SORTING BY TRANSPOSITIONS: FIXED-PARAMETER ALGORITHMS AND STRUCTURAL PROPERTIES

by
Chris Whidden

SUBMITTED IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE OF
BACHELOR OF COMPUTER SCIENCE, HONOURS

AT

DALHOUSIE UNIVERSITY
HALIFAX, NOVA SCOTIA
DECEMBER 5, 2007

DALHOUSIE UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

The undersigned hereby certify that they have read and recommend to the Faculty of Graduate Studies for acceptance a thesis entitled "**Sorting by Transpositions: Fixed-parameter Algorithms and Structural Properties**" by **Chris Whidden** in partial fulfillment of the requirements for the degree of **Bachelor of Computer Science, Honours**.

Dated: December 5, 2007

Supervisor:	Dr. Norbert Zeh
Reader:	Dr. Robert Beiko

DALHOUSIE UNIVERSITY

Date: **December 5, 2007**

Author: **Chris Whidden**

Title: **Sorting by Transpositions: Fixed-parameter Algorithms and Structural Properties**

Department: **Computer Science**

Degree: **B.CSc. (Hon)** Convocation: **May** Year: **2008**

Table of Contents

List of Figures

Abstract

Molecular biologists study the evolutionary relationships of species by comparing their gene sequences. One approach is to compare the gene orders of their genomes, represented as a permutation, which evolve by inversions and transpositions, as well as the deletion, insertion, and duplication of fragments. Recent years have seen much work on the problem of SORTING BY TRANSPOSITIONS, which is related to the evolution of genomes through transpositions. However, It is not known if the problem is *NP-hard* or admits a polynomial-time solution.

This thesis examines several structural properties of SORTING BY TRANSPOSITIONS to aid understanding the problem and possibly lead to a complexity proof. A problem kernelization is given, which reduces the size of the permutation to at most $3k$, where k is the number of transpositions needed to sort the permutation. We show that some permutations can only be optimally sorted by inverting predecessor-successor pairs. We also show that any permutation can be sorted optimally with a transposition sequence that does not split any pair of elements more than once. The effect of adding or removing elements on the problem is examined. An algorithm to generate all optimal transposition sequences is provided based on this concept, but it currently has poor results. This also provides a parameter for induction that may be useful in proving results.

The problem is studied from the point of view of fixed-parameter tractability, where the objective is to obtain an efficient exact algorithm with a running time that is exponential in some parameter related to the input, but polynomial in the input size. We provide two FPT algorithms where the parameter is the length of a transposition sequence. Both rely on the kernelization provided. The first algorithm has a running time of $O(k^3 \cdot (3k)!)$, and examines every permutation of the correct size. The second algorithm has a running time of $O(k \log k \cdot (3k)^{1.5k})$, by using a double-ended search, which we show is faster than the first algorithm. Both algorithms have memory complexity of the same order as their time complexity.

Acknowledgements

I would like to thank Dr. Norbert Zeh, my Honours supervisor, for all of his help in researching and working on this problem, as well as his many helpful comments while I was writing this thesis.

Chapter 1

Introduction

Molecular biologists study the evolutionary relationships of species by comparing their gene sequences, using the reasonable assumption that species with more similar gene sequences are more closely related. Methods of sequence comparison are often done on DNA sequences of organisms to study their similarity, but a better approach may be to compare the gene orders of mitochondrial DNA, instead of comparing DNA sequences. Mitochondrial DNA shares many genes among different organisms and is small enough to be extensively sequenced by current technology.

This method was examined by Sankoff et al [11]. They examined several mitochondrial genomes with 30–50 genes, such as various yeasts, chicken, and sea urchins. Comparisons between genomes with this many genes have used many simplifications, as exact comparisons would require an enormous amount of computation. While the distances between these genomes are often 10 or more genome rearrangement operations, closely related genomes with few rearrangement operations may be able to be compared exactly by using an algorithm that requires computation proportional to the number of genome rearrangement operations instead of the genome length.

Gene order comparisons may be particularly useful for organisms with highly divergent genomes, which may be closely related yet have very different gene sequences. Genomes evolve by inversions and transpositions, as well as the deletion, insertion, and duplication of fragments. Some genomes are thought to evolve almost exclusively by inversions (for example plant mitochondrial DNA) [10], and some by both inversions and transpositions (such as the Herpes virus).

The problem of Sorting By Reversals, related to the evolution of genomes through inversions, has been extensively studied [1]. The problem has been proven to be NP-Hard [4], and has an approximation algorithm with performance guarantee 1.375 [3]. However, the oriented version, where genes are represented as signed permutations, has a linear-time algorithm. [7]

Recent years have seen much work on the similar problem of SORTING BY TRANSPOSITIONS, which is related to the evolution of genomes through transpositions [2]. An approximation algorithm with a performance guarantee of 1.375 is known [6]. This seems to be a more difficult problem to analyze, however, as the complexity of the problem is currently unknown. In particular, it is not known if the problem is *NP-hard* or admits a polynomial-time solution.

This thesis examines several structural properties of SORTING BY TRANSPOSITIONS to aid understanding the problem and possibly lead to a complexity proof. A problem kernelization is given, which reduces the size of the permutation that must be examined. We prove that some permutations can only be optimally sorted by inverting predecessor-successor pairs. We also prove that any permutation can be sorted optimally with a transposition sequence that does not split any pair of elements more than once. The effect of adding or removing one element to the problem is also examined. An algorithm to generate all optimal transposition sequences is examined, based on this concept, but it currently has very poor results. Adding and removing elements also provides a basis for induction that may be useful in proving results.

The problem is also studied from the point of view of fixed-parameter tractability, where the objective is to obtain an efficient exact algorithm with a running time that is exponential in some parameter related to the input, but polynomial in the input size. We provide two algorithms where the parameter is the transposition distance. The first, based on a breadth-first search of a transposition graph, has a factorial running time. The second uses a double-ended search and has an exponential but provably faster running time.

1.1 Sorting by Transpositions

This section defines concepts and introduces notation that is necessary to discuss the results in this thesis. These concepts are based on the material from [2].

A *permutation* $S = x_1 x_2 \ldots x_n$ of the numbers 1 to n represents the order of genes in a genome. To represent the beginning and end of the genome we often prepend

0 and append $n+1$ to the permutation. Circular permutations are also commonly used, as sorting a linear permutation of length n is equivalent to sorting a circular permutation of length $n+1$ [8]. We say that x_i and x_{i+1} are *adjacent* for $1 \leq i < n$. The *identity permutation*, I, of length n is the sorted permutation $I = 1, 2, \ldots, n$

Given a permutation $S = x_1 x_2 \ldots x_n$ and integers $0 \leq i < j < k \leq n$ the *transposition* $\tau^{i,j,k}$ transforms S into the permutation

$$(x_1 x_2 \ldots x_i)(x_{j+1} x_{j+2} \ldots x_k)(x_{i+1} x_{i+2} \ldots x_j)(x_{k+1} x_{k+2} \ldots x_n)$$

that is, the subsequences $(x_{i+1} x_{i+2}\ x_j)$ and $(x_{j+1} x_{j+2} \ldots x_k)$ are swapped by $\tau^{i,j,k}$.

We say that a transposition $\tau^{i,j,k}$ *joins* element x_i with x_{j+1}, x_k with x_{i+1}, and x_j with x_{k+1}. We also say that $\tau^{i,j,k}$ *splits* element x_i from x_{i+1}, x_j from x_{j+1}, and x_k from x_{k+1}.

This leads to the Transposition Distance Problem, where the objective is to find a minimal length sequence of transpositions that act on one permutation to give another permutation.

Transposition Distance Problem
Input: Two permutations $S_1 = x_1 x_2 \ldots x_n$ and $S_2 = x_1 x_2 \ldots x_n$
Goal: Find a minimal length sequence $T = \tau_1 \tau_2 \ldots \tau_k$ of *transpositions* such that $(\tau_k \circ \tau_{k-1} \circ \ldots \circ \tau_1)(S_1) = S_2$

The length of T is the *transposition distance* between S_1 and S_2. It is important to note that the transposition distance between S_1 and S_2 is the same as the transposition distance between $S_2^{-1} \circ S_1$ and the identity permutation I. Then any instance of the Transposition Distance Problem becomes an equivalent instance of the following problem:

Sorting by Transpositions
Input: A permutation $S = x_1 x_2 \ldots x_n$
Goal: Find a minimal-length sequence $T = \tau_1 \tau_2 \ldots \tau_k$ of *transpositions* such that $(\tau_k \circ \tau_{k-1} \circ \ldots \circ \tau_1)(S) = I$ We say that T *sorts* S.

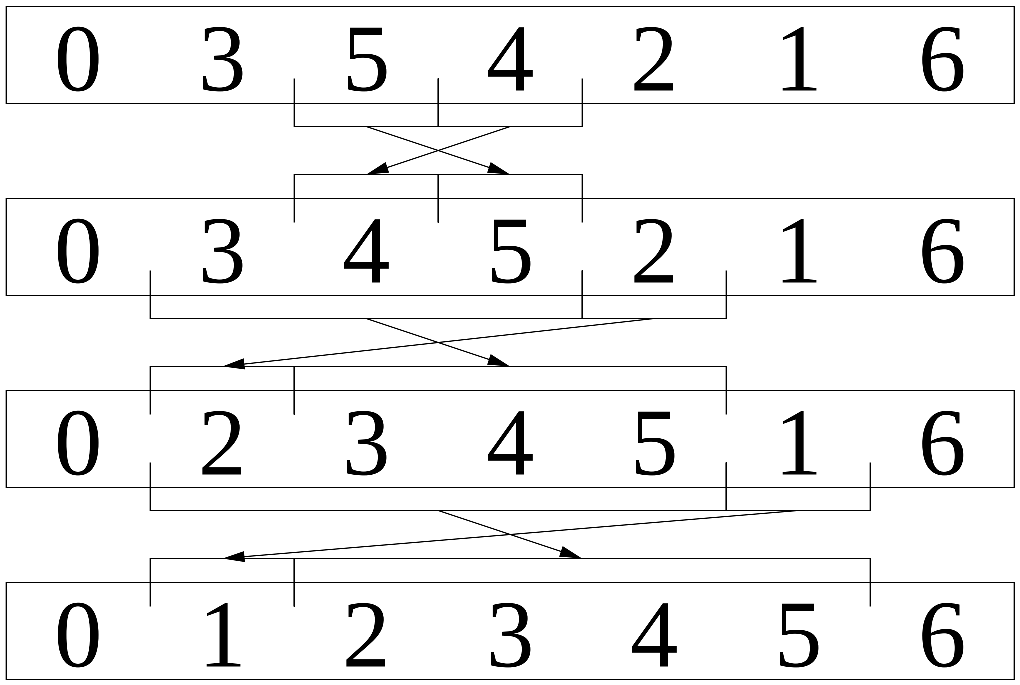

Figure 1.1: A Sorting by Transpositions example

An example is shown in figure 1.1. As a simplification, for the remainder of this thesis we will only consider the problem of Sorting by Transpositions with the understanding that results are generalizable to the Transposition Distance Problem.

A *breakpoint* of a permutation $S = x_1 x_2 \ldots x_n$ is a pair (x_i, x_{i+1}) where $x_{i+1} \neq x_i + 1$. Breakpoints are adjacent elements that are not a predecessor-successor pair. The identity permutation is the only permutation with 0 breakpoints. Then a natural way to see the problem is as reducing the number of breakpoints to 0.

A *p-transposition* is a transposition that changes the number of breakpoints by p.

Lemma 1.1. *Any transposition is a p-transposition with* $-3 \leq p \leq 3$*.*

Proof. Three locations in a permutation are affected by a transposition, so a transposition can change the number of breakpoints by at most 3. □

One useful tool that has aided the analysis of the problem is the notion of a cycle graph [2]. An example is shown in figure 1.2. A directed edge-colored *cycle graph*

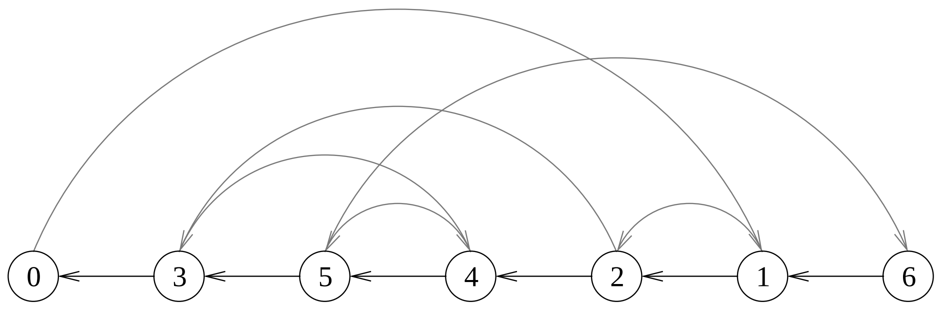

Figure 1.2: A cycle graph example

of S, denoted by $G(S)$, is the graph with vertex set $\{0, 1, 2, \dots, n+1\}$ and edge set defined as follows. For all $1 \leq i \leq n+1$, *gray edges* are directed from $i-1$ to i, and *black edges* from s_i to s_{i-1}.

An *alternating cycle* of $G(S)$ is a directed cycle where the edges alternate colours. We will simply refer to these as *cycles.* The *length* of a cycle is the number of black edges contained in it. As the identity permutation is the only permutation with $n+1$ cycles, sorting the permutation corresponds to increasing the number of cycles.

A *p-move* is a transposition that changes the number of cycles by p.

Lemma 1.2 (Bafna and Pevzner [2]). *Any transposition is a p-move with $p \in \{-2, 0, 2\}$.*

Note that the set of 2-transpositions and 3-transpositions is a subset of the set of 2-moves, since each adjacent predecessor-successor pair forms a cycle of length 1. Similarly, the set of (-2)-transpositions and (-3)-transpositions is a subset of the set of (-2)-moves.

1.2 Fixed Parameter Tractability

Fixed-parameter algorithms, similar to approximation algorithms, are a tool for solving NP-hard problems. The difference is that fixed-parameter algorithms provide exact solutions, which implies that exponential running times are unavoidable unless $P = NP$. The main idea of fixed-parameter algorithms is to restrict the combinatorial explosion in the running time to some function depending only on some parameter that is specific to the problem. It is hoped that in a real application of this problem, that parameter is relatively “small” so that the exponential growth is reasonable. The fixed-parameter algorithm then efficiently solves the given parameterized problem. A good resource on fixed-parameter algorithms is the book ”Invitation to Fixed-Parameter Algorithms” by Rolf Niedermier [9].

Parameterized complexity was formalized by Rod G. Downey and Michael R. Fellows and co-authors in the 1990’s [5]. They developed a theory of parameterized complexity which is a mathematical tool for fixed-parameter algorithms. The material studied in this thesis uses this theory where necessary but uses an application-oriented look at the use and design of fixed-parameter algorithms.

The advantages of fixed-parameter algorithms over approximation algorithms and heuristics are the guaranteed optimality of the solution and the provable upper bounds on the computational complexity. The disadvantage is that exponential running times are expected. Parameterized algorithm design searches for the cause of the hardness of a problem. The size of an instance and some parameter are examined to find whether the difficult part of the problem can be confined to the parameter. That parameter may be much smaller than the input size in practice.

Formally, a *parameterized problem* is a language $L \subseteq \Sigma^* \times \Sigma^*$, where Σ is a finite alphabet. The first component is an instance of the problem. The second component is the parameter of the problem and is often confined to $\mathbb{N}$. Thus every member of L is of the form (X, k), where k is the parameter. For Vertex Cover, for example, X is some encoding of the given graph G and k is the desired size of the vertex cover. Then (X, k) belongs to L if and only if G has a vertex cover of size k. Vertex Cover is examined below.

A parameterized problem L is *fixed-parameter tractable* if it can be determined in $f(k) \cdot n^{O(1)}$ time whether or not $(X, k) \in L$, where f is a computable function only depending on k. The corresponding complexity class is called FPT.

1.2.1 Example - Vertex Cover

Vertex Cover is a popular problem for fixed-parameter tractability research and examples. Many fixed-parameter approaches work on it and new approaches are often first tested with Vertex Cover.

Vertex Cover

Input: a graph $G = (V, E)$ and a non-negative integer k.

Goal: Find a subset of at most k vertices $C \subseteq V$ such that every edge $\{u, v\} \in E$ has at least one endpoint in C.

A simple observation leads to a fixed-parameter algorithm for Vertex Cover: Every edge $\{u, v\}$ must be covered by C. Thus, at least one of u and v must be in C. Then a search tree can be built where each node of the tree represents a partially solved version of the graph. The root of the tree is the original graph G and an empty cover C. At each node of the tree, branch to bring either u or v into C, removing that vertex and any incident edges from the graph, and continue recursively on the reduced graph.

Each instance of the problem in a search tree is a single node. A search tree with depth at most k has less than 2^{k+1} nodes, since the branching is done on both ends of an edge. Each node can be checked in linear time to determine if it is a solution. Therefore this algorithm requires $O(2^k \cdot n)$ time.

1.2.2 Creating a Fixed-parameter Algorithm

There are two main techniques that can be combined to create a fixed-parameter algorithm, *kernelization* and *depth-bounded search trees.*

Reducing a problem to a "problem kernel", or kernelization, uses data reduction rules to replace the original instance of a problem with a reduced instance and reduced parameter. This reduced instance is generally the difficult core of a problem.

For Vertex Cover, there is a simple reduction rule. Each edge must be covered in a vertex cover, so one of the two endpoints of every edge must be in the vertex cover. Then, for any vertex, either that vertex or all of its neighbours must be in a vertex cover. Since at most k vertices can be in the vertex cover, any vertex with degree at least $k + 1$ must be in the vertex cover, as not all of its neighbours can be. Then any such vertex can be put in the vertex cover, k can be reduced by one, and the vertex and its adjacent edges removed from the graph, without altering the solution. After reducing the problem with this rule, there can be at most k^2 edges and $k^2 + k$ vertices.

Let L be a parameterized problem, that is, L consists of input pairs (I, k), where I is the problem instance and k is the parameter. *Reduction to a problem kernel* means to replace (I, k) with a reduced instance (I', k') such that $k' \leq k$, $|I'| \leq g(k)$ for some function g only depending on k, and $(I, k) \in L$ iff $(I', k') \in L$. The reduction must be computable in polynomial time. $g(k)$ is the size of the problem kernel.

With data reduction and problem kernels, the focus is on polynomial-time preprocessing of the input. For NP-hard problems, there is usually no other method than an exhaustive search that can finally determine the optimal solution. Organizing the exhaustive search in a tree whose depth is bounded by some function depending only on the parameter gives a depth-bounded search tree.

One method of creating a depth-bounded search tree is to find in polynomial time a "small subset" of the input instance such that at least one element of the subset is guaranteed to be in the solution. For Vertex Cover, for example, the two vertices of an edge in a vertex cover are such a subset. This leads to the previously mentioned size-$O(2^k)$ search tree.

If there is a reduction to a problem kernel and a depth-bounded search tree method for a particular problem, we can reduce the problem and then apply the bounded search tree method. A method that may provide better results is to repeatedly kernelize the problem while applying the search tree method. Because the parameter decreases during the search tree, kernelization may remove more elements.

Chapter 2

Structural properties

2.1 Kernelization

There is a simple kernelization for SORTING BY TRANSPOSITIONS that limits the problem size to $3k$, where k is the transposition distance. Intuitively, it never helps to split two elements i and $i+1$ that are already adjacent. The kernelization requires the following lemma:

Lemma 2.1. *Given a permutation $S = x_1x_2 \dots x_n$ with a transposition distance of k, there always exists a transposition sequence $T = \tau_1\tau_2 \dots \tau_k$ of length k that sorts S in k transpositions and such that no transposition τ_i splits any elements x_j, x_{j+1} that satisfy $x_j + 1 = x_{j+1}$.*

Proof. The proof proceeds by induction on $|T|$. As a basis, consider $T = \tau_k$, that is, $|T| = 1$. The result of τ_k is I, which has no elements out of order. Then τ_k can not split any two elements i and $i+1$ that are already adjacent.

Assume the lemma is true for $|T| < k$, and S is a permutation that is sorted using k transpositions. Let $T = \tau_1\tau_2 \dots \tau_k$ be a transposition sequence that sorts S. Then by the inductive hypothesis, the length $k-1$ transposition sequence $\tau_2\tau_3 \dots \tau_k$ has an alternative transposition sequence $T' = \tau'_2\tau'_3 \dots \tau'_k$ that does not split any two elements x_j, x_{j+1} that satisfy $x_j + 1 = x_{j+1}$.

Consider the transposition sequence $T'' = \tau_1 T'$. If τ_1 does not split two elements that are already adjacent, then we are done. Otherwise, consider two elements x_j, x_{j+1} that satisfy $x_j + 1 = x_{j+1}$ that are split by τ_1. Since x_j and x_{j+1} are adjacent in I, and T' does not split x_j and x_{j+1} by the inductive hypothesis, T'' must split and join x_j, x_{j+1} exactly once. Let τ_f be the transposition that joins the pair.

We will now alter T'' such that a split occurs after x_{j+1} in any transposition that would split x_j, x_{j+1}. Construct $T''' = \tau''_1\tau''_2\tau''_3 \dots \tau''_k$ from T'' as follows: $\tau''_1 = \tau_1$ with one index changed so that τ''_1 splits x_{j+1}, x_{j+2} instead of x_j, x_{j+1}. For each subsequent τ''_i, split after x_{j+1} instead of x_j, x_{j+1} if those two elements are split.

We must show that T''' results in I. Consider how T'' and T''' each alter S. Each transposition τ''_i of T''' has the same effect as the corresponding transposition τ'_i of T'' except that x_j and x_{j+1} remain adjacent. Then after τ''_f, where x_j, x_{j+1} would be joined by τ'_f, T''' and T'' result in the same permutation. Then the result of both is I.

However, T''' may still fail to satisfy the lemma if x_{j+1}, x_{j+2} are adjacent in I. Since the above alterations always increase the indices of τ_1, this process can be repeated a finite number of times, resulting in a transposition sequence of length k where the first transposition satisfies the lemma. The alterations may have caused a later transposition to fail the lemma, but we can once again apply the inductive hypothesis to give a transposition sequence of length k that satisfies the lemma. □

Lemma 2.1 implies the following reduction rule, R_1: Let $S = x_1 x_2 \ldots x_n$ be a permutation such that x_j, x_{j+1} are two elements where $x_j + 1 = x_{j+1}$. Replace x_j and x_{j+1} with a single element x'_j with the predecessor of x_j and the successor of x_{j+1} to give S'. A transposition sequence that sorts S can be created from a transposition sequence that sorts S' by replacing x'_j with x_j and x_{j+1}.

Theorem 2.2. *Reducing an instance of* Sorting by Transpositions *with R_1 gives a problem kernel of size $3k$, where k is the transposition distance of the instance.*

Proof. By Lemma 1.1, at most 3 elements of a permutation can be joined by one transposition. Then a permutation with a transposition distance of k can have at most $3k$ breakpoints. Reducing a permutation S with R_1 results in an equivalent permutation, S', where every element boundary is a breakpoint. The size of S' is at most $3k$. □

This reduction gives a linear-size problem kernel that will be very important for our FPT results.

2.2 Splitting Pairs of Elements

Since the problem of sorting one permutation into another is equivalent to Sorting by Transpositions, it is interesting to consider Lemma 2.1 applied to the intermediate results of a transposition sequence. This leads to the following Theorem:

Theorem 2.3. *Given a permutation $S = x_1x_2 \dots x_n$, there exists an optimal transposition sequence that splits any pair of elements at most once.*

Proof. Let $S = x_1x_2 \dots x_n$ be a permutation. Let $T = \tau_1\tau_2 \dots \tau_k$ be an optimal transposition sequence on S. We will show how to convert T into an optimal transposition sequence that splits any pair of elements at most once. Denote by S_i the permutation before τ_i.

Consider two elements x and y that are split, joined, and split again at transpositions τ_a, τ_b, and τ_c. Now, consider the problem of sorting S_a to S_{b+1}. x and y are adjacent in both S_a and S_{b+1}, so by Lemma 2.1, x and y can be treated as one element if we were to sort S_a to S_{b+1}. Similar to the proof of Lemma 2.1, there is a transposition sequence T' that can be constructed from T that keeps x and y together up to S_c, increases one index of τ'_a and otherwise splits and joins the same pairs. Subsequent permutations will have xy where x was previously, at least for permutations up to S_{b+1}.

Applying this to join one such pair may cause another pair to be split twice. Repeatedly applying this to suitable elements can give a transposition sequence T' that splits any pair of elements at most once. Some care must be taken in how this is done, however.

To analyze repeated applications of this, we will store each pair that is joined this way, for each transposition. Let L be a list of 4-tuples (x, y, i, j) where x and y are a pair of elements, i is the first permutation generated by T that has x and y adjacent, and j is the last permutation generated by T where x and y are adjacent. Then each such 4-tuple represents $(j - i + 1)$ joined pairs. Each time we alter T to T' by keeping elements x and y together, we add (x, y, i, j) to L. Then L reflects the pairs of elements that we have grouped together.

Our first chosen pair is simply kept together, and the corresponding 4-tuple is all that L contains. Now, consider how keeping elements x and y together and thus adding (x, y, i, j) to L affects pairs that have already been grouped in L. We will show that the number of joined pairs in L always increases. Then this process can only be repeated a finite number of times. In the following, any elements that are represented with different letters are not the same element. Figures are shown for important cases as follows: the left box represents a set of existing groupings, the second box represents the new grouping, and the final box shows the result from carrying out that case.

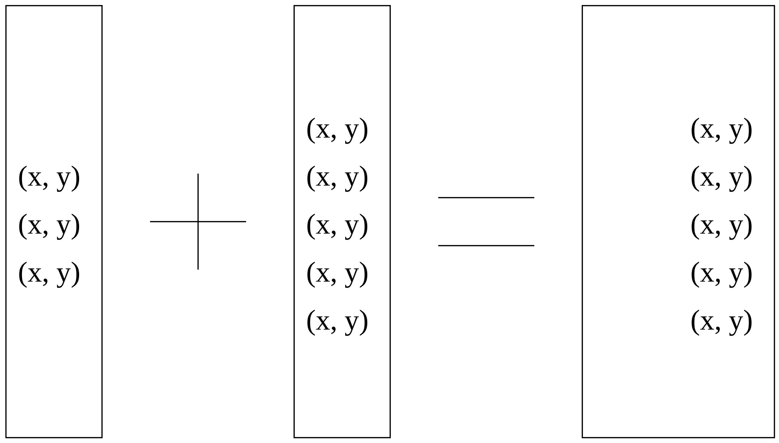

Figure 2.1: Proof of Theorem 2.3: Case 1

1. $(x, y, k, l) \in L$

   The new grouping supersedes this grouping, and will only occur if some change has split x, y before S_k or joined x, y after S_l. In either case, the new grouping encompasses the previous one which can be removed from L.

2. $(p, q, k, l) \in L$

   Grouping x and y only affects pairs of elements adjacent to x or y, so no change occurs.

3. $(a, y, k, l) \in L$

   S_i and S_j must have the pair x, y, so there are three subcases:

   (a) $i < k$ and $j > l$

Grouping elements x and y as specified will remove the a, y grouping, so it is deleted from L.

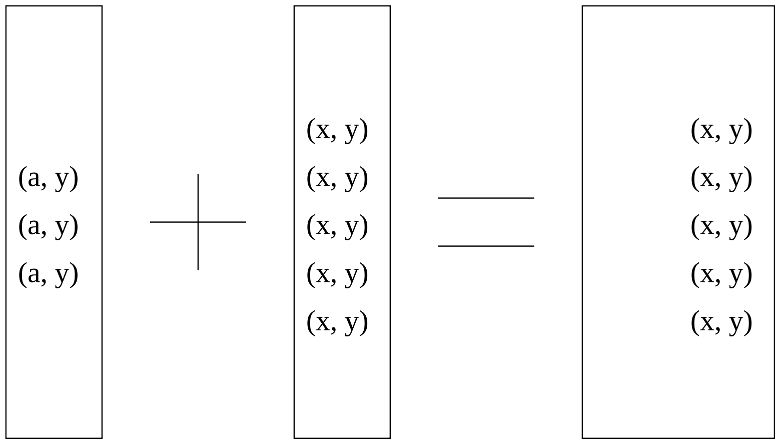

Figure 2.2: Proof of Theorem 2.3: Case 3a

(b) $i < k$ and $j < k$

The groupings do not affect the same transpositions so no change occurs.

(c) $i > l$ and $j > l$

The groupings do not affect the same transpositions so no change occurs.

4. $(y, b, k, l) \in L$

(a) $i < k$ and $j > l$

The x, y grouping encompasses the old y, b grouping which is deleted from L.

(b) $i < k$ and $j < k$ or $i > k$ and $j > k$

The two groupings are independent so no change occurs.

(c) $i < k$ and $k < j < l$

Element y is moved with x but b is also moved at some index m where $k \leq m \leq j$. This is because the pairs x, y and y, b are adjacent at index j, due to the y, b grouping existing and the x, y grouping being necessary. Then the y, b grouping is altered to (y, b, k, m).

(d) $k < i$ and $i < l < j$

Symmetric with the previous subcase.

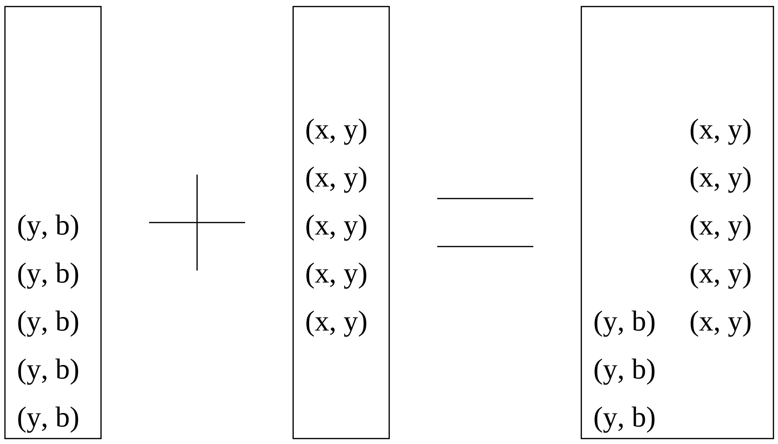

Figure 2.3: Proof of Theorem 2.3: Case 4c

If there previously existed a, y and y, b groupings, then an a, b grouping is created, where the y, b grouping was diminished. This is because element y was moved from between elements a and b.

5. $(x, c, k, l) \in L$

   S_i and S_j must have the pair x, y, so there are three subcases:

   (a) $i < k$ and $j > l$

   Grouping elements x and y as specified will alter the x, c grouping to (y, c, k, l). This altered grouping must be tested against all cases when x, y is finished to move any groupings that applied on the left to c to apply to x.

   (b) $i < k$ and $j < k$

   The groupings are independent and no change occurs.

   (c) $i > l$ and $j > l$

   The groupings are independent and no change occurs.

6. $(d, x, k, l) \in L$

   Pairings to the left of x are not altered by the way we specified keeping x and y together, so there are no changes.

Each 4-tuple (x, y, i, j) denotes $(j-i+1)$ joined pairs in the transposition sequence. One transposition can only have two pairs involving one element, and there can be

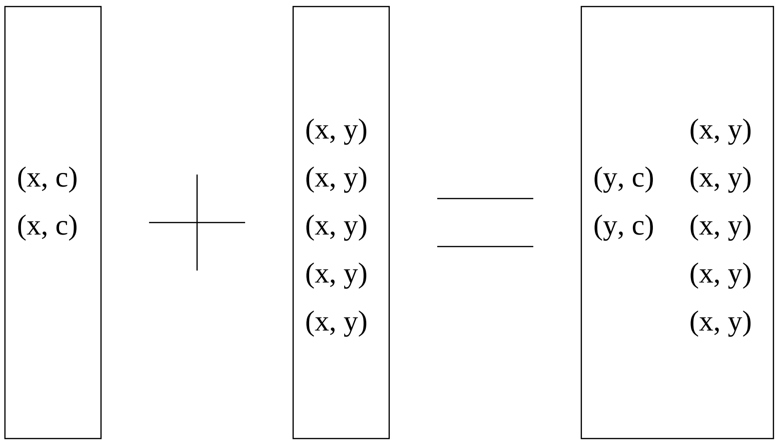

Figure 2.4: Proof of Theorem 2.3: Case 5a

only one grouping for a pair, as no case will split a grouping in two or add a second grouping for a pair already in L. The only types of groupings that can be diminished are a, y groupings encompassed by x, y in Case 3a or y, b groupings in Case 4c. If both occur then the a, y grouping is altered to an a, b grouping, so both occuring still increases the number of joined pairs. Also, altered groupings from x, c to y, c in Case 5a may require updating any c, d groupings since c has moved. Since only the d can be moved by this update and the c has already been tested, this can only occur a finite number of times and will not decrease the number of groupings. Then adding an x, y grouping increases the number of joined pairs.

If there is a pair of elements x and y that are split and joined by T' then they can be grouped. Then since each case increases the number of joined pairs and there can be only finitely many pairs joined in a finite number of transpositions, repeatedly applying this process will result in a transposition sequence that does not join any pair that it splits. □

Theorem 2.3 provides an interesting result that could potentially reduce the subproblems that need to be considered when solving an instance of SORTING BY TRANSPOSITIONS. However, we do not as of yet have any useful results using it.

2.3 Creating Inversions

One concept that has been useful in the development of sorting algorithms is that of *inversions*. There is an inversion between two elements of a permutation if they are out of order. Then two elements x_i and x_j are inverted if $x_i > x_j$ and $i < j$.

One may wonder if there is always an optimal transposition sequence that does not create new inversions. This seems to be a fairly strong claim, so we first consider only inversions between two elements adjacent in I. Then two elements x_i and x_j are inverted if $x_i + 1 = x_j$ and $i > j$.

With this definition of inversions, the claim becomes that there is always an optimal transposition sequence that does not create new inversions between elements adjacent in I. This claim holds for permutations with transposition distance at most 2, but does not hold for all permutations with transposition distance 3. Consider the permutation $S = 0426183759$. There are only two optimal transposition sequences for this permutation:

1. (0)(426)(183)(759) → (0)(183)(426)(759)
   (01)(834)(267)(59) → (01)(267)(834)(59)
   (012)(678)(345)(9) → (012)(345)(678)(9)

2. (042)(618)(375)(9) → (042)(375)(618)(9)
   (04)(237)(561)(89) → (04)(561)(237)(89)
   (0)(456)(123)(789) → (0)(123)(456)(789)

Clearly the first transposition of transposition sequence 1 inverts elements 2 and 3, while the first transposition of transposition sequence 2 inverts elements 6 and 7.

Note that this permutation is sorted optimally by three 3-transpositions, such that the indices of any two of the transpositions interleave. Also note that if transposition sequence 1 is denoted $T_1 = \tau_1\tau_2\tau_3$ then transposition sequence 2 is $T_2 = \tau_3\tau_2\tau_1$, the inverse of transposition sequence 1. It may still be interesting to characterize permutations that do not allow a transposition sequence which does not create inversions between elements adjacent in I. Another question to examine is whether it is ever necessary to increase the number of inversions.

2.4 Building Permutations One Element at a Time

One method we have used for proving results about Sorting by Transpositions is induction on the length of optimal transposition sequences. Another proof technique that may be useful is induction on the length of permutations. To examine permutations in this manner, we must consider what happens when we add or remove an element from a permutation. We define these operations as follows: Let $S = x_1 x_2 \dots x_n$ be a permutation.

Removing an element x_i creates a new permutation $(S - x_i) = x'_1 x'_2 \dots x'_{n-1}$ such that elements larger than x_i are decreased by one, and elements x_j to the right of x_i are now referred to as $x'_{(j-1)}$.

Adding an element x_i creates a new permutation $(S + x_i) = x'_1 x'_2 \dots x'_{n+1}$ such that elements as large or larger than x_i are increased by one, and elements x_j to the right of x_i are now referred to as $x'_{(j+1)}$.

Lemma 2.4. *Removing an element from a permutation is equivalent to the transposition that moves that element directly before its successor*

Proof. Let $S = x_1 x_2 \dots x_n$ be a permutation. Let $(S - x_i) = S$ with element x_i removed. Consider $S' = S$ after transposition $\tau^{(i-1),i,(j-1)}$ or $\tau^{(j-1),(i-1),i}$ that moves x_i directly before its successor x_j. $S' = x_1 x_2 \dots x_{(j-1)} x_i x_j x_{(j+1)} \dots x_n$. By Lemma 2.1 we can reduce S' to $(S - x_i)$, since x_i and x_j are an adjacent predecessor-successor pair. Then any transposition sequence that sorts S' has an equivalent transposition sequence that sorts $(S - x_i)$ and vice versa. $\square$

Moving a single element directly before its successor can either decrease the transposition distance by one or keep it the same, since any transposition sequence that sorts permutation S has an equivalent transposition sequence that sorts $(S - x_i)$, possibly with an empty transposition that corresponds to moving x_i directly before its successor or after its predecessor. Let k be the transposition distance of S. This gives the following two cases:

1. Removing element x_i does not change the transposition distance

2. Removing element x_i decreases the transposition distance by one

By the same reasoning, we have analogous cases for adding a single element:

1. Adding element x_i does not change the transposition distance

2. Adding element x_i increases the transposition distance by one

In case 1 for adding an element, since adding x_i does not change the transposition distance, x_i must be carried around inside transpositions while either staying or moving with transpositions that border it, in some optimal transposition sequence for S, to give an optimal transposition sequence for $(S + x_i)$.

In case 2, it must be optimal to move element x_i directly before its successor with transposition τ_x, since τ_x will give a permutation that is equivalent to S and then S can be sorted in k transpositions. Note that optimal transposition sequences for $(S + x)$ in case 2 can also occur by having x_i move along the boundaries of transposition sequences of length $k + 1$ that sort S.

These results lead to the following algorithm to solve the problem, based on constructing all optimal transposition sequences. for a permutation S:

Consider permutation T, initially empty. Maintain a tree Q of transposition sequences for T. For each element i of S, $1 \leq i \leq n$, add element i to T such that T is a subsequence of S. If T is sorted, continue to the next element. Otherwise, update the transposition sequences in Q to reflect the new element. Branch whenever a transposition could either move i or not. if any of these new transposition sequences sort T and have the same length as an optimal transposition sequence before adding element i, then we have case 1 and each of those is optimal. Otherwise, we can construct all optimal transposition sequences by moving i directly before its successor or after its predecessor. Unfortunately, we must branch and do this at every possible location in Q.

Based on the cases above, this will generate all optimal transposition sequences for S. However, the running time will be quite large. Each branch for Case 1 can double the size of Q for each level of Q. Additionally, the branches for Case 2 can increase the size of Q by the depth of Q, since i can be moved individually at any level of a transposition sequence.

We had hoped that non-optimal transposition sequences could be pruned from the tree, which would greatly limit its size, but since non-optimal transposition sequences can become optimal when a new element is added, both Case 1 and Case 2 must be explored at every step, and every generated transposition sequence must be stored. If that obstacle can be overcome, this algorithm could potentially be quite efficient.

As previously mentioned, the effects of adding and removing elements from permutations are still useful. They could possibly be used for considering Sorting by Transpositions along with other genome rearrangement operations. They can also be used with induction to prove results about the problem. Adding an element to a permutation changes the transposition distance by either one or zero, which may be very useful in an inductive proof.

Chapter 3

FPT results

3.1 An Algorithm Based on Kernelization

The kernelization from Theorem 2.2 can be used to give a fixed-parameter algorithm for Sorting by Transpositions. Recall that the kernelization gives a problem kernel of size $3k$, where k is the transposition distance of a problem intance. Also, note that there are $n!$ different permutations of size n. Then after reducing an instance of the problem of size n, there are at most $3k!$ different permutations that are possible.

We can create a transposition graph of permutations, where each vertex is a permutation of size n and two vertices have an edge if the corresponding permutations can be transposed to one another. We can then do a breadth-first search of the permutation graph, starting at S and looking for I. We will examine the algorithm from the perspective that S is of size n for simplicity and generality. Then we will apply the kernelization before the algorithm and examine the results. This suggests the following algorithm for finding the transposition distance:

Let S be a permutation. Let G be a graph of permutations, the permutation graph. Let $d(M)$ be the transposition distance from S to M. Let L be a list of permutations that have not been visited yet, ordered by $d(M)$.

PERMUTATION-GRAPH-SBT(S)

```
$G = (V, E)$
$V$ = every permutation of the same size as S.
$d(S) = 0$
$L = \{S\}$
VISITED = $\emptyset$
while ($L \neq \emptyset$)
    $P$ = dequeue($L$)
    add $P$ to VISITED
    foreach $M$ that $P$ can be transposed to
        if $M \notin$ VISITED
            enqueue $M$ in $L$
            $d(M) = d(P) + 1$
            add $(P, M)$ to $E$
return $d(I)$
```

This algorithm examines at most each permutation of the same size as S. Permutations are examined in order of their distance from S, so every permutation at distance i from S will be examined before every permutation at distance $i+1$. Then a new permutation M adjacent to P that is not yet in G must be one farther from S than P. Then when the identity permutation I is found, $d(I)$ gives the transposition distance of S. The transposition sequence can be recovered from the edges of E. This is a breadth-first search of the transposition graph of permutations that S can be transposed to. Newly found permutations are added to L, and only permutations from L are examined, so each permutation is examined once. Thus the algorithm will terminate after examining each permutation once, and correctly gives the transposition distance of S.

Because permutations are examined in order of their distance from S, new permutations can be added to the end of L and L will remain sorted by d. Then L can be implemented as a linked list, and adding elements to L or removing the first element of L require costant time. Because the algorithm may examine every permutation eventually, VISITED can be an array of size $n!$ and checking if a permutation is in VISITED requires constant time. Then the running time of this algorithm relies solely on the number of permutations examined and the time required to examine each. There are $\binom{n}{3}$ transpositions from each permutation, which is $O(n^3)$, so the time complexity of the algorithm is $O(n^3 \cdot n!)$. The space required is dominated by the size of G which is $O(n!)$.

Applying our kernelization to the permutation gives a permutation that is at most $3k$ elements long, so if we first apply our kernelization, the algorithm has a running time of $O((3k)^3 \cdot (3k)!)$, which is $O(k^3 \cdot (3k)!)$, and requires $O((3k)!)$ space.

This algorithm can be optimized in several ways which may improve performance but do not improve the worst-case time complexity. First, the algorithm can return as soon as I is found, since $d(I)$ is known then. Second, instead of preallocating G as an array, G can be a hash table or other dictionary. The second change would actually increase the time complexity, due to dictionary accesses, but the entire permutation set would not necessarily be examined and stored.

Note that this algorithm solves the optimization problem for Sorting by Transpositions and uses the transposition distance as the parameter. We can use this algorithm to solve the relevant decision problem: does a permutation have a transposition distance at most k?

```
FPT-PERMUTATION-GRAPH-SBT($S, k$)
$S'$ = KERNELIZE($S$)
if ($k < 1/3 \cdot |S'|$)
    return NO
$d$ = PERMUTATION-GRAPH-SBT($S'$)
if ($d <= k$)
    return YES
else
    return NO
```

Theorem 3.1. Sorting by Transpositions *can be solved in* $O(k^3 \cdot (3k)!)$ *time and* $O((3k)!)$ *space and is in FPT.*

Proof. From the kernelization, we know that the transposition distance of S is at least $1/3 \cdot |S'|$, since one transposition can eliminate at most 3 breakpoints. Then we know that the answer is NO if k is less than $1/3 \cdot |S'|$. Otherwise, we can use **PERMUTATION-GRAPH-SBT** to find the transposition distance and compare it to the given k. Since **PERMUTATION-GRAPH-SBT** is only used when $n <= 3k$, our previous analysis holds and we have a fixed-parameter algorithm for Sorting by Transpositions with a running time of $O(k^3 \cdot (3k)!)$ which requires $O((3k)!)$ space. Thus the problem is in FPT. □

3.2 A Faster Algorithm Using a Bounded Search Tree

The analysis of **FPT-PERMUTATION-GRAPH-SBT** relied on the fact that there are $n!$ permutations of size n. A more naive method that may allow improvements is a a naive bounded search tree algorithm. With this we examine every permutation that can be transposed to with k transposition from S.

Let S be a permutation and k be a nonnegative integer. Let T be a bounded-depth search tree, initially empty, where each node of the tree will have a permutation and its distance from S. Let I be the identity permutation.

```
FPT-NAIVE-SBT($S, k$)
$S' =$ KERNELIZE($S$)
if ($k < 1/3 \cdot |S'|$)
    return NO
$T = \{(s, 0)\}$
return FPT-NAIVE-EXAMINE-NODE($(S, 0), k$)
```

```
FPT-NAIVE-EXAMINE-NODE($(S, d), k$)
if($d > k$)
    return NO
if($S = I$)
    return YES
for each permutation $v$ that $S$ can be transposed to
    add $(v, d + 1)$ to $T$ as a child of $(S, d)$
    if(FPT-NAIVE-EXAMINE-NODE($(v, d + 1), k$) = YES)
        return YES
return NO
```

These two subroutines detail a recursive algorithm that creates a depth-bounded search tree for the problem. The depth of the tree is at most k and each node branches into $\binom{n}{3}$ children, which is $O((3k)^3$ due to the kernelization. Then the size of the tree is $O(((3k)^3)^k) = O((3k)^{3k})$. Since every node does a constant amount of work other than the branching, this algorithm uses time and space $O((3k)^{3k})$.

This algorithm is asymptotically slower and requires more space than **FPT-PERMUTATION-GRAPH-SBT**. However, This type of search allows improvements.

We can improve the algorithm by using a double-ended search. To solve an instance of Sorting by Transpositions with permutation S and distance k, use **FPT-NAIVE-SBT** to generate a depth-bounded search tree of depth $\lceil k/2 \rceil$ starting from S and a depth-bounded search tree of depth $\lfloor k/2 \rfloor$ for the inverse problem of sorting I to S. Sorting I to S gives a new permutation S_2 that is sorted to I. Then normalize the inverse tree by applying S_2^{-1} to each permutation so that I is the first element, instead of S_2 and the trees can be compared. If there is a permutation x that exists in both trees, then there must be a path of length at most k from S to I, via x.

Let S be a permutation and k be a nonnegative integer. Let I be the identity permutation. Let T_1 and T_2 be bounded bounded-depth search trees, initially empty, where each node of T_1 will have a permutation and its distance from S and each node of T_2 will have a permutation and its distance from I.

```
**FPT-DOUBLE-ENDED-SBT**($S, k$)
$S_1 =$ **KERNELIZE(**$S$**)**
$S_2 = S_1^{-1} \cdot I$
**FPT-NAIVE-SBT(**$S_1, \lceil k/2 \rceil$**)**
**FPT-NAIVE-SBT(**$S_2, \lfloor k/2 \rfloor$**)**
normalize $T_2$ so that its permutations can be compared with $T_1$
$X =$ the intersection of permutations in $T_1$ and $T_2$
if ($X = \emptyset$)
    return NO
else
    return YES
```

This generates two depth-bounded search trees which are both of size $O((3k)^{3 \cdot k/2})$ which is $O((3k)^{1.5k})$. The kernelization and inverse problem creation take an insignificant amount of time compared to the tree sizes, but normalizing T_2 will take $O(|T_2|)$

time. The most time consuming part of this algorithm is generating the intersection X of permutations in T_1 and T_2, which can be done in $O((|T_1| + |T_2|) \cdot \log(|T_1| + |T_2|))$ by inserting each permutation into a sorted binary search tree. This is $O(\log((3k)^{1.5k}) \cdot (3k)^{1.5k})$ which is $O(k \log k \cdot (3k)^{1.5k})$. The memory required is $O((3k)^{1.5k})$.

How do the factorial algorithm **FPT-DOUBLE-ENDED-SBT** and the exponential algorithm **FPT-PERMUTATION-GRAPH-SBT** compare? It suffices to compare $n^{n/2}$ and $n!$. $n! = n \cdot (n-1) \cdot (n-2) \ldots 2$ and can be rewritten as a product of $n/2$ terms if n is even: $(n \cdot ((n-1) \cdot 2) \cdot ((n-2) \cdot 3) \ldots ((n/2) \cdot ((n/2)+1)))$. Each term is larger than n, so clearly $n!$ is larger than $n^{n/2}$ as $n^{n/2}$ is the product of $n/2$ terms that are n. for even n. If n is odd, then the final term of the rewritten $n!$ is $\lfloor n/2 \rfloor$. However, since n is odd, $n^{n/2}$ is the product of $\lfloor n/2 \rfloor$ terms of n and one term of $\sqrt{n}$. Since $n/2$ is larger than $\sqrt{n}$ for $n > 3$, $n!$ is larger than $n^{n/2}$.

Note that each term needs at most to be multiplied by 2 to be larger than n. Then instead of using the $\lfloor n/2 \rfloor!$ multiple, we only need $2^{\lfloor n/2 \rfloor}$. Then $n^{n/2}$ and $n!$ differ by at least

$$\frac{(\lfloor n/2 \rfloor!)}{2^{\lfloor n/2 \rfloor}}$$

.

Therefore $O(k \log k \cdot (3k)^{1.5k}$ increases slower than $O(k^3 \cdot (3k)!)$. Thus a double-ended search causes the running time of **FPT-DOUBLE-ENDED-SBT** to grow slower than that of **FPT-PERMUTATION-GRAPH-SBT** as n increases.

That being said, both algorithms require a large amount of time and memory. Structural considerations can potentially decrease the number of permutations generated in the depth-bounded search tree, and could potentially improve **FPT-DOUBLE-ENDED-SBT**.

Chapter 4

Conclusions

We have provided a simple kernelization of at most $3k$ elements for Sorting by Transpositions, by treating adjacent elements of a permutation which are also adjacent in the identity permutation as a single element. This concept was used with the intermediate results of a transposition sequence to show that there is always an optimal transposition sequence that splits no pair of elements more than once. It remains to be seen if the second result can lead to an improved algorithm for the problem.

The concept of inversions does not seem to give much insight into the problem of Sorting by Transpositions. In particular, some permutations are only optimally sorted by inverting elements i and $i+1$. Future research should be done that examines such permutations to determine their structure and rarity. Perhaps they are easy to solve and can be worked around in an algorithm based on not creating new inversions.

Adding and removing elements from a permutation can potentially be useful for an inductive proof of properties of Sorting by Transpositions. However, our algorithm based on adding elements to generate all optimal transposition sequences can not avoid looking at nonoptimal transposition sequences as well. Future work should examine this further to see if nonoptimal transposition sequences can be pruned without risk of losing at least one optimal transposition sequence.

We provided two fixed-parameter algorithms for Sorting by Transpositions. Both rely on the kernelization provided. The first algorithm has a running time of $O(k^3 \cdot (3k)!)$, and examines every permutation of the correct size. The second algorithm has a running time of $O(k \log k \cdot (3k)^{1.5k})$, by using a double-ended search, which we showed is faster than the first algorithm. Both algorithms have memory requirements similar to their running times.

We conjecture that 2-transpositions are optimal, ie. every 2-transposition is the start of some optimal transposition sequence. This may be useful practically, even if it does not lead to better worst-case bounds. We have so far been unable to prove or disprove this. Approximation algorithms often use a stronger version of this, choosing any available 2-moves.

We had hoped to find a faster FPT algorithm than those discussed here, or possibly an NP-hardness proof. We examined several structual properties of the problem which could lead to improvements. We have an efficient linear-size problem kernel; however, we have so far found no better depth bounded search tree method than an exhaustive double-ended search. Future research should focus on improving this.

Bibliography

[1] Bafna, V., and Pevzner, P. Genome rearrangements and sorting by reversals. *SIAM Journal of Computing 25* (1996), 272–289.

[2] Bafna, V., and Pevzner, P. Sorting by transpositions. *SIAM Journal on Discrete Mathematics 11*, 2 (1998), 224–240.

[3] Berman, P., Hannenhalli, S., and Karpinski, M. 1.375-approximation algorithm for sorting by reversals. *Electronic Colloquium on Computational Complexity (ECCC) 8*, 47 (2001).

[4] Caprara, A. Sorting by reversals is difficult. In *RECOMB '97: Proceedings of the first annual international conference on Computational molecular biology* (New York, NY, USA, 1997), ACM, pp. 75–83.

[5] Downey, R., and Fellows, M. Fixed-parameter tractability and completeness I: Basic results. *SIAM Journal of Computing 24* (1995), 873–921.

[6] Elias, I., and Hartman, T. A 1.375-approximation algorithm for sorting by transpositions. *IEEE/ACM Transactions on Computational Biology and Bioinformatics 3*, 4 (2006), 369–379.

[7] Hannenhalli, S., and Pevzner, P. Transforming cabbage into turnip: Polynomial algorithm for sorting signed permutations by reversals. *Journal of the ACM 46*, 1 (1999), 1–27.

[8] Hartman, T. A simpler 1.5-approximation algorithm for sorting by transpositions. *Combinatorial Pattern Matching (CPM 03) 2676* (2003), 156–169.

[9] Niedermeier, R. *Invitation to Fixed-Parameter Algorithms.* Oxford University Press, 2006.

[10] Palmer, J., and Herbon, L. Plant mitochondrial DNA evolves rapidly in structure, but slowly in sequence. *Journal of Molecular Evolution 27* (1988), 87–97.

[11] Sankoff, D., Leduc, G., Antoine, N., Paquin, B., Lang, B. F., and Cedergren, R. J. Gene order comparisons for phylogenetic inference: Evolution of the mitochondrial genome. In *Proceedings of the National Academy of Sciences* (1992), vol. 89, pp. 6575–6579.