_{r,c}} \tag{3.5}$$

  We assume for the rest of this paper that the kernels are normalized.

3.2.2 Padding

Computing values on the edges require values outside of the matrix. There are a few solutions to get the value of $v = M_{r,c}$ of a $m \times n$ matrix M, where $r \notin \{0, \cdots, m-1\}$ or $c \notin \{0, \cdots, n-1\}$

- **Extension**: Values on the edges are copied as far as necessary to provide values for the convolution.
  - If $r < 0$ take the bottom row, if $r \geq m$ take the top row
  - If $c < 0$ take the left column, if $c \geq n$ take the right column
- **ZerosAndOnes**: Values on the edges are replaced by either zero or one.
  - If $r < 0$ or $c < 0$ the padding is equal to one
  - Otherwise the padding is equal to zero
- **Cropping**: Values on the edge are replaced by zero.
  - $v = 0$

3.3 Objective function

Now we can define the objective function that will have to be minimized. As explained before, the convolution of the matrix with a Gaussian kernel gives a measure of how good the permutation is. Hence, we define the objective function as the difference between original matrix M and a new matrix $(M * K)$

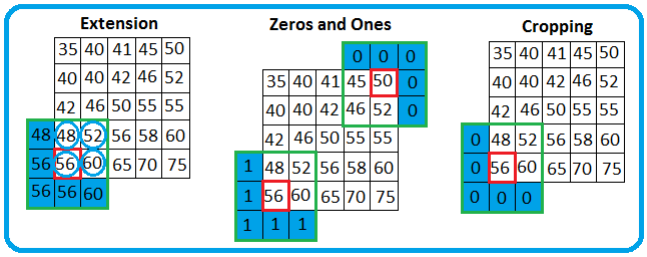


Fig. 3.2 Types of padding

obtained by convolution with some kernel. The kernel is a parameter to be chosen, the one we use is the GBS and was chosen experimentally (see chapter 6).

$$\text{Error}_{\text{quadratic}} = \sum_{r=0}^{m-1}\sum_{c=0}^{n-1}(M_{r,c} - (M * K)_{r,c})^2 \tag{3.6}$$

$$\text{Error}_{\text{abs}} = \sum_{r=0}^{m-1}\sum_{c=0}^{n-1}|M_{r,c} - (M * K)_{r,c}| \tag{3.7}$$

Using the sum of the squares of the differences (eq:3.6) favors many small errors over a few big errors. The sum of the absolute values (eq:3.7) can also be used, which would favor few big errors over many small ones. For `ConvoMap`, we use the absolute value.

One can notice that when using the absolute value, computing the error only for ones is enough when using a symmetric kernel.

Theorem For symmetric kernels, minimizing $\text{Error}_{\text{abs}}$ only for $r,c : M_{r,c} = 1$ $\iff$ Minimizing $\text{Error}_{\text{abs}}$ for all r,c

Proof We reformulate equation 3.7 to give it an intuitive meaning. First, we introduce the concept of disagreement between two entries of the matrix. We say that entries $M_{a,b}$ and $M_{c,d}$ disagree if they have different values. We define:

- $disagreement(M_{a,b}, M_{c,d}) = 1$ if $M_{a,b} \neq M_{c,d}$
- $disagreement(M_{a,b}, M_{c,d}) = 0$ otherwise

We can give a weight $w_K(M_{a,b}, M_{c,d})$ to the pair $(M_{a,b}, M_{c,d})$ defined as $w_K(M_{a,b}, M_{c,d}) = K_{\lfloor k_m/2 \rfloor + c - a, \lfloor k_n/2 \rfloor + d - b}$ where k_m and k_n are respectively the number of rows and columns of the kernel. This weight corresponds to the

weight associated to entry $M_{c,d}$ when the kernel is centered on $M_{a,b}$. If the kernel is symmetric, we have $w_K(M_{a,b}, M_{c,d}) = w_K(M_{c,d}, M_{a,b})$. Entries outside of the kernel are equal to zero.

We define V_0 (resp. V_1) the set of the entries equal to zero (resp. one) and $V = V_0 \cup V_1$ the set of all the entries. We show in the following that the error can be written as the weighted sum of the disagreements:

$$\text{Error}_{\text{abs}} = \sum_{x \in V} \sum_{y \in V} w_K(x,y) * \mathit{disagreement}(x,y) = \sum_{x \neq y \in V} w_K(x,y) \tag{3.8}$$

We know that the convolution of an entry x can be expressed as:

$$\begin{aligned} (M * K)(x) &= \sum_{y \in V_1} 1 * w_K(x,y) + \sum_{y \in V_0} 0 * w_K(x,y) \\ &= \sum_{y \in V_1} w_K(x,y) \end{aligned} \tag{3.9}$$

We also know that the kernel is normalized and that values outside of the kernel are given a weight of zero, therefore for each entry x we have:

$$\sum_{y \in V} w_K(x,y) = \sum_{y \in V_0} w_K(x,y) + \sum_{y \in V_1} w_K(x,y) = 1 \tag{3.10}$$

We now compute the error for every entry x in the matrix. There are two situations:

1. $x \in V_1$. The error for this element is defined as:

$$\begin{aligned} \mathit{error}(x) &= |1 - (M * K)(x)| \\ &\overset{3.9}{=} \left| 1 - \sum_{y \in V_1} w_K(x,y) \right| \\ &\overset{3.10}{=} \left| 1 - \left(1 - \sum_{y \in V_0} w_K(x,y) \right) \right| \\ &= \sum_{y \in V_0} w_K(x,y) \end{aligned} \tag{3.11}$$

2. $x \in V_0$. The error for this element is defined as:

$$\mathit{error}(x) = |0 - (M * K)(x)| \overset{3.9}{=} \left| 0 - \sum_{y \in V_1} w_K(x,y) \right| = \sum_{y \in V_1} w_K(x,y) \tag{3.12}$$

Finally the error is equal to:

$$\begin{aligned}\sum_{x\in V} error(x) &= \sum_{x\in V_1} \text{error}(x) + \sum_{x'\in V_0} \text{error}(x') \\ &\overset{3.11,3.12}{=} \underbrace{\sum_{x\in V_1}\sum_{y\in V_0} w_K(x,y)}_{(a)} + \underbrace{\sum_{x'\in V_0}\sum_{y'\in V_1} w_K(x',y')}_{(b)} \\ &= \sum_{x\neq y\in V} w_K(x,y)\end{aligned} \tag{3.13}$$

This proves equation 3.8.

From 3.13, one can interpret the error as the weighted count of disagreements in the matrix. For 3×3 kernel, the error computes the weighted sum of immediate transitions in the matrix. Indeed, $w_K(M_{a,b},M_{c,d})$ is equal to zero if $M_{a,b}$ and $M_{c,d}$ are not direct or diagonal neighbors. For 3×3 cross kernels, this further simplifies to counting the weighted sum of horizontal and vertical transitions.

For 3×3 kernels, this global error is minimal if the overall number of transitions is minimal (with the corresponding weights taken into account). Minimizing the error 3.7 is thus minimizing the number of 1-0 transitions in the matrix.

To prove the theorem, we observe in equation 3.13 that for a symmetric kernel, $(a)=(b)$ and thus equation 3.13 becomes:

$$\sum_{x\in V} error(x) = 2*\sum_{x\in V_1}\sum_{y\in V_0} w_K(x,y) = 2*\sum_{x\in V_1} error(x) \tag{3.14}$$

Minimizing 3.14 is the same as minimizing 3.7. Therefore, for the rest of this document, we only minimize the error for entries equal to 1.

3.4 Categorical datasets

So far, only binary matrices were discussed. This section explains how to extend this objective function to categorical datasets.

In a categorical dataset, attributes can take one of a limited and usually fixed number of possible values. If the categorical dataset contains only two different values, the dataset can be converted into a binary dataset.

The issues thus arise when the dataset contains more than two different values, and even more issues arise when the values can have different types. For example, the attribute gender could contain text whereas age could contain a number.

3.4.1 Values to binary vector mapping

The fix to the above issue is done by mapping each different pair (attribute, value) to a vector of binary values. The one to one mapping is referred as $\texttt{Map}_{\text{bin}}$. Several values for different attributes can be mapped to the same vector.

As an example, assume we have a dataset that contains information about people. For each person, the dataset stores his gender and his eye color. The gender is a value in the set {"man", "woman"} and the eye color is a value in {"blue", "green", "brown", "gray"}. One possible mapping is then:

Attribute	*Value*	*Binary vector*
gender	man	$(0,0)$
gender	woman	$(1,1)$
eye color	blue	$(0,0)$
eye color	green	$(0,1)$
eye color	brown	$(1,0)$
eye color	gray	$(1,1)$

Observe that *man* and *blue* (and *woman* and *gray*) map to the same binary vector. However this is not an issue because theses values belong to different attributes.

3.4.2 Values to color mapping

Once we have the mapping $\texttt{Map}_{\text{bin}}$ from pairs (attribute, value) to binary vectors, we can define another mapping $\texttt{Map}_{\text{col}}$ from binary vectors to colors. The colors allow to display the categorical matrix. In fact, we have a mapping from pairs (attribute,value) to color defined as $\texttt{Map}_{\text{full}} : x \to (\texttt{Map}_{\text{col}} * \texttt{Map}_{\text{bin}})(x)$. Since a binary vector of length n can support 2^n different values for an attribute, if the maximum number of different values for an attribute is max, then a vector of size $\lceil \log_2(max) \rceil$ is needed.

3.4.3 Objective function

The objective is to find a permutation of the rows and columns and a mapping from pairs (attribute, value) to colors such that elements with the same color are close to each other. The mapping from values to colors for each attributes is arbitrary and can thus be changed. The objective is to get a matrix with few transitions between colors, because displaying such a matrix is easier when the matrix is very large. Lastly, similar rows and columns are more likely to be close to each other, which makes the extraction of patterns from the data easier as well.

A good objective function is one that, when minimized, minimizes the number of transitions between colors. Again, we use a convolution with a Gaussian kernel, except that this time, each element of the matrix is a binary vector. Note that the objective function for binary matrices is a special case of the one for categorical matrices for vectors of size 1.

Definition: Convolution of matrix of vectors Let M be a $m \times n$ matrix. Let each element of M be a binary vector of size s and K be a $k_m \times k_n$ kernel (with k_m and k_n) odd, for symmetry). The convolution $(M * K)$ of M and K is still defined as:

$$(M * K)_{r,c} = \sum_{k_r=0}^{k_m-1} \sum_{k_c=0}^{k_n-1} M_{r+k_r-\lfloor k_m/2 \rfloor, c+k_c-\lfloor k_n/2 \rfloor} * K_{k_r,k_c} \tag{3.15}$$

The only difference from the binary case is that $M_{r,c}$ is a vector of size s, which makes $(M * K)_{r,c}$ a linear combination of binary vectors. The sum is over vectors and is done element-wise.

Since M only contains binary vectors and that the kernel K is normalized, we have $(M * K)_{r,c} \in [0,1]$ for all r, c in the matrix.

The error between M and $(M * K)$ (i.e. the convoluted matrix) is then defined as follows:

$$\text{Error} = \sum_{r=0}^{m-1} \sum_{c=0}^{n-1} \text{norm}(M_{r,c} - (M * K)_{r,c}) \tag{3.16}$$

Since $M_{r,c} - (M * K)_{r,c}$ is now a vector, we use the norm. We can take the *Euclidean* norm or the *Manhattan* norm. For a vector V of size s, they are defined as follows:

$$\text{Euclidean}_{\text{norm}}(V) = \sqrt{\sum_{i=0}^{s-1} V_i^2} \tag{3.17}$$

$$\text{Manhattan}_{\text{norm}}(V) = \sum_{i=0}^{s-1} |V_i| \tag{3.18}$$

3.4.4 Minimizing the number of transitions

This part explains how minimizing the objective function 3.16 also minimize the number of transitions between colors.

The error is minimal when similar elements are close to each other. Indeed, if two vectors are very similar, the norm of their difference will be small. Given an element $M_{r,c}$, if the elements around $M_{r,c}$ are equal to $M_{r,c}$, then $(M * K)_{r,c}$ is equal to $M_{r,c}$ and the error is zero. If one of the neighbors is different from $M_{r,c}$, then $(M * K)_{r,c}$ will not be equal to $M_{r,c}$ and the norm of the difference will be greater than zero. When the number of neighbors that are different increases, the error also increases.

3.4.5 Norm issues

Using norms induces an order among colors that depends on the mapping that we chose. For example, assume that red, green and blue are respectively mapped to $(0,0)$, $(1,0)$ and $(1,1)$. Then red is closer to green than to blue. Indeed the norm of $(0,0)$ - $(1,0)$ is smaller than the one of $(0,0)$ - $(1,1)$.

Minimizing the objective function tends to put elements whose color are more "similar" next to each other, but the colors were chosen arbitrarily. Ideally, the choice of the colors should not have an impact on the error. The only thing that should matter is to put elements with the same color next to each other, not the color themselves. Normalizing the vectors does not solve this issue.

A possible solution is to use one-hot encoding, which consists in choosing a mapping from pairs (attribute, value) to binary vectors that have exactly one element equal to one. The size of the binary vectors must then be equal to the number of different pairs. This solves the above issue, but has the drawback of making the matrix bigger as well as increasing the computations because of the increased vector size.

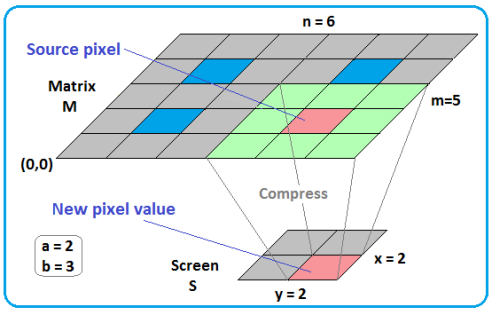


Fig. 3.3 Compression example

3.5 Compression of the matrix

Since a screen only has so many pixels, some of the values of the matrix have to be aggregated in order to display that matrix. We present two techniques, summarizing and sampling, to compress a matrix.

Summarizing Consider a $m \times n$ binary matrix M and a screen S that can fully display a matrix of size at most $x \times y$, such that $x = \text{scale}_r * m$ and $y = \text{scale}_c * n$ with $\text{scale}_r, \text{scale}_c > 0$. There is no need to scale the matrix down if $m < x$ (resp. $n < y$), therefore $scale_r$ (resp. $scale_c$) is set to be 1 in that case. Let $a = \lfloor \frac{1}{\text{scale}_r} \rfloor$ and $b = \lfloor \frac{1}{\text{scale}_c} \rfloor$. We also define a_{odd} as a if a is even and $a+1$ otherwise; b_{odd} is defined similarly.

To reduce the size of M to the one of S, we apply the following transformation to obtain each value (r,c) to display on the screen, where $r \in \{0, \cdots, x-1\}$ and $c \in \{0, \cdots, y-1\}$:

$$S_{r,c} = compress\,(M, ar + \lfloor a_{odd}/2 \rfloor, bc + \lfloor b_{odd}/2 \rfloor) \tag{3.19}$$

where *compress* is a function that will compress the $a_{odd} \times b_{odd}$ sub matrix centered on $M_{ar+\lfloor a_{odd}/2 \rfloor, bc+\lfloor b_{odd}/2 \rfloor}$ and output a single value in $[0,1]$ that represents this sub matrix. *compress* can be a majority vote, a weighted majority vote, a convolution using a $a_{odd} \times b_{odd}$ kernel or another function. An example is shown on figure 3.3. The coordinates that are the center of the 3×3 sub matrix are shown in light blue.

In practice we find that the convolution with a uniform kernel works well for binary matrices. This also has the advantage of giving a scale of which values are the most present in that sub matrix, since a value close to 1 means "lots of ones in the sub matrix", the same goes for 0's. Such a value can then

be displayed as a scale of gray.

For categorical matrices, we first convert the one hot encoding back into an integer number, and then apply a majority vote. We cannot use a summarizing convolution on categorical matrices as the resulting value will likely not correspond to any category (taking the floor or the ceiling of that value does not give this value any meaning either).

We distinguish between two types of categorical matrices:

- *commonAttributes*: matrices for which attributes for the categories are common across the whole matrix (ex: the news datasets, see section 6.1.2)
- *notCommonAttributes* : matrices for which attributes for the categories are not common across the whole matrix (ex: the categorical mushrooms dataset, see section 6.1.2)

For the *commonAttributes*, we can use a sub matrix with more than one columns without issues when rescaling. For the *notCommonAttributes* kinds, we cannot use a wide sub matrix because we could end up mixing attributes of non-related categories in a single column when rescaling. For these matrices, we can only do a majority vote column by column, and then sample which columns are displayed.

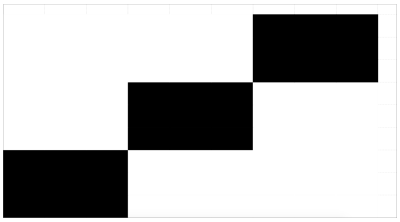
The majority vote has the advantage that we can get a sense of how well a value represents its corresponding sub matrix. If the vote is unanimous, the value represents the sub matrix well. If there was one of every category, the value does not represent well the sub matrix. This score can then be used to assess the quality of the rescaled image.

Sampling Another possibility is to sample the rows (resp. columns) to only keep a number of rows (resp. columns) that can be displayed on the screen, which amounts to keeping only one row (resp. column) out of r (resp. c). However, as we only keep some rows and columns, the information given by the rows and columns that are discarded is lost, which is a drawback. In other words, the rows and columns discarded do not have an impact on the image that is displayed. Consider the 9×9 matrix in figure 3.4a whose associated image is depicted in figure 3.4b.

Sampling this matrix to only keep 5 rows yields the image in figure 3.5a. This image shows distorted features as the original shape of the matrix is not

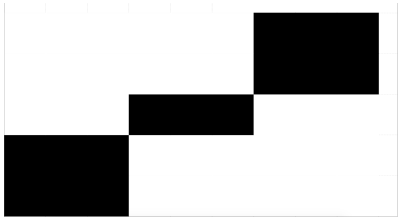
$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

(a) 9×9 matrix

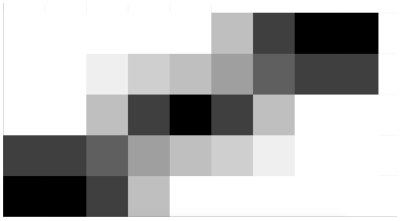


(b) Matrix to be sampled

Fig. 3.4 Matrix before sampling



(a) Sampled matrix



(b) Sampled convoluted matrix

Fig. 3.5 Matrix after sampling

maintained. Indeed, every block should be of the same size as in the original matrix.

The convolution helps mitigate this problem as every element that is sampled is convoluted and therefore takes into account its neighbors.

Convoluting the matrix given as example and then sampling the result gives the image in figure 3.5b. The gray points in the image help mitigate this issue because they give more information about the original matrix. If the points in the matrix have few black/white transitions, the value of the convolution will better represent the full matrix.

Chapter 4

Complexity of the problem

4.1 Size of the search space

In this chapter, we discuss more details about the search problems and how we tackled them.

As we mentioned earlier, we are looking for a permutations of the rows and columns such that the score of the matrix will be as small as possible (`ConvoMap`). A discussion of the size of the search space as already been presented in [18]. A naive way to tackle this problem would be to try every permutations.

The number of permutations for m rows is $m!$. For each of those permutations, we have to consider all the possible permutations of n columns. Therefore, the number of possibilities is $m! * n!$. In the case where the rows and columns are reordered independently, the number of possibilities becomes $m! + n!$. The values on table 4.1 show that this number grows very fast.

m	*m!*	*m! m!*	*2 m!*
6	7.2 x 10^2	5.2 x 10^5	1.4 x 10^3
10	3.6 x 10^6	1.3 x 10^{13}	7.2 x 10^6
12	4.8 x 10^8	2.3 x 10^{17}	9.6 x 10^8

Table 4.1 Factorials

Even for 10 by 10 matrices, finding a solution would take a few hours with a laptop. Pruning of the search space increases the size of the matrices we can process, as we will see later with `Branch and Bound` (Section 5.4). Note that even with pruning, solving problems with a few hundreds of rows and columns within a reasonable amount of time remains impossible.

4.2 Proof of NP-hardness

As seen in section 4.1, the problem we are tackling is quite complex. In this section, we provide a formal proof of its complexity by reducing the Hamiltonian Path Problem (`HamPath`) which is NP-hard to `ConvoMap` for a cross kernel.

The following proof of NP-hardness assumes the use of a cross kernel as this amounts to minimizing the number of direct transitions multiplied by the weight for neighbors in the cross kernel (See section 3.3). This does not prove the NP-hardness of the general problem with an arbitrary kernel, but the cross kernel is already very useful. Moreover, using a Gaussian kernel should make the problem even more difficult because the rows and columns cannot be reordered independently anymore. We also assume that the error is only computed for elements equal to one and the use of cropping padding.

4.2.1 Hamiltonian Path Problem

The Hamiltonian Path Problem is NP-Complete [8], therefore it is also NP-Hard. To solve `HamPath` is to find a path in a directed graph that visits all nodes, only once.

4.2.2 Reduction of the Hamiltonian Path to `ConvoMap`

The following proof is based on the proof in [21].

ConvoMap is the problem that consists in finding the permutation of the rows that minimizes the absolute objective function defined in 3.3. The cross kernel allows us to tackle the problem of reordering the rows and the columns independently (See section 3.2.1). Therefore we only focus on the permutation of the rows in this proof.

Theorem $\texttt{HamPath} \leq_p \texttt{ConvoMap}$

Proof In order to prove the NP-Hardness of `ConvoMap` we need to give a polynomial reduction of `HamPath` to this problem. Given an instance of the `HamPath` problem, an undirected graph $G(V,E)$, construct an instance of `ConvoMap` as follows.

We first construct the incidence matrix of the graph. This is a binary matrix I with $|V|$ rows and $|E|$ columns such that $I(v,e) = 1$ iff vertex v is incident upon edge e.

Note that if no incident nodes are adjacent in the ordering, the number of vertical transitions is maximal and equal to $4|E|$. Indeed there are 4 transitions in every column. Moreover, every time two ones in a column are put next to each other, the number of vertical transitions reduces by two.

Claim The graph G has an Hamiltonian path $\iff$ the number of vertical transitions in the order of `ConvoMap` is $4|E| - 2(|V| - 1)$.

The value $4|E| - 2(|V| - 1)$ is the minimal number of vertical transitions as at most |V-1| columns have adjacent ones. If that was not true, several edges would be adjacent to the same pair of nodes. However multiple edges can be removed because a Hamiltonian path visits every node exactly once. We now prove the correctness of this claim.

- $\Longrightarrow$: Suppose that G contains an Hamiltonian path.

  Reordering the rows of the incidence matrix according to the order of the vertices in the Hamiltonian path yields a matrix such that $|V-1|$ columns have adjacent ones.

  Indeed, for every pair of rows $(i, i+1)$ with $1 \leq i < |V|$ there is a column c such that $I(i,c) = 1 = I(i+1,c)$. If c did not exist, vertices i and $i+1$ would not be consecutive in the Hamiltonian path. This proves that there is at least one permutation of the rows such that $|V-1|$ columns have adjacent ones. Therefore this ordering has $4|E| - 2|V-1|$ vertical transitions.

- $\Longleftarrow$: Suppose the number of vertical transitions in the order of `ConvoMap` is $4|E| - 2(|V| - 1)$.

  This means that the permutation of the rows is such that $|V-1|$ columns have adjacent ones. Therefore every vertex i in this ordering is connected to vertex $i+1$. If we follow this ordering, we will visit every node exactly once. Hence this ordering defines an Hamiltonian path.

This shows that `HamPath` $\leq_p$ `ConvoMap`. Therefore, there is no known polynomial algorithm to solve `ConvoMap` with a cross kernel.

Chapter 5

Algorithm

5.1 Representation of the matrix

For efficiency, when changing the order of the rows and columns of the matrix, we do not change all the values of the matrix to reflect the new permutation. Instead, we only change the mapping of the rows and columns that keeps track of where each row or column is located, this way the original matrix never changes.

An example is given on figure 5.1. The bottom left of the matrix is at coordinates $(0,0)$. When in need to access element $(2,0)$ of the permuted (also called mapped) matrix, those indices have to be looked up in the rows and cols mappings, which yield the element $(0,4)$ in the original matrix (the mapped matrix is not kept in memory).

To swap rows 0 and 3 of the mapped matrix (rows 2 and 1 of the original matrix), only those two indices in the mapping of the rows need to be swapped, and not the whole rows. This effectively allows the swapping of two rows in $\mathcal{O}(1)$ instead of $\mathcal{O}(n)$ where n is the number of columns, while keeping $\mathcal{O}(1)$ access to the elements.

To transpose the matrix, one only needs to swap the mapping of the rows and of the columns, done in $\mathcal{O}(1)$. Given such an efficient transpose operation, all the moves that will be described in the following subsections are applied on rows. To apply a move on columns, just transpose the matrix, apply the move and then transpose the matrix again.

Transposing the matrix does not change the error as long as the kernel is equal to its transpose. When using non square kernels we also transpose the

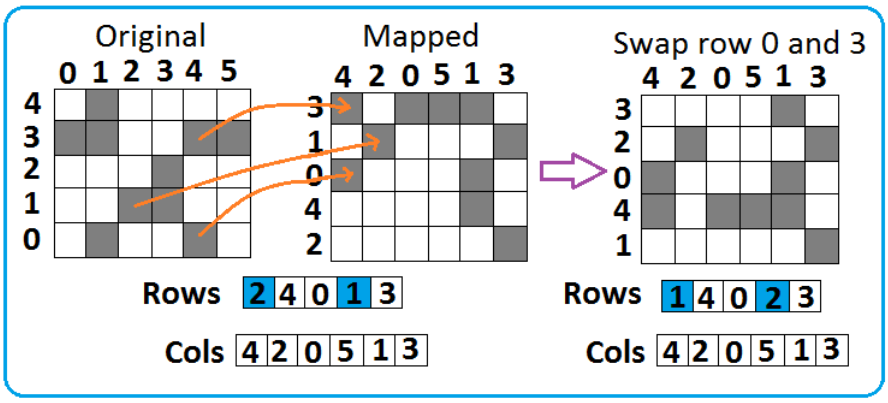


Fig. 5.1 Mapping of the rows and columns after a swap

kernel. This way the error does not have to be updated, as long as the kernel is row and column symmetric. For these reasons, we limit ourselves to kernels that are row and column symmetric. This requirement also becomes useful when we talk about the update of the error for the reverse move (Sect. 5.2.2).

Since every matrix is binary (categorical matrices can be converted in a set of binary matrices), we store them in bit sets, which makes them memory efficient.

5.2 `Local Search` algorithm

In section 4.2.2, `ConvoMap` was shown to be NP-Hard for a cross kernel. Therefore we cannot apply classical deterministic algorithms to solve large instances. A typical solution for solving computationally hard problems is to explore only a fraction of the search space, which can be done with a `Local Search` algorithm. Such an algorithm explores the search space by moving from one solution to another.

First, we have to define what are *candidate solutions* and the *scoring function*. The latter has been described in details in section 3.3. A candidate solution is described by a certain permutation of the rows and columns. The *initial candidate solution* can be a random permutation or the output of another algorithm.

Then we have to define the *set of moves* that can be applied to a candidate solution. After applying a move, the *score* of the candidate must be updated.

The last step is to choose a function that will output true when we want to keep a solution and false when we the solution should be rejected.

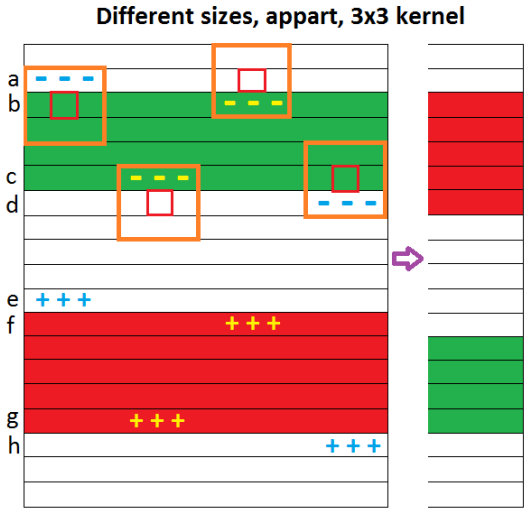


Fig. 5.2 Update of the error for the swap move

The *neighborhood* of a candidate is defined as the set of candidates that can be reached by applying a move from a chosen set of moves on the current candidate. The neighborhood has to be large enough otherwise we could get stuck in a local minimum.

5.2.1 Swaps

The first basic move is to select two random blocks of consecutive rows, and swap them. At first glance, we could allow the two blocks to be of any size and anywhere in the matrix, as long as they do not overlap. We will show later why that is not very efficient. We thus first discuss this move in a generic way, before making the move more specialized for reasons of efficiency. To begin with, we explain the update of the error for the generic case.

Update of the error A naive way to update the global error function after the swap would be to recompute the latter for the whole matrix. However, this is not efficient as only some values of the convolution error must be updated. If we use a 3×3 kernel, we must recompute the convolution error for at most 8 rows, as shown in figure 5.2.

One could also keep in memory the convolution of the mapped matrix, and update its values as needed. However, this requires twice the memory, which might not be suitable for big matrices. A more efficient solution is to keep the global error in memory and just make updates when we change the permutations.

The error for element $M_{r,c}$ is:

$$error_{r,c} = error\left(M_{r,c} - (M * K)_{r,c}\right) \tag{5.1}$$

where K is the kernel used and the error function is either quadratic (Eq:3.6) or absolute value (Eq:3.7). The error for the same element after the swap is:

$$error'_{r,c} = error\left(M'_{r,c} - (M' * K)_{r,c}\right) \tag{5.2}$$

where M' is the matrix after the swap. If we define by S the set of coordinates for which the convolution must be recomputed after the swap, the new global error becomes:

$$\begin{aligned}\text{global error} = \text{old global error} \\ + \sum_{(r,c)\in S} error\left(M_{r,c} - (M * K)_{r,c}\right) - error\left(M'_{r,c} - (M' * K)_{r,c}\right)\end{aligned} \tag{5.3}$$

As an example, figure 5.2 shows the rows for which the error is updated when the green and red blocks are swapped, for a 3×3 kernel. We analyze how the global error must be updated for line a.

1. First, we compute the error for every element of row a considering that row b is below row a.

2. Then we compute the same error but considering that row f is below row a.

3. Finally, we compute the difference between values obtained in (1) and (2) and add that difference to the global error.

The reasoning is the same for the other lines. Only the updates for the green lines are shown in figure 5.2, updates are done similarly for the red lines.

This update can be done for bigger kernels too in a similar way. Special cases arise when the kernel overlaps the two blocks, or when the two blocks touch each other. For a swap of adjacent blocks, the update (for a 3x3 kernel) is almost the same except that fewer lines have to be updated (6 instead of 8). (Fig. 5.2). These special cases are treated similarly and are not further developed here.

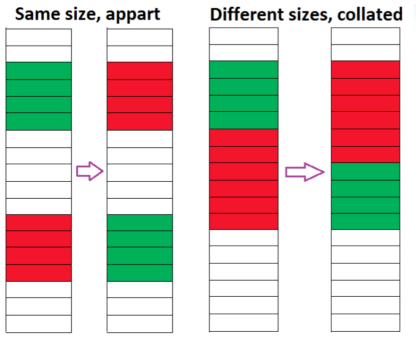


Fig. 5.3 Swap move for two separated blocks (left) and for two contiguous blocks (right)

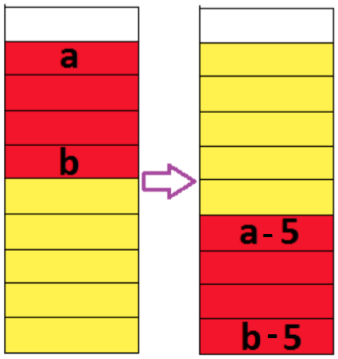


Fig. 5.4 Relocate move where $k = 5$

Efficiency concerns Earlier, we hinted at the fact that such flexibility on the swap move might not be efficient. Indeed, allowing such a move is costly, since the indices in the row mapping have to be shifted if the two blocks are of different sizes. This could be done more efficiently with a linked list, but we would then lose $\mathcal{O}(1)$ access time to the elements. For that reason, we restrict ourselves to two types of swaps (Fig. 5.3):

1. Swapping of two blocks of rows of the same size.
2. Swapping of two blocks of different size but which are adjacent.

Note how these two types only require the indices belonging to the two blocks to be changed in the mapping, all other indices remain unchanged. This makes this move possible in $\mathcal{O}(n)$ where n is the combined size of the two blocks (not taking the updating of the error into account for the complexity).

We further split these swaps into three different kinds, which we implemented as moves for the `Local Search`.

1. *Row swap*: Take two random rows and swap them
2. *Adjacent row swap*: Takes two adjacent rows and swaps them.
3. *Relocate*: Select 2 random rows a and b, which define a random block of rows. The block is then moved by a random number (k) of rows down, using the swap of adjacent blocks (Fig. 5.4).

5.2.2 Reverse

The reverse move consists in choosing two random row indices a and b ($a < b$) and reversing the order of the rows between them. The reverse move is defined by the following operations:

$$\forall i \in \left\{0, \ldots, \frac{b-a}{2}\right\}: \quad \text{swap the rows } a+i \text{ and } b-i \tag{5.4}$$

The update of the error can be carried out in a smarter way than by making updates on the error after every swap. Indeed, if the kernel is row symmetric, only a few rows will have a different convolution after the move.

For example, assume that for the matrix 5.5 (letters represent binary values) we choose the row indices $a = 1$ and $b = 4$. Once the rows have been reversed we obtain the matrix 5.6.

$$\begin{pmatrix} c & d \\ e & f \\ g & h \\ i & j \\ k & l \\ m & n \end{pmatrix} \tag{5.5}$$

$$\begin{pmatrix} c & d \\ \mathbf{k} & \mathbf{l} \\ \mathbf{i} & \mathbf{j} \\ \mathbf{g} & \mathbf{h} \\ \mathbf{e} & \mathbf{f} \\ m & n \end{pmatrix} \tag{5.6}$$

First, notice that the rows (i, j) and (g, h) have the same neighbors, except that they have been swapped. If the kernel is row symmetric, the convolution of those points are the same and there is no need to update them. Only the error of the other lines have to be updated.

The new global error is computed in a similar way as for the swap move. First, we must compute the errors for those rows before the move is applied. Then we must compute the same error but considering the new order of the rows, after the reverse. This new order can be obtained in the following way. For each row index r, there are two cases to consider:

1. If $a \leq r \leq b$ then the new position of row r is $a + b - r$
2. Otherwise the position of the row at index r is unchanged

After computing these two errors, we add their difference to the global error. Using this technique avoids many unnecessary operations compared to using the error update of the swap, especially if the block to reverse is large.

5.2.3 Swap rectangles

The following move is inspired from the algorithm described in [6]. All the moves described in the previous sections define a large neighborhood for a candidate solution. Indeed, we can reach many other candidates by applying one of those moves randomly. However, as the size of the matrix grows, the number of possibilities also grows very fast. Just for swapping two rows, we have to take two random values among the set of m rows, which gives us m^2 possibilities.

The problem

Assume that we have the following matrix.

$$\begin{array}{c} \\ b \\ c \\ d \\ e \\ f \\ \\ \end{array} \begin{pmatrix} \dots & \dots & \dots & \dots \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ \dots & \dots & \dots & \dots \end{pmatrix}$$

To improve the current solution we have to put the blocks of ones (b,c) and (e,f) next to each other. Just moving one of the rows will not improve the solution. Assume we are using the relocate move, how likely is that move going to shift block (b,c) one step down, or shift (e,f) one step up ?

To move the block (b,c), we have to select row c and then any row above c. Then we have to select a step of 1. Let n be the number of rows. The probability to select row c is $\frac{1}{n}$.

The probability to select a row above c is $\frac{c}{n}$, as there are c rows above row c. Finally, we can choose a step between 1 and $n-c-1$ (the number of rows below c). The probability of this move is thus $\frac{c}{n^2(n-c-1)}$. This value represents the probability of selecting row c and any row above c and a step of 1. We can compute the probability of moving the block (e,f) next to (b,c) in the same way. We obtain $\frac{e}{n^2(n-e-1)}$. The total probability is thus

$$p_{\text{success}} = \frac{c}{n^2(n-c-1)} + \frac{e}{n^2(n-e-1)} \tag{5.7}$$

Assuming that $c = e - 2 = 499$ and $n = 1001$, that probability equals $1.98 * 10^{-6}$.

To get a sense of how likely this is going to happen, we compute the number of times we have to try on average to successfully move the two blocks. As there are only two outputs (success and failure) and that the probability of success is the same for each trial (we cancel the move if the wrong blocks were moved), this corresponds to a geometric distribution.

The average number of failures before success is then given by $\frac{1-p_{\text{success}}}{p_{\text{success}}}$. Finally, by using only relocate moves, we have to wait on average ~ 500000 iterations before the two blocks are moved next to each other, which is far too many.

Defining a new move

Given the above issue, we are in need of a new move that will perform this operation much earlier.

The main idea behind this move is to select two rectangles full of ones and then put one of them next to the other. We do this by first selecting two elements of the matrix that are equal to one. Then for both elements, we find a rectangle that contains that element and that maximizes some scoring function. Finally, we move one of those rectangles next to the other by using the relocate move (we only move the rows, not the columns).

This problem is related to that of determining the largest rectangle of ones in an (ordered) matrix [22], but requires one specific entry to be included in the rectangle.

We first describe how to find a set of rectangles full of 1's that contains a given point. Then, we explain how to find two of these rectangles and discuss the scoring function that should be used in order to select them. At that point, we look at the complexity of our algorithm after which we explore improvements that can be made to take noise into account. Lastly, we discuss how this algorithm can be used with categorical matrices.

How to find the rectangles that contain point

The goal Our goal is to find a set of rectangles that contain a point with coordinate (r,c), and that only contains ones in matrix M. We only keep maximal rectangles: those that are not fully included in other rectangles.

The boundaries To achieve that goal, we need to define the concept of a boundary of a particular column which, intuitively, gives the highest and lowest row indices that a rectangle spanning column c up to that particular column can take, under the constraint that the rectangle must contain only 1's. Since the bottom left of the matrix has coordinate $(0,0)$ (by convention), we call these indices respectively the upper and lower bounds of the rectangle. As we go further away from column c (right or left), those boundaries can only shrink (the upper bound decreases, and the lower bound increases). If this was not the case, at least one of the boundaries between c and that particular column is not maximal.

We define the set of columns on the right of c as $R = \{c, c+1, c+2, ...\}$ where $M(r, c_r) = 1 \quad \forall c_r \in R$. L is defined in a similar way for columns on the left of c. We also define $LR = L \cup R$.

For each column $c_{lr} \in LR$, we keep the upper and lower bound (thus the rows) of the maximal rectangle that goes from c to c_{lr} and that only contains ones. We call those upper and lower bounds the boundary $B_{c_{lr}} = (upper, lower)$ of the rectangle for the particular column c_{lr}. Note that such boundaries always include row r.

For convenience, we also define the term *previous boundary* with regard to boundary $B_{c_{lr}}$ as $B_{c_{lr}+1}$ if $c_{lr} \in L$ and $B_{c_{lr}-1}$ if $c_{lr} \in R$. This term is not defined for B_c.

We now describe how to find the boundaries when going right (Algorithm 2). Finding the boundaries when going left is done in a similar way. We start by computing B_c by going up and down from (r,c) until we find a 0 (Algorithm 1). The computation of the next boundary B_{c+1} (assuming $c+1 \in R$) is done similarly, but we do not go further than the previous boundary since boundaries can only shrink. We then proceed in the same way for the rest of the columns in R.

Since we are only interested in finding the maximal rectangles, there is no need to keep every boundary in memory. In fact, we only keep a boundary if

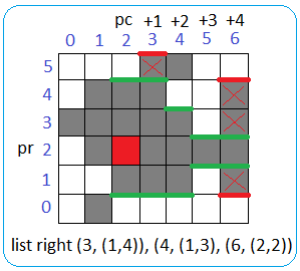


Fig. 5.5 The boundaries

that boundary is different from the next boundary (or if that boundary is the last one). An example of the boundaries obtained when going right is shown on figure 5.5, where the boundary (second element of the list) for a particular column (first element of the list) is shown.

One could wonder if precomputing the boundary for every point in the matrix before starting the search would be more efficient. Every time we perform a move, the boundaries could be updated such that we always have the correct boundary for each point.

The problem with this approach is that if we do not use the move swap rectangles often, we would spend most of the time updating the boundaries even if we do not need them. Even if this move is used a lot, we would still update the boundaries for many points that will not be used later. Moreover, we have noticed that this move is quite fast, therefore this is not a bottleneck for the search algorithm.

Algorithm 1: FindBoundary

Input: Matrix, (p_r, p_c), previous boundary: $(\text{prev}_{\text{up}}, \text{prev}_{\text{down}})$
Output: The maximal boundary
begin
  $up = p_r$
  while $Matrix(up+1, p_c) \neq 0$ ***and*** $up \neq prev_{up}$ **do**
    Increment up by one.
  $down = p_r$
  while $Matrix(down-1, p_c) \neq 0$ ***and*** $up \neq prev_{down}$ **do**
    Decrement $down$ by one.
return *(up, down)*

Algorithm 2: FindRectangleLimits

Input: Matrix M, (p_r, p_c), dir: left or right
Output: The list of boundaries for each column, going in direction dir
begin
  current $\longleftarrow$ `FindBoundary` $(M, (p_r, p_c), (0, \mathrm{M}_{\mathrm{rows}} - 1))$
  boundaries $\longleftarrow$ List()
  while *Moving in direction dir from* (p_r, p_c) *yields a 1* **do**
    previous $\longleftarrow$ current
    $n_c \longleftarrow$ Move one column from p_c in direction dir
    current $\longleftarrow$ current $\cap$ `FindBoundary` $(M, (p_r, n_c), previous)$
    if $current \neq previous$ **then**
      `Append` $(p_c, \text{previous})$ to boundaries
    $p_c \longleftarrow n_c$
  `Append` $(p_c, \text{current})$ to boundaries
return *boundaries*

Creating rectangles and selecting the best one

For all pairs of boundaries (B_l, B_r) where $l \in L$ and $r \in R$, we can take their intersection $B_{lr} = (up, down) = B_l \cap B_r$ to create new boundaries that span columns l to r. This defines a rectangle whose top left corner will be (up, l) and bottom right corner $(down, r)$. The notation we use to denote such a rectangle is: *Rectangle(top left corner, bottom right corner)*. Using a scoring function described in the next section, we compute the score for this rectangle and remember the best one found so far.

Algorithm 3: FindMaxRectangle

Input: Matrix, (p_r, p_c), a `Score` function for the rectangles
Output: The rectangle containing (p_r, p_c) that maximizes the score
begin
  $\text{list}_{\text{right}} \longleftarrow$ `FindRectangleLimits` (Matrix, (p_r, p_c), Right)
  $\text{list}_{\text{left}} \longleftarrow$ `FindRectangleLimits` (Matrix, (p_r, p_c), Left)
  max rect $\longleftarrow$ Rectangle((p_r, p_c), (p_r, p_c))
  for $(r_{col}, r_{boundary}) \in list_{right}$ **do**
    for $(l_{col}, l_{boundary}) \in list_{left}$ **do**
      $(up, down) \longleftarrow l_{\text{boundary}} \cap r_{\text{boundary}}$
      rec $\longleftarrow$ Rectangle((up, l_{col}), (down, r_{col}))
      if `Score` *(rec)* > `Score` *(max rect)* **then**
        max rect $\longleftarrow$ rec
return *max rect*

Scoring function

The scoring function gives a score to a rectangle and chooses which rectangles should be moved next to each other.

Since we only keep the maximal rectangles, the scoring function must satisfy the following condition: if two rectangles have the same height (resp. width) but different width (resp. height), the one with the longer width (resp. height) must have a bigger score. This limits the number of possibilities that we consider as we only have to add a triplet to the list when the maximal height changes.

Indeed, assume that the maximal height remains unchanged between two consecutive columns i and i' to the right. For every element of $\text{list}_{\text{left}}$, we would then have the choice between two rectangles of the same height but different widths. Thanks to the above condition, we only have to keep the one with the biggest width, hence the triplet with the biggest first value.

When moving the two rectangles next to each other, we only move the rows. The scoring function thus has to favor large width for the rectangles. The scoring function must also take into account the height of the rectangle because the goal is to select blocks of ones in the matrix to move next to each other. The scoring function that we found experimentally for a rectangle is:

$$width * \sqrt{height} \tag{5.8}$$

Finding two rectangles

We first randomly select a coordinate p_1 of the matrix such that $Matrix(p_1) = 1$. Using `FindMaxRectangle` (algorithm 3), we find among all the rectangles that contain p_1 the one that maximizes the score function. To be able to put the second rectangle next to the first one, we select another coordinate p_2 such that $Matrix(p_2) = 1$, on the same column as p_1. The second rectangle containing p_2 is found using `FindMaxRectangle` again (the scoring function could be different). Finally, we move the rectangles next to each other by moving the rows of one of them.

Complexity

For algorithm 3, the worst case scenario is when the row that we select for p_1 is full of ones, and that the ones create a staircase starting from p_1. There can then be no pruning of the boundaries. If $l = \text{length}(\text{list}_{\text{left}})$ and $r = \text{length}(\text{list}_{\text{right}})$, we have that $l + r = n$ where n is the number of columns of the matrix. In

that case, we have to consider $l * r = -l^2 + ln$ different rectangles. This is maximal when $l = n/2$. The highest number of rectangles we have to consider is therefore in $\mathcal{O}(n^2)$.

The complexity of `FindBoundaries` is $\mathcal{O}(m)$ where m is the number of rows of the matrix.

The total complexity for algorithm 3 is thus $\mathcal{O}(m * n^2)$

Improvements

In some situations more pruning is possible. Consider the figure 5.6. Let the red point be the one that needs to be included in the rectangle, and let its coordinates be $(2,4)$. Then we have $\text{list}_{\text{right}} = [(4,(0,4)),(5,(1,3))]$ and $\text{list}_{\text{left}} = [(4,(0,4)),(3,(1,3)),(2,(1,2)),(1,(2,2))]$. We denote by r_i (resp. l_i) the i^{th} element of $\text{list}_{\text{right}}$(resp. $\text{list}_{\text{left}}$) in the double for loops. In figure 5.6:

A We consider r_1 and l_1, we then obtain the brown rectangle.

B After one iteration, we now consider r_1 and l_2. We observe that $r_2 \subset l_2$, so we can skip r_1 and move to r_2 in the external for loop. Indeed, the brown rectangle of figure B is included in the rectangle of the figure C, so we do not have to consider that rectangle. All the other rectangles that will be created by considering r_1 are included in the rectangles created by considering r_2, so we can ignore them all. This is why we directly move to r_2.

C Here we are directly considering l_2. Without the pruning, we would have considered l_1. The rectangle created by using r_2 and l_1 is included in the rectangle created by using r_2 and l_2. For the same reasons as above, we do not need to consider that rectangle. In other words, for each boundary on the right, we start by considering the left-most boundary such that the boundary on the right is included in that left-most boundary.

We thus observe two improvements:

1. For each element $r_i \in \text{list}_{\text{right}}$, if this is not the last element and that the element of $\text{list}_{\text{left}}$that we consider is included in r_{i+1}, then we can stop the inner loop and directly consider the next element of $\text{list}_{\text{right}}$.

2. For each element $r_i \in \text{list}_{\text{right}}$, we must start by the last element l_j of $\text{list}_{\text{left}}$such that $r_i \subset l_j$

These two improvements yield algorithm 4.

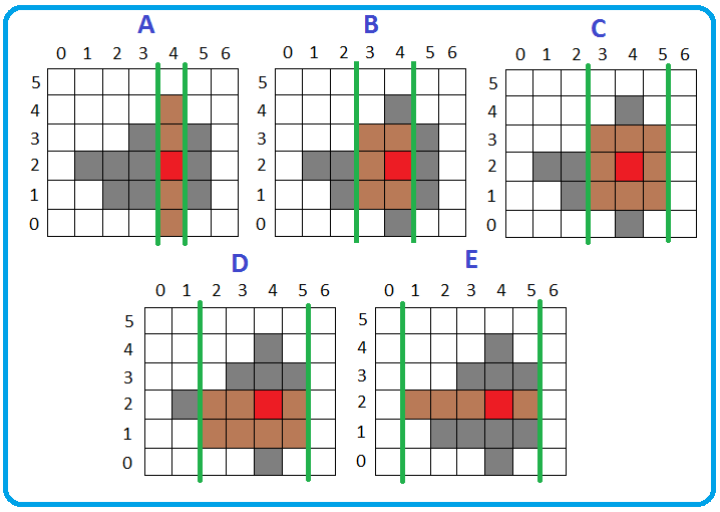


Fig. 5.6 Example showing the boundaries of the rectangles at various steps

Algorithm 4: FindMaxRectangle improved

Input: mat, (p_r, p_c), `Score`
Output: The maximal rectangle
begin
  list$_{\text{right}}$ ⟵ `FindRectangleLimits` (mat, (p_r, p_c), Right)
  list$_{\text{left}}$ ⟵ `FindRectangleLimits` (mat, (p_r, p_c), Left)
  max rect ⟵ Rectangle((p_r, p_c), (p_r, p_c))
  for $i \leftarrow 0$ **until** `Size` *(list$_{right}$)* **do**
    isLast ⟵ $i =$ `Size` (list$_{\text{right}}$) - 1
    j ⟵ the biggest j such that $r_{i:\text{boundary}} \subset l_{j:\text{boundary}}$
    while $j <$ `Size` *(list$_{left}$)* ***and*** $(isLast$ ***or*** $l_{j:boundary} \not\subset r_{i+1:boundary})$
    do
      $(up, down)$ ⟵ $l_{j:\text{boundary}} \cap r_{i:\text{boundary}}$
      rec ⟵ Rectangle((up, $l_{j:col}$), (down, $r_{i:col}$))
      if `Score` *(rec)* > `Score` *(max rect)* **then**
        max rect ⟵ rec
      $j \longleftarrow j+1$
return *max rect*

Noise

A last improvement to this algorithm is to take into account the noise. For example, finding a large rectangle in the matrix that contains mainly ones with a few zeros is interesting. With the current version, this is not possible because the algorithm stops at the first zero found.

A possibility to implement this is to change `FindBoundary` by using a threshold t on the number of zeros. We are then going up and down and allowing to cross t zeros before stopping and returning the boundary. Another possibility is to allow to cross more zeros (say z) than t if many ones (say u) have been crossed in a row. In practice we found that $t = 2$, $z = 1$ and $u = 4$ work well, however the performances depend on the dataset. For example, if we cross $16 = 4 * u$ ones, we allow to cross $4 = 4 * z$ more zeros, after having depleted the $2 = t$ zeros limit of the threshold.

In the function `FindRectangleLimits`, we could also allow to cross some zeros on the columns. This can be done by slightly modifying the behavior of the condition in the while loop.

Example One example for which including noise works well is on the following matrix. Assume $p = (2,2)$

$$\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

If we disallow a threshold the algorithm picks the rectangle containing the last two lines. If we allow a threshold of one, the algorithm selects all three rows and ends up with a bigger rectangle, which is better.

Possible problems of allowing noise One problem with noise tolerance is illustrated by the following matrix:

$$\begin{pmatrix} 0 & 0 & 0 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix}$$

If we allow a threshold on the number of zeros of one, the algorithm picks the entire matrix. Indeed having a threshold of one means that we accept one 1 in every column. Therefore the first line is selected because if we take it, every column has exactly one 1. However, this line only contains zeros and degrades

the quality of the rectangle. The technique we use to avoid selecting that first line is by starting from the top and bottom and removing rows until we find a row that has a density of 1's of at least 80%.

Categorical matrices

For categorical matrices, this move is slightly different. As explained earlier, those matrices contain categorical values. The goal is to reorder the rows and columns and find a mapping between pairs (attribute, value) and colors such that we get big clusters of elements with the same color. For those matrices, the technique used is different when we are reordering the rows or the columns.

When we are reordering the rows, the technique is very similar to what was explained before. We have to find two rectangles of the same color such that the intersection of their columns is not empty. The two rectangles must have the same color because changing their color such that both have the same color would change the data that the matrix represent. This is because each value of each attribute must be mapped to a different color, and attributes are arranged per column.

When we are reordering the columns, the situation is different. This time we are looking for rectangles such that the intersection of their sets of columns is empty. Hence, we do not have the same problem as before and we can pick two rectangles with different colors. We just have to change the color of one of them with the transformation described above and put them next to each other.

A last point that needs to be discussed is how we can find two rectangles with different colors. As explained earlier, we transform a categorical matrix into a list of binary matrices.

We denote T^i the matrix at index i in this list. Those matrices are such that $T^i_{r,s}$ is equal to one if and only if $A_{r,s}$ is equal to i (from the one hot encoding of the attribute values, see section 3.4.5). Hence, if we want to find a rectangle of the color mapped to number i, we select the i^{th} matrix in the list and find a rectangle that contains only ones. This way, we can select two rectangles of the same or different colors. If the move does not improve the current solution we reverse its effect by moving the rectangles back where they were and by reversing the transformation of the mapping.

5.2.4 Accept function

Now that the different moves have been defined, we describe the Local Search algorithm in more details. First the algorithm is given an initial candidate solution as input (discussed in section 5.2.6). At every step of the algorithm, we choose a move randomly and accept it only if the solution is improved. The goal is to find an accept function that allows the algorithm to converge to a good solution fast. That same function must also prevent falling into a local minimum that cannot be escaped.

The accept function that we have chosen is greedy, in the sense that we only accept solutions that improve the objective function. Indeed we have noticed that for large matrices reaching a local minimum is already difficult. If we do not succeed to reach a local minimum fast enough, there is no point in trying to escape from that local minimum by diversifying the solution. However, once such a local minimum has been reached, we could use some other methods, as described in section 5.2.5.

In order to avoid making copies of the current solution when performing a move, we reverse the move by applying its opposite if the new solution is rejected. If the move is accepted, the current solution becomes the state of the matrix after the move is applied. This way, no copy must be made, even when the move is rejected. In practice, we have noticed that keeping a copy does not improve performances.

5.2.5 Escaping local minima

The Local Search that we present here is a random walk hill climbing. Therefore the found solution might be a local minimum. This is why we have to design a way to escape local minima. The idea is to reorder the matrix globally. We do this by splitting the matrix horizontally into blocks of rows, and then reordering those blocks as if they were single rows. Each block is composed of all the columns of the matrix. The reordering can be done using Local Search, or if the number of blocks is small enough (say 10), with branch and bound. As the number of blocks is small, we use a 3×3 kernel.

Splitting of the matrix To split a $m \times n$ matrix, we cut the matrix into blocks of b rows: $[0,b[,[b,2b[,\cdots,[\lfloor m/b \rfloor * b, m[$. Note how the last block may be

smaller than the other blocks if b does not divide m. We thus end up with $\lceil m/b \rceil$ blocks.

Reusing the other moves In order to reuse the previous moves on that sub matrix, we "squash" each block of rows into a single row. Squashing the rows together is done by a majority vote: if the number of 1's (resp. 0's) in the column is greater than the number of 0's (resp. 1's), the value for that column will be 1 (resp. 0). We then end up with a $\lceil m/b \rceil \times n$ sub matrix, that we can feed into the branch and bound or `Local Search` algorithms.

Given that each row represents a block of rows, after the sub matrix has been reordered, care must be taken to reinsert the blocks correctly into the original matrix.

Update of the error We recompute the entire error from scratch, on the full matrix.

The error of the matrix after applying this ordering often increases because big blocks of rows are replaced by single rows. However, this allows the search to jump to other parts of the search space where a better solution is more likely to be found. This method to reorder the matrix should be used when the solution did not improve much for some time.

5.2.6 Initial solution

In order to start the `Local Search`, we have to choose an initial candidate solution. A simple idea would be to just take the raw matrix.

The problem is that if the matrix is large, putting similar rows and columns next to each other might be slow at the beginning. Indeed, all the moves except swap rectangles are random and only slightly improve the current solution.

For example, swap just takes two random rows and swaps them. The algorithm thus needs to carry out many iterations before most rows and columns have been considered. The other moves are not very useful at the beginning because they mainly improve the solution once the ones are already clustered.

A better idea is to run an algorithm that is faster to put similar rows and columns next to each other. Once this is done, we can use `Local Search` to improve the solution. This works better because ones will already be approximately gathered and the swap rectangle move will be more efficient.

To choose which algorithm should be used as a preprocessing step, recall the definition of ConvoMap. We are looking for the permutation of the rows and columns such that the number of transitions between ones and zeros is minimized.

The transitions can be direct if we use a 3×3 kernel or indirect if we use a bigger kernel, as explained before. If we take two rows, we can define a distance between them. We define this distance as the number of transitions between the two rows if they are put next to each other (Hamming distance). In other words, we define the distance between rows r_1 and r_2 as follows

$$distance(r_1, r_2) = \sum_{c=0}^{cols-1} r_1(c) \oplus r_2(c) \tag{5.9}$$

The goal is then to find the best permutation of the rows and columns such that the total distances between adjacent rows is minimized. If we consider that we can reorder the rows and the columns independently, we get a traveling salesman problem (TSP). This idea is inspired from the technique used in [21] and [12].

A practical issue with this approach is that TSPs are well-known NP-complete problems; however, as Hamming distances satisfy the triangle inequality, a reasonably good approximation algorithm can be used: the Christofides algorithm has an approximation factor of 3/2 [3].

One can interpret every row as a city and we are looking for the permutation that minimizes the total distance between adjacent cities (rows) (Fig. 5.7).

More precisely, we can proceed as follows. First, reorder the rows. To do so, construct a $n \times n$ distance matrix D such that $D_{i,j}$ is the distance between rows i and j. Such a matrix is symmetric. Then input that matrix to a TSP solver. The same can be done for the columns. Finally, the output is a new permutation for the rows and the columns. We can apply the permutations to the original matrix and use this as the input for the Local Search.

For categorical matrices, the distance function that we use is the number of values that are different between the two vectors. Indeed, we want to put rows with similar values next to each other.

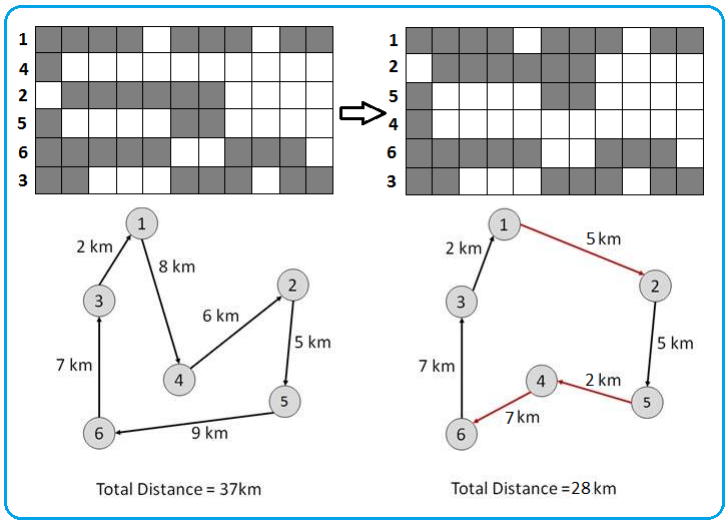


Fig. 5.7 Transformation of a matrix into a TSP instance

For the columns the distance function is different as we can change the mapping from pairs (attribute, value) to colors for an attribute.

Consider a $m \times n$ categorical matrix. In the following, we call Pairs(c_1, c_2) the set composed of the pairs $(color(c_1, i), color(c_2, i))$ for $i \in \{0, \cdots, m-1\}$ where $color(c_x, i)$ is the color at the i^{th} row of column c_x.

The goal is to have few elements in Pairs(c_1,c_2) and only a few pairs that appear often. In other words, if we count the number of times that each pair appears, we do not want those counts to be close to each other. This way, changing one of the mappings will affect many pairs.

We can achieve this by counting the number of times each different pair in Pairs(c_1, c_2) appears, followed by computing their sum of squares. This value is maximal when we have only one pair with a count of m. The higher the number of different pairs and the closer the counts, the smaller this value will be. The distance function that we want is:

$$distance(c_1, c_2) = n^2 - \sum_{(i,j) \in Pairs(c_1,c_2)} count(i, j)^2 \tag{5.10}$$

where *count* is the number of times that the pair (i, j) appears in Pairs(c_1,c_2).

This distance is zero if there is only one pair and increases as the number of different pairs gets bigger and as the counts get closer.

A last step is to preprocess the mappings from pairs (attribute, value) to colors (i.e. the colors) to have a better initial solution. We use the method described in algorithm 5. This is only done for *notCommonAttributes* types of categorical matrices (see section 3.5).

Algorithm 5: Preprocessing of the colors

```
Input: Matrix
Output: The matrix with the colors preprocessed
begin
    for i <- 0 until n-1 do
        P ⟵ GetMappingCounts (Matrix, i,i+1)
        forbiddenColors ⟵ empty array
        (color_i, color_{i+1}) ⟵ GetMaxPair (P, forbiddenColors)
        forbiddenColors ⟵ Append (forbiddenColors, color_i)
        while (color_i, color_{i+1}) ≠ (-1,-1) do
            Matrix ⟵ SwapColors (Matrix, i+1,color_i,color_{i+1})
            (color_i, color_{i+1}) ⟵ GetMaxPair (Matrix,
            forbiddenColors)
            forbiddenColors ⟵ Append (forbiddenColors, color_i)
    return Matrix
return Matrix
```

Algorithm 5 considers every consecutive pairs of columns starting from the left and does the following. For each pair of columns $(i, i+1)$, GetMappingCounts returns a mapping between each pair and the number of times that pair appears in columns i and $i+1$. GetMaxPair returns the pair of colors $(color_i, color_{i+1})$ that has the highest count in P and such that neither $color_i$ nor $color_{i+1}$ is contained in forbiddenColors. If there is no such pair, the function returns the pairs $(-1,-1)$.

After that we add $color_i$ to the list of forbidden colors. If such a pair was found, we swap the colors $color_i$ and $color_{i+1}$ in the $i+1^{th}$ of the matrix. This way we create more pairs of the same colors between those adjacent columns. Finally we repeat the content of the loop.

The purpose of the list forbiddenColors is to remember which colors we cannot change anymore without destructing what we did before. As an example, assume that GetMaxPair first returns the pair (red, green) for columns $(i, i+1)$. We swap colors red and green in column $i+1$ to create more pairs of the same color (red). If then GetMaxPair returns (red, blue) for the same columns, we would swap colors red and blue in column $i+1$ destructing all

the uniform pairs that we created before. Indeed, instead of having many pairs (red, red), we would now have many pairs (red, blue).

The solver that we use first finds an initial solution with a greedy algorithm. Then this solution is improved by applying several times the 2-opt algorithm presented in [4].

Why still use `Local Search` after `TSP` ? A first reason is that the traveling salesman problem is NP complete and thus cannot be solved in polynomial time. For big matrices `TSP` is very unlikely to find the optimal solution.

A second reason is that in order to be able to use a `TSP` solver to solve `ConvoMap`, we had to make the assumption that rows and columns could be independently reordered. This is the same as assuming that `ConvoMap` is using a cross kernel.

However, the kernels that we use do not have this property as they consider the whole neighbourhood of each point. We want to find some structure in the matrix. Consider the following matrix:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}$$

With a cross kernel, this would be considered as a good solution because putting the ones next to each other just by permuting the rows or columns is impossible. If we want to find the banded structure below, we have to use a kernel with non zero values on the diagonals.

$$\begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix}$$

A third reason is that we might want to use a bigger kernel than just 3×3. However, `TSP` only defines a distance between two rows and does not take into account a specific ordering. In order to optimize the score for a bigger matrix, we still have to run `Local Search` after the `TSP` solver. Nevertheless, the `TSP` solver already does a good preprocessing.

The `Local Search` algorithm will just try to improve the solution by taking more elements into account when computing the error for one element. Also, an advantage of using bigger kernels than 3×3 is that they allow to

better account for noise in the matrix, since each individual 0 will have a lower impact on the convolution (see section 6.5.1).

5.2.7 Learning of the moves

In the previous sections, we defined a number of moves that can be applied to a matrix in order to explore the space of its neighbors. In this section, we discuss how to choose the next move to use in Local Search, given a set of possible moves.

One possibility is to select a single move for the whole search. However, some moves are better suited for some particular instances, and might not work well on others. For example, in the following matrix, a single reverse move between the second and last line gives the optimal solution. If only the swap move was selected, the probability to apply the swaps such as to obtain the same result is lower.

$$\text{non-optimal} \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \quad \text{optimal} \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix}$$

Another possibility is to use all possible moves, and let the algorithm learn which ones are the most effective on a particular instance. This technique has already been studied as part of the adaptive large neighborhood search (ALNS) [24] [23]. ALNS is a search technique that uses several different heuristics and chooses between them using statistics collected during the search. Given k moves, each move is attributed a weight w_j, $j \in \{1 \cdots k\}$. The probability that move j is selected is:

$$\frac{w_j}{\sum_{i=1}^{k} w_i} \tag{5.11}$$

The weights can be seen as a score that gives how well a particular move is currently performing. The better the move performs, the better its chances to be selected as the next move. To find theses weights, we split the search into segments, which correspond to a fixed number of iterations. At the start of the algorithm, each move has an equal chance to be selected, and has a weight

of 1.0. At the beginning of a segment, the score of each move is set to zero. If a move j improves the current solution by reducing the error by $delta$ in segment i, then its score s_{ij} is increased by $delta$. At the end of segment i, the weight $w_{i+1,j}$ of move j for the segment $i+1$ is updated as follows:

$$w_{i+1,j} = w_{ij}(1-r) + r\frac{s_{ij}}{\theta_j} \tag{5.12}$$

Where the reaction factor r adjusts how fast the weights will react to changes in how effective a particular move is, and θ_j is the number of times the move j was selected during the last segment.

One could also consider the time a particular move takes in the computation of its score.

5.3 Parallelization of the `Local Search`

As the `Local Search` algorithm can take a long time to execute, we introduce a technique that makes use of parallelization, that we will use in the experiments. Indeed, if we cut the matrix into blocks, we can optimize each block independently. This parallelization makes the search faster.

In the following we call `N-SplitSearch` the method that splits the rows of the matrix into N blocks of consecutive rows of equal size (except maybe the last one) and performs `Local Search` on each of them in parallel. Then the same is done for the columns. Finally all those permutations are applied to the original matrix.

We also call `block ordering` the method that consists in cutting the matrix into 10 blocks. Each block is aggregated into one line by using a majority vote for each column. Then we use the branch and bound search of section 5.4 to reorder the rows of this new matrix perfectly. Finally, we apply the same permutation to the blocks of the original matrix. This might cause the error to increase but this method is very useful for escaping local minima, especially for banded matrices.

The overall parallelized `Local Search` ordering consists in the following steps.

1. Perform a block ordering

2. Perform a 3-SplitSearch
3. Perform a 2-SplitSearch
4. Perform a block ordering
5. Perform a 3-SplitSearch again
6. Perform a normal `Local Search`

Note that we reuse a `3-SplitSearch` after the block ordering, because that block ordering might increase the error. However this is important to escape local minima.

A second remark is that we perform several `N-SplitSearch` because local information is not enough to obtain a very good solution. As an example, after running a `3-SplitSearch`, each of the blocks is optimized but they do not take into account what is inside the other blocks. Therefore the solution can be improved by cutting the matrix into less blocks so that more rows will be taken into account.

Finally, we return the best matrix we encountered, because there are are no guarantees that the last matrix is going to be the best one.

In the rest of this chapter, `P-LS` stands for `Parallelized Local Search`.

5.4 Brute force

As discussed in section 4.1, the search space for `ConvoMap` is huge. Trying all permutations can only be performed for 9×9 matrices in reasonable time (i.e. in less than a few hours). Finding the optimal solution is however possible much faster for bigger matrices by using `Branch and Bound`.

Motivation Even with pruning one cannot expect to use branch and bound to solve large matrices. However `Branch and Bound` can still be used locally by selecting a small subset of adjacent rows. We can use this algorithm as a move in the `Local Search` algorithm (as explained before).

Another solution is to try to use `Branch and Bound` on the whole matrix by reducing its size to something small (ex: 25×25). This is done by cutting

the matrix into a small number of blocks of n rows. `Branch and Bound` can directly reorder those blocks between each other. Those blocks can possibly be aggregated into a few lines by using a majority vote for each column or some other aggregation method. This does not ensure that the global error will decrease, however this can be used to move to other parts of the search space if we are stuck in a local optimum.

Main idea The idea behind `Branch and Bound` is to search through the space of solutions for the best one according to some scoring function. However, not all solutions are explored. Lower bounds are computed on partial solutions to prune part of the search space. If this lower bound is higher than the best solution found so far, solutions built from that partial solutions can be pruned. High quality lower bounds allow to prune more of the search space, at the expense of being more expensive to compute.

Scoring function The scoring function is the error defined in equation 3.7, which requires to choose a kernel. Using the Gaussian kernel poses the issue that rows and columns cannot be reordered independently (cfr. 3.2.1), therefore the number of candidate solutions would be $rows! * cols!$. On the other hand, reordering the rows and columns independently yields a search space of size $rows! + cols!$ (See section 4.1). Using a cross kernel allows independent reordering of rows and columns, so this is what we will use. We first explain the algorithm assuming that the kernel is 3×3 and then we show how to generalize to larger kernels.

Symmetries Two solutions are symmetric if applying a reverse move on all the rows and/or all columns on one yields the other. Solutions that are symmetric have the same score if the kernel is symmetric and that the type of padding is symmetric too. Symmetries do not need to be explored, and are thus pruned, which reduces the search space by a factor of two.

Algorithm Solutions are built incrementally, at each step a column or row is added to the partial solution following a depth-first search technique. When all rows and columns have been added, compare the solution with the best one found so far, and keep the best one. This process is described in algorithm 6, which is initially called with an empty partial solution and the set of lines of the matrix to order.

Algorithm 6: Branch and Bound (B&B)

```
Input: Partial solution P, The lines L that can be added to P
Output: Matrix with minimal score
begin
    if L is empty then
        // P is complete
        if score of P < score of best solution then
            Set the best to solution to P
    else
        for each line in L do
            Add the line at the bottom of P
            if Lower bound of P < score of best solution then
                Update score of P incrementally
                Recursive call B&B(M, P, L\{line})
```

Update of the error For the following, we assume the use of a 3×3 cross kernel. Adding a line at the bottom of a partial solution P_x (which has x lines) with error e_x gives another partial solution P_{x+1} and its error e_{x+1}. Note that e_x can only take into account the first $x-1$ lines, since the error related to the x^{th} line requires the $x+1^{th}$ line which is unknown. The error e_{x+1} is computed from the error e_x as follows: $e_{x+1} = e_x +$ error due to line x. Initially, $e_0 = 0$.

Two special cases arise:

1. e_1 cannot be updated, because at least 2 lines are needed to compute a partial score. Thus $e_1 = e_0$. For e_2, the padding is used to compute the error due to line 1.

2. When all lines have been added to the partial solution with m rows, $e_m = e_{m-1} +$ error due to line $m-1 +$ error due to line m. The error due to line m can be computed using the padding, and the error becomes complete.

For a $k_m \times k_n$ kernel, we must have chosen $\lfloor \frac{k_m}{2} \rfloor$ rows above and below a row to be able to compute its error. We can generalize the two special cases this way:

1. e_i with $i < \lfloor \frac{k_m}{2} \rfloor$ cannot be updated. Indeed at least $\lfloor \frac{k_m}{2} \rfloor + 1$ lines are needed to compute a partial score, the row itself and the $\lfloor \frac{k_m}{2} \rfloor$ rows below it. Thus $e_i = e_0, \forall i < \lfloor \frac{k_m}{2} \rfloor$. For e_i with $i > \lfloor \frac{k_m}{2} \rfloor$, the padding is used to compute the error due to the $\lfloor \frac{k_m}{2} \rfloor$ first lines.

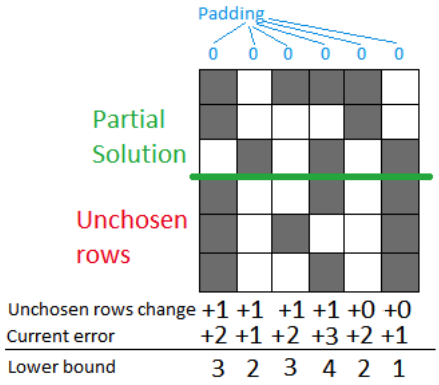


Fig. 5.8 Computation of the fast lower bound

2. When all lines have been added to the partial solution with m rows, $e_m = e_{m-1} +$ error due to $\lfloor \frac{k_m}{2} \rfloor + 1$ last lines. The error due to line m can be computed using the padding, and the error becomes complete.

Lower bounds We have two different lower bounds, one that is good and fast to calculate and one that is better but more expensive to compute. They are obtained by relaxing some constraints of the search.

Fast lower bound By allowing to change the order of elements within a column independently of all other columns, the number of transitions in that column is equal to the number of transitions present in the partial solution (the rows that are fixed):

- $+0$ if all other elements yet unchosen of that column are zeros (resp. ones) and the bottom element of the partial solution is zero (resp. one).
- $+1$ otherwise

The lower bound is obtained by adding that number for each column (example on figure 5.8).

For a bigger kernel, the previous explanation can be generalized by ordering the elements within each column independently of all the others. This means that for each column, all the ones will be put first and then all the zeros. Then the error for each line that still has to be chosen is computed with this particular configuration.

Slow lower bound For the following, we assume a 3×3 cross kernel. For each line l that is yet unassigned, select the two lines that are the closest to l from the set of lines that have not yet been assigned to the partial solution union the last line of the partial solution. Lines can be chosen multiple times. For each line l and its two closest lines, compute the error for l considering that l is between those two lines (the order does not matter if the kernel is row symmetric). Two lines are "close" if the number of values by which they differ is low (Hamming distance: equation 5.9).

For a $k_m \times k_n$ kernel, the previous explanation can be generalized. In section 3.3, we explained that when using a cross kernel and an absolute error, the error for row r can be computed as the sum of the number of transitions between r and each of the $\frac{k_m-1}{2}$ rows above and below it multiplied by its weight in the kernel. We define $center_r = \frac{k_m-1}{2}$, $center_c = \frac{k_n-1}{2}$ and $trans$ as a function taking two rows and returning the number of transitions between them. We get the following formula for a row symmetric kernel K:

$$error(r) = \sum_{i=1}^{\frac{k_m-1}{2}} K_{center_r-i,center_c} * (trans(M_{r,.}, M_{r-i,.}) + trans(M_{r,.}, M_{r+i,.})) \tag{5.13}$$

Therefore the overall lower bound can be found by computing the sum of the lower bounds of this error for every row. We notice that if $K_{center_r-i,center_c}$ decreases or stays constant as i increases, the rows with the fewest number of transitions should be put the closest to r (as the numbers of transitions are always positive).

We now describe how this can be done in practice. In the following, we call A the unassigned rows and P the $\frac{k_m-1}{2}$ last rows of the partial solution. We also call S their union. For every row r for which the error has not been computed yet (i.e. the rows in S), we consider two different cases:

- $r \in P$: the row is in the partial solution. We call end the index of the last row in the partial solution. If the index of r is r_i, we know that all the rows above r are already assigned as well as the $end - r_i$ rows below it. Therefore we find the $\frac{k_m-1}{2} - (end - r_i)$ rows that are the closest to r in $A \setminus r$ and we put them after the row at index end. Finally we compute the error for r with this configuration.

- $r \in A$: first we order $A \setminus r$ by increasing distance to r and we call this new order $A_{ordered}$. Then, we find value $j \in \{0, \cdots, \frac{k_m-1}{2}\}$ such that the error of the following configuration is as small as possible:

  - Put the $2j$ first rows in $A_{ordered}$ around r in order of increasing Hamming distance to r (j rows above and j below).
  - Then put the $\frac{k_m-1}{2} - j$ following rows in $A_{ordered}$ below those j lower rows.
  - Finally, put the $\frac{k_m-1}{2} - j$ last rows of the partial solution above the j upper rows (use the padding if there are not enough rows in the partial solution).

  Note that a worse lower bound can be obtained faster by just taking the $k_m - 1$ rows that are the closest to r in $S \setminus r$. This relaxes the constraint that the rows in P are already ordered.

A drawback of using those lower bounds is that the pruning occurs deeper in the search tree because a partial solution with enough rows is needed to get a good lower bound.

In the next section, we assess our algorithm and compare it with other algorithms.

Chapter 6

Experiments

In this last section, we assess the performances of our algorithm and study the results produced on several datasets. We also compare our `Local Search` algorithm to existing algorithms. We also study the influence of several parameters of our method such as the kernel or the error function. Note that on the figures, all the datasets look rectangular because they had to be squeezed to fit on one page.

6.1 Description of the datasets

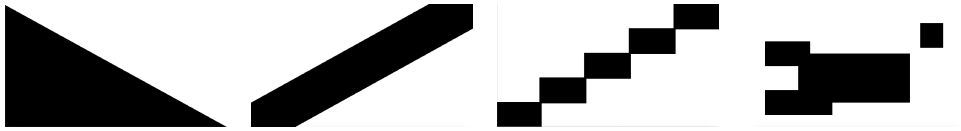
The datasets that we use can be divided into two categories: binary or categorical. Each category can be further split into two categories: artificial and real data. We use artificial datasets because we know the expected structure for them and therefore we can compare the results better. We can also confirm that our method finds what we expect. Moreover, some of the real datasets have labels that can be used for validation. Those datasets are presented below.

6.1.1 Binary datasets

Artificial datasets

The four artificial datasets that we study are shown in figure 6.1. They all have 300 rows and 300 columns. Each of them has a certain pattern that could be interesting to discover in a dataset.

- **Nested** has a nested structure. More precisely, this is a lower triangular matrix (Fig. 6.1a).



(a) nested (b) Banded (c) Blocks (d) Banded Blocks

Fig. 6.1 Artificial binary datasets

- **Banded** has a banded structure. The matrix has ones for every position (r,c) such that $|r-c| \leq 60$ (Fig. 6.1b).
- **Blocks** is a set of four blocks placed such that three of them overlap (Fig. 6.1c).
- **Banded Blocks** is a set of blocks of size 60 that form a banded structure (Fig. 6.1d).

The amount of noise can be changed to make the pattern more or less obvious. If we want to add x percents of noise, we flip every bit in the matrix with a probability of x.

Real datasets

We study five real datasets of different sizes.

Mammals has been introduced in [20] and represents the location of European mammals. This dataset is composed of 2200 locations (rows) of size 40km x 40km and of 124 mammal species (columns). The entry (r,c) is a one if the mammal r has been observed on location c. The paper [14] shows that this dataset has a nested structure.

Gene expression has been introduced in [16]. This dataset consists in 2112 genes (rows) and 363 tumour samples (columns) for the breast cancer. The original data is numerical. Therefore the numerical data has been ranked and categorized to obtain binary data.

Football [1] consists in the adjacency matrix of the network of the 115 American football games between division IA colleges during regular season Fall 2000 studied in [10]. This dataset contains labels that represent 12 football associations.

[1] Available from https://networkdata.ics.uci.edu/data.php?id=5

Mushroom binary [2] is a dataset which describes types of mushrooms and whether they are edible or poisonous. This dataset has been introduced in [25]. There are 8124 mushrooms (rows) described by 22 attributes that can take several values. We transformed this categorical dataset into a binary one by creating one column for each pair (attribute, value). Each column, which represents a pair (a, v), contains a one for mushroom r if that mushroom has value v for attribute a. This is a one hot encoding of each value of each attribute. Each mushroom has a single value for each attribute. The result of this processing gives a 8124×112 matrix. Each mushroom is given a label: e for edible and p for poisonous.

UCL dataset This dataset describes the set of courses that students followed during their stay at the university. Each row represents a student, and each column a course. If a student followed a specific course, he/she will have a one in the corresponding column. Each student is also labeled with its major and minor. Each major is given a label that is displayed as a color on the side of the dataset. This dataset has 4187 rows and 108 columns.

Paleo dataset [3] is a dataset where the rows represent locations and the columns species. It has 1597 rows and 4215 columns. A black pixel at (r, c) signifies that species c was found at location r.

6.1.2 Categorical datasets

Artificial datasets

The artificial categorical dataset is composed of 300 rows and 300 columns and six colors (Fig. 6.2). There is a 100×120 rectangle inside the matrix which is split into six rectangles (one for each color) of sizes 100×20. The other points are assigned one of the six colors at random. The objective is to obtain a rectangle of size 100×120 of uniform color.

Real datasets

We study three categorical datasets:

[2] Available from https://archive.ics.uci.edu/ml/datasets/mushroom

[3] Available from http://www.helsinki.fi/science/now/excel/NOW_public_030717.zip

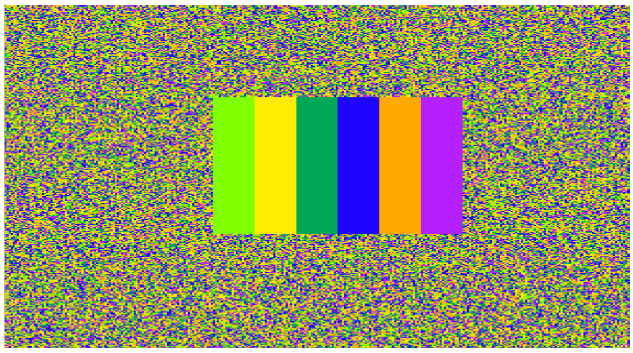


Fig. 6.2 Categorical artificial dataset

Mushroom categorical [4] is identical to mushroom binary but has not been made binary. This dataset has 22 columns (one for each attribute) and each value of an attribute is mapped to one of the 12 colors. This dataset also has labels to point out which mushrooms are edible or poisonous.

news_3cl_1 [5] is the adjacency matrix of a weighted graph of 600 newsgroup documents described in [5]. The larger the number of words in common between two documents, the higher the weight of the edge between them in the graph. Moreover, each node is given a label and nodes with similar labels should have a large number of words in common. The label indicates to which class this document belongs. This matrix has been multiplied by its transposed and then normalized. Finally we have discretized the values in four categories: [0, 0.05[, [0.05, 0.1[, [0.1, 0.25[and [0.25, 1[.

news_5cl_1 [5] has the same structure and interpretation as news_3cl_1 but instead represents 998 newsgroup documents. This dataset has also been preprocessed in the same way.

6.2 Parameters of the `Local Search`

In this section we briefly describe the choices that we made for the `Local Search` ordering presented in chapter 5.

- **Moves** We use all the moves described in chapter 5. We have modified slightly the swap move to help obtain better results. Instead of choosing two random rows, we first select a random row and then choose another

[4] Available from https://archive.ics.uci.edu/ml/datasets/mushroom
[5] Available from http://qwone.com/~jason/20Newsgroups/

row among its 20 nearest neighbors. This is based on the observation that swap of rows that are close to each other are usually more useful.

- **Learning of the moves** We let the `Local Search` learn which moves are the best and changes the probability of selecting each move itself as described in 5.2.7. The segment length and reaction factor introduced in this section are respectively set to 500 and 0.6.
- **Maximal number of iterations** We have set this number to 100000.
- **Automatic stop** We let the algorithm stop automatically when the solution does not improve enough for a certain number of iteration. More precisely, after each segment length (i.e. 500 iterations) we stop if the decrease of the error divided by the segment length is greater than some threshold that we set to -0.0001. In other words, every time we have performed 500 iterations, we check if the error has decreased enough, otherwise we stop. This way, the search stops automatically once the solution reaches an asymptote and we do not have to worry about the maximum number of iterations to allow.
- **Initialization** If not mentioned otherwise, we use the `TSP` ordering as a preprocessing step.

For the block ordering described in 5.3, we cut the matrix into 10 blocks. Then we use a branch and bound with a 3×3 Gaussian Blur Smooth Cross kernel. The lower bound that we use is the slow but efficient one.

6.3 Assessment of the solutions

The different solutions that we obtain are assessed and compared in three different ways:

- **Qualitative assessment** How nice the image looks. This way of assess results is quite subjective but already gives a good idea of the quality of the obtained image. Images can be roughly compared.
- **Assessment based on error** The error of solutions obtained with different algorithms or different settings are compared. This can only be used to compare results between each other as the error does not give information about how far from the optimal value we are (except if we know it beforehand).

- **Assessment based on accuracy** For labeled datasets, we can estimate the accuracy obtained with a k-nearest neighbor classification. More precisely, we use a stratified cross validation with 10 splits. This ensures that the proportion between the classes is kept as much as possible in the train and test sets. First we run our algorithm on the whole data without the labels. Then for each split, we classify the test samples. We assign to each unlabeled test row the most frequent label among the 10 nearest labeled rows in the matrix (i.e. neighbors that are in the train set). For the football dataset we use the 5 nearest neighbors because this dataset is quite small. Finally we compute the mean of the accuracies obtained on the different test sets.

  For datasets that represent adjacency matrices (football and the news datasets), this procedure can be applied to both the rows and the columns. Therefore both accuracies are displayed in the results.

6.4 Questions

In the following sections, we answer a set of questions to see how our algorithm performs in different conditions.

We first discuss the tuning of the parameters of the search, namely the type of the kernel, its size and the error function. The effect of these parameters are linked to each other. Indeed, the choice of the type of kernel influences the results when we change its size, and the same way around. We also discuss the impact that the size of the kernel has on the running time.

We then examine how the heuristics influence the result of `Local Search`. These heuristics include the choice of the moves for the `Local Search` as well as learning which moves are the best dynamically. We also look at the effect of choosing different initial solutions.

After that, we discuss how the dataset itself can influence the result, namely the effect of noise as well as the difference of width/height in matrices.

At that point, we compare our algorithm to other algorithms on quality and on run time.

Finally, we look at the results we obtain on real datasets.

These are the questions we are answering in the upcoming sections:

- *Tuning of the parameters* (Section 6.5.1)
  - Q1: What is the quality impact of the types of kernel?
  - Q2: What is the quality impact of the size of the kernel?
  - Q3: What is the quality impact of the error function?
  - Q4: What is the running time impact of the size of the kernel?
- *Influence of heuristics* (Section 6.5.2)
  - Q5: What is the impact of the heuristics for the `Local Search`?
  - Q6: What is the impact of learning which of the different moves are the best?
  - Q7: What is the impact of changing the initial solution?
- *Influence of the dataset* (Section 6.5.3)
  - Q8: What is the impact of noise?
  - Q9: How does the `Local Search` scale for the difference of width/height in matrices?
- *Comparison with other algorithms* (Section 6.5.4)
  - Q10: What is the quality of the other algorithms?
  - Q11: What is the run time of the other algorithms?
- *Real datasets* (Section 6.5.5)
  - Q12: What are the results on real data?

6.5 Results

This section presents the results of the algorithms on different datasets by answering the questions of section 6.4. The code used to generate them is available at https://github.com/GLeurquin/Faithful-visualization-of-categorical-datasets/ under the MIT license.

Note that all the experiments have been produced by two different laptops. They were not run in totally clean environments as there might have been other processes running at the same time, however the run times should still give a good idea of how fast an algorithm is. The specifications of the machines used are given below:

- I7-3632QM 2.2 GHz with turboboost to 3.2 GHz and 16GB DDR3 ram
- I5-520M 2.4 GHz with turboboost to 2.93 GHz and 8GB DDR3 ram

All images are scaled down to 300 rows and 300 columns (or less if they originally have less rows or columns) using the summarizing techniques described in section 3.5.

In the tables, <kernel type>(S) means that the kernel is of size $S \times S$

6.5.1 Tuning of the parameters

Q1: What is the quality impact of the types of kernel?

We run our parallelized algorithm on the *nested* and *banded* datasets with three different types of kernels: Gaussian Blur Smooth, Gaussian Blur Exponential and Gaussian Blur Smooth Cross, as described in section 3.2.1. We choose a kernel size of 49×49. Indeed, question 2 will show that bigger kernels give results that better bring out the pattern even if they are slower. We use the absolute value for the error and not the square, as question 3 will show that the absolute error gives better results.

In table A.1, we observe that the `Parallelized Local Search` with a Gaussian Blur Smooth kernel is the best to bring out the full pattern. The two other kernels only succeed to bring out parts of the patterns.

This observation can also be seen by looking at the errors. The original dataset is the one where the pattern is the most obvious, therefore this matrix should have the smallest error. For the GBS kernel, this is true for both *banded* and *nested*. However, for the two other kernels the errors obtained after the `Local Search` are sometimes lower than the error of the original dataset. This lower error does not however reflect the quality of the images, which are poorer than the original. A good kernel type should give lower errors to images of better quality (i.e. where the pattern is the most visible).

A possible explanation for why the GBS is better is that its weights decrease more slowly with the distance. Therefore, neighbors that are further away are given more importance than with the two other kernels (i.e. because they have higher weights). When using a 49×49 GBE kernel, distant neighbors almost do not play a role while computing the convolution of a point, because of the exponential decay in the weights. In other words, the GBS kernel is the one

that gives the most information about what is around each point.

Moreover, GBS is also the kernel that deals the best with the noise. Indeed, the smoother the kernel, the more every point will influence the convolution. The impact of the noisy points is therefore less important. A last remark is that the cross kernel is the one providing the least information about neighboring points because any value that is not on the middle row/column is equal to zero.

In the rest of the experiments we will therefore use the GBS kernel.

Q2: What is the quality impact of the size of the kernel?

We study the impact of the size of the kernel on the result. We run the `Parallelized Local Search` with kernels of five different sizes: $3\times3, 11\times11, 11\times3, 25\times25, 49\times49$. We study these kernels on *nested* and *banded*.

As can be seen on table A.2, increasing the size of the kernel gives better results. This observation can be done from the images but also from the errors. Indeed 3×3 kernels give lower errors to the matrix after the `Local Search` even if the pattern is less visible. We notice that the 11×3 kernel does not give very good results.

As mentioned before, using a bigger kernel gives more information about the distant neighbors of a point and helps deal with the noise. The more points we use in the convolution, the smaller the impact of noisy points.

We will use big kernels in the rest of the experiments.

Q3: What is the quality impact of the error function?

Another important parameter is the function that we use to compute the error of each point. That function can either be the absolute value or quadratic, as described in section 3.3. We try those two error functions on *nested* and *banded*.

In table A.3, we notice that the quadratic error gives worse results for the *nested* dataset than the absolute value error. The former favors having many small errors over a few big ones. However, the results show that using the quadratic error is not a good idea to bring out patterns. Therefore we only use the absolute value error in the following experiments.

Q4: What is the run time impact of the size of the kernel?

The size of the kernel has an impact on the run time. We experiment on all the artificial binary data and categorical real data with kernels of sizes 3×3, 11×11, 11×3 and 25×25, 29×5. We use the `Parallelized Local Search`.

From tables A.2, A.10 and A.11, we conclude that increasing the size of the kernel increases the execution time. Indeed, the number of operations needed to compute the convolution of each point increases quadratically with the size of the kernel (for square kernels). As an example, computing the convolution of one point for a 25×25 kernel requires 625 operations. For a 49×49, this requires 2401 operations. This is almost four times more operations.

Moreover, if we increase the size of the kernel, more convolutions must be recomputed after a move because it affects more points. For categorical data, the execution time also increases with the number of colors as we have to compute one convolution for each color.

The run times are a few seconds for a 3×3, less than a hour for 11×11, 11×3 and 25×25 kernels and almost 7 hours for a 49×49 kernel.

6.5.2 Influence of heuristics

Q5: What is the impact of the heuristics for the `Local Search`?

In section 5.2 we described several moves that can be used in the `Local Search`. In this experiment, we study the effect of each move to determine if some are better. To achieve that, we run `Local Search` once for each move, using only one of the five moves. We test our algorithm on the *nested* and *banded* datasets. We do not preprocess the dataset with a `TSP` ordering in order to make the impact more visible. We also show the result of only running `TSP` in order to observe its performances on its own.

A first observation from table A.4 is that no move succeeds to reorder the dataset perfectly on its own. That is why we have to use a combination of several moves to get good results. All the moves are useful because they all improve the solution. We can rank them by increasing error: `Swap Rectangles`, `Relocate` and `TSP`, `Reverse`, `Swap` and `Swap Adjacent`.

Swap Rectangles gives the best results because this move is more sophisticated and less random than the others. On the other hand, it takes more time to execute because we have to find rectangles in the matrix at each iteration.

`TSP` **ordering** is a good preprocessing move because of its speed and effectiveness.

Relocate gives good results, especially for *nested*. This move is similar to `Swap Rectangles` except that it chooses the rows at random instead of trying to find a rectangle. This move is also slower because it usually affects more rows. We notice that those moves affect the matrix globally.

Reverse, Swap and Swap Adjacent are very good at reordering the data locally. Indeed they were designed for this purpose as they only change the order of adjacent rows and columns. We notice that for *nested* they find many small diamonds and for *banded* many small banded structures. However, `Reverse` finds larger patterns because this move affects more rows and is less local than the two others. This makes the latter slower, since the number of points for which the convolution has to be updated increases with number of rows/columns that are affected by a move.

The execution times can be ordered in this way: `Swap`, `Swap Adjacent`, `Reverse`, `Swap Rectangles` and `Relocate`. This ordering is strongly correlated with the number of rows/columns affected by the moves. This is also roughly the reverse of the ordering for the quality of the moves. We conclude that there is a trade-off between quality of the move and execution time.

In table A.5, we observe that using all the moves (with or without learning the moves) gives a better solution as expected but takes slightly more time than the slowest move. This is due to the fact that the method can keep improving the result longer.

Q6: What is the impact of learning which of the different moves are the best?

As described in section 5.2.7 we can make the `Local Search` algorithm learn by itself which moves perform well in order to choose them with a higher probability. The goal of this experiment is therefore to study if updating the

probabilities has an impact on the result or on the execution time. We run our algorithm without TSP preprocessing on *nested* and *banded*.

On table A.5, we observe that learning which moves are the best does not improve the solution significantly. For *nested*, this is even the contrary as the search returns a worse solution than without learning the moves. However, the difference between the errors is small.

On the other hand, learning which moves are the best makes the algorithm faster. This can be explained by the following two (linked) reasons:

1. The search selects moves that are more likely to improve the solution and thus loses less time trying moves that are useless at this stage of the search.

2. If we mainly select moves that improve the solution, we do not have to spend as much time reversing moves that do not improve the solution.

Q7: What is the impact of changing the initial solution?

The `Local Search` algorithm must be given an initial solution. In this section, we study the effect of using different algorithms to generate that initial solution.

Table A.11 presents the results obtained by the `Parallelized Local Search` when the results of `HeatMap`, `Bidirectional` and `TSP` ordering are used as initial solution. For *mammals* we observe that `HeatMap` is the best initial solution for the `Local Search`, even if it has the higher initial error than `TSP`. However the `Bidirectional` ordering followed by a `Local Search` seems to better reveal the nested structure. For *mushrooms*, the `Bidirectional` ordering gives the initial solution that will lead to the best result. Finally, for *UCL* we see that the best initial solution is given by `TSP`.

We conclude that for real datasets, the quality of the solution obtained after the `Parallelized Local Search` depends on the algorithm used as preprocessing. Moreover, no single algorithm performs the best for all datasets. Hence the algorithm that should be used depends on the dataset. We also notice that initial solutions with bigger errors can lead to better solutions at the end of the search.

Table A.6 shows results obtain on artificial datasets. The main conclusion that we can draw from this is that most algorithms generate initial solutions

that will lead to good results. However the `Nested` ordering does not yield a good initial solution for the *banded* dataset, which makes sense as `Nested` is designed to find nested structures. The `Bidirectional` ordering seems to give the best initial solution for those types of datasets.

6.5.3 Influence of the dataset

Q8: What is the impact of noise?

As mentioned earlier, we can influence the percentage of noise in the artificial datasets. Knowing how our algorithm performs when the noise increases is valuable as this better reflects real datasets. We try three different levels of noise: 0%, 10% and 25%. We run our parallelized algorithm on *nested* and *banded*.

From tables A.7, A.8 and A.9, we observe that the `Local Search` algorithm gives a solution very close to the optimal solution for all three levels of noise, at the cost of a high run time. From these experiments, we conclude that our algorithm fares well against different levels of noise.

Q9: How does the `Local Search` scale for the difference of width/height in matrices?

Some real datasets are have a much higher number of rows (resp. columns) than columns (resp. rows). Therefore we study if our algorithm still performs well on them. The *mushrooms* dataset has 8124 rows and only 22 columns and is therefore a good candidate to experiment on. We also study this effect on *mammals* and *gene expression*.

Table A.11 shows the results obtained for the *mammals* dataset. Even if the dataset has a large ratio rows/columns (18) the `TSP` ordering still finds a good solution. However, the `Local Search` is only able to improve this solution slightly when using a 25×25 kernel. We also tried with a 29×5 kernel because as the dataset only has a few columns, our intuition was that we could take less columns into account when computing the convolution and obtain similar results. However, the results show that this is not the case. With this kernel, the `Local Search` algorithm is not able to improve the solution at all.

The results are more convincing for the *gene expression* dataset in table A.10. This dataset has a ratio rows/columns of 6. Here we see that the `Local search` improved the solution even after the `TSP` ordering.

Finally, the conclusions are very similar for the *mushroom* categorical dataset in table A.12 with a rows/columns ratio of 369. The `Local Search` could improve the results only with a 3×3 kernel. From those experiments, it seems like the `TSP` ordering works well even on datasets with a large rows/-columns ratio. However, the `Local Search` is mainly efficient when this ratio is not too high or that at least the kernel used is square.

6.5.4 Comparison with other algorithms

Q10: What is the quality of the different algorithms?

From table A.7, we observe that provided that the structure of the matrix is known before hand, the selection of the appropriate algorithm gives very good results in a very small amount of time for datasets with no noise. Our algorithm takes some time even if the solution is not improved compared to the `TSP` ordering. This is because we have to wait for some time to ensure that the solution cannot be improved. From these experiments, we observe that the `Nested` algorithm only works on nested matrices (as expected), where as the `Barycentric` and `Bidirectional` algorithms can find both a banded and nested structure.

In tables A.8 and A.9, we observe that increasing the noise makes the solution worse for the `Barycentric` algorithm. The `HeatMap` ordering does not give good results on artificial datasets as this method does not bring out the patterns we are trying to find. The `Bidirectional` ordering deals well with the noise, especially for the *nested* dataset. For the *banded* dataset, this method finds an almost banded structure but the result is not perfect. The `TSP` ordering gives poor results when the noise increases, because the distances between the rows and columns changes. Finally, our `Local Search` deals very well with the noise as this method perfectly brings out the patterns we are looking for. We conclude that our algorithm is useful because it is the only one that always finds the perfect solution.

In table A.12, we observe that our method finds a big rectangle. This means that the rows and columns were reordered and the colors changed as expected. The `HeatMap` ordering fails to find this pattern.

Q11: What is the run time of the different algorithms?

Earlier we studied the run time of our `Local Search` algorithm. Now we compare the run times of all the algorithms presented before.

On table A.9 we observe that `Nested` ordering is very fast (i.e. only a few milliseconds). As mentioned before, this is because this is a very simple algorithm. The second fastest method is the `Barycentric` ordering that only takes a few seconds. This method is followed by the `Bidirectional` ordering that has a slightly longer execution time. The `TSP` ordering is also efficient as this method gives similar run times on the artificial data.

We now compare those run times with the ones we obtained for the different sizes of kernel in A.2. From this table, we conclude that our method is at least twice as slow even for the smallest kernel. For bigger kernels, the run time increases and becomes much bigger than for the other methods.

The last step is to study how those algorithms scale to bigger datasets. To achieve that, we use only real binary data. In tables A.10 and A.11 we confirm that the `Nested` ordering is the fastest and scales very well. The same applies to the `Barycentric` ordering that also scales well and have run times of a few seconds. The results for the `Bidirectional` ordering are convincing even if its run times are much longer than the one of the `Barycentric` ordering. The former takes a few minutes on most big datasets. Finally, our `Local Search` usually takes between one and four hours for a 25×25 kernel.

6.5.5 Real datasets

Q12: What are the results on real data?

This last experiment consists in running all the algorithms on all the real datasets. For our `Local Search`, we choose a Gaussian Blur Smooth kernel of size $25x25$ (for binary) and $21x21$ for the categorical datasets since these sizes have a good trade-off between quality and execution time.

The results are displayed in tables A.10, A.11, A.12 and A.13.

Mammals For this dataset, the `HeatMap` ordering does not give conclusive results as it does not bring out any pattern and the points are not well clustered.

`Nested`, against intuition, does not perform well as the algorithm does not take noise into account. `Barycentric` does not work well because *mammals* has a nested structure and that `Barycentric` tries to find a banded one. The structures they find do not have a high density of black points and therefore do not convey much information.

The `Bidirectional` ordering yields better results as the nested structure becomes more visible and has a higher density of black points. However, the image is still blurry due to the transitions between black and white points.

The `TSP` ordering does a very good job at gathering black points together. The nested structure is less visible than with the `Bidirectional` ordering but we can still guess it.

Our `Local Search` ordering does not improve the solution much. For the 25×25 kernel this search only improves the solution slightly. The nested pattern is a bit less visible than after the `TSP` but it looks like some black points have been put closer to each other.

We conclude that this dataset has a decent nested structure. The information we can get from those results is that there is some kind of hierarchy between locations depending on the number of different species that live there.

Mushrooms binary This dataset is binary but we added one column on the right to display the label associated to each mushroom. The rows with a green label correspond to edible mushrooms whereas the ones with a red color correspond to poisonous mushrooms.

The `HeatMap` ordering is not very good on this dataset as the points could be clustered better. However, this method helps separates the labels well and gives a high accuracy.

The `Nested` and `Barycentric` orderings do not give good results as they do not bring out any structure. Moreover the labels are still mixed after applying them, which results in low accuracies.

The `Bidirectional` ordering finds a decent banded patterns and separates the edible and poisonous mushrooms better, hence the accuracy is very good.

The `TSP` ordering separates the edible and poisonous mushrooms quite well. This means that this is possible to classify the mushrooms based on their characteristics, as confirmed by the high accuracy.

The `Local Search` with a 25×25 kernel further improves the objective function. We see that some columns contain many ones. As a column corresponds to a pair (attribute, value), this observation means that most mushrooms have the same value for those attributes.

Gene expression The `HeatMap` ordering yields a good image as points are more clustered than in the original data. This method also gives one of the smallest error.

Once again we notice that the `Nested` and `Barycentric` orderings yield poor results.

The `Bidirectional` ordering finds a band that has a higher density of black points than the rest of the image. However, points are not very well separated as this algorithm mainly focuses on finding a banded pattern.

The `TSP` ordering yields an image that is less blurry than the others. We notice some clusters of black points but the data is not very well ordered yet.

Finally, our `Local Search` improves the result by gathering the white and blacks points even more. Now we notice a large rectangle with a high density of black points as well as several smaller rectangles with high density of blacks or white points. The large rectangle means that this group of tumorous samples (columns) have all those genes in common.

Football The column to the right and the upper row represent the association to which each team belongs. There are 12 associations.

The `HeatMap` ordering yields a very clear banded structure composed of several blocks. We observe that most of those blocks can be mapped to a

football association.

The Nested and Barycentric orderings do not give convincing results.

The Bidirectional ordering finds a banded structure composed of many small rectangles nested. They mostly correspond to the different associations.

The TSP ordering also finds several rectangles but does not succeed to put them in a banded structure. They also mostly correspond to the 12 different associations of sports teams that play each other. This makes sense because those teams are more likely to play against each other. However this is not exactly true because not all associations are represented by a rectangle in the matrix. This might be due to the fact that they almost did not play. We notice that the associations can be retrieved by looking for blocks of consecutive ones in the rows or in the columns.

The Local Search does not improve the image.

UCL The column to the right represents the main option (major) of each student.

The HeatMap ordering clusters the points very well. The result is a clear image with big clusters of points. This method is also one of the best (among the ones we tried) for classification.

The Nested ordering does not yield a good image and does not separate the students well.

The Barycentric ordering performs better, even though the band does not have a very high density. This method still manages to find some clusters of students.

The image can be separated in two parts. The lower part corresponds to the lectures that are common to many options and are followed by many students. The upper part is composed of the lectures specific to some options. We can distinguish some clusters of points that correspond to all the students with a specific option. This is easily seen in the last column where we can see rectangles of the same color.

The `Bidirectional` ordering finds a banded structure with a higher density. We observe that the separation in different options is even better.

The `TSP` ordering does not bring out a pattern in the image but does a decent job at clustering the students. We observe that most clusters in the image can be mapped to a rectangle of uniform color in the most right column (i.e. to a main option).

The `Local Search` ordering seems to cluster the points even more but this does not result in a better separation of the options.

Mushroom Categorical The column to the right indicates whether the mushroom is edible or poisonous. A green label indicates that the corresponding mushroom is edible whereas a red label means that it is poisonous.

The `HeatMap` ordering does not cluster the points well. Moreover, the mushrooms are not well separated. Furthermore, this method does not change the color mapping, and thus the patterns are less visible.

As for the binary mushroom dataset, we observe that some attributes have the same value for most mushrooms. They corresponds to the big and high rectangles. The other rectangles correspond to group of mushrooms that have the same values for some attributes.

We notice that the `TSP` ordering separates the edible and poisonous mushrooms well and that there are less transitions between colors than in the original dataset. This result could be used to form groups of mushrooms based on the value of some of their attributes.

News_3cl_1 The upper row and the column to the right represent the labels associated to each document. The levels of similarity are displayed with a gradient: the lighter the color, the higher the similarity between those two documents.

In the original image, we notice that blue documents are very similar between each other (i.e. this region of the image is very bright). We also observe that this is not the case for the green documents. Therefore they will likely be

more difficult to classify.

The TSP ordering mainly finds two large bright areas. They correspond to the blue documents and to a fraction of the gray documents. We observe that the algorithm almost perfectly separates the blue documents from the others. This confirms our previous observation that blue documents are more easily distinguishable.

We can classify the documents by determining which documents are associated to the rows or to the columns of the bright zones. Here we notice that looking at the columns is better to distinguish them. The first third of the columns is dark and corresponds to green points. The second third has a lighter zones. We see that its columns mainly correspond to gray documents. The last third has also a bright zone that corresponds to blue documents. Hence cutting the columns into three blocks and mapping them to the associated documents would allow us to roughly retrieve the different groups.

Running the `Local Search` helps improve the solution slightly. However the TSP preprocessing has already done much of the job. We notice that using a larger kernel makes the solution look better.

News_5cl_1 The results for this dataset are a bit worse because there are more labels and the bright regions are less visible, even in the original matrix. This means that documents of one class are also similar to documents of other classes. The documents of the blue class have a very low level of similarity.

We observe that the TSP ordering almost perfectly separates the documents of the cyan and green classes. This makes sense since green documents are very similar to each other.

The `Local Search` helps improve the solution by clustering the points even more. Once again we also notice that using a bigger kernel gives better results. A last observation is that the run times are much longer than for `News_3cl_1`, because the matrix is almost three times bigger.

Chapter 7

Conclusion

7.1 Future work

In this section, we briefly present some improvements or extensions that could be made to our algorithm.

GPU usage In the previous section, we noticed that our algorithm yields good results but is quite slow compared to the others. This is due to the huge amount of time needed to compute the convolution for big kernels. A possibility to improve this would be to compute the convolution more efficiently by using GPUs or by using the convolution of neighbors of a point. Indeed, the convolution of two adjacent points should be similar if the kernel is big enough because only a few values will change. Instead of computing the exact convolution for the matrix, we could also try to find approximations that would be faster to compute.

Parallelization A second possibility to improve the run time would be to use even more parallelization. This could be done by using GPUs or even maybe a cluster of computers that would be used to compute the convolutions of the points in parallel. Another way to increase the parallelization would be to try several moves in parallel and then keep the one improving the solution.

Automatic kernel tuning In this thesis we have defined a few kernels that we can use in the objective function. However it might be interesting to learn new kernels with Deep Learning methods. As a example, we could present several images obtained with different kernels to a user and let him decide which image he prefers. The kernels could then be modified and improved

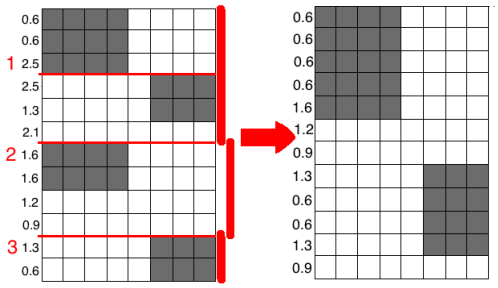


Fig. 7.1 KMax with $k = 3$ with a 3×3 GBS and a ZerosAndOnes padding

based on his feedback. This way, the user could personalize the output and favor certain structures.

Zooming interface Once the algorithm has been made faster, it would be interesting to design an interface that would let the user zoom in and out on the image. The size of the kernel used to display the image would change accordingly. The user might also define constraints on the data (e.g. those two blocks should be put next to each other ...).

Relational database visualization Another application of this work could be used to visualize relational databases. For example, consider a client/product matrix where a black pixel at (r,c) would say that client r bought product c. There could also be a category/product matrix where a black pixel at (r,c) would mean that the product c belongs to category r. This could be extended to categorical matrices as well. These two matrices can be displayed on top of each other and can be linked together, such that reordering the client/product matrix would also impact the columns of the category/product matrix. Such representations can be extended to more matrices in order to form a "puzzle" of matrices linked together. We could then try to optimize all theses matrices at the same time, to find interesting patterns spanning the whole database.

Additional move: KMax We present here a new move that could be used to further improve the results of the `Local Search`. In the following, we assume that a kernel with k_n rows is used. The idea behind this move is that rows with big errors are likely to be delimiters between clusters of ones (Fig. 7.1). Therefore the idea would be to select at most k rows with high error and then to reorder the blocks between those rows.

The rows are selected by repeating the following procedure until we have k rows or until no more rows can be selected.

1. Find row r_{max} in the set of rows with maximal error and add it to the selected rows.

2. Filter the set of rows to remove every row that is within a distance of k_n from r_{max}. This ensure that blocks do not interact with each other.

Figure 7.1 shows an example with $k = 3$. The black numbers represent the error of the rows, the horizontal red lines delimit the different blocks, the red numbers represent the rows that are picked and in which order. The vertical red lines show the rows that are filtered out after each row is picked.

The $(k+1)$ blocks defined by those rows can be reordered by reducing this problem to a `TSP`. One way to do this is to compute a directed distance between each pair of blocks. This distance is the error obtained by putting one of the block on top of the other. In this error, only the $k_n/2$ rows from the borders of each block are taken into account.

Finally, the distance matrix can be given as input to a `TSP` solver. This solver would return a new order for the blocks. Note that as this matrix is likely asymmetric, we have to convert it into a symmetric matrix if the `TSP` solver only works for symmetric distances. This can be done with the method explained in [13].

A limitation of this method is that the order of the rows in a block cannot be changed. It would be interesting to let the algorithm reverse some blocks. This can be done by using $2(k+1)$ nodes. There will be $(k+1)$ nodes for the different blocks and $(k+1)$ nodes for their reverse. We have to ensure that only one of the nodes in this pair will be taken. This can be done by using a generalization of the `TSP` called `Equality Generalized Traveling Salesman Problem` (E-GTSP). Moreover the `E-GTSP` problem can be transformed into an asymmetric `TSP` problem using the method described in [2].

Finally we would iteratively apply this method on the matrix and its transpose until convergence.

Table A.14 shows the results obtained on the artificial datasets with $k = 5$ (without reversing the blocks). We observe that this method only manages to form some clusters of points. However, this algorithm can be further improved by using it in combination with other moves or methods.

7.2 Conclusion

In this master thesis, we have studied the problem of reordering a categorical dataset in order to visualize it faithfully. We began by studying and implementing methods described in the literature. Then we studied the complexity of `ConvoMap` and showed that it is NP-Hard for some simple settings. Based on this observation, we have decided to develop a local search algorithm using the difference between the original image and its convolution as objective function to minimize. We defined several moves to reorder the matrix with the heuristic that the number of transitions should be minimized.

We then assessed our algorithm and compared it to the other existing algorithms on both artificial and read datasets. Our conclusion is that our algorithm yields very good results on datasets that have a clear pattern. Moreover, it also deals well with noise. We also note that using big kernels gives better results, at the expense of run time.

Our algorithm gives results similar to the `Bidirectional` ordering on most real datasets (except on the gene expression data). But unlike the `Bidirectional` ordering, our method deals with both binary and categorical data, which allows us to tackle a wider range of problems.

We conclude that the use of convolution as an optimization criterion to find patterns in matrices works well, as the results give meaningful information that can be used to better understand the data, extract patterns or cluster data.

References

[1] Atkins, J. E., Boman, E. G., and Hendrickson, B. (1998). A spectral algorithm for seriation and the consecutive ones problem. *SIAM Journal on Computing*, 28:297–310.

[2] Behzad, A. and Modarres, M. (2002). A new efficient transformation of the generalized traveling salesman problem into traveling salesman problem. In *Proceedings of the 15th International Conference of Systems Engineering*, pages 6–8.

[3] Christofides, N. (1976). Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, DTIC Document.

[4] Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations research*, 6(6):791–812.

[5] Ding, C. H., He, X., Zha, H., Gu, M., and Simon, H. D. (2001). A min-max cut algorithm for graph partitioning and data clustering. In *Data Mining, 2001. ICDM 2001, Proceedings IEEE International Conference on*, pages 107–114. IEEE.

[6] Edmonds, J., Gryz, J., Liang, D., and Miller, R. J. (2001). Mining for empty rectangles in large data sets. In *International Conference on Database Theory*, pages 174–188. Springer.

[7] Garriga, G. C., Junttila, E., and Mannila, H. (2011). Banded structure in binary matrices. *Knowledge and information systems*, 28(1):197–226.

[8] Gary, M. R. and Johnson, D. S. (1979). Computers and intractability: A guide to the theory of np-completeness.

[9] Geerts, F., Goethals, B., and Mielikäinen, T. (2004). Tiling databases. In *International Conference on Discovery Science*, pages 278–289. Springer.

[10] Girvan, M. and Newman, M. E. J. (2002). Community structure in social and biological networks. *Proc. Natl. Acad. Sci. USA*, 99:7821–7826.

[11] Hahsler, M., Hornik, K., and Buchta, C. (2008). Getting things in order: An introduction to the r package seriation. *Journal of Statistical Software, Articles*, 25(3):1–34.

[12] Hubert, L. (1974). Some applications of graph theory and related non-metric techniques to problems of approximate seriation: The case of symmetric proximity measures. *British Journal of Mathematical and Statistical Psychology*, 27(2):133–153.

[13] Jonker, R. and Volgenant, T. (1983). Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, 2(4):161–163.

[14] Junttila, E. (2011). Patterns in permuted binary matrices. Master's thesis, University of Helsinki.

[15] Junttila, E. and Kaski, P. (2011). Segmented nestedness in binary data. In *Proceedings of the 2011 SIAM International Conference on Data Mining*, pages 235–246. SIAM.

[16] Le Van, T., van Leeuwen, M., Carolina Fierro, A., De Maeyer, D., Van den Eynden, J., Verbeke, L., De Raedt, L., Marchal, K., and Nijssen, S. (2016). Simultaneous discovery of cancer subtypes and subtype features by molecular data integration. *Bioinformatics*, 32(17):i445.

[17] Liiv, I. (2010). Seriation and matrix reordering methods: An historical overview. *Statistical Analysis and Data Mining*, 3(2):70–91.

[18] Mäkinen, E. and Siirtola, H. (2000). Reordering the reorderable matrix as an algorithmic problem. In *International Conference on Theory and Application of Diagrams*, pages 453–468. Springer.

[19] Mäkinen, E. and Siirtola, H. (2005). The barycenter heuristic and the reorderable matrix. *Informatica (Slovenia)*, 29(3):357–364.

[20] Mitchell-Jones, A. J., Mitchell, J., Amori, G., Bogdanowicz, W., Spitzenberger, F., Krystufek, B., Vohralík, V., Thissen, J., Reijnders, P., Ziman, J., et al. (1999). *The atlas of European mammals*, volume 3. Academic Press London.

[21] Mohapatra, A. (2009). Optimal sort ordering in column stores is np-complete. Technical report, Technical report, Stanford University.

[22] Orlowski, M. (1990). A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5(1-4):65–73.

[23] Ropke, S. and Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *TRANSPORTATION SCIENCE*, 40(4):455–472.

[24] Ropke, S. and Pisinger, D. (2010). Large neighborhood search. *Handbook of Metaheuristics (2 ed.)*, pages 399–420.

[25] Schlimmer, J. (1981). Mushroom records drawn from the audubon society field guide to north american mushrooms. *GH Lincoff (Pres), New York*.

Appendix A

Experimental results

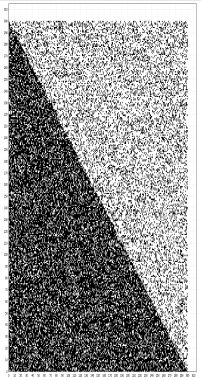
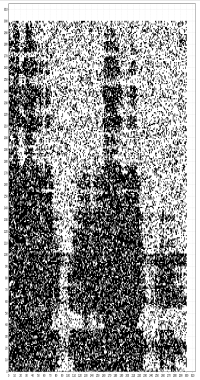
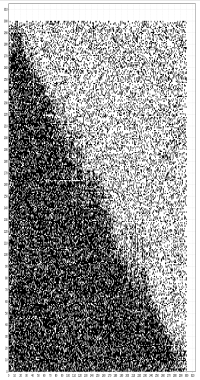
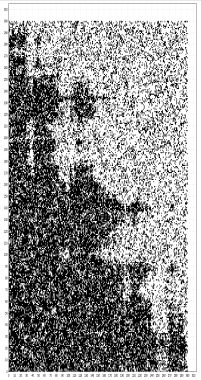
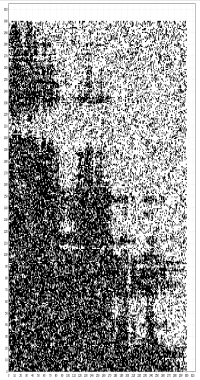
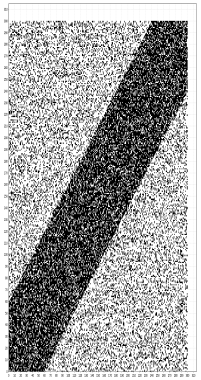
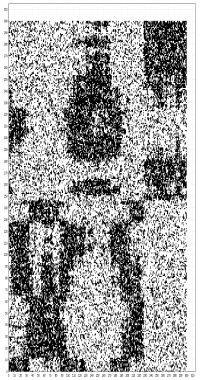
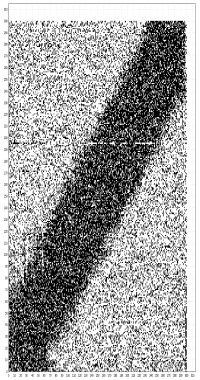
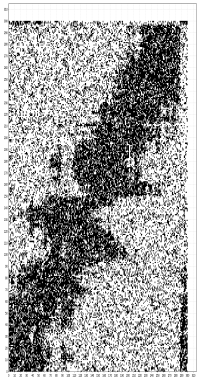
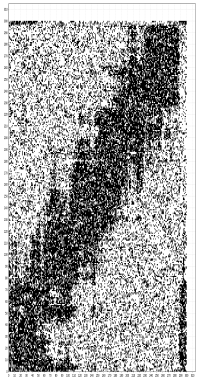
Config	Original	TSP	TSP+P-LS(GBS)	TSP+P-LS(GBE)	TSP+P-LS(CGBS)
Nested					
GBS(49)	16981	18732	17111	/	/
GBE(49)	15028	15177	/	14966	/
CGBS(49)	13534	13208	/	/	13174
Time(s)	0.00	8.32	18308.49	24604.85	21353.64
Banded					
GBS(49)	17533	19339	17586	/	/
GBE(49)	15118	15329	/	15165	/
CGBS(49)	13598	13350	/	/	13276
Time(s)	0.00	6.95	21782.34	24108.63	19578.18

Table A.1 49x49 GBS, GBE and CGBS kernels on the Nested and Banded artificial datasets with 25% noise. Abs error and ZerosAndOnes padding. Cfr. Q1

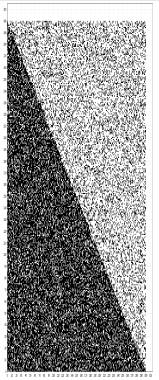
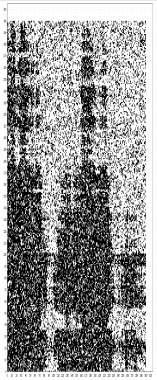
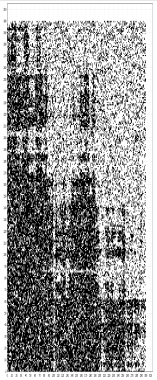
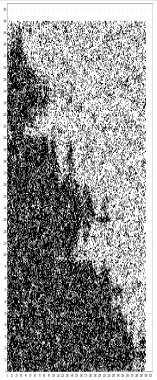
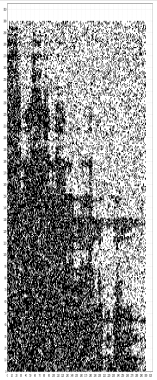
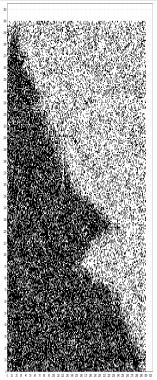
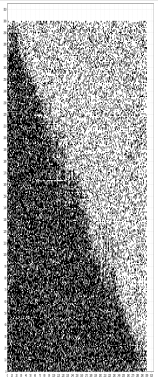
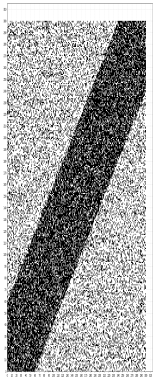
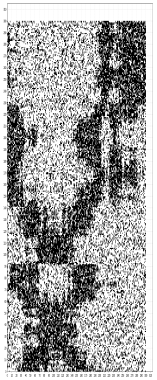
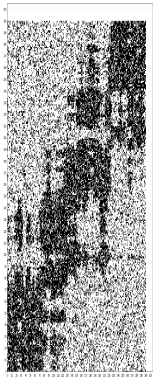
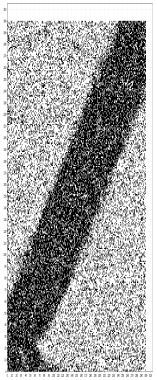
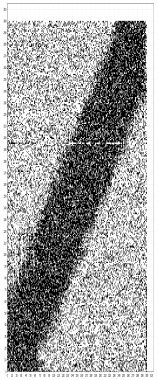
Config	Original	TSP	TSP+P-LS(3x3)	TSP+P-LS(11x11)	TSP+P-LS(11x3)	TSP+P-LS(25x25)	TSP+P-LS(49x49)
Nested							
GBS(3)	13480	12934	12966	/	/	/	/
GBS(11)	16554	17292	/	16624	/	/	/
GBS(11x3)	15949	16152	/	/	16071	/	/
GBS(25)	16868	18194	/	/	/	16941	/
GBS(49)	16981	18732	/	/	/	/	17111
Time(s)	0.00	0.53	21.75	380.36	82.48	3025.15	24604.85
Banded							
GBS(3)	13514	13084	13066	/	/	/	/
GBS(11)	16688	17410	/	16726	/	/	/
GBS(11x3)	16040	16271	/	/	16231	/	/
GBS(25)	17155	18354	/	/	/	17130	/
GBS(49)	17533	19339	/	/	/	/	17586
Time(s)	0.00	0.49	16.23	330.50	54.09	4482.11	21782.34

Table A.2 Different sizes of GBS kernels, on the Nested and Banded artificial datasets with 25% noise. Abs error and ZerosAndOnes padding. Cfr. Q2, Q4, Q11

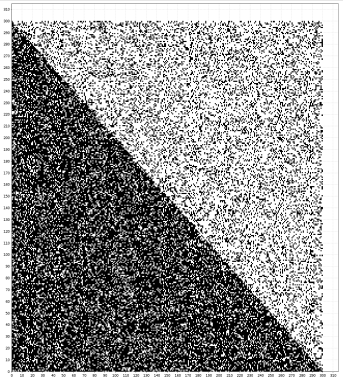
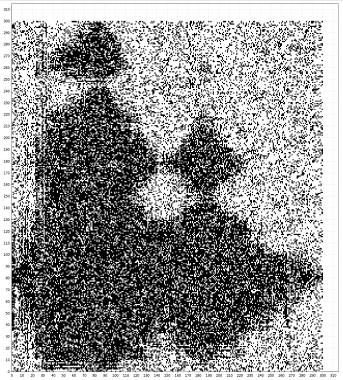
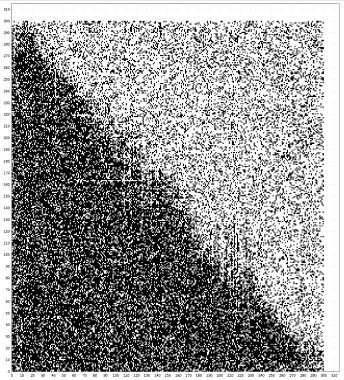
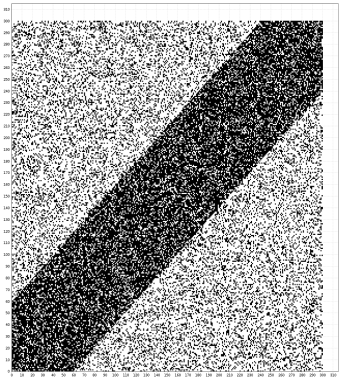
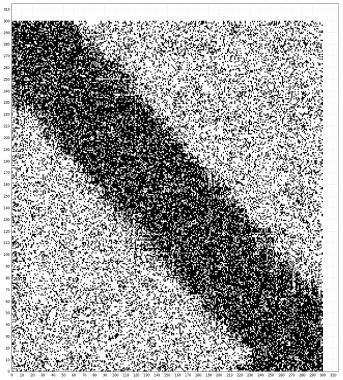
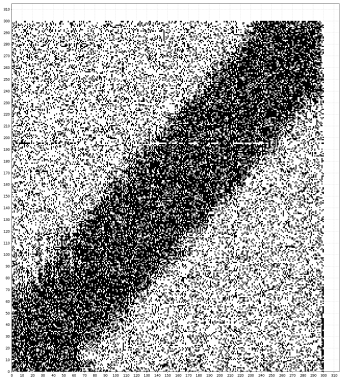
Config	Original	TSP + LS(Quadr)	TSP+LS(Abs)
Nested			
Quadratic	8589	8632	/
Abs	16981	/	17111
Time(s)	0.00	33088.38	18308.49
Banded			
Quadratic	9748	9723	/
Abs	17533	/	17586
Time(s)	0.00	9038.25	21782.34

Table A.3 Quadratic and Abs error on the Nested and Banded artificial datasets with 25% noise. 49x49 GBS kernel, ZerosAndOnes padding. Cfr. Q3

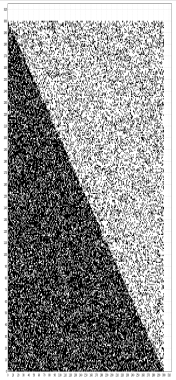
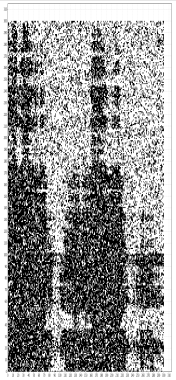
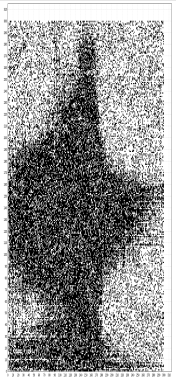
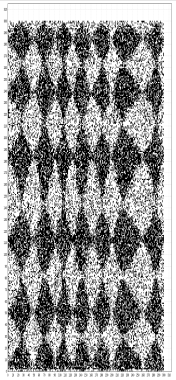
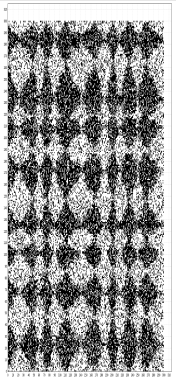
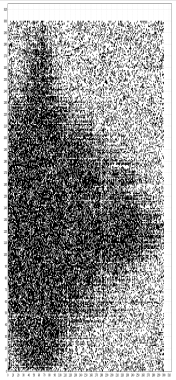
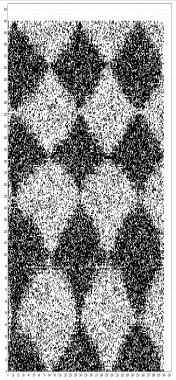
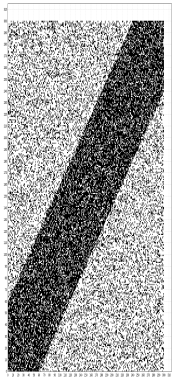
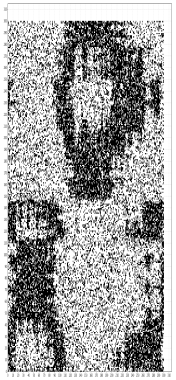
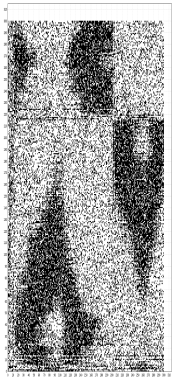
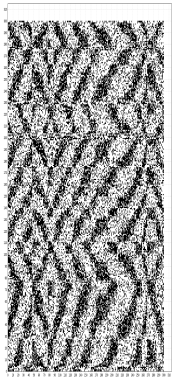
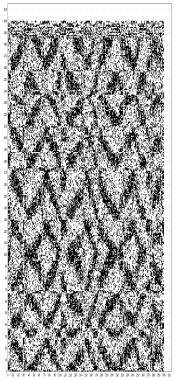
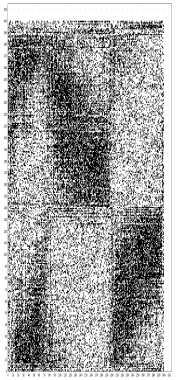
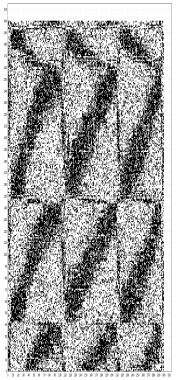
Config	Original	TSP	LS(Rect)	LS(Swap)	LS(SwapA)	LS(Reloc)	LS(Rev)
Nested							
GBS(29)	16898	18320	**17735**	19811	20177	18063	18465
Time(s)	0.00	5.41	10748.09	1848.85	2460.03	15423.88	5139.28
Banded							
GBS(29)	17229	18532	**18363**	20952	21049	19180	19392
Time(s)	0.00	3.81	10724.35	1196.76	1685.69	22227.87	5550.54

Table A.4 Single move with LS on the Nested and Banded artificial datasets with 25% noise. 49x49 GBS kernel, Abs error and ZerosAndOnes padding. Cfr. Q5

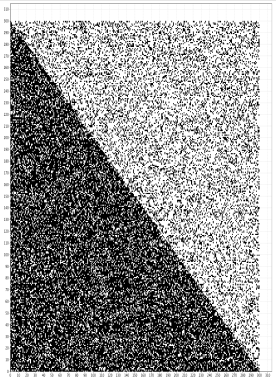
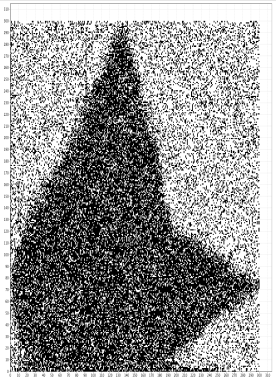
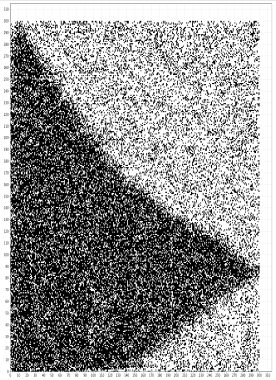
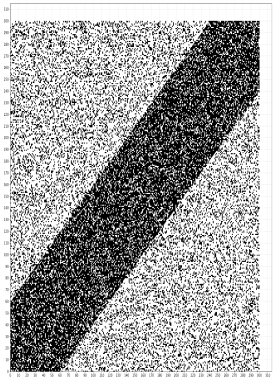
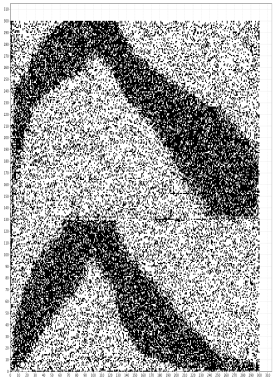
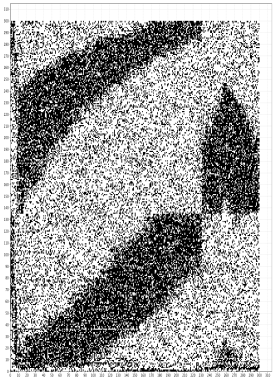
Config	Original	LS (with learning)	LS (without learning)
Nested			
GBS(29)	16898	17086	**16983**
Time(s)	0	17197.91	19256.64
Banded			
GBS(29)	17229	**17728**	17804
Time(s)	0	14014.97	18318.83

Table A.5 Impact of learning the moves probabilities on the Nested and Banded artificial datasets with 25% noise.
Cfr. Q5, Q6

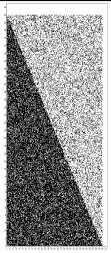
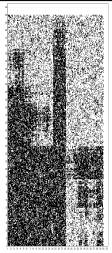
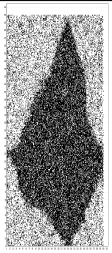
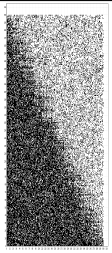
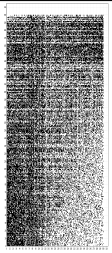
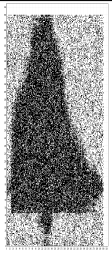
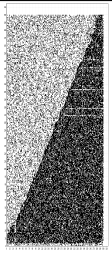
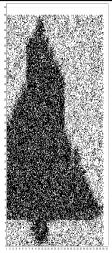
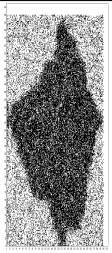
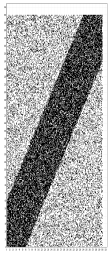
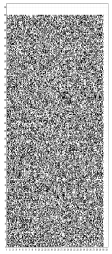
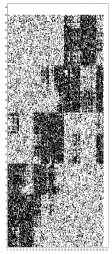
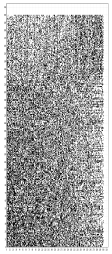
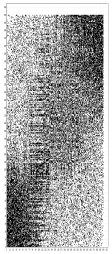
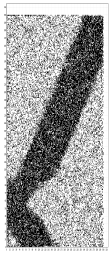
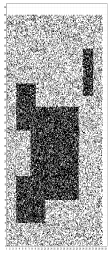
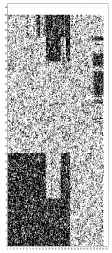
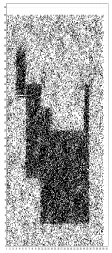
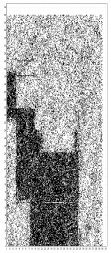
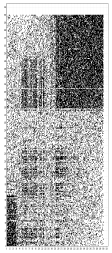
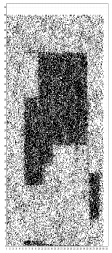
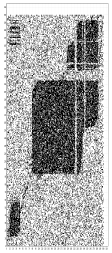
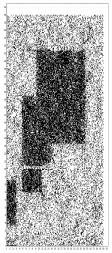
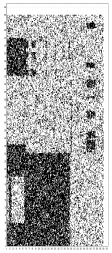
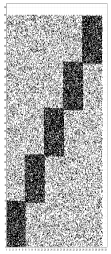
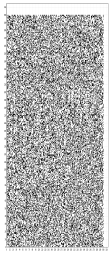
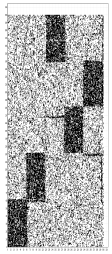
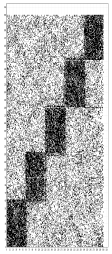
Config	Original	Shuffled	Heatmap	Heat+P-LS	Nested	Nest+P-LS	Bary	Bary+P-LS	Bidi	Bidi+P-LS	TSP	TSP+P-LS
Nested												
GBS(25)	16868	22160	17889	17101	17325	17078	20097	17219	17435	**17044**	18471	17085
Time(s)	0.00	0.00	1.52	11661.17	0.04	11759.50	2.58	10002.71	6.40	7968.18	4.69	8282.42
Banded												
GBS(25)	17155	21846	18360	17117	21264	17854	19556	17456	17217	**17083**	18447	17204
Time(s)	0.00	0.00	3.07	7656.26	0.01	8636.43	1.57	8133.27	3.35	3371.08	1.95	4305.23
Blocks												
GBS(25)	17241	21227	17415	16887	17973	**16849**	18410	16979	17489	16871	17358	16906
Time(s)	0.00	0.00	1.54	9021.81	0.59	11618.04	1.32	8302.27	3.50	6566.09	5.04	7832.64
Banded B												
GBS(25)	17369	20411	17487	17101	20156	17969	18753	17264	17404	17098	17573	**17064**
Time(s)	0.00	0.00	1.14	6027.07	0.01	8016.02	0.85	6026.73	2.55	5821.72	3.03	5465.31

Table A.6 Artificial datasets with 25% noise ran with local search using different initializations.
Cfr. Q7

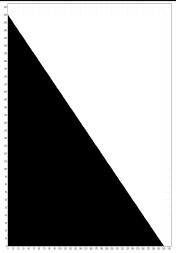
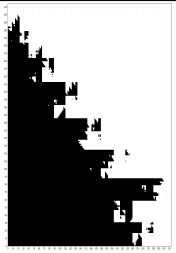
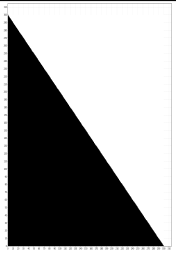
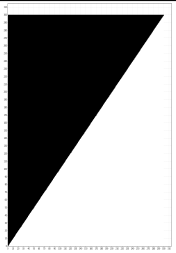
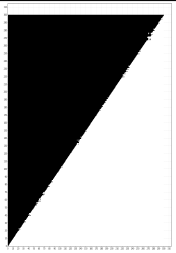
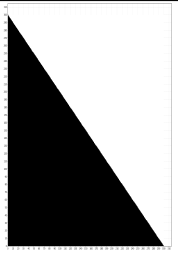
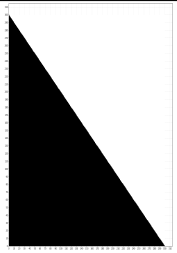
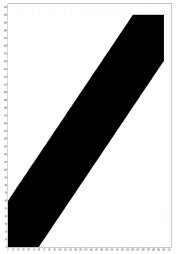
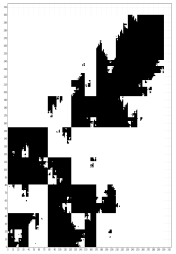
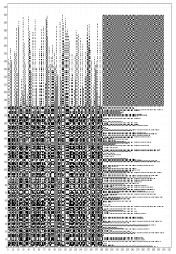
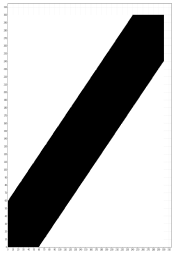
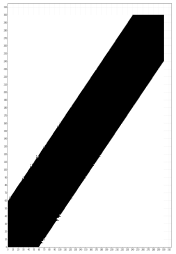
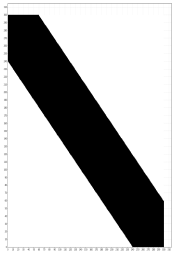
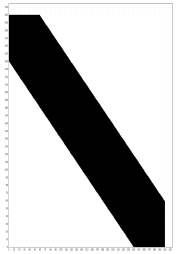
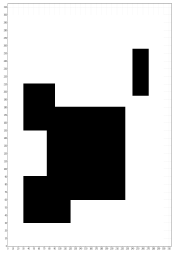
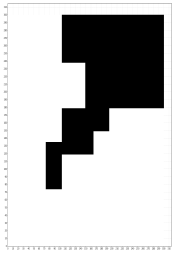
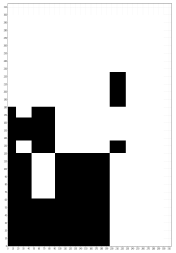
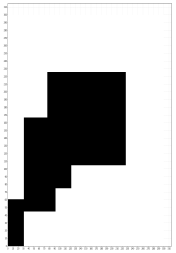
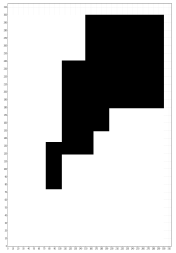
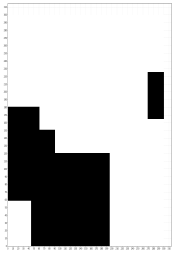
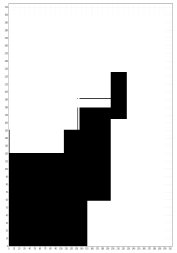
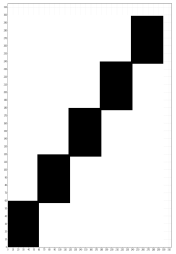
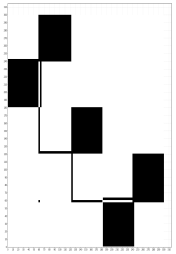
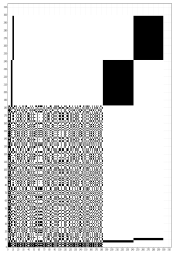
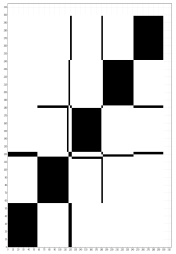
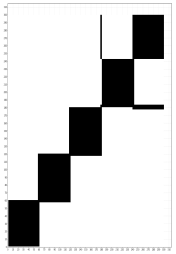
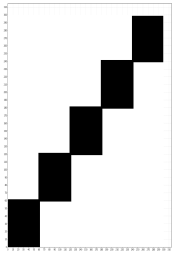
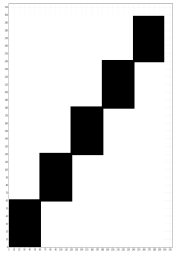
Config	Original	Heatmap	Nested	Barycentric (10)	Bidirectional (100)	TSP	TSP + P-LS
Nested							
GBS(49)	1806	4023	1806	3137	3146	**1806**	**1806**
Time(s)	0.00	3.60	0.48	4.95	64.19	12.68	2001.98
Banded							
GBS(49)	3458	7748	17140	3458	3461	**3435**	**3435**
Time(s)	0.00	4.62	0.44	3.45	10.95	5.39	1225.43
Blocks							
GBS(49)	4438	4286	3719	3504	3911	**2786**	**2708**
Time(s)	0.00	5.08	0.56	2.49	30.42	6.89	3346.99
Banded Blocks							
GBS(49)	4495	5191	9723	6166	4927	**4476**	**4476**
Time(s)	0.00	5.33	0.01	1.84	27.70	4.84	706.08

Table A.7 Artificial datasets with no noise, using a 49x49 GBS kernel with the absolute error and the ZerosAndOnes padding.
Cfr. Q8, Q10

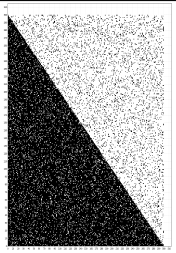
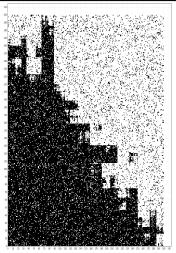
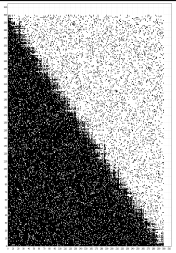
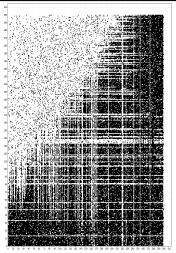
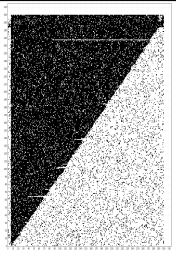
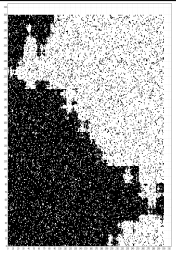
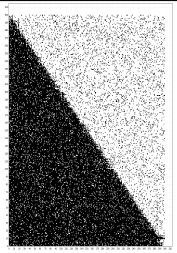
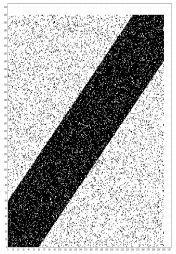
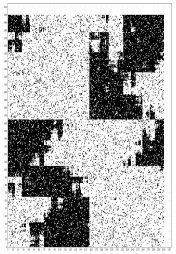
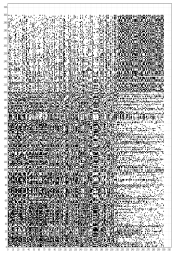
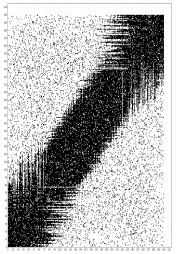
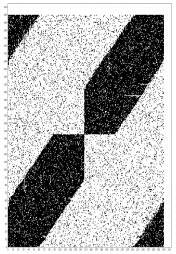
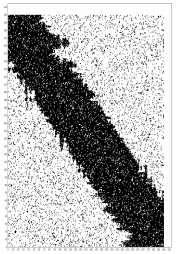
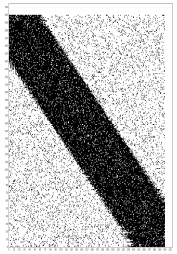
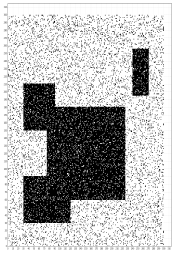
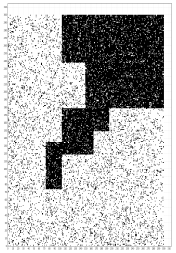
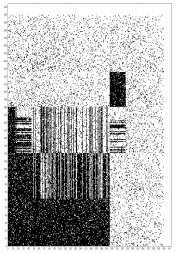
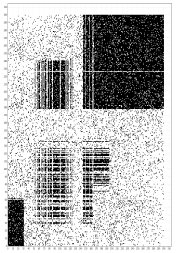
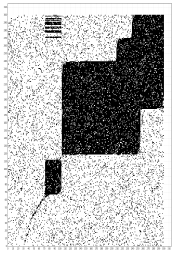
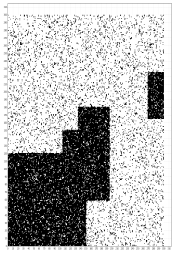
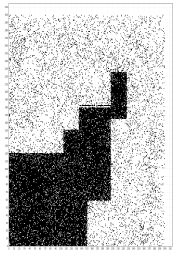
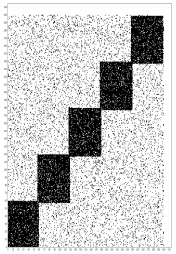
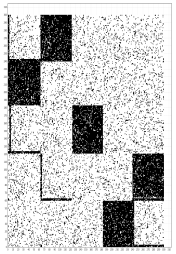
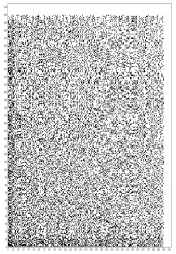
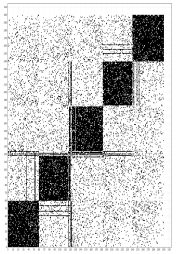
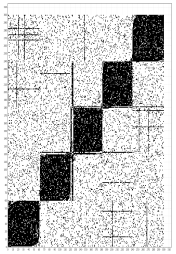
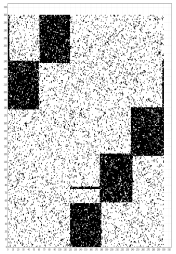
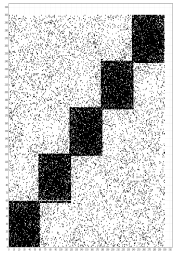
Config	Original	Heatmap	Nested	Barycentric (10)	Bidirectional (100)	TSP	TSP + P-LS
Nested							
GBS(49)	9031	10387	9305	15875	10698	10478	**9021**
Time(s)	0.00	4.29	0.35	11.04	57.05	19.19	26371.55
Banded							
GBS(49)	10201	13062	19354	11552	11185	10489	**10148**
Time(s)	0.00	3.89	0.43	3.61	12.33	5.40	16695.25
Blocks							
GBS(49)	10692	10761	11439	12756	10621	9631	**9224**
Time(s)	0.00	4.31	0.01	3.21	17.16	8.28	32596.49
Banded Blocks							
GBS(49)	10829	11254	17019	11339	11472	10964	**10403**
Time(s)	0.00	3.98	0.01	2.46	22.97	6.61	29139.47

Table A.8 Artificial datasets with 10% noise, using a 49x49 GBS kernel with the absolute error and the ZerosAndOnes padding.
Cfr. Q8, Q10

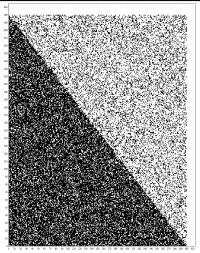
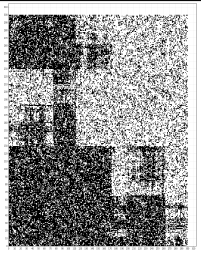
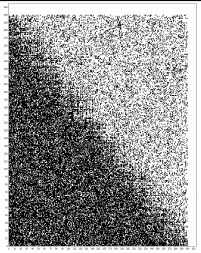
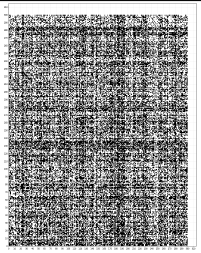
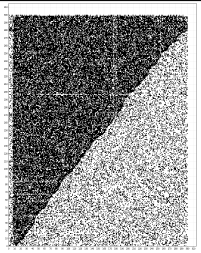
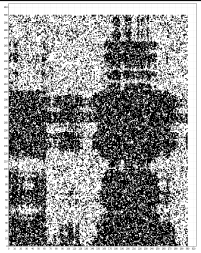
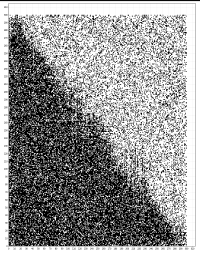
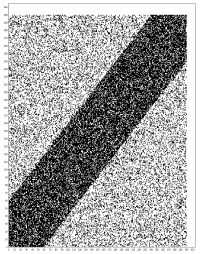
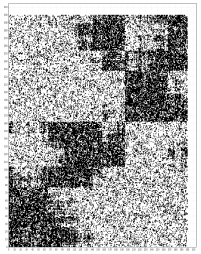
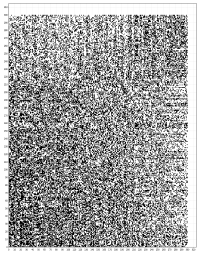
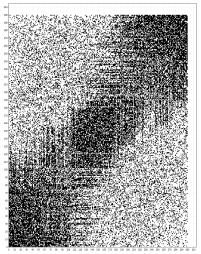
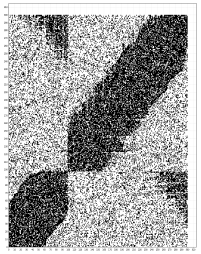
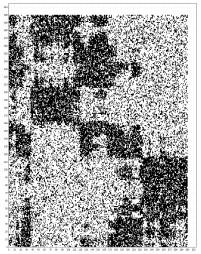
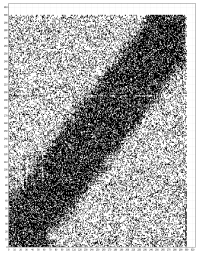
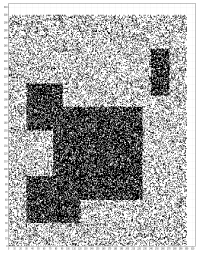
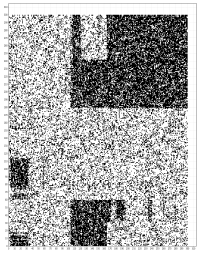
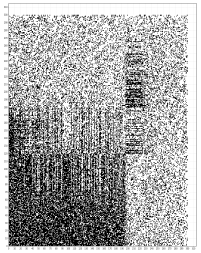
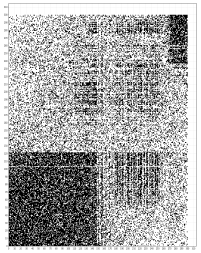
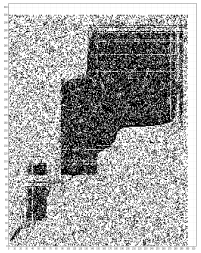
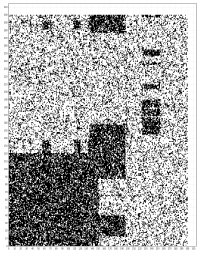
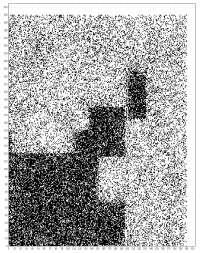
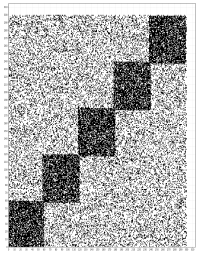
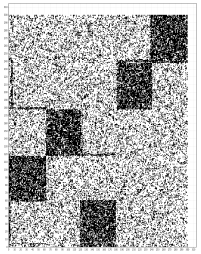
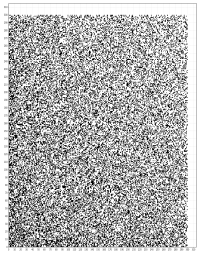
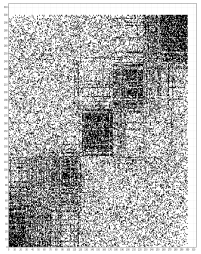
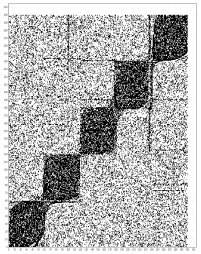
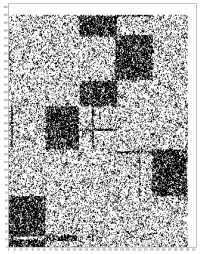
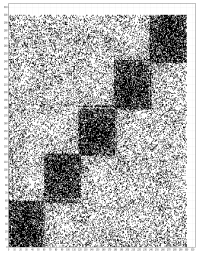
Config	Original	Heatmap	Nested	Barycentric (10)	Bidirectional (10)	TSP	TSP+P-LS
Nested							
GBS(49)	16981	18341	17311	22242	17962	18820	**17111**
Time(s)	0.00	3.56	0.03	11.74	14.41	10.09	21353.64
Banded							
GBS(49)	17533	18872	21193	18969	18202	19269	**17586**
Time(s)	0.00	4.30	0.75	8.42	10.62	12.35	21782.34
Blocks							
GBS(49)	17640	18118	17940	18416	17744	17728	**16953**
Time(s)	0.00	3.56	0.01	5.39	24.63	14.68	45855.61
Banded Blocks							
GBS(49)	17822	17888	20019	18717	17831	18088	**17438**
Time(s)	0.00	3.74	0.01	3.29	16.18	8.81	43510.83

Table A.9 Artificial datasets with 25% noise, using a 49x49 GBS kernel with the absolute error and the ZerosAndOnes padding. Cfr. Q8, Q10, Q11

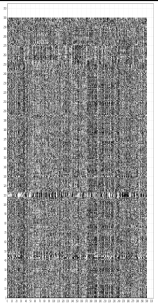
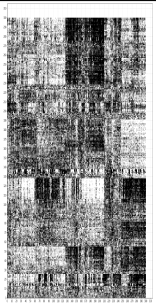
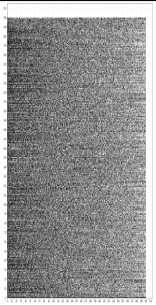
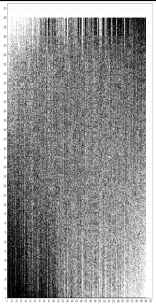
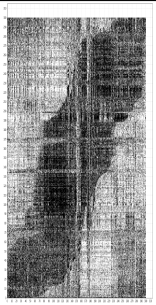
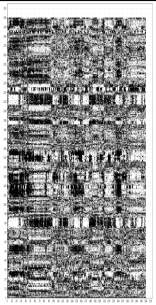
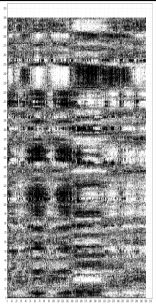
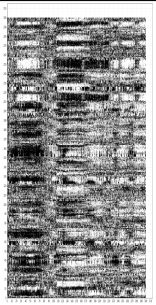
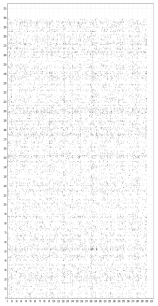
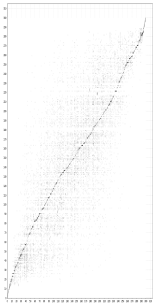
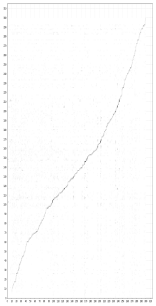
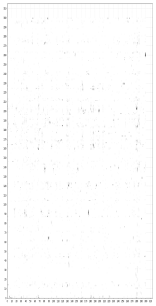
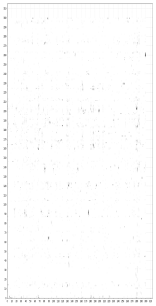
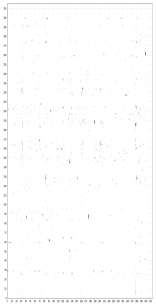
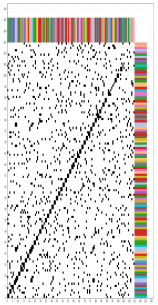
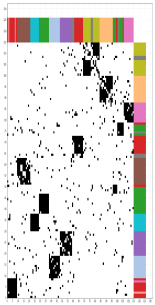
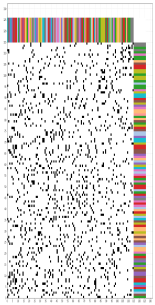
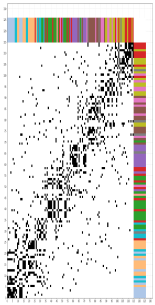
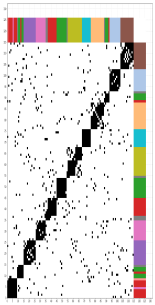
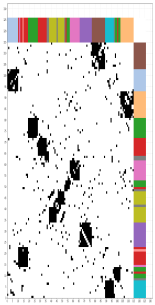
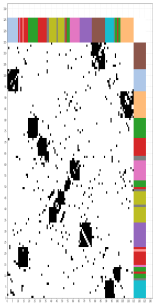
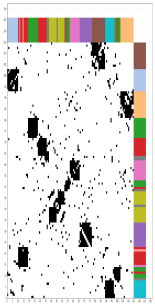
Config	Original	Heatmap	Nested	Barycentric (10)	Bidirectional (10)	TSP	TSP+P-LS	TSP+P-LS
Gene expression								
GBS(25)	189529	161640	185616	172142	162298	166303	**158741**	/
GBS(29x5)	186242	**152121**	184373	169254	158296	155110	/	152661
Time(s)	0.00	23.30	20.05	10.44	64.11	30.65	12653.78	3067.16
Paleo								
GBS(25)	14012	13304	13475	13845	13323	**12898**	**12898**	/
GBS(29x5)	13879	12782	13449	13669	13057	12283	/	**12135**
Time(s)	0.00	317.60	0.62	5.80	36.49	178.23	397.03	100.34
Football								
GBS(7)	1096	658	1119	915	641	**618**	**618**	/
GBS(11)	1127	763	1143	957	735	**725**	/	**725**
Time(s)	0.00	4.17	0.05	0.07	0.22	0.19	15.27	25.59
Accuracies rows	7.23	86.16	13.98	47.66	86.83	86.85	85.90	87.47
Accuracies columns	6.82	**87.38**	11.80	48.49	86.22	84.19	82.95	83.85

Table A.10 Gene expression, Paleo and Football binary datasets.
Cfr. Q4, Q9, Q11, Q12

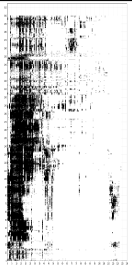
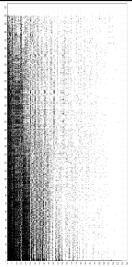
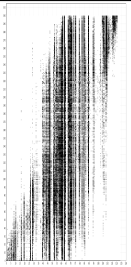
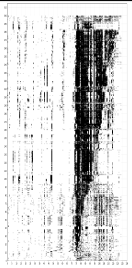
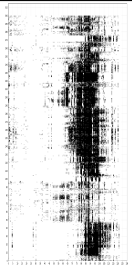
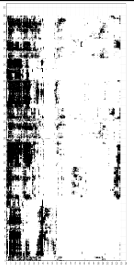
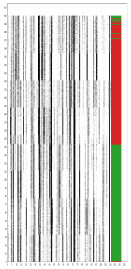
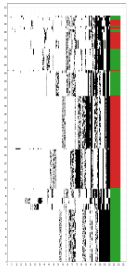
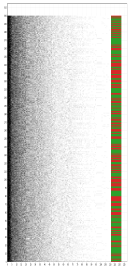
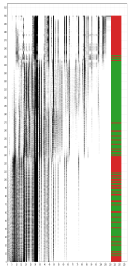
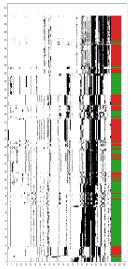
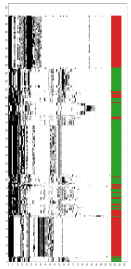
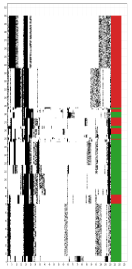
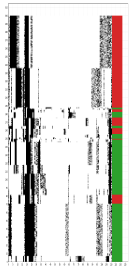
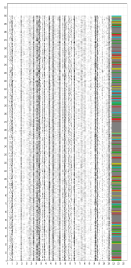
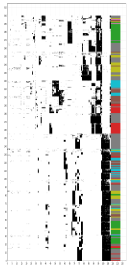
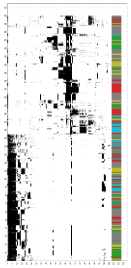
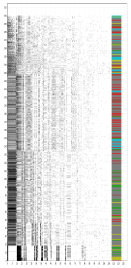
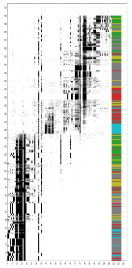
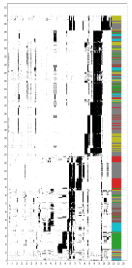
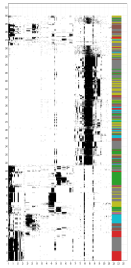
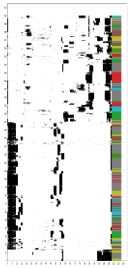
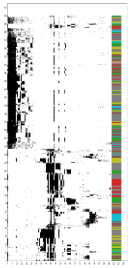
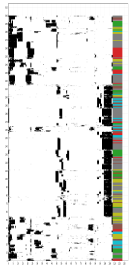
Config	Original	Heatmap	Heat+P-LS	Nested	Barycentric	Bidirectional	Bidi+P-LS	TSP	TSP+P-LS	TSP+P-LS
Mammals										
GBS(25)	41405	25168	**20776**	23058	32524	26316	22387	21475	21384	/
GBS(29x5)	36946	17938	/	22613	29160	21056	/	**16000**	/	**16000**
Time(s)	0.00	8.52	15816.69	0.35	0.53	25.25	54.74	10658.39	4956.01	1227.34
Mushrooms										
GBS(25)	128144	95796	77854	86903	105449	92699	**77189**	91125	78122	/
GBS(29x5)	119301	66741	/	86109	91824	70211	/	**59512**	/	**59512**
Time(s)	0.00	73.32	37738.22	0.21	9.59	764.08	37861.564	154.66	23961.56	5011.06
Accuracies	79.57	**99.82**	99.26	49.67	78.57	96.81	96.85	99.80	99.45	99.74
UCL										
GBS(25)	46021	35092	26732	35282	32404	33330	28068	29592	**26500**	/
GBS(29x5)	45127	18651	/	34573	25871	21305	/	**16148**	/	16196
Time(s)	0.00	19.81	4155.84	17.70	2.00	61.39	3892.67	34.28	6455.08	6826.96
Accuracies	8.43	61.01	57.34	13.92	47.89	45.31	44.04	**62.20**	55.06	58.03

Table A.11 Mammals, Mushrooms and UCL binary datasets.
Cfr. Q4, Q7, Q9, Q11, Q12

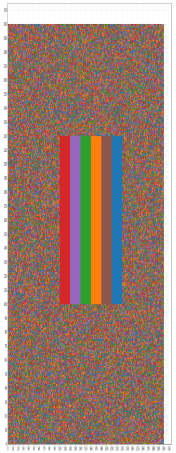
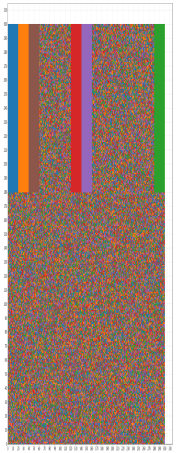
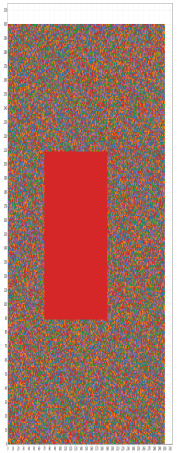
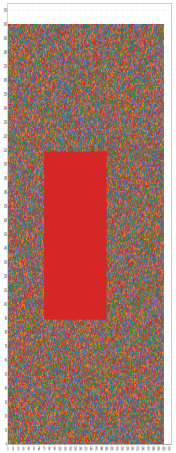
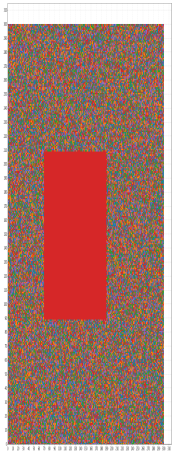
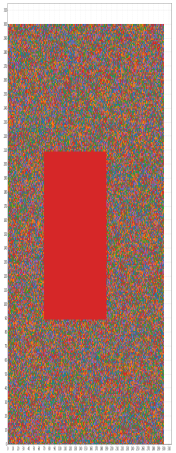
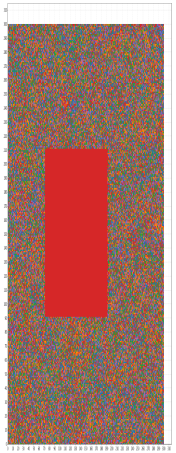
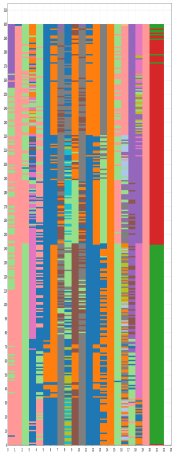
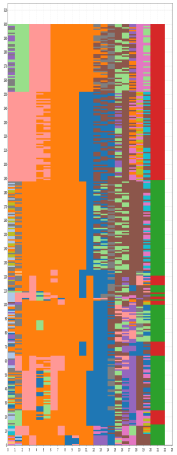
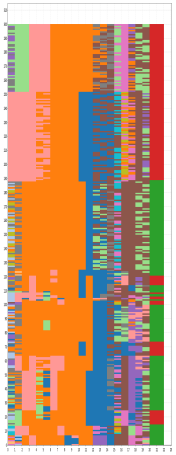
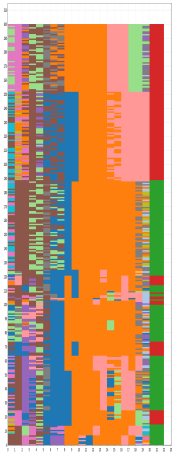
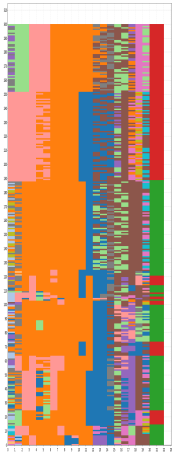
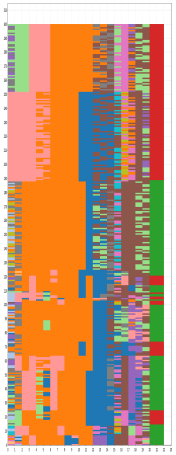
Config	Original	Heatmap	TSP	TSP+P-LS(3x3)	TSP+P-LS(11x3)	TSP+P-LS(11x11)	TSP+P-LS(21x21)
Categorical							
GBS(3)	50738	49947	48226	**48135**	/	/	/
GBS(11x3)	59826	59313	58357	/	**58103**	/	/
GBS(11)	62967	62670	61243	/	/	**60933**	/
GBS(21)	64971	64716	62448	/	/	/	**62119**
Time(s)	0.00	4.49	4.30	14.49	133.55	1431.01	8893.02
Categorical Mushrooms							
GBS(3)	90248	67661	45841	**45799**	/	/	/
GBS(11x3)	103343	76189	53481	/	**53421**	/	/
GBS(11)	119628	100873	**82760**	/	/	**82760**	/
GBS(21)	121527	108641	98007	/	/	/	**45799**
Time(s)	0.00	25.69	302.56	31.77	183.77	2207.66	31772
Accuracies	79.69	**99.84**	99.75	**99.84**	99.82	99.79	99.78

Table A.12 Artificial categorical dataset and the categorical mushrooms dataset.
Cfr. Q9, Q10, Q12

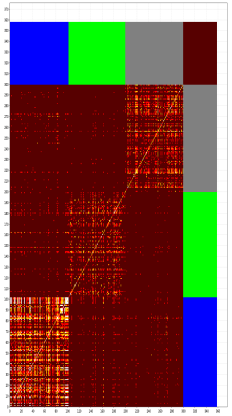
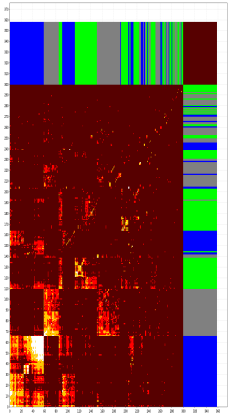
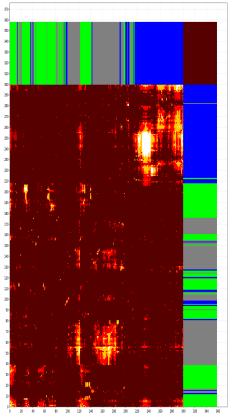
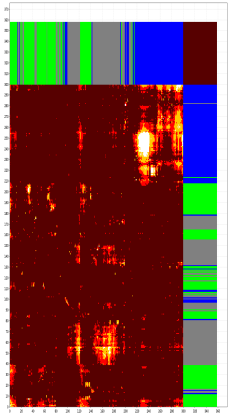
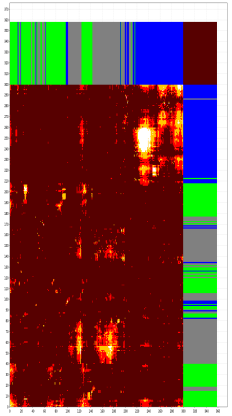
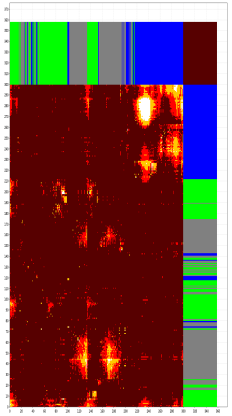
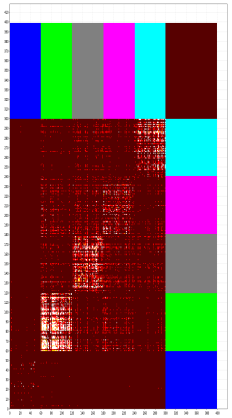
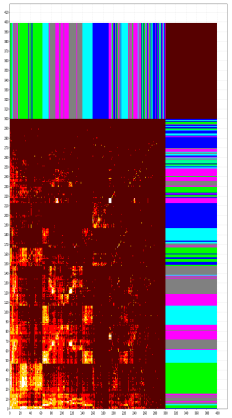
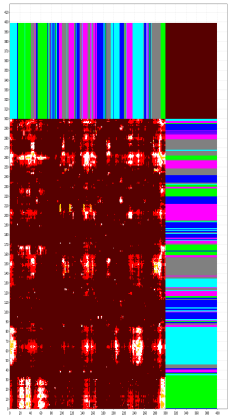
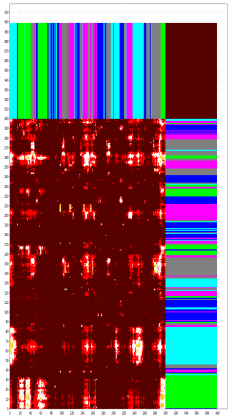
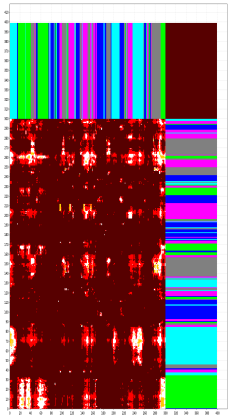
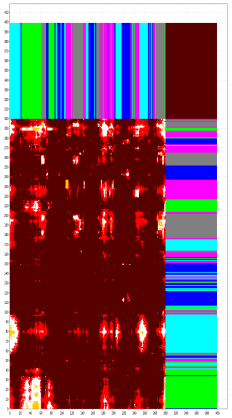
Config	Original	Heatmap	TSP	TSP + P-LS	TSP + P-LS	TSP + P-LS
news_3cl_1						
GBS(7)	50609.1	26519	23640	**23177.7**	/	/
GBS(11)	53378.3	30643	27841.9	/	**26765.5**	/
GBS(21)	56043.9	36761	34236	/	/	**31593.9**
Time(s)	0.00	0.00	9.32	306.38	1773.35	19846.24
Accuracies rows	99.83	87.63	**91.68**	91.67	90.50	88.97
Accuracies cols	99.83	87.92	88.13	89.31	**90.31**	89.67
news_5cl_1						
GBS(7)	251425	129953	118353.2	**117423**	/	/
GBS(11)	264812.2	145995	136843.6	/	**133782**	/
GBS(21)	275998.6	167744	164398.6	/	/	**154107.9**
Time(s)	0.00	31.22	124.3	990.985	7122.527	51115.464
Accuracies rows	99.40	78.66	**81.59**	81.57	80.78	80.28
Accuracies cols	99.60	78.53	81.05	81.29	81.58	**81.89**

Table A.13 News datasets.
Cfr. Q12

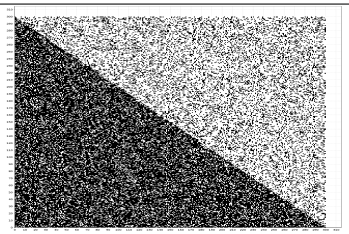
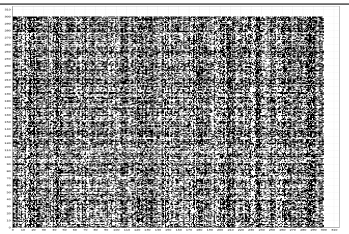
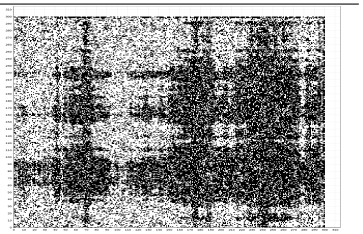
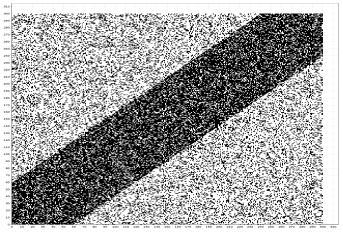
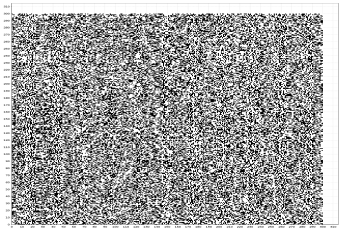
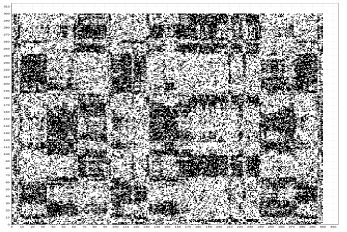
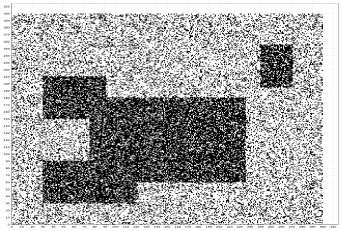
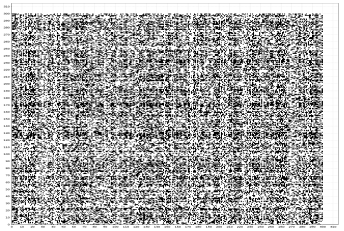
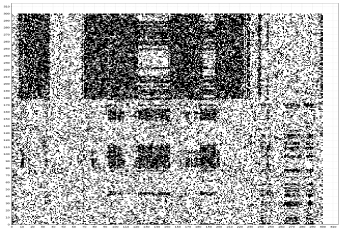
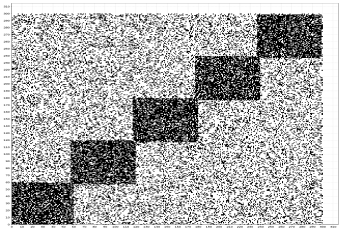
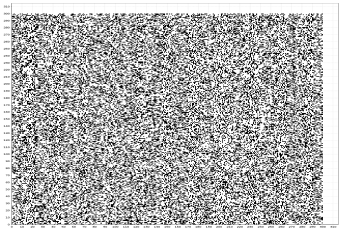
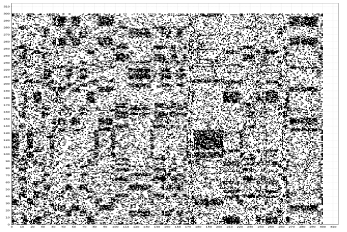
Config	Original	Shuffled	Kmax
Nested			
GBS(5)	15430	19887	16329
Time(s)	0.00	0.00	98.84
Banded			
GBS(5)	15499	20164	17034
Time(s)	0.00	0.00	75.27
Blocks			
GBS(5)	15549	19224	16014
Time(s)	0.00	0.00	64.01
Banded Blocks			
GBS(5)	15595	18836	16700
Time(s)	0.00	0.00	107.51

Table A.14 KMax for $k = 5$ on the artificial datasets with 25% noise. Abs error and ZerosAndOnes padding

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**