UNIVERSITY
of
GLASGOW

Department of
Computing Science

Genome Rearrangement Problems

David Alan Christie

Submitted for the degree of Doctor of Philosophy
to
The University of Glasgow,
August, 1998.

Abstract

Various global rearrangements of permutations, such as reversals and transpositions, have recently become of interest because of their applications in computational molecular biology. A *reversal* is an operation that reverses the order of a substring of a permutation. A *transposition* is an operation that swaps two adjacent substrings of a permutation. The problem of determining the smallest number of reversals required to transform a given permutation into the identity permutation is called *sorting by reversals*. Similar problems can be defined for transpositions and other global rearrangements.

Related to sorting by reversals is the problem of establishing the reversal diameter. The *reversal diameter* of S_n (the symmetric group on n elements) is the maximum number of reversals required to sort a permutation of length n. Of course, diameter problems can be posed for other global rearrangements.

These various problems are of interest because the permutations can be used to represent sequences of genes in chromosomes, and the global rearrangements then represent evolutionary events. As a result, we call these problems *genome rearrangement problems*.

Genome rearrangement problems seem to be unlike previously studied algorithmic problems on sequences, so new methods have had to be developed to deal with them. These methods predominantly employ graphs to model permutation structure. However, even using these methods, often a genome rearrangement problem has no obvious polynomial-time algorithm, and in some cases can be shown to be NP-hard. For example, the problem of sorting by reversals is NP-hard, whereas the computational complexity of sorting by transpositions is open. For problems like these, it is natural to seek polynomial-time approximation algorithms that achieve an approximation guarantee.

In this thesis, we study several genome rearrangement problems as interesting and challenging algorithmic problems in their own right, including some problems for which the global rearrangement has no immediate biological equivalent. For example, we define a *block-interchange* to be a rearrangement that swaps any two substrings of the permutation. We examine, in particular, how the graph theoretic models relate to the genome rearrangement problems that we study.

The major new results contained in this thesis are as follows:

- We present a 3/2-approximation algorithm for sorting by reversals. This is the best known approximation algorithm for the problem, and improves upon the 7/4 approximation bound of the previous best algorithm.

- We give a polynomial-time algorithm for a significant special case of sorting by reversals, thereby disproving a conjecture of Kececioglu and Sankoff, who had suggested that this special case was likely to be NP-hard.
- We analyse the structure of the so-called *cycle graph* of a permutation in the context of sorting by transpositions, and thereby gain a deeper insight into this problem. Among the consequences are: a tighter lower bound for the problem, a simpler 3/2-approximation algorithm than had previously been described, and algorithms that, in empirical tests, almost always find the exact transposition distance of random permutations.
- We introduce a natural generalisation of sorting by transpositions called sorting by block-interchanges, and present a polynomial-time algorithm for this problem.
- We initiate the study of analogous problems on strings over a fixed length alphabet. We establish upper and lower bounds and diameter results for the problems over a binary alphabet. We also prove that the problems analogous to sorting by reversals and sorting by block-interchanges are NP-hard.

The thesis has the following structure:

In Chapter 1 we introduce several genome rearrangement problems, review the existing literature on the algorithmic aspects of these problems, and summarise the fundamental results and notations.

We study the problem of sorting by reversals in Chapter 2. At the start of this chapter, we introduce a new graph, called the reversal graph of a permutation, that turns out to be useful for finding sequences of reversals that sort the permutation. We then use these graphs to obtain the 3/2-approximation algorithm, and the polynomial-time algorithm for the special case.

Chapter 3 contains a number of contributions that help to improve our understanding of the algorithmic issues involved in the problem of sorting by transpositions. We note that an optimal sequence of transpositions never has to break apart elements i and $i+1$ once they are together in a permutation. Then we analyse the structure of cycles in the cycle graph leading to a better understanding of how these cycles affect the general problem. This leads to a simpler 3/2-approximation algorithm for the problem. We also obtain an improved lower bound for sorting by transpositions and improve the lower bound for transposition diameter. This latter result is obtained by describing an optimal sequence of transpositions that sorts the reverse permutation.

In Chapter 3 we also describe various approximation algorithms and an exact branch and bound algorithm for sorting by transpositions. In empirical tests, described in this chapter, we observe that our approximation algorithms require much fewer than 3/2 times the optimal number of transpositions to sort a random permutation (even though we do not prove any approximation bound for the approximation algorithms we test). We also observe that the exact algorithm could almost always find an optimal sequence

of transpositions for random permutations. We note that for most permutations the optimal length of sequence of transpositions is equal to or very close to the lower bound.

Chapter 4 is devoted to the problem of sorting by block-interchanges. We describe a sequence of block-interchanges that sorts a given permutation. We then show that properties of the cycle graph, in the context of this problem, mean that the sequence we describe is optimal. We also establish the block-interchange diameter of S_n.

In Chapter 5 we study problems on strings, over a fixed length alphabet, that are analogous to sorting by reversals, sorting by prefix-reversals (reversals that act on prefixes), sorting by transpositions and sorting by block-interchanges. We adapt the concept of the *breakpoint*, from problems on permutations, for use with these problems. Then, for all of the problems, we obtain upper and lower bounds for the distance between two binary strings, and we establish a diameter result on binary strings. We also show that the problems analogous to sorting by reversals and sorting by block-interchanges are NP-hard, even when the alphabet is binary.

Contents

Abstract ii

Contents v

List of Figures viii

List of Tables x

Preface xi

1 Introduction and Background 1
- 1.1 Introduction . . . 1
- 1.2 Sorting by reversals . . . 3
- 1.3 Sorting signed permutations by reversals . . . 6
- 1.4 Sorting by prefix-reversals . . . 7
- 1.5 Sorting by restricted reversals . . . 9
- 1.6 Sorting by transpositions . . . 9
- 1.7 Sorting by restricted transpositions . . . 11
- 1.8 Sorting by block-interchanges . . . 12
- 1.9 Sorting by reversals and transpositions . . . 12
- 1.10 Generating the symmetric group . . . 13
- 1.11 Sorting by translocations . . . 13
- 1.12 Syntenic edit distance . . . 14
- 1.13 Other related problems . . . 15

2 Sorting by Reversals 16
- 2.1 Introduction . . . 16
- 2.2 Definitions . . . 16
- 2.3 Reversal graphs simplify sorting by reversals . . . 17
- 2.4 A 3/2-approximation algorithm for sorting by reversals . . . 28
  - 2.4.1 The 3/2-bound . . . 28
  - 2.4.2 The cycle decomposition . . . 29
  - 2.4.3 The sequence of reversals . . . 32

2.4.4 The algorithm . . . 34
2.4.5 Conclusion . . . 34
2.5 An easy case of sorting by reversals . . . 36
2.5.1 2-reversals, exposed and hidden . . . 37
2.5.2 Kernels . . . 38
2.6 Conclusion and open problems . . . 46

3 Sorting by Transpositions **47**
3.1 Introduction . . . 47
3.2 Fundamental definitions and results . . . 48
3.2.1 Definitions . . . 48
3.2.2 Transposition equivalent permutations . . . 48
3.2.3 The transposition cycle graph . . . 49
3.2.4 Even length cycles . . . 52
3.3 Fully oriented cycles . . . 52
3.3.1 Fully oriented triples . . . 53
3.3.2 Canonical labellings . . . 53
3.3.3 Fully oriented cycles . . . 55
3.3.4 Knots . . . 58
3.3.5 Strongly oriented cycles . . . 62
3.3.6 Super oriented cycles . . . 62
3.4 A new 3/2-approximation algorithm . . . 63
3.5 A tighter lower bound . . . 77
3.5.1 Hurdles . . . 77
3.5.2 How tight are the lower bounds? . . . 79
3.5.3 The transposition diameter . . . 81
3.6 Solving instances of sorting by transpositions in practice . . . 83
3.6.1 The algorithms . . . 84
3.6.2 Permutations . . . 89
3.6.3 How the algorithms performed . . . 92
3.6.4 Analysis of the performance of the algorithms . . . 100
3.7 Conclusion and open problems . . . 105

4 Sorting by Block-Interchanges **107**
4.1 Introduction . . . 107
4.2 Minimal block-interchanges . . . 108
4.3 The graph model . . . 109
4.4 Block-interchange distance . . . 110
4.5 Block-interchange diameter . . . 110
4.6 Conclusion . . . 113

5 Sorting Strings by Global Transformations 114
5.1 Introduction . . . 114
5.2 Definitions . . . 115
5.3 Reversal distance between binary strings . . . 116
5.3.1 A lower bound . . . 116
5.3.2 An upper bound . . . 118
5.3.3 Reversal diameter of $\mathcal{B}_n$ . . . 119
5.3.4 Sorting by reversals . . . 121
5.4 Prefix-reversal distance between binary strings . . . 122
5.4.1 A lower bound . . . 122
5.4.2 An upper bound . . . 122
5.4.3 Prefix-reversal diameter of $\mathcal{B}_n$ . . . 123
5.4.4 Sorting by prefix-reversals . . . 126
5.5 Transposition distance between binary strings . . . 127
5.5.1 A lower bound . . . 127
5.5.2 An upper bound . . . 128
5.5.3 Transposition diameter of $\mathcal{B}_n$ . . . 129
5.5.4 Sorting by transpositions . . . 130
5.6 Block-Interchange distance between binary strings . . . 131
5.6.1 A lower bound . . . 131
5.6.2 An upper bound . . . 132
5.6.3 Block-interchange diameter of $\mathcal{B}_n$ . . . 135
5.6.4 Sorting by block-interchanges . . . 136
5.7 NP-completeness of reversal distance and block-interchange distance . . . 137
5.8 Conclusion . . . 142

Glossary 144

Bibliography 147

List of Figures

1.1 Some example reversals. 1
1.2 Some example transpositions. 2
1.3 Some examples of reversals on signed permutations. 6
1.4 Some example prefix-reversals. 8
1.5 Some example block-interchanges. 12
1.6 Some example translocations. 14
1.7 An example syntenic edit distance problem. 15

2.1 An example cycle decomposition of $rG'(\pi)$. 19
2.2 An example reversal graph 19
2.3 The effect of $\rho(u)$ when u is red. 20
2.4 The effect of $\rho(u)$ when u is blue. 21
2.5 The resulting cycle decomposition. 22
2.6 The resulting reversal graph. 22
2.7 A type 1 vertex. 24
2.8 A type 2 vertex. 25
2.9 Some short ladders. 30
2.10 $rG(\pi)$ for $\pi = [0\ 4\ 2\ 6\ 8\ 7\ 3\ 5\ 1\ 9\ 14\ 15\ 12\ 13\ 10\ 11\ 16]$. 31
2.11 $rF(\pi)$ for $\pi = [0\ 4\ 2\ 6\ 8\ 7\ 3\ 5\ 1\ 9\ 14\ 15\ 12\ 13\ 10\ 11\ 16]$. 31
2.12 $rL(M)$ where $M = \{a, b, c, d, e\}$ in Figure 2.11. 31
2.13 The cycle decomposition $\mathcal{C}$. 31
2.14 Part of a long ladder. 33
2.15 A short ladder. 33
2.16 The 3/2-approximation algorithm 34
2.17 Finding an elimination sequence for $rR(\mathcal{C})$. 35
2.18 $rF^*(\pi_3)$ 43
2.19 $rF^*(\pi_4)$ 44
2.20 $rF^*(\pi_5)$ 44

3.1 $tG(\pi)$, for $\pi = [3\ 4\ 1\ 2]$. 49
3.2 $tG(\pi)$, for $\pi = [6\ 1\ 5\ 7\ 12\ 4\ 11\ 2\ 9\ 3\ 8\ 10]$. 51
3.3 How a transposition changes the cycle graph. 54
3.4 $tG(\pi)$, for $\pi = [4\ 3\ 2\ 1]$. 54

3.5 How to sort a knot of length 5. 59
3.6 A sequence of transpositions that sorts ω_5. 60
3.7 The oriented cycle obtained by an applying an transposition that interleaves with an unoriented cycle. 65
3.8 How C and D are interleaved initially. 66
3.9 How the cycles are interleaved after the 0-transposition. 67
3.10 How the cycle graph changes. 69
3.11 How $tG(\pi)$ changes as the result of the transposition. 70
3.12 The first transposition of a $(0, 2, 2)$-sequence. 72
3.13 Special case of (d). 73
3.14 Special case of (e). 74
3.15 The new 3/2-approximation algorithm 75
3.16 Bafna and Pevzner's 3/2-approximation algorithm 76
3.17 A cycle graph with three components. 78
3.18 An optimal length sorting of κ_2. 80
3.19 A sequence of transpositions that sorts R_{11}. 83
3.20 The algorithm: `approx` 85
3.21 The algorithm: `tlis` 87
3.22 The algorithm: `random` 88
3.23 The algorithm: `branch` 90
3.24 Performance of `best`. 103
3.25 A sequence of 13 transpositions that sorts π. 104

4.1 An example of sorting by block-interchanges 108
4.2 The black edges that change when a block-interchange is applied: (a) when the block-interchange is not a transposition, and (b) when the block-interchange is a transposition. 109
4.3 How a minimal block-interchange changes $tG(\pi)$. 111
4.4 An algorithm to calculate $bd(\pi)$. 112
4.5 An algorithm to perform an optimal sequence of block-interchanges. . . 112

List of Tables

3.1 random permutations, `lower_bound` . . . 94
3.2 random permutations, `approx` . . . 94
3.3 random permutations, `alt_approx` . . . 94
3.4 random permutations, `tlis` . . . 94
3.5 random permutations, `random` . . . 95
3.6 random permutations, `rpt_random` . . . 95
3.7 random permutations, `branch` . . . 95
3.8 random permutations, `best` . . . 95
3.9 3-cycle permutations, `lower_bound` . . . 96
3.10 3-cycle permutations, `approx` . . . 96
3.11 3-cycle permutations, `alt_approx` . . . 96
3.12 3-cycle permutations, `tlis` . . . 96
3.13 3-cycle permutations, `random` . . . 97
3.14 3-cycle permutations, `rpt_random` . . . 97
3.15 3-cycle permutations, `branch` . . . 97
3.16 3-cycle permutations, `best` . . . 97
3.17 transposition tight permutations, `lower_bound` . . . 98
3.18 transposition tight permutations, `approx` . . . 98
3.19 transposition tight permutations, `alt_approx` . . . 98
3.20 transposition tight permutations, `tlis` . . . 98
3.21 transposition tight permutations, `random` . . . 99
3.22 transposition tight permutations, `rpt_random` . . . 99
3.23 transposition tight permutations, `branch` . . . 99
3.24 transposition tight permutations, `best` . . . 99

4.1 The number of permutations of size n that achieve $bD(n)$. . . . 113

5.1 The 18 possible combinations of S and T. . . . 133
5.2 The complexity of the different problems, and the diameter of $\mathcal{B}_n$. . . . 143

Preface

Acknowledgements

First of all I would like to thank Rob Irving for his supervision. Rob's guidance has been crucial in getting this thesis into its current state. In particular, I am grateful for the many helpful comments he has given that have helped to improve the quality of my writing. Thanks too to everyone else who has been part of my supervisory team.

I would like to thank my family for all their support, especially in this final year.

I would like to thank EPSRC (the Engineering and Physical Sciences Research Council) for their financial support in the form of a Research Studentship from October 1994 to September 1997.

Lastly, I would like to un-thank Partick Thistle for being relegated twice during the course of my postgraduate studies.

Publications and papers

- R. W. Irving and D. A. Christie. Sorting by Reversals: on a conjecture of Kececioglu and Sankoff. Technical Report TR-95-12, Department of Computing Science of The University of Glasgow, May 1995.

  The contents of this paper are presented in Section 2.5. However, the presentation of this work has been changed, so as to make it more consistent with the rest of Chapter 2.

  Tran [Tra97] subsequently and independently obtained an identical result to that presented in this paper.

- D. A. Christie. Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165–169, November 1996.

  Chapter 4 is based on this paper.

- D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, San Francisco CA., pages 244–252, January 1998.

The contents of this paper are presented in Sections 2.3 and 2.4, with corrections to a few minor errors in the paper.

- D. A. Christie and R. W. Irving. Sorting Strings by Global Transformations. Submitted for publication.

  This paper is based on parts of Chapter 5.

Statement of collaboration

The following summarises the role of my supervisor, Rob Irving, in work undertaken jointly with him.

Chapter 2

The reversal graphs of Section 2.3 were developed from graphs in [IC95]. Rob is due most credit for the graphs in [IC95], and suggested the method used to develop those graphs into the form shown in Section 2.3. Rob proposed using the matching and ladder graphs to generate the cycle decomposition used in Section 2.4. Section 2.5 was originally ([IC95]) written jointly with Rob.

Chapter 3

The canonical labellings of cycles (Section 3.3.2) were suggested by Rob. He also proposed Theorem 3.3.3. An initial proof of Theorem 3.5.5 was found by Rob, but the proof presented in this thesis is quite different.

Chapter 5

The NP-completeness proofs in this chapter were developed with Rob.

Overlaps with work by other authors

Theorem 3.5.6 was proved independently by Meidanis, Walter and Dias, and this theorem appears as the main result in [MWD97b].

Declaration

This dissertation is submitted in accordance with the regulations for the degree of Doctor of Philosophy in the University of Glasgow. No part of it has been previously submitted by the author for a degree at any university and all results contained within are claimed as original, except where indicated above.

Chapter 1

Introduction and Background

1.1 Introduction

It is common to compare strings by calculating the minimum number of operations required to transform one into the other, where the set of allowable operations varies according to the context. When the set of allowable operations consists of insertions, deletions and substitutions, efficient algorithms to calculate such 'edit distances' are well known , e.g., [WF74], and are described in most algorithms textbooks.

We say that insertions, deletions and substitutions are *local* operations. However, it is possible to envisage *global* operations that can change the entire string in one step. Global operations like this include the *reversal* and the *transposition*. A reversal inverts the order of a substring of any length, and a transposition swaps two adjacent substrings of any length. Examples of these global operations are shown in Figures 1.1 and 1.2. Note that in these figures, and in others too, the operations act on the underlined substrings.

In this thesis we consider several edit distance problems based on global operations. Motivation for studying these problems comes firstly from computational molecular biology. At the chromosome level, genetic sequences mutate much more commonly by global operations like reversals and transpositions, than by local operations. So evolutionary relationships between genomes can be inferred by calculating such global edit distances.

We call any problem to do with calculating edit distances based on global operations

7 6 3 1 5 8 2 4
7 6 3 1 2 8 5 4

7 6 3 1 5 8 2 4
1 3 6 7 5 8 2 4

Figure 1.1: Some example reversals.

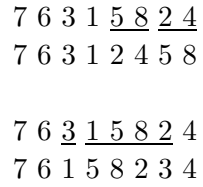

Figure 1.2: Some example transpositions.

a *genome rearrangement problem*. This definition is very broad, and includes problems that are not directly motivated by biological applications, that involve more than two strings, and that are based on sets instead of strings. Typically, however, we consider genome rearrangement problems that involve calculating the edit distance between two strings relative to one kind of global operation.

For each problem, we seek an efficient algorithm or, alternatively, a proof that the problem is NP-hard. If we cannot obtain an efficient algorithm then we seek good heuristics for solving instances of the problem, and efficient algorithms that find provably good solutions. For minimisation problems, like all the problems we consider, an algorithm is called an *r-approximation algorithm* if the solution produced by the algorithm is guaranteed to be no greater than r times the optimal solution.

Genome rearrangement problems have been studied for only a relatively short while. New methods have had to be developed for these problems, because the dynamic programming solutions typical of local edit distance algorithms are not obviously applicable in the new setting.

One step that is almost always taken to simplify the problems is to assume that the sequences being compared contain no repeated characters. The resulting simplified problem is still biologically valid since a genome is unlikely to contain duplicate genes. If there are no repeated characters then the strings are actually permutations, since the global operations we consider cannot insert or delete any characters.

In this thesis we use π and ϕ to denote permutations. The length of a permutation is denoted by n. (We also occasionally use $|\pi|$ to denote the length of π.) We denote the ith element of π by $\pi(i)$. So, for example, if $\pi = [4\ 1\ 3\ 2]$, then $\pi(1) = 4$, $\pi(2) = 1$, and so on. For reasons that will become apparent later, we always assume that π is extended so that $\pi(0) = 0$ and $\pi(n+1) = n+1$. However, usually when we write π, as above, we do not include these extra elements. The identity permutation $[1\ 2\ 3\ \ldots\ n]$ is denoted by $\imath_n$, or $\imath$ when the length of the permutation is obvious. The set (actually the symmetric group) containing all permutations of length n is denoted by S_n.

Global edit distance problems on permutations can be viewed as sorting problems. For example, if $\pi = [4\ 1\ 3\ 2]$, and $\phi = [2\ 4\ 1\ 3]$, we can relabel the elements of the permutations so that $\phi = [\boxed{1}\ \boxed{2}\ \boxed{3}\ \boxed{4}]$. That is, we have relabelled 2 as $\boxed{1}$, 4 as $\boxed{2}$, 1 as $\boxed{3}$, and 3 as $\boxed{4}$. With this relabelling $\pi = [\boxed{2}\ \boxed{3}\ \boxed{4}\ \boxed{1}]$, and finding the distance between π and ϕ is equivalent to finding the shortest sequence of operations that sorts π. Note that $\phi^{-1} \cdot \pi = [2\ 3\ 4\ 1]$, and, in general, the distance between π and ϕ is equal

to the distance between $\phi^{-1} \cdot \pi$ and the identity permutation.

For a particular global operation X, *sorting by Xs* (e.g., sorting by reversals) is the problem of finding a shortest sequence of Xs that sorts a given permutation. For such a problem the *distance of* π, $d(\pi)$, is the minimum number of operations required to sort π. Note that by relabelling elements the distance between $\imath$ and π is equal to the distance between π^{-1} and $\imath$, so $d(\pi) = d(\pi^{-1})$. The *diameter of the symmetric group*, $D(n)$, is the maximum value of $d(\pi)$ taken over all n element permutations.

In this thesis we prefix $d(\pi)$ with letters representing the global operation(s) being considered. Similarly, in general, we prefix all functions by letters representing the operation(s) being considered.

A simple concept that is often useful when studying genome rearrangement problems is that of the *breakpoint*. It is a position in the permutation that needs to be changed by an operation, at some point in the transformation to the identity permutation. The exact definition of a breakpoint differs from problem to problem. A lower bound for $d(\pi)$ can be calculated from the number of breakpoints in π. Often this lower bound can be improved by counting cycles in a graph, called the *cycle graph*, that is particularly appropriate for capturing the 'hidden' difficulties that make genome rearrangement problems non-trivial.

There are many different variations of genome rearrangement problems. One variation is derived from the fact that genes have an orientation in chromosomes. To model this fact, elements of the permutations can be given signs, '+' or '−', that can change when the elements take part in global operations. Other variations involve restricting the size or positions of the rearrangements, e.g., to only allow reversals of length two. Yet other variations involve different global operations to be described in detail later.

The genome rearrangement problems are interesting in an algorithmic sense, because some of the problems have been shown to be NP-hard, some are polynomial-time solvable, and some have unresolved computational complexity. Sometimes changing a problem's definition just a little changes the problem from an NP-hard problem to a problem in P, or vice-versa.

We can also view genome rearrangement problems as combinatorial puzzles like, for example, the Rubik's cube. In this way solving them is viewed as something of an intellectual challenge, and motivation to solve the problems becomes intrinsic.

Recent textbooks by Setubal and Meidanis [SM97] and Gusfield [Gus97] both contain a chapter on genome rearrangement problems and, in particular, the problem of sorting by reversals. In the sections below we describe specific genome rearrangement problems and review the literature on these and related problems.

1.2 Sorting by reversals

A *reversal* $\rho = \rho(i, j)$ (where $1 \leq i < j \leq n+1$) transforms π into $\pi \cdot \rho = [\pi(0) \ \ldots \ \pi(i-1) \ \pi(j-1) \ \ldots \ \pi(i) \ \pi(j) \ \ldots \ \pi(n+1)]$. The *reversal distance*, $rd(\pi)$, between π and

$\imath$ is the length of a shortest sequence of reversals that transforms π into $\imath$. *Sorting by reversals* is the problem of finding a sequence of reversals of length $rd(\pi)$ that sorts π.

When dealing with reversals, a *(reversal) breakpoint* is defined as a position i ($1 \leq i \leq n+1$) such that $|\pi(i) - \pi(i-1)| \neq 1$. (Remember that π is extended so that $\pi(0) = 0$ and $\pi(n+1) = n+1$.) The number of reversal breakpoints in π is denoted by $rb(\pi)$. The identity permutation is the only permutation with no reversal breakpoints, and a permutation can have no more than $n+1$ reversal breakpoints. A *strip* is a substring $\pi[i..j]$ ($i \leq j$) of π such that i and $j+1$ are breakpoints, but no breakpoints lie between these positions. A *singleton* is a strip of length one. A strip is *increasing* if $\pi(j) > \pi(i)$, otherwise it is *decreasing*.

A reversal can reduce the number of breakpoints in a permutation by at most two. Therefore, a simple lower bound for reversal distance is

$$rd(\pi) \geq \frac{rb(\pi)}{2}.$$

Watterson, Ewens, Hall and Morgan [WEHM82] describe a simple approximation algorithm for sorting by reversals[1] that moves 1 to the front of the permutation with the first reversal, then moves 2 to the correct place with the next reversal, and so on until the permutation is sorted. This algorithm requires at most $n-1$ reversals to sort any given permutation. However, this algorithm does not achieve any constant approximation bound.

Kececioglu and Sankoff [KS95][2] present a greedy approximation algorithm that repeatedly applies a reversal that removes the most breakpoints from the permutation, resolving ties among reversals that remove one breakpoint in favour a reversal that leaves a decreasing strip. They are able to show that, on average, their algorithm removes at least one breakpoint with every reversal, and is therefore a 2-approximation algorithm. The worst-case time complexity of this algorithm is $O(n^2)$.

Bafna and Pevzner [BP96][3] introduce the important concept of the *(reversal) cycle graph*[4] $rG(\pi)$. It is an edge coloured undirected graph derived from π that contains a vertex for each element in the permutation (including 0 and $n+1$). So $rG(\pi)$ has vertex set $\{0, \ldots, n+1\}$. Vertices are joined by black edges if the elements they represent form a breakpoint in π, i.e., $rG(\pi)$ has black edge set $\{\{\pi(i-1), \pi(i)\} : 1 \leq i \leq n+1,\ i \text{ is a breakpoint}\}$. Vertices i and $i+1$ are joined by a grey edge if these elements are not consecutive in π, i.e., $rG(\pi)$ has grey edge set $\{\{i, i+1\} : 0 \leq i \leq n,\ i \text{ and } i+1 \text{ are not consecutive in } \pi\}$. Some example cycle graphs are shown in Chapter 2.

The cycle graph is important because Bafna and Pevzner obtain an improved lower bound for reversal distance by considering cycles in this graph. An *alternating cycle* is

[1] In fact, they actually consider a similar problem on circular permutations in which mirror image permutations are considered to be identical.

[2] an earlier version of this paper appeared as [KS93]

[3] an earlier version of this paper appeared as [BP93]

[4] In fact, Bafna and Pevzner call these graphs breakpoint graphs.

a cycle that has edges of alternating colours. The cycle graph can be completely decomposed into edge-disjoint alternating cycles, because each vertex has an equal number of incident grey and black edges. The number of cycles in a *maximum alternating-cycle decomposition* of $rG(\pi)$ is denoted by $rc(\pi)$. Bafna and Pevzner show that

$$rd(\pi) \geq rb(\pi) - rc(\pi).$$

This is a better bound than the bound described earlier because each cycle accounts for at least two breakpoints.

Bafna and Pevzner obtain a 7/4-approximation algorithm for sorting by reversals, by considering signed permutations (Section 1.3) and properties of the cycle graph. The worst-case time complexity of their algorithm is $O(n^2)$.

Christie [Chr98] presents a 3/2-approximation algorithm for sorting by reversals. We describe this algorithm in Chapter 2. Frings [Fri98] describes experiments comparing this 3/2-approximation algorithm with Kececioglu and Sankoff's 2-approximation algorithm. He finds that, on average, the 3/2-approximation algorithm finds 'significantly' shorter sequences than the 2-approximation algorithm.

Hannenhalli and Pevzner [HP96] use their algorithm for sorting signed permutations by reversals (Section 1.3) to prove a conjecture of Kececioglu and Sankoff [KS95] that an optimal length sequence of reversals exists that sorts a permutation and does not break apart strips of length three or more. They also explain precisely when strips of length two must be split apart. In fact, for permutations that have fewer than $\log n$ singletons they obtain a polynomial-time algorithm for sorting by reversals.

Kececioglu and Sankoff [KS95] describe some novel graphs that give improved lower bounds for reversal distance that they use in an exact algorithm that can determine the reversal distance of permutations of about 30 elements.

Caprara, Lancia and Ng [CLN95] describe an exact branch and bound algorithm that sorts random permutations containing up to 100 elements in a few minutes.

Kececioglu and Sankoff [KS95] conjecture that deciding if a permutation can be sorted with $rb(\pi)/2$ reversals is NP-complete and hence that sorting by reversals is NP-hard. However, Irving and Christie [IC95], and Tran [Tra97] disprove this conjecture. We explain this further in Section 2.5.

In fact, the general problem of sorting by reversals is NP-hard as proved by Caprara [Cap97b] (see also [Cap97c]). Essentially, he obtains this result by proving that determining the value of $rc(\pi)$ is NP-hard.

Caprara [Cap98] shows that the probability that the lower bound for $rd(\pi)$ based on $rG(\pi)$ is not exact is low (asymptotically $O(1/n^5)$) for a random permutation π. He uses this result to explain the good experimental performance of algorithms like that of Caprara, Lancia and Ng.

The *reversal diameter*, $rD(n)$, of the symmetric group S_n is the maximum value of $rd(\pi)$ taken over all permutations of length n. Bafna and Pevzner [BP96] prove that $rD(n) = n - 1$ and that the only permutations needing this many reversals are the

$$-7\ -6\ +3\ +1\ \underline{+5\ +8\ -9\ -2}\ -4$$
$$-7\ -6\ +3\ +1\ +2\ +9\ -8\ -5\ -4$$

$$\underline{-7\ +6\ +3\ -1}\ +5\ +8\ -9\ -2\ -4$$
$$+1\ -3\ -6\ +7\ +5\ +8\ -9\ -2\ -4$$

Figure 1.3: Some examples of reversals on signed permutations.

Gollan permutation, γ_n, and its inverse, where the Gollan permutation is defined as

$$\gamma_{n+1} = \begin{cases} [1], & \text{if } n = 0, \\ [\gamma_n(1)\ \gamma_n(2)\ \ldots\ \gamma_n(n-1)\ n+1\ \gamma_n(n)], & \text{if } n \text{ is odd}, \\ [\gamma_n(1)\ \gamma_n(2)\ \ldots\ \gamma_n(n-2)\ n+1\ \gamma_n(n-1)\ \gamma_n(n)], & \text{otherwise.} \end{cases}$$

1.3 Sorting signed permutations by reversals

In fact, there is a more biologically relevant version of sorting by reversals in which every element of the permutation is given a sign, '+' or '−', that indicates the orientation of the gene in the chromosome. Such permutations are called *signed* permutations. (Permutations with no signs are called *unsigned* permutations.) In this version of the problem, a reversal flips the sign of all the elements it acts upon. Some examples of reversals on signed permutations are shown in Figure 1.3.

The identity signed permutation is the permutation $+\imath = [+1\ +2\ \ldots\ +n]$. The *signed reversal distance* $srd(\pi)$ between π and $+\imath$ is the length of a shortest sequence of reversals that transforms π into $+\imath$. *Sorting signed permutations by reversals* is the problem of determining a sequence of reversals of length $srd(\pi)$ that sorts π.

For signed permutations, a *signed reversal breakpoint* is a position i $(1 \leq i \leq n+1)$ such that $\pi(i) - \pi(i-1) \neq 1$.

It would appear that sorting signed permutations by reversals is more complicated than sorting by reversals, since any algorithm to solve the signed problem has to keep track of the orientation of the elements as well as sort the permutation. However, despite the apparent extra complication, Hannenhalli and Pevzner [HP95b] show that the problem of sorting signed permutations by reversals is solvable in polynomial-time. They achieve this result by first of all transforming the signed permutation π into an unsigned permutation π' of length $2n$ by replacing $+i$ by $[2i-1\ 2i]$ and $-i$ by $[2i\ 2i-1]$. Significantly, $rG(\pi')$ has a unique cycle decomposition.

By considering how reversals can affect this unique cycle decomposition Hannenhalli and Pevzner are able to show that

$$srd(\pi) = rb(\pi') - rc(\pi') + rh(\pi') + rf(\pi),$$

where $rh(\pi')$ is the number of so-called *hurdles* in the cycle graph, and $rf(\pi)$ is 1 if π is a so-called *fortress* and 0 otherwise. An excellent explanation of this result is given by Setubal and Meidanis [SM97].

Berman and Hannenhalli [BH96] improve the $O(n^4)$ algorithm contained in [HP95b] by producing an $O(n^2\alpha(n))$ algorithm to solve the problem, where α is the inverse Ackerman function (a function that is unbounded but grows remarkably slowly). Kaplan, Shamir and Tarjan [KST97] present an $O(n^2)$ algorithm that is also simpler. If the value of $srd(\pi)$ is required only, without a sequence of reversals, then Berman and Hannenhalli's method can be used in a $O(n\alpha(n))$ algorithm.

Before Hannenhalli and Pevzner described their polynomial-time algorithm, Bafna and Pevzner [BP96] had described a 3/2-approximation algorithm for the problem, and Kececioglu and Sankoff [KS94] had observed that the lower bound based on the cycle graph of π' was very tight.[5]

We denote by Σ_n the set of n element signed permutations. The *signed reversal diameter* of Σ_n, $srD(n)$, is the the maximum value of $srd(\pi)$ taken over all π in Σ_n. For $n > 3$, $srD(n) = n + 1$. When n is even, the diameter is achieved by the permutation $+R_n = [+n\ +(n-1)\ \ldots\ +1]$. When n is odd and $n > 3$, the diameter is achieved by the permutation $[+2\ +1\ +3\ +n\ +(n-1)\ \ldots\ +4]$. These permutations are described by Knuth in [Knu98] (Exercise 5.1.4–43).

Caprara [Cap97a] considers a generalisation of sorting signed permutations by reversals called *multiple sorting by reversals*. In an instance of multiple sorting by reversals we are given q signed permutations, $\pi_1, \ldots, \pi_q$, and we must construct a signed permutation ϕ such that $\Sigma_{i=1}^{q} srd(\phi, \pi_i)$ is minimised, where $srd(\phi, \pi_i)$ is the signed reversal distance between ϕ and π_i. Caprara shows that multiple sorting by reversals is NP-hard, even when $q = 3$.

1.4 Sorting by prefix-reversals

A problem related to sorting by reversals is the problem of *sorting by prefix-reversals*, alternatively known as the *pancake problem* because it was posed [Dwe75] in the following way:

> The chef in our place is sloppy, and when he prepares a stack of pancakes they come out all different sizes. Therefore, when I deliver them to a customer, on the way to the table I rearrange them (so the smallest winds up on top, and so on, down to the largest on the bottom) by grabbing several from the top and flipping them over, repeating this (varying the number I flip) as many times as necessary. If there are n pancakes, what is the maximum number of flips (as a function of n) that I shall ever have to use to rearrange them.

[5] In fact, Kececioglu and Sankoff consider a version of the problem on circular permutations.

$$\underline{7\ 6\ 3\ 1\ 5\ 8\ 2}\ 4$$
$$2\ 8\ 5\ 1\ 3\ 6\ 7\ 4$$

$$\underline{7\ 6\ 3\ 1}\ 5\ 8\ 2\ 4$$
$$1\ 3\ 6\ 7\ 5\ 8\ 2\ 4$$

Figure 1.4: Some example prefix-reversals.

A prefix-reversal is a reversal of the form $\rho(1, j)$, for $2 \leq j \leq n+1$. Some example prefix-reversals are shown in Figure 1.4. The *prefix-reversal distance* $prd(\pi)$ of a permutation π is the minimum number of prefix-reversals necessary to sort the permutation. *Sorting by prefix-reversals* is the problem of finding a sequence of prefix-reversals of length $prd(\pi)$ that sorts π.

We define *prefix-reversal breakpoints* in the same way as we defined reversal breakpoints except that we ignore the first position in the permutation since this value must change with every prefix-reversal. So the total number of prefix-reversal breakpoints $prb(\pi)$ in π can be defined as

$$prb(\pi) = \begin{cases} rb(\pi), & \text{if } \pi(1) = 1, \\ rb(\pi) - 1, & \text{otherwise.} \end{cases}$$

A prefix-reversal can decrease the number of prefix-reversal breakpoints by at most one, so

$$prd(\pi) \geq prb(\pi).$$

A simple algorithm to sort a permutation is to bring n to the front of the permutation with the first reversal, then move it to the end with the next reversal. We can then use two reversals to move $n-1$ to the correct place, and so on until the permutation has been sorted. This process requires at most $2n-3$ reversals to sort π.

Heydari and Sudborough [HS] show that sorting by prefix-reversals is an NP-hard problem.

The prefix-reversal diameter $prD(n)$ of the symmetric group S_n is the maximum value of $prd(\pi)$ taken over all n-element permutations. Gates and Papadimitriou [GP79] show that

$$prD(n) \leq \frac{5n+5}{3}.$$

Heydari and Sudborough [HS97] improve a lower bound in [GP79] and show that

$$\frac{15n}{14} \leq prD(n).$$

A signed version of sorting by prefix-reversals, also called the burnt pancake problem, is introduced in [GP79]. The *signed prefix-reversal distance* of π, $sprd(\pi)$, is the minimum number of prefix-reversals that sort a signed permutation π. The *signed*

prefix-reversal diameter, $sprD(n)$, of Σ_n is the maximum value of $sprd(\pi)$ over all n element signed permutations. Cohen and Blum [CB95] tighten bounds in [GP79] and show that

$$\frac{3n}{2} \leq sprD(n) \leq 2n - 2.$$

Cohen and Blum also conjecture that the signed permutation prefix-reversal diameter is achieved by the permutation $-\iota = [-1\ -2\ \ldots\ -n]$. Heydari and Sudborough [HS97] show that $sprd(-\iota) \leq 3(n+1)/2$, and that if the conjecture is true then $prD(n) \leq sprD(n) \leq 3(n+1)/2$.

1.5 Sorting by restricted reversals

Chen and Skiena [CS96][6] investigate the problem of sorting permutations using reversals of a fixed-length only, i.e., using reversals of the form $\rho(i, i+k)$ for some fixed k. Note that sometimes it is impossible to sort a permutation using only reversals of a certain length. For example, an odd length reversal does not change the parity of the position of any element it acts on, so odd length reversals cannot sort any permutation of the form $\pi = [2\ 1\ \ldots\ ]$.

Chen and Skiena give a complete description for all n and k of how many permutations of length n can be sorted using only reversals of length k. They present upper and lower bounds for the reversal diameter of the group of permutations of length n that can be sorted by reversals of length k. They also study the problem on circular permutations.

A reversal of length two simply swaps adjacent elements of π. Bubble sort is an algorithm that sorts a permutation using the minimum number of reversals of length two. The number of swaps it uses is the so called *inversion number* of the permutation. Jerrum [Jer85] describes an algorithm that optimally sorts circular permutations using only reversals of length two.

A short reversal is a reversal of the form $\rho(i, i+2)$ or $\rho(i, i+3)$. Vergara [Ver97] studies the problem of sorting by short reversals. He presents a polynomial-time 2-approximation algorithm for calculating the short reversal distance of a given permutation. He also presents bounds for the short reversal diameter of S_n.

1.6 Sorting by transpositions

A *transposition* $\tau = \tau(i, j, k)$ (where $1 \leq i < j < k \leq n+1$) transforms π into $\pi \cdot \tau = [\pi(0)\ \ldots\ \pi(i-1)\ \pi(j)\ \ldots\ \pi(k-1)\ \pi(i)\ \ldots\ \pi(j-1)\ \pi(k)\ \ldots\ \pi(n+1)]$. We view a transposition as an operation that swaps two adjacent substrings in a permutation. (An alternative and equivalent view is to think of a transposition as an operation that removes the substring $\pi[i..j-1]$ from the permutation, before re-inserting this

[6] an earlier version of this paper appeared as [CS95]

substring in front of the element in position k.) Transpositions are also sometimes called *block-moves*. (Note that a cycle of length two in the disjoint cycle form of a permutation is sometimes called a transposition, but we are not concerned with these kinds of transpositions.)

The *transposition distance* $td(\pi)$ of a permutation π is the length of a shortest sequence of transpositions that transforms π into ι. *Sorting by transpositions* is the problem of finding a sequence of transpositions of length $td(\pi)$ that sorts π. The *transposition diameter*, $tD(n)$, is the maximum value of $td(\pi)$ taken over all n element permutations.

When dealing with transpositions a *(transposition) breakpoint* is defined as a position i $(1 \leq i \leq n+1)$ such that $\pi(i) - \pi(i-1) \neq 1$. (Remember, π is extended so that $\pi(0) = 0$ and $\pi(n+1) = n+1$.) A *strip* is a substring $\pi[i..j]$ of π $(i < j)$ such that i and $j+1$ are breakpoints and there are no breakpoints between these positions. The number of breakpoints in a permutation π is represented by $tb(\pi)$. The identity permutation is the only permutation that contains no breakpoints. A permutation can contain at most $n+1$ breakpoints.

A transposition can remove at most three breakpoints from a permutation. So we can immediately obtain the following lower bound:

$$td(\pi) \geq \frac{tb(\pi)}{3}.$$

Bafna and Pevzner [BP98][7] introduce the important concept of the *transposition cycle graph* for sorting by transpositions. The transposition cycle graph of π, $tG(\pi)$, is a directed edge-coloured graph with vertex set $\{0, \ldots, n+1\}$, grey edge set $\{(i, i+1) : 0 \leq i \leq n\}$, and black edge set $\{(\pi(i), \pi(i-1)) : \ 1 \leq i \leq n+1\}$. Note that the edge (u, v) is directed from u to v.

Since every incoming edge of a vertex can be uniquely paired with an outgoing edge of the alternative colour, the graph can be completely decomposed into alternating cycles, and furthermore this decomposition is unique. An alternating cycle is *odd* if it contains an odd number of black edges. The number of odd alternating cycles in $tG(\pi)$ is denoted by $tc_{odd}(\pi)$.

The transposition cycle graph is important because Bafna and Pevzner show that

$$td(\pi) \geq \frac{n+1-tc_{odd}(\pi)}{2}.$$

This is generally a much better lower bound for transposition distance than the bound based on breakpoints. It is still an open question as to whether sorting by transpositions can be solved in polynomial time, but Bafna and Pevzner use properties of cycles in the transposition cycle graph to obtain a polynomial time 3/2-approximation algorithm for the problem.

[7] an earlier version of this paper appeared as [BP95b]

The transposition diameter of the symmetric group, $tD(n)$, is still unknown. Bafna and Pevzner prove that

$$\frac{n}{2} \leq tD(n) \leq \frac{3n}{4}.$$

They note that, for $3 \leq n \leq 10$, $tD(n) = \lfloor n/2 \rfloor + 1$, and that this value is achieved by the *reverse permutation* $R_n = [n\ n-1\ \ldots\ 1]$. They also state that $td(R_n) \leq \lfloor n/2 \rfloor + 1$. In Chapter 3 we prove that $td(R_n) = \lfloor n/2 \rfloor + 1$, and therefore $tD(n) \geq \lfloor n/2 \rfloor + 1$. This result was obtained independently by Meidanis, Walter and Dias [MWD97b].

Guyer, Heath and Vergara [GHV95] describe heuristics for sorting by transpositions that are based on the length of a longest increasing subsequence and the length of a longest increasing substring of the permutation. In Chapter 3 we present evidence that these heuristics are not very good.

Jordan [Jor95] makes an interesting connection between topology and sorting by transpositions. He defines a surface based on $tG(\pi)$, and shows that the lower bound for $td(\pi)$ based on $tG(\pi)$ is the so-called genus of this surface.

1.7 Sorting by restricted transpositions

An insertion of the leading element is a transposition of the form $\tau(1, 2, i)$. Aigner and West [AW87] investigate the problem of sorting a list by repeated insertion of the leading element. They show that $n - k$ insertions are required, where n is the length of the list and k is the largest value such that the last k elements in the list form an increasing sequence. This is proved by observing that the elements in positions $n - k$ and $n - k + 1$ must be in the wrong relative order at first. The only way their ordering can change is if the element at position $n - k$ reaches the front. This can only happen after at least $n - k - 1$ insertions of the leading element. By extending the increasing sequence at the end of the list with every insertion of the leading element, the list can be sorted by $n - k$ insertions.

An insertion of any element is a transposition of the form $\tau(i, i+1, j)$ or $\tau(i, j, j+1)$. Knuth [Knu73] (Exercise 5.2.1–39), and Heath and Vergara [HV97] show that exactly $n - lis(\pi)$ insertions are required to sort π, where $lis(\pi)$ is the length of a longest increasing subsequence of π. The result is obtained because every element that is not in the longest increasing subsequence must be moved, and every such element can be moved into the correct position in the increasing subsequence by one insertion.

A *bounded block-move* is a transposition of the form $\tau(i, j, k)$, where $k \leq i + b$ for some fixed parameter b. Heath and Vergara [HV97] investigate the problem of sorting by bounded block-moves. A *short block-move* is a transposition of the form $\tau(i, i + 1, i + 2)$, $\tau(i, i + 2, i + 3)$, or $\tau(i, i + 1, i + 3)$, i.e., a bounded block-move with $b = 3$. Heath and Vergara [HV97] show that the short block-move diameter of S_n is $\lceil \binom{n}{2}/2 \rceil$. In [HV98] they obtain a polynomial-time 4/3-approximation algorithm for calculating the short block-move distance of a given permutation. They also describe

7 6 3 1 5 8 2 4
7 6 3 1 2 4 5 8

7 6 3 1 5 8 2 4
1 5 6 3 7 8 2 4

Figure 1.5: Some example block-interchanges.

classes of permutations for which they can calculate the exact short block-move distance in polynomial-time.

1.8 Sorting by block-interchanges

A block-interchange is a generalisation of a transposition. In a block-interchange two non-intersecting substrings of any length are swapped in the permutation, whereas in a transposition the substrings must be adjacent. Some example block-interchanges are shown in Figure 1.5. The *block-interchange distance* of π, $bd(\pi)$, is the length of a shortest sequence of block-interchanges that transforms π into $\imath$. *Sorting by block-interchanges* is the problem of finding a sequence of block-interchanges of length $bd(\pi)$ that sorts π.

Christie [Chr96] introduces the problem of sorting by block-interchanges, describes a polynomial time algorithm for sorting by block-interchanges, and determines the block-interchange diameter of S_n. We present these results in Chapter 4.

1.9 Sorting by reversals and transpositions

All of the problems considered so far, allow only one kind of global rearrangement operation. However, we can, of course, define problems in which more than one kind of global rearrangement can be performed. For example, *Sorting by reversals and transpositions* is the problem of finding a shortest sequence of reversals and/or transpositions that sorts a given permutation. Walter, Dias, and Meidanis [WDM98] investigate this problem. It remains open as to whether the problem can be solved in polynomial time, but they describe a 3-approximation algorithm for the problem.

They also investigate a signed version of sorting by reversals and transpositions. For this problem they describe a 2-approximation algorithm, and show that the reversal and transposition diameter of Σ_n is at least $\lfloor n/2 \rfloor + 2$. Hannenhalli, Chappey, Koonin and Pevzner [HCKP95] use exhaustive search to solve a particular instance of length 7 of sorting signed permutations by transpositions and reversals.

Gu, Peng and Sudborough [GPS96] investigate the the problem of sorting signed permutations by reversals, transpositions and *trans-reversals*, where a trans-reversal is a simultaneous transposition and reversal. More formally, the trans-reversal $\tau\rho(i, j, k)$

removes the substring $\pi[i..j-1]$ from π, reverses this substring, and inserts it in front of the element in position k of π. They describe a polynomial time $2(1+1/k)$-approximation algorithm for their problem, where $k \geq 3$ is any fixed integer [8]. Gu, Peng, Iwata, and Chen [GPIC97] describe a simple greedy approximation algorithm for this problem that in tests finds solutions that are very close to optimal.

Blanchette, Kunisawa and Sankoff [BKS96] conclude from experimental evidence that a weighted version of sorting signed permutations by reversals and transpositions in which transpositions are given roughly twice the weight of reversals is likely to give more biologically meaningful results than the unweighted version.

1.10 Generating the symmetric group

Let $g_1, \ldots, g_k$ be permutations of the set $\{1 \ \ldots \ n\}$. The set containing all the permutations that can be expressed as a product of these permutations is a group. We say that $g_1, \ldots, g_k$ are *generators* of this group. The size of the group that is *generated* may be exponential in n and k. However, Furst, Hopcroft and Luks [FHL80] describe a polynomial-time algorithm to test for membership of such a group.

Reversals, transpositions and other global rearrangements can be represented by permutations, e.g., $[3\ 2\ 1\ 4\ \ldots\ n]$ is a permutation that reverses the first three elements of a permutation. Finding $d(\pi)$ for a particular kind of rearrangement is equivalent to finding a minimum length sequence of such permutations that produces π. Even and Goldreich [EG81] show, by a transformation from the 3-exact-cover problem, that the general problem of finding a minimal length sequence of generators that produce a given permutation from a given set of generators is NP-hard. Jerrum [Jer85] has shown that the problem is PSPACE-complete even when there are only two generators given.

However, in the problems that we study, the generator sets are fixed, i.e., they are not part of the problem instances. This can make the problems more tractable. Some computationally tractable fixed generator set problems are described in [Jer85].

1.11 Sorting by translocations

Organisms with more than one chromosome can evolve by genome rearrangements called translocations. This leads to a genome rearrangement problem based on translocations.

A *translocation* as an operation that acts on two strings X and Y such that a prefix X' of X is swapped with a prefix or a reversed suffix Y' of Y. Note that the swapped strings can have different lengths, but must have length at least one and must be shorter than the entire string (i.e. $1 \leq |X'| \leq |X| - 1$ and $1 \leq |Y'| \leq |Y| - 1$). Some example translocations are shown in Figure 1.6. *Sorting by translocations* is the

[8] In a forthcoming paper Gu, Peng and Sudborough [GPS99] describe a 2-approximation algorithm for this problem.

[5 6 4] [1 2 3 7 8]
[1 2 3 4] [5 6 7 8]

[8 7 3 4] [5 6 2 1]
[1 2 3 4] [5 6 7 8]

Figure 1.6: Some example translocations.

problem of finding a shortest sequence of of translocations, of length $tld(A, B)$, that transforms A into B, where A and B are sets of strings. In this problem it is assumed that each element appears in exactly one string of A and in exactly one string of B. We also consider strings X and Y to be identical, if $X = Y$ or $X = \overline{Y}$, where $\overline{Y}$ is the reverse of Y.

Kececioglu and Ravi [KR95] describe a 2-approximation algorithm for sorting by translocations, and a 2-approximation algorithm for sorting by translocations and reversals. Hannenhalli [Han96][9] describes a signed version of sorting by translocations. He shows, using methods similar to those used to study sorting signed permutations by reversals, that the signed version of the problem can be solved in polynomial-time. The time complexity of the unsigned version remains open.

It is possible to extend the definition of a translocation to include operations that act on empty prefixes and suffixes, as well as prefixes and suffixes that are the entire string (i.e. to allow $0 \leq |X'| \leq |X|$ and $0 \leq |Y'| \leq |Y|$ in the definition of translocation given above). With this definition of translocation, a *fusion* is a translocation that joins two strings together, and a *fission* is an translocation that splits a single string into two strings. Hannenhalli and Pevzner [HP95c] describe a polynomial algorithm for a signed version of sorting by translocations, fusions, fissions and reversals.

1.12 Syntenic edit distance

Ferretti, Nadeau and Sankoff [FNS96] introduce a problem that is similar to sorting by translocations except that in their problem translocations act on sets instead of strings. Their motivation for this problem is that, in practice, the order of genes in a chromosome is often unknown, whereas the chromosomal assignment of genes is known.

They define a *translocation* to be an operation that transforms sets X and Y into $(X - X') \cup Y'$ and $(Y - Y') \cup X'$, for some $X' \subseteq X$ and $Y' \subseteq Y$. Note that if $Y = \{\}$ then the operation is a *fission*, and that if $X = X'$ and $Y' = \{\}$ then the operation is a *fusion*.

The *syntenic edit distance*, $sd(A, B)$, is the minimum number of these translocations that can transform A into B, where A and B are sets of sets. We define $|A|$ to be the number of sets contained in A.

[9] an earlier version of this paper appeared as [Han95a]

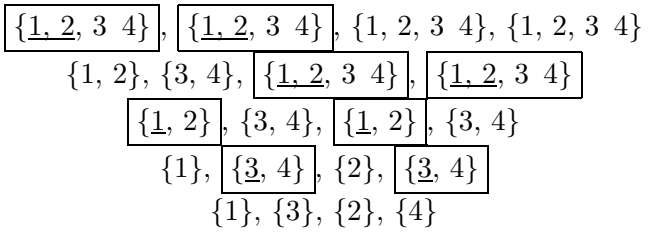

Figure 1.7: An example syntenic edit distance problem.

There is a *compact representation* of the problem in which we relabel all the elements that occur in the first set of A by 1, and all the elements that occur in the second set of A by 2, and so on (removing any duplicates in the same set). Solving the problem, in this form, requires a sequence of translocations that transforms B into $\{\{1\}, \{2\}, \ldots, \{|A|\}\}$. So in a sense the syntenic edit distance problem is a set sorting problem rather than a permutation sorting problem. An example syntenic edit distance problem and its solution is shown in Figure 1.7.

Ferreti, Nadeau and Sankoff [FNS96] present a simple heuristic for approximating the syntenic edit distance. DasGupta, Jiang, Kannan, Li, and Sweedyk [DJK+97] show that calculating syntenic edit distance is an NP-hard problem. They also describe a simple 2-approximation algorithm for the problem.

1.13 Other related problems

Pevzner and Waterman [PW95] present a list of open problems in computational molecular biology that includes a section on genome rearrangement problems.

Lowrance and Wagner [LW75] study the problem of calculating the edit distance between two strings where the set of allowable edit operations is extended to include swapping two adjacent characters. They describe a polynomial algorithm for the problem when there are special restrictions on the weight of a swap operation. Wagner [Wag83] shows that in general calculating the extended edit distance is NP-hard.

A mathematical puzzle that is somewhat related to sorting by reversals, is the problem of reversing a train using a short spur line attached to the main track. This problem is introduced by Dewdney [Dew87], and studied by Amato, Blum, Irani and Rubinfeld [ABIR89] and Aggarwal and Leighton [AL90].

A shuffle operation on a deck of playing cards is similar to a rearrangement operation in a genome rearrangement problem. Aldous and Diaconis [AD86] and Diaconis, McGrath and Pitman [DMP95] present statistical analyses of card shuffling techniques.

Similarly, twist operations on a Rubik's cube are like rearrangements in genome rearrangement problems. Eidswick [Eid86] presents some methods that are useful for solving the Rubik's cube and similar problems.

Chapter 2

Sorting by Reversals

2.1 Introduction

In this chapter we study the problem of sorting by reversals. The significant results of this chapter are as follows:

- we define a graph called the reversal graph that is useful for finding sequences of reversals that sort permutations (Section 2.3).
- we present a polynomial-time 3/2-approximation algorithm for sorting by reversals (Section 2.4). This is currently the best known approximation guarantee for sorting by reversals.
- we show that it is possible to decide in polynomial-time whether a permutation can be sorted using only reversals that remove two breakpoints (Section 2.5). This disproves a conjecture of Kececioglu and Sankoff [KS95] that this special case of sorting by reversals is NP-hard.

We begin the chapter with a section of standard definitions and results that are useful for studying sorting by reversals.

2.2 Definitions

Let π be a permutation of length n. As is standard, we assume that π is extended so that $\pi(0) = 0$ and $\pi(n+1) = n+1$, so that the first and last elements of the permutation can be dealt with in the same way as all the other elements of the permutation. The *reversal* $\rho = \rho(i, j)$ (where $1 \leq i < j \leq n+1$) transforms π into $\pi \cdot \rho = [\pi(0) \ \ldots \ \pi(i-1) \ \pi(j-1) \ \ldots \ \pi(i) \ \pi(j) \ \ldots \ \pi(n+1)]$. The *reversal distance*, $rd(\pi)$, between π and ι is the length of a shortest sequence of reversals that transforms π into ι. *Sorting by reversals* is the problem of finding a sequence of reversals of length $rd(\pi)$ that sorts π.

We say that $\pi(i)$ is to the *left* of $\pi(j)$ if $i < j$. Similarly $\pi(i)$ is to the *right* of $\pi(j)$ if $i > j$. A *reversal breakpoint* is a position i in the permutation such that $1 \leq i \leq n+1$

and $|\pi(i) - \pi(i-1)| \neq 1$. The number of reversal breakpoints in a permutation π is denoted by $rb(\pi)$. A *reversal adjacency* is a position i in the permutation such that $1 \leq i \leq n+1$ and $|\pi(i) - \pi(i-1)| = 1$. A *strip* is a substring $\pi[i, j]$, $(i \leq j)$, of π such that i and $j+1$ are breakpoints, but no breakpoints lie between these positions. Note that, for conciseness, in the rest of this chapter we may talk of breakpoints and adjacencies where, of course, we mean reversal adjacencies and reversal breakpoints.

The identity permutation is the only permutation containing no breakpoints, and a permutation of length n can have at most $n+1$ breakpoints. For any reversal ρ, $rb(\pi) - rb(\pi \cdot \rho) \in \{-2, -1, 0, 1, 2\}$. A reversal ρ is a *k-reversal* if $rb(\pi) - rb(\pi \cdot \rho) = k$.

The *(reversal) cycle graph* $rG(\pi)$ is an edge coloured graph that was introduced by Bafna and Pevzner [BP96]. The graph contains a vertex for each element in the permutation (including 0 and $n+1$). So $rG(\pi)$ has vertex set $\{0, \ldots, n+1\}$. Two vertices are joined by a black edge if the elements they represent form a breakpoint in π. Therefore $rG(\pi)$ has black edge set $\{\{\pi(i-1), \pi(i)\} : 1 \leq i \leq n+1,\ i \text{ is a breakpoint}\}$. Two vertices i and $i+1$ are joined by a grey edge if these elements are not consecutive in π. Therefore $rG(\pi)$ has grey edge set $\{\{i, i+1\} : 0 \leq i \leq n,\ i \text{ and } i+1 \text{ are not consecutive in } \pi\}$. In fact, the relationship between vertices of $rG(\pi)$ and the elements of π that they represent is so close that we often treat both objects as if they were the same thing. For example, we often refer to elements of $rG(\pi)$ instead of vertices of $rG(\pi)$.

An *alternating cycle* is a cycle that has edges of alternating colours. Henceforth all cycles referred to in the cycle graph will be alternating cycles. The *length* of a cycle is the number of black edges it contains. A *k-cycle* is a cycle of length k. A *long cycle* is a cycle of length greater than two.

The cycle graph can be completely decomposed into edge-disjoint cycles, because each vertex has an equal number of incident grey and black edges. However there are likely to be many different such *cycle decompositions* of $rG(\pi)$. The maximum number of cycles in any cycle decomposition of $rG(\pi)$ is denoted by $rc(\pi)$. The cycle graph and cycle decompositions are important because they give us the following lower bound for $rd(\pi)$.

Theorem 2.2.1 (Bafna and Pevzner) *For any permutation π,*

$$rd(\pi) \geq rb(\pi) - rc(\pi).$$

2.3 Reversal graphs simplify sorting by reversals

In this section we show how to use a cycle decomposition $\mathcal{C}$ of $rG(\pi)$ to find a sequence of reversals that sorts π. To help us find the sequence of reversals we use another graph called the *reversal graph* $rR(\mathcal{C})$. Note that, of course, $rR(\mathcal{C})$ does depend on π, but for conciseness we do not show this in our notation.

To construct the reversal graph we shall use the *augmented cycle graph* $rG'(\pi)$. This graph is constructed from $rG(\pi)$ by adding black edges and grey edges between adjacent elements of π, and directing all black edges in the graph from $\pi(i)$ to $\pi(i+1)$. In other words, $rG'(\pi)$ is an edge coloured graph with vertex set $\{0, \ldots, n+1\}$, grey edge set $\{\{i, i+1\} : 0 \leq i \leq n\}$, and black edge set $\{(\pi(i), \pi(i+1)) : 0 \leq i \leq n\}$. For a particular black edge $(\pi(i), \pi(i+1))$, $\pi(i)$ is said to be the *tail*, and $\pi(i+1)$ is said to be the *head*.

A (alternating) cycle in $rG'(\pi)$ does not need to respect the directions of the black edges. So the augmented cycle graph can be decomposed into cycles just like the cycle graph. Note that an adjacency in π is a 1-cycle in $rG'(\pi)$. We can use this fact to transform a cycle decomposition $\mathcal{C}$ of $rG(\pi)$ into a cycle decomposition of $rG'(\pi)$. Let $\mathcal{C}^+$ denote the cycle decomposition of $rG'(\pi)$ obtained from a cycle decomposition $\mathcal{C}$ of $rG(\pi)$ by adding 1-cycles to $\mathcal{C}$.

Given a permutation π, and a particular cycle decomposition $\mathcal{C}$ of $rG'(\pi)$, we construct the *reversal graph* $rR(\mathcal{C})$ as follows. We begin with the vertex set $\{u_0, \ldots, u_n\}$. Each vertex in this set is associated with a grey edge of $rG'(\pi)$. The vertex associated with grey edge $\{i, i+1\}$ is denoted by u_i. We colour u_i *blue* if $\{i, i+1\}$ is part of a cycle, according to $\mathcal{C}$, in which it connects the head of a black edge to the tail of a black edge. Otherwise we colour u_i *red*. The grey edge of $rG'(\pi)$ associated with a vertex u of $rR(\mathcal{C})$ is denoted by $g(u)$.

Let u be a vertex in $rR(\mathcal{C})$ such that $g(u)$ is part of cycle C of the cycle decomposition $\mathcal{C}$. We define $l_g(u)$ and $r_g(u)$ to be the positions in π of the leftmost and rightmost elements, respectively, that are incident to $g(u)$. So $l_g(u_i) = \min(\pi^{-1}(i), \pi^{-1}(i+1))$, and $r_g(u_i) = \max(\pi^{-1}(i), \pi^{-1}(i+1))$. Similarly, we define $l_b(u)$ and $r_b(u)$ to be the positions of the leftmost and rightmost black edges, respectively, that are joined by $g(u)$ in C. (The position of a black edge is the position of the rightmost element it is incident to.) Note that, unlike $l_g(u)$ and $r_g(u)$, the definitions of $l_b(u)$ and $r_b(u)$ depend on the cycle decomposition $\mathcal{C}$. However $l_g(u) \leq l_b(u) \leq l_g(u)+1$, and $r_g(u) \leq r_b(u) \leq r_g(u)+1$.

We connect vertices u and v of $rR(\mathcal{C})$ with an edge if

$$
\begin{array}{ll}
(i) & l_b(u) < l_b(v) < r_b(u) < r_b(v), \text{ or} \\
(ii) & l_b(v) < l_b(u) < r_b(v) < r_b(u), \text{ or} \\
(iii) & l_g(u) < l_g(v) < r_g(u) < r_g(v), \text{ or} \\
(iv) & l_g(v) < l_g(u) < r_g(v) < r_g(u).
\end{array}
$$

Example A cycle decomposition of the augmented cycle graph of $\pi = [7\ 5\ 6\ 3\ 2\ 4\ 1]$ is shown in Figure 2.1. Given this cycle decomposition $l_g(u_0) = 0$, $r_g(u_0) = 7$, $l_g(u_2) = 4$, and $r_g(u_2) = 5$, whereas $l_b(u_0) = 1$, $r_b(u_0) = 7$, $l_b(u_2) = 5$, and $r_b(u_2) = 5$. The reversal graph of this cycle decomposition is shown in Figure 2.2. Note that in this figure vertex u_i is given the label i. □

Of course, reversal graphs are not unique for π, because cycle decompositions are not unique.

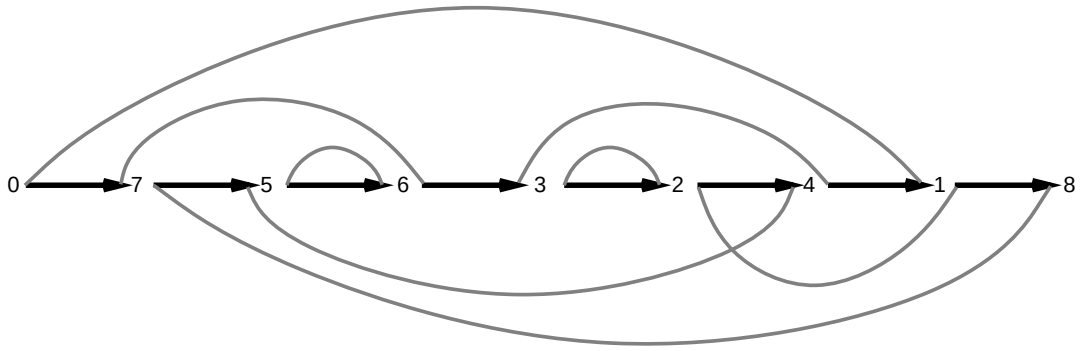

Figure 2.1: An example cycle decomposition of $rG'(\pi)$.

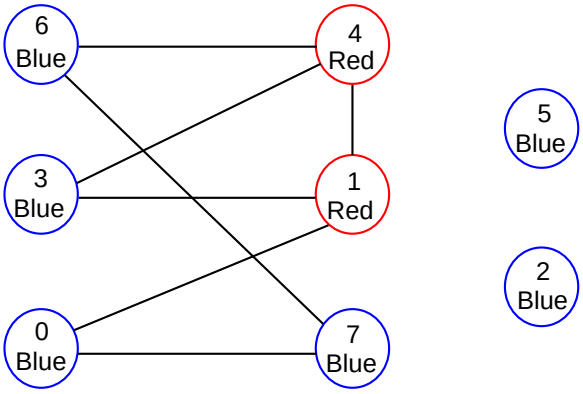

Figure 2.2: An example reversal graph

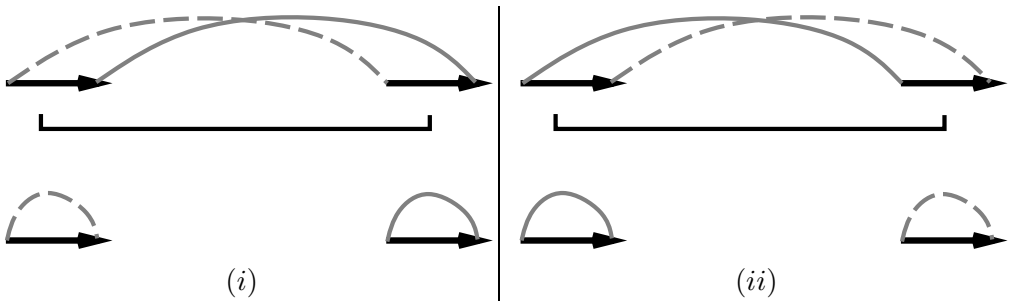

Figure 2.3: The effect of $\rho(u)$ when u is red.

The reversal $\rho(i,j)$ *acts* on the black edges $(\pi(i-1),\pi(i))$ and $(\pi(j-1),\pi(j))$ of $rG'(\pi)$. The augmented cycle graph of the resulting permutation is identical to $rG'(\pi)$ except that these two black edges are removed and replaced by $(\pi(i-1),\pi(j-1))$ and $(\pi(i),\pi(j))$, and the direction is flipped of each black edge of the form $(\pi(k),\pi(k+1))$ where $i \leq k \leq j-2$.

Reversal graphs are given their name because each vertex u in the graph has a reversal $\rho(u)$ associated with it. We define $\rho(u)$ to be the reversal that acts on the two black edges of $rG'(\pi)$ that are joined by $g(u)$ in C, where C is the cycle of $\mathcal{C}$ containing $g(u)$. If C is a 1-cycle, then $\rho(u)$ is the identity reversal (i.e. the reversal that does nothing).

Now let us consider the effect of applying $\rho(u)$ when this reversal is not the identity reversal. As was described above $rG'(\pi \cdot \rho(u))$ is identical to $rG'(\pi)$ except that two black edges have been removed, two have been added, and the direction of some black edges has been flipped. Clearly every cycle in $\mathcal{C}$, except the cycle C containing the removed black edges, exists in $rG'(\pi \cdot \rho(u))$. So we can find a cycle decomposition of $rG'(\pi \cdot \rho(u))$ that contains all the cycles in $\mathcal{C}$ except C. The edges of $rG'(\pi \cdot \rho(u))$ that are not part of these cycles form either one or two cycles depending on the colour of u.

If u is a red vertex then the edges of C now form two cycles in $rG'(\pi \cdot \rho(u))$. These cycles are length 1 and length $l-1$, where l is the length of C. This is illustrated in Figure 2.3. In this figure $g(u)$ is represented by a grey edge, and the other path connecting the black edges joined by $g(u)$ is represented by a dotted grey edge.

If u is a blue vertex then the edges of C form a cycle in $rG'(\pi \cdot \rho(u))$. This is illustrated in Figure 2.4.

We denote by $\mathcal{C} \cdot \rho(u)$ the cycle decomposition of $rG'(\pi \cdot \rho(u))$ that contains all the cycles in $\mathcal{C}$ except C, replacing C with one or two cycles as described above. The following lemma can be used to generate the reversal graph $rR(\mathcal{C} \cdot \rho(u))$ of $\pi \cdot \rho(u)$ from $rR(\mathcal{C})$.

Lemma 2.3.1 *Let u be a vertex of $rR(\mathcal{C})$ such that $g(u)$ is part of cycle C. Then $rR(\mathcal{C} \cdot \rho(u))$ can be derived from $rR(\mathcal{C})$ by making the following changes to $rR(\mathcal{C})$:*
(i) For each vertex v ($v \neq u$), change the colour of v if and only if $\{u,v\}$ is an edge in

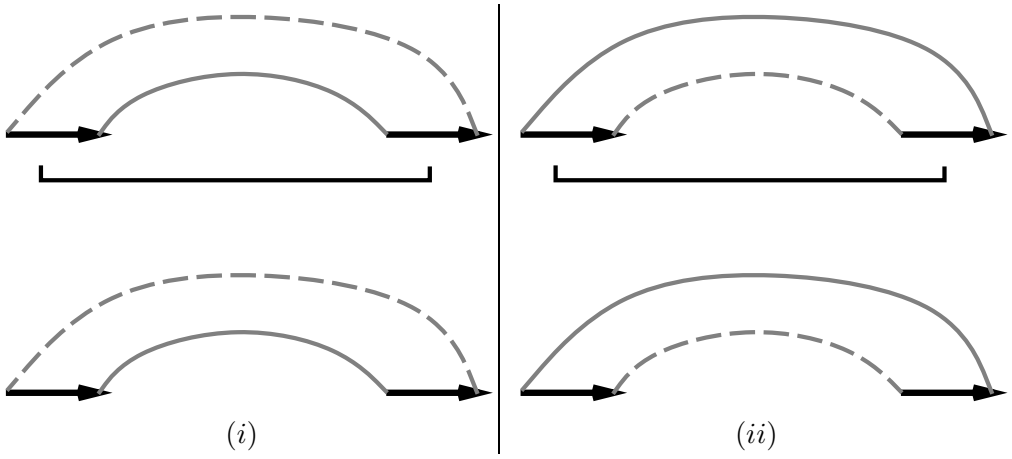

Figure 2.4: The effect of $\rho(u)$ when u is blue.

$rR(\mathcal{C})$.
(ii) For each pair of vertices v and w in $rR(\mathcal{C})$, such that $\{u, v\}$ and $\{u, w\}$ are edges in $rR(\mathcal{C})$, flip the adjacency of v and w.
(iii) If u is a red vertex, make it an isolated blue vertex.

Proof We first prove that these alterations to the graph are necessary.
(i) Suppose $\{u, v\}$ is an edge of $rR(\mathcal{C})$. Then $\rho(u)$ changes the direction of one of the black edges (in the augmented cycle graph) that is incident to $g(v)$, but leaves the direction of the other black edge alone. So the colour of v is different in $rR(\mathcal{C} \cdot \rho(u))$ than in $rR(\mathcal{C})$.
(ii) Suppose that $\{u, v\}$ and $\{u, w\}$ are edges of $rR(\mathcal{C})$. Then because the order of elements of π between positions $l_b(u)$ and $r_b(u) - 1$ (inclusive) are reversed by $\rho(u)$ it transpires that $\{v, w\}$ is an edge of $rR(\mathcal{C} \cdot \rho(u))$ if and only if $\{v, w\}$ is not an edge of $rR(\mathcal{C})$.
(iii) If u is red, then as shown in Figure 2.3, $\rho(u)$ transforms $rG'(\pi)$ so that $g(u)$ is part of a 1-cycle. So u should be blue and isolated in $rR(\mathcal{C} \cdot \rho(u))$.

We now prove that no more alterations need to be made to the graph.
(i) If $\{u, v\}$ is not an edge of $rR(\mathcal{C})$ then the directions of both black edges incident to $g(v)$ remain the same or are both reversed by the action of $\rho(u)$. Therefore the colour of any vertex not adjacent to u should remain the same.
(ii) If $\{u, v\}$ is not an edge of $rR(\mathcal{C})$ then $\rho(u)$ does not change which vertices are adjacent to v. So the adjacencies of vertices not adjacent to u remain the same.
(iii) If u is blue then as shown in Figure 2.4, u is still blue in $rR(\mathcal{C} \cdot \rho(u))$. It is also clear that $\{u, v\}$ is an edge of $rR(\mathcal{C} \cdot \rho(u))$ if and only if $\{u, v\}$ was an edge of $rR(\mathcal{C})$.
□

From this lemma we see that, if u is a red vertex in $rR(\mathcal{C})$, then we obtain $rR(\mathcal{C} \cdot \rho(u))$ from $rR(\mathcal{C})$ by flipping the colour of every vertex adjacent to u, flipping the adjacency

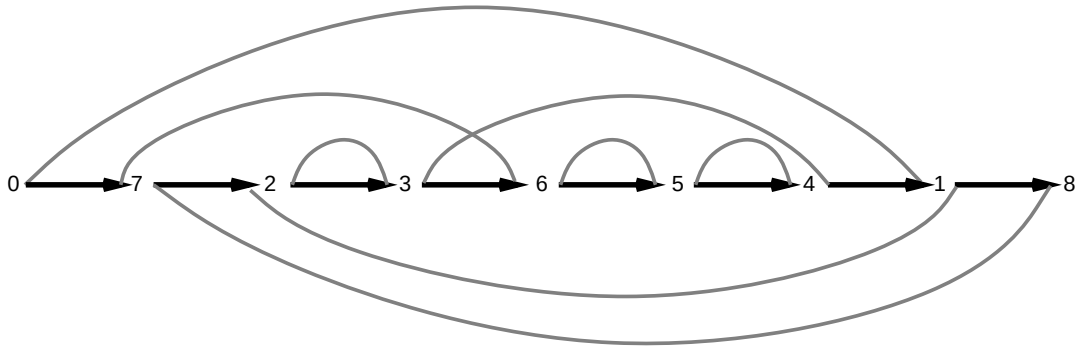

Figure 2.5: The resulting cycle decomposition.

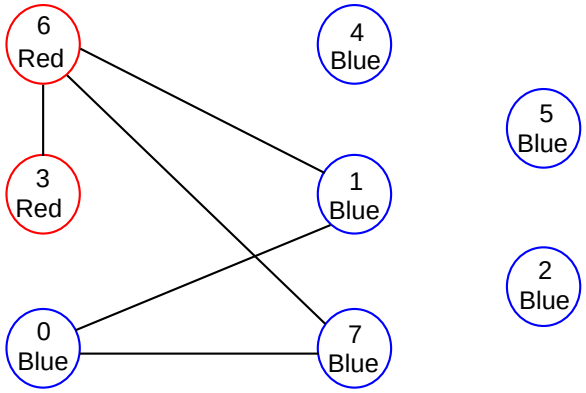

Figure 2.6: The resulting reversal graph.

of every pair of vertices adjacent to u, and making u an isolated blue vertex.

Similarly, if u is a blue vertex then we obtain $rR(\mathcal{C} \cdot \rho(u))$ from $rR(\mathcal{C})$ by flipping the colour of every vertex adjacent to u, and flipping the adjacency of every pair of vertices adjacent to u. Note that in this case u remains blue and is still adjacent to the same vertices.

Example Applying the reversal represented by vertex 4 in Figure 2.2 results in the cycle decomposition in Figure 2.5. The reversal graph of this cycle decomposition is shown in Figure 2.6. □

As a simple consequence of Lemma 2.3.1 we state the following corollary without proof.

Corollary 2.3.1 *A reversal represented by a vertex u in $rR(\mathcal{C})$ affects only vertices that are in the same connected component as u.*

The reversal graph for the identity permutation consists of $n+1$ isolated blue vertices. So to sort π, a sequence of reversals is required that transforms the reversal graph into a graph with $n+1$ isolated blue vertices. A sequence of reversals, each of the form $\rho(u)$ for some vertex u in $rR(\mathcal{C})$, that transforms $rR(\mathcal{C})$ into $n+1$ isolated blue vertices is called an *elimination sequence for* $rR(\mathcal{C})$. Similarly, a sequence of reversals, each of the form $\rho(u)$ for some vertex u in $rR(\mathcal{C})$, that transforms a connected component of $rR(\mathcal{C})$ containing k vertices into k isolated blue vertices is called an *elimination sequence* for that component. We now show that there are two kinds of component in $rR(\mathcal{C})$ and we show how to eliminate each kind, using only reversals represented in the graph. But first, we need some lemmas about the structure of $rR(\mathcal{C})$.

Cycles C and D of $\mathcal{C}$ *interleave* if $rR(\mathcal{C})$ contains vertices u and v, arising from cycles C and D respectively, such that $\{u, v\}$ is an edge of the reversal graph. Grey edges of $rG'(\pi)$ *interleave* if the vertices that represent them are adjacent in the reversal graph.

Lemma 2.3.2 *Isolated blue vertices in $rR(\mathcal{C})$ correspond to 1-cycles in $\mathcal{C}$.*

Proof From the definition of cycle graphs it is easy to verify that all 1-cycles in $\mathcal{C}$ have corresponding isolated blue vertices in $rR(\mathcal{C})$.

Now, suppose, for a contradiction, that v is an isolated blue vertex that arises from a cycle C in $\mathcal{C}$ of length greater than one. Then v represents a grey edge g that joins two black edges of C. Name the elements of π incident to the grey edge, x and x', and the two remaining elements incident to these black edges, y and z. Since v is blue, π can take only two forms as shown below.

$$\pi = \begin{array}{ll} \ldots\ x\ y\ \ldots\ z\ x'\ \ldots & \text{(i)} \\ \ldots\ y\ x\ \ldots\ x'\ z\ \ldots & \text{(ii)} \end{array}$$

Now consider an algorithm that visits each vertex of the augmented cycle graph in the order 0, 1, ..., n, $n+1$. This algorithm follows each grey edge in the graph. Now in order to visit the smallest value between x and x' it must travel along a grey edge that interleaves with g. But that would mean v was not isolated, so x must be contiguous with x' in π. But then the cycle containing the black edge (x, x') must give rise to vertices in $rR(\mathcal{C})$ that are connected to v. This contradiction proves the lemma. □

Let $\mathcal{C}$ be a cycle decomposition of $rG'(\pi)$. A cycle C of $\mathcal{C}$ is said to be *unoriented* if every grey edge in the cycle connects the head of a black edge to the tail of a black edge. Otherwise, the cycle is said to be *oriented*. (Caprara, perhaps more logically, calls unoriented cycles directed cycles, and oriented cycles undirected, but we keep to the more standard terminology.) Note that all vertices of $rR(\mathcal{C})$ arising from an unoriented cycle are blue.

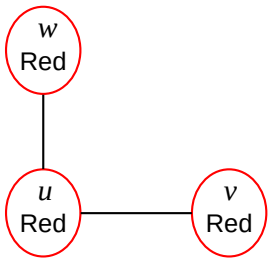

Figure 2.7: A type 1 vertex.

Lemma 2.3.3 *An unoriented* 2*-cycle of* $rG(\pi)$ *must interleave with another cycle in* $rG(\pi)$*.*

Proof Both of the grey edges in an unoriented 2-cycle connect a head to a tail, so both of the vertices of $rR(\pi)$ that represent the 2-cycle are blue. By the definition of $rR(\pi)$ these two vertices are not adjacent. Now by Lemma 2.3.2 the vertices are not isolated. So the vertices must be adjacent to vertices arising from another cycle. Therefore, an unoriented 2-cycle must interleave with another cycle. □

A connected component of $rR(\mathcal{C})$ is *oriented* if it contains a red vertex, or it consists solely of an isolated blue vertex. Otherwise the component is *unoriented*. Let A be a connected component of $rR(\mathcal{C})$, and let u be a vertex in A. We define A_u to be the subgraph of $rR(\mathcal{C} \cdot \rho(u))$ that contains all the vertices of A.

Lemma 2.3.4 *If a component* A *of* $rR(\mathcal{C})$ *is oriented then it contains a red vertex* u *such that every component of* A_u *is oriented (or the component* A *consists of a single isolated blue vertex).*

Proof To set up a contradiction, suppose that A is not a single isolated blue vertex, and that A_u contains an unoriented component for every red vertex u in A. By the definition of unoriented components, the unoriented component in A_u cannot be an isolated blue vertex. So in order that the unoriented component of A_u contains two blue vertices that are joined by an edge, it must be the case that, for every red vertex u in A, there is a pair of distinct vertices v and w such that, either
1) v and w are both red, $\{u, v\}$, $\{u, w\}$ are edges in A and $\{v, w\}$ is not an edge in A (Figure 2.7), or
2) v is red and w blue, $\{u, v\}$, $\{v, w\}$ are edges in A and $\{u, w\}$ is not an edge in A (Figure 2.8).
Further, these vertices, v and w, are part of an unoriented component of A_u.

Call u a *type* 1 *vertex* if only case 1) applies, and otherwise a *type* 2 *vertex*. Let V_r be the set of red vertices in A, so that every vertex in V_r is either a type 1 or a type 2 vertex. Define a mapping $f : V_r \to V_r$ by means of $f(u) = v$ where v is defined

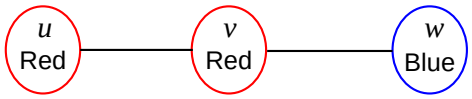

Figure 2.8: A type 2 vertex.

by either case 1) or case 2) above, depending on whether v is a type 1 or a type 2 vertex. For a given red vertex u in A, define the sequence $u_0 = u$, $u_{i+1} = f(u_i)$ for $i \geq 0$. Then because the set V_r is finite, the sequence must cycle, i.e., there must be a smallest integer k ($k \geq 2$) such that $u_j = u_{j+k}$, for any $j \geq x$, where x is some positive integer.

Suppose that the cycle contains a type 2 vertex, u_s. Then there is a blue vertex w such that $\{u_{s+1}, w\}$ is an edge and $\{u_s, w\}$ a non-edge of A. But since u_{s+2} is in a unoriented component of $A_{u_{s+1}}$, it follows that $\{u_{s+2}, w\}$ must be an edge of A. Similarly, $\{u_{s+3}, w\}, \{u_{s+4}, w\}, \ldots$ must be edges of A, but this leads to a contradiction since $u_s = u_{s+k}$, and $\{u_s, w\}$ is a non-edge of A.

Therefore every vertex in the cycle is a type 1 vertex. Let u_s be a vertex in the cycle. Since u_s is a type 1 vertex, there is a red vertex w such that $\{u_s, u_{s+1}\}$, $\{u_s, w\}$ are edges, and $\{u_{s+1}, w\}$ is a non-edge in A. Recall that $u_s = u_{s+k}$. Note that it is impossible to have $u_{s+k-1} = w$ because then u_s would be connected to a red vertex u_{s+1} in $A_{u_{s+k-1}}$. It follows that $\{u_{s+k-1}, w\}$ must be an edge of A. Similarly, $\{u_{s+k-2}, w\}, \{u_{s+k-3}, w\}, \ldots$ must be edges of A, but this leads to a contradiction since $\{u_{s+1}, w\}$ is a non-edge of A. □

Lemma 2.3.5 *Let B be an unoriented component of $rR(\mathcal{C})$. Then B_v is an oriented component for all vertices v in B.*

Proof This is a simple consequence of Lemma 2.3.1. □

It is possible to use Lemma 2.3.4 to find an elimination sequence for an oriented component A of $rR(\mathcal{C})$. We simply apply a reversal represented by a red vertex v, for which all the components of A_v are oriented, and repeat until A has been eliminated completely. Note that this observation together with Lemma 2.3.5 mean that we can always find an elimination sequence for $rR(\mathcal{C})$.

A reversal $\rho(u)$ is a *k-move* if the number of 1-cycles in $\mathcal{C} \cdot \rho(u)$ is k greater than in $\mathcal{C}$. If u is a red vertex then $\rho(u)$ is a 1-move or a 2-move. If u is a blue vertex then $\rho(u)$ is a 0-move.

The following simple proposition will help us prove an important lemma about components of the reversal graph.

Proposition 2.3.1 *Let v be a vertex in $rR(\mathcal{C})$. Then $\rho(v)$ is a 2-move if and only if v arises from an oriented 2-cycle.*

Proof Suppose $\rho(v)$ is a 2-move. Then $\mathcal{C} \cdot \rho(v)$ contains two more 1-cycles than $\mathcal{C}$ by Lemma 2.3.2. So v must be red because otherwise the lengths of cycles in $\mathcal{C} \cdot \rho(v)$ are no different from the lengths of cycles in $\mathcal{C}$. Now because v is a red vertex, $\rho(v)$ splits a cycle into a 1-cycle and a cycle of length $l - 1$, where l is the length of the original cycle. In order for the reversal to be a 2-move, l must be 2, because the lengths of the other cycles are not altered by $\rho(v)$. Therefore v arises from an oriented 2-cycle.

Clearly a reversal on an oriented 2-cycle is a 2-move. □

Lemma 2.3.6 *All the vertices that arise from the same cycle in $\mathcal{C}$ are part of the same connected component of $rR(\mathcal{C})$.*

Proof Suppose, for a contradiction, that a cycle C in $\mathcal{C}$, gives rise to vertices in $rR(\mathcal{C})$ that are in different components.

Now let us imagine an elimination sequence S for $rR(\mathcal{C})$. Each reversal, $\rho(u)$, in this elimination sequence, where u is a red vertex arising from C, splits the cycle into a 1-cycle and a cycle of length $l - 1$. The last reversal of this kind, $\rho(w)$, must therefore be a 2-move, and it must increase the number of isolated blue vertices in the reversal graph by two (Proposition 2.3.1). Let u be a vertex arising from C that is not part of the same component as w.

By Corollary 2.3.1 two components of $rR(\mathcal{C})$ never merge together as a result of applying a reversal during the elimination sequence. So the elimination sequence S defines elimination sequences for each of the components of $rR(\mathcal{C})$, and u and w are never part of the same component. Also by Corollary 2.3.1 the elimination sequence on the component containing w can be applied before any of the other elimination sequences. If we do this, then when we apply $\rho(w)$ the number of isolated blue vertices in the reversal graph must still increase by two (by Corollary 2.3.1). But by Proposition 2.3.1, C must be an oriented 2-cycle at that stage. However, by the definition of $rR(\mathcal{C})$, the two vertices representing an oriented 2-cycle are connected by an edge, whereas C is represented by vertices u and w (at least) that do not share an edge. This contradiction proves the lemma. □

It is possible to use Lemma 2.3.4 and Lemma 2.3.6 to calculate how many reversals are required to eliminate an oriented component of $rR(\mathcal{C})$ as shown by the following proposition.

Proposition 2.3.2 *Let A be an oriented component of $rR(\mathcal{C})$ that contains vertices arising from k different cycles of $rG(\pi)$. Then there is an elimination sequence for A that contains k 2-moves with all the other reversals being 1-moves.*

Proof The elimination sequence found by the method described above always applies a reversal that is represented by a red vertex in the reversal graph. So each reversal is a 1-move or a 2-move. By Proposition 2.3.1 k of the reversals are 2-moves. □

We now describe how to deal with unoriented components. To eliminate an unoriented component we first make it an oriented component as described in Lemma 2.3.5. The resulting oriented component can then be solved as before. So the number of reversals needed to eliminate an unoriented component of $rR(\mathcal{C})$, using only reversals represented in $rR(\mathcal{C})$, is the number stated in the following proposition that can be easily verified.

Proposition 2.3.3 *Let A be an unoriented component of $rR(\mathcal{C})$ that contains vertices arising from k different cycles of $rG(\pi)$. Then there is an elimination sequence for A contains one 0-move, and k 2-moves, with all the other reversals being 1-moves.*

We now combine the last two propositions to find an upper bound on the reversal distance of the permutation.

Theorem 2.3.1 *Let $\mathcal{C}$ be a cycle decomposition of $rG(\pi)$. Then, using only reversals represented in $rR(\mathcal{C}^+)$, π can be sorted by $rb(\pi) - |\mathcal{C}| + ru(\mathcal{C})$ reversals, where $ru(\mathcal{C})$ is the number of unoriented components in $rR(\mathcal{C}^+)$, and $|\mathcal{C}|$ is the number of cycles in $\mathcal{C}$.*

Proof By repeatedly applying Propositions 2.3.2 and 2.3.3 we will find a sequence of reversals that sorts π and contains $ru(\mathcal{C})$ 0-moves, $|\mathcal{C}|$ 2-moves, and $rb(\pi) - 2|\mathcal{C}|$ 1-moves. So the total length of this sequence is $rb(\pi) - |\mathcal{C}| + ru(\mathcal{C})$. □

Note that it may be possible to sort π with fewer reversals by using a different cycle decomposition, or by using reversals that act on black edges of two different cycles.

In fact the bound of Theorem 2.3.1 is very similar to the bound obtained by Hannenhalli and Pevzner [HP95b] for the related problem of sorting signed permutations by reversals. Applied to unsigned permutations, they proved that, for a permutation π and a cycle decomposition $\mathcal{C}$,

$$rd(\pi) \leq rb(\pi) - |\mathcal{C}| + rh(\mathcal{C}) + rf(\mathcal{C}),$$

where $rh(\mathcal{C})$ is the number of *hurdles* contained in $\mathcal{C}$, and $rf(\pi) = 0$, or 1. Hurdles are a subset of the unoriented components of $rR(\mathcal{C})$, and in fact $rh(\mathcal{C}) + rf(\mathcal{C}) \leq ru(\mathcal{C})$. So their bound is tighter than the bound obtained in Theorem 2.3.1.

However, the elimination sequences of Proposition 2.3.2 and Proposition 2.3.3 are crucial for a counting argument used in a later proof that establishes the 3/2 bound of an approximation algorithm. So we prefer not to use the bounds established in [HP95b]. This preference also has the added advantage that in the next section, we do not need to use properties of signed permutations in order to establish results that are purely about unsigned permutations.

2.4 A 3/2-approximation algorithm for sorting by reversals

In this section we describe a 3/2-approximation algorithm for sorting by reversals. The rest of this section is split into five subsections. In the first of these subsections (Section 2.4.1) an upper bound is described that is a sufficient condition for an algorithm to achieve a 3/2-approximation bound. A method of generating cycle decompositions is presented in Section 2.4.2. These cycle decompositions have reversal graphs, as shown in Section 2.4.3, that can be used to sort π with a sequence of reversals that achieves the bound. All the ideas presented in the earlier sections are pulled together in Section 2.4.4, where the 3/2-approximation algorithm is presented. Some concluding remarks appear in Section 2.4.5.

2.4.1 The 3/2-bound

In this section we describe a sufficient condition for an algorithm to achieve a 3/2-approximation bound. In the later sections we set out to produce an algorithm that satisfies this condition.

Theorem 2.4.1 *If $rc_2(\pi)$ is the minimum number of 2-cycles in any maximum cycle decomposition of $rG(\pi)$ then*

$$rd(\pi) \geq \frac{2}{3} rb(\pi) - \frac{1}{3} rc_2(\pi).$$

Proof Bafna and Pevzner [BP96] proved the fundamental lower bound

$$rd(\pi) \geq rb(\pi) - rc(\pi). \tag{2.1}$$

Now let $\mathcal{C}$ be a maximum cycle decomposition of $rG(\pi)$ with the minimum number of 2-cycles. Suppose, in particular, that $\mathcal{C}$ contains $rc_{3*}(\pi)$ cycles of length greater than two. Then, by (2.1),

$$rd(\pi) \geq rb(\pi) - rc_2(\pi) - rc_{3*}(\pi).$$

However, because each long cycle accounts for at least three of the black edges that are not in shorter cycles of the decomposition, it must be that

$$rc_{3*}(\pi) \leq \frac{1}{3}(rb(\pi) - 2rc_2(\pi)).$$

Combining these two inequalities proves the theorem. □

In view of the above theorem it is apparent that an algorithm that sorts π using at most $rb(\pi) - rc_2(\pi)/2$ reversals will achieve a 3/2-approximation bound for sorting by reversals.

2.4.2 The cycle decomposition

In Section 2.4.1 we obtained a bound for reversal distance based on the minimum number of 2-cycles in a maximum cycle decomposition. Perhaps perversely, we now seek a cycle decomposition that contains as many 2-cycles as possible.

We will use a new graph called the *matching graph* $rF(\pi)$ to find our cycle decomposition of $rG(\pi)$. The matching graph contains a vertex for each black edge in $rG(\pi)$. Two vertices, u and v, of $rF(\pi)$ are connected by an edge if $rG(\pi)$ has a 2-cycle that contains both of the black edges represented by u and v. So each edge in $rF(\pi)$ corresponds to a 2-cycle in $rG(\pi)$.

A maximum cardinality matching M of $rF(\pi)$ can certainly be found in polynomial-time. (Algorithms for finding the maximum cardinality matching of a graph are described, for example, in Chapter 9 of [LP86]).

Unfortunately the cycles represented by M may not be edge-disjoint. For example, if $\pi = [0\ 9\ 3\ 4\ 6\ 5\ 8\ 7\ 1\ 10\ 2\ 11]$ then M contains edges representing the cycles $(0, 1, 10, 9)$ and $(9, 10, 2, 3)$, but both these cycles involve the same grey edge $(9, 10)$. However, it is possible to find a cycle decomposition that contains at least $\lceil |M|/2 \rceil$ 2-cycles.

To find an appropriate set of edge-disjoint 2-cycles we use a graph derived from M, called the *ladder graph* $rL(M)$. Let $rL(M)$ be a graph with a vertex for every 2-cycle represented in the matching M. Connect vertices of $rL(M)$ if the 2-cycles that they represent have an edge in common.

Proposition 2.4.1 *$rL(M)$ consists of isolated vertices and simple paths.*

Proof Vertices of $rF(\pi)$ represent black edges of $rG(\pi)$, and edges of M represent 2-cycles of $rG(\pi)$. So, if any 2-cycles represented in M share black edges then M could not be a matching. So the 2-cycles represented by M can share only grey edges. A 2-cycle has two grey edges that could each be shared with a different cycle, so vertices of $rL(M)$ have maximum degree two. If two 2-cycles share a grey edge, but not a black edge then they must share two vertices of $rG(\pi)$. So the two cycles must be as shown in Figure 2.9. If we extend these figures with more 2-cycles, that share grey edges but not black edges, then we end up with 2-cycles sandwiched between other 2-cycles. From the way that the cycles are sandwiched together, it is clearly impossible for $rL(M)$ to contain a cycle. □

We call the 2-cycles represented by edges in M that are represented by isolated vertices in $rL(M)$ *independent 2-cycles*. By contrast we call the other 2-cycles represented by edges in M *ladder cycles*, and the vertices of $rL(M)$ representing these cycles *ladder vertices*. This is because we call a collection of 2-cycles of $rG(\pi)$ represented by vertices that form a path in $rL(M)$ a *ladder*. (We leave it to the reader to work out why we call these structures ladders.) Two very short ladders are shown in Figure 2.9.

Note that either all the cycles in a ladder are oriented, or all the cycles are unoriented. (A cycle of $rG(\pi)$ is oriented or unoriented if and only if the cycle is oriented or

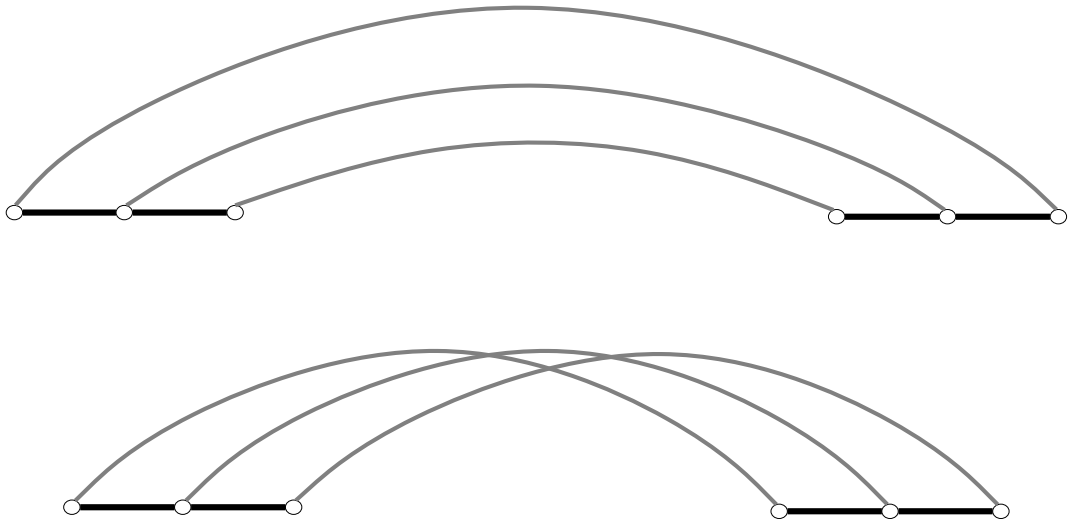

Figure 2.9: Some short ladders.

unoriented respectively in $rG'(\pi)$.) So we define a ladder to be *oriented* or *unoriented* depending on whether the cycles it contains are oriented or not.

Let us suppose that $rL(M)$ contains z isolated vertices and y ladder vertices. Note that $|M| = y + z$.

Theorem 2.4.2 *Given a maximum cardinality matching M of $rF(\pi)$ it is possible to find a cycle decomposition $\mathcal{C}$ of $rG(\pi)$ that contains at least $\lceil y/2 \rceil$ ladder 2-cycles and z independent 2-cycles.*

Proof Construct $rL(M)$ for the matching M. This graph consists of simple paths and isolated vertices. Let $\mathcal{C}$ contain all the independent 2-cycles from $rL(M)$ and every alternate cycle from each ladder. Clearly these 2-cycles are edge-disjoint. The rest of $\mathcal{C}$ can be obtained by adding any cycle decomposition of the remaining edges of $rG(\pi)$. □

Example Suppose $\pi = [0\ 4\ 2\ 6\ 8\ 7\ 3\ 5\ 1\ 9\ 14\ 15\ 12\ 13\ 10\ 11\ 16]$. Then $rG(\pi)$ is the graph shown in Figure 2.10. The matching graph $rF(\pi)$ of this permutation is shown in Figure 2.11. A maximum cardinality matching of this graph is indicated by the edges a, b, c, d, and e. The ladder graph induced by this matching is shown in Figure 2.12. Vertices a, c, d, and e may be selected from this graph. The cycle decomposition $\mathcal{C}$ found by selecting these 2-cycles is shown in Figure 2.13. □

Let $\mathcal{C}$ be the cycle decomposition found in Theorem 2.4.2. A *selected 2-cycle* is a 2-cycle of $rG(\pi)$ that is represented in M, and is part of $\mathcal{C}$. The edges of $rG(\pi)$ that are in 2-cycles represented in M, but not selected cycles, are called *spare edges*.

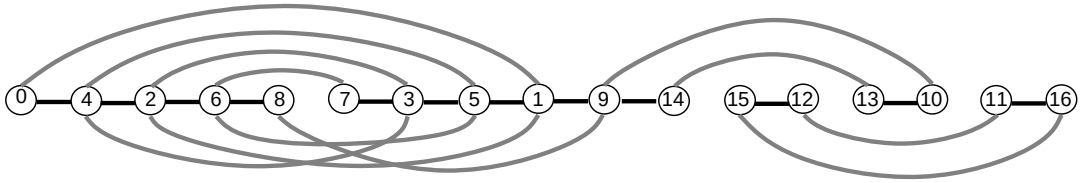

Figure 2.10: $rG(\pi)$ for $\pi = [0\ 4\ 2\ 6\ 8\ 7\ 3\ 5\ 1\ 9\ 14\ 15\ 12\ 13\ 10\ 11\ 16]$.

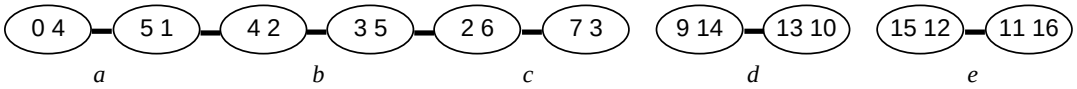

Figure 2.11: $rF(\pi)$ for $\pi = [0\ 4\ 2\ 6\ 8\ 7\ 3\ 5\ 1\ 9\ 14\ 15\ 12\ 13\ 10\ 11\ 16]$.

Figure 2.12: $rL(M)$ where $M = \{a, b, c, d, e\}$ in Figure 2.11.

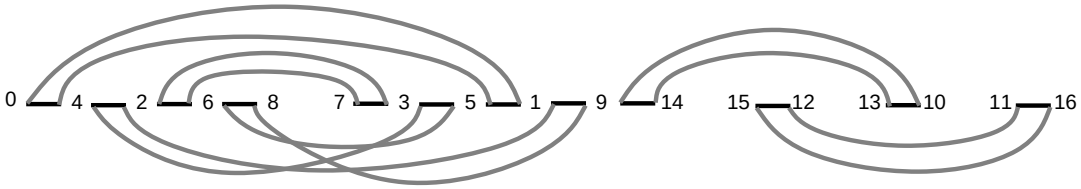

Figure 2.13: The cycle decomposition $\mathcal{C}$.

We now consider the reversal graph $rR(\mathcal{C})$ of this cycle decomposition.

Lemma 2.4.1 *Let C be an unoriented ladder 2-cycle of $\mathcal{C}$. Then the vertices of $rR(\mathcal{C}^+)$ that represent C are part of a component that contains vertices representing a cycle that is not a selected 2-cycle.*

Proof Let C be an unoriented ladder 2-cycle in $\mathcal{C}$.

Suppose that the ladder containing C consists of three or more 2-cycles. Then there must be another selected ladder 2-cycle D such that two spare black edges x and y connect the black edges of C and D in the fashion shown in Figure 2.14. (Note that the roles of C and D can be interchanged but the following argument holds with only minor modifications.)

Now consider the cycle E containing x. Note that E cannot be a 1-cycle because we are dealing with $rG(\pi)$ and not $rG'(\pi)$. Now E interleaves with C or D if it contains a black edge that occurs: (i) before the leftmost edge of C, (ii) between the two black edges of D, or (iii) after the rightmost black edge of C. So E intersects with C of D unless it is a 2-cycle containing x and y. However such 2-cycles cannot exist because $rG(\pi)$ would then contain a cycle with only grey edges, and that is obviously impossible from the definition of $rG(\pi)$. Therefore E intersects with C or D.

By a similar argument we can show that F, the cycle containing y, intersects with C or D. (Note that E and F may be the same cycle.)

Now suppose both of E and F interleave with D but not with C. Then by Lemma 2.3.3 a cycle G interleaves with C. Now G must have a black edge, a, between the two black edges of C, and a black edge b either before the leftmost black edge of C or after the rightmost black edge of C. But since G is not C, E or F, it must be the case that a is between the two black edges of D. So G interleaves with D.

Therefore, by Lemma 2.3.6, vertices representing C are part of the same component in $rR(\mathcal{C}^+)$ as vertices representing E and F which contain spare edges.

Suppose that the ladder containing C is a ladder of length two, as shown in Figure 2.15. (Note that the roles of C and the spare edge can be interchanged but the following argument holds with only minor modifications.) Suppose that the cycle or cycles containing the spare black edges of the ladder do not interleave with C. Then, in a similar way as was explained above, a cycle that interleaves with C must interleave with the cycle containing the spare grey edge of the ladder. So vertices representing C are part of the same component in $rR(\mathcal{C}^+)$ as vertices representing a cycle containing a spare edge.

The proof is completed by noting that cycles in $\mathcal{C}$ containing spare edges cannot be selected 2-cycles. □

2.4.3 The sequence of reversals

We now explain how to find a sequence of reversals that sorts π in no more than $rb(\pi) - rc_2(\pi)/2$ reversals. We can assume by, Theorem 2.4.2, that we have a cycle

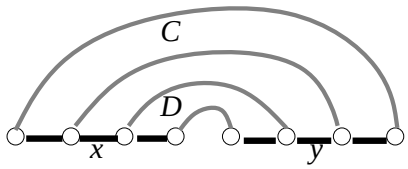

Figure 2.14: Part of a long ladder.

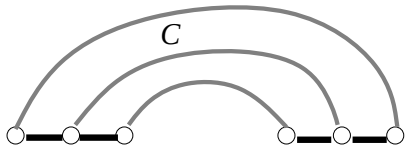

Figure 2.15: A short ladder.

decomposition $\mathcal{C}$ of $rG(\pi)$ that contains at least $\lceil y/2 \rceil$ 2-cycles that are part of ladders and z independent 2-cycles.

Theorem 2.4.3 *It is possible to sort π using no more than $rb(\pi) - rc_2(\pi)/2$ reversals.*

Proof Suppose that k of the selected 2-cycles in $\mathcal{C}$ are part of oriented components of $rR(\mathcal{C}^+)$. Then, by Proposition 2.3.2, there is a sequence of reversals that eliminates all the oriented components and includes at least k 2-moves and no 0-moves.

Suppose that there are u unoriented components in $rR(\mathcal{C}^+)$ that contain vertices representing cycles that are not selected 2-cycles, and that together these components include vertices representing l of the selected 2-cycles in $\mathcal{C}$. Then, by Proposition 2.3.3, there is a sequence of reversals that eliminates these components and includes at least $l + u$ 2-moves and only u 0-moves.

Suppose that the remaining v unoriented components of $rR(\mathcal{C}^+)$, which, by Lemma 2.4.1, consist only of vertices representing independent selected 2-cycles, contain vertices representing m 2-cycles. Then, by Proposition 2.3.3, there is a sequence of reversals that eliminates these components and includes at least m 2-moves and only v 0-moves. Note that $v \leq \lfloor z/2 \rfloor$, since, by Lemma 2.3.3, every component of this kind represents at least two cycles.

The sequence of reversals found by the steps described will sort π using at least $k + l + m + u$ 2-moves and only $u + v$ 0-moves. So at most $rb(\pi) - k - l - m + v$ reversals are required to sort the permutation. But $k + l + m - v \geq \lceil y/2 \rceil + z - v$, since $k + l + m \geq \lceil y/2 \rceil + z$. Further $\lceil y/2 \rceil + z - v \geq \lceil y/2 \rceil + \lceil z/2 \rceil$, since $v \leq \lfloor z/2 \rfloor$. Now $|M| \geq rc_2(\pi)$ and $y + z = |M|$, so $\lceil y/2 \rceil + \lceil z/2 \rceil \geq rc_2(\pi)/2$. This establishes the theorem. □

algorithm approximation(π: Permutation) **is**
begin
  construct the matching graph $rF(\pi)$;
  find a matching M for this graph;
  construct $\mathcal{C}$ using this matching;
  construct the reversal graph $rR(\mathcal{C}^+)$;
  find an elimination sequence for $rR(\mathcal{C}^+)$;
end approximation;

Figure 2.16: The 3/2-approximation algorithm

Hence an algorithm that finds the elimination sequence described above is a 3/2-approximation algorithm for sorting by reversals.

2.4.4 The algorithm

The approximation algorithm that has been described in the previous sections is outlined in Figure 2.16. The matching graph can be constructed in $O(n)$ time. A maximum cardinality matching for any graph can be found in $O(|V||E|^{\frac{1}{2}})$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph. The matching graph contains at most n edges and n vertices so, even without exploiting the special structure of the matching graph, the matching M can be found in $O(n^{\frac{3}{2}})$ time. The cycle decomposition can then be found in $O(n)$ time. Constructing the reversal graph can be performed in $O(n^2)$ time since there are at most $O(n^2)$ edges. The elimination sequence for components of $R(C)$ can be found as shown in Figure 2.17. Clearly finding the elimination sequence in this way can be achieved in $O(n^4)$ time. So the overall time-complexity of the algorithm is $O(n^4)$.

Note that by using methods described by Kaplan, Shamir and Tarjan [KST97] for the related problem of sorting signed permutations by reversals, it is possible to find the elimination sequence more efficiently. In fact, using their method of finding the elimination sequence requires only $O(n^2)$ time and so the overall time-complexity of the algorithm can be reduced to $O(n^2)$.

2.4.5 Conclusion

The approximation algorithm we have described has an approximation ratio of 3/2. In fact the algorithm does perform that badly for some permutations. For instance if $\pi = [3\ 4\ 1\ 2]$ then the algorithm requires 3 reversals, when 2 is the minimum possible. We define a sequence of permutations, of increasing length: $\pi_1 = \pi$, $\pi_{i+1} = \pi_i +\!\!+ [6i-1\ 6i\ 6i+3\ 6i+4\ 6i+1\ 6i+2]$, where $i \geq 1$, and $+\!\!+$ represents concatenation.

```
algorithm find an elimination sequence for rR(C) is
begin
  for each unoriented component loop
    apply a reversal represented by a blue vertex;
  end loop;
  while π ≠ ι loop
    r:= first red vertex;
    found:= false;
    while not found loop
      generate rR(C · ρ(r));
      count unoriented components in rR(C · ρ(r));
      if there are no unoriented components then
        π:= π · ρ(r);
        C:= C · ρ(r);
        found:= true;
      else
        r:= next red vertex;
      end if;
    end loop;
  end loop;
end find an elimination sequence for rR(C);
```

Figure 2.17: Finding an elimination sequence for $rR(\mathcal{C})$.

All these permutations require 3/2 times the minimum number of reversals to be sorted by our algorithm. Hence the algorithm achieves this worst bound asymptotically too.

On the other hand there are permutations for which the lower bound of Theorem 2.4.1 is exactly 2/3 of the actual reversal distance. For instance if $\pi = [5\ 6\ 3\ 4\ 1\ 2]$ then $rd(\pi) = 3$, whereas $rb(\pi) = 4$ and $rc_2(\pi) = 2$, so that the lower bound has value 2 in this case. In a similar way to the above, we can define a sequence of permutations which all have this property. Of course the approximation algorithm must sort all permutations like this optimally in order to achieve the approximation bound. Therefore unless we improve the lower bound we will not be able to improve the approximation ratio.

A strange property of our 3/2-approximation algorithm is that when we are constructing the cycle decomposition $\mathcal{C}$, any cycle decomposition of those edges that are not part of the selected 2-cycles will suffice. In general, the larger the cycle decomposition we take of these edges, the better the solution we will find.

Similarly, a 0-reversal is used in the elimination sequence of every unoriented component of $rR(\mathcal{C})$. However, in general, by using reversals that act on black edges of two different cycles it is not necessary to use so many 0-reversals.

Frings [Fri98] has implemented the 3/2-approximation algorithm. He has tested the algorithm with permutations of length 10, 20 and 40, that were generated by applying k reversals to the identity permutation. In comparison with Kececioglu and Sankoff's 2-approximation algorithm he discovered that 3/2-approximation algorithm found 'significantly' shorter sequences of reversals on average. For example, on permutations with $n = 20$ and $k = 7$ the 3/2-approximation algorithm found an optimal length sequence of reversals nearly 80% of the time, whereas the 2-approximation algorithm found an optimal length sequence less than 60% of the time. However, there were some permutations for which 2-approximation found shorter sequences of reversals than the 3/2-approximation algorithm. For example the 2-approximation algorithm uses only two reversals to sort $\pi = [3\ 4\ 1\ 2]$, but the 3/2-approximation algorithm uses three reversals.

2.5 An easy case of sorting by reversals

Since the identity permutation has no breakpoints, and a reversal can reduce the number of breakpoints by at most two, an immediate lower bound for the reversal distance is

$$rd(\pi) \geq \frac{rb(\pi)}{2}. \tag{2.2}$$

Let us call a permutation π *reversal-tight* if (2.2) is satisfied with equality. Kececioglu and Sankoff [KS95] made the following conjecture.

Conjecture 2.5.1 *The problem of determining whether a given permutation is reversal-tight is NP-complete.*

In this section we describe a polynomial-time algorithm to determine whether a permutation is reversal-tight, and thereby disprove the Kececioglu/Sankoff conjecture. In fact, the same result has been obtained independently by Tran [Tra97].

2.5.1 2-reversals, exposed and hidden

A permutation π has an *exposed 2-reversal* $\rho = \rho(i,j)$ $(1 \leq i,j \leq n+1, i < j-1)$ if and only if there are breakpoints at positions i and j, $|\pi(i-1) - \pi(j-1)| = 1$, and $|\pi(i) - \pi(j)| = 1$. It is clear that application the reversal ρ on π will reduce the number of breakpoints by 2 if and only if ρ is an exposed 2-reversal. (Exposed 2-reversals correspond to oriented 2-cycles in $rG(\pi)$.)

A permutation π has a *hidden 2-reversal* $\rho = \rho(i,j)$ $(1 \leq i,j \leq n+1, i < j-1)$ if and only if there are breakpoints at positions i and j, $|\pi(i-1) - \pi(j)| = 1$ and $|\pi(i) - \pi(j-1)| = 1$. Application of a hidden 2-reversal does not reduce the number of breakpoints by 2. However, a hidden 2-reversal may be transformed to an exposed 2-reversal by the application of one or more other reversals, depending (in a precise way to be elaborated upon below) on the overlapping pattern of the reversals. Hidden 2-reversals correspond to unoriented 2-cycles in $rG(\pi)$. In what follows, we use the term *2-reversal* to mean either an exposed or a hidden 2-reversal.

Let $\mathcal{R}(\pi)$ be the set of exposed and hidden 2-reversals in π. At any stage during the process of sorting π by reversals, application of an exposed 2-reversal cannot create any new exposed or hidden 2-reversals — it will destroy any 2-reversal that shares an end-point with it, and it may flip some exposed 2-reversals so that they become hidden, and vice versa. So the lower bound in (2.2) can be achieved only if $\mathcal{R}(\pi)$ contains an appropriate subset $\mathcal{R}'(\pi)$ of exactly $rb(\pi)/2$ 2-reversals. Here, 'appropriate' means:

> (a) $\mathcal{R}'(\pi)$ contains exactly one exposed or hidden 2-reversal with an end-point at breakpoint position i, for each breakpoint position i in π; and
> (b) the 2-reversals in $\mathcal{R}'(\pi)$ can be ordered, say $\rho_1, \rho_2, \ldots, \rho_r$ $(r = rb(\pi)/2)$ so that, for each j $(1 \leq j \leq r)$, ρ_j is exposed after $\rho_1, \ldots, \rho_{j-1}$ have been applied (in that order).

(Note that, when a reversal is applied, the 'parameters', or end positions, of any overlapping reversal will be changed, but we continue to think of it as essentially the same reversal.)

Each of conditions (a) and (b) above can be expressed in terms of a (different) graph model. For condition (a), we use the matching graph $rF(\pi)$ of Section 2.4.2, that has a vertex for each breakpoint i in π, with vertices i and j $(i < j)$ adjacent if and only if $\rho(i,j)$ is a 2-reversal in $\mathcal{R}(\pi)$. (Note that in Section 2.4.2, $rF(\pi)$ was defined in terms of cycles of length 2 in $rG(\pi)$ instead of 2-reversals. However, both definitions give rise to the same graph.) Then condition (a) is clearly satisfied if and only if the graph $rF(\pi)$ has a perfect matching. So this is a necessary condition for π

to be reversal-tight (and can easily be checked in polynomial time). That it is not a sufficient condition may be seen from the second of the following illustrative examples.

Example

$$\pi_1 = [0\ 9\ 6\ 4\ 2\ 8\ 1\ 7\ 3\ 5\ 10].$$

π_1 has 10 breakpoints, and has two exposed 2-reversals — $\rho_1 = \rho(3, 8)$ and $\rho_2 = \rho(5, 7)$, and four hidden 2-reversals — $\rho_3 = \rho(1, 6)$, $\rho_4 = \rho(2, 10)$, $\rho_5 = \rho(3, 9)$, and $\rho_6 = \rho(4, 9)$. ρ_1, ρ_2, ρ_3, ρ_4, and ρ_6 form a perfect matching in the graph $rF(\pi_1)$. The reversals may be applied in the order ρ_2, ρ_3, ρ_4, ρ_1, ρ_6, so the reversal distance of π_1 is 5, and π_1 is reversal-tight. □

Example

$$\pi_2 = [0\ 5\ 3\ 4\ 1\ 2\ 6\ 8\ 7\ 9].$$

π_2 has 6 breakpoints, and has one exposed reversal — $\rho_1 = \rho(7, 9)$, and two hidden 2-reversals — $\rho_2 = \rho(1, 4)$ and $\rho_3 = \rho(2, 6)$, and these 2-reversals form a perfect matching in the graph $rF(\pi)$. However, application of ρ_1 leaves no exposed 2-reversal, so π_2 is not reversal-tight. □

From now on we assume that the graph $rF(\pi)$ has a perfect matching, since if it does not, we can be sure that the permutation π is not reversal-tight. To investigate condition (b) we use reversal graphs.

Let M be a perfect matching for $rF(\pi)$. This perfect matching defines a cycle decomposition $\mathcal{C}_M$ of $rG(\pi)$, in which all the cycles are 2-cycles. Our disproof of the Kececioglu/Sankoff conjecture is based on the following theorem.

Theorem 2.5.1 *A permutation π is reversal-tight if and only if there is some perfect matching M in $rF(\pi)$ such that every component of $rR(\mathcal{C}_M^+)$ is oriented.*

Proof If M is a perfect matching in $rF(\pi)$ such that every component of $rR(\mathcal{C}_M^+)$ is oriented, then the elimination sequence found by using Proposition 2.3.2 on each component sorts π such that every reversal is a 2-move. Further, each reversal must remove 2-breakpoints because $\mathcal{C}_M$ contains only 2-cycles. So π is reversal tight.

If π is reversal tight, then there must be a sequence of 2-reversals that sorts π. Each 2-reversal corresponds to a 2-cycle in $rG(\pi)$. So $rF(\pi)$ has a perfect matching. Further each component of $rR(\mathcal{C}_M^+)$ is oriented, because the sequence of 2-reversals that sorts π corresponds to an elimination sequence of $rR(\mathcal{C}_M^+)$, and none of the reversals are 0-moves. So each reversal corresponds to applying the reversal represented by a red vertex of the reversal graph. If there was an unoriented component in $rR(\mathcal{C}_M^+)$ then we would not be able to sort π. □

2.5.2 Kernels

Theorem 2.5.1 on its own does not give a polynomial-time algorithm to check whether a permutation is reversal-tight. Certainly, for a given permutation π we can check

in polynomial time whether the graph $rF(\pi)$ has a perfect matching M, and for any such perfect matching M we can check whether each component of $rR(\mathcal{C}_M)$ is oriented. But there may be many such matchings M — exponentially many in the worst case — and we can only conclude that π is not reversal tight if $rR(\mathcal{C}_M)$ has an unoriented component for every such matching M. We now investigate how multiple matchings can arise, and how such situations can be handled.

Let π be a permutation and consider the graph $rF(\pi)$. Recall that we are assuming that the graph $rF(\pi)$ has a perfect matching. In forming a perfect matching in this graph, any vertex of degree 1 must be matched with the unique vertex to which it is adjacent. We may therefore iteratively pair off vertices until we reach a subgraph of $rF(\pi)$ in which every vertex has degree ≥ 2. Furthermore, if this subgraph has a *bridge*, i.e., an edge whose removal disconnects the graph, then just by considering the parity of the two resulting components we can decide whether that edge is forced into or out of every perfect matching. Dealing with every bridge in this way leads us to a subgraph of $rF(\pi)$ — say $rF^*(\pi)$ — in which every vertex has degree 2 or 3, and which contains no bridges. Let us call each connected component of $rF^*(\pi)$ a *kernel* in $rF(\pi)$. It turns out that kernels have a very special structure, as will be revealed in the following sequence of lemmas. (Of course, since we are assuming that $rF(\pi)$ has a perfect matching, each kernel necessarily has an even number of vertices.)

Vertices of $rF(\pi)$ are associated with black edges of $rG(\pi)$ which are in-turn associated with breakpoints of π. We shall blur the distinction between vertices of $rF(\pi)$ and breakpoints of π. So we say i is a breakpoint of a kernel K, if K contains a vertex that is associated with breakpoint i of π. We use notation v_i to denote the vertex of K that is associated with the breakpoint of π at position i.

We need some additional terminology. Suppose that permutation π has a breakpoint i, and this breakpoint is represented by a vertex, v say, in the graph $rF(\pi)$. We call $\pi(i-1)$ and $\pi(i)$ the *components* of vertex v. If $\{v, w\}$ is an edge in a kernel K then this edge is an $\{\alpha, \alpha+1\}$-*connection* if α is a component of v and $\alpha+1$ is a component of w, or vice-versa.

Call an element α of π *one-sided* with respect to kernel K if there is a breakpoint of K immediately to one side of α but not to the other. An element of π is *two-sided* with respect to K if it has breakpoints of K on both sides.

Suppose $\{v, w\}$ is an edge of a kernel K, and that v and w represent, respectively, breakpoints in positions i and j of the permutation, with $i < j$. Then v is called a *left* breakpoint and w a *right* breakpoint with respect to K. (Note that later we show that a breakpoint cannot be both a left and a right breakpoint with respect to K.) For convenience, in what follows, we use α' and α^* to represent $\alpha+1$ and $\alpha-1$, not necessarily respectively, for any element α of permutation π. If α' represents $\alpha+1$ then α'' represents $\alpha+2$, and so on.

Lemma 2.5.1 *For all permutations π, if a kernel K of the graph $rF(\pi)$ contains an $\{\alpha, \alpha+1\}$-connection, then K contains at least two $\{\alpha, \alpha+1\}$-connections.*

Proof Suppose that the end-vertices of the given $\{\alpha, \alpha+1\}$-connection are v and w. Without loss of generality, we may assume that α and $\alpha+1$ are the smaller components of v and w respectively. If there is no $\{\alpha, \alpha-1\}$-connection in K, then since v has degree ≥ 2, there must be a further $\{\alpha, \alpha+1\}$-connection incident on v. Likewise, if there is no $\{\alpha+1, \alpha+2\}$-connection, there must be a further $\{\alpha, \alpha+1\}$-connection incident on w. Otherwise, let x be any vertex of K whose smaller component is $< \alpha$, and let y be any vertex of K whose smaller component is $> \alpha$. Then, since $\{v, w\}$ cannot be a bridge, there must be a path in K from x to y that does not use the edge $\{v, w\}$. But if the smaller components of the vertices in this path are listed in order, they must form a sequence with initial value $< \alpha$, final value $> \alpha$, and each pair of successive values differing by 1. It follows that the sequence must contain α followed by $\alpha+1$, and the corresponding edge of K is a second $\{\alpha, \alpha+1\}$-connection. □

Lemma 2.5.2 *Let K be a kernel of $rF(\pi)$. If the leftmost occurrence of a right breakpoint (with respect to K) is in position i of π, then position $i-1$ is not a breakpoint of K.*

Proof We can assume that $\pi(i-1) = \alpha$ and $\pi(i) = \beta$ are the components of the leftmost right breakpoint, v say, of K. Since v is a right breakpoint, there must be a breakpoint with components α' and β', say, lying to the left of v in π. Since there is no right breakpoint in position $i-1$, a second $\{\alpha, \alpha'\}$-connection can only arise if α' is sandwiched between β' and β^*, so that all the $\{\alpha, \alpha'\}$-connections involve the breakpoint v. If there is a breakpoint of K, u say, in position $i-1$, it must be a left breakpoint, so α^* must lie to the right of α, and any $\{\alpha, \alpha^*\}$-connections must involve the breakpoint u, since they cannot involve v. But then there cannot be a path in K from a breakpoint having α' as a component to one having α^* as a component, so u and v cannot both be part of K. Hence, position $i-1$ is not a breakpoint of K. □

Lemma 2.5.3 *If α_1 and α_2 are the smaller (respectively larger) components of two vertices in a kernel K, then each value between α_1 and α_2 is also the smaller (respectively larger) component of a vertex in K.*

Proof Let α_1 and α_2 be the smaller components of vertices v and w respectively. Since the kernel K is connected, there is a path from v to w, and the successive vertices on this path have smaller components differing by exactly one. Hence the path contains vertices with smaller components covering all values between α_1 and α_2. The argument for larger components is similar. □

Lemma 2.5.4 *For a permutation π of length n, the breakpoints of a kernel are in positions $i, i+1, \ldots, i+m-1$ and $j, j+1, \ldots, j+m-1$ for some i, j and m with $1 \leq i$, $i+m < j$, $j \leq n-m+2$. Further, any edge of K connects breakpoints in positions p and q for some p, q with $i \leq p \leq i+m-1$ and $j \leq q \leq j+m-1$.*

Proof Suppose α is one-sided with respect to K. If there is an $\{\alpha, \alpha+1\}$-connection in K, then there must be at least two such connections, by Lemma 2.5.1. Hence there can be no $\{\alpha, \alpha-1\}$-connection because there would have to be two such connections, and this would imply a vertex of degree 4 in K which is impossible. Similarly, if there is an $\{\alpha, \alpha-1\}$-connection there can be no $\{\alpha, \alpha+1\}$-connection. So, by Lemma 2.5.3, the only candidates for one-sided elements are

(a) the minimum of the smaller components of breakpoints in K;
(b) the maximum of the smaller components of breakpoints in K;
(c) the minimum of the larger components of breakpoints in K;
(d) the maximum of the larger components of breakpoints in K.

Since there can be at most four one-sided elements it follows that the breakpoints of K lie in at most two intervals of the permutation. It is a consequence of Lemma 2.5.2 that the breakpoints must lie in two disjoint intervals $[i, i+l-1]$ and $[j, j+m-1]$ with $i+l<j$, and that all breakpoints in the first interval are left breakpoints and all those in the second interval are right breakpoints. Finally, since we are assuming that $rF(\pi)$ has a perfect matching, it follows that K does also, and since all edges of K connect breakpoints in the two separate intervals, it follows that $l=m$. □

If all the vertices in a kernel have degree 2, then the kernel is just a cycle (and there are only two possible matchings of the vertices in this kernel). We now seek to characterise kernels containing at least one vertex of degree 3.

Lemma 2.5.5 *If a breakpoint (vertex) v has degree 3 in a kernel K of a permutation π, then it is adjacent to three successive breakpoints x, y and z of π, and the middle one of these, y, must be adjacent to the breakpoints u and w on either side of v.*

Proof Let the components of v be α and β, with α to the left of β, and suppose, without loss of generality, that v is a left breakpoint. The fact that v is adjacent to three successive breakpoints is immediate. There are then two cases to consider:

(a) x, y and z occur in the sequence $\alpha'\beta'\alpha^*\beta^*$;
(b) x, y and z occur in the sequence $\beta'\alpha'\beta^*\alpha^*$.

We prove the result only for case (a), the proof for case (b) being entirely analogous.

So the permutation takes the form

$$\ldots\ldots\ldots\alpha\beta\ldots\ldots\ldots\alpha'\beta'\alpha^*\beta^*\ldots\ldots\ldots$$

To get a second $\{\alpha, \alpha'\}$-connection, there must be a breakpoint of K immediately to the left of α, and to enable that breakpoint to have degree 2 in K, the value to the left of α must be β'' or β^{**}. Similarly, the value to the right of β must be α'' or α^{**}. So this leads to four subcases, as follows:

case (i) $\ldots\ldots\ldots\beta''\alpha\beta\alpha^{**}\ldots\ldots\ldots\alpha'\beta'\alpha^*\beta^*\ldots\ldots\ldots$
case (ii) $\ldots\ldots\ldots\beta^{**}\alpha\beta\alpha''\ldots\ldots\ldots\alpha'\beta'\alpha^*\beta^*\ldots\ldots\ldots$
case (iii) $\ldots\ldots\ldots\beta^{**}\alpha\beta\alpha^{**}\ldots\ldots\ldots\alpha'\beta'\alpha^*\beta^*\ldots\ldots\ldots$
case (iv) $\ldots\ldots\ldots\beta''\alpha\beta\alpha''\ldots\ldots\ldots\alpha'\beta'\alpha^*\beta^*\ldots\ldots\ldots$

We will show that cases (ii), (iii) and (iv) are impossible. The conclusion of the lemma follows immediately in case (i).

case (ii) To enable breakpoint (β, α'') to have degree 2, β^* must be followed by α'''. To enable breakpoint (β^*, α''') to have degree 2, β^{**} must be preceded by α''''. To enable breakpoint (β', α^*) to have degree 2, α^{**} must be a component of some breakpoint in K. But then it is easy to verify that two $\{\alpha^*, \alpha^{**}\}$-connections cannot exist, showing that this case cannot arise.

case (iii) To enable breakpoint (β^{**}, α) to have degree 2, β^{***} must precede α'. To enable breakpoint (α', β') to have degree 2, (α'', β'') (or (β'', α'')) must be a breakpoint in K.

Now, since K contains no bridge there must be a path in K from breakpoint (α', β') to (α'', β'') that does not use the edge joining them, and the first step in this path must lead to (α, β), the only other available edge from (α', β'). This path must pass through, at some point, the other breakpoint with component β' — i.e., (β', α^*) — so this may as well be the next step on the path. It must also pass through the other breakpoint with component α — i.e., (β^{**}, α) — but to reach that vertex from (β', α^*), the path must pass through the other breakpoint with component β, namely (β, α^{**}). But then both breakpoints having β as a component would have already appeared in the path, and so it would impossible to get from (β^{**}, α) to (α'', β''). Hence this case cannot arise.

case (iv) To enable breakpoint (β, α'') to have degree 2, β^* must be followed by α'''. To enable breakpoint (α^*, β^*) to have degree 2, there must be a breakpoint with components α^{**} and β^{**}. This case can now be ruled out by an argument analogous to that used for case (iii), considering the possibility of an alternative path from (α^*, β^*) to $(\alpha^{**}, \beta^{**})$. □

Lemma 2.5.6 *If any vertex in a kernel has degree 3, then for some $i < j$, the edges in the kernel are either*

(i) $\{v_i, v_j\}$, $\{v_i, v_{j+1}\}$; $\{v_{i+m-1}, v_{j+m-2}\}$, $\{v_{i+m-1}, v_{j+m-1}\}$; *and* $\{v_k, v_{k+j-i-1}\}$, $\{v_k, v_{k+j-i}\}$, $\{v_k, v_{k+j-i+1}\}$ *for* $i+1 \leq k \leq i+m-2$; *or*
(ii) $\{v_i, v_{j+m-1}\}$, $\{v_i, v_{j+m-2}\}$; $\{v_{i+m-1}, v_{j+1}\}$, $\{v_{i+m-1}, v_j\}$; *and* $\{v_k, v_{i+j+m-k-2}\}$, $\{v_k, v_{i+j+m-k-1}\}$, $\{v_k, v_{i+j+m-k}\}$ *for* $i+1 \leq k \leq i+m-2$.

Proof Suppose v_k has degree 3 for some k ($i \leq k \leq i+m-1$). Then v_k is adjacent in K to v_{l-1}, v_l, and v_{l+1} for some l ($j < l < j+m-1$). By Lemma 2.5.5, it follows

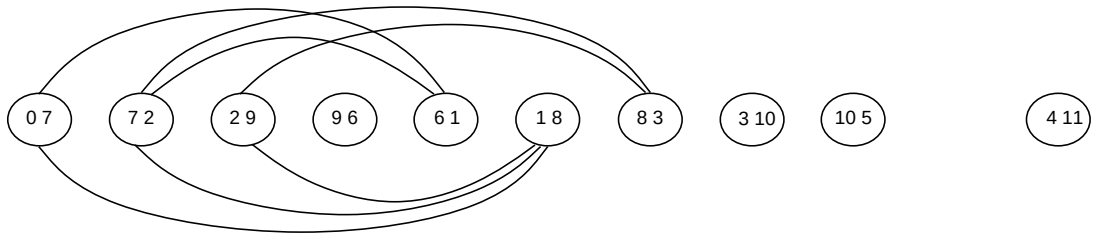

Figure 2.18: $rF^*(\pi_3)$

that $i < k < i + m - 1$ and that positions $k-1, k, k+1, k+2$ and $l-1, l, l+1, l+2$ are occupied in one of the two possible ways shown:

case (a) $\ldots\ldots\ldots \beta''\alpha\beta\alpha^{**} \ldots\ldots\ldots \alpha'\beta'\alpha^*\beta^* \ldots\ldots\ldots$
case (b) $\ldots\ldots\ldots \beta^{**}\alpha\beta\alpha'' \ldots\ldots\ldots \beta'\alpha'\beta^*\alpha^* \ldots\ldots\ldots..$

If $m = 3$ then we see that these two possibilities correspond to cases (i) and (ii) in the statement of the lemma. Otherwise, one of v_{k+1} or v_{k-1} has degree 3. Applying Lemma 2.5.5 leads to configurations (i) and (ii) with $m = 4$. The argument may be continued inductively to complete the proof. □

Let us call kernels of the form described in Lemma 2.5.6 parts (i) and (ii) *overlapped* and *nested* kernels, respectively. A kernel in which every vertex has degree 2 will be called a *cyclic* kernel.

At this point it may be helpful to include some illustrative examples.

Example

$$\pi_3 = [0\ 7\ 2\ 9\ 6\ 1\ 8\ 3\ 10\ 5\ 4\ 11].$$

The permutation π_3 contains an overlapped kernel with 6 breakpoints (see Figure 2.18). It is a split kernel — see the definition below, split by the breakpoint $(9, 6)$, since it is forced to be matched in an oriented cycle with the breakpoint $(10, 5)$. □

Example

$$\pi_4 = [0\ 11\ 2\ 9\ 4\ 7\ 6\ 5\ 8\ 3\ 10\ 1\ 12].$$

The permutation π_4 contains a nested kernel that includes all 10 breakpoints (see Figure 2.19). It is a whole kernel — see the definition below. □

Example

$$\pi_5 = [0\ 8\ 4\ 10\ 2\ 6\ 11\ 3\ 7\ 1\ 9\ 5\ 12].$$

The permutation π_5 contains a cyclic kernel with 10 breakpoints (see Figure 2.20). It is a split kernel, split by the single breakpoint $(6, 11)$. □

Now that we have characterised the structure of kernels, we have to consider how to deal with them when they arise. Recall that a permutation is reversal-tight if and

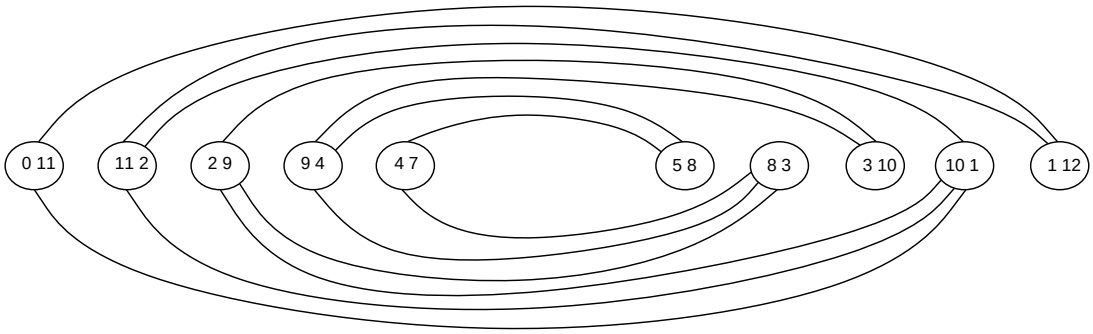

Figure 2.19: $rF^*(\pi_4)$

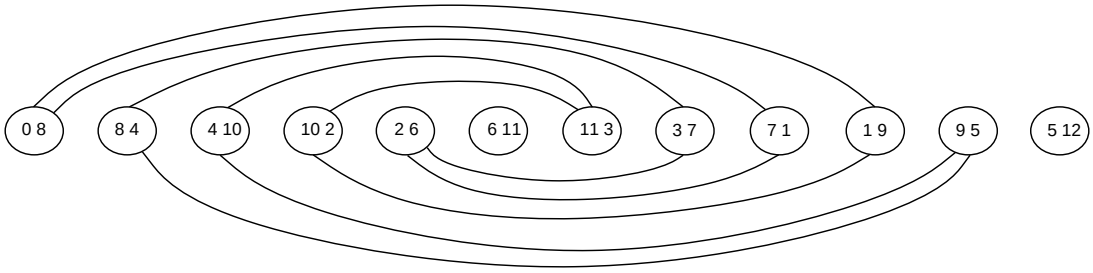

Figure 2.20: $rF^*(\pi_5)$

only if there is a perfect matching M in the graph $rF(\pi)$ such that every component of $rR(\mathcal{C}_M)$ is oriented. For a given kernel K, and a perfect matching M in $rF(\pi)$, denote by K_M the subgraph of $rR(\mathcal{C}_M)$ comprising of those components that contain vertices arising from 2-cycles (of $rG(\pi)$) represented by edges in K and M.

According to Lemma 2.5.4, a kernel K has its left and right breakpoints in two disjoint intervals of the permutation. Suppose that there is a 2-reversal with exactly one of its endpoints lying between these two intervals, and that this 2-reversal is either part of another kernel or is forced, during the construction of $rF^*(\pi)$ from $rF(\pi)$, to be part of any perfect matching. In this case, K will be called a *split* kernel. If there is no such 2-reversal the kernel will be called *whole*. If $rF(\pi)$ has a perfect matching M such that every component of K_M is oriented, then K will be called a *reversal-tight* kernel.

Lemma 2.5.7 *Let π be a permutation with a kernel K, and suppose that the graph $rF(\pi)$ has a perfect matching.*

> *(i) If K is a whole kernel which is overlapped or nested then it is reversal-tight.*
> *(ii) If K is a whole kernel which is cyclic, then there is a polynomial-time algorithm to check whether it is reversal-tight.*
> *(iii) If K is a split kernel then it is reversal-tight.*

Proof We prove the three cases separately:

(i) Let K be an overlapped kernel, with breakpoints v_i, ..., v_{i+m-1} and v_j, ..., v_{j+m-1} as in the statement of Lemma 2.5.6. If we take a perfect matching in K that includes the edges $\{v_i, v_{j+1}\}$ and $\{v_{i+1}, v_j\}$, and it is clear that such a matching exists, then both of these edges represent exposed 2-reversals, and as vertices in K_M, both are represented by vertices that are adjacent to all of the vertices arising from K. Hence the single component of K_M is oriented.

On the other hand, if K is a nested kernel, we may take the perfect matching that pairs v_i with v_{j+m-1}, v_{i+m-1} with v_j, and, for $k = 1, \ldots, m-2$, v_{i+k} with $v_{j+m-1-k}$. Every 2-reversal in this perfect matching is exposed, so that every component of K_M is oriented.

(ii) In this case, since the kernel is whole, the components of K_M contain reversals only on breakpoints of K. There are only two possible perfect matchings in K, and it is a simple matter to check, in polynomial time, whether for either one, all the components of K_M are oriented.

(iii) For any perfect matching M of $rF(\pi)$, K_M will be a connected component, since the edge that splits the kernel (or any of the chosen edges of a splitting kernel) will be represented in K_M by a vertex that is adjacent to all vertices arising from K. So in this case, it suffices to determine whether K contains a perfect matching that includes at least one exposed 2-reversal.

If K is cyclic (necessarily with an even number of vertices) then there are just two perfect matchings in K. The leftmost breakpoint in K is an end-point of two 2-reversals, one of which is exposed and one of which is hidden. So we may choose the matching that includes this particular exposed 2-reversal.

If K is overlapped or nested then the perfect matchings described in the proof of (i) each include at least one exposed 2-reversal, and this is all that is required. □

Note that, by the definition of a kernel, the choices we make for the matching restricted to each kernel are independent. So we can construct a global matching from the matchings for each kernel. Therefore, it follows from Lemma 2.5.7 that, if π is a permutation for which the graph $rF(\pi)$ contains a perfect matching, then in order to determine whether π is reversal-tight, the following steps suffice:

1. identify the kernels;
2. for any whole cyclic kernel K, establish whether at least one of the two possible matchings gives components of $rR(\mathcal{C}_M^+)$ that are all oriented;
3. establish whether all components of $rR(\mathcal{C}_M^+)$ that do not involve kernels are all oriented.

We have therefore proved the following theorem:

Theorem 2.5.2 *For a given permutation π, it can be established in polynomial time whether or not π is reversal-tight.*

This result has been obtained independently by Tran [Tra97]. He proves the first half (Theorem 2.5.1) of this result in roughly the same way as we do. However, he proves the second half in quite a different way from us. In effect, Tran shows that if π is reversal tight then all components of $rR(\mathcal{C}_M^+)$ are oriented, where M is a maximum weight perfect matching of $rF(\pi)$ with edges given weight $+1$ if they represent an exposed 2-reversal and weight -1 otherwise. So his proof of this part is simpler than our proof because it does not need to describe the complicated structure of kernels.

2.6 Conclusion and open problems

The 3/2-approximation algorithm of Section 2.4 has the best known approximation guarantee for sorting by reversals. It is an open problem to find a polynomial-time algorithm with a better approximation guarantee.

The question of whether sorting by reversals is MAXSNP-complete [Pap94], and the related question of whether there is a polynomial time approximation scheme for the problem, are both open.

Chapter 3

Sorting by Transpositions

3.1 Introduction

In this chapter we study the problem of sorting by transpositions. The computational complexity of this problem remains open, but we make a number of contributions to the better understanding of this problem. The significant new results of this chapter are as follows:

- we analyse the structure of cycles in the so-called transposition cycle graph, leading to a better understanding of how these cycles affect the problem (Section 3.3). This includes defining cycles called *knots*, and describing a method for dealing with such cycles in an efficient manner.
- we present a new 3/2-approximation algorithm for sorting by transpositions that is somewhat simpler than that presented by Bafna and Pevzner (Section 3.4).
- we present an improved lower bound for sorting by transpositions (Section 3.5).
- we present empirical evidence that in practice we can solve most instances of sorting by transpositions (Section 3.6).

Some other interesting new results in this chapter include:

- a proof that it is never necessary to break apart any strips in a minimal length sequence of transpositions that sorts a permutation (Section 3.2.2).
- a description of a family of permutations whose transposition distance can be arbitrarily far from Bafna and Pevzner's lower bound for transposition distance (Section 3.5.2).
- a proof that the transposition distance of the reverse permutation R_n is $\lfloor n/2 \rfloor + 1$ (Section 3.5.3).

We begin the chapter with a section containing fundamental definitions, lemmas and theorems used to study the problem of sorting by transpositions.

3.2 Fundamental definitions and results

3.2.1 Definitions

Let π be a permutation of length n. The *transposition* $\tau = \tau(i, j, k)$ (where $1 \leq i < j < k \leq n + 1$) transforms π into $\pi \cdot \tau = [\pi(0) \ \ldots \ \pi(i-1) \ \pi(j) \ \ldots \ \pi(k-1) \ \pi(i) \ \ldots \ \pi(j-1) \ \pi(k) \ \ldots \ \pi(n+1)]$. The *transposition distance*, $td(\pi)$, between π and $\imath$ is the length of a shortest sequence of transpositions that transforms π into $\imath$. A sequence of transpositions that transforms π into $\imath$ is called a *sorting sequence*. *Sorting by transpositions* is the problem of finding a sorting sequence for π of length $td(\pi)$.

A *transposition breakpoint* is a position i in the permutation such that $1 \leq i \leq n+1$ and $\pi(i) - \pi(i-1) \neq 1$. The number of transposition breakpoints in a permutation π is represented by $tb(\pi)$. By contrast, a *transposition adjacency* is a position i in the permutation such that $1 \leq i \leq n+1$ and $\pi(i)-\pi(i-1) = 1$. A *strip* is a substring $\pi[i..j]$ of π $(i < j)$ such that i and $j+1$ are breakpoints and there are no breakpoints between these positions. Note that, for conciseness, in the rest of this chapter we may talk of breakpoints and adjacencies where, of course, we mean transposition breakpoints and transposition adjacencies.

Let $f(\pi)$ be some function on π, e.g., $tb(\pi)$. We are often interested in the change in value of $f(\pi)$ as a result of applying a transposition τ. We shall use $\Delta f(\pi, \tau)$ to denote the change. Formally, $\Delta f(\pi, \tau) = f(\pi \cdot \tau) - f(\pi)$.

It is easy to show that the value of $\Delta tb(\pi, \tau)$ is characterised by the following lemma.

Lemma 3.2.1 *For all permutations π and transpositions τ, $\Delta tb(\pi, \tau) \geq -3$.*

This lemma leads to the following lower bound for transpositions distance, since the identity permutation contains no breakpoints.

Theorem 3.2.1 *For all permutations π, $td(\pi) \geq \lceil tb(\pi)/3 \rceil$.*

3.2.2 Transposition equivalent permutations

A little thought suggests that it is unlikely that a shortest sorting sequence would ever have to break apart a strip, since that strip would later have to be put back together. In fact this intuition is correct, as we prove below.

Let r be the number of strips in π, excluding any initial strip beginning with 1 and any final strip ending with n. We define *the minimal permutation* $gl(\pi)$ to be the permutation of $\{1, \ \ldots, \ r\}$ formed from π by 'gluing' all the adjacencies together, i.e., replacing each strip by a single element. Note that if π begins with 1 then the strip at the start of π is removed completely in $gl(\pi)$ and, similarly, if π ends with n then the strip at the end of π is removed completely in $gl(\pi)$.

Example If $\pi = [4\ 5\ 6\ 3\ 1\ 2\ 7]$ then $gl(\pi) = [3\ 2\ 1]$. □

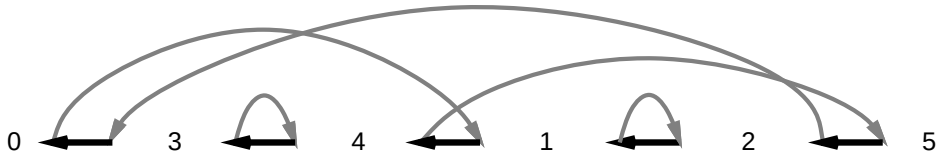

Figure 3.1: $tG(\pi)$, for $\pi = [3\ 4\ 1\ 2]$.

Theorem 3.2.2 *For every permutation π, $td(\pi) = td(gl(\pi))$.*

Proof Clearly $td(\pi) \leq td(gl(\pi))$, since any transposition on $gl(\pi)$ may be mimicked by a transposition on π.

We now show that $td(gl(\pi)) \leq td(\pi)$. Let π be a permutation of length n. We may assume that $gl(\pi)$ is length m, such that $m \leq n$. Form a string S_π of length n by inserting $n - m$ asterisks $(*)$ into $gl(\pi)$, such that $S_\pi(i)$ is an asterisk if i is an adjacency in π, or $\pi(j) = j$ for all $j \geq i$. For example, if $\pi = [1\ 3\ 4\ 5\ 7\ 8\ 2\ 6\ 9]$, then $gl(\pi) = [2\ 4\ 1\ 3]$, and $S_\pi = [*\ 2\ *\ *\ 4\ *\ 1\ 3\ *]$. Then any transposition on π can be applied to S_π, and ignoring asterisks this transposition can be applied to $gl(\pi)$. Note that if any transposition on S_π moves a block that consists only of asterisks then the corresponding transposition on $gl(\pi)$ does nothing, so can be ignored. A sequence of transpositions that sorts π will sort S_π (ignoring asterisks), and so a sequence of transpositions of at most the same length exists that sorts $gl(\pi)$. Hence $td(gl(\pi)) \leq td(\pi)$, and the theorem has been proved. □

We say that permutations π and ϕ are *transposition equivalent* if $gl(\pi) = gl(\phi)$.

As a consequence of Theorem 3.2.2 we can essentially restrict our attention to permutations with no adjacencies. A *strip free permutation* π is a permutation of length n such that π contains no adjacencies, $\pi(1) \neq 1$, and $\pi(n) \neq n$.

3.2.3 The transposition cycle graph

The *transposition cycle graph* of π, $tG(\pi)$, is a directed edge-coloured graph with vertex set $\{0, \ldots, n+1\}$, grey edge set $\{(i, i+1) :\ 0 \leq i \leq n\}$, and black edge set $\{(\pi(i), \pi(i-1)) :\ 1 \leq i \leq n+1\}$. Note that the edge (u, v) is directed from u to v. This graph was introduced by Bafna and Pevzner [BP95b].

Example Let π be the permutation $[3\ 4\ 1\ 2]$. Then $tG(\pi)$ is shown in Figure 3.1. Note that by convention $tG(\pi)$ is drawn without any circles representing the vertices. We also, by a convention to be explained later, draw $tG(\pi)$ so that the black edge $(\pi(i), \pi(i-1))$ points directly to the grey edge $(\pi(i-1), \pi(i-1)+1)$, and the grey edge $(i, i+1)$ points directly to the black edge $(i+1, \pi(\pi^{-1}(i+1)-1))$. □

The black edge $(\pi(i), \pi(i-1))$ is denoted by b_i. This edge, b_i, is also known as *the black edge at position i*. As shown in Figure 3.1, we draw $tG(\pi)$ so that the black edges occur in the order $b_1, \ldots, b_{n+1}$ from left to right. It is because of this convention that we define an edge b_i to be to the *left* of b_j if $i < j$. Similarly, we define b_i to be to the *right* of b_j if $i > j$. The *leftmost* edge of a set of black edges is the edge b_i from the set such that i is minimised. Similarly, the *rightmost* edge of a set of black edges is the edge b_i from the set such that i is maximised. A grey edge $(i, i+1)$ is said to be *directed to the right* if $\pi^{-1}(i) < \pi^{-1}(i+1)$. Similarly, a grey edge is said to be *directed to the left* if $\pi^{-1}(i) > \pi^{-1}(i+1)$. In Figure 3.1, the grey edge $(2, 3)$ is directed to the left, and all the other grey edges are directed to the right.

A path in $tG(\pi)$ is an *alternating path* if its edges alternate in colour. Similarly, a cycle is an *alternating cycle* if its edges alternate in colour.

At each vertex of $tG(\pi)$, each incoming edge can be paired with an outgoing edge of the alternative colour. This pairing of edges decomposes the graph into alternating cycles, and furthermore the decomposition is unique, since each vertex has at most one incoming edge of each colour, and one outgoing edge of each colour. Such *cycle decompositions* are obvious in our figures because of the conventions we use when drawing cycle graphs. The number of alternating cycles in the cycle decomposition of $tG(\pi)$ is denoted by $tc(\pi)$. The maximum value of $tc(\pi)$ is $n+1$, and is achieved only when π is the identity permutation. In the rest of this chapter it is assumed that all the cycles we consider are alternating cycles.

Let C be a cycle in $tG(\pi)$. The *length* of C, $l(C)$, is the number of black edges in C. A *k-cycle* is a cycle of length k. A *proper cycle* is a cycle that has length greater than one. A *long cycle* is a cycle that has length greater than two. If $l(C)$ is odd then the cycle is *odd*, and otherwise the cycle is *even*. The number of odd cycles in $tG(\pi)$ is denoted by $tc_{odd}(\pi)$. Similarly, $tc_{even}(\pi)$ denotes the number of even cycles in $tG(\pi)$.

Example If π is the permutation [3 4 1 2] then $tc(\pi) = tc_{odd}(\pi) = 3$ and $tc_{even}(\pi) = 0$, as illustrated in Figure 3.1. □

Bafna and Pevzner [BP95b] proved the following lemmas about the number of cycles and odd cycles in the cycle graph.

Lemma 3.2.2 *For all permutations π and transpositions τ, $\Delta tc(\pi, \tau) \in \{-2, 0, 2\}$.*

Lemma 3.2.3 *For all permutations π and transpositions τ, $\Delta tc_{odd}(\pi, \tau) \in \{-2, 0, 2\}$.*

A transposition τ is a *t-transposition* if $\Delta tc_{odd}(\pi, \tau) = t$. Lemma 3.2.3, combined with the fact that the identity permutation contains $n+1$ odd length cycles, allowed Bafna and Pevzner to prove the following lower bound for transposition distance.

Theorem 3.2.3 *For all permutations π, $td(\pi) \geq (n + 1 - tc_{odd}(\pi))/2$.*

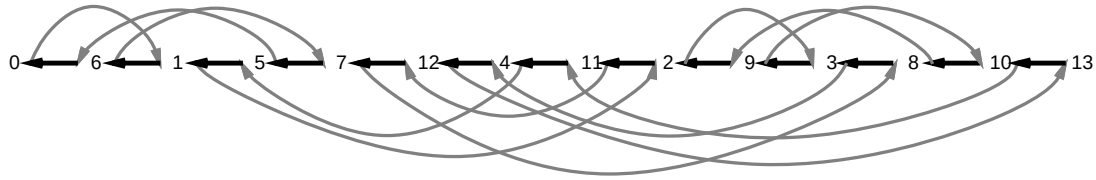

Figure 3.2: $tG(\pi)$, for $\pi = [6\ 1\ 5\ 7\ 12\ 4\ 11\ 2\ 9\ 3\ 8\ 10]$.

The positions of the leftmost black edge and the rightmost black edge of a cycle C are denoted by $C_{\min}$ and $C_{\max}$ respectively. More formally, these values can be defined as $C_{\min} = \min\{i : b_i \text{ is an edge in } C\}$, and $C_{\max} = \max\{i : b_i \text{ is an edge in } C\}$.

By convention, a cycle is described by listing the positions of the black edges of the cycle in the order they occur in the cycle, starting from position $C_{\max}$. A cycle C that visits black edges in the order $b_{i_1}, \ldots, b_{i_{l(C)}}$ is denoted by $(i_1, \ldots, i_{l(C)})$. (Note $i_1 = C_{\max}$.)

Example Let π be the permutation [6 1 5 7 12 4 11 2 9 3 8 10]. The cycle graph of this permutation contains three cycles: $(4, 1, 2)$, $(12, 9, 10)$, and $(13, 7, 3, 8, 5, 11, 6)$, as shown in Figure 3.2. □

The transposition $\tau = \tau(i, j, k)$ is said to *act on* black edges b_i, b_j and b_k of $tG(\pi)$, since the transformation from $tG(\pi)$ to $tG(\pi \cdot \tau)$ involves removing these three edges and introducing three new edges, whereas all the other grey and black edges of $tG(\pi)$ remain intact in $tG(\pi \cdot \tau)$.

Three black edges of $tG(\pi)$ form an *oriented triple* if τ, the transposition that acts on them, is such that $\Delta tc(\pi, \tau) = 2$. Three black edges of a cycle form an *unoriented triple* if they do not form an oriented triple.

Example In Figure 3.2, (b_1, b_2, b_4) is an oriented triple and (b_3, b_5, b_8) is an unoriented triple. □

An *oriented cycle* contains an oriented triple of black edges. A cycle that is not oriented is *unoriented*. Bafna and Pevzner noted the following theorem characterising unoriented cycles.

Theorem 3.2.4 *Cycle C is unoriented if and only if $(i_1, \ldots, i_{l(C)})$ is a decreasing sequence.*

When a transposition is applied to a permutation the cycle graph of the resulting permutation is the same as the original cycle graph except that three black edges have been removed, and three black edges have been introduced. It is likely that the positional labelling (b_i) of a black edge that was not removed or introduced will have changed as the result of the transposition. We often want a label for an edge that does

not change if the edge is not removed. We use the letters s, t, u, v, w, x, y and z as *positionally independent labels* of black edges. The position of such a black edge u in π is denoted by $p(\pi, u)$. For brevity we sometimes write a positionally independent label, where we should really write the position of the label. For example we may write $u < v$ in place of $p(\pi, u) < p(\pi, v)$.

3.2.4 Even length cycles

Bafna and Pevzner do not give much emphasis to the following results about even length cycles, but we will show them to be very useful in later sections. The first result is a lemma that can be derived easily from Lemma 3.2.2 and Lemma 3.2.3.

Lemma 3.2.4 *For all permutations π and transpositions τ, $\Delta tc_{even}(\pi, \tau) \in \{2, 0, -2\}$.*

This lemma leads naturally to the following corollary.

Corollary 3.2.1 *$tc_{even}(\pi)$ is an even number, for all permutations π.*

Proof This follows from the preceding lemma and the fact that the cycle graph of the identity permutation contains no even length cycles. □

This corollary allows us to prove the following useful lemma, that helps us determine if it is possible to apply a 2-transposition to a permutation.

Lemma 3.2.5 *For all permutations π, if $tG(\pi)$ contains an even cycle we may apply a 2-transposition on π.*

Proof By Corollary 3.2.1, if $tG(\pi)$ contains an even cycle then it must contain at least two even cycles. Let C and D be even cycles of $tG(\pi)$. Let x and y be black edges of C such that the alternating path connecting x to y (but excluding both edges) in $tG(\pi)$ contains an even number of black edges. Let z be any black edge of D. Then the transposition acting on black edges x, y and z is a transposition that increases the number of odd length cycles in the cycle graph by 2 (although it does not increase the total number of cycles in the cycle decomposition). □

3.3 Fully oriented cycles

In this section we describe how to determine if a 2-transposition exists on a permutation. In particular, we achieve this by characterising those cycles (the fully oriented cycles) that admit 2-transpositions that act on their edges. We also describe a method for efficiently dealing with cycles that are oriented, but do not admit 2-transpositions on their edges. We then compare our fully oriented cycles with Bafna and Pevzner's strongly oriented cycles. Finally we define cycles, super oriented cycles, that permit a sequence of 2-transpositions on their edges, and show a preliminary result towards characterising these cycles.

3.3.1 Fully oriented triples

Three black edges x, y and z of $tG(\pi)$ form a *fully oriented triple* (x, y, z) if τ, the transposition on those edges, is a 2-transposition, and x, y, and z are all part of the same cycle in $tG(\pi)$. A cycle that contains a fully oriented triple is said to be *fully oriented*.

Fully oriented triples are useful because of the following simple proposition.

Proposition 3.3.1 *Let τ be a 2-transposition that acts on black edges x, y, and z of $tG(\pi)$. Then either x, y and z are all part of the same cycle in $tG(\pi)$, or they are from two even length cycles in $tG(\pi)$*

Proof Edges x, y, and z are edges of one, two or three cycles in $tG(\pi)$. All possible cases are shown in Figure 3.3. Note that vertices are not shown in this figure because the figure represents only part of $tG(\pi)$ and so there may be many vertices between the black edges. In this figure dotted grey edges represent alternating paths in $tG(\pi)$ that start and finish with a grey edge.

A transposition can only be a 2-transposition if it is like that shown in Figure 3.3 (b), (c), (d) or (e). So x, y, and z must be part of one or two cycles. □

As a consequence of this proposition, a 2-transposition either acts on two even length cycles, or it acts on a fully oriented triple of edges. Now, if a 2-transposition acts on two even cycles, then we can show that the edges the transposition acts on must be arranged as described in the proof of Lemma 3.2.5. We characterise 2-transpositions on one cycle by characterising fully oriented triples and fully oriented cycles.

Given a cycle we can easily tell, using Theorem 3.2.4, whether the cycle is oriented, or unoriented. However an oriented cycle is not necessarily a fully oriented cycle. For example, if $\pi = [4\ 3\ 2\ 1]$ then the cycle $(5, 3, 1, 4, 2)$ in $tG(\pi)$ (Figure 3.4) is oriented, but not fully oriented. What conditions make an oriented cycle fully oriented? We answer this question in Section 3.3.3, but to help us find the answer we need a new notation for cycles, that we call the canonical labelling of a cycle.

3.3.2 Canonical labellings

With the cycle notation introduced in Section 3.2.3 we can tell if a cycle is oriented, but the notation is not very useful for anything else, in particular it is difficult to compare the structure of two cycles using this notation.

We now introduce a new cycle notation called the *canonical labelling* in which the structures of different cycles can be compared more easily. The first step towards obtaining this new notation for a cycle C is to (re)label its black edges 1, 2, ..., $l(C)$ so as to preserve the relative positional order of the edges. So edge $b_{C_{\min}}$ is labelled 1, the edge b_i of C such that i is minimised but greater than $C_{\min}$ is labelled 2, and so on up to $b_{C_{\max}}$, which is labelled $l(C)$.

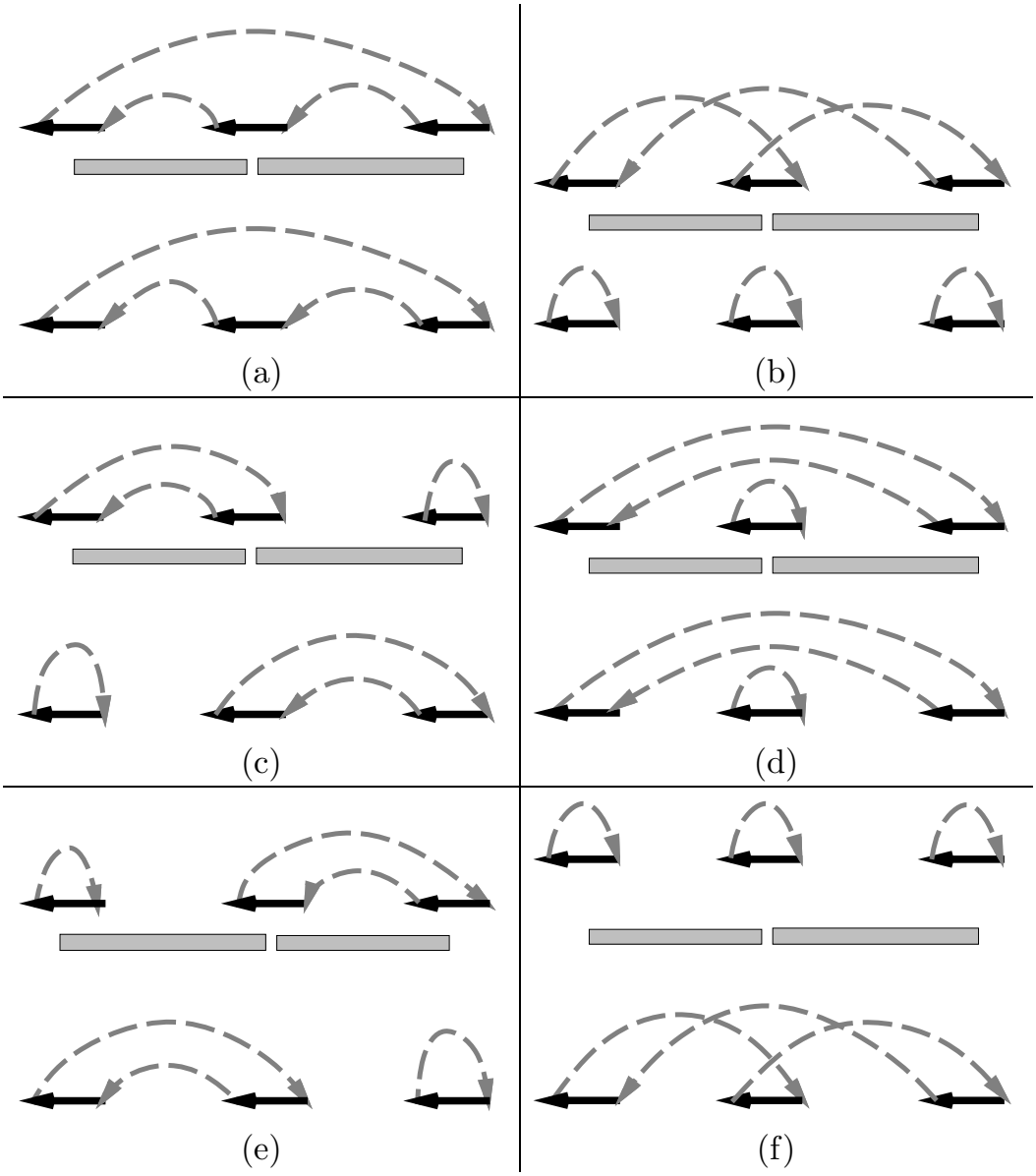

Figure 3.3: How a transposition changes the cycle graph.

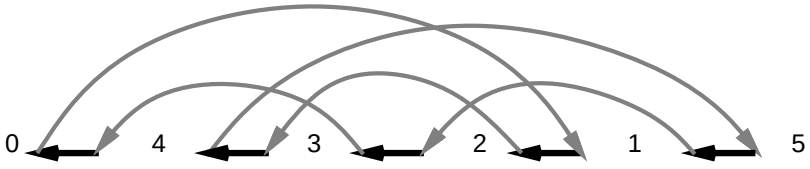

Figure 3.4: $tG(\pi)$, for $\pi = [4\ 3\ 2\ 1]$.

The canonical labelling, $L(C)$, of the cycle C is the permutation of the set $\{1, \ldots, l(C)\}$ obtained from this labelling by starting with the edge labelled 1 then reading the labels in the order they occur around the cycle. Note that all canonical labellings begin with 1.

It will prove useful to split the canonical labelling into two subsequences. The *outer sequence* $O(C)$ is defined by taking the 1st, 3rd, etc. labels from the canonical labelling. The *inner sequence* $I(C)$ is defined by taking the 2nd, 4th, etc. labels from the canonical labelling. The kth elements of these sequences are denoted by $i(C, k)$ and $o(C, k)$ respectively.

Example Let π be the permutation [6 1 5 7 12 4 11 2 9 3 8 10]. The cycle graph of this permutation contains three cycles: $(4, 1, 2)$, $(12, 9, 10)$, and $(13, 7, 3, 8, 5, 11, 6)$, as shown in Figure 3.2. These cycles have canonical labellings [1 2 3], [1 2 3], and [1 5 2 6 3 7 4]. The third cycle has outer labelling [1 2 3 4] and inner labelling [5 6 7]. From this new notation it is perhaps more obvious that two of the cycles are very similar. □

In the old cycle notation, a cycle is unoriented if it is represented by a decreasing sequence. So we may restate Theorem 3.2.4 in terms of canonical labellings as follows

Theorem 3.3.1 *A cycle C is unoriented if and only if*

$$L(C) = [1\ l(C)\ l(C) - 1\ \ldots\ 2].$$

3.3.3 Fully oriented cycles

We now show how to determine if a cycle is fully oriented based on its canonical labelling. We show that all even oriented cycles are fully oriented, whereas some odd oriented cycles are not fully oriented. First we need a proposition characterising oriented triples.

Proposition 3.3.2 *Let $[x\ y\ z]$ be a subsequence of the canonical labelling of a cycle. Then (x, y, z) is an oriented triple if and only if $x < y < z$, or $y < z < x$, or $z < x < y$.*

Proof If x, y and z satisfy one of these three conditions, then the cycle must be oriented. The transposition on edges x, y, and z splits the cycle into three cycles, as shown in Figure 3.3 (b). If x, y and z do not satisfy any of the three conditions, then they must be arranged as shown in Figure 3.3 (a), and the transposition, as shown, does not increase the number of cycles. Therefore the edges form an unoriented triple. □

Now we show that every oriented even length cycle is a fully oriented cycle.

Lemma 3.3.1 *Let $[x\ y\ z]$ be a subsequence of the canonical labelling of an even length cycle C, such that $x < y < z$, or $y < z < x$, or $z < x < y$. Then (x, y, z) is a fully oriented triple if and only if $[x\ y\ z]$ is neither a subsequence of $I(C)$ nor a subsequence of $O(C)$.*

Proof A transposition on these three edges will split C into 3 cycles (Proposition 3.3.2). If $[x\ y\ z]$ is a subsequence of $I(C)$ or $O(C)$ then the paths between x, y and z all contain an odd number of black edges. So the resulting cycles are all even in length. However if $[x\ y\ z]$ is not a subsequence of $I(C)$ or $O(C)$ then two of the resulting cycles will be odd in length. □

Theorem 3.3.2 *If C is an even length cycle that is oriented, then C is also fully oriented.*

Proof Since C is oriented it must have the canonical labelling $[1\ \ldots\ i\ \ldots\ j\ \ldots]$ such that $i < j$. Obviously, 1 is in $O(C)$. If at least one of i and j is in $I(C)$ then $(1, i, j)$ is a fully oriented triple, by Lemma 3.3.1. If both of i and j are in $O(C)$ then there must be a k in $I(C)$ such that C has canonical labelling $[1\ \ldots\ i\ \ldots\ k\ \ldots\ j\ \ldots]$. If $k < j$ then $(1, k, j)$ is a fully oriented triple. If $k > j$ then $(1, i, k)$ is a fully oriented triple. □

We now show that no analogous result can be proved for odd length cycles. However, we are able to characterise those odd length cycles that are oriented but not fully oriented.

Lemma 3.3.2 *Let $[x\ y\ z]$ be a subsequence of the canonical labelling of an odd length cycle C. Then (x, y, z) is a fully oriented triple if and only if one of the following six conditions on x, y and z is true:*

(i) $x < y < z$, *x and z are in $O(C)$ and y is in $I(C)$*
(ii) $x < y < z$, *x and z are in $I(C)$ and y is in $O(C)$*
(iii) $y < z < x$, *x and z are in $O(C)$ and y is in $I(C)$*
(iv) $y < z < x$, *x and z are in $I(C)$ and y is in $O(C)$*
(v) $z < x < y$, *x and z are in $O(C)$ and y is in $I(C)$*
(vi) $z < x < y$, *x and z are in $I(C)$ and y is in $O(C)$*

Proof Suppose that x, y and z satisfy one of these conditions. Then the three edges must form an oriented triple (Proposition 3.3.2). Furthermore the paths in C between x, y and z all contain an even number of black edges, and so the transposition on these edges increases the number of odd length cycles in the cycle graph by two. So the edges form a fully oriented triple.

Suppose that x, y and z do not satisfy any of the above conditions. If (x, y, z) is not an oriented triple then it cannot be a fully oriented triple. If (x, y, z) is an oriented triple then two of the paths in C between x, y and z contain an odd number of black edges, and so the transposition on these edges increases the number of even length cycles in the cycle graph. So in neither case is (x, y, z) a fully oriented triple □

Theorem 3.3.3 *For an odd length cycle of length $2r+1$ ($r > 1$), the canonical labelling $[1\ r+2\ 2\ r+3\ 3\ \ldots\ 2r+1\ r+1]$ is the only canonical labelling possible for a cycle that is oriented, but not fully oriented.*

Proof Clearly this is a canonical labelling of an oriented cycle. However the cycle is not fully oriented, because $O(C)$ is the sequence $[1, 2, \ldots, r+1]$ and $I(C)$ is the sequence $[r+2, r+3, \ldots, 2r+1]$. Hence, it is impossible to find edges x, y and z that satisfy any of the six conditions of Lemma 3.3.2.

We now show that this is the only possible canonical labelling for an oriented cycle that is not fully oriented. Let C be an odd length cycle of length $2r+1$ that is oriented, but is not fully oriented.

In particular let us first suppose that $o(C,k) \leq r+1$ for all values of k. This restriction means that $o(C,k) < i(C,l)$ for all values of k and l. If $i(C,k) > i(C,l)$ but $k < l$ then by Lemma 3.3.2 the black edges at positions $o(C,l)$, $i(C,l)$, and $i(C,k)$ form a fully oriented triple. Therefore $I(C)$ must be the sequence $[r+2\ r+3\ \ldots\ 2r+1]$. Similarly $O(C)$ must be the sequence $[1\ 2\ \ldots\ r+1]$.

This means that the canonical labelling $[1\ r+2\ 2\ r+3\ 3\ \ldots\ 2r+1\ r+1]$ is the only canonical labelling that represents a cycle that is not fully oriented and for which $o(C,k) \leq r+1$ for all values of k. So in any alternative canonical labelling of a cycle that is not fully oriented there must be an $o(C,k)$ such that $o(C,k) > r+1$. Let C be a cycle with just such an alternative canonical labelling.

Let $o(C,k)$ be the largest value in $O(C)$, and let $i(C,l)$ be the smallest value in $I(C)$. Note that $o(C,k) > r+1$ and so $i(C,l) \leq r+1$. In particular $l \geq k$ because otherwise $(1, i(C,l), o(C,k))$ would be a fully oriented triple, by Lemma 3.3.2. So C has canonical labelling $[1\ \ldots\ o(C,k)\ \ldots\ i(C,l)\ \ldots]$.

Now since C is not fully oriented it must be the case that $i(C,m) > o(C,k)$ for all $1 \leq m < k$, since otherwise $(i(C,m), o(C,k), i(C,l))$ would be a fully oriented triple. Similarly $o(C,m) < i(C,l)$ for all $l < m \leq r+1$, because otherwise $(1, i(C,l), o(C,m))$ would be a fully oriented triple. Furthermore $o(C,s) > i(C,l)$ for all $2 \leq s \leq l$ since otherwise $(i(C,1), o(C,s), i(C,l))$ would form a fully oriented triple.

If $2r+1$ were in $O(C)$ then $(1, i(C,1), 2r+1)$ would be a fully oriented triple, so $2r+1$ is in $I(C)$. Suppose $i(C,1) \neq 2r+1$ then $i(C,p) = 2r+1$ for some p, $1 < p \leq r$. Then $(o(C,2), i(C,p), o(C,r+1))$ would be a fully oriented triple because then $o(C,r+1) < i(C,l) < o(C,2) < i(C,p)$. So $i(C,1) = 2r+1$. If $2r$ were in $I(C)$ then $(2r+1, o(C,2), 2r)$ would be a fully oriented triple, so $2r$ is in $O(C)$. In particular $o(C,k) = 2r$, and so $k = 2$ since $i(C,m) > o(C,k)$ for all $1 \leq m < k$.

Now suppose that $C \neq [1\ 2r+1\ 2r\ \ldots 3\ 2]$. Then there must be an $i(C,q) < o(C,r)$ such that $q < r$ or an $o(C,q) < i(C,r)$ such that $q \leq r$ and $q > 1$. If the first case were true then $(2r, i(C,q), o(C,r))$ would be a fully oriented triple. If the second case were true then $(2r+1, o(C,q), i(C,r))$ would be a fully oriented triple. So $C = [1\ 2r+1\ 2r\ \ldots 3\ 2]$, which is not oriented. □

We can use these results about the canonical labellings of fully oriented cycles, to prove that small cycles are less likely to be fully oriented.

Lemma 3.3.3 *A cycle of length l is less likely to be fully oriented than a cycle of length $l+1$, assuming all canonical labellings are equally likely.*

Proof If an odd length cycle (of length $2r+1$) is not fully oriented then it must have canonical labelling $[1\ 2r+1\ \ldots\ 2]$, or $[1\ r+2\ 2\ r+3\ 3\ \ldots\ 2r+1\ r+1]$. Now there are $(2r)!$ possible canonical labellings of an odd length cycle. So the chance that an odd length cycle is not fully oriented is only $2/(2r)!$.

If an even length cycle (of length $2r$) is not fully oriented then it must have canonical labelling $[1\ 2r\ \ldots\ 2]$. So the chance that an even length cycle is not fully oriented is only $1/(2r-1)!$.

The proof is completed by noting that $1/(2r-1)! > 2/(2r)! > 1/(2r+1)!$. □

We shall use this lemma as the basis of a successful heuristic for solving instances of sorting by transpositions in a later section.

3.3.4 Knots

In this section we prove some results about cycles that are oriented, but not fully oriented. We call such cycles *knots*. We define ω_r to be the permutation of length $2r$ whose cycle graph consists of a knot of length $2r+1$, if such a permutation exists.

Lemma 3.3.4 *If* $r \equiv 1 \pmod 3$ *then* ω_r *is not defined, but* ω_r *exists (and is unique) for all other values of* r.

Proof By Theorem 3.3.3, a knot of length $2r+1$ has canonical labelling $[1\ r+2\ 2\ r+3\ 3\ \ldots\ 2r+1\ r+1]$. So if ω_r exists then $tG(\pi)$ contains r grey edges directed to the right, namely $\{(\pi(i), \pi(i+r+2)) : 0 \leq i \leq r-1\}$, and $r+1$ grey edges directed to the left, namely $\{(\pi(i), \pi(i-(r-1))) : r \leq i \leq 2r\}$. Therefore all grey edges are directed from $\pi(i)$ to $\pi((i+r+2) \bmod (2r+1))$ (except the edge $(\pi(r-1),\pi(2r+1))$).

Let S be the sequence that is defined by the recurrence relation:

$$\begin{aligned} S(0) &= 0 \\ S(i+1) &= (S(i)+r+2) \bmod (2r+1), \text{ (for } i \geq 0). \end{aligned}$$

Then $S = [0\ r+2\ 3\ \ldots\ ]$. Suppose that ω_r exists. Let us follow grey edges in $tG(\omega_r)$, starting from 0, i.e., visit vertices of $tG(\pi)$ in the order 0, 1, 2, etc.. Now the positions of these elements in π is given by the sequence S. Therefore ω_r exists if and only if the period of S is $2r+1$.

Now the period of S is $2r+1$ if and only if $r+2$ and $2r+1$ are co-prime, i.e., $gcd(r+2, 2r+1) = 1$ where $gcd(x, y)$ is the greatest common divisor of x and y. Now $gcd(r+2, 2r+1) = 1 \Leftrightarrow gcd(r+2, r-1) = 1 \Leftrightarrow gcd(3, r-1) = 1$. Therefore ω_r exists if and only if $r \not\equiv 1 \pmod 3$. □

By considering S, the sequence defined in the proof of Lemma 3.3.4, it is easy to verify that $\omega_r(i) = (i.4(r+1)/3) \bmod (2r+1)$ if $r \equiv 2 \pmod 3$, and $\omega_r(i) = (i.(2r/3+1)) \bmod (2r+1)$ if $r \equiv 0 \pmod 3$

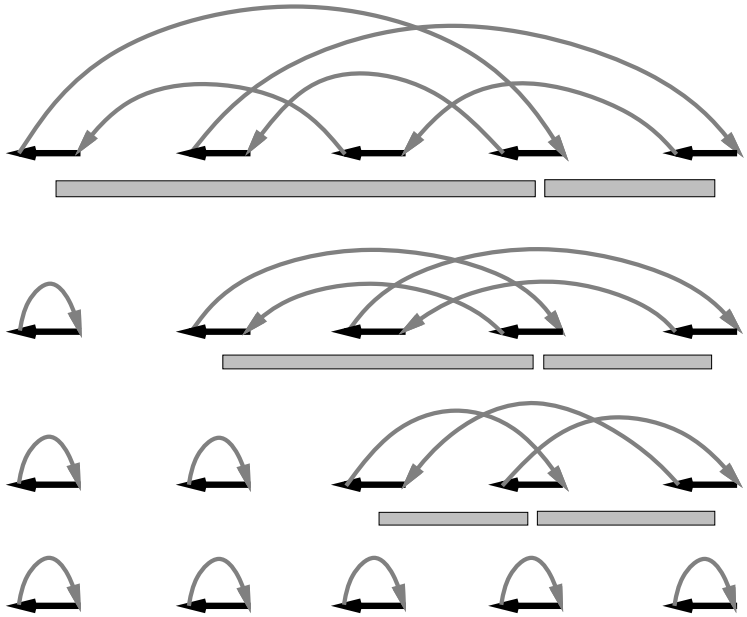

Figure 3.5: How to sort a knot of length 5.

Lemma 3.3.5 *If $tG(\pi)$ contains a knot, then we may apply a $(0, 2, 2)$-sequence of transpositions on the knot.*

Proof We can sort ω_2 using a $(0, 2, 2)$-sequence as shown in Figure 3.5. Edges 1, 2, $r + 1$, $r + 2$, and $2r + 1$ of a larger knot are connected in exactly the same way as the five edges of ω_2 are connected, except that there is an alternating path (that contains an even number of black edges) between edges 2 and $2r + 1$ instead of a single grey edge. The sequence of transpositions on ω_2 may be applied to these five edges of the larger knot. The resulting transpositions form a $(0, 2, 2)$-sequence. □

Lemma 3.3.5 is sufficiently powerful to obtain the 3/2-approximation algorithm in Section 3.4. However, below we show that in fact a much stronger result can be proved, though we warn the reader that the proof of this theorem involves a lengthy and detailed case by case analysis.

Theorem 3.3.4 *If $r \not\equiv 1 \pmod 3$ then $td(\omega_r) = r + 1$.*

Proof Since, ω_r contains only one cycle, and it is not fully oriented, an initial 2-transposition cannot be applied to ω_r. So $td(\omega_r) \geq r + 1$.

$\underline{8\ 5\ 2\ 10\ 7\ 4}\ \underline{1}\ 9\ 6\ 3$
$1\ \underline{8\ 5}\ \underline{2\ 10\ 7\ 4\ 9}\ 6\ 3$
$1\ 2\ 10\ 7\ \underline{4\ 9}\ \underline{8\ 5\ 6\ 3}$
$1\ 2\ \underline{10}\ \underline{7\ 8\ 5\ 6\ 3\ 4\ 9}$
$1\ 2\ \underline{7\ 8}\ \underline{5\ 6}\ 3\ 4\ 9\ 10$
$1\ 2\ \underline{5\ 6\ 7\ 8}\ \underline{3\ 4}\ 9\ 10$
$1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 10$

Figure 3.6: A sequence of transpositions that sorts ω_5.

Let us define a sequence of transpositions $\tau_1, \tau_2, \ldots, \tau_{r+1}$, as follows.

$$\tau_i = \begin{cases} \tau(1, r+2, r+3) & i = 1 \\ \tau(2+\frac{i-2}{3}, 2+i, r+4+2\frac{i-2}{3}) & 2 \le i \le r-1,\ i \equiv 2 \pmod 3 \\ \tau(r-2\frac{i-3}{3}, r+2, 2r+1-\frac{i-3}{3}) & 2 \le i \le r-1,\ i \equiv 0 \pmod 3 \\ \tau(3+\frac{i-4}{3}, r-1-2\frac{i-4}{3}, 2r+1-\frac{i-4}{3}) & 2 \le i \le r-1,\ i \equiv 1 \pmod 3 \\ \tau(1+\frac{r+1}{3}, 2\frac{r+1}{3}+1, r+2) & i = r,\ r \equiv 2 \pmod 3 \\ \tau(1+\frac{r+1}{3}, r+2, 4\frac{r+1}{3}+1) & i = r+1,\ r \equiv 2 \pmod 3 \\ \tau(\frac{r}{3}+2, \frac{2r}{3}+2, \frac{4r}{3}+2) & i = r,\ r \equiv 0 \pmod 3 \\ \tau(\frac{r}{3}+2, \frac{2r}{3}+2, r+2) & i = r+1,\ r \equiv 0 \pmod 3 \end{cases}$$

In Figure 3.6 it is demonstrated that this sequence of transpositions sorts ω_5. We prove by induction that this sequence of transpositions sorts ω_r for all r. Let π_i be the permutation obtained after applying the first i transpositions to ω_r, i.e., $\pi_i = \omega_r \cdot \tau_1 \cdots \tau_i$.

Let us define δ to be equal to $\omega_r(1)$. By the proof of Lemma 3.3.4 this definition is equivalent to

$$\delta = \begin{cases} 4\left(\frac{r+1}{3}\right) & r \equiv 2 \pmod 3 \\ \frac{2r}{3}+1 & r \equiv 0 \pmod 3 \end{cases}$$

Note that, for both cases $2\delta \equiv 2r+3-\delta \pmod{2r+1}$.

To help us describe π_i at each step, we need the following definitions of subsequences of ω_r and $\imath$. Let $I(i,k) = [i\ i+1\ \ldots\ i+k-1]$. Let $I^*(i,k) = [i-k+1\ \ldots\ i-1\ i]$. Let $K(i,j,k) = [\omega_r(m)\ \omega_r(m+1)\ \ldots\ \omega_r(m+k-1)]$, where $\omega_r(m) = i$, and $\omega_r(m+k-1) = j$.

Let $B_1(k) = I(1,k)$, $B_2(k) = I^*(2r+2-\delta, k)$, $B_3(k) = I(\delta+1, k)$, $B_4(k) = I^*(\delta, k)$, $B_5(k) = I(2r+3-\delta, k)$, and $B_6(k) = I^*(2r, k)$.

For $1 \le i \le r-2$ we shall show that π_i is defined as follows:
If $i \equiv 1 \pmod 3$ then $\pi_i = [B_1((i-1)/3+1)\ B_4((i+2)/3)\ B_5((i+2)/3)\ K((2+(i-1)/3, \delta-1-(i-1)/3, r-1-i)\ B_2((i+2)/3)\ B_3((i+2)/3)\ K(2r+3-\delta+(i+2)/3, 2r+2-\delta-(i+2)/3, r-2-i)\ B_6((i-1)/3)]$.

If $i \equiv 2 \pmod 3$ then $\pi_i = [B_1((i+1)/3+1)\ K(\delta+1+(i+1)/3, \delta-(i+1)/3, r-1-i)\ B_2((i+1)/3)\ B_3((i+1)/3)\ B_4((i+1)/3)\ B_5((i+1)/3+1)\ K(2+(i+1)/3, 2r+2-\delta-(i+1)/3, r-2-i)\ B_6((i-2)/3)]$.
If $i \equiv 0 \pmod 3$ then $\pi_i = [B_1(i/3+1)\ K(\delta+1+i/3, 2r+1-i/3, r-1-i)\ B_4(i/3+1)\ B_5(i/3+1)\ K((2+i/3, \delta-1-i/3, r-2-i)\ B_2(i/3+1)\ B_3(i/3)\ B_6(i/3-1)]$.

We shall prove this by induction. The base case is π_1. The first transposition is $\tau_1 = \tau(1, r+2, r+3)$. This transposition moves 1 to the front of the permutation since $\omega_r(r+2) = 1$. Note also that $\pi_1(2) = \omega_r(1) = \delta$, $\pi_1(3) = \omega_r(2) = 2\delta = 2r+3-\delta$, $\pi_1(4) = \omega_r(3) = 2$, $\pi_1(r+1) = \omega_r(r) = 2r+2-2\delta = 2r+1-(2r+3-\delta) = \delta-1$, $\pi_1(r+2) = \omega_r(r+1) = 2r+2-\delta$, $\pi_1(r+3) = \omega_r(r+3) = 1+\delta$, $\pi_1(r+4) = \omega_r(r+4) = 1+2\delta = 2r+4-\delta$, and $\pi_1(2r) = \omega_r(2r) = 2r+1-\delta$. So $\pi_1 = [1\ \delta\ 2r+3-\delta\ K(2, \delta-1, r-2)\ 2r+2-\delta\ 1+\delta\ K(2r+4-\delta, 2r+1-\delta, r-3)]$, which can be re-written as $[B_1(1)\ B_4(1)\ B_5(1)\ K(2, \delta-1, r-2)\ B_2(1)\ B_3(1)\ K(2r+4-\delta, 2r+1-\delta, r-3)\ B_6(0)]$, which is the required form.

Now the induction step. Suppose π_i is as described above. We shall apply transposition τ_{i+1} and show that π_{i+1} is as described above. There are three different cases that need to be considered.

Suppose $i \equiv 1 \pmod 3$. Then $\tau_{i+1} = \tau(2+(i-1)/3, 2+i, r+4+2(i-1)/3)$, so applying this transposition results in the permutation $[B_1((i-1)/3+1)\ K((2+(i-1)/3, \delta-1-(i-1)/3, r-1-i)\ B_2((i+2)/3)\ B_3((i+2)/3)\ B_4((i+2)/3)\ B_5((i+2)/3)\ K(2r+3-\delta+(i+2)/3, 2r+2-\delta-(i+2)/3, r-2-i)\ B_6((i-1)/3)]$. So we can extend B_1 and B_5 by one. Note also that $2+(i-1)/3+\delta = \delta+1+(i+2)/3$, and $2r+3-\delta+(i+2)/3+\delta = 2r+3+(i+2)/3$. So we can rewrite the permutation as $[B_1((i+2)/3+1)\ K((\delta+1+(i+2)/3, \delta-1-(i-1)/3, r-1-(i+1))\ B_2((i+2)/3)\ B_3((i+2)/3)\ B_4((i+2)/3)\ B_5((i+2)/3+1)\ K(2r+3+(i+2)/3, 2r+2-\delta-(i+2)/3, r-2-(i+1))\ B_6((i-1)/3)]$, which is the required form.

Suppose $i \equiv 2 \pmod 3$. Then $\tau_{i+1} = \tau(r-2(i-2)/3, r+2, 2r+1-(i-2)/3)$, so applying this transposition results in the permutation $[B_1((i+1)/3+1)\ K(\delta+1+(i+1)/3, \delta-(i+1)/3, r-1-i)\ B_4((i+1)/3)\ B_5((i+1)/3+1)\ K(2+(i+1)/3, 2r+2-\delta-(i+1)/3, r-2-i)\ B_2((i+1)/3)\ B_3((i+1)/3)\ B_6((i-2)/3)]$. So we can extend B_4 and B_2 by one. Note also that $\delta > (i+1)/3$. So $2r+1-(i+1)/3+\delta = \delta-(i+1)/3$, and $\delta-1-(i+1)/3+\delta = 2r+2-\delta-(i+1)/3$. So we can rewrite the permutation as $[B_1((i+1)/3+1)\ K(\delta+1+(i+1)/3, 2r+1-(i+1)/3, r-1-(i+1))\ B_4((i+1)/3+1)\ B_5((i+1)/3+1)\ K(2+(i+1)/3, \delta-1-(i+1)/3, r-2-(i+1))\ B_2((i+1)/3+1)\ B_3((i+1)/3)\ B_6((i-2)/3)]$, which is the required form.

Suppose $i \equiv 0 \pmod 3$. Then $\tau_{i+1} = \tau(3+(i-3)/3, r-1-2(i-3)/3, 2r+1-(i-3)/3)$, so applying this transposition results in the permutation $[B_1(i/3+1)\ B_4(i/3+1)\ B_5(i/3+1)\ K((2+i/3, \delta-1-i/3, r-2-i)\ B_2(i/3+1)\ B_3(i/3)\ K(\delta+1+i/3, 2r+1-i/3, r-1-i)\ B_6(i/3-1)]$. Clearly B_3 and B_6 can be extended by one. Note also that $\delta+1+i/3+\delta = 2r+4-\delta+i/3$ and $2r+1-\delta-i/3+\delta = 2r+1-i/3$. So we can re-write this permutation as $[B_1(i/3+1)\ B_4(i/3+1)\ B_5(i/3+1)\ K((2+i/3, \delta-1-i/3, r-1-$

$(i+1))$ $B_2(i/3+1)$ $B_3(i/3)$ $K(2r+4-\delta+i/3, 2r+1-\delta-i/3, r-2-(i+1))$ $B_6(i/3-1)]$, which is the required form.

We now need to show that performing the last three transpositions results in the identity permutation.

If $r \equiv 2 \pmod 3$ then $\pi_{r-1} = [B_1((r+1)/3)$ $[\delta + (r+1)/3]$ $B_4((r+1)/3)$ $B_5((r+1)/3)$ $B_2((r+1)/3)$ $B_3((r-2)/3)$ $B_6((r-2)/3-1)]$. The next transposition is $\tau_{r-1} = \tau(1 + (r+1)/3, 2 + (r+1)/3, 5(r+1)/3 + 1)$. So $\pi_{r-1} = [B_1((r+1)/3)$ $B_4((r+1)/3)$ $B_5((r+1)/3)$ $B_2((r+1)/3)$ $B_3((r+1)/3)$ $B_6((r-2)/3-1)]$. The next transposition is $\tau_r = ((r+1)/3 + 1, 2(r+1)/3 + 1, r+2)$, therefore $\pi_r = [B_1((r+1)/3)$ $B_5((r+1)/3)$ $B_4((r+1)/3)$ $B_2((r+1)/3)$ $B_3((r+1)/3)$ $B_6((r-2)/3-1)]$. The next transposition is $\tau_{r+1} = \tau((r+1)/3 + 1, r+2, 4(r+1)/3 + 1)$. So $\pi_{r+1} = [B_1((r+1)/3)$ $B_2((r+1)/3)$ $B_5((r+1)/3)$ $B_4((r+1)/3)$ $B_3((r+1)/3)$ $B_6((r-2)/3-1)]$, which is the identity permutation.

If $r \equiv 0 \pmod 3$ then $\pi_{r-2} = [B_1(r/3)$ $B_4(r/3)$ $B_5(r/3)$ $[r/3 + 1]$ $B_2(r/3)$ $B_3(r/3)$ $B_6(r/3 - 1)]$. The next transposition is $\tau_{r-1} = \tau(r/3 + 1, r + 1, 5r/3 + 2)$. So $\pi_{r-1} = [B_1(r/3)$ $[r/3 + 1]$ $B_2(r/3)$ $B_3(r/3)$ $B_4(r/3)$ $B_5(r/3)$ $B_6(r/3 - 1)]$. The next transposition is $\tau_r = \tau(r/3 + 2, 2r/3 + 2, 4r/3 + 2)$. So $\pi_r = [B_1(r/3 + 1)$ $B_3(r/3)$ $B_4(r/3)$ $B_2(r/3)$ $B_5(r/3)$ $B_6(r/3 - 1)]$. The last transposition is $\tau_{r+1} = \tau(r/3 + 2, 2r/3 + 2, r + 2)$. So $\pi_{r+1} = [B_1(r/3 + 1)$ $B_4(r/3)$ $B_3(r/3)$ $B_2(r/3)$ $B_5(r/3)$ $B_6(r/3 - 1)]$ which is the identity permutation. □

3.3.5 Strongly oriented cycles

Bafna and Pevzner define *strongly oriented cycles* to be oriented cycles that 'have the simplest "self-interleaving" structure among all oriented cycles'. They define a cycle to be strongly oriented if it is oriented, and an interleaving transposition (that does not act on any edges of the cycle) can transform the cycle into an unoriented cycle. A cycle that is strongly oriented must be fully oriented. This is because it contains only two grey edges directed to the right, and knots of length five (the only knots with only two grey edges directed to the right) are not strongly oriented cycles. However, many fully oriented cycles are not strongly oriented.

Bafna and Pevzner use strongly oriented cycles because they show that these cycles admit 2-transpositions on their edges. However, we now have a better definition of such cycles, the fully oriented cycles, that encompasses all cycles that admit 2-transpositions on their edges, so we shall use fully oriented cycles instead of strongly oriented cycles. Later in this chapter we prove several results on fully oriented cycles that are similar to results Bafna and Pevzner proved about strongly oriented cycles.

3.3.6 Super oriented cycles

A cycle C is said to be *super oriented* if it is possible to apply a 2-transposition C followed by a 2-transposition on one of cycles that C was split into.

If we could characterise super oriented cycles in the same way that we can characterise fully oriented cycles, then we may be able to improve the lower bounds for transposition distance. Unfortunately, such a characterisation of super oriented cycles has so far proved elusive. However, it is possible to detect some cycles that are not super oriented, and for some permutations this helps us to improve the lower bound for transposition distance. To describe this result we need an extra definition.

Define a grey edge $(i, i+1)$ to be *crossing relative to the transposition* $\tau(x, y, z)$ if either:

$$(i) \quad x \leq \pi^{-1}(i) \leq y-1 \text{ and } y \leq \pi^{-1}(i+1) \leq z-1, \text{ or}$$
$$(ii) \quad x \leq \pi^{-1}(i+1) \leq y-1 \text{ and } y \leq \pi^{-1}(i) \leq z-1.$$

This definition is motivated by the following lemma.

Lemma 3.3.6 *Let τ be a transposition that transforms π into π'. Then a grey edge g has different directions in $tG(\pi)$ and $tG(\pi')$ if and only if g is crossing relative to τ.*

Proof We can use the transposition $\tau = \tau(x, y, z)$ to partition the set $\{0, \ldots, n+1\}$ into three sets I_1, I_2, and I_3 as follows. Let $I_1 = \{\pi(i) : 0 \leq i < x\} \cup \{\pi(i) : z \leq i \leq n+1\}$, $I_2 = \{\pi(i) : x \leq i < y\}$, and $I_3 = \{\pi(i) : y \leq i < z\}$. Now if i and $i+1$ are in the same set under this partitioning, then it is obvious that the direction of the grey edge $(i, i+1)$ does not change as the result of applying τ. Similarly if i or $i+1$ is in I_1 then it is easy to see that the direction of $(i, i+1)$ does not change as the result of applying τ. In the remaining two cases $(i, i+1)$ is crossing relative to τ, and the action of τ does change the direction of the grey edge. □

Lemma 3.3.7 *Let C be a cycle that contains only two grey edges that are directed from left to right. Then C is not super oriented.*

Proof Consider any oriented triple (x, y, z) of C, such that $x < y < z$. Then only one grey edge of C is crossing relative to $\tau(x, y, z)$, and it must occur on the path between z and x. By Lemma 3.3.6, this crossing edge is the only grey edge in C that changes direction, and it becomes an edge directed to the right. The transposition $\tau(x, y, z)$ splits cycle C into three cycles that each contain only one grey edge directed to the right, so each cycle must be unoriented. Therefore C is not super oriented. □

It is an open problem to determine if any cycle exists that is not super oriented, but is fully oriented and has more than two grey edges directed to the right. Perhaps such a cycle, if one exists, must produce a knot as the result of a 2-transposition on its edges?

3.4 A new 3/2-approximation algorithm

The observations about even cycles, fully oriented cycles, and knots made in earlier sections of this chapter, allow us to present a new 3/2-approximation algorithm that

is somewhat simpler than that presented by Bafna and Pevzner. We achieve the 3/2-approximation bound by describing a sequence of transpositions that sorts a permutation and contains no (-2)-transposition, and at least two 2-transpositions for every 0-transposition.

Such a sequence of transpositions requires no more than 3/2 times the minimum number of transpositions required to sort the permutation as predicted by Theorem 3.2.3. A useful definition for describing the 3/2-approximation algorithm is that of a (0,2,2)-sequence. A $(0, 2, 2)$-*sequence* of transpositions is a sequence of three transpositions such that the first transposition is a 0-transposition, and the next two transpositions are 2-transpositions.

If for all permutations, apart from the identity permutation, we can find a 2-transposition or a (0,2,2)-sequence, then we can achieve the approximation bound by repeatedly using this result, until the permutation has been sorted. Now we already know from the previous sections that we can apply a 2-transposition to a permutation if its cycle graph contains a fully oriented cycle or an even length cycle. If $tG(\pi)$ contains a knot then we know that we can find a (0,2,2)-sequence, at least, on π.

So now we consider the case when $tG(\pi)$ contains only odd length cycles that are all unoriented. To study this case we need some extra definitions and some auxiliary results.

Two pairs of black edges (b_i, b_j) and (b_k, b_l) are said to *intersect* if $i < k < j < l$, or $k < i < l < j$. Similarly, a cycle intersects with two black edges x and y if the cycle contains two black edges that intersect with x and y. Similarly, cycles C and D intersect if C contains two black edges that intersect with two black edges of D.

Triples of black edges (b_i, b_j, b_k) and (b_l, b_m, b_n) *interleave* if $i < l < j < m < k < n$ or $l < i < m < j < n < k$. Similarly, cycles C and D interleave if C contains a triple of black edges that interleave with a triple of black edges of D. Two transpositions interleave if the triples of black edges that the transpositions act on interleave. A transposition interleaves with a cycle C if the three edges that the transposition acts on interleave with three black edges of C.

Bafna and Pevzner stated the following lemma for oriented cycles.

Lemma 3.4.1 *Let (x, y, z) be a triple in a cycle C, and let $u,\ v,\ w \notin C$ be black edges of $tG(\pi)$. Then $\tau(u, v, w)$ changes the orientation of triple (x, y, z) (i.e., it transforms an oriented triple into an unoriented triple, and vice versa) if and only if (u, v, w) interleaves with (x, y, z).*

We prove several similar lemmas for fully oriented triples.

Lemma 3.4.2 *Let (x, y, z) be a fully oriented triple in a cycle C, and let u, v, $w \notin C$ be black edges of $tG(\pi)$. If (u, v, w) interleaves with (x, y, z) then $\tau(u, v, w)$ transforms (x, y, z) into an unoriented triple in $tG(\pi \cdot \tau)$. Otherwise, (x, y, z) is a fully oriented triple in $tG(\pi \cdot \tau)$.*

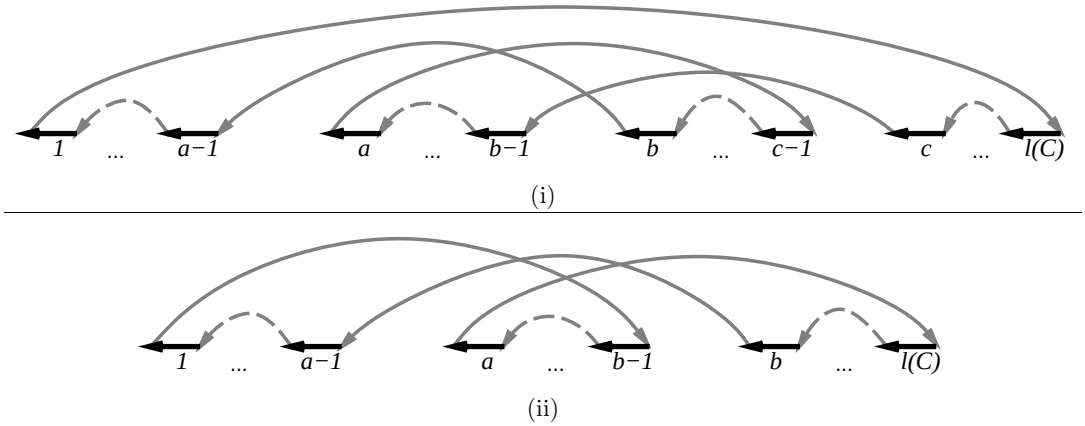

Figure 3.7: The oriented cycle obtained by an applying an transposition that interleaves with an unoriented cycle.

Proof If the triples are interleaved then (x, y, z) is an unoriented triple in $tG(\pi \cdot \tau)$ by Lemma 3.4.1. Otherwise, after the transposition has been applied, $x < y < z$, or $y < z < x$ or $z < x < y$ so (x, y, z) is an oriented triple by Proposition 3.3.2. Further, (x, y, z) is a fully oriented triple, because the lengths of the paths connecting the edges are no different than before. □

Note that it is not true that an unoriented triple can be converted into a fully oriented triple just by applying a transposition on edges that interleave with the unoriented triple. However, we can prove the following lemma about unoriented cycles.

Lemma 3.4.3 *A transposition that interleaves with an unoriented cycle converts the unoriented cycle into a fully oriented cycle.*

Proof Let C be an unoriented cycle, and let τ be a transposition that interleaves with C. Suppose that $C_{\min}$ is to the left of the leftmost black edge that τ acts upon, and $C_{\max}$ is to the right of the rightmost black edge that τ acts upon. Then after applying τ, C is a cycle with canonical labelling $[1\ l(C)\ \ldots\ c\ b-1\ \ldots\ a\ c-1\ \ldots\ b\ a-1\ \ldots\ ]$ for some $1 < a < b < c < l(C)$ as shown in Figure 3.7 (i) (in which the black edges are given their canonical labelling). Otherwise C becomes a cycle with canonical labelling $[1\ b-1\ \ldots\ a\ l(C)\ \ldots\ b\ a-1\ \ldots\ ]$ for some $1 < a < b < l(C)$ as is shown in Figure 3.7 (ii). In both cases the cycle is oriented, and in neither case is the cycle a knot, so the cycle is fully oriented. □

We now have enough auxiliary results to find a $(0, 2, 2)$-sequence when $tG(\pi)$ contains only odd length unoriented cycles, and contains two interleaving cycles.

Lemma 3.4.4 *Let π be a permutation, such that $tG(\pi)$ contains only unoriented odd cycles. Suppose that $tG(\pi)$ contains two cycles, C and D, that interleave. Then there exists a $(0, 2, 2)$-sequence of transpositions on π.*

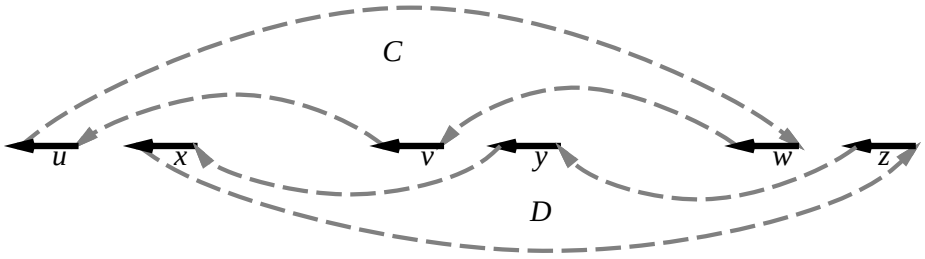

Figure 3.8: How C and D are interleaved initially.

Proof Let C be the cycle with the leftmost black edge. The two cycles are interleaved, so let u, v and w be labels of edges of C and x, y and z be labels of edges of D such that these triples interleave, and such that u is the edge $b_{C_{\min}}$. Further we may pick x to be the leftmost edge of D to the right of u (so x is the edge $b_{D_{\min}}$), y to be the leftmost edge of D to the right of v, and z to be the leftmost edge of D to the right of w. So $tG(\pi)$ is as shown in Figure 3.8.

The transposition $\tau = \tau(u, v, w)$ converts C into an unoriented cycle C' of $tG(\pi \cdot \tau$ and D into a fully oriented by Lemma 3.4.3.

We show that D' contains a fully oriented triple of edges that interleave with a triple of edges of C'. If (y, x, z) is a fully oriented triple of D' then this is obvious. If (y, x, z) is not a fully oriented triple of D' then, as a consequence of Lemma 3.3.2, two of the paths connecting x, y and z must contain an odd number of black edges.

Suppose that it is the paths connecting x to z and y to x that both contain an odd number of black edges. (Then since D' is odd the path connecting z to y contains an even number of black edges.) Let s be the label of the first black edge on the path connecting y to x. Then $p(\pi, s) < p(\pi, v)$ since D is unoriented in $tG(\pi)$, and y is the leftmost edge of D to the right of v. So (y, s, z) is a fully oriented triple in $tG(\pi \cdot \tau)$ (Figure 3.9(a)).

Suppose that it is the paths connecting y to x and z to y that both contain an odd number of black edges. Let s be the label of the first edge on the path connecting z to y. Then $p(\pi, s) < p(\pi, w)$ since D is unoriented in $tG(\pi)$, and z is the leftmost edge of D to the right of w. So (s, x, z) is a fully oriented triple in $tG(\pi \cdot \tau)$ (Figure 3.9(b)).

Suppose that it is the paths connecting z to y and x to z that both contain an odd number of black edges. Let s be the label of the first edge on the path connecting x to z. Then $p(\pi, s) > p(\pi, z)$, since D is unoriented in $tG(\pi)$, and x is the leftmost edge of D. So (y, x, s) is a fully oriented triple in $tG(\pi \cdot \tau)$ as required (Figure 3.9(c)).

So D' contains a fully oriented triple of edges that interleaves with C'. If we apply the transposition on the fully oriented triple then C' will be a fully oriented cycle in the cycle graph of the resulting permutation, by Lemma 3.4.3. So a 2-transposition can

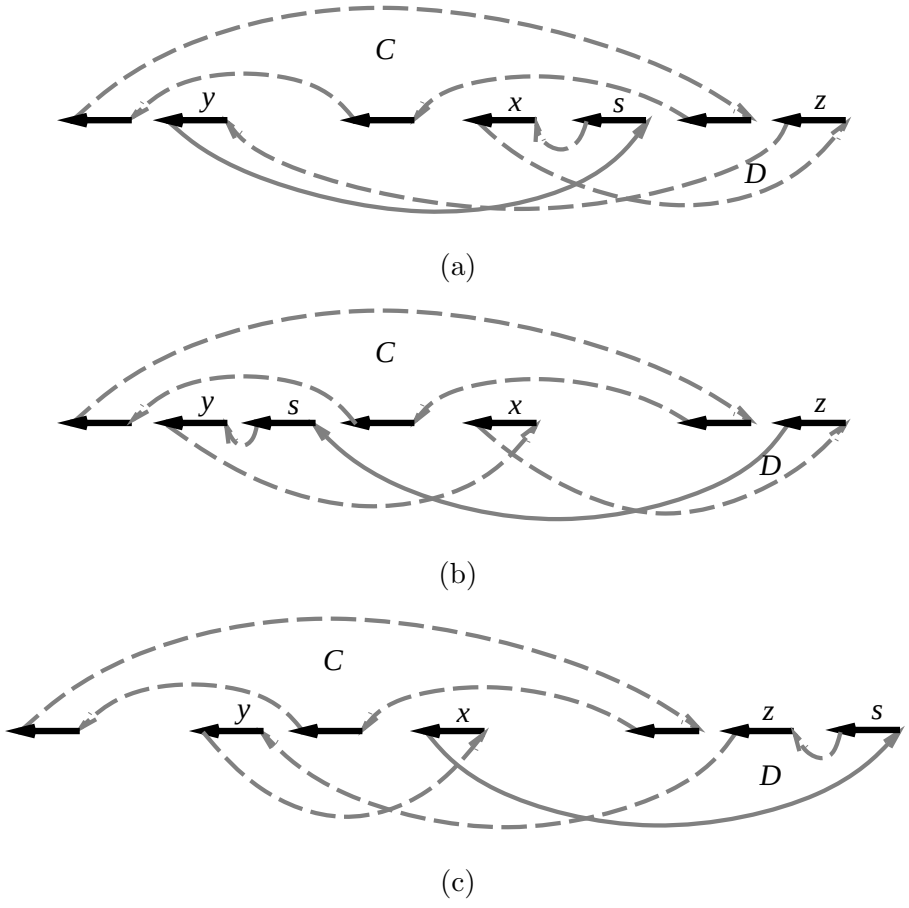

Figure 3.9: How the cycles are interleaved after the 0-transposition.

then be performed on C'. Hence we can perform a $(0, 2, 2)$-sequence of transpositions on π. □

So now we only need to be able to find a (0,2,2)-sequence when all the cycles in $tG(\pi)$ are odd and unoriented, and no pair are interleaving. In fact we can find such a (0,2,2)-sequence, but to prove this we require several auxiliary results. We begin with a lemma proved by Bafna and Pevzner that shows that unoriented cycles must intersect with other cycles.

Lemma 3.4.5 *Let C be an unoriented cycle of $tG(\pi)$. Let x and y be labels of two black edges of C. Then there is a cycle D in $tG(\pi)$ that intersects with x and y.*

Lemma 3.4.6 *Let C and D be unoriented cycles of $tG(\pi)$, such that C and D intersect, and C has the leftmost black edge of the two cycles. Then the transposition $\tau(C_{\min}, D_{\min}, D_{\max})$ converts C and D into a 1-cycle and an oriented cycle. Further, the oriented cycle is fully oriented unless C and D are 3-cycles that interleave.*

Proof Let x be the leftmost edge of C to the right of $D_{\min}$, and let y be the rightmost edge of C to the left of $D_{\min}$. Then depending on whether $C_{\max} < D_{\max}$ or $D_{\max} < C_{\max}$ the transposition $\tau(C_{\min}, D_{\min}, D_{\max})$ acts as shown in Figure 3.10 (i) or Figure 3.10 (ii). (Of course, the edge x could be the edge $b_{C_{\max}}$, and the edge y could be the edge $b_{C_{\min}}$, and therefore the graphs would be not exactly as shown in the figure, but in either case the resulting cycle decomposition is very similar to that shown.) The transposition converts C and D into a 1-cycle and a long cycle C'. In both cases C' is an oriented cycle, so C' is fully oriented unless it is a knot. If $D_{\max} < C_{\max}$ then C' cannot be a knot because its canonical labelling begins $[1\ l\ \ldots\ ]$, where l is the length of the cycle. Otherwise, C' contains only two grey edges directed to the right (the edge that comes after $b_{C'_{\min}}$, and the edge between x and y). So C' could not be a knot of length greater than five, because such knots contain at least three grey edges directed to the right.

Suppose that C' is a knot of length five. Then this cycle has canonical labelling [1 4 2 5 3]. The edge with canonical label 1 is the edge $b_{C'_{\min}}$, and the edge with canonical label 2 is the edge x because these edges precede the two grey edges that are directed to the right. Let w be the edge with canonical label 4, y' be the edge with canonical label 5 (this is the edge y if y differs from $b_{C_{\min}}$, and z be the edge with canonical label 3. Now $x < z < w$ in $\pi \cdot \tau$, and since the transposition doesn't change the relative positions of these edges it must be that $x < z < w$ in π. So z must be an edge of D because otherwise C would be an oriented cycle. So C and D are interleaving 3-cycles. □

Lemma 3.4.7 *Let C and D be two unoriented cycles of $tG(\pi)$, such that C and D intersect, and C has the rightmost black edge of the two cycles. Then the transposition $\tau(D_{\min}, D_{\max}, C_{\max})$ converts C and D into a 1-cycle and an oriented cycle. Further, the oriented cycle is fully oriented unless C and D are 3-cycles that interleave.*

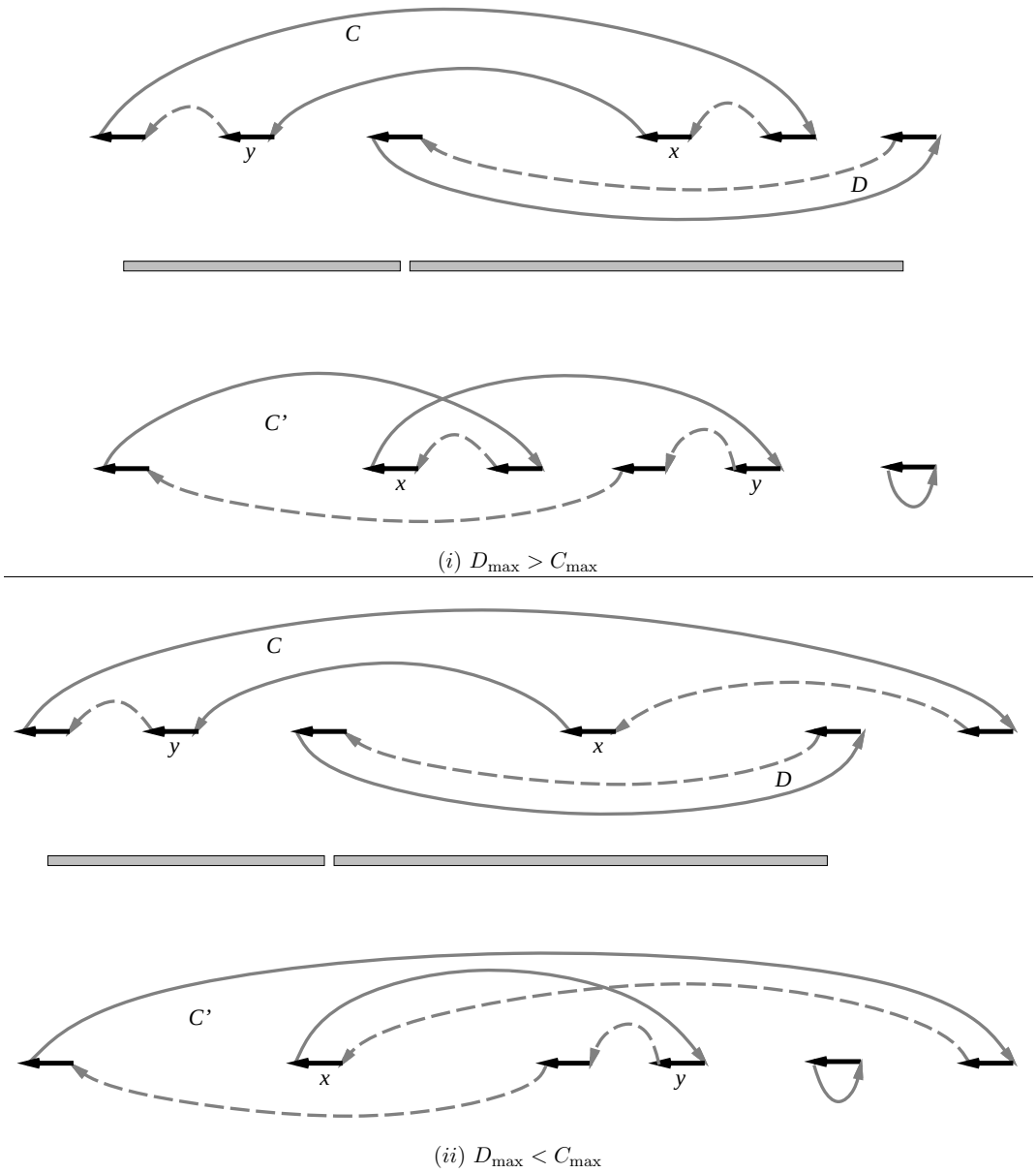

Figure 3.10: How the cycle graph changes.

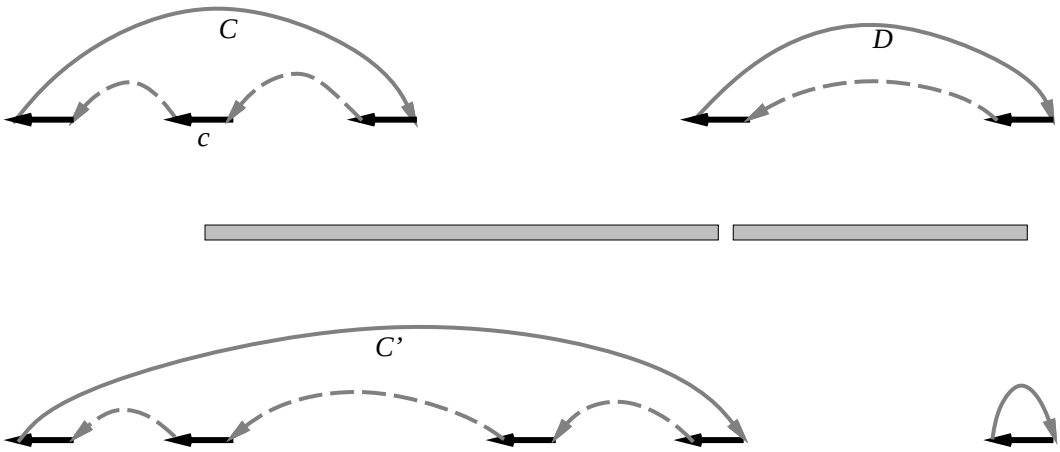

Figure 3.11: How $tG(\pi)$ changes as the result of the transposition.

Proof The proof of Lemma 3.4.6 may be easily transformed to prove this lemma. □

We now, as an aside, prove an interesting result about even length cycles.

Lemma 3.4.8 *If $tG(\pi)$ contains two unoriented even cycles C and D such that C and D intersect, then it is possible to apply a sequence of two 2-transpositions on π.*

Proof We can assume without loss of generality that C is the cycle that has the leftmost black edge. The transposition $\tau(C_{\min}, D_{\min}, D_{\max})$ is a 2-transposition since by Lemma 3.4.6 it transforms the two even cycles into two odd length cycles. By Lemma 3.4.6 there is a fully oriented cycle in the resulting cycle decomposition, so we can perform a further 2-transposition on this cycle. □

Now back to auxiliary lemmas that will help us obtain the (0,2,2)-sequence that we are seeking.

Lemma 3.4.9 *Let C and D be two unoriented cycles of $tG(\pi)$, such that C and D do not intersect, and C has the leftmost black edge of the two cycles. Let c be the position of a black edge of C. Then the transposition $\tau(c, D_{\min}, D_{\max})$ converts C and D into a 1-cycle and an unoriented cycle.*

Proof If $C_{\max} < D_{\min}$ then the transposition $\tau(c, D_{\min}, D_{\max})$ acts as shown in Figure 3.11. (Note that, of course, c could be $b_{C_{\min}}$ or $b_{C_{\max}}$, but these cases are similar to the case illustrated.) So the transposition converts C and D into a 1-cycle and a proper cycle C'. It is easy to verify that C' is an unoriented cycle, since it has only one grey edge that is directed to the right. The case when $C_{\max} > D_{\max}$ is similar. □

Lemma 3.4.10 *Let C and D be two unoriented cycles of $tG(\pi)$, such that C and D do not intersect, and D has the rightmost black edge of the two cycles. Let d be the*

position of a black edge of D. Then the transposition $\tau(C_{\min}, C_{\max}, d)$ converts C and D into a 1-cycle and an unoriented cycle.

Proof The proof of Lemma 3.4.9 may be easily transformed to prove this corollary. □

Lemma 3.4.11 *Let C and D be odd unoriented cycles of $tG(\pi)$ that do not interleave, and let C be the cycle with the leftmost black edge. Let $\tau(i, j, k)$ be a transposition that acts on three of four black edges $b_{C_{\min}}$, $b_{C_{\max}}$, $b_{D_{\min}}$, $b_{D_{\max}}$, such that either:*

$$\begin{array}{ll} (i) & i = C_{\min} \text{ and } j = C_{\max}, \text{ or} \\ (ii) & j = D_{\min} \text{ and } k = D_{\max}. \end{array}$$

Then $\tau(i, j, k)$ is a 0-transposition that converts C and D into a 1-cycle and a proper cycle that is unoriented if C and D do not intersect, and is fully oriented otherwise.

Proof This lemma is a simple consequence of Lemmas 3.4.6, 3.4.7, 3.4.9 and 3.4.10. □

We now have enough auxiliary results to obtain the (0,2,2)-sequence.

Lemma 3.4.12 *Let π be a permutation such that $tG(\pi)$ contains only unoriented odd length cycles. Further suppose that no cycles in $tG(\pi)$ interleave. Then there exists a $(0, 2, 2)$-sequence of transpositions on π.*

Proof Let C be the proper cycle in $tG(\pi)$ with the leftmost black edge. By Lemma 3.4.5 another cycle must intersect with $b_{C_{\min}}$ and $b_{C_{\max}}$. Let D be the cycle that intersects with $b_{C_{\min}}$ and $b_{C\max}$, and that, among all the cycles that intersect with these edges, contains the rightmost black edge to the left of $b_{C_{\max}}$. Let s be the second leftmost black edge of C, and let t be the edge $b_{C_{\max}}$. (Note $s \neq t$, because $l(C) \geq 3$).

If $p(\pi, s) < D_{\min}$ then there must be a cycle in $tG(\pi)$ that intersects with edges $b_{C_{\min}}$ and s of C (Lemma 3.4.5). Let E be a cycle that does intersect with these edges. Now $E_{\min} > C_{\min}$ because C is the leftmost cycle of $tG(\pi)$. Therefore, $C_{\min} < E_{\min} < s$ and $E_{\max} > s$ because E intersects with $b_{C_{\min}}$ and s. So either (a) $s < E_{\max} < D_{\min}$, or (b) $D_{\min} < E_{\max} < t$, or (c) $t < E_{\max} < D_{\max}$, or (d) $D_{\max} < E_{\max}$. These four different ways that cycles D and E can be related, are shown in the top halves of Figures 3.12 (a), (b), (c) and (d).

If $p(\pi, s) > D_{\min}$ then there must be a cycle that intersects with edges s and $b_{C_{\max}}$ of C (by Lemma 3.4.5). Let E be a cycle that does intersect with these edges. Now $E_{\min} > C_{\min}$ because C is the leftmost cycle of $tG(\pi)$. Note that D cannot contain black edges between s and t because otherwise it would interleave with C. Therefore $E_{\max} < t$ because otherwise E would contradict the choice of D (since E would intersect with C and contain a black edge to the left of $b_{C_{\max}}$ that was further to the right than any such edge of D). Therefore, $C_{\min} < E_{\min} < s$ and $E_{\max} > s$ because E intersects

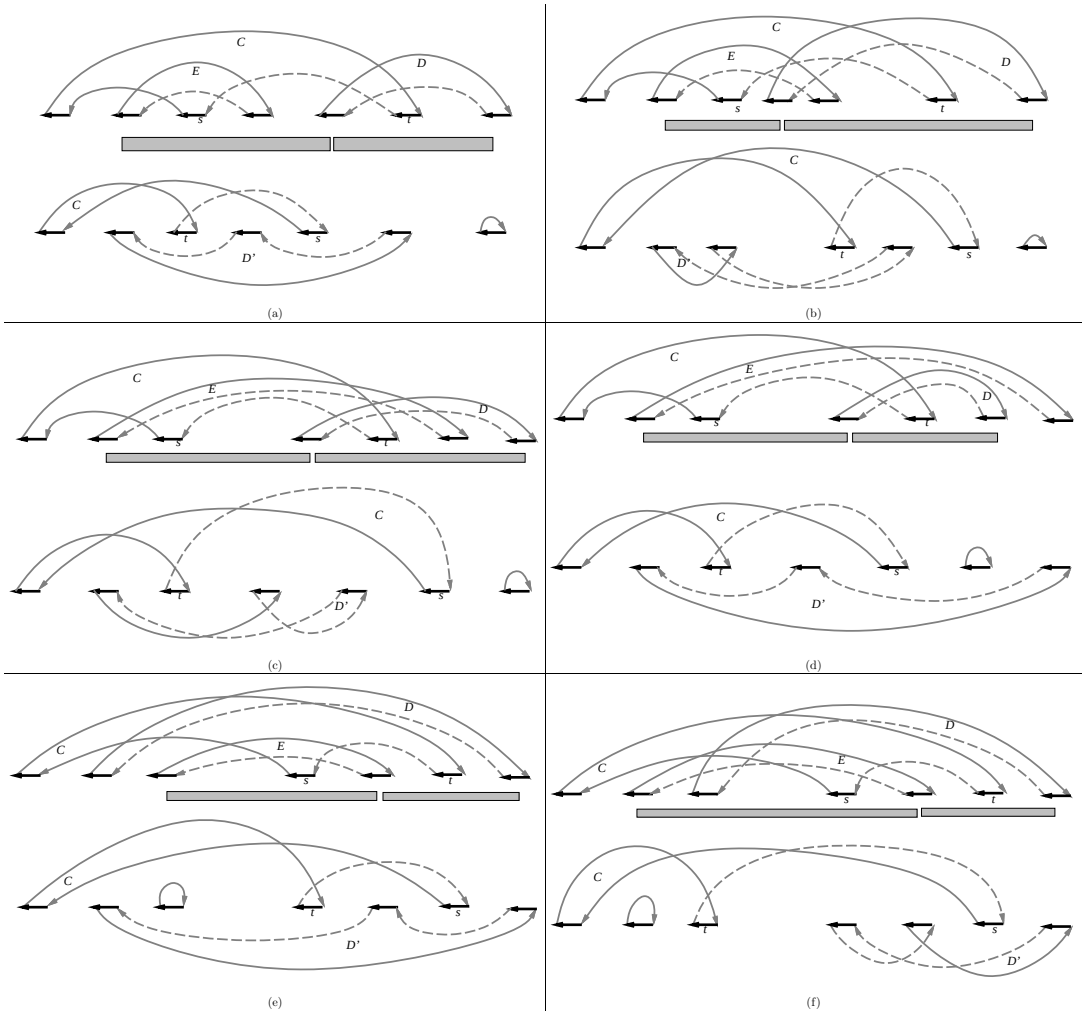

Figure 3.12: The first transposition of a $(0, 2, 2)$-sequence.

with s and $b_{C_{\max}}$. So either (e) $D_{\min} < E_{\min} < s$, or (f) $C_{\min} < E_{\min} < D_{\min}$. These two ways D and E can be related, are shown in at the top halves of Figures 3.12 (e) and (f).

In Figure 3.12 a 0-transposition τ (by Lemma 3.4.11) is shown for each case. In each case cycles D and E are converted into a 1-cycle and a long cycle D'. The transposition shown interleaves with cycle C, so by Lemma 3.4.3 C is a fully oriented cycle in the resulting cycle decomposition. In fact $(b_{C_{\min}}, t, s)$ is a fully oriented triple in the resulting cycle decomposition because all three paths connecting these edges are even in length. We now explain why in each case it is possible to find a sequence of two 2-transpositions on the resulting permutation.

(a) Cycles D and E do not intersect, so D' is unoriented by Lemma 3.4.11. Then $(b_{C_{\min}}, t, s)$ is a fully oriented triple of C that interleaves with D' so by Lemma 3.4.3 a

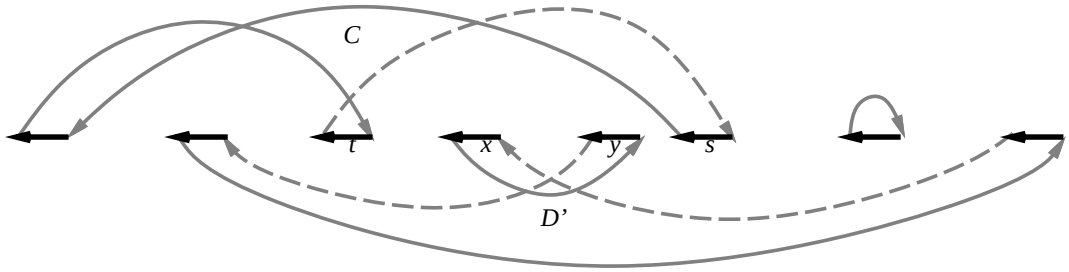

Figure 3.13: Special case of (d).

sequence of two 2-transpositions can be performed on the permutation.

(b) Cycles D and E intersect, so D' is fully oriented by Lemma 3.4.11. Then $(b_{C_{\min}}, t, s)$ is a fully oriented triple of C that does not interleave with any fully oriented triples of D', since $C_{\min} < D'_{\min}$, and $s > D'_{\max}$. So by Lemma 3.4.2 a sequence of two 2-transposition can be performed.

(c) Cycles D and E intersect, so D' is fully oriented by Lemma 3.4.11. Then $(b_{C_{\min}}, t, s)$ is a fully oriented triple of C that does not interleave with any fully oriented triples of D', since $C_{\min} < D'_{\min}$, and $s > D'_{\max}$. So by Lemma 3.4.2 a sequence of two 2-transposition can be performed.

(d) If D and E do not intersect then D' is unoriented, and this case is similar to (a). If D and E intersect then D' is fully oriented. Since D and E intersect, there must be an edge x of E such that $D_{\min} < x < D_{\max}$. Further $C_{\max} < x$ because otherwise C and E would be interleaved. In particular, let x be the leftmost edge of E to the right of $b_{C_{\max}}$. Let y be the edge that comes after x in D'. So $tG(\pi \cdot \tau)$ is as shown in Figure 3.13. Then either $(b_{D'_{\min}}, x, y)$ or $(x, y, b_{D'_{\max}})$ is a fully oriented triple that does not interleave with $(b_{C_{\min}}, t, s)$ which is a fully oriented triple of C. So by Lemma 3.4.2 a sequence of two 2-transposition can be performed.

(e) If D and E do not intersect then D' is unoriented, and this case is similar to (a). If D and E intersect then D' is fully oriented. Since D and E intersect, there must be an edge x of D such that $E_{\min} < x < E_{\max}$. Further $x < s$ because otherwise C and D would be interleaved. Let x be the rightmost edge of D to the left of s. Let y be the edge that precedes x in D'. So $tG(\pi \cdot \tau)$ is as shown in Figure 3.14. Then either $(b_{D'_{\min}}, y, x)$ or $(y, x, b_{D'_{\max}})$ is a fully oriented triple that does not interleave with $(b_{C_{\min}}, t, s)$ which is a fully oriented triple of C. So by Lemma 3.4.2 a sequence of two 2-transposition can be performed.

(f) Cycles D and E intersect, so D' is fully oriented by Lemma 3.4.11. Then $(b_{C_{\min}}, t, s)$ is a fully oriented triple of C that does not interleave with any fully oriented triples of D', since $C_{\min} < D'_{\min}$, and $t < D'_{\min}$. So by Lemma 3.4.2 a sequence of two 2-transposition can be performed. □

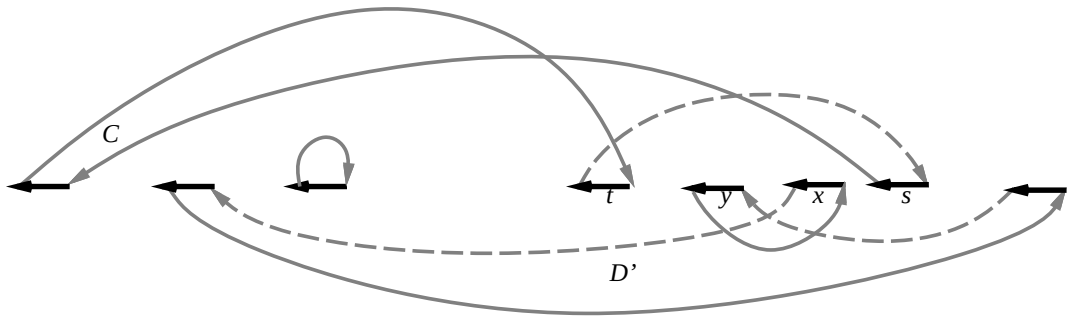

Figure 3.14: Special case of (e).

We now state formally the argument that proves that all the lemmas in this section allow us to describe a 3/2-approximation algorithm.

Lemma 3.4.13 *For every permutation, except the identity permutation, we can find either a 2-transposition or a* $(0, 2, 2)$*-sequence of transpositions on the permutation.*

Proof By the definition of fully oriented cycles, a 2-transposition exists if $tG(\pi)$ contains a fully oriented cycle. By Lemma 3.2.5 a 2-transposition also exists if $tG(\pi)$ contains an even cycle. If $tG(\pi)$ contains a knot then we can perform a $(0, 2, 2)$-sequence of transpositions, by Lemma 3.3.5.

If $tG(\pi)$ does not contain any fully oriented cycles, or even length cycles, or knots, then $tG(\pi)$ must contain only unoriented odd cycles. If $tG(\pi)$ contains two interleaving cycles then there is a $(0, 2, 2)$-sequence by Lemma 3.4.4. Otherwise $tG(\pi)$ contains only unoriented cycles, and no pair of these cycles are interleaving, so there is a $(0, 2, 2)$-sequence by Lemma 3.4.12. □

Theorem 3.4.1 *There is a 3/2-approximation algorithm for sorting by transpositions.*

Proof An entire sequence of transpositions that sorts a permutation can be generated by re-applying Lemma 3.4.13 again and again until the permutation has been sorted. This algorithm is illustrated in Figure 3.15. The sequence that is generated in this way contains at least two 2-transpositions for every 0-transposition, and no (-2)-transpositions. So the length of this sequence is at most 3/2 times the lower bound of Theorem 3.2.3. Hence we have described a 3/2-approximation algorithm for sorting by transpositions. □

We now compare Bafna and Pevzner's 3/2-approximation algorithm with the algorithm that has just been presented. In order to perform this comparison we briefly introduce Bafna and Pevzner's algorithm, which is summarised in Figure 3.16.

```
algorithm 3/2_Approximation(π: Permutation) is
  τ, τ1, τ2, τ3: Transposition;
begin
  while π ≠ ι loop
    if tG(π) contains a fully oriented cycle C then
      τ is a 2-transposition that acts on C;
      π:= π · τ;
    elsif tG(π) contains an even cycle then
      τ is a 2-transposition that acts on two even cycles (Lemma 3.2.5);
      π:= π · τ;
    elsif tG(π) contains a knot C then
      there is a (0, 2, 2)-sequence τ1, τ2, τ3 that acts on C (Lemma 3.3.5);
      π:= π · τ1 · τ2 · τ3;
    elsif tG(π) contains two interleaving cycles then
      there is a (0, 2, 2)-sequence τ1, τ2, τ3 that acts on C (Lemma 3.4.4);
      π:= π · τ1 · τ2 · τ3;
    else
      there is a (0, 2, 2)-sequence τ1, τ2, τ3 that acts on C (Lemma 3.4.12);
      π:= π · τ1 · τ2 · τ3;
    end if;
  end loop;
end 3/2_Approximation;
```

Figure 3.15: The new 3/2-approximation algorithm

```
algorithm BP_Approximation(π: Permutation) is
begin
  while π ≠ ι loop
    if tG(π) contains an oriented cycle C then
      apply a 2-transposition or a (0,2,2)-sequence on π;
    elsif tG(π) contains a long cycle then
      if tG(π) contains two interleaving cycles then
        apply a (0,2,2)-sequence on π (like Lemma 3.4.4);
      else
        apply a (0,2,2)-sequence on π (like Lemma 3.4.12);
      end if;
    else
      apply a sequence of two 2-transpositions;
    end if;
  end loop;
end BP_Approximation;
```

Figure 3.16: Bafna and Pevzner's 3/2-approximation algorithm

There are two ways in which their algorithm may apply a (0,2,2)-sequence where ours would apply a 2-transposition instead, so perhaps our algorithm will find shorter sequences of transpositions. Their algorithm will apply either a 2-transposition or a (0,2,2)-sequence on an oriented cycle, whereas our algorithm will always apply a 2-transposition if one exists, and will apply a (0,2,2)-sequence only if the cycle is a knot. Also, on occasion, their algorithm will apply (0,2,2)-sequences when the cycle graph contains 2-cycles, whereas our algorithm would apply a 2-transposition on the even cycles.

So we would expect our algorithm to sort permutations more efficiently. However as will be shown in Section 3.6 there are some heuristics for sorting by transpositions that we would expect to be even more efficient at solving instances of sorting by transpositions, even though the heuristics have no approximation guarantee. So, perhaps more efficient algorithms exist anyway.

Unfortunately, although we can detect fully oriented cycles easily, we know of no quicker means of finding a fully oriented triple in a fully oriented cycle than trying every triple of edges in the cycle. So finding a strongly oriented triple has $O(n^3)$ worst case time-complexity. The other steps inside the main loop of our algorithm have the following worst case time-complexities: building $tG(\pi)$ is $O(n)$, determining if $tG(\pi)$ contains a fully oriented cycle is $O(n)$, determining if $tG(\pi)$ contains interleaving cycles is $O(n^2)$, and performing a transposition is $O(n)$. Hence, the overall worst case time-complexity of the algorithm is $O(n^4)$, because we need to perform at most $O(n)$

transpositions. However we believe that it may be possible to improve the worst-case complexity of this algorithm, by finding a more clever algorithm for finding fully oriented triples in fully oriented cycles, or by improving the algorithmic analysis above.

Bafna and Pevzner claim that their algorithm is $O(n^2)$, but it seems to be $O(n^3)$ since before each transposition it must determine if any two cycles are interleaving, which is $O(n^2)$, and it must perform $O(n)$ transpositions. They are able to find transpositions on oriented cycles in $O(n)$ because they accept a $(0, 2, 2)$-sequence even when a 2-transposition exists.

We prove some results that are similar to results proved by Bafna and Pevzner. Our proof of Lemma 3.4.12 is simpler than their proof of a similar result, because it involves fewer cases, and does not involve arguments based on symmetries. To be fair though, their proof of a result like Lemma 3.4.4 based on strongly oriented cycles is shorter and perhaps simpler than our proof, though ours is more precise.

It is worth noting that Guyer, Heath and Vergara [GHV95] were "unable to" implement Bafna and Pevzner's 3/2-approximation algorithm because they could not resolve some "implementation details." Therefore a new 3/2-approximation algorithm that is simpler and is proved in a more precise manner is a useful contribution.

3.5 A tighter lower bound

3.5.1 Hurdles

In this section improved lower bounds for transposition distance are presented. These improved lower bounds are achieved by identifying subgraphs of the cycle graph that are difficult to sort. These difficult subgraphs are called *hurdles*, by analogy with the terminology of Hannenhalli and Pevzner for sorting by reversals. Below we define hurdles and then we describe some improved lower bounds for transposition distance based on counting hurdles.

Let us construct an *overlap graph* $tH(\pi)$ that contains a vertex for each cycle in $tG(\pi)$. Connect two vertices by an edge if their respective cycles intersect. Collections of cycles of $tG(\pi)$ that form components in $tH(\pi)$ will be of interest. We shall call a collection of cycles of $tG(\pi)$ a *component of* π if the collection consists of all the cycles in one component of $tH(\pi)$.

Example Let π be the permutation [8 10 9 11 1 6 5 4 3 2 7]. Then $tG(\pi)$ contains four cycles: $C_1 = (12, 1, 5)$, $C_2 = (11, 9, 7)$, $C_3 = (10, 8, 6)$, and $C_4 = (4, 2, 3)$, as shown in Figure 3.17. The components of π are $\{C_1\}$, $\{C_2, C_3\}$, and $\{C_4\}$. □

Let $\mathcal{C}$ be a component of $tG(\pi)$. Now suppose that we obtain a graph from $tG(\pi)$ by deleting all edges and vertices that are not part of $\mathcal{C}$, and then merging vertices that are next to each other in the resulting graph, as we draw it, but are not adjacent by a black edge. For example, if $\mathcal{C}$ is the component $\{C_1\}$ in Figure 3.17, we would merge vertices 8 and 11, and 7 and 1. This resulting graph can have its vertices relabelled

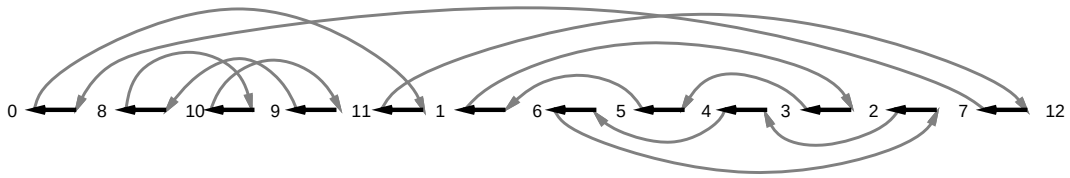

Figure 3.17: A cycle graph with three components.

so that the leftmost vertex is labelled 0, and each grey edge is directed from vertex i to vertex $i+1$. Reading vertices from left to right we obtain a permutation called the *component permutation* $\pi_{\mathcal{C}}$.

If $\mathcal{C}$ contains no even cycles and $\pi_{\mathcal{C}}$ cannot be sorted using only 2-transpositions then we say that $\mathcal{C}$ is a *hurdle*. If $\pi_{\mathcal{C}}$ can be sorted using only 2-transpositions then $\mathcal{C}$ is a *sortable component* (whether $\mathcal{C}$ contains an even cycle or not). Let $th(\pi)$ denote the number of hurdles in $tG(\pi)$. A transposition τ *clears* a hurdle $\mathcal{C}$ if no part of any cycle in $\mathcal{C}$ is part of a hurdle in $tG(\pi \cdot \tau)$.

Example For the example permutation above, $\pi_{\{C_1\}} = [2\ 1]$, $\pi_{\{C_2,C_3\}} = [5\ 4\ 3\ 2\ 1]$ and $\pi_{\{C_4\}} = [2\ 1]$. Components $\{C_1\}$ and $\{C_4\}$ are sortable components, but the component $\{C_2, C_3\}$ is a hurdle. □

It is possible to improve the lower bound of Theorem 3.2.3 by counting hurdles in $tG(\pi)$. Bafna and Pevzner's lower bound is based on assuming that each transposition in a minimal length sorting sequence is a 2-transposition. Note that a 2-transposition on a cycle in one component does not affect cycles in other components. Also, if a 2-transposition acts on two cycles then both cycles must be even in length. So if $tG(\pi)$ contains a hurdle then at least one transposition that is not a 2-transposition will be required to sort π. In fact we can improve the lower bound as shown below.

Theorem 3.5.1 *For every permutation* π, $td(\pi) \geq ((n+1-tc_{odd}(\pi))/2 + \lceil th(\pi)/2 \rceil$.

Proof All hurdles of $tG(\pi)$ must be cleared by at least one 0-transposition, or (-2)-transposition. A transposition can act on at most three hurdles, so a single transposition can clear at most three hurdles. If a transposition clears three hurdles then it must merge three cycles, from different components, together into one cycle. Such a transposition is a (-2)-transposition. So for every transposition that clears three hurdles, an extra 2-transposition is required. In fact hurdles can be cleared more efficiently by clearing two at a time. A 0-transposition on two cycles in different components may clear two hurdles. For example, the first transposition in Figure 3.18 is a 0-transposition that clears two hurdles. Clearing hurdles in pairs with 0-transpositions in this way requires $\lceil th(\pi)/2 \rceil$ 0-transpositions to clear all the hurdles. □

A problem with this bound based on hurdles is that detecting hurdles may not be any easier than determining the transposition distance of a given permutation.

However, there are some hurdles that can be detected easily. These hurdles consist only of odd length cycles that are not fully oriented. Let $th_d(\pi)$ represent the number of *detectable* hurdles like this in π. The lower bound that we obtain using this count of hurdles is as follows:

Theorem 3.5.2 *For every permutation π, $td(\pi) \geq (n + 1 - tc_{odd}(\pi))/2 + \lceil th_d(\pi)/2 \rceil$.*

Proof Clearly $th_d(\pi) \leq th(\pi)$. The theorem then follows from Theorem 3.5.1. □

Let $tl(\pi) = (n + 1 - tc_{odd}(\pi))/2 + \lceil th_d(\pi)/2 \rceil$. In Section 1.1, we observed that $td(\pi) = td(\pi^{-1})$. Let π = [7 5 3 1 8 6 4 2], so that π^{-1} = [4 8 3 7 2 6 1 5]. Now $tl(\pi) = 3$, but $tl(\pi^{-1}) = 4$, and given the above identity it must be that $td(\pi) \geq 4$. This example inspires the following lower bound:

Theorem 3.5.3 *For all permutations π, $td(\pi) \geq \max\{tl(\pi), tl(\pi^{-1})\}$.*

Proof This theorem can be derived easily from Theorem 3.5.2 and the identity $td(\pi) = td(\pi^{-1})$. □

We have found that this lower bound is useful to speed up an implemented branch and bound algorithm for sorting by transpositions. The bound can also be used to obtain permutations that have transposition distances arbitrarily far away from the lower bound of Theorem 3.2.3, as we show in the next section.

3.5.2 How tight are the lower bounds?

We now show that Bafna and Pevzner's lower bound for transposition distance (Theorem 3.2.3) can be arbitrarily far away from the exact transposition distance. We need some extra definitions to describe permutations with this property.

If π is a permutation of the set $\{1, \ldots, n\}$ and k is an integer then we define $\pi + k$ to be the permutation of the set $\{k+1, \ldots, k+n\}$ obtained by adding k to each of the elements of π. In particular this means that $(\pi + k)(i) = \pi(i) + k$, for $1 \leq i \leq n$.

If π is a permutation of the set $\{1, \ldots, n\}$ and ϕ is a permutation of the set $\{n + 1, \ldots, m\}$ then we can define $\pi +\!\!+ \phi$ to be the permutation of the set $\{1, \ldots, m\}$ obtained by appending ϕ to π. For example if π = [2 4 3 1] and ϕ = [6 5] then $\pi +\!\!+ \phi$ = [2 4 3 1 6 5].

We now build a permutation κ_k that has transposition distance arbitrarily distant from the lower bound of Theorem 3.2.3. Let κ = [4 3 2 1]. Define $\kappa_1 = \kappa$, and $\kappa_k = \kappa_{k-1} +\!\!+ [5(k-1)] +\!\!+ (\kappa + 5(k-1))$, for $k \geq 2$. So, for example, κ_3 = [4 3 2 1 5 9 8 7 6 10 14 13 12 11]. We prove that κ_k has the property that we want.

Theorem 3.5.4 *The transposition distance of κ_k is given by the formula*

$$td(\kappa_k) = 2k + \left\lceil \frac{k}{2} \right\rceil = \frac{n + 1 - tc_{odd}(\kappa_k)}{2} + \left\lceil \frac{k}{2} \right\rceil .$$

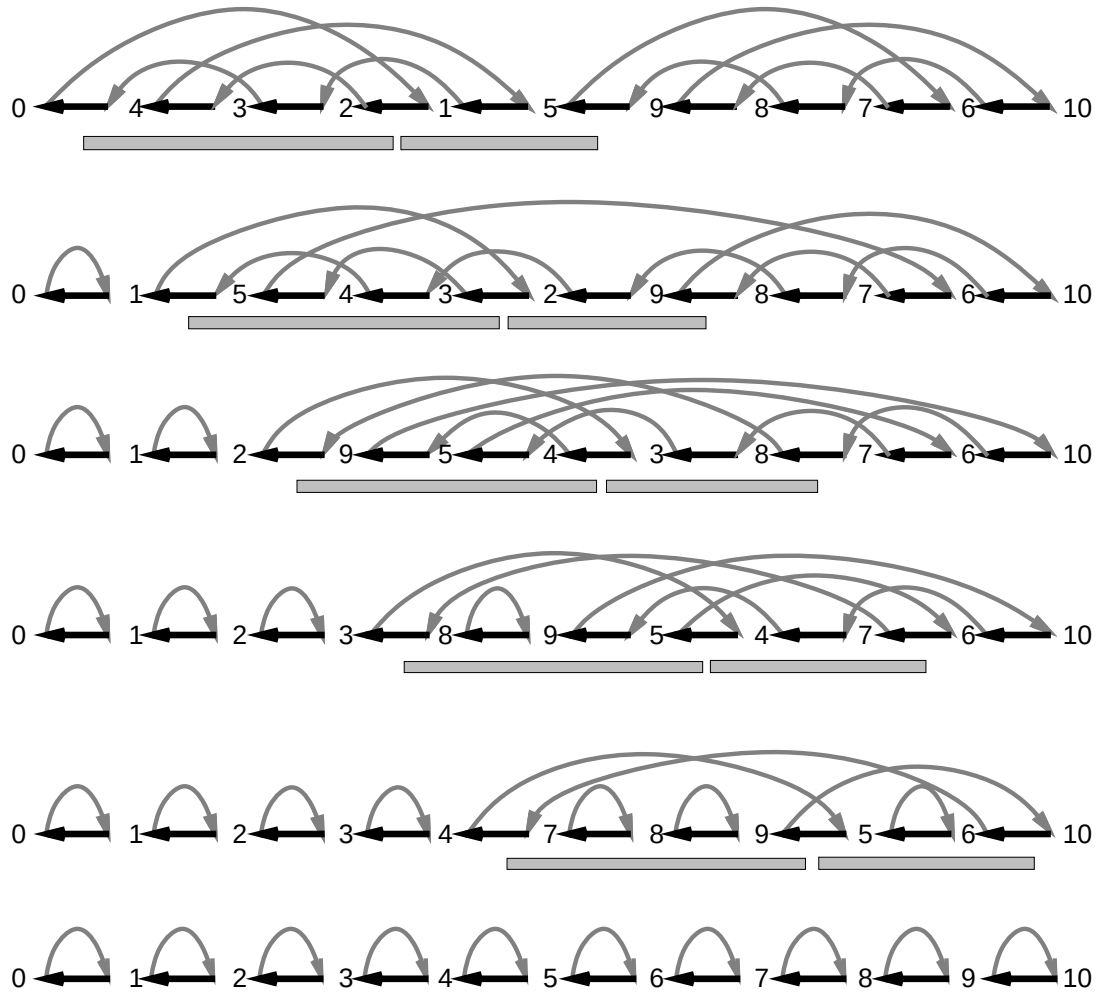

Figure 3.18: An optimal length sorting of κ_2.

Proof The cycle graph of κ_k consists of k knots of size five that do not intersect with each other. Therefore, by Theorem 3.5.1, the transposition distance of κ_k is at least $(n+1-tc_{odd}(\kappa_k))/2 + \lceil k/2 \rceil$.

It is possible to sort κ_k by sorting pairs of knots together as shown in Figure 3.18. If k is odd, then one knot must also be sorted on its own. The sequence of transpositions that we have just described has the length given by the above formula. □

In fact we can generalise the above theorem as follows. Let π be a permutation that contains only odd length cycles and for which $td(\pi) \geq (n+1-tc_{odd}(\pi))/2 + 1$. Let $\pi_1 = \pi$, and $\pi_k = \pi_{k-1} +\!\!+ [(k-1)(n+1)] +\!\!+ (\pi + (k-1)(n+1))$. Then $td(\pi_k) \geq (n+1-tc(\pi_k))/2 + \lceil k/2 \rceil$.

So we have shown that κ_k is an example of a family of permutations whose transpo-

sition distance can be arbitrarily far from the lower bound of Theorem 3.2.3. However it appears that such permutations are rare, and the bound of Theorem 3.2.3 is a good bound in practice (Section 3.6).

Note also that $td(\kappa_1) = 3$, but the lower bound for κ_1 is only 2. So any approximation algorithm that has its approximation ratio based only on the lower bound of Theorem 3.2.3 cannot have an approximation ratio better than 3/2.

Of course, the lower bound of Theorem 3.5.1 based on hurdles is exact for κ_k. So, it is natural to ask if are there any permutations that can be arbitrarily distant from this lower bound? In fact this question remains open, however we conjecture that the answer is yes for reasons explained below.

Consider the permutation $\pi_1 = [3\ 2\ 1\ 6\ 5\ 4\ 9\ 8\ 7\ 12\ 11\ 10]$. The lower bound based on hurdles (Theorem 3.5.1) is 5 transpositions, but $td(\pi_1) = 6$. (The exact distance of this permutation was calculated using a branch and bound algorithm described in section 3.6.) Note that the overlap graph for this permutation $tH(\pi)$ contains only one component, but that component is not a hurdle or a sortable component. This permutation demonstrates a way in which the hurdles lower bound can underestimate the transposition distance, because of the way it deals with components containing even cycles.

Consider also the permutation $\pi_2 = [5\ 4\ 3\ 8\ 7\ 6\ 11\ 10\ 9\ 14\ 13\ 12\ 17\ 16\ 15\ 2\ 1]$. The cycle graph, $tG(\pi_2)$ contains six 3-cycles that are all unoriented, such that no pair of the cycles are interleaving. The overlap graph, $tH(\pi_2)$ contains only one component, that is a hurdle. In can be shown that $td(\pi_2) = 9$ (using the branch and bound algorithm) but the lower bound of Theorem 3.5.1 is 7. So π_2 is a permutation distance 2 greater than the lower bound based on hurdles.

Note also, that the permutation π_2 demonstrates a manner in which hurdles of the transposition cycle graph are unlike hurdles of the reversal cycle graph, because the hurdle in the transposition graph requires more than one transposition in order to be cleared, whereas hurdles of the reversal cycle graph require only one reversal to be cleared (with some special permutations, fortresses, requiring an extra reversal).

We conjecture that it may be possible to build permutations that have transposition distances arbitrarily far from the lower bound of Theorem 3.5.1 by concatenating permutations like π_1 and π_2 together, or building larger permutations that have structures similar to those of π_1 and π_2.

3.5.3 The transposition diameter

The *transposition diameter* $tD(n)$ is the maximum value of $td(\pi)$ taken over all permutations π of length n. Bafna and Pevzner proved that $tD(n) \leq 3n/4$, since their 3/2-approximation algorithm sorts any permutation using at most $3/4.(n+1-c_{odd}(\pi))$ transpositions. They also discovered that, for $3 \leq n \leq 10$, $tD(n) = \lfloor n/2 \rfloor + 1$, and that this value is achieved by the *reverse permutation*, R_n. They also stated that, for all n, $td(R_n) \leq \lfloor n/2 \rfloor + 1$. We prove that $td(R_n) = \lfloor n/2 \rfloor + 1$, and make a conjecture about

transposition diameter.

For odd $n \geq 3$ we define *the broken reverse permutation*, BR_n, to be the permutation that results from applying the transposition $\tau(1, (n+1)/2, (n+1)/2+2)$ to R_n. If $n = 3$ then $BR_n = [2\ 1\ 3]$, and for larger values of n then $BR_n = [\lceil n/2 \rceil\ \lfloor n/2 \rfloor\ n\ \ldots\ \lceil n/2 \rceil + 1\ \lfloor n/2 \rfloor - 1\ \ldots\ 1]$.

Lemma 3.5.1 *For odd n, $td(BR_n) \leq \lfloor n/2 \rfloor$.*

Proof We prove this lemma by induction. The base case is when $n = 3$. Clearly one transposition is enough to sort $[2\ 1\ 3]$.

Now suppose that the lemma is true for all $n \leq 2k+1$. Let $n = 2k+3$. Now apply the transposition $\tau(2, (n+1)/2+1, (n+1)/2+3)$. If $n = 5$, we obtain the permutation $\pi = [3\ 4\ 1\ 2\ 5]$, and otherwise we obtain the permutation $\pi = [\lceil n/2 \rceil\ \lceil n/2 \rceil + 1\ \lfloor n/2 \rfloor - 1\ \lfloor n/2 \rfloor\ n\ \ldots\ \lceil n/2 \rceil + 2\ \lfloor n/2 \rfloor - 2\ \ldots\ 1]$. Now $gl(\pi) = gl(BR_{2k+1})$, so by the inductive hypothesis, and Theorem 3.2.2, $td(\pi) \leq k$. Therefore $td(BR_{2k+3}) \leq k+1$, and we have proved the lemma by induction. □

We now prove the following theorem, which is stated without proof by Bafna and Pevzner.

Theorem 3.5.5 *$td(R_n) \leq \lfloor n/2 \rfloor + 1$, for all $n \geq 2$.*

Proof If n is even, we make the transposition $\tau(1, 2, n+1)$ to move the element n to its correct position. We then treat what remains as R_{n-1}. The fact that, for n even, $\lfloor n/2 \rfloor + 1 = 1 + (\lfloor (n-1)/2 \rfloor + 1)$ means that the truth of the result for n even will follow if we establish its validity for n odd.

For n odd, perform the transposition $\tau(1, (n+1)/2, (n+1)/2+2)$. The resulting permutation is BR_n. By Lemma 3.5.1 we may sort this permutation with $\lfloor n/2 \rfloor$ transposition. So R_n can be sorted using $\lfloor n/2 \rfloor + 1$ transpositions. □

In fact, for n odd we can prescribe a sequence of transpositions, $\tau_1, \ldots, \tau_r$ (where $r = (n+1)/2$), that sorts R_n as follows:

$$\tau_i = \begin{cases} \tau(i, i + (n-1)/2, i + (n+3)/2), & i = 1, \ldots, r-1 \\ \tau(1, (n+1)/2, n), & i = r. \end{cases}$$

In Figure 3.19 this sequence of transpositions is applied to R_{11}.

Theorem 3.5.6 *$td(R_n) = \lfloor n/2 \rfloor + 1$, for all $n \geq 2$.*

Proof By Theorem 3.5.5 $td(R_n) \leq \lfloor n/2 \rfloor + 1$.

If $n \equiv 3 (\text{mod } 4)$ then $tG(R_n)$ consists of only 2 even length cycles. So by Theorem 3.2.3 it is impossible to sort R_n using fewer than $\lfloor n/2 \rfloor + 1$ transpositions.

If $n \equiv 1 (\text{mod } 4)$ then $tG(R_n)$ consists of only 2 odd length cycles. Both cycles are unoriented, so again it is impossible to sort R_n using fewer than $\lfloor n/2 \rfloor + 1$ transpositions.

11 10 9 8 7 6 5 4 3 2 1
6 5 11 10 9 8 7 4 3 2 1
6 7 4 5 11 10 9 8 3 2 1
6 7 8 3 4 5 11 10 9 2 1
6 7 8 9 2 3 4 5 11 10 1
6 7 8 9 10 1 2 3 4 5 11
1 2 3 4 5 6 7 8 9 10 11

Figure 3.19: A sequence of transpositions that sorts R_{11}.

However, when n is even, $tG(R_n)$ consists of one oriented odd length cycle. So in this case the lower bound of Theorem 3.2.3 is only $\lfloor n/2 \rfloor$ transpositions. However, the cycle in $tG(R_n)$ has canonical labelling $[1\ n\ n-2\ \ldots\ 2\ n+1\ n-1\ \ldots\ 3]$. Therefore, by Lemma 3.3.7, the cycle is not super oriented since it contains only two grey edges that are directed to the right. So it is impossible to find a sequence of two 2-transpositions on R_n, and therefore it is impossible to sort R_n using fewer than $\lfloor n/2 \rfloor + 1$ transpositions. □

As a result of Theorem 3.5.6, $\lfloor n/2 \rfloor + 1 \leq tD(n) \leq 3n/4$. In fact, we conjecture that the reverse permutation achieves the transposition diameter for all n.

Conjecture 3.5.1 *For all n, $tD(n) = \lfloor n/2 \rfloor + 1$.*

We make this conjecture because, on the one hand, the more cycles there are in $tG(\pi)$ the fewer 2-transpositions are needed to sort π by the lower bound of Theorem 3.2.3, and on the other hand, the fewer cycles there are in $tG(\pi)$, then the longer these cycles must be and so the more likely it is that the cycles are fully oriented, or are knots, and hence the more likely it is that a sequence of many 2-transpositions can be applied to π. We believe these two factors together make it impossible to have $td(\pi) > \lfloor n/2 \rfloor + 1$. Some evidence supporting this conjecture is presented in Section 3.6.

3.6 Solving instances of sorting by transpositions in practice

In order to investigate how well instances of sorting by transpositions can be solved in practice, several approximation algorithms and an exact algorithm were implemented and tested. The various algorithms use heuristics based on results of the previous sections. We describe the algorithms in Section 3.6.1. The algorithms were tested with

various kinds of permutation that are described in Section 3.6.2. Tables of raw data describing each algorithm's performance are presented in Section 3.6.3. In Section 3.6.4 we analyse these tables, and try to explain the observed behaviour. We conclude that most instances of sorting by transpositions can be solved optimally, at least for the range of sizes covered by our experiments.

3.6.1 The algorithms

The different algorithms that were implemented are described below. Each description begins with the name of the algorithm, followed by an explanation of the heuristics the algorithm uses and a statement about the complexity of the algorithm.

`lower_bound`

This algorithm is not an approximation algorithm or exact algorithm for sorting by transpositions. Instead, it simply calculates the lower bound for transposition distance based on detectable hurdles and the inverse permutation (Theorem 3.5.3). To calculate this bound the algorithm must build the cycle graph $tG(\pi)$ and the overlap graph $tH(\pi)$. Counting the number of cycles in $tG(\pi)$ can be performed in $O(n)$ steps, but building $tH(\pi)$, and identifying its components takes $O(n^2)$ steps. So this algorithm is $O(n^2)$.

`approx`

This algorithm uses several heuristics, based on the results of the previous sections, to find a sequence of transpositions that sorts a given permutation. The algorithm has no approximation guarantee, but in practice we expect it to perform better than the 3/2-approximation algorithm described in Section 3.4, for reasons explained below.

The algorithm is described in Figure 3.20. The first step of the algorithm checks if the permutation is transposition equivalent to the reverse permutation and sorts the permutation optimally if it is. Otherwise a sequence of steps is followed that is similar to the sequence of steps used in the 3/2-approximation algorithm.

If $tG(\pi)$ contains a strongly oriented cycle then a 2-transposition τ is applied on a shortest strongly oriented cycle C, splitting C into three cycles C_1, C_2 and C_3. Note that, in particular, τ is chosen so that the size of C_1 is maximised, and then so that the size of C_2 is maximised. This strategy was implemented because, by Lemma 3.3.3, larger cycles are more likely to be strongly oriented than smaller cycles. Therefore when we have a choice between applying a 2-transposition on a small cycle or a 2-transposition on a large cycle, we apply the transposition on the small cycle, because it is less likely that the small cycle should have been fully oriented, and it is more likely that the large cycle will remain fully oriented in the cycle graph of the resulting permutation. We choose the transposition that maximises C_1 and then C_2 because we are attempting to maximise the probability that these cycles are fully oriented.

```
algorithm approx(π: Permutation) is
  τ: Transposition;
  ts: array of Transposition;
  i: Integer;
begin
  i:= 0;
  while π ≠ ι loop
    i:= i+1;
    if gl(π) = R_|gl(π)| then
      sort π optimally from here (Theorem 3.5.6);
      exit loop;
    else
      if tG(π) contains a fully oriented cycle then
        τ:= a 2-transposition that acts on a smallest fully oriented cycle;
      elsif tG(π) contains interleaving even cycles then
        τ:= a 2-transposition that acts on two interleaving even cycles (Lemma 3.4.8);
      elsif tG(π) contains even cycles then
        τ:= a 2-transposition that acts on two even cycles (Lemma 3.2.5);
      elsif tG(π) contains a knot then
        τ:= the 0-transposition from the (0, 2, 2)-sequence of Lemma 3.3.5;
      elsif tG(π) contains two interleaving cycles then
        τ:= the 0-transposition from the (0, 2, 2)-sequence of Lemma 3.4.4;
      else
        τ:= the 0-transposition from the (0, 2, 2)-sequence of Lemma 3.4.12;
      end if;
      π:= π · τ;
      ts(i):= τ;
    end if;
  end loop;
  return i, ts;
end approx;
```

Figure 3.20: The algorithm: `approx`

The algorithm does not achieve the 3/2-approximation bound because we have no guarantee that it applies two 2-transpositions after every 0-transposition. But the strategy for selecting 2-transpositions is designed with the hope that it will find a long sequence of 2-transpositions, if one exists, and so we hope that the algorithm will find at least two 2-transpositions after every 0-transposition. The algorithm performs so well in practice that implementing the 3/2-approximation algorithm did not seem to be worthwhile.

The algorithm has $O(n^4)$ worst-case time complexity by the same reasons that the 3/2-approximation algorithm has this complexity. In practice, though, the runtime of the algorithm does not appear to increase as rapidly as this.

`alt_approx`

This algorithm uses the same strategy as `approx` except that when a fully oriented cycle exists, the algorithm favours 2-transpositions on odd length cycles before 2-transpositions on even length cycles.

The reason for this variation is that we know, by Lemma 3.2.5, that if $tG(\pi)$ contains even cycles then we can perform a 2-transposition on π, whether the even cycles are fully oriented or not. So the probability of being able to apply a 2-transposition on an odd cycle could be thought of as being much smaller than the probability of applying a 2-transposition on even cycles.

Another way of thinking about this algorithm is that it keeps 2-transpositions on even cycles in reserve until it gets stuck trying to apply 2-transpositions on odd cycles.

The time complexity of this algorithm is $O(n^4)$ just like `approx`.

`tlis`

This algorithm is based on an algorithm described by Guyer, Heath and Vergara [GHV95]. The algorithm repeatedly applies a transposition τ to π such that $\pi \cdot \tau$ has the longest possible increasing subsequence. This algorithm is described in Figure 3.21. A modification we have made to this algorithm (that is not indicated in the figure) is that the transposition is chosen so that it maximises the length of the smaller block moved, over all transpositions that increase the longest increasing sequence maximally. Without this modification, if the length of the longest increasing subsequence can be increased by only one element then the algorithm could repeatedly move only a single element in the permutation until the permutation is sorted, which is likely to be a very inefficient method of sorting the permutation.

There are $O(n^3)$ transpositions that need to be considered on each iteration of the loop, and for each transposition the length of the longest increasing subsequence is calculated in $O(n \log n)$ time. The loop is iterated $O(n)$ times, so the overall time-complexity of the algorithm is $O(n^5 \log n)$.

```
algorithm tlis(π: Permutation) is

  algorithm lis(π: Permutation) is
  begin
    return the length of a longest increasing subsequence of π;
  end lis;

  τ: Transposition;
  ts: array of Transposition;
  i: Integer;
begin
  i:= 0;
  while π ≠ ι loop
    τ:= a transposition such that lis(π · τ) is maximised;
    π:= π · τ;
    i:= i+1;
    ts(i):= τ;
  end loop;
  return i, ts;
end tlis;
```

Figure 3.21: The algorithm: `tlis`

```
algorithm random(π: Permutation) is
  τ: Transposition;
  ts: array of Transposition;
  i: Integer;
begin
  i:= 0;
  while π ≠ ι loop
    i:= i+1;
    if π admits a 2-transposition then
      τ:= a 2-transposition chosen uniformly at random;
    elsif tG(π) contains a knot then
      τ:= the 0-transposition from the (0, 2, 2)-sequence of Lemma 3.3.5;
    elsif tG(π) contains two interleaving cycles then
      τ:= the 0-transposition from the (0, 2, 2)-sequence of Lemma 3.4.4;
    else
      τ:= the 0-transposition from the (0, 2, 2)-sequence of Lemma 3.4.12;
    end if;
    π:= π · τ;
    ts(i):= τ;
  end loop;
  return i, ts;
end random;
```

Figure 3.22: The algorithm: `random`

`random`

This algorithm repeatedly selects a 2-transposition uniformly at random from among all the 2-transpositions that are possible on the permutation until the permutation is sorted. If no 2-transposition is possible then a 0-transposition is applied using the same strategy that `approx` uses. The algorithm is summarised in Figure 3.22. This algorithm has $O(n^4)$ complexity because in the worst case counting the total number of 2-transpositions on π is an $O(n^3)$ operation, and we may need to perform $O(n)$ transpositions. Note that we have optimised the way we count 2-transpositions so that, in practice, this algorithm appears to have a better runtime complexity.

`rpt_random`

This algorithm repeatedly applies the algorithm `random` until an exact solution is found, or a specified amount of processing time has elapsed, in which case the algorithm returns the shortest sequence of transpositions that was obtained within the time-limit. The algorithm uses `alt_approx` to produce the initial sorting sequence. Of course, an

application of `random` is abandoned as soon as it cannot improve upon the best sorting sequence found so far.

`branch`

This algorithm uses branch-and-bound search to find the exact transposition distance of a given permutation. This algorithm is summarised in Figure 3.23. The lower bound used by the algorithm is the bound based on detectable hurdles and the inverse permutation (Theorem 3.5.3). The search is bounded by a specified time limit. Of course, there are permutations for which this algorithm will not be able to find the exact solution in the specified time. If time runs out during the search then the algorithm returns the shortest sequence of transpositions that was obtained during the time-limit.

`best`

This algorithm tries all the algorithms described above (except for `lower_bound`) and selects the shortest sorting sequence found. Note that this algorithm also tries a few variants of the algorithm `branch` that can sometimes establish the exact transposition distance of a permutation faster than `branch`. However in the experiments these variant algorithms could only determine the exact transposition distance of a handful of permutations that the other algorithms could not establish. Furthermore, these variant algorithms could not establish the transposition distance of many permutations that the other algorithms could. Therefore we do not include tables for these algorithms, or describe these algorithms fully, though we do describe the variants a little in Section 3.6.4.

3.6.2 Permutations

The algorithms were tested with three different kinds of permutations. Of course, we test the algorithms with random permutations (*random permutations*). We also test the algorithms with two kinds of permutations that have cycle graphs containing only 3-cycles. We are interested in these permutations because they are important when we consider the concept of *transposition tightness* which is defined by analogy with reversal tightness (Section 2.5). We say that a permutation π is *transposition tight* if it is possible to sort π by applying a sequence of transpositions to π such that each transposition removes three breakpoints. If a permutation is transposition tight then it must have a cycle graph containing only 3-cycles. We generate permutations containing only 3-cycles in their cycle graph that may or may not be transposition tight (*3-cycle permutations*), as well as permutations that are transposition tight (*transposition tight permutations*).

We describe the three kinds permutations in more detail below.

```
algorithm branch(π: Permutation) is
  bound: Integer
  current, best: array of Transposition;

  algorithm search(π, depth) is
  begin
    if π = ι then
      if depth < bound then
        bound:= depth;
        best:= current;
      end if;
    else
      for each transposition τ in order of decreasing Δtc_odd(π, τ) loop
        if lower_bound(π · τ) + depth + 1 < bound then
          current(depth + 1):= τ;
          search(π · τ, depth + 1);
        end if;
      end loop;
    end if;
  end search;

begin
  bound, best:= alt_approx(π);
  search(π, 0);
  return bound, best;
end branch;
```

Figure 3.23: The algorithm: `branch`

random permutations

These are generated uniformly at random from among all strip free permutations of a given length.

3-cycle permutations

These are generated uniformly at random from among all permutations of a given length that have cycle graphs containing only 3-cycles such that no 3-cycle contains both b_i and b_{i+1} for any i. Note that these permutations can only be generated if $n \equiv 2 \pmod 3$.

We generate these permutations by randomly partitioning $\{b_1, \ldots, b_{n+1}\}$ into sets of size three, such that no set contains b_i and b_{i+1} for any i. Each of these sets represents a potential 3-cycle in the cycle graph. We then randomly decide if each potential 3-cycle is oriented or unoriented. Having made these decisions we can add grey edges to the black edges, so as to obtain a candidate cycle graph. We then attempt to label all the vertices in this graph by starting from vertex 0 and following grey edges in the graph (since each grey edges is directed from vertex i to vertex $i + 1$). (Of course, we can identify vertex 0 to begin with because it is adjacent to b_1.) If we are able to label all the vertices in the graph, then a permutation π exists with this candidate cycle graph and it is easy to read π from the graph. If we cannot label all the vertices then we redo the entire process until it works.

Note that if a 3-cycle contains black edges b_i and b_{i+1}, for some i, then that 3-cycle must be fully oriented. Such a 3-cycle remains fully oriented until a 2-transposition is applied on the cycle, or a 0-transposition or (-2)-transposition is applied that acts on at least one of the edges of the cycle. Furthermore, if we apply a 2-transposition on such a cycle, the overlapping pattern of the other cycles in the cycle graph is unaffected. We conjecture that if $tG(\pi)$ contains such a cycle, we may always obtain an optimal length sequence of transpositions by initially applying a 2-transposition on this cycle. In fact, this conjecture holds when the permutation is transposition tight. Therefore we believe that these 3-cycles are not very interesting, and so we generate permutations that do not contain these cycles.

transposition tight permutations

We generate these permutations by randomly partitioning $\{b_1, \ldots, b_{n+1}\}$ into sets of size three, such that no set contains b_i and b_{i+1} for any i. Each of these partitions represents a transposition that acts on the three black edges in the set. We then apply a sequence of these $(n + 1)/3$ transpositions, in random order, to the identity permutation. The resulting permutation has a cycle graph that contains only 3-cycles such that no 3-cycle contains b_i and b_{i+1}, for some i. Furthermore, we know that the permutation is transposition tight. (This is in contrast to 3-cycle permutations which may or may not be transposition tight.) Note, however, that we cannot claim that a

permutation generated in this way is chosen uniformly at random from the set of all such permutations. Note also that these permutations can again only be generated if $n \equiv 2 \pmod 3$.

3.6.3 How the algorithms performed

Each algorithm was tested with each kind of permutation against 100 permutations of sizes 8, 17, 26, 53, 107, 431 and 1727. (Remember that $n+1$ must be divisible by three in order for it to be possible to decompose a cycle graph into 3-cycles only.) Some tests were not performed because the space or time requirements of the algorithms were too great.

The algorithm `branch` was allowed to search for 15 minutes of processing time. The algorithm `rpt_random` was allowed to search for 5 minutes of processing time. All the algorithms were written in Ada 95, and compiled with the Gnat Ada 95 compiler. All the tests were performed on a Sun SPARCstation-20.

The performance of the various algorithms is presented in tables later in this section. Recall that, for a given permutation, each algorithm (except `lower_bound`) returns a sequence of transpositions that sorts the permutation (a sorting sequence). We now describe the column headings of the tables. Of course, for `lower_bound` the tables present figures derived from values calculated as the lower bound, rather than from the length of sorting sequence obtained.

Average

The average length of sorting sequence found by the algorithm over the 100 instances.

Min

The shortest length of sorting sequence found by the algorithm.

Max

The longest length of sorting sequence found by the algorithm.

Av gap

The average difference between the length of the sorting sequence found by the algorithm and the lower bound of Theorem 3.5.3 over the 100 instances.

Max gap

The biggest difference between the length of the sorting sequence found by the algorithm and the lower bound of Theorem 3.5.3.

Match lb

The number of times the length of the sorting sequence found by the algorithm was equal to the lower bound of Theorem 3.5.3.

Exact

The number of times the length the sorting sequence found by the algorithm was equal the exact transposition distance of the permutation (when the exact transposition distance of the permutation is known).

Av Time

The average time in seconds required by the algorithm. Note that for `branch` and `rpt_random` this time excludes the time required to establish initial upper bound using `alt_approx`.

For the algorithm `branch` there are two more table headings, they are:

Finished

The number of times the algorithm completed its search inside the time allowed.

Longest

The longest time (in seconds) required by the algorithm to complete its search, when the algorithm did complete its search.

n	Average	Min	Max
8	3.96	3	4
17	8.43	7	9
26	12.75	12	13
53	26.18	25	27
107	53.07	52	54
431	214.67	213	216
1727	862.45	860	864

Table 3.1: random permutations, `lower_bound`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	4.20	3	5	0.24	1	76	98	0.0
17	8.75	7	10	0.32	2	69	88	0.0
26	13.02	12	14	0.27	1	73	73	0.0
53	26.51	25	28	0.33	2	68	68	0.0
107	53.34	52	55	0.27	1	73	73	0.0
431	214.96	213	217	0.29	1	71	71	0.2
1727	862.83	860	865	0.38	2	63	63	3.2

Table 3.2: random permutations, `approx`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	4.20	3	5	0.24	1	76	98	0.0
17	8.75	7	10	0.32	2	69	88	0.0
26	13.00	12	14	0.25	1	75	75	0.0
53	26.54	25	28	0.36	1	64	64	0.0
107	53.36	52	55	0.29	1	71	71	0.0
431	214.98	213	217	0.31	2	70	70	0.2
1727	862.70	860	865	0.25	2	76	76	3.3

Table 3.3: random permutations, `alt_approx`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	4.36	3	5	0.40	1	60	82	0.0
17	10.01	8	12	1.58	5	17	20	2.3
26	16.12	14	19	3.37	7	0	0	30.2

Table 3.4: random permutations, `tlis`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	4.26	3	5	0.30	1	70	92	0.0
17	9.04	7	11	0.61	2	44	62	0.0
26	13.34	12	15	0.59	2	47	47	0.0
53	27.14	25	30	0.96	3	31	31	0.0
107	54.28	52	58	1.21	4	23	23	0.2
431	215.93	213	219	1.26	4	16	16	8.2
1727	863.72	861	867	1.27	3	27	27	514.6

Table 3.5: random permutations, `random`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	4.18	3	5	0.22	1	78	100	66.0
17	8.64	7	9	0.21	2	80	99	60.0
26	12.90	12	14	0.15	1	85	85	45.0
53	26.29	25	27	0.11	1	89	89	33.0
107	53.13	52	54	0.06	1	94	94	18.2
431	214.70	213	216	0.03	1	97	97	19.9
1727	862.61	860	865	0.16	1	84	84	259.8

Table 3.6: random permutations, `rpt_random`

n	Average	Min	Max	Av gap	Max gap	Match lb	Finished	Exact	Av time	Longest
8	4.18	3	5	0.22	1	78	100	100	0.0	0.0
17	8.63	7	9	0.20	1	80	100	100	23.1	697.5
26	12.91	12	14	0.16	1	84	84	84	144.4	42.6
53	26.29	25	27	0.11	1	89	89	89	99.0	0.2
107	53.13	52	54	0.06	1	94	94	94	54.0	0.2
431	214.70	213	216	0.03	1	97	97	97	27.1	1.1
1727	862.45	860	864	0.00	0	100	100	100	2.4	66.0

Table 3.7: random permutations, `branch`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact
8	4.18	3	5	0.22	1	78	100
17	8.63	7	9	0.20	1	80	100
26	12.90	12	14	0.15	1	85	85
53	26.29	25	27	0.11	1	89	89
107	53.13	52	54	0.06	1	94	94
431	214.70	213	216	0.03	1	97	97
1727	862.45	860	864	0.00	0	100	100

Table 3.8: random permutations, `best`

n	Average	Min	Max
8	3.19	3	4
17	6.05	6	7
26	9.00	9	9
53	18.00	18	18
107	36.00	36	36
431	144.00	144	144
1727	576.00	576	576

Table 3.9: 3-cycle permutations, `lower_bound`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.32	3	4	0.13	1	87	100	0.0
17	6.57	6	8	0.52	2	53	92	0.0
26	9.78	9	11	0.78	2	37	86	0.0
53	19.21	18	22	1.21	4	20	22	0.0
107	37.18	36	39	1.18	3	24	24	0.0
431	145.12	144	148	1.12	4	21	21	0.2
1727	577.13	576	580	1.13	4	24	24	2.9

Table 3.10: 3-cycle permutations, `approx`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.32	3	4	0.13	1	87	100	0.0
17	6.57	6	8	0.52	2	53	92	0.0
26	9.78	9	11	0.78	2	37	86	0.0
53	19.21	18	22	1.21	4	20	22	0.0
107	37.18	36	39	1.18	3	24	24	0.0
431	145.12	144	148	1.12	4	21	21	0.2
1727	577.13	576	580	1.13	4	24	24	2.9

Table 3.11: 3-cycle permutations, `alt_approx`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	4.35	3	6	1.16	2	25	25	0.0
17	9.92	6	13	3.87	6	1	1	2.2
26	15.87	11	18	6.87	9	0	0	27.8

Table 3.12: 3-cycle permutations, `tlis`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.32	3	4	0.13	1	87	100	0.0
17	6.57	6	8	0.52	2	53	92	0.0
26	9.85	9	12	0.85	3	33	83	0.0
53	19.07	18	22	1.07	4	30	32	0.0
107	37.23	36	40	1.23	4	21	21	0.0
431	145.24	144	147	1.24	3	20	20	0.2
1727	577.42	576	579	1.42	3	11	11	3.5

Table 3.13: 3-cycle permutations, `random`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.32	3	4	0.13	1	87	100	39.0
17	6.54	6	8	0.49	2	55	94	135.0
26	9.69	9	11	0.69	2	41	92	177.0
53	18.44	18	20	0.44	2	57	61	129.1
107	36.55	36	37	0.55	1	45	45	165.0
431	144.53	144	145	0.53	1	47	47	159.2
1727	576.54	576	577	0.54	1	46	46	166.3

Table 3.14: 3-cycle permutations, `rpt_random`

n	Average	Min	Max	Av gap	Max gap	Match lb	Finished	Exact	Av time	Longest
8	3.32	3	4	0.13	1	87	100	100	0.0	0.0
17	6.48	6	8	0.43	1	57	100	100	0.0	1.0
26	9.61	9	11	0.61	2	41	100	100	8.1	630.5
53	18.47	18	20	0.47	2	54	54	59	419.0	262.4
107	36.69	36	39	0.69	3	39	39	39	550.5	83.7
431	144.59	144	145	0.59	1	41	41	41	538.4	278.5

Table 3.15: 3-cycle permutations, `branch`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact
8	3.32	3	4	0.13	1	87	100
17	6.48	6	8	0.43	1	57	100
26	9.61	9	11	0.61	2	41	100
53	18.43	18	19	0.43	1	57	62
107	36.55	36	37	0.55	1	45	45
431	144.53	144	145	0.53	1	47	47
1727	576.54	576	577	0.54	1	46	46

Table 3.16: 3-cycle permutations, `best`

n	Average	Min	Max
8	3.00	3	3
17	6.00	6	6
26	9.00	9	9
53	18.00	18	18
107	36.00	36	36
431	144.00	144	144
1727	576.00	576	576

Table 3.17: transposition tight permutations, `lower_bound`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.00	3	3	0.00	0	100	100	0.0
17	6.00	6	6	0.00	0	100	100	0.0
26	9.02	9	11	0.02	2	99	99	0.0
53	18.26	18	20	0.26	2	85	85	0.0
107	36.59	36	39	0.59	3	69	69	0.0
431	145.02	144	148	1.02	4	47	47	0.2
1727	577.16	576	580	1.16	4	40	40	2.8

Table 3.18: transposition tight permutations, `approx`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.00	3	3	0.00	0	100	100	0.0
17	6.00	6	6	0.00	0	100	100	0.0
26	9.02	9	11	0.02	2	99	99	0.0
53	18.26	18	20	0.26	2	85	85	0.0
107	36.59	36	39	0.59	3	69	69	0.0
431	145.02	144	148	1.02	4	47	47	0.2
1727	577.16	576	580	1.16	4	40	40	2.8

Table 3.19: transposition tight permutations, `alt_approx`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.46	3	5	0.46	2	55	55	0.0
17	8.18	6	11	2.18	5	7	7	1.7
26	13.98	10	17	4.98	8	0	0	23.4

Table 3.20: transposition tight permutations, `tlis`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.00	3	3	0.00	0	100	100	0.0
17	6.00	6	6	0.00	0	100	100	0.0
26	9.02	9	11	0.02	2	99	99	0.0
53	18.20	18	20	0.20	2	87	87	0.0
107	36.69	36	38	0.69	2	62	62	0.0
431	145.01	144	148	1.01	4	45	45	0.2
1727	577.13	576	580	1.13	4	42	42	3.4

Table 3.21: transposition tight permutations, `random`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact	Av time
8	3.00	3	3	0.00	0	100	100	0.0
17	6.00	6	6	0.00	0	100	100	0.0
26	9.00	9	9	0.00	0	100	100	0.0
53	18.00	18	18	0.00	0	100	100	0.0
107	36.00	36	36	0.00	0	100	100	0.0
431	144.00	144	144	0.00	0	100	100	0.2
1727	576.00	576	576	0.00	0	100	100	5.4

Table 3.22: transposition tight permutations, `rpt_random`

n	Average	Min	Max	Av gap	Max gap	Match lb	Finished	Exact	Av time	Longest
8	3.00	3	3	0.00	0	100	100	100	0.0	0.0
17	6.00	6	6	0.00	0	100	100	100	0.0	0.0
26	9.00	9	9	0.00	0	100	100	100	0.0	0.0
53	18.01	18	19	0.01	1	99	99	99	11.7	264.3
107	36.11	36	37	0.11	1	89	89	89	102.3	118.5
431	144.20	144	146	0.20	2	81	81	81	195.7	713.1

Table 3.23: transposition tight permutations, `branch`

n	Average	Min	Max	Av gap	Max gap	Match lb	Exact
8	3.00	3	3	0.00	0	100	100
17	6.00	6	6	0.00	0	100	100
26	9.00	9	9	0.00	0	100	100
53	18.00	18	18	0.00	0	100	100
107	36.00	36	36	0.00	0	100	100
431	144.00	144	144	0.00	0	100	100
1727	576.00	576	576	0.00	0	100	100

Table 3.24: transposition tight permutations, `best`

3.6.4 Analysis of the performance of the algorithms

The most important conclusions from these experiments are:

- for most permutations π, $td(\pi)$ is equal to or very close to $tl(\pi)$.
- for most randomly generated permutations π, we can evaluate $td(\pi)$ and find an optimal length sorting sequence.

We now review and summarise the performances of the various algorithms. First of all we describe features of the algorithms that were noticeable in the experiment results. Later we discuss general features of the experiment results.

`approx` and `alt_approx`

The algorithms `approx` and `alt_approx` have almost identical behaviour in all the tests (Note that for 3-cycles permutations, and transposition tight permutations both algorithms actually do behave in an identical manner, because there are no even cycles), except that `alt_approx` performs marginally better for larger values of n. It is because of this slight advantage that `alt_approx` was chosen as the upper bound used in `branch`.

We conclude that both algorithms contain good strategies for finding sorting sequences, since both algorithms find shorter sorting sequences, on average, than `random`.

It is perhaps a little surprising that the performance of `approx` is comparable to `alt_approx` when `alt_approx` keeps 2-transpositions on even cycles up its sleeve in case it gets stuck with the odd cycles. Might `approx` not waste its 2-transpositions on even cycles, and get stuck looking for a 2-transposition more often? Perhaps these algorithms have similar performance because `approx` hardly ever gets stuck for a 2-transposition to apply, so it does not need to hold back 2-transpositions on even cycles for emergencies.

Another reason for the similar performance is that the following lemma tells us that after applying a 2-transposition that acts on an even cycle, we must be able to apply a 2-transposition on the resulting permutation.

Lemma 3.6.1 *If $tG(\pi)$ contains an oriented even cycle C, then it is possible to apply a sequence of two 2-transpositions on π.*

Proof Since C is oriented, it must be fully oriented, so a 2-transposition exists that splits C into two odd cycles and an even cycle. Now by Lemma 3.2.5 a 2-transposition can be applied on the resulting permutation, because it contains an even cycle in its cycle graph. □

Therefore `approx` is unlikely to waste 2-transpositions on even cycles, since any 2-transposition on an even cycle must lead to at least one further 2-transposition.

Though neither algorithm achieves an approximation bound, each typically uses only a handful of 0-transpositions to sort a given permutation, and far fewer 0-transpositions than the maximum allowed in order to achieve a 3/2-approximation bound for transposition distance.

The algorithms are reasonably fast for all the permutations tested. The observed run-time complexity was $O(n^2)$ for both algorithms.

`tlis`

The performance of `tlis` is clearly not as good as `approx` or `alt_approx`. On average it produces longer length sequences of transpositions to sort the given permutations, and it requires much more time.

The results of these experiments would appear to refute Heath and Vergara's claim that this algorithm "often produces near-optimal results" [HV97]. As n increases the number of permutations for which this algorithm finds an exact solution rapidly decreases. In fact, when $n = 26$ the algorithm fails to find an exact solution for any of the cases generated of the transposition tight permutations which, as we will explain later, we expect to be particularly easy to solve exactly.

Note that the other algorithms described by Guyer, Heath, and Vergara [GHV95], which are based on repeatedly selecting a strip from the permutation and moving it by a transposition so as to remove at least one breakpoint, require approximately n transpositions to sort a random permutation, whereas `approx` and `alt_approx` appear to require approximately $n/2$ transpositions. In fact, as a result of Theorem 3.2.2, we know that strips are an unimportant feature of a permutation, with respect to its transposition distance.

The observed run-time complexity was $O(n^5 \log n)$ as expected. In fact this means that the algorithm requires too much time to solve the instances when n was larger than 26.

`random` and `rpt_random`

The algorithm `random` produces sequences of transpositions that are longer, on average, than those produced by `approx` or `alt_approx`. This is viewed as evidence that the strategies employed by `approx` and `alt_approx` are good strategies.

However, the algorithm `rpt_random` finds sequences that are shorter on average than those produced by `approx` or `alt_approx`. In fact, for 3-cycle permutations and transposition tight permutations this algorithm finds sorting sequences that are shorter, on average, than all the other algorithms. For the random permutations `rpt_random` has similar performance to `branch`, except for larger values of n when, perhaps, counting the 2-transpositions takes so long that only a few iterations of `random` can be performed inside the time limit.

The observed run-time complexity of `random` was $O(n^3)$ for the random permutations, and $O(n^2)$ for the other two types of permutations. Note that when the cycle graph contains only 3-cycles, the method used to count 2-transpositions on π requires only $O(n)$ steps so the algorithm has $O(n^2)$ overall time-complexity. Note that, if n is much larger than 1727 there are too many 2-transpositions on the random permutations to be able to count them in a reasonable amount of time.

`branch`

For random permutations `branch` finds the shortest sorting sequences, on average. It is worth noting that for the random permutations tested, when $n \geq 53$ the algorithm finds an exact sorting sequence in about a second, if it can determine an exact sorting sequence. Note, of course, that this timing excludes the first call of `alt_approx`.

For $n = 1727$ the algorithm ran out of space when attempting to solve the 3-cycle permutation and transposition tight permutation instances. Therefore the tables do not contain any figures for these instances.

`best`

A strange property in the tables for the hypothetical algorithm `best` on the random permutations is the way that the column Exact starts with 100, then decreases before increasing to near 100 again. We believe that three factors have created this behaviour.

The first factor is that, in some sense, proving $td(\pi) = tl(\pi)$ is easier than proving $td(\pi) > tl(\pi)$. This is because if $td(\pi) = tl(\pi)$ then we only need to find one sequence of length $tl(\pi)$ that sorts π. However if $td(\pi) > tl(\pi)$ then we must show that all sequences of length $tl(\pi)$ fail to sort π, and this may involve a great deal of work.

The second factor is that we believe that if $td(\pi) = tl(\pi)$ then it is likely that there will be many sequences of this length that sort π, which makes it easier to find such a sorting sequence. The evidence that supports this conjecture is the good performance of the randomised algorithm `rpt_random`, and the approximation algorithms `alt_approx` and `approx`.

The third factor is that, for random permutations, it would appear that as n increases, the lower bound for transposition distance becomes a more accurate measure of exact transposition distance. That is, as n increases, it is more likely that $td(\pi) = tl(\pi)$.

So for small values of n exhaustive search is enough to find the exact transposition distance of all the permutations given. Then for slightly larger values of n, exhaustive search is unable to determine the exact transposition distance of permutations that have $td(\pi) > tl(\pi)$ in a reasonable amount of time. For much larger values of n, it is very likely that $td(\pi) = tl(\pi)$ and that `branch` or `rpt_random` will quickly find a sequence that proves it.

We summarise this behaviour in Figure 3.24. The white ribbon in the graph represents the number of permutations for which the lower bound was demonstrated to

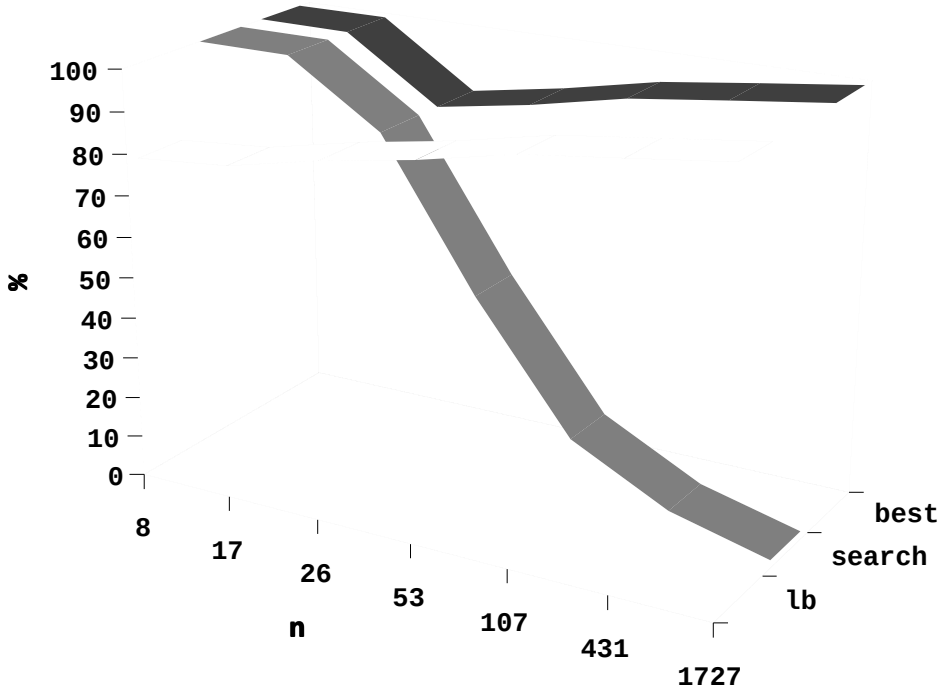

Figure 3.24: Performance of `best`.

be the exact distance. The grey ribbon is an estimate of the percentage of permutations of length n, that have $td(\pi) > tl(\pi)$, for which exhaustive search can prove $td(\pi) > tl(\pi)$ in a reasonable amount of time. The black ribbon is the maximum of the other two edges, and represents the number of permutations for which the exact distance is known.

For all the permutations tested, the length of the sorting sequence found by `best` was at most three greater than the lower bound of Theorem 3.5.3. This suggests that, in practice, this lower bound is a very accurate lower bound.

For all the permutations tested, the length of sorting sequence found by `best` was never greater than $\lfloor n/2 \rfloor + 1$. We view this as supporting evidence for Conjecture 3.5.1. If the transposition diameter were to be close to $3n/4$, the tightest known upper bound for $tD(\pi)$, then we would expect to have found permutations with transposition distance noticeably greater than $\lfloor n/2 \rfloor + 1$.

For random permutations the minimum distance and maximum distance found by `best` are remarkably close to the average distance, which itself is remarkably close to $\lceil n/2 \rceil$. Perhaps it may be possible to show that the expected transposition distance of a random permutation is $\lceil n/2 \rceil$.

3 18 12 17 21 25 8 23 2 13 10 19 9 20 5 7 4 11 24 16 14 6 1 15 22
3 18 12 17 21 25 8 23 2 13 10 19 9 20 5 6 7 4 11 24 16 14 1 15 22
1 3 18 12 17 21 25 8 23 2 13 10 19 9 20 5 6 7 4 11 24 16 14 15 22
1 2 13 10 19 9 20 5 6 7 4 11 3 18 12 17 21 25 8 23 24 16 14 15 22
1 2 3 18 13 10 19 9 20 5 6 7 4 11 12 17 21 25 8 23 24 16 14 15 22
1 2 3 4 11 12 17 21 25 18 13 10 19 9 20 5 6 7 8 23 24 16 14 15 22
1 2 3 4 11 12 17 18 13 10 19 9 20 21 25 5 6 7 8 23 24 16 14 15 22
1 2 3 4 5 6 7 8 23 24 16 14 15 22 11 12 17 18 13 10 19 9 20 21 25
1 2 3 4 5 6 7 8 9 23 24 16 14 15 22 11 12 17 18 13 10 19 20 21 25
1 2 3 4 5 6 7 8 9 23 24 16 17 18 13 14 15 22 11 12 10 19 20 21 25
1 2 3 4 5 6 7 8 9 13 14 15 22 11 12 10 23 24 16 17 18 19 20 21 25
1 2 3 4 5 6 7 8 9 13 14 15 16 17 18 19 20 21 22 11 12 10 23 24 25
1 2 3 4 5 6 7 8 9 11 12 10 13 14 15 16 17 18 19 20 21 22 23 24 25
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25

Figure 3.25: A sequence of 13 transpositions that sorts π.

Discussion

The algorithm `rpt_random` very quickly finds an exact sequence of transpositions to sort each of the transposition tight permutations. These permutations are generated by applying $\lceil n/3 \rceil$ non-interfering transpositions on π. Therefore, a transposition tight permutation for which there are many optimal length sorting sequences is more likely to be generated by this method than a transposition tight permutation for which there are relatively few optimal length sorting sequences. So we cannot conclude that `rpt_random` is a good test for determining if a permutation is transposition tight, because there may be some permutations that were not generated in our tests, for which it is much more difficult to find an exact sequence of transpositions.

However, the algorithm `rpt_random` does perform much better with these permutations than the other algorithms, for all of which the number of permutations for which the exact solution is found decreases as n increases.

Unfortunately, there are some permutations with n as small as 25 for which our algorithms are unable to determine the exact transposition distance. For example, $\pi = [3\ 18\ 12\ 17\ 21\ 25\ 8\ 23\ 2\ 13\ 10\ 19\ 9\ 20\ 5\ 7\ 4\ 11\ 24\ 16\ 14\ 6\ 1\ 15\ 22]$ has $tl(\pi) = 12$. The algorithm `approx` finds a sorting sequence, shown in Figure 3.25, of length 13. However a week's computation by algorithms `branch` and `rpt_random` has not been able to establish if $td(\pi) = 12$, or $td(\pi) = 13$.

In general, it would appear that, when $td(\pi) = tl(\pi)$, then there are many sequences of length $tl(\pi)$ that sort π. Therefore we conjecture that $td(\pi) = 13$, and that the exact algorithms run out of time before proving this. However, it is possible that there are relatively few sorting sequences of length 12, and that is why the algorithms have been unable to find one of these sequences.

It is possible to discard some transpositions during the search because we know that they would lead to a permutation that we have already searched from. For instance for many transpositions τ_1 and τ_2, $\pi \cdot \tau_1 \cdot \tau_2 = \pi \cdot \tau_2 \cdot \tau_1$. A branch and bound algorithm was implemented that discarded many possible search paths for this reason. Certainly this alternative algorithm detects permutations that have $td(\pi) > tl(\pi)$ much more quickly than the standard algorithm. However, there appear to be very few permutations like this, and in tests the alternative algorithm found hardly any new permutations like this.

It is also possible to change the branch and bound algorithm by removing the use of `alt_approx` from inside the search procedure of `branch`. This alteration to the algorithm speeds up the detection of permutations that have $td(\pi) > tl(\pi)$, but slows down the detection of permutations that have $td(\pi) = tl(\pi)$. Since most permutations have $td(\pi) = tl(\pi)$, this algorithm finds the exact distance of fewer permutations than the standard algorithm, when given the same time constraints.

We also tested all the algorithms with so called *m-permutations*. These permutations are generated by applying a sequence of m random transpositions to the identity permutation. We generated m-permutations of various sizes with $m = \lceil n/2 \rceil$, $\lceil n/3 \rceil$, and $\lceil n/4 \rceil$. As with the other tests, the algorithms `branch` and `rpt_random` found the shortest sorting sequences on average. These algorithms could determine the exact transposition distance of about 90 percent of these permutations. Interestingly $td(\pi)$ was noticeably smaller than m, on average.

3.7 Conclusion and open problems

We conclude this chapter with some conjectures and open problems. We begin with a simple conjecture that would allow us to disregard (-2)-transpositions.

Conjecture 3.7.1 *For any permutation π, there is a sequence of transpositions of length $td(\pi)$ that sorts π and contains no (-2)-transpositions.*

We make this conjecture because applying a (-2)-transpositions would appear to be a negative step towards sorting π, however we have no insight as to how to prove this conjecture.

We may chip away at the problem of sorting by transposition, by cataloguing a list of classes of permutations for which the exact transposition distance is known.

Here is a list of permutations for which the exact transposition distance is known (of course, permutations transposition equivalent to the following permutations have known transposition distance too):

- the reverse permutation R_n (Section 3.5.3).
- the permutation ω_r (Section 3.3.4).
- permutations with only 2-cycles in their cycle graphs. We can sort such permutations by repeated application of the 2-transpositions described in the proof of Lemma 3.4.8.
- the permutation $\pi_n = [2\ 4\ \ldots\ n\ 1\ 3\ \ldots\ n-1]$ (where n is even). Such a permutation has only one cycle in its cycle graph, therefore by Theorem 3.2.3, $td(\pi) \geq n/2$. It is clear that this bound can be achieved by a sequence of $n/2$ transpositions that each move one odd element into the correct place in the initial increasing sequence. Note that a permutation formed by concatenating two increasing sequences is transposition equivalent to π_n, for some n.

It is an open problem to extend this list. Of course a polynomial-time algorithm for sorting by transpositions would make this list redundant, and proving whether sorting by transpositions is in P remains an open problem. It may be possible to find a hard special case of sorting by transpositions. The problem of deciding if a permutation is transposition tight (TTDP) may be a useful problem for resolving whether sorting by transpositions is NP-hard.

A permutation must have a cycle graph containing only 3-cycles if it is to be transposition tight, and at least one cycle in each component of the overlap graph must be oriented. However, we know little more about the problem. The way that 3-cycles can interact with each other is very rich and varied. Deciding if a permutation is transposition tight seems to be much more complicated than deciding if a permutation is reversal tight. The experiments show that the algorithm `rpt_random` could recognise the permutations that were generated to be transposition tight. However, as was explained before, the method used to generate these permutations is more likely to generate a permutation that has many optimal length sorting sequences, than a permutation with relatively few optimal length sorting sequences.

The approximation algorithms of Section 3.6 appear to perform least effectively when given input permutations containing only 3-cycles in their cycle graphs. So an algorithm to solve TTDP, or some good heuristics for the problem would be useful to help improve our approximation algorithm.

Chapter 4

Sorting by Block-Interchanges

4.1 Introduction

In this chapter we introduce an operation, the *block-interchange*, in which two substrings, or blocks, are swapped in a permutation. We demonstrate a polynomial-time algorithm for calculating the block-interchange distance of a permutation (i.e. the minimum number of block-interchanges required to transform the permutation to the identity). We also determine the block-interchange diameter of the symmetric group.

A block-interchange can be viewed as a generalisation of a transposition. In a block-interchange two non-intersecting substrings of any length are swapped in the permutation. In a transposition the substrings must be adjacent.

A block-interchange $\beta = \beta(i, j, k, l)$, where $1 \leq i < j \leq k < l \leq n + 1$, is applied to π by exchanging the blocks $[\pi(i) \ \ldots \ \pi(j-1)]$ and $[\pi(k) \ \ldots \ \pi(l-1)]$. So, if $j \neq k$ then the resulting permutation $\pi \cdot \beta = [\pi(0) \ \ldots \ \pi(i-1) \ \pi(k) \ \ldots \ \pi(l-1) \ \pi(j) \ \ldots \ \pi(k-1) \ \pi(i) \ \ldots \ \pi(j-1) \ \pi(l) \ \ldots \ \pi(n+1)]$, and if $j = k$ then $\pi \cdot \beta = [\pi(0) \ \ldots \ \pi(i-1) \ \pi(k) \ \ldots \ \pi(l-1) \ \pi(i) \ \ldots \ \pi(j-1) \ \pi(l) \ \ldots \ \pi(n+1)]$. (As is standard, we assume $\pi(0) = 0$ and $\pi(n+1) = n+1$.) Note that when $j = k$ the block-interchange is also a transposition.

The block-interchange distance of π, $bd(\pi)$, is the length of a shortest sequence of block-interchanges that transforms π into the identity permutation. For example the permutation $\pi = [5\ 2\ 4\ 1\ 3]$ has block-interchange distance 2, as demonstrated in Figure 4.1. *Sorting by block-interchanges* is the problem of finding a sequence of block-interchanges of length $bd(\pi)$ that sorts π.

We find transposition breakpoints, as defined in Chapter 3, to be useful in studying the problem of sorting by block-interchanges. To recall, a transposition breakpoint is a value of i such that $1 \leq i \leq n+1$ and $\pi(i) - \pi(i-1) \neq 1$. In the rest of this chapter, for conciseness, we talk of breakpoints where we mean transposition breakpoints.

The number of breakpoints in the permutation π is denoted by $tb(\pi)$. Note that the identity permutation is the only permutation with no breakpoints, and a block-interchange can change $tb(\pi)$ by at most 4.

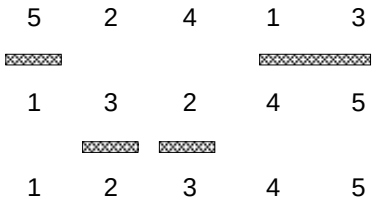

Figure 4.1: An example of sorting by block-interchanges

In this chapter we describe a polynomial-time algorithm for evaluating $bd(\pi)$. This is achieved by first demonstrating that for all permutations, except the identity permutation, a block-interchange exists that removes at least two breakpoints. Then it is proved using a graph model for the permutations that an algorithm which repeatedly applies these block-interchanges sorts π using as few block-interchanges as is possible. We then study the permutations of n elements which require the most block-interchanges in order to sort them and show that the block-interchange diameter of the symmetric group S_n is $\lfloor n/2 \rfloor$.

4.2 Minimal block-interchanges

Lemma 4.2.1 *It is always possible to find a block-interchange that removes at least two breakpoints from a given permutation, π, unless π is the identity permutation.*

Proof Since π is not the identity permutation, there must be at least two elements in π that appear in the wrong order, i.e., there must be an x and a y such that $x < y$ but $\pi = [\ldots\ y\ \ldots\ x\ \ldots]$.

Now choose x to be the smallest such value and choose y to be the largest value in π to the left of this x. Then $x-1$ must be to the left of y in π since otherwise this would contradict the choice of x. Similarly $y+1$ must be to the right of x in π. Note that in fact x is the smallest value such that $\pi(x) \neq x$. Hence π has the form

$$\pi = [1\ \ldots\ x-1\ \ldots\ y\ \ldots\ x\ \ldots\ y+1\ \ldots].$$

The block-interchange $\beta = \beta(\pi^{-1}(x-1)+1, \pi^{-1}(y)+1, \pi^{-1}(x), \pi^{-1}(y+1))$ transforms π into $\pi \cdot \beta$:

$$\pi \cdot \beta = [1\ \ldots\ x-1\ x\ \ldots\ \ldots\ \ldots\ y\ y+1\ \ldots].$$

Notice that before this transformation there were breakpoints at each place where the block-interchange cut π. After the block-interchange there are at least two fewer breakpoints in the permutation. Hence the lemma. □

We call the block-interchange described in the above lemma the *minimal block interchange*. Since we can remove at most four breakpoints with any block-interchange

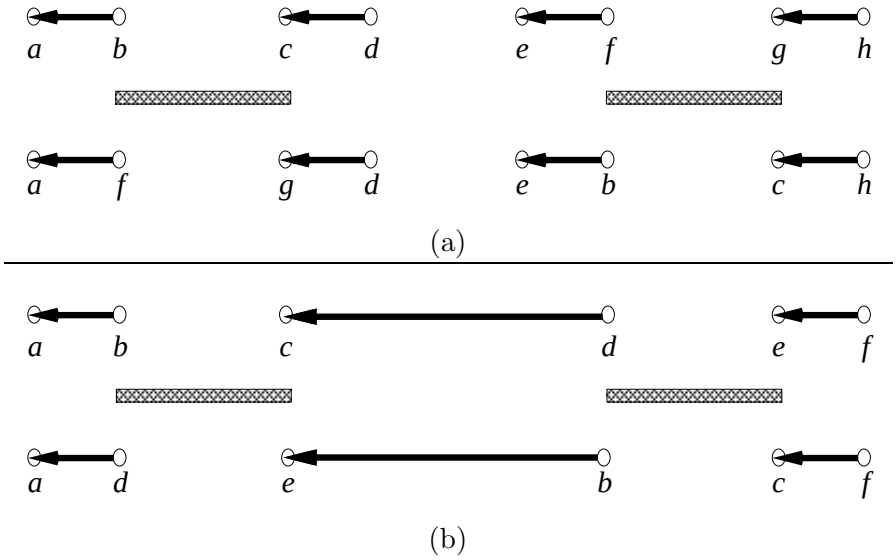

Figure 4.2: The black edges that change when a block-interchange is applied: (a) when the block-interchange is not a transposition, and (b) when the block-interchange is a transposition.

and a minimal block-interchange removes at least two breakpoints, then an algorithm that repeatedly applies the minimal block-interchanges is a 2-approximation algorithm for block-interchange distance. However when we examine these block-interchanges on a graph model we discover that, in fact, this algorithm is an exact algorithm.

4.3 The graph model

The graph model we use is that of the *transposition cycle graph* as presented in Chapter 3. The *transposition cycle graph*, $tG(\pi)$, of π is a directed edge-coloured graph with vertex set $\{0, \ldots, n+1\}$, grey edge set $\{(i, i+1) : \ 0 \leq i \leq n\}$, and black edge set $\{(\pi(i), \pi(i-1)) : \ 1 \leq i \leq n+1\}$. Note that the edge (u, v) is directed from u to v.

Since every incoming edge of a vertex can be uniquely paired with an outgoing edge of the other colour, we can easily completely decompose the graph into alternating cycles, and furthermore this decomposition is unique. The identity permutation is the only permutation that has $n+1$ alternating cycles in its graph. The number of alternating cycles in $tG(\pi)$ is denoted by $tc(\pi)$.

A block-interchange changes only black edges in the graph. Three or four black edges are removed from the graph and replaced by new black edges as shown in Figure 4.2.

Lemma 4.3.1 *When a minimal block-interchange is applied $tc(\pi)$ increases by two.*

Proof A minimal block interchange may change 3 or 4 black edges of $tG(\pi)$. If it changes 3 edges then those edges must be part of one cycle. If it changes 4 edges then those edges must be part of either one or two cycles. The three possible cases are shown in Figure 4.3 (a), (b) and (c). In each of these cases $tc(\pi)$ increases by two. Note that in the figure, paths which start and finish with a grey edge are represented by dotted grey edges. □

Lemma 4.3.2 *It is impossible to increase $tc(\pi)$ by more than two with a single block-interchange.*

Proof A block interchange changes at most four black edges in the graph, so the only way that we could increase $tc(\pi)$ by more than two with a single block-interchange would be if one big cycle was broken into four cycles, but this is impossible. A block interchange that changes black edges from four different cycles results in two cycles. By symmetry, a block interchange that results in four cycles must have acted on black edges from two cycles (an example of this is shown in Figure 4.3 (c)). So $tc(\pi)$ can be increased by at most two with a single block-interchange. □

4.4 Block-interchange distance

We can now present the main result of this chapter.

Theorem 4.4.1 *The block-interchange distance $bd(\pi)$ of a permutation π is*

$$bd(\pi) = \frac{(n+1) - tc(\pi)}{2}$$

where $tc(\pi)$ is the number of alternating cycles in the cycle graph of π.

Proof By Lemma 4.3.1, $bd(\pi) \leq (tc(\imath_n) - tc(\pi))/2$, and by Lemma 4.3.2, $bd(\pi) \geq (tc(\imath_n) - tc(\pi))/2$, where $\imath_n$ is the identity permutation of length n. So $bd(\pi) = ((n+1) - tc(\pi))/2$, since $tc(\imath_n) = n + 1$. □

An algorithm that calculates the block-interchange distance is shown in Figure 4.4. The algorithm performs a depth-first search of $tG(\pi)$, counting the number of cycles, and hence runs in linear time.

An algorithm that performs an optimal sequence of block-interchanges is shown in Figure 4.5. This algorithm has $O(n^2)$ complexity since all the steps inside the while loop can be executed in linear time, and the loop itself is executed $\lfloor n/2 \rfloor$ times at most.

4.5 Block-interchange diameter

The *block-interchange diameter*, $bD(n)$, of the symmetric group S_n, is the maximum value of $bd(\pi)$ taken over all n-element permutations. R_n is the reverse permutation

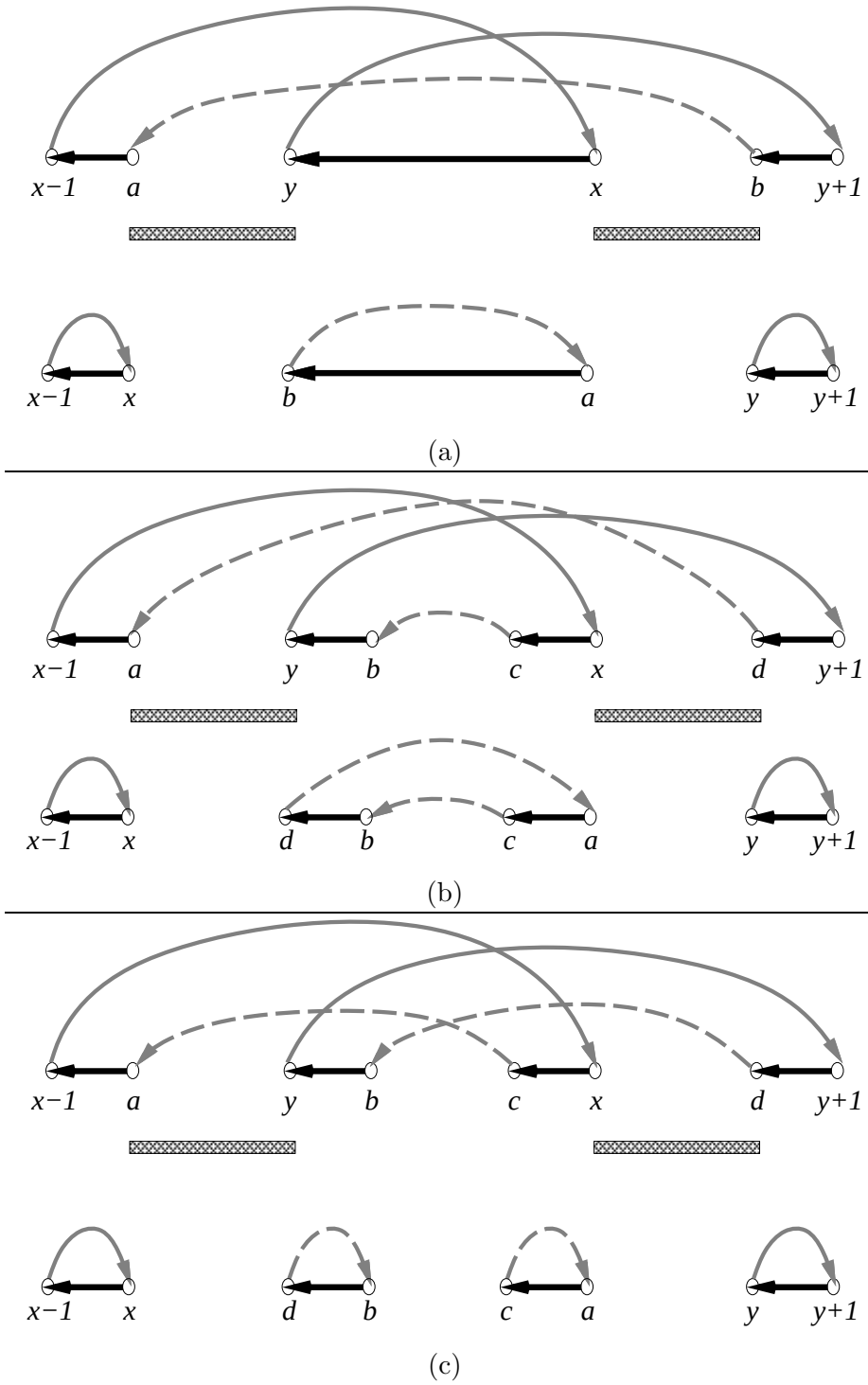

Figure 4.3: How a minimal block-interchange changes $tG(\pi)$.

```
algorithm bd(π: Permutation) is
  visited: array of Boolean;
  j, cycles: Integer;
begin
  initialise all components of visited as false;
  cycles:= 0;
  for i in 0 .. |π| loop
    if not visited(i) then
      cycles:= cycles+1;
      j:= i;
      while not visited(j) loop
        visited(j):= true;
        j:= j + 1; — follow grey edge
        j:= π(π^-1(j) − 1); — follow black edge
      end loop;
    end if;
  end loop;
  return (n + 1 − cycles)/2;
end bd;
```

Figure 4.4: An algorithm to calculate $bd(\pi)$.

```
algorithm bd_sequence(π: Permutation) is
  x, y: Integer;
begin
  while π ≠ ι loop
    locate the smallest x s.t. π(x) ≠ x;
    find the largest value, y, between x − 1 and x in π;
    π:= π · β(x, π^-1(y) + 1, π^-1(x), π^-1(y + 1));
  end loop;
end bd_sequence;
```

Figure 4.5: An algorithm to perform an optimal sequence of block-interchanges.

of length n, i.e., $R_n = [n\ n-1\ \ldots\ 1]$. It is easy to show that $tc(R_n)$ is given by the formula

$$tc(R_n) = \begin{cases} 2 & \text{if } n \text{ is odd,} \\ 1 & \text{otherwise.} \end{cases}$$

So $tc(R_n)$ is as small as possible (since $tc(\pi) \equiv n+1 \pmod 2$ for any permutation π of length n). Hence $bD(n)$ is achieved by the reverse permutation and is $\lfloor n/2 \rfloor$.

There are many other permutations which achieve this upper bound, e.g., the permutation [2 4 6 8 1 3 5 7] contains only one cycle in its cycle graph, and the family of permutations like it (that have even numbers in ascending order preceding odd numbers in ascending order) have the same value for $tc(\pi)$ as the reverse permutation. Table 4.1 shows how many permutations achieve the $bD(n)$ for small values of n.

n	1	2	3	4	5	6	7	8	9	10
number	1	1	5	8	84	180	3044	8064	193248	604800

Table 4.1: The number of permutations of size n that achieve $bD(n)$.

It is perhaps a little surprising that even though a block-interchange would appear to be a much more powerful primitive operation for sorting than a transposition, the block-interchange diameter ($\lfloor n/2 \rfloor$) is only one less than the conjectured transposition diameter ($\lfloor n/2 \rfloor + 1$).

4.6 Conclusion

A block-interchange is a generalisation of a transposition. However, in contrast with what is known for transpositions, we have derived a polynomial-time algorithm for sorting by block-interchanges, and have also evaluated the block-interchange diameter of S_n, The complexity of sorting by transpositions is unresolved, and the transposition diameter of S_n is not known.

We can think of a block-interchange as an operation in which three adjacent substrings of π are rearranged in any order. This is an alternative generalisation of a transposition, since a transposition, of course, rearranges two adjacent substrings of π in the only different order possible. As far as we know, the problem of sorting π using operations that rearrange k adjacent substrings of π in any order, has not been studied for $k \geq 4$.

Now that we have a polynomial-time algorithm for sorting by block-interchanges, we might ask, what can we say about the problem if we allow the sequences being compared to contain repeated characters? In Chapter 5 we study this and other similar problems. In particular, we show that this generalised version of sorting by block-interchanges is NP-hard.

Chapter 5

Sorting Strings by Global Transformations

5.1 Introduction

The previous chapters were devoted to the problems of sorting permutations by reversals, transpositions and block-interchanges. Corresponding problems can be defined on strings (that may contain repeated characters) instead of permutations. These problems, together with the problem on strings that corresponds to sorting permutations by prefix-reversals, are investigated in this chapter.

It was shown in a previous chapter that, for permutations, transforming π into ϕ is equivalent to transforming $\phi^{-1} \cdot \pi$ into $\imath$. However, there is no analogue of this result for strings. Therefore the string problems are expressed in terms of two strings.

Given strings S and T: the *reversal distance* $rd(S,T)$ is the minimum number of reversals required to transform S into T; the *prefix-reversal distance* $prd(S,T)$ is the minimum number of prefix-reversals required to transform S into T; the *transposition distance* $td(S,T)$ is the minimum number of transpositions required to transform S into T; and the *block-interchange distance* $bd(S,T)$ is the minimum number of block-interchanges required to transform S into T. It is impossible to insert or delete characters using reversals, prefix-reversals, transpositions, or block-interchanges, so T must be a rearrangement of S, since otherwise it would be impossible to transform S into T. We say that S and T are *related* if T is a rearrangement of S.

For each of the four transformations, the distance problem on permutations is a special case of the distance problem on strings. Since, for example, $rd(\pi) = rd(\pi, \imath_n)$. Therefore, for each of the transformations, the distance problem on strings is at least as hard as the sorting problem on permutations. Thus, finding reversal distance on strings is NP-hard since sorting permutations by reversals is NP-hard [Cap97b]. Similarly, finding prefix-reversal distance is NP-hard since sorting permutations by prefix-reversals is NP-hard [HS].

If the strings are drawn from a fixed size alphabet that is smaller than the length of the strings, then the sorting problems on permutations are no longer special cases of the distance problems on strings. In this chapter, distance problems on strings that are drawn from a binary alphabet $\{0, 1\}$ are studied. Of course, the distance problems on a larger alphabet include the distance problems on a binary alphabet as a special case. So the distance problems on a larger alphabet are at least as hard as the distance problems on a binary alphabet.

The next section (Section 5.2) contains a few useful definitions. Sections follow for each of the four distance problems. NP-completeness results are proved in Section 5.7 for two of the problems.

5.2 Definitions

A reversal, or prefix-reversal, on a string will be represented by enclosing in square brackets the substring that has to be reversed. For example, $0[1010110]1 = 001101011$. A similar notation will be used to describe transpositions and block-interchanges, though in both cases two substrings need to be bracketed.

Let 0^k represent a string of zeros of length k, 1^k represent a string of ones of length k, and in general let S^k represent the string obtained by concatenating k copies of S, for any string S. Larger strings can be built from smaller strings by concatenating them together using $+\!+$. For example, $0^2 +\!+ 1^3 = 00111$. Concatenation ($+\!+$) can also be used in a similar way to summation ($\sum$). For example, $+\!+_{i=1}^{3} 0^i 1 = 010010001$. Note that sometimes, as in the last example, the $+\!+$ symbol is omitted, and the strings to be concatenated are simply placed side by side.

Let $\mathcal{B}_n$ be the set of binary strings of length n. For a particular transformation, the *diameter* of $\mathcal{B}_n$ is the maximum distance between any two related binary strings of length n. Define $B_k = 0^k +\!+ 1^k$, and $C_k = (10)^k$. For example, $B_4 = 00001111$ and $C_4 = 10101010$. These strings are particularly useful for establishing diameter results in later sections.

Let S^{-1} represent the string derived from S by switching zeros and ones. Let $\overline{S}$ represent the string S in reverse order. If $S = 0100110001$ then $S^{-1} = 1011001110$, $\overline{S} = 1000110010$, and $\overline{S}^{-1} = 0111001101$.

Strings X and Y are *isomorphic* to strings S and T if:

$$
\begin{array}{ll}
(i) & X = S \text{ and } Y = T, \text{ or } X = T \text{ and } Y = S, \text{ or} \\
(ii) & X = S^{-1} \text{ and } Y = T^{-1}, \text{ or } X = T^{-1} \text{ and } Y = S^{-1}, \text{ or} \\
(iii) & X = \overline{S} \text{ and } Y = \overline{T}, \text{ or } X = \overline{T} \text{ and } Y = \overline{S}, \text{ or} \\
(iv) & X = \overline{S}^{-1} \text{ and } Y = \overline{T}^{-1}, \text{ or } X = \overline{T}^{-1} \text{ and } Y = \overline{S}^{-1}.
\end{array}
$$

Obviously, if X and Y are isomorphic to S and T, then $rd(X, Y) = rd(S, T)$, $td(X, Y) = td(S, T)$, and $bd(X, Y) = bd(S, T)$.

Strings X and Y are *prefix isomorphic* to strings S and T if:

$$\begin{array}{ll}(i) & X = S \text{ and } Y = T, \text{ or } X = T \text{ and } Y = S, \text{ or} \\ (ii) & X = S^{-1} \text{ and } Y = T^{-1}, \text{ or } X = T^{-1} \text{ and } Y = S^{-1}.\end{array}$$

If X and Y are prefix isomorphic to S and T, then $prd(X, Y) = prd(S, T)$.

Define $lcp(S, T)$ and $lcs(S, T)$ to be the lengths of the longest common prefix and the longest common suffix, respectively, of S and T.

A *block of zeros* is a maximal length substring that consists only of the character 0 repeated. A *block of ones* is defined similarly. Let $b(S)$ denote the total number of blocks in S, and $z(S)$ denote the number of blocks of zeros in S. So, for example, $b(001110101) = 6$, and $z(001110101) = 3$.

In the sections that follow we will often know a prefix, a suffix and sometimes a substring of a string, but not know (or care about) the rest of the string. In such cases we represent the string by joining the known parts with dots (...). For example, if S has prefix '01', a substring '00' and suffix '11' then we write $S = 01 \ldots 00 \ldots 11$.

By contrast, sometimes we will need to represent strings that have a repeating pattern. In such cases we will use a twiddle symbol ($\sim$). For example, $C_k = 1010 \sim 10$.

5.3 Reversal distance between binary strings

In this section, we present a lower bound and an upper bound for reversal distance between binary strings. These bounds are then used to determine the reversal diameter of $\mathcal{B}_n$, and also to identify some strings that achieve the reversal diameter. A restricted version of the problem, that is in some ways analogous to sorting permutations by reversals, is shown to be solvable in polynomial-time. However, in Section 5.7 the general problem of determining reversal distance between two strings is shown to be NP-hard.

5.3.1 A lower bound

In this section the concept of breakpoint from permutation sorting problems is adapted for use with string sorting problems. This new kind of breakpoint is then used to find a lower bound for reversal distance. Remember that, for permutations, two elements form a breakpoint if they are adjacent in π but not adjacent in the identity permutation. Substrings of length 2 represent adjacencies in strings S and T, so our definition of breakpoints on strings will be based on these substrings.

If S contains more '00' substrings than T, then each extra '00' must be broken, by a reversal, at some time in the transformation from S into T. Similarly if T contains more '00' substrings than S, then each extra '00' must be created by a reversal. Each extra '00' in S or T is a *reversal breakpoint*. An obvious difference between breakpoints on strings and on permutations is that, on strings, the specific location of a breakpoint may not necessarily be identified. For instance, if S contains three '00' substrings and

T contains only two '00' substrings, then one of the '00' substrings in S is a breakpoint, but no particular '00' substring of S is selected as the breakpoint. Breakpoints occur with '01', '10', and '11' substrings as well. However, care must be taken, because reversals can convert '01' substrings into '10' substrings. Therefore, '01' substrings and '10' substrings are counted together when considering reversal breakpoints.

Breakpoints can also be contributed from the beginning and end of both strings. For example, if $S(1) \neq T(1)$, then position one contributes breakpoints. In order to deal with these breakpoints, S and T are extended by adding special characters α at the beginning, and ω at the end of both strings. These breakpoints can then be counted by comparing the number of occurrences of the substrings '$\alpha 0$', '$\alpha 1$', '0ω', and '1ω' in both strings. Adding α and ω to S and T, is similar to adding 0 and $n+1$ to π when dealing with permutations.

The number of times the substring 'ab' occurs in S is denoted by $f_{ab}(S)$, where $a \in \{\alpha, 0, 1\}$ and $b \in \{0, 1, \omega\}$.

The number of reversal breakpoints (over a binary alphabet), $rb(S,T)$, is therefore

$$\begin{aligned} rb(S,T) &= |f_{00}(S) - f_{00}(T)| + |f_{11}(S) - f_{11}(T)| \\ &\quad +|f_{01}(S) + f_{10}(S) - f_{01}(T) - f_{10}(T)| \\ &\quad +|f_{\alpha 0}(S) - f_{\alpha 0}(T)| + |f_{\alpha 1}(S) - f_{\alpha 1}(T)| \\ &\quad +|f_{0\omega}(S) - f_{0\omega}(T)| + |f_{1\omega}(S) - f_{1\omega}(T)|. \end{aligned}$$

Clearly, if $S = T$, then $rb(S,T) = 0$. However, it is possible to have $rb(S,T) = 0$, even when $S \neq T$. For example, if $S = 100101$ and $T = 101001$, then $rb(S,T) = 0$.

We now derive a lower bound for reversal distance based on these breakpoints.

Lemma 5.3.1 *Suppose that S' is obtained from S by a single reversal. Then*

$$rb(S',T) \geq rb(S,T) - 4.$$

Proof A reversal on S cuts two substrings of length two in the extended version of S. Suppose that the first of these substrings is ab and the other is cd. Recall that the frequency counts of 01 substrings and 10 substrings are added together when counting breakpoints. So the frequency counts of substrings in S' will be the same as the frequency counts of substrings in S except that S' contains one less ab and cd and one more ac and bd. Given the definition of $rb(S',T)$, this means that $rb(S',T) \geq rb(S,T) - 4$. □

This lemma can be used to easily deduce the following lower bound for reversal distance.

Theorem 5.3.1 *Let S and T be related binary strings. Then*

$$rd(S,T) \geq \lceil rb(S,T)/4 \rceil.$$

Unfortunately this lower bound is not tight. For example, if $S = 0011000111$ and $T = 1110011000$, then $rb(S,T) = 4$, but $rd(S,T) = 2$. Note also that sometimes it is impossible to apply a reversal that will reduce the number of reversal breakpoints. For example, $rb(\overline{S},T) = 0$, but $\overline{S} \neq T$.

5.3.2 An upper bound

In this section a simple upper bound is derived for reversal distance.

Lemma 5.3.2 *Let S and T be related strings of length n, such that $S \neq T$. Then it is possible either:*
a) to apply a reversal on S resulting in the string S', such that $lcp(S',T)+lcs(S',T) \geq lcp(S,T)+lcs(S,T)+2$, or
b) to apply a reversal on T resulting in the string T' such that $lcp(S,T')+lcp(S,T') \geq lcp(S,T)+lcs(S,T)+2$.

Proof Without loss of generality, it can be assumed that $S(1) = 0$, since otherwise we could consider S^{-1} and T^{-1}, and apply the resulting reversal to S and T. Further, it can be assumed that $S(1) \neq T(1)$, and $S(n) \neq T(n)$, because otherwise we could reduce n by removing any common prefix or suffix from both strings.

The reversal applied to S or T depends on S and T. We describe seven cases, and show a reversal with the required property for each one. The seven cases cover all possibilities for S and T, though the cases are not necessarily mutually exclusive.

Case (i). $S(n) = 1$: then $S = 0\ldots1$ and $T = 1\ldots0$, so take $S' = [0\ldots1]$.

Case (ii). $T(2) = 0$: then $S = 0\ldots0$, and $T = 10\ldots1$, so take $S' = [0 \sim 01]\ldots0$, where the '01' substring at the end of the reversal is the first '01' substring in S.

Case (iii). $T(n-1) = 0$: then $S = 0\ldots0$, and $T = 11\ldots01$, so, by considering $\overline{S}$, and $\overline{T}$, this case can be dealt with in a similar way to Case (ii).

Case (iv). $f_{11}(S) > 0$: then $S = 0\ldots11\ldots0$ and $T = 11\ldots11$, so take $S' = [0\ldots11]\ldots00$, where the '11' substring at the end of the reversal is the first '11' substring in S.

Case (v). $S(2) = 1$: then $S = 01\ldots0$ and $T = 11\ldots11$ so, by considering S^{-1}, and T^{-1}, this case can be dealt with in a similar way to Case (ii).

Case (vi). $S(n-1) = 1$: then $S = 00\ldots10$ and $T = 11\ldots11$ so, by considering $\overline{S}^{-1}$, and $\overline{T}^{-1}$, this case can be dealt with in a similar way to Case (ii).

Case (vii). $f_{00}(T) > 0$: then $S = 00\ldots00$ and $T = 11\ldots00\ldots11$, so, by considering S^{-1}, and T^{-1}, this case can be dealt with in a similar way to Case (iv).

These cases are exhaustive because Cases (i) to (iv) can only fail to apply if S contains more zeros than ones, whereas Cases (i), and (v) to (vii) can only fail to apply if T contains more ones than zeros. □

Theorem 5.3.2 *Let S and T be related binary strings of length n. Then*

$$rd(S,T) \leq \lfloor n/2 \rfloor.$$

Proof Lemma 5.3.2 describes a way to increase the combined length of the common prefix and suffix of S and T by at least two using a single reversal. So a sequence of $\lfloor n/2 \rfloor$ such reversals will be enough to transform S into T. □

For example, if $S = 010101010$, and $T = 110000011$, then applying reversals as described in the proof of Lemma 5.3.2 results in the following sequence of strings. (In this example the substrings to be reversed are underlined).

$$\begin{array}{c} 010\underline{10}1010 \\ 011001\underline{010} \\ 01100\underline{010}1 \\ \underline{011}000011 \\ 110000011 \end{array}$$

Note that the first reversal found as described in the proof of Lemma 5.3.2 is the one that reverses the first three characters of T, and so it is the last reversal shown in the illustration.

5.3.3 Reversal diameter of $\mathcal{B}_n$

The *reversal diameter* of $\mathcal{B}_n$, $rD_2(n)$, is defined to be the maximum value of $rd(S,T)$ over all related binary strings S and T of length n. More formally

$$rD_2(n) = \max\{rd(S,T) : S,\ T \text{ are related binary strings of length } n\}.$$

Lemma 5.3.3 *For all $k \geq 1$, $rd(B_k, C_k) = k$, and $rd(0 +\!\!+ B_k, 0 +\!\!+ C_k) = k$.*

Proof This follows at once by application of Theorem 5.3.1, and Theorem 5.3.2 to these strings. □

Theorem 5.3.3 *For all $n \geq 1$, $rD_2(n) = \lfloor n/2 \rfloor$.*

Proof This is an immediate consequence of Theorem 5.3.2 and Lemma 5.3.3. □

Theorem 5.3.4 *Let S and T be related binary strings of length $2n \geq 6$. Then $rd(S,T) = n$, if and only if S and T are isomorphic to C_n and B_n.*

Proof We prove this theorem by induction. The base case is when $2n = 6$. Then, by complete search, it may be verified that $rd(S,T) = 3$ if and only if S and T are isomorphic to B_3 and C_3. Now suppose that the theorem holds for $n \leq k$. Let S and T be strings of length $2k+2$ such that $rd(S,T) = k+1$. We show that S and T are isomorphic to C_{k+1} and B_{k+1}.

We can assume, without loss of generality, that $S(1) = 0$. By Lemma 5.3.2, we can apply a reversal to S or T that increases the combined length of the common prefix and suffix by at least two. By the proof of Lemma 5.3.2 we can assume that the reversal is applied to S resulting in the string S'. It must be that $lcp(S', T) + lcs(S', T) = 2$, and $rd(S', T) = k$, since any alternative would contradict $rd(S, T) = k + 1$. Let S'_e and T_e be the strings S' and T excluding common prefix and suffix. By the induction hypothesis, S'_e and T_e must be isomorphic to B_k and C_k. Therefore, since $B_k^{-1} = \overline{B_k}$, and $C_k^{-1} = \overline{C_k}$, either:

a) $S'_e = B_k$ and $T_e = C_k$; or
b) $S'_e = C_k$ and $T_e = B_k$; or
c) $S'_e = \overline{B_k}$ and $T_e = \overline{C_k}$; or
d) $S'_e = \overline{C_k}$ and $T_e = \overline{B_k}$.

By the proof of Lemma 5.3.2 there are essentially three ways that the reversal can be applied to S, as typified by cases (i), (ii) and (iv) in that proof. We take each of these three cases in turn, and show that for the four possible values of S'_e and T_e, $rd(S, T) = k + 1$ if and only if S and T are isomorphic to B_{k+1} and C_{k+1}.

Case (i). $S = 0 \ldots 1$, $T = 1 \ldots 0$. In this case the whole of S is reversed. We show that cases a) and b) for S'_e and T_e lead to contradictions, whereas cases c) and d) establish the induction step.

a) $S = 0 +\!\!+ \overline{B_k} +\!\!+ 1 = 011 \sim 100 \sim 01$, $T = 1 +\!\!+ C_k +\!\!+ 0 = 11010 \sim 100$. Suppose that instead of applying the reversal of Lemma 5.3.2 we apply the reversal $[011] \sim 100 \sim 01$ to obtain S''. This reversal extends the common prefix by three characters, so $rd(S'', T) < k$. So $rd(S, T) < k + 1$, a contradiction.

b) $S = 0 +\!\!+ \overline{C_k} +\!\!+ 1 = 00101 \sim 011$, $T = 1 +\!\!+ B_k +\!\!+ 0 = 100 \sim 011 \sim 10$. These strings are isomorphic to the strings in a), so $rd(S, T) < k + 1$, a contradiction.

c) $S = 0 +\!\!+ B_k +\!\!+ 1 = B_{k+1}$, $T = 1 +\!\!+ \overline{C_k} +\!\!+ 0 = C_{k+1}$.

d) $S = 0 +\!\!+ C_k +\!\!+ 1 = \overline{C_{k+1}}$, $T = 1 +\!\!+ \overline{B_k} +\!\!+ 0 = \overline{B_{k+1}}$.

Case (ii). $S = 0 \ldots 0$, $T = 10 \ldots 1$. In this case the reversal results in a string S' that has prefix '10'. So S'_e and T_e must be suffixes of S' and T. Now since T ends with a 1, only cases b) and c) need to be considered. Both cases lead to a contradiction.

b) $S' = 10 +\!\!+ C_k = 1010 \sim 10$, and $T = 10 +\!\!+ B_k = 1000 \sim 011 \sim 1$. The reversal on S ends with the first '01' substring in S, so $S = 011010 \sim 10$. Then the reversal $[01101]0 \sim 10$ produces string S'' that has $rd(S'', T) < k$ by the induction hypothesis. So $rd(S, T) < k + 1$, a contradiction.

c) $S' = 10 +\!\!+ \overline{B_k} = 1011 \sim 100 \sim 0$, and $T = 10 +\!\!+ \overline{C_k} = 100101 \sim 01$. Given the nature of the reversal on S, $S = 0111 \sim 100 \sim 0$. Then the reversal $0111 \sim [100 \sim 0]$ results in a string S'', that has $rd(S'', T) < k$ by the induction hypothesis. So $rd(S, T) < k + 1$, a contradiction.

Case (iv). $S = 0 \ldots 11 \ldots 0$, and $T = 11 \ldots 11$. In this case the reversal is applied to S to obtain a string S' that has prefix 11. So S'_e and T_e must be suffixes of S' and

T. Now, since T ends with '11' only case b) need be considered. However, in fact, even this case cannot occur.

b) $S' = 11 +\!\!+ C_k = 111010 \sim 10$, and $T = 11 +\!\!+ B_k = 1100 \sim 011 \sim 1$. But then the reversal on S could not have moved the first '11' substring in S. So this case cannot occur.

So $rd(S,T) = k+1$, if and only if S and T are isomorphic to B_{k+1} and C_{k+1}. Therefore, by induction, we have proved the theorem. □

Theorem 5.3.4 describes the strings of length n that achieve the reversal diameter, when n is even. When n is odd, significantly more pairs of strings achieve the reversal diameter.

5.3.4 Sorting by reversals

Let S_ι denote the string that is related to S and consists only of a block of zeros, followed by a block of ones. For example, if $S = 01100110$ then $S_\iota = 00001111$. Then determining $rd(S, S_\iota)$ is an analogue of determining the reversal distance of a permutation. We show that $rd(S, S_\iota)$ can be determined in polynomial-time.

Recall that $z(S)$ denotes the number of blocks of zeros contained in S. Obviously, $z(S_\iota) = 1$ (unless S does not contain any zeros). The following lemma can be verified easily.

Lemma 5.3.4 *Let S' be a string obtained from S by a single reversal. Then*

$$z(S') \geq z(S) - 1.$$

With this lemma we can determine $rd(S, S_\iota)$.

Theorem 5.3.5 *For any binary string S,*

$$rd(S, S_\iota) = \begin{cases} z(S) - 1, & \text{if } S(1) = 0, \\ z(S), & \text{otherwise.} \end{cases}$$

Proof By Lemma 5.3.4, $rd(S, S_\iota) \geq z(S) - 1$. If $S(1) = 0$, then $z(S) - 1$ reversals of the form $0 \sim 0[1 \ldots 0]b \ldots$, where $b \in \{1, \omega\}$, transform S into S_ι. If $S(1) = 1$, then an extra reversal is required, because it is impossible to change the character at the beginning to a 0, and also reduce the value of z. This bound can be achieved by performing the reversal $[1 \ldots 0]b \ldots$, before performing the $z(S) - 1$ reversals described for when $S(1) = 0$. □

The distance described in Theorem 5.3.5 can be calculated easily in polynomial-time. In Section 5.7 it is shown that, in general, determining $rd(S,T)$ is NP-hard.

5.4 Prefix-reversal distance between binary strings

In this section, we present an upper bound and a lower bound for prefix-reversal distance between binary strings. These bounds are used to determine the prefix-reversal diameter of $\mathcal{B}_n$, and to identify some strings that achieve the prefix-reversal diameter. A restricted version of the problem, that is in some ways analogous to the problem of sorting by prefix-reversals, is shown to be solvable in polynomial-time.

5.4.1 A lower bound

The lower bound for prefix-reversal distance is based on blocks instead of breakpoints. It is possible to base the lower bound on breakpoints, but such a bound is not as elegant because the first character needs to be dealt with in a special way. Note, it is possible to obtain a lower bound for reversal distance (and transposition distance and block-interchange distance) based on blocks instead of breakpoints. However, in general, such a bound is not as tight as the bound based on breakpoints.

The following lemma can be verified easily.

Lemma 5.4.1 *Suppose that S' is obtained from S by a single prefix-reversal. Then*

$$b(S) - b(S') \in \{-1, 0, 1\}.$$

This lemma can be used to deduce the following lower bound for prefix-reversal distance.

Theorem 5.4.1 *Let S and T be related binary strings of length n. Then*

$$prd(S,T) \geq \begin{cases} |b(S) - b(T)| + 1, & \text{if } S(n) \neq T(n), \\ |b(S) - b(T)|, & \text{otherwise.} \end{cases}$$

Proof By Lemma 5.4.1 at least $|b(S) - b(T)|$ prefix-reversals are required, just to ensure that both strings contain the same number of blocks. If $S(n) \neq T(n)$ then at one point in the transformation from S into T, the whole string will have to be reversed. Such a prefix-reversal cannot change the value of b, so at least one more prefix-reversal is required in such instances. □

This lower bound is not exact. For example, if $S = 011001$ and $T = 101100$, then the bound states $prd(S,T) \geq 1$, whereas $prd(S,T) = 2$.

5.4.2 An upper bound

In this section a simple upper bound is obtained for prefix-reversal distance.

Lemma 5.4.2 *Let S and T be related binary strings, such that $S \neq T$. Then it is possible either:*
a) to apply a prefix-reversal to S or T, obtaining the string S' or T', such that $lcs(S', T) \geq lcs(S, T) + 1$, or $lcs(S, T') \geq lcs(S, T) + 1$, or
b) to apply a sequence of two prefix-reversals to S or T resulting in the string S'' or T'', such that $lcs(S'', T) \geq lcs(S, T) + 2$, or $lcs(S, T'') \geq lcs(S, T) + 2$.

Proof It can be assumed without loss of generality that $S(1) = 0$, and $S(n) \neq T(n)$. The prefix-reversals that are applied to S or T depend on S and T. We describe six cases, and show prefix-reversals with the required property in each case. The six cases cover all possibilities for S and T, though the cases are not necessarily mutually exclusive.

Case (i). $T(n) = 0$: then $S = 0 \ldots 1$ and $T = \ldots 0$, so take $S' = [0 \ldots 1]$.

Case (ii). $T(1) = 0$: then $S = 0 \ldots 0$, and $T = 0 \ldots 1$, so swapping the roles of S and T, this case is similar to Case (i).

Case (iii). $T(n-1) = 0$: then $S = 0 \ldots 0$, and $T = 1 \ldots 01$, so take $S' = [0 \sim 01] \ldots 0$, then $S'' = [10 \ldots 0]$, where the '01' substring at the end of the first prefix-reversal is the first '01' substring in S.

Case (iv). $S(n-1) = 1$: then $S = 0 \ldots 10$, and $T = 1 \ldots 11$, so, by considering S^{-1}, and T^{-1}, this case is similar to Case (iii).

Case (v). $f_{11}(S) > 0$: then $S = 0 \ldots 11 \ldots 00$, and $T = 1 \ldots 11$, so take $S' = [0 \ldots 11] \ldots 00$, then $S'' = [11 \ldots 00]$, where the '11' substring at the end of the first prefix-reversal is the first '11' substring in S.

Case (vi). $f_{00}(T) > 0$: then $S = 0 \ldots 00$, and $T = 1 \ldots 00 \ldots 11$, so by considering S^{-1}, and T^{-1}, this case is similar to Case (v).

These cases are exhaustive because Cases (i) to (v) can only fail to apply if S contains more zeros than ones, whereas Cases (i) to (iv), and (vi) can only fail to apply if T contains more ones than zeros. □

Theorem 5.4.2 *Let S and T be related binary strings of length n. Then*

$$prd(S, T) \leq n - 1.$$

Proof The prefix-reversals described by Lemma 5.4.2 extend the common suffix of the two strings by at least one character on average. The sequence of two prefix-reversals is only selected if the strings are at least three characters long. Therefore, at most $n-1$ prefix-reversals like those indicated by Lemma 5.4.2 will be required to transform S into T. □

5.4.3 Prefix-reversal diameter of $\mathcal{B}_n$

The *prefix-reversal diameter* of $\mathcal{B}_n$, $prD_2(n)$, is defined to be the maximum value of $prd(S, T)$ over all related binary strings S and T of length n. More formally

$$prD_2(n) = \max\{prd(S, T) : \text{S, T are related binary strings of length } n\}.$$

Lemma 5.4.3 *For all $k \geq 1$, $prd(B_k, C_k) = 2k - 1$, and $prd(0 +\!\!+ B_k, 0 +\!\!+ C_k) = 2k$.*

Proof This follows at once by application of Theorem 5.4.1 and Theorem 5.4.2 to these strings. □

Theorem 5.4.3 *For all $n \geq 1$, $prD_2(n) = n - 1$.*

Proof This is a simple consequence of Theorem 5.4.2 and Lemma 5.4.3. □

Theorem 5.4.4 *Let S and T be related strings of length $n \geq 2$. Then $prd(S, T) = n - 1$, when n is even, if and only if S and T are prefix isomorphic to $C_{n/2}$ and $B_{n/2}$, and $prd(S, T) = n - 1$, when n is odd, if and only if S and T are prefix isomorphic to $0 +\!\!+ C_{\lfloor n/2 \rfloor}$ and $0 +\!\!+ B_{\lfloor n/2 \rfloor}$.*

Proof We prove this theorem by induction. The base cases are when $n = 2$ and $n = 3$. It is easy to verify the theorem for these values of n. Now suppose that the theorem holds for $n \leq k$. Let S and T be strings of length $k + 1$ such that $prd(S, T) = k$. We show that S and T are isomorphic to the strings indicated.

We can assume, without loss of generality, that $S(1) = 0$. By Lemma 5.4.2 we can apply a prefix-reversal or two prefix-reversals to S or T, in such a way that each prefix-reversal, on average, increases the common suffix of the two strings by a character. By the proof of Lemma 5.4.2, we can assume that r prefix-reversals are applied to S, resulting in the string S^*. It must be that $lcs(S^*, T) = r$, and $prd(S^*, T) = k - r$, since any alternative would contradict the choice of S, and T. Let S_e^*, and T_e be the strings S^* and T excluding the common suffix. By the induction hypothesis, S_e^* and T_e must be prefix isomorphic to the strings indicated in the statement of the lemma. Let $l = k + 1 - r$. Then l is the length of S_e^* and T_e. If l is even then either:

$$\begin{array}{ll} \text{a)} & S_e^* = B_{\frac{l}{2}}, \text{ and } T_e = C_{\frac{l}{2}}, \\ \text{b)} & S_e^* = C_{\frac{l}{2}}, \text{ and } T_e = B_{\frac{l}{2}}, \\ \text{c)} & S_e^* = B_{\frac{l}{2}}^{-1}, \text{ and } T_e = C_{\frac{l}{2}}^{-1}, \text{ or} \\ \text{d)} & S_e^* = C_{\frac{l}{2}}^{-1}, \text{ and } T_e = B_{\frac{l}{2}}^{-1}. \end{array}$$

If l is odd then either:

$$\begin{array}{ll} \text{e)} & S_e^* = 0 +\!\!+ B_{\frac{l-1}{2}}, \text{ and } T_e = 0 +\!\!+ C_{\frac{l-1}{2}}, \\ \text{f)} & S_e^* = 0 +\!\!+ C_{\frac{l-1}{2}}, \text{ and } T_e = 0 +\!\!+ B_{\frac{l-1}{2}}, \\ \text{g)} & S_e^* = 1 +\!\!+ B_{\frac{l-1}{2}}^{-1}, \text{ and } T_e = 1 +\!\!+ C_{\frac{l-1}{2}}^{-1}, \text{ or} \\ \text{h)} & S_e^* = 1 +\!\!+ C_{\frac{l-1}{2}}^{-1}, \text{ and } T_e = 1 +\!\!+ B_{\frac{l-1}{2}}^{-1}. \end{array}$$

By the proof of Lemma 5.4.2, there are essentially three different ways that the prefix-reversal or prefix-reversals could be applied to S, as typified by cases (i), (iii) and (v) in that proof. We take each case in turn, and show that S and T must be prefix isomorphic to the strings indicated above if $prd(S,T) = n-1$.

First of all, suppose that $k+1$ is even.

Case (i). $S = 0\ldots1$ and $T = \ldots0$. In this case the prefix-reversal reverses the whole of S. Only one prefix-reversal is applied so l is odd. Only two cases g) and h) need to be considered since S^* must begin with 1. Both cases establish the induction step.

g) $S = 0 +\!\!+ B_{\frac{k-1}{2}} +\!\!+ 1 = 000 \sim 011 \sim 11 = B_{\frac{k+1}{2}}$ and $T = 1 +\!\!+ C^{-1}_{\frac{k-1}{2}} +\!\!+ 0 = 10101 \sim 010 = C_{\frac{k+1}{2}}$.

h) $S = 0 +\!\!+ C_{\frac{k-1}{2}} +\!\!+ 1 = 01010 \sim 101 = C^{-1}_{\frac{k+1}{2}}$ and $T = 1 +\!\!+ B^{-1}_{\frac{k-1}{2}} +\!\!+ 0 = 111 \sim 100 \sim 00 = B^{-1}_{\frac{k+1}{2}}$.

Case (iii). $S = 0\ldots0$, and $T = 1\ldots01$. In this case a sequence of two prefix-reversals is applied to S to obtain S'', where $S' = [0 \sim 01]\ldots0$, and $S'' = [10\ldots0]$. Therefore l is even. Only two cases a) and d) need to be considered, since T must begin with 1. However both cases lead to a contradiction.

a) $S'' = B_{\frac{k-1}{2}} +\!\!+ 01 = 00 \sim 011 \sim 101$, and $T = C_{\frac{k-1}{2}} +\!\!+ 01 = 1010 \sim 1001$. So $S = 0111 \sim 100 \sim 0$. Take $T' = [1010\ldots100]1$, then $T'' = [001\ldots011]$. Then $prd(S,T'') < k-2$ by the induction hypothesis, so $prd(S,T) < k$.

d) $S'' = C^{-1}_{\frac{k-1}{2}} +\!\!+ 01 = 0101 \sim 0101$, and $T = B^{-1}_{\frac{k-1}{2}} +\!\!+ 01 = 11 \sim 100 \sim 001$. So $S = 0110\ldots1010$. Take $T' = [11\ldots10]0\ldots01$, and $T'' = [011\ldots10\ldots01]$. Then $prd(S,T'') < k-2$ by the induction hypothesis, so $prd(S,T) < k$.

Case (v). $S = 0\ldots11\ldots00$, and $T = 1\ldots11$. In this case two prefix-reversals are applied to S to obtain S'', where $S' = [0\ldots11]\ldots00$, and $S'' = [11\ldots00]$. Therefore l is even. Only case a) needs to be considered, since S'' begins with '00', but, in fact, even this case cannot occur.

a) $S'' = B_{\frac{k-1}{2}} +\!\!+ 11 = 00 \sim 011 \sim 111$ and $T = C_{\frac{k-1}{2}} +\!\!+ 11 = 1010 \sim 1011$. But the first prefix-reversal on S should end with the first '11' substring in S. However, that would mean that S'' could not possibly have the form shown. So this case cannot occur.

Now, let us suppose $k+1$ is odd.

Case (i). $S = 0\ldots1$ and $T = \ldots0$. The prefix-reversal reverses the whole of S. Therefore l is even. Only case b) and c) need to be considered, since S' begins with 1. Case b) leads to a contradiction, and case c) establishes the induction step.

b) $S = 0 +\!\!+ C^{-1}_{\frac{k}{2}} = 00101 \sim 01$ and $T = B_{\frac{k}{2}} +\!\!+ 0 = 00 \sim 011 \sim 10$. Take $T' = [00 \sim 01]1 \sim 10$, and then $T'' = [100 \sim 011 \sim 10]$. Now $prd(S,T'') < k-2$, by the inductive hypothesis, so $prd(S,T) < k$.

c) $S = 0 +\!\!+ B_{\frac{k}{2}} = 000 \sim 011\ldots1$ and $T = C^{-1}_{\frac{k}{2}} +\!\!+ 0 = 0101 \sim 010 = 0 +\!\!+ C_{\frac{k}{2}}$.

Case (iii). $S = 0 \dots 0$, and $T = 1 \dots 01$. In this case two prefix-reversals are applied to S to obtain S'', where $S' = [0 \dots 01] \dots 0$, and $S'' = [10 \dots 0]$. Therefore l is odd. But $T(1) = 1$, and $S''(1) = 0$ and so by the induction hypothesis $prd(S'', T) < k - 2$. Therefore, $prd(S, T) < k$.

Case (v). $S = 0 \dots 11 \dots 00$, and $T = 1 \dots 11$. In this case two prefix-reversals are applied to S to obtain S'', where $S' = [0 \dots 11] \dots 00$ and $S'' = [11 \dots 00]$. Therefore l is odd. But $T(1) = 1$, and $S''(1) = 0$ and so by the induction hypothesis $prd(S'', T) < k - 2$. Therefore, $prd(S, T) < k$.

So, by induction, we have proved the theorem. □

5.4.4 Sorting by prefix-reversals

We show that $prd(S, S_\iota)$ can be determined in polynomial-time.

The following lemma can be verified easily.

Lemma 5.4.4 *Let S' be a string obtained from S by a single prefix-reversal. Then*

$$z(S') \geq \begin{cases} z(S) - 1, & \text{if } S(1) = 0, \\ z(S), & \text{otherwise.} \end{cases}$$

With this lemma we can determine $prd(S, S_\iota)$.

Theorem 5.4.5 *For any binary string S,*

$$prd(S, S_\iota) = \begin{cases} 2(z(S) - 1), & \text{if } S(1) = 0, \\ 2z(S) - 1, & \text{otherwise.} \end{cases}$$

Proof If a prefix-reversal reduces the value of z, then it must bring 1 to the front of the string, and then by Lemma 5.4.4 the next prefix-reversal cannot reduce the value of z. Note also that the final prefix-reversal cannot reduce the value of z because it must bring 0 to the front. Therefore $prd(S, S_\iota) \geq 2(z(S) - 1)$. If $S(1) = 0$, then $z(S) - 1$ prefix-reversals of the form $[0 \sim 01 \sim 1]0 \dots$ interleaved with $z(S) - 1$ prefix-reversals of the form $[1 \sim 10 \sim 0]b \dots$, where $b \in \{1, \omega\}$, transform S into S_ι. If $S(1) = 1$, then an extra prefix-reversal is required, since the first prefix-reversal cannot reduce the value of z. This bound can be achieved by performing the prefix-reversal $[1 \sim 10 \sim 0]b \dots$, where $b \in \{1, \omega\}$, before performing the $2(z(S) - 1)$ prefix-reversals just described for when $S(1) = 0$. □

The distance described in Theorem 5.4.5 can be calculated easily in polynomial-time. The question of whether, in general, the prefix-reversal distance between any two strings can be calculated in polynomial-time remains open.

5.5 Transposition distance between binary strings

In this section, we present an upper bound and a lower bound for transposition distance between binary strings. These bounds are used to determine the transposition diameter of $\mathcal{B}_n$, and identify some strings that achieve the diameter. A restricted version of the problem, that is in some ways analogous to the problem of sorting by transpositions, is shown to be solvable in polynomial-time.

5.5.1 A lower bound

In this section, breakpoints are used to obtain a lower bound for transposition distance.

Transposition breakpoints are defined in a similar way to reversal breakpoints. For example, if S contains more 11 substrings than T then each extra 11 substring contributes a breakpoint. However a crucial difference between reversal breakpoints and transposition breakpoints is that 01 substrings and 10 substrings are counted separately. As before prepend α and append ω to each string.

The number of *transposition breakpoints* (over a binary alphabet) is therefore

$$\begin{aligned} tb(S,T) &= |f_{00}(S) - f_{00}(T)| + |f_{01}(S) - f_{01}(T)| + |f_{10}(S) - f_{10}(T)| \\ &\quad + |f_{11}(S) - f_{11}(T)| + |f_{\alpha 0}(S) - f_{\alpha 0}(T)| + |f_{\alpha 1}(S) - f_{\alpha 1}(T)| \\ &\quad + |f_{0\omega}(S) - f_{0\omega}(T)| + |f_{1\omega}(S) - f_{1\omega}(T)|. \end{aligned}$$

Clearly, $tb(T,T) = 0$. However it is possible that $tb(S,T) = 0$, even when $S \neq T$. For example, if $S = 101001$ and $T = 100101$, then $tb(S,T) = 0$.

Lemma 5.5.1 *Suppose that S' is obtained from S by a single transposition. Then*

$$tb(S',T) \geq \begin{cases} tb(S,T) - 6, & \text{if the transposition moves the first and last letters of } S, \\ tb(S,T) - 4, & \text{otherwise.} \end{cases}$$

Proof The transposition must have the form $\ldots a[b\ldots c][d\ldots e]f\ldots$, where $a \in \{\alpha, 0, 1\}$, $b, c, d, e \in \{0, 1\}$, and $f \in \{0, 1, \omega\}$. The transposition results in the string $\ldots a[d\ldots e][b\ldots c]f\ldots$. Now let us suppose that the none of the substrings 'ab', 'cd', or 'ef' is the same as any of the substrings 'ad', 'eb', or 'cf'. Then 'cd'$\neq$'cf', so $d \neq f$. Similarly, $c \neq a$, $b \neq f$, and $e \neq a$. Now suppose that $a \neq \alpha$. Then $c = e$. But then 'ef'='cf', a contradiction. Similarly if $f \neq \omega$ then 'ab'='ad'. Therefore, if $a \neq \alpha$, or $f \neq \omega$, then at least one of the substrings 'ab', 'cd', or 'ef' is the same as one of the substrings 'ad', 'eb', or 'cf'.

If a transposition does move the first and last characters of S then at most three substrings of length two may change as a result of the transposition. Therefore, in such cases, the frequency counts of substrings in S' are the same as the frequency counts of substrings in S except that the frequency counts of up to three substrings are reduced

by one, and the frequency counts of up to three substrings are increased by one. Given the definition of $tb(S', T)$, this means that $tb(S', T) \geq tb(S, T) - 6$.

However, if a transposition does not move the first and last characters of S, then at most two substrings of length two may change as a result of the transposition. In such cases $tb(S', T) \geq tb(S, T) - 4$. □

Theorem 5.5.1 *Let S and T be related binary strings, of length n. Then*

$$td(S, T) \geq \begin{cases} \lceil tb(S, T)/4 \rceil, & \text{if } S(1) = T(1), \text{ or } S(n) = T(n), \\ \lceil (tb(S, T) - 2)/4 \rceil, & \text{otherwise.} \end{cases}$$

Proof A sequence of transpositions that transforms S into T can contain at most one transposition that reduces the number of breakpoints by six. Such a transposition is only possible if $S(1) \neq T(1)$, and $S(n) \neq T(n)$. Every other transposition can reduce the number of breakpoints by at most four. The theorem follows easily from these observations. □

This lower bound is not exact. For example, if $S = 011100110001$ and $T = 100011001110$, then the bound is 1, but $td(S, T) = 2$.

5.5.2 An upper bound

In this section a simple upper bound is derived for transposition distance.

Lemma 5.5.2 *Let S and T be related strings of length n, such that $S \neq T$. Then it is possible either:*
a) to apply a transposition to S resulting in a string S' such that $lcp(S', T) + lcs(S', T) \geq lcp(S, T) + lcs(S, T) + 2$, or
b) to apply a transposition to T resulting in a string T' such that $lcp(S, T') + lcs(S, T') \geq lcp(S, T) + lcs(S, T) + 2$.

Proof Without loss of generality, it can be assumed that $S(1) = 0$, $S(1) \neq T(1)$, and $S(n) \neq T(n)$. The transposition that is applied to S or T depends on S and T. Three cases are described, and for each case a transposition is shown with the required property. The three cases cover all possibilities for S and T, though the cases are not necessarily mutually exclusive.

Case (i). $S(n) = 1$: Then $S = 0 \ldots 1$ and $T = 1 \ldots 0$, so take $S' = [0 \sim 0][1 \ldots 1]$, where the '01' substring that is split apart by the transposition is the first '01' substring in S.

Case (ii). $f_{11}(S) > 0$: Then $S = 0 \ldots 11 \ldots 0$, and $T = 1 \ldots 1$ so take $S' = [00 \sim 01][1 \ldots 0]$, where the '11' substring that is split apart by the transposition, is the first '11' substring in S.

Case (iii). $f_{00}(T) > 0$: Then $S = 0 \ldots 0$, and $T = 1 \ldots 00 \ldots 1$ so, by considering S^{-1}, and T^{-1}, this case is similar to Case (ii).

These cases are sufficient to prove the lemma because Cases (i) and (ii) can only fail to apply if S contains more zeros than ones, whereas Cases (i) and (iii) can only fail to apply when T contains more ones than zeros. □

Theorem 5.5.2 *Let S and T be related binary strings, of length n. Then*

$$td(S,T) \leq \lfloor n/2 \rfloor.$$

Proof Lemma 5.5.2 describes a way to increase the the combined length of the common prefix and suffix of S and T by at least two using a single transposition. So a sequence of $\lfloor n/2 \rfloor$ such transpositions will be enough to transform S into T. □

5.5.3 Transposition diameter of $\mathcal{B}_n$

The *transposition diameter* of $\mathcal{B}_n$, $tD_2(n)$, is the maximum value of $td(S,T)$ taken over all related binary strings of length n. More formally

$$tD_2(n) = \max\{td(S,T) : S,\ T \text{ are related binary strings of length } n\}.$$

Lemma 5.5.3 *For all $k \geq 1$, $td(B_k, C_k) = k$, and $td(0 +\!\!+ B_k, 0 +\!\!+ C_k) = k$.*

Proof In both cases, this follows at once by application of Theorem 5.5.1, and Theorem 5.5.2. □

Theorem 5.5.3 *For all $n \geq 1$, $tD_2(n) = \lfloor n/2 \rfloor$.*

Proof This is an immediate consequence of Theorem 5.5.2 and Lemma 5.5.3. □

Theorem 5.5.4 *Let S and T be related binary strings of length $2n \geq 4$. Then $td(S,T) = n$, if and only if S and T are isomorphic to C_n and B_n.*

Proof We prove this theorem by induction. The base case is when $2n = 4$. Then the only strings for which $td(S,T) = 2$ are isomorphic to B_2 and C_2. Now suppose that the theorem holds for $n \leq k$. Let S and T be strings of length $2k+2$, such that $td(S,T) = k+1$. We show that S and T are isomorphic to C_{k+1} and B_{k+1}.

We can assume, without loss of generality, that $S(1) = 0$. By the proof of Lemma 5.5.2 we can assume that the transposition is applied to S resulting in the string S'. It must be that $lcp(S',T) + lcs(S',T) = 2$, and $td(S',T) = k$, since any alternative would contradict $td(S,T) = k+1$. Let S'_e and T_e be the strings S' and T excluding common prefix and suffix. By the induction hypothesis, S'_e and T_e must be isomorphic to B_k and C_k. Therefore, either:

a) $S'_e = B_k$ and $T_e = C_k$; or
b) $S'_e = C_k$ and $T_e = B_k$; or
c) $S'_e = \overline{B_k}$ and $T_e = \overline{C_k}$; or
d) $S'_e = \overline{C_k}$ and $T_e = \overline{B_k}$.

By the proof of Lemma 5.5.2, there are essentially two ways that the transposition can be applied to S, as typified by cases (i) and (ii) in that proof. We take each case in turn, and show that for the four possible values of S'_e and T_e, $td(S,T) = k+1$ if and only if S and T are isomorphic to B_{k+1} and C_{k+1}.

Case (i). $S = 0\ldots1$, $T = 1\ldots0$. In this case the transposition moves the block of zeros at the front of S to the end. We show that cases c) and d) for S'_e and T_e establish the induction step, whereas cases a) and b) lead to contradictions.

a) $S' = 1 +\!\!+ B_k +\!\!+ 0 = 100 \sim 011 \sim 10$ and $T = 1 +\!\!+ C_k +\!\!+ 0 = 11010 \sim 100$. So $S = 0100 \sim 011 \sim 1$. But then the transposition $0[100][\ldots011\ldots1]$ produces a string S'' such that $td(S'',T) < k$, by the induction hypothesis. So $td(S,T) < k+1$, giving a contradiction.

b) $S' = 1 +\!\!+ C_k +\!\!+ 0 = 11010 \sim 100$ and $T = 1 +\!\!+ B_k +\!\!+ 0 = 100 \sim 011 \sim 10$. Then $S = 0011010\ldots01$. But then the transposition $[0011010\ldots0][1]$ produces a string S'' such that $td(S'',T) < k$, by the induction hypothesis. So $td(S,T) < k+1$, giving a contradiction.

c) $S' = 1 +\!\!+ \overline{B_k} +\!\!+ 0 = \overline{B_{k+1}}$ and $T = 1 +\!\!+ \overline{C_k} +\!\!+ 0 = C_{k+1}$. So $S = B_{k+1}$.

d) $S' = 1 +\!\!+ \overline{C_k} +\!\!+ 0 = C_{k+1}$ and $T = 1 +\!\!+ \overline{B_k} +\!\!+ 0 = \overline{B_{k+1}}$. So $S = \overline{C_{k+1}}$.

Case (ii). $S = 0\ldots11\ldots0$, $T = 1\ldots1$. In this case the transposition is applied to S to obtain $S' = [0\ldots1][1\ldots0]$. Now S' must contain 00, so only cases a) and c) need to be considered. However, both cases lead to contradictions.

a) $S' = 1 +\!\!+ B_k +\!\!+ 1 = 100 \sim 011 \sim 11$ and $T = 1 +\!\!+ C_k +\!\!+ 1 = 11010 \sim 101$. However the transposition that splits the first '11' in S cannot produce a string like S'. So this case cannot occur.

c) $S' = 1 +\!\!+ \overline{B_k} +\!\!+ 1 = 111 \sim 100 \sim 01$ and $T = 1 +\!\!+ \overline{C_k} +\!\!+ 1 = 10101 \sim 011$. Then $S = 00 \sim 011 \sim 100 \sim 0$. However the transposition $00 \sim 011 \sim [11][00 \sim 0]$ produces a string S'' such that $td(S'',T) < k$, by the induction hypothesis. So $td(S,T) < k+1$, giving a contradiction.

So $td(S,T) = k+1$ if and only if S and T are isomorphic to B_{k+1} and C_{k+1}. Therefore by induction the theorem is true. □

5.5.4 Sorting by transpositions

We show that $td(S,S_\iota)$ can be determined in polynomial-time. The following lemma can be verified easily.

Lemma 5.5.4 *Let S' be a string obtained from S by a single transposition. Then*

$$z(S') \geq z(S) - 1.$$

With this lemma we can determine $td(S, S_\iota)$.

Theorem 5.5.5 *For any binary string S,*

$$td(S, S_\iota) = \begin{cases} z(S) - 1, & \text{if } S(1) = 0, \\ z(S), & \text{otherwise.} \end{cases}$$

Proof By Lemma 5.5.4, $td(S, S_\iota) \geq z(S) - 1$. If $S(1) = 0$, then $z(S) - 1$ transpositions of the form $0 \sim 0[1 \ldots 1][0 \sim 0]b \ldots$, where $b \in \{1, \omega\}$, transform S into S_ι. If $S(1) = 1$, an extra transposition is required, because is is impossible to change the character at the front of the string to 0 with a transposition, and also reduce the value of z. The bound in this case can be achieved by performing the transposition $[1 \ldots 1][0 \sim 0]b \ldots$, where $b \in \{1, \omega\}$, before the sequence of transpositions used when $S(1) = 0$. □

The distance described in Theorem 5.5.5 can be calculated easily in polynomial-time. The question of whether, in general, the transposition distance between any two strings can be calculated in polynomial-time remains open.

5.6 Block-Interchange distance between binary strings

In this section, we present a lower bound and an upper bound for block-interchange distance between binary strings. These bounds are then used to determine the block-interchange diameter, and also to identify some strings that achieve the diameter. A restricted version of the problem, that is in some ways analogous to sorting permutations by block-interchanges, is shown to be solvable in polynomial-time. However, in Section 5.7 the general problem of determining block-interchange distance between two strings is shown to be NP-hard.

5.6.1 A lower bound

In this section, the concept of a breakpoint is adapted to obtain a lower bound for block-interchange distance.

Block-interchange breakpoints are defined in exactly the same way as transposition breakpoints. So the number of block-interchange breakpoints (over a binary alphabet) is

$$\begin{aligned} bb(S,T) \;=\; & |f_{00}(S) - f_{00}(T)| + |f_{01}(S) - f_{01}(T)| + |f_{10}(S) - f_{10}(T)| \\ & + |f_{11}(S) - f_{11}(T)| + |f_{\alpha 0}(S) - f_{\alpha 0}(T)| + |f_{\alpha 1}(S) - f_{\alpha 1}(T)| \\ & + |f_{0\omega}(S) - f_{0\omega}(T)| + |f_{1\omega}(S) - f_{1\omega}(T)|. \end{aligned}$$

Clearly, $bb(T, T) = 0$. However it is possible that $bb(S, T) = 0$, even when $S \neq T$. For example, if $S = 101001$ and $T = 100101$, then $bb(S, T) = 0$.

Lemma 5.6.1 *Suppose that S' is obtained from S by a single block-interchange. Then*

$$bb(S', T) \geq bb(S, T) - 8.$$

Proof Four substrings of length two may change as the result of a block-interchange. So the frequency counts of substrings in S' are the same as the frequency counts of substrings in S except that the frequency counts of the four removed substrings are reduced by one, and the frequency counts of the four new substrings are increased by one. Given the definition of $bb(S', T)$, this means that $bb(S', T) \geq bb(S, T) - 8$. □

The following lower bound for block-interchange distance can be deduced easily from the above lemma.

Theorem 5.6.1 *Let S and T be related binary strings. Then*

$$bd(S, T) \geq \lceil bb(S, T)/8 \rceil.$$

This lower bound is not exact. For example, if $S = 011100110001$ and $T = 100011001110$, then the bound is one , whereas $bd(S, T) = 2$. In fact this lower bound can be strengthened, since a block interchange can only reduce the number of breakpoints by eight if f_{00} and f_{11} are both reduced by two, and f_{01} and f_{10} are both increased by two, or vice versa. However, this lower bound is sufficiently tight for our purposes.

5.6.2 An upper bound

In this section, a simple upper bound is derived for block-interchange distance.

Lemma 5.6.2 *Let S and T be related binary strings of length n, such that $S \neq T$ and $n \geq 4$. Then it is possible to apply either:*
a) a block-interchange to S obtaining S' such that $lcp(S', T) + lcs(S', T) \geq lcp(S, T) + lcs(S, T) + 4$, or
b) a block-interchange to T obtaining T' such that $lcp(S, T') + lcs(S, T') \geq lcp(S, T) + lcs(S, T) + 4$.

Proof Without loss of generality, it can be assumed that $S(1) = 0$, $S(1) \neq T(1)$, and $S(n) \neq T(n)$. If the strings are represented by their first two and last two characters then, up to isomorphism, all possible remaining combinations of S and T are represented by the 18 pairs shown in Table 5.1.

If $S(2) = 0$ then cases (i) to (xvi) cover all the possibilities for S and T. If S ends with 00 or 11, or T begins or ends with 00 or 11 then strings isomorphic to S and T are covered by one of the first sixteen cases. Otherwise there are only two possible cases remaining and these are cases (xvii) and (xviii).

Case	S	T
(i)	00...00	10...01
(ii)	00...00	10...11
(iii)	00...00	11...01
(iv)	00...00	11...11
(v)	00...01	10...00
(vi)	00...01	10...10
(vii)	00...01	11...00
(viii)	00...01	11...10
(ix)	00...10	10...01
(x)	00...10	10...11
(xi)	00...10	11...01
(xii)	00...10	11...11
(xiii)	00...11	10 ...00
(xiv)	00...11	10...10
(xv)	00...11	11...00
(xvi)	00...11	11...10
(xvii)	01...01	10...10
(xviii)	01...10	10...01

Table 5.1: The 18 possible combinations of S and T.

These eighteen cases are now investigated in turn and for each case, either a block-interchange is demonstrated that satisfies the statement of the lemma, or the case is shown to be similar to a previous case.

Case (i). S must contain at least two ones, so take $S' = [00\ldots01]\ldots[10\ldots00]$.

Case (ii). This case can be subdivided into two cases:
a) S contains 11 before 10: take $S' = [00\ldots11]\ldots[10\ldots00]$; else
b) T contains 00 before 00: take $T' = [10\ldots00]\ldots[00\ldots11]$.
These cases are exhaustive, because if a) fails then S contains more zeros than ones, whereas if b) fails then T contains at least as many ones as zeros.

Case (iii). $\overline{S} = 00\ldots00$ and $\overline{T} = 10\ldots11$ so this case is identical to Case (ii) by symmetry.

Case (iv). This case can be subdivided into two cases:
a) S contains 11 before 11: take $S' = [00\ldots11]\ldots[11\ldots00]$; else
b) T contains 00 before 00: take $T' = [11\ldots00]\ldots[00\ldots11]$.
These cases are exhaustive, because if a) fails then S contains more zeros than ones, whereas if b) fails then T contains more ones than zeros.

Case (v). This case can be subdivided into two cases:
a) S ends with 001: take $S' = [00\ldots00][1]$; else
b) S ends with 101: take $S' = [00]\ldots[101]$.
These cases are obviously exhaustive.

Case (vi). This case can be subdivided into two cases:
a) S contains 100: take $S' = [00\ldots10]0\ldots0[1]$; else
b) S ends with 101: take $S' = [00\ldots10][1]$.
These cases are exhaustive because if S does not contain 100 then S must end with 101.

Case (vii). Clearly S must contain another one since T contains two ones. So take $S' = [00\ldots00]1\ldots0[1]$.

Case (viii). This case can be subdivided into seven cases:
a) S contains 10 before 11: take $S' = [00\ldots10]\ldots[11\ldots01]$; else
b) S contains 111: take $S' = [0]0\ldots1[11\ldots01]$; else
c) S contains 101, but not only as a suffix: take $S' = [00\ldots10]1\ldots0[1]$; else
d) $S = 001101$ and $T = 110010$: take $S' = [00][11]01$; else
e) T contains 01 before 00: take $T' = [11\ldots01]\ldots[00\ldots10]$; else
f) T contains 000: take $T' = [1]1\ldots0[00\ldots10]$; else
g) T contains 010, but not only as a suffix: take $T' = [11\ldots01]0\ldots1[0]$.
These cases are exhaustive, because if a), b), c) and d) fail then S contains more zeros than ones, whereas if d), e), f) and g) fail then T contains more ones than zeros.

Case (ix). Clearly S must contain another one since T contains two ones. So take $S' = [00\ldots01]\ldots[10]$.

Case (x). This case can be subdivided into two cases:
a) S contains 11 before 10: $S' = [00\ldots11]\ldots[10\ldots10]$; else

b) T contains 10 before 00: $T' = [10 \dots 10] \dots [00 \dots 11]$.
These cases are exhaustive, because if a) fails S then contains more zeros than ones, whereas if b) fails then T contains more ones than zeros.

Case (xi). This case can be subdivided into two cases:
a) S contains 01 before 11: $S' = [00 \dots 01] \dots [11 \dots 10]$; else
b) T contains 10 before 00: $T' = [11 \dots 10] \dots [00 \dots 01]$.
These cases are exhaustive, because if a) fails S then contains more zeros than ones, whereas if b) fails then T contains more ones than zeros.

Case (xii). $\overline{S}^{-1} = 10 \dots 11$ and $\overline{T}^{-1} = 00 \dots 00$ so this case is isomorphic to Case (ii).

Case (xiii). $\overline{S} = 11 \dots 00$ and $\overline{T} = 00 \dots 01$ so this case is isomorphic to Case (vii).

Case (xiv). Take $S' = [0]0 \dots 1[1]$.

Case (xv). Take $S' = [00] \dots [11]$.

Case (xvi). $S^{-1} = 11 \dots 00$ and $T^{-1} = 00 \dots 01$ so this case is isomorphic to Case (vii).

Case (xvii). This case can be subdivided into two cases:
a) S contains 100: take $S' = [01 \dots 10]0 \dots 0[1]$; else
b) S ends with 101: take $S' = [01 \dots 10][1]$.
These cases are exhaustive, because if S does not contain 100 then it must end with 101.

Case (xviii). Take $S' = [01] \dots [10]$. □

Theorem 5.6.2 *Let S and T be related binary strings of length n. Then*

$$bd(S, T) \leq \lceil (n-1)/4 \rceil.$$

Proof Lemma 5.6.2 describes a way to extend the combined length of the common prefix and suffix of S and T by at least four using a single block-interchange. So a sequence of $\lceil (n-1)/4 \rceil$ such block-interchanges will be enough to transform S into T. □

5.6.3 Block-interchange diameter of $\mathcal{B}_n$

The *block-interchange diameter* of $\mathcal{B}_n$, $bD_2(n)$, is the maximum value of $bd(S, T)$ taken over all related binary strings S and T of length n. More formally

$$bD_2(n) = \max\{bd(S, T) : S, T \text{ are related binary strings of length } n\}.$$

Lemma 5.6.3 *For all $k \geq 1$, $bd(B_{2k}, C_{2k}) = k$, $bd(0 +\!\!+ B_{2k}, 0 +\!\!+ C_{2k}) = k$, $bd(B_{2k+1}, C_{2k+1}) = k + 1$, and $bd(0 +\!\!+ B_{2k+1}, 0 +\!\!+ C_{2k+1}) = k + 1$.*

Proof In each case, this follows at once by application of Theorem 5.6.1 and Theorem 5.6.2. □

Theorem 5.6.3 *For all* $n \geq 1$, $bD_2(n) = \lceil (n-1)/4 \rceil$.

Proof This is trivially true for $n = 1$. For $n \geq 2$, it is an immediate consequence of Theorem 5.6.2 and Lemma 5.6.3. □

Conjecture 5.6.1 *Let* S *and* T *be related binary strings of length* $4k+2$. *Then* $bd(S,T) = k+1$, *if and only if* S *and* T *are isomorphic to* C_{2k+1} *and* B_{2k+1}.

It is conjectured that for other lengths of strings, more pairs of strings achieve the block-interchange diameter.

5.6.4 Sorting by block-interchanges

We show that $bd(b, S_\iota)$ can be determined in polynomial-time. The following lemma can be verified easily.

Lemma 5.6.4 *Let* S' *be a string obtained from* S *by a single block-interchange. Then*

$$z(S') \geq z(S) - 2.$$

With this lemma we can determine $bd(S, S_\iota)$.

Theorem 5.6.4 *For any binary string* S,

$$bd(S, S_\iota) = \begin{cases} \lceil (z(S)-1)/2 \rceil, & \text{if } S(1) = 0 \\ \lceil z(S)/2 \rceil, & \text{otherwise} \end{cases}$$

Proof By Lemma 5.6.4, $bd(S, S_\iota) \geq \lceil (z(S)-1)/2 \rceil$. If $S(1) = 0$, and $z(S)$ is odd, then $\lceil (z(S)-1)/2 \rceil$ block-interchanges of the form $0 \sim 0[1 \ldots 1]0 \sim 01 \ldots 1[0 \sim 0]b \ldots$, where $b \in \{1, \omega\}$, transform S into S_ι. If $S(1) = 0$, and $z(S)$ is even, then not every block-interchange can reduce z by two, therefore at least $\lceil z(S)/2 \rceil$ block-interchanges are required. This bound can be achieved with a block-interchange of the form $0 \sim 0[1 \sim 1][0 \sim 0]b \ldots$, where $b \in \{1, \omega\}$ followed by $\lfloor (z(S)-1)/2 \rfloor$ block-interchanges like those used above when $z(S)$ was odd.

If $S(1) = 1$, an extra block-interchange may be required, because is is impossible to change the character at the front of the string to 0 with a block-interchange, and also reduce the value of z by two. If we apply the block-interchange $[1 \ldots 1]0 \sim 01 \ldots 1[0 \sim 0]b \ldots$ (or $[1 \sim 1][0 \sim 0]b \ldots$, when $z(S) = 1$) , where $b \in \{1, \omega\}$, before the sequence of $\lceil (z(S)-2)/2 \rceil$ block-interchanges used when $S(1) = 0$, the bound in this case, can be achieved. □

The distance described in Theorem 5.6.4 can be calculated easily in polynomial-time. In Section 5.7 it is shown that, in general, determining $bd(S,T)$ is NP-hard.

5.7 NP-completeness of reversal distance and block-interchange distance

In this section, we prove that the general problem of finding the reversal distance between two strings is NP-hard, even when the strings are drawn from a binary alphabet. We also prove a similar result for block-interchange distance.

We begin with a definition of the reversal distance problem as a decision problem (RD):

> **RD**
> Instance: Related strings S and T of length n, over an alphabet of size $t \geq 2$, and a bound $d \in Z^+$.
> Question: Is $rd(S,T) \leq d$?

We transform an NP-complete problem into RD to obtain the NP-completeness result. The problem we start from is 3-Partition:

> **3-Partition**
> Instance: A set A of $3m$ elements, a bound $B \in Z^+$, and a size $s(a) \in Z^+$ for each $a \in A$ such that $B/4 < s(a) < B/2$ and such that $\sum_{a \in A} s(a) = mB$.
> Question: Can A be partitioned into m disjoint sets A_1, A_2, ..., A_m such that, for $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$? (Note that each A_i must contain exactly three elements from A).

However, the transformation from 3-Partition to RD is a little unusual in that it is a pseudo-polynomial transformation. We now define pseudo-polynomial transformations, and also present the definition of NP-complete in the strong sense.

Let I be an instance of a decision problem that involves numbers. Then *Length*(I) is defined to be the size of the problem instance, and *Max*(I) is defined to be the magnitude of the largest number in the problem instance. A decision problem Π is *NP-complete in the strong sense* if there is a polynomial p such that Π_p is NP-complete, where Π_p is the problem Π restricted to instances I with $\mathrm{Max}(I) \leq p(\mathrm{Length}(I))$.

The following lemma is due to Garey and Johnson [GJ75] (see also Chapter 4.2 of [GJ79]).

Lemma 5.7.1 *3-Partition is NP-complete in the strong sense.*

Let Π and Π' denote arbitrary decision problems with instance sets D_Π and $D_{\Pi'}$, and yes sets Y_Π and $Y_{\Pi'}$, with specified functions Max and Length, and Max$'$ and Length$'$ respectively. A *pseudo-polynomial transformation* from Π to Π' is a function $f : D_\Pi \rightarrow D_{\Pi'}$ such that:

a) for all $I \in D_\Pi$, $I \in Y_\Pi$ if and only if $f(I) \in Y_{\Pi'}$.

b) f can be computed in time polynomial in the two variables $\mathrm{Max}(I)$ and $\mathrm{Length}(I)$.

c) there exists a polynomial q_1 such that for all $I \in D_\Pi$

$$q_1(\mathrm{Length}'(f(I))) \geq \mathrm{Length}(I).$$

d) there exists a two variable polynomial q_2 such that, for all $I \in D_\Pi$

$$\mathrm{Max}'(f(I)) \leq q_2(\mathrm{Max}(I), \mathrm{Length}(I)).$$

Lemma 5.7.2 *If* Π *is NP-complete in the strong sense,* $\Pi' \in NP$*, and there exists a pseudo-polynomial transformation from* Π *to* Π'*, then* Π' *is NP-complete (in the strong sense).*

Proof This is proved in Chapter 4.2 of [GJ79]. Essentially the pseudo-polynomial transformation is a polynomial-time transformation for instances of Π_p. □

We can now prove the NP-completeness result for reversals.

Theorem 5.7.1 *RD is NP-complete, even when* $t = 2$.

Proof RD is in NP because, given a sequence of reversals, it can easily be checked in polynomial-time that the sequence transforms S into T and has length at most d. By Lemma 5.7.2, the proof will be completed by demonstrating a pseudo-polynomial transformation from 3-Partition to RD. We now describe the transformation.

Let I be an instance of 3-Partition. From this instance construct an instance, I', of RD with $S = (+\!\!+_{i=1}^{3m-1} 0^{s(a_i)} 1^3) +\!\!+\, 0^{s(a_{3m})}$, $T = (+\!\!+_{i=1}^{m} 0^B 1) +\!\!+\, 1^{8m-3}$, and $d = 3m - 1$. So the blocks of zeros in S represent elements of A, and the lengths of the blocks represent the sizes of the elements.

We first show that $rd(S, T) \geq 3m - 1$.

Let U be a string that is related to S and T. Recall that $z(U)$ is the number of blocks of zeros in the string U. Define $o(U)$ as the number of blocks of ones of length one in U. Then the function $f(U) = z(U) - o(U) - 1$ can be viewed as a kind of distance function between strings U and T, since $f(T) = 0$. Furthermore, $f(S) = 3m - 1$. We show that, if U' is obtained from U by applying a single reversal, then $f(U') \geq f(U) - 1$.

Suppose that ρ is a reversal that transforms U into U' with $f(U') < f(U)$. Then ρ must reduce the number of blocks of zeros or increase the number of blocks of ones of length one.

If ρ reduces the number blocks of zeros then it must have the form $\ldots 1[0 \ldots 1]0 \ldots$ or $\ldots 0[1 \ldots 0]1 \ldots$, so $f(U) - f(U') = 1$. So it is impossible for ρ to increase the number of blocks of ones of length one as well as reduce the number of blocks of zeros.

If ρ increases the number of blocks of ones of length one then it must be of the form $\ldots 01[10\ldots 0]0\ldots$, or $\ldots 1[10\ldots 11]0\ldots$, or $\ldots 01[1\ldots 0]11\ldots$, or the mirror image of one of these three reversals. (Note, the reversals $\ldots 01[11\ldots 0]0\ldots$, and $\ldots 11[10\ldots 0]0\ldots$ do not reduce the value of f.) In each case $f(U) - f(U') = 1$. So $rd(S,T) \geq 3m-1$.

Note that the first of the reversals in the previous paragraph is special because it increases the number of blocks of length one by two, but also increases the number of blocks of zeros. We call this kind of reversal a *bad reversal*.

We now show that the given transformation from 3-Partition to an instance of RD is a pseudo-polynomial transformation. We verify the four properties, a), b), c) and d) in turn.

For property a), we have to show that I is a yes instance of 3-Partition if and only if $rd(S,T) \leq 3m-1$.

We have already shown that $rd(S,T) \geq 3m-1$. We now show that, if $rd(S,T) = 3m-1$, then no minimal length sequence of reversals that transform S into T contains a bad reversal.

Suppose that $rd(S,T) = 3m-1$. Every reversal in a minimal length sequence that transforms S into T must reduce the value of f by one. For a reversal to be bad the string must contain 0110 as a substring. However S contains no such substring, and no reversal that reduces the value of f by one can create such a substring. So if $rd(S,T) = 3m-1$, then no minimal length sequence of reversals that transforms S into T contains a bad reversal.

This means that, if $rd(S,T) = 3m-1$, each block of zeros in T is constructed from three blocks of zeros in S. It follows that I is a yes instance of 3-Partition if $rd(S,T) = 3m-1$.

Now we show that if I is a yes instance of 3-Partition then $rd(S,T) \leq 3m-1$. Since I is a yes instance, we can partition A into m disjoint sets A_1, ..., A_m, each of which contains three elements, and sums to B. For each subset A_i in turn, we can use two reversals, of the form $\ldots 0[1\ldots 0]a\ldots$, where $a \in \{1, \omega\}$ (where ω is the special character used to denote the end of the string), to merge the three blocks of zeros representing the elements of A_i into a single block of zeros of length B, without affecting any other block of zeros. (Note that the reversals shown do not move the block of zeros at the front of the string). Then we can use $m-1$ reversals, of the form $\ldots 01[11\ldots 0]1\ldots$, to create blocks of ones of length one separating the blocks of zeros. This sequence of reversals has length $3m-1$, so $rd(S,T) \leq 3m-1$. This establishes the required property a).

To prove properties b), c) and d) we need Length and Max functions for 3-Partition and RD. For 3-Partition, reasonable definitions are $\text{Length}(I) = |A| + \sum_{a \in A} \lceil \log_2 s(a) \rceil$, and $\text{Max}(I) = \max\{s(a) : a \in A\}$ [GJ79]. For RD, reasonable definitions are, for example, $\text{Length}'(I') = 2n + \lceil \log_2 d \rceil$, and $\text{Max}'(I') = 1$ (since RD is not a number problem). Note that $n = \sum_{a \in A} s(a) + |A| - 3$.

Given these functions, properties b), c) and d) can be proved quite easily.

So the transformation is a pseudo-polynomial time transformation, and therefore, by Lemma 5.7.2, RD is NP-complete. □

The transformation just described was obtained after several other simpler transformations had been shown to fail. For example, we tried the following transformation from sorting by reversals to *RD*. Given a permutation π, define strings $S = (+\!+_{i=1}^{n-1} 0^{\pi(i)}1) +\!+ 0^{\pi(n)}$, and $T = (+\!+_{i=1}^{n-1} 0^{i}1) +\!+ 0^{n}$. One might conjecture that determining the value of $rd(S,T)$ must also determine the value of $rd(\pi)$. However, if $\pi = 3142$, then $rd(\pi) = 3$, but $S = 0001010000100$, $T = 0100100010000$ and $rd(S,T) = 2$. So this transformation does not work.

We now prove that the general problem of finding the block-interchange distance between two strings is NP-hard, even when the strings are drawn from a binary alphabet. The block-interchange distance problem can be defined as a decision problem (BD), as follows:

> **BD**
> Instance: Related strings S and T of length n, over an alphabet of size $t \geq 2$, and a bound $d \in Z^+$.
> Question: Is $bd(S,T) \leq d$?

This time the transformation starts from a special case of 3-Partition called Odd-3-Partition.

> **Odd-3-Partition**
> Instance: A set A of $3m$ elements, where m is odd , a bound $B \in Z^+$, and a size $s(a) \in Z^+$ for each $a \in A$ with $B/4 < s(a) < B/2$ and $\sum_{a \in A} s(a) = mB$.
> Question: Can A be partitioned into m disjoint sets A_1, A_2, ..., A_m such that, for $1 \leq i \leq m$, $\sum_{a \in A_i} s(a) = B$? (Note that each A_i must contain exactly three elements from A)

We now show that this special case of 3-Partition is NP-complete in the strong sense.

Lemma 5.7.3 *Odd-3-Partition is NP-complete in the strong sense.*

Proof Let I be an instance of 3-Partition. We transform I into an instance of Odd-3-Partition. If m is odd then I is already an instance of Odd-3-Partition. Otherwise, let $m' = m + 1$, $A' = A \cup \{a_{3m+1}, a_{3m+2}, a_{3m+3}\}$, $s'(a) = 3.s(a)$ for $a \in A$, $s'(a_{3m+1}) = s'(a_{3m+2}) = B - 1$, $s'(a_{3m+3}) = B + 2$, and $B' = 3B$. Then A', m', s', and B' define an instance of Odd-3-Partition. It is easy to verify that this is a yes instance of Odd-3-Partition if and only if I is a yes instance of 3-Partition. The transformation can be

performed in polynomial-time, therefore Odd-3-Partition must be NP-complete in the strong sense. □

Theorem 5.7.2 *BD is NP-complete, even when* $t = 2$.

Proof We prove this theorem using a transformation that is very similar to the one used in the proof of Theorem 5.7.1.

BD is in NP because, given a sequence of block-interchanges, it can easily be checked in polynomial-time that the sequence transforms S into T and has length at most d. By Lemma 5.7.2 the proof is completed by demonstrating a pseudo-polynomial transformation from Odd-3-Partition to BD. The transformation is now described.

Let I be an instance of Odd-3-Partition. From this instance construct an instance I' of BD with $S = (+\!\!+_{i=1}^{3m-1} 0^{s(a_i)}1^3) +\!\!+\, 0^{s(a_{3m})}$, $T = (+\!\!+_{i=1}^{m} 0^B 1) +\!\!+\, 1^{8m-3}$, and $d = (3m-1)/2$. So the blocks of zeros in S represent elements of A and the lengths of the blocks represent the the sizes of the elements.

We now show that this transformation from Odd-3-Partition to BD is a pseudo-polynomial transformation. There are four properties a), b), c) and d), that must be verified, and because the transformation is so similar to the transformation used in Theorem 5.7.1 properties b), c) and d) follow easily from the proof of that theorem.

For property a), we have to show that I is a yes instance of Odd-3-Partition if and only if $bd(S,T) \leq (3m-1)/2$.

First of all we prove that $bd(S,T) \geq (3m-1)/2$.

As in the proof of Theorem 5.7.1, $f(U) = z(U) - o(U) - 1$. We show that, for any block-interchange that transforms U into U', $f(U') \geq f(U) - 2$. This will prove that $bd(S,T) \geq (3m-1)/2$, since $f(S) = 3m-1$, and $f(T) = 0$.

Suppose ρ is a block-interchange that transforms U into U' such that $f(U') < f(U)$. Then ρ must reduce the number of blocks of zeros, or increase the number of blocks of ones of length one, or do both.

Suppose ρ is a transposition such that $f(U') < f(U)$. If ρ reduces the number of blocks of zeros, then $f(U') = f(U)-1$ because a transposition can reduce the number of blocks of zeros by at most one (Lemma 5.5.4), and such a transposition cannot increase the number of blocks of ones of length one. If ρ increases the number of blocks of ones of length one then $f(U') \geq f(U)-2$, because a transposition can increase o by at most two. If U does not contain any 0110 substrings then then $f(U') = f(U) - 1$ because then ρ can only increase o by one without increasing z, or it can increase o by two while also increasing z by one, e.g., $\ldots 11[10\ldots01][11\ldots0]0\ldots$. If U does contain a 0110 substring then $f(U') \geq f(U) - 2$, and this bound can be achieved by a transposition that creates two blocks of ones of length one without changing the number of blocks of zeros, e.g., $\ldots 01[10\ldots111][0\ldots0]11\ldots$.

If ρ is not a transposition then ρ must have the form

$$\ldots a[b\ldots c]d\ldots e[f\ldots g]h\ldots = \ldots af\ldots gd\ldots eb\ldots ch.$$

Note that changes at ab and ef happen independently of changes at cd and gh. Suppose that the number of blocks of zeros is decreased by the action of ρ at ab and ef. Then ρ must have the form $\ldots 0[1\ldots]\ldots 1[0\ldots]\ldots$, or alternatively the form $\ldots 1[0\ldots]\ldots 0[1\ldots]\ldots$. Therefore, f is decreased by one as a result of this part of the block-interchange. Now suppose that the number of blocks of ones of length one is increased by the action of ρ at ab and ef. Then ρ has the form:

$$\begin{array}{ll}
(i) & \ldots 01[1\ldots]\ldots 11[0\ldots]\ldots, \\
(ii) & \ldots 11[0\ldots]\ldots 01[1\ldots]\ldots, \\
(iii) & \ldots 0[11\ldots]\ldots 1[10\ldots]\ldots, \\
(iv) & \ldots 1[10\ldots]\ldots 0[11\ldots]\ldots, \\
(v) & \ldots 01[10\ldots]\ldots 0[0\ldots]\ldots, \text{ or} \\
(vi) & \ldots 0[0\ldots]\ldots 01[10\ldots]\ldots.
\end{array}$$

In each case f is reduced by one as a result of this part of the block-interchange. Note that other apparent possibilities, e.g., $\ldots 01[11\ldots]\ldots 0[0\ldots]\ldots$, increase the number of blocks of zeros and hence do not decrease f. The changes that can happen at cd and gh are identical to those that can happen at ab and ef, therefore f can be reduced overall by two by a single block-interchange. Hence $bd(S,T) \geq (3m-1)/2$.

A block-interchange is *bad* if it splits a 00 substring or it is a transposition. Suppose that $bd(S,T) = (3m-1)/2$. Then we show that no sequence of block-interchanges that achieves this bound contains a bad block-interchanges. Clearly every block-interchange in the sequence that transforms S into T must reduce f by two. For a block-interchange in a minimal length sequence to be bad, the string must contain 0110 as a substring. However S contains no such substring, and no block-interchange that reduces f by two can create such a substring. So if $bd(S,T) = (3m-1)/2$, then no shortest sequence of block-interchanges that transforms S into T contains a bad block-interchange.

Therefore, if $bd(S,T) = (3m-1)/2$, then any minimum length sequence of block-interchanges must assemble each block of zeros in T out of three blocks of zeros in S. So if $bd(S,T) = (3m-1)/2$ then I is a yes instance of Odd-3-Partition.

We now show that if I is a yes instance of Odd-3-Partition, then $bd(S,T) \leq (3m-1)/2$. For each subset A_i use a block-interchange of the form $\ldots 0[1\ldots 1]0\ldots 1[0 \sim 0]1\ldots$ to merge the three blocks of zeros representing elements of A_i into a single block of zeros. Then use $(m-1)/2$ block-interchanges of the form $\ldots 01[1\ldots 1]10 \sim 01\ldots 1[0 \sim 0]1$, to create the blocks of ones of length one separating the blocks of zeros.

This establishes the required property a). So, by Lemma 5.7.2, BD is NP-complete. □

5.8 Conclusion

In this chapter we have shown that, just as sorting permutations by reversals is NP-hard, so also is finding the reversal distance between two strings, even when the

strings are drawn from a binary alphabet. We have also shown that finding the block-interchange distance between two strings is NP-hard. This contrasts with sorting permutations by block-interchanges, for which a polynomial-time algorithm was presented in Chapter 4.

The complexity of finding the transposition distance between two strings remains open, just as the complexity of sorting permutations by transpositions is open. The complexity of finding the prefix-reversal distance between two strings (over a fixed size alphabet) is also open, unlike the complexity of sorting permutations by prefix-reversals, which is known to be NP-hard.

For all four problems on strings, we derived lower and upper bounds for the distance between binary strings, and used these bounds to find the diameter of $\mathcal{B}_n$.

A summary of the known complexity of the distance problems on strings (over a fixed size alphabet) and the distance problems on permutations is presented in Table 5.2. The table also shows the diameter of $\mathcal{B}_n$ under each of the operations.

	Reversals	Prefix-Reversals	Transpositions	Block-Interchanges
permutations	NP-hard	NP-hard	Open	P
strings	NP-hard	Open	Open	NP-hard
diameter of $\mathcal{B}_n$	$\lfloor n/2 \rfloor$	$n-1$	$\lfloor n/2 \rfloor$	$\lceil (n-1)/4 \rceil$

Table 5.2: The complexity of the different problems, and the diameter of $\mathcal{B}_n$.

Pevzner and Waterman discuss the problem of sorting strings by reversals (they call the problem sorting words by reversals) in their open problems paper [PW95]. Their Problem 4 is to devise a performance guarantee algorithm for sorting words by reversals. Finding an algorithm with a performance guarantee for any of the distance problems on strings remains an open problem.

Glossary

Symbol	*Short description*	*Page*
B_k	$B_k = 0^k +\!\!+ 1^k$	115
BR_n	the broken permutation of length n	82
$\mathcal{B}_n$	the set of binary strings of length n	115
$bb(S,T)$	the number of block-interchange breakpoints	131
$bD(n)$	the block-interchange diameter of S_n	110
$bD_2(n)$	the block-interchange diameter of $\mathcal{B}_n$	135
$bd(S,T)$	the block-interchange distance between strings S and T	114
$bd(\pi)$	the block-interchange distance of π	12
b_i	the black edge $(\pi(i), \pi(i-1))$	50
$b(S)$	the number of blocks in S	116
C_k	$C_k = (10)^k$	115
$C_{\max}$	the position of the rightmost edge of cycle C	51
$C_{\min}$	the position of the leftmost edge of cycle C	51
$\lvert\mathcal{C}\rvert$	the number of cycles in $\mathcal{C}$	27
$\mathcal{C}_M$	a cycle decomposition derived from M	38
$\mathcal{C}^+$	the augmented cycle decomposition derived from $\mathcal{C}$	18
$\mathcal{C} \cdot \rho(u)$	the cycle decomposition derived from $\mathcal{C}$ after applying $\rho(u)$	20
$f_{ab}(S)$	the number of occurrences of the substring 'ab' in S	117
$gl(\pi)$	the 'glued' permutation derived from π	48
$g(v)$	the grey edge of $rG'(\pi)$ associated with v	18
$I(C)$	the inner sequence of cycle C	55
$i(C,k)$	the kth element of the inner sequence of C	55
$\imath$	the identity permutation	2
$\imath_n$	the identity permutation of length n	2
$+\imath$	$+\imath = [+1\ +2\ \ldots\ +n]$	6
$-\imath$	$-\imath = [-1\ -2\ \ldots\ -n]$	9
K_M	the subgraph of $rR(\mathcal{C}_M)$ that is related to kernel K	45
$L(C)$	the canonical labelling of cycle C	55
$l_b(u)$	the position of the leftmost black edge adjacent to $g(u)$ in $\mathcal{C}$	18
$l(C)$	the length of cycle C	50
$lcp(S,T)$	the length of the longest common prefix of S and T	116

$lcs(S,T)$	the length of the longest common suffix of S and T	116
$l_g(u)$	the position of the leftmost element incident to $g(u)$	18
$lis(\pi)$	the length of the longest increasing subsequence of π	11
n	the length of a permutation	2
$O(C)$	the outer sequence of cycle C	55
$o(C,k)$	the kth element of the outer sequence of C	55
$o(S)$	the number of blocks of ones of length one in S	138
$prb(\pi)$	the number of prefix reversal breakpoints in π	8
$prD(n)$	the prefix reversal diameter of S_n	8
$prD_2(n)$	the prefix reversal diameter of $\mathcal{B}_n$	124
$prd(S,T)$	the prefix reversal distance between strings S and T	114
$prd(\pi)$	the prefix reversal distance of π	8
$p(\pi,u)$	the position of the edge u in $tG(\pi)$	52
R_n	the reverse permutation of length n	11
$+R_n$	$+R_n = [+n\ +(n-1)\ \ldots\ +1]$	7
$\mathcal{R}(\pi)$	set of 2-reversals on π	37
$rb(S,T)$	the number of reversal breakpoints	117
$r_b(u)$	the position of the rightmost black edge incident to $g(u)$ in $\mathcal{C}$	18
$rb(\pi)$	the number of reversal breakpoints in π	4
$rc(\pi)$	the size of a maximum alternating cycle decomposition of $rG(\pi)$	5
$rc_2(\pi)$	the least number of 2-cycles in any maximum cycle decomposition	28
$rc_{3*}(\pi)$	the number of cycles of length greater than 2	28
$rD(n)$	the reversal diameter of S_n	5
$rD_2(n)$	the reversal diameter of $\mathcal{B}_n$	119
$rd(S,T)$	the reversal distance between strings S and T	114
$rd(\pi)$	the reversal distance of π	3
$rF(\pi)$	the matching graph	29
$rF^*(\pi)$	the bridge free subgraph of $rF(\pi)$	39
$rf(\pi)$	the fortress value of π	7
$rG(\pi)$	the reversal cycle graph of π	4
$rG'(\pi)$	the augmented reversal cycle graph of π	18
$r_g(u)$	the position of the rightmost element adjacent to $g(u)$	18
$rh(\pi)$	the number of hurdles in $rG(\pi)$	7
$rL(M)$	the ladder graph	29
$rR(\mathcal{C})$	the reversal graph based on $\mathcal{C}$	18
$ru(\mathcal{C})$	the number of unoriented components in $rR(\mathcal{C}^+)$	27
$\overline{S}$	the string S in reverse order	115
S_ι	the string related to S that has the form $0 \sim 01 \sim 1$	121
S^k	the string obtained by concatenating k copies of S	115
S_n	the symmetric group	2
S^{-1}	the string obtained by swapping 1s and 0s in S	115

$sd(X, Y)$	the syntenic edit distance between X and Y	14
$sprD(n)$	the signed prefix reversal diameter	9
$sprd(\pi)$	the signed prefix reversal distance of π	8
$srD(n)$	the signed reversal diameter of Σ_n	7
$srd(\pi)$	the signed reversal distance of π	6
$tb(S, T)$	the number of transposition breakpoints	127
$tb(\pi)$	the number of transposition breakpoints in π	10
$tc(\pi)$	the number of alternating cycles in $tG(\pi)$	50
$tc_{odd}(\pi)$	the number of odd length alternating cycles in $tG(\pi)$	10
$tc_{even}(\pi)$	the number of even length alternating cycles in $tG(\pi)$	50
$tD(n)$	the transposition diameter of S_n	10
$tD_2(n)$	the transposition diameter of $\mathcal{B}_n$	129
$td(S, T)$	the transposition distance between strings S and T	114
$td(\pi)$	the transposition distance of π	10
$tG(\pi)$	the transposition cycle graph of π	10
$tH(\pi)$	the transposition cycle overlap graph of π	77
$th(\pi)$	the number of hurdles in $tG(\pi)$	78
$th_d(\pi)$	the number of detectable hurdles in $tG(\pi)$	79
$tld(A, B)$	the reciprocal translocation distance between A and B	14
$tl(\pi)$	a lower bound for $td(\pi)$	79
u_i	a vertex of $rR(\mathcal{C})$	18
$z(S)$	the number of blocks of zeros in S	116
$\sim$	a symbol used to represent repetition in strings	116
$+\!\!+$	concatenation operator	34
$(i_1, \ldots, i_k)$	an alternating cycle	51
α	a special character used to represent the start of a string	117
$\beta(i, j, k, l)$	a block-interchange	107
$\Delta f(\pi, \tau)$	The change in f as a result of applying τ to π	48
κ_k	a permutation with k knots in $tG(\pi)$	79
$\|\pi\|$	the length of π	2
$\pi_{\mathcal{C}}$	the component permutation	78
$\pi \cdot \gamma$	the permutation obtained by applying γ to π	3
$\rho(i, j)$	a reversal	3
$\rho(u)$	the reversal represented by u	20
Σ_n	the set containing all signed permutations of length n	7
$\tau(i, j, k)$	a transposition	9
ω	a special character used to represent the end of a string	117
ω_r	the permutation of length $2r$ whose cycle graph is a knot	58

Bibliography

[ABIR89] N. Amato, M. Blum, S. Irani, and R. Rubinfeld. Reversing trains: A turn of the century sorting problem. *Journal of Algorithms*, 10:413–428, 1989.

[AD86] D. Aldous and P. Diaconis. Shuffling cards and stopping times. *American Mathematical Monthly*, 93(5):333–348, 1986.

[AL90] A. Aggarwal and T. Leighton. A tight lower bound for the train reversal problem. *Information Processing Letters*, 35:301–304, 1990.

[AW87] M. Aigner and D. B. West. Sorting by insertion of leading elements. *Journal of Combinatorial Theory, Series A*, 45:306–309, 1987.

[BH96] P. Berman and S. Hannenhalli. Fast sorting by reversals. In *Combinatorial Pattern Matching, Proceedings of the 7th Annual Symposium (CPM'96)*, number 1075 in Lecture Notes in Computer Science, pages 168 – 185, 1996.

[BKS96] M. Blanchette, T. Kunisawa, and D. Sankoff. Parametric genome rearrangement. *Gene*, 172:GC 11–GC 17, 1996.

[BP93] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. In *Proceedings of the 34th IEEE Symposium on Foundations of Computer Science*, pages 148–157, 1993.

[BP95a] V. Bafna and P. A. Pevzner. Sorting by reversals: Genome rearrangements in plant organelles and evolutionary history of X chromosome. *Molecular Biology and Evolution*, 12(2):239–246, 1995.

[BP95b] V. Bafna and P. A. Pevzner. Sorting by transpositions. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 614–623, 1995.

[BP96] V. Bafna and P. A. Pevzner. Genome rearrangements and sorting by reversals. *SIAM Journal on Computing*, 25(2):272–289, 1996.

[BP98] V. Bafna and P. A. Pevzner. Sorting by transpositions. *SIAM Journal on Discrete Mathematics*, 11(2):224–240, 1998.

[Cap97a] A. Caprara. Formulations and complexity of multiple sorting by reversals. Technical Report OR-97-15, DEIS - Operations Research Group, University of Bologna, 1997.

[Cap97b] A. Caprara. Sorting by reversals is difficult. In *Proceedings of the First International Conference on Computational Molecular Biology (RECOMB'97)*, pages 75–83. ACM Press, 1997.

[Cap97c] A. Caprara. Sorting permutations by reversals and eulerian cycle decompositions. Technical Report OR-97-4, DEIS - Operations Research Group, University of Bologna, 1997.

[Cap98] A. Caprara. On the tightness of the alternating-cycle lower bound for sorting by reversals. Technical Report OR-98-4, DEIS - Operations Research Group, University of Bologna, 1998.

[CB95] D. S. Cohen and M. Blum. On the problem of sorting burnt pancakes. *Discrete Applied Mathematics*, 61:105–120, 1995.

[Chr96] D. A. Christie. Sorting permutations by block-interchanges. *Information Processing Letters*, 60(4):165–169, 1996.

[Chr98] D. A. Christie. A 3/2-approximation algorithm for sorting by reversals. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 244–252, 1998.

[CKb] Just be.

[CLN95] A. Caprara, G. Lancia, and S. K. Ng. A column-generation based branch-and-bound algorithm for sorting by reversals. Presented at the DIMACS 4th Annual Implementation Challenge Workshop, 1995. Also Technical Report OR-95-10, DEIS - Operations Research Group, University of Bologna.

[CS95] T. Chen and S. S. Skiena. Sorting with fixed-length reversals. Technical Report 95-18, DIMACS, June 1995.

[CS96] T. Chen and S. S. Skiena. Sorting with fixed-length reversals. *Discrete Applied Mathematics*, 71:269–295, 1996.

[Dew87] A. K. Dewdney. Computer recreations. *Scientific American*, 256(6):128–129, 1987.

[DJK+97] B. DasGupta, T. Jiang, S. Kannan, M. Li, and Z. Sweedyk. On the complexity and approximation of syntenic distance. In *Proceedings of the 1st Annual International Conference on Computational Molecular Biology (RECOMB'97)*, pages 99–108, 1997.

[DMP95] P. Diaconis, M. McGrath, and J. Pitman. Riffle shuffles, cycles, and descents. *Combinatorica*, 15:11–29, 1995.

[Dwe75] H. Dweighter. *American Mathematical Monthly*, 82:1010, 1975.

[EG81] S. Even and O. Goldreich. The minimum-length generator sequence problem is NP-hard. *Journal of Algorithms*, 2:311–313, 1981.

[Eid86] J. A. Eidswick. Cubelike puzzles—what are they and how do you solve them? *American Mathematical Monthly*, 93(3):157–176, 1986.

[FHL80] M. Furst, J. Hopcroft, and E. Luks. Polynomial-time algorithms for permutation groups. In *Proceedings of the 21st IEEE Symposium on Foundations of Computer Science*, pages 36–41, 1980.

[FNS96] V. Ferretti, J. H. Nadeau, and D. Sankoff. Original synteny. In *Combinatorial Pattern Matching, Proceedings of the 7th Annual Symposium (CPM'96)*, number 1075 in Lecture Notes in Computer Science, pages 159 – 167, 1996.

[Fri98] T. Frings. Approximationsalgorithmen für das Problem "Sorting by Reversals" auf Permutationen. Diplomarbeit, Rheinische Friedrich-Wilhelms-Universität Bonn, March 1998.

[GHV95] S. A. Guyer, L. S. Heath, and J. P. C. Vergara. Subsequence and run heuristics for sorting by transpositions. Presented at the DIMACS 4th Annual Algorithm Implementation Challenge, 1995. Also Technical Report TR-97-20, Virginia Tech Department of Computer Science.

[GJ75] M. R. Garey and D. S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal of Computing*, 4:397–411, 1975.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[GP79] W. H. Gates and C. H. Papadimitriou. Bounds for sorting by prefix reversal. *Discrete Mathematics*, 27:47–57, 1979.

[GPIC97] Q.-P. Gu, S. Peng, K. Iwata, and Q. M. Chen. A heuristic algorithm for genome rearrangements. In *Proceedings of the Genome Informatics Workshop*, Tokyo, 1997. Universal Academy Press. http://www.tokyo-center.genome.ad.jp/manuscripts/GIW97/Poster/GIW97P08.pdf.

[GPS96] Q.-P. Gu, S. Peng, and H. Sudborough. Approximation algorithms for genome rearrangements. In *Proceedings of the Genome Informatics Workshop*, Tokyo, 1996. Universal Academy Press. http://www.tokyo-center.genome.ad.jp/manuscripts/GIW96/Oral/GIW96O02.ps.

[GPS99] Q.-P. Gu, S. Peng, and H. Sudborough. A 2-approximation algorithm for genome rearrangements by reversals and transpositions. *Theoretical Computer Science*, 210(2):327–339, 1999.

[Gus97] D. Gusfield. *Algorithms On Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, Cambridge, 1997.

[Han95a] S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. In *Combinatorial Pattern Matching, Proceedings of the 6th Annual Symposium (CPM'95)*, number 937 in Lecture Notes in Computer Science, pages 162–176, 1995.

[Han95b] S. Hannenhalli. *Transforming men into mice (a computational theory of genome rearrangements).* PhD thesis, Pennsylvania State University Department of Computer Science and Engineering, 1995.

[Han96] S. Hannenhalli. Polynomial-time algorithm for computing translocation distance between genomes. *Discrete Applied Mathematics*, 71:137–151, 1996.

[HCKP95] S. Hannenhalli, C. Chappey, E. V. Koonin, and P. A. Pevzner. Genome sequence comparison and scenarios for gene rearrangements: A test case. *Genomics*, 30:299–311, 1995.

[HP95a] S. Hannenhalli and P. A. Pevzner. Towards a computational theory of genome rearrangements. *Lecture Notes in Computer Science*, 1000:184–202, 1995.

[HP95b] S. Hannenhalli and P. A. Pevzner. Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals). In *Proceedings of the 27th Annual ACM Symposium on Theory of Computing*, pages 178–189, 1995.

[HP95c] S. Hannenhalli and P. A. Pevzner. Transforming men into mice (polynomial algorithm for genomic distance problem). In *Proceedings of the 36th Annual IEEE Symposium on Foundations of Computer Science (FOCS'95)*, pages 581–592, 1995.

[HP96] S. Hannenhalli and P. A. Pevzner. To cut ... or not to cut (applications of comparative physical maps in molecular evolution). In *Proceedings of the*

7th Annual ACM-SIAM Symposium on Discrete Algorithms, pages 304–313, 1996.

[HS] M. H. Heydari and I. H. Sudborough. Pancake sorting is NP-complete. manuscript.

[HS97] M. H. Heydari and I. H. Sudborough. On the diameter of the pancake network. *Journal of Algorithms*, 25:67–94, 1997.

[HV95] L. S. Heath and J. P. C. Vergara. Some experiments on the sorting by reversals problem. Technical Report TR-95-16, Virginia Tech Department of Computer Science, September 1995.

[HV97] L. S. Heath and J. P. C. Vergara. Sorting by bounded block-moves. Technical Report TR-97-09, Virginia Tech Department of Computer Science, May 1997. To appear in Discrete Applied Mathematics.

[HV98] L. S. Heath and J. P. C. Vergara. Sorting by short block-moves. Technical Report TR-98-03, Virginia Tech Department of Computer Science, February 1998.

[IC95] R. W. Irving and D. A. Christie. Sorting by reversals: on a conjecture of Kececioglu and Sankoff. Technical Report TR-95-12, Department of Computing Science of The University of Glasgow, May 1995.

[Jer85] M. R. Jerrum. The complexity of finding minimum-length generator sequences. *Theoretical Computer Science*, 36:265–289, 1985.

[Jor95] E. M. Jordan. Visualising measures of genetic distance. Presented at the DIMACS 4th Annual Implementation Challenge Workshop, 1995. http://medg.lcs.mit.edu/people/emjordan/final.ps.

[Knu73] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, first edition, 1973.

[Knu98] D. E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition, 1998.

[KR95] J. D. Kececioglu and R. Ravi. Of mice and men: Algorithms for evolutionary distances between genomes with translocation. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 604–613, 1995.

[KS93] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for the inversion distance between two chromosomes. In *Combinatorial Pattern*

Matching, Proceedings of the 4th Annual Symposium (CPM'93), volume 684 of *Lecture Notes in Computer Science*, pages 87–105, 1993.

[KS94] J. D. Kececioglu and D. Sankoff. Efficient bounds for oriented chromosome inversion distance. In *Combinatorial Pattern Matching, Proceedings of the 5th Annual Symposium (CPM'94)*, volume 807 of *Lecture Notes in Computer Science*, pages 307–325, 1994.

[KS95] J. Kececioglu and D. Sankoff. Exact and approximation algorithms for sorting by reversals, with application to genome rearrangement. *Algorithmica*, 13:180–210, 1995.

[KST97] H. Kaplan, R. Shamir, and R. E. Tarjan. Faster and simpler algorithm for sorting signed permutations by reversals. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 344–351, 1997.

[LP86] L. Lovász and M. D. Plummer. *Annals of Discrete Mathematics (29): Matching Theory.* Number 121 in North-Holland Mathematics Studies. North-Holland, Amsterdam, 1986.

[LW75] R. Lowrance and R. A. Wagner. An extension of the string-to-string correction problem. *Journal of the Association for Computing Machinery*, 22(2):177–183, April 1975.

[MWD97a] J. Meidanis, M. E. M. T. Walter, and Z. Dias. Distância de reversão de cromossomos cirulares com sinais. In *Proceedings of the 14th SEMISH - Software and Hardware Seminar*, 1997.

[MWD97b] J. Meidanis, M. E. M. T. Walter, and Z. Dias. Transposition distance between a permutation and its reverse. In *Proceedings of the 4th South American Workshop on String Processing*, pages 70–79, 1997.

[Pap94] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, Reading, Massachusetts, 1994.

[PW95] P. A. Pevzner and M. S. Waterman. Open combinatorial problems in computational molecular biology. In *Proceedings of the 3rd Israel Symposium on the Theory of Computing and Systems*, pages 158–173, 1995.

[SM97] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology.* PWS Publishing Company, Boston, 1997.

[Tra97] N. Tran. An easy case of sorting by reversals. In *Combinatorial Pattern Matching, 8th Annual Symposium*, volume 1264 of *Lecture Notes in Computer Science*, pages 83–89. Springer, 1997.

[Ver97] J. P. C. Vergara. *Sorting by Bounded Permutations*. PhD thesis, Virginia Tech Department of Computer Science, April 1997.

[Wag83] R. A. Wagner. On the complexity of the extended string-to-string correction problem. In D. Sankoff and J. B. Kruskal, editors, *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, pages 215–235. Addison-Wesley, Reading, Massachusetts, 1983.

[WDM98] M. E. M. T. Walter, Z. Dias, and J. Meidanis. Reversal and transposition distance of linear chromosomes. To be presented at SPIRE'98 - String Processing and Information Retrieval: A South American Symposium, September 1998.

[WEHM82] G. A. Watterson, W. J. Ewens, T. E. Hall, and A. Morgan. The chromosome inversion problem. *Journal of Theoretical Biology*, 99:1–7, 1982.

[WF74] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the Association for Computing Machinery*, 21(1):168–173, January 1974.