

Getting Things in Order: An Introduction to the R Package *seriation*

Michael Hahsler
Southern Methodist University

Kurt Hornik
Wirtschaftsuniversität Wien

Christian Buchta
Wirtschaftsuniversität Wien

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.

Keywords: combinatorial data analysis, seriation, permutation, R.

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* (Arabie and Hubert 1996). 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, Hubert, Arabie, and Meulman (2001) propose dynamic programming and Brusco and Stahl (2005) 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. Petrie (1899) 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, [Robinson \(1951\)](#), [Kendall \(1971\)](#) 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 [Ihm \(2005\)](#).

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** ([Chessel, Dufour, and Dray 2007](#); [Dray and Dufour 2007](#)) and **vegan** ([Oksanen, Kindt, Legendre, and O'Hara 2007](#))). 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.

A previous version of this manuscript was published in the *Journal of Statistical Software* ([Hahsler, Hornik, and Buchta 2008](#)).

2. Seriation as a combinatorial optimization problem

To seriate a set of n objects $\{O_1, \dots, O_n\}$ 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

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

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, \dots, 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 Robinson (1951). Formally, an $n \times n$ dissimilarity matrix \mathbf{D} is in anti-Robinson form if and only if the following two gradient conditions hold (Hubert *et al.* 2001):

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

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

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 (Hubert *et al.* 2001). 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 Hubert *et al.* (2001) as

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

where $f(\cdot, \cdot)$ is a function which defines how a violation or satisfaction of a gradient condition for an object triple $(O_i, O_k \text{ and } O_j)$ is counted. Hubert *et al.* (2001) suggest two functions.

The first function is given by:

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

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

The second function is defined as:

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

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.

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

To only count the violations we use

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

$I(\cdot)$ is an indicator function returning 1 only for violations. [Chen \(2002\)](#) presented a formulation for an equivalent loss function and called the violations *anti-Robinson events*. [Chen \(2002\)](#) also introduced a weighted versions of the loss function resulting in

$$f(z, y) = |y - z| I(z, y) \quad (9)$$

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., [Hubert 1974](#); [Caraux and Pinloche 2005](#)). 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:

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

Note that the length of the Hamiltonian path is equal to the value of the *minimal span loss function* (as used by [Chen 2002](#)), and both notions are related to the *traveling salesperson problem* ([Gutin and Punnen 2002](#)).

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 ([Caraux and Pinloche 2005](#)) defined as

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

$|i - j|^2$ 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 ([Caraux and Pinloche 2005](#)) defined by

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

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 ([de Leeuw 2005](#)), where the objective is to place all n objects on a straight line using a position vector $\mathbf{z} = z_1, z_2, \dots, z_n$ 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

[McCormick, Schweitzer, and White \(1972\)](#) defined the *measure of effectiveness (ME)* for an $n \times m$ matrix $\mathbf{X} = (x_{ij})$ as

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

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

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

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. [Niermann \(2005\)](#) 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

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

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

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

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:

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

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* (Hubert *et al.* 2001) and *branch-and-bound* (Brusco and Stahl 2005).

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 (Brusco and Stahl 2005) 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. Brusco and Stahl (2005) 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., Gutin and Punnen 2002). 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. Garfinkel (1985) 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 (Held and Karp 1962) and larger instances of several hundred objects can be solved using *branch-and-cut* algorithms (Padberg and Rinaldi 1990). 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 (Rosenkrantz, Stearns, and Philip M. Lewis 1977) to complex heuristics like the Lin-Kernighan heuristic (Lin and Kernighan 1973). A comprehensive overview of heuristics and exact methods can be found in Gutin and Punnen (2002).

3.3. Bond energy algorithm

The *bond energy algorithm* (BEA; McCormick *et al.* 1972) 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 McCormick *et al.* (1972) use BEA also for non-binary data, Arabie and Hubert (1990) 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.

Lenstra (1974) 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 these 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.

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. Gruvaeus and Wainer (1972) 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.

Bar-Joseph, Demaine, Gifford, and Jaakkola (2001) 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 (Fowler 2004) 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

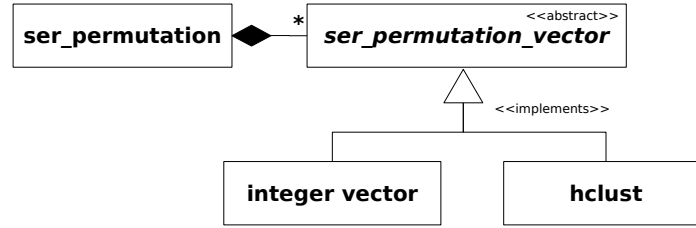


Figure 1: UML class diagram of the data structures for permutations provided by **seriation**

integer vector or as an object of class **hclust** (defined in package **stats**). **hclust** describes 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 (Brusco, Köhn, and Stahl 2008) and the two branch-and-bound implementations (Brusco and Stahl 2005) was obtained from the authors. The TSP solvers (exact solvers and a variety of heuristics) is provided by package **TSP** (Hahsler and

Algorithm	method	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" "OLO_single" "OLO_average" "OLO_complete"	Hamiltonian path length (restricted)	dist
Gruvaeus and Wainer	"GW" "GW_single" "GW_average" "GW_complete"	Hamiltonian path length (restricted)	dist
Bond Energy Algorithm	"BEA"	Measure of effectiveness	matrix
TSP to optimize ME	"BEA_TSP"	Measure of effectiveness	matrix
MDS	"MDS" "MDS_metric" "MDS_nonmetric" "MDS_angle"	Least square crit.	dist
Principal component analysis	"PCA" "PCA_angle"	Least square crit.	matrix
Spectral seriation	"Spectral" "Spectral_norm"	Inertia criterion	dist
Hierarchical clustering	"HC" "HC_single" "HC_average" "HC_complete"	Other	dist
Rank-two ellipse seriation	"R2E"	Other	dist
Sorting Points Into Neighbor- hoods	"SPIN_NH" "SPIN_STS"	Other	dist
Visual Assessment of (Clus- tering) Tendency	"VAT"	Other	dist

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

Name	method	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()`.

Hornik 2007b,a). For optimal leaf ordering we implemented the algorithm by Bar-Joseph *et al.* (2001). The BEA code was kindly provided by Fionn Murtagh. For the Gruvaeus and Wainer algorithm, the implementation in package `gclus` (Hurley 2007) is used. For the rank-two ellipse seriation we implemented the algorithm by Chen (2002). Spectral seriation is described by Ding and He (2004). 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

```
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 Bertin (1981) 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 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
```

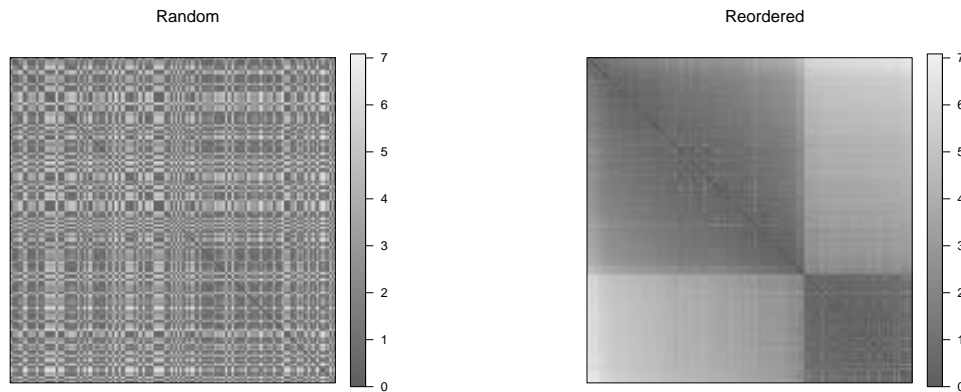


Figure 2: Matrix shading of the distance matrix for the iris data.

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))
```

	random	reordered
AR_events	5.517e+05	5.492e+04
AR_deviations	9.433e+05	9.431e+03
RGAR	5.004e-01	4.981e-02
Gradient_raw	-1.386e+03	9.921e+05
Gradient_weighted	9.154e+03	1.772e+06
Path_length	3.595e+02	8.627e+01
Inertia	2.154e+08	3.569e+08
Least_squares	7.884e+07	7.649e+07
ME	3.014e+05	4.023e+05
Moore_stress	8.857e+05	1.390e+04
Neumann_stress	4.156e+05	5.484e+03

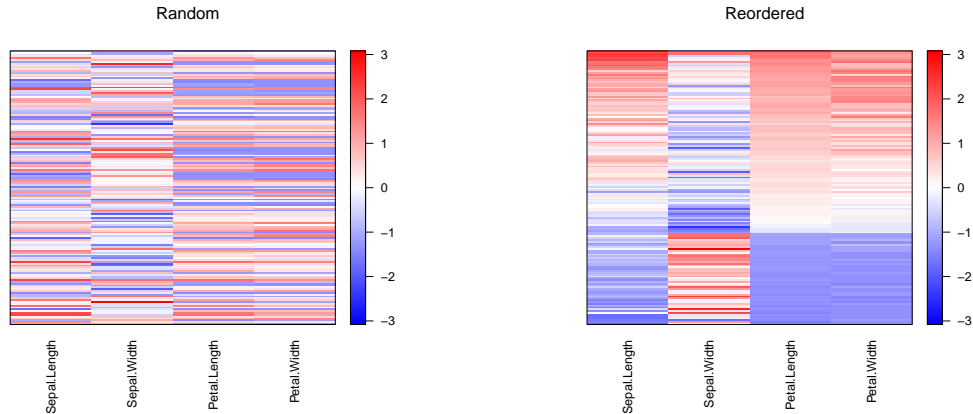


Figure 3: Matrix shading of the iris data matrix.

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()`. We standardize the data using scale such that the visualized value is the number of standard deviations an object differs from the variable mean. For matrices containing negative values, `pimage` uses automatically a divergent palette. 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 (represented by `NA`) to the permutations object using the `combine` function `c()`. This new permutation object for 2-mode data is used for displaying the reordered scaled data. The two plots are shown in Figure 3.

```
> pimage(scale(x), main = "Random")

> o_2mode <- c(o, NA)
> pimage(scale(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", "R2E", "ARSA", "HC", "GW", "OLO")
> o <- sapply(methods, FUN = function(m) seriate(d, m))
```

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.

Seriation Method	TSP	R2E	ARSA	HC	GW	OLO
Execution time [sec]	0.012	0.038	0.795	0.003	0.005	0.004

Table 3: Execution time of seriation of the iris data set for different methods.

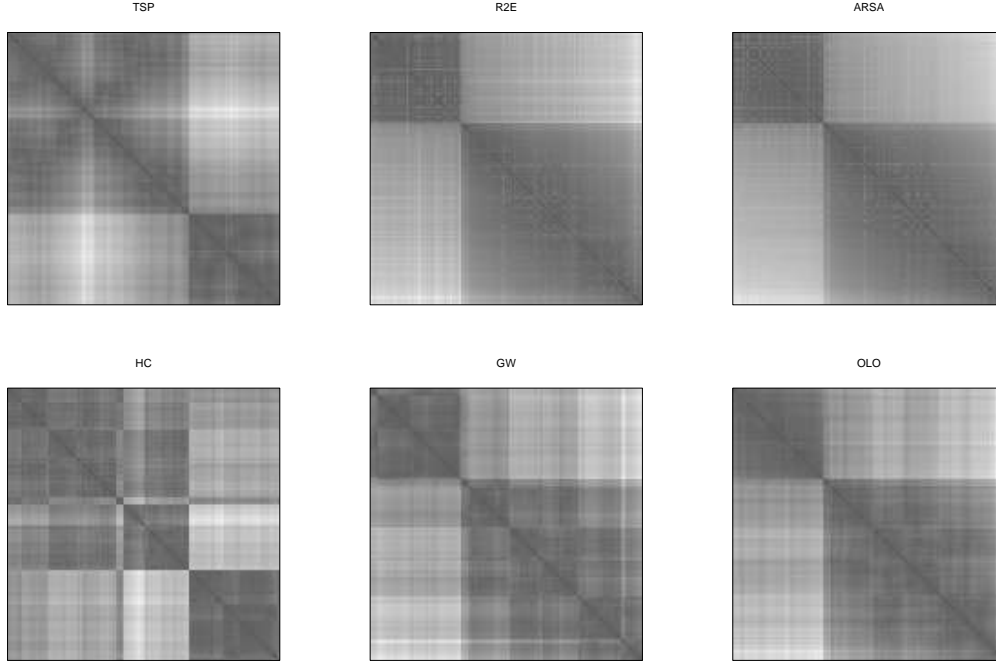


Figure 4: Image plot of the distance matrix for the iris data using rearrangement by different seriation methods.

The direction of the resulting orderings is first normalized (aligned) and then the orderings are displayed using matrix shading (see Figure 4).

```
> o <- seriation_align(o)
> for(s in o) pimage(d, s, main = get_method(s), key = FALSE)
```

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

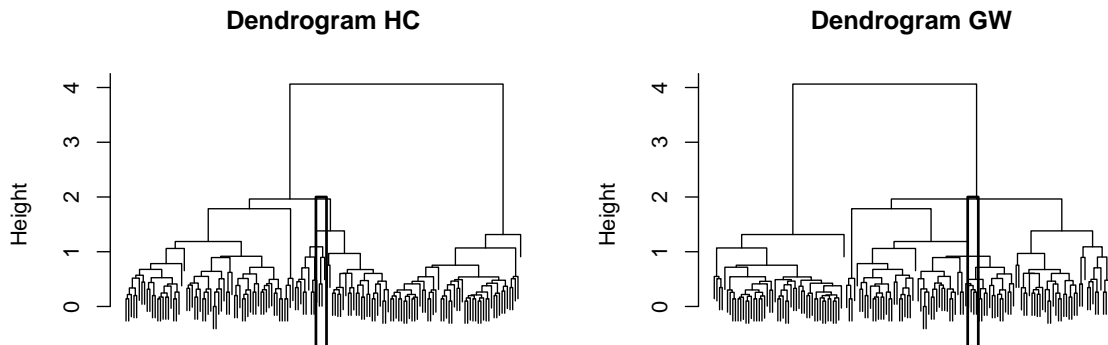


Figure 5: Dendrograms for the seriation with HC and GW.

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

```
> plot(o[["HC"]], labels = FALSE, main = "Dendrogram HC")
> plot(o[["GW"]], 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_events	AR_deviations	RGAR	Gradient_raw	Gradient_weighted	Path_length
TSP	276110	179758	0.25042	549654	1323294	51.79
R2E	90031	18184	0.08165	921858	1746766	85.43
ARSA	54882	9439	0.04978	992142	1772124	85.77
HC	387861	329553	0.35177	326113	877752	70.40
GW	184425	53158	0.16726	733016	1644378	56.96
OLO	162486	43484	0.14737	776925	1669805	52.02
	Inertia	Least_squares	ME	Moore_stress	Neumann_stress	
TSP	311169902	77086081	404603	15104	5168	
R2E	354057418	76521451	402017	19358	7305	
ARSA	356948894	76487641	402283	13888	5468	
HC	272005465	77680136	401656	50281	17325	
GW	346811290	76657969	402100	21973	7535	
OLO	346780582	76624065	403335	17113	5848	

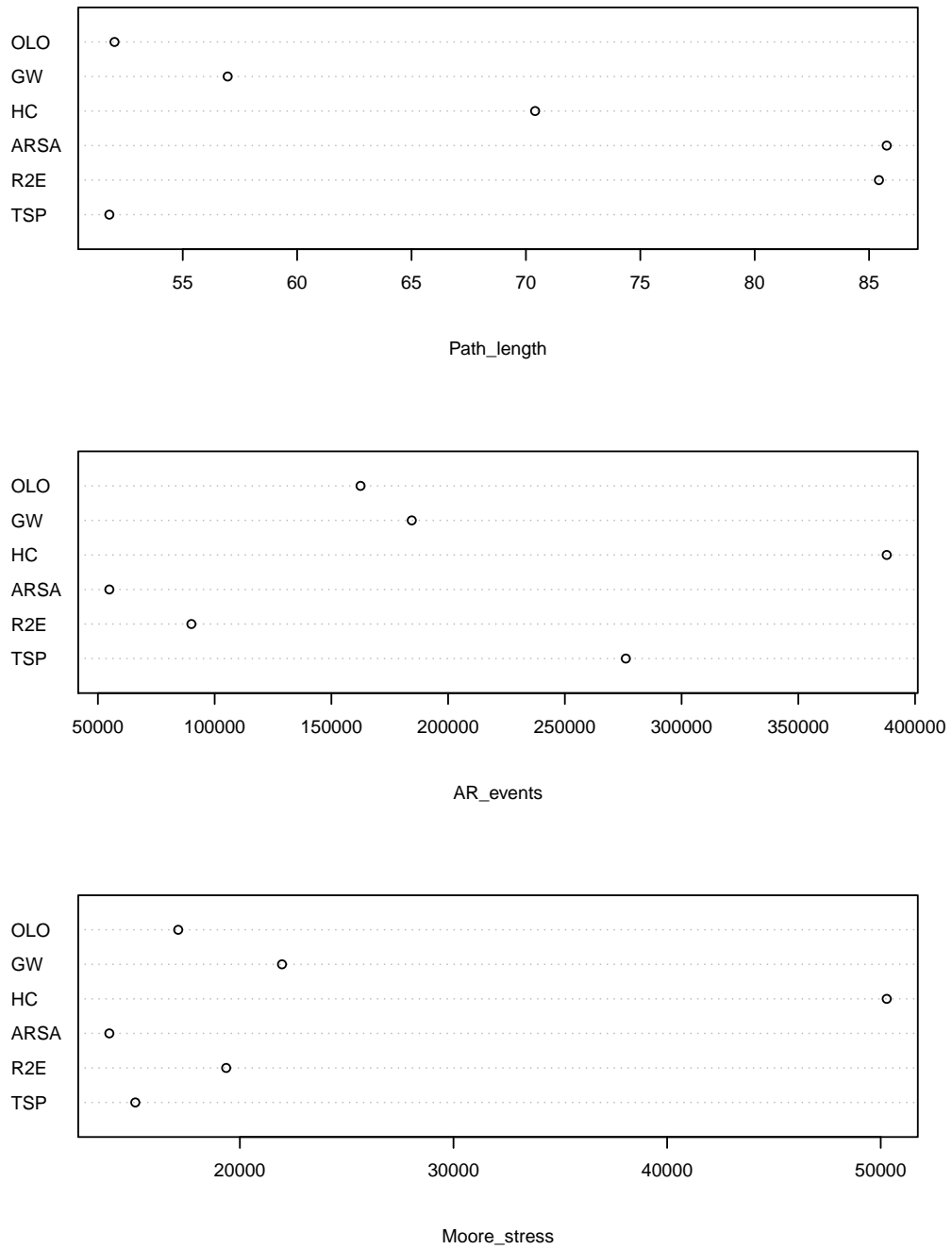


Figure 6: Comparison of different methods and seriation criteria

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.

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 the first few available seriation methods for input data of class `dist` and `matrix`.

```
> show_seriation_methods("dist")[1:3]

$dist_Identity
name:      Identity
kind:      dist
description: Identity permutation

$dist_Random
name:      Random
kind:      dist
description: Random permutation

$dist_ARSA
name:      ARSA
kind:      dist
description: Minimize Anti-Robinson events using simulated annealing

> show_seriation_methods("matrix")[1:3]

$matrix_BEA_TSP
name:      BEA_TSP
kind:      matrix
description: TSP to maximize ME

$matrix_BEA
name:      BEA
kind:      matrix
description: Bond Energy Algorithm to maximize ME
```

```
$matrix_PCA
name:      PCA
kind:      matrix
description: 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_TSP"  "BEA"      "PCA"      "PCA_angle" "Identity"  "Random"
```

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 reverses the original order of the objects, i.e., which returns the reverse identity order.

```
> seriation_method_reverse <- function(x, control = NULL) {
+   lapply(dim(x), function(n) rev(seq(n)))
+ }
```

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 ‘Reverse’.

```
> set_seriation_method("matrix", "Reverse", seriation_method_reverse,
+   "Reverse identity order")
> set_seriation_method("array", "Reverse", seriation_method_reverse,
+   "Reverse identity order")
```

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

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

```
$matrix_BEA_TSP
name:      BEA_TSP
kind:      matrix
description: TSP to maximize ME
```

```
$matrix_BEA
name:      BEA
```

```

kind:      matrix
description: Bond Energy Algorithm to maximize ME

$matrix_PCA
name:      PCA
kind:      matrix
description: First principal component

$matrix_PCA_angle
name:      PCA_angle
kind:      matrix
description: First two principal components (angle)

$matrix_Identity
name:      Identity
kind:      matrix
description: Identity permutation

$matrix_Random
name:      Random
kind:      matrix
description: Random permutation

$matrix_Reverse
name:      Reverse
kind:      matrix
description: Reverse identity order

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

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

  vector length seriation method
1          4          Reverse
2          3          Reverse

> get_order(o, 1)

[1] 4 3 2 1

> get_order(o, 2)

[1] 3 2 1

```

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 implemented 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., [Eisen, Spellman, Brownlager, and Botstein 1998](#)).

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 (using by default Optimal Leaf Ordering), the function `hmap()` can be used with its default settings.

```
> hmap(x, margin = c(7, 4), cexCol=1, labRow = FALSE)
```

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, method = "MDS")
```

If a seriation method is used that does not depend on dendrograms, instead of hierarchical clustering, seriation on the dissimilarity matrices for rows and columns is performed 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

[Bertin \(1981, 1999\)](#) 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 **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.

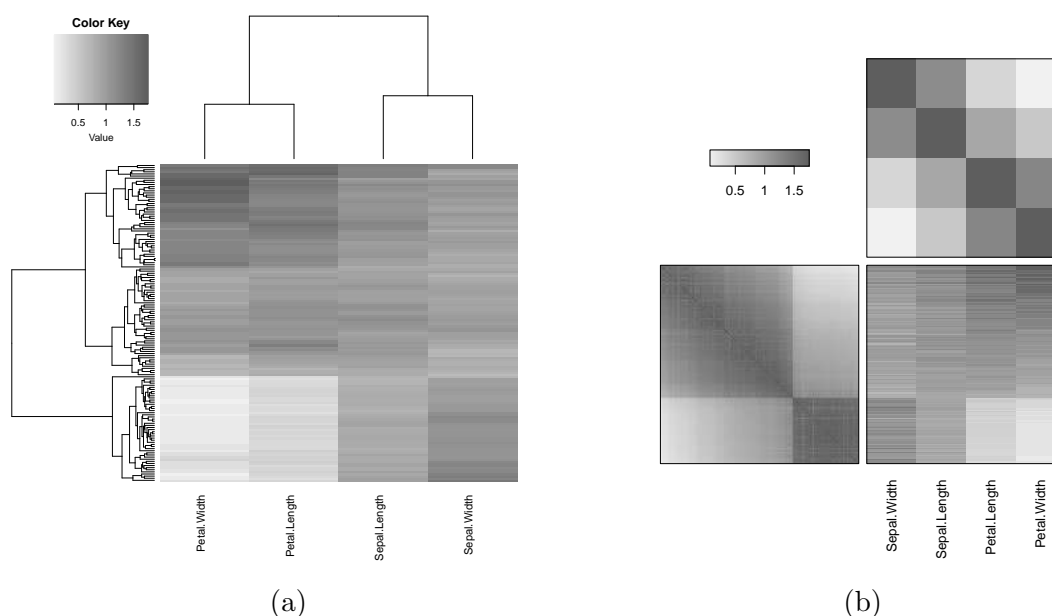


Figure 7: Two presentations of the rearranged iris data matrix. (a) as an optimally reordered heat map and (b) as a seriated data matrix with reordered dissimilarity matrices to the left and on top.

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 (de Falguerolles, Friedrich, and Sawitzki 1997)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"),
```

¹The 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/>

```
+      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)
```

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 Ihm (2005) 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 Bertin (1981) 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))
```

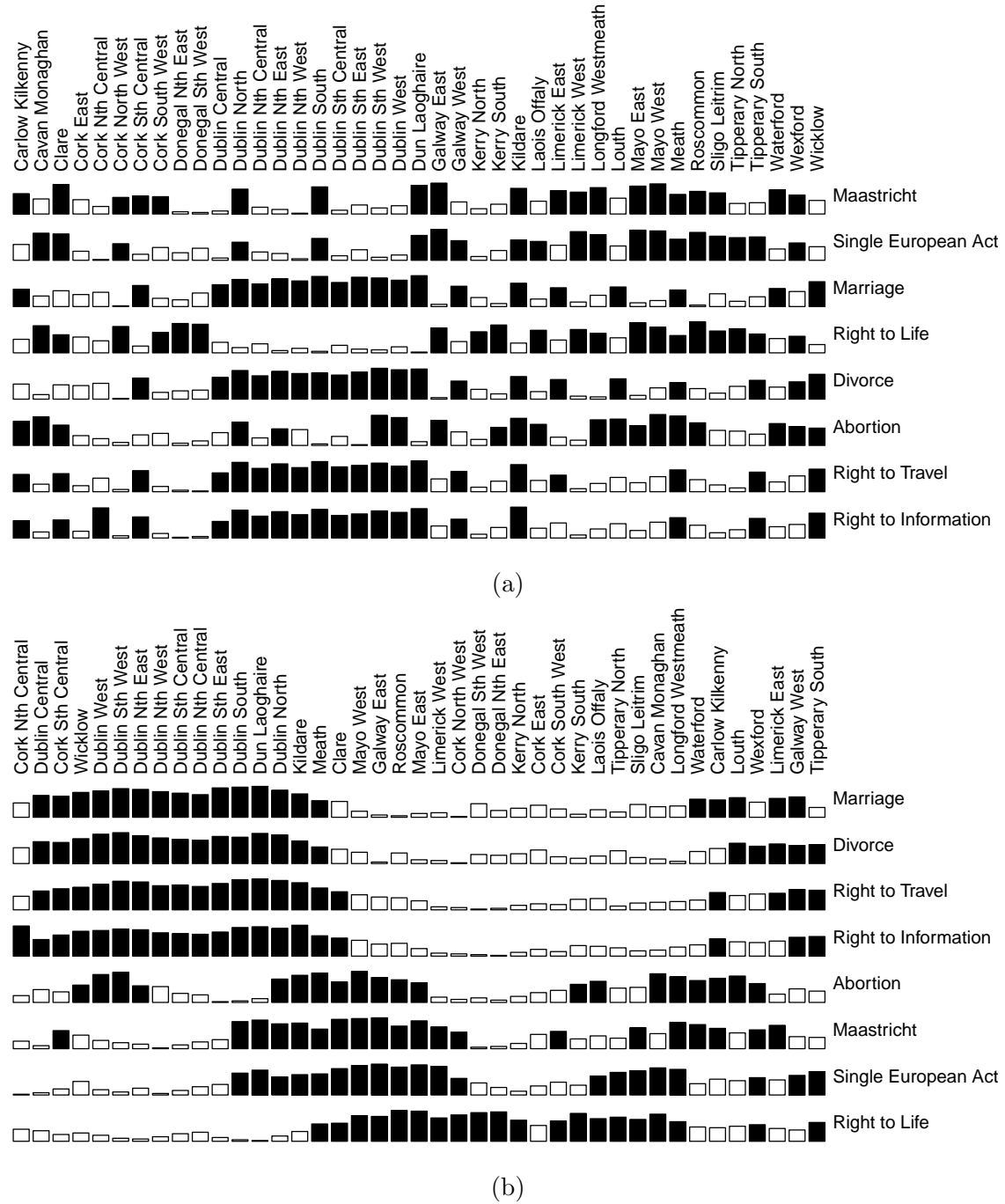


Figure 8: Bertin plot for the (a) original arrangement and the (b) reordered Irish data set.

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	80

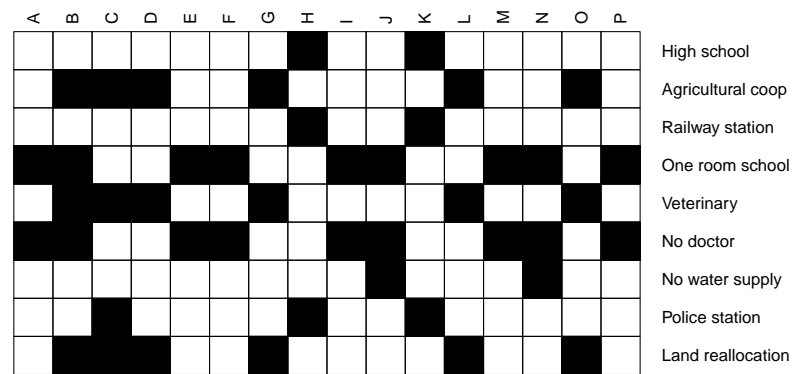
BEA tries to maximize the measure of effectiveness which is much higher in the reordered matrix (in fact, 65 is the maximum for the data set). Also the two types of stress are improved significantly.

5.7. Dissimilarity plot

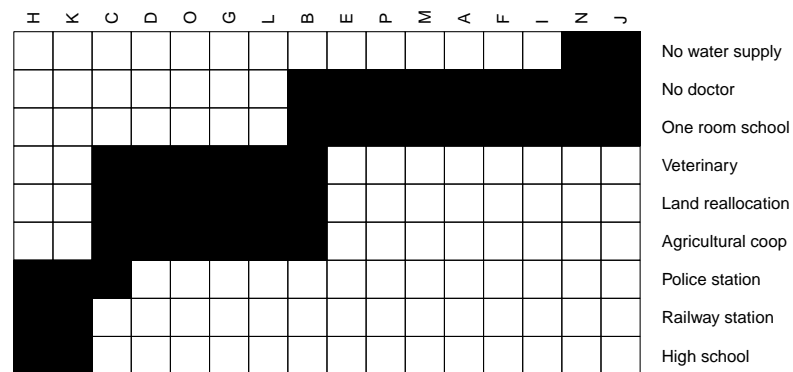
Assessing the quality of an obtained cluster solution has been a research topic since the invention of cluster analysis. This is especially important since all popular cluster algorithms produce a clustering even for data without a “cluster” structure.

Matrix shading is an old technique to visualize clusterings by displaying the rearranged matrices (see, e.g., [Sneath and Sokal 1973](#); [Ling 1973](#); [Gale, Halperin, and Costanzo 1984](#)). 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.

[Strehl and Ghosh \(2003\)](#) 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.



(a)



(b)

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

The dissimilarity plots implemented by the function `dissplot()` in **seriation** improve *CLUSTERION* 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.

```
> ## plot original matrix
> dissplot(d, method = NA)

> ## plot reordered matrix
> 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
```

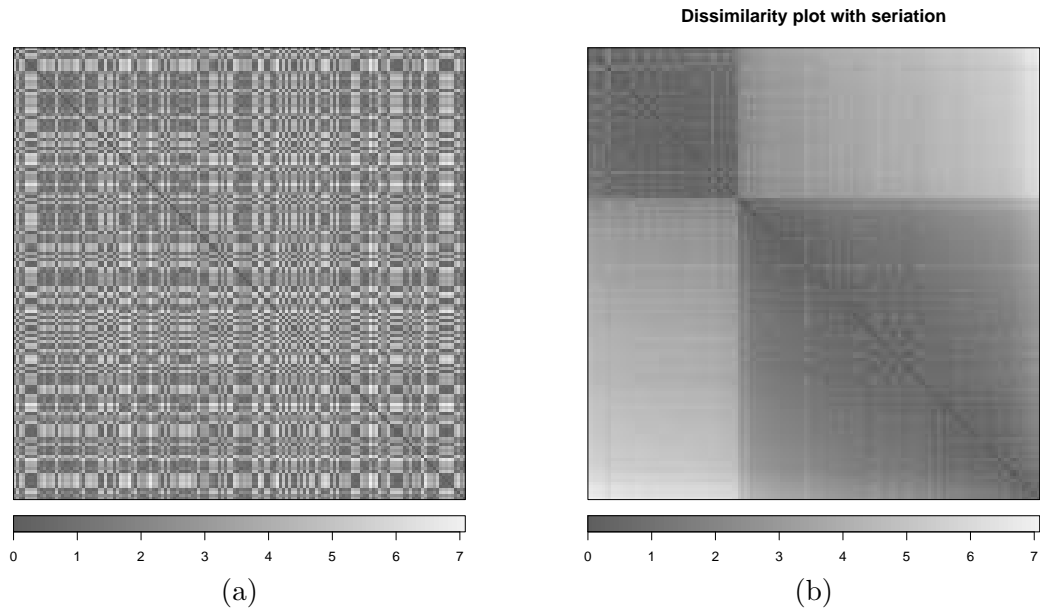


Figure 10: Two dissimilarity plots. (a) the original dissimilarity matrix and (b) the seriated dissimilarity matrix.

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

cluster description

	position	label	size	aggregated_dissimilarity	avg_silhouette_width
1	1	3	6	0.3146	0.09850
2	2	10	15	0.8533	0.39962
3	3	7	15	0.5950	0.35982
4	4	5	5	0.5426	0.29692
5	5	2	9	0.3177	-0.08095
6	6	6	8	0.6524	0.30633
7	7	8	21	0.7597	0.33593
8	8	9	35	0.7052	0.24743
9	9	1	27	0.5795	0.34854
10	10	4	9	0.4939	0.39738

used seriation methods

```
inter-cluster: 'ARSA'
intra-cluster: 'ARSA'
```

dissimilarity aggregation method: 'avg'

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

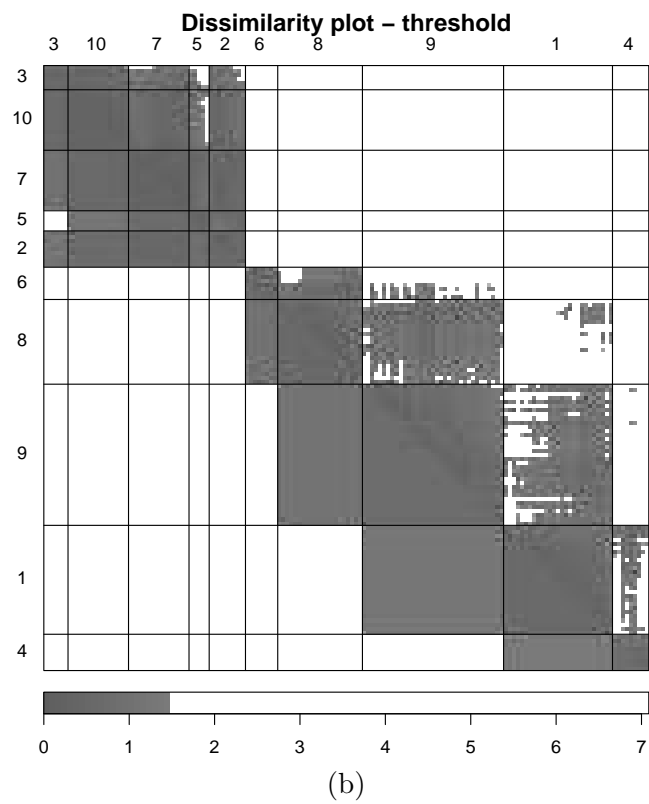
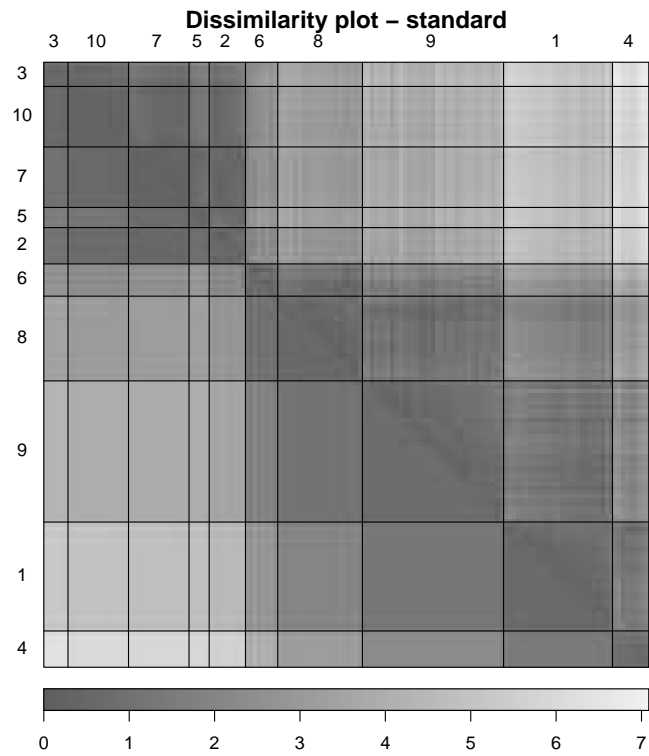


Figure 11: Dissimilarity plot for k -means solution with 10 clusters. (a) standard plot and (b) plot with threshold.

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.

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.

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

	3	10	7	5	2	6	8	9	1	4
setosa	6	15	15	5	9	0	0	0	0	0
versicolor	0	0	0	0	0	8	20	22	0	0
virginica	0	0	0	0	0	0	1	13	27	9

```
> # $
```

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", package="cluster")
> 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.

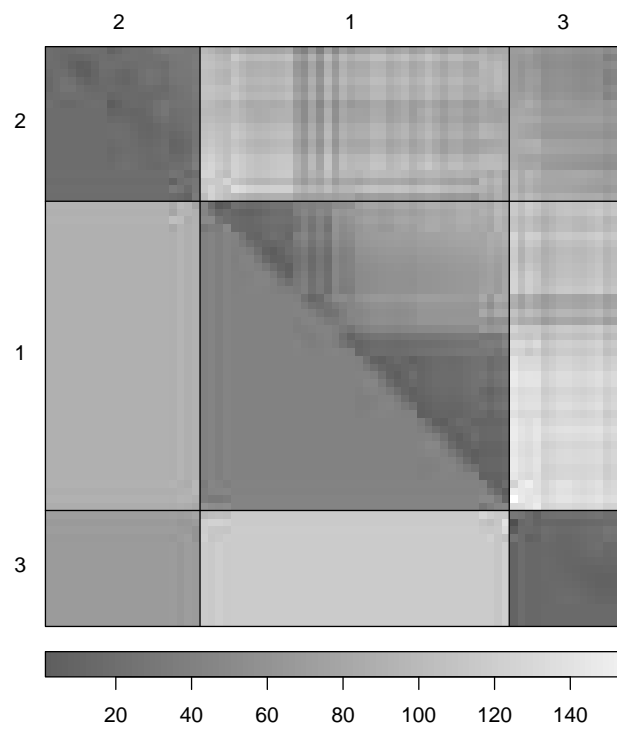


Figure 12: Dissimilarity plot for k -means solution with 3 clusters for the Ruspini data set with 4 groups.

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. A more detailed treatment of dissimilarity plots as a tool for exploring partitional clustering can be found in [Hahsler and Hornik \(2011\)](#).

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. [Jurman, Merler, Barla, Paoli, Galea, and Furlanello \(2008\)](#) for a recent application of such ideas in a molecular profiling context). The R extension package **relations** ([Hornik and Meyer 2008](#)) 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., [Marcotorchino 1987](#)).

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.

References

- Arabie P, Hubert LJ (1990). “The Bond Energy Algorithm Revisited.” *IEEE Transactions on Systems, Man, and Cybernetics*, **20**(1), 268–74.
- Arabie P, Hubert LJ (1996). “An Overview of Combinatorial Data Analysis.” In P Arabie, LJ Hubert, GD Soete (eds.), *Clustering and Classification*, pp. 5–63. World Scientific, River Edge, NJ.
- Bar-Joseph Z, Demaine ED, Gifford DK, Jaakkola T (2001). “Fast Optimal Leaf Ordering for Hierarchical Clustering.” *Bioinformatics*, **17**(1), 22–29.

- Bertin J (1981). *Graphics and Graphic Information Processing*. Walter de Gruyter, Berlin. Translated by William J. Berg and Paul Scott.
- Bertin J (1999). “Graphics and Graphic Information Processing.” In SK Card, JD Mackinlay, B Shneiderman (eds.), *Readings in Information Visualization*, pp. 62–65. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. ISBN 1-55860-533-9.
- Brusco M, Köhn HF, Stahl S (2008). “Heuristic Implementation of Dynamic Programming for Matrix Permutation Problems in Combinatorial Data Analysis.” *Psychometrika*. In press.
- Brusco M, Stahl S (2005). *Branch-and-Bound Applications in Combinatorial Data Analysis*. Springer.
- Caraux G, Pinloche S (2005). “Permutmatrix: A Graphical Environment to Arrange Gene Expression Profiles in Optimal Linear Order.” *Bioinformatics*, **21**(7), 1280–1281.
- Chen CH (2002). “Generalized Association Plots: Information Visualization via Iteratively Generated Correlation Matrices.” *Statistica Sinica*, **12**(1), 7–29.
- Chessel D, Dufour AB, Dray S (2007). *ade4: Analysis of Ecological Data : Exploratory and Euclidean methods in Multivariate data analysis and graphical display*. R package version 1.4-3, URL <http://CRAN.R-project.org/package=ade4>.
- de Falguerolles A, Friedrich F, Sawitzki G (1997). “A Tribute to J. Bertin’s Graphical Data Analysis.” In W Bandilla, F Faulbaum (eds.), *SoftStat ’97 (Advances in Statistical Software 6)*, pp. 11–20. Lucius & Lucius.
- de Leeuw J (2005). “Unidimensional Scaling.” In B Everitt, D Howelll (eds.), *Encyclopedia of Statistics in Behavioral Science*, volume 4. Wiley.
- Ding C, He X (2004). “Linearized cluster assignment via spectral ordering.” In *Proceedings of the Twenty-first International Conference on Machine Learning (ICML ’04)*, p. 30. ACM Press.
- Dray S, Dufour AB (2007). “The ade4 Package: Implementing the Duality Diagram for Ecologists.” *Journal of Statistical Software*, **22**(4). URL <http://www.jstatsoft.org/v22/i04>.
- Eisen MB, Spellman PT, Brownagger PO, Botstein D (1998). “Cluster Analysis and Display of Genome-wide Expression Patterns.” *Proceedings of the National Academy of Science of the United States*, **95**(25), 14863–14868.
- Fowler M (2004). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. third edition. Addison-Wesley Professional.
- Gale N, Halperin WC, Costanzo CM (1984). “Unclassed Matrix Shading and Optimal Ordering in Hierarchical Cluster Analysis.” *Journal of Classification*, **1**, 75–92.
- Garfinkel RS (1985). “Motivation and Modeling.” chapter 2, pp. 17–36. Wiley, New York.
- Gruvaeus G, Wainer H (1972). “Two Additions to Hierarchical Cluster Analysis.” *British Journal of Mathematical and Statistical Psychology*, **25**, 200–206.

- Gutin G, Punnen AP (eds.) (2002). *The Traveling Salesman Problem and Its Variations*, volume 12 of *Combinatorial Optimization*. Kluwer, Dordrecht.
- Hahsler M, Hornik K (2007a). **TSP**: *Traveling Salesperson Problem (TSP)*. R package version 0.2-2, URL <http://CRAN.R-project.org/package=TSP>.
- Hahsler M, Hornik K (2007b). “TSP – Infrastructure for the Traveling Salesperson Problem.” *Journal of Statistical Software*, **23**(2), 1–21. URL <http://www.jstatsoft.org/v23/i02>.
- Hahsler M, Hornik K (2011). “Dissimilarity Plots: A Visual Exploration Tool for Partitional Clustering.” *Journal of Computational and Graphical Statistics*. Accepted for publication.
- Hahsler M, Hornik K, Buchta C (2008). “Getting Things in Order: An Introduction to the R Package seriation.” *Journal of Statistical Software*, **25**(3), 1–34. ISSN 1548-7660. URL <http://www.jstatsoft.org/v25/i03>.
- Held M, Karp RM (1962). “A Dynamic Programming Approach to Sequencing Problems.” *Journal of SIAM*, **10**, 196–210.
- Hornik K, Meyer D (2008). **relations**: *Data Structures and Algorithms for Relations*. R package version 0.3-1, URL <http://CRAN.R-project.org/package=relations>.
- Hubert L, Arabie P, Meulman J (2001). *Combinatorial Data Analysis: Optimization by Dynamic Programming*. Society for Industrial Mathematics.
- Hubert LJ (1974). “Some Applications of Graph Theory and Related Nonmetric Techniques to Problems of Approximate Seriation: The Case of Symmetric Proximity Measures.” *British Journal of Mathematical Statistics and Psychology*, **27**, 133–153.
- Hurley C (2007). **gclus**: *Clustering Graphics*. R package version 1.2, URL <http://CRAN.R-project.org/package=gclus>.
- Ihm P (2005). “A Contribution to the History of Seriation in Archaeology.” In C Weihs, W Gaul (eds.), *Classification - the Ubiquitous Challenge, Proceedings of the 28th Annual Conference of the Gesellschaft für Klassifikation e.V., University of Dortmund, March 9–11, 2004*, Studies in Classification, Data Analysis, and Knowledge Organization, pp. 307–316.
- Jurman G, Merler S, Barla A, Paoli S, Galea A, Furlanello C (2008). “Algebraic Stability Indicators for Ranked Lists in Molecular Profiling.” *Bioinformatics*, **24**(2), 258–264.
- Kendall DG (1971). “Seriation from Abundance Matrices.” In F Hodson, D Kendall, P Tautu (eds.), *Mathematics in the Archaeological and Historical Sciences*, pp. 214–252.
- Lenstra JK (1974). “Clustering a Data Array and the Traveling-Salesman Problem.” *Operations Research*, **22**(2), 413–414.
- Lin S, Kernighan BW (1973). “An Effective Heuristic Algorithm for the Traveling-Salesman Problem.” *Operations Research*, **21**(2), 498–516.
- Ling RL (1973). “A Computer Generated Aid for Cluster Analysis.” *Communications of the ACM*, **16**(6), 355–361.

- Marcotorchino F (1987). “Block Seriation Problems: A Unified Approach.” *Applied Stochastic Models and Data Analysis*, **3**, 73–91.
- McCormick WT, Schweitzer PJ, White TW (1972). “Problem Decomposition and Data Reorganization by a Clustering Technique.” *Operations Research*, **20**(5), 993–1009.
- Niermann S (2005). “Optimizing the Ordering of Tables With Evolutionary Computation.” *The American Statistician*, **59**(1), 41–46.
- Oksanen J, Kindt R, Legendre P, O’Hara B (2007). *vegan: Community Ecology Package*. R package version 1.8-6, URL <http://CRAN.R-project.org/package=vegan>.
- Padberg M, Rinaldi G (1990). “Facet Identification for the Symmetric Traveling Salesman Polytope.” *Mathematical Programming*, **47**(2), 219–257. ISSN 0025-5610.
- Petrie FWM (1899). “Sequences in Prehistoric Remains.” *Journal of the Anthropological Institute*, **29**, 295–301.
- Robinson WS (1951). “A Method for Chronologically Ordering Archaeological Deposits.” *American Antiquity*, **16**, 293–301.
- Rosenkrantz DJ, Stearns RE, Philip M Lewis I (1977). “An Analysis of Several Heuristics for the Traveling Salesman Problem.” *SIAM Journal on Computing*, **6**(3), 563–581.
- Sneath PHA, Sokal RR (1973). *Numerical Taxonomy*. Freeman and Company, San Francisco.
- Strehl A, Ghosh J (2003). “Relationship-based Clustering and Visualization for High-dimensional Data Mining.” *INFORMS Journal on Computing*, **15**(2), 208–230.

Affiliation:

Michael Hahsler
 Engineering Management, Information, and Systems
 Lyle School of Engineering
 Southern Methodist University
 P.O. Box 750123
 Dallas, TX 75275-0123
 E-mail: mhahsler@lyle.smu.edu
 URL: <http://lyle.smu.edu/~mhahsler>

Kurt Hornik
 Department für Statistik & Mathematik
 Wirtschaftsuniversität Wien
 1090 Wien, Austria
 E-mail: kurt.hornik@wu.ac.at
 URL: <http://statmath.wu.ac.at/~hornik/>

Christian Buchta
 Department für Welthandel
 Wirtschaftsuniversität Wien

1090 Wien, Austria

E-mail: christian.buchta@wu.ac.at

URL: <http://www.wu.ac.at/itf/institute/staff/buchta>