Delft University of Technology
Faculty Electrical Engineering, Mathematics and Computer Science
Delft Institute of Applied Mathematics

Ranking items: Is solving a seriation problem a good alternative for existing methods? (Dutch title: Items rangschikken: Is het oplossen van een seriatie probleem een goed alternatief voor bestaande methodes?)

A thesis submitted to the
Delft Institute of Applied Mathematics
in partial fulfilment of the requirements

for the degree

BACHELOR OF SCIENCE
in
APPLIED MATHEMATICS

by

Quinten Cederhout

Delft, the Netherlands
June 2017

Copyright © 2017 by Q. Cederhout. All rights reserved.

BSc thesis APPLIED MATHEMATICS

"Ranking items: Is solving a seriation problem a good alternative for existing methods?"
(Dutch title: "Items rangschikken: Is het oplossen van een seriatie probleem een goed alternatief voor bestaande methodes?")

QUINTEN CEDERHOUT

Delft University of Technology

Responsible Professor

Prof.dr. E. de Klerk

Other members of the thesis committee

Dr. B. van den Dries

Dr.ir. F.J. Vermolen

June, 2017

Delft

Abstract

In this dissertation we look at the seriation problem and the applications of this problem. Given a set of items, we try to find an ordering based on the similarity between the items.
We start by explaining the mathematical theory behind the seriation problem. Then we describe a couple of different methods that can be used to find a solution for the problem. After that, we apply these methods to various different datasets. The results of these tests will be analysed.
Solving a seriation problem can be an alternative way to already existing methods when finding a ranking of items for a given dataset. The goal is to find out if is also a viable method to use in practice.

Contents

Chapter 1

Introduction

1.1 The seriation problem

You have got your favourite books lying before you. When putting them back in the closet you want to order them from best to least good. That is not an easy task. For example, how do you decide what is your seventh favourite book? It is much easier to compare the books in relative terms 'I like this book better than that one', which you can decide for every pair of books. Constructing a ranking from these pairwise comparisons is an example of a seriation problem. Given a set of items, we want to construct an ordering for these items based on the similarity between them. Often only pairwise comparisons between items are known from which similarity scores can be computed. Solving a seriation problem has many different applications, often related to finding a ranking for different items. For example, finding which teams are the best in a sport competition or finding a ranking for the best universities.

1.2 Origin of the seriation problem

The seriation problem originates in the late nineteenth century with a man called Flinders Petrie [16]. He was an archaeologist and was excavating tombs in Egypt. The tombs they found had no evidence of their date and common dating techniques we use nowadays were not yet available. Petrie invented a seriation technique to sequence the tombs by looking at the pottery within them. First, he classified the different styles of the pottery they found inside the tombs and assumed that each style came from a different time period. Then he made the assumption that if two styles had many similarities, they would originate with a closer period of time in between. In this way he reordered the different styles of pottery into a ranking and could determine the chronological order of the tombs by looking at the style of pottery present within them.
After that, the seriation problem was studied throughout the years in many different environments [12]. The particular problem often does not have a clear mathematical formulation. Therefore, finding a solution involves the constructing and solving of a mathematical model. Because of this, the problem also has known different descriptions and definitions. One of these definitions, the one we will use, can be found in e.g. Fogel, Daspremont and Vojnovic (p. 3, [7]).

Definition 1.2.1 (Seriation problem)**.** *The seriation problem seeks to reorder n items given a similarity matrix between these items, such that the more similar two items are, the closer they should be.*

In practice, this often translates to finding a perfect ranking for a set of items, which is often not possible. We will explain more details about this perfect ranking in Chapter 2. Our task

will not be to find a perfect ranking, but to find the 'best' ranking possible, which is sometimes a subjective process.

1.3 Goal of our research

Solving a seriation problem can be used to create a ranking for a set of items. In reality, there are many instances where a ranking for items is created through some sort of method. We want to see if our methods, that solve a seriation problem, can be equally good or even better than the existing methods. Therefore we have three different datasets we can try to find a ranking for (See Section 4.1).
We have a 'Test set' which is a set of items with a given ranking which we can use to test the solving methods we find.
Furthermore, we have data from a korfball competition [14], from which we know which team beat which team within the competition. This Korfbal League competition produces a ranking by using the classical point system seen in most sports. We will compare our rankings with this known ranking.
Finally, we have a dataset with different rankings for universities in Europe. We will try to combine these different rankings into a singular one by translating it to seriation problem we can solve. This combining of different rankings is part of a discipline sometimes called 'voting theory', where you try to elect a winner from a set of different votes or rankings. There exist many methods to do this and we will compare our rankings to some of them.

1.4 Outline of dissertation

We will start of with a mathematical analysis of the seriation problem in Chapter 2. Here we discover what needs to be done in order to get to a solution and we will also find the reason such a solution is often not fully satisfactory.
Because of this, many different ways were developed over the years to tackle the problem. We will have a look at a couple different existing methods in Chapter 3. The first is a spectral method from Fogel et al. [7], which is based on an older work from Atkins et al. [2]. Further one we will also rewrite the problem, namely as a quadratic assignment problem (QAP), which is a well known problem inside the optimisation discipline. The thesis of M. Seminaroti [18] describes a special case of the QAP called the 2-Sum problem, which we will use in the first place. To find a solution for the quadratic assignment problem we will study the heuristics and algorithms found in Burkard et al. [4].
After describing different solution methods, we will apply them in Chapter 4 to the three datasets and analyse the results. Finally, in Chapter 5, we will we argue which solution method gave the best results. We will also determine if solving a seriation problem can be a good alternative to existing ranking methods.

Chapter 2

Mathematical analysis

We will start of with describing the seriation problem in a mathematical setting. In doing so, we will discover why the problem does not always have a satisfactory solution.
Solving the seriation problem (Definition 1.2.1) consist of finding a ranking of items out of a similarity matrix, in which is stored how similar two items are compared to each other. The datasets we want to analyse do not give us any similarity matrices, but only pairwise comparisons, for example which korfball team lost to which other teams. So before we look at methods to find a ranking, we will first have to translate these pairwise comparisons into a similarity matrix.

2.1 Comparison matrices

From a given dataset we have the pairwise comparisons between the n items. A good way to display and work with this information is a comparison matrix. In a comparison matrix all the pairwise comparisons are stored. It is constructed as follows.

Definition 2.1.1 (Comparison matrix)**.** *Given a set of n items and their pairwise comparisons. Then matrix $C \in \{-1, 0, 1\}^{n \times n}$ is the comparison matrix with*

$$C_{i,j} = -C_{j,i} \quad \text{and} \quad C_{i,j} = \begin{cases} 1 & \text{if } i \text{ is higher ranked than } j \text{ or } i = j, \\ 0 & \text{if } i \text{ and } j \text{ are not compared or in a draw,} \\ -1 & \text{if } i \text{ is lower ranked than } j. \end{cases} \tag{2.1}$$

At first it may seem a bit strange that the diagonal elements are set to 1. It looks like an element is higher ranked than itself. However, the following part gives us a good reason to define the comparison matrix in this way.
From a comparison matrix we want to construct a pairwise similarity matrix. The question to ask here is 'How do we define the similarity between two items?' If we look at a tournament setting we can give an answer. Two teams that defeated the same teams and lost to the same teams should have a similar ranking in the end, and thus have a high similarity score. In other words, we compute the similarity score for two items i and j by looking at the similarity between the pairwise comparisons with all the other $n - 2$ items.

Definition 2.1.2 (Similarity matrix)**.** *The similarity score between item i and item j is equal to:*

$$S^{match}_{i,j} = \sum_{k=1}^{n} \left(\frac{1 + C_{i,k} C_{j,k}}{2} \right). \tag{2.2}$$

The matrix S^{match} is called the similarity matrix.

If $C_{i,k}$ and $C_{j,k}$ have matching signs that means that both items i and j compare in the same way to item k, which makes them more similar. Therefore, if the signs match we get $C_{i,k}C_{j,k} = 1$ and that has a positive effect on the similarity score like we want. If the signs are not matching, then $C_{i,k}C_{j,k} = -1$ and the similarity score does not get increased. This also makes sense because items i and j relate in a different way to item k and should not be considered more similar because of these comparisons. Since the pairwise comparisons are not always complete, it could happen that either i or j are not compared to item k. Then $C_{i,k}C_{j,k} = 0$ and this will give a slight positive effect to the similarity.
The definition of the similarity matrix clarifies why all the diagonal elements of the comparison matrix (2.1.1) have a 1. Intuitively, an items is very similar to itself, so when creating the similarity matrix the entry $C_{i,i}$ should always have a positive effect on the similarity of that particular item i.
Instead of calculating the similarity score between each pair of items with the sum (2.1.2), we can do it all at once by translating this to matrix multiplications and additions. With $\mathbf{1}$ being a all-ones vector of length n we get:

$$S^{match} = \frac{1}{2}(n\mathbf{1}\mathbf{1}^T + CC^T). \tag{2.3}$$

2.2 Partial orders and Chains

Now that we have translated the pairwise comparisons to a similarity matrix we can begin with analysing the seriation problem. We will have a look at what it means to have a solution for the problem and also in which occasions such an solution does or does not exist. We start of with defining what a proper solution is, therefore we need to following definitions as found in e.g. Aliprantis and Burkinshaw (pp. 7-8, [1]).

Definition 2.2.1 (Partial order)**.** *A relation $\preceq$ is said to be a partial order for a set X if it satisfies the following three properties:*

1. *$x \preceq x$ for every $x \in X$ (reflexivity)*
2. *If $x \preceq y$ and $y \preceq x$, then $x = y$ (antisymmetry)*
3. *If $x \preceq y$ and $y \preceq z$, then $x \preceq z$ (transitivity)*

The set X is then called a partially ordered set.

Definition 2.2.2 (Chain)**.** *A subset Y of a partially ordered set is called a chain if for every pair $x, y \in Y$, either $x \preceq y$ or $y \preceq x$. The chain Y is also called a totally ordered set.*

Looking at the last definition, a chain of all items seems like the ranking we are looking for when solving the seriation problem. Therefore, finding such a chain is our ultimate goal in solving the seriation problem. We define this as a perfect ranking.

Definition 2.2.3 (Perfect ranking)**.** *Given a set $\mathcal{S}$ of n items and pairwise relations between these items. If the pairwise relations define a chain on all the n items (i.e. $\mathcal{S}$ is a totally ordered set), then we call such a chain a perfect ranking of $\mathcal{S}$.*

A chain requires that their is an comparison between every two items in the chain, which gives an immediate problem. Often in reality, not all pairwise comparisons are available or there are ties between items, as described earlier while defining the comparison matrix (Def. 2.1.1). In general, this makes finding a perfect ranking impossible. We generalise the perfect ranking by removing the requirement of having every comparison available.

Definition 2.2.4 (Consistent ranking)**.** *Given a set $\mathcal{S}$ of n items and pairwise relations between these items. If the pairwise relations define a partial order on $\mathcal{S}$, than there exists an ordering of $\mathcal{S}$ which respects the partial order. Such an ordering is called a consistent ranking.*

It is quickly seen that any consistent ranking can be expanded to a perfect ranking by adding missing comparison relations. Such a perfect ranking no longer has to be unique.
If we have a seriation problem with missing comparisons between items, our goal will be to find a consistent ranking, which can be expanded to a perfect ranking if needed. Such an consistent ranking often does not exist, which is the very reason the seriation problem is more like solving a mathematical model as argued in Paragraph 1.2. In the following example we show a situation where no consistent ranking exists.

Example 2.2.5 (No consistent ranking)**.** *Consider a set of three different sport teams A, B and C. The pairwise comparisons are defined by the teams playing against each other with the following results. Team A beat team B, team B beat team C and team C beat team A. Or in other words $A \succeq B, B \succeq C, C \succeq A$. With every team losing once and winning once we cannot declare a clear victor and loser, therefore it will be impossible to find a consistent ranking for this set of items.*

One can imagine that this type of situations can easily come up in any dataset.
In the next section we will find in which cases a consistent ranking exists and in which cases it does not.

2.3 R-matrices

In order to prove for which similarity matrices we can find a consistent ranking we have to introduce some new theory. We start with the Robinson similarity matrix (R-matrix).

Definition 2.3.1 (R-matrix)**.** *(Fogel et al., Definition 2.1, [7]). Let A be a symmetric $n \times n$ matrix. If $A_{i,j} \leq A_{i,j+1}$ and $A_{i+1,j} \leq A_{i,j}$ whenever $1 \leq j < i \leq n$ (the lower triangle), then A is called a Robinson similarity matrix or R-matrix.*

So in other words, an R-matrix is a symmetrical matrix in which coefficients do not increase as we move away from the diagonal.

Example 2.3.2 (Example of an R-matrix)**.**

$$A = \begin{bmatrix} 7 & 4 & 2 & 1 \\ 4 & 8 & 6 & 3 \\ 2 & 6 & 7 & 4 \\ 1 & 3 & 4 & 7 \end{bmatrix} \tag{2.4}$$

We are going to rearrange different items. With the definition of the comparison and similarity matrices (Def. 2.1.1 and 2.1.2) we have seen that each row (and each column with the same number) corresponds to an item. Rearranging these items would be the same as swapping rows and columns at the same time. In matrix notation this rearrangement translates to applying a permutation on both the rows and columns. Let $\pi \in \mathcal{S}_n$ be a permutation, where we define $\mathcal{S}_n$ as the collection of all permutations of $\{1, \ldots, n\}$. Let A be an $n \times n$ matrix, then we will denote the appliance of permutation π to matrix A as $A_\pi = (A_{\pi(i),\pi(j)})_{i,j=1}^n$.

You could find a permutation for a given R-matrix where the new matrix is still an R-matrix, but this is not always possible. We define the matrices for which it is not possible as a special kind of R-matrix.

Definition 2.3.3 (strict-R)**.** *An R-matrix A is strict-R if and only if the identity and reverse identity permutations are the only permutation that reorder A as an R-matrix.*

Here the *reverse identity* permutation is the permutation that reverses the row and columns of a matrix, so $(1, 2, \ldots, n-1, n) \to (n, n-1, \ldots, 2, 1)$.
To solve the seriation problem we have to rearrange items to find a consistent ranking, but in terms of given matrices we will find that this is the same as finding a permutation that permutes the similarity matrix into an R-matrix. We know that the seriation problem does not always have a consistent ranking, so we cannot always find such a permutation, therefore we define the so-called pre-R-matrix.

Definition 2.3.4 (pre-R)**.** *A matrix A is pre-R if there exists a permutation π such that the permuted matrix A_π is an R-matrix. If there exist only two of such permutation matrices, then A is called pre-strict-R.*

Logically, a matrix A that can be permuted into a strict-R-matrix is called a pre-strict-R-matrix. This happens exactly when there are only two permutations that permute A into an R-matrix.

Example 2.3.5 (Example of an pre-R matrix)**.**

$$B = \begin{bmatrix} 7 & 2 & 4 & 6 \\ 2 & 7 & 1 & 4 \\ 4 & 1 & 7 & 3 \\ 6 & 4 & 3 & 8 \end{bmatrix} \tag{2.5}$$

Let $\pi = (2, 4, 1, 3)$ be a permutation. Then B_π is the matrix A from Example 2.3.2, which is an R-matrix. This means that B is a pre-R-matrix.

2.4 R-matrices and the seriation problem

Now that we have defined R-matrices we want to relate them to the similarity matrices we found before and thus to the seriation problem.
Say we have a set of n items with a perfect ranking. If we made the comparison matrix according to this ranking we will have a matrix with in the bottom triangle -1 and for the rest only 1. Creating a similarity matrix from this comparison matrix will give us an R-matrix, as shown by the next proposition from Fogel et al..

Proposition 2.4.1. *(Fogel et al., Proposition 2.3, p. 4, [7]) Given all pairwise comparisons between items ranked according to a chain, the similarity matrix S^{match} constructed in (2.3) is a strict R-matrix and*

$$S^{match}_{i,j} = n - |i - j| \qquad \forall i, j = 1, \ldots, n \tag{2.6}$$

Proof. As noted before we have $C_{i,j} = -1$ if $i < j$ and $C_{i,j} = 1$ otherwise. Definition (2.2) gives us:

$$\begin{aligned} S^{match}_{i,j} &= 1 \cdot (min(i,j) - 1) + 0 \cdot (max(i,j) - (min(i,j) - 1)) + 1 \cdot (n - (max(i,j) - 1)) \\ &= n - (max(i,j) - min(i,j)) \\ &= n - |i - j| \end{aligned}$$

Because $|i - j| < n$, S^{match} is strictly positive. And the further away you are from the diagonal the larger $|i - j|$ is, so the smaller the coefficients of S^{match} will be, hence S^{match} is a strict R-matrix. □

So when there exists a perfect ranking and we chose this ranking as the order of the items to make the comparison matrix with, we will get a similarity matrix that is strict-R.

Example 2.4.2 (Comparison and similarity matrices for perfect ranking)**.** *We consider four sport teams* $\{A, B, C, D\}$*. Let* $A \succeq B \succeq C \succeq D$ *be the perfect ranking of this set. If we chose this as the starting order, the comparison matrix* C *and similarity matrix* S *will be:*

$$C = \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & 1 & 1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & 1 \end{bmatrix} \qquad S = \begin{bmatrix} 4 & 3 & 2 & 1 \\ 3 & 4 & 3 & 2 \\ 2 & 3 & 4 & 3 \\ 1 & 2 & 3 & 4 \end{bmatrix} \tag{2.7}$$

You can see the described pattern of ones and minus ones in the comparison matrix and we see that the similarity matrix indeed is a strict-R matrix.

Now lets say that the pairwise comparisons again define a perfect ranking for the items, but we did not choose the proper order to make the comparison matrix. Proposition 2.4.1 implies that the similarity matrix we get from this will be a pre-strict-R matrix. So to find the ranking we are looking for we have to find the permutation that changes the pre-strict-R matrix to the strict-R matrix. This permutation will tell us what the final ranking should be, because applying it to the starting order will change it to the order corresponding with the strict-R-matrix and we know that that order should be a perfect ranking.
In the next chapter we will explore different methods to permute the similarity matrix into an R-matrix, if this is possible, and thus find the ranking we are looking for.

Chapter 3

Solution methods

We are looking for algorithms that can solve the seriation problem, i.e. that can find a consistent ranking for n items given pairwise comparisons if such an ranking exists. In the previous chapter we have seen that this corresponds to rearranging the similarity matrix (Def. 2.1.2) to an strict-R matrix (Def. 2.3.3). We do this by finding the right permutation that achieves this goal. Applying the found permutation to the starting order will then define the ranking we are looking for, because the items will then be in the same order as they would be for creating the strict-R-matrix (Theorem 2.4.1). In this chapter we will look at different methods that can find this permutation.

3.1 SerialRank

3.1.1 Defining the method

The first method (Fogel et. al., [7]) is a spectral computation method and it is based on sorting the so-called Fiedler vector of the similarity matrix.

Definition 3.1.1 (Fiedler vector)**.** *The Fiedler vector of a matrix A is the eigenvector corresponding to the Fiedler value. The Fiedler value is the second smallest eigenvalue of the Laplacian matrix of A ($L_A = \text{diag}(A\mathbf{1}) - A$).*

Here $\text{diag}(v)$ is the function that makes a matrix out of a vector v by putting the entries of the vector on the diagonal of a matrix of equal size. All non-diagonal elements of the matrix are zero.
$\mathbf{1}$ is an all-ones vector.
With this we get the following algorithm to find a ranking out of pairwise comparisons.

Algorithm 1 (SerialRank)**.** *(Fogel et al. [7]).*
Input: *A comparison matrix C as defined in (2.1.1).*
Result: *A permutation π.*

Step 1: *Compute the similarity matrix S from C (2.3).*
Step 2: *Compute the Laplacian matrix $L_S = \text{diag}(S\mathbf{1}) - S$.*
Step 3: *Compute a Fiedler vector of S.*
Step 4: *Find the permutation that sorts the Fiedler vector of S in either increasing or decreasing order to minimize the number of upsets.*

Minimizing the number of upsets means that we pick the permutation that would make the most sense for the problem we are solving. For example, for a given sports competition the algorithm finds two rankings, one by sorting in decreasing order and one by sorting in increasing order. Both rankings will be the exact opposite. One of these will have teams that won a lot of games as the highest ranking teams and one will have teams that lost a lot of games as the highest ranking teams. Teams that won most games should be high ranked, so we would pick the first permutation in this case. This decision is very easy to make when solving a seriation problem with the Algorithm SerialRank. A Matlab[19] implementation of this algorithm can be found in Appendix B.2. The following Theorem is found in another wording in Fogel et al. (Theorem 3.6, [7]).

Theorem 3.1.2 (Fogel et al., Theorem 3.6, [7])**.** *Given the pairwise comparisons of a totally ordered set, the algorithm SerialRank recovers the perfect ranking of the items.*

This Theorem tells us that the algorithm SerialRank (Algorithm 1) actually works and finds the permutation we are looking for. We have to take multiple steps in order to proof this. First of all, we note that the similarity matrix S created in step 1 is an pre-strict-R-matrix, as we described earlier as a consequence of Proposition 2.4.1. Theorem 3.1.2 now says that we can sort the Fiedler vector of this matrix, and that this permutation gives us the perfect ranking.

The perfect ranking is unique. This means that the permutation which induces the perfect ranking is also unique. Therefore, for Theorem 3.1.2 to hold, it can only output one valid permutation. This poses a couple of problems, because we are getting the permutation from sorting the Fiedler vector:

1. The Fiedler vector could contain complex entries, making it impossible to sort.
2. The Fiedler vector could not be unique, suggesting multiple possible permutations.
3. The Fiedler vector could contain repeated entries, suggesting multiple possible permutations.

We will show that the algorithm is not affected by any of these three problems. After that, we will prove that the algorithm indeed recovers the perfect ranking.

3.1.2 Proving Theorem 3.1.2

First of all, we consider the case in which the Fiedler value is zero. This can occur when the similarity matrix is a so-called reducible matrix.

Definition 3.1.3 (Reducible matrix)**.** *(Hiai et al., p.57, [10]) A square $n \times n$ matrix A is called reducible if there exists a permutation $\pi \in \mathcal{S}_n$ such that A_π is of the form $A_\pi = \begin{pmatrix} A_1 & B \\ 0 & A_2 \end{pmatrix}$, where A_1 and A_2 are square and nonempty matrices. A is called irreducible if it is not reducible.*

According to the Perron-Frobenius Theorem (Hogben, section 9.2, [11]), if the similarity matrix S is an irreducible matrix, there will be at least one positive eigenvalue and thus a positive Fiedler value. However, it is certainly possible that S is not an irreducible matrix. This problem is easily avoidable. If the matrix S is reducible, that means that there exists a permutation π such that S_π is a block diagonal matrix, since $B = 0$ because of symmetry. We can then apply the algorithm to each block separately and merge the results together to find a ranking induced by the first similarity matrix. If one of the smaller blocks happened to be a reducible matrix as well, we repeat the same process for that matrix. The following Theorem from Atkins et al. [2] describes this more formally.

Theorem 3.1.4 (Atkins et al. , Lemma 4.2, p. 303, [2]). *Let $S_i, i = 1, \ldots, k$, be the irreducible blocks of a pre-R-matrix S, and let π_i be a permutation of block S_i such that the submatrix $(S_i)_{\pi_i}$ is an R-matrix. Then the permutation formed by concatenating the π_i's will make S become an R-matrix.*

With this Theorem it suffices to show that the algorithm SerialRank (Algorithm 1) works for irreducible similarity matrices, so for now, we assume that this is the case.

Eigenvectors can contain complex entries. If the Fiedler vector would have complex entries we would not be able to sort it, luckily this will not be a problem for us. The similarity matrix S is a symmetric matrix and one can quickly see that the Laplacian matrix of S will also be a symmetric matrix. The eigenvalues and eigenvectors of a symmetric matrix are never complex and thus we know that the Fiedler vector has no complex entries and we will be able to sort it.

There exists only one perfect ranking. If there are multiple permutations that sort the Fiedler vector in ascending order(or descending order), the algorithm would give multiple permutations that induce the perfect ranking which would imply that there are multiple different perfect rankings. This should be impossible because the perfect ranking is unique.
The Fiedler vector is an eigenvector belonging to the Fiedler value, which is a smallest non zero eigenvalue. If the algebraic multiplicity of the Fiedler value is more than one, the Fiedler value induces multiple independent Fiedler vectors. We would have a choice of different eigenvectors to sort, which each eigenvector giving a possible different permutation. For the algorithm to work in every occasion the Fiedler value should have an algebraic multiplicity of one, because there is only one perfect ranking. In other words, the Fiedler value has to be a simple eigenvalue and the Fiedler vector should be a unique eigenvector up to a multiplicative constant.

Lemma 3.1.5. *Let S be an irreducible pre-R matrix, then S has a simple Fiedler value.*

Proof. There exists a permutation π that sorts the matrix S as a R-matrix S'. If x is a Fiedler vector of S, than we know that πx is a Fiedler vector of S' corresponding to the same Fiedler value. Therefore, it suffices to show that S' has a simple Fiedler value, implying that S has a simple Fiedler vector. S' is an R-matrix. This means that $S'_{n,1}$ is among its minimal elements. As Fogel et al. argues (Proof of Lemma 3.3, [7]), subtracting $S'_{n,1}$ from S' does not change the nonnegativity of S' and it still is an R-matrix. [Atkins et al., Theorem 4.6, [2]] now tells us that S' has a simple Fiedler value. □

An eigenvector could have repeated entries. If the Fiedler vector would have repeated entries there would not be an unique way of sorting the vector in ascending or descending order and thus the algorithm would give multiple permutations that induce perfect rankings, which is not possible. With results from Fogel et al. [7] we can prove the following lemma.

Lemma 3.1.6. *Let S be an irreducible pre-R-matrix, then the Fiedler vector of S does not have repeated entries.*

Proof. From [Fogel et al, Lemma 3.5, [7]] follows that if we have a irreducible R-matrix that is strict R, then there are no distinct indices $r < s$ such that for any $k \notin [r, s], A_{r,k} = A_{r+1,k} = \ldots = A_{s,k}$. By [Fogel et al, Lemma 3.4, [7]], such a matrix has a Fiedler vector that is strictly monotonic. Let π be the permutation that permutes S into an R-matrix, then S_π has an strictly monotonic Fiedler vector. This is a Fiedler vector without repeated entries. Permuting a matrix does not change entries of its Fiedler vector, but only the order. This means that if the Fiedler vector of S_π has no repeated entries, neither has the Fiedler vector of S. □

We now know that the Fiedler vector can be sorted and that the permutation to sort the Fiedler vector is unique. Left to show is that this permutation turns a pre-strict-R-matrix into a strict-R-matrix and thus induces the perfect ranking. Therefore we will need the following theorem found in Atkins et al.

Theorem 3.1.7. *(Atkins et al., Theorem 3.2, [2]) If A is an R-matrix then it has a monotone Fiedler vector.*

With this we can now prove the next major Theorem. For easier notation, we use permutation matrices to describe a permuted matrix. So consider a permutation π of n items and a $n \times n$-matrix A. The permuted matrix A_π, where the rows and columns of A are permuted according to permutation π, would be the same as $A_\pi = \Pi A \Pi^T$. Here $\Pi \in \{0,1\}^{n \times n}$ is the permutation matrix with $\Pi_{i,j} = 1$ if $\pi(i) = j$.

Theorem 3.1.8. *(Fogel et al., Theorem 3.2, [7]) Let S be an irreducible pre-R-matrix with a simple Fiedler value and a Fiedler vector v that has no repeated entries. Let Π_1 and Π_2 respectively be permutation matrices such that $\Pi_1 v$ is strictly increasing and $\Pi_2 v$ is strictly decreasing. Then $\Pi_1 S \Pi_1^T$ and $\Pi_2 S \Pi_2^T$ are R-matrices and no other permutations of S produce R-matrices.*

Proof. The Fiedler value of S is simple, so the Fiedler vector is unique. Because of this uniqueness we can observe that if v is the Fiedler vector of S, then Πv is the Fiedler vector of $\Pi S \Pi^T$ for any permutation matrix Π. So permuting the matrix permutes the Fiedler vector in the same way. Let Π_s be the permutation such that $\Pi_s S \Pi_s^T$ is an R-matrix. Theorem 3.1.7 tells that the Fiedler vector $\Pi_s v$ is monotone. From Lemma 3.1.6, we know its strictly monotone because there are no repeated entries and thus Π_s is the permutation that sorted the Fiedler vector in ascending or descending order. □

With these Lemmas and Theorems we can prove that the SerialRank algorithms works properly.

Proof of Theorem 3.1.2. Proposition 2.4.1 tells us that S is a pre-strict-R-matrix. Lemmata 3.1.6 and 3.1.5 tell us that the Fiedler value of this matrix is simple and the Fiedler vector has no repeated entries. Theorem 3.1.8 shows that only the permutations that sort the Fiedler vector permute S into an strict-R-matrix are the ones that sort the Fiedler vector in increasing or decreasing order. We know from Theorem 2.4.1 that a perfect ranking would give such an strict-R-matrix. We chose between the two potential perfect rankings by choosing the one with the least upsets to find the perfect ranking. □

We now have a proper method to solve the seriation problem. However, this is not the only method we are going to use. In the next section we will discuss a second way of solving the seriation problem.

3.2 2-SUM and QAP

3.2.1 2-SUM

Seminaroti [18] describes a different method of finding a solution to a seriation problem. Solving the so-called 2-SUM problem is another way to go from the described similarity matrix (Def. 2.1.2) to a ranking for the items by finding a permutation. The idea is to model this problem as a discrete optimisation problem. To do this we want to find a permutation while minimizing a given objective function, called the seriation measure. With $\mathcal{S}_n$ the set of all different permutations of $\{1, 2, \ldots, n\}$, the seriation measure of 2-SUM is defined in the following way.

Definition 3.2.1 (2-SUM problem)**.** *(Seminaroti, relation (7.2), [18])*

$$\min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{\pi(i),\pi(j)}(i-j)^2. \tag{3.1}$$

Where in our problem A_π is the similarity matrix with permuted rows and columns according to π. Intuitively, one can see that this criterion pushes high values in A, the items with high similarity, closer to the diagonal. Possibly so that the further away from the diagonal you go, the smaller the values get. This reminds of the special matrix we have seen before, the R-matrix (Def. 2.3.1). In Proposition 2.4.1 we have seen that such an R-matrix corresponds with a consistent or perfect ranking. So, it looks like the permutation which is optimal for the 2-SUM problem (3.1), is also the permutation that sorts A into an R-matrix and thus is the permutation that gives the consistent or perfect ranking we are looking for, if one of them exists.
The following result from Fogel et al. [8] shows that, when the pairwise comparisons induce a perfect ranking, the permutation that sorts the similarity matrix into an R-matrix is indeed the optimal solution to the 2-SUM problem (3.1). Before we can show the theorem, we will first have to introduce interval-cut matrices.

Definition 3.2.2 (Interval-cut matrix)**.** *(Seminaroti, p. 133, [18]) Given two integers u and v with $1 \leq u \leq v \leq n$. The interval-cut matrix $I(u,v)$ is the symmetric $n \times n$ matrix with*

$$I_{i,j} = \begin{cases} 1, \text{ if } u \leq i \text{ and } j \leq v \\ 0, \text{ otherwise} \end{cases} . \tag{3.2}$$

Theorem 3.2.3. *(Fogel et al., Theorem [8]) If a symmetric $n \times n$ matrix A can be written as a conic combination of interval- cut matrices, then the identity permutation is optimal for 2-SUM (3.1). More generally if, for some $\pi \in \mathcal{S}_n$, A_π can be written as a conic combination of interval-cut matrices, then π is optimal for 2-SUM (3.1).*

In Proposition 2.4.1 we have seen that the similarity matrix of a set with a perfect ranking corresponds to $S_{i,j} = n - |i-j|$. One can easily see that such a matrix can be written as a conic combination of interval-cut matrices. Thus, Theorem 3.2.3 shows that if the pairwise comparisons induce a perfect ranking, then the optimal solution for the 2-SUM problem (3.1) is the permutation that sorts the similarity matrix into the strict-R-matrix corresponding with the perfect ranking and so this permutation induces the perfect ranking we are looking for.
We now know that solving the 2-SUM problem can give us the perfect ranking if it exists. However, if there is no perfect ranking, the corresponding R-matrix will often not be able to be written as a conic combination of interval-cut matrices. Seminaroti [18] has removed the restriction of Theorem 3.2.3 and shown that solving 2-SUM works for every pre-R-matrix. Before we look at the theorem from Seminaroti (Theorem 3.2.8), we will first have to go back to a more general version of the 2-SUM problem, as it is a special version of the quadratic assignment problem.

3.2.2 QAP

The Quadratic Assignment Problem, QAP in short, is an optimisation problem introduced by Koopmans and Beckmann [15] in 1957 as a mathematical model for location problems. In an QAP(A,B), we have to assign n facilities to n different locations. We have a so-called flow matrix A, where each entry A_{ij} stands for the flow of activity between facility i and facility j. B is the distance matrix and each entry B_{ij} represents the distant between facility i and j. The objective of QAP is to assign the facilities to the locations in such a way so that the total cost of

the flows and distances combined is minimized. An example could be when new buildings have to be placed on building sites. The entries of flow matrix A would contain the amount of people that need to walk from one building to another on a given day. The entries of B would be the amount of meters needed to walk from one building site to another building site. The objective would then be to minimize the amount of meters walked by all people together in a given day. We can see a certain assignment of facilities to locations as a permutation of the flow matrix A. This explains the following definition of a QAP.

Definition 3.2.4 (QAP(A, B))**.** *(Burkard et al., pp. 203-206, [4]). Given a symmetric flow matrix A and a symmetric distance matrix B of sizes $n \times n$, find a permutation π of $\{1, \ldots, n\}$ minimizing:*

$$\min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{\pi(i)\pi(j)} B_{ij}. \tag{3.3}$$

From this definition one can quickly see that 2-SUM (3.1) is a special case of QAP. Namely the case where the entries of the distance matrix are $B_{ij} = (i - j)^2$.

Before we can formulate the theorem that proves that 2-SUM is a good way to solve a seriation problem we first need to introduce two special matrices.

Definition 3.2.5 (anti-R-matrix)**.** *The matrix B is called a Robinson dissimilarity matrix or anti-R-matrix if $-B$ is an R-matrix (Def. 2.3.1).*

In other words, where the entries of an R-matrix do not increase as we move away from the diagonal, the entries of an anti-R-matrix do not decrease as we move away from the diagonal. We call B pre-anti-R if $-B$ is pre-R (Def. 2.3.4).

Definition 3.2.6 (Toeplitz)**.** *A matrix B is called Toeplitz if $B_{ij} = B_{i+1,j+1}, \forall\ 1 \leq i, j \leq n-1$.*

In other words, the entries on the diagonals are all the same.

Example 3.2.7 (Toeplitz matrix)**.** *An example of an Toeplitz matrix B:*

$$B = \begin{bmatrix} 2 & 0 & 4 & 1 \\ 3 & 2 & 0 & 4 \\ 2 & 3 & 2 & 0 \\ 7 & 2 & 3 & 2 \end{bmatrix} \tag{3.4}$$

Seminaroti (Theorem 7.2.5, [18]) has shown that, with some restrictions, R-matrices are the optimal solution for the QAP and thus 2-SUM. He extends this to the following result.

Theorem 3.2.8. *(Seminaroti, Corollary 7.2.6, p. 144, [18]) Let A and B be symmetric $n \times n$ matrices. Assume that A is a pre-R-matrix and B an pre-anti-R-matrix. Let π and τ be permutations that reorder A and B as an R-matrix and anti-R-matrix respectively. Assume that one of the matrices A_π or B_τ is a Toeplitz matrix. Then the permutation $\tau^{-1}\pi$ is optimal for QAP(A, B).*

Earlier, with Theorem 3.2.3, we needed to be able to write the similarity matrix as a conic combination of interval-cut matrices to solve the 2-SUM problem. With Theorem 3.2.8, we can see that this restriction is not needed, as the matrix B of the 2-SUM problem always is an anti-R and Toeplitz matrix. The following example clarifies this.

Example 3.2.9 (B matrix from 2-SUM)**.** *We have seen that the 2-SUM problem (3.1) is a QAP(A, B) with $B_{ij} = (i-j)^2$. With $n = 4$ the dimensions of the $n \times n$ matrix A the matrix B would be:*

$$B = \begin{bmatrix} 0 & 1 & 2 & 3 \\ 1 & 0 & 1 & 2 \\ 2 & 1 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \tag{3.5}$$

From this example one can quickly derive that the matrix B from the 2-SUM problem will always be an anti-R-matrix and a Toeplitz matrix. Therefore we can use Theorem 3.2.8 with the 2-SUM problem.

Solving the seriation problem with QAP

In Chapter 2 we have seen that solving the seriation problem involves finding the permutation that sorts the pre-R similarity matrix as an R-matrix. With Theorem 3.2.8 we have a way to find such a permutation. Say we have a similarity matrix A from a given seriation problem that is pre-R. We can then choose an anti-R-matrix B that is Toeplitz and solve the QAP(A, B). Theorem 3.2.8 tells us that the optimal solution of this QAP will be a permutation that sorts A into an R-matrix and thus is the permutation that induces the perfect ranking.
We note again that the 2-SUM problem (3.1) is the same as an QAP with distance matrix $B_{ij} = (i-j)^2$, which is anti-R and Toeplitz. According to Theorem 3.2.8 this means the permutation π that would sort a pre-R-matrix A as an R-matrix is also the optimal solution of the 2-SUM problem (3.1).
Our goal now is to find a method to get a solution for the discrete optimisation problem QAP.

Gilmore-Lawler bound

The quadratic assignment problem is a NP-hard problem, so it can be very hard to find a solution. Therefore, bounds on the problem can be very useful. The Gilmore-Lawler bound (GLB)(Burkard et al. , Paragraph 7.5.1, pp. 225-227, [4]) is one of these bounds.
We consider a

$$QAP(A, B) = \min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{\pi(i)\pi(j)} B_{ij} = \min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} \sum_{j=1}^{n} A_{ij} B_{\pi(i)\pi(j)}.$$

First, we define the following minimum scaler product

$$\langle a, b \rangle^- = \min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} a_i b_{\pi(i)}, \text{ with } a, b \in \mathbb{R}^n. \tag{3.6}$$

Proposition 5.8 from Burkard et al. [4] tells us that this value can easily be calculated.
We now acquire the (n-1)-vectors $\hat{a}_i$ by taking the ith row of matrix $A = (a_{ij})$ from the QAP and deleting the element a_{ii}. We get a $\hat{b}_i$ for every row in matrix $B = (b_{ij})$ in the same way. We then note that for any $i \in \{1, \ldots, n\}$ and $\pi \in \mathcal{S}_n$ the following holds.

$$\langle \hat{a}_i, \hat{b}_{\pi(i)} \rangle^- + a_{ii} b_{\pi(i)\pi(i)} \leq \sum_{k=1}^{n} a_{ik} b_{\pi(i)\pi(k)}. \tag{3.7}$$

This gives us a lower bound for each row, given a certain permutation π. In order to find a lower bound on the optimum value of QAP, we want to find the permutation π which minimizes the

lower bounds. So we want to find the permutation π that satisfies:

$$\min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} \langle \hat{a}_i, \hat{b}_{\pi(i)} \rangle^- + a_{ii} b_{\pi(i)\pi(i)}. \tag{3.8}$$

To solve this we define a cost matrix $L = (l_{ij}) := (\langle \hat{a}_i, \hat{b}_j \rangle^- + a_{ii} b_{jj})$. So we need to find the permutation that satisfies:

$$\min_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} l_{i\pi(i)}.$$

This problem is called a *linear sum assignment problem* (LSAP) ([4], equation (1.6)). It is a well known problem within the optimisation discipline and there exist many algorithms to solve it. These algorithms have a fairly fast runtime. For example, the Hungarian algorithm ([4], pp 85-87) can be implemented with a runtime of $\mathcal{O}(n^3)$.
By solving the LSAP with cost matrix L we obtain the Gilmore-Lawler lower bound for the QAP .
We can obtain a upper bound by following the same procedure, but instead of $\langle a, b \rangle^-$ we use

$$\langle a, b \rangle^+ = \max_{\pi \in \mathcal{S}_n} \sum_{i=1}^{n} a_i b_{\pi(i)}.$$

In the next section we will discuss a way to find the optimal solution of the discrete optimisation problem QAP.

3.2.3 Simulated Annealing

As said before, a quadratic assignment problem, and thus the 2-SUM problem are NP-hard to solve. However, there are methods that can find a solution. One of these methods is simulated annealing.
Simulated annealing (SA) is a probabilistic technique based on simulating a thermodynamic process. It is designed to approximate the global optimum of a function without getting stuck in a local optimum worse than the global optimum. The first real study of this algorithm was done in 1983 by Kirckpatrick et al. [13].
The goal of the SA algorithm is to find the optimal solution for a given objective function $f(x)$. The SA process begins with a starting solution x and a parameter called the initial 'temperature' T_0. It is an iterative process, which means it finds a new solution after each iteration. In an iteration the algorithm finds a neighbouring solution x' of the starting solution and then calculates the costs $f(x)$ and $f(x')$ of these solutions. If the neighbouring solution x' has a lower cost than the initial solution, i.e. $f(x') < f(x)$, x' is accepted as the new solution. But when $f(x) > f(x')$, the neighbouring solution will not automatically get discarded as it would be with a local search algorithm. Instead, the worse neighbouring solution gets accepted as the new solution with a certain probability $\exp(-(f(x') - f(x))/T_0)$.
This possibility of accepting a worse solution makes it so the algorithm does not get stuck in a local optimum. The probability of it happening depends on two things. The first is the difference $f(x') - f(x)$ between the initial solutions score and the score of the neighbouring solution or in other words, how much worse is the new solution we try to accept. If this difference is very large, the changes of it being accepted becomes slim. The other influence on the probability comes from the current temperature T. The temperature start at an initial value T_0 and decreases as we go through more iterations of the algorithm. A high temperature makes it very likely we accept

a new solution and makes the objective function $f(x)$ of little importance. As the temperature decreases, the objective function becomes more and more significant to the point where the temperature approaches zero, making acceptance of a worse solution almost impossible. At that point the algorithm essentially is a local search algorithm where the solution becomes trapped in the lowest minima.
In summary, the SA algorithm for a given objective function $f(x)$ looks like this:

Algorithm 2 (Simulated Annealing)**.** *(Kirkpatrick et al., [13])*
Input:

- *Objective function* $f(x)$
- *Starting solution* x_0
- *Starting Temperature* T_0
- *Amount of iterations* I

Result: *An optimal solution for the objective function*
initialization, with $x = x_0$ *and* $T = T_0$*;*
for $i = 0$ **to** I **do**
  Find neighbouring solution x' *of* x *;*
  if $f(x') \leq f(x)$ **then**
    Accept x' *as new solution* x *;*
  else
    Accept x' *as new solution* x *with chance* $\exp(-(f(x') - f(x))/T)$*;*
  end
  Decrease the temperature T *according to a cooling method ;*
end

Convergence of SA algorithm

The idea of the simulated annealing algorithm is fairly intuitive, but can we know for sure that it always finds the optimal solution? Hajek (Theorem 1, 1998, [9]) has proven that certain SA algorithms indeed always convergence to the optimal solution in a probabilistic sense. He has shown that this convergence depends a bit on the initial values of the problem, but mostly on the cooling method that is used to decrease the temperature. In the next section we will describe how we handle the different aspects of the SA algorithm. Which each aspect we will explain the criteria Hajek has found in order to be able to prove the convergence of the algorithm. We will also explain whether we hold on to these criteria or not.

3.2.4 Solving 2-SUM with SA

We will use simulated annealing (Algorithm 2) to find a solution for the 2-SUM (3.1) problem. Naturally, the objective function of our SA algorithm will be the 2-SUM measure.

$$f(\pi) = \sum_{i=1}^{n} \sum_{j=1}^{n} A_{\pi(i),\pi(j)}(i-j)^2. \tag{3.9}$$

This means that the solution we are looking for is, of course, a permutation. From Theorem 3.2.8 we know that finding the optimal solution to the 2-SUM problem will give us the permutation that sorts the similarity matrix into an R-matrix. This is also the permutation which gives us

the perfect ranking as consequence of Proposition 2.4.1. We will now discuss how we approach the different aspects of the algorithm when solving the 2-SUM problem.

Starting solution

To start the algorithm we need a initial permutation π. We can either pick a random permutation or a permutation that we think might already be a good solution. An example could be the permutation we find by Algorithm 1 SerialRank. The upside of using such a permutation over a random one is that the convergence to the optimum solution is likely faster. The downside is that we need to compute this fixed permutation first. We will start by picking a random permutation, if it turns out that the computation time takes too long, we will use the fixed permutation given by SerialRank (Algorithm 1).

Neighbouring solution

We define the distance between to permutations π and π' as

$$d(\pi, \pi') = |\{i : \pi(i) \neq \pi'(i)\}|. \tag{3.10}$$

A Neighbouring solution of π will then be an element of the collection

$$\mathcal{N}_k(\pi) = \{\pi' \in \mathcal{S}_n : d(\pi, \pi') \leq k, \pi' \neq \pi\}. \tag{3.11}$$

Were k is an integer that can be chosen.
We will find our neighbouring solution for π by picking a random element from $\mathcal{N}_2(\pi)$, which is the same as switching to elements inside π.
In this way, it is possible to go from any permutation to any other permutation while only travelling through neighbours, in other words the set of permutations would form a connected graph if we connect neighbouring permutations. This is one of the criteria needed to prove the convergence of the algorithm, as shown by Hajek [9]. It is a very intuitive one, for when the graph would not be connected the possibility exists that the optimal solution is not reachable from the starting permutation.

Starting temperature

In the first iterations of the algorithm we want to make sure that it is very likely to accept a worse solution, even when the difference between the current solution and worse solution score is large. Therefore, we want to pick a temperature

$$T_0 > \max f(\pi) - \min f(\pi).$$

This ensures that, in the first iterations, solutions with the largest difference objective score possible still give a good change of picking the worse solution to use in the next iteration. Because we do not know $\max f(\pi)$ and $\min f(\pi)$, as $\min f(\pi)$ is exactly the solution we are trying to find, we can use upper and lower bounds for $f(\pi)$ instead. We will use the Gilmore-Lawler bounds (3.2.2). We can also double the starting temperature to make sure the algorithm has enough time to escape local optima.

Cooling method

As Tsuzuki et al. (section 2.7, [20]) shows, many ways have been found to define the cooling method of the SA algorithm. The only real restriction we have for a cooling method is that

the temperature has to decrease over time, making it so that the change of accepting a worse solution eventually approaches zero.
However, we are not able to proof for every cooling method that the algorithm convergence to the optimal solution. Bertsimas and Tsitsiklis [3] use Hajeks Theorem (Theorem 1, [9]) to show that using the following cooling method does ensure convergence of the algorithm.

$$T_k = \frac{T_0}{\ln(k)}.$$

Here T_0 is the starting temperature as defined earlier and k the iteration. Because this method ensures convergence to the optimal solution when performing infinite iterations, it is a very popular cooling method.
However, while the convergence with this method is certain, nothing is said about the speed of this convergence and it turns out that it is pretty slow in practice.
For our purpose, it is sufficient to use a rather simple but efficient cooling scheme, first used by Kirkpatrick et al. [13].
If T_k is our current temperature. The temperature used in the next iteration will be

$$T_{k+1} = \alpha T_k.$$

Where $\alpha \in (0, 1)$. Tsuzuki et al. (p. 13, [20]) numerical experiments have shown that a $\alpha = 0.99$ is a good choice for α. This cooling method will make the temperature converge rather quickly to zero, because we apply it in every iteration. By using this cooling method, we can not mathematically proof that the algorithm will actually convergence to the optimal solution. However, we can still almost guarantee that we find the optimal solution by performing multiple numerical experiments and comparing results, because our datasets are relatively small.

3.3 Summary so far

In Chapter 2 we have seen that sorting the similarity matrix as an R-matrix gives us the consistent or perfect ranking we are looking for. In this chapter we have described two algorithms (Alg. 1 and Alg. 2) that can find the permutation from which the desired ranking can be derived. However, as said before, it often happens that a consistent ranking does not exist for a given dataset. For those kind of data, the algorithms can still create a permutation. We want to test how good the rankings given by these permutations are. To do this we will apply our algorithms to various different datasets in the next chapter.

Chapter 4

Numerical results

In the previous section we have explored different methods that can find a solution for the seriation problem. In this chapter we apply our algorithms to different datasets in order to find a ranking. We will analyse these results. Sometimes a dataset comes with a given ranking for the items (e.g. a tournament result). We will call such a ranking the historical ranking. A historical ranking will be a useful tool in analysing the results from the algorithms.

4.1 Datasets

Their is a wide variety of datasets the seriation problem applies to, but almost all of them start with pairwise comparisons between items. The solving methods are made in such a way that they will always find a perfect ranking if such a ranking exists. In reality the perfect ranking often does not exist, but we can still apply the algorithms and see how good the results are.
All datasets we look at have an existing ranking. Our goal is to see if our algorithms produce a ranking that is equally good or arguably even better than the historical rankings for the datasets. A short description of the three data sets we use is found below.

Test set This dataset of nine items will serve as a test set. All comparisons are known and follow a total ordering, which means that there exists a perfect ranking. Our algorithms should always be able te find this ranking.

Dutch Korfbal League This is a small dataset of ten items. Not every comparison is known and a consistent ranking does not exist. We know the historical ranking of this competition and thus we can see how our algorithms results compare to the historical ranking.

Europe universities This is larger dataset of thirty items. All comparisons are known, but a consistent ranking does not exist. We will use our algorithms to combine three rankings into one. There exist many other methods to do this, which we will compare to our algorithms.

4.1.1 Test set

We have a test set of nine items, the numbers 1 through 9. We define the perfect ranking of these numbers as $(9, 8, 7, 6, 5, 4, 3, 2, 1)$. We will start of with a randomly generated ordering $(6, 8, 5, 9, 2, 4, 3, 7, 1)$, which the algorithms will have to sort. Before we can apply the algorithms we have to create a comparison matrix for this set:

$$C_{ij} = \begin{cases} 1 & \text{if number } i \text{ is higher than number j} \\ -1 & \text{if number } i \text{ is less than number j} \end{cases} \tag{4.1}$$

Applying this equation creates the following matrix:

$$C = \begin{bmatrix} 1 & -1 & 1 & -1 & 1 & 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 1 & -1 & 1 & 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & -1 & -1 & -1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & -1 & 1 \\ -1 & -1 & -1 & -1 & 1 & -1 & 1 & -1 & 1 \\ 1 & -1 & 1 & -1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix} \tag{4.2}$$

In the next section we will apply our algorithms to this comparison matrix. Because there exist a perfect ranking, this should yield us with the correct permutation to reorder the random ordering of the nine items into the perfect ranking. With this dataset it is very easy to see what the correct permutation should be, the first item in the ranking is the nine, which is in the fourth place in the starting order, meaning the permutation should start with a 4. The eight is located in the second place in the starting order, meaning that the second entry of the permutation should be a 2, etc.. In this way one can find the permutation $(4, 2, 8, 1, 3, 6, 7, 5, 9)$ to be the permutation we are looking for.

4.1.2 Dutch Korfbal League

For this dataset we will look at the results from the Dutch Korfbal League from the season 2015/2016 [14]. There are ten teams competing in this tournament, playing each opponent twice. Based on the results of these games the teams score points, two points for winning, one point for a tie and zero points if they lose. This point system is often used in sporting competitions and yields a ranking at the end of the tournament. We will compare the rankings of our algorithms with this historical ranking.
There are ten teams attending the League tournament. For easy reference we have numbered them according to alphabetical order. This will also be our starting order when inputting the data in the algorithms:

1	2	3	4	5	6	7	8	9	10
AW	Blauw Wit	Dalto	DOS'46	DVO	Fortuna	KZ	LDODK	PKC	TOP

Table 4.1: Korfbal League teams in their starting order.

The historical ranking, found by the point system from the tournament itself, is found below. From left to right we go from best ranked team to least ranked team. Meaning that team PKC has won the competition.

9	10	2	7	6	8	4	5	1	3
PKC	TOP	Blauw Wit	KZ	Fortuna	LDODK	DOS'46	DVO	AW	Dalto

Table 4.2: Historical ranking Korfbal League teams.

Because we numbered the starting order with numbers 1 through 10 (See Table 4.1), we can very quickly find the permutation that is needed to permute the starting order in the historical ranking. It is the same as the order of the numbers as depicted above the teams in Table 4.2. In

the same way, when we find a permutation later the numbers in the permutation will correspond to the different teams. Therefore, the permutation itself will be equal to the ranking it produces.

For our algorithms to find a ranking, we need pairwise comparisons, derived from the games the teams played against each other. We will translate the played games into a comparison matrix in the following way.

$$C_{ij} = \begin{cases} 1 & \text{if team } i \text{ beat team } j \text{ twice} \\ 0 & \text{if team } i \text{ beat and lost once to team } j \\ -1 & \text{if team } i \text{ lost to team } j \text{ twice} \end{cases} \tag{4.3}$$

Applying this equation with the starting order of the teams given by first table shown above, we receive this comparison matrix, hereby we used the results of the competition [14]:

$$C = \begin{bmatrix} 1 & -1 & 1 & -1 & 0 & -1 & 0 & 0 & -1 & -1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ -1 & -1 & 1 & -1 & 1 & -1 & -1 & 0 & -1 & -1 \\ 1 & -1 & 1 & 1 & 0 & -1 & 0 & 0 & -1 & -1 \\ 0 & 0 & -1 & 0 & 1 & 0 & -1 & 1 & -1 & -1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & -1 & -1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & -1 & -1 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & 1 & -1 & -1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & -1 & 1 \end{bmatrix} \tag{4.4}$$

Because each team played against each other twice it is possible that a team beats another team once but also loses to that team ones, resulting in a zero entry in the comparison matrix. A zero entry means that those two teams are not compared, there is no clear winner. Thereby not all comparisons are known, which means that a perfect ranking is not possible. The algorithms will thus try and find a consistent ranking instead.
There does not necessary exist a consistent ranking either. A team could have won almost all their games and thus ended up with the highest score en best place in the ranking, but they still may have lost to a team that lost a lot of games and has a low ranking, resulting in a non consistent final ranking. This means that it is very well possible that our algorithms will not produce a ranking similar to historical ranking of the tournament, as it is not presumably not a consistent ranking.

4.1.3 Europe Universities

There are a lot of good universities in Europe. From time to time an organisation or research center publishes a ranking for the top universities. These rankings often have many differences. We have picked out thirty of the top universities in Europe. The data consists of three different rankings for these universities made by three different organisations. We will try to combine these three rankings into a singular one. The three rankings can be found in the Appendix A. We numbered the universities 1 through 30 according to the first ranking, this will also be the starting order we use as input for the algorithms. As with the Korfbal League teams, the numbering will make a lot easier to switch between permutations and rankings, as they are essentially the same ordering of the numbers.

From the three rankings we will create a comparison matrix in the following way.

$$C_{ij} = \begin{cases} 1 & \text{if university } i \text{ is higher ranked than university } j \text{ in most rankings} \\ -1 & \text{if university } i \text{ is lower ranked than university } j \text{ in most rankings} \end{cases} \tag{4.5}$$

To make this matrix by hand would be a lot of work, but with a program it can quickly be computed, see Appendix B.1.

Voting Theory

Combining multiple rankings into a singular one can be compared to combining vote results to elect a winner. In our case we have three different authorities who all 'voted' for the best universities and from those results we try to find a ranking. There exist many methods to elect a winner from votes and many criteria on which these methods are marked good or bad. Wallis (2014, [22]) describes a couple of these methods and criteria in his book. To see if our algorithms produce reasonable rankings we can compare them to the rankings derived from more traditional methods.
However, we can not use just any traditional method. Most methods are designed to produce a single winner out of different votes, but we are interested in producing a whole ranking. Instead of altering a lot of methods to make them produce rankings instead of a winner we have chosen two methods that already produce a ranking and adhere to different good criteria inside the voting theory. We will use the so-called 'Borda count' method (Wallis, Section 2.6, [22]) and the 'Nanson' method (Wallis, Section 3.6, [22]) to combine the three given rankings into a single one. Both implementations can be found in Appendix B.6. The details of these methods are beyond the scope of this dissertation, since we only use their results for comparison.

4.2 Results

4.2.1 Means of analysis

We will apply the algorithms to the datasets and have a look at the different rankings the algorithms give. There are a couple of different criteria we can analyse with each result we get from the algorithms:

- The runtime of the algorithm.
- The amount of 'errors' present in the given ranking. We define a single error as follows: Item i is higher ranked than item j by the algorithm while the pairwise comparison implies that i should be lower ranked than j. We can count how many of these errors occur given a comparison matrix and ranking (see Matlab[19] code B.4).
- Comparison with historical ranking.
- Comparison with Gilmore-Lawler bounds (3.2.2)

4.2.2 Test set

The main reason we apply our algorithms to the test set is to check if our programs work correctly. All the algorithms should find the permutation that leads to the perfect ranking. This ranking should have no errors when compared to the comparison matrix and the lower Gilmore-Lawler bound should be close to the 2-SUM score of the permutation that the algorithm gives.

SerialRank

With Theorem 3.1.2 we have proven that the algorithm SerialRank should always give us the perfect ranking if it exists. To test if our Matlab[19] implementation (see Appendix B.2) is correct we apply the algorithm to the Test set as described earlier. We started with a randomly chosen order of the nine numbers in this test set. Running the algorithm with the corresponding comparison matrix (4.2) yielded the following Fielder vector and permutation derived from that.

$$\begin{bmatrix} 0.0855 \\ 0.3243 \\ 0.0000 \\ 0.5944 \\ -0.3243 \\ -0.0855 \\ -0.1849 \\ 0.1849 \\ -0.5944 \end{bmatrix}$$

$$\text{Permutation } = (4, 2, 8, 1, 3, 6, 7, 5, 9).$$

We can see that the permutation corresponds with sorting the found Fiedler vector in descending order. Applying this permutation to the shuffled numbers indeed yields us the numbers nine to one in decreasing order, which we defined as the perfect ranking of this set. This means that the algorithm works correctly for this dataset.
With finding the final permutation we also got some additional results.

$$\begin{aligned} \text{Number of errors} &= 0 \\ \text{2-SUM score} &= 3936 \\ \text{Lower Gilmore-Lawler bound} &= 3936 \end{aligned}$$

The found ranking has zero 'errors' when compared to the pairwise comparisons. This is what we would expect, because this data set has a perfect ranking.
The 2-SUM score (3.1) of this permutation was calculated at 3936. Interestingly, This score is equal to the Lower Gilmore-Lawler bound.
The starting order for the items should not make a difference on the results, as changing the starting order would accordingly change the order of the Fiedler vector. To make sure that this is indeed the case we have tested the algorithm with many different starting orders. We found that it indeed always yields the same results.
All these results make it very plausible that the implementation of the algorithm and calculation programs is correct. The algorithm needed approximately zero seconds to run.

Simulated Annealing

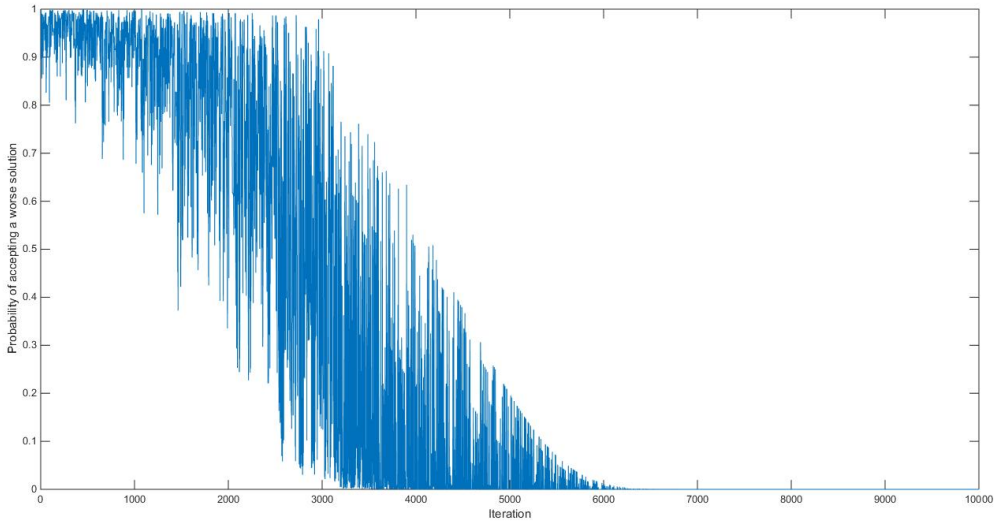
With the comparison matrix (4.2) from the test set we create the similarity matrix. Then we apply the Simulated Annealing algorithm 2 to solve the 2-SUM problem (3.1) with this similarity matrix. This gave us the following result.

$$\text{Permutation } = (4, 2, 8, 1, 3, 6, 7, 5, 9).$$

The algorithm yields the same desired permutation as the SerialRank algorithm did and therefore there are again no errors when this ranking is compared to the pairwise comparisons.
Before the algorithm settled on this final permutation we can see that the algorithm had 1339

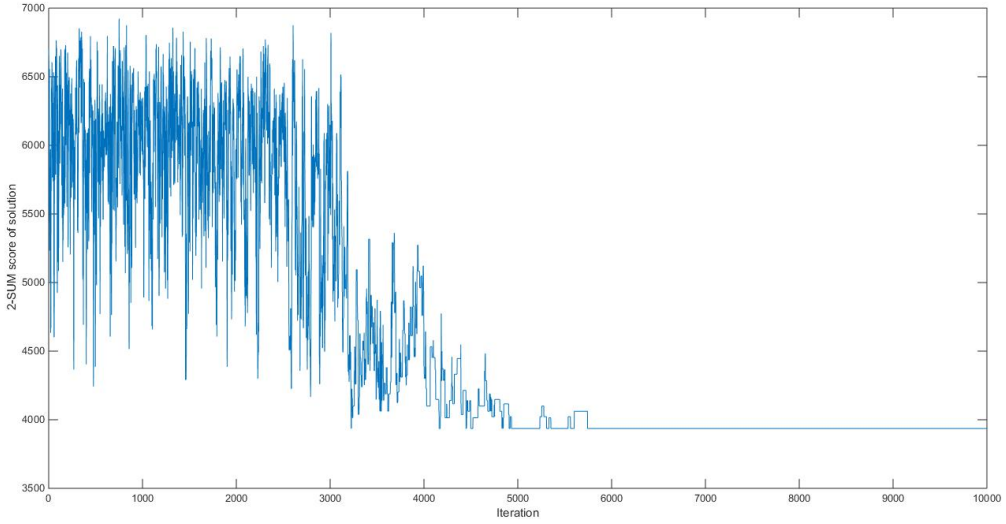
incidents where it accepted a worse solution. The next graph (Figure 4.1) will show what the probabilities were of accepting these worse solutions. For better visibility the cases where the new solution was better than the current solution, an thus when the probability of accepting was equal to 1, are not displayed in the graph.

Figure 4.1: The probability a worse solution is accepted



We can clearly see that the as we move through the iterations, the general probability of accepting a worse solution lowers. This is of course a consequence of the cooling method lowering the temperature with each iteration in the SA algorithm. The next graph (Figure 4.2) shows the 2-SUM scores of each iteration.

Figure 4.2: The 2-SUM score for every iteration



We can see that the score jumps all over the place in the first 3000 iterations. This is in sync with the probability of accepting worse solutions being very high at that point in time.

When the probability begins to lower, we can see that the scores begin to settle more towards the optimum value were it is eventually trapped at a score of 3936, equal to the lower Gilmore-Lawler bound. It is interesting to see that the scores never get close to the upper GLB of 7516. The algorithm needed approximately 2.2 seconds to run.

4.2.3 Korfbal League

Historical ranking

This dataset is about a korfball competition from 2015/2016. The competition has ended, so we know the historical ranking this competition produced. With our starting order in mind (Table 4.1), the historical ranking looks like this in permutation form.

$$\text{Permutation for historical ranking} = (9, 10, 2, 7, 6, 8, 4, 5, 1, 3).$$

The historical ranking itself, among with the other rankings we find in this section, can be found in Table 4.3 at the end of the section.
We can compare this ranking with the comparison matrix to find the number of errors. We can also determine the 2-SUM score of the historical ranking.

$$\begin{aligned} \text{Number of errors of historical ranking} &= 3 \\ \text{2-SUM score of historical ranking} &= 7951 \end{aligned}$$

SerialRank

With the comparison matrix (4.4) as input the algorithm SerialRank (1) found the following Fiedler vector and the permutation that sorts the Fiedler vector.

$$\begin{bmatrix} 0.2038 \\ -0.2738 \\ 0.4695 \\ 0.0603 \\ 0.3280 \\ -0.0648 \\ 0.0297 \\ 0.1818 \\ -0.6636 \\ -0.2709 \end{bmatrix}$$

$$\text{Permutation} = (9, 2, 10, 6, 7, 4, 8, 1, 4, 3).$$

It is interesting to see that in this case the Fiedler vector gets sorted in ascending order, where it was in descending order with the Test set.
The ranking that follows from this permutation, by applying it to the initial order, can be found in Table 4.3. The first thing we notice here is that the highest and lowest scoring team get the same rank compared to the historical ranking. The eight teams in between all get a different ranking than the tournament result gave them. The difference in these rank is never larger than one, as the eight teams are divided in four neighbouring pares that switch positions in the ranking compared to the historical ranking.
From the ranking given by SerialRank we also find the following results.

$$\begin{aligned} \text{Number of errors} &= 3 \\ \text{2-SUM score} &= 7787 \end{aligned}$$

This dataset does not have a perfect ranking. As a consequence, the ranking found by SerialRank has 3 errors when compared to the pairwise comparisons, instead of zero.
We can see that the number of errors is the same as with the historical ranking, which is surprising as the two rankings differ quite a bit. Because of this we could say that both rankings are equally good. However, the 2-SUM score of this ranking is lower than that of the historical ranking, implying that the newly found ranking is better. The runtime of the algorithm approximated zero seconds.

Simulated Annealing

The SA algorithm gives the following result.

$$\text{Permutation } = (9, 2, 10, 6, 7, 4, 1, 8, 4, 3).$$

The ranking this permutation produces can again be found in Table 4.3. We can see that it is almost exactly the same as the ranking found by SerialRank, only team 8 and team 1 got switched around.
This ranking found by SA gives us the following values.

$$\begin{aligned} \text{Number of errors} &= 3 \\ \text{2-SUM score} &= 7787 \end{aligned}$$

We also found that the Lower Gilmore-Lawler bound for the 2-SUM problem was equal to 7661 for this dataset. The 2-SUM found by the ranking is a bit higher than this bound, which is expected because there is no perfect ranking.
It is interesting to see that both the number of errors as the 2-SUM score given by the SA ranking is equal to those of the SerialRank ranking. This means that the SA algorithm could also have ended on the same ranking and thus that there is no unique optimal solution for the 2-SUM problem in this case.
The runtime of the algorithm was approximately 1.7 seconds.

Rankings

If we apply the three permutations to the initial order of the teams we get the rankings found in Table 4.3:

Historical ranking	Ranking by SerialRank	Ranking by SA
9 PKC	9 PKC	9 PKC
10 TOP	2 Blauw Wit	2 Blauw Wit
2 Blauw Wit	10 TOP	10 TOP
7 KZ	6 Fortuna	6 Fortuna
6 Fortuna	7 KZ	7 KZ
8 LDODK	4 DOS'46	4 DOS'46
4 DOS'46	8 LDODK	1 AW
5 DVO	1 AW	8 LDODK
1 AW	5 DVO	5 DVO
3 Dalto	3 Dalto	3 Dalto

Table 4.3: Ranking of Korfbal League teams

4.2.4 Universities

SerialRank

Applying SerialRank to the comparison matrix (4.5) yields the following permutation.

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 14, 20, 26, 15, 21, 12, 27, 16, 19, 22, 18, 24, 30, 28, 29, 13, 23, 17, 11, 25)$$

Applying this permutation gives the ranking found in Table 4.4 at the end of the section, the higher up in the table, the higher the university is ranked.
It is interesting to notice that the first ten elements stay in the exact same order they started in, which order is the same as one of the three rankings used in making the comparison matrix. With this ranking we get the following results.

$$\begin{aligned} \text{Number of errors} &= 12 \\ \text{2-SUM score} &= 1663262 \end{aligned}$$

The runtime of the algorithm was approximately 0.6 seconds.

Simulated Annealing

The SA algorithm gives us the following permutation.

$$(1, 2, 3, 4, 5, 6, 7, 8, 9, 14, 10, 20, 26, 15, 21, 12, 27, 16, 19, 22, 18, 24, 30, 28, 29, 23, 13, 17, 11, 25).$$

The ranking by applying this permutation to the initial order can be found in Table 4.4. We can see that the ranking is quite similar to the ranking found by SerialRank. There are two pairs of universities that are switched around. The SA ranking also gives us the following results.

$$\begin{aligned} \text{Number of errors} &= 10 \\ \text{2-SUM score} &= 1663138 \end{aligned}$$

This result is very interesting, as both values indicate that the SA ranking is better than the ranking found by SerialRank. Apparently, switching the two pairs of universities in the ranking causes two less errors when you compare the ranking to the pairwise comparisons. This also results in a lower 2-SUM score. The 2-SUM score is still always above the Lower Gilmore-Lawler bound, which is calculated at 1648901.
The runtime of the algorithm was approximately 13.1 seconds.

Borda Count and Nanson method

The rankings found by both classical methods can be found in Table 4.4. We have calculated the number of errors and 2-SUM score of these rankings.

$$\begin{aligned} \text{Number of errors of Borda Count} &= 36 \\ \text{2-SUM score of Borda Count} &= 1723094 \\ \text{Number of errors of Nanson method} &= 35 \\ \text{2-SUM score of Nanson method} &= 1754494 \end{aligned}$$

Both methods make quite an number of errors if we compare the ranking to the pairwise comparisons. A lot more than SerialRank and the 2-SUM method do. It is interesting to see that the Nanson method has one fewer error, but a higher 2-SUM score than the Borda count.

Combined University rankings

In Table 4.4 all rankings can be found. For the classical methods only the numbers we gave to the universities are displayed to make the table better readable.

Table 4.4: Combined rankings given by the algorithms and other methods

Ranking by SerialRank	Ranking by SA	Borda Count	Nanson
1 University of Cambridge	1 University of Cambridge	1	1
2 University of Oxford	2 University of Oxford	2	2
3 University College London	3 University College London	3	3
4 ETH Zurich	4 ETH Zurich	4	4
5 Imperial College London	5 Imperial College London	5	5
6 EPFL	6 EPFL	6	6
7 The University of Edinburgh	7 The University of Edinburg	7	7
8 King's College London	8 King's College London	9	14
9 The University of Manchester	9 The University of Manchester	8	8
10 École normale supérieure	14 The University of Sheffield	20	9
14 The University of Sheffield	10 École normale supérieure	10	10
20 Technical University of Munich	20 Technical University of Munich	15	20
26 KU Leuven	26 KU Leuven	14	26
15 Londen School of Economics	15 Londen School of Economics	12	15
21 École Polytechnique	21 École Polytechnique	21	21
12 University of Glasgow	12 University of Glasgow	26	12
27 University of Birmingham	27 University of Birmingham	16	16
16 Universität Heidelberg	16 Universität Heidelberg	27	19
19 LMU Munich	19 LMU Munich	18	13
22 University of Bristol	22 University of Bristol	22	27
18 University of Amsterdam	18 University of Amsterdam	19	18
24 University of Copenhagen	24 University of Copenhagen	13	11
30 University of Zurich	30 University of Zurich	24	22
28 Lund University	28 Lund University	17	17
29 University of Southampton	29 University of Southampton	11	24
13 The University of Warwick	23 TU Delft	30	30
23 TU Delft	13 The University of Warwick	23	23
17 University of Nottingham	17 University of Nottingham	28	28
11 University of St Andrews	11 University of St Andrews	29	29
25 Durham University	25 Durham University	25	25

We can see that there are many differences between the rankings we found with our algorithms and the classical methods, but also a lot of similarities. The same university almost never has a drastically higher or lower rank if we look at two different rankings in the table. Even better, the top seven scoring universities and the lowest scoring university are the same in all four rankings.

Chapter 5

Conclusion and Recommendations

5.1 Korfbal League ranking

With our algorithms, SerialRank (Alg. 1) and simulated annealing (Alg. 2), we have found two rankings for the teams in the Korfbal League competition. The rankings where almost the same, only two teams got switched around. Both rankings had a total of three errors. From this we can conclude that there exists no consistent ranking for this dataset, as we had assumed. Interestingly, while the two rankings differ a little bit, they still had the exact same 2-SUM score. This means that the ranking found by the simulated annealing algorithm was not unique, because both permutations that lead to the different ranking would be optimal.
The historical ranking, created by the point system of the tournament itself had a couple of differences when compared to our found rankings. The best and worst team were the same, but most other teams had a slightly different rank. Upon testing the historical ranking we found that it had three errors when compared to the pairwise comparisons, similar to the rankings we found. Based on that observation alone, we could conclude that the rankings found by SerialRank and SA are equally good when compared to the historical ranking if we look at errors made. However, the 2-SUM score of the historical ranking turned out to be higher than that of the other rankings. From this we could conclude that based on the 2-SUM measure, the historical ranking is actually worse than the two rankings we found with the algorithms.

All in all, we have found that the point system used in the Korfbal League competition to rank the different teams might not be the best method to find a ranking, at least not based on the criteria we used. Revising the ranking system or even replacing it with either SerialRank or the solving of a 2-SUM problem with simulated annealing can result in a ranking system that is even 'fairer' for all teams within.

5.2 Merging different rankings

We have applied the algorithms to the dataset of universities to try and merge three different rankings into a singular one. The two rankings we got from our algorithms were almost the same. The slight differences made us think that the ranking gained by solving the 2-SUM problem with SA was a little better than the ranking gained by SerialRank, as the one from SA had fewer errors and a lower 2-SUM score.
We have compared our algorithms results to two existing methods, the Borda Count and Nanson method. The rankings gained by the classical methods were not identical, but had a lot of similarities. The number of errors and the 2-SUM score were almost the same. We saw that there were many differences between the rankings of these classical methods and the SerialRank

and 2-SUM method, but there were also a significant number of universities that got the same rank or close to the same rank when comparing all four rankings.

The similarities between rankings give the implication that solving a seriation problem could very well be a good method to produce a singular ranking out of multiple ones or, if that is the goal, to produce a winner out of a set of votes. We could even argue that, based on the number of errors and 2-SUM measure, the rankings produced by solving the seriation problem were better. However, their are many more criteria one could apply when electing a winner out of a set of votes and testing all of these was beyond the scope of this dissertation. Still the results are promising and we would recommend the testing of more criteria for a further study.

5.3 Discussion: SerialRank versus 2-SUM

Below a quick summary of the differences in results from the two algorithms.

Test set The results from both algorithms were the same.

Korfbal set The final rankings of both algorithms had a slight difference, but both the errors as well as the 2-SUM score were equal.

University set The final rankings of both algorithms were slightly different. The ranking from the SA algorithm had fewer errors and a lower 2-SUM score.

We have also seen that the SerialRank algorithm had a significantly faster runtime when compared to the simulated annealing algorithm.
In all cases, solving the 2-SUM problem with SA was either equally good or better than using the SerialRank algorithm, since the number of errors and the 2-SUM score was always lower or equal with 2-SUM compared to SerialRank.
For small datasets however, the rankings were equally good. Therefore, we would advise the use of SerialRank for small datasets because of the very fast runtime. For the larger dataset, we have found that the result of solving the 2-SUM problem was better than using the SerialRank algorithm. The 2-SUM problem may be NP-hard, be we have seen that it can still be solved fairly quick using simulated annealing. So based on our results, we would advise the use of the SA algorithm to solve the 2-SUM problem when trying to find a ranking for a larger dataset.

It is important to note that these results came from just three datasets. Applying the algorithms in different situations could further enhance the understanding of these algorithms and this would be a recommendation for a further study.

Bibliography

[1] Aliprantis, C. D., & Burkinshaw, O. (1998). *Principles of real analysis* (3rd ed.). San Diego, USA: Academic Press.

[2] Atkins, J. E., Boman, E. G., & Hendrickson, B. (1998). A spectral algorithm for seriation and the consecutive ones problem. *SIAM Journal on Computing*, 28(1), 297-310. doi:10.1137/S0097539795285771

[3] Bertsimas, D., & Tsitsiklis, J. (1993). Simulated Annealing. *Statistical Science*, 8(1), 10-15. doi:10.1214/ss/1177011077

[4] Burkard, R., Dell'Amico, M., & Martello, S. (2009). *Assignment Problems: Revised Reprint*. Philadelphia, USA: SIAM.

[5] Cao, Y. (2011). *Hungarian Algorithm for Linear Assignment Problems (V2.3)* [Matlab-function]. MATLAB Central File Exchange. Retrieved from https://nl.mathworks.com/matlabcentral/fileexchange/20652-hungarian-algorithm-for-linear-assignment-problems--v2-3-

[6] Center for World University Rankings. (2016). *CWUR 2016 - World University Rankings* [Dataset]. Retrieved March 24, 2016, from http://cwur.org/2016.php

[7] Fogel, F., Daspremont, A., & Vojnovic, M. (2016). Spectral ranking using seriation. *ARXIV*. arXiv:1406.5370

[8] Fogel, F., Jennaton, R., Bach, F., & Daspremont, A. (2015). Convex Relaxations for Permutation Problems. *SIAM Journal on Matrix Analysis and Applications*, 36(4), 1465-1488. doi:10.1137/130947362

[9] Hajek, B. (1988). Cooling schedules for optimal annealing. *Mathematics of Operations Research*, 13(2), 311-329.

[10] Hiai, F., & Petz, D. (2014). *Introduction to matrix analysis and applications*. Cham: Springer. doi:10.1007/978-3-319-04150-6

[11] Hogben, L. (2006). *Handbook of linear algebra*. Boca Raton: Chapman & Hall/CRC.

[12] Kendall, M. G. and Smith, B. B. (1940). On the method of paired comparisons. *Biometrika*, 31(3-4), 324-345.

[13] Kirkpatrick, S. C. Gelatt, Jr. D., & Vecchi., M. P. (1983). Optimization by simulated annealing. *Science*, 220(4598), 671-680.

[14] Koninklijk Nederlands Korfbalverbond 2012 - 2017. (2016). *Korfbal League uitslagen* [Dataset]. Retrieved from http://www.knkv.nl/topkorfbal/korfballeague/uitslagen/

[15] Koopmans, T.C., & Beckmann., M. (1957). Assignment problems and the location of economic activities. *Econometrica*, 25(1), 53-76.

[16] Petrie, F. W. M. (1899). Sequences in prehistoric remains. *Journal of the Anthropological Institute*, 29(3-4), 295-301.

[17] QS Quacquarelli Symonds Limited. (2017). *QS World University Rankings* [Dataset]. Retrieved March 24, 2017, from https://www.topuniversities.com/university-rankings/world-university-rankings/2016

[18] Seminaroti, M. (2016). *Combinatorial algorithms for the seriation problem* [PHD thesis]. Tilburg: CentER, Center for Economic Research.

[19] The MathWorks, Inc. (2011). MatLab Student Version with Simulink, Version r2014b [Software]. Retrieved from http://www.mathworks.com/

[20] Tsuzuki, M.S.G., & Martins, T.C. (2014). *Simulated Annealing : Strategies, potential uses and advantages* Hauppauge, New York: Nova Science.

[21] U.S. News & World Report. (2016, October 24). *Best Global Universities in Europe* [Dataset]. Retrieved March 24, 2017, from https://www.usnews.com/education/best-global-universities/europe?int=9b5208

[22] Wallis, W. D. (2014). *The mathematics of elections and voting.* Cham: Springer. doi:10.1007/978-3-319-09810-4

Appendices

Appendix A

University rankings

Table A.1: Ranking from QS topuniversities [17]

1	University of Cambridge
2	University of Oxford
3	University College London
4	ETH Zurich
5	Imperial College London
6	EPFL
7	The University of Edinburgh
8	King's College London
9	The University of Manchester
10	École normale supérieure
11	Londen School of Economics
12	University of Bristol
13	The University of Warwick
14	École Polytechnique
15	University of Amsterdam
16	Technical University of Munich
17	TU Delft
18	University of Glasgow
19	LMU Munich
20	University of Copenhagen
21	Universitt Heidelberg
22	Lund University
23	Durham University
24	University of Nottingham
25	University of St Andrews
26	KU Leuven
27	University of Zurich
28	University of Birmingham
29	The University of Sheffield
30	University of Southampton

Table A.2: Ranking from U.S. News Education[21]

1	University of Oxford
2	University of Cambridge
3	Imperial College London
4	University College Londen
5	ETH Zurich
6	The University of Edinburgh
7	EPFL
8	King's College London
9	University of Copenhagen
10	KU Leuven
11	The University of Manchester
12	University of Amsterdam
13	Universitt Heidelberg
14	University of Zurich
15	University of Bristol
16	Technical University of Munich
17	Lund University
18	University of Southampton
19	University of Glasgow
20	University of Birmingham
21	The University of Sheffield
22	University of Nottingham
23	Durham University
24	The University of Warwick
25	TU Delft
26	École Polytechnique
27	École normale supérieure
28	University of St Andrews
29	Londen School of Economics
30	LMU Munich

Table A.3: Ranking from CWUR [6]

1	University of Cambridge
2	University of Oxford
3	ETH Zurich
4	University College London
5	Imperial College London
6	EPFL
7	École Polytechnique
8	École normale supérieure
9	The University of Edinburgh
10	The University of Manchester
11	University of Copenhagen
12	LMU Munich
13	KU Leuven
14	Universitt Heidelberg
15	Kings's College London
16	University of Zurich
17	Technical University of Munich
18	University of Amsterdam
19	Lund University
20	University of Bristol
21	University of Glasgow
22	University of Nottingham
23	University of Southampton
24	University of Birmingham
25	The University of Sheffield
26	Durham University
27	TU Delft
28	The University of Warwick
29	Londen School of Economics
30	University of St Andrews

Appendix B

Matlab code

B.1 Creating university comparison matrix

```
function [ Comp ] = Creatematrix()
%Creates the comparison matrix of the University dataset.
A = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
    25 26 27 28 29 30]; %ranking QS university
B = [2 1 5 3 4 7 6 8 20 26 9 15 21 27 12 16 22 30 18 28 29 24 23 13
    17 14 10 25 11 19]; %ranking U.S. News.
C = [1 2 4 3 5 6 14 10 7 9 20 19 26 21 8 27 16 15 22 12 18 24 30 28
    29 23 17 13 11 25]; %ranking CWUR

Comp = zeros(30,30);
%Creating the matrix.
for i = 1:30
%Run over all elements to compare.
        for j = 1:30
        %Run over all elements to be compared with.
                result = 0;
                Aplacei = find(A==i);
                %Find the places in the ranking of the elements that
                   we will compare.
                Aplacej = find(A==j);
                Bplacei = find(B==i);
                Bplacej = find(B==j);
                Cplacei = find(C==i);
                Cplacej = find(C==j);
                if Aplacei < Aplacej
                %Compare two elements in the first ranking.
                        result = result + 1;
                elseif Aplacei > Aplacej
                        result  = result - 1;
                end
                if Bplacei < Bplacej
                %Compare two elements in the second ranking.
                        result = result + 1;
                elseif Bplacei > Bplacej
```

```
31                          result  = result - 1;
32                  end
33                  if Cplacei < Cplacej
34                  %Compare two elements in the thirth ranking.
35                          result = result + 1;
36                  elseif Cplacei > Cplacej
37                          result  = result - 1;
38                  end
39                  if result > 0
40                  %If element i was higher more often than j it has a
                        positive entry in the comparison matrix.
41                          Comp(i, j) = 1;
42                  elseif result < 0
43                  % If element i was higher less often than j it has a
                        negative entry.
44                          Comp(i, j) = -1;
45                  end
46          end
47  end
48  diagonaalmat = diag(ones(30, 1));
49  %Elements are not compared to themselves before, and we give them the
        default 1.
50  Comp = Comp + diagonaalmat;
51  %Final matrix.
52  end
```

B.2 Serial Rank

```
1   function permutatie  = SerialRank(Comp)
2   %Creates the permutation used for sorting, and thus the final order
        of the target elements.
3   S = 0.5*(size(Comp,1)*ones(size(Comp,1)) + Comp*Comp')
4   %Constructs the Similarity matrix.
5   L = diag(S*ones(size(S,1), 1)) - S;
6   %Constructs the Laplacian matrix from the Similarity matrix.
7   [V, D] = eig(L);
8   %Calculates the eigenvectors and eigenvalues of the Laplacian matrix.
9   D = diag(D);
10  %Takes the eigenvalues as a vector.
11  [Dsort, plaats] = sort(D);
12  if round(Dsort(1), 10, 'decimals') == 0
13          Fiedlerplaats = plaats(2);
14  else
15          Fiedlerplaats = plaats(1);
16          %Finds the place of the smallest nonzero eigenvalue and
                corresponding eigenvector.
17  end
18  Fiedlervec = V(1:end , Fiedlerplaats)
```

```
%Gets the Fiedlervector of the similarity matrix.
[~, permutatie1] = sort(Fiedlervec, 'descend');
%Sorts the Fiedlervector in descending order and remembers the
    reverse permutation used.
[~, permutatie2] = sort(Fiedlervec, 'ascend');
%Sorts the Fiedlervector in ascending order and remembers the reverse
    permutation used.
if ErrorAmount(Comp, permutatie1) < ErrorAmount(Comp, permutatie2)
%Checks which permutation has the least errors.
        permutatie = permutatie1;
else
        permutatie = permutatie2;
end
end
```

B.3 SimulatedAnnealing

```
function [ Permutation ] = SimAnn( Comp, Permutation, Temperature,
    Time )
%Simulated Annealing 2-SUM. A function to perform Simulated annealing
    for the 2-SUM problem.
%Given a comparison matrix, starting permutation, starting
    temperature and
%iteration time, this function calculates the permutation that
    minimizes
%the 2-SUM problem.
Probabilities = [1];
% A list to keep track of the probability of choosing a worse
    solution.
counter = 0;
% A counter that tracks how often a worse solution is chosen.
Scores = zeros(1,Time-1);
% A list to keep track of the 2-SUM score of each iteration.
for t = 2:Time
        Startscore = sumscore(Comp, Permutation);
        %Caluculates the score of current permutation.
        Scores(t-1) = Startscore;
        NewPerm = NeighPermutation(Permutation, 2);
        %Creates a neighbouring permutation.
        Newscore = sumscore(Comp, NewPerm);
        %Calculates the score of neighbouring permutation.
        if Newscore <= Startscore
                Permutation = NewPerm;
                %If the New score is lower, we accept the
                    neighbouring permutation.
                Probabilities = [Probabilities, Probabilities(end)];
        else
                x = rand;
```

```
26          if x < (exp((Startscore - Newscore)/Temperature))
27                  Permutation = NewPerm;
28                  % If the New score is higher, we accept the
                        neighbouring permutation with a certain change.
29                  counter = counter  + 1;
30          end
31                  Probabilities = [Probabilities, exp((Startscore -
                        Newscore)/Temperature)];
32          end
33          Temperature = Temperature * 0.999;
34          %Decreases the Temperature.
35  end
36  if ErrorAmount(Comp, Permutation) > ErrorAmount(Comp, fliplr(
        Permutation))
37          Permutation = fliplr(Permutation);
38          %Flips the permutation if necessary.
39  end
40  fig1 = figure();
41  %Creates graphs of the data.
42  plot(Probabilities)
43  xlabel('Iteration');
44  ylabel('Probability of accepting a worse solution');
45  fig2 = figure();
46  plot(Scores)
47  xlabel('Iteration');
48  ylabel('2-SUM score of solution');
49  AcceptingWorseSolution = counter + 1 - 1
50  end
```

B.4 ErrorAmount

```
1  function [ errors ] = ErrorAmount( Comp, permutation )
2  %Calculates number of errors between final ranking and comparison
       matrix.
3  % Checks for every pair of elements if the ranking in the permutation
4  % corresponds with the Comparison matrix.
5  errors = 0;
6  len = length(permutation);
7  for i = 1:(len-1)
8          for j = (i+1):(len)
9                  if Comp(permutation(i),permutation(j)) < 0
10                         errors = errors + 1;
11                 end
12         end
13 end
14 end
```

B.5 Gilmore-Lawler bounds

As explained in Section 3.2.2, to calculate the Gilmore-Lawler bound we have to solve a LSAP. To do this we use the Hungarian method. We use a Matlab [19] implementation written by Y. Cao [5]. In the program below, this function is called munkres().

```
function [ score ] = GilLaw( A, B, Bound )
%Calculates the Gilmore-Lawler upper and lower bound for a given QAP(A,B).
n = size(A,1);
L = zeros(n);
for i = 1:n
        a = A(i, :);
        a(i) = [];
        a = sort(a, 'ascend');
        aii = A(i,i);
        for j = 1:n
                b = B(j,:);
                b(j) = [];
                if Bound == 1
                        b = sort(b, 'ascend');
                else
                        b = sort(b, 'descend');
                end
                bii = B(j,j);
                L(i,j) =(a*b') + (aii*bii);
        end
end
%Calculates the cost matrix L
[~, score] = munkres(L);
%Determines the solution to the LSAP with cost matrix L
end
```

B.6 Borda count & Nanson method

```
function [ ranking ] = Borda()
% Determines the ranking found by the Borda Count from three given rankings.
A = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30]; %ranking QS university
B = [2 1 5 3 4 7 6 8 20 26 9 15 21 27 12 16 22 30 18 28 29 24 23 13 17 14 10 25 11 19]; %ranking U.S. News.
C = [1 2 4 3 5 6 14 10 7 9 20 19 26 21 8 27 16 15 22 12 18 24 30 28 29 23 17 13 11 25]; %ranking CWUR
Points = linspace(1.0, 0.0, 30);
Scores = zeros(1,30);
for i = 1:30
        Scores(A(i)) = Scores(A(i)) + Points(i);
        Scores(B(i)) = Scores(B(i)) + Points(i);
```

```
        Scores(C(i)) = Scores(C(i)) + Points(i);
        %Gives points to the universities based on their ranking.
end
[~, ranking] = sort(Scores, 'descend');
%The more points a university scores, the higher they get ranked.
end
```

```
function [ Ranking ] = Nanson( )
% Determines the ranking found by the Nanson method from three given
    rankings.
A = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
    25 26 27 28 29 30]; %ranking QS university
B = [2 1 5 3 4 7 6 8 20 26 9 15 21 27 12 16 22 30 18 28 29 24 23 13
    17 14 10 25 11 19]; %ranking U.S. News.
C = [1 2 4 3 5 6 14 10 7 9 20 19 26 21 8 27 16 15 22 12 18 24 30 28
    29 23 17 13 11 25]; %ranking CWUR
Ranking = zeros(1,30);
for k = 1:29
        Points = linspace(2.0, 1.0, length(A));
        Scores = zeros(1,30);
        for i = 1:length(A)
                Scores(A(i)) = Scores(A(i)) + Points(i);
                Scores(B(i)) = Scores(B(i)) + Points(i);
                Scores(C(i)) = Scores(C(i)) + Points(i);
                %Gives points to the universities based on their
                    ranking.
        end
        m=min(Scores(Scores>0));
        loser = find(Scores==m);
        loser = loser(1);
        %Determines the lowest ranking university
        Ranking(length(A)) = loser;
        %Adds it to the end of the final ranking.
        A = A(A~=loser);
        B = B(B~=loser);
        C = C(C~=loser);
        %Removes the losing university and repeats the process.
end
Ranking(1) = A(1);
%Adds the last university.
end
```