Getting Things in Order: An introduction to the R package **seriation**

Michael Hahsler, Kurt Hornik and Christian Buchta

February 4, 2008

Abstract

Seriation, i.e., finding a suitable linear order for a set of objects given data and a loss or merit function, is a basic problem in data analysis. Caused by the problem's combinatorial nature, it is hard to solve for all but very small sets. Nevertheless, both exact solution methods and heuristics are available. In this paper we present the package **seriation** which provides an infrastructure for seriation with R. The infrastructure comprises data structures to represent linear orders as permutation vectors, a wide array of seriation methods using a consistent interface, a method to calculate the value of various loss and merit functions, and several visualization techniques which build on seriation. To illustrate how easily the package can be applied for a variety of applications, a comprehensive collection of examples is presented.

1 Introduction

A basic problem in data analysis, called *seriation* or sometimes *sequencing*, is to arrange all objects in a set in a linear order given available data and some loss or merit function in order to reveal structural information. Together with cluster analysis and variable selection, seriation is an important problem in the field of *combinatorial data analysis* (**?**). Solving problems in combinatorial data analysis requires the solution of discrete optimization problems which, in the most general case, involves evaluating all feasible solutions. Due to the combinatorial nature, the number of possible solutions grows with problem size (number of objects, n) by the order $O(n!)$. This makes a brute-force enumerative approach infeasible for all but very small problems. To solve larger problems (currently with up to 40 objects), partial enumeration methods can be used. For example, **?** propose dynamic programming and **?** use a branch-and-bound strategy. For even larger problems only heuristics can be employed.

It has to be noted that seriation has a rich history in archeology. **?** was the first to use seriation as a formal method. He applied it to find a chronological order for graves discovered in the Nile area given objects found there. He used a cross-tabulation of grave sites and objects and rearranged the table using row and column permutations till all large values were close to the diagonal. In the rearranged table graves with similar objects are closer to each other. Together with the assumption that different objects continuously come into and go out of fashion, the order of graves in the rearranged table suggests a chronological order. Initially, the rearrangement of rows and columns of this contingency table was done manually and the adequacy was only judged subjectively by the researcher. Later, **?**, **?** and others proposed measures of agreement between rows to quantify optimality of the resulting table. A comprehensive description of the development of seriation in archeology is presented by **?**.

Techniques related to seriation are also popular in several other fields. Especially in ecology scaling techniques are used under the name *ordination*. For these applications several R packages already exist (e.g., **ade4** (**?**) and **vegan** (**?**)). This paper describes the new package **seriation** which differs from existing packages in the following ways:

- **seriation** provides a flexible infrastructure for seriation;
- **seriation** focuses on seriation as a combinatorial optimization problem.

This paper starts with a formal introduction of the seriation problem as a combinatorial optimization problem in Section 2. In Section 3 we give an overview of seriation methods. In

Section 4 we present the infrastructure provided by the package **seriation**. Several examples and applications for seriation are given in Section 5. Section 6 concludes.

2 Seriation as a combinatorial optimization problem

To seriate a set of n objects $\{O_1, \ldots, O_n\}$ one typically starts with an $n \times n$ symmetric dissimilarity matrix $\mathbf{D} = (d_{ij})$ where d_{ij} for $1 \leq i, j \leq n$ represents the dissimilarity between objects O_i and O_j, and $d_{ii} = 0$ for all i. We define a permutation function Ψ as a function which reorders the objects in $\mathbf{D}$ by simultaneously permuting rows and columns. The seriation problem is to find a permutation function Ψ^* which optimizes the value of a given loss function L or merit function M. This results in the optimization problems

$$\Psi^* = \underset{\Psi}{\operatorname{argmin}}\, L(\Psi(\mathbf{D})) \quad \text{or} \quad \Psi^* = \underset{\Psi}{\operatorname{argmax}}\, M(\Psi(\mathbf{D})), \tag{1}$$

respectively.

A symmetric dissimilarity matrix is known as *two-way one-mode* data since it has columns and rows (two-way) but only represents one set of objects (one-mode). Seriation is also possible for two-way two-mode data which are represented by a general nonnegative matrix. In such data columns and rows represent two sets of objects which are reordered simultaneously. For loss/merit functions for two-way two-mode data the optimal order of columns can depend of the order of rows and vice versa or it can be independent allowing for breaking the optimization down into two separate problems, one for the columns and one for the rows. Another way to deal with the seriation for two-way two-mode data is to calculate two dissimilarity matrices, one for each mode, and then solve two seriation problems for two-way one-mode data. Furthermore, seriation can be generalized to k-way k-mode data in the form of a k-dimensional array by defining suitable loss/merit functions for such data or by breaking the problem down into several lower dimensional independent problems.

To assess the complexity of seriation of k-way k-mode data, let us assume the data is a k-dimensional array with the dimensions containing $n_1, n_2, \ldots, n_k$ objects. If the loss/merit function allows for separating the problem into k independent problems, the problem size is just the sum of the individual problems. By using complete enumeration the size is $O(\sum_{i=1}^{k} n_i!)$. If the problem is not separable and the optimal seriation of each dimension depends on the order of the objects of the other dimensions, the problem size is $O((\sum_{i=1}^{k} n_i)!)$. For example for $k = 5$ and all dimensions containing 5 objects, the search space for separable dimensions is only 600 while without separability it is larger than 10^{25} clearly too big to be solvable in reasonable time. This shows that for data with even only a few dimensions and a few objects each, finding the optimal solution is infeasible and loss/merit functions which allow for separating the problem are highly desirable.

In the following subsections, we review some commonly employed loss/merit functions. Most functions are used for two-way one-mode data but the measure of effectiveness and stress can be also used for two-way two-mode data. For the implementation of various loss or merit measures see function `criterion()` in Section 4.

2.1 Column/row gradient measures

A symmetric dissimilarity matrix where the values in all rows and columns only increase when moving away from the main diagonal is called a perfect *anti-Robinson matrix* after the statistician **?**. Formally, an $n \times n$ dissimilarity matrix $\mathbf{D}$ is in anti-Robinson form if and only if the following two gradient conditions hold (**?**):

$$\text{within rows:} \quad d_{ik} \leq d_{ij} \quad \text{for} \quad 1 \leq i < k < j \leq n; \tag{2}$$

$$\text{within columns:} \quad d_{kj} \leq d_{ij} \quad \text{for} \quad 1 \leq i < k < j \leq n. \tag{3}$$

In an anti-Robinson matrix the smallest dissimilarity values appear close to the main diagonal, therefore, the closer objects are together in the order of the matrix, the higher their similarity. This provides a natural objective for seriation.

It has to be noted that $\mathbf{D}$ can be brought into a perfect anti-Robinson form by row and column permutation whenever $\mathbf{D}$ is an ultrametric or $\mathbf{D}$ has an exact Euclidean representation

in a single dimension (**?**). However, for most data only an approximation to the anti-Robinson form is possible.

A suitable merit measure which quantifies the divergence of a matrix from the anti-Robinson form was given by **?** as

$$M(\mathbf{D}) = \sum_{i<k<j} f(d_{ik}, d_{ij}) + \sum_{i<k<j} f(d_{kj}, d_{ij}) \tag{4}$$

where $f(\cdot,\cdot)$ is a function which defines how a violation or satisfaction of a gradient condition for an object triple (O_i, O_k and O_j) is counted. **?** suggest two functions. The first function is given by:

$$f(z,y) = \text{sign}(y-z) = \begin{cases} +1 & \text{if} \quad z < y; \\ 0 & \text{if} \quad z = y; \\ -1 & \text{if} \quad z > y. \end{cases} \tag{5}$$

It results in the raw number of triples satisfying the gradient constraints minus triples which violate the constraints.

The second function is defined as:

$$f(z,y) = |y-z|\text{sign}(y-z) = y - z \tag{6}$$

It weighs each satisfaction or violation by its magnitude given by the absolute difference between the values.

2.2 Anti-Robinson events

An even simpler loss function can be created in the same way as the gradient measures above by concentrating on violations only.

$$L(\mathbf{D}) = \sum_{i<k<j} f(d_{ik}, d_{ij}) + \sum_{i<k<j} f(d_{kj}, d_{ij}) \tag{7}$$

To only count the violations we use

$$f(z,y) = I(z,y) = \begin{cases} 1 & \text{if} \quad z < y \quad \text{and} \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

$I(\cdot)$ is an indicator function returning 1 only for violations. **?** presented a formulation for an equivalent loss function and called the violations *anti-Robinson events*. **?** also introduced a weighted versions of the loss function resulting in

$$f(z,y) = |y-z|I(z,y) \tag{9}$$

using the absolute deviations as weights.

2.3 Hamiltonian path length

The dissimilarity matrix $\mathbf{D}$ can be represented as a finite weighted graph $G = (\Omega, E)$ where the set of objects Ω constitute the vertices and each edge $e_{ij} \in E$ between the objects $O_i, O_j \in \Omega$ has a weight w_{ij} associated which represents the dissimilarity d_{ij}.

Such a graph can be used for seriation (see, e.g., **??**). An order Ψ of the objects can be seen as a path through the graph where each node is visited exactly once, i.e., a Hamiltonian path. Minimizing the Hamiltonian path length results in a seriation optimal with respect to dissimilarities between neighboring objects. The loss function based on the Hamiltonian path length is:

$$L(\mathbf{D}) = \sum_{i=1}^{n-1} d_{i,i+1}. \tag{10}$$

Note that the length of the Hamiltonian path is equal to the value of the *minimal span loss function* (as used by **?**), and both notions are related to the *traveling salesperson problem* (**?**).

2.4 Inertia criterion

Another way to look at the seriation problem is not to focus on placing small dissimilarity values close to the diagonal, but to push large values away from it. A function to quantify this is the moment of inertia of dissimilarity values around the diagonal (**?**) defined as

$$M(\mathbf{D}) = \sum_{i=1}^{n}\sum_{j=1}^{n} d_{ij}|i-j|^2. \tag{11}$$

$|i-j|^2$ is used as a measure for the distance to the diagonal and d_{ij} gives the weight. This is a merit function since the sum increases when higher dissimilarity values are placed farther away from the diagonal.

2.5 Least squares criterion

Another natural loss function for seriation is to quantify the deviations between the dissimilarities in $\mathbf{D}$ and the rank differences of the objects. Such deviations can be measured, e.g, by the sum of squares of deviations (**?**) defined by

$$L(\mathbf{D}) = \sum_{i=1}^{n}\sum_{j=1}^{n} (d_{ij} - |i-j|)^2, \tag{12}$$

where $|i-j|$ is the rank difference or gap between O_i and O_j.

The least squares criterion defined here is related to uni-dimensional scaling (**?**), where the objective is to place all n objects on a straight line using a position vector $\mathbf{z} = z_1, z_2, \ldots, z_n$ such that the dissimilarities in $\mathbf{D}$ are preserved by the relative positions in the best possible way. The optimization problem of uni-dimensional scaling is to find the position vector $\mathbf{z}^*$ which minimizes $\sum_{i=1}^{n}\sum_{j=1}^{n}(d_{ij} - |z_i - z_j|)^2$. This is close to the seriation problem, but in addition to the ranking of the objects also takes the distances between objects on the resulting scale into account.

Note that if Euclidean distance is used to calculate $\mathbf{D}$ from a data matrix $\mathbf{X}$, using the order of the elements in $\mathbf{X}$ as they occur projected on the first principal component of $\mathbf{X}$ minimizes the loss function of uni-dimensional scaling (using squared distances). Using this order, also provides a good solution for the least square seriation criterion.

2.6 Measure of effectiveness

? defined the *measure of effectiveness (ME)* for an $n \times m$ matrix $\mathbf{X} = (x_{ij})$ as

$$M(\mathbf{X}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}[x_{i,j+1} + x_{i,j-1} + x_{i+1,j} + x_{i-1,j}] \tag{13}$$

with, by convention $x_{0,j} = x_{n+1,j} = x_{i,0} = x_{i,m+1} = 0$. ME is maximized if each element is as closely related numerically to its four neighboring elements as possible.

ME was developed for two-way two-mode data, however, ME can also be used for a symmetric matrix (one-mode data) and gets maximal only if all large values are grouped together around the main diagonal.

Note that the definition in equation (13) can be rewritten as

$$M(\mathbf{X}) = \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}[x_{i,j+1} + x_{i,j-1}] + \sum_{i=1}^{n}\sum_{j=1}^{m} x_{ij}[x_{i+1,j} + x_{i-1,j}] \tag{14}$$

showing that the contributions of column and row order to the merit function are independent.

2.7 Stress

Stress measures the conciseness of the presentation of a matrix (two-mode data) and can be seen as a purity function which compares the values in a matrix with their neighbors. The stress measures used here are computed as the sum of squared distances of each matrix entry from its adjacent entries. **?** defined for an $n \times m$ matrix $\mathbf{X} = (x_{ij})$ two types of neighborhoods:

- The Moore neighborhood comprises the (at most) eight adjacent entries. The local stress measure for element x_{ij} is defined as

$$\sigma_{ij} = \sum_{k=\max(1,i-1)}^{\min(n,i+1)} \sum_{l=\max(1,j-1)}^{\min(m,j+1)} (x_{ij} - x_{kl})^2 \tag{15}$$

- The Neumann neighborhood comprises the (at most) four adjacent entries resulting in the local stress of x_{ij} of

$$\sigma_{ij} = \sum_{k=\max(1,i-1)}^{\min(n,i+1)} (x_{ij} - x_{kj})^2 + \sum_{l=\max(1,j-1)}^{\min(m,j+1)} (x_{ij} - x_{il})^2 \tag{16}$$

Both local stress measures can be used to construct a global measure for the whole matrix by summing over all entries which can be used as a loss function:

$$L(\mathbf{X}) = \sum_{i=1}^{n} \sum_{j=1}^{m} \sigma_{ij} \tag{17}$$

The major difference between the Moore and the Neumann neighborhood is that for the later the contributions of row and column order to stress are independent.

Stress can be also used as a loss function for symmetric proximity matrices (one-mode data). Note also, that stress with Neumann neighborhood is related to the measure of effectiveness defined above (in Section 2.6) since both measures are optimal if for each cell the cell and its four neighbors are numerically as similar as possible.

3 Seriation methods

Solving the discrete optimization problem for seriation with most loss/merit functions is clearly very hard. The number of possible permutations for n objects is $n!$ which makes an exhaustive search for sets with a medium to large number of objects infeasible. In this section, we describe some methods (partial enumeration, heuristics and other methods) which are typically used for seriation. For each method we state for which type of loss/merit functions it is suitable and whether it finds the optimum or is a heuristic. For the implementation of various seriation methods see function `seriate()` in Section 4.

3.1 Partial enumeration methods

Partial enumeration methods search for the exact solution of a combinatorial optimization problem. Exploiting properties of the search space, only a subset of the enormous number of possible combinations has to be evaluated. Popular partial enumeration methods which are used for seriation are *dynamic programming* (**?**) and *branch-and-bound* (**?**).

Dynamic programming recursively searches for the optimal solution checking and storing $2^n - 1$ results. Although $2^n - 1$ grows at a lower rate than $n!$ and is for $n \gg 3$ considerably smaller, the storage requirements of $2^n - 1$ results still grow fast, limiting the maximal problem size severely. For example, for $n = 30$ more than one billion results have to be calculated and stored, clearly a number too large for the main memory capacity of most current computers.

Branch-and-bound has only very moderate storage requirements. The forward-branching procedure (**?**) starts to build partial permutations from left (first position) to right. At each step, it is checked if the permutation is valid and several fathoming tests are performed to check if the algorithm should continue with the partial permutation. The most important fathoming test is the boundary test, which checks if the partial permutation can possibly lead to a complete permutation with a better solution than the currently best one. In this way large parts of the search space can be omitted. However, in contrast to the dynamic programming approach, the reduction of search space is strongly data dependent and poorly structured data can lead to very poor performance. With branch-and-bound slightly larger problems can be solved than with dynamic programming in reasonable time. **?** state that depending on the data, in some cases proximity matrices with 40 or more objects can be handled with current hardware.

Partial enumeration methods can be used to find the exact solution independently of the loss/merit function. However, partial enumeration is limited to only relatively small problems.

3.2 Traveling salesperson problem solver

Seriation by minimizing the length of a Hamiltonian path through a graph is equal to solving a traveling salesperson problem. The traveling salesperson or salesman problem (TSP) is a well known and well researched combinatorial optimization problem (see, e.g., **?**). The goal is to find the shortest tour that, starting from a given city, visits each city in a given list exactly once and then returns to the starting city. In graph theory a TSP tour is called a *Hamiltonian cycle*. But for the seriation problem, we are looking for a Hamiltonian path. **?** described a simple transformation of the TSP to find the shortest Hamiltonian path. An additional row and column of 0's is added (sometimes this is referred to as a *dummy city*) to the original $n \times n$ dissimilarity matrix $\mathbf{D}$. The solution of this $(n+1)$-city TSP, gives the shortest path where the city representing the added row/column cuts the cycle into a linear path.

As the general seriation problem, solving the TSP is difficult. In the seriation case with $n+1$ cities, $n!$ tours have to be checked. However, despite this vast searching space, small instances can be solved efficiently using dynamic programming (**?**) and larger instances of several hundred objects can be solved using *branch-and-cut* algorithms (**?**). For even larger instances or if running time is critical, a wide array of heuristics are available, ranging from simple nearest neighbor approaches to construct a tour (**?**) to complex heuristics like the Lin-Kernighan heuristic (**?**). A comprehensive overview of heuristics and exact methods can be found in **?**.

3.3 Bond energy algorithm

The *bond energy algorithm* (BEA; **?**) is a simple heuristic to rearrange columns and rows of a matrix (two-way two-mode data) such that each entry is as closely numerically related to its four neighbors as possible. To achieve this, BEA tries to maximize the measure of effectiveness (ME) defined in Section 2.6. For optimizing the ME, columns and rows can be treated separately since changing the order of rows does not influence the ME contributions of the columns and vice versa. BEA consists of the following three steps:

1. Place one randomly chosen column.
2. Try to place each remaining column at each possible position left, right and between the already placed columns and calculate every time the increase in ME. Choose the column and position which gives the largest increase in ME and place the column. Repeat till all columns are placed.
3. Repeat procedure with rows.

This greedy algorithm works fast and only depends on the choice of the first column/row. This dependence can be reduced by repeating the procedure several times with different choices and returning the solution with the highest ME.

Although **?** use BEA also for non-binary data, **?** argue that the measure of effectiveness only serves its intended purpose of finding an arrangement which is close to Robinson form for binary data and should therefore only be used for binary data.

? notes that the optimization problem of BEA can be stated as two independent traveling salesperson problems (TSPs). For example, the row TSP for an $n \times m$ matrix $\mathbf{X}$ consists of n cities with an $n \times n$ distance matrix $\mathbf{D}$ where the distances are

$$d_{ij} = -\sum_{k=1}^{m} x_{ik} x_{jk}.$$

BEA is in fact a simple suboptimal TSP heuristic using this distances and instead of BEA any TSP solver can be used to obtain an order. With an exact TSP solver, the optimal solution can be found.

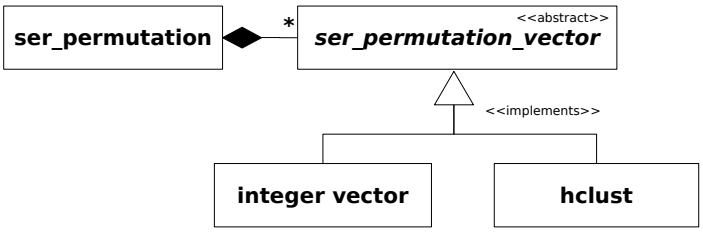


Figure 1: UML class diagram of the data structures for permutations provided by **seriation**

3.4 Hierarchical clustering

Hierarchical clustering produces a series of nested clusterings which can be visualized by a dendrogram, a tree where each internal node represents a split into subtrees and has a measure of similarity/dissimilarity attached to it. As a simple heuristic to find a linear order of objects, the order of the leaf nodes in a dendrogram structure can be used. This idea is used, e.g., by heat maps to reorder rows and columns with the aim to place more similar objects and variables closer together.

The order of leaf nodes in a dendrogram is not unique. A binary (two-way splits only) dendrogram for n objects has 2^{n-1} internal nodes and at each internal node the left and right subtree (or leaves) can be swapped resulting in 2^{n-1} distinct leaf orderings. To find a unique or optimal order, an additional criterion has to be defined. **?** suggest to obtain a unique order by requiring to order the leaf nodes such that at each level the objects at the edge of each cluster are adjacent to that object outside the cluster to which it is nearest.

? suggest to rearrange the dendrogram such that the Hamiltonian path connecting the leaves is minimized and called this the optimal leaf order. The authors also present a fast algorithm with time complexity $O(n^4)$ to solve this optimization problem. Note that this problem is related to the TSP described above, however, the given dendrogram structure significantly reduces the number of permissible permutations making the problem easier.

Although hierarchical clustering solves an optimization problem different to the seriation problem discussed in this paper, hierarchical clustering still can produce useful orderings, e.g., for visualization.

4 The package infrastructure

The **seriation** package provides the data structures and some algorithms to efficiently handle seriation with R. As the input data for seriation R already provides

- for two-way one-mode data the class `dist`,
- for two-way two-mode data the class `matrix`, and
- for k-way k-mode data the class `array`.

However, R provides no classes for representing permutation vectors. **seriation** adds the necessary data structure (using the S3 class system) as depicted in the UML class diagram (**?**) in Figure 1. In this diagram classes are represented by rectangles and different symbols are used to state the type of relationship between the classes. The class `ser_permutation` in Figure 1 represents the permutation information for k-mode data (including the cases of $k = 1$ and $k = 2$). It consists of k permutation vectors (class `ser_permutation_vector`). This relationship is represented by the solid diamond and the star above the connection between the two classes. Class `ser_permutation_vector` is defined *abstract* and only its concrete implementations (classes connected with the triangle symbol) are used to store a permutation vector. This design with an abstract class was chosen to allow to use different representations for the permutation vectors. Currently, the permutation vector can be stored as a simple integer vector or as an object of class `hclust` (defined in package **stats**). `hclust` describes

Algorithm	`method` argument	Optimizes	Input data
Simulated annealing	`"ARSA"`	Gradient measure	`dist`
Branch-and-bound	`"BBURCG"`	Gradient measure	`dist`
Branch-and-bound	`"BBWRCG"`	Gradient measure (weighted)	`dist`
TSP solver	`"TSP"`	Hamiltonian path length	`dist`
Optimal leaf ordering	`"OLO"`	Hamiltonian path length (restricted)	`dist`
Bond Energy Algorithm	`"BEA"`	Measure of effectiveness	`matrix`
TSP to optimize ME	`"BEA_TSP"`	Measure of effectiveness	`matrix`
Hierarchical clustering	`"HC"`	Other	`dist`
Gruvaeus and Wainer	`"GW"`	Other	`dist`
Rank-two ellipse seriation	`"Chen"`	Other	`dist`
MDS – first dimension	`"MDS"`	Other	`dist`
First principal component	`"PCA"`	Other	`matrix`

Table 1: Currently implemented methods for `seriation()`.

a hierarchical clustering tree (dendrogram) including an ordering for the tree's node leaves which provides a permutation for all objects (see Section 3.4).

Class `ser_permutation_vector` has a constructor `ser_permutation_vector()` which converts data into the correct concrete subclass of `ser_permutation_vector` and checks if it contains a proper permutation vector. For `ser_permutation_vector` the methods `print()`, `length()` for the length of the permutation vector, `get_method()` to get the method used to generate the permutation, and `get_order()` to access the raw (integer) permutation vector are available. To use an additional class to represent permutations as a concrete subclass of `ser_permutation_vector` only an appropriate accessor method `get_order()` has to be implemented for the new class.

For `ser_permutation` a constructor is provided which can bind k `ser_permutation_vector` objects together into an object for k-mode data. `ser_permutation` is implemented as a list of length k and each element contains a `ser_permutation_vector` object. Methods like `length()`, accessing elements with `[[`, `[[<-`, subsetting with `[`, and combining with `c()` work as expected. Also a `print()` method is provided. Finally, direct access to the raw permutation vectors is available using `get_order()`. Here a second argument (which defaults to 1) specifies the dimension (mode) for which the order vector is requested.

All seriation algorithms are available via the function `seriate()` defined as:

```
seriate(x, method = NULL, control = NULL, ...)
```

where `x` is the input data, `method` is a string defining the seriation method to be used and `control` can contain a list with additional information for the algorithm. `seriate()` returns an object of class `ser_permutation` with a length conforming to the number of dimensions of `x`. Typical input data are a dissimilarity matrix (class `dist`; see package **stats** for more information) for one-mode two-way data, `matrix` for two-mode two-way data and `array` for k-mode k-way data. For `matrix` and `array` the additional argument `margin` can be used to restrict the dimensions which should be seriated (e.g., with `margin = 1` only the first dimension, i.e., the columns of a matrix, are seriated).

Various seriation methods were already introduced in this paper in Section 3. In Table 1 we summarize the methods currently available in the package for seriation. The code for the simulated annealing heuristic (**?**) and the two branch-and-bound implementations (**?**) was obtained from the authors. The TSP solvers (exact solvers and a variety of heuristics) is provided by package **TSP** (**??**). For optimal leaf ordering we implemented the algorithm by **?**. The BEA code was kindly provided by Fionn Murtagh. For the Gruvaeus and Wainer algorithm, the implementation in package **gclus** (**?**) is used. For the rank-two ellipse seriation we implemented the algorithm by **?**. Note that some methods implemented (e.g., the rank-two ellipse seriation) do not fall within the combinatorial optimization framework of this paper and thus are not dealt with here in detail. They are included in the package since they can be useful for various applications. Over time more methods will be added to the package.

To calculate the value of a loss/merit function for data and a certain permutation, the function

Name	`method` argument	merit/loss	Input data
Anti-Robinson events	`"AR_events"`	loss	`dist`
Anti-Robinson deviations	`"AR_deviations"`	loss	`dist`
Gradient measure	`"Gradient_raw"`	merit	`dist`
Gradient measure (weighted)	`"Gradient_weighted"`	merit	`dist`
Hamiltonian path length	`"Path_length"`	loss	`dist`
Inertia criterion	`"Inertia"`	merit	`dist`
Least squares criterion	`"Least_squares"`	loss	`dist`
Measure of effectiveness	`"ME"`	merit	`matrix`
Stress (Moore neighborhood)	`"Moore_stress"`	loss	`matrix`
Stress (Neumann neighborhood)	`"Neumann_stress"`	loss	`matrix`

Table 2: Implemented loss/merit functions in function `criterion()`.

```
criterion(x, order = NULL, method = NULL, ...)
```

is provided. `x` is the data object, `order` contains a suitable object of class `ser_permutation` (if omitted no permutation is performed) and `method` specifies the type of loss/merit function. A vector of several methods can be used resulting in a named vector with the values of the requested functions. If `method` is omitted (`method = NULL`), the values for all applicable loss/merit functions are calculated and returned. We already defined different loss/merit functions for seriation in Section 2. In Table 2 we indicate the loss/merit functions currently available in the package.

All methods for `seriate()` and `criterion()` are managed by a registry mechanism which makes the seriation framework easily extensible for users. For example, a new seriation method can be registered using `set_seriation_method()` and then used in the same way as the built-in methods with `seriate()`. All available methods in the registry can be viewed using `list_seriation_methods()` and `show_seriation_methods()`. For criterion methods, the same interface is available by just substituting 'seriation' by 'criterion' in the function names. An example for how to add new methods can be found in section 5.3 of this paper.

In addition the package offers the (generic) function

```
permute(x, order)
```

where `x` is the data (a `dist` object, a matrix, an array, a list or a numeric vector) to be reordered and `order` is a `ser_permutation` object of suitable length.

For visualization, the package offers several options:

- Matrix shading with `pimage()`. In contrast to the standard `image()` in package **graphics**, `pimage()` displays the matrix as is with the first element in the top left-hand corner and using a gamma-corrected gray scale.
- Different heat maps (e.g., with optimally reordered dendrograms) with `hmap()`.
- Visualization of data matrices in the spirit of `?` with `bertinplot()`.
- *Dissimilarity plot*, a new visualization to judge the quality of a clustering using matrix shading and seriation with `dissplot()`.

We will introduce the package usage and the visualization options in the examples in the next section.

5 Examples and applications

We start this section with a simple first session to demonstrate the basic usage of the package. Then we present and discuss several seriation applications.

5.1 A first session using seriation

In the following example, we use the well known iris data set (from R's **datasets** package) which gives the measurements in centimeters of the variables sepal length and width and

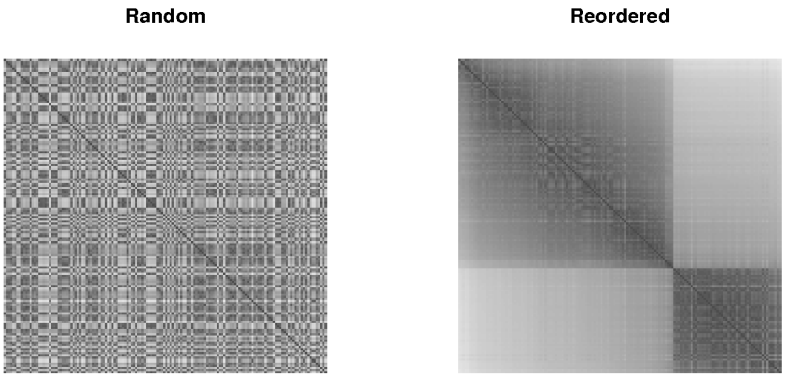


Figure 2: Matrix shading of the distance matrix for the iris data.

petal length and width, respectively, for 50 flowers from each of 3 species of the iris family (Iris setosa, versicolor and virginica).

First, we load the package **seriation** and the iris data set. We remove the species classification and reorder the objects randomly since they are already sorted by species in the data set. Then we calculate the Euclidean distances between objects.

```
> library("seriation")
> data("iris")
> x <- as.matrix(iris[-5])
> x <- x[sample(seq_len(nrow(x))), ]
> d <- dist(x)
```

To seriate the objects given the dissimilarities, we just call `seriate()` with the default settings.

```
> o <- seriate(d)
> o
```

```
object of class 'ser_permutation', 'list'
contains permutation vectors for 1-mode data

  vector length seriation method
1           150             ARSA
```

The result is an object of class `ser_permutation` for one-mode data. The permutation vector length is 150 for the 150 objects in the iris data set and the used seriation method is `"ARSA"`, a simulated annealing heuristic (see Table 1). The actual order can be accessed using `get_order()`. In the following we show the first 15 elements in the permutation vector.

```
> head(get_order(o), 15)
```

```
 [1] 116 126  49 145 144  22  66 140  52   5 110  60  64 137  99
```

To visually inspect the effect of seriation on the distance matrix, we use matrix shading with `pimage()` (the result is shown in Figure 2).

```
> pimage(d, main = "Random")
> pimage(d, o, main = "Reordered")
```

We can also compare the improvement for different loss/merit functions using `criterion()`.

```
> cbind(random = criterion(d), reordered = criterion(d, o))
```

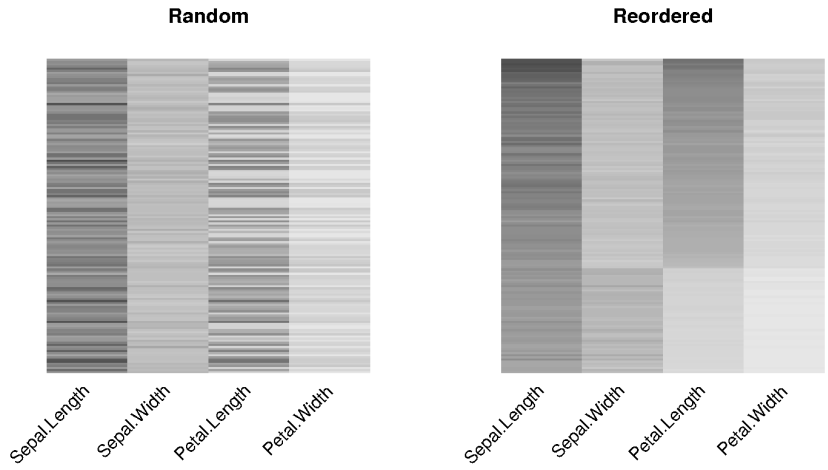


Figure 3: Matrix shading of the iris data matrix.

```
                       random    reordered
AR_deviations        943320.0      9442.40
AR_events            551693.0     54915.00
Gradient_raw          -1392.0    992073.00
Gradient_weighted      9153.5   1772117.91
Inertia           215367939.3 356947608.76
Least_squares      78838267.7  76487648.47
ME                   301406.5    402260.78
Moore_stress         885654.0     14060.78
Neumann_stress       415557.8      5556.31
Path_length             359.5        86.44
```

Naturally, the reordered dissimilarity matrix achieves better values for all criteria. Note that the gradient measures, inertia and the measure of effectiveness are merit functions and for these measures larger values are better (use `show_criterion_methods("dist")` to find out which measures are loss and merit functions).

To visually compare the original data matrix and the result of seriation, we can also use `pimage()`. After using `pimage()` for the original (random) data matrix, we create a suitable `ser_permutation` object for the original two-mode data. Since the seriation above only produced an order for the rows of the data, we add an identity permutation vector for the columns (which leaves them in original order) to the permutations object using the combine function `c()`. This new permutation object for 2-mode data is used for displaying the reordered data. The two plots are shown in Figure 3.

```
> pimage(x, main = "Random")
> o_2mode <- c(o, ser_permutation(seq_len(ncol(x))))
> pimage(x, o_2mode, main = "Reordered")
```

5.2 Comparing different seriation methods

To compare different seriation methods we use again the randomized iris data set and the distance matrix `d` from the previous example. We include in the comparison several seriation methods for dissimilarity matrices described in Section 3.

```
> methods <- c("TSP", "Chen", "ARSA", "HC", "GW", "OLO")
> o <- sapply(methods, FUN = function(m) seriate(d, m), simplify = FALSE)
```

Seriation Method	TSP	Chen	ARSA	HC	GW	OLO
Execution time [sec]	0.396	0.2	4.661	0.028	0.056	0.032

Table 3: Execution time of seriation of the iris data set for different methods (1.5 GHz ix86 system running Ubuntu Linux).

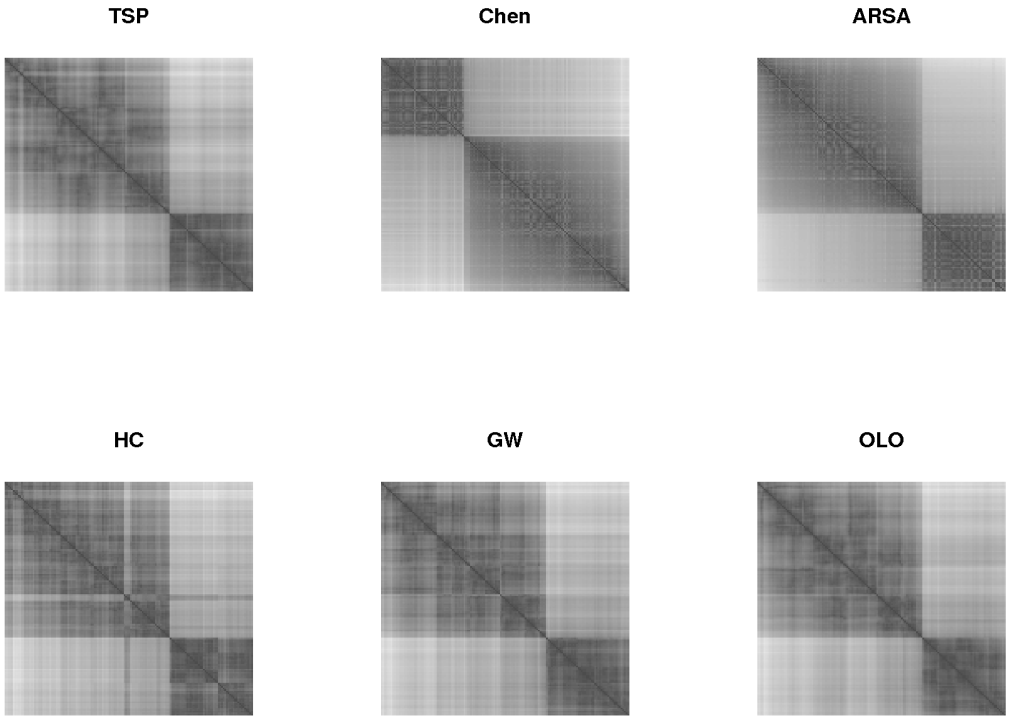


Figure 4: Image plot of the distance matrix for the iris data using rearrangement by different seriation methods.

Table 3 contains the execution times for running seriation with the different methods. Except for the simulated annealing method (ARSA) the seriation only takes a fraction of a second. The resulting orderings are displayed using matrix shading (see Figure 4).

```
> tmp <- lapply(o, FUN = function(x) pimage(d, x, main = get_method(x[[1]])))
```

The first row of matrices in Figure 4 contains the orders obtained by a TSP solver the rank-two ellipse seriation by Chen and using the simulated annealing method (ARSA). The results of Chen and ARSA are very similar (except that the order is reversed). The TSP solver produces a smoother image with some lighter lines visible. The reason for these lines is that the TSP only optimizes distances locally between two neighboring objects. Therefore it is possible that in a quite homogeneous block several objects are enclosed gradually getting more different and then getting more similar again (see, e.g., the light line close to the upper left corner of the TSP image in Figure 4).

The second row of Figure 4 contains three images based on hierarchical clustering. The visual impression gets better from left (just hierarchical clustering) to right (first using the Gruvaeus Wainer heuristic and then optimal leaf ordering to rearrange the branches of the dendrogram obtained by hierarchical clustering). The most striking feature in the image for hierarchical clustering (HC in Figure 4) is the distinct cross going right through the center of the plot. This indicates that several relatively dissimilar objects are caught in an otherwise homogeneous block. This effect vanishes after rearranging the dendrogram branches (see GW and OLO in Figure 4). To investigate this effect, we plot the dendrogram obtained by

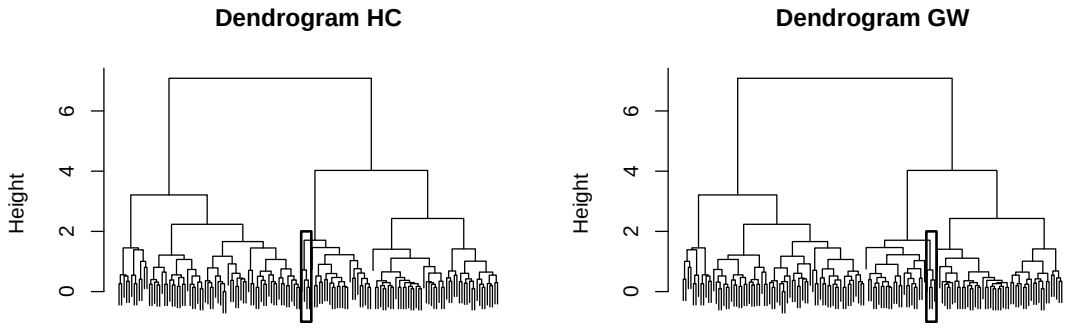


Figure 5: Dendrograms for the seriation with HC and GW.

hierarchical clustering which is used to order the objects and compare it to the dendrogram rearranged using the Gruvaeus Wainer heuristic.

```
> plot(o[["HC"]][[1]], labels = FALSE, main = "Dendrogram HC")
> plot(o[["GW"]][[1]], labels = FALSE, main = "Dendrogram GW")
```

Comparing the two dendrograms in Figure 5, we see that the branch left from the top is almost unchanged. The branch which is responsible for the light cross in the shaded image is highlighted by a box. The Gruvaeus Wainer heuristic rotates the highlighted branch towards the right since the objects in it are more similar to the objects in there.

Finally, we compare the values of the loss/merit functions for the different seriation methods.

```
> crit <- sapply(o, FUN = function(x) criterion(d, x))
> t(crit)
```

```
     AR_deviations AR_events Gradient_raw Gradient_weighted   Inertia
TSP          51174    165453       770981            1651215 345485410
Chen         18184     90015       921885            1746766 354057418
ARSA          9424     55010       991884            1772114 356944288
HC           53167    173729       754425            1644981 345722302
GW           44776    171903       758071            1664667 346564518
OLO          46900    174625       752619            1660309 346172124
     Least_squares     ME Moore_stress Neumann_stress Path_length
TSP       76648852 402674        15294           5238       52.41
Chen      76521451 402018        19358           7305       85.43
ARSA      76487654 402346        13333           5217       85.44
HC        76657164 402560        30347          10401       64.15
GW        76630917 402822        18636           6426       57.24
OLO       76636726 403053        14979           5126       51.11
```

For easier comparison, Figure 6 contains a plot of the criteria Hamiltonian path length, anti-Robinson events (`AR_events`) and stress using the Moore neighborhood. Clearly, the methods which directly try to minimize the Hamiltonian path length (hierarchical clustering with optimal leaf ordering (`OLO`) and the TSP heuristic) provide the best results concerning the path length. For the number of anti-Robinson events, using the simulated annealing heuristic (`ARSA`) provides the best result since it directly aims at minimizing this loss function. Regarding stress, the simulated annealing heuristic also provides the best result although, it does not directly minimize this loss function.

5.3 Registering new methods

New methods to calculate criterion values and to compute a seriation can be easily added by the user via the method registry mechanism provided in **seriation**. Here we give a simple example of how to implement and register a new seriation method.

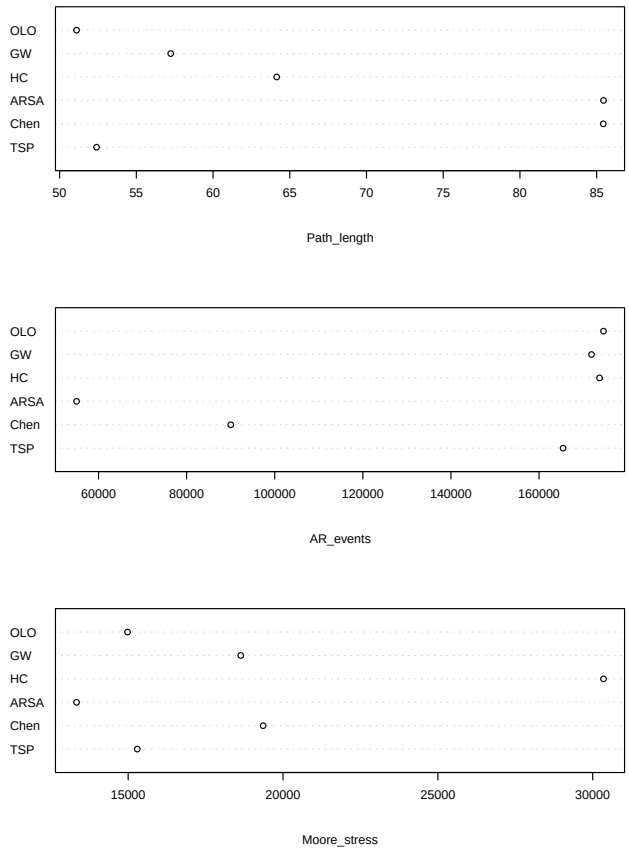


Figure 6: Comparison of different methods and seriation criteria

In the registry we distinguish between methods for different types of input data. With the following two commands we produce a short overview of the available seriation methods for input data of class `dist` and `matrix`.

```
> show_seriation_methods("dist")
```

```
ARSA: Minimize Anti-Robinson events using simulated annealing
BBURCG: Minimize the unweighted row/column gradient by
       branch-and-bound
BBWRCG: Minimize the weighted row/column gradient by
       branch-and-bound
Chen: Rank-two ellipse seriation
GW: Hierarchical clustering reordered by Gruvaeus and Wainer
       heuristic
HC: Hierarchical clustering
MDS: MDS - first dimension
OLO: Hierarchical clustering with optimal leaf ordering
TSP: Minimize Hamiltonian path length with a TSP solver
```

```
> show_seriation_methods("matrix")
```

```
BEA: Bond Energy Algorithm to maximize ME
BEA_TSP: TSP to maximize ME
PCA: First principal component
```

The overview is intended to make it convenient for the user to choose an appropriate method. It contains the name of the method used as the `method` argument for `seriate()` and a short description. To get just the names the following function is also available:

```
> list_seriation_methods("matrix")
```

```
[1] "BEA"     "BEA_TSP" "PCA"
```

To add a new seriation method, we first have to implement the seriation code as a function with the two formal arguments `x` and `control`. `x` is the data object and `control` contains a list with additional information for the method passed on from `seriate()`. The function has to return a list of objects which can be coerced into `ser_permutation_vector` objects (e.g., a list of integer vectors). The elements in the list have to be in order corresponding to the dimensions of `x`.

In this example we just create a method to return a permutation which maintains the original order of the objects, i.e., which returns the identity order.

```
> seriation_method_identity <- function(x, control) {
+     lapply(dim(x), seq)
+ }
```

The function produces integer sequences of the correct lengths, one for each dimension of `x` (`control` is not used). Since the function works for `matrix` and `array` we can register it for both data types under the short name 'identity'.

```
> set_seriation_method("matrix", "identity", seriation_method_identity,
+     "Identity order")
> set_seriation_method("array", "identity", seriation_method_identity,
+     "Identity order")
```

Now the new seriation method is registered and can be found by the user and applied to data.

```
> show_seriation_methods("matrix")
```

```
BEA: Bond Energy Algorithm to maximize ME
BEA_TSP: TSP to maximize ME
identity: Identity order
PCA: First principal component
```

```
> o <- seriate(matrix(1, ncol = 3, nrow = 4), "identity")
> o

object of class 'ser_permutation', 'list'
contains permutation vectors for 2-mode data

  vector length seriation method
1             4         identity
2             3         identity

> get_order(o, 1)

[1] 1 2 3 4

> get_order(o, 2)

[1] 1 2 3
```

Criterion methods can be added in the same way. We refer the interested reader to the documentation accompanying the package for detailed information and an example.

If you have implemened a new criterion or seriation method, please consider submitting the code to one of the maintainers of **seriation** for inclusion in a future release of the package.

5.4 Heat maps

A heat map is a shaded/color coded data matrix with a dendrogram added to one side and to the top to indicate the order of rows and columns. Typically, reordering is done according to row or column means within the restrictions imposed by the dendrogram. Heat maps recently became popular for visualizing large scale genome expression data obtained via DNA microarray technology (see, e.g., **?**).

From Section 3.4 we know that it is possible to find the optimal ordering of the leaf nodes of a dendrogram which minimizes the distances between adjacent objects in reasonable time. Such an order might provide an improvement over using simple reordering such as the row or column means with respect to presentation. In **seriation** we provide the function `hmap()` which uses optimal ordering and can also use seriation directly on distance matrices without using hierarchical clustering to produce dendrograms first.

For the following example, we use again the randomly reordered iris data set `x` from the examples above. To make the variables (columns) comparable, we use standard scaling.

```
> x <- scale(x, center = FALSE)
```

To produce a heat map with optimally reordered dendrograms, the function `hmap()` can be used with its default settings.

```
> hmap(x)
```

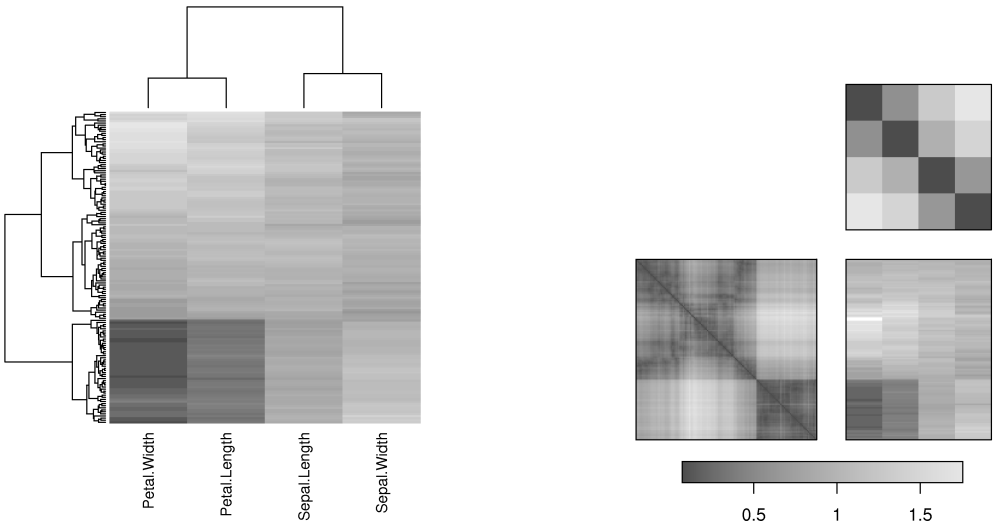
With these settings, the Euclidean distances between rows and between columns are calculated (with `dist()`), hierarchical clustering (`hclust()`) is performed, the resulting dendrograms are optimally reordered, and `heatmap()` in package **stats** is used for plotting (see Figure 7(a) for the resulting plot).

```
> hmap(x, hclustfun = NULL)
```

If `hclustfun = NULL` is used, instead of hierarchical clustering, seriation on the dissimilarity matrices for rows and columns is performed (using a TSP heuristic by default) and the reordered matrix with the reordered dissimilarity matrices to the left and on top is displayed (see Figure 7(b)). A `method` argument can be used to choose different seriation methods.

5.5 Bertin's permutation matrix

?? introduced permutation matrices to analyze multivariate data with medium to low sample size. The idea is to reveal a more homogeneous structure in a data matrix $\mathbf{X}$ by simultaneously rearranging rows and columns. The rearranged matrix is displayed and cases and variables can be grouped manually to gain a better understanding of the data.



(a) (b)

Figure 7: Two presentations of the rearranged iris data matrix. (a) as an optimally reordered heat map and (b) as a seriated data matrix with reordered dissimilarity matrices to the left and on top.

To find a rearrangement of columns and rows which reveals structure a purity function is used. A possible purity function is: Given distances between rows and columns of the data matrix, define purity as the sum of distances of adjacent rows/columns. Using this purity function, finding the optimal permutation means solving two (independent) TSPs, one for the columns and one for the rows which can be done very conveniently using the infrastructure provided by **seriation**.

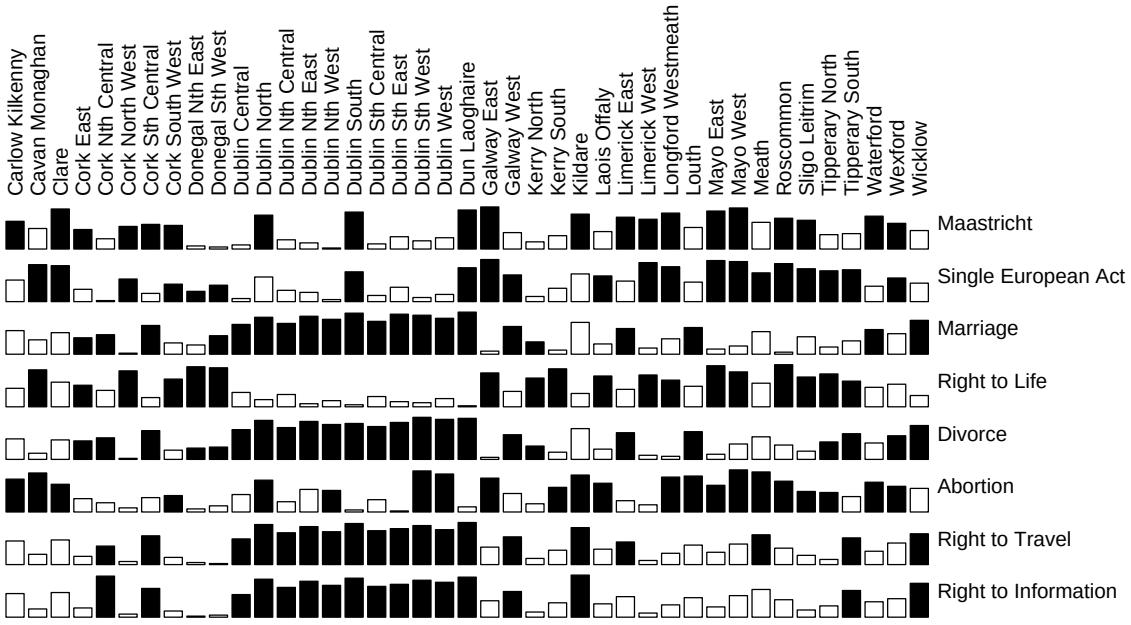
As an example, we use the results of 8 constitutional referenda for 41 Irish communities (**?**)[1]. To make values comparable across columns (variables), the ranks of the values for each variable are used instead of the original values.

```
> data("Irish")
> orig_matrix <- apply(Irish[, -6], 2, rank)
```

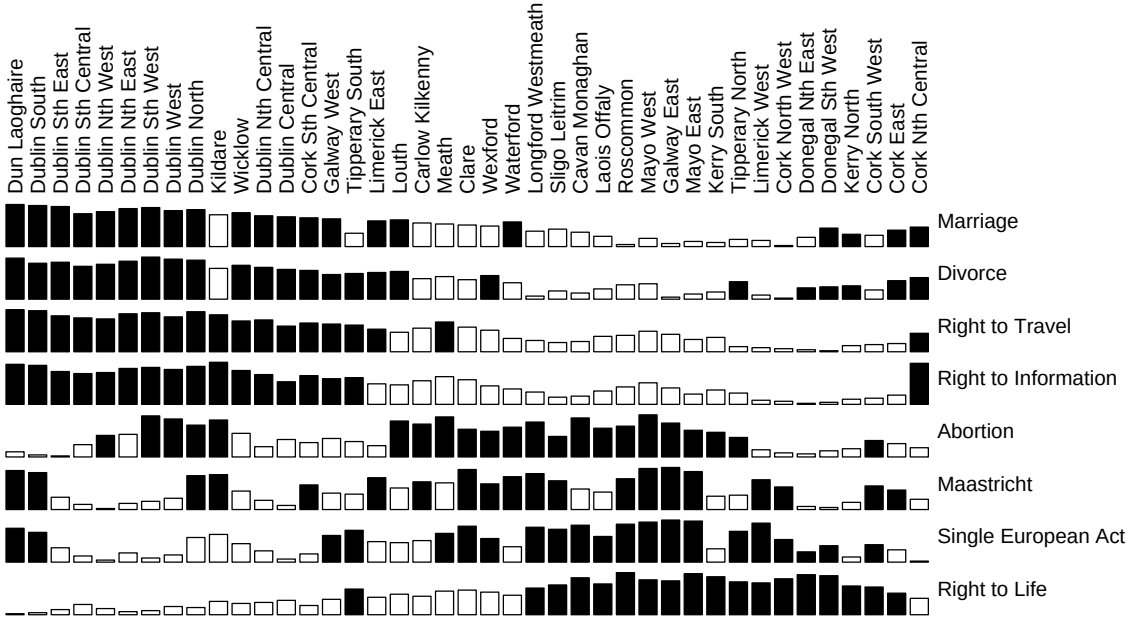
For seriation, we calculate distances between rows and between columns using the sum of absolute rank differences (this is equal to the Minkowski distance with power 1). Then we apply seriation (using a TSP heuristic) to both distance matrices and combine the two resulting `ser_permutation` objects into one object for two-mode data. The original and the reordered matrix are plotted using `bertinplot()`.

```
> o <- c(seriate(dist(orig_matrix, "minkowski", p = 1), method = "TSP"),
+     seriate(dist(t(orig_matrix), "minkowski", p = 1), method = "TSP"))
> o

object of class 'ser_permutation', 'list'
contains permutation vectors for 2-mode data

  vector length seriation method
1            41              TSP
2             8              TSP

> bertinplot(orig_matrix)
> bertinplot(orig_matrix, o)
```

[1]The Irish data set is included in this package. The original data and the text of the referenda can be obtained from `http://www.electionsireland.org/`



(a)



(b)

Figure 8: Bertin plot for the (a) original arrangement and the (b) reordered Irish data set.

The original matrix and the rearranged matrix are shown in Figure 8 as a matrix of bars where high values are highlighted (filled blocks). Note that following Bertin, the cases (communities) are displayed as the columns and the variables (referenda) as rows. Depending on the number of cases and variables, columns and rows can be exchanged to obtain a better visualization.

Although the columns are already ordered (communities in the same city appear consecutively) in the original data matrix in Figure 8(a), it takes some effort to find structure in the data. For example, it seems that the variables 'Marriage', 'Divorce', 'Right to Travel' and 'Right to Information' are correlated since the values are all high in the block made up by the columns of the communities in Dublin. The reordered matrix confirms this but makes the structure much more apparent. Especially the contribution of low values (which are not highlighted) to the overall structure becomes only visible after rearrangement.

5.6 Binary data matrices

Binary or 0-1 data matrices are quite common. Often such matrices are called *incidence matrices* since a 1 in a cell indicates the incidence of an event. In archeology such an event could be that a special type of artifact was found at a certain archaeological site. This can be seen as a simplification of a so-called *abundance matrix* which codes in each cell the (relative) frequency or quantity of an artifact type at a site. See **?** for a comparison of incidence and abundance matrices in archeology.

Here we are interested in binary data. For the example we use an artificial data set from **?** called *Townships*. The data set contains 9 binary characteristics (e.g., has a veterinary or has a high school) for 16 townships. The idea of the data set is that townships evolve from a rural to an urban environment over time.

After loading the data set (which comes with the package), we use `bertinplot()` to visualize the data (`pimage()` could also be used but `bertinplot()` allows for a nicer visualization). Bars, the standard visualization of `bertinplot()`, do not make much sense for binary data. We therefore use the panel function `panel.squares()` without spacing to plot black squares.

```
> data("Townships")
> bertinplot(Townships, options = list(panel = panel.squares,
+     spacing = 0, frame = TRUE))
```

The original data in Figure 9(a) does not reveal structure in the data. To improve the display, we run the bond energy algorithm (BEA) for columns and rows 10 times with random starting points and report the best solution.

```
> o <- seriate(Townships, method = "BEA", control = list(rep = 10))
> bertinplot(Townships, o, options = list(panel = panel.squares,
+     spacing = 0, frame = TRUE))
```

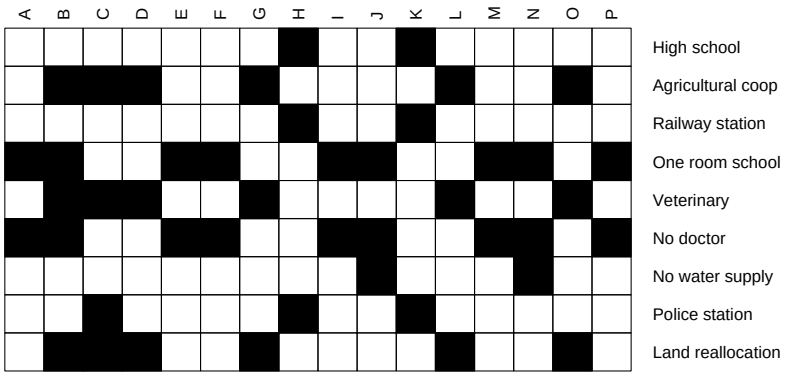
The reordered matrix is displayed in Figure 9(b). A clear structure is visible. The variables (rows in a Bertin plot) can be split into the three categories describing different evolution states of townships:

1. Rural: No doctor, one-room school and possibly also no water supply
2. Intermediate: Land reallocation, veterinary and agricultural cooperative
3. Urban: Railway station, high school and police station

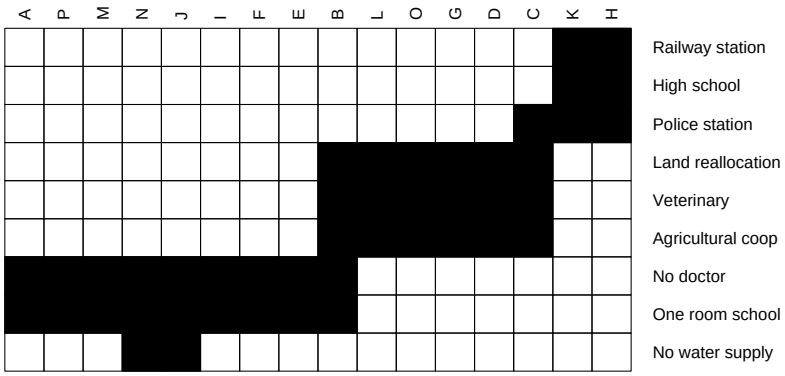
The townships also clearly fall into these three groups which tentatively can be called villages (first 7), towns (next 5) and cities (final 2). The townships B and C are on the transition to the next higher group.

```
> rbind(original = criterion(Townships), reordered = criterion(Townships,
+     o))
          ME Moore_stress Neumann_stress
original  19          464            260
reordered 65          212             82
```

BEA tries to maximize the measure of effectiveness which is much higher in the reordered matrix (in fact, 65 is the maximum for the data set). Also the two types of stress are improved significantly.



(a)



(b)

Figure 9: The townships data set in original order (a) and reordered using BEA (b).

5.7 Dissimilarity plot

Assessing the quality of an obtained cluster solution has been a research topic since the invention of cluster analysis. This is especially important since all popular cluster algorithms produce a clustering even for data without a "cluster" structure.

Matrix shading is an old technique to visualize clusterings by displaying the rearranged matrices (see, e.g., **???**). Initially matrix shading was used in connection with hierarchical clustering, where the order of the dendrogram leaf nodes was used to arrange the matrix. However, with some extensions, matrix shading can also be used with any partitional clustering method.

? suggest a matrix shading visualization called *CLUSION* where the dissimilarity matrix is arranged such that all objects pertaining to a single cluster appear in consecutive order in the matrix. The authors call this *coarse seriation*. The result of a "good" clustering should be a matrix with low dissimilarity values forming blocks around the main diagonal. However, using coarse seriation, the order of the clusters has to be predefined and the objects within each cluster are unordered.

The dissimilarity plots implemented by the function `dissplot()` in **seriation** improve *CLUSION* using seriation methods. It aims at visualizing global structure (similarity between different clusters is reflected by their position relative to each other) as well as the micro structure within each cluster (position of objects).

To position the clusters in the dissimilarity plot, an inter-cluster dissimilarity matrix is calculated using the average between cluster dissimilarities. `seriate()` is used on this inter-cluster dissimilarity matrix to arrange the clusters relative to each other resulting in on average more similar clusters to appear closer together in the plot. Within each cluster, `seriate()` is used again on the sub-matrix of the dissimilarity matrix concerning only the objects in the cluster.

For the example, we use again Euclidean distance between the objects in the iris data set.

```
> data("iris")
> iris <- iris[sample(seq_len(nrow(iris))), ]
> d <- dist(as.matrix(iris[-5]), method = "euclidean")
```

First, we use `dissplot()` without a clustering. We set `method` to `NA` to prevent reordering and display the original matrix (see Figure 10(a)). Then we omit the method argument which results in using the default seriation technique from `seriate()`. Since we did not provide a clustering, the whole matrix is reordered in one piece. From the result shown in Figure 10(b) it seems that there is a clear structure in the data which suggests a two cluster solution.

```
> dissplot(d, method = NA)
```

```
> dissplot(d, options = list(main = "Dissimilarity plot with seriation"))
```

Next, we create a cluster solution using the k-means algorithm. Although we know that the data set should contain 3 groups representing the three species of iris, we let k-means produce a 10 cluster solution to study how such a misspecification can be spotted using `dissplot()`.

```
> l <- kmeans(d, 10)$cluster
```

We create a standard dissimilarity plot by providing the cluster solution as a vector of labels. The function rearranges the matrix and plots the result. Since rearrangement can be a time consuming procedure for large matrices, the rearranged matrix and all information needed for plotting is returned as the result.

```
> res <- dissplot(d, labels = l, options = list(main = "Dissimilarity plot - standard"))
```

```
> res
```

```
object of class 'cluster_dissimilarity_matrix'
matrix dimensions: 150 x 150
dissimilarity measure: 'euclidean'
number of clusters k: 10

cluster description
```

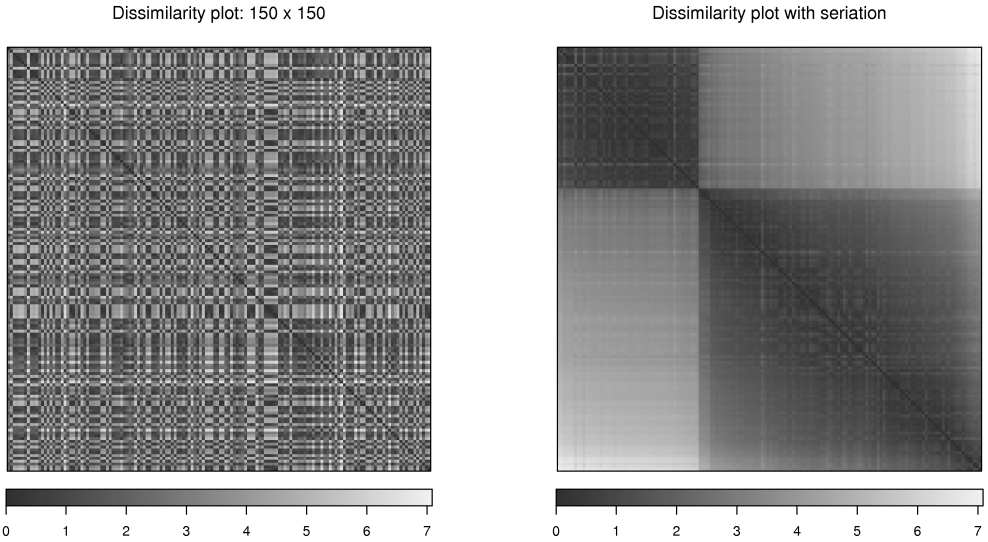


Figure 10: Two dissimilarity plots. (a) the original dissimilarity matrix and (b) the seriated dissimilarity matrix.

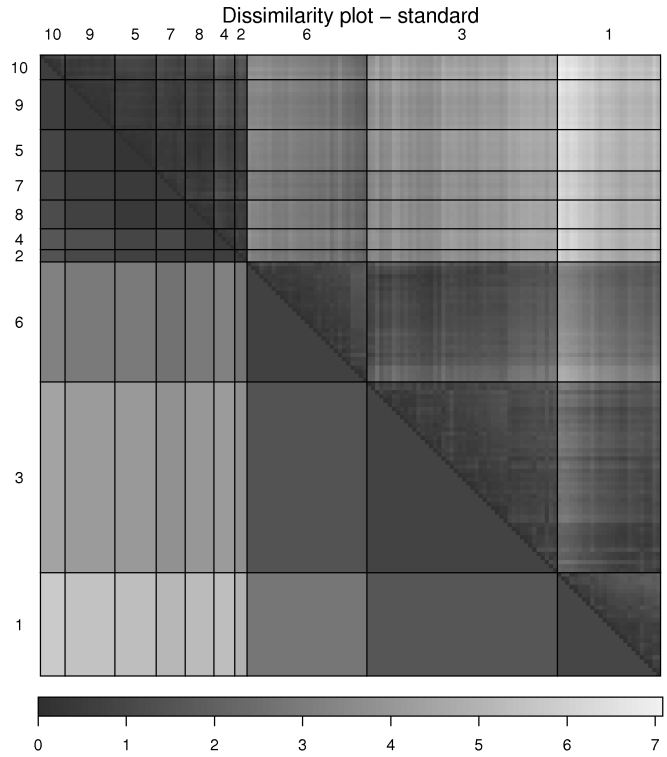
```
   position label size avg_dissimilarity avg_silhouette_width
1         1    10    6             0.9794               0.03654
2         2     9   12             0.8411               0.36239
3         3     5   10             0.3124               0.34195
4         4     7    7             0.4487              -0.08724
5         5     8    7             0.7754               0.18984
6         6     4    5             0.4663               0.14042
7         7     2    3             0.2942               0.10950
8         8     6   29             0.4939               0.43920
9         9     3   46             0.2396               0.31231
10       10     1   25             0.5795               0.34635

used seriation methods
inter-cluster: 'ARSA'
intra-cluster: 'ARSA'
```

The resulting plot is shown in Figure 11(a). The inter-cluster dissimilarities are shown as solid gray blocks and the average object dissimilarity within each cluster as gray triangles below the main diagonal of the matrix. Since the clusters are arranged such that more similar clusters are closer together, it is easy to see in Figure 11(a) that clusters 6, 3 and 1 as well as clusters 10, 9, 5, 7, 8, 4 and 2 are very similar and form two blocks. This suggests again that a two cluster solution would be reasonable.

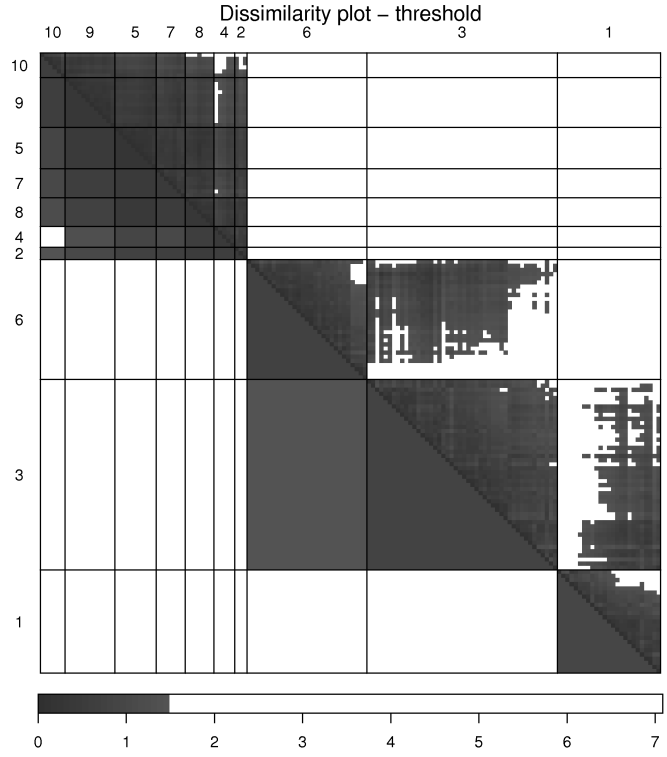
Since slight variations of gray values are hard to distinguish, we plot the matrix again (using `plot()` on the result above) and use a threshold on the dissimilarity to suppress high dissimilarity values in the plot.

```
> plot(res, options = list(main = "Seriation - threshold",
+     threshold = 1.5))
```

In the resulting plot in Figure 11(b), we see that the block containing 10, 9, 5, 7, 8, 4 and 2 is very well defined and cleanly separated from the other block. This suggests that these clusters should form together a cluster in a solution with less clusters. The other block is less well defined. There is considerable overlap between clusters 6 and 3, but also cluster 3 and 1 share similar objects.



(a)



(b)

Figure 11: Dissimilarity plot for k-means solution with 10 clusters. (a) standard plot and (b) plot with threshold.

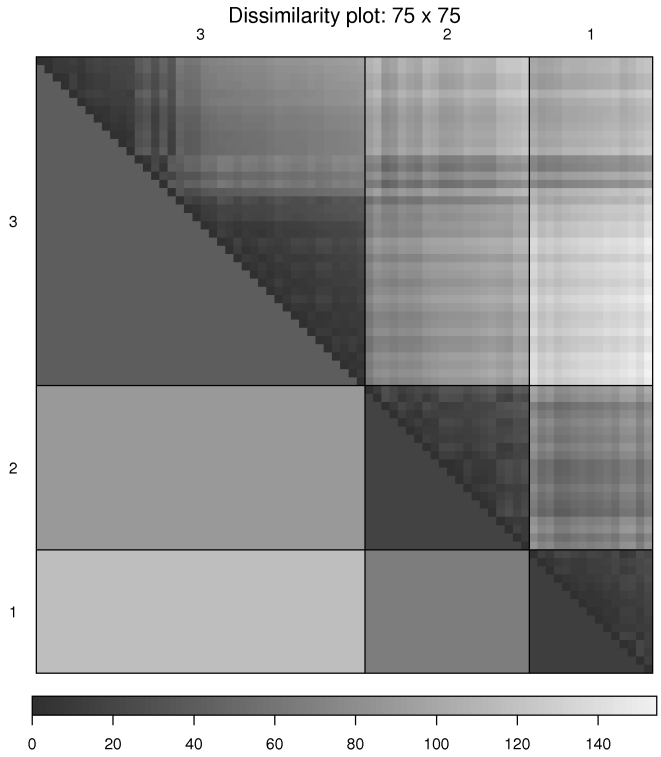


Figure 12: Dissimilarity plot for k-means solution with 3 clusters for the Ruspini data set with 4 groups.

Using the information stored in the result of `dissplot()` and the class information available for the iris data set, we can analyze the cluster solution and the interpretations of the dissimilarity plot.

```
> table(iris[res$order, 5], res$label)[, res$cluster_order]
```

```
            10  9  5 7 8 4 2  6  3  1
  setosa     6 12 10 7 7 5 3  0  0  0
  versicolor 0  0  0 0 0 0 0 28 22  0
  virginica  0  0  0 0 0 0 0  1 24 25
```

As the plot in Figure 11 indicated, the clusters 10, 9, 5, 7, 8, 4 and 2 should be a single cluster containing only flowers of the species Iris setosa. The clusters 6, 3 and 1 are more problematic since they contain a mixture of Iris versicolor and virginica.

To illustrate the results of the dissimilarity plot in case a clustering with a k smaller than the actual number of groups in the data is used, we use the Ruspini data set which consists of 75 points in four groups and is also often used to illustrate clustering techniques. We load the data set, calculate distances, perform k-means clustering with $k = 3$ (although the real number of groups is 4) and produce a dissimilarity plot.

```
> data("ruspini")
> d <- dist(ruspini)
> l <- kmeans(d, 3)$cluster
> dissplot(d, labels = l)
```

The dissimilarity plot in Figure 12 shows that cluster 3 actually should be two separate clusters represented by the two clearly visible darker triangles next to the main diagonal.

The dissimilarity plot using seriation is a useful tool to inspect the result of clustering. It is especially useful to spot misspecifications of the number of clusters employed.

6 Conclusion

In this paper we presented the infrastructure provided by the package **seriation**. The infrastructure contains the necessary data structures to store the linear order for one-, two- and k-mode data. It also provides a wide array of seriation methods for different input data, e.g., dissimilarities, binary and general data matrices focusing on combinatorial optimization. New seriation methods can be easily incorporated into the **seriation** framework by the user with the method registry mechanism provided.

Based on seriation, **seriation** features several visualization techniques. In particular, the optimally reordered heat map, the Bertin plot and the dissimilarity plot present clear improvements over standard plots.

A natural extension to **seriation** is the synthesis of ensembles of seriations into a "consensus" one. Such ensembles do not only arise when using different seriation methods, but also when varying data or control parameters to obtain more robust solutions (see e.g. **?** for a recent application of such ideas in a molecular profiling context). The R extension package **relations** (**?**) contains a variety of methods for obtaining consensus *relations*, covering consensus seriation (where the relations are linear orders on the objects) as a special case.

Future work on **seriation** will focus on adding further seriation methods, such as for example methods for higher dimensional arrays and methods for block seriation which aim at finding simultaneous partitions of rows and columns in a data matrix (see, e.g., **?**).

Acknowledgments

The authors would like to thank Michael Brusco, Hans-Friedrich Köhn and Stephanie Stahl for their seriation code, Fionn Murtagh for his BEA implementation and the anonymous reviewers for their valuable comments and suggestions.