Rank aggregation by criteria

Minimizing the maximum Kendall-τ distance

– Diplomarbeit –

zur Erlangung des akademischen Grades Diplom-Mathematiker

eingereicht von

Niko Schwarz

niko.schwarz@googlemail.com

Betreuer

Dipl.-Bioinf. Nadja Betzler
Prof. Dr. Rolf Niedermeier

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Lehrstuhl Theoretische Informatik I
Institut für Mathematik und Informatik
Ernst-Abbe-Platz 2
07743 Jena
Germany

22. April 2009

Abstract

We propose a new method of combining ranking results which each rank the same set of items according to different criteria. It will choose a ranking that is closest as possible to each ranking result to be combined. In the context of the Internet, it can be used to rank web pages into an order that best reflects a balance between several criteria. In the context of sports, we propose to use the method to determine the winner of competitions, when the performance of an athlete is naturally judged according to different criteria, such as figure skating and show jumping.

The rank aggregation method we propose is known to be NP-hard. This thesis aims to develop efficient algorithms to compute aggregated rankings for practically relevant instances. By employing parameterized complexity theory, we can identify the structural hardness of an instance and allow for choosing a high-performing algorithm accordingly.

We present efficient and effective data reduction rules which will reduce parameters measuring the structural difficulty of an instance provably by simplifying the instance or removing unneeded parts.

We provide efficient search tree algorithms which will solve practically relevant instances, where the criteria correlate strongly. Experiments with synthetic data confirm that for instances with high correlation between the rankings, even large instances can be computed in short time.

For general instances, we present two enumeration algorithms, which will likely outperform the trivial algorithm of trying all rankings and comparing their qualities as aggregations. We prove the enumeration algorithms to perform well in scenarios with both few items to be ranked and some correlation between the rankings.

We present methods to approximate the solution quality by a factor of two. We present parameters which can not be used to improve computation by proving and we will prove the NP-hardness of the problem even for small values of these parameters.

Contents

CHAPTER 1

Introduction

There appear to be many scenarios where items need to be ranked according to several criteria—a search engine is just one instance. The criteria need to be balanced with one another to find a ranking that represents all criteria. Where possible, one can rank all items separately for each criterion. The separate rankings can be aggregated into one. This process is called "rank aggregation."

A well-studied method of aggregating several rankings into one is the Kemeny-Young method [38]. The Kemeny-Young method looks at the rankings to be aggregated as error-prone, independent judges of the same, sole criterion. The idea of the Kemeny-Young method stems from Condorcet [13], whose work was generalized into the Kemeny-Young method. He referred to Rousseau's Social Contract [51] as follows:

> When in the popular assembly a law is proposed, what the people is asked is not exactly whether it approves or rejects the proposal, but whether it is in conformity with the general will, which is their will. Each man, in giving his vote, states his opinion on that point; and the general will is found by counting votes. When therefore the opinion that is contrary to my own prevails, this proves neither more nor less that I was mistaken, and that what I thought to be the general will was not so. —J.J. Rousseau [52, p. 93]

As M. Truchon puts it:

> Condorcet's objective was to formulate this proposition rigorously, using the calculus of probability, which was new at that time. There is a best alternative, a second best, etc. Voters may have different opinions because they are imperfect judges. However, if they are right more often than they are wrong, then the opinion of the majority should yield the true order of the alternatives. —M. Truchon [59]

In this thesis, we propose an aggregation method closely related to the Kemeny-Young method, but based on the assumption that the judges are not imperfect approximations of the true ranking but rather perfect judges of their own specific criteria, and that an optimal aggregation is identified as the best possible in all criteria. In other words, we propose an aggregation that tries to resemble each judge's opinion as closely as possible.

To compare two rankings, we propose using the Kendall-τ distance. A ranking can be turned into any other through repeated swapping of adjacent entries. The Kendall-τ distance between two rankings is the number of adjacent swaps required to obtain one list from another. This matches our intuition: the two entries might be similar and it could be difficult to tell which of them to rank ahead.

We propose aggregating several rankings into the aggregation that has the least maximum Kendall-τ distance to the input rankings. We refer to the computational problem of finding such an aggregation as the CENTER RANKING problem, and to

its usage as an aggregation method as the CENTER RANKING method. The CENTER RANKING problem is known to be NP-hard by Popov [50][1] .

The CENTER RANKING method contrasts to the well-studied Kemeny-Young method, which aggregates several input rankings into the ranking that has the least possible total distance to all input rankings. Let us study the aggregation method we propose in an example and compare it directly to the Kemeny-Young method. Search engines aim to deliver a ranking of websites when given a certain input phrase; the quality of the ranking can be described using specific criteria. For some criteria that define a good search engine ranking, it is difficult to find a ranking to represent them, such as the first few results delivered by the search engine should be as diverse as possible, which would be useful if the search term has many common meanings. However, for many other criteria, it appears possible to determine a ranking representing these criteria. Such criteria could include the quality of the match of a website to the search phrase, the number of links pointing to it, the website's probability of not being spam, etc. Quite naturally, we would not want to give up any of the above criteria in favor of another. We can, however, determine the rankings and aggregate them using the CENTER RANKING aggregation method.

Let us consider three websites called 1, 2, and 3. Three rankings of the web pages according to three criteria are presented in Table 1.

Table 1: Rankings of three web pages 1, 2, and 3 according to various criteria. Here, "$<$" means "better than." The Kemeny-Young aggregation is $1 < 2 < 3$, while the CENTER RANKING aggregation is $2 < 1 < 3$.

criterion	ranking
number of links from other websites	$1 < 2 < 3$
number of occurrences of the search term	$1 < 2 < 3$
probability of not being spam	$2 < 3 < 1$

Using the Kemeny-Young method, the aggregation we obtain is $1 < 2 < 3$. But, 1 is almost certainly spam. Apparently, two votes just overrule one in the Kemeny-Young method. In our example, that is unacceptable. Spam is spam and we should refrain from putting something at the top of the result list that is very probably spam.

If we choose to use the CENTER RANKING aggregation method, where no criterion can be easily overruled, we arrive at the aggregation $2 < 1 < 3$. Here, 2 might not fit the search term so well; but then again, it is not spam. The aggregation appears to be more sensible.

If only two rankings are aggregated, the Kemeny-Young method regards both input rankings as optimum aggregations, while the CENTER RANKING method considers only a balance of both input rankings to be of optimum quality. The Kemeny-Young method tries to obtain an average of the input rankings, thus focusing on the multitude of the occurrences of certain relative rankings. In contrast, the CENTER RANKING method tries to obtain a ranking that resembles every input ranking as closely as possible by minimizing the maximum distance to the input rankings, completely disregarding of multitude.

Despite the NP-hardness, just as was argued for the Kemeny-Young method [43], we claim that in practice the criteria-based ranking can be determined in a short

[1]Note that in his paper, he refers to the CENTER RANKING problem as "ADJACENT SWAP CENTER PERMUTATION PROBLEM."

time even on a vintage laptop computer. To establish just that, we investigate a bevy of algorithms which aggregate rankings using our method in very short time for a large range of different instances of the problem.

To provide provable bounds of the qualities of the presented algorithms, we employ parameterized complexity [18, 47, 21] analysis as pioneered by Downey and Fellows. An outline of parameterized complexity analysis can be found in Section 1.4.4.

1.1. Aggregating ranks by criteria

When the performance of a competitor in a competition is determined by different and not easily comparable criteria, the aggregation method used still needs to be able to determine the final ranking of the competitors.

In figure skating contests, the couples receive both a Technical Score and a Program Component Score. Then, "the final score is calculated by adding the total Technical Score and the Program Component Score and subtracting any program deductions (for example 1.0 for a fall)"[62]. In figure skating, program deductions are used to enforce a certain criterion, namely to make it significantly more difficult to win if certain rules are broken. In show jumping contests against the clock, riders are urged not to knock down any obstacles while staying within the time limit. For each second that a competitor stays behind the time limit, he or she will receive a penalty point; likewise, knocking off obstacles will result in penalty points. The riders with the least penalty points (often nil) will finally enter a jump-off. Both figure skating and show jumping competitions, in principle, allow for compensating failure in one criterion with success in the other. However, compensating knocking down an obstacle by speeding up is limited to competing the course within the time limit. When a horse disobeys for the first time, i.e. refuses to jump over an obstacle once, the rider receives 4 penalties. The second refusal will lead to the rider's elimination from the competition. Figure skaters are penalized heavily for disobeying a certain rule, e.g. playing vocal music. It appears that the desired effect of many penalty rules is near complete enforcement of a certain rule, i.e. to lower significantly the final rank of the competitor that violated a certain rule. The value of the penalty or program deduction imposed needs to be fine-tuned to show the desired effect. Many other sports have similar concepts.

The CENTER RANKING rank aggregation method that we propose provides another concept which strongly enforces certain rules, i.e. significantly lowers the rank of a competitor. It does this without requiring the creation of penalty scores—which seems to be a burdensome business, because the value of the penalty must be compared directly to the other scores a competitor might obtain. Instead, we propose ranking for each criterion separately. Ranking the competitors according to their compliance to this rule (or criterion) might be trivial. Then, we propose aggregating the ranking that expresses the performance in the competition with the rankings for compliance with certain rules. The aggregation we propose is defined to be the closest possible to each separate ranking.

In the example of show jumping, we have argued that the rules try to severely punish the disobedience of a rider. The CENTER RANKING aggregation method might start by ranking the riders three times, by their knock-down penalties, by their time to complete the course and finally by their penalties for disobedience. As an example, let us assume four riders A,B,C,D which are ranked as seen in Table 2. All rankings are fairly similar to A<B<C<D, except that A raced through the course much too fast, knocking down all obstacles, C surprisingly had fewer penalties for disobedience than B. The CENTER RANKING aggregation lists A on the second rank. A competitor who completely failed in the criterion of knocking

down as few obstacles as possible cannot assume to win. On the other hand, B wins over C, since otherwise the final ranking would only poorly reflect the ranking of the times to complete the course. Of course, D is ranked last.

Table 2: Three rankings of candidates A,B,C,D in a show jumping competition. Here, "X<Y" means "X performed better than Y." When aggregated by the CENTER RANKING aggregation method, the only best ranking is B<A<C<D.

criterion	ranking
ranking by time to complete the course	A<B<C<D
ranking by knock-downs	B<C<D<A
ranking by disobediences	A<C<B<D

It might be worthwhile to note that there is a chance to have several optimal aggregations when using the CENTER RANKING method. In the example of Table 2, there is only one optimal aggregation.

1.1.1. Mixing and post-processing the CENTER RANKING aggregation. The CENTER RANKING aggregation method shares a common advantage with all aggregation methods. Even if we are interested in only the first position of a ranking, we receive a complete ranking. While this might appear wasteful at first, it leaves room for corrections in case our modelling of the criteria was imprecise. Receiving the complete list enables us to identify the runners-up and compare them with the winner. We can change the final ranking to match criteria that were not part of the aggregation.

Another advantage of the CENTER RANKING aggregation method would be that, in order to aggregate several rankings, it allows itself to be mixed with the Kemeny-Young method, which might be tempting as one of them allows compensation while the other one does not. We could aggregate rankings that are meant to represent the same criterion but are error-prone, using the Kemeny-Young method, and then combine the aggregation with other rankings that reflect criteria using the CENTER RANKING aggregation method.

For example, a DVD rental store can have one ranking of all movies for each user, representing the users' ratings of the movies. There might be several thousand rankings representing the users' votes and another one representing the number of scratches on DVDs. Here, we could aggregate the users' votes using the Kemeny-Young method and then combine it with the scratch-rating using the CENTER RANKING aggregation method. For it might be a significant drawback of a DVD to contain scratches. The drawback can hardly be compensated by more and more good reviews.

Known results. Despite the many interesting applications, and indeed quite surprisingly, the only research on the CENTER RANKING problem known to the author is the proof of NP-hardness by Popov [50][2]. In Popov's work, the problem is found to be of interest to bio-informatics.

1.2. Main results

This section gives an overview of the most original, novel and exciting results of this thesis.

The proposal of using the CENTER RANKING method for obtaining aggregations that balance between several criteria appears to be original to this thesis. Currently,

[2]Note that in his paper, he refers to the CENTER RANKING problem as "ADJACENT SWAP CENTER PERMUTATION PROBLEM."

in many sports compliance to criteria is measured in penalties, which must be fine-tuned to be fair and to show the desired effect, since every penalty is directly comparable to every other penalty. This author hopes that the CENTER RANKING aggregation method provides a way to determine the final ranking of all competitors in a way that depends less on the (possibly debatable) decisions on the exact sizes of the penalties and still provides a final ranking of the competitors that matches common sense.

It appears to be plausible to expect a correlation between several rankings representing criteria that constitute the overall quality of the ranked items. The problem is NP-hard, but we can expect at least some correlation between the rankings that are aggregated. Thus, the neighbor permutation search algorithm appears to be of interest. The neighbor permutation search algorithm in Section 4.2 solves instances of high correlation between the rankings in only short time. While the general idea of limiting the search tree to logarithmic depth by branching into many cases on every node in the search tree has been provided by Ma and Sun [44], the CENTER RANKING problem is more difficult than the CLOSEST STRING problem (we can reduce CLOSEST STRING to CENTER RANKING such that the reduced instance closely resembles the original instance, see Section 2.2.1). It is difficult to adapt the branching proposed by Ma and Sun, because strings can easily be truncated to sub-strings, as the algorithm by Ma and Sun does, but truncating a ranking to some of its relative rankings requires careful and complex modelling.

This author believes the sphere intersection enumeration algorithm in Section 5.3 to provide a novel and interesting technique of enumerating permutations. The algorithm achieves to enumerate the intersection of two balls in the metric space of permutations, with respect to the Kendall-τ distance, to be enumerated in time that is linear in the output size. This is surprising since the computation of the Kendall-τ distance alone could be expected to take more than linear time per permutation that is enumerated. To achieve this surprising performance, the algorithm does not compute Kendall-τ distances directly, but rather computes numbers of inversions and spreads these computations so evenly across the enumeration, that no asymptotic extra-cost is required. Further, the algorithm directly enumerates the intersection of two balls, it does not need to select an intersection out of a greater set. The algorithm builds up the permutations within the enumeration digit-wise and thereby never errs and rejects a choice that was made earlier in the build-up process. Therefore, together with the astonishingly little work required per enumeration item, the algorithm is expected to perform very well in practice.

1.3. Organization

This thesis is divided into the following chapters.

- Chapter 1 presents the motivation of our analysis and the preliminaries needed for the later chapters.
- Chapter 2 presents the CENTER RANKING problem from an election theory point of view, introduces data reduction rules, discusses the relation to the KEMENY RANKING problem and CLOSEST STRING problem.
- Chapter 3 examines fixed-parameter tractability with respect to the most interesting parameters.
- Chapter 4 presents parameterized algorithms for instances where the rankings to be aggregate correlate strongly. For these, the radius parameter is small.
- Chapter 5 presents parameterized algorithms for general instances, which likely outperforms the trivial algorithm enumerating all possible rankings.

- Chapter 6 presents the results of experiments to determine the practical run times of some of the algorithms presented.
- Chapter 7 presents a summary of the algorithms examined and lists questions that appeared, or remained open.

1.4. Preliminaries

The basic concepts and notations we will use are presented in this section.

1.4.1. Rankings. We define a ranking or vote as a total order of the set $[n] = \{1, \dots, n\}$ for $n \in \mathbb{N}$. The set of all rankings of set $[n]$ will be denoted as $\mathrm{Sym}_{[n]}$.

A ranking can represent the preference of a voter on the candidates available in an election, where the candidates would be represented by numbers. In social choice theory, a ranking according to the preference of a voter is called a "vote." The entries of the votes are called "candidates." Because of the close relation to voting theory we will use the terms "votes" and "rankings" as synonyms, and hence we will refer to the ranked entries as "candidates." Since a total order on the set $[n]$ can be identified with permutations, we further identify rankings and permutations, perhaps oddly therefore referring to the entries of permutations as "candidates" and to permutations themselves as "votes."

1.4.2. Kendall-τ distance. Different rankings can be compared by their distance. Numerous metrics on rankings have been proposed. Perhaps the most natural one (e.g. argued by Noether [48]) is the Kendall-τ distance, proposed by Kendall [39], which is central for our investigations. We may assume that the smallest possible difference between two rankings may be that they list two adjacent candidates in different order.

Each unordered pair of candidates that two rankings list in different order may be seen as a *discordance*, as Kendall names it [39].

Definition 1 (Kendall [39])**.** Let $\lambda = p_1 p_2 \cdots p_n$ and $\mu = q_1 q_2 \dots q_n$ be two permutations. Then $\{p_i, p_j\}$ is a discordance between λ and μ if and only if λ lists p_i before p_j, but μ does not.

The concept of discordances between two permutations allows for a natural way to define the Kendall-τ distance.

Definition 2 (Kendall-τ distance)**.** Let $\mu, \lambda \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. The Kendall-$\tau$ distance is equivalent to the number of discordances between μ and λ,

$$\tau(\mu, \lambda) \coloneqq \#\{(a, b) | a <_\mu b \wedge a >_\lambda b\}.$$

Here, # denotes the cardinality of a set. Note that the number of discordances between two permutations equals the number of adjacent swaps required to turn one permutation into the other.

The Kendall-τ distance is a metric which means that for three permutations $\mu, \lambda, \nu \in \mathrm{Sym}_{[n]}$, the following holds.

- $\tau(\lambda, \mu) = 0$ if and only if $\lambda = \mu$ (identity of indiscernibles)
- $\tau(\lambda, \mu) = \tau(\mu, \lambda)$ (symmetry)
- $\tau(\lambda, \mu) \leq \tau(\lambda, \nu) + \tau(\nu, \mu)$ (triangle inequality)

The values τ assumes range from 0 to $\binom{n}{2}$ for permutations of length $n \in \mathbb{N}$, where $\binom{n}{2}$ refers to the binomial coefficient. The values must be smaller than $\binom{n}{2}$ since the number of pairs of discordance, which define the Kendall-τ distance must be smaller than the number of pairs of n candidates. The bound of $\binom{n}{2}$ is reached if and only if the two permutations are discordant on all pairs, which is equivalent to the rankings being each other's reverse.

1.4.3. Definition of the CENTER RANKING problem. The CENTER RANKING problem seeks to obtain the aggregation of a set of rankings $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$. It chooses the aggregation which has the least maximum distance from the input rankings. In other words, it tries to find a new permutation $\mu \in \mathrm{Sym}_{[n]}$ such that $\max_{\lambda \in V} \tau(\lambda, \mu)$ is minimal.

Problem 3 (CENTER RANKING)**.** We can describe CENTER RANKING as follows:

> **Given:** (V, r), where V is a set of votes—i.e. a set of permutations out of $\mathrm{Sym}_{[n]}$, where $[n] = \{1, \ldots, n\}, n \in \mathbb{N}$ is a set of candidates; and $r \in \mathbb{N}$ is a radius.
> **Question:** Is there a $\pi \in \mathrm{Sym}_{[n]}$ such that $\max_{\nu \in V} \tau(\pi, \nu) < r$?

We denote the set of $n \in \mathbb{N}$ candidates usually as $[n] \coloneqq \{1, \ldots, n\}$.

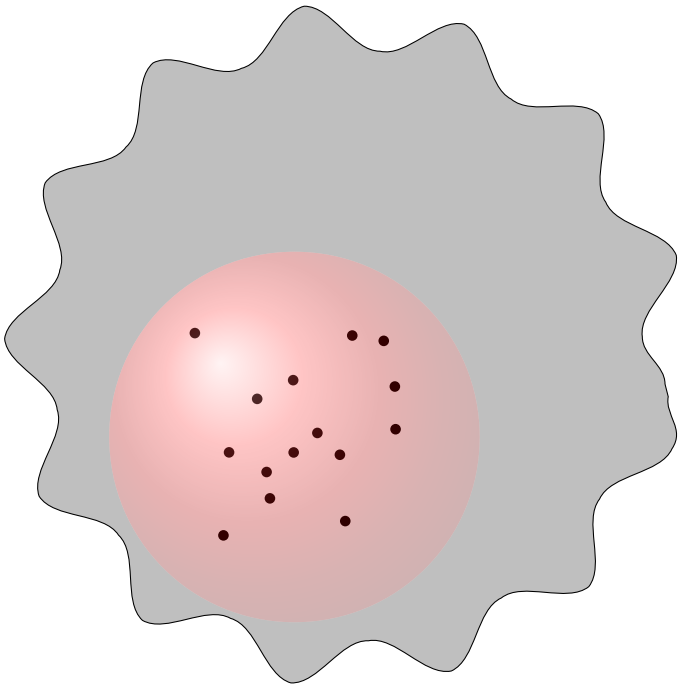

Figure 1.4.1: A visualization of a minimum KT-distance vote as introduced in Section 1.4.3, with its aggregation result. The points represent rankings, the middle of the red ball represents the aggregation results. The vote has an election result of quality r if the red disc of radius r can be translocated so that all points are covered, as is the case here.

We could imagine the votes to be points on a plane and the question would be if we can translocate the disc such that it completely covers all the points. The aggregation would be the ranking represented by the center of that disc. See Figure 1.4.1. The problem can be looked at as a decision problem, as defined above, but also as an optimization problem, changing the question to: "Find $\min_{\pi \in \mathrm{Sym}_{[n]}} \max_{\nu \in V} \tau(\pi, \nu) < r$?"

Problem 4. CENTER RANKING as an optimization problem:

Given: V, which is V is a set of votes—i.e. a set of permutations out of $\mathrm{Sym}_{[n]}$, where $[n] = \{1, \ldots, n\}, n \in \mathbb{N}$ is a set of candidates.
Question: Which is the minimal solution radius $r = \max_{\lambda \in V} \tau(\sigma, \lambda)$ for any permutation $\sigma \in \mathrm{Sym}_{[n]}$?

In the context of this definition, we refer to $\max_{\lambda \in V} \tau(\sigma, \lambda)$ as the *solution quality* of σ.

We could also ask for the actual solution of the problem: "For which $\pi \in \mathrm{Sym}_{[n]}$ is $\max_{\nu \in V} \tau(\pi, \nu) < r$ minimal?" For us, these three questions are essentially equivalent. If we can answer the decision problem, a binary search for the smallest radius solves the optimization problem[3]. If we can answer the optimization problem, we can also answer the decision problem simply by comparing parameter r to the optimum. Being able to solve the question for the optimal solution enables us to determine the maximum radius from the optimal solution to any input vote and with that number answer the optimization problem. It is a little bit more difficult to see how being able to solve the optimization problem enables us to answer the question for the optimal solution. This would indeed require some extra work. However, it is of no particular concern to us; all algorithms we present naturally emit the optimal solution. We therefore see no harm in treating the three formalizations of the CENTER RANKING problem as one problem and we deliberately interchange between understanding the CENTER RANKING problem one way or another.

1.4.4. Parameterized complexity. The CENTER RANKING problem was shown to be NP-hard [50][4]. It is an established assumption that NP-hardness inevitably leads to run times growing exponentially with the input size, as has been argued, among many others, by Garey and Johnson [23]. Hüffner[31] lists several ways out of the dilemma: "Heuristics drop the demand for useful run time guarantees or for useful quality guarantees, and are tuned to run fast with good results on typical instances. Approximation algorithms trade the demand for optimality for a provably efficient run time behavior, while still providing provable bounds on the solution quality."

For determining the final ranking of a sports competition, it is neither acceptable to expect exceptionally long computation times for corner cases, nor is it satisfactory to approximate the final ranking of the competitors. Parameterized complexity analysis as pioneered by Downey and Fellows [18, 21, 47] aims to help us identify the structural complexity of instances and find algorithms whose run time can be high-performing for instances of low structural complexity. We argue that most practically relevant instances are of low structural complexity.

We define a parameterized problem as follows [18].

Definition 5. A parameterized problem is a language $L \subset \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The second component is called the parameter of the problem.

Note that throughout this thesis, we will refer to an instance $(I, k) \subset L$ as instance I with parameter k.

A parameterized algorithms for an instance of input size n, is an algorithm with run time $f(k)p(n)$ where p is a polynomial, f is any function n is the size of the overall input; and k is the problem parameter. We can restrict the overall

[3]This has in fact been employed for the algorithm solving the neighbor permutation optimization problem, as seen in Appendix A.

[4]Note that in his paper, he refers to the CENTER RANKING problem as "ADJACENT SWAP CENTER PERMUTATION PROBLEM."

combinatorial explosion to the parameter k, such that for some values of k we can answer the instance quickly, even if the instance is large. Compare Figure 1.4.2 and Figure 1.4.3.

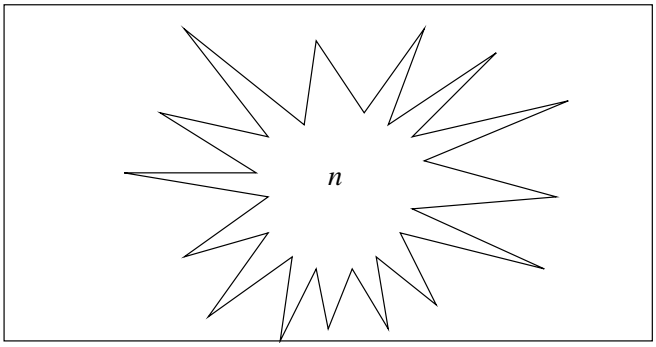

Figure 1.4.2: Classically considered, an NP-hard problem yields a combinatorial explosion in the input size. In this picture, n depicts the size of the overall input.

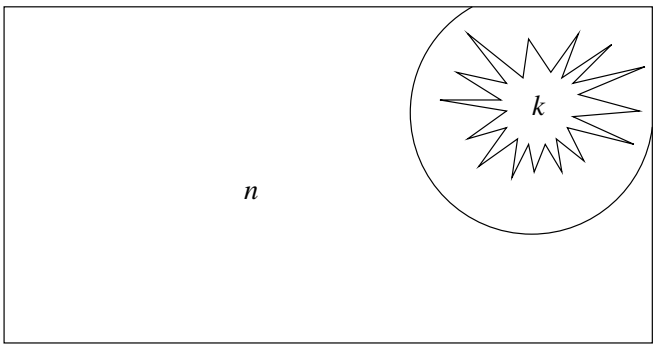

Figure 1.4.3: Parameterized complexity theory tries to confine the combinatorial explosion to the parameter k only. For small values of k, an algorithm with run time $O(f(k)p(n))$, where f is any function and p is a polynomial. This means that for some values of k, the problem is efficiently solvable for even large values of n.

Parameterized algorithms with time bounds of the form $O(f(k)p(n))$ can be considered to be solutions for variable k, with precise bounds on how much parameter k influences the run time. There can be several aspects introducing structural hardness, each reflected in their own parameter.

Having a range of parameterized algorithms for different parameters allows for automatic selection of an appropriate algorithm for a given instance. We can determine the different aspects of the instance that can constitute the structural hardness of the instance, choose an aspect of low structural hardness for the instance and then select an algorithm which runs high-performing for low values of the parameter which reflects the aspect.

1.4.5. Permutations. There are many different, yet compatible definitions of permutations, each favoring its own notation. A permutations can be understood as a sequence of symbols such that no symbol appears on two positions of the sequence.

The sequence is denoted as, for example, $\pi = 3657124$. We could understand a permutation π as a bijection from $[n]$ to $[n]$, Herein, $[n]$ denotes the set $\{1, 2, \ldots, n\}$. The bijection π is often denoted table-like as

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 3 & 6 & 5 & 7 & 1 & 2 & 4 \end{pmatrix}.$$

The function value of π is identified in the table as the table entry right under i.

The set of permutations of entries $[n]$ is denoted as $\mathrm{Sym}_{[n]}$. In this thesis, we assume permutations to be presented and stored in a computer in sequence notation, e.g. as $\pi = 3657124$. Here, π is the permutation which maps the number one to number three, number two to number six and so forth. Nonetheless, we understand permutations as functions, too. The composition of permutations is denoted using the circle operator "$\circ$". The permutation $\pi \circ \mu$ is the permutation for which $(\lambda \circ \mu)(i) = \lambda(\mu(i))\ \forall i \in [n]$ holds. Understanding a permutation as a function, we will denote the inverse of a permutation of a permutation μ as μ^{-1}. For example, for $\pi = 3657124$, the inverse would be $\pi^{-1} = 5617324$.

1.4.5.1. *Inversions.* Counting discordances is closely related to counting inversions in a permutation. Bona [10] gives the following definition of inversions.

Definition 6. Let $\lambda = p_1 p_2 \cdots p_n$ be a permutation. We say that (p_i, p_j) is an inversion of λ if $i < j$ but $p_i > p_j$.

An inversion discordance between λ and the permutation $12 \ldots n$, as defined in Definition 4.1.2.

Example 7. Permutation 3657124 has 11 inversions. They are $(3,1)$, $(3,2)$, $(6,5)$, $(6,1)$, $(6,2)$, $(6,4)$, $(5,1)$, $(5,2)$, $(5,4)$, $(7,1)$, $(7,2)$, and $(7,4)$.

We denote the set of all inversions in λ as $\mathrm{inv}(\lambda)$. For a permutation π, we write $1 <_\pi 2$, if 1 appears before 2 in π. We refer to the permutation $12 \ldots n$ as the *identity permutation*, and denote it as $\mathbf{1}$.

1.4.5.2. *Balls.* As it might be interesting to analyze the neighborhood of a certain permutation, let us define balls. A ball is meant to express the set of permutations proximate to a permutation.

Definition 8. Let (M, d) be a metric space, $m \in M$, $k \in \mathbb{N}$. Then,

$$\mathbb{B}_{d,k}(m) \coloneqq \{n \in M : d(m, n) \leq k\}.$$

The metric space we deal with most of the time is $(\mathrm{Sym}_{[n]}, \tau)$. If no other metric is given, we implicitly refer to τ.

Definition 9. Let $k \in \mathbb{N}, \mu \in \mathrm{Sym}_{[n]}$,

$$\mathbb{B}_k(\mu) \coloneqq \mathbb{B}_{\tau,k}(\mu) \coloneqq \{\nu \in M : \tau(\mu, \nu) \leq k\}.$$

1.4.6. Parallel algorithms.

> There are two critical forces shaping software development today. One is the popular adoption of Parallel Computing and the other is the trend toward Service Oriented Architecture.
> —L. Chen from Sun Microsystems [11]

We give brief glances at the parallelizability of some of our algorithms. Search tree algorithms can often be turned into parallel algorithms rather easily by referring all recursive calls to different threads. For several search tree algorithms we state the theoretical run time of these parallel algorithms. The computational model we have in mind when stating these run times is the EREW PRAM, the parallel random access machine where only one thread can read from or write into a specific cell of memory. This EREW model fits today's vintage computers and concurrent read or write is of no particular benefit to our algorithms. For a more complete introduction to the computational model and parallel algorithms, see [33].

1.4.7. Oriented graphs. We model some of our data as graphs to leverage the multitude of existing and well-analyzed algorithms operating on graphs.

A *simple graph* is defined as a pair (V, E), where V is understood as a set of vertices, and a set E of edges between two nodes each. We only consider simple directed graphs. A simple graph is *directed* if all of its edges are directed, i.e. if they have a start point and end point. Also, we only consider oriented graphs. A graph is *oriented* if and only if between any two vertices, there can be at most one directed edge. A *tournament* is an oriented, simple, directed graph that contains an edge between every pair of vertices. See [3] for an overview over directed graphs.

1.4.8. Fast computation of the Kendall-τ distance. The Kendall-τ distance between two permutations in $\mathrm{Sym}_{[n]}, n \in \mathbb{N}$, can easily be computed in time $O(n^2)$. A faster approach, computing the distance in time $O(n \log n)$, closely related to merge-sort, is given by Springsteel and Stojmenovic. [56, 40].

We suggest another algorithm whose run time is within $O(n \log n)$, with best case run time $\Omega(n)$. It is related to the parallel computation of prefix sums, described in [33].

Instead of counting discordances, we will count the inversions in one permutation. This suffices, since we show that for $\lambda, \mu \in \mathrm{Sym}_{[n]}$,

$$\#\operatorname{inv}(\lambda^{-1} \circ \mu) = \tau(\lambda, \mu). \tag{1.4.1}$$

To prove equation (1.4.1), we first present an important observation.

Lemma 10. *Let $\mu, \lambda, \pi \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. Then for the Kendall-$\tau$ distance,*

$$\tau(\lambda, \mu) = \tau(\pi \circ \lambda, \pi \circ \mu).$$

Proof. Recall that a discordance between permutations $\lambda, \mu \in \mathrm{Sym}_{[n]}$ is defined to be $\{a, b\}$ with $a, b \in [n], a \neq b$ where λ and μ list a and b in different order. We show that if $\{a, b\}$ is a discordance between λ and μ, then $\{\pi(a), \pi(b)\}$ must be a discordance between $\pi \circ \lambda$ and $\pi \circ \mu$. We show that there is a bijection between sets A and B, where A is the set of discordances between λ and μ, and B is the set of discordances between $\pi \circ \lambda$ and $\pi \circ \mu$.

Let $\{a, b\}$ be a discordance between λ and μ, we show that $\{a', b'\} \coloneqq \{\pi(a), \pi(b)\}$ is a discordance between $\pi \circ \lambda$ and $\pi \circ \mu$. Let us assume, without loss of generality, that $a <_\lambda b$, but $a >_\mu b$. From here, we find

$$\pi^{-1}(a') <_\lambda \pi^{-1}(b') \wedge \pi^{-1}(a') >_\mu \pi^{-1}(b') \tag{1.4.2}$$

$$\Leftrightarrow \lambda^{-1}(\pi^{-1}(a')) < \lambda^{-1}(\pi^{-1}(b')) \wedge \mu^{-1}(\pi^{-1}(a')) > \mu^{-1}(\pi^{-1}(b')) \tag{1.4.3}$$

$$\Leftrightarrow (\pi \circ \lambda)^{-1}(a') < (\pi \circ \lambda)^{-1}(b') \wedge (\pi \circ \mu)^{-1}(a') > (\pi \circ \mu)^{-1}(b') \tag{1.4.4}$$

$$\Leftrightarrow a' <_{(\pi \circ \lambda)} b \wedge a' >_{(\pi \circ \mu)} b'. \tag{1.4.5}$$

The number of discordances between two permutations equals the Kendall-τ distance. This completes the proof. □

We can deduce equation (1.4.1).

Corollary 11. *Let $\lambda, \mu \in \mathrm{Sym}_{[n]}, n, m \in \mathbb{N}$, then*

$$\#\operatorname{inv}(\lambda^{-1} \circ \mu) = \tau(\lambda, \mu).$$

Proof. It is easy to verify by the definition that

$$\operatorname{inv}(\nu) = \tau(\nu, \mathbf{1}). \tag{1.4.6}$$

Then, because of Lemma 10,

$$\tau(\lambda, \mu) = \tau(\mathbf{1}, \lambda^{-1} \circ \mu) = \operatorname{inv}(\lambda^{-1} \circ \mu).$$

□

The number of inversions can be counted as follows. For every entry, we count the number of entries it precedes, that are smaller than itself. The sum of these numbers is the number of inversions of the permutation (see [10, p. 50]).

Hence, it suffices to present an algorithm that quickly computes the number of inversions of a permutation. The pseudo-code of this operation is given in Algorithm 1.

Algorithm 1 Algorithm computing the number of inversions in μ. It is used in Section 1.4.8 to compute the Kendall-τ distance between two permutations, as $\tau(\lambda, \mu) = \mathrm{inv}(\lambda^{-1} \circ \mu)$. Here, v_i denotes the i^{th} entry in vector v.

```
Input: permutation μ ∈ Sym_[n].
Output: #inv(μ)
A1: Initialize vector v := (1,...,1) of length n.
A2: Initialize the number of inversions z := 0.
A3: for 1 ≤ i ≤ n do begin
    Set z := z + Σ_{j≤i} v_{μ(j)}. (*)
    Set v_{μ(i)} := 0. (**)
    end.
A4: return z.
```

Proposition 12. *Algorithm 1 correctly computes* $\#\mathrm{inv}(\mu)$.

PROOF. The invariant to prove this algorithm correct is that $\sum_{j \le i} v_{\mu(i)}$ is the number of elements smaller than $\mu(i)$ which $\mu(i)$ precedes. It holds right before the execution of line `(*)` in loop `A3`. The entry $v_{\mu(i)}$ can be interpreted as $v_i = 1$ iff the element i has been read before. □

To reach the claimed bound, we need to be able to compute instruction `(*)` in time $O(n \log n)$. We will merely outline the computation and how it leads to the time bound.

Theorem 13. *Algorithm 1computes* $\#\mathrm{inv}(\mu)$ *in time* $O(n \log n)$.

PROOF (OUTLINE). The algorithm looks n times into the prefix sums of vector v. We will show that instruction `(*)` can be computed in time $O(\log n)$ after some preparation. In every iteration, one entry of v changes. Since v does not change much, neither do its prefix sums. Similar to JáJá in [33], we can compute the prefix sums of v using a structure in which every entry can be read in $O(\log n)$ time and in which every entry of v can be updated in $O(\log n)$ time. We build a binary tree with some partial sums. See Figure 1.4.4 on page 13 for an example. There, the brown (light) nodes show the partial sums which are preserved throughout the re-iterations of line `(*)`. The blue (dark) nodes depict the steps required to compute line `(*)`. Here, $\sum_k^l := \sum_{i=k}^l v_i$. The example shows how $\sum_1^n$ is computed recursively using $\sum_1^j = \sum_1^{j/2} + \sum_{j/2+1}^j, j \le n$, but keeps intermediate results in the brown (light) nodes. Precisely, it keeps a vector s of length n, such that $s_i = \sum_a^i$, where a is $i - p + 1$, where p is the biggest power of 2 dividing i. In the diagram, the elements of s are displayed by the topmost brown nodes in each column. Using the vector, we can compute $\sum_1^k = \sum_i s_i |k|_i$, where $|k|$ is the binary presentation of k, and s_i is the i^{th} entry entry of vector s. In See Figure 1.4.4 on page 13, the brown (light) nodes represent vector s. Using vector s allows us to compute line `(*)` by only computing the blue (dark) nodes of the search tree for the desired sum, which can

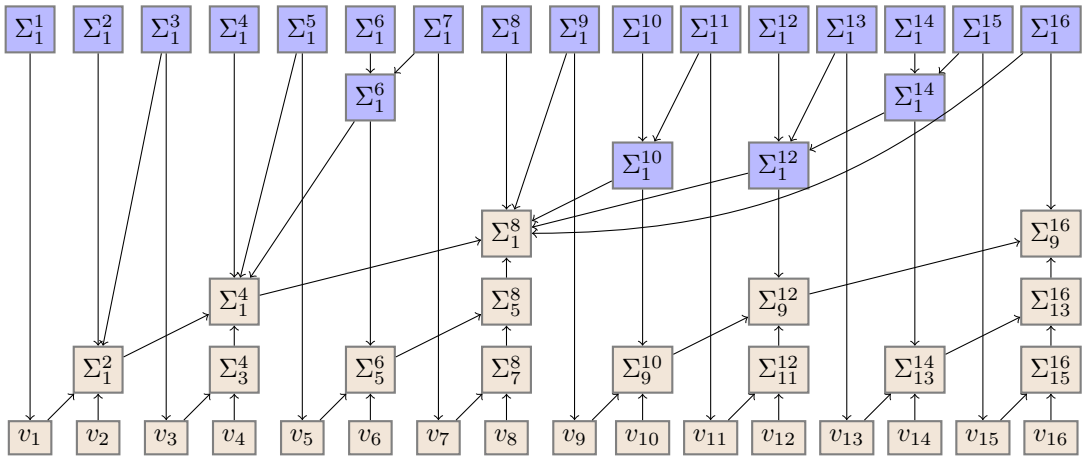

Figure 1.4.4: Computing the prefix sums of vector v. This is used in Section 1.4.8 to quickly execute Algorithm 1 on the preceding page. In this structure, any time an entry in v changes, the brown (light) entries it is connected with need be updated. This can be done in $O(\log n)$. To obtain an output value o_i at the top, all blue (dark) nodes it is connected to need be computed. There are at most $O(\log n)$ such nodes.

be done in time $O(\log n)$. In line `(**)`, s needs to be re-computed, which requires us to update the brown (light) nodes in the tree depicted in Figure 1.4.4 on page 13, which can done in time $O(\log n)$, too, since only one of the input nodes at the bottom changed due to instruction `(**)`. The initial build-up can before the looping can be done in time $O(n)$.

Summarizing, the loop can be executed in time $O(\log n)$, and it is executed $O(n)$ times, giving overall run time $O(n \log n)$.

□

Algorithm 1 can be further improved by deleting brown (light) nodes that have been read before and replacing them with a shortcut. The improvement leaves the worst-case bound at $O(n \log n)$, but the best case can be solved in linear time. It is currently unknown what the expected run time of the improved algorithm is for randomized inputs.

CHAPTER 2

Simple considerations on CENTER RANKING

2.1. The CENTER RANKING aggregation method

The optimum solution is defined to be the permutation with the least maximum Kendall-τ distance to the input rankings. Since the optimum solution needs to be as close as possible to all input rankings, we expect the aggregation to reflect all input rankings as closely as possible.

A function $e : \mathrm{Sym}_{[n]}^{m} \to \mathrm{Sym}_{[n]}$, $n, m \in \mathbb{N}$, is called an election method [60][1]. The CENTER RANKING method can be viewed as an election method, too. The theory of social choice proposes criteria to judge the quality of election methods. Truchon [59] lists the following four criteria as the probably most famous ones, all of which were proposed by Arrow [2].

Monotonicity: This criterion says that if all voters were to maintain or improve the relative ranking of candidate s with respect to candidate t, then the final ranking of candidate s with respect to candidate t should be at least as good as it was before the change.

Binary Independence: This criterion says that only the relative rankings of two candidates should matter in establishing the final relative ranking of these two candidates.

Weak Pareto: This criterion says that if all voters are unanimous on the relative rankings of two candidates, the final relative ranking of these two candidates should agree with the unanimous view of the voters.

Non-Dictatorship: This criterion says that no voter may be able to impose his or her ranking as the final ranking in all circumstances.

As an election method, the CENTER RANKING aggregation method satisfies all of these criteria except for Binary Independence. It can be easily verified by the definition that the CENTER RANKING aggregation method satisfies Monotonicity and Non-Dictatorship. We will prove the weak Pareto criterion to be satisfied and give an example that proves the Binary Independence criterion to not be satisfied.

To see that the CENTER RANKING aggregation method does not satisfy Binary Independence, let us prove that erasing a candidate from an election can indeed change the winner. Consider the example given in Table 2 on page 4. Erasing competitor C changes the aggregation to be A<B<D, while it is B<A<C<D when C is still considered.

It is not surprising that the CENTER RANKING election method does not satisfy all four criteria. Arrow [2] proved that an election method cannot satisfy all four criteria. However, note that the failure to satisfy Binary Independence might be desired. In the counter-example, we saw how adding candidate C to the competition can change the outcome. Adding a candidate also adds information. Candidate C seems to be a rather weak competitor. Yet, competitor A loses to him in one criterion. Common sense might tell us that the additional information should make us reconsider the ranking of competitor A, and favor competitor B over competitor A.

[1] In the the theory of social choice, the term social welfare function is used, too. Such a function determines the election result, given all ballots of an election.

Let us proceed to show that the weak Pareto criterion is satisfied. The weak Pareto criterion can be defined more formally as follows.

Definition 14 (Weak Pareto criterion [2, 60])**.** Let $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$, be a set of votes for (possibly containing duplicates), then an election method $e : \mathrm{Sym}_{[n]}^m \to \mathrm{Sym}_{[n]}$, complies with the weak Pareto criterion if and only if the following is true:

Given that all votes agree on the order of a given pair $\{a, b\}$, i. e.

$$\forall \lambda : a <_\lambda b \text{ or } \forall \lambda : a >_\lambda b, \tag{2.1.1}$$

the aggregation $e(V)$ maintains their relative ranking, i. e.

$$a <_{e(V)} b \text{ or } a >_{e(V)} b, \tag{2.1.2}$$

respectively.

We find that CENTER RANKING complies with the weak Pareto criterion.

Theorem 15. *The* CENTER RANKING *aggregation method satisfies the weak Pareto criterion.*

PROOF. Let us assume that an optimal aggregation ν would permit $b <_\nu a$, even though $\forall \lambda : a <_\lambda b$. We show that by swapping around a and b in ν, we can obtain a better result. Let ν' be the permutation we obtain from ν after swapping a and b. We show that ν' yields a better score than ν.

Let us first note the easy observation that the Kendall-τ distance can be written as the sum

$$\tau(\lambda, \nu) := \sum_{\{c,d\} \subseteq [n], c \neq d} d_{\lambda,\nu}(c, d),$$

where $d_{\lambda,\nu}(c, d)$ is defined as

$$d_{\lambda,\nu}(c, d) = \begin{cases} 0 & \text{if } \lambda \text{ and } \mu \text{ agree on the relative ranking of } c \text{ and } d. \\ 1 & \text{if } \lambda \text{ and } \mu \text{ disagree on the relative ranking of } c \text{ and } d. \end{cases}$$

For every $\lambda \in V$, we show that $\tau(\lambda, \nu') < \tau(\lambda, \nu)$. In the following inequality, the indices in the sums are $c, d \in [n]$.

$$\tau(\lambda, \nu) = \sum_{\{c,d\}} d_{\lambda,\nu}(c, d) \tag{2.1.3}$$

$$= \frac{1}{2} \sum_c \sum_d d_{\lambda,\nu}(c, d) \tag{2.1.4}$$

$$= \frac{1}{2} \sum_{c \notin \{a,b\}} \sum_{d \notin \{a,b\}} d_{\lambda,\nu}(c, d) + \sum_d d_{\lambda,\nu}(a, d) + \sum_d d_{\lambda,\nu}(b, d) \tag{2.1.5}$$

$$< \frac{1}{2} \sum_{c \notin \{a,b\}} \sum_{d \notin \{a,b\}} d_{\lambda,\nu'}(c, d) + \sum_d d_{\lambda,\nu'}(a, d) + \sum_d d_{\lambda,\nu'}(b, d) \tag{2.1.6}$$

$$= \frac{1}{2} \sum_c \sum_d d_{\lambda,\nu'}(c, d) \tag{2.1.7}$$

$$= \tau(\lambda, \nu'). \tag{2.1.8}$$

To see inequality (2.1.6), note that the first term stays the same, while the other two terms must shrink or stay the same. The inequality is strict, because

$$d_{\lambda,\nu}(a, b) < d_{\lambda,\nu'}(a, b). \tag{2.1.9}$$

We showed that $\tau(\lambda, \nu') < \tau(\lambda, \nu)$ for every $\lambda \in V$.

We have seen that an optimum solution needs to list two candidates whose relative order is agreed on by all rankings, in that relative order. □

The CENTER RANKING aggregation method lacks any sort of majority principle, such as the extended Condorcet criterion proposed in [59]. Again, we argue that this may be wanted. If we regard the rankings we need to aggregate as preferences according to different criteria, we may not want to compensate one criterion with several other.

2.2. Relation to the CLOSEST STRING and KEMENY SCORE problems

As a computational problem, the CENTER RANKING problem is fairly similar to both the KEMENY SCORE problem and the CLOSEST STRING problem. Some of the algorithms designed for either of the problems can be adapted to solve the CENTER RANKING problem.

2.2.1. Relation to CLOSEST STRING. Just like the CENTER RANKING problem, the CLOSEST STRING problem with alphabet $\Sigma = \{0, 1\}$ is concerned with finding the center of a given set of input items. The difference is that the CLOSEST STRING problem defines the input to be strings of equal length, while the CENTER RANKING problem accepts votes of the same number of candidates as its input. Also, while the center in the CLOSEST STRING problem is understood as the center with respect to the Hamming distance, the CENTER RANKING problem seeks the center with respect to the Kendall-τ distance. The CLOSEST STRING problem is defined as follows.

Problem 16. The CLOSEST STRING problem over alphabet $\Sigma = \{0, 1\}$.

> **Given:** (S, k), where S is a set of strings of the same length, over the alphabet $\{0, 1\}^D, D \in \mathbb{N}$, i. e. $S \subset \{0, 1\}^D$, and a solution parameter $k \in \mathbb{N}$.
> **Question:** Is there a string $s \in \{0, 1\}^D$, such that $S \subseteq \mathbb{B}_k(s)$?

However, we show that sets of strings over the alphabet $\{0, 1\}^D$ can be transformed into permutations whose pairwise Kendall-τ distances equal original Hamming distances. We deduce that every algorithm solving the CENTER RANKING problem can also solve CLOSEST STRING instances after a transformation.

Popov [50][2] proved the CENTER RANKING problem to be NP-hard by reducing the CLOSEST STRING problem to it. We give a simplification of his proof, as he obtains it as a corollary of showing the similar SWAP CENTER PERMUTATION problem to be NP-complete, which uses another metric between the votes, known as the Caley distance.

Theorem 17 (POPOV)**.** *The* CENTER RANKING *problem is NP-complete.*

PROOF. The CENTER RANKING problem clearly is in NP.

We show the Karp-reduction [1] CLOSEST STRING $\leq_P$ CENTER RANKING. A Karp-reduction transforms the input of a CLOSEST STRING instance and outputs an input to CENTER RANKING, such that the latter is solvable if and only if the prior is solvable. This proves NP-completeness, since the CLOSEST STRING problem was proved to be NP-complete even for alphabet size[3] 2 by Lanctot et al. [42].

A run of a permutation is a consecutive sub-sequence of the permutation. By reversing a run of length 2, we mean swapping the both entries around. The reduction is performed by the function f, described in Algorithm 2.

[2]Note that in his paper, he refers to the CENTER RANKING problem as "ADJACENT SWAP CENTER PERMUTATION PROBLEM."

[3]Frances and Litman. [22] show NP-completeness for the general case.

Algorithm 2 Algorithm describing function f that reduces from CLOSEST STRING to CENTER RANKING.

```
F1: Take as input a string s.
F2: Set p to be the identity permutation of length 2|s|.
F3: Partition p into runs of length 2.
For each 1 ≤ i ≤ |s|:
    F4: Reverse the i^th run in p
    if and only if s_i = 1.
F5: Output p.
```

For example, $f(0101) = 12435687$. The algorithm would start with identity permutation 12345678, split into four runs: 12, 34, 56, and 78. The second and fourth run would be reversed, and the final result would be 12435687.

Apparently, Algorithm 2 runs in polynomial time.

We show that for all $s, t \in \{0,1\}^D$

$$d_H(s,t) = \tau(f(s), f(t)). \tag{2.2.1}$$

For a position $i \in [D]$ that s and t differ, the i^{th} run in $f(s)$ is in reverse order of $f(t)$, proving

$$d_H(s,t) \leq \tau(f(s), f(t)).$$

Let $\{b, c\}$ be a discordance between $f(s)$ and $f(t)$. Due to the definition of f, b and c are adjacent in both $f(s)$ and $f(t)$, and appear in the same run that ranges from position $i-1$ to position i for some $i \in [D]$. Further, since $\{b, c\}$ is a discordance, one run is the reverse of the other, and hence s and t have different symbols on position i, proving

$$d_H(s,t) \geq \tau(f(s), f(t)).$$

We proved equation (2.2.1). We show that (S, k) is CLOSEST STRING yes-instance if and only if $(f(S), k)$ is a CENTER RANKING yes-instance.

Let s be a solution of the CLOSEST STRING instance (S, k). Equation (2.2.1) tells us that $f(s)$ is a solution of the CENTER RANKING instance $(f(S), k)$.

We show that if $(f(S), k)$ is a CENTER RANKING yes-instance, so is the CLOSEST STRING instance(S, k). Let π be a solution solution of the CENTER RANKING instance $(f(S), k)$. We show that π can be assumed to have inversions only in adjacent candidates. If there were a solution π with a discordance between any other two elements, transposing them would yield a better solution. Hence, we have that f is bijective on solutions of $(f(S), k)$. By equation 2.2.1, we conclude that $f^{-1}(\pi)$ must be a solution of CLOSEST STRING instance (S, k).

We reduced CLOSEST STRING to CENTER RANKING, completing the proof. □

Any two permutations of the reduced instance will have discordances only between adjacent candidates. The proof implies the following corollary.

Corollary 18. *Let (S, k) be a* CLOSEST STRING *instance and let f be the transformation described by Algorithm 2, then $(f(S), k)$ is a* CENTER RANKING *yes-instance if and only if (S, k) is a* CLOSEST STRING *yes-instance, and equation* (2.2.1) *holds.*

Equation (2.2.1) is especially interesting. Let us note it as a separate corollary.

Corollary 19. *Let f be the transformation described by Algorithm 2, then f maps a set of strings to a set of permutations retaining their respective distances.*

The reduction implies that impossibility results for the CLOSEST STRING problem still hold for the CENTER RANKING problem. Transformation f leaves the structure of CLOSEST STRING instances largely untouched.

2.2.2. Relation to KEMENY SCORE. The computational problem of obtaining the final ranking of the Kemeny-Young method is known as the KEMENY SCORE problem [32]. Bartholdi III et al. [32] showed the KEMENY SCORE problem to be NP-hard. It has since been studied rather intensively[7, 20, 19, 30, 5, 32]. We attempt to adapt many of the techniques used for obtaining parameterized algorithms for the KEMENY SCORE problem.

The KEMENY SCORE problem accepts the same input structure as the CENTER RANKING problem, i.e. a set of votes, and it, too, aggregates rankings. However, the definition of the quality of an optimal solution differs. It is defined as follows.

Problem 20. The KEMENY SCORE problem.

Given: (S, k), where S is a set of permutations out of $\mathrm{Sym}_{[n]}$ for some $n \in \mathbb{N}$ and a score $k \in N^+$.
Question: Is there a permutation $\pi \in \mathrm{Sym}_{[n]}$, such that $\sum_{\sigma \in S} \tau(\sigma, \pi) < k$?

The solution quality in the KEMENY SCORE of a solution σ is equivalent to a double sum, as opposed to the maximum of a sum in the CENTER RANKING problem is a significant difference. The double sum can be separated and re-ordered at will. The KEMENY SCORE of a solution σ is defined as

$$\sum_{\sigma \in S} \tau(\sigma, \pi) = \sum_{i} \sum_{\{a,b\} \subset [n]} d_{\sigma,\lambda_i}(a, b),$$

where

$$d_{\sigma,\lambda_i}(a, b) = \begin{cases} 0 & \text{if } \lambda_i \text{ and } \sigma \text{ list } a \text{ and } b \text{ in the same order} \\ 1 & \text{otherwise} \end{cases}.$$

Betzler et al. [7] provide a plethora of parameterized algorithms for the KEMENY SCORE problem. Many of them make use of the separability of the KEMENY SCORE problem. These algorithms provide a solution for a slice of the instance and then use this sub-solution to speed up significantly the computation of the next, possibly overlapping slice of the instance.

Specifically, it is possible to re-arrange the double sum in order to completely consider the effect of a certain preference pair. For example, for two candidates $b, c \in [n]$, the contribution of the relative ranking of b and c to the overall Kemeny score is

$$\sum_{i} d_{\sigma,\lambda_i}(b, c).$$

We can isolate the effect of a relative ranking of two candidates in the KEMENY SCORE problem, which we will refer to as the "separability" of the KEMENY SCORE problem. In the CENTER RANKING problem, we cannot isolate the influence of a pair as easily, because a certain relative ranking of two candidates in the solution may or may not change the CENTER RANKING solution quality at all. If some input vote λ is a lot closer to the solution than another input vote μ, then the precise structure of λ matters less than that of μ. We could swap random adjacent pairs of λ without changing the aggregation outcome. That may not be true for μ.

Betzler et al. [7] proposed a bounded-depth search tree algorithm that solves the KEMENY SCORE problem. A simple version of the algorithm could be brought forward to solve the CENTER RANKING problem. The algorithm proposed by Betzler et al. uses the fact the Kemeny-Young election method maintain the order of a certain pair of candidates, if the majority of the votes asserts that order. The CENTER RANKING election method does not necessarily preserve the preference of the majority.

Dynamic programming algorithms generally seem to require some degree of separability of the computational problem, which the CENTER RANKING problem can hardly permit. It may thus appear unsurprising that we have not found a single dynamic programming algorithm for the CENTER RANKING problem.

If all votes agree on the relative ranking of two candidates, the Kemeny-Young election method preserves that relative ranking in the election outcome. This property is called weak Pareto criterion. The fact that the Kemeny-Young method satisfies the weak Pareto criterion permits us to simplify certain instances by removing candidates whose position is entirely determined by the weak Pareto criterion. This sort of simplifying of an instance is called "data reduction." We discuss data reduction rules for the CENTER RANKING problem in Section 2.3.

2.3. Data reduction rules and kernelizations

Even though a problem might be NP-hard to solve, it is not necessarily impossible to gather information about the solution of a problem instance. For example, if we were to select a minimal set of vertices whose deletion would delete all edges (a problem known as VERTEX COVER), it is easy to show that if some vertex is adjacent to no other vertex, then the deletion of the vertex does not change the solution.

Downey and Fellows [18] explain that the purpose of the method of data reduction is to "reduce a problem instance I to an 'equivalent' instance I', where the size of I' is bounded by some function of [a parameter]. The instance I' is then exhaustively analyzed, and a solution for I' can be lifted to a solution for I, in the case where a solution exists." For a more complete introduction, together with many successful examples of the method, see [25, 64].

A *kernelization* with respect to parameter k is a transformation of an instance (I, k) of a parameterized problem to an instance (I', k'), such that for $n \in \mathbb{N}$ being the size of I and for any function $g : \mathbb{N} \to \mathbb{N}$,

- the problem instance I is a yes-instance if and only if I' is a yes-instance
- the size of I' does not exceed $g(n)$
- the transformation can be done in polynomial time in n
- $k' \leq k$

The result of the exhaustive application of the kernelization is called a *problem kernel*. We will refer to the rules that reduce instance I to instance I' in a kernelization as *data reduction rules*. Our ultimate goal is to prove bounds on the problem kernels of our data reduction rules.

2.3.1. Identical votes. In Problem 3 on page 7, the problem definition assumed the input of votes to be a set rather than a list of votes. The frequency of a certain vote in the input is not of significance when it comes to answering the question asked by the CENTER RANKING problem[4]. This follows from the definition of the CENTER RANKING problem in Problem 3 on page 7 as a minimum of a maximum. We formalize the disposability of multiple votes as a data reduction rule. In the following rule, V is a list rather than a set.

Data reduction rule 21. *Let (V, r) be an instance of the* CENTER RANKING *problem. Remove all duplicates from V.*

[4]This is a difference from the Kemeny-Young method, where the multitude of votes can have a significant impact on the impact of the vote. In [67] the KEMENY SCORE problem was formalized to have the input votes given as a function $n_\sigma : \mathrm{Sym}_{[n]} \to \mathbb{N}$ which mapped a ranking to its number of occurrences in the given instance.

As a consequence, we can limit the number of votes in V by knowing the number of candidates.

Proposition 22. *After application of Data Reduction Rule 21, an instance of the* CENTER RANKING *problem contains at most $n!$ votes where n is the number of candidates in the instance.*

We can therefore bound the number of votes in the problem kernel, if we assume the number of candidates to be bounded.

2.3.2. Weak Pareto. The CENTER RANKING problem satisfies the weak Pareto criterion which states that if all votes agree on the order of a certain pair of candidates, the order is maintained in the final ranking, see Lemma 15 on page 16.

The data reduction rule uses the term "dirty pair", which is defined as an unordered pair of candidates whose relative ranking is not agreed on by all candidates[7, Definition 1].

Definition 23. An unordered pair $\{a, b\}$ of candidates $a, b \in [n], n \in \mathbb{N}$, is a dirty pair of an instance $V \subset \mathrm{Sym}_{[n]}, \#V = m$ if and only if there are $\lambda, \mu \in V$ such that $a <_\mu b \wedge b <_\lambda a$.

We can deduce the following data reduction rule from the weak Pareto criterion.

Data reduction rule 24. *Let (V, r) be an instance of the* CENTER RANKING *problem, where $V \subset \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. If there is a candidate $a \in [n]$ which appears in no dirty pair of instance V, then strike candidate a out of all input votes.*

Note that candidate a must have appeared in the same position in each input vote $\lambda \in V$, since all candidates to its left agree in every vote that a must be to their right and by analog, all candidates to the right of λ agree in every vote that λ must be to their left. After solving the simplified instance and obtaining solution σ', we can obtain the solution of the original instance by inserting the struck candidate a into the position it was struck from.

We prove an easy limit on the kernel.

Proposition 25. *Let $V \in \mathrm{Sym}^m_{[n]}, n, m \in \mathbb{N}$ be an instance of the* CENTER RANKING *problem. After exhaustive application of Data Reduction Rule 24, the kernel of V has at least $n/2$ dirty pairs.*

PROOF. After exhaustive application of Data Reduction Rule 24, an instance has at least $n/2$ dirty pairs left, since otherwise there would be a candidate without a dirty pair, which the rule would have deleted. □

As it is trivial to obtain a parametrization on the number of candidates by merely trying all orderings, we might be interested in relations between parameters that implicitly bound the number of candidates.

We show that bounding the maximum pairwise distance of an instance and bounding the number of votes implies a bound on the number of candidates after exhaustive application of Data Reduction Rule 24 and Data Reduction Rule 21.

Proposition 26. *Let $V \subset \mathrm{Sym}_{[n]}, n \in \mathbb{N}, \#V = m$ be an instance of the* CENTER RANKING *problem. Let $d_{\max} \coloneqq \max_{i,j}\{\tau(\lambda_i, \lambda_j)\}$ denote the maximum pairwise distance of the input votes. After exhaustive application of Data Reduction Rule 24 and Data Reduction Rule 21 the number of candidates in the problem kernel is within $O(m^2 d_{\max})$.*

PROOF. Since $\tau(\mu, \lambda) < d_{\max}\ \forall \mu, \lambda \in V$ the number of discordances between any two votes is bounded by $d_{\max}$. Let $P \coloneqq \{\{a, b\} : \exists \mu, \lambda : a <_\mu b \wedge a >_\lambda b\}$. Set P contains the *dirty pairs* of the instance.

We show that the number of dirty pairs does not exceed $\frac{1}{2}m^2 d_{\max}$.

$$\#P = \#\left\{\{a,b\} : \exists \mu, \lambda : a <_\mu b \wedge a >_\lambda b\right\} \tag{2.3.1}$$

$$\leq \sum_{\mu,\lambda} \#\left\{\{a,b\} : a <_\mu b \wedge a >_\lambda b\right\} \tag{2.3.2}$$

$$= \sum_{\mu,\lambda} \tau(\mu,\lambda) \tag{2.3.3}$$

$$\leq \sum_{\mu,\lambda} d_{\max} \tag{2.3.4}$$

$$= m^2 d_{\max} \tag{2.3.5}$$

After exhaustive application of Data Reduction Rules 24 and 21, each candidate must be in at least one dirty pair. There are at most $\frac{1}{2}m^2 d_{\max}$ dirty pairs. Each dirty pair contains two candidates. After exhaustive application of Data Reduction Rules 24 and 21 an instance can contain no more than $m^2 d_{\max}$ candidates. □

The above lemma is related to Popov's rather lengthy proof [50, Theorem 4], which shows that SWAP CENTER PERMUTATION problem is fixed-parameter tractable, in our terminology, with respect to the parameter "radius and number of votes combined." The SWAP CENTER PERMUTATION problem is equivalent to the CENTER RANKING problem, except that it measures distances using the Caley distance[5]. While Popov uses the same technique, he does not put it into context with data reduction.

We deduce yet another data reduction rule from the observation, that in yes-instances, the maximum pairwise distance may not exceed $2r$.

Proposition 27. *If $V \in \mathrm{Sym}_{[n]}^m, nm \in \mathbb{N}$ and $r \in \mathbb{N}$ constitute an instance of the* CENTER RANKING *problem, then if $\exists \lambda, \mu \in V : \tau(\lambda,\mu) > 2r$ then there cannot be $\pi \in \mathrm{Sym}_{[n]} : \max_{\lambda \in V} \tau(\pi,\lambda) \leq r$.*

PROOF. If a consensus string π were to exist, we would find, by the very definition of the problem

$$\tau(\lambda,\mu) \leq \tau(\lambda,\pi) + \tau(\pi,\mu) \leq 2r,$$

thus contradicting our assumption. □

Data reduction rule 28. *Let (V,r) be an instance of the* CENTER RANKING *problem, where $V \subset \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. Seek for a pair $\lambda_1, \lambda_2 \in V$ with $\tau(\lambda_1,\lambda_2) > 2r$. If one exists, answer "no".*

Next, compute the maximum distance between any two $\lambda_1, \lambda_2 \in V$. If it is less than r, answer yes.

2.3.3. Impossibility of a data reduction rule based on neighborhood examinations. It is intriguing to consider the possibility that, as the final outcome is figuratively trying to "please" the most extreme voters, there could be votes which do not have any impact on the final outcome, because other votes are more extreme than they are. However, the idea seems to be misguided.

Specifically, we would imagine that if a vote is surrounded, then it can be disposed. This turns out to be false.

Proposition 29. *Let R be the rule that removes a vote μ from an instance of the* CENTER RANKING *problem, if all possible votes of votes of Kendall-τ distance 1 to μ are part of the instance, too. Then, R is* not *a correct data reduction rule*

[5]We briefly discuss the Caley distance at the end of Section 4.2.

PROOF. Consider the instance $V = \{\lambda : \lambda \in \mathrm{Sym}_{[n]}\}$ containing all possible votes for some $n \in \mathbb{N}$.

Permutation $\mathbf{1} = 12 \dots n$ would be removed due to rule R. However, in the original instance, the optimum solution quality would be $\binom{n}{2}$, while the reduced instant would allow solution quality $\binom{n}{2} - 1$ for solution $n \dots 21$. □

CHAPTER 3

Multivariate analysis

Fixed-parameter analysis tries to confine the combinatorial explosion in the run time of a problem to a certain parameter or a combination of several parameters. Since the run time is exponential in these parameters, we may want to choose parameters which give us reason to believe that they are small in instances worth solving. We examine interesting parameters and try to give parameterized algorithms or hardness results wherever we can.

3.1. Overview of the parameters under consideration

We present an overview of the parameters that appear to be worth considering, and refer to a more detailed analysis in subsequent sections and chapters. The results for the various parameters are summarized in Table 3.

Table 3: Fixed-parameter tractability results on the CENTER RANKING problem presented in this thesis listed per parameter, where #candidates means "number of candidates." If a parameterized algorithm was found, its run time is presented. If the problem was found to be NP-hard for fixed values of the parameter, the respective entry reads "NP-hard." It is unknown whether the CENTER RANKING problem is fixed-parameter tractable with respect to parameter "number of votes."

parameter	result	section
number of candidates, n	$O(n! \cdot n \log n)$	Section 3.2
number of votes, m	unknown	Section 3.8
radius, r	$O(2^{4r}mn^2 + mn^2 \log n)$	Chapter 4
maximum pairwise distance, $d_{\max}$	$O(2^{8d_{\max}}mn^2 + mn^2 \log n)$	Section 3.4
position range, p	NP-hard	Section 3.5
average pairwise distance, d_a	NP-hard	Section 3.6
number of dirty pairs, p_r	$O(2^{p_r} \cdot mn \log n + n^2 m)$	Section 3.7
#candidates and radius combined, (n, r)	$O(mn \log n \cdot \min\{rn^r, n!\})$	Chapter 5

Let $V \in \mathrm{Sym}_{[n]}^m, m, n \in \mathbb{N}$ be an instance of the CENTER RANKING problem. We consider the following parameters.

Number of candidates: denoted as n. The number of candidates n is defined as the number of entries in each input permutation. The CENTER RANKING problem can be solved in time $O(n! \cdot n \log n)$. See Section 3.2.

Number of votes: denoted as m. The number of votes m is defined as the number of input votes in V. When aggregating according to criteria, the number of votes equals the number of criteria under consideration. In many scenarios there are only two rankings under consideration. For $m = 2$, the CENTER RANKING problem can be solved in time $O(n^2)$, see Section 3.8. For general m, it is unknown whether the CENTER RANKING problem is fixed-parameter tractable with respect to the parameter "number of votes." See Section 3.8.

Radius: denoted as r. The radius of an instance is defined to be the best solution quality possible for an instance. If σ is an optimum solution of an instance of the CENTER RANKING problem, the radius is the maximum Kendall-τ distance from σ to any input vote. The radius of an instance will also be referred to as the *optimum solution quality*. The parameter may be very small for correlating input rankings. The CENTER RANKING problem is solvable in time $O(2^{4r}mn^2 + mn^2 \log n)$. See Section 3.3 and Chapter 4.

Maximum pairwise distance: denoted as $d_{\max}$. The maximum pairwise distance is defined to be the maximum pairwise Kendall-τ distance between any two input permutations. It is equivalent to the parameter "radius" in the sense that $r \leq d_{\max} \leq 2r$. See Section 3.4.

Position range: denoted as p. The position range is the maximum distance of positions any candidate assumes in the input votes. For example, if a candidate appears on position 1 in one vote and on position 15 in another, then the position range is at least 15. The parameter can be very small for large instances. The CENTER RANKING problem is NP-hard even for $p = 1$. See Section 3.5.

Average pairwise distance: denoted as d_a. The average Kendall-τ distance between all pairs of input permutations. This parameter can be very small even for large instances if many of the votes correlate. The CENTER RANKING problem is NP-hard for $d_a = 1$. See Section 3.6.

Number of dirty pairs: denoted as p_r. The number of pairs of candidates are of different relative ranking for any two input votes. The number of dirty pairs can be very small, even for large instances. The CENTER RANKING problem can be solved in time $O(2^{p_r} \cdot mn \log n + n^2 m)$. See Section 3.7.

Number of candidates and radius combined: Since the CENTER RANKING problem is fixed-parameter tractable with respect to both r and n, it is naturally fixed-parameter tractable to their combination, too. We present two algorithms, which present the currently best-known algorithms to solve instances chosen uniformly at random. The first solves the CENTER RANKING problem in time $O\left(mn \log n \cdot \min\left\{r(2r)^n, rn^{2r}, n!\right\}\right)$, the second in time $O(mn \log n \cdot \min\{rn^r, n!\})$. See Chapter 5.

The relationships between the parameters are summarized in Table 4.

Table 4: Relationships between the parameters.

parameters	relationship
n, m	$m \leq n!$ after data reduction, see Proposition 22
n, m, r	$n \leq m^2 r$ after data reduction, see Lemma 26.
n, r	$r \leq \binom{n}{2}$,$\mathbb{E}(r) \geq \binom{n}{2}/4$, see Section 3.3.
$r, d_{\max}$	$r \leq d_{\max} \leq 2r$, see Section 3.4.
$d_a, d_{\max}, m$	$d_{\max} \geq d_a m$, see Section 3.6.
$d_a, d_{\max}$	$d_a \leq d_{max}$, see Section 3.6.
$d_{\max}, m, n$	$d_a \leq m\binom{n}{2}$, see Section 3.6.
$d_{\max}, p$	$d_{\max} \geq p$, see Section 3.5.
$d_{\max}, n$	$d_{\max} \leq n(n-1)$, see Section 3.4.
p_r, n	$\frac{n}{2} \leq p_r \leq \binom{n}{2}$ after data reduction, see Section 3.7.

3.2. Parameter "number of candidates"

In practical implementations solving NP-hard problems, it is not without precedence that the trivial algorithm that tries all possible solutions is preferred to more refined algorithms. Besides being easily implementable in most cases and providing predictable run times, algorithms that try all possible solutions may may be of competitive performance, since they usually involve less complicated data structures and computations, thus keeping the hidden constants small.

Perhaps the most natural algorithm to solve a CENTER RANKING instance of n candidates is to merely try all $n!$ permutations and see which one is the best solution. The run time of this simple algorithm would be within $O(mn \log n \cdot n!)$, where n is the number of candidates and m is the number of votes. Enumerating all $n!$ permutations can be done in $O((n+1)!)$ time using standard techniques such as backtracking, see e.g. [55]. The solution quality of a permutation can be checked in $O(mn \log n)$ time. This is achieved by computing m Kendall-τ distances, which can be done in $O(n \log n)$ time as we have discussed in Section 1.4.8.

Algorithm 3 The trivial algorithm solving CENTER RANKING enumerating all permutations of length $n \in \mathbb{N}$ and choosing the permutation with the best solution quality.

```
Input: An instance (λ_1, λ_m) of the CENTER RANKING problem
    with λ_i ∈ Sym_[n] ∀i ≤ m.
T1: [Initialize] Set g := ∞.
T2: [Loop control]
    Using a permutation enumeration algorithm,
    e.g. [55, p. 711], execute T3 for each π ∈
    Sym_[n]. Then go to T4.
        T3: [Loop] If max_i{τ(π, λ_i)} < g,

            then σ := π, g := max_i{τ(π, λ_i)}
T4: [Result] Return σ.
```

The above algorithm already yields a parametrization with respect to the parameter "number of candidates." Despite seeming rather crude, the algorithm is tested in practical experiments in Section 6.2 and in fact outperforms more refined algorithms for some instances.

3.3. Parameter "radius"

The *radius* of an instance of the CENTER RANKING $(\lambda_1, \ldots, \lambda_m)$ problem is defined to be the quality of an optimum solution. The radius is the smallest possible $r \in \mathbb{N}$ such that there is a permutation σ with $\max_{i \in [m]} \tau(\lambda_i, \sigma) \leq r$. Parameterized algorithms for this parameter are deduced and discussed in Chapter 4. In this section, we limit the size of the parameter and put it into context with the other parameters.

For any instance, we find that $r \leq \binom{n}{2}$, since any two permutations' distance needs to be smaller than or equal to $\binom{n}{2}$. The bound cannot be improved, since an instance whose set of input permutations includes all permutations of length n must inevitably contain the reverse of whatever permutation is chosen as a solution. The Kendall-τ distance between a permutation and its reverse is $\binom{n}{2}$. Hence, the instance's optimum solution would be of quality $\binom{n}{2}$.

In situations where the criteria we want to aggregate do not correlate, the radius must be expected to be rather large. We prove the following theorem.

Theorem 30. *Let $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$ be an instance of the* CENTER RANKING *problem whose input permutations are chosen uniformly at random, the expected minimal solution quality is greater than or equal to $\binom{n}{2}/4$, where n is the number of candidates.*

PROOF. Let us assume that the number of votes equals $m = 2$. Otherwise, the expected value needs clearly be higher. We show that the expected value is exactly $\binom{n}{2}/4$ for $m = 2$.

Let us show that $\binom{n}{2}/2$ is the expected Kendall-τ distance between two permutations chosen uniformly at random out of $\mathrm{Sym}_{[n]}$. While this was stated by Kendall in [39, p. 63] and is in fact quite central to the book, it was not proved in [39].

Let $\pi, \pi' \in \mathrm{Sym}_{[n]}$ be chosen uniformly at random. Due to Lemma 10 on page 11, for $\mu \coloneqq \pi^{-1} \circ \pi'$, we have $\mathbb{E}(\tau(\pi, \pi')) = \mathbb{E}(\tau(\mathbf{1}, \mu))$, where $\mathbb{E}$ denotes the expected value. Apparently μ follows the uniform distribution on $\mathrm{Sym}_{[n]}$. We seek the average for $\tau(\mathbf{1}, \mu)$ for $\mu \in \mathrm{Sym}_{[n]}$. Let μ^R denote the permutation obtained by reversing the sequence notation of μ. To compute the average distance, we pair every permutation with its reverse. The average value of the distance between $\mathbf{1}$ and μ is

$$\begin{aligned}
\frac{1}{n!} \sum_{\mu \in \mathrm{Sym}_{[n]}} \tau(\mathbf{1}, \mu) &= \frac{1}{2(n!)} \sum_{\mu \in \mathrm{Sym}_{[n]}} \tau(\mathbf{1}, \mu) + \tau(\mathbf{1}, \mu^R) \\
&= \frac{1}{2(n!)} \sum_{\mu \in \mathrm{Sym}_{[n]}} \binom{n}{2} \\
&= \binom{n}{2}/2.
\end{aligned}$$

For $m = 2$, proper analysis of Algorithm 4 on page 36, which solves the CENTER RANKING problem in case $m = 2$, shows that the minimal solution quality needs to be half the distance between the two input votes, which completes the proof. □

The expected value of the optimal solution quality needs to be smaller than or equal to $\binom{n}{2}$. We can say that the estimated solution quality must be within $\Theta\left(\binom{n}{2}\right) = \Theta\left(n^2\right)$. We can expect r to be significantly greater than n in instances chosen uniformly at random out of $\mathrm{Sym}_{[n]}$, which is confirmed by our experiments in Section 6.2.

3.4. Parameter "maximum pairwise distance"

Let us define the position range and maximum pairwise distance for a CENTER RANKING instance (V, r). The definition given here is equivalent to that given by Betzler et al. [7].

Definition 31. Let $(V, r), V \in \mathrm{Sym}_{[n]}^m, n, m, r \in \mathbb{N}$ be an instance of the CENTER RANKING problem. Then,

$$d_{\max} \coloneqq \max_{\lambda,\mu \in V} \tau(\lambda, \mu).$$

For the similar KEMENY SCORE problem, Betzler et al. [7] presented an algorithm parameterized by the maximum distance parameter using dynamic programming. In this section, we establish that CENTER RANKING, too, is fixed-parameter tractable with respect to the parameter "maximum pairwise distance." We examine the maximum pairwise distance parameter and then examine the dynamic programming approach presented in [7].

In relation to the radius parameter, we find a close relation between the maximum pairwise distance and the optimum solution quality of an instance. The optimum solution quality of an instance is the answer to the question asked in Problem 4 on page 7:

> **Given:** $V \in \mathrm{Sym}_{[n]}^m, m, n \in \mathbb{N}$, a set of permutations, interpreted as votes on the order of the candidates $[n]$.
> **Question:** Which is the minimal solution radius $r = \max_{\lambda \in V} \tau(\sigma, \lambda)$ for any permutation $\sigma \in \mathrm{Sym}_{[n]}$?

We show that the parameter "maximum pairwise distance" is equivalent to the solution radius parameter r in the same sense as two metrics are equivalent. More specifically, we find the following lemma.

Lemma 32. *Let $V \in \mathrm{Sym}_{[n]}^m, m, n \in \mathbb{N}$ be an instance of the* CENTER RANKING *optimization problem. Let $d_{\max}$ be the maximum pairwise distance in V and let r be the optimum solution radius, then after exhaustive application of Data Reduction Rule 28,*

$$\frac{d_{\max}}{2} \leq r \leq d_{\max}.$$

PROOF. To see that $r \leq d_{\max}$, note that trivially $\tau(\lambda_1, \lambda_j) \leq d_{\max} \forall j$. This means that vote λ_1, if tested for its solution quality, would have solution radius at most $d_{\max}$. Since the optimal solution is at least as good as any permutation's solution quality, we have $r \leq d_{\max}$.

In Lemma 27 on page 22 we already proved $d_{\max} \leq 2r$, completing the proof. □

In Section 1.4.3 we figuratively described the CENTER RANKING problem to be the problem of translocating a disc over the space of permutations such that all permutations are under the disc. Lemma 32 says that for non-trivial instances, the radius of the disc needs to be within $[\frac{d_{\max}}{2}, d_{\max}]$. See Figure 3.4.1 on page 30.

Lemma 32 implicitly puts the $d_{\max}$ parameter into context with the parameter "number of candidates."

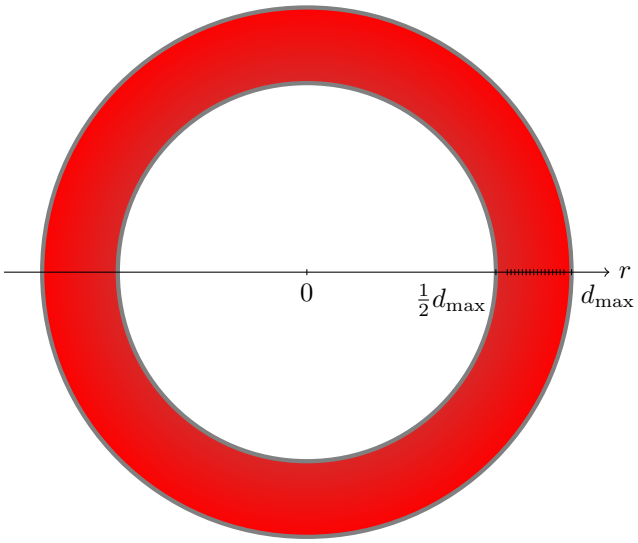

Figure 3.4.1: Discs with interesting radii. When the CENTER RANKING problem is viewed as the problem of translocating discs over the space of permutations until all permutations are covered, then Lemma 32 implies that non-trivial discs must have radii within $[\frac{d_{\max}}{2}, d_{\max}]$.

Corollary 33. *Let* $V \in \mathrm{Sym}^m_{[n]}, n, m \in \mathbb{N}$ *be an an instance of the* CENTER RANKING *problem, then*

$$d_{\max} \leq 2r \leq 2\binom{n}{2} = n(n-1).$$

A consequence of Corollary 33 combined with Corollary 51 on page 45 is the following.

Corollary 34. The CENTER RANKING *problem is fixed-parameter tractable with respect to the parameter* $d_{\max}$.

Lemma 32 has another possibly interesting consequence. By choosing just any input permutation as a solution, its solution quality can be at most twice as bad as that of the optimal solution. We can use Lemma 32 to give a constant factor approximation algorithm that solves the optimization problem associated with CENTER RANKING. An approximation algorithm is an algorithm for an optimization problem that finds a solution that is possibly not optimal, but approximates the optimum. The ratio between the optimal solution and the approximated solution is called "approximation ratio." A constant factor approximation algorithm is an algorithm with a constant approximation ratio[63].

We outline the approximation algorithm and its properties in the following theorem.

Theorem 35. *For the optimization problem associated with* CENTER RANKING*, the algorithm that chooses any input permutation and outputs it is a factor-2 approximation algorithm.*

PROOF. This is a consequence of Lemma 32. □

The algorithm described in Theorem 35 runs in time only $O(n)$, which is merely the time needed to output the first input permutation. Apparently, this algorithm is of debatable practical value. However, it does imply that looking for approximation

algorithms might be fruitful for the CENTER RANKING problem. This would be further suggested by the success of finding good approximation algorithms for the CLOSEST STRING problem by Ma and Sun [44].

3.5. Parameter "position range"

The maximum range of candidate positions is the size of the largest interval $[b, c]$, such that a candidate appears on position b in some vote and position c in another. For large instances, the maximum position range might be very small. In this section, we discuss a dynamic programming algorithm proposed for the related KEMENY SCORE problem that yields a parameterized algorithm with respect to parameter "position range," and we prove the CENTER RANKING problem to be NP-hard with respect to the parameter.

Betzler et al. [7] provide a parameterized algorithm for the KEMENY SCORE problem that looks at instances block-wise. All input rankings are truncated to a given and fixed number of consecutive positions. The algorithm solves the truncated input, which we will call a frame. When a frame of a certain size has already been analyzed, it speeds up the analysis of the frame that starts one position further to the right, by looking at the old frame's solution. For a given frame, the qualities of all possible solutions are computed and written into a table. Then, for the computation of the next frame, lookups in the table speed up the next frame computation. The algorithm is considered to be a dynamic programming algorithm.

Even though it is just as possible to split the input to the CENTER RANKING problem into frames, knowing all the qualities of all possible solutions for one frame does not seem to help computing the next frame. In Section 2.2.2, we have discussed that the KEMENY SCORE problem allows for a certain degree of separability, which appears to be a general necessity for dynamic programming approaches.

We prove NP-hardness even for position range 1. Let us first give a precise definition of the position range, as used by Betzler et al. [7]. Recall that the position of element i in a permutation λ can be denoted as $\lambda^{-1}(i)$.

Definition 36. Let $(V, r), V \in \mathrm{Sym}_{[n]}^m, n, m, r \in \mathbb{N}$ be an instance to the CENTER RANKING problem. Then we call

$$p \coloneqq \max_{\lambda,\mu \in V, i \in [n]} |\lambda^{-1}(i) - \mu^{-1}(i)|$$

the *position range* of V.

Here, $|\cdot|$ denotes the absolute value.

To put the parameter "position range" into context, let us show that $p \leq d_{\max}$. Betzler et al. prove the following proposition as a part of a larger proof [7, Corollary 1]. Let us isolate the needed piece and repeat it here.

Proposition 37 (Betzler et al.)**.** *Let $(V, r), V \in \mathrm{Sym}_{[n]}^m, m, n, r \in \mathbb{N}$, be an instance of the* CENTER RANKING *problem. Then, the position range $p \in \mathbb{N}$ of instance V is smaller than or equal to the maximum pairwise distance $d_{\max} \in \mathbb{N}$, i.e.*

$$p \leq d_{\max}.$$

PROOF. We prove the theorem by contradiction. For the sake of contradiction, assume that there are two permutations $\alpha, \beta \in V$ and a candidate $k \in \mathbb{N}$ such that the position of k differs by more than d, more precisely, $\alpha^{-1}(k) = p$ and $\beta^{-1}(k) \geq p + d_{\max} + 1$. We can examine α and β:

$$\alpha = \alpha_1 \dots \alpha_{p-1} k \alpha_{p+1} \dots \alpha_{p+d} \alpha_{p+d+1} \dots \alpha_n$$
$$\beta = \beta_1 \dots \beta_{p-1} \beta_p \beta_{p+1} \dots \beta_{p+d} k \beta_{p+d+2} \dots \beta_n.$$

Note that in α, the number of candidates to the right of k is at least $d_{\max}+1$ larger than than the number of candidates to the right of k in β. Each of these candidates contributes to at least one discordance between α and β, because α lists the candidates after k, but β lists them before k. At least $d_{\max}+1$ inversions between α and β imply that $\tau(\alpha,\beta) \geq d+1$. This means that if the position range were greater than $d_{\max}$, then $d_{\max}$ would not be the maximum distance between any two input permutations, thus proving $p \leq d_{\max}$. □

It is possible to reduce the CLOSEST STRING problem with alphabet $\{0,1\}$ to CENTER RANKING such that the position range is solely 1. And in fact we have done so in the proof of Theorem 17 on page 17. We deduce the following observation.

Proposition 38. *The* CENTER RANKING *problem is NP-hard even for instances of position range 1.*

Therefore, there cannot be a parameterized algorithm with respect to the "position range," unless P=NP.

3.6. Parameter "average distance"

The average distance of an instance V to the CENTER RANKING problem is defined as the mean pairwise distance between the input votes.

In this section, we put the parameter into context with other parameters, discuss a dynamic programming algorithm for the KEMENY SCORE problem, and prove that the CENTER RANKING problem is NP-hard for every average distance greater than any $\epsilon > 0$.

Definition 39. Let V be an instance of the CENTER RANKING problem. Then we call

$$d_a(V) \coloneqq \frac{1}{\binom{\#V}{2}} \sum_{\{\lambda,\mu\},\mu,\lambda\in V} \tau(\lambda,\mu)$$

the average distance of V.

Due to the definition, of the average distance, we can easily conclude the following.

Proposition 40. *Let $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$, an an instance of the* CENTER RANKING *problem, where d_a is the average distance and $d_{\max}$ the maximum pairwise distance. Then,*

$$d_a \leq d_{\max}.$$

For the KEMENY SCORE problem, Betzler et al. [5, 6] presented a dynamic programming algorithm that was parameterized with respect to the parameter "average distance." Due to the poor separability of the CENTER RANKING problem, see Section 2.2.2, this author was unable to bring forward their algorithm to the CENTER RANKING problem.

We prove that CENTER RANKING is NP-hard even for average distance 1, if duplicate input votes are allowed.

Theorem 41. CENTER RANKING *is NP-hard for pairwise average distance $d_a = \epsilon\ \forall \epsilon > 0$, if input permutations are allowed to be identical.*

The general idea is that, given any CENTER RANKING instance, copying an input permutation reduces the average score, without making the problem any easier.

PROOF. We reduce any CENTER RANKING problem instance to another CENTER RANKING instance with pairwise average distance ϵ. We show that the transformed instance is solvable if and only if the original instance is solvable. Let

$(\lambda_1, \ldots, \lambda_m) = V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$, be an instance of the CENTER RANKING decision problem. We transform V to another instance

$$V_q' \coloneqq (\lambda_1, \ldots, \lambda_m, \mu_1, \ldots \mu_q), \tag{3.6.1}$$

where $\mu_i = \lambda_1 \forall i$.

We show that there is a q such that the average pairwise distance in V' is smaller than any given ϵ. Let

$$\sigma := \sum_{\{\lambda,\nu\},\lambda,\nu \in V} \tau(\lambda, \nu) \tag{3.6.2}$$

be the total of all pairwise distances in V. Note that

$$\sigma \leq \binom{m}{2}\binom{n}{2}. \tag{3.6.3}$$

We choose q as

$$q \coloneqq 2(\sigma/\epsilon + 2m - m^2). \tag{3.6.4}$$

Then we find that the average pairwise distance of V' is

$$d_a(V_q') = \frac{1}{\binom{m+q}{2}} \sum_{\{\pi,\rho\} \subset V_q'} \tau(\pi, \rho) \tag{3.6.5}$$

$$= \frac{1}{\binom{m+q}{2}} \Big(\sum_{\{\pi,\rho\} \subset V} \tau(\pi, \rho) + \sum_{1 \leq i \leq q, \lambda \in V} \tau(\mu_i, \lambda) + \sum_{1 \leq i,j \leq q} \tau(\mu_i, \mu_j) \Big) \tag{3.6.6}$$

$$= \frac{1}{\binom{m+q}{2}} (\sigma + q \sum_{\pi \in V} \tau(\pi, \lambda_1) + 0 \tag{3.6.7}$$

$$\leq \frac{1}{\binom{m+q}{2}} (\sigma + q\sigma) \tag{3.6.8}$$

$$= \frac{1}{\binom{m+q}{2}} (\sigma(q+1)) \tag{3.6.9}$$

$$= \frac{\sigma}{-1 + m + q/2 + (2 - 3m + m^2)/(2(1+q))} \tag{3.6.10}$$

$$\leq \frac{\sigma}{-1 + m + q/2 + (2 - 3m + m^2)} \tag{3.6.11}$$

$$= \frac{\sigma}{1 - 2m + m^2 + q/2} \tag{3.6.12}$$

$$< \epsilon, \tag{3.6.13}$$

where inequality (3.6.13) follows from the definition of q in equation (3.6.4). We proved that $d_a(V_q') < \epsilon$, where V_q' is the transformed instance of average distance smaller than ϵ.

To show that the reduction is correct, it needs to be shown that V has solution radius r if and only if V_q' has solution radius r. Data Reduction Rule 21 on page 20 says that erasing duplicates does not alter the solution radius. Using the same argument, adding one does not alter the solution radius either. Hence, V has solution radius r if and only if V_q' has solution radius r.

Finally, we argue that the reduction can be done in polynomial time in the instance size. This is clear, since we only need to create q copies of λ_1, and due to equation (3.6.4) and inequality (3.6.3), we have (note that ϵ is a constant)

$$q = 2(\sigma/\epsilon + 2m - m^2) = O(n^2 m^2),$$

showing that the reduction can be carried out in polynomial time in the size of V, completing the proof. □

3.7. Parameter "number of dirty pairs"

A dirty pair is defined to be an unordered pair $\{a, b\}$ of candidates $a, b \in [n]$ of an instance $V \in \mathrm{Sym}^m_{[n]}, n, m \in \mathbb{N}$, if and only if there are $\lambda, \mu \in V$ such that $a <_\mu b \wedge a >_\lambda b$, see Definition 23 on page 21. We will refer to the cardinality of the set of dirty pairs as the *number of dirty pairs.*

In this section, we put the parameter into context with the parameter "number of candidates." We outline a search tree algorithm that branches on all dirty pairs, and thus solves the CENTER RANKING problem solvable in $O(2^{p_r} \cdot mn \log n + n^2 m)$, where p_r denotes the number of dirty pairs, and n refers to the number of candidates, and m refers to the number of votes. We can count the number of dirty pairs in the instance upfront in $O(n^2 m)$ time. This allows estimating the run time of the algorithm and therefore aids choosing the appropriate algorithm for an instance.

Since the CENTER RANKING aggregation method satisfies the weak Pareto criterion (see Section 2.1), we know that if all votes share a preference on two candidates, every aggregation must adhere to that preference. Hence, rather than trying all possible permutations of the candidates, it suffices to only consider those permutations that adhere to the preferences shared by all voters.

After exhaustive application of Data Reduction Rule 24 on page 21, an instance has at least $n/2$ dirty pairs left due to Lemma 25. The lower bound can be reached. Consider the following CENTER RANKING instance.

$$\begin{aligned} V &= \{\lambda_1, \lambda_2\} \\ \lambda_1 &= 12 \dots n \\ \lambda_2 &= 2143 \dots n(n-1) \end{aligned}$$

This instance has exactly $n/2$ dirty pairs, while Data Reduction Rule 24 on page 21 would not erase any candidate. As an upper bound, the number of dirty pairs is clearly smaller than $\binom{n}{2}$, the number of all possible pairs. Let us summarize these easy observations into the following proposition.

Proposition 42. *After exhaustive application of Data Reduction Rule 24, the number of dirty pairs p_r in an instance of* CENTER RANKING *with n candidates is within $[n/2, \binom{n}{2}]$.*

In terms of parameterized complexity analysis, the CENTER RANKING problem is fixed-parameter tractable with respect to the parameter "dirty pairs in the instance." We outline a search tree algorithm on all dirty pairs that was proposed for the KEMENY SCORE problem [7], but can be used to solve the CENTER RANKING problem, too. The algorithm computes a set S of pairs of candidates on whose relative orders all votes agree. The algorithm starts by choosing a pair that is not present in S, chooses a preference for it and adds the pair to S. It repeats the process until either all preferences have been decided, or S contains conflicting pairs. That is, S contains three pairs that express a cyclic preference. The algorithm algorithm tries all possibilities to add a new pair in the branch step.

The algorithm solves the CENTER RANKING problem in $O(2^{p_r} \cdot mn \log n + n^2 m)$ time. The search tree would clearly have $O(2^p)$ leaves, and the time spent per leaf would be the time required to check if the selection of preferences is a solution. The checking can be done in $O(mn \log n)$ time. Obtaining all dirty pairs initially is possible in time $O(n^2 m)$. Since a more elaborate description can be found in [7], let us merely state the result.

Theorem 43. *The* CENTER RANKING *problem is fixed-parameter tractable with respect to the parameter "number of dirty pairs". An instance (V, r) with p_r dirty*

pairs can be solved by a search tree in time $O(2^{p_r} \cdot mn \log n + n^2 m)$*, where* p_r *denotes the number of dirty pairs,* n *the number of candidates and* m *the number of votes.*

Betzler et al. [7] give a refinement of the above search tree which this author has not been able to transfer. The crux is to branch on dirty triples rather than on dirty pairs. As defined in [7], "A dirty triple consists of three candidates such that at least two pairs of them are dirty pairs." A KEMENY SCORE instance without dirty triples is trivial, because the Kemeny-Young method satisfies the extended Condorcet criterion, which says that if the majority of the voters prefer some candidate over another, that should give the final ranking. This allows a more sophisticated branching. However, the CENTER RANKING problem stays NP-hard even if restricted to instances without dirty triples. This can be deduced from the reduction from CENTER RANKING to CLOSEST STRING in Section 20 on page 19 which never generates dirty triples.

The search tree algorithm described above can also be parallelized by referring all recursive calls to different threads. Then, all nodes on one level of the search tree are computed in parallel, reducing the time needed to the depth of the search tree times the time per node. In terms of parallel algorithms [33], the algorithm would run in time $T = O(pn^2m)$ and with work (number of operations across all threads) $W = O(2^p \cdot mn \log n + n^2 m)$, where p refers to the number of processors.

3.8. Parameter "number of votes"

The number of votes of an instance $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$, refers to the number of input rankings m. For some rather interesting applications of the CENTER RANKING problem, such as obtaining the final ranking of a sports competition, it is well justifiable that the number of input permutations is small while the number of candidates is rather large, as in the search engine applications proposed in Section 1.1. Therefore, several approaches to proving and disproving the fixed-parameter tractability of the CENTER RANKING problem have been tried and are outlined in this section. Unfortunately, up to the current point, fixed-parameter tractability of the CENTER RANKING problem with respect to the parameter "number of votes" could be neither proved nor disproved. On the positive side, we show that for $m = 2$ the CENTER RANKING problem is solvable in $O(n^2)$ time.

A ranking can be turned into any other through repeated swapping of adjacent entries. Let $V = \{\lambda_1, \lambda_2\}, \lambda_1, \lambda_2 \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$ be an instance of the CENTER RANKING problem. We show that V can be solved in time $O(n^2)$. If permutation λ_2 can be obtained from permutation λ_1 by swapping around k adjacent entries, then after $\lfloor \frac{k}{2} \rfloor$ of these swaps we arrive at an optimum solution to the problem. See Algorithm 4.

Algorithm 4 Solving CENTER RANKING for $m = 2$ input votes.

```
Input: Two votes λ1, λ2 ∈ Sym_[n], n ∈ ℕ.
Output: σ such that max_{i∈{1,2}} τ(σ, λi)
    is minimal.
S1: Collect all unordered pairs
    {a, b} ⊂ [n] into set A such that
    the relative ranking of a and b
    differs in λ1 and λ2.
S2: Repeat ⌊τ(λ1, λ2)/2⌋ times:
    S2.1: Remove a pair {p, q} from A such
          that p and q are neighbors in λ2.
    S2.2: Replace λ2 by λ2 with p and q
          swapped.
S3: Return λ2.
```

Note that in instruction `S2.1`, the pair $\{p, q\}$ can always be found in set A. This is because A is initialized to be the set of inversions between λ_1 and λ_2. Each execution of the loop maintains the property that A is the set of inversions between λ_1 and λ_2. Now, if $\lambda_1 \neq \lambda_2$ then at least one of their inversion must be between neighboring entries, see. [41, p. 108].

Theorem 44. *Let $(\lambda_1, \lambda_2) \in \mathrm{Sym}_{[n]}^2, n \in \mathbb{N}$ be an instance of the* CENTER RANKING *problem. Algorithm 4 correctly solves instance (λ_1, λ_2) of the* CENTER RANKING *problem.*

PROOF. We show that the solution returned by Algorithm 4 is optimal. Algorithm 4 returns a permutation with distance at most $\lceil \tau(\lambda_1, \lambda_2)/2 \rceil$ to both λ_1 and λ_2. If there were a better solution σ, then

$$\tau(\lambda_1, \lambda_2) \leq \tau(\lambda_1, \sigma) + \tau(\sigma, \lambda_2) < 2 \lceil \tau(\lambda_1, \lambda_2)/2 \rceil \leq \tau(\lambda_1, \lambda_2),$$

which is false. Therefore, the solution returned by Algorithm 4 is optimal. □

Let us prove the run time bound.

Theorem 45. *Algorithm 4 solves the* CENTER RANKING *problem for $V \in \mathrm{Sym}_{[n]}^m$, $n \in \mathbb{N}, m = 2$, in time $O(n^2)$.*

PROOF. This is clear since since the number of bubble sort steps performed is half the number of discordances between λ_1 and λ_2, which is defined to be $\tau(\lambda_1, \lambda_2) \leq \binom{n}{2}$. □

3.8.1. Integer program. The CLOSEST STRING problem can be parametrized by the number of input strings, the analog of the number of votes of the CENTER RANKING problem. A parameterized algorithm is provided by Gramm et al. [24] that makes use of integer programming. As the CENTER RANKING problem and the CLOSEST STRING problem are similar, see Section 2.2.1, we examine the parameterized algorithm proposed by by Gramm et al. [24], and then discuss how it could be brought forward to solving the CENTER RANKING problem.

In the CLOSEST STRING problem, if all strings are written in a table below one another, it is insignificant in which order the columns appear. We prove this fact in Appendix B. In the parameterized algorithm in [24], identical columns are compressed to only one. It is counted how often a each column appears in the input in a table listing all possible columns. The possible columns are called *column types*. The input is thus compressed into one counter for each "column type" and then solved using an integer program. An integer program is a linear optimization

problem whose variables are restricted to integer values[66]. Solving an integer program requires exponential time in the number of variables only, due to a result by Lenstra [26]. Let us examine how the strings are encoded.

Example 46. An instance of the CLOSEST STRING problem.

$$s_1 = 100$$
$$s_2 = 010$$
$$s_3 = 110$$
$$s_4 = 001$$

In the example, the first column is of column type 1010, the second column is of column type 0110, and the third column would be of column type 0001.

There can be at most 2^m column types. Let us identify column types with subsets $v \subset V$, such that a column type would equal v if all $\mathbf{t} \in v, \mathbf{t} \in \{0,1\}^D$ would be set to 1 in the columns of the specified column type. In this example, we would identify column type 1010 with the set $\{s_1, s_3\}$, we would identify column type 0110 with the set $\{s_2, s_3\}$, and we would identify column type 0001 with the set $\{s_4\}$. Note that the identification is biunique.

The integer linear program solving CLOSEST STRING is presented in Algorithm 5.

Algorithm 5 Integer program that provides a parameterized algorithm that solves the CLOSEST STRING problem.

- Variable $\#\sigma_v$ of the integer program will encode how many of the columns of type v in the desired solution σ are set to 1.
- Number $\#_v \in \mathbb{N}$ is defined to be defined to be the number of occurrences of column type $v \subset V$ in the input.
- Number $\mathbf{t}_v \in \{0,1\}$ for $\mathbf{t} \in V$,$v \subset V$ is set to 1 if input string $\mathbf{t}$ is set to 1 in columns of type v, otherwise 0.
- The integer linear program seeks to minimize

$$\max_{\lambda \in V} \sum_{v \subset V} (\mathbf{t}_v(\#_v - \#\sigma_v) + (1 - \mathbf{t}_v)(\#\sigma_v)$$

such that $0 \leq \#\sigma_v \leq \#_v$.

Note that the only variables of the integer program are variables $\#\sigma_v$ that encode the result of the integer program. Numbers $\#_v$ and $\mathbf{t}_v$ are *not* variables of the integer program but encode the input.

The elegance lies in the fact that the only variables needed are $\#\sigma_v$. There are $2^{\#V}$ of them. Together with Kannan's improvement of Lenstra's theorem [36], Gramm et al. proved that CLOSEST STRING with an alphabet of size 2 can be solved in time $O(m^{9 \cdot 2^{m-1}} n)$. While this yields large numbers for $m > 4$, it gives hope that a combinatorial solution to the CLOSEST STRING problem might exist and that its run time is smaller.

We can try to model CENTER RANKING similarly. Let $V \in \mathrm{Sym}_{[n]}^m$ be an instance of the CENTER RANKING problem. We exploit the idea of storing the solution in variables $\#\sigma_v$ of which there are only $2^{\#V}$.We denote our permutations as strings over the alphabet $\{0,1\}$. We denote a permutation by listing all relative rankings of the candidates available. Let $\pi \in \mathrm{Sym}_{[n]}$ be a permutation, we can encode it as a string of ones and zeroes as follows.

Definition 47 (String representation of a permutation)**.** Let $\{a_i, b_i\}, a_i \neq b_i$ be the i^{th} entry of the list of all unordered pairs on $[n]$ in some fixed order. Let $\pi \in \text{Sym}_{[n]}, n \in \mathbb{N}$, be a permutation. Then for

$$s_i(\pi) \coloneqq \begin{cases} 1 & \text{if } a_i <_\pi b_i \\ 0 & \text{otherwise} \end{cases},$$

we call $\mathbf{s}(\pi) \coloneqq s_1 \dots s_{\binom{n}{2}}$ the *string encoding* of permutation π.

The string notations' Hamming distance of two permutations equals the permutations' Kendall-τ distance. We will look more closely at this fact in Section 5.3.1.

- Number $\#_v \in \mathbb{N}$ is defined to be defined to be the number of occurrences of column type $v \subset V$ in the input encoded as strings.
- Number $\lambda_v \in \{0, 1\}$ for $\lambda \in V$,$v \subset V$ is set to 1 if the string encoding of input vote λ is set to 1 in columns of type v, 0 otherwise.
- Variables $\#\sigma_v$ define the string encoding of the solution permutation. Variable $\#\sigma_v$ equals the number of positions in columns of column type v that are set to 1 in the string encoding of the solution permutation.

Then, $\max_{\lambda \in V} \sum_{v \subset V} (\lambda_v(\#_v - \#\sigma_v) + (1 - \lambda_v)(\#\sigma_v)$ is the maximum distance from σ to any input permutation. However, we need to restrict $\#\sigma_v$ such that there always is a permutation that variables $\#\sigma_v$ define. In other words, not every setting of the variables $\#\sigma_v$ represents a permutation. The number of restrictions needed to restrict $\#\sigma_v$ to only encode permutations appears to be very big and depend on the exact values $\#\sigma_v$ assumes. What is more, each restriction requires its own variable. If we cannot give an upper bound on the number of variables that depends only on m, we cannot assume to obtain fixed-parameter tractability. This author has not been able to identify an integer program that solves the CENTER RANKING problem which has a number of variables that is bounded by $f(m)$, where f is any function.

Even though we could not prove fixed-parameter tractability with respect to the parameter "number of votes," integer programming nonetheless is a promising technique for obtaining high-performing algorithms that solve the CENTER RANKING problem. We present an integer program that solves the CENTER RANKING problem in Section 6.1.

3.8.2. Feedback edge set reduction to Kemeny score. The KEMENY SCORE problem is NP-hard even for as few as 4 votes. The KEMENY SCORE and CENTER RANKING problems share the same input and they both try to aggregate rankings—yet, their aggregations usually differ. In this sub-section, we outline a proof proposed by Dwork et al. [20], which shows that KEMENY SCORE is NP-hard for only 4 votes. We will see that it relies on properties of the KEMENY SCORE problem that are not present in the CENTER RANKING problem.

To show the NP-hardness of an instance of only 4 votes, Dwork et al. [20] reduce FEEDBACK EDGE SET instances to KEMENY SCORE instances of only 4 votes. The FEEDBACK EDGE SET problem was shown to be NP-complete by Karp [37]. Karp defined FEEDBACK EDGE SET as the following problem.

Problem. The FEEDBACK EDGE SET problem.

> **Given:** (G, k), where G is a directed graph and k is a natural number.
> **Question:** Is there a set of k edges whose deletion breaks all cycles?

Dwork et al. could encode the entire directed graph into just one vote. Each edge is encoded as a candidate in the vote. For two edges e, d that are adjacent to

a vertex v in the input graph as follows. The relative ranking of the candidates a, b representing edges e, d was defined as follows. If edge d enters vertex v and edge e leaves vertex v, then a is sorted before b. If d leaves v and e enters v, then b is sorted before a. If e and d would both enter or leave v, then an arbitrary order is chosen. Thus, some relative orders in the vote were important, while other relative orderings of candidates were arbitrarily chosen. By adding new votes which revers all arbitrarily chosen relative orders, it is possible to "cancel out" the effect of the arbitrary choices in the final ranking.

It seems hard to try "cancelling out" one vote with another in the CENTER RANKING problem, since the contributions of discordances do not add up to determine the quality of a solution. In fact, we have advertised the CENTER RANKING aggregation method in Chapter 1 as an aggregation method that specifically disallows compensating one criterion by another.

3.8.3. Summary. Solving CENTER RANKING for $m = 2$ can be done quickly and in polynomial time. For $m > 2$ the question whether CENTER RANKING is fixed-parameter tractable with respect to the parameter m remains open. The complexity of the CENTER RANKING problem with respect to the parameter "number of votes" appears to be between CLOSEST STRING and KEMENY SCORE. While the CLOSEST STRING problem is fixed-parameter tractable with respect to the parameter "number of votes," the KEMENY SCORE problem is NP-hard even for only 4 votes. There lies a large number of classes of different fixed-parameter intractability between these two extremes, proposed by Downey and Fellows. [18]. From the analysis carried out so far, this author does not dare to conjecture into which of the categories the CENTER RANKING problem falls.

CHAPTER 4

Parameterized algorithms for parameter "radius"

The radius parameter, denoted r, is defined to be the solution quality of an optimum instance in the CENTER RANKING problem. That is, if σ is an optimal solution, than the maximum distance from σ to any input vote is defined to be the "radius" of the instance.

The solution quality of an optimization problem is a natural choice of a parameter to be analyzed from a parameterized complexity viewpoint. In the CENTER RANKING problem, the radius parameter is of particular interest if we interpret the CENTER RANKING problem as a method to aggregate rankings according to criteria and if we have reason to assume that the criteria correlate. In such cases, the maximum pairwise distance between two votes is small; the radius, therefore, is even smaller.

We provide two search tree algorithms that allow us to solve large instances of the CENTER RANKING problem in short time on a vintage computer for radius $r \leq 6$, even though the problem is NP-complete. We present two search tree algorithms, the first one solving the CENTER RANKING problem in $O(r^r \cdot mn \log n)$ time and the second one solving it in time $O(2^{4r} n^2 m + mn^2 \log n)$, which means that CENTER RANKING is fixed-parameter tractable with respect to the radius parameter r.

Both algorithms might be considered practically relevant, since the hidden constants are reasonable and they both are well parallelizable, which might be of interest to search engine applications. The first algorithm can be expected to fare better for small radii and may be easier to implement. The second algorithm's run time is exponentially better for variable radius.

4.1. Search tree algorithm branching on swaps

In this section we examine a search tree algorithm that solves a CENTER RANKING instance in run time $O(r^r \cdot mn \log n)$. Let us give the outline of the algorithm. The root of the search tree represents any input vote and radius r. We start by swapping around two candidates in the vote the root node represents. For the swapping, we choose two candidates that form a discordance between the vote represented by the root node and an input vote of distance greater than r to the vote represented by the root node. We repeat the procedure and branch into the various cases produced, taking care to lower r by one when each new level of the search tree is reached. By doing this, we ensure that the search tree nodes have at most $r + 1$ children, and that the depth of the tree does not exceed r. The search tree size is no more than $O(r^r)$.

The work done per node is relatively small and so is memory consumption, which is why the algorithm might be considered practically relevant in applications where the radius parameter is known to be small.

The branching strategy of the algorithm presented in this section is similar to a search tree algorithm for the CLOSEST STRING problem. The analog of a string position in the CENTER RANKING problem is the relative ranking of two candidates. The CENTER RANKING problem is more difficult, because reversing

Algorithm 6 An algorithm that solves CLOSEST STRING in $O(nm)$ time for fixed radius, proposed by Gramm et al. [24]. It accesses the global variables $S = \{s_1, \ldots, s_k\}$, a set of strings and d, the maximum distance of a solution from any input string, which together form the input of the CLOSEST STRING problem.

```
Input: Candidate string s and integer Δd.
    In the initial call, set s = s_1 and Δd = d.
Output: A string ŝ with d_H(ŝ, s_i) ≤ d ∀i and
    d_H(ŝ, s) ≤ Δd, if a solution exists.
    The output is "not found," otherwise.
D0: If Δd < 0, then return "not found".
D1: If ∃i : d_H(s_i, s) > d + Δd, then return "not found".
D2: If d_H(s, s_i) ≤ d ∀i, then return solution s.
D3: Seek for an input string s' that is
    not within B_{d_H,k}(s).
D4: Collect any d + 1 different strings
    out of B_{d_H,1}(s) into
    a set named A, such that
    ∀a ∈ A : d_H(a, s') < d_H(s, s').
D5: For each a ∈ A, run this algorithm. As
    arguments, pass it a as the input
    string and k − 1 as the ball size.
```

the relative ranking of two candidates potentially requires changing other relative rankings, too, while the positions of strings are independent from one another.

To simplify understanding the branching strategy, we first present the CLOSEST STRING algorithm before we present the similar search tree algorithm that solves CENTER RANKING.

4.1.1. A search tree algorithm that solves CLOSEST STRING. The search tree algorithm we establish for CENTER RANKING makes use of the fact that if two permutations have Kendall-τ distance greater than r, then neither one can have put the candidates which create the discordances into the same order as the solution. A similar statement holds for strings in the CLOSEST STRING problem and it led to a good algorithm for the CLOSEST STRING problem in [24].

For quick reference, let us recall that the CLOSEST STRING problem accepts as input a set S of strings of the same length m and a integer number d, the distance parameter. The cardinality of S is denoted as n. The CLOSEST STRING problem asks: Is there a string s' such that $d_H(s, s') < d \ \forall s \in S$, where d_H denotes the Hamming distance? See Section 2.2.1 on the CLOSEST STRING problem.

Let us outline the search tree algorithm. We assign any input string s to the root node of the search tree. We seek any input string s_i of distance greater than d to string s. We start by changing one of the positions of the string assigned to the root node to match s_i. We could now choose to branch into all of the $2d$ possible strings derived from s by changing one of its positions to match s_i. However, while this branching strategy would correctly retrieve a solution, it can be shown that it suffices to branch into $d + 1$.

Remember that for a string $c \in \{0,1\}^n$ and a radius $k \in \mathbb{N}$, a ball is defined as

$$\mathbb{B}_{d_H,k}(c) \coloneqq \{b \in \{0,1\}^n : d_H(c, b) \leq k\},$$

where d_H refers to the Hamming distance, and $\{0,1\}^n$ refers to the set of all 0-1 strings of length n. The algorithm is outlined as pseudo-code in Algorithm 6.

Algorithm 6 solves CLOSEST STRING in time $O(nm + nd \cdot d^d)$, where n is the number of input strings and m is the length of every input string[1].

4.1.2. Search tree algorithm that solves CENTER RANKING. We provide a search tree algorithm that solves a CENTER RANKING instance (V, r) in time $O(r^r mn \log n)$. To describe the algorithm, we need to remember the concept of a discordance between two permutations that we defined in Definition 1 on page 6.

Definition. Let $\lambda, \mu \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. Then $\{a, b\} \subset \mathbb{N}$ is a discordance between λ and μ, if and only if λ lists a and b in another order than μ.

Note that in permutation terminology, an inversion of a permutation μ is a pair (a, b) with $a < b$, while $a >_\mu b$. Let us define the swap operation.

Definition 48. Let $(a, b) \in [n]^2, n \in \mathbb{N}$. Then $\mathrm{swap}(a, b) \in \mathrm{Sym}_{[n]}$ is the permutation that swaps around entries a and b, i.e.

$$(\mathrm{swap}(a, b))(a) = b, (\mathrm{swap}(a, b))(b) = a, \mathrm{swap}(a, b)(i) = i, a \neq i \neq b.$$

The pseudo-code solving a CENTER RANKING instance $((\lambda_1, \ldots, \lambda_m), r)$ is given in Algorithm 7. In the initial call, the candidate permutation λ is set to λ_1 and Δr is set to r.

Algorithm 7 An algorithm that solves CENTER RANKING in polynomial time for fixed radius. It accesses the global variables $V = \{\lambda_1, \ldots, \lambda_m\}$ and r, which form the input to the CENTER RANKING problem (see Problem 3 on page 7 for the definition).

```
Input: Candidate permutation λ and
    rest-radius Δr ∈ ℕ.
    In the initial call, set λ ≔ λ_1 and Δr ≔ r.
Output: A permutation λ̂ with
    τ(λ̂, λ_i) ≤ r ∀i and τ(λ̂, λ) ≤ Δd,
    if a solution exists.
    Return "not found," otherwise.
D0: If Δr < 0, then return "not found."
D1: If τ(λ, λ_1) > r − Δr,
    then return "not found."
D2: If τ(λ, λ_i) ≤ r ∀i, then return solution λ̂ ≔ λ.
D3: Seek for an j such that τ(λ, λ_j) > r.
D4: Collect any r + 1 discordances between
    λ_j and λ into a set named A.
D5: For each (a, b) ∈ A, run this algorithm.
    As arguments, pass it swap(a, b) ∘ λ as
    the candidate permutation and
    Δr − 1 as rest-radius.
```

Theorem 49. *Algorithm 7 solves* CENTER RANKING *correctly.*

PROOF. The algorithm accepts solutions only if they meet the problem. On every recursive call we decrease Δr and once Δr reaches 0, we abandon the branch, thus ensuring the termination of the algorithm in all cases. Therefore, it is clear that the algorithm handles the case where no solution exists, correctly.

We assume that the instance has a solution. Further, we assume the solution to be the identity permutation. Notice that, while simplifying the proof, this is not

[1]For details and a proof of correctness, see [24]

a real restriction, as we do not use this knowledge in the algorithm. The general case can be shown to be correct by multiplying every input permutation with $\hat{\lambda}^{-1}$, where $\hat{\lambda}$ is the solution.

The correctness follows from the observation that, in terms of Algorithm 7,

$$\exists (a,b) \in A : \tau(\mathrm{swap}(a,b) \circ \lambda, \mathbf{1}) < \tau(\lambda, \mathbf{1}), \tag{4.1.1}$$

which is to say that one of the children in the search tree has smaller distance to the solution than its parent node. As a consequence, in at least one recursive call

$$\tau(\mathbf{1}, \lambda) \leq \Delta r, \tag{4.1.2}$$

which is to say that for at least one node in each level of the search tree, the solution, if it exists, is at most Δr steps away from the permutation to the node.

To see equation (4.1.1), let us examine the set Q, defined to be the set of discordances between λ and λ_1. Each of these discordances is either an inversion in λ_1 or it is an inversion in λ. Let I_λ be the set of inversions of λ and I_{λ_1} be the set of inversions in λ_1. Now we have $A \subset Q \subset I_\lambda \cup I_{\lambda_1}$, where $\#I_\lambda \leq r, \#I_{\lambda_1} \leq r$. On the other hand, $\#A \geq r+1$; hence, A contains an inversion in λ by the pigeonhole principle. The swap that corresponds to the inversion in λ, i.e. (a,b), satisfies equation (4.1.1). To see this, a simple consideration on the candidates suffices. For candidates right and left from the two swapped elements, no new inversions are created. For candidates in between a and b, the number of inversions they are involved in can only decrease. But for the two swapped elements, one inversion disappears.

Since all $(c,d) \in A$ are tried, for at least one of the recursive calls equation (4.1.2) must hold. If a solution exists, it is found, as in every recursion level, one step leads towards it.

To justify instruction `D1`, consider the condition is not met; then one recursion step has led back to λ_1, the original input, although there must be a path to the solution where every swap decreases the distance to the solution, if a solution exists. □

Even though instruction `D1` is not strictly necessary, and does not contribute to our time bound, it might improve practical performance.

Instruction `D5` might be a little counterintuitive, as it decreases the rest-radius Δr by only 1, even though the "step" we have taken towards the solution might have removed more than one inversion. Indeed, the algorithm could be sped up—if we only knew the number of inversions that "step" $\mathrm{swap}(a,b)$ removes. However, to know the number, we would need to know the solution beforehand.

Theorem 50. *Let $V \in \mathrm{Sym}^m_{[n]}, n, m \in \mathbb{N}$ an instance of the* CENTER RANKING *problem of optimum solution radius $r \in \mathbb{N}$. Algorithm 7 solves V in time $O(r^r mn \log n)$.*

PROOF. The recursive algorithm describes a search tree of depth at most r, as parameter Δr has value r at first and is decreased to at least 0 in every level of the call hierarchy. Each node has at most $r+1$ children, as in instruction `D5`, set A is limited to $r+1$ elements. Thus the number of nodes in the search tree is in $O(r^r)$. The time to run is $O(r^r f(m,n))$, where f is the time to spent on every node.

- Instruction `D0` runs in time $O(1)$.
- Instruction `D1` runs in time $O(n \log n)$.
- Instruction `D2` runs in time $O(mn \log n)$ (see Section Section 1.4.8).
- Instruction `D3` runs in time $O(mn \log n)$.
- Instruction `D4` runs in time $O(n \log n)$.

Thus the run time per node is $O(mn \log n)$. Since the search tree size is in $O(r^r)$, the overall run time is $O(r^r mn \log n)$, completing the proof. □

Corollary 51. CENTER RANKING *is fixed-parameter tractable with respect to the parameter radius* r.

Since we know that the radius parameter is closely related to the maximum pairwise distance parameter, we find that the CENTER RANKING problem is also fixed-parameter tractable with respect to the parameter "maximum pairwise distance."

Discussion. Figuratively speaking, the algorithm walks step by step through the permutation space, where eligible permutations for a next step would be permutations of Caley distance 1.

Definition 52 (Caley distance[17])**.** Let $\nu, \nu' \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. The Caley distance $T(\nu, \nu')$ is the minimum number of transpositions $\pi_1, \ldots, \pi_k \in \mathrm{Sym}_{[n]}$ such that

$$\nu = \pi_1 \circ \cdots \circ \pi_k \circ \nu'.$$

The Caley distance is the number of transpositions needed to obtain ν from ν'. Here, a *transposition* is a permutation which can be obtained from the identity permutation by swapping two elements. For example, permutation $\pi = 321$ is a transposition, because it is obtained from the identity permutation by swapping 3 and 1. The Caley distance between $\mu = 213$ and $\mu' = 231$ is one, because $\mu = \pi \circ \mu'$.

Note that in instruction `D4`, in order to solve a CENTER RANKING instance, it does not suffice to gather all permutations out of $\mathbb{B}_{\tau,1}(\lambda)$, as $\mathbb{B}_{\tau,1}(\lambda)$ might be too small. In metric terms, the child nodes are chosen all out of $\mathbb{B}_{T,1}(\lambda)$. Note that $\mathbb{B}_{T,1}(\lambda)$ is always of greater cardinality than $\mathbb{B}_{\tau,1}(\lambda)$. We give a short proof[2].

Lemma 53 (Cohen and Deza)**.** *Let* $\lambda, \mu \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$ *be any two permutations, then*

$$T(\lambda, \mu) \leq \tau(\lambda, \mu).$$

PROOF. By definition, $T(\lambda, \mu)$ is the minimum number of transpositions needed to obtain μ from λ. This is less than or equal to the number of pairwise transpositions needed to obtain μ from λ, defined to be $\tau(\lambda, \mu)$. □

Further improvement: Computing the distances in linear time. The time to run the algorithm can be reduced down to $O(r^r mn + mn \log n)$ by creating a dictionary with one entry for every input permutation, listing the discordances of this permutation with λ. When λ gets replaced by $\mathrm{swap}(a, b) \circ \lambda$ in the recursive call, updating one dictionary entry can be carried out in time $O(n)$. Hence, all dictionary entries can be updated in time $O(mn)$, which is linear in the size of the input.

Further improvement: eliminating multiple computations of identical branches. Algorithm 7 creates a large search tree whose size is dominant to the run time. Maintaining a list of previously visited nodes might bring some speedup at the cost of increased memory usage and decreased parallelizability.

Further improvement: Parallelization. The algorithm is excellently parallelizable by referring all recursive calls in instruction `D5` to different threads. Since the search tree depth may not exceed r and the time spent per node is $O(mn \log n)$, in terms of parallel algorithms [33, p. 27ff], we can conclude the following proposition.

Proposition 54. A parallel version of Algorithm 7 that refers all recursive calls to new threads solves an instance $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$, of optimum radius r in time $O(mn \log n)$ with work $O(r^r mn \log n)$.

[2]This lemma was proved in [17, 12].

Proposition 54 implies that there is no extra cost for the parallelization. The speedup of a parallel algorithm is defined as

$$S_p(n) \coloneqq \frac{T^*(n)}{T_p(n)},$$

where T^* refers to the best known sequential run time and T_p refers to run time of the algorithm using p processors. Since the algorithm immediately branches into $r+1$ cases with equal run time, for $p \leq r+1$ we obtain a speedup of p, which is the best we can hope for. For greater p the speedup degrades slowly. We have analyzed how the time spent per node might be further parallelized. It is well-known that the Kendall-τ distance can be computed with significant speedup in parallel.

4.1.3. Concluding remarks. Algorithm 7 on page 43 has fairly good hidden constants, but the explosion in the parameter radius is enormous. After application of Data Reduction Rule 24 on page 21, it is clear that every candidate contributes to at least one discordance with another input vote. While this does not give strict bounds to the number of candidates the algorithm can reasonably handle, it is clear that the instances that the algorithm can solve must have a very specific format. They pair-wise agree on the order of nearly all of their candidates, i.e. they highly correlate.

4.2. Neighbor permutation search

We present a search tree algorithm with run time $O(2^{4d}n^2m+mn^2\log n)$, which is asymptotically exponentially better than the algorithm given in Section 4.1.2. Let us state the idea of the search tree focusing on our goal of answering the decision problem of a CENTER RANKING instance.

Each node represents a set of restrictions on the permutations that is considered as possible solutions in subsequents nodes in the search tree. These restrictions are expressed as sets of preference pairs (a, b). A set that contains pair (a, b) restricts possible solutions examined in descendants of the node, i.e. in the nodes examined through recursive calls, to permutations which list candidate a before candidate b. The root node imposes no restrictions at all. On every node of the search tree, a new set P of preference pairs is chosen which complements the set of restrictions differently in every child node. Every child node receives the set of restrictions inherited from its parent complemented by the new set of preference pairs P with a subset of the pairs reversed. For every subset of P, one child node is generated.

Since we try to solve the decision problem, we are supplied with a radius parameter. For a carefully picked choice of P, we show that we can halve the radius parameter in subsequent nodes in the search tree while still guaranteeing that a solution is found if one exists. The search tree depth is thus upper-bounded by $\log r$.

The introduction of some terminology lets us formulate the algorithm more naturally. Following a line of thoughts, a vote can be defined as a strict total order of its candidates, a strict total order in turn can be defined as a binary relation, and a binary relation can be identified by a set of pairs. For the binary relation $<_\lambda$ which expresses the order of candidates in λ, these pairs would be $\{(a, b) : a <_\lambda b\}$, as defined by Halmos [27]. We call pairs of the set $\{(a, b) : a <_\lambda b\}$ "preference pairs."

Definition 55 (Preference pairs)**.** A *preference pair* is a pair $(a, b) \in [n]\times[n], a \neq b$, $n \in \mathbb{N}$. A vote $\lambda \in \mathrm{Sym}_{[n]}$ *follows* a preference pair (a, b), if and only if $a <_\lambda b$.

In descendant nodes of the search tree, i.e. nodes that spawn due to the recursive call of the current nodes, we only examine permutations that follow the preference pairs associated with the node.

Definition 56 (Permutations following a set of preference pairs)**.** Let

$$R \subset \{(a,b) : a,b \in [n], a \neq b\}, n \in \mathbb{N}$$

be a set of preference pairs. We call $\mathrm{Sym}_{R,[n]}$ the subset of $\mathrm{Sym}_{[n]}$ that follows each preference pair in R.

Each node of the search tree is associated with a set of pairs serving as restrictions on the solution. The solutions that can be found in descendants of a node with associated set of pairs serving as a restriction is $\mathrm{Sym}_{R,[n]}$. Our algorithm starts with an empty set of preference pairs of the solution. It acquires a set of preference pairs that need to be decided in the next level of the search tree. For a possibility to decide these preferences, it adds the decided preference pairs to the set. The algorithms branches into all cases of deciding the preference pairs. A branch of the search tree can be abandoned once $\mathrm{Sym}_{R,[n]}$ is empty.

To computationally determine the emptiness of $\mathrm{Sym}_{R,[n]}$, we use two graph-related terms. The advantage of modelling the criterion in terms of graph theory instead of terms of sets is that for both concepts to be introduced, the run time on on graphs is well-researched. A directed graph is *acyclic* if and only if it contains no path that starts and ends on the same vertex [3, pp. 32-44]. A *topological sorting* is a linear ordering of the nodes of a graph in which each node comes before all nodes to which it has outbound edges [15, pp. 485-488].

Lemma 57. *Let $n \in \mathbb{N}$, and R be a set of preference pairs $R \subset \{(a,b) : a,b \in [n], a \neq b\}$, and let h be the directed graph with vertices $[n]$ and an edge from b to c if and only if $(b,c) \in R$. Then the set $\mathrm{Sym}_{R,[n]}$ is empty if and only if h is acyclic.*

Proof. If the directed graph contains no cycles, then it has a topological sorting $\lambda \in \mathrm{Sym}_{[n]}$. The topological λ, as a permutation, follows all preference pairs in R, hence it is in $\mathrm{Sym}_{R,[n]}$.

If $\mathrm{Sym}_{R,[n]}$ is not empty, it must contain a permutation μ. Let g be the tournament graph that contains vertices $[n]$ and an edge (a,b) with $a,b \in [n]$ if and only if $a <_\mu b$. Since μ shows no cyclic preferences, g must be acyclic. Since g contains every edge that h contains, h must be acyclic, too. □

The set of restrictions contains preference pairs solution if a solution is to be found in the current sub-tree. Since the solution is a total order of the candidates, the set of restrictions that defines the solution must be *transitive*, by which we mean that if it contains two pairs (a,b) and (b,c), it also needs to contain (a,c). In the above proof, we related the set of pairs serving as a restriction R to the graph of vertices $[n]$ and and an edge for every member in R, such that there is an edge in the graph from d to e if and only if the pair $(d,e) \in R$. This technique helps us determine the pairs we can add to the set of preference pairs serving as a restriction. We refer to the graph with vertices $[n]$ and edges R as the *graph associated with* R. The set of preference pairs which are implied by R is called the *transitive closure* of R. On each node, we can replace the set R by its transitive closure without changing the solvability of the instance. The term transitive closure is defined both for relations and graphs; in our terms, the transitive closure of R is the same as the set of edges of the transitive closure of the graph associated with R. A branch of the search tree can be abandoned once the graph associated with R becomes complete, i.e. there is an edge between each two vertices. However, because of the

computational complexity of determining the transitive closure, we compute it at most once. To ensure the transitivity of R it suffices to abandon branches where the graph associated with R is acyclic, since a complete directed graph is acyclic if and only if it is transitive [29, Corollary 5a].

In the subtree, the contribution of some preference pairs to the CENTER RANKING score is already known. Branching from there, we only need to add inversions that were unclear before. We only consider discordances that have not been considered already in an ancestor need in the search tree. Let us define the space of all unordered pairs of candidates.

Definition 58. Let $n \in \mathbb{N}$, then

$$\Omega_n \coloneqq \{\{a, b\} \subset [n], a \neq b\}.$$

To see whether or not a permutation follows a set of preference pairs, we can compare the preferences of the permutation with the given set.

Definition 59. Let $\lambda \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$, and $Q \subset \Omega_n$ a set of unordered pairs of candidates. Then we call

$$\lambda|_Q \coloneqq \{(a, b) : a <_\lambda b, \{a, b\} \in Q\}$$

the *preferences of λ restricted to Q.*

The Kendall-τ distance is defined to be the discordance, or the number of dirty pairs, between two permutations. On every node of the search tree, we know that if a solution is found in this branch, it follows the restrictions associated with the node. We can compute the part of the Kendall-τ distance associated with the restrictions associated with the node.

Definition 60 (Restricted Kendall-τ distance)**.** Let $\mu, \lambda \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$ and $R \subset \Omega_n$, then

$$\tau^R(\lambda, \mu) \coloneqq \#(\lambda|_R - \mu|_R)$$

is the number of pairs of candidates in R that λ and μ disagree upon.

Note that for $R = \Omega_n$, the restricted Kendall-τ distance equals the classical Kendall-τ distance. The restricted Kendall-τ distance is a pseudometric. We are mostly interested in the fact that it satisfies the triangle inequality.

Lemma 61. *Let $R \subset \Omega_n, n \in \mathbb{N}$. Function τ^R is a pseudometric.*

PROOF. Remember that a pseudometric must satisfy the following conditions for any $\lambda, \mu, \nu \in \mathrm{Sym}_{[n]}$[58]:

(1) $\tau^R(\lambda, \mu) = \tau^R(\mu, \lambda)$ (Symmetry)
(2) $\tau^R(\lambda, \lambda) = 0$
(3) $\tau^R(\lambda, \mu) \leq \tau^R(\mu, \nu) + \tau^R(\mu, \nu)$ (triangle inequality)

The first two conditions are easy to verify. We will a generalized version of the third condition, which we need to defer since it makes use of notation unintroduced at this point. The generalization are given in Lemma 64 and following it we show how it proves the triangle inequality of the restricted Kendall-τ distance. □

On every node of the search tree, we determine the part of distance from the solution to each input permutation that can be determined from the restriction associated with the node. We associate a distance d_i to each input permutation λ_i. On every node of the search tree, we decrease d_i by the part of the distance from the solution to the input permutation that can already be determined from the set of restrictions associated with the node that were not present in its parent. On every node, $r - d_i$ serves as a lower bound of the distance from the solution to λ_i.

Once $d_i < 0$ for some i at some node of the search tree, the branch of the search tree can be abandoned. Once all preference pairs have been decided, the algorithm verifies if the permutation represented by the restrictions is a solution or not.

The above branching can be understood as assigning separate distances for each input permutation. We help formalizing the cases we branch into by formulating the NEIGHBOR PERMUTATION problem[3], which generalizes the CENTER RANKING problem by allowing the specification of a maximum allowed distance from the solution to each input ranking separately. Further, the solution to the NEIGHBOR PERMUTATION problem must follow a set of given restrictions. The NEIGHBOR PERMUTATION problem asks whether there is a permutation following all of the preference pairs in a set R serving as a restriction, which have restricted Kendall-τ distance less than d_i to the each input permutation λ_i, where d_i is a distance assigned to each input permutation. The Kendall-τ distance is restricted to the preference pairs outside of the given restriction, to reflect the unknown part of the Kendall-τ distance in the original CENTER RANKING problem.

To express a set of preference pairs being outside of what has been set as a restriction before, let us define Ω_n as a "universe" of pairs of candidates, thus simplifying the notation of the complement.

Definition 62. Given a set of unordered preference pairs $R \subset \Omega_n, n \in \mathbb{N}$, we set

$$\bar{R} \coloneqq \Omega_n - R.$$

The restrictions associated with each node in the search tree is a set of ordered pairs. The Kendall-τ distance limited to the pairs of candidates present in the set can be expressed by the Kendall-τ distance if the pairs are given as unordered. Let us define a notation to refer of the unordered pairs associated with a list of ordered pairs.

Definition 63. For a set of preference pairs $R \subset \{(a,b) : a,b \in [n], a \neq b\}, n \in \mathbb{N}$, we call $R^\dagger \coloneqq \{\{a,b\} : (a,b) \in R\}$, the *unpacking* of R.

Using this notation, we can state that on any node in the search tree associated with the set of preference pairs R serving as a restriction, the distance from the solution to some input permutation λ_i must exceed $\#(R - \lambda_i|_{R^\dagger})$. The cardinality of the complement serves as a pseudometric. We prove the triangle inequality, which is a generalization of the triangle inequality of the restricted Kendall-τ distance.

Lemma 64. *For three sets of preference pairs a, b, c with*

$$a^\dagger = b^\dagger = c^\dagger, \tag{4.2.1}$$

we find

$$\#(a-b) \leq \#(a-c) + \#(b-c). \tag{4.2.2}$$

PROOF. First, let us prove

$$a - b \subset (a-c) \cup (b-c). \tag{4.2.3}$$

Let $t \in a - b$. We show $t \in (a-c) \cup (b-c) = (a-c) \cup (c-b)$. The equality holds because equation (4.2.1) implies that $b - c = c - b$. To see $t \in (a-c) \cup (c-b)$, note that either $t \in c$ or $t \notin c$. In the first case $t \in c - b$, in the other $t \in a - c$. Either way, $t \in (a-c) \cup (c-b)$, thus proving inequality (4.2.3), which directly implies inequality (4.2.2). □

We can now prove the triangle inequality for the restricted Kendall-τ distance.

[3]The name is chosen after the neighbor string problem, which is a generalization of the CLOSEST STRING problem and was employed and defined by Ma and Sun [44] to speed up the solving of CLOSEST STRING instances.

PROOF OF THE TRIANGLE INEQUALITY IN LEMMA 61 . The triangle inequality in follows from the fact that $\lambda|_R^\dagger = \mu|_R^\dagger = \nu|_R^\dagger$. Now we find, using Lemma 64,

$$\tau^R(\lambda,\mu) = \#(\lambda|_R - \mu|_R) \tag{4.2.4}$$

$$\leq \#(\lambda|_R - \nu|_R) + \#(\mu|_R - \nu|_R) \tag{4.2.5}$$

$$= \tau^R(\mu,\nu) + \tau^R(\mu,\nu), \tag{4.2.6}$$

which is the triangle inequality. □

Let us define the generalized problem that matches the behaviour of our search tree. It extends the CENTER RANKING problem by assigning a distance to each input permutation separately and demanding the solution to follow a set of preference pairs serving as a restriction. Also, the metric used is the restricted Kendall-τ distance, restricted to $\overline{R^\dagger}$, where R is the set of preference pairs serving as a restriction.

Definition 65 (NEIGHBOR PERMUTATION problem)**.** Let $(\lambda_1,\ldots,\lambda_m) \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$, and let $d_1, d_2, \ldots, d_m \in \mathbb{N}$, and let $R \subset \{(a,b) : a,b \in [n], a \neq b\}$ be a set of preference pairs. The NEIGHBOR PERMUTATION problem seeks for a permutation $\sigma \in \mathrm{Sym}_{R,[n]}$ such that

$$\tau^{\bar{R^\dagger}}(\lambda_i,\sigma) \leq d_i \forall i \in [1,n].$$

To stress the resemblance between the CENTER RANKING problem and the NEIGHBOR STRING problem, let us state how easily the CENTER RANKING problem can be reduced to NEIGHBOR PERMUTATION problem, i.e. how we can make use of an algorithm which solves the NEIGHBOR PERMUTATION problem to solve the CENTER RANKING problem.

Remark 66. We can reduce the CENTER RANKING problem to the NEIGHBOR PERMUTATION problem by transforming a CENTER RANKING instance $((\lambda_1,\ldots,\lambda_m),r)$ into a NEIGHBOR PERMUTATION instance $((\lambda_1,r),\ldots,(\lambda_m,r),\emptyset))$ in time $O(nm)$. Thus, the run time to solve the NEIGHBOR PERMUTATION problem is the same as the run time to solve the CENTER RANKING problem.

We still need to identify the pairs of candidates whose preferences are added to the set of preference pairs serving as a restriction on every node in the search tree. The following choice[4] allows us to halve d_1 in every child node.

Definition 67 (Permutation disagreement)**.** Let $\lambda,\mu \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. We call $P(\lambda,\mu)$ the *disagreement choices of* λ *despite* μ. We set $P(\lambda,\mu)$ to be the set of ordered pairs (a,b) with $a <_\lambda b$, but $a >_\mu b$.

Note that Definition 67 is not symmetric. It reflects the preferences of λ.

We halve d_1 on every new level in the search tree of the search tree algorithm. Yet, if a solution exists, it still is found and identified as a solution. To prove that we can losslessly decrease d_1 so greatly, we use the following Lemma. It is similar to [44, Lemma 1] by Ma and Sun.

Lemma 68. *Let* $((\lambda_1,d_1),\ldots,(\lambda_m,d_m),R)$ *be an instance of the* NEIGHBOR PERMUTATION *problem, where* $n,m \in \mathbb{N}, \lambda_i \in \mathrm{Sym}_{[n]}\ \forall i$ *and* $R \subset \Omega_n$*. If* j *satisfies* $\tau^{\overline{R^\dagger}}(\lambda_1,\lambda_j) > d_j$*, then for* P *being the disagreement between* λ_1 *despite* λ_j*, we find for any solution* σ *of the* NEIGHBOR PERMUTATION *problem,* $\tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma,\lambda_1) < \frac{d_1}{2}$.

[4]Note that the disagreement between two permutations is essentially the set of all discordances between them, as defined in Definition 4.1.2 on page 43. We use this term for easy comparison with the Ma and Sun algorithm.

PROOF. With the conditions of the lemma met, we find

$$d_j < \tau^{\overline{R^\dagger}}(\lambda_1, \lambda_j) = \tau^{\overline{R^\dagger} \cap P^\dagger}(\lambda_j, \lambda_1) < \tau^{\overline{R^\dagger} \cap P^\dagger}(\sigma, \lambda_1) + \tau^{\overline{R^\dagger} \cap P^\dagger}(\sigma, \lambda_j). \tag{4.2.7}$$

Furthermore,

$$\tau^{\overline{R^\dagger} \cap P^\dagger}(\sigma, \lambda_1) + \tau^{\overline{R^\dagger} \cap P^\dagger}(\sigma, \lambda_j) \tag{4.2.8}$$

$$= (\tau^{\overline{R^\dagger}}(\sigma, \lambda_1) - \tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_1)) + (\tau^{\overline{R^\dagger}}(\sigma, \lambda_j) - \tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_j)) \tag{4.2.9}$$

$$= \tau^{\overline{R^\dagger}}(\sigma, \lambda_1) + \tau^{\overline{R^\dagger}}(\sigma, \lambda_j) - 2\tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_1) \tag{4.2.10}$$

$$\leq d_1 + d_j - 2\tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_1). \tag{4.2.11}$$

Here, equation (4.2.10) holds because $\tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_1) = \tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_j)$, because $\lambda_1|_{\overline{P^\dagger}} = \lambda_j|_{\overline{P^\dagger}}$. By inequality (4.2.7), we get $d_1 + d_j - 2\tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_i) > d_j$, and as a consequence $\tau^{\overline{R^\dagger \cup P^\dagger}}(\sigma, \lambda_1) < \frac{d_1}{2}$, as was claimed. □

Algorithm 8 Algorithm NEIGHBOR PERMUTATION SEARCH, solving the CENTER RANKING problem for a given radius.

Input: An instance of NEIGHBOR PERMUTATION SEARCH
$((\lambda_1, d_1), \ldots, (\lambda_m, d_m), R)$ with $\lambda_i \in \mathrm{Sym}_{[n]} \forall i$
and $R \subset \{(a, b) : a \neq b, a, b \in [n]\}$.
In the initial call, R must be empty.
Output: Permutation $\sigma \in \mathrm{Sym}_{R,[n]}$ such that
$\tau^{\overline{R^\dagger}}(\lambda_i, \sigma) <$
$d_i) \forall i$. Return "not found," if there is none.
N0: [Ensure termination, edge conditions]

Check if R is cyclic. If it is, return "Not found."
If $d_i < 0$ for some i, return "not found."
If $\#R \geq \binom{n}{2}$, then check if the topological
sorting of the graph corresponding to R
is a solution. If so, return it, otherwise
return "Not found."
N1: [Compute q]
Try to find q s.t. $\tau^{\overline{R^\dagger}}(\lambda_q, \lambda_1) > d_q$. If there
is none, compute the graph corresponding to R.
Compute the transitive closure of the graph.
Set the preference pairs R' to be the set of
edges of the computed transitive closure.
If the graph corresponding to
$\lambda_1|_{R'^\dagger} \cup R'$ is acyclic,
its topological sorting is the solution.
Otherwise, return "Not found."
N2: [Check if the instance is trivially unsolvable]
Verify that $\tau^{\overline{R^\dagger}}(\lambda_1, \lambda_q) \leq d_1 + \max_i d_i$;
otherwise, return "not found."
N3: [Compute the disagreement]
Compute the disagreement of λ_1 despite λ_q,
intersected with $\overline{R^\dagger}$ and store it in P.
N4: [Compute the set of cases to branch into]
$A' \coloneqq \{\{(y, x) : (x, y) \in p\} \cup P - p : p \subset P\}$, i.e.
set A' to be all ways to flip a subset of P.
N5: [Complete the cases to branch into]
Set $A \coloneqq \{a \cup R : a \in A'\}$.
N6: [Distances in the branches]
Set $e_{a,i} \coloneqq d_i - \#(\lambda_i|_{c^\dagger} - a)$ for $i \neq 1$ and
$e_{a,1} \coloneqq \min\{d_1 - \#(\lambda_1|_{c^\dagger} - a), \lceil d_1/2 \rceil - 1\}$,
where $c = a - R$.
N7: [Recursive call]
Run NEIGHBOR PERMUTATION SEARCH for each
$a \in A$ with input $((\lambda_1, e_{a,1}), \ldots, (\lambda_m, e_{a,m}), a)$.
N8: [Unless a recursive call found a result]
Return "not found."

We can now formulate the search tree algorithm. See Algorithm 8.

The algorithm restricts the allowed sets of restrictions R, because the algorithm would not operate correctly for arbitrary R. For example, for an input

$((\lambda_1 = 123, d_1 = 500)), R = \{(3,1)\}$, instruction `N1` is unable to compute q, and therefore attempt to compute the topological sorting of the graph corresponding to $\lambda_1|_{R^{\bar{\gamma}\dagger}} \cup R'$. The algorithm fails, since the graph corresponding to $\lambda_1|_{R^{\bar{\gamma}\dagger}} \cup R'$ is cyclic. In our example, $\lambda_1|_{R^{\bar{\gamma}\dagger}} \cup R'$ would be the 3-cycle depicted in Figure 4.2.1 on page 55.

The algorithm requires $\lambda_1|_{R^{\bar{\gamma}\dagger}} \cup R'$ to be acyclic in case `N1` cannot determine q. It does compute the result correctly for $R = \emptyset$ and sets R that are generated by the algorithm itself in recursive calls. Generalizing the algorithm to be able to handle all sets R could be done by trying all ways of breaking the cycles in $\lambda_1|_{R^{\bar{\gamma}\dagger}} \cup R'$ by reversing edges originating from $\lambda_1|_{R^{\bar{\gamma}\dagger}}$.

To understand Algorithm 8 better, it might be advisable to examine some of its intermediate steps during execution. Appendix A contains a complete run of the algorithm with all intermediate results printed.

Theorem 69. *Algorithm 8 correctly solves the* NEIGHBOR PERMUTATION *problem.*

PROOF. Let us prove that if the instance has a solution, the algorithm finds it. First note that instruction `N0` correctly determines the answer in case $\#R \geq \binom{n}{2}$, thus ensuring that the algorithm terminates, since R grows in every recursive call. We need to show that if the algorithm returns a solution, it is the correct one; we then show that if a solution exists, it is reached by the algorithm. We then show that a solution is found, if one exists.

We need to show that if σ is a solution of an instance I' that we branched into,

$$I' \coloneqq ((\lambda_1, e_{a,1}), \ldots, (\lambda_m, e_{\alpha,m}), a), \tag{4.2.12}$$

then $\sigma \in \mathrm{Sym}_{[n],a}$ it is also a solution of

$$I \coloneqq ((\lambda_1, d_1), \ldots, (\lambda_m, d_m), R). \tag{4.2.13}$$

To show that a solution returned by a recursive call also solves the original problem, let σ be a solution of an instance that we branched into, i.e. $\tau^{\overline{a^\dagger}}(\sigma, \lambda_i) \leq e_{a,i}$. We show that $\tau(\sigma, \lambda_i) \leq d_i$. In the terms of the algorithm, note that $\tau^{\overline{a^\dagger}} = \tau^{\overline{R^\dagger}} - \tau^{a^\dagger - R^\dagger}$, because $R \subset a$.

For $i \neq 1$, given a solution σ of I' and $e_{i,\alpha}$ as computed by the algorithm,

$$\tau^{\overline{R^\dagger}}(\sigma, \lambda_i) - \#\left(\sigma_i|_{a^\dagger - R^\dagger} - \lambda|_{a^\dagger - R^\dagger}\right) \tag{4.2.14}$$

$$= \tau^{\overline{R^\dagger}}(\sigma, \lambda_i) - \tau^{a^\dagger - R^\dagger}(\sigma, \lambda_i) \tag{4.2.15}$$

$$= \tau^{\overline{a^\dagger}}(\sigma, \lambda_i) \tag{4.2.16}$$

$$\leq e_{i,\alpha} \tag{4.2.17}$$

$$= d_i - \#\left(\lambda_i|_{a^\dagger - R^\dagger} - a\right) \tag{4.2.18}$$

$$= d_i - \#\left(\lambda_i|_{a^\dagger - R^\dagger} - \sigma|_{a^\dagger - R^\dagger}\right) \tag{4.2.19}$$

To see equation (4.2.19), note that $\sigma \in \mathrm{Sym}_a$. Equation (4.2.19) holds because $a = \sigma|_{a-R}$. After regrouping, we find $\tau^{\overline{R^\dagger}}(\sigma, \lambda_i) \leq d_i$, implying that σ solves I for $i \neq 1$. For $i = 1$, the above calculation combined with Lemma 68 show that if a recursive call returns a solution, it is also a solution of its parent node.

We use induction on d_1 to show that a solution is found, if one exists. Clearly, for $d_1 < 0$, the algorithm is correct. Let us examine the case $d_1 \geq 0$.

If instruction `N1` cannot find q, then, since none of the conditions in `N0` applied, if λ_1 were within $\mathrm{Sym}_{[n],R}$, clearly λ_1 would solve the instance. However, we cannot generally assume that. We deal with the case that q cannot find q separately at the end of the proof.

If instruction `N1` finds q successfully, then at least one of the sets of preference pairs in A' must reflect σ's preferences, that is $\sigma \in \mathrm{Sym}_{a,[n]}$ for some $a \in A'$ after step `N4`. Since $\sigma \in \mathrm{Sym}_{R,[n]}$, we find that $\sigma \in \mathrm{Sym}_{R \cup a,[n]}$ as obtained by instruction `N5`. Analog the above, for a we find $\tau^{\overline{a^\dagger}}(\sigma, \lambda_i) < e_{a,i}$, and specifically $\tau^{\overline{a^\dagger}}(\sigma, \lambda_1) < e_{a,1}$. Hence, if a solution exists, it also solves one of the instances that instruction `N7` tries, namely the instance with restriction set $a \cup R$. The recursive call with restriction set $a \cup R$ must at least halve d_1. Therefore, we conclude per induction that the algorithm operates properly for the instance with restriction set $a \cup R$, and therefore correctly returns σ. We proved that if a solution exists, it is found.

We discuss the case that instruction `N1` did not find q and prove that the answer returned by the algorithm is correct. If λ_1 is not within $\mathrm{Sym}_{[n],R}$, instruction `N1` changes the preference pairs of λ_1 that are discordant to the entries of R to match R. A solution must follow all preference pairs in R. It also needs to follow all implied preference pairs in R, that is, given two preference pairs $(a, b), (b, c) \in R$, the solution must also follow the preference pair (a, c). Therefore, we can replace R by R' which is obtained from R by inserting all implied preference pairs. This is equivalent to replacing R with the set of pairs representing the edges on transitive closure on the graph representing[5] R. The result of changing all preference pairs of λ_1 discordant with R' to match R' is the set $\lambda_1|_{\overline{R'^\dagger}} \cup R'$.

We will show the existence of a permutation with preference pairs $\lambda_1|_{\overline{R'^\dagger}} \cup R'$. We will show that the permutation with preference pairs $\lambda_1|_{\overline{R'^\dagger}} \cup R'$ either is a solution, or otherwise, there cannot be a solution, which is precisely what instruction `N1` tests.

A permutations with preference pairs $\lambda_1|_{\overline{R'^\dagger}} \cup R'$ exists if and only if the graph corresponding to $\lambda_1|_{\overline{R'^\dagger}} \cup R'$ is acyclic. Note that the permutation we find does not need to be a solution. Note that the graph corresponding to R' is transitive. We show the graph represented by $R' \cup \lambda_1|_{\overline{R'^\dagger}}$ to be acyclic by contradiction. Let us assume that the graph would contain a cycle of length 2. Since both the graphs representing R' and $\lambda_1|_{\overline{R'^\dagger}}$ are acyclic, one of the edges would represent a preference in R', the other one would represent a preference in $\lambda_1|_{\overline{R'^\dagger}}$ of the same two candidates, contradicting the restriction of $\lambda_1|_{\overline{R'^\dagger}}$ to pairs of candidates R' has no preference upon.

Note that the graph corresponding to $\lambda_1|_{\overline{R'^\dagger}} \cup R'$ is a tournament. If a tournament graph has a cycle of length $k \geq 3$, then it also contains a cycle of length 3 [29, Proof of theorem 7]. Since we know that R' is transitive and acyclic, only one of the edges of the cycle of length 3 can stem from R. That is because when any two edges from a cycle of length 3 are chosen, their transitive closure implies the third, left out edge. Hence, two of the edges cycle of length three must originate from $\lambda_1|_{\overline{R'^\dagger}}$. Without loss of generality, let the three nodes be 1, 2 and 3 as depicted in Figure 4.2.1 on the facing page, where only the red (light) edge, (3,1) originates from R'. However, R' cannot be any set of restrictions, it must be either empty or have been created earlier by the algorithm, further down the call stack. Since we already concluded that one edge must stem from R', R' cannot be empty. By carefully examining the creation of R' as the transitive closure of R, as an entry of A after instruction `N4` down the stack, the edge $(1, 3)$ must have got into R' because for a previous $\tilde{q}$, λ_1 and $\lambda_{\tilde{q}}$ were discordant on the order of 1 and 3, with $\lambda_{\tilde{q}}$ favoring 3 over 1, while λ_1 favored 1 over 3. However, $\lambda_{\tilde{q}}$ must also have been discordant also on the preference on 1 and 2 or on the preference on 2 and 3, because otherwise $\lambda_{\tilde{q}}$

[5]Indeed, R indeed meets the definition of a binary relation, and the concept of transitive closures is also defined for relations in a way compatible with our usage, see [65]. We could say that R' is defined to be the transitive closure of R.

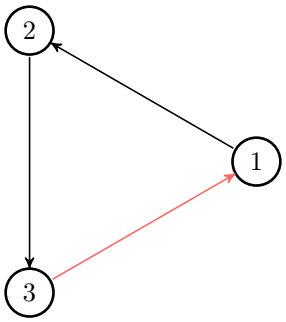

Figure 4.2.1: Cyclic graph of length three, referred to from the proof of Theorem 69.

would have had cyclic preferences. At least one of the other pairs must be within R, too, since all discordances between the two permutations λ_1 and $\lambda_{\tilde{q}}$ are reflected in all entries of A. This contradicts our assumption that only one of the entries originates from R'. Hence, the graph represented by $R' \cup \lambda_1|_{\overline{R'^\dagger}}$ is acyclic. Let π be the topological sorting of the graph corresponding to $R' \cup \lambda_1|_{\overline{R'^\dagger}}$. We showed that a permutation with preference pairs $\lambda_1|_{\overline{R'^\dagger}} \cup R'$ exists.

Since $\tau^{\overline{R'^\dagger}}(\pi, \lambda_i) \leq \tau^{\overline{R^\dagger}}(\pi, \lambda_i) \leq d_i \ \forall i$, π is a solution of the instance $((\lambda_1, d_1), \ldots, (\lambda_m, d_n))$ with restriction set R'. Since R' only adds necessary pairs to R, every solution of the given instance with restriction R must be a solution of the instance with restriction set R', too. Hence, in case π is not a solution to the original instance with restrictions R, there cannot be a solution. If π is a solution of the instance with restriction set R, it is found and correctly returned. Hence, if a solution exists and q cannot be computed, instruction `N1` finds and returns the solution. If q cannot be computed and there is no solution, instruction `N1` correctly returns that there is no solution. This completes the proof. □

Theorem 70. *Let* $((\lambda_1, d_1), \ldots, (\lambda_m, d_m), R)$ *be an instance of the* NEIGHBOR PERMUTATION *problem, where* $n, m \in \mathbb{N}, \lambda_i \in \mathrm{Sym}_{[n]}\ \forall i$ *and* $R \subset \Omega_n, d \coloneqq \max_i d_i$. *The neighbor permutation search algorithm runs in time* $O(2^{4d} n^2 m + mn^2 \log n)$. *This means that the* NEIGHBOR PERMUTATION *problem is fixed-parameter tractable with respect to the parameter* d_1.

Proof. Let us compute the number of leaves of the search tree, since it dominates the run time. The number of children of an inner node is $\#A$. Let us recall that $\#A = \#A'$ and

$$A' \coloneqq \{\{(y, x) : (x, y) \in p\} \cup P - p : p \subset P\}.$$

Since $P^\dagger \subset \overline{R^\dagger}$, we find that $\#A \leq 2^{\#\left(\overline{R^\dagger}\right)}$. Unfortunately, this inequality appears to be too rough to prove the time bound that we claimed. Instead, we present a recurrence equation for the size of the set of leaves of the search tree, which splits A into slices depending on the distance to λ_1.

For each $a \in A$, where $k \coloneqq \#(\lambda_1|_{P^\dagger} - a)$ and $d \coloneqq \max_i d_i$ we find:

$$\#(P^\dagger) = \tau^{\overline{R^\dagger}}(\lambda_1, \lambda_q) \tag{4.2.20}$$

$$= \#\left(\lambda_1|_{\overline{R^\dagger}} - \lambda_q|_{\overline{R^\dagger}}\right) \tag{4.2.21}$$

$$\leq \#\left(\lambda_1|_{\overline{R^\dagger}} - a\right) + \#\left(\lambda_q|_{\overline{R^\dagger}} - a\right) \tag{4.2.22}$$

$$\leq k + d_q \tag{4.2.23}$$

$$\leq d + k \tag{4.2.24}$$

Inequality (4.2.22) follows from Corollary 64. Note that the size of $\#(P^\dagger)$ does not depend on k.

We bound the sizes of $A_k \coloneqq \{a \in A' : \#(a - \lambda_1|_{P^\dagger}) = k\}$, which constitute A in the sense that

$$A' = \bigcup_{k\in[1,d_1]} A_k. \tag{4.2.25}$$

We have

$$\#A_k \leq \binom{\#(P^\dagger)}{k} \leq \binom{d+k}{k}. \tag{4.2.26}$$

Let $T(d, d_1)$ be the size of the search tree. Then the size of the set of leaves rooted at α can be bounded by $T(d, \min\{d_1 - k, \lceil d_1/2\rceil - 1\})$. The number of leaves satisfies

$$T(d, d_1) \leq \sum_{k=\lfloor d_1/2\rfloor+1}^{d_1} \binom{d+k}{k} T(d, d_1 - k) \tag{4.2.27}$$

$$+ \sum_{k=0}^{\lfloor d_1/2\rfloor} \binom{d+k}{k} T(d, \lceil d_1/2\rceil - 1) \tag{4.2.28}$$

$$= I_1 + I_2. \tag{4.2.29}$$

Clearly, $T(d, 0) = 1$, because of instruction `NO`.

We prove by induction on k that for $k \geq 1$,$k < d_1$:

$$T(d, \tilde{d}) \leq 2^{2d} \binom{d+\tilde{d}}{\tilde{d}}. \tag{4.2.30}$$

For $\tilde{d} = 1$, we know

$$T(d, 1) \leq d + 2, \tag{4.2.31}$$

which implies inequality (4.2.30).

For $\tilde{d} = 2$, because of inequality (4.2.29), we find

$$T(d, 2) \leq \binom{d+2}{2} + d + 2, \tag{4.2.32}$$

which implies inequality (4.2.30).

For the rest of the proof by induction, suppose $d_1 > 2$ and inequality (4.2.30) is true for $0 \leq \tilde{d} < d_1$. We bound I_1 and I_2 separately.

$$I_1 = \sum_{k=\lfloor d_1/2 \rfloor+1}^{d_1} \binom{d+k}{k} T(d, d_1-k) \tag{4.2.33}$$

$$\leq \sum_{k=\lfloor d_1/2 \rfloor+1}^{d_1} \binom{d+d_1}{k} T(d, d_1-k) \tag{4.2.34}$$

$$\leq \sum_{k=\lfloor d_1/2 \rfloor+1}^{d_1} \binom{d+d_1}{k}\binom{d+d_1}{d_1-k} 2^{2(d_1-k)} \tag{4.2.35}$$

$$= \binom{d+d_1}{d_1} \sum_{k=\lfloor d_1/2 \rfloor+1}^{d_1} \binom{d_1}{k} 2^{2(d_1-k)} \tag{4.2.36}$$

$$\leq \binom{d+d_1}{d_1} 2^{d_1-1} \sum_{k=\lfloor d_1/2 \rfloor+1}^{d_1} \binom{d_1}{k} \tag{4.2.37}$$

$$\leq \binom{d+d_1}{d_1} 2^{2d_1-2}. \tag{4.2.38}$$

We can proceed to show

$$I_2 \leq 3(\binom{d+d_1}{d_1} 2^{2d_1-2}, \tag{4.2.39}$$

where $k_0 \coloneqq \lceil d_1/2 \rceil$.

$$I_2 = \sum_{k=0}^{k_0-1} \binom{d+k}{k} T(d, d_1-k_0) \tag{4.2.40}$$

$$\leq \sum_{k=0}^{k_0-1} \binom{d+k}{k}\binom{d+d_1-k_0}{d_1-k_0} 2^{2(d_1-k_0)} \tag{4.2.41}$$

$$= \binom{d+d_1-k_0}{d_1-k_0} 2^{2(d_1-k_0)} \sum_{k=0}^{k_0-1} \binom{d+k}{k} \tag{4.2.42}$$

$$\leq \binom{d+d_1-k_0}{d_1-k_0} 2^{2(d_1-(\lceil d_1/2 \rceil-1))} \binom{d+k_0}{k_0} \tag{4.2.43}$$

Now to show the bound for I_2, we only need to show

$$\binom{d+d_1-k_0}{d_1-k_0}\binom{d+k_0}{k_0} 2^{-2k_0} \leq \frac{3}{4}\binom{d+d_1}{d_1}, \tag{4.2.44}$$

or equivalently

$$\binom{d+d_1-k_0}{d_1-k_0}\binom{d_1}{k_0} \leq \frac{3}{4} 2^{2k_0} \binom{d+d_1}{d_1-k_0}. \tag{4.2.45}$$

The last equation is true, because

$$\binom{d+d_1-k_0}{d_1-k_0} \leq \binom{d+d_1}{d_1-k_0}, \binom{d_1}{k_0} \leq \frac{1}{2} 2^{d_1+1} < \frac{3}{4} 2^{2k_0}, \tag{4.2.46}$$

thus showing inequality (4.2.39).

Summarizing, we have

$$\begin{aligned} T(d,d_1) \leq & \quad I_1 + I_2 & \text{due to (4.2.29)} \\ \leq & \quad \tbinom{d+d_1}{d_1} 2^{2d_1-2} + 3(\tbinom{d+d_1}{d_1} 2^{2d_1-2} & \text{due to (4.2.33) and (4.2.39)} \\ \leq & \quad 2^{2d}\tbinom{d+d_1}{d_1} & \text{due to induction and (4.2.30)} \end{aligned}$$

The implication is that the search tree has at most $O(2^{2d}\binom{d+d_1}{d_1}) = O(2^{4d})$ nodes. Let us analyze the run time needed per node.

- `N0` can be solved in run time $O(mn^2)$. Checking if the graph corresponding to R is acyclic can be achieved by computing a topological sorting which is possible in time $O(\text{edges} + \text{vertices}) = O(n^2)$, see e.g. [35].
- Instruction `N1` can be executed in time $O(mn^2)$ if the transitive closure does not need to be computed. It needs to be computed at most once for all recursive calls of the algorithm. The transitive closure of the algorithm can be computed expected time $O(n^2)$ [53], or in time $O(n^2 \log n)$ with reasonable practical constants [9], or in time $O(n^{2.376})$ with large hidden constants [34, 14]. We assume $O(n^2 \log n)$ deterministic time for the sake of simplicity.
- Instruction `N2` can be executed in time $O(n^2 m)$, as computing the restricted Kendall-τ distance can be done in time $O(n^2)$.
- Instruction `N3` can be executed in time $O(n^2)$.
- When the cost of instruction `N4` is assigned to its recursive calls, the run time per node of instruction `N4` and `N5` is $O(n^2)$.
- Similarly, the run time per node of instruction `N7` is $O(1)$.
- Instruction `N8` runs in time $O(1)$.

Altogether, the time needed per node is within $O(n^2 m)$ and instruction `N1` additionally might require a run of time $O(mn^2 \log n)$. As the number of leaves of the search tree is within $O(2^{4d})$, so is the total number of nodes. Summarizing, we find the total run time to be within $O(2^{4d} \cdot n^2 m + mn^2 \log n)$. □

For a reference implementation for which all of the above bounds hold, see Appendix A.

Corollary 71. *Let $V \in \mathrm{Sym}^m_{[n]}, n, m \in \mathbb{N}$ be an instance of the* CENTER RANKING *decision problem with radius r. The neighbor permutation search algorithm solves V in time $O(2^{4r} \cdot n^2 m + mn^2 \log n)$.*

PROOF. Using Remark 66 on page 50, we can reduce a CENTER RANKING instance to an instance of NEIGHBOR PERMUTATION in time $O(nm)$ such that the latter is solvable if and only if the prior is solvable. The NEIGHBOR PERMUTATION instance can be solved in time $O(2^{4d} \cdot n^2 m + mn^2 \log n)$. □

Since the sets R in the leaves of the algorithm must be acyclic, the algorithm cannot have more than $n!$ leaves. Since the leaves dominate the run time, we can establish a corollary that restricts the run time of the neighbor permutation search algorithm depending only on n.

Corollary 72. *Let $V \in \mathrm{Sym}^m_{[n]}, n, m \in \mathbb{N}$ be an instance of the* CENTER RANKING *decision problem with radius r. The neighbor permutation search algorithm solves V in time*

$$O(n! \cdot n^2 m + mn^2 \log n).$$

The neighbor permutation search has been practically implemented and can solve even large arbitrary instances for $r \leq 6$ in less than 20 seconds.

For random instances with possibly large r, the algorithm performs significantly worse than the trivial algorithm which determines the solution quality of

every permutation of length n and then chooses the permutation with the best solution quality. Recall that the performance of the trivial algorithm is within $O(n! \cdot mn \log n)$. The hidden constant of work per recursive call in the neighbor permutation search algorithm might be reasonable, but it is significantly worse than obtaining the solution quality of a specific permutation.

For some instances the algorithm might really consume time $\Theta(2^{4r}n^2m)$. If, for example, in instruction `N4` the two permutations compared differ only in adjacent pairs and are of distance $2r$, then A is of size 2^{2r}. The practical performance of the algorithm is discussed and compared more thoroughly in Chapter 6.

Further improvement: finding the minimum radius for the CENTER RANKING *problem.* We have seen how we can solve the decision problem CENTER RANKING by making use of the neighbor permutation search algorithm. A common technique to solve the optimization problem associated with a decision problem is to run a binary search for the smallest parameter in question. A binary search, as defined by Sedgewick[55, p. 236], seeks within a specified range. To determine the range, the radius parameter can be doubled until the decision problem becomes solvable for the radius. Then, the minimum parameter must be within 0 and the radius parameter for which it was solvable.

We present an adaptation of the Sedgewick algorithm on binary search adapted to our needs. Instead of running the initial subsequent doubling, we seek within range $[\frac{d_{\max}}{2}, d_{\max}]$, see Section 3.4.

Algorithm 9 solves the CENTER RANKING optimization problem in time $O(\log r(n^2m2^{4r} + mn^2 \log n))$. An actual implementation can be found in Appendix A. The run times of the algorithm for practical instances are thoroughly examined in Chapter 6.

Algorithm 9 Algorithm that solves the CENTER RANKING optimization problem in time $O(\log r(n^2m2^{4r} + mn^2 \log n))$. It runs a binary search for the optimum parameter and recursively calls the neighbor permutation search algorithm for each candidate.

```
Input: An instance (λ_1, ..., λ_m) of the optimization
    problem CENTER RANKING.
Output: The minimum r such that ∃σ∀i : τ(σ, λ_i) ≤ r.
B1: [Initialization]
    Set r := max_{i,j∈[m]} τ(λ_i, λ_j). Set l := ⌊r/2⌋.
B2: [Terminate loop if a solution was found]
    If r = l, return r.
B3: [Set the candidate] Set x := ⌊(l+r)/2⌋.
B4: [Reduce to neighbor permutation search] Run the

    neighbor permutation search algorithm with input
    ((λ_1, x), ..., (λ_m, x)), R = ∅. If the instance
    was solvable, set r := x, otherwise l := x + 1.
B5: [Loop] Return to B2.
```

Further improvement: initializing the set of restrictions. The algorithm solving CENTER RANKING might be sped up by initializing the set of restrictions not just by the empty set. We know from Lemma 15 on page 16 that the CENTER RANKING problem satisfies the weak Pareto criterion, which implies that if all votes agree on the ordering of a certain pair, that ordering shall be found in the final aggregation. Hence, we can initialize the set of restrictions in our input to the NEIGHBOR PERMUTATION problem to be the intersection of all pairwise agreements between the

votes. Unfortunately, the problem stays NP-hard even for instances where this set is empty, as for example the instances generated by the reduction in the proof of Theorem 17 on page 17.

If the set of restrictions is already initialized with some preferences, this restricts the number of branches to go into, which can speed up the computation.

Further improvement: parallelization. The algorithm described above can also be parallelized by referring all recursive calls to different threads. On this algorithm, all nodes on one level of the search tree would be computed in parallel, reducing the time needed to the depth of the search tree times the time per node. In terms of parallel algorithms [33], on $p \in \mathbb{N}$ processors, the algorithm would run in time $T = O(pn^2m)$ and with work (number of operations across all threads) $W = O(2^p \cdot mn \log n + n^2m)$.

Concluding remarks. The developed algorithm is similar to that given by Ma and Sun [44]. The proof of Theorem 70 closely resembles [44, Theorem 1] for an alphabet $\Sigma = \{0, 1\}$. The algorithm in [44] solves CLOSEST STRING and yields a run time of $O(nm + nd \cdot 2^{4d}(\#\Sigma - 1)^d)$, where n is the string length, m the number of input strings and d is he ball size al input strings need to fit in. It is perhaps surprising that while bringing forward the CLOSEST STRING algorithm, the equivalent of a certain string position in the CLOSEST STRING problem is indeed a preference pair (a, b) with either $a <_\lambda b$ or $a >_\lambda b$ for some permutation λ. The Hamming distance that is to be minimized in the CLOSEST STRING problem counts the positions two strings differ in, while the Kendall-τ distance counts the preference pairs that differ for two permutations. Thus, identifying string positions with preference pairs perhaps appears more natural.

CHAPTER 5

Parameterized for "number of candidates and radius combined"

The run time complexity of solving the CENTER RANKING problem can be confined to both the number of votes and to the radius parameter. It comes as no surprise that the CENTER RANKING problem is fixed-parameter tractable with respect to the parameter "number of candidates and radius combined." We present two algorithms which are assumed to perform well in case the radius and the number of candidates are both assumed to be small.

The first algorithm is discussed in Section 5.2. It is proved to run in time $O\left(mn \log n \cdot \min\left\{r(2r)^n, rn^{2r}, n!\right\}\right)$. The second algorithm is discussed in Section 5.3 and proved to run in time $O(mn \log n \cdot \min\{rn^r, n!\})$. For $n < 16$, the second algorithm represents an improvement over the neighbor permutation search algorithm in Section 4.2. For $n < r$ it represents an improvement over the search tree algorithm presented in Section 4.1. Comparing both algorithms presented in this chapter, the first one is a little simpler and may be easier to implement. However, the second one is expected to outperform the first algorithm, since it enumerates less permutations and does not seem to require more work to be done per enumeration. In case that two of the input permutations have Kendall-τ distance greater than or equal to $2r - 1$, we prove the algorithm to perform in $O(2^{2r}mn \log n)$ time.

In the first algorithm, rather than trying all permutations of the given n candidates, we attempt to try only permutations of distance at most r to some input permutation. We know that the solution of an instance (V, r) of the CENTER RANKING problem can have at most Kendall-τ distance r from any input permutation.

We can further shrink the number of permutations tried. After choosing two input permutations, we observe that the solution of an instance can have at most distance r to both rankings. Further, we demand that the two input rankings be far from one another. The second algorithm proposed thus limits the permutations tried.

The first section provides an observation that will allow us to translocate permutations at will, which we make use of in both algorithms. We then present the two algorithms.

5.1. Translocating instances

In the following sections, we will enumerate balls in the metric space of permutations with respect to the Kendall-τ distance. We can save a great deal of computation by assuming the center of a ball to be the identity permutation. This section provides a lemma that allows us to translocate permutations while retaining their original pairwise distances.

We make use of a property of the Kendall-τ distance that we proved in Lemma 10 on page 11. It lets us translocate the instances to our liking. We repeat the lemma for quick reference.

Lemma. *Let $\mu, \lambda, \pi \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. Then for the Kendall-$\tau$ distance,*

$$\tau(\lambda, \mu) = \tau(\pi \circ \lambda, \pi \circ \mu).$$

When analyzing balls of permutations, we can translocate the center. We will use the following corollary in the following chapter, where we will enumerate balls.

Corollary 73. *Let $\mu \in \mathrm{Sym}_{[n]}$ and $n, r \in \mathbb{N}$, then $\mathbb{B}_r(\mu) = \mu^{-1}\left(\mathbb{B}_r(\mathbf{1})\right)$.*

Proof. This follows from

$$\pi \in \mathbb{B}_r(\mu) \tag{5.1.1}$$

$$\Leftrightarrow \tau(\pi, \mu) \leq r \tag{5.1.2}$$

$$\Leftrightarrow \tau(\mu^{-1} \circ \pi, \mu^{-1} \circ \mu) \leq r \tag{5.1.3}$$

$$\Leftrightarrow \mu^{-1} \circ \pi \in \mathbb{B}_r(\mathbf{1}). \tag{5.1.4}$$

□

5.2. Enumerating all permutations within a given ball

One trivial algorithm enumerates all $n!$ permutations and tests them for being a solution or not. It is presented in Section 3.2. We suppose an algorithm that always checks less permutations than the trivial algorithm, while retaining the property of the trivial algorithm of having little costs per iteration. We prove the run time bound $O\left(mn \log n \cdot \min\left\{r(2r)^n, rn^{2r}, n!\right\}\right)$, which proves the algorithm to be coequal with the trivial algorithm in terms of run time, and surpass it for small values of r. The algorithm surpasses the search tree algorithms for small values of n.

The definition of the CENTER RANKING problem in 1.4.3 implies that the set of solutions of an instance is the set $\bigcap_{v \in V} \mathbb{B}_r(v)$. The first algorithm enumerates $\mathbb{B}_r(\mu)$ for any $\mu \in V$ and checks for each element whether it qualifies as a solution. To do so, we learn to enumerate all permutations within $\mathbb{B}_r(\mu)$ for permutation μ and radius r. It is easier to enumerate $\mathbb{B}_r(\mathbf{1})$, where $\mathbf{1}$ refers to the identity permutation. We can find the enumeration of $\mathbb{B}_r(\mu)$ by translocating the enumeration of $\mathbb{B}_r(\mathbf{1})$.

To enumerate the ball around the identity permutation, we introduce a notation of permutations we call *inversion vectors.* We enumerate the inversion vectors of permutations around the identity permutation, obtain their sequence notations and then translocate them to obtain an enumeration of $\mathbb{B}_r(\mu)$. The inversion vector can be obtained from a permutation by replacing every element in the sequence notation of the permutation by the number of inversions that element has with elements to its right. For example, the permutation 45231 would be denoted as $(3, 3, 1, 1)$.

Since only $n - i$ positions follow after position i, it is clear that entry i of the inversion vector needs be smaller than $n - i$. In this inversion vector notation, it is easy to count inversions. The number of inversions is the sum of the entries of the inversion vector.

Let $(a_1, a_2, \ldots, a_{n-1})$ be a inversion vector with $a_i \leq n - i \ \forall i$. Then there can be only one permutation π whose inversion vector would be $(a_1, a_2, \ldots, a_{n-1})$, as we will prove in the following lemma. The proof also shows how to translate between the usual sequence notation and our vector notation.

Lemma 74. *Let $(a_1, a_2, \ldots, a_{n-1}), n \in \mathbb{N}$ be a inversion vector with $a_i \leq n - i \ \forall i$. Then there can be at most one permutation π whose inversion vector is $(a_1, a_2, \ldots, a_{n-1})$.*

Proof. Let $\pi = p_1 p_2 \ldots p_n$ and $\rho = q_1 q_2 \ldots q_n$ denote two permutations out of $\mathrm{Sym}_{[n]}, n \in \mathbb{N}$, whose inversion vectors both equal $(a_1, a_2, \ldots, a_{n-1})$. We prove $\pi = \rho$.

For any number $b \in [n]$ to go in the first position, we already know how many entries it must precede that are smaller than itself. That number would be $b-1$. We deduce $p_1 = q_1$, since

$$p_1 = a_1 + 1 = q_1. \tag{5.2.1}$$

For the next position i, let us for the sake of induction assume that all positions $p_j = q_j, j < i$ have been set already. We know that p_i and q_i precede exactly a_i elements that are smaller than they are. Thus, the $a_i + 1^{\text{st}}$ smallest element out of the set of entries that have not been placed yet must equal p_i and q_i. By induction, we find that $\pi = \rho$. □

Careful consideration of the above proof shows precisely how to obtain the sequence notation of the permutation described by an inversion vector. We reformulate the method into an algorithm, see Algorithm 10. We will refer to the function that maps between inversion vectors and sequence notation as `P`.

Algorithm 10 Alogarithm of the `P` function, which obtains the permutation described by an inversion vector, as described in the proof of Lemma 74.

`Input:` $(a_1, a_2, \ldots, a_{n-1})$`, the inversion vector.`
`Output:` $p_1 p_2 \ldots p_n \coloneqq$ `P` $((a_1, \ldots, a_{n-1}))$`, the permutation associated with inversion vector` $(a_1, a_2, \ldots, a_{n-1})$ `in sequence notation.`
`P1: [Init] Set` $i \coloneqq 1$.
`P2: [Find candidates for` p_i`] Set` $S \coloneqq \{s \in [n] : s \neq p_j \forall j < i\}$.
`P3: [Set` p_i`] Choose the` $a_i + 1^{\text{st}}$ `smallest element from` S `and declare it to be` p_i.
`P4: [Loop] If` $i = n$`, return` $p_1 \ldots p_n$`, otherwise set` $i \coloneqq i + 1$ `and return to P2.`

We enumerate the inversion vectors of $\mathbb{B}_r(\mathbf{1})$. Enumerating inversion vectors, i.e. vectors $(a_1, \ldots, a_{n-1})$ with $\sum_i a_i \leq r$ and $a_i \leq n - i \ \forall i$ can be done very similarly to enumerating all numbers in a certain range, only that each position has its own limit after which it is overflowed. We describe how to enumerate the inversion vectors by giving the immediate next vector after `NEXT`$((a_1, \ldots, a_{n-1}))$ in Algorithm 11.

Algorithm 11 Algorithm for the NEXT function, finding the next inversion vector when enumerating $\mathbb{B}_r(\mathbf{1})$.

`Input: An inversion vector` $(a_1, \ldots, a_{n-1}) \in [n]^{n-1}$ `with` $a_i \leq n - i \ \forall i$`; radius` $r \in \mathbb{N}$.
`Output:` $(b_1, \ldots, b_{n-1}) \coloneqq$ `NEXT`$((a_1, \ldots, a_{n-1}))$
`Initialize:` $(b_1, \ldots, b_{n-1}) \coloneqq (a_1, \ldots, a_{n-1})$
`FOR` i `FROM` $n-1$ `DOWNTO 1 DO BEGIN`
    `(*) IF` $a_i \leq n - i$ `AND` $\sum_{j \leq n} a_j < r$ `THEN BEGIN`
        $b_i \coloneqq a_i + 1$;
        `RETURN` $(b_1, \ldots, b_{n-1})$
    `END ELSE BEGIN`
        `(**)` $b_i \coloneqq 0$
    `END`
`END;`
`RETURN ‘‘None’’`

To understand the behaviour of Algorithm 11 better, let us examine the overflowing vectors it enumerates for $n = 3$ and $r = 2$.

Example. For $n = 3$ and $r = 2$, Algorithm 11 outputs vectors o_1 through o_9, where

$$o_1 = (0, 0, 0)$$
$$o_2 = (0, 0, 1)$$
$$o_3 = (0, 1, 0)$$
$$o_4 = (0, 1, 1)$$
$$o_5 = (0, 2, 0)$$
$$o_6 = (1, 0, 0)$$
$$o_7 = (1, 0, 1)$$
$$o_8 = (1, 1, 0)$$
$$o_9 = (2, 0, 0).$$

Lemma 75. *Let* $\mathbf{1} \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. *Algorithm 11 enumerates the inversion vectors of* $\mathbb{B}_r(\mathbf{1}), r \in \mathbb{N}$.

PROOF. For each $\pi \in \mathbb{B}_r(\mathbf{1})$, the inversion vector of π can be obtained by a finite number of executions of Algorithm 11 with the result of the previous invocation as input. The first input should be $(0, \ldots, 0)$. We merely outline the proof by stating that the inversion vectors are enumerated exactly as numbers in digit representation are, only that the overflow occurs at different limits for each position. □

The algorithm we present enumerates $\mathbb{B}_r(\mu)$ using Algorithm 11 and Algorithm 10, then checks for each member of $\mathbb{B}_r(\mu)$ whether it is a solution. See Algorithm 12 for the pseudo-code of the procedure.

Theorem 76. *Let* $V \in \mathrm{Sym}^m_{[n]}, n \in \mathbb{N}$, *an instance of the* CENTER RANKING *problem. Algorithm 12 correctly solves* V.

PROOF. Due to the definition of the CENTER RANKING problem, for μ as chosen by the algorithm, the solution needs be within $\mathbb{B}_r(\mu)$, if it exists. The $(a_1, \ldots, a_{n-1})$ vector in the repeat loop enumerates the inversion vectors of $\mathbb{B}_r(\mathbf{1})$ due to Lemma 75. Due to the correctness of Algorithm 10 and Corollary 73, π enumerates $\mathbb{B}_r(\mu)$. If a solution exists, the algorithm enumerates and checks it. Otherwise, "None" is returned correctly. □

In summary, to enumerate all permutations, we generate all integer vectors of length $n-1$ with $0 \leq a_i \leq n-i \ \forall i$ with $a_1 + a_2 + \cdots + a_{n-1} = r$, we retrieve their sequence notations, translocate them using μ and then test for each enumerated permutation whether it is a solution. The algorithm enumerates $\mathbb{B}_r(\mu)$ which is of the same size as $\mathbb{B}_r(\mathbf{1})$. Therefore, the algorithm's run time depends on the size of $\mathbb{B}_r(\mathbf{1})$. We therefore investigate the asymptotic size of $\#\mathbb{B}_r(\mathbf{1})$.

The size of $\mathbb{B}_k(\mathbf{1})$ can be expressed exactly using Tahonian numbers[1]. The Tahonian number $b(n, k)$ is the number of permutations of n entries and k inversions. The Tahonian numbers can be described by a simple recurrence equation for $k \leq n$ as follows:

$$b(n+1, k) = b(n+1, k-1) + b(n, k).$$

The Tahonian numbers $b(n, k)$ are defined to be the number of permutations of length n with exactly k inversions. We have

[1]Named after MacMahon, see Bona [10, p. 50].

Algorithm 12 Algorithm that enumerates all permutations of a given distance r around permutation μ. It uses function `P` which translates between the inversion vector and the sequence representation of a permutation, see Algorithm 10. It uses function `NEXT` which enumerates inversion vectors, see Algorithm 11. The run time is $O(mn \log n \cdot r(2r)^n)$.

```
Input: An instance (V, r) of CENTER RANKING of
    votes ∅ ≠ V ⊂ Sym_[n] and radius r ∈ ℕ
Output: A solution π, i.e.  π ∈ ⋂_{μ∈V} 𝔹_r(μ)
    if one exists, ‘‘None’’ otherwise.

[Initialize vector (a_1, ..., a_{n-1})]
a_1 := ... := a_{n-1} := 0.

[Choose μ to be any element of V]
μ := V[1];

[Loop]
REPEAT (a_1, ..., a_{n-1}) := NEXT((a_1, ..., a_{n-1}), r)
    π := μ^{-1} ∘ P(NEXT((a_1, ..., a_{n-1})));
    IF τ(π, μ) ≤ r' ∀μ ∈ V THEN RETURN π
UNTIL (a_1, ..., a_{n-1}) = "None";

RETURN ‘‘None’’
```

$$\#\mathbb{B}_r(\mathbf{1}) = \#\left(\bigcup_{k'}\{\lambda \in \mathrm{Sym}_{[n]} : \mathrm{inv}(\lambda) = k'\}\right) \tag{5.2.2}$$

$$= \sum_{k'} \#\{\lambda \in \mathrm{Sym}_{[n]} : \mathrm{inv}(\lambda) = k'\} \tag{5.2.3}$$

$$= \sum_{k'} b(n, k'). \tag{5.2.4}$$

We upper-bound the number of permutations of n candidates and k inversions as follows.

Lemma 77. *Let $n, k \in \mathbb{N}$, then Tahonian number*

$$b(n, k) \leq (2k + 2)^n.$$

Proof. The inversion vectors of a permutations of exactly k inversion needs to have an entry out of $[1, k+1]$ in the first position of the solution sequence, because if the entry were bigger than $k + 1$, more than k smaller elements would follow, and hence it would have more than k inversions. Similarly, we know that for every position i, the element of the solution string must be chosen out of $[i-k-1, i+k+1]$. We can conclude that for every position, out of which n exist, we can choose between $2k+2$ elements. Concluding, there cannot be more than $(2k+2)^n$ Tahonian permutations of length n with exactly k inversions. □

We can give yet another bound. We bound the number of permutations of n candidates and k inversions.

Lemma 78. *Let $n, k \in \mathbb{N}$, then Tahonian number*

$$b(n, k) \leq \binom{n}{2}^k = O(n^{2k}).$$

PROOF. By definition $b(n,k)$ is the number of permutations with k inversions. This is of course smaller than the number of sets of r unordered pairs of k entries which is $\binom{n}{2}^k$. □

We are now able to prove a bound on the number of permutations of Kendall-τ distance at most r from the identity permutation $\#\mathbb{B}_r(\mathbf{1})$.

Theorem 79. *Let* $\mathbf{1} \in \mathrm{Sym}_{[n]}, n, r \in \mathbb{N}$*, then*

$$\#\mathbb{B}_r(\mathbf{1}) = O\left(\min\{r(2r)^n, rn^{2r}, n!\}\right).$$

PROOF. Of course, $\#\mathbb{B}_r(\mathbf{1}) \leq n!$ is trivial, since $\mathbb{B}_r(\mathbf{1}) \subset \mathrm{Sym}_{[n]}$. The claim $\#\mathbb{B}_r(\mathbf{1}) = O\left(r(2r)^n\right)$ is a consequence of equations (5.2.3) and Lemma 77. We can establish it by

$$\#\mathbb{B}_r(\mathbf{1}) \leq \sum_{k'=1}^{r} b(n,k') \tag{5.2.5}$$

$$\leq \sum_{k'=1}^{r} (2k'+2)^n \tag{5.2.6}$$

$$\leq r(2r+2)^n. \tag{5.2.7}$$

Finally, $\#\mathbb{B}_r(\mathbf{1}) = O\left(rn^{2r}\right)$ can be established similarly using Lemma 78.

$$\#\mathbb{B}_r(\mathbf{1}) \leq \sum_{k'=1}^{r} b(n,k') \tag{5.2.8}$$

$$\leq \sum_{k'=1}^{r} \binom{n}{2}^{k'} \tag{5.2.9}$$

$$\leq \frac{1}{2} rn^{2r}. \tag{5.2.10}$$

□

Theorem 79 tells us the run time of Algorithm 12 on the preceding page which solved the CENTER RANKING problem by trying only permutations out of a ball of radius r. The following was claimed at the start of the chapter.

Theorem 80. *Let* (V,r) *be an instance of the decision* CENTER RANKING *decision problem,* $r \in \mathbb{N}$*, and* $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$*. Algorithm 12 solves* (V,r) *in*

$$O\left(mn \log n \cdot \min\left\{r(2r)^n, rn^{2r}, n!\right\}\right)$$

time.

PROOF. Let us first note that Algorithm 11 runs in time $O(n)$. To see it, let us realize that the loop is executed n times. The sum in line `(*)` needs to be computed completely only once. In every later loop, we only need to update the previous computation by subtracting a_i from the previously computed value every time line `(**)` is executed.

Algorithm 10 runs in time $O(n \log n)$. We can store set S as a binary search tree, by which we mean a tree where for every node the left subtree of a node contains only values less than the node's value and the right subtree of a node contains only values greater than the node's value. It is important to create the search tree such that it has depth of at most $\log n$, which is the case for a complete binary tree. We then attach the number of descendants to every node of the search tree. The algorithm needs to find and delete the $a_i + 1^{\text{st}}$ smallest entry of the tree in instruction `P3`. We can find the $a_i + 1^{\text{st}}$ smallest entry by navigating through the tree according to the information attached to every node. We need to keep the

information up to date upon deletions, which requires $O(\log n)$ time upon every deletion. Each find-and-delete step can thus be executed in $O(\log n)$ time. Since the loop `P4` is executed n times, Algorithm 10 runs in time $O(n \log n)$ using a binary search tree. Note that it is not necessary to re-balance the tree after the deletions.

Algorithm 12 calls Algorithm 11 and then Algorithm 10 once for every permutation $\nu \in \mathbb{B}_r(\mu)$, precisely to obtain $\nu \in \mathbb{B}_r(\mu)$ in sequence notation, which takes $O(n \log n)$ time. It then proceeds to verify if ν is a solution, which requires computing the Kendall-τ distance m times, requiring $O(mn \log n)$ time in total, see Section 1.4.8. The distance is computed once for each $\nu \in \mathbb{B}_r(\mu)$.

Therefore, Algorithm 12 runs in time $O(mn \log n \cdot \#\mathbb{B}_r(\mu))$. We can deduce by Theorem 79 and Corollary 73 that the run time needs to be within

$$O(mn \log n \cdot \mathbb{B}_r(\mu)) = O(mn \log n \cdot \#\mathbb{B}_r(\mathbf{1})) \tag{5.2.11}$$

$$= O\left(mn \log n \cdot \min\left\{r(2r)^n, rn^{2r}, n!\right\}\right). \tag{5.2.12}$$

This completes the proof. □

5.2.1. Lower bound. The bounds on the ball size $\#\mathbb{B}_r(\mathbf{1})$ appear rather rough in many respects. In particular, we might have hopes that the size of ball $\#\mathbb{B}_r(\mathbf{1})$ might be only polynomial in n or r if we just find a more accurate proof on the bounds. That is, however, not the case.

Lemma 81. *Let $n, k \in \mathbb{N}$, $n \geq 2$, and let n be even. Then Tahonian number*

$$b(n, k) \geq \binom{n/2}{k}.$$

Proof. We count permutations of k inversions of the form $(2i-1, 2i)$. For example, for $n = 3$, and $k = 2$, these permutations are 124365, 213465, 214356.

In a permutation of length $2n$, exactly n such inversions can occur and they are completely independent. For $k \leq 2n$, we can choose k of the inversions. We have $\binom{n}{k}$ possibilities to do so, which all lead to different permutations. □

The implication for the ball size $\#\mathbb{B}_r(\mathbf{1})$ would be the following lemma.

Lemma 82. *Let $\mathbf{1} \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$ denote the identity permutation, let $k \in \mathbb{N}, r \leq \binom{n}{2}$. Then*

$$\#\mathbb{B}_r(\mathbf{1}) = \Omega((\frac{n}{2r})^r).$$

Proof. We have

$$\mathbb{B}_r(\mathbf{1}) = \sum_{k' \leq r} b(n, k') \tag{5.2.13}$$

$$\geq b(n, r) \tag{5.2.14}$$

$$\geq \binom{n/2}{r} \tag{5.2.15}$$

$$= \prod_{i \leq r} \frac{n/2 + 1 - i}{r + 1 - i} \tag{5.2.16}$$

$$\geq \prod_{i \leq r} \frac{n/2}{r} \tag{5.2.17}$$

$$= (\frac{n}{2r})^r \tag{5.2.18}$$

Inequality (5.2.17) holds since $n/r < \frac{(n-i)+1}{(r-i)+1}$ for $i > 0$. □

Looking at merely one input vote does not bring us fixed-parameter tractability with respect to either parameter radius r or candidates n.

Theorem 83. *Let p be a polynomial and f be any function. Let $\mathbf{1} \in \mathrm{Sym}_{[n]}, n, k \in \mathbb{N}$. Then*

$$\#\mathbb{B}_k(\mathbf{1}) \notin O(p(n)f(k)).$$

Proof. If $\#\mathbb{B}_k(\mathbf{1}_{[n]}) = O(p(n)f(k))$ for a polynomial p and some function f, then due to Corollary 82

$$\begin{aligned}\#\mathbb{B}_k(\mathbf{1}) \in O(p(n)f(k)) &\Rightarrow \lim_{n\to\infty} \frac{(\frac{n}{2k})^k}{(\frac{n}{2k'})^{k'}} < \infty \ \forall k, k' \\ &\Rightarrow \lim_{n\to\infty} \frac{n^2/4}{n/2} < \infty \text{ for } k = 2 \text{ and } k' = 1 \\ &\Rightarrow \lim_{n\to\infty} n/2 < \infty.\end{aligned}$$

Which is false. Thus $\#\mathbb{B}_k(\mathbf{1}) \notin O(p(n)f(k))$, completing the proof. □

Algorithm 12 provides an easy to implement algorithm which does relatively little work per tested permutation and provably tries less permutations than the trivial algorithm which tries all permutations. For small values of n, it can outperform the search tree algorithms from Chapter 4.

5.3. Enumerating the intersection of two balls

In the Euclidean plane, two balls of sufficient distance have a rather small intersection, as depicted in Figure 5.3.1. While the intuition the Euclidean plane gives us might be encouraging, it is also misleading. The metric space of permutations with Kendall-τ distance is very much unlike the Euclidean plane. For instance, there are numerous shortest paths between two permutations.

The algorithm presented in this section is proved to run in $O(n^r \cdot rmn \log n)$ time. The algorithm is especially good when the input contains two votes of Kendall-τ distance at least $2r - 1$. In that case, the size of the intersection is bounded by 2^{2r}. For distances greater than $2r$, the instance can trivially be identified as a no-instance, see Data Reduction Rule 28 on page 22.

For an instance (V, r) of the CENTER RANKING decision problem, we know that the solution must be within range r from every input permutation. Therefore, if we choose any two input permutations, the solution must be within $\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mu)$. To keep the intersection small, we will choose λ and μ such that $\tau(\lambda, \mu) > r$. We can do so, because if no two such two input permutations exist, Data Reduction Rule 28 on page 22 can decide the instance.

We present an algorithm that enumerates the intersection of two balls in the metric space of permutations. We will choose the balls such that a solution must be enumerated if it exists. We study the sizes of the intersections rigorously, showing both upper and lower bounds for the size of the intersection.

We translocate the two balls we want to enumerate such that the center of one of them is the identity permutation. The following proposition is an easy consequence of Lemma 10 on page 11.

Proposition 84. *Let $\lambda, \mu \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$ be two permutations and $r \in \mathbb{N}$, then*

$$\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mu) = \lambda^{-1}(\mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\lambda^{-1} \circ \mu)).$$

We present an algorithm that enumerates the intersection $(\mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\lambda^{-1} \circ \mu)$. The general idea of the algorithm enumerating the ball intersection is to build up the permutations from left to right. We start by setting the first digit of the permutation. We check if any permutation within the intersection starts with that

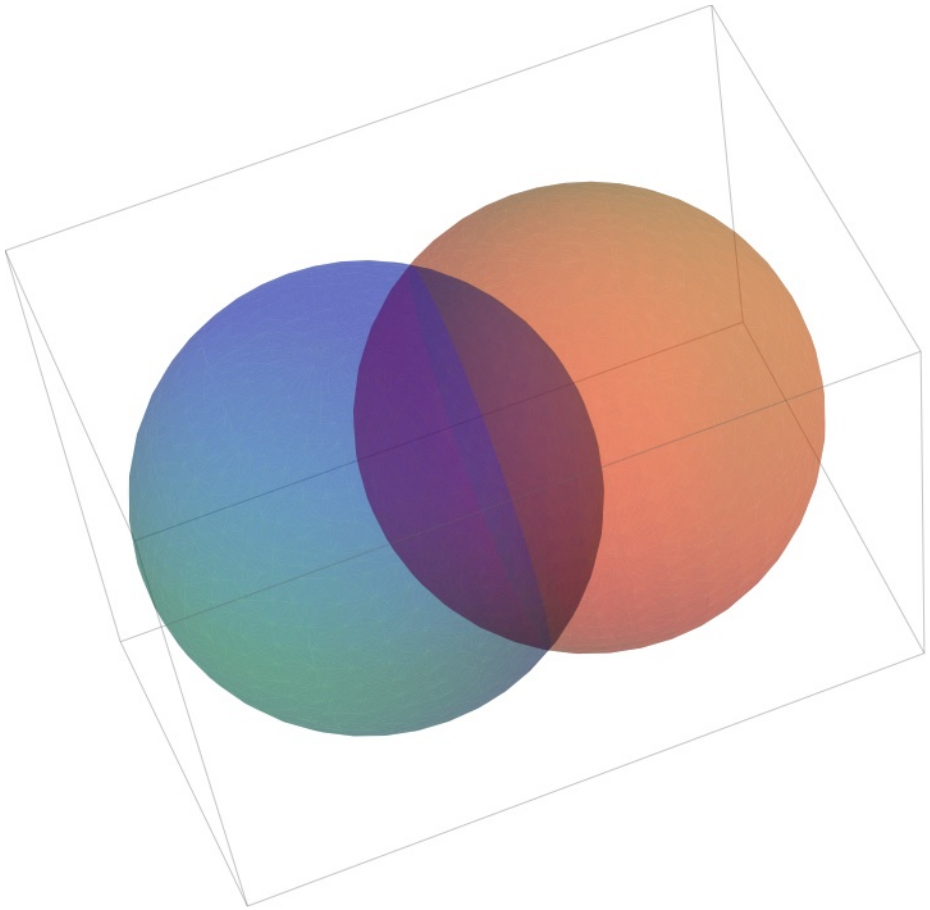

Figure 5.3.1: Intersection of two balls in Euclidean space.

digit. We add the next digit and check whether there is a permutation starting with the starting sequence we have chosen so far. We repeat the process, each time choosing a new digit, making sure that after each time a digit is added, the new starting sequence can be completed to a permutations within the intersection. The algorithm enumerates the intersection by trying all ways of adding a new digit.

To understand how the algorithm enumerates $\mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\lambda^{-1} \circ \mu)$, it is decisive to understand that computing the Kendall-τ distance and inversion counting are closely related, since the algorithm computes all Kendall-τ distances by counting the number of inversions. One advantage of counting inversions is that it is fairly easy to tell how many inversions the first entry of a permutation participates in, in case the set of entries of the permutation is $[n]$. The first entry b of a permutation must participate in $b-1$ inversions, since exactly $b-1$ entries smaller than itself are listed to its right. See [10, p. 50] for an explanation of this way of counting inversions.

The algorithm keeps count on the distance left to both ball centers in every recursive call, which is decreased by the inversions of the newly set digit with entries to its right. The observation that the first digit b of a permutation participates in $b-1$ inversions holds only if the set of entries of the permutation equals $\{1, \ldots, k\}$ for some $k \in \mathbb{N}$. That is why once the algorithm chooses a first position, it changes the remaining entries so that in the recursive call the set of entries is $\{1, \ldots, k\}$.

For a newly determined entry b, the distance to $\mathbf{1}$ is increased by $b-1$, which is the number of entries smaller than itself to its right. Similarly, we can increase the distance to λ by $\lambda^{-1}(b)-1$. See Algorithm 13 for the pseudo-code of the operation.

Algorithm 13 Enumerates the intersection of two balls in permutation space. The ball centers have distance greater than r but smaller or equal $2r$ and both radii are r.

Input: Ball center λ, radius r,
discordance counters i_λ, and $i_{\mathbf{1}}$.
Inequality $r < \tau(\lambda, \mathbf{1}) \leq 2r$ must hold.
Output: $\mathbb{B}_{r-i_\lambda}(\lambda) \cap \mathbb{B}_{r-i_{\mathbf{1}}}(\mathbf{1})$.
J1: Initialize the answer set as $A \coloneqq \emptyset$.
J2: Initialize D to be the set
$\{d : d-1+\lambda^{-1}(d)-1+\tau(\lambda|_{[n]-\{d\}}, \mathbf{1}) \leq$
$2r - i_\lambda - i_{\mathbf{1}} \wedge d-1 \leq i_{\mathbf{1}} \wedge \lambda^{-1}(d)-1 \leq i_\lambda\}$
J3: Let $d \in D$ be any element from D.
Remove d from D.
J4: Update i_λ and $i_{\mathbf{1}}$:

$$\begin{aligned} i_{\lambda,d} &:= i_\lambda + \lambda^{-1}(d) - 1 \\ i_{\mathbf{1},d} &:= i_{\mathbf{1}} + d - 1 \end{aligned}$$

J5: Run this algorithm with input $\rho(\lambda|_{[n]-\{d\}})$ and
discordance counters $i_{\lambda,d}$ and $i_{\mathbf{1},d}$,
and store the answer in B.
Here, function ρ lowers all entries in
$\lambda_{[n]-\{d\}}$ that are greater than d by one.
J6: Let ρ^{-1} denote the function that undoes
the effect of ρ from J5.
Prepend d to each entry of $\rho^{-1}(B)$ and
add the resulting set to A.
J7: If D is not empty, return to J3.
J8: Return A.

Note that J5 makes sure that the set of candidates in recursive calls is $\{1, \ldots, k\}$. For example, if $\lambda = 45321$ and $d = 3$, then $\rho(\lambda|_{[n]-\{d\}}) = 3421$ is λ after striking out d and lowering all entries greater than d by one.

The complete algorithm merely enumerates the intersection and checks for each member whether it qualifies as a solution. See Algorithm 14 on the next page.

Lemma 85. *Let $\lambda \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$. Algorithm 13 correctly enumerates $\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$.*

Proof. The algorithm builds up the permutation in sequence notation from left to right. We call the first entries in the sequence notation of a permutation the permutation's *starting sequence*. At any point, the algorithm already knows a starting sequence s, which consists of all starting digits chosen by previous recursive calls, and decides for each digit d if there is a permutation within $\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$ with starting sequence sd. If there is, the algorithm branches into the new starting sequence, otherwise it abandons the digit. We prove the correctness of the algorithm by induction on the length of the starting sequence l.

Clearly, unless $\tau(\mathbf{1}, \lambda) > 2r$, which we excluded, there is a permutation with the empty starting sequence. Assume for the sake of induction that we already know

Algorithm 14 Solves an instance of CENTER RANKING using Algorithm 13 on the facing page.

```
Input: An instance (V, r) of the CENTER RANKING problem.
Output: σ such that max_{λ∈V} τ(σ, λ) ≤ r,
    "None" if there is none.
I1: From the input, verify that
    ∀λ, μ ∈ V : τ(λ, μ) ≤ 2r.
    Otherwise, answer "None."
I2: Choose any two λ1, λ2 ∈ V with τ(λ1, λ2) > r.
    If no such two exist,
    any λ1 ∈ V is a solution.
I3: Collect  B_k(1) ∩ B_k(λ^{-1} ∘ μ)
    into a set A using Algorithm 13.
    As Arguments, pass it the ball center
    λ^{-1} ∘ μ, radius r and i_λ = i_1 = 0.
I4: Replace A with {λ^{-1} ∘ α : α ∈ A}.
I5: Remove any a ∈ A from A.
I6: Check if a is a solution. If it is, return a.
I7: If A ≠ ∅, return to I5.
I8: Return "None".
```

if for a starting sequences of length l if a permutation exists. Now given a starting sequence s of length l for which there is a permutation with that starting sequence, for $d \in [n]$ we need to decide whether or not a permutation $\pi \in \mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$ exists with starting sequence sd, where sd is the concatenation of s and d. For example, for $s = 432$ and $d = 1$, we have $sd = 4321$.

First, we show that after choosing $d \in D$ in instruction J2, there exists a permutation with starting sequence sd in $\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$, which we denote as $sd\pi$, where sd is the starting sequence and π denotes all symbols to the right of the start sequence. We refer to π as the tail of the permutation. Let $\lambda|_C$ denote permutation λ after striking out all entries except those present in C. Let $C = [n]$ denote the set of candidates and let $C - sd$ denote the symbols out of C that are not part of the start sequence sd.

We present precise instructions how to choose π such that $sd\pi \in \mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$. We choose the π in the "middle" between permutations $\lambda|_{C-sd}$ and $\mathbf{1}|_{C-sd}$. A permutation can be turned into any other through repeated swapping of adjacent entries and the number of swaps needed equals the Kendall-τ distance. We start on permutation $\mathbf{1}|_{C-sd}$ and repeatedly swap adjacent entries that are discordant with $\lambda|_{C-sd}$. Let $r' \coloneqq \tau(\lambda, \mathbf{1})$. Let i_λ be the cardinality of the subset of the set of discordances $\{b, c\}$ between $sd\pi$ and λ, where either b or c occurs in s. Note that i_λ is independent from π. Further, $i_{\mathbf{1}}$ is the cardinality of the analog subset of the set of discordances between $sd\pi$ and $\mathbf{1}$. Note that both i_λ and $i_{\mathbf{1}}$ are part of the input to Algorithm 13, but we will show that they are passed as arguments such that the above holds. Figuratively, i_λ is the distance from s to λ and $i_{\mathbf{1}}$ is the distance from s to $\mathbf{1}$.

After q of the swaps that turn $\mathbf{1}$ into λ, we obtain a permutation π for which the following holds:

$$\tau(\pi, \mathbf{1}|_{C-sd}) = q \wedge \tau(\pi, \lambda|_{C-sd}) = \tau(\lambda|_{C-sd}, \mathbf{1}|_{C-sd}) - q = r' - i_\lambda - i_{\mathbf{1}} - q. \tag{5.3.1}$$

We set q to be any element of $I \coloneqq [r' - r + i_{\lambda,d}, r - i_{\mathbf{1},d}]$, where

$$i_{\lambda,d} \coloneqq i_\lambda + \lambda^{-1}(d) - 1$$

and

$$i_{\mathbf{1},d} \coloneqq i_{\mathbf{1}} + d - 1,$$

as defined in instruction J4. Before every recursive call, instruction J5 ensures that the set of candidates in the recursive call is the set $[n-1]$—preventing the occurrence of gaps, and thus allowing us to compute the number of inversions in $d\pi$ that d participates in, as $d-1$ on every level of the recursion, because in π, exactly $d-1$ entries that are smaller than d must occur. Therefore, for any $sd\pi \in \mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$, discordance counter $i_{\lambda,d}$ is the cardinality of the subset of the set of discordances $\{b, c\}$ between $sd\pi$ and λ, where either b or c occurs in sd, and $i_{\mathbf{1},d}$ is the analog discordance counter for $\mathbf{1}$ instead of λ. We have shown that the inversion counters are passed, such that they describe the discordances of the starting sequence with $\mathbf{1}$ and λ, respectively.

We show that $sd\pi \in \mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$. Remember that r denotes the radius of the instance. It is easy to verify that the first condition in J2, which is the following

$$d - 1 + \lambda^{-1}(d) - 1 + i(\lambda|_{[n]-\{d\}}) \leq 2r - i_\lambda - i_{\mathbf{1}} \tag{5.3.2}$$

holds if and only if I is not empty. The other two conditions in J2 hold if and only if $i_{\lambda,d}, i_{\mathbf{1},d} \geq 0$. Then, $q \in I$ together with equation (5.3.1) imply that for π

$$\begin{aligned} \tau(\pi, \mathbf{1}|_{C-sd}) &= q \leq r - i_{\mathbf{1},d} \\ \text{and } \tau(\pi, \lambda|_{C-sd}) &= r' - q \geq r' - r + i_{\lambda,d}, \end{aligned} \tag{5.3.3}$$

and therefore

$$\tau(\pi, \lambda|_{C-sd}) \leq r - i_{\lambda,d}. \tag{5.3.4}$$

As for the distance from π to λ, we then have

$$\begin{aligned} i_\lambda + \lambda^{-1}(d) - 1 = r - i_{\lambda,d} \geq \tau(\pi, \lambda|_{C-sd}) &= \tau(d\pi, \lambda|_{C-s}) + \lambda^{-1}(d) - 1 \\ &\Rightarrow \tau(d\pi, \lambda|_{C-s}) \leq r - i_\lambda \\ &\Rightarrow \tau(sd\pi, \lambda) \leq r. \end{aligned} \tag{5.3.5}$$

Analogously, we deduce $\tau(\pi, \mathbf{1}) \leq r$. We have shown that $sd\pi \in \mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$.

Let us show the converse, which is that if a permutation $sd\nu \in \mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1})$ exists, then $d \in D$ as defined in instruction J2, when the prefix set is s. We can choose a q so that we can obtain ν by subsequently swapping q adjacent pairs of candidates in $\mathbf{1}|_{C-sd}$ that are discordant with $\lambda|_{C-sd}$. For this q, we can trace back all the above computation and find that $d \in D$ as defined in J2.

We have proved that the digit is correctly chosen. We conclude by induction, that the rest of π will be chosen correctly in the remaining recursive calls, since in each of them the starting sequence will be longer than l. Once all digits have been correctly chosen, all permutations have been correctly enumerated, completing the proof. □

Corollary 86. *Let $(V, r), V \in \mathrm{Sym}^m_{[n]}, n, m, r \in \mathbb{N}$ be an instance of the* CENTER RANKING *decision problem. Algorithm 14 correctly solves V.*

PROOF. Due to the definition of the CENTER RANKING problem, if a solution exists, it is in $\mathbb{B}_r(\mu) \cap \mathbb{B}_r(\lambda)$ for μ, λ as chosen by the algorithm. Note that if λ, μ cannot be picked, then the algorithm correctly solves the instance due to Data Reduction Rule 28 on page 22.

Algorithm 13 correctly enumerate $\mathbb{B}_r(\mu) \cap \mathbb{B}_r(\lambda)$ due to Lemma 85 and Corollary 73 on page 62. For every element in the intersection, it is tested whether it solves the instance, and if so, it is returned. Otherwise "Not found" is returned. Since the solution, if one exists,is in the enumerated intersection, the algorithm works correctly. □

The run time of Algorithm 13 depends on the number of permutations in the intersection $(\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1}))$. We upper-bound the size of $\#\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mathbf{1}))$ under the condition $\tau(\mathbf{1}, \mu) > r$, which is met by the algorithm. We show the following.

Lemma 87. *Let $\mu, \mathbf{1} \in \mathrm{Sym}_{[n]}, n, r \in \mathbb{N}$, where $\mathbf{1}$ is the identity permutation and $\tau(\mathbf{1}, \mu) > r$, then $\#(\mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\mu)) = O(rn^r + 2^r)$.*

Proof. Let Q denote all "non-inversions", i.e. all unordered pairs $\{a, b\}$ with $a < b$ and $a <_\mu b$. Let P denote all inversions, i.e. all unordered pairs $\{a, b\}$ with $a < b$ and $a >_\mu b$. We know that $\lambda \in \mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\mu)$ may contain at most r inversions. For each pair of λ of candidates a and b, either $\{a, b\} \in Q$ or $\{a, b\} \in P$. We show that out of the inversions in λ at most $r/2$ inversions can be picked out of Q and at most r can be picked out of P.

Using the restricted Kendall-τ distance notation from Definition 60 on page 48, we can denote

$$\tau = \tau^P + \tau^Q. \tag{5.3.6}$$

For each pair $\{a, b\} \in P$, λ must show a preference, where $\#P$ must be greater than r. Each preference increases the τ^P-distance to either $\mathbf{1}$ or μ by one. The pigeonhole principle now says, that either $\tau^P(\lambda, \mathbf{1}) \geq \frac{r}{2}$ or $\tau^P(\lambda, \mu) \geq \frac{r}{2}$. Without loss of generality, let us assume that

$$\tau^P(\lambda, \mathbf{1}) \geq \frac{r}{2}. \tag{5.3.7}$$

.

Equation (5.3.6) and equation (5.3.7) imply

$$r \geq \tau(\lambda, \mathbf{1}) = \tau^Q(\lambda, \mathbf{1}) + \tau^P(\lambda, \mathbf{1}) \geq \tau^Q(\lambda, \mathbf{1}) + r/2, \tag{5.3.8}$$

and hence $\tau^Q(\lambda, \mathbf{1}) \leq r/2$, which means that at most $r/2$ inversions may be chosen out of Q, which has size $\binom{n}{2} - \#P = O(n^2)$. Naturally, since λ cannot have more than r inversions, only r can be picked out of P.

To choose at most $\frac{r}{2}$ inversions out of Q, we have at most

$$\sum_{j \leq \frac{r}{2}} \binom{\#Q}{j} = O\left(\sum_{j \leq \frac{r}{2}} (\#Q)^j\right) = O\left(\frac{r}{2}(\#Q)^{r/2}\right) = O\left(rn^r\right) \tag{5.3.9}$$

possibilities. To choose r inversions out of P, we can choose any subset of elements in P, out of which $2^{\#P} = O(2^r)$ exist.

Together, we have that

$$\#(\mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\mu)) = O(rn^r + 2^r), \tag{5.3.10}$$

completing the proof. □

Theorem 88. *Let (V, r) be an instance of the* center ranking *problem, where $r \in \mathbb{N}$ and $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$. For $n \geq 2$, Algorithm 14 solves (V, r) in time*

$$O(n^r \cdot rmn \log n).$$

Proof. Algorithm 14 chooses $\lambda, \nu \in V$ and for $\mu \coloneqq \lambda^{-1} \circ \nu$, it uses Algorithm 13 to enumerate the set $\mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\mu)$. Note that τ in instruction `J2` does not need to be recomputed for every recursive call. It can instead be decreased by $d - 1$ in every recursive call. Hence, the time consumed by Algorithm 14 equals the number of permutations checked times the number of time spent on each permutation.

- Instruction `I1` can be run once in time $O(n \log n)$.
- Instruction `I2` can be run once in time $O(mn \log n)$.
- Instruction `I3`, `I4`, and `I5` can be run in time $O(n)$ per permutation enumerated.

- Instruction `I6` can be performed in time $O(mn \log n)$ per permutation enumerated, by computing m Kendall-τ distances in time $n \log n$, see Section 1.4.8.
- Instruction `I7` and `I8` can be executed in $O(1)$.

We have that Algorithm 14 runs in time

$$O(\#(\mathbb{B}_r(\mathbf{1}) \cap \mathbb{B}_r(\mu))mn \log n) = O(n^r \cdot rmn \log n),$$

due to Theorem 87 and 73. □

Algorithm 14 performs especially well in case that $\tau(\lambda, \mu) \geq 2r - 1$, as we see in the following lemma.

Lemma 89. *Let $\lambda, \mu \in \mathrm{Sym}_{[n]}, n \in \mathbb{N}$ with $\tau(\lambda, \mu) \geq 2r - 1$, then*

$$\#(\mathbb{B}_r(\lambda) \cap \mathbb{B}_r(\mu)) \leq 2^{2r}.$$

Proof. In Chapter 4 we noted that permutations can be defined as binary relations, and those in turn as sets of ordered pairs, which we call "preference pairs." Recall that a "discordance" between permutation λ and μ was defined to be a pair $\{a, b\}$, $a, b \in [n], a \neq b$ such that λ and μ list a and b in different order. There are at least $2r-1$ discordances between λ and μ. Let A be the set of discordances between λ and μ. Permutation $\nu \in \mathbb{B}_r(\mu) \cap \mathbb{B}_r(\lambda)$ needs to assign its own preference on each pair of candidates in A. For a pair of candidates in A, no matter which preference we choose, the pair must be in discordance with either λ or μ. By the pigeonhole principle, at least r of ν's preference pairs must be discordant with one one of λ and μ. Let us assume without loss of generality that ν has at least r discordances with λ. There cannot be a discordance between ν and λ that is not present in A already—or otherwise there would be more than r discordances between ν and λ, implying $\nu \notin \mathbb{B}_r(\lambda)$. Hence, ν can be obtained from λ by reversing a subset of size r of A. Since $\#A \leq 2r$, there are at most 2^{2r} subsets of A. There are at most 2^{2r} possibilities to obtain ν. Hence, there can be no more than 2^{2r} permutations in $\mathbb{B}_r(\mu) \cap \mathbb{B}_r(\lambda)$. □

In case that there are two permutations λ and μ of distance greater than or equal to $2r-1$, the intersection between them is of a cardinality which is exponential solely in the radius. We present the run time in case two input permutations are of distance greater than $2r - 1$ in the following corollary.

Corollary 90. *For an instance $(\lambda_1, \ldots, \lambda_m) \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$, of the* CENTER RANKING *problem, Algorithm 14 runs within time $O(2^{2r}mn \log n)$ for radius r.*

Since the run time very strongly improves the farther the two ball centers are, it might be of advantage to change instruction `I2` in Algorithm 14 to choose the two input permutation with the greatest pairwise distance. This would, however, add a one-time run time cost of $O(m^2 n \log n)$.

5.3.1. Lower bounds for Algorithm 13. As we did in Section 5.2, we may ask if we merely failed to prove a better bound, even though one exists. We show that the enumerating the intersection of two balls whose centers are r apart, where r is within $]r, 2r]$, cannot lead to a fixed-parameter algorithm. Specifically, we show that for $n \in \mathbb{N}$, we can give an example of a ball intersection for the CLOSEST STRING problem, which can be translated to understanding that for each $n \in \mathbb{N}$, there are permutations $\lambda, \mu \in \mathrm{Sym}_{[n]}$ such that

$$\#\mathbb{B}_r(\mu) \cap \mathbb{B}_r(\lambda) = \Omega\left(\max\left\{2^{r/2}, \frac{(n/2 - r - 1)^{r/2}}{\frac{r}{2}!}\right\}\right).$$

To show lower bounds for the run time of Algorithm 13, we lower bound the cardinality of the intersection of the balls which is enumerated. We do so by giving a lower bound for a similar intersection in the metric space of strings, with the Hamming distance used as the metric. This is helpful, since we can translate a set of strings to a set of permutations of equal distances, see Corollary 19 on page 18.

We show that the intersections of two balls of radius d in the metric space of strings of equal same length, with the Hamming distance as a metric, is within $\Omega\left(2^{r/2}\right)$. Using the transformation from strings to permutations from Section 2.2.1, we prove that the enumeration cannot lead to a parameterized algorithm with respect to the parameter r.

We find two strings s_1'' and s_2'', such that $\#(B_{d_H,r}(\mathbf{s}_1'') \cap \mathbb{B}_{d_H,r}(\mathbf{s}_2'')) = \Omega\left(2^{r/2}\right)$. Let $s_1'' \coloneqq 0^D$ and $s_2'' \coloneqq 0^{D-r-1}1^{r+1}$, $r > 3$. hence $d_H(s_1'', s_2'') = r + 1$. We show a lower bound for $\#I$, $I \coloneqq B_{d_H,r}(\mathbf{s}_1'') \cap \mathbb{B}_{d_H,r}(\mathbf{s}_2'')$, by listing numerous elements contained in the intersection. They shall have the form $\mathbf{bd}$ where $\mathbf{b}$ is of length $D - r - 1$ and $\mathbf{d}$ is of length $r + 1$. All strings $\mathbf{bd} \in I$ are to share the same $\mathbf{d}$, but differ in the $\mathbf{b}$ part.

We find numerous strings of the form $\mathbf{bd} \in \{0,1\}^D$ with

$$d_H(\mathbf{s}_i'', (\mathbf{b}, \mathbf{d})) \leq r,\ i = 1, 2, \tag{5.3.11}$$

which are all be members of I. We choose $\mathbf{d} \coloneqq 1^{\lfloor \frac{r+1}{2} \rfloor} 0^{\lceil \frac{r+1}{2} \rceil})$. For $\mathbf{b} = \mathbf{0}$, we have

$$d_H(\mathbf{s}_i'', \mathbf{bd}) = \lceil \frac{1}{2} d_H(\mathbf{s}_1'', \mathbf{s}_2'') \rceil,\ i = 1, 2 \tag{5.3.12}$$

We can still change $\mathbf{b}$ while staying within the bounds set by equation (5.3.11). We can choose $\mathbf{b}$ to be any string with

$$\mathbf{b} \in \{0,1\}^{D-r-1}, d_H(\mathbf{b}, 0^{D-r-1}) = \lfloor \frac{r}{2} \rfloor - 1, \tag{5.3.13}$$

because then

$$d_H(\mathbf{bd}, s_i'') \leq \left\lfloor \frac{r}{2} \right\rfloor - 1 + \left\lfloor \frac{r+1}{2} \right\rfloor \leq r.$$

Therefore, $\mathbf{bd}$ for an $\mathbf{b}$ satisfying equation (5.3.13) is within I. To see how many choices we have on $\mathbf{b}$. We can choose $\lfloor \frac{r}{2} \rfloor - 1$ ones to go into any position out of $D - r - 1$ positions. The number of ways to do so is

$$\#(B_{d_H,r}(\mathbf{s}_1'') \cap \mathbb{B}_{d_H,r}(\mathbf{s}_2'')) \geq \binom{D-r-1}{\lfloor \frac{r}{2} \rfloor - 1} \tag{5.3.14}$$

$$= \frac{(D-r-1)!}{(\lfloor \frac{r}{2} \rfloor - 1)!(D-r-1-\lfloor \frac{r}{2} \rfloor + 1)!} \tag{5.3.15}$$

$$= \Omega\left(\frac{(D-r-1)^{r/2}}{\frac{r}{2}!}\right) \tag{5.3.16}$$

For $D \geq 2r$ we can lower-bound the above using a result by Stanica et al. [57, Corollary 2.9], which gives lower bounds on binomial coefficients. We have

$$\#(B_{d_H,r}(\mathbf{s}_1'') \cap \mathbb{B}_{d_H,r}(\mathbf{s}_2'')) = \Omega\left(2^{r/2}\right). \tag{5.3.17}$$

Using the transformation from strings to permutations from Section 2.2.1, equations (5.3.16) and (5.3.17) prove that the intersection algorithm gives a fixed-parameter algorithm neither with respect to candidates nor with parameter radius.

5.3.1.1. *Intersecting three or more balls.* Given an input (S, r) to the CLOSEST STRING problem with $\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3 \in S$ and $d_H(\mathbf{s}_i, \mathbf{s}_j) > r$, if $i \neq j$. We lower-bound $\bigcap_{i \leq 3} \mathbb{B}_{d_H, r}(\mathbf{s}_i)$. Even if 3 strings of pairwise distances greater than r would exist, we would still find examples for $\mathbf{s}_1, \mathbf{s}_2, \mathbf{s}_3$ such that $\#\left(\bigcap_{i \leq 3} \mathbb{B}_{d_H, r}(\mathbf{s}_i)\right) = \Omega\left(\binom{D}{\frac{1}{2}r}\right)$.

Example 91. Let us consider three input strings like these:

$$\mathbf{s}_1 = 0000000000$$
$$\mathbf{s}_2 = 0000001111$$
$$\mathbf{s}_3 = 0000111100$$

We denote the concatenation of two strings $\mathbf{y}$ and $\mathbf{z}$ as $\mathbf{yz}$. Here, $\bigcap_{i \leq 3} \mathbb{B}_{d_H, r}(\mathbf{s}_i)$ would contain all strings of the form $\mathbf{b}001100, \mathbf{b} \in \{0,1\}^4$, with $d_H(\mathbf{b}, 0^4) \leq 1$ are in the intersection.

Generalizing, we can set the following.

We denote the string of q repeated ones as 1^q. Let $q \leq D \in \mathbb{N}$ be even. Then consider

$$\mathbf{s}_1 = 0^D$$
$$\mathbf{s}_2 = 0^{D-q}1^q$$
$$\mathbf{s}_3 = 0^{D-\frac{3}{2}q}1^q 0^{\frac{1}{2}q}$$

Here, we have that $d_H(\mathbf{s}_i, \mathbf{s}_j) = q$, if $i \neq j$. Let us choose radius $r \coloneqq q + 1$. All strings of the form $\mathbf{b}0^{\frac{1}{2}q}1^{\frac{1}{2}q}0^{\frac{1}{2}q}$ with $\mathbf{b} \in \{0,1\}^{D-\frac{3}{2}q}, d_H(\mathbf{b}, 0^{D-\frac{3}{2}q}) \leq r - \frac{1}{2}q$ are in the intersection. The number of such strings $\mathbf{b}$ is within $\Omega\left(\binom{D}{r-\frac{1}{2}q}\right) = \Omega\left(\binom{D}{\frac{1}{2}r}\right)$. We can even say that with a fairly large number of such balls, we would not reach fixed-parameter tractability by merely intersecting a fixed number of balls. We present an example of n input strings with pairwise distances q. The size of the intersection of the balls of radius r cannot be upper-bounded by a function of the form $p(D)f(m)$, where p is a polynomial.

Example 92. Let, us consider the following strings for $i \leq n$:

$\mathbf{s}_1 = 0^D$

$\mathbf{s}_i = \mathbf{a}_i 1^{\frac{1}{2}q}$

where $\mathbf{a}_i = \mathbf{0}1^{\frac{q}{2}}0^{\frac{qi}{2}}$, $\mathbf{a}_i \in \{0,1\}^{D-\frac{1}{2}q}$. Here, $\mathbf{0}$ denotes the string of only zeroes, such that $\mathbf{b}_i$ has length $D - \frac{1}{2}q$.

By choosing$D$ sufficiently great, all those $\mathbf{s}_i$ still have an arbitrarily large number of zeroes on their left. We can lower bound the cardinality of the intersection, $\#\left(\bigcap_{i \leq n} \mathbb{B}_{d_H, r}(\mathbf{s}_i)\right)$, because all strings of the form $\mathbf{b}1^{\frac{1}{2}q}$, $d_H(\mathbf{b}, 0^{D-\frac{1}{2}q}) \leq r - \frac{1}{2}q$ are in that intersection. There would be $\Omega\left(\binom{D-\frac{1}{2}q}{r-\frac{1}{2}q}\right)$ of those, where $r \geq q$, because $\tau(\mathbf{s}_i, \mathbf{s}_j) = q$ if $i \neq j$.

5.3.2. Summary. The technique of enumerating the intersections of balls improve the trivial algorithm. The amount of computation per enumerated permutation is not much higher, but the number of enumerated permutations may be significantly lower. For fixed radius, the algorithm are exponentially better than the trivial algorithm.

Algorithm 14 enumerates the intersection of two balls and checks the members of the intersection for whether they qualify as a solution or not. The algorithm is the fastest known algorithm for solving the CENTER RANKING problem in case that

$n < r \leq 16$. Compared to the search tree algorithm 8 which constructs a search tree of size 16^r, Algorithm 14 is easier to implement and the cost per permutation is a lot smaller. Compared to the search tree algorithm 6 which constructs a search tree of size r^r, Algorithm 14 carries the advantage of never examining the same permutation twice. In case that $n < r$, the size of enumerated permutations is smaller than the search tree sizes of both search tree algorithms.

CHAPTER 6

Practical experiments

Some of the algorithms presented in this thesis have been implemented their run times have been measured for different problem instances. The algorithms compared here are the trivial algorithm, which tries the solution quality of every possible permutation (see Section 3.2), the neighbor permutation search algorithm (see Section 4.2), and an integer linear program implementation.

An integer linear program that solves the CENTER RANKING problem with no proved performance bound is presented in Section 6.1. Section 6.2 contains a comparison of the practical run times of all implemented algorithms. The exact implementations of all algorithms are presented in Appendix A.

6.1. Integer linear program solving THE CENTER RANKING problem

There are various software packages aimed at quickly solving integer programs. In practical scenarios it occurs that the solution of an integer program formulation of a problem has a run time good enough to be of practical importance. Integer programming was used to obtain exact algorithms for the CLOSEST STRING problem without any complexity bounds, but with respectable practical performance [45]. We present an integer program that solves the CENTER RANKING problem. The integer program we present has no guarantees on the run time of the solver. We examine the practical run time in Mathematica 7, which will significantly exceed the time needed to try all permutations.

The formulation of an integer program distinguishes between restrictions and variables. The restrictions act on the variables. They are expressed through constants. The solution of an integer program is the minimum that a variable we pick may achieve while all variables satisfy all restrictions.

The integer program we propose resembles an integer program presented by Ben-Dor et al. [4], which solves the CLOSEST STRING problem. We add restrictions to the variables of the CLOSEST STRING problem that allow us to identify the solution string with a permutations. We will encode each input permutation as a string, too. The integer program is presented in Algorithm 15

Algorithm 15 Integer program solving the CENTER RANKING problem with no proved time bound. Here, $a, b \in [n], a \neq b, \lambda_i \in V$.

- Variables $S_{\{a,b\}} \in \{0,1\}$, encoding the solution, $S_{\{a,b\}} = 1 \Leftrightarrow a <_\sigma b$.
- Variables $H_{\lambda_i,\{a,b\}} \in \{0,1\}$,
- Numbers $l_{\lambda_i,\{a,b\}} \in \{0,1\}$, encoding the input, $l_{\lambda_i,\{a,b\}} = 1 \Leftrightarrow a <_{\lambda_i} b$
- Restriction $S_{\{a,b\}} + S_{\{b,c\}} - S_{\{a,c\}} \leq 1$
- Restriction $-S_{\{a,b\}} - S_{\{b,c\}} + S_{\{a,c\}} \leq 0$
- Restriction $l_{\lambda_i,\{a,b\}} + S_{\{a,b\}} \leq 2H_{\lambda_i,\{a,b\}} + 1$
- Restriction $l_{\lambda_i,\{a,b\}} + S_{\{a,b\}} \geq 2H_{\lambda_i,\{a,b\}}$
- Minimize $m' := \max_i \sum_{\{a,b\}} l_{\lambda_i,\{a,b\}} + S_{\{a,b\}} - 2H_{\lambda_i,\{a,b\}}$

Theorem 93. *Algorithm 15 correctly solves the* CENTER RANKING *problem.*

PROOF. Let $V \in \mathrm{Sym}_{[n]}^m, n, m \in \mathbb{N}$ be an instance of the CENTER RANKING problem. The variables used to encode the solution $\sigma \in \mathrm{Sym}_{[n]}$ are denoted as

$$S_{\{a,b\}} \in \{0,1\},\ a,b \in [n], a \neq b, \tag{6.1.1}$$

which we will interpret as $S_{\{a,b\}} = 1 \Leftrightarrow a <_\sigma b$.

To ensure that S denotes a permutation, we restrict variables $S_{\{a,b\}}$ as follows.

$$S_{\{a,b\}} = v \wedge S_{\{b,c\}} = v \Rightarrow S_{\{a,c\}} = v, \forall v \in \{0,1\}. \tag{6.1.2}$$

We can express implication (6.1.2) in a linear program for all $a,b \in [n], a \neq b$ as follows.

$$S_{\{a,b\}} + S_{\{b,c\}} - S_{\{a,c\}} \leq 1 \tag{6.1.3}$$

$$-S_{\{a,b\}} - S_{\{b,c\}} + S_{\{a,c\}} \leq 0. \tag{6.1.4}$$

We encode the input permutations $\lambda_i \in V$ in constants

$$l_{\lambda_i,\{a,b\}} \in \{0,1\}, l_{\lambda_i,\{a,b\}} = 1 \Leftrightarrow a <_{\lambda_i} b \tag{6.1.5}$$

.

The encoding we chose guarantees the strings to have the same respective distances as the original permutations. We proved this fact in Section 5.3.1.

The distance from the solution to the input permutations is expressed in the target function

$$\max_i \sum_{\{a,b\}} l_{\lambda_i,\{a,b\}} \oplus S_{\{a,b\}}. \tag{6.1.6}$$

Here, $\oplus$ denotes the XOR operation. We encode the XOR operation with helper variables $H_{\lambda_i,\{a,b\}} \in \{0,1\}$, such that

$$l_{\lambda_i,\{a,b\}} + S_{\{a,b\}} - 2H_{\lambda_i,\{a,b\}} = l_{\lambda_i,\{a,b\}} \oplus S_{\{a,b\}}. \tag{6.1.7}$$

Restriction (6.1.7) can be expressed in the following two inequalities.

$$l_{\lambda_i,\{a,b\}} + S_{\{a,b\}} \leq 2H_{\lambda_i,\{a,b\}} + 1 \tag{6.1.8}$$

$$l_{\lambda_i,\{a,b\}} + S_{\{a,b\}} \geq 2H_{\lambda_i,\{a,b\}} \tag{6.1.9}$$

The final target variable m that needs be minimized is defined as

$$m' \coloneqq \max_i \sum_{\{a,b\}} l_{\lambda_i,\{a,b\}} + S_{\{a,b\}} - 2H_{\lambda_i,\{a,b\}}, \tag{6.1.10}$$

subject to constraints (6.1.3), (6.1.4), (6.1.8), and (6.1.9), which is the integer program that was presented. □

The number of variables in the integer program we present is within $O(n^2 m)$. We discuss the performance of solving the program in the next section.

6.2. Performance of the implemented algorithms

The practical performance of three of the algorithms proposed in this thesis has been evaluated in experiments. These algorithms were the integer linear program, see Section 6.1, the trivial algorithm enumerating all $n!$ permutations, see Section 3.2, and the neighbor permutation search algorithm, see Section 4.2.

6.2.1. Implementation notes. The integer linear program from Section 6.1 could be implemented straightforwardly. Mathematica is able to compute the matrix that defines the linear program's restrictions by itself when given the inequalities and equations forming the restrictions. The integer program was given as input to Mathematica with no further optimization.

To implement the trivial algorithm, it was necessary to implement a new function that generates all permutations of a given length, despite the fact that Mathematica already ships with a function for precisely that purpose, the reason being that the Mathematica built-in function returns all permutations at once, requiring an immense amount of memory. To enumerate the permutations, a modification of the algorithm proposed by Sedgewick [55] was implemented, which requires no more than $O(n)$ memory.

The neighbor permutation search algorithm could be implemented straightforwardly from the description in Section 4.2. Since Mathematica allows the use of mathematical notations and ships with a large collection of graph algorithms, it was possible to retain nearly all of the notation used in Section 4.2 to describe the neighbor permutation search algorithm.

6.2.2. Test environment and experiment outline. All measurement and implementation was done in Mathematica 7 on a dual core (all computation has been single-threaded) 2.4 GHz notebook computer with 2 GB of RAM.

The run time of the neighbor permutation search algorithm very much depends on the supplied radius. For a non-minimal radius the run time may be significantly better than for the minimal radius. Thus, the performance is dominated by the distance from the supplied radius parameter to the minimum radius possible. To avoid measuring this rather unexciting effect, we have measured the performance of solving the optimization problem CENTER RANKING rather than the decision problem.

Since the neighbor permutation search algorithm is formulated inherently to solve the decision problem, the benchmark tests several radii, until the minimum radius is found, for which the decision problem answers "yes." The minimal radius is sought using a binary search. It is possible that seeking for the minimal radius by trying all possible radii in ascending order would have permitted better performance, because the run time of a run for a radius that is greater than the minimal radius might exceed the run time of all runs for smaller radii. It has not been practically compared whether a binary search or sequential search yields better performance.

Three systematic experiments were carried out. The first test was intended to measure the integer program implementation alone. It was not tested alongside the other implementations, because in exploratory computations, the integer program proved to be several magnitudes slower than both other implementations, such that for an instance small enough for the integer program to solve, the other two benchmarks would have meaninglessly short run times.

The second experiment compared the trivial algorithm with the neighbor permutation search algorithm for instances where the rankings were expected to correlate. Each instance was solved once by the trivial algorithm and once by the neighbor permutation search algorithm. A run that exceeded 20 seconds of tie was aborted. Then, the benchmarks were grouped such that each group would represent a triple (n, m, r), where n is a number of candidates, m is the number of votes and r is the minimum radius of the instance solved. In each group, the average run time and the standard deviation of the run time were computed both for the neighbor permutation search algorithm and for the trivial algorithm. If the average could not be computed, because the group contained the measuring of a run that had to

be aborted because it exceeded the 20 second limit, the average was not computed will be presented as absent in the upcoming subsection. If for a combination of candidates, votes and radii, both entries would have had to be left empty, the data set was disposed, because the instance radius was unknown.

The third experiment compared the run times of the trivial algorithm with the run time of the neighbor permutation search algorithm for instances chosen uniformly at random. The results were grouped only after the number of candidates and votes, not per radius. However, the average radius and the standard deviation of the radius was recorded. Also, in the third experiment computations were not aborted.

6.2.3. Instances. Three experiments were run, each with its own kind of instance.

The first kind of instance was designed to be small enough so the integer program would be able to solve it. For each combination of numbers of candidates (3, 4, or 5) and numbers of votes (3, 5, and 10), three instances were chosen uniformly at random. The sizes were chosen because they appeared to allow instances to be solved in less than 20 seconds. For any greater number of candidates, experimental computations took more than 2 minutes.

The second kind of instance was supposed to simulate several rankings with high correlation, i.e. small radii. To create an instance of minimal radius close to r, the input permutations were obtained as follows. We start with the identity permutation. We then chose any random pair of adjacent candidates and swap them around. This process is repeated k times, where k was is with binomial distribution with expected value r out of the range $\left[0, \binom{n}{2}\right]$. For each combination of numbers of candidates (7 or 20), number of votes (3, 8, and 30), and radii (1 through 7), 10 instances were created. Altogether 1470 instances were created. The numbers of candidates was chosen to be 7 because it is the highest number of candidates that the trivial algorithm can solve within 20 seconds. The neighbor permutation search algorithm still performs reliably fast for 20 candidates and small radii. The number of votes did not exceed 30, because otherwise the trivial algorithm might have exceeded the 20 second time limit.

Additionally, 30 instances were chosen uniformly at random for each combination of candidates (5, 6) and votes (3, 5, 15). The number of votes differs from the second experiment, because for 30 votes, the radii that were obtained grew too high such that the neighbor permutation algorithm took very long to complete. The number of candidates had to be chosen smaller than 7, because for 7 candidates, the neighbor permutation search algorithm takes several minutes of time to solve instances chosen uniformly at random.

6.2.4. Experimental results. The observed run time of the integer program has proved to be consistently worse by several orders of magnitude than that of the trivial algorithm. Many solvers for linear programs allow fine-tuning of the branching strategy. Fine-tuning would be particularly useful in our case to fine-tune the branching, since the integer program needs to find encodings of valid permutations, which might not be trivial by itself already. Finding valid encodings could be guided in a more refined integer program solver—however, the Mathematica solver for integer programs does not support interference with its "branch and bound" strategy. The measurings can be found in Table 5 on the facing page.

The benchmarking of instances with high correlation revealed that for radius less than 7, the neighbor permutation search algorithm can compute instances with very large numbers of candidates in short time. For example, for radii 4 and 7 candidates, the neighbor permutation performed on average more than double as

Table 5: Run time of the integer linear program solving the CENTER RANKING problem after 3 runs for each combination of parameters. The runs were aborted after 20 seconds; empty fields indicate that the run time exceeded 20 seconds. Here, $\bar{r}$ is the sample mean optimum solution quality, s_r is the sample standard deviation of the optimum solution quality; $\bar{t}$ refers to the mean time spent solving in seconds, s_t is the standard deviance of the run time.

parameters		measurements			
candidates	votes	$\bar{r}$	s_r	$\bar{t}$	s_t
3	3	1.67	0.57	0.14	0.05
3	5	2.00	0	0.32	0.13
3	10	2.33	0.57	1.99	0.61
4	3	2.67	0.57	1.69	0.09
4	5	2.33	0.57	5.57	0.34
4	10				
5	3	2.67	0.57	9.60	2.77
5	5				
5	10				

fast than the trivial algorithm. For 20 candidates, the trivial algorithm would have required years of run time, while the the neighbor permutation search algorithm could still reliably solve instances of 20 candidates, 3 votes and radius 6, in less than 4 seconds on average.

The results confirm that the run time of the neighbor permutation algorithm vastly depends on the minimal radius, while the trivial algorithm's run time is nearly constant for instants of the same number of candidates and votes. See Table 6 on page 84 for the precise measurements.

The systematic experiments depicted in Table 6 have been extended in a more exploratory way. Instances of 5 votes were created to see how far the algorithm can be taken, see Table 7 on page 84. The exploratory results reveal that the neighbor permutation search algorithm does become significantly slower with a raising number of candidates. For 2000 candidates, and radius 2, the algorithm did not terminate within 10 minutes. The amount of work done per recursive call of the neighbor permutation call is relatively big. A graph of size quadratic in the number of candidates needs to be created for every recursive call and it is tested whether the graph is acyclic.

For instances chosen uniformly at random it was uncertain before the implementation whether the neighbor permutation search algorithm would be able to compete with the trivial algorithm. The neighbor permutation search algorithm runs in time $O(n! \cdot n^2m + mn^2 \log n)$, see Corollary 72 on page 58; the trivial algorithm runs $O(n! \cdot mn \log n)$, see Section 3.2. While the run time of the trivial algorithm appears to be better, it was unclear how many of the $O(n!)$ possible sets of restriction the neighbor permutation search algorithm would really be trying. It was thought to be possible that in practical instances, the many criteria that abort the branching would lead to a small number of nodes in the search tree. For an instance of 15 permutations of length 6, the neighbor permutation tries an average of 491 restrictions (in the complete binary search for the optimum parameter), where there are only 720 permutations. The results show that the run time of the neighbor permutation search algorithm for instances chosen uniformly at random is significantly worse than the run time of the trivial algorithm, supposedly due to the raised amount of time spent per node in the search tree. Nonetheless, the

Table 6: Run times of the neighbor permutation search algorithm on average after at least 3 runs for each combination of the 3 parameters. Here, $\bar{t}$ refers to the average time consumed in seconds; s_t refers to the sample standard deviation of the time consumed. "Trivial" refers to the trivial algorithm computing the solution quality of every permutation. NPS refers to the neighbor permutation search algorithm from Section 4.2.

parameters			trivial		NPS	
candidates	votes	radius	$\bar{t}$	s_t	$\bar{t}$	s_t
7	3	1	1.982	0.022	0.010	0.010
7	3	2	1.983	0.020	0.049	0.043
7	3	3	1.979	0.016	0.219	0.215
7	3	4	1.985	0.020	0.859	1.006
7	3	5	2.003	0.031	3.146	4.731
7	8	1	4.422	0.064	0.032	0.004
7	8	2	4.393	0.054	0.040	0.031
7	8	3	4.397	0.045	0.135	0.195
7	8	4	4.407	0.055	0.656	0.727
7	8	5	4.422	0.069	2.184	2.145
7	8	6	4.414	0.065		
7	30	2	14.959	0.079	0.159	0.038
7	30	3	15.097	0.247	0.200	0.108
7	30	4	15.067	0.189	0.513	0.288
7	30	5	15.061	0.191	1.790	2.242
7	30	6	15.080	0.182		
7	30	7	15.014	0.149		
20	3	1			0.013	0.015
20	3	2			0.183	0.166
20	3	3			0.736	0.760
20	3	4			1.238	1.114
20	3	5			4.065	4.813
20	3	6			3.769	0.572
20	8	2			0.181	0.176

Table 7: Experiments that explore how large instances are handled by the neighbor permutation search algorithm. All instances contain 5 votes. Column "candidates" contains the number of candidates, column "radius" contains the minimal solution radius, and column "t" contains the time needed to solve the instance in seconds.

candidates	radius	t
1000	2	137
200	4	36
100	4	2

run time of the neighbor permutation search algorithm also greatly varies among instances.

See Table 8 on the facing page for the measurements of the benchmark.

Table 8: Run times of the neighbor permutation search and trivial algorithm for permutations chosen uniformly at random after 5 runs for each combination of candidates and votes. Here, $\bar{r}$ refers to the mean optimal solution quality; s_r refers to the sample standard deviance of the optimum solution quality; $\bar{t}$ refers to the average time consumed in seconds; s_t refers to the sample standard deviation of the time consumed. "Mean calls" refers to the mean number of recursive calls for the entire search for the minimum parameter.

		parameters		trivial		neighbor permutation search		
candidates	votes	$\bar{r}$	s_r	$\bar{t}$	s_t	$\bar{t}$	s_t	mean calls
5	3	3.2	0.447	0.040	0.000	0.261	0.410	13.6
5	5	4.4	0.894	0.059	0.003	1.035	1.413	18.4
5	15	6.0	0.707	0.151	0.001	2.409	1.961	79.0
6	3	4.6	1.673	0.263	0.002	11.475	12.645	57.8
6	5	6.0	1.000	0.388	0.014	38.243	29.938	265.2
6	15	8.6	0.548	1.023	0.019	59.519	48.361	491.0

CHAPTER 7

Conclusions and outlook

We proposed the CENTER RANKING aggregation method to be used for aggregating rankings that represent different criteria of judgement, especially in sports competitions. We provided several hardness results and numerous parameterized algorithms that help computing results efficiently for instances with correlation between the rankings, but also for instances chosen uniformly at random.

7.1. Summary of the algorithms presented

We have provided a large range of algorithms that solve the CENTER RANKING problem. For instances chosen uniformly at random or instances with only moderate correlation between rankings, Algorithm 14 on page 71, which enumerates a ball intersection, promises to be most useful. For instances with strong correlation between rankings and $r \leq 16$, Algorithm 7 on page 43, which runs a search tree of size $O(r^r)$, is expected to perform well. For instances with strong correlation between the rankings and $r > 16$, neighbor permutation search, Algorithm 8 on page 52, is most promising. Experiments show that Algorithm 8 can solve large instances with strong correlation in little time.

Table 9 contains a summary of all algorithms presented. All tractability results are summarized in Table 10

Table 9: Algorithms presented and their associated run times. Here, n refers to the number of candidates, m refers to the number of votes, r refers to optimum solution radius and p_r refers to the number of dirty pairs.

algorithm	asymptotic time bound
trivial, Algorithm 3 on page 27	$O(mn \log n \cdot n!)$
search tree on swaps, Algorithm 7 on page 43	$O(r^r mn \log n)$
neighbor permutation search, Algorithm 8 on page 52	$O(2^{4d} n^2 m + mn^2 \log n)$
search tree algorithm on dirty pairs in Section 3.7	$O(2^{p_r} \cdot mn \log n + n^2 m)$
ball enumeration, Algorithm 12 on page 65	$O\left(mn \log n \cdot \min\left\{r(2r)^n, rn^{2r}, n!\right\}\right)$
ball intersection enumeration, Algorithm 14 on page 71	$O(mn \log n \cdot \min\{rn^r, n!\})$

7.2. Open questions

7.2.1. Tie breaking. The CENTER RANKING problem can have several optimal solution. The algorithms we present all find only one optimal solutions. If we

Table 10: Fixed-parameter tractability results on the CENTER RANKING problem presented in this thesis listed per parameter, where #candidates means "number of candidates." If a parameterized algorithm was found, its run time is presented. If the problem was found to be NP-hard for fixed values of the parameter, the respective entry reads "NP-hard." It is unknown whether the CENTER RANKING problem is fixed-parameter tractable with respect to parameter "number of votes."

parameter	result	section
number of candidates, n	$O(n! \cdot n \log n)$	Section 3.2
number of votes, m	unknown	Section 3.8
radius, r	$O(2^{4r}mn^2 + mn^2 \log n)$	Chapter 4
maximum pairwise distance, $d_{\max}$	$O(2^{8d_{\max}}mn^2 + mn^2 \log n)$	Section 3.4
position range, p	NP-hard	Section 3.5
average pairwise distance, d_a	NP-hard	Section 3.6
number of dirty pairs, p_r	$O(2^{p_r} \cdot mn \log n + n^2 m)$	Section 3.7
#candidates and radius combined, (n, r)	$O(mn \log n \cdot \min\{rn^r, n!\})$	Chapter 5

wish to distinguish between the optimal solutions, we need to enumerate all optimal solutions and then find criteria which allow us to choose one aggregation out of all optimal aggregations.

The choice between several optimal aggregations may be domain-specific. It may not be of importance in all applications. However, if indeed sports competitions rankings are decided by the CENTER RANKING aggregation method, a choice has to be made.

7.2.2. Resemblance to existing ranking methods. It would be interesting to compare how the aggregation results we propose compare to the aggregation methods currently used to obtain final rankings in sport competitions. A close resemblance would make a strong case for our conjecture that in many cases, the CENTER RANKING aggregation method expresses naturally the intuitively expected balance between several criteria.

7.2.3. Fixed-parameter tractability with respect to the parameter "number of votes". It is still an open question whether the CENTER RANKING problem is fixed-parameter tractable with respect to the number of votes. If a combinatorial parameterized algorithm were found, it could also be used to solve the CLOSEST STRING problem. This would be of advantage since until today, the only parameterized algorithm with respect to the number of strings for the CLOSEST STRING problem is the integer linear program presented in Gramm et al. [24], which yields fixed-parameter tractability due to a rather theoretical result by Lenstra et al. [26].

Gramm et al. ask whether there is a combinatorial parameterized algorithm with respect to the parameter "number of strings" that solves the CLOSEST STRING problem. The parameterized algorithm for CLOSEST STRING provided by Gramm et al. shows an enormous combinatorial explosion in the parameter that could be

improved if a parameterized algorithm for the CENTER RANKING problem were found, because CLOSEST STRING instances can be reduced to CENTER RANKING instances, such that the reduced instances' number of votes equal the respective original instances' number of strings.

7.2.4. Fixed-parameter tractability with respect to the parameter "average distance". While we were able to prove NP-hardness with respect to parameter "pairwise average distance" in Theorem 41 in case that duplicate input votes are allowed, it is currently unknown whether the CENTER RANKING problem is fixed-parameter tractable with respect to the parameter "pairwise average distance," if duplicates are disallowed.

7.2.5. Generalizations. The CENTER RANKING method can be extended to ties between competitors by making use of the extension of the Kendall-τ distance proposed by Hemaspaandra et al. [30]. To allow votes with ties in the definition of the CENTER RANKING problem in Problem 3 on page 7, we need to extend the Kendall-τ metric to tell the similarity between such votes. Hemaspaandra et al. propose the generalized Kendall-τ distance to be defined as

$$d_{\mu,\lambda}(c,d) = \begin{cases} 0 & \text{if } \lambda \text{ and } \mu \text{ agree on } c \text{ and } d. \\ 1 & \text{if one of } \lambda, \mu \text{ has a preference among} c, d, \text{ while the other one has not.} \\ 2 & \text{if } \lambda \text{ and } \mu \text{ strictly disagree on } c \text{ and } d. \end{cases}$$

The ordinary Kendall-τ distance is obtained by setting $d_{\mu,\lambda}(c,d)$ to be 0 if permutations μ and λ rank candidates c and d in the same order, 1 otherwise.

The definition of the generalized Kendall-τ distance by Hemaspaandra et al. still expects all votes to be complete, i.e. to put all available candidates into an order. In the case that not all votes include the same candidates, Dwork at al. [19] propose to restrict two votes to the intersection of their respective sets of candidates ranked by erasing all other votes but leaving the relative orders of the remaining candidates untouched. A more sophisticated definition is proposed by Sculley [54], which accesses similarity information available to the data.

Betzler et al. [7] propose to weight votes with positive integer or real values. In case criteria might be of different importance to us, we could do the same and search for a permutation π with $\max_{\nu \in V} W_\nu \tau(\pi, \nu) < r$, where W_ν would be the weight of vote ν.

These generalizations can all be of some practical benefit. Yet, they might also significantly improve the complexity of the computation. It remains for the matter of this thesis an open but interesting question to ask which of the proposed algorithms can be extended to solve the above generalized definitions.

7.2.6. Randomizations. It is an open question how much the algorithms can be sped up by randomization techniques. A particularly interesting approach would be to try Schöning's "local search and restart" technique [16]. A rough outline of "local search and restart" might be the following. We choose a starting point at random and then try a random walk towards the solution, where the chance of the random walk to go towards the solution is only marginally better than sheer guessing.

When search tree algorithm 7 on page 43 would be modified into randomly choosing one of the input permutations as a starting point and then, instead of branching, choose one of the children in a search tree as a next step in a random walk, and do so repeatedly. The random walk would be aborted after $3r$ steps, if did not lead to a solution, then a repeated execution of this process yields an expected run time which is by a factor of r better than the worst case bound shown

for the algorithm. The improvement is not any better because, while the search is local and restarted, an important part of the technique is a good random choice of the starting point. Unfortunately, blindly choosing any starting out of $\mathrm{Sym}_{[n]}$ gives significantly worse run time than starting from the same member of the set of input votes V over and over again. If it were possible to choose a starting point with expected distance r to the solution that is binomially distributed, similar to the situation discussed in [16], then the expected run time might be significantly better than that of the search tree algorithms presented in this thesis.

7.2.7. Approximations. Algorithm 8 on page 52, the neighbor permutation search algorithm, is similar to the neighbor string search algorithm presented in [44]. The neighbor string search algorithm allows a polynomial time approximation algorithm for the CLOSEST STRING problem with time complexity $O(n^{O(\epsilon^{-2})})$. It is an open question whether the result can be brought forward to solving the CENTER RANKING problem.

7.2.8. Parallelization. While the search tree algorithms in Chapter 4 and Section 3.7 can be parallelized very well by referring the recursive calls to, we have not thoroughly examined how much they can be sped up by parallelizing the computation per search tree node.

APPENDIX A

Implementations

This chapter contains commented implementations of the neighbor search algorithm, i.e. Algorithm 8, the trivial algorithm, i.e. Algorithm 3, and the integer linear program described in Section 6.1, all implemented in Mathematica 7. We present the complete Mathematica notebooks, including the output, where applicable.

Algorithm 8, Algorithm 3 are both implemented to solve the CENTER RANKING optimization problem, as opposed to the decision problem.

Following is the code used to obtain the benchmark results presented in Chapter 6.

Implementation of the neighbor permutation search algorithm

This file will solve the neighbor permutation problem. See Niko Schwarz's thesis for details.
4/2009

Example run with explanations

In[201]:=
```
input =
  {{{4, 1, 3, 2}, 3}, {{2, 4, 3, 1}, 3}, {{1, 3, 2, 4}, 3}, {{2, 3, 1, 4}, 3}};
R =
  {};
```

First, we check if any of the distances in d is smaller than 0, equal to zero, or if R has reached maximum size already

In[203]:=
```
(*N0*)
If[Min[d] < 0, Return["Not found"]];
Module[{solutions, cas},
  cas = Cases[input, {_List, 0}][[All, 1]];
  If[cas ≠ {},
   solutions = Select[Function[p, Flatten[sym[Union[p, R]]]] /@ cas,
     # ≠ {} && solutionQ[#] &];
   If[solutions == {}, Return["Not found"], result[solutions // First]]];
If[Length[R] == Binomial[n, 2],
  Module[{sol = TopologicalSort[FromOrderedPairs[R]]},
   If[! solutionQ[sol], Return["Not found"], result[sol]]]];
If[! AcyclicQ[FromOrderedPairs[R]], Return["Not found"]];
```

It appears that none of these is the case. As a next step, let us compute the q needed by the algorithm. If none can be determined, the algorithm is done.

In[207]:=
```
(*N1*)
Module[{qcands = Select[Range[m],
    Function[i, τR[λ[[i]], λ[[1]]] > d[[i]] ]] },
 If[qcands == {},
  Module[{RR = ToOrderedPairs[TransitiveClosure[FromOrderedPairs[R]]],
    solution, p} ,
   p = Join[λ[[1]], RR];
   solution = TopologicalSort[ FromOrderedPairs[p]];
   If[solutionQ[solution], result[solution], Return["Not found"] ]]];
 q = qcands // First]; q
```

Out[207]= 2

Let us try a criterion that excludes the existence of a solution in the case.

In[208]:=
```
(*N2*)
If[τR[λ[[q]], λ[[1]]] ≤ d[[1]] + Max[d], , Return["Not found"]];
```

We compute P that determines the cases into which cases we branch:

In[209]:=
```
(*N3*)
P = Module[{prs = R^†},
  disagreement[λ[[1]], λ[[q]]] ⋂ Join[prs, Reverse[prs, {2}]]]
```

Out[209]=
```
{{1, 2}, {1, 3}, {3, 2}, {4, 2}}
```

Let us compute the field A which will contain all pairs we will branch into. All options need be compatible with R, and the graph of R needs to be transitive.

In[210]:=
```
(*N4*)
A' = Function[p, Join[Complement[P, p], Reverse[p, 2]]] /@ Subsets[P]
```

Out[210]=
```
{{{1, 2}, {1, 3}, {3, 2}, {4, 2}}, {{1, 3}, {3, 2}, {4, 2}, {2, 1}},
 {{1, 2}, {3, 2}, {4, 2}, {3, 1}}, {{1, 2}, {1, 3}, {4, 2}, {2, 3}},
 {{1, 2}, {1, 3}, {3, 2}, {2, 4}}, {{3, 2}, {4, 2}, {2, 1}, {3, 1}},
 {{1, 3}, {4, 2}, {2, 1}, {2, 3}}, {{1, 3}, {3, 2}, {2, 1}, {2, 4}},
 {{1, 2}, {4, 2}, {3, 1}, {2, 3}}, {{1, 2}, {3, 2}, {3, 1}, {2, 4}},
 {{1, 2}, {1, 3}, {2, 3}, {2, 4}}, {{4, 2}, {2, 1}, {3, 1}, {2, 3}},
 {{3, 2}, {2, 1}, {3, 1}, {2, 4}}, {{1, 3}, {2, 1}, {2, 3}, {2, 4}},
 {{1, 2}, {3, 1}, {2, 3}, {2, 4}}, {{2, 1}, {3, 1}, {2, 3}, {2, 4}}}
```

In[191]:=
```
(*N5*)
A = Join[R, #] & /@ A'
```

Out[191]=
```
{{{1, 2}, {1, 3}, {3, 2}, {4, 2}}, {{1, 3}, {3, 2}, {4, 2}, {2, 1}},
 {{1, 2}, {3, 2}, {4, 2}, {3, 1}}, {{1, 2}, {1, 3}, {4, 2}, {2, 3}},
 {{1, 2}, {1, 3}, {3, 2}, {2, 4}}, {{3, 2}, {4, 2}, {2, 1}, {3, 1}},
 {{1, 3}, {4, 2}, {2, 1}, {2, 3}}, {{1, 3}, {3, 2}, {2, 1}, {2, 4}},
 {{1, 2}, {4, 2}, {3, 1}, {2, 3}}, {{1, 2}, {3, 2}, {3, 1}, {2, 4}},
 {{1, 2}, {1, 3}, {2, 3}, {2, 4}}, {{4, 2}, {2, 1}, {3, 1}, {2, 3}},
 {{3, 2}, {2, 1}, {3, 1}, {2, 4}}, {{1, 3}, {2, 1}, {2, 3}, {2, 4}},
 {{1, 2}, {3, 1}, {2, 3}, {2, 4}}, {{2, 1}, {3, 1}, {2, 3}, {2, 4}}}
```

Let us compute the distances for each entry in A that we will branch into.

In[211]:=
```
(*N6*)
e[p_List, 1] := Module[{c = Complement[p, R]},
   Min[d[[1]] - Length[Complement[λ[[1]]_c^†, p]], ⌈d[[1]] / 2⌉ - 1]];
e[p_List, i_] := Module[{c = Complement[p, R]},
   d[[i]] - Length[Complement[λ[[i]]_c^†, p]]];
Outer[e, A, Range[m],
 1]
```

Out[213]=
```
{{1, -1, 2, -1}, {1, 0, 1, 0}, {1, 0, 1, 0}, {1, 0, 1, 0}, {1, 0, 3, 0},
 {1, 1, 0, 1}, {1, 1, 0, 1}, {1, 1, 2, 1}, {1, 1, 0, 1}, {1, 1, 2, 1}, {1, 1, 2, 1},
 {0, 2, -1, 2}, {0, 2, 1, 2}, {0, 2, 1, 2}, {0, 2, 1, 2}, {-1, 3, 0, 3}}
```

Let us branch into all elements of A. If a solution is found in one, the computation will be aborted.

In[214]:=
```
(*N7*)
Scan[
 Function[p,
  Module[{inp =
     {λ[[#]], e[p, #]} & /@ Range[1, m]},
   NPSrec [inp, p]
  ]],
 A]
```

Throw::nocatch : Uncaught Throw[{1, 2, 4, 3}] returned to top level. ≫

Out[214]=
```
Hold[Throw[{1, 2, 4, 3}]]
```

Here, {1, 2, 4, 3} was returned by the 11 th branch. Let us examine how, by resetting input and R to the input of the eighth branch.

In[215]:=
```
{input, R} = Module[{inp =
      {λ[[#]], e[a, #]} & /@ Range[1, m]}, {inp, a}] /. a → A[[11]]
```

Out[215]=
```
{{{{4, 1, 3, 2}, 1}, {{2, 4, 3, 1}, 1}, {{1, 3, 2, 4}, 2}, {{2, 3, 1, 4}, 1}},
 {{1, 2}, {1, 3}, {2, 3}, {2, 4}}}
```

Since several distances are set to 0, N0 will dutifully return the solution :

In[216]:=
```
(*N0*)
If[Min[d] < 0, Return["Not found"]];
Module[{solutions, cas},
  cas = Cases[input, {_List, 0}][[All, 1]];
  If[cas ≠ {},
   solutions = Select[Function[p, Flatten[sym[Union[p_(R†)⁻, R]]]] /@ cas,
     # ≠ {} && solutionQ[#] &];
   If[solutions == {}, Return["Not found"], result[solutions // First ]]];
If[Length[R] == Binomial[n, 2],
  Module[{sol = TopologicalSort[FromOrderedPairs[R]]},
   If[! solutionQ[sol], Return["Not found"], result[sol]]]];
If[! AcyclicQ[FromOrderedPairs[R]], Return["Not found"]];
```

N0 does not abort the branch. Let us compute q:

In[220]:=
```
(*N1*)
Module[{qcands = Select[Range[m],
    Function[i, τR[λ[[i]], λ[[1]]] > d[[i]] ]] },
 If[qcands == {},
  Module[{RR = ToOrderedPairs[TransitiveClosure[FromOrderedPairs[R]]],
    solution, p} ,
   p = Join[λ[[1]]_(RR†)⁻, RR];
   solution = TopologicalSort[ FromOrderedPairs[p]];
   If[solutionQ[solution], result[solution], Return["Not found"] ]]];
 q = qcands // First]; q
```

Out[220]=
```
4
```

The condition in N2 is not met :

In[221]:=
```
(*N2*)
If[τR[λ[[q]], λ[[1]]] ≤ d[[1]] + Max[d], , Return["Not found"]];
```

We compute P that determines the cases into which cases we branch:

In[222]:=
```
(*N3*)
P = Module[{prs = R†⁻},
   disagreement[λ[[1]], λ[[q]]] ⋂ Join[prs, Reverse[prs, {2}]]]
```

Out[222]=
```
{{4, 1}, {4, 3}}
```

Let us compute the field A which will contain all pairs we will branch into. All options need be compatible with R, and the graph of R needs to be transitive.

In[253]:=
```
(*N4*)
As = Function[p, Join[Complement[P, p], Reverse[p, 2]]] /@ Subsets[P]
```

Out[253]=
```
{{{4, 1}, {4, 3}}, {{4, 3}, {1, 4}}, {{4, 1}, {3, 4}}, {{1, 4}, {3, 4}}}
```

In[254]:=
```
(*N5*)
A = Join[R, #] & /@ As
```

Out[254]=
```
{{{1, 2}, {1, 3}, {2, 3}, {2, 4}, {4, 1}, {4, 3}},
 {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {4, 3}, {1, 4}},
 {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {4, 1}, {3, 4}},
 {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {1, 4}, {3, 4}}}
```

Let us compute the distances for each entry in A that we will branch into.

In[255]:=
```
(*N6*)
e[p_List, 1] := Module[{c = Complement[p, R]},
   Min[d[[1]] - Length[Complement[λ[[1]]c↑, p]], ⌈d[[1]] / 2⌉ - 1]];
e[p_List, i_] := Module[{c = Complement[p, R]},
   d[[i]] - Length[Complement[λ[[i]]c↑, p]]];
Outer[e, A, Range[m],
 1]
```

Out[257]=
```
{{0, 1, 0, -1}, {0, 0, 1, 0}, {0, 0, 1, 0}, {-1, -1, 2, 1}}
```

Apparantly, the solution can only be branch 2 or 3.
Let us branch into all elements of A. If a solution is found in one, the computation will be aborted.

In[258]:=
```
(*N7*)
Scan[
 Function[p,
  Module[{inp =
     {λ[[#]], e[p, #]} & /@ Range[1, m]},
   NPSrec [inp, p]
  ]],
 A]
```

Throw::nocatch : Uncaught Throw[{1, 2, 4, 3}] returned to top level. ≫

Out[258]=
```
Hold[Throw[{1, 2, 4, 3}]]
```

We will find be found in the second branch. Let us reset the input to simulate the recursive call on level 3 into branch 2.

In[270]:=
```
{input, R} = Module[{inp =
      {λ[[#]], e[a, #]} & /@ Range[1, m]}, {inp, a}] /. a → A[[2]]
```

Out[270]=
```
{{{{4, 1, 3, 2}, 0}, {{2, 4, 3, 1}, 0}, {{1, 3, 2, 4}, 1}, {{2, 3, 1, 4}, 0}},
 {{1, 2}, {1, 3}, {2, 3}, {2, 4}, {4, 3}, {1, 4}}}
```

Since several distances are set to 0, N0 will dutifully return the solution :

In[275]:=
```
(*N0*)
If[Min[d] < 0, Return["Not found"]];
Module[{solutions, cas},
  cas = Cases[input, {_List, 0}][[All, 1]];
  If[cas ≠ {},
   solutions = Select[Function[p, Flatten[sym[Union[p_R↑, R]]]] /@ cas,
     # ≠ {} && solutionQ[#] &];
   If[solutions == {}, Return["Not found"], result[solutions // First]]];
If[Length[R] == Binomial[n, 2],
  Module[{sol = TopologicalSort[FromOrderedPairs[R]]},
   If[! solutionQ[sol], Return["Not found"], result[sol]]]];
If[! AcyclicQ[FromOrderedPairs[R]], Return["Not found"]];
```

Throw::nocatch : Uncaught Throw[{1, 2, 4, 3}] returned to top level. ≫

Out[276]=
```
Hold[Throw[{1, 2, 4, 3}]]
```

General Definitions

The definitions used closely resemble the definitions given in the Thesis.

```
Needs["Combinatorica`"]

(*global definitions*)
Clear[Ω, n, m, OverBar, Subscript, τ, SuperDagger]
Ω_Integer := Ω_n = Subsets[Range[n], {2}]
set_† := Function[arg,
   Module[{a = arg[[1]], b = arg[[2]]}, If[b < a, {b, a}, {a, b}]]] /@ set

λ_Q_List /; PermutationQ[λ] := Module[
  {inv = InversePermutation[λ]},
  Function[q, Module[{a = q[[1]], b = q[[2]]},
     If[inv[[a]] > inv[[b]], {b, a}, {a, b}]]] /@ Q]

_____
set_ := Complement[Ω_n, set]

disagreement[p_, q_] := Complement[p_Ω_n, q_Ω_n]

τ[μ_List, ν_List] := Inversions[InversePermutation[μ][[ν]]];
(*τ[μ_List,ν_List]:=τ_Ω_n[μ,ν];*)
τ_R_List[μ_List, ν_List] := Complement[μ_R, ν_R] // Length;
τ_Ω_n := τ

(*small function helpers for testing*)
solutionQ[μ_] :=
  And @@
   Function[arg, Module[{γ = arg[[1]], d = arg[[2]]}, τR[γ, μ] ≤ d]] /@ input;
τR[a_, b_] := τ_R†‾[a, b];
τRP[a_, b_] := τ_P†∩R†‾[a, b];

λ := input[[All, 1]];
d := input[[All, 2]];
m := input // Length
n := λ // First // Length

Clear[sym];
Intersection[sym[R_], sym[Q_]] ^:= sym[Union[Q, R]]
sym[res_] := Module[{P = res‾},
   Select[TopologicalSort /@
     FromOrderedPairs /@
      (Function[s, Join[Complement[P, s], Reverse[s, 2], res]] /@
        Subsets[P]),
    PermutationQ]];
calls = 0;
```

The NPS Algorithm

```
Clear[NPS, NPSrec, NPSdec];
result[μ_] := Throw[μ];
NPSrec[input_, R_] := Module[
   {λ = input[[All, 1]],
    d = input[[All, 2]], τR, τRP, P, Q, q, A, solutionQ, Ra, e},
   calls += 1;
   τR[a_, b_] := τ_(R^†)[a, b];
   solutionQ[μ_] :=
    And @@ Function[arg,
       Module[{γ = arg[[1]], d = arg[[2]]}, τR[γ, μ] ≤ d]] /@ input;

   (*N0*)
   If[Min[d] < 0, Return["Not found"]];
   Module[{solutions, cas},
    cas = Cases[input, {_List, 0}][[All, 1]];
    If[cas ≠ {},
     solutions = Select[Function[p, Flatten[sym[Union[p_(R^†), R]]]] /@ cas,
       # ≠ {} && solutionQ[#] &];
     If[solutions == {}, Return["Not found"], result[solutions // First ]]]];
   If[Length[R] == Binomial[n, 2],
    Module[{sol = TopologicalSort[FromOrderedPairs[R]]},
     If[! solutionQ[sol], Return["Not found"], result[sol]]]];

   (*N1*)
   Module[{qcands = Select[Range[m],
       Function[i, τR[λ[[i]], λ[[1]]] > d[[i]] ]] },
    If[qcands == {}, Module[{RR =
        ToOrderedPairs[TransitiveClosure[FromOrderedPairs[R]]], solution} ,
      solution = TopologicalSort[ FromOrderedPairs[Join[λ[[1]]_(RR^†), RR]]];
      If[solutionQ[solution], result[solution], Return["Not found"] ]]];
    q = qcands // First];

   (*N2*)
   If[τR[λ[[q]], λ[[1]]] ≤ d[[1]] + Max[d], , Return["Not found"]];

   (*N3*)
   P = Module[{{prs = R^†}},
     disagreement[λ[[1]], λ[[q]]] ⋂ Join[prs, Reverse[prs, {2}]]];

   (*N4,N5*)
   A = Function[p, Join[R, Complement[P, p], Reverse[p, 2]]] /@ Subsets[P];

   (*extra: branch restrict*)
   A = Select[A, Composition[AcyclicQ, FromOrderedPairs]];
```

```
    (*N6*)
    e[p_List, 1] := Module[{c = Complement[p, R]},
      Min[d[[1]] - Length[Complement[λ[[1]]c†, p]], ⌈d[[1]] / 2⌉ - 1]];
    e[p_List, i_] := Module[{c = Complement[p, R]},
      d[[i]] - Length[Complement[λ[[i]]c†, p]]];

    (*N7*)
    Scan[
     Function[a,
      Module[{inp =
          {λ[[#]], e[a, #]} & /@ Range[1, m]},
       NPSrec [inp, a]
      ]],
     A];

    (*N8*)
    Return["Not found"]
   ];
NPS[input : {{_List, _Integer} ..}, R_List] := Catch[NPSrec[input, R]];
NPSdec[input : {{_List, _Integer} ..}, R_List] := (NPS[input, R]) // ListQ;

Clear[binarySearch, q]
binarySearch[expr_, var_Symbol] :=
 Module[{r, l, x, dists = Function[q, τ @@ q] /@ Subsets[λ, {2}]},
  r = Max[dists]; l = Ceiling[r / 2];
  While[r ≠ l,
   x := Quotient[l + r, 2];
   If[Evaluate[expr /. var → x], r = x, l = x + 1];
  ];
  r
 ]

SetAttributes[binarySearch, HoldFirst]
NPS[] := binarySearch[NPSdec[input /. r → q, R], q]
```

Integer Programming for computing the Kendall τ – distance

This program solves the Minimum Kendall τ-distance problem, using integer programming. The input is created as a random problem instance consisting of m permutations of length m. To solve other instances, replace variable inst with another instance. We will compute the score of the optimal solution, print one optimal solution, both in permutation form and in a special 0-1-presentation, defined in stringView. The stringView of a permutation is a string of zeroes and ones where each position has a specified meaning.
Each position is assigned to a special pair (a, b) ∈ [n]^2 . Now, the position is set to 1 iff the permutation lists b before a, otherwise 0. This makes computing the Kendall-τ distance easy: The Kendall τ - distance is now the sum of the column-wise XOR in the transformed representation.
The Integer Program will use helper variables HH to compute the XOR. They will be defined using conditions "xorHelperRestrictions". To make sure that the defined pairs of the solution string will be transitive (that means, s[{a, b}] == 0, s[{b, c}] == 0, then s[{a, c}] == 0), we will use conditions transitivityRestrictions.

```
Needs["Combinatorica`"]
CreateInstance[n_: n, m_: m] := Table[RandomPermutation[n], {m}]
inst = CreateInstance[4, 3];

pairs := Subsets[Range[n], {2}];
stringView[permutation_] := If[#, 1, 0] & /@
  Module[{a, b, inv = InversePermutation[permutation]},
   Function[pr,
     {a, b} = pr; inv[[a]] < inv[[b]]] /@ pairs]
OO[perm_, pr_] := stringView[perm][[positionOf[pr]]]
stringView /@ inst
```

```
{{0, 0, 0, 1, 1, 0}, {1, 0, 0, 0, 0, 1}, {1, 1, 1, 0, 0, 1}}
```

HH contains the helper variables to compute the XOR. They are defined using"xorHelperRestriction".
S contains the variables making up the solution. s[{a, b}] == 1, iff s lists a before b.
To make sure that the defined pairs of the solution string will be transitive (that means, s[{a, b}] == 0, s[{b, c}] == 0, then s[{a, c}] == 0), we will use conditions transitivityRestrictions.

```
HH := Outer[h, inst, pairs, 1] // Flatten;
S := Thread[s[pairs]];
vars := Union[HH, S, {maximum}];
positionOf[pr_] := Position[pairs, pr, 1] // Flatten // First
transitivityRestriction := And @@ Module[{a, b, c}, Function[tri,
       {a, b, c} = tri; s[{a, b}] + s[{b, c}] - s[{a, c}] ≤ 1
        && -s[{a, b}] - s[{b, c}] + s[{a, c}] ≤ 0
      ]] /@ Subsets[Range[n], {3}] ;

xorHelperRestriction := And @@ Module[{perm, pr},
     Function[arg, {perm, pr} = arg;
        OO[perm, pr] + s[pr] ≤ 2 h[perm, pr] + 1
         && OO[perm, pr] + s[pr] ≥ 2 h [perm, pr]]
      /@ Flatten[Outer[List, inst, pairs, 1], 1] ];
variableConstraints := And @@ Thread[1 ≥ S ≥ 0] && And @@ Thread[1 ≥ HH ≥ 0];
values := Function[w, Total[OO[w, #] + s[#] - 2 h[w, #] & /@ pairs]] /@ inst;
constraints := Apply[List,
    Join[transitivityRestriction, xorHelperRestriction, variableConstraints,
     Apply[And, Thread[maximum ≥ values]]]];
Timing[{res, rules} = Minimize[{maximum, constraints}, vars, Integers]] // First
{res, rules}
```

```
1.97463
```

```
{3, {maximum → 3, h[{1, 3, 4, 2}, {1, 2}] → 0, h[{1, 3, 4, 2}, {1, 3}] → 0,
  h[{1, 3, 4, 2}, {1, 4}] → 1, h[{1, 3, 4, 2}, {2, 3}] → 0,
  h[{1, 3, 4, 2}, {2, 4}] → 0, h[{1, 3, 4, 2}, {3, 4}] → 1, h[{2, 4, 3, 1}, {1, 2}] → 0,
  h[{2, 4, 3, 1}, {1, 3}] → 0, h[{2, 4, 3, 1}, {1, 4}] → 0, h[{2, 4, 3, 1}, {2, 3}] → 0,
  h[{2, 4, 3, 1}, {2, 4}] → 1, h[{2, 4, 3, 1}, {3, 4}] → 0, h[{3, 4, 1, 2}, {1, 2}] → 0,
  h[{3, 4, 1, 2}, {1, 3}] → 0, h[{3, 4, 1, 2}, {1, 4}] → 0, h[{3, 4, 1, 2}, {2, 3}] → 0,
  h[{3, 4, 1, 2}, {2, 4}] → 0, h[{3, 4, 1, 2}, {3, 4}] → 1, s[{1, 2}] → 0,
  s[{1, 3}] → 0, s[{1, 4}] → 1, s[{2, 3}] → 0, s[{2, 4}] → 1, s[{3, 4}] → 1}}
```

Here, the linear program put together and run. The output is the optimum solution quality and the assignments done to the variables.

We will compute the result expressed by these assignments. Since the solution was coputed in a string view, we will also show the string view of the solution and the input permutations. The first rows of the table are the stringViews of the input permutations. The last row is the stringView of the solution.

```
solution = Sort[Range[n], (S /. rules)[[positionOf[{#1, #2}]]] == 1 &]

TableView[Insert[Insert[stringView /@ inst , S /. rules, -1], pairs, 1] , ItemSize → 4]
```

```
{3, 2, 1, 4}
```

	1	2	3	4	5	6
1	{1,2}	{1,3}	{1,4}	{2,3}	{2,4}	{3,4}
2	0	0	0	1	1	0
3	1	0	0	0	0	1
4	1	1	1	0	0	1
5	0	0	1	0	1	1

Verifying the result

Here, we define the distance computation differently, to verify the previous results. We will use the τ function from the NPS implementation file here.

The diameter of the instance. The solution has to be greater than half the diameter, but not bigger than the diameter :

```
diameter = Module[{a, b},
   Function[arg, {a, b} = arg; τ[a, b]] /@ Subsets[inst, {2}] ] // Max
```

```
6
```

Let us examine our solution compared with all input strings :

```
τ[#, solution] & /@ inst
```

```
{3, 3, 3}
```

This implies that the computed score is correct. Is there a better solution?

```
score[perm_?PermutationQ] := Max[ τ[#, perm] & /@ inst];
{time, result} = score /@ Permutations[Range[n]] // Min // Timing
```

```
{0.008667, 3}
```

No, there is not. Note that, unfortunately, the brute force algorithm is a LOT faster than the linear program.

Benchmark code

Niko Schwarz, 4/2009

The code presented here is used to provide the performance measurements shown and discussed in Niko Schwarz' s Diplom Thesis . It was run in *Mathematica* 7.

Definitions used in the benchmark

```
CreateInstance[n_, m_, r_] :=
 Module[{}, {input, R} = {{#, r} & /@  Table[RandomPermutation[n], {m}], {}}]

CreateInstance[n_: n, m_: m] :=  Table[RandomPermutation[n], {m}]

swap[ll_List, i_Integer] :=
 Module[{l = ll, t = ll[[i]]}, l[[i]] = l[[i + 1]]; l[[i + 1]] = t; l]

next[n_, expectedDist_] := Module[{t, perm = Range[n], swaps = RandomInteger[
      BinomialDistribution[Binomial[n, 2], expectedDist / Binomial[n, 2]]]},
  Do[perm = swap[perm, RandomInteger[{1, n - 1}]], {swaps}]; perm]

CreateEasyInstance[n_, m_, r_, expectedDist_: 5] :=
 Module[{}, {input, R} = {{#, r} & /@  Table[next[n, expectedDist], {m}], {}}]

score[perm_List] := Max[τ[perm, #] & /@ λ]

(*enumerate all permutaions and call f for each*)
gen[f_, n_] := Module[{id = -1, val = Table[Null, {n}], visit},
  visit[k_] := Module[{t},
    id++; If[k ≠ 0, val[[k]] = id];
    If[id == n, f[val]];
    Do[If[val[[t]] == Null, visit[t]], {t, 1, n}];
    id--; val[[k]] = Null;];
  visit[0];
 ]

mean[x_List] /; MemberQ[x, $Aborted] := $Aborted
var[x_List] /; MemberQ[x, $Aborted] := $Aborted
mean[x : ___] := N[Mean[x], 3]
var[x : ___] := N[StandardDeviation[x], 4]

triv[] := Module[{min = ∞, s, ret},
  gen[Function[p, s = score[p]; If[s < min, min = s; ret = p]], n]; min]
```

Benchmarks

```
runtimes = {};

NN = {7, 20}; MM = {3, 8, 30}; RR = Range[1, 7];
benchmark[n_, m_, r_] :=
 Module[{}, CreateEasyInstance[n, m, q, r]; AppendTo[runtimes, {n, m,
    TimeConstrained[triv[] // Timing, 20], TimeConstrained[NPS[] // Timing, 20]}]]

Do[Outer[benchmark, NN, MM, RR], {10}]
```

```
runtimes;
ReplaceAll[%, {nn_, mm_, {a_, r_}, {b_, _}} → {nn, mm, r, a, b}];
ReplaceAll[%, {nn_, mm_, {a_, r_}, $Aborted} → {nn, mm, r, a, $Aborted}];
ReplaceAll[%, {nn_, mm_, $Aborted, {b_, r_}} → {nn, mm, r, $Aborted, b}];
Select[%, Length[#] == 5 &] (*drop samples where both runs were interrupted*);

GatherBy[%, Take[#, 3] &];
Select[%, Length[#] ≥ 3 &];
Join[Take[# // First, 3],
    {mean[#[[All, 4]]], var[#[[All, 4]]], mean[#[[All, 5]]], var[#[[All, 5]]]} & /@ %;
SortBy[%, Dot[Reverse[Array[Function[i, 500^(i/.$Aborted→0)], 7]], #] &];
% /. f_Real → Round[f, 0.001];
Export["Dropbox/Thesis/tabgen1.csv", %, "CSV"];
%% // TableView;



Import["Dropbox/Thesis/tabgen1.csv"]
SortBy[%, Dot[Reverse[Array[Function[i, 10^(i/.$Aborted→0)], 7]], #] &];
% /. f_Real → Round[f, 0.001];
Export["Dropbox/Thesis/tabgen11.csv", %, "CSV"];
%% // TableView

runtimes2 = {};

benchmark2[n_, m_] := Module[{}, CreateInstance[n, m, q]; calls = 0;
  AppendTo[runtimes2, {n, m, triv[] // Timing, NPS[] // Timing, calls}]]

MemoryConstrained[
 Do[Outer[benchmark2, {5, 6}, {3, 5, 15}], {5}], (3 / 2) * 1000 * 1000 * 1000]


runtimes2;
ReplaceAll[%, {nn_, mm_, {a_, r_}, {b_, _}, t_} → {nn, mm, r, a, b, t}];
ReplaceAll[%, {nn_, mm_, {a_, r_}, $Aborted, t_} → {nn, mm, r, a, $Aborted, t}];
ReplaceAll[%, {nn_, mm_, $Aborted, {b_, r_}, t_} → {nn, mm, r, $Aborted, b, t}];
GatherBy[%, Take[#, 2] &];
Join[Take[# // First, 2],
    {mean[#[[All, 3]]], var[#[[All, 3]]], mean[#[[All, 4]]], var[#[[All, 4]]],
     mean[#[[All, 5]]], var[#[[All, 5]]], mean[#[[All, 6]]]} & /@ %;
SortBy[%, Dot[Reverse[Array[Function[i, 50^(i/.$Aborted→0)], 9]], #] &];
% /. f_Real → Round[f, 0.001];
Export["Dropbox/Thesis/tabgen2.csv", %]
%%


runtimes3 = {};

benchmark3[n_, m_] := Module[{time, erg},
   inst = CreateInstance[n, m];
   time = TimeConstrained[
     Timing[erg = Minimize[{maximum, constraints}, vars, Integers]][[1]] , 20];
   rules = erg[[2]]
     AppendTo[runtimes3, {n, m, erg[[1]], time}];
  ];

Do[Outer[benchmark3, {3, 4, 5}, {3, 5, 10}], {3}];

runtimes3;
ReplaceAll[%, {nn_, mm_, r_, $Aborted} → {nn, mm, $Aborted, $Aborted}];
GatherBy[%, Take[#, 2] &];
Join[Take[# // First, 2],
    {mean[#[[All, 3]]], var[#[[All, 3]]], mean[#[[All, 4]]], var[#[[All, 4]]]} & /@ %;
Export["Dropbox/Thesis/tabgen3.csv", %]
%% // TableView
```

Exploratory experiments

```
CreateEasyInstance[100, 5, q, 3];
NPS[] // Timing
```

```
{2.36686, 4}
```

```
CreateEasyInstance[200, 5, q, 2];
NPS[] // Timing
```

```
{9.25836, 4}
```

```
CreateEasyInstance[1000, 5, q, 1];
NPS[] // Timing
```

```
{102.865, 1}
```

Helper function for testing

To test the functionality, it might be convenient to enumerate sym[a_]. The easiset way is this:

```
Clear[sym]
sym[R_List] :=  Select[Permutations[Range[n]], Function[e, Module[{prs = e⟦Ω_n⟧},
     And @@ (MemberQ[prs, #] & /@ R)]]]
```

However, we are interested in an algorithm that can is exponential only in the output size. One could improve the above by re-implementing the enumeration of all permutations (as a depth-first search that tracks back, see Sedgewick's "Algorithms") and abort as soon as the properties are violated. However, this can still have run-time exponential in n eventhough the output is not exponential in n:

```
Clear[sym]
Intersection[sym[a_], sym[b_]] ^:= sym[a ⋃ b]
sym[R_] := Module[{id, val, visit},
   id = -1; val = Table[Null, {n}];

   visit[k_] := Module[{t},
     id++;
     If[k ≠ 0, val[[k]] = id;
      If[(R ⋂ Reverse[pairsWith[val, id], 2]) ≠ {},
       id--; val[[k]] = Null; Return[]]];

     If[id == n, Sow[val]];
     For[t = 1, t ≤ n, t++,
      If[val[[t]] == Null, visit[t]]
     ];
     id--; val[[k]] = Null;
    ];
   Flatten[Reap[visit[0]][[2]], 1]];

(*Pairs in perm that have i as one entry,
in exactly the order in which they appear in perm*)
pairsWith[perm_, i_] := Module[{flip = False},
  Flatten[Reap[
     Scan[Function[el,
       If[el == i, flip = True,
        Sow[If[flip, {i, el}, {el, i}]]]], perm]][[2]], 1]]

Block[{n = 6}, Intersection[sym[{{1, 2}, {2, 3}, {3, 4}}], sym[{{4, 5}}]]] // Timing
```

```
{0.169979, {{1, 2, 3, 4, 5, 6}, {1, 2, 3, 4, 6, 5},
  {1, 2, 3, 6, 4, 5}, {1, 2, 6, 3, 4, 5}, {1, 6, 2, 3, 4, 5}, {6, 1, 2, 3, 4, 5}}}
```

The previous algorithm is already a lot faster than the naive procedure; unfortunately, it is still exponential in n. This is the speed for n=10:

```
Block[{n = 11}, Intersection[sym[{}], sym[Table[i, {i, 1, 12}]⟦Ω_12⟧]]] // Timing // First
```

```
1.27948
```

Fortunately, we can speed it up a great deal by only trying the possible inversions we are interested in, and checking if, together with the restrictions, they make up a permutation.

```
Clear[sym];
sym[res_] := Module[{P = R̄},
  TopologicalSort /@
   Select[
    FromOrderedPairs /@
     (Function[s, Join[Complement[P, s], Reverse[s, 2], res]] /@
       Subsets[P] )
    , AcyclicQ]]
```

Which yields acceptable speed :

```
Block[{n = 13}, Intersection[sym[{}], sym[Table[i, {i, 1, 12}]_Ω12]]] // Timing // First
```

```
0.009672
```

Tests that show the intented behaviour of the definitions.

These tests are convenient to verify that all tests set in the NPS algorithm implementation work as intended. All these statements should evaluate to true.

```
{{3, 2}, {1, 4}}† == {{2, 3}, {1, 4}}
{1, 2, 4, 3}_{{1,2},{3,4}} == {{1, 2}, {4, 3}}
Module[{n = 5}, (Range[n]_Ωn // Length) == Binomial[n, 2]]
Block[{n = 11}, Length[{{1, 2}}‾] == Binomial[n, 2] - 1]

τ[{4, 2, 3, 1, 5}, {1, 2, 3, 4, 5}] == 5
τ_{{1,2},{2,3},{4,2}}[{4, 2, 3, 1, 5}, {1, 2, 3, 4, 5}] == 2
```

```
True

True

True

True

True

True
```

APPENDIX B

Transformation rules for CLOSEST STRING

We provide rules that help transform a CLOSEST STRING instance into another CLOSEST STRING that is solvable if and only if the original instance is solvable.

Example. Let us assume that we have an input (S, r) and two input strings with maximum distance $\mathbf{s}_1, \mathbf{s}_2 \in S$ which are

$$\mathbf{s}_1 = 01001011001$$
$$\mathbf{s}_2 = 01011100100.$$

We can tidy up the instance see the structure of $\mathbf{s}_1$ and $\mathbf{s}_2$ more easily. The idea is to replace every $\mathbf{s} \in S$ by some $\mathbf{s}'$, without changing anything essential.

Lemma (B.1. Isometric transformation)**.** *Let $f : \{0,1\}^D \to \{0,1\}^D, D \in \mathbb{N}$ be bijective and isometric on the Hamming distance, i.e. f^{-1} exists and $\forall \mathbf{a}, \mathbf{b} \in \{0,1\}^D$: $d_H(\mathbf{a}, \mathbf{b}) = d_H(f(\mathbf{a}), f(\mathbf{b}))$, then (S, r) is a* CLOSEST STRING *instance that can be answered in the affirmative if and only if $(f(S), r)$ is a yes-instance.*

PROOF. Note first that if f is isometric, then f^{-1} must be isometric, too. This is because

$$\begin{aligned} & d_H(f^{-1}(\mathbf{a}), f^{-1}(\mathbf{b})) \\ =& d_H(f(f^{-1}(\mathbf{a})), f(f^{-1}(\mathbf{b}))) \\ =& d_H(\mathbf{a}, \mathbf{b}) \end{aligned}$$

Hence, for all $\mathbf{s} \in \{0,1\}^D$ we find that

$$\begin{aligned} & f(S) \subset \mathbb{B}_{d_H, r}(\mathbf{s}) \\ \Leftrightarrow & \forall \mathbf{s}' \in f(S) : d_H(\mathbf{s}', \mathbf{s}) \leq r \\ \Leftrightarrow & \forall \mathbf{s}' \in S : d_H(f(\mathbf{s}'), \mathbf{s}) \leq r \\ \Leftrightarrow & \forall \mathbf{s}' \in S : d_H(f^{-1}(f(\mathbf{s}')), f^{-1}(\mathbf{s})) \leq r \\ \Leftrightarrow & \forall \mathbf{s}' \in S : d_H(\mathbf{s}', f^{-1}(\mathbf{s})) \leq r \\ \Leftrightarrow & S \subset \mathbb{B}_{d_H, r}(f^{-1}(\mathbf{s})) \end{aligned}$$

Thus we are proving that a solution of $(f(S), r)$ yields a solution of (S, r). And, if we replace $\mathbf{s}$ by $f(\mathbf{s})$ in the above, we see that

$$S \subset \mathbb{B}_{d_H, r}(s) \Leftrightarrow f(S) \subset \mathbb{B}_{d_H, r}(f(\mathbf{s}))$$

Which proves that a solution in (S, r) yields a solution in $(f(S), r)$. □

This allows us to tidy up our example instance. We transform the whole instance using Lemma B.1 . Applying an XOR with $\mathbf{s}_1$ on the whole instance would achieve $\mathbf{s}_1$ to consist only of zeroes, while not changing the solvability of the problem. We would get:

$$\mathbf{s}_1' = 00000000000$$
$$\mathbf{s}_2' = 00010111101$$

To show that we can use Lemma B.1, we need to show that the xor operation is isometric.

Lemma (B.2)**.** *Let* $\mathbf{a} \in \{0,1\}^D, D \in \mathbb{N}$*, then function* f

$$f : \{0,1\}^D \to \{0,1\}^D, f(\mathbf{q}) \coloneqq \mathbf{a} \oplus \mathbf{q}$$

is invertible and isometric on d_H*.*

Proof. We prove that f is bijective.
Assume that for $\mathbf{s}, \mathbf{t} \in \{0,1\}^D$, we would find $f(\mathbf{s}) = f(\mathbf{t})$. We would get

$$\begin{aligned} & f(\mathbf{s}) = f(\mathbf{t}) \\ \Rightarrow & \mathbf{s} \oplus \mathbf{a} = \mathbf{t} \oplus \mathbf{a} \\ \Rightarrow & \mathbf{s} \oplus \mathbf{a} \oplus \mathbf{a} = \mathbf{t} \oplus \mathbf{a} \oplus \mathbf{a} \\ \Rightarrow & \mathbf{s} = \mathbf{t} \end{aligned}$$

Hence, f is injective.
Given any $\mathbf{s} \in \{0,1\}^D$, we find that

$$f(\mathbf{a} \oplus \mathbf{s}) = \mathbf{s}$$

Hence, f is surjective.
We prove that f is isometric.
We can define the Hamming distance as follows:

$$d_H(\mathbf{s}, \mathbf{t}) = \sum_{1 \le i \le D} (\mathbf{s} \oplus \mathbf{t})_i = \mathbf{1}(\mathbf{s} \oplus \mathbf{t})^T$$

And now we get

$$d_H(\mathbf{s}_1 \oplus \mathbf{t}, \mathbf{s}_2 \oplus \mathbf{t}) = \mathbf{1}(\mathbf{s}_1 \oplus \mathbf{t} \oplus \mathbf{s}_2 \oplus \mathbf{t})^T = \mathbf{1}(\mathbf{s}_1 \oplus \mathbf{s}_2)^T = d_H(\mathbf{s}_1, \mathbf{s}_2)$$

The second equation follows from $\mathbf{t} \oplus \mathbf{t} = \mathbf{0}$. Hence, we find

$$d_H(\mathbf{s}, \mathbf{t}) = d_H(f(\mathbf{s}), f(\mathbf{t}))$$

□

We can still order a little more. We would like to sort all ones in $\mathbf{s}_2$ to the right and obtain

$$\mathbf{s}_1'' = 00000000000$$
$$\mathbf{s}_2'' = 00000111111.$$

Again, we use Lemma B.1 , for which we have to show that reordering the positions of the strings does not change the solvability.

Lemma (B.3)**.** *Each* $\pi \in \mathrm{Sym}_{[D]}$ *is isometric with respect to* d_H *and reversible. Here, a permutation* π *is considered isometric with respect to* d_H *if and only if for all* $\mathbf{s}_1, \mathbf{s}_2 \in \{0,1\}^D$

$$d_H(\pi\mathbf{s}_1, \pi\mathbf{s}_2) = d_H(\mathbf{s}_1, \mathbf{s}_2).$$

PROOF. The invertibility of permutations is well-known. The isometry follows from

$$\begin{aligned} d_H(\mathbf{s}_1, \mathbf{s}_2) &= \sum_i (\mathbf{s}_1 \oplus \mathbf{s}_2)_i = \sum_i (\mathbf{s}_1 \oplus \mathbf{s}_2)_{\pi(i)} \\ &= \sum_i (\pi(\mathbf{s}_1 \oplus \mathbf{s}_2))_i = \sum_i (\pi\mathbf{s}_1 \oplus \pi\mathbf{s}_2)_i = d_H(\pi\mathbf{s}_1, \pi\mathbf{s}_2) \end{aligned}$$

□

Using Lemma B.1 and Lemma B.2, we can assume that $\mathbf{s}_1 = \mathbf{0}$. Using Lemma B.1 and Lemma B.3, we can assume that for $\mathbf{s}_2$,

$$(\mathbf{s}_2'')_i \leq (\mathbf{s}_2'')_j, \text{ if } i < j$$

holds.

We can also change the input by applying the XOR operation to all strings, without changing the respective distances.

Lemma (B.4)**.** *Let $s, t, p \in \{0,1\}^D$, then*

$$d_H(s,t) = d_H(s \oplus p, t \oplus p).$$

Here, $s \oplus t$ stands for the XOR operation.

PROOF. We have

$$\begin{aligned} d_H(s,t) &= \sum_i s[i] \leftrightarrow t[i] \\ &= \sum_i s[i] \oplus p[i] \leftrightarrow t[i] \oplus p[i] \\ &= d_H(s \oplus p, t \oplus p) \end{aligned}$$

Here, the symbol $\leftrightarrow$ denotes the biconditional, also known as XNOR. □

Next we show that the hamming distance is not affected by the order of the columns.

Let $s, t \in \{0,1\}^D$ and $\pi \in \mathrm{Sym}_{[D]}$. We define the product

$$\lambda t := (t[\lambda(1)], t[\lambda(2)], \ldots, t[\lambda(D)]).$$

Lemma (B.5)**.** *Let $s, t \in \{0,1\}^D$, then*

$$d_H(s,t) = d_H(\lambda s, \lambda t).$$

PROOF. We have $d_H(s,t) = \sum_i s[i] \leftrightarrow t[i]$, where $\leftrightarrow$ denotes the biconditional (the negation of the XOR operation). The lemma is hence equivalent to reordering the sum. □

List of Tables

List of Figures

Nomenclature

$\oplus$	Binary addition; that is, the addition on the vector space over F_2^D
$\leftrightarrow$	The biconditional, also known as XNOR.
$\#$	Cardinality of a set.
$\circ$	Concatenation of permutations. See Section 1.4.5.
$\mathbb{E}$	The expected value.
$-$	The set complement. For two sets A, B, $A - B$ is all of A except all of B.
$[n]$	For a natural number $n \in \mathbb{N}$, the set $[n]$ is defined to be $\{1, \ldots, n\}$
$\binom{n}{k}$	The binomial coefficient n choose k.
$\subset$	We will write $A \subset B$ to express that A is a non-strict subset of B, sometimes written as $A \subseteq B$.
$\mathrm{Sym}_{[n]}$	The symmetric group on $[n] = \{1, 2, \ldots, n\}$. I. e., all permutations of length n
$\mathrm{Sym}_{R,[n]}$	The permutations of length n which follow the set of restrictions R. See Section 4.2.

Bibliography

[1] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach (Draft)*. January 2007. 17

[2] Kenneth Joseph Arrow. *Social Choice and Individual Values*. Wiley, New York, 1951. 15, 16

[3] Jørgen Bang-Jensen and Gregory Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer Monographs in Mathematics. Springer-Verlag, London, second edition edition, December 2008. 11, 47

[4] Amir Ben-Dor, Giuseppe Lancia, R. Ravi, and Jennifer Perone. Banishing bias from consensus sequences. In *CPM '97: Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching*, pages 247–261, London, UK, 1997. Springer-Verlag. 79

[5] Nadja Betzler, Michael R. Fellows, Jiong Gong, Rolf Niedermeier, and Frances A. Rosamond. Computing Kemeny rankings, parameterized by the average KT-distance. In Ulle Endriss & Paul W. Goldberg, editor, *Proceedings of the Second International Workshop on Computational Social Choice (COMSOC-2008)*, pages 85–96. Department of Computer Science University of Liverpool, 2008. http://www.csc.liv.ac.uk/ pwg/COMSOC-2008/. 19, 32

[6] Nadja Betzler, Michael R. Fellows, Jiong Gong, Rolf Niedermeier, and Frances A. Rosamond. How similarity helps to efficiently compute Kemeny rankings. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS-2009)*, 2009. 32

[7] Nadja Betzler, Michael R. Fellows, Jiong Guo, Rolf Niedermeier, and Frances A. Rosamond. Fixed-parameter algorithms for kemeny scores. In *AAIM*, pages 60–71, 2008. 19, 21, 29, 31, 34, 35, 89

[8] Johannes Bisschop. *AIMMS - Optimization Modeling*. Lulu.com, 2006.

[9] Peter A. Bloniarz, Michael J. Fischer, and Albert R. Meyer. A note on the average time to compute transitive closures. In *ICALP*, pages 425–434, 1976. 58

[10] Miklós Bóna. *Combinatorics of Permutations*. Chapman & HALL/CRC, 2004. 10, 12, 64, 69

[11] Liang T. Chen. Developing applications for parallel computing. http://developers.sun.com/solaris/articles/parallel.html, 12 2005. 10

[12] G. Cohen and M. Deza. Distances invarantes et l-cliques sur certains demigroups finis. *Graph and Combinatorics*, 5:49–69, 1989. 45

[13] Marquis de Condorcet. *Essai sur l'application de l'analyse à la probabilité des décisions rendues à la pluralité des voix*. Paris: Imprimerie royale, 1785. 1

[14] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM. 58

[15] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001. 47

[16] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, Ravi Kannan, Jon Kleinberg, Christos Papadimitriou, Prabhakar Raghavan, and Uwe Schöning. A deterministic $(2-2/(k+1))n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 289(1):69–83, 2002. 89, 90

[17] Michael Deza and Tayuan Huang. Metrics on permutations, a survey. *Journal of Combinatorics, Information and System Sciences*, 23:173–185, 1998. 45

[18] R.G. Downey and M.R. Fellows. *Parameterized Complexity.* Springer-Verlag, 1997. 3, 8, 20, 39

[19] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW '01: Proceedings of the 10th international conference on World Wide Web*, pages 613–622, New York, NY, USA, 2001. ACM. 19, 89

[20] Cynthia Dwork, Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation revisited. Technical report, 2001. 19, 38

[21] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series).* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. 3, 8

[22] Moti Frances and Ami Litman. On covering problems of codes. *Theory of Computing Systems*, 30(2):113–119, 1997. 17

[23] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman, 1979. 8

[24] Jens Gramm, Rolf Niedermeier, and Peter Rossmanith. Fixed-parameter algorithms for closest string and related problems. *Algorithmica*, (37):25–42, 2003. 36, 42, 43, 88

[25] Jiong Guo and Rolf Niedermeier. Invitation to data reduction and problem kernelization. *SIGACT News*, 38(1):31–45, 2007. 20

[26] Jr. H. W. Lenstra. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8:538–548, 1983. 37, 88

[27] Paul R Halmos. *Naive Set Theory.* Springer-Verlag, New York, 1960. 46

[28] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 26(2):147–160, 1950.

[29] Frank Harary and Leo Moser. The theory of round robin tournaments. *American Mathematical Monthly*, (73):231–246, 1966. 48, 54

[30] Edith Hemaspaandra, Holger Spakowski, and Jörg Vogel. The complexity of kemeny elections. *Theoretical Computer Science*, 349(3):382–391, 2005. 19, 89

[31] Falk Hüffner. *Algorithms and Experiments for Parameterized Approaches to Hard Graph Problems.* PhD thesis, Institut für Informatik, Friedrich-Schiller-Universität Jena, 2007. 8

[32] J. Bartholdi III, C. A. Tovey, and M. A. Trick. Voting schemes for which it can be difficult to tell who won the election. *Social Choice and Welfare*, 6(2):157–165, April 1989. 19

[33] Joseph Jaja. *An Introduction to Parallel Algorithms.* Addison-Wesley Professional, March 1992. 10, 11, 12, 35, 45, 60

[34] Hakan Jakobsson. Mixed-approach algorithms for transitive closure (extended abstract). In *PODS '91: Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 199–205, New York, NY, USA, 1991. ACM. 58

[35] A. B. Kahn. Topological sorting of large networks. *Commununications of the ACM*, 5(11):558–562, 1962. 58

[36] Ravi Kannan. Minkowski's convex body theorem and integer programming. *Math. Oper. Res.*, 12(3):415–440, 1987. 37

[37] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972. 38

[38] John Kemeny. Mathematics without numbers. *Daedalus*, 88:577–591, 1959. 1

[39] Maurice Kendall and Jean D. Gibbons. *Rank Correlation Methods*. A Charles Griffin Title, 5 edition, September 1990. 6, 28

[40] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Pearson Education, Inc., 2006. 11

[41] Donald E. Knuth. *Art of Computer Programming, Volume 3: Sorting and Searching (2nd Edition)*. Addison-Wesley Professional, April 1998. 36

[42] J. Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, J. Kevin Lanctot, Ming Li, Bin Ma, Shaojiu Wang, and Louxin Zhang. Distinguishing string selection problems. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, pages 633–642, 1999. 17

[43] Michel Le Breton and Michel Truchon. A Borda measure for social choice functions. *Mathematical Social Sciences*, 34(3):249–272, October 1997. 2

[44] Bin Ma and Xiaoming Sun. More efficient algorithms for closest string and substring problems. In *Research in Computational Molecular Biology, Twelfth International Conference, RECOMB 2008*, pages 396–409, 2008. 5, 31, 49, 50, 60, 90

[45] Cláudio N. Meneses, Zhaosong Lu, Carlos A. S. Oliveira, and Panos M. Pardalos. Optimal solutions for the closest-string problem via integer programming. *INFORMS Journal on Computing*, 16:2004, 2004. 79

[46] X. Munoz. Fair ranking in dancesport competitions. In Michael H. Albert, editor, *Proceedings, Permutation Patterns*, number Technical Report OUCS-2003-02, pages 39–48, Dunedin, New Zealand, 2003. Department of Computer Science, University of Otago.

[47] Rolf Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006. 3, 8

[48] G. E. Noether. Why kendall tau? In Best of Teaching Statistics. Available at http://rsscse.org.uk/ts/bts/noether/text.html, 1986. 6

[49] James Noyes and Eric W. Weisstein. Linear programming. MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/LinearProgramming.html, March 2009.

[50] V. Yu. Popov. Multiple genome rearrangement by swaps and by element duplications. *Theoretical Computer Science*, 385(1-3):115–126, 2007. 2, 4, 8, 17, 22

[51] J.J Rousseau. *Du Contrat Social ou Droit Politique*. Strasbourg : De l'Impr. de la Societe typographique, 1791. 1

[52] J.J Rousseau. The social contract. London: J.M. Dent I& Sons, 1913. 1

[53] Claus-Peter Schnorr. An algorithm for transitive closure with linear expected time. *SIAM Journal on Computing*, 7(2):127–133, 1978. 58

[54] D. Sculley. Rank aggregation for similar items. In *SDM*. SIAM, 2007. 89

[55] Robert Sedgewick. *Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. 27, 59, 81

[56] Frederick Springsteel and Ivan Stojmenovic. *Parallel General Prefix Computations with Geometric, Algebraic, and Other Applications*, volume 18. Kluwer Academic Publishers, Norwell, MA, USA, 1990. 11

[57] Pantelimon Stanica. Good lower and upper bounds on binomial coefficients. *Journal of Inequalities in Pure and Applied Mathematics*, 2(3):Article 30, 11 2000. 75

[58] Lynn Arthur Steen and J. Arthur Seebach Jr. *Counterexamples in Topology.* Dover, 1995. 48

[59] Michel Truchon. Figure skating and the theory of social choice. Technical report, 1998. 1, 15, 17

[60] Michel Truchon. An extension of the Concordet criterion and Kemeny orders. Written 1998. (michel.truchon@ecn.ulaval.ca). 15, 16

[61] Anya Tsalenko, Roded Sharan, Hege Edvardsen, Vessela Kristensen, Anne-Lise Børresen-Dale, Amir Ben-Dor, and Zohar Yakhini. Analysis of SNP-expression association matrices. *Journal of Bioinformatics and Computational Biology*, 4:259–274, 2006.

[62] International Skating Union. Official rules summary. http://www.sportcentric.com/vsite/vfile/page/fileurl/0,11040,4844-181384-198602-118436-0-file,00.pdf. 3

[63] Vijay V. Vazirani. *Approximation Algorithms.* Springer, March 2004. 30

[64] Karsten Weihe. Covering trains by stations or the power of data reduction. In *Proceedings of Algorithms and Experiments (ALEX98)*, pages 1–8, 1998. 20

[65] E. W. Weisstein. Transitive closure. MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/TransitiveClosure.html. 54

[66] Eric W. Weisstein. Integer programming. MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/IntegerProgramming.html. 37

[67] H.P. Young and A. Levenglick. A consistent extension of Condorcet's election principle. *SIAM Journal of Applied Mathematics*, 35:285–300, 1978. 20