Sorting by Bounded Permutations

by

John Paul C. Vergara

Dissertation submitted to the faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science and Applications

APPROVED:

Lenwood S. Heath, Chairman

Donald C. S. Allison

Ezra A. Brown

Edward L. Green

Clifford A. Shaffer

April, 1997

Blacksburg, Virginia

Sorting by Bounded Permutations

by

John Paul C. Vergara

Committee Chairman: Lenwood S. Heath

Computer Science

(ABSTRACT)

Let P be a predicate applicable to permutations. A permutation that satisfies P is called a *generator*. Given a permutation π, $MinSort_P$ is the problem of finding a shortest sequence of generators that, when composed with π, yields the identity permutation. The length of this sequence is called the P distance of π. $Diam_P$ is the problem of finding the longest such distance for permutations of a given length. $MinSort_P$ and $Diam_P$, for some choices of P, have applications in the study of genome rearrangements and in the design of interconnection networks.

This dissertation considers generators that are swaps, reversals, or block-moves. Distance bounds on these generators are introduced and the corresponding problems are investigated. Reduction results, graph-theoretic models, exact and approximation algorithms, and heuristics for these problems are presented. Experimental results on the heuristics are also provided.

When the bound is a function of the length of the permutation, there are several sorting problems such as sorting by block-moves and sorting by reversals whose bounded variants are at least as difficult as the corresponding unbounded problems. For some bounded problems, a strong relationship exists between finding optimal sorting sequences and correcting the relative order of individual pairs of elements. This fact is used in investigating $MinSort_P$ and $Diam_P$ for two particular predicates.

A short block-move is a generator that moves an element at most two positions away from its original position. Sorting by short block-moves is solvable in polynomial time for two large classes of permutations: woven bitonic permutations and woven double-strip permutations. For general permutations, a polynomial-time $\frac{4}{3}$-approximation algorithm that computes short block-move distance is devised. The short block-move diameter for length-n permutations is determined.

A short swap is a generator that swaps two elements that have at most one element between them. A polynomial-time 2-approximation algorithm for computing short swap distance is devised and a class of permutations where the algorithm computes the exact short swap distance is determined. Bounds for the short swap diameter for length-n permutations are determined.

To Marichelle, Ian, and Camille

ACKNOWLEDGEMENTS

First, I thank my advisor, Dr. Lenny Heath, whose guidance was most valuable. His advise, patience, and friendship have made this academic pursuit both worthwhile and pleasant. I also wish to thank the following people for their comments and suggestions: my committee members, Dr. Donald Allison, Dr. Bud Brown, Dr. Ed Green, and Dr. Cliff Shaffer; and the members of the Computer Science Theory Group, Ben, Craig, Joe, Sanjay, Scott, and Sriram. I thank Scott Guyer, in particular, for his work on the ETD document class, and his patience and help with my various typesetting problems.

I thank the Department of Computer Science of Virginia Tech, particularly Dr. Verna Schuetz, for the financial support and teaching opportunities I have received. Also, I thank the Ateneo de Manila University, Fr. Ben Nebres, Dr. Mari-jo Ruiz, Eileen Lolarga, and Arnie Del Rosario, for their support and confidence, and for permitting me to go on study leave.

I thank my parents, Milo and Eden, my brothers, Mylo, Mimick, Romel, Jomel, and sisters, Mylene and Eileen, for their love and confidence. To Alan, Allan, Chinky, EJ, Girlie, Jess, Leah, Lucia, Mel, Migs, Rene, Rose, Topsie, Waldo, and all other members of the Filipino Student Association, I express special thanks for making my stay in Blacksburg pleasant and memorable. I have made several friends in the Department of Computer Science, among them, Andy, Ben, Olivier, Scott, and Siva; I thank them as well.

Finally, I wish to thank the Lord God Almighty, for making everything possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

In molecular biology, an organism is identified by the sequences of genes found in its chromosomes. A gene in a chromosome is in turn determined by its DNA sequence. The set of chromosomes for a particular species is called the *genome* of that species. A genome typically mutates through insertions, deletions, duplications, reversals, and block-moves of its fragments. A "fragment" occurs on two different levels: on the gene level, where a fragment means part of the DNA sequence of a gene; or on the genome level, where a fragment means a subsequence of adjacent genes on a chromosome.

It is useful to measure or estimate the distance between two genomes, that is, the number of mutations that occurred when one organism evolved into another. Phylogenetic trees of different species are constructed based on information about how closely their genomes are related [12, 13]. It has been observed that some species are very similar in genetic makeup and differ mainly in the order of genes in their genomes [14, 28]. When comparing such species, it can be assumed that the mutations occur only on the genome level and that genes are neither introduced nor removed. Under this assumption, one may think of a genome as a *permutation* of genes[1] and mutations as *rearrangements* applied to the permutation. When estimating the distance between two genomes (permutations),

[1] This is particularly appropriate for genomes with a single chromosome. When a genome consists of more than one chromosome, a *concatenation* of these chromosomes corresponds to the permutation.

the biologist wants to determine the minimum number of mutations (rearrangements) required to transform one into the other.

1.2 Group-Theoretic Formulation

We formulate this problem in terms of group theory. Most of the notation and terminology we use in this dissertation are consistent with usual group-theoretic usage found in the literature (such as [18]). Given the set $T_n = \{1, 2, \ldots, n\}$, a permutation π of T_n is a bijective function $\pi : T_n \to T_n$. The *symmetric group* S_n is the set of all the permutations of T_n. We can view a permutation $\pi \in S_n$ as an ordered arrangement of the elements in T_n where $\pi(i)$ is the element in position i. In this view, the integers $1, 2, \ldots, n$ are used to indicate both positions and elements; we shall use regular italics when indicating positions and boldface (e.g., $\mathbf{1}, \mathbf{2}, \ldots, \mathbf{n}$) when indicating elements. If n represents the number of genes that compose two different species, then the genome for a particular species is precisely a permutation in S_n.

Composition between permutations illustrates a rearrangement occurring on a genome. For example, suppose that $\pi, \gamma \in S_n$ are the permutations

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

and

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

Then,

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

That is, composing π with γ has the effect of switching the second and fourth elements in π. Notice that, in cycle notation, $\gamma = (2\ 4)$. We call the permutation γ a *generator*, to emphasize that it operates on or rearranges the elements in the permutation it is composed with. In this context, a generator maps positions to positions so none of the integers in γ are specified in boldface.

In the context of evolution, we wish to restrict the generators to a particular set R of plausible genome rearrangements (mutations). It is therefore appropriate to characterize this set, a subset of

S_n, using some predicate P:

$$R = \{\gamma \in S_n \mid P(\gamma) \text{ is true}\}.$$

An evolutionary sequence between two genomes π_1 and π_2 is a sequence of rearrangements

$$\gamma_1, \gamma_2, \ldots, \gamma_k$$

from R satisfying

$$\pi_1 \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_k = \pi_2.$$

Equivalently the sequence satisfies

$$(\pi_2^{-1} \cdot \pi_1) \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_k = \boldsymbol{\iota},$$

where $\boldsymbol{\iota}$ is the identity permutation

$$\begin{pmatrix} 1\ 2\ \ldots\ n \\ \mathbf{1\ 2\ \ldots\ n} \end{pmatrix}.$$

Finding an evolutionary sequence for two genomes is therefore equivalent to sorting the permutation $\pi_2^{-1} \cdot \pi_1$.

1.3 The Problems

It is clear that we are interested in a family of sorting problems, one problem for each predicate P. Also, as we want the sorting to be by a sequence of minimum length, we state the general optimization problem formally as follows:

MINIMUM SORTING BY P **(**$MinSort_P$**)**

INSTANCE: A permutation $\pi \in S_n$.

SOLUTION: A sequence of generators $\gamma_1, \gamma_2, ..., \gamma_k$ such that $P(\gamma_i)$ is true for all i with $1 \leq i \leq k$, $\pi \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_k = \boldsymbol{\iota}$, and k is as small as possible.

Previous work on $MinSort_P$ problems for various predicates P is motivated mainly by their applications in biology and are surveyed in Chapter 3.

Let $MinSort_P(\pi)$ denote the length of a shortest sorting sequence for π. We call this length the P *distance* of π and is exactly the integer k in the formulation of $MinSort_P$ above. A related

problem is that which determines, among all permutations[2] in S_n, one that yields the longest such distance. This distance is called the P *diameter* of S_n, and the problem is formulated as follows:

P **DIAMETER** ($Diam_P$)

INSTANCE: An integer n.

SOLUTION: $Diam_P(n) = \max_{\pi \in S_n} MinSort_P(\pi)$.

The significance of P diameter in biology is that it represents the worst-case distance between two genomes containing the same genes.

For a predicate P and integer n, the *Cayley graph* $\mathcal{G}^c(P, n) = (\mathcal{V}^c, \mathcal{A}^c)$ has the vertex set $\mathcal{V}^c = S_n$ and an arc $(\pi_1, \pi_2) \in \mathcal{A}^c$ exactly when $\pi_2 = \pi_1 \cdot \gamma$ for some γ where $P(\gamma)$ is true. $MinSort_P$ is equivalent to determining a shortest path between π and $\boldsymbol{\iota}$ in $\mathcal{G}^c(P, n)$. The $Diam_P$ problem is equivalent to determining the diameter (longest shortest path between vertices) of $\mathcal{G}^c(P, n)$.

In this dissertation, the $MinSort_P$ and $Diam_P$ problems are investigated for the following generators: element-swaps, reversals (a contiguous subsequence or block in π is inverted), and block-moves (a block in π is repositioned elsewhere in the permutation). Bounds on these generators are introduced:

- for swaps, on the distance between the swapped elements;
- for reversals, on the length of the inverted block; and,
- for block-moves, on the lengths of the blocks moved.

Investigating bounded $MinSort_P$ and $Diam_P$ problems is useful as it gives us a more complete understanding of the computational complexity of these problems in general. Sorting by unbounded swaps is solvable [23]; sorting by unbounded reversals is NP-complete [7]; and sorting by unbounded block-moves remains open. It is interesting theoretically to determine whether the computational complexities of these problems are affected when distance bounds are applied to the problems. Furthermore, it has been suggested that research be directed towards finding minimum *weighted* sorting sequences where weights are assigned to the generators based on type or length [27]. Certainly a first step in exploring such a problem is to understand the problems we investigate in this dissertation.

[2] We assume here that all permutations in S_n can be sorted by generators that satisfy the predicate P. Equivalently, we assume that the inverses of the these generators *generate* S_n, in the group-theoretic sense.

1.4 Summary of Results

This dissertation obtains a variety of results on bounded sorting problems. We prove that, when the bound is a function of the length of the input permutation, then there are several sorting problems whose bounded variants are at least as difficult as the corresponding unbounded problems (Theorem 4.6). *Short* generators are those that affect the positions of a block of at most 3 elements (the bound is exactly 3). For the problem of sorting by short block-moves, we show that it suffices to consider only block-moves that correct the relative order of the elements moved (Theorem 5.2). We show that similar statements can be made for other bounded sorting problems, in particular, sorting by short generators (Theorem 5.4), sorting by short swaps or short reversals (Theorem 5.6), and sorting by bounded swaps for any given bound (Theorem 5.8).

This dissertation focuses on two problems: sorting by short block-moves ($MinSort_{Bk^3}$) and sorting by short swaps ($MinSort_{Sw^3}$). Using the proven statements on relative order, graph models are devised for both problems. For $MinSort_{Bk^3}$, we use permutation graphs (Section 6.1) and arc graphs (Section 6.3). We show that $MinSort_{Bk^3}$ is solvable in polynomial time for two large classes of permutations: woven bitonic permutations (Theorem 6.9) and woven double-strip permutations (Theorem 6.14). For general permutations, we present a $\frac{4}{3}$-approximation algorithm for computing short block-move distance (Section 6.8). We also show that $Diam_{Bk^3}(n)$, the short block-move diameter of S_n, equals $\lceil \binom{n}{2}/2 \rceil$ (Theorem 6.17).

Sorting by short block-moves can be viewed as a bounded variant of sorting by single element insertions. We show that sorting by unbounded single element insertions is solvable in polynomial time using longest increasing subsequences (Theorem 7.1). A corresponding algorithm for $MinSort_{Bk^3}$ based on a longest increasing subsequence is devised (Section 7.4).

For $MinSort_{Sw^3}$, we use the permutation graph model and define an alternate model, vector diagrams (Section 8.2). Using permutation graphs, we obtain a straightforward 3-approximation algorithm (Section 8.1) for computing short swap distance and a lower bound of $\lceil \binom{n}{2}/3 \rceil$ for short swap diameter (Theorem 8.4). Vector diagrams, on the other hand, allow us to obtain a 2-approximation algorithm for computing short swap distance (Section 8.3), and an upper bound of $\lfloor \frac{n^2}{4} \rfloor$ for short swap diameter (Theorem 8.19). Vector diagrams also provide a lower bound for the short swap distance of a permutation and we prove that it can be determined in polynomial time whether there exists an optimal solution whose length is exactly the lower bound (Theorem 6.2).

This dissertation also provides experimental results. We find that the heuristics we devise based

on permutation graphs, longest increasing subsequences, and vector diagrams all perform well in practice (Chapter 9).

1.5 Organization

The rest of this dissertation is structured as follows. Chapter 2 defines terms and notation. Chapter 3 surveys previous work on $MinSort_P$ and $Diam_P$. In Chapter 4, we prove that some groups of bounded $MinSort_P$ variants are at least as difficult as the unbounded problems. Chapter 5 shows that for some bounded $MinSort_P$ problems, there exists an optimal sorting sequence for a permutation such that each generator corrects the relative order of particular pairs of elements in the permutation. In Chapter 6, we discuss sorting by short block-moves. Chapter 7 discusses sorting by single element insertions using longest increasing subsequences. Chapter 8 discusses sorting by short swaps (short reversals). Chapter 9 evaluates experimentally the algorithms devised in the previous chapters. We conclude in Chapter 10 where we summarize our results and provide future directions for this research.

Chapter 2

Technical Background

In this chapter, we provide the necessary technical background for this dissertation. Section 2.1 describes how permutations and generators are specified. In Sections 2.2, 2.3, and 2.4, swaps, reversals, and block-moves are discussed in detail and are viewed within the context of distance bounds. Section 2.5 generalizes the notion of a distance bound to any predicate P. Sections 2.6 and 2.7 describe alternate ways of specifying generators. In Section 2.8, we define several concepts related to order and subsequences in a permutation. In Section 2.9, we describe our notation for graphs. Finally, in Section 2.10, we describe the various ways of referring to the problems we investigate in this dissertation.

2.1 Permutations and Generators

A particular permutation $\pi \in S_n$, when viewed as an ordered arrangement of elements, is denoted by the sequence of elements written in boldface and delimited by spaces; for example, $\pi = \mathbf{2\ 5\ 1\ 3\ 6\ 4}$. Recall that the term $\pi(i)$ denotes the element of π in position i; in the example, $\pi(4) = \mathbf{3}$. The *length* of a permutation is its number of elements; in the example, the length of π is 6.

Whenever the permutation is viewed as a generator, it is specified in the standard form,

$$\gamma = \begin{pmatrix} 1 & 2 & \dots & n \\ a_1 & a_2 & \dots & a_n \end{pmatrix},$$

where $\gamma(i) = a_i$. In our formulation of $MinSort_P$, the predicate P on a generator γ allows us to

restrict the generators to a particular set. Let $P[n]$ denote the set of generators γ of length n for which $P(\gamma)$ holds. We say P *describes* the set $P[n]$. This dissertation focuses on $MinSort_P$ and $Diam_P$, where $P[n]$ contains either swaps, reversals, or block-moves (Sections 2.2–2.4).

We denote the identity permutation $\mathbf{1\ 2\ \ldots\ n}$ or the identity generator

$$\begin{pmatrix} 1\ 2\ \ldots\ n \\ 1\ 2\ \ldots\ n \end{pmatrix}$$

by the symbol $\boldsymbol{\iota}$. We use the expression $\boldsymbol{\iota}[n]$ whenever it is important to clearly indicate the length of this permutation. Another special type of permutation is the *decreasing permutation* $\boldsymbol{\delta}[n] = \mathbf{n\ n-1\ \ldots\ 1}$.

2.2 Swaps

The *swap* $\sigma\langle i, j\rangle$, where $i < j$, is the generator

$$\begin{pmatrix} 1 & 2 & \ldots & i-1 & i & i+1 & \ldots & j-1 & j & j+1 & \ldots & n \\ 1 & 2 & \ldots & i-1 & j & i+1 & \ldots & j-1 & i & j+1 & \ldots & n \end{pmatrix}.$$

When composed with a permutation π, the swap σ has the effect of switching elements $\pi(i)$ and $\pi(j)$ in π. Consider the permutation $\pi = \mathbf{2\ 3\ 6\ 7\ 4\ 5\ 9\ 8\ 1}$, for example. Applying the swap $\sigma = \sigma\langle 3, 6\rangle$ to π yields $\pi \cdot \sigma = \mathbf{2\ 3\ 5\ 7\ 4\ 6\ 9\ 8\ 1}$.

In the product $\pi \cdot \sigma\langle i, j\rangle$, the elements $\pi(i)$ and $\pi(j)$ are called the *swapped elements* in π. The swap $\sigma\langle i, j\rangle$ is called an m-swap if $m = j - i + 1$; we call m the *extent* of σ. An m-swap is *short* if $m \leq 3$. When $\sigma\langle i, j\rangle$ is a 3-swap, the element $\pi(i+1)$ is the *middle element*; the elements on either side of the middle element are the swapped elements in this case. Sw is the predicate that describes the set of all swaps, that is, $Sw(\gamma)$ holds when γ is a swap. For an integer ℓ, Sw^ℓ describes the set of all m-swaps where $m \leq \ell$; we call such swaps *ℓ-bounded* swaps. Clearly, $Sw^\ell[n] \subseteq Sw[n]$. Here, ℓ bounds the distance between the swapped elements. Note, in particular, that Sw^3 describes short swaps.

2.3 Reversals

A *block* is a subsequence of contiguous elements in π. The *reversal* $\rho\langle i,j\rangle$, where $1 \leq i < j \leq n$, is the generator

$$\begin{pmatrix} 1 & 2 & \dots & i-1 & i & i+1 & \dots & j-1 & j & j+1 & \dots & n \\ 1 & 2 & \dots & i-1 & j & j-1 & \dots & i+1 & i & j+1 & \dots & n \end{pmatrix}.$$

This generator has the effect of inverting the block $\pi(i)\ \pi(i+1)\ \dots\ \pi(j)$ in π. Consider the permutation $\pi = \mathbf{2\ 3\ 6\ 7\ 4\ 5\ 9\ 8\ 1}$, for example. Applying the reversal $\rho = \rho\langle 4,8\rangle$ to π yields $\pi \cdot \rho = \mathbf{2\ 3\ 6\ 8\ 9\ 5\ 4\ 7\ 1}$.

Consider the product $\pi \cdot \rho\langle i,j\rangle$. The reversal $\rho\langle i,j\rangle$ is an m-reversal if $m = j-i+1$; intuitively, m represents the length of the inverted block. The m-reversal $\rho\langle i,j\rangle$ is *short* if $m \leq 3$. When $\rho\langle i,j\rangle$ is a 3-reversal, the element $\pi(i+1)$ is the *middle element*. Rv is the predicate that describes the set of all reversals. For an integer ℓ, Rv^ℓ describes the set of all m-reversals, where $m \leq \ell$. Clearly, $Rv^\ell[n] \subseteq Rv[n]$. Note that Rv^3 describes short reversals and that $Rv^3[n] = Sw^3[n]$.

2.4 Block-Moves

The *block-move* $\beta\langle i,j,k\rangle$, where $1 \leq i < j < k \leq n+1$, is the generator

$$\begin{pmatrix} 1 & 2 & \dots & i-1 & i & i+1 & \dots & j-1 & j & j+1 & \dots & k-1 & k & k+1 & \dots & n \\ 1 & 2 & \dots & i-1 & j & j+1 & \dots & k-1 & i & i+1 & \dots & j-1 & k & k+1 & \dots & n \end{pmatrix}.$$

A block-move repositions a block of elements in π, in particular, the block $\pi(i)\ \pi(i+1)\ \dots\ \pi(j-1)$ is moved to the immediate left of element $\pi(k)$ in π. Equivalently, the block $\pi(j)\ \pi(j+1)\ \dots\ \pi(k-1)$ is moved to the immediate left of element $\pi(i)$ in π. Another equivalent description of the effect of block-move $\beta\langle i,j,k\rangle$ on π is that the adjacent blocks $\pi(i)\ \pi(i+1)\ \dots\ \pi(j-1)$ and $\pi(j)\ \pi(j+1)\ \dots\ \pi(k-1)$ are switched. Consider the permutation $\pi = \mathbf{2\ 3\ 6\ 7\ 4\ 5\ 9\ 8\ 1}$, for example. Applying the block-move $\beta = \beta\langle 3,5,8\rangle$ to π yields $\pi \cdot \beta = \mathbf{2\ 3\ 4\ 5\ 9\ 6\ 7\ 8\ 1}$.

Consider the product $\pi \cdot \beta\langle i,j,k\rangle$. Recall that in a block-move, adjacent blocks are swapped. The block-move $\beta\langle i,j,k\rangle$ is called an (x,y)-block-move where x and y represent the individual lengths of the two blocks, $j-i$ and $k-j$. An (x,y)-block-move is *short* if $x+y \leq 3$. A (1,1)-block-move is a *skip*. A *hop* is a (1,2)-block-move (a *right hop*) or a (2,1)-block-move (a *left hop*). The set of short block-moves consists of skips and hops. Bk describes the set of all block-moves. For an integer

ℓ, Bk^{ℓ} describes the set of all (x, y)-block-moves, where $x + y \leq \ell$. Clearly, $Bk^{\ell}[n] \subseteq Bk[n]$. For integers p and q, $Bk^{p,q}$ describes the set of all (x, y)-block-moves, satisfying $x \leq p$ and $y \leq q$ or $x \leq q$ and $y \leq p$. The predicates Bk^3 and $Bk^{1,2}$ both describe short block-moves.

2.5 Distance Bounds

In general, if P is a predicate and ℓ is an integer, P^{ℓ} describes all generators $\gamma \in P[n]$ such that there exists two integers a, b, with $1 \leq a < b \leq n + 1$, $b - a \leq \ell$, and $\gamma(i) = i$ for all i, $1 \leq i < a$ or $b \leq i \leq n$. That is, the generator γ, when composed with a permutation π, preserves the position of all elements in π except perhaps for elements in some block in π of length at most ℓ. Here, ℓ is called a *distance bound*. For instance, *short* generators are the case when the distance bound $\ell = 3$. In particular, let Gn be the predicate that holds true for all permutations. The predicate Gn^3 describes all short generators, that is, generators that permute the elements in a block of length 3. It can be verified that $Gn^3[n] = Bk^3[n] \cup Rv^3[n]$.

2.6 Value-Based Specification

It is also useful to specify generators in terms of the values of elements in π instead of their positions. In particular, the swap $\sigma\langle i, j\rangle$ is alternately denoted by $S\langle \mathbf{a}, \mathbf{b}\rangle$ or $S\langle \mathbf{b}, \mathbf{a}\rangle$, where $\mathbf{a} = \pi(i)$ and $\mathbf{b} = \pi(j)$. The reversal $\rho\langle i, j\rangle$ is alternately denoted by $R\langle s\rangle$, where $s = \pi(i)\ \pi(i+1)\ \ldots \pi(j)$ (the block inverted in π). Finally, the block-move $\beta\langle i, j, k\rangle[n]$ is alternately denoted by $B\langle s, t)[n]$, where $s = \pi(i)\ \pi(i+1)\ \ldots \pi(j-1)$ and $t = \pi(j)\ \pi(j+1)\ \ldots \pi(k-1)$ (the blocks moved in π). For example, let $\pi = \mathbf{2\ 4\ 5\ 1\ 3\ 6}$. The product $\pi \cdot \sigma\langle 2, 4\rangle$ is the same as $\pi \cdot S\langle \mathbf{4}, \mathbf{1}\rangle$. The product $\pi \cdot \rho\langle 2, 4\rangle$ is the same as $\pi \cdot R\langle \mathbf{4\ 5\ 1}\rangle$. Finally, the product $\pi \cdot \beta\langle 1, 3, 6\rangle$ is the same as $\pi \cdot B\langle \mathbf{2\ 4}, \mathbf{5\ 1\ 3}\rangle$. Value-based specification is particularly useful during illustrations or when specifying long sorting sequences since we simply indicate the elements repositioned in the permutation.

It should be noted that generators specified by value are dependent on context particularly for reversals and block-moves. The reversal $R\langle \mathbf{4\ 5\ 1}\rangle$ for example, is valid only when the elements **4**, **5**, and **1** appear contiguously in the permutation the generator is composed with. Swaps specified by value, on the other hand, are valid regardless of the permutation π. In fact, applying a value-based swap has some group-theoretic significance: the permutation π composed with the swap $S\langle \mathbf{a}, \mathbf{b}\rangle$ is

the same as π *left composed* with the generator $(a\ b)$ specified in cycle notation. That is,

$$\pi \cdot S\langle \mathbf{a}, \mathbf{b} \rangle = (a\ b) \cdot \pi.$$

2.7 Specifying Lengths

The parameters of the generators $\sigma\langle i, j\rangle$, $\rho\langle i, j\rangle$ and $\beta\langle i, j, k\rangle$ are positions in π assuming these are generators composed with π. These generators are permutations themselves so it is more precise to specify, in addition, a *length* parameter to our notation in order to distinguish, say, $\sigma\langle 2, 4\rangle$ as a permutation of length 5 from $\sigma\langle 2, 4\rangle$ as a permutation of length 8. Whenever such a distinction needs to be made we use the following notation for the three types of generators: $\sigma\langle i, j\rangle[n]$, $\rho\langle i, j\rangle[n]$, and $\beta\langle i, j, k\rangle[n]$. Whenever the length n is clear from context, we omit this parameter.

Length parameters may also be included when specifying generators by value, using similar notation: $S\langle \mathbf{a}, \mathbf{b}\rangle[n]$, $R\langle s\rangle[n]$, and $B\langle s, t\rangle[n]$.

2.8 Order and Subsequences

Let π be a permutation. An *unpositioned element* in π is an element $\pi(i)$ such that $\pi(i) \neq i$. An *inversion* is a pair of elements $\pi(i)$ and $\pi(j)$ in π that are not in their correct relative order; that is, $i < j$ and $\pi(i) > \pi(j)$.

A *subsequence* of a permutation $\pi \in S_n$ is a sequence $\pi(i_1)\ \pi(i_2)\ \ldots\ \pi(i_\ell)$, where $1 \leq i_j \leq n$ for each i_j and $i_1 < i_2 < \ldots < i_\ell$. The subsequence is *increasing* if $\pi(i_j) < \pi(i_{j+1})$ for $1 \leq j < \ell$; the subsequence is *decreasing* if $\pi(i_j) > \pi(i_{j+1})$ for $1 \leq j < \ell$. A *longest increasing subsequence* (LIS) of π is an increasing subsequence of π, $\pi(i_1)\pi(i_2)\ldots\pi(i_\ell)$, such that ℓ is as large as possible. In the permutation $\pi = \mathbf{4\ 2\ 3\ 5\ 1\ 7\ 6\ 8}$, for example, a longest increasing subsequence is $\mathbf{2\ 3\ 5\ 6\ 8}$. Longest increasing subsequences of permutations can be determined in $O(n \log\log n)$ time particularly because the values of the elements in a permutation are bounded by n [24].

Let $\mathbf{a}$ be an element in π. The *restriction* $\pi[\mathbf{a}]$ is the subsequence in π containing all elements $\leq \mathbf{a}$. For example, let $\pi = \mathbf{2\ 4\ 1\ 3\ 6\ 5}$; the restriction $\pi[\mathbf{3}] = \mathbf{2\ 1\ 3}$. Restrictions generalize to any sequence of integers, not just permutations. That is, given a sequence s of integers and an integer b, the restriction $s[b]$ is the subsequence of s consisting of all elements $\leq b$. For example, let $s = \mathbf{7\ 2\ 4\ 6\ 3}$; the restriction $s[4] = \mathbf{2\ 4\ 3}$.

2.9 Graph Notation

This dissertation uses graphs to model the problems we investigate. We use the symbol $\mathcal{G}$ to denote a graph. We use the symbol $\mathcal{V}$ to denote its vertices. A graph may be directed or undirected. If $\mathcal{G}$ is directed, then the graph is given by $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ where $\mathcal{A}$ is its set of arcs (ordered pairs of vertices from $\mathcal{V}$). If $\mathcal{G}$ is undirected, then the graph is given by $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{E}$ is its set of edges (unordered pairs of vertices from $\mathcal{V}$).

We use additional superscript symbols to distinguish between different types of graphs. For example, as defined earlier in Chapter 1, $\mathcal{G}^c = (\mathcal{V}^c, \mathcal{A}^c)$ denotes a Cayley graph. To distinguish between different *instances* of graphs, we use additional subscript symbols. For example, $\mathcal{G}^p_\pi = (\mathcal{V}^p_\pi, \mathcal{A}^p_\pi)$ denotes the permutation graph (see Section 6.1) for the permutation π.

2.10 Problem Names

Minimum sorting by P for a particular predicate P is denoted by $MinSort_P$ where P is replaced by the actual predicate. In addition, whenever the predicate describes generators with some associated name (such as "short block-move"), this name is used when referring to the actual $MinSort_P$ problem. For example, $MinSort_{Bk^3}$ denotes *minimum sorting by short block-moves*, the problem of determining the *short block-move distance* of a permutation. The same convention applies to $Diam_P$; for example, $Diam_{Bk^3}$ denotes the *short block-move diameter* problem.

Chapter 3

Related Research

In this chapter, we survey previous results on $MinSort_P$ and $Diam_P$. Most of the work discussed here focuses on a particular predicate P.

Even and Goldreich [11] show that both $MinSort_P$ and $Diam_P$, when P is considered part of the problem instance, are NP-hard. The results are strengthened by Jerrum [23] where he shows that these problems are PSPACE-complete.

Jerrum also identifies some cases of $MinSort_P$ that are polynomial-time solvable. One case is minimum sorting by adjacent element swaps. In fact, the operations performed by the well-known sorting algorithm *bubble-sort* is precisely an optimal sorting sequence of swaps in this case. Observe that this problem is a simple example of a *bounded* variant of $MinSort_P$ and the predicate P is exactly $Sw^2 = Rv^2 = Bk^2 = Bk^{1,1}$. Other cases identified by Jerrum where $MinSort_P$ is solvable in polynomial time are when P describes either swaps, generators that permute any 3 elements, or particular combinations of these generators. Driscoll and Furst [10] show that $Diam_P(n) = O(n^2)$ for the predicates P that describe the generators indicated above and for other general cases for P.

Kececioglu and Sankoff [26, 27] and Bafna and Pevzner [6] have results on $MinSort_{Rv}$ and $Diam_{Rv}$. They introduce the notions of breakpoints and the breakpoint graph of a permutation. Employing these notions, they identify upper and lower bounds for $MinSort_{Rv}(\pi)$ (the *reversal distance* of π) and devise corresponding approximation algorithms. Bafna and Pevzner's algorithm, in particular, has a performance guarantee of 1.75. They also show that $Diam_{Rv}(n) = n - 1$. Heath and Vergara [20] test some conjectures on $MinSort_{Rv}$ and provide a mechanism to generate

counterexamples. Caprara [7] shows that $MinSort_{Rv}$ is NP-complete. Hannenhalli and Pevzner [19] and Kaplan, Shamir, and Tarjan [25] show that a variant of the problem, sorting *signed* permutations by reversals, is solvable in polynomial time.

Minimum sorting by *prefix reversals*, also known as the pancake problem, is the case where $P[n] = \{\rho(1, j) \mid 2 \leq j \leq n\}$. Heydari [21] shows that this problem is NP-complete. The prefix-reversal diameter of S_n is at least $\frac{17}{16}n$ [15, 17] and at most $\frac{3}{2}(n+1)$ [22].

Chen and Skiena [8] investigate minimum sorting by fixed-length reversals. Here, $P[n]$ is the set of all m-reversals for some fixed m. They describe the equivalence classes of permutations under generators of this type. They also introduce the problem of determining the m-reversal diameter for *circular* permutations and provide upper and lower bounds for this diameter.

Bafna and Pevzner [5] study minimum sorting by block-moves ($MinSort_{Bk}$) and provide a metric on a permutation that counts cycles in a graph representation of the permutation. Based on this metric, they derive upper and lower bounds for $MinSort_{Bk}(\pi)$ (i.e., the *block-move distance* of π) and present an approximation algorithm with a 1.5 performance guarantee. They also conjecture that $Diam_{Bk}(n) = \lfloor \frac{n}{2} \rfloor + 1$ and prove that it is at least $\lfloor \frac{n}{2} \rfloor$. Guyer, Heath, and Vergara [16] employ some heuristics based on subsequences and devise corresponding algorithms for the problem. They observe that selecting block-moves that maximally lengthen the longest increasing subsequence in the permutation appear to produce near-optimal results. It is not yet known whether or not $MinSort_{Bk}$ is NP-complete. Aigner and West [2] provide a polynomial-time algorithm for a variant of this problem where the block-moves are restricted to those that re-insert the first element in a permutation.

Block-interchanges are generators that switch blocks that are not necessarily adjacent (a generalization of block-moves). Christie [9] shows that minimum sorting by block-interchanges is solvable in $O(n^2)$ time and establishes that the block-interchange diameter of S_n equals $\lfloor \frac{n}{2} \rfloor$.

Cayley graphs have applications in the study of interconnection networks [3, 4]. A pancake network [3, 22] (also called a pancake graph), for instance, corresponds to the Cayley graph $\mathcal{G}^c(P, n)$ where $P[n]$ is the set of prefix reversals. Gates and Papadimitriou [15] and Heydari and Sudborough [22] provide bounds on the diameter of pancake networks. Determining the diameter of interconnection networks is the same as solving $Diam_P$ for the appropriate P. A Cayley graph need not be connected; if it is not connected then $\mathcal{G}^c(P, n)$ consists of a collection of isomorphic connected components. In particular, it is possible that the predicate P is chosen so that only some permuta-

tions in S_n can be sorted by composition with generators satisfying P (although for the generators considered in this paper, all permutations in S_n can be sorted). In such a case, each component of $\mathcal{G}^c(P, n)$ consists of vertices that constitute a subset of S_n. The component containing the identity $\boldsymbol{\iota}$ is called a subgroup or *permutation group* of S_n. Taking an example from the study of interconnection networks, observe that several isomorphic copies of the n-degree hypercube are exhibited in the Cayley graph $\mathcal{G}^c(P, 2n)$ where P describes the set of swaps $\sigma\langle i, i+1\rangle[2n]$ for odd i. Cayley graphs with high symmetry and low degree and diameter are preferable; the pancake network and the hypercube are examples of such graphs. Akers and Krishnamurthy [3] show that every symmetric interconnection network corresponds to a Cayley graph for some predicate P.

Chapter 4

Reduction Results

In this chapter, we show that many bounded problems are at least as difficult as the corresponding unbounded problems. We begin by restating $MinSort_P$ as a decision problem.

MINIMUM SORTING BY P ($MinSort_P$)

INSTANCE: A permutation $\pi \in S_n$ and an integer K.

QUESTION: Is there a sorting sequence of generators for π such that each generator in the sequence satisfies P and the length of the sequence is at most K?

Section 4.1 presents a reduction from the unbounded $MinSort_{Bk}$ problem to a family of bounded $MinSort_{Bk}$ variants. Section 4.2 presents similar results for a larger class of predicates including Rv and $Bk \vee Rv$.

4.1 Bounded Block-Moves

Consider $MinSort_{Bk^{\lfloor f(n) \rfloor}}$ where $f(n)$ is some function proportional to n; specifically, $f(n) = n/r + c$ for constants $r \geq 1$ and $c \geq 0$. As an example, $MinSort_{Bk^{\lfloor n/2 \rfloor}}$ is minimum sorting where the block-moves are such that the total lengths of the blocks do not exceed half the permutation length. The following result shows that there is a polynomial reduction from $MinSort_{Bk}$ to a problem such as $MinSort_{Bk^{\lfloor n/2 \rfloor}}$.

Theorem 4.1 *Let $f(n) = n/r + c$ for constants $r \geq 1$ and $c \geq 0$. Then, $MinSort_{Bk}$ reduces in linear time to $MinSort_{Bk\lfloor f(n)\rfloor}$.*

Proof: We construct a $MinSort_{Bk\lfloor f(n)\rfloor}$ instance (π_2, K) from a $MinSort_{Bk}$ instance (π_1, K). Let m denote the length of the permutation π_1, to distinguish it from the length of the permutation for the resulting instance. That is,

$$\pi_1 = \pi_1(1)\ \pi_1(2)\ \ldots\ \pi_1(m).$$

Obtain the $MinSort_{Bk\lfloor f(n)\rfloor}$ instance (π_2, K) as follows. The same integer K applies and the permutation π_2 is

$$\pi_2 = \pi_1(1)\ \pi_1(2)\ \ldots\ \pi_1(m)\ \mathbf{m+1}\ \mathbf{m+2}\ \ldots\ \mathbf{n}$$

where $n = \lceil f^{-1}(m)\rceil = \lceil (m-c)r\rceil$. Observe that in the bounded problem $MinSort_{Bk\lfloor f(n)\rfloor}$, the block-moves allowed are those whose lengths are at most

$$\lfloor f(n)\rfloor = \left\lfloor \frac{1}{r}\lceil (m-c)r\rceil + c \right\rfloor.$$

It can be verified that this value is just m, the length of the $MinSort_{Bk}$ permutation π_1, from which π_2 is derived. Intuitively, π_2 is just π_1 followed by additional elements already in their correct positions in the permutation. Constructing the permutation π_2 of length n requires time linear in m (recall that $n = \lceil (m-c)r\rceil$). To prove that this construction is appropriate, it remains to show that a sorting sequence (under $MinSort_{Bk}$ constraints) of length $k_1 \leq K$ for π_1 exists if and only if a sorting sequence (under $MinSort_{Bk\lfloor f(n)\rfloor}$ constraints) of length $k_2 \leq K$ for π_2 exists.

(Only if) Let $\beta_1, \beta_2, \ldots, \beta_{k_1}$ be a sorting sequence for π_1 where $Bk(\beta_i)$ holds for $1 \leq i \leq k_1$; that is,

$$\pi_1 \cdot \beta_1 \cdot \beta_2 \cdots \beta_{k_1} = \boldsymbol{\iota}[m].$$

For all i, $1 \leq i \leq k_1$, let $\beta'_i = \beta_i[n]$. Then,

$$\pi_2 \cdot \beta'_1 \cdot \beta'_2 \cdots \beta'_{k_1} = \boldsymbol{\iota}[n],$$

since π_2 is just π_1 concatenated with $\mathbf{m+1}\ \mathbf{m+2}\ \ldots\ \mathbf{n}$. It is appropriate to use the same indices for each β'_i because the positions of the elements $\mathbf{1}, \mathbf{2}, \ldots, \mathbf{m}$ in π_1 are preserved in π_2. In effect, each β'_i repositions only those elements in π_2 that are also in π_1 (leaving the positions of elements $\mathbf{m+1}, \mathbf{m+2}, \ldots, \mathbf{n}$ unchanged). Finally, each β'_i moves blocks with total length at most m (since no elements $> \mathbf{m}$ are in the blocks), which follows $MinSort_{Bk\lfloor f(n)\rfloor}$ constraints.

(If) A sorting sequence for π_2 may involve elements $\mathbf{m+1}, \mathbf{m+2}, \ldots, \mathbf{n}$. This suggests that a derivation similar to the one given above will not apply particularly because the positions of the elements $\mathbf{1}, \mathbf{2}, \ldots, \mathbf{m}$ may not be preserved as the block-moves are applied to the permutation. We instead perform a derivation based on the values of the elements moved.

Recall that a restriction of a sequence of integers is the subsequence consisting of elements less than or equal to some integer. Notice, for instance, that $\pi_1 = \pi_2[\mathbf{m}]$.

Now, or every block-move that switches two blocks of elements in π_2, we derive a corresponding block-move that switches the same blocks of elements in π_1 ignoring elements $\mathbf{m+1}, \mathbf{m+2}, \ldots, \mathbf{n}$, since these elements do not appear in π_1. We use restrictions in specifying such block-moves. More precisely, suppose that

$$\pi_2 \cdot B_1 \cdot B_2 \cdots B_{k_2} = \boldsymbol{\iota}[n],$$

where the block-moves are specified by values instead of positions. For each B_i, $1 \leq i \leq k_2$, let $B'_i = B\langle s[m], t[m]\rangle[m]$ whenever $B_i = B\langle s, t\rangle[n]$ and neither $s[m]$ nor $t[m]$ is an empty sequence; let $B'_i = \boldsymbol{\iota}[m]$ if at least one of $s[m]$ or $t[m]$ is an empty sequence. Then,

$$\pi_1 \cdot B'_1 \cdot B'_2 \cdots B'_{k_2} = \boldsymbol{\iota}[m].$$

The effect of each B_i on π_2 is simulated by each B'_i on the $MinSort_{Bk}$ permutation π_1, insofar as the elements $\mathbf{1\ 2\ \ldots\ m}$ are concerned. As an example, consider Figure 4.1 where where $B'_i = B\langle \mathbf{4\ 1}, \mathbf{5\ 2}\rangle$ is derived from $B_i = B\langle \mathbf{4\ 1}, \mathbf{5\ 2\ 6}\rangle$. The figure specifies the block-moves by the elements' indices and the actual blocks of elements moved are shown in boxes. Observe that in the derivation, elements in the blocks that are greater than $\mathbf{5}$ (particularly element $\mathbf{6}$) are simply omitted to obtain a corresponding block-move.

Since the sequence $B_1, B_2, \ldots B_{k_2}$ sorts π_2, the sequence $B'_1, B'_2, \ldots B'_{k_2}$ sorts π_1. Whenever $B'_i = \boldsymbol{\iota}$, it is removed from the sorting sequence since it has no effect on the permutation. The length of the resulting sequence is therefore $k_1 \leq k_2 \leq K$. Finally, we note that there are no block-length constraints for $MinSort_{Bk}$ so the proof is complete. □

The result immediately extends to other bounds $f(n)$. In particular, suppose the function $f(n)$ is such that the inverse f^{-1} exists and is computable in polynomial time. Furthermore, suppose $f^{-1}(m)$ is $O(g(m))$ for some polynomial $g(m)$. Then, the permutation for $MinSort_{Bk}$ is simply extended accordingly; that is, the corresponding length of the $MinSort_{Bk^{\lfloor f(n)\rfloor}}$ instance is $n = \lceil f^{-1}(m)\rceil$.

$$
\begin{aligned}
\pi_2 \cdot B_i &= \mathbf{3}\ \boxed{\mathbf{4\ 1}}\ \boxed{\mathbf{5\ 2\ 6}}\ \mathbf{7\ 8\ 9\ 10} \cdot \beta\langle 2,4,7\rangle \\
&= \mathbf{3}\ \boxed{\mathbf{5\ 2\ 6}}\ \boxed{\mathbf{4\ 1}}\ \mathbf{7\ 8\ 9\ 10} \\
\\
\pi_1 \cdot B'_i &= \mathbf{3}\ \boxed{\mathbf{4\ 1}}\ \boxed{\mathbf{5\ 2}} \cdot \beta\langle 2,4,6\rangle \\
&= \mathbf{3}\ \boxed{\mathbf{5\ 2}}\ \boxed{\mathbf{4\ 1}}
\end{aligned}
$$

Figure 4.1: Deriving a $MinSort_{Bk}$ block-move from a $MinSort_{Bk^{n/2}}$ block-move.

The constraints we have given on the function f guarantees that the reduction is polynomial.

Theorem 4.2 *Let $f(n)$ be a function such that f^{-1} is computable in polynomial time and $f^{-1}(m)$ is $O(g(m))$ for some polynomial $g(m)$. Then, $MinSort_{Bk}$ reduces in polynomial time to $MinSort_{Bk^{\lfloor f(n)\rfloor}}$.*

For example, suppose $f(n) = \frac{1}{r}n^{1/p} + c$, for constants $r, p \geq 1$ and $c \geq 0$. In the reduction from $MinSort_{Bk}$ to $MinSort_{Bk^{\lfloor f(n)\rfloor}}$, the length of the resulting instance is

$$n = \lceil f^{-1}(m)\rceil = \lceil ((m-c)r)^p\rceil.$$

Observe that $f^{-1}(m) = ((m-c)r)^p$ is computable in polynomial time and is $O(m^p)$ which implies that lengthening the permutation from m to n can be accomplished in polynomial time.

This suggests that, as the time complexity of $MinSort_{Bk}$ is not resolved, it can be worthwhile to investigate $MinSort_{Bk^\ell}$ cases where ℓ is a fixed integer. Chapter 6 addresses one of these cases.

The next section uses the technique developed in this section to obtain similar results for $MinSort_{Rv}$ and $MinSort_{Bk\vee Rv}$.

4.2 Restriction-Preserving Predicates

Let P be a predicate on generators. Let I be the predicate that holds true only for identity generators. We say P is *restriction-preserving* if the following holds: if π is a permutation of length n, γ is a generator in $P[n]$, their product is $\pi' = \pi \cdot \gamma$, and b is an integer $\leq n$, then there exists a $\gamma' \in (P \vee I)[b]$ such that $\pi'[b] = \pi[b] \cdot \gamma'$. Intuitively, a restriction-preserving predicate allows us to derive corresponding generators for restrictions of a permutation. We use the same adjective

"restriction-preserving" for the sets of generators described by restriction-preserving predicates. For example, block-moves and reversals are restriction-preserving.

Lemma 4.3 *The predicate Bk is restriction-preserving.*

Proof: Given permutations π and γ together with an integer b, derive $\gamma' = B\langle s[b], t[b]\rangle[b]$ from $\gamma = B\langle s, t\rangle[n]$, where $B' = \boldsymbol{\iota}[b]$ when either $s[b]$ or $t[b]$ is empty. □

Lemma 4.4 *The predicate Rv is restriction-preserving.*

Proof: Given permutations π and γ together with an integer b, derive $\gamma' = R'\langle s[b]\rangle[n]$ from $\gamma = R\langle s\rangle[n]$, where $R' = \boldsymbol{\iota}[b]$ when $s[b]$ is empty. □

As a consequence of the above lemmas, the set consisting of block-moves and reversals is also restriction-preserving.

Lemma 4.5 *The predicate $Bk \vee Rv$ is restriction-preserving.*

Swaps are an example of generators that are *not* restriction-preserving as the following example suggests:

$$\begin{aligned}
\pi &= \mathbf{5\ 3\ 2\ 1\ 4} \\
\pi[3] &= \mathbf{3\ 2\ 1} \\
\pi' &= \pi \cdot \sigma\langle 2, 5\rangle = \mathbf{5\ 4\ 2\ 1\ 3} \\
\pi'[3] &= \mathbf{2\ 1\ 3}.
\end{aligned}$$

In this example, there is no swap σ' such that $\pi'[3] = \pi[3] \cdot \sigma'$.

The following theorem generalizes the results in the previous section. Recall that given a predicate P, P^ℓ is just the predicate P with an additional distance bound ℓ.

Theorem 4.6 *Let P be a restriction-preserving predicate and let $f(n)$ be a function such that f^{-1} is computable in polynomial time and $f^{-1}(m)$ is $O(g(m))$ for some polynomial $g(m)$. Then, $MinSort_P$ reduces in polynomial time to $MinSort_{P^{\lfloor f(n)\rfloor}}$.*

Proof: Given a $MinSort_P$ instance (π_1, K) where $\pi_1 = \pi_1(1)\ \pi_1(2)\ \ldots\ \pi_1(m)$, we obtain the $MinSort_{P^{\lfloor f(n)\rfloor}}$ instance (π_2, K) where

$$\pi_2 = \pi_1(1)\ \pi_1(2)\ \ldots\ \pi_1(m)\ \mathbf{m+1}\ \mathbf{m+2}\ \ldots\ \mathbf{n},$$

such that $n = \lceil f^{-1}(m) \rceil$. Note that $\lfloor f(n) \rfloor = m$, the additional distance bound we have imposed on generators for $MinSort_{P^{\lfloor f(n) \rfloor}}$. It remains to show that a sorting sequence (under $MinSort_P$ constraints) of length $k_1 \leq K$ for π_1 exists if and only if a sorting sequence (under $MinSort_{P^{\lfloor f(n) \rfloor}}$ constraints) of length $k_2 \leq K$ for π_2 exists.

(Only if) Let $\gamma_1, \gamma_2, \ldots, \gamma_{k_1}$ be a sorting sequence for π_1; that is,

$$\pi_1 \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_{k_1} = \boldsymbol{\iota}[m].$$

A similar sorting sequence applies to π_2; that is, if $\gamma_i = \gamma_i(1)\ \gamma_i(2)\ \ldots \gamma_i(m)$, then

$$\gamma_i' = \gamma_i(1)\ \gamma_i(2)\ \ldots \gamma_i(m)\ \mathbf{m+1}\ \mathbf{m+2}\ \ldots\ \mathbf{n},$$

for all i, $1 \leq i \leq k_1$ ($k_2 = k_1$). We then conclude

$$\pi_2 \cdot \gamma_1' \cdot \gamma_2' \cdots \gamma_{k_1}' = \boldsymbol{\iota}[n].$$

Extending γ_i to obtain γ_i' is appropriate because the elements in positions $m+1, m+2, \ldots, n$ in π_2 remain unaffected by the resulting generators. Each γ_i', therefore, repositions only the elements $\mathbf{1}, \mathbf{2}, \ldots \mathbf{m}$, simulating the effect that γ_i makes on π_1. It also follows that each γ_i' affects the positions of elements in a block of at most length $m = \lfloor f(n) \rfloor$.

(If) Observe that a sorting sequence for π_2 may involve elements $\mathbf{m+1}, \mathbf{m+2}, \ldots, \mathbf{n}$. For every generator that rearranges elements in π_2, we derive a corresponding generator that ignores elements $\mathbf{m+1}, \mathbf{m+2}, \ldots, \mathbf{n}$. We use the fact that P is a restriction-preserving predicate. More precisely, let

$$\pi_2 \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_{k_2} = \boldsymbol{\iota}[n].$$

Since P is restriction-preserving, there exists a generator $\gamma_i' \in (P \vee I)[m]$, for each γ_i, $1 \leq i \leq k_2$, such that,

$$\pi_2[m] \cdot \gamma_1' \cdot \gamma_2' \cdots \gamma_{k_2}' = \boldsymbol{\iota}[m].$$

Now, $\pi_2[m] = \pi_1$ so that,

$$\pi_1 \cdot \gamma_1' \cdot \gamma_2' \cdots \gamma_{k_2}' = \boldsymbol{\iota}[m].$$

The intention is that each γ_i' simulates the effect of γ_i on π_1 insofar as the elements $\mathbf{1}, \mathbf{2}, \ldots, \mathbf{m}$ are concerned. Whenever $\gamma_i' = \boldsymbol{\iota}$, it is simply removed from the derived sequence. A sorting sequence for π_2 therefore implies a (possibly shorter) sorting sequence for π_1, which completes the proof. $\square$

Theorem 4.1 in the previous section is just a consequence of the above theorem by Lemma 4.3. Now, our intended results for this section are summarized in the following corollaries and are implied by Theorem 4.6 and Lemmas 4.4 and 4.5.

Corollary 4.7 *Let $f(n)$ be a function such that f^{-1} is computable in polynomial time and $f^{-1}(m)$ is $O(g(m))$ for some polynomial $g(m)$. Then, $MinSort_{Rv}$ reduces in polynomial time to $MinSort_{Rv^{\lfloor f(n) \rfloor}}$.*

Corollary 4.8 *Let $f(n)$ be a function such that f^{-1} is computable in polynomial time and $f^{-1}(m)$ is $O(g(m))$ for some polynomial $g(m)$. Then, $MinSort_{Bk \vee Rv}$ reduces in polynomial time to $MinSort_{(Bk \vee Rv)^{\lfloor f(n) \rfloor}}$.*

Furthermore, in light of the result by Caprara [7], where $MinSort_{Rv}$ is shown to be NP-complete, all bounded $MinSort_{Rv}$ variants of this kind are likewise NP-complete.

Corollary 4.9 *Let $f(n)$ be a function such that f^{-1} is computable in polynomial time and $f^{-1}(m)$ is $O(g(m))$ for some polynomial $g(m)$. $MinSort_{Rv^{\lfloor f(n) \rfloor}}$ is NP-complete.*

Chapter 5

Relative Order

Recall that when P describes the generators that swap adjacent elements ($P = Sw^2 = Rv^2 = Bk^2 = Bk^{1,1}$), $MinSort_P$ is polynomial-time solvable and the sequence of operations performed in bubble-sort is precisely an optimal sorting sequence. In such a case, observe that any swap in this sequence is a "correcting" swap, i.e., it corrects the relative order of the two adjacent elements. For example, in the permutation **1 4 2 5 3**, an optimal sequence of generators (assuming only adjacent swaps are allowed) is: $S\langle \mathbf{4}, \mathbf{2} \rangle$, $S\langle \mathbf{5}, \mathbf{3} \rangle$, $S\langle \mathbf{4}, \mathbf{3} \rangle$. In each of the three swaps, the relative order of the swapped elements is corrected.

In this chapter, we prove that a similar statement can be made for a host of bounded sorting problems. The goal is to assert that it suffices to consider only those generators with some "correcting" characteristic when seeking an optimal sorting sequence. We can then restrict our attention to sorting sequences that contain only correcting generators, even though there may be optimal sorting sequences that contain non-correcting generators.

We first define what this correcting characteristic means for the different generators. In general, a *correcting generator* is one that corrects the relative order of all pairs of elements whose positions are affected by the generator. The block-move $\beta\langle i, j, k \rangle$ is a *correcting block-move* for π if the following property holds: $\pi(p) > \pi(q)$ for all p, q where $i \leq p < j \leq q < k$. The reversal $\rho\langle i, j \rangle$ is a *correcting reversal* for π if $\pi(i) > \pi(i+1) > \ldots > \pi(j)$. An exception is made in the case of a swap: the swap $\sigma\langle i, j \rangle$ is a *correcting swap* for π if $\pi(i) > \pi(j)$. Notice that only the swapped elements are considered when determining whether a swap is correcting or not and that, in general, the relative

orders of other pairs of elements (specifically those pairs that consist of one of the swapped elements and an element between these elements) are affected by the swap.

Section 5.1 proves that, for any permutation, there exists an optimal sorting sequence of short block-moves such that each block-move is a correcting block-move. Using a similar proof, Section 5.2 shows that only correcting generators need to be considered when optimally sorting a permutation with short generators. In Section 5.3, we show that, for $MinSort_{Sw^3}$ (equivalently $MinSort_{Rv^3}$), correcting short swaps are sufficient when determining a sorting sequence for a permutation. Finally, Section 5.4 extends the result to sorting by bounded swaps in general, that is, for $MinSort_{Sw^\ell}$ for any bound ℓ.

5.1 Sorting by Short Block-Moves

Recall that a correcting block-move simply corrects the relative order between all elements moved. Furthermore, recall that an inversion is a pair of elements in π not in their correct relative order. For correcting short block-moves, exactly one inversion is eliminated in the case of a skip ((1,1)-block-move), and exactly two inversions in the case of a hop ((1,2)-block-move or (2,1)-block-move). Non-correcting block-moves, on the other hand, *introduce* inversions. We can therefore associate a metric $newinversions(\pi, \beta)$ to a short block-move β that counts the number of inversions that β introduces to π. Correcting short block-moves have $newinversions(\pi, \beta) = 0$ while non-correcting short block-moves have $newinversions(\pi, \beta) \in \{1, 2\}$. The following is a result that allows us to improve on this metric when a non-correcting block-move is detected.

Lemma 5.1 *Let π be a permutation, and let $\beta_1, \beta_2, \ldots, \beta_k$ be a sequence of short block-moves that sorts π. Suppose β_i is a non-correcting block-move and that $\beta_1, \beta_2, \ldots, \beta_{i-1}$ are correcting block-moves. Then, there exists an alternate sorting sequence of short block-moves such that one of the following is true:*

- *The sequence has length less than k and begins with $\beta_1, \beta_2, \ldots, \beta_{i-1}$.*
- *The sequence has length exactly k and begins with $\beta_1, \beta_2, \ldots, \beta_{i-1}, \hat{\beta}_i$, where $\hat{\beta}_i$ is a short block-move satisfying*

$$newinversions(\pi \cdot \beta_1 \cdot \beta_2 \cdots \beta_{i-1}, \hat{\beta}_i) = newinversions(\pi \cdot \beta_1 \cdot \beta_2 \cdots \beta_{i-1}, \beta_i) - 1.$$

Table 5.1: Possibilities for non-correcting short block-moves.

Case	π^{i-1}	β_i	$\pi^i = \pi^{i-1} \cdot \beta_i$
1	$\ldots\ \mathbf{a}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}, \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{a}\ \ldots$
2	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}, \mathbf{x}\ \mathbf{b} \rangle$	$\ldots\ \mathbf{x}\ \mathbf{b}\ \mathbf{a}\ \ldots$
3	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}\ \mathbf{x}, \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{a}\ \mathbf{x}\ \ldots$
4	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{x}\ \mathbf{a}, \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{x}\ \mathbf{a}\ \ldots$
5	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$	$B\langle \mathbf{a}, \mathbf{b}\ \mathbf{x} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{x}\ \mathbf{a}\ \ldots$

Table 5.2: Alternate short block-moves $\hat{\beta}_i$ for β_i.

Case	π^{i-1}	$\hat{\beta}_i$	$\hat{\pi}^i = \pi^{i-1} \cdot \hat{\beta}_i$
1	$\ldots\ \mathbf{a}\ \mathbf{b}\ \ldots$	$\boldsymbol{\iota}$	$\ldots\ \mathbf{a}\ \mathbf{b}\ \ldots$
2	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}, \mathbf{x} \rangle$	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$
3	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{x}, \mathbf{b} \rangle$	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$
4	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{x}, \mathbf{a} \rangle$	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$
5	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$	$B\langle \mathbf{b}, \mathbf{x} \rangle$	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$

Proof: Let $\pi^r = \pi \cdot \beta_1 \cdot \beta_2 \cdots \beta_r$ denote the permutation after the first r block-moves have been applied to π; in particular, $\pi^0 = \pi$ and $\pi^k = \boldsymbol{\iota}$.

Since β_i is a non-correcting block-move, there are two elements $\mathbf{a}, \mathbf{b}$ in π such that:

1. $\mathbf{a} < \mathbf{b}$,
2. $\mathbf{a}$ occurs to the left of $\mathbf{b}$ in π^{i-1}, and,
3. $\mathbf{a}$ occurs to the right of $\mathbf{b}$ in $\pi^i = \pi^{i-1} \cdot \beta_i$.

Table 5.1 enumerates the 5 cases of possible positions of $\mathbf{a}$ and $\mathbf{b}$ in π^{i-1} and π^i. Notice that a third element $\mathbf{x}$ may be involved in the block-move. For clarity, the block-moves are specified by element values. The goal is to replace β_i with $\hat{\beta}_i$ as enumerated in Table 5.2. The revised block-move $\hat{\beta}_i$ simulates the effect of β_i except that it does not destroy the already correct relative order between elements $\mathbf{a}$ and $\mathbf{b}$. That is, for the cases enumerated, $newinversions(\pi^{i-1}, \hat{\beta}_i) = newinversions(\pi^{i-1}, \beta_i) - 1$. As a consequence, the permutations π^i and $\hat{\pi}^i = \pi^{i-1} \cdot \hat{\beta}_i$ are different only in that $\mathbf{a}$ and $\mathbf{b}$ are switched. We note Case 1 as a special case in this discussion since it is the case where β_i is a skip, so that it suffices to simply omit β_i instead of replacing it with some $\hat{\beta}_i$. Now, since the sequence of block-moves $\beta_1, \beta_2, \ldots, \beta_k$ eventually sorts the permutation π, there is a

Table 5.3: Possibilities where β_j corrects the order of **a** and **b**.

Case	π^{j-1}	β_j	$\pi^j = \pi^{j-1} \cdot \beta_j$
1	$\ldots$ **b a** $\ldots$	$B\langle \mathbf{b}, \mathbf{a} \rangle$	$\ldots$ **a b** $\ldots$
2	$\ldots$ **b y a** $\ldots$	$B\langle \mathbf{y\ a}, \mathbf{b} \rangle$	$\ldots$ **y a b** $\ldots$
3	$\ldots$ **b y a** $\ldots$	$B\langle \mathbf{b\ y}, \mathbf{a} \rangle$	$\ldots$ **a b y** $\ldots$
4	$\ldots$ **y b a** $\ldots$	$B\langle \mathbf{y\ b}, \mathbf{a} \rangle$	$\ldots$ **a y b** $\ldots$
5	$\ldots$ **b a y** $\ldots$	$B\langle \mathbf{b}, \mathbf{a\ y} \rangle$	$\ldots$ **a y b** $\ldots$

Table 5.4: Revised block-moves $\hat{\beta}_j$ for β_j.

Case	$\hat{\pi}^{j-1}$	$\hat{\beta}_j$	$\hat{\pi}^{j-1} \cdot \hat{\beta}_j = \pi^j$
1	$\ldots$ **a b** $\ldots$	$\boldsymbol{\iota}$	$\ldots$ **a b** $\ldots$
2	$\ldots$ **a y b** $\ldots$	$B\langle \mathbf{a}, \mathbf{y} \rangle$	$\ldots$ **y a b** $\ldots$
3	$\ldots$ **a y b** $\ldots$	$B\langle \mathbf{y}, \mathbf{b} \rangle$	$\ldots$ **a b y** $\ldots$
4	$\ldots$ **y a b** $\ldots$	$B\langle \mathbf{y}, \mathbf{a} \rangle$	$\ldots$ **a y b** $\ldots$
5	$\ldots$ **a b y** $\ldots$	$B\langle \mathbf{b}, \mathbf{y} \rangle$	$\ldots$ **a y b** $\ldots$

subsequent block-move β_j ($j > i$) that returns elements **a** and **b** to their correct relative order. Table 5.3 enumerates the possibilities (in the table, a third element **y** may be part of the block-move). Consider $\hat{\pi}^{j-1} = \hat{\pi}^i \cdot \beta_{i+1} \cdots \beta_{j-1}$. Since $\hat{\pi}^i$ is just π^i with elements **a** and **b** switched, $\hat{\pi}^{j-1}$ is just π^{j-1} with elements **a** and **b** switched.

We replace β_j with a block-move $\hat{\beta}_j$ that leaves **a** and **b** in their correct relative positions. Table 5.4 enumerates the corresponding $\hat{\beta}_j$ for each β_j in Table 5.3. Again, note Case 1 as a special case where β_j is simply omitted instead of replaced with some $\hat{\beta}_j$. We may confirm that for all cases except Case 1 in the table, the following now holds: $\hat{\pi}^{j-1} \cdot \hat{\beta}_j = \pi^j$. In effect, by replacing block-moves β_i and β_j with $\hat{\beta}_i$ and $\hat{\beta}_j$, we yield the same permutation. The transformation is summarized in the following equation:

$$\pi \cdot \beta_1 \cdot \beta_2 \cdots \beta_k = \pi \cdot \beta_1 \cdot \cdots \beta_{i-1} \cdot \hat{\beta}_i \cdot \beta_{i+1} \cdots \beta_{j-1} \cdot \hat{\beta}_j \cdot \beta_{j+1} \cdots \beta_k,$$

Whenever Case 1 holds for either β_i or β_j in the derivation, we have a shorter sequence that starts with $\beta_1, \beta_2, \ldots \beta_{i-1}$. Otherwise, we have the sequence above of length k such that

$$newinversions(\pi^{i-1}, \hat{\beta}_i) = newinversions(\pi^{i-1}, \beta_i) - 1.$$

□

The above lemma allows us to transform an optimal sorting sequence into a sequence that contains only correcting block-moves. The goal is to show that only correcting block-moves need to be considered when solving $MinSort_{Bk^3}$.

Theorem 5.2 *For a permutation π, there exists an optimal sequence of short block-moves $\beta_1, \beta_2, \ldots, \beta_k$ that sorts π such that each block-move is a correcting block-move.*

Proof: We convert a given optimal sequence to an alternate sequence of correcting short block-moves by repeatedly applying the transformation process given in Lemma 5.1. The conversion replaces a non-correcting block-move β_i with a correcting block-move in either one or two applications of Lemma 5.1, depending on the value of $newinversions(\pi, \beta_i)$. Since the block-moves $\beta_1, \beta_2, \ldots, \beta_{i-1}$ are unchanged and remain as correcting block-moves at each step, the conversion process eventually terminates. □

It is enlightening to note that Case 1 never holds for either β_i or β_j in any step of the conversion described in the above proof, since this would imply a shorter resulting sorting sequence (a contradiction, since the proof starts with an optimal sequence).

5.2 Sorting by Short Generators

In this section, we consider $MinSort_{Gn^3}$, minimum sorting by short generators. A short generator permutes any three contiguous elements in a permutation. Recall that short block-moves and short reversals comprise the set of short generators. Also, recall that a correcting generator corrects the relative order between all pairs of elements repositioned by the generator; in particular, a correcting generator does not introduce inversions. The following is a lemma (similar to Lemma 5.1 in the previous section) that allows us to replace a non-correcting short generator with one that introduces fewer inversions. As in the previous section, $newinversions(\pi, \gamma)$ represents the number of inversions introduced by the generator γ when it is applied to π.

Lemma 5.3 *Let π be a permutation, and let $\gamma_1, \gamma_2, \ldots, \gamma_k$ be an optimal sequence of short generators that sorts π. Suppose γ_i is a non-correcting generator and that $\gamma_1, \gamma_2, \ldots, \gamma_{i-1}$ are correcting generators. Then, there exists an alternate optimal sorting sequence of short generators that begins*

Table 5.5: Possibilities for non-correcting short generators.

Case	π^{i-1}	γ_i	$\pi^i = \pi^{i-1} \cdot \gamma_i$
1	$\ldots\ \mathbf{a}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}, \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{a}\ \ldots$
2	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}, \mathbf{x}\ \mathbf{b} \rangle$	$\ldots\ \mathbf{x}\ \mathbf{b}\ \mathbf{a}\ \ldots$
3	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}\ \mathbf{x}, \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{a}\ \mathbf{x}\ \ldots$
4	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$R\langle \mathbf{a}\ \mathbf{x}\ \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{x}\ \mathbf{a}\ \ldots$
5	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{x}\ \mathbf{a}, \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{x}\ \mathbf{a}\ \ldots$
6	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$	$R\langle \mathbf{x}\ \mathbf{a}\ \mathbf{b} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{a}\ \mathbf{x}\ \ldots$
7	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$	$B\langle \mathbf{a}, \mathbf{b}\ \mathbf{x} \rangle$	$\ldots\ \mathbf{b}\ \mathbf{x}\ \mathbf{a}\ \ldots$
8	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$	$R\langle \mathbf{a}\ \mathbf{b}\ \mathbf{x} \rangle$	$\ldots\ \mathbf{x}\ \mathbf{b}\ \mathbf{a}\ \ldots$

Table 5.6: Alternate short generators $\hat{\gamma}_i$ for γ_i.

Case	π^{i-1}	$\hat{\gamma}_i$	$\hat{\pi}^i = \pi^{i-1} \cdot \hat{\gamma}_i$
1	$\ldots\ \mathbf{a}\ \mathbf{b}\ \ldots$	$\boldsymbol{\iota}$	$\ldots\ \mathbf{a}\ \mathbf{b}\ \ldots$
2	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{a}, \mathbf{x} \rangle$	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$
3	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{x}, \mathbf{b} \rangle$	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$
4	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$	$\boldsymbol{\iota}$	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$
5	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{x}, \mathbf{a} \rangle$	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$
6	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$	$B\langle \mathbf{x}, \mathbf{a}\ \mathbf{b} \rangle$	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$
7	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$	$B\langle \mathbf{b}, \mathbf{x} \rangle$	$\ldots\ \mathbf{a}\ \mathbf{x}\ \mathbf{b}\ \ldots$
8	$\ldots\ \mathbf{a}\ \mathbf{b}\ \mathbf{x}\ \ldots$	$B\langle \mathbf{a}\ \mathbf{b}, \mathbf{x} \rangle$	$\ldots\ \mathbf{x}\ \mathbf{a}\ \mathbf{b}\ \ldots$

with $\gamma_1, \gamma_2, \ldots, \gamma_{i-1}, \hat{\gamma}_i$, *where* $\hat{\gamma}_i$ *is a short generator satisfying*

$$newinversions(\pi \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_{i-1}, \hat{\gamma}_i) < newinversions(\pi \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_{i-1}, \gamma_i).$$

Proof: A proof similar to Lemma 5.1 applies. That is, we replace the non-correcting generator in the given optimal sequence with an alternate generator that introduces fewer inversions. Table 5.5 enumerates the possibilities for a short generator γ_i that destroys the relative order between two elements $\mathbf{a}$ and $\mathbf{b}$ while Table 5.6 lists the corresponding alternate short generators $\hat{\gamma}_i$. It can be verified that these are the only cases possible and that for each case, the alternate generator $\hat{\gamma}_i$ introduces fewer inversions than γ_i. In fact, the permutations $\pi^i = \pi^{i-1} \cdot \gamma_i$ and $\hat{\pi}^i = \pi^{i-1} \cdot \hat{\gamma}_i$ are different only in that the elements $\mathbf{a}$ and $\mathbf{b}$ are switched. Equivalently,

$$\pi^{i-1} \cdot \gamma_i \cdot S\langle \mathbf{a}, \mathbf{b} \rangle = \pi^{i-1} \cdot \hat{\gamma}_i.$$

Table 5.7: Possibilities where γ_j corrects the order of **a** and **b**.

Case	π^{j-1}	γ_j	$\pi^j = \pi^{j-1} \cdot \gamma_j$
1	$\ldots$ **b a** $\ldots$	$B\langle \mathbf{b}, \mathbf{a} \rangle$	$\ldots$ **a b** $\ldots$
2	$\ldots$ **b y a** $\ldots$	$B\langle \mathbf{y\ a}, \mathbf{b} \rangle$	$\ldots$ **y a b** $\ldots$
3	$\ldots$ **b y a** $\ldots$	$B\langle \mathbf{b\ y}, \mathbf{a} \rangle$	$\ldots$ **a b y** $\ldots$
4	$\ldots$ **b y a** $\ldots$	$R\langle \mathbf{b\ y\ a} \rangle$	$\ldots$ **a y b** $\ldots$
5	$\ldots$ **y b a** $\ldots$	$B\langle \mathbf{y\ b}, \mathbf{a} \rangle$	$\ldots$ **a y b** $\ldots$
6	$\ldots$ **y b a** $\ldots$	$R\langle \mathbf{y\ b\ a} \rangle$	$\ldots$ **a b y** $\ldots$
7	$\ldots$ **b a y** $\ldots$	$B\langle \mathbf{b}, \mathbf{a\ y} \rangle$	$\ldots$ **a y b** $\ldots$
8	$\ldots$ **b a y** $\ldots$	$R\langle \mathbf{b\ a\ y} \rangle$	$\ldots$ **y a b** $\ldots$

Table 5.8: Revised generators $\hat{\gamma}_j$ for γ_j.

Case	$\hat{\pi}^{j-1}$	$\hat{\gamma}_j$	$\hat{\pi}^{j-1} \cdot \hat{\gamma}_j = \pi^j$
1	$\ldots$ **a b** $\ldots$	$\boldsymbol{\iota}$	$\ldots$ **a b** $\ldots$
2	$\ldots$ **a y b** $\ldots$	$B\langle \mathbf{a}, \mathbf{y} \rangle$	$\ldots$ **y a b** $\ldots$
3	$\ldots$ **a y b** $\ldots$	$B\langle \mathbf{y}, \mathbf{b} \rangle$	$\ldots$ **a b y** $\ldots$
4	$\ldots$ **a y b** $\ldots$	$\boldsymbol{\iota}$	$\ldots$ **a y b** $\ldots$
5	$\ldots$ **y a b** $\ldots$	$B\langle \mathbf{y}, \mathbf{a} \rangle$	$\ldots$ **a y b** $\ldots$
6	$\ldots$ **y a b** $\ldots$	$B\langle \mathbf{y}, \mathbf{a\ b} \rangle$	$\ldots$ **a b y** $\ldots$
7	$\ldots$ **a b y** $\ldots$	$B\langle \mathbf{b}, \mathbf{y} \rangle$	$\ldots$ **a y b** $\ldots$
8	$\ldots$ **a b y** $\ldots$	$B\langle \mathbf{a\ b}, \mathbf{y} \rangle$	$\ldots$ **y a b** $\ldots$

In foresight, we note that Cases 1 and 4 will not occur in an optimal sequence as that would imply a sequence shorter than the optimal (See Lemma 5.1 and Theorem 5.2). As in the proof of Lemma 5.1, a subsequent generator γ_j (the one that returns **a** and **b** to their correct relative order in the original sequence) needs to be replaced accordingly by some $\hat{\gamma}_j$ (the one that leaves the elements **a** and **b** in their already correct relative order) so that the sequence remains a sorting sequence. The possibilities and replacements are given in Tables 5.7 and 5.8. Again, we note Cases 1 and 4 as cases that will not occur. We thus have our desired result, that is,

$$\pi \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_k = \pi \cdot \gamma_1 \cdot \cdots \cdot \gamma_{i-1} \cdot \hat{\gamma}_i \cdot \gamma_{i+1} \cdots \gamma_{j-1} \cdot \hat{\gamma}_j \cdot \gamma_{j+1} \cdots \gamma_k,$$

such that

$$newinversions(\pi \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_{i-1}, \hat{\gamma}_i) < newinversions(\pi \cdot \gamma_1 \cdot \gamma_2 \cdots \gamma_{i-1}, \gamma_i).$$

□

Of course, the goal is to show that it suffices to consider only correcting generators when seeking an optimal sorting sequence for a permutation.

Theorem 5.4 *For a permutation π, there exists an optimal sequence of short generators $\gamma_1, \gamma_2, \ldots, \gamma_k$ that sorts π such that each generator is a correcting generator.*

Proof: Repeatedly applying Lemma 5.3 to any optimal sorting sequence for π eventually results in an optimal sequence of generators, none of which introduces inversions. The theorem follows. □

5.3 Sorting by Short Swaps (Short Reversals)

In $MinSort_{Sw^3}$, the set of allowable generators includes 2-swaps (adjacent element swaps) and 3-swaps (swaps between elements that have exactly one element between them). First, note that the problem is equivalent to $MinSort_{Rv^3}$ where 2-swaps and 3-swaps are just 2-reversals and 3-reversals, respectively. Also, since we have defined a correcting swap as one that corrects the relative order of the swapped elements, correcting short swaps may cause the middle element to be placed out of order with respect to one of the swapped elements. The following lemma states that given a sorting sequence of short swaps, a non-correcting swap can always be eliminated without lengthening the sorting sequence.

Lemma 5.5 *Let π be a permutation, and let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be a sequence of short swaps that sorts π. Suppose σ_i is a non-correcting swap and that $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ are correcting swaps. Then, there exists an alternate sorting sequence of short swaps that begins with $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$, and one of the following is true:*

- *The sequence has length less than k.*
- *The sequence has length exactly k and has fewer 3-swaps than the original sequence.*

Proof: Let $\pi^r = \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_r$ denote the permutation after the first r swaps have been applied to π and let $\pi^0 = \pi$.

Suppose $\sigma_i = \sigma\langle p, q\rangle$. Since σ_i is not a correcting swap, it swaps two elements, $\mathbf{a} = \pi^{i-1}(p)$ and $\mathbf{b} = \pi^{i-1}(q)$, that are already in their correct relative order; that is, $\mathbf{a} < \mathbf{b}$. Also, since the

sequence of swaps eventually sorts the permutation π, there is a subsequent swap σ_j $(j > i)$ that causes the elements **a** and **b** to be returned to their correct relative order. The strategy is to remove the "offending" swap σ_i from the sequence and (if necessary) replace σ_j with two swaps $\hat{\sigma}_j$ and $\breve{\sigma}_j$ while preserving the effect of the sorting sequence on π. That is, we want

$$\begin{aligned}
\pi^k &= \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_k \\
&= \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1} \cdot \hat{\sigma}_j \cdot \breve{\sigma}_j \cdot \sigma_{j+1} \cdots \sigma_k \\
&= \boldsymbol{\iota}.
\end{aligned}$$

Observe that π^{i-1} is just π^i with elements **a** and **b** switched so that $\hat{\pi}^{j-1} = \pi^{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1}$ is just π^{j-1} with elements **a** and **b** switched; that is, $\hat{\pi}^{j-1} = \pi^{j-1} \cdot S\langle \mathbf{a}, \mathbf{b} \rangle$. If σ_j simply swaps elements **a** and **b**, that is, $\sigma_j = S\langle \mathbf{a}, \mathbf{b} \rangle$, then removing σ_j from the sequence is sufficient because

$$\begin{aligned}
\pi^j &= \pi^{j-1} \cdot \sigma_j \\
&= \pi^{j-1} \cdot S\langle \mathbf{a}, \mathbf{b} \rangle \\
&= \hat{\pi}^{j-1} \\
&= \pi^{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1} \\
&= \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1}.
\end{aligned}$$

Hence,

$$\pi^k = \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_k = \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1} \cdot \sigma_{j+1} \cdots \sigma_k,$$

a sequence that begins with $\beta_1, \beta_2, \ldots, \beta_{i-1}$, and is shorter than the original sequence.

However, it is possible that σ_j does not swap elements **a** and **b**. This occurs when σ_j is a 3-swap such that one of **a** and **b** is a middle element, thereby *indirectly* causing a change in the relative order of these two elements. It remains to show that the theorem holds in this case.

Table 5.9 lists the two possible ways **a** and **b** can be indirectly repositioned by a 3-swap. For clarity, we use values instead of indices when specifying the swaps. In the table, a third element **x** indicates one of the swapped elements. The intention is to replace σ_j with two 2-swaps that produce the same resulting permutation. Table 5.10 exhibits these replacements.

Recall that π^{j-1} is just $\hat{\pi}^{j-1}$ with elements **a** and **b** switched. It can be verified that for the two cases given, π^{j-1} composed with σ_j produces the same permutation when $\hat{\pi}^{j-1}$ is composed with $\hat{\sigma}_j$ and $\breve{\sigma}_j$ as exhibited in Tables 5.9 and 5.10. We conclude

$$\pi^k = \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1} \cdot \hat{\sigma}_j \cdot \breve{\sigma}_j \cdot \sigma_{j+1} \cdots \sigma_k,$$

Table 5.9: Cases where σ_j indirectly changes the relative order of **a** and **b**.

Case	π^{j-1}	σ_j	$\pi^j = \pi^{j-1} \cdot \sigma_i$
1	... **b a x** ...	$S\langle \mathbf{b}, \mathbf{x} \rangle$	... **x a b** ...
2	... **x b a** ...	$S\langle \mathbf{x}, \mathbf{a} \rangle$	... **a b x** ...

Table 5.10: Replacements $\hat{\sigma}_j, \breve{\sigma}_j$ for σ_j.

Case	$\hat{\pi}^{j-1} = \pi^{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1}$.	$\hat{\sigma}_j, \breve{\sigma}_j$	$\hat{\pi}^{j-1} \cdot \hat{\sigma}_j \cdot \breve{\sigma}_j = \pi^j$
1	... **a b x** ...	$S\langle \mathbf{b}, \mathbf{x} \rangle, S\langle \mathbf{a}, \mathbf{x} \rangle$	... **x a b** ...
2	... **x a b** ...	$S\langle \mathbf{x}, \mathbf{a} \rangle, S\langle \mathbf{x}, \mathbf{b} \rangle$	... **a b x** ...

where the sorting sequence begins with $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$, the total number of swaps stays the same, and the 3-swap σ_j has been effectively replaced with two 2-swaps $\hat{\sigma}_j$ and $\breve{\sigma}_j$, resulting in fewer 3-swaps, which completes the proof. □

We use the above lemma to prove the following theorem which states that, in an optimal sequence, each swap can be chosen such that the relative order of the two swapped elements is always corrected.

Theorem 5.6 *For a permutation π, there exists an optimal sequence of short swaps $\sigma_1, \sigma_2, \ldots, \sigma_k$ that sorts π such that each swap is a correcting swap.*

Proof: Given an optimal sorting sequence of short swaps, the transformation presented in Lemma 5.5 is simply repeated until all swaps are correcting swaps. It is easy to verify that the process will end in a finite number of steps. First, notice that the first case presented in the lemma will not occur since we start with an optimal sequence (the resulting sequence can not be shorter). Each step in the transformation therefore implies a reduction in the total number of 3-swaps. Since there are only a finite number of 3-swaps in the original optimal sequence, the process terminates. □

To demonstrate a step in this transformation process, consider the permutation $\pi =$ **2 3 5 1 6 4**. An optimal sequence of swaps for π is $\sigma_1\langle 1, 2\rangle$, $\sigma_2\langle 3, 5\rangle$, $\sigma_3\langle 3, 4\rangle$, $\sigma_4\langle 1, 3\rangle$, $\sigma_5\langle 4, 6\rangle$. The intermediate permutations are shown in Figure 5.1.

Here, the swap σ_1 does not correct the relative order of the swapped elements so this is subject to elimination by an application of Lemma 5.5. The 3-swap $\sigma_4\langle 1, 3\rangle$ is replaced with the two 2-swaps

$$
\begin{aligned}
\pi^0 = \pi &= \mathbf{2\ 3\ 5\ 1\ 6\ 4} \\
\pi^1 = \pi^0 \cdot \sigma_1\langle 1,2\rangle &= \mathbf{3\ 2\ 5\ 1\ 6\ 4} \\
\pi^2 = \pi^1 \cdot \sigma_2\langle 3,5\rangle &= \mathbf{3\ 2\ 6\ 1\ 5\ 4} \\
\pi^3 = \pi^2 \cdot \sigma_3\langle 3,4\rangle &= \mathbf{3\ 2\ 1\ 6\ 5\ 4} \\
\pi^4 = \pi^3 \cdot \sigma_4\langle 1,3\rangle &= \mathbf{1\ 2\ 3\ 6\ 5\ 4} \\
\pi^5 = \pi^4 \cdot \sigma_5\langle 4,6\rangle &= \mathbf{1\ 2\ 3\ 4\ 5\ 6}
\end{aligned}
$$

Figure 5.1: An optimal sequence of swaps.

$$
\begin{aligned}
\pi^0 = \pi &= \mathbf{2\ 3\ 5\ 1\ 6\ 4} \\
\hat{\pi}^1 = \pi^0 \cdot \sigma_2\langle 3,5\rangle &= \mathbf{2\ 3\ 6\ 1\ 5\ 4} \\
\hat{\pi}^2 = \hat{\pi}^1 \cdot \sigma_3\langle 3,4\rangle &= \mathbf{2\ 3\ 1\ 6\ 5\ 4} \\
\hat{\pi}^3 = \hat{\pi}^2 \cdot \hat{\sigma}_4\langle 2,3\rangle &= \mathbf{2\ 1\ 3\ 6\ 5\ 4} \\
\hat{\pi}^4 = \hat{\pi}^3 \cdot \breve{\sigma}_4\langle 1,2\rangle &= \mathbf{1\ 2\ 3\ 6\ 5\ 4} \\
\hat{\pi}^5 = \hat{\pi}^4 \cdot \sigma_5\langle 4,6\rangle &= \mathbf{1\ 2\ 3\ 4\ 5\ 6}
\end{aligned}
$$

Figure 5.2: An optimal sequence of swaps that has fewer 3-swaps.

$\hat{\sigma}_j\langle 2,3\rangle$ and $\breve{\sigma}_j\langle 1,2\rangle$ to obtain a revised sequence of swaps as shown in Figure 5.2. The next step, of course, is to eliminate σ_2, another swap that does not correct the relative order of the swapped elements. The reader can verify that another application of Lemma 5.5 completes the transformation described in Theorem 5.6, for this example.

5.4 Sorting by ℓ-Bounded Swaps

Recall that the predicate Sw^ℓ describes ℓ-bounded swaps. The integer ℓ bounds the extent of a swap so that $Sw^\ell[n]$ contains all m-swaps having extent $m \leq \ell$. Section 5.3 addresses the special case of $MinSort_{Sw^3}$. In this section, we generalize this result to $MinSort_{Sw^\ell}$, for all bounds ℓ. First, we introduce a new measure for a sorting sequence. Let π be a permutation and let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be a sequence of swaps that sorts π. Also, let $m_1, m_2, \ldots m_k$ denote the respective extents of the swaps $\sigma_1, \sigma_2, \ldots, \sigma_k$. Define the *total extent* of a sorting sequence as the sum of the extents of all the swaps

in the sequence. That is,

$$totextent(\sigma_1, \sigma_2, \ldots, \sigma_k) = \sum_{i=1}^{k} m_i.$$

Since a swap has extent at least 2, the total extent of a sorting sequence of length k is at least $2k$. The following lemma allows us to eliminate a non-correcting ℓ-bounded swap in a sequence and, at the same time, reduce the total extent of the sequence.

Lemma 5.7 *Let π be a permutation, and let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be an optimal sequence of ℓ-bounded swaps that sorts π. Suppose σ_i is a non-correcting swap and $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ are correcting swaps. Then, there exists an alternate optimal sorting sequence of short swaps that begins with $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$, and has total extent less than the total extent of the original sequence.*

Proof: We present a proof similar to that of Lemma 5.5 in the previous section. We omit some details and instead emphasize the main differences between that proof and the current one. The same strategy applies; that is, we eliminate the swap σ_i from the sequence, locate a subsequent swap σ_j, and then replace σ_j with two swaps $\hat{\sigma}_j$ and $\breve{\sigma}_j$. The swap σ_i swaps two elements **a** and **b** already in their correct relative order while σ_j returns these elements to their correct order.

Table 5.11 presents the two possibilities for σ_j. We exclude the possibility that σ_j simply swaps elements **b** and **a** (see Lemma 5.5 and Theorem 5.6). Thus, elements **a** and **b** are indirectly returned to their correct relative order as given in the table; that is, one of **a** and **b** is swapped with a third element **x** and the other element occurs between these two elements. In the table, each of the subsequences V and W represents the other elements between two of these three elements in π^{j-1}.

Table 5.12, on the other hand, specifies the replacements $\hat{\sigma}_j$ and $\breve{\sigma}_j$ for σ_j. It can be verified that we have achieved an alternate sorting sequence; that is,

$$\begin{aligned}
\pi^k &= \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_k \\
&= \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1} \cdot \hat{\sigma}_j \cdot \breve{\sigma}_j \cdot \sigma_{j+1} \cdots \sigma_k \\
&= \boldsymbol{\iota}.
\end{aligned}$$

Let $\hat{m}_j$ and $\breve{m}_j$ be the extents of $\hat{\sigma}_j$ and $\breve{\sigma}_j$, respectively. It remains to show that $\hat{m}_j + \breve{m}_j < m_i + m_j$ for the lemma to hold. Clearly, by considering the positions of **a**, **b**, and **x** as given in the tables, $\hat{m}_j + \breve{m}_j = m_j + 1$, for both cases. In addition, $m_i \geq 2$ (a swap has extent at least 2). It follows that $\hat{m}_j + \breve{m}_j < m_i + m_j$ and that

$$totextent(\sigma_1, \sigma_2, \ldots, \sigma_{i-1}, \sigma_{i+1}, \ldots, \sigma_{j-1}, \hat{\sigma}_j, \breve{\sigma}_j, \sigma_{j+1}, \ldots, \sigma_k) < totextent(\sigma_1, \sigma_2, \ldots, \sigma_k).$$

Table 5.11: Cases where an ℓ-bounded swap σ_j changes the relative order of $\mathbf{a}$ and $\mathbf{b}$.

Case	π^{j-1}	σ_j	$\pi^j = \pi^{j-1} \cdot \sigma_i$
1	$\ldots\ \mathbf{b}\ V\ \mathbf{a}\ W\ \mathbf{x}\ \ldots$	$S\langle \mathbf{b}, \mathbf{x} \rangle$	$\ldots\ \mathbf{x}\ V\ \mathbf{a}\ W\ \mathbf{b}\ \ldots$
2	$\ldots\ \mathbf{x}\ V\ \mathbf{b}\ W\ \mathbf{a}\ \ldots$	$S\langle \mathbf{x}, \mathbf{a} \rangle$	$\ldots\ \mathbf{a}\ V\ \mathbf{b}\ W\ \mathbf{x}\ \ldots$

Table 5.12: Replacements $\hat{\sigma}_j, \breve{\sigma}_j$ for the ℓ-bounded swap σ_j.

Case	$\hat{\pi}^{j-1} = \pi^{i-1} \cdot \sigma_{i+1} \cdots \sigma_{j-1}$.	$\hat{\sigma}_j, \breve{\sigma}_j$	$\hat{\pi}^{j-1} \cdot \hat{\sigma}_j \cdot \breve{\sigma}_j = \pi^j$
1	$\ldots\ \mathbf{a}\ V\ \mathbf{b}\ W\ \mathbf{x}\ \ldots$	$S\langle \mathbf{b}, \mathbf{x} \rangle, S\langle \mathbf{a}, \mathbf{x} \rangle$	$\ldots\ \mathbf{x}\ V\ \mathbf{a}\ W\ \mathbf{b}\ \ldots$
2	$\ldots\ \mathbf{x}\ V\ \mathbf{a}\ W\ \mathbf{b}\ \ldots$	$S\langle \mathbf{x}, \mathbf{a} \rangle, S\langle \mathbf{x}, \mathbf{b} \rangle$	$\ldots\ \mathbf{a}\ V\ \mathbf{b}\ W\ \mathbf{x}\ \ldots$

□

Theorem 5.8 *For a permutation π, there exists an optimal sequence of ℓ-bounded swaps $\sigma_1, \sigma_2, \ldots, \sigma_k$ that sorts π such that each swap is a correcting swap.*

Proof: Repeatedly applying Lemma 5.7 to any optimal sorting sequence results in an optimal sequence consisting entirely of correcting ℓ-bounded swaps since, at each step, the first non-correcting swap is always eliminated from the sequence. The process is guaranteed to terminate because the total extent of the sequence can be at most $2k$ and it strictly decreases at each step in the conversion.

□

Chapter 6

Short Block-Moves

In this chapter, results for $MinSort_{Bk^3}$ and $Diam_{Bk^3}$ are presented. Recall that Bk^3 is the predicate that describes short block-moves. Section 6.1 defines permutations graphs, the graph model we use in investigating these problems. We illustrate how correcting short block-moves map to arcs in a permutation graph and then establish bounds for short block-move distance. In Section 6.2, we introduce the notion of pending elements in a permutation and show how it relates to permutation graphs. Section 6.3 builds on the permutation graph model and defines arc graphs to more accurately model optimal sorting sequences for permutations. A tighter lower bound for short block-move distance is then established. Section 6.4 defines special types of permutations where the models we have developed apply. In Sections 6.5 and 6.6, we devise algorithms to solve $MinSort_{Bk^3}$ for these special types of permutations. Section 6.7 establishes the value for $Diam_{Bk^3}(n)$, the short block-move diameter for S_n. In Section 6.8, we devise an algorithm that sorts any permutation with a sequence of short block-moves whose length is within $\frac{4}{3}$ of the optimal sequence. Finally, in Section 6.9, we use the permutation graph model to obtain results for sorting by short generators.

6.1 Permutation Graphs

By Theorem 5.2, it is sufficient to consider only correcting short block-moves when seeking an optimal sorting sequence. This leads to the following graph representation of a permutation π. The *permutation graph* of π is the directed graph $\mathcal{G}_\pi^p = (\mathcal{V}_\pi^p, \mathcal{A}_\pi^p)$ where $\mathcal{V}_\pi^p = \{\pi(1), \pi(2), \ldots, \pi(n)\}$ and $(\pi(i), \pi(j)) \in \mathcal{A}_\pi^p$ whenever $\pi(i) > \pi(j)$ and $i < j$. Intuitively, the arcs of a permutation

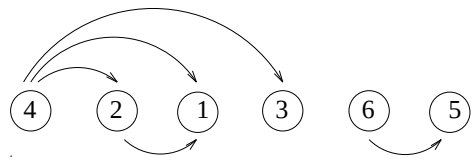

Figure 6.1: The permutation graph for $\pi = \mathbf{4\ 2\ 1\ 3\ 6\ 5}$.

Table 6.1: Arcs that correspond to an optimal block-move sequence ($\pi = \mathbf{4\ 2\ 1\ 3\ 6\ 5}$).

Block-move	Arcs	Permutation graph
$B_1\langle \mathbf{4\ 2, 1}\rangle$	$(4,1),(2,1)$	Figure 6.2(a)
$B_2\langle \mathbf{4, 2\ 3}\rangle$	$(4,2),(4,3)$	Figure 6.2(b)
$B_3\langle \mathbf{6, 5}\rangle$	$(6,5)$	Figure 6.2(c)

graph indicate those pairs of elements that are not in their correct relative order; that is, every arc represents an inversion in π. Figure 6.1 illustrates the permutation graph for $\pi = \mathbf{4\ 2\ 1\ 3\ 6\ 5}$. Permutation graphs have been studied extensively (e.g., [31]) and are examples of transitive graphs [30].

It can be verified that a correcting short block-move corresponds to the removal of one or two arcs from the corresponding permutation graph. Specifically, a skip corresponds to the removal of a single arc in the graph (an inversion in π is eliminated). On the other hand, a hop corresponds to the removal of two arcs in the graph since two inversions are eliminated. In either case, the other arcs in the graph are not affected since the relationships among the other vertices remain the same. Consider again the permutation $\pi = \mathbf{4\ 2\ 1\ 3\ 6\ 5}$, whose permutation graph is shown in Figure 6.1. An optimal sorting sequence for π is:

$$B_1\langle \mathbf{4\ 2, 1}\rangle, B_2\langle \mathbf{4, 2\ 3}\rangle, B_3\langle \mathbf{6, 5}\rangle.$$

Figure 6.2 illustrates the effect of this sequence on the permutation graph of π. Table 6.1 shows the correspondences of the block-moves with the arcs of $\mathcal{G}_\pi^p$ drawn in dashed arcs in the figure. In general, the correcting block-move $B\langle \mathbf{b}, \mathbf{a}\rangle$ corresponds to the arc (b, a). The correcting block-move $B\langle \mathbf{a}, \mathbf{b\ c}\rangle$ corresponds to the arcs (a, b) and (a, c). Finally, the correcting block-move $B\langle \mathbf{b\ c}, \mathbf{a}\rangle$ corresponds to the arcs (b, a) and (c, a).

Two arcs in a permutation graph are *compatible* if their heads are identical or their tails are

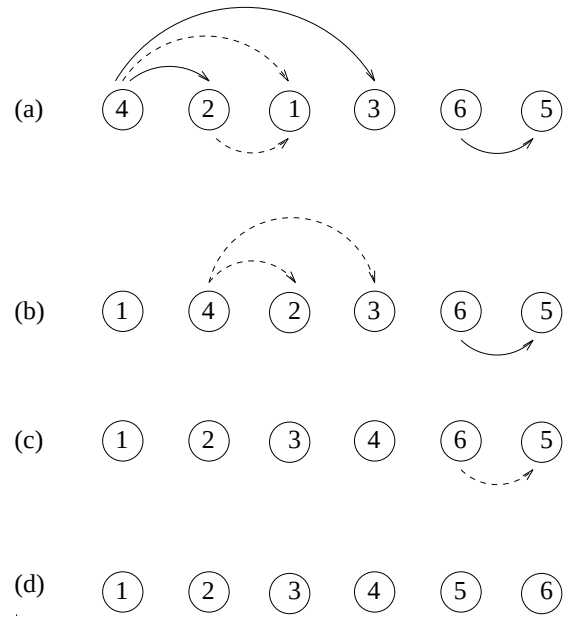

Figure 6.2: Effects of correcting short block-moves on a permutation graph.

identical. The following proposition summarizes the potential of permutation graphs as a model for sorting by short block-moves.

Proposition 6.1 *Let π be a permutation, and let $\beta_1, \beta_2, \ldots, \beta_k$ be an optimal sequence of correcting short block-moves that sorts π. The sequence corresponds to a set D_π of arc-disjoint subgraphs of $\mathcal{G}_\pi^p$ where each subgraph is either a single arc or a pair of compatible arcs.*

The set D_π in Proposition 6.1 is called an *arc-partition* of $\mathcal{G}_\pi^p$, since it partitions the arcs of the graph into single arcs or pairs of compatible arcs. In seeking an optimal sorting sequence, we want to reverse the process; that is, first obtain the arc-partition and then derive the optimal sequence. It should be evident that the goal is to maximize the number of arc-pairs in the partition since this minimizes the total number of subgraphs derived from $\mathcal{G}_\pi^p$, and, equivalently, the number of block-moves that sort π. By Theorem 5.2, there exists an optimal sorting sequence of short block-moves such that each block-move in the sequence removes at least one arc from $\mathcal{G}_\pi^p$. This means that in the best case, all arcs are paired; in the worst case, none of the arcs are paired. Bounds for the length of an optimal sequence of block-moves that sorts π, $MinSort_{Bk^3}(\pi)$, can therefore be determined just from the size of $\mathcal{A}_\pi^p$ as stated in the following lemma.

Lemma 6.2 *The short block-move distance of a permutation π satisfies the inequality:*

$$\lceil |\mathcal{A}_\pi^p|/2 \rceil \leq MinSort_{Bk^3}(\pi) \leq |\mathcal{A}_\pi^p|.$$

The arcs paired in $\mathcal{A}_\pi^p$ have to be compatible (have identical heads or tails)—this is a necessary condition for the pair to correspond to a hop in a sorting sequence. As a simple example, the permutation $\pi = \mathbf{5\ 1\ 2\ 3\ 4\ 6}$ can be sorted optimally with hops $B\langle \mathbf{5}, \mathbf{1\ 2}\rangle$ and $B\langle \mathbf{5}, \mathbf{3\ 4}\rangle$. Observe that the corresponding arcs in $\mathcal{A}_\pi^p$ all have the element **5** as their tail. The next section characterizes this particular situation.

6.2 Pending Elements

An element $\mathbf{a} = \pi(p)$ in a permutation π is *right d-pending* for some integer d, where $0 < d \leq n-1$, if all the d elements to its immediate right are less than $\mathbf{a}$; that is, $\pi(q) < \mathbf{a}$ for all q satisfying $p+1 \leq q \leq p+d$. A symmetric definition holds for a *left d-pending* element; that is, an element $\mathbf{a} = \pi(p)$ in a permutation π is left d-pending if $\pi(q) > \mathbf{a}$ for all q satisfying $p-d \leq q \leq p-1$.

Right (left) d-pending elements are associated with a non-zero magnitude (d) and a direction (right or left). In our discussion, we sometimes simplify by omitting one of these two parts. For instance, *right (left) pending* means the element is right (left) d-pending for some d. Similarly, a *d-pending* element means a right d-pending or left d-pending element. It should be clear that a d-pending element, where $d > 1$, is also a $(d-1)$-pending element.

Right (left) pending elements are important in that correcting short block-moves can be immediately performed by moving the element $\mathbf{a}$ to its right (left) in π. The following lemma is a straightforward result that specifies the correcting hops that can be applied to a permutation given that a right or left d-pending element is identified. Recall that $\mathcal{G}_\pi^p = (\mathcal{V}_\pi^p, \mathcal{A}_\pi^p)$ is the permutation graph of the permutation π.

Lemma 6.3 *Let π be a permutation and let $\mathbf{a} = \pi(p)$ be a d-pending element in π such that d is even. Then, there is sequence of correcting hops $\beta_1, \beta_2, \ldots, \beta_{d/2}$, that, when applied to π results in a permutation*

$$\pi' = \pi \cdot \beta_1 \cdot \beta_2 \cdots \beta_{d/2},$$

such that $|\mathcal{A}_{\pi'}^p| = |\mathcal{A}_\pi^p| - d$. Specifically, if $\mathbf{a}$ is right pending, then we have

$$\beta_1 \quad = \quad \beta\langle p, p+1, p+3\rangle,$$

$$\begin{aligned}
\beta_2 &= \beta\langle p+2, p+3, p+5\rangle, \\
&\vdots \\
\beta_{d/2} &= \beta\langle p+d-2, p+d-1, p+d+1\rangle.
\end{aligned}$$

If **a** *is left pending, then we have*

$$\begin{aligned}
\beta_1 &= \beta\langle p-2, p, p+1\rangle, \\
\beta_2 &= \beta\langle p-4, p-2, p-1\rangle, \\
&\vdots \\
\beta_{d/2} &= \beta\langle p-d, p-d+2, p-d+3\rangle.
\end{aligned}$$

A naive algorithm to sort π using short block-moves is to repeatedly detect d-pending elements and perform the corresponding hops. This is unfortunately insufficient because there will not always be a d-pending element where d is even. For example, **2 1 3 5 4 6** is a permutation where all of its pending elements are 1-pending. With this notion of pending elements, however, we have imposed some method of pairing the arcs in $\mathcal{G}_\pi^p$. In the next section, we reinvestigate the constraints necessary to pair such arcs.

6.3 Arc Graphs

Recall that a necessary condition for a pair of arcs in $\mathcal{A}_\pi^p$ to be a hop is that they be compatible. This condition is by no means sufficient. It is possible that a pair of compatible arcs may not be realized as a hop. Consider the example in Figure 6.3. The compatible arc-pair $(2,1)-(6,1)$ can not be a realized as a hop because the element **4** occurs between elements **2** and **6** in the permutation and will therefore not "get out of the way" as it is already in its correct relative order with respect to elements **2** and **6** (precisely, $\mathbf{2} < \mathbf{4} < \mathbf{6}$). We are, of course, simplifying the problem in that we specify that only correcting block-moves are acceptable in a sorting sequence. The element **4** is called an obstacle in the above situation; that is, given a pair of compatible arcs $(a,b)-(a,c)$ or $(b,a)-(c,a)$ in a permutation graph $\mathcal{G}_\pi^p$, the element $\mathbf{x}$ is an *obstacle* if $\mathbf{b} < \mathbf{x} < \mathbf{c}$ and $\mathbf{x}$ occurs between $\mathbf{b}$ and $\mathbf{c}$ in π. A pair of arcs is *feasible* if they are compatible and no obstacles exists between them. In this case, we say the arc-pair is a *feasible hop*.

A refined strategy is to construct yet another graph $\mathcal{G}_\pi^a$, which we call the *arc graph* of π.

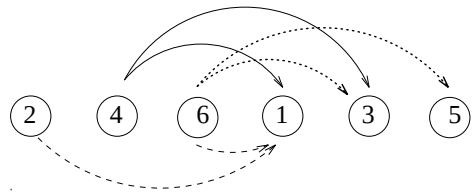

Figure 6.3: $(2,1)-(6,1)$ cannot be realized because of element **4**.

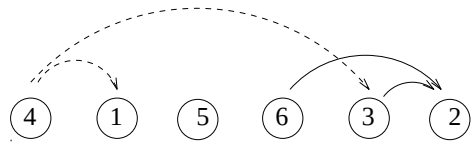

Figure 6.4: $(4,1)-(4,3)$ is disabled when $(6,2)-(3,2)$ is applied.

$\mathcal{G}_\pi^a = (\mathcal{V}_\pi^a, \mathcal{E}_\pi^a)$ is an undirected graph whose vertices are the arcs of $\mathcal{G}_\pi^p$ and whose edges connect two vertices whenever the pair of arcs is feasible.

The strategy starts with obtaining a maximum matching M in $\mathcal{G}_\pi^a$ (recall that the intention is to maximize the hops, i.e., arc-pairs). The matching M is a set of edges from $\mathcal{G}_\pi^a$, equivalently, a set of pairs of feasible arcs from $\mathcal{G}_\pi^p$. Let U be the set of unmatched vertices in $\mathcal{G}_\pi^a$; U, in turn corresponds to a set of arcs from $\mathcal{G}_\pi^p$. From M and U, we derive the corresponding set D_π of subgraphs in $\mathcal{G}_\pi^p$. The arc-pairs in D_π correspond to hops, single arcs to skips. The intention is to construct an optimal sorting sorting sequence from these hops and skips. Unfortunately, the feasibility of arc-pairs is not "static"; that is, when a sorting sequence is derived from D_π and then applied to π, it is possible to *disable* an arc-pair, that is, make infeasible a feasible hop. As an example, consider the permutation in Figure 6.4. When the hop for arc-pair $(6,2)-(3,2)$ is applied, element **2** is placed between **1** and **3**, thereby becoming an obstacle for the arc-pair $(4,1)-(4,3)$.

The example just given is discouraging since a potentially polynomial-time strategy involving maximum matching does not work in all cases. However, we can identify particular permutations where this graph-theoretic strategy applies. In addition, this strategy does give us a lower bound better than that given in Lemma 6.2 for the length of an optimal sorting sequence for a permutation.

Lemma 6.4 *Let π be a permutation, and let M be a maximum matching in the arc graph $\mathcal{G}_\pi^a$ of π. Furthermore, let U denote the set of unmatched vertices in $\mathcal{G}_\pi^a$. The short block-move distance of π*

satisfies the inequality:

$$MinSort_{Bk^3}(\pi) \geq |M| + |U|.$$

In the next sections, we identify particular types of permutations where the graph models we have devised are sufficient for optimally sorting a permutation with short block-moves.

6.4 Decomposing a Permutation into Subsequences

We can decompose a permutation into two or more element-disjoint subsequences. More precisely, an *m-decomposition* of π is a set $\{\chi_\pi^1, \chi_\pi^2, \ldots, \chi_\pi^m\}$ of subsequences of π obtained as follows. Using a set of m labels $\{a_1, a_2, \ldots, a_m\}$, assign a label to each element of π such that each label is assigned to at least one element (it is helpful to think of these labels as colors). We then define the component χ_π^k, where $1 \leq k \leq m$, as the subsequence consisting of elements assigned the label a_k. For example, let

$$\pi = \mathbf{8\ 2\ 4\ 5\ 1\ 3\ 6\ 9\ 7}.$$

We obtain a 3-decomposition of π by assigning a_1 to **1** and **2**, a_2 to **4**, **7** and **8**, and, a_3 to **3**, **5**, **6**, and **9**. We arrive at the following 3-decomposition:

$$\{\mathbf{2\ 1},\ \ \mathbf{8\ 4\ 7},\ \ \mathbf{5\ 3\ 6\ 9}\}.$$

A *woven bitonic permutation* π is a permutation that has a 2-decomposition in which one sequence is increasing and the other sequence is decreasing. That is, there exists a 2-decomposition $\{inc_\pi, dec_\pi\}$ for π where

$$inc_\pi = \pi(i_1)\ \pi(i_2) \ldots \pi(i_r)$$

and

$$dec_\pi = \pi(j_1)\ \pi(j_2) \ldots \pi(j_s),$$

satisfying

$$\pi(i_1) < \pi(i_2) < \ldots < \pi(i_r)$$

and

$$\pi(j_1) > \pi(j_2) > \ldots > \pi(j_s).$$

We call inc_π and dec_π the *increasing component* and the *decreasing component* of π, respectively. For example, the permutation

$$\pi = \mathbf{1\ 3\ 9\ 4\ 8\ 6\ 5\ 7\ 2}$$

is shown to be a woven bitonic permutation by this 2-decomposition

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{3}}\ \mathbf{9}\ \boxed{\mathbf{4}}\ \mathbf{8\ 6\ 5}\ \boxed{\mathbf{7}}\ \mathbf{2}.$$

The elements in boxes comprise the increasing component while the rest of the elements comprise the decreasing component; in particular, $inc_\pi = \mathbf{1\ 3\ 4\ 7}$ and $dec_\pi = \mathbf{9\ 8\ 6\ 5\ 2}$. Woven bitonic permutations generalize the more common notion of a *bitonic* permutation, which is a permutation consisting of two subsequences of contiguous elements, an increasing one and a decreasing one. In a woven bitonic permutation, the elements in the subsequences need not be contiguous but are more generally interwoven.

A *woven double-strip permutation* π is a permutation that has a 2-decomposition whose components are both increasing. That is, there exists a 2-decomposition $\{inc_\pi^1, inc_\pi^2\}$ for π where

$$inc_\pi^1 = \pi(i_1)\ \pi(i_2) \ldots \pi(i_r)$$

and

$$inc_\pi^2 = \pi(j_1)\ \pi(j_2) \ldots \pi(j_s)$$

satisfying

$$\pi(i_1) < \pi(i_2) < \ldots < \pi(i_r)$$

and

$$\pi(j_1) < \pi(j_2) < \ldots < \pi(j_s).$$

For example, the permutation

$$\pi = \mathbf{2}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{4}}\ \mathbf{3}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \mathbf{6\ 9}\ \boxed{\mathbf{8}}$$

is a woven double-strip permutation where $inc_\pi^1 = \mathbf{1\ 4\ 5\ 7\ 8}$ and $inc_\pi^2 = \mathbf{2\ 3\ 6\ 9}$.

In indicating a particular 2-decomposition for woven bitonic or a woven double-strip permutation, it is sufficient to specify one component as the other component is readily derived.

The identity permutation $\boldsymbol{\iota}$ is woven bitonic where any single element comprises dec_π and the rest is an increasing subsequence and therefore comprises inc_π. The decreasing permutation $\boldsymbol{\delta}$ is

CheckWovenBitonic(π). Validating a woven bitonic permutation.

INPUT: A permutation π.
OUTPUT: Whether or not π is a woven bitonic permutation.

1 $L \leftarrow$ a longest increasing subsequence of π
2 $D \leftarrow$ the subsequence of elements in π not in L
3 **if** $|D| = 0$ **or** D is a decreasing subsequence
4 **then return true**
5 **else return false**

Figure 6.5: Algorithm CheckWovenBitonic.

also woven bitonic—any single element can constitute inc_π and the rest of the elements constitute dec_π. The permutation $\boldsymbol{\iota}$ is woven double-strip since any 2-decomposition results in two increasing subsequences. The following theorems show that it can be determined in polynomial time whether a given permutation is woven bitonic or woven double-strip.

Theorem 6.5 *Determining whether a permutation π is a woven bitonic permutation is solvable in $O(n \log\log n)$ time.*

Proof: Algorithm CheckWovenBitonic in Figure 6.5 determines whether or not a given permutation π is woven bitonic. The algorithm first computes the longest increasing subsequence (LIS) of π and stores this subsequence in L (line 1). It then constructs the subsequence D of elements in π that are not in L (line 2). Finally, it returns **true** if D is either empty or a decreasing subsequence (lines 3–4), **false** otherwise (line 5).

It is easy to show that Algorithm CheckWovenBitonic returns **true** *only if* the permutation is woven bitonic: L and D comprise the decomposition of π into inc_π and dec_π (the case $|D| = 0$ occurs when $\pi = \boldsymbol{\iota}$).

It remains to show that the algorithm returns **true** *if* the input permutation is woven bitonic. Let π be a woven bitonic permutation π with decomposition inc_π and dec_π and let L be the LIS of π computed by the algorithm. Clearly, all elements in inc_π are in L and at most one element in dec_π may be in L. If none of the elements in dec_π are in L, then $D = dec_\pi$ and the algorithm returns **true**. If exactly one element **a** in dec_π is in L, then D is just dec_π without **a**. The sequence

CheckWovenDoubleStrip(π)**.** Validating a woven double-strip permutation.

INPUT: A permutation π.
OUTPUT: Whether or not π is woven double-strip permutation.

```
1   let n be the length of π
2   s_1 ← π(1)
3   i ← 2
4   k ← 1
5   ℓ ← 0
6   while i ≤ n
7         do if π(i) > s_k
8               then k ← k + 1
9                    s_k ← π(i)
10              else ℓ ← ℓ + 1
11                   t_ℓ ← π(i)
12            i ← i + 1
13  if ℓ = 0 or t_1 t_2 ... t_ℓ is an increasing sequence
14    then return true
15    else return false
```

Figure 6.6: Algorithm CheckWovenDoubleStrip.

D is therefore either decreasing or empty (if $dec_\pi = \mathbf{a}$) and the algorithm returns **true**.

Lines 2 and 3 of the algorithm require straightforward linear scans of π and D, respectively. The algorithm is therefore dominated by the determination of the LIS (line 1) so that it runs in $O(n \log\log n)$ time [24]. □

Theorem 6.6 *Determining whether a permutation π is a woven double-strip permutation is solvable in $O(n)$ time.*

Proof: Algorithm CheckWovenDoubleStrip in Figure 6.6 determines whether or not a given permutation π is woven double-strip. The algorithm builds a 2-decomposition of π into subsequences $S = s_1\ s_2\ \ldots\ s_k$ and $T = t_1\ t_2\ \ldots\ t_\ell$. The element s_1 is assigned the first element in π (line 2). Then, a **while** loop (lines 6–12) builds the rest of the subsequence S by performing a linear scan of π that repeatedly selects the next element greater than the previously selected element (lines 7–9). This guarantees that the resulting subsequence S is increasing. The elements in π not included in

S are made elements of T (lines 10–11) during the scan. If the resulting sequence T is empty or an increasing sequence (line 13), the algorithm returns **true** (line 14); it returns **false** otherwise (line 15).

It is easy to show that Algorithm CheckWovenDoubleStrip returns **true** *only if* the permutation is woven double-strip: S and T comprise the decomposition of π into inc^1_π and inc^2_π (the case $|T| = \ell = 0$ occurs when $\pi = \boldsymbol{\iota}$).

To show that the algorithm returns **true** *if* the input permutation is woven double-strip, let π be a woven double-strip permutation π with 2-decomposition inc^1_π and $inc^2\pi$ and let S and T be the 2-decomposition of π produced by the algorithm. Consider the sequence $T = t_1\ t_2\ \ldots t_\ell$. An element t_p, where $1 < p \leq \ell$, is called a *shifter* if either of the following holds:

- $t_p = t_1$
- t_p is in inc^1_π and t_{p-1} is in inc^2_π, or,
- if t_p is in inc^2_π and t_{p-1} is in inc^1_π.

Observe that shifters allow us to view T as a series of *runs*, subsequences of contiguous elements that begin with a shifter. The runs in T are alternately contained in inc^1_π and inc^2_π so that each run is actually an increasing subsequence. Now, if all shifters t_p in T except the first shifter t_0 satisfy the condition $t_{p-1} < t_p$, then clearly, T is either empty or an increasing sequence and the algorithm returns **true**. It suffices to show that such is indeed the case for all shifters $\neq t_1$.

Let t_p be a shifter other than t_1. Without loss of generality, suppose that t_p is in inc^1_π and t_{p-1} is in inc^2_π. The element t_{p-1} is an element in π not selected by the algorithm to be an element in S. This can only occur if the previously selected element, say s_q, is from inc^1_π. Now, $t_{p-1} < s_q$, since t_{p-1} was not selected in the loop. Also, $s_q < t_p$, since both of these are elements of inc^1_π and t_p occurs later (to the right of s_q) in π. By transitivity, $t_{p-1} < t_p$, as desired. The correctness of Algorithm CheckWovenDoubleStrip follows.

The algorithm runs in $O(n)$ time since the **while** loop performs a simple scan of π. □

In the next two sections we solve $MinSort_{Bk^3}$ for both woven bitonic and woven double-strip permutations.

6.5 Woven Bitonic Permutations

In this section, we devise a polynomial-time algorithm for optimally sorting woven bitonic permutations by short block-moves. We begin with an example. Let $\pi = \mathbf{7\ 1\ 4\ 5\ 6\ 8\ 3\ 2}$ whose components are exhibited by

$$\mathbf{7}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{6}}\ \boxed{\mathbf{8}}\ \mathbf{3\ 2}.$$

The strategy of the upcoming algorithm is to insert elements from the decreasing component into the increasing component. For instance, observe that the element **7** is right 4-pending so that Lemma 6.3 applies and **7** can be moved to the right using 2 correcting hops. After the two hops, element **7** can be included in the increasing component as illustrated by

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{6}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \mathbf{3\ 2}.$$

Now, element **2** is left 6-pending and therefore takes 3 correcting hops to be inserted in the increasing component resulting in

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{6}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \mathbf{3}.$$

Now we are left with element **3**, a left 5-pending element. Since it is 4-pending as well, 2 correcting hops and then one skip successfully inserts the element, resulting in

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{3}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{6}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}},$$

the identity permutation. This is optimal since all but one block-move in the sort is a correcting hop (see Lemma 6.2). The above sort is performed using an "insert" metric, which we define next.

Let π be a woven bitonic permutation and fix inc_π and dec_π so that

$$inc_\pi = \pi(i_1)\ \pi(i_2) \ldots \pi(i_r)$$

and

$$dec_\pi = \pi(j_1)\ \pi(j_2) \ldots \pi(j_s).$$

Let $\mathbf{a} = \pi(p)$ be an element in dec_π. There exists a unique integer x, where $0 \leq x \leq r$, such that

- $i_\ell < p,$ for all $\ell, 0 < \ell \leq x$, and,
- $i_\ell > p,$ for all $\ell, x < \ell \leq r$;

that is,

$$i_1 < i_2 \cdots < i_x < p < i_{x+1} < \cdots < i_r.$$

we call i_x the *left neighbor index* of a; for notational convenience, we let $i_0 = 0$ and $\pi(i_0) = 0$ in case $x = 0$. For example, let

$$\pi = \boxed{\mathbf{2}}\ \mathbf{7}\ \boxed{\mathbf{4}}\ \mathbf{5}\ \mathbf{3}\ \boxed{\mathbf{6}}\ \mathbf{1}\ \boxed{\mathbf{8}}.$$

For element $\pi(2) = \mathbf{7}$, its left neighbor index $i_x = 1$. The other elements in the decreasing sequence $\pi(4)$, $\pi(5)$, and $\pi(7)$ have left neighbor indices 3, 3, and 6, respectively. The intention is to determine the *insert position* i' in π where element $\mathbf{a}$ can be inserted in inc_π while retaining the increasing property of the sequence. There are three cases for insert position.

Case 1. $\pi(i_x) < \pi(p) < \pi(i_{x+1})$: Here, $\mathbf{a}$ is already properly inserted in inc_π, so $i' = p$. In the above example, the element $\pi(4) = \mathbf{5}$ has $i' = p = 4$.

Case 2. $\pi(i_{x+1}) < \pi(p)$: Here, the insert position of $\mathbf{a}$ is to its right so that i' is the position i_y where y is the largest integer $\leq r$ such that $\pi(i_y) < \pi(p)$. In the example, element $\pi(2) = \mathbf{7}$ has $i' = 6$, the position of element $\mathbf{6}$.

Case 3. $\pi(p) < \pi(i_x)$: Here, the insert position of $\mathbf{a}$ is to its left so that i' is the position i_y where y is the smallest integer > 0 such that $\pi(p) < \pi(i_y)$. In the example, element $\pi(5) = \mathbf{3}$ has $i' = 3$, the position of element $\mathbf{4}$. The element $\pi(7) = \mathbf{1}$, on the other hand, has $i' = 1$, the position of element $\mathbf{2}$.

Define the *insert distance* $insertdist(\pi, dec_\pi, \mathbf{a})$ of an element $\mathbf{a} = \pi(p)$ in dec_π to be the distance of $\mathbf{a}$ from its insert position; that is,

$$insertdist(\pi, dec_\pi, \mathbf{a}) = i' - p.$$

The sign of $insertdist(\pi, dec_\pi, \mathbf{a})$ indicates whether the insert position is to the right or to the left of the element. The insert distances of the elements in the previous example are given in the following table.

Element	Insert Distance
7	4
5	0
3	-2
1	-6

An important observation is that when an element is inserted in a woven bitonic permutation, the resulting permutation remains woven bitonic, as stated in the following lemma.

Lemma 6.7 *Let π be a woven bitonic permutation and let inc_π and dec_π be the increasing and decreasing components of π. Let $\mathbf{a} = \pi(p)$ be an element in dec_π whose insert position is i' and let $d = insertdist(\pi, dec_\pi, \mathbf{a})$. If $d > 0$, then the permutation*

$$\pi' = \pi \cdot B\langle \mathbf{a}, \pi(p+1)\ \pi(p+2) \ldots \pi(i')\rangle$$

is also a woven bitonic permutation. If $d < 0$, then the permutation

$$\pi' = \pi \cdot B\langle \pi(i')\ \pi(i'+2) \ldots \pi(p-1), \mathbf{a}\rangle$$

is also a woven bitonic permutation.

Proof: For either case, as long as dec_π contains at least two elements, the permutation π' is a woven bitonic permutation where $inc_{\pi'}$ is just inc_π with element $\mathbf{a}$ in its correct place and $dec_{\pi'}$ is just dec_π without the element $\mathbf{a}$. If dec_π contains only one element, observe that $\pi' = \boldsymbol{\iota}$ and is therefore woven bitonic. □

The following lemma determines pending elements from insert distances.

Lemma 6.8 *Let $\mathbf{a}$ be an element in the decreasing component of a woven bitonic permutation π and let $d = insertdist(\pi, dec_\pi, \mathbf{a})$. If $d > 0$, then $\mathbf{a}$ is right d-pending. If $d < 0$, then $\mathbf{a}$ is left $|d|$-pending.*

Proof: First, suppose $d > 0$. We need to show that all d elements to the immediate right of element $\mathbf{a} = \pi(p)$ are less than $\mathbf{a}$. Let $\mathbf{b} = \pi(q)$, where $p + 1 \leq q \leq p + d$, be one such element. The element $\mathbf{b}$ can either be in dec_π or inc_π. If $\mathbf{b}$ is in dec_π, then $\mathbf{b} < \mathbf{a}$, since dec_π is decreasing and $\mathbf{b}$ occurs to the right of $\mathbf{a}$. If $\mathbf{b}$ is in inc_π, then $\mathbf{b} = \pi(q) < \mathbf{a}$, by transitivity as $\pi(q) < \pi(i')$ since inc_π is increasing and $\pi(i') < \mathbf{a}$ by definition. It follows that $\mathbf{a}$ is right d-pending.

Now, suppose $d < 0$. We need to show that all $|d|$ elements to the immediate left of element $\mathbf{a} = \pi(p)$ are greater than $\mathbf{a}$. Let $\mathbf{b} = \pi(q)$, where $p + d \leq q \leq p - 1$, be one such element. The element $\mathbf{b}$ can either be in dec_π or inc_π. If $\mathbf{b}$ is in dec_π, then $\mathbf{b} > \mathbf{a}$, since dec_π is decreasing and $\mathbf{b}$ occurs to the left of $\mathbf{a}$. If $\mathbf{b}$ is in inc_π, then $\mathbf{b} = \pi(q) > \mathbf{a}$, by transitivity as $\pi(i') < \pi(q)$ since inc_π is increasing and $\mathbf{a} < \pi(i')$ by definition. It follows that $\mathbf{a}$ is left $|d|$-pending. The lemma follows. □

InsertEven(π, a, d)**.** Inserting a pending element in a permutation.

INPUT: A permutation π, an element a in π, and an even integer d such that $d = 0$ or a is $|d|$-pending in π.
OUTPUT: The resulting permutation after a has been inserted in π and the actual number of correcting hops applied.

```
1   if d > 0
2     then π ← RightHops(π, a, d/2)
3     else  π ← LeftHops(π, a, |d|/2)
4   return (π, |d|/2)
```

Figure 6.7: Algorithm InsertEven.

The above lemma implies that as long as the permutation being sorted contains an element with even insert distance, then sorting is straightforward since we can repeatedly insert each element using corresponding hops by Lemma 6.3. Algorithm InsertEven, shown in Figure 6.7, performs the hops for such an element. The algorithm takes as input, a permutation π, an element a, and an even integer d, such that either $d = 0$ or a is $|d|$-pending in π. If $d \neq 0$, then the element a is either right or left pending, depending on the sign of d. Presumably, d is the insert distance of a, computed by the algorithm (discussed later) that calls InsertEven. Algorithm InsertEven applies the corresponding hops to π and returns the resulting permutation and $|d|/2$, the actual number of hops performed. The algorithm calls one of two functions, described as follows:

RightHops$(\pi, \mathbf{a}, k)$**.** This function applies k hops to π, specifically, hops that move element $\mathbf{a}$ to its right in π. If $\mathbf{a}$ is right $2k$-pending, then Lemma 6.3 applies and the actual correcting hops are specified in the lemma. This function returns the permutation that results after the hops have been applied to π.

LeftHops$(\pi, \mathbf{a}, k)$**.** This function applies k hops to π, specifically, hops that move element $\mathbf{a}$ to its left in π. If $\mathbf{a}$ is left $2k$-pending, then Lemma 6.3 applies and the actual correcting hops are specified in the lemma. This function returns the permutation that results after the hops have been applied to π.

We cannot guarantee that an element with positive, even insert distance exists in the permutation

InsertOdd(π, a, d)**.** Inserting a pending element in a permutation.

INPUT: A permutation π, an element a in π, and an odd integer d such that a is $|d|$-pending in π.
OUTPUT: The resulting permutation after a has been inserted in π and the actual number of correcting block-moves applied.

```
1   if d > 0
2     then π ← RightHops(π, a, ⌊d/2⌋)
3          π ← RightSkip(π, a)
4     else π ← LeftHops(π, a, ⌊|d|/2⌋)
5          π ← LeftSkip(π, a)
6   return (π, ⌊|d|/2⌋ + 1).
```

Figure 6.8: Algorithm InsertOdd.

as it is sorted. In particular, the insert distance measure changes dynamically as block-moves are applied to the permutation. It may be possible that only odd distances remain and perhaps a skip is necessary as in the last block-move in the example given earlier in this section.

Algorithm InsertOdd, presented in Figure 6.8 spends one skip to insert a $|d|$-pending element, where d is odd. If $|d| = 1$, then a single skip suffices. If $|d| \geq 3$, then the algorithm uses the fact that a $|d|$-pending element where d is odd is $(|d| - 1)$-pending where $|d| - 1$ is even. Lemma 6.3 then applies and performing $(|d| - 1)/2$ hops plus an extra skip inserts the pending element a. In addition to the subroutines described earlier, the algorithm calls one of the functions RightSkip or LeftSkip described as follows.

RightSkip$(\pi, \mathbf{a})$**.** This function applies a skip to π, specifically, the skip $\beta(p, p+1, p+2)$ assuming $\mathbf{a} = \pi(p)$. This function returns $\pi \cdot \beta$, the permutation that results.

LeftSkip$(\pi, \mathbf{a})$**.** This function applies a skip to π, specifically, the skip $\beta(p-1, p, p+1)$ assuming $\mathbf{a} = \pi(p)$. This function returns $\pi \cdot \beta$, the permutation that results.

Algorithm InsertOdd returns the resulting permutation and the actual number of block-moves applied $(\lfloor |d|/2 \rfloor + 1)$.

An algorithm to sort woven bitonic permutations described later in this section repeatedly calls InsertEven whenever any elements with even insert distances remain in the decreasing component

and then calls InsertOdd only when it is necessary, perhaps if only one element with odd insert distance remains. It is still possible, however, that several elements remain, all of which have odd insert distances. When this situation occurs, we select pairs of these elements and find a way for the elements to "correct" each other's insert distance parity, as the following example illustrates.

Let

$$\pi = \mathbf{13\ 12\ 1\ 2\ 5\ 7\ 8\ 9\ 6\ 4\ 10\ 3\ 11}$$

whose components are exhibited by

$$\mathbf{13\ 12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6\ 4}\ \boxed{\mathbf{10}}\ \mathbf{3}\ \boxed{\mathbf{11}}.$$

The insert distances of the elements in the decreasing sequence are shown below:

Element	Insert Distance
13	12
12	11
6	-3
4	-5
3	-7

Element **13** is the only element with even insert distance so it is moved to the extreme right resulting in

$$\mathbf{12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6\ 4}\ \boxed{\mathbf{10}}\ \mathbf{3}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}}.$$

The insert distances of the other elements remain unchanged after **13** has joined the increasing subsequence. We can insert elements **4** and **3** as follows. First, apply the following correcting hops:

$$B\langle \mathbf{4\ 10}, \mathbf{3}\rangle, B\langle \mathbf{6}, \mathbf{3\ 4}\rangle.$$

The first hop causes elements **3** and **4** to be adjacent. The second hop shifts the parity of the insert distance of both elements. The resulting permutation is

$$\mathbf{12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{3\ 4\ 6}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}}.$$

Note that the insert distance of **3** is now even, as is the insert distance of **4**, after **3** moves out of the way. We thus apply the following hops

$$B\langle \mathbf{8\ 9}, \mathbf{3}\rangle, B\langle \mathbf{5\ 7}, \mathbf{3}\rangle, B\langle \mathbf{8\ 9}, \mathbf{4}\rangle, B\langle \mathbf{5\ 7}, \mathbf{4}\rangle.$$

as direct applications of Lemma 6.3 resulting in the permutation

$$\mathbf{12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{3}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}},$$

where the insert distances for elements **12** and **6** remain as in the table above. These two elements are inserted by first applying the following correcting hops:

$$B\langle \mathbf{12}, \mathbf{1\ 2}\rangle, B\langle \mathbf{12}, \mathbf{3\ 4}\rangle, B\langle \mathbf{12}, \mathbf{5\ 7}\rangle.$$

These three hops move element **12** to the right. The third block-move, in particular, causes the insert distance of element **6** to shift from 3 to 4. At this point, the permutation is

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{3}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \mathbf{12}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{6}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}},$$

where **12** is still right 5-pending. Applying $B\langle \mathbf{8\ 9}, \mathbf{6}\rangle$ and $B\langle \mathbf{7\ 12}, \mathbf{6}\rangle$ to this permutation inserts element **6** resulting in the following:

$$\boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{3}}\ \boxed{\mathbf{4}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{6}}\ \boxed{\mathbf{7}}\ \mathbf{12}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \boxed{\mathbf{13}}.$$

Observe that the moves causes **12** to be right 4-pending so that the block-moves $B\langle \mathbf{12}, \mathbf{8\ 9}\rangle$ and $B\langle \mathbf{12}, \mathbf{10\ 11}\rangle$ complete the sort. To summarize, whenever elements with odd insert distances remain, a pair of elements is selected and the pair is inserted so that the parities of their insert distances are made even. This is always possible because any pair of elements from the decreasing sequence dec_π have to cross each other during the sort as the elements are not yet in their correct relative order. We shall see later that it is preferable that the insert distances of the pair have the same sign as in the first pair in the above sorting example (elements **3** and **4**).

We now describe two additional algorithms, InsertSameSign and InsertDiffSign that handle pairs of elements with odd insert distance. Algorithm InsertSameSign is shown in Figure 6.9 and takes as input the permutation π, two elements a and b, and two odd integers d_a and d_b with identical signs. The integers are presumed to be the insert distances of elements a and b so that these elements are $|d_a|$-pending and $|d_b|$-pending, respectively (Lemma 6.8). The goal is to insert these elements in inc_π using only correcting hops. The algorithm returns the resulting permutation and the number of hops performed. First, the algorithm establishes the positions of elements a and b (line 1) and then stores the distance between these two elements in the variable *gap* (line 2). Then, the algorithm distinguishes between two cases: whether the distances are positive (lines 4–12) or negative (lines 13–22). We discuss the positive case and the negative case follows by symmetry. The first step is to move element a to the right using correcting hops until it is next to element b. Depending on whether *gap* is odd or even, a will be immediately to the right (line 5) or immediately

InsertSameSign(π, a, d_a, b, d_b)**.** Used in sorting woven bitonic permutations.

INPUT: A permutation π, elements a and b in π, and odd integers d_a and d_b such that a is $|d_a|$-pending, b is $|d_b|$-pending, and $d_a d_b > 0$.

OUTPUT: The resulting permutation after elements a and b have been inserted in π and the actual number of correcting hops applied.

```
1   let π(i) = a, π(j) = b   ▷ i < j
2   gap ← j − i   ▷ distance between elements a and b
3   if d_a > 0   ▷ direction is to the right
4     then if gap is odd
5            then π ← RightHops(π, a, (gap − 1)/2)
6                 π ← LeftHops(π, π(j + 1), 1)
7                 π ← RightHops(π, a, (d_a − 1)/2 − gap/2)
8                 π ← RightHops(π, b, (d_b − 1)/2)
9            else π ← RightHops(π, a, gap/2 − 1)
10                π ← LeftHops(π, π(j + 1), 1)
11                π ← RightHops(π, a, (d_a − 1)/2 − (gap/2 − 1))
12                π ← RightHops(π, b, (d_b − 1)/2)
13    else   ▷ direction is to the left
14         if gap is odd
15           then π ← LeftHops(π, b, (gap − 1)/2)
16                π ← RightHops(π, π(i − 1), 1)
17                π ← LeftHops(π, b, (|d_a| − 1)/2 − gap/2)
18                π ← LeftHops(π, a, (|d_b| − 1)/2)
19           else π ← LeftHops(π, b, gap/2 − 1)
20                π ← RightHops(π, π(i − 1), 1)
21                π ← LeftHops(π, b, (|d_a| − 1)/2 − (gap/2 − 1))
22                π ← LeftHops(π, a, (|d_b| − 1)/2)
23  return (π, (|d_a| + |d_b|)/2)
```

Figure 6.9: Algorithm InsertSameSign.

to the left (line 9) of b. Next, a correcting left hop is performed that shifts the remaining insert distance of the two elements (line 6 or 10). Finally, both elements are individually moved to the right using correcting hops until their insert positions are reached (lines 7–8 or lines 11–12). It is important to move a first particularly if *gap* is odd since a is already to the right of b and $a > b$.

Algorithm InsertDiffSign is shown in Figure 6.10 takes as input the permutation π, two elements a and b, and two odd integers d_a and d_b with different signs, presumed to be the insert distances of elements a and b. Assuming a occurs to the left of b, a and b are d_a-pending and $|d_b|$-pending, respectively. As in the previous algorithm, it begins by establishing the positions of elements a and b (line 1). It then stores in the variable *span* the distance between these two elements (line 2). Next, the algorithm compares d_a and $|d_b|$ with *span*. Four cases result:

Case 1. $d_a < span$ **and** $|d_b| < span$ (lines 4–8): Here, the element a is moved to the right using correcting hops and one correcting skip. The insertion of a causes the insert distance of b to be increased by 1. Lemma 6.3 and Lemma 6.8 then apply and b is inserted using correcting left hops.

Case 2. $d_a < span$ **and** $|d_b| \geq span$ (lines 9–12): The element b is moved to the left until it is 1 or 2 (depending on the parity of *span*) positions to the right of a. This shifts the parity of the insert distance of a so Lemma 6.3 and Lemma 6.8 apply and right hops cause the insertion of element a. Inserting a, in turn, shifts the parity of the insert distance of b and element b is inserted using right hops.

Case 3. $d_a \geq span$ **and** $|d_b| < span$ (lines 13–16): Symmetric to Case 2.

Case 4. $d_a \geq span$ **and** $|d_b| \geq span$ (lines 17–20): Element a is inserted using right hops and a single skip, causing a shift in the parity of the insert distance of b. Then, element b is inserted using left hops.

We are now ready to describe our algorithm for sorting woven bitonic permutations. Algorithm BitonicSort, shown in Figure 6.11, takes as input a woven bitonic permutation π. The algorithm returns the short block-move distance of π and uses at most one skip in sorting π. It first establishes the decreasing component of π and sets a variable D to be this component (line 1). The variable D is a sequence of elements that represents the decreasing component of π even as π is updated. The **while** loop in lines 3–28 dynamically updates π and D by repeatedly inserting elements from D to

InsertDiffSign(π, a, d_a, b, d_b)**.** Used in sorting woven bitonic permutations.

INPUT: A permutation π, elements a and b in π, and odd integers d_a and d_b such that a is $|d_a|$-pending, b is $|d_b|$-pending, and $d_a d_b < 0$.

OUTPUT: The resulting permutation after elements a and b have been inserted in π and the actual number of correcting hops applied.

```
let π(i) = a, π(j) = b   ▷ i < j
span ← j − i   ▷ distance between elements a and b
dist ← (d_a + |d_b|)/2)   ▷ value may be incremented; see first case below
switch case d_a < span and |d_b| < span
            do π ← RightHops(π, a, (d_a − 1)/2)
               π ← RightSkip(π, a)
               dist ← dist + 1
               π ← LeftHops(π, b, (|d_b| + 1)/2)
       case d_a < span and |d_b| ≥ span
            do π ← LeftHops(π, b, ⌈span/2⌉ − 1)
               π ← RightHops(π, a, (d_a + 1)/2)
               π ← LeftHops(π, b, (|d_b| − 1)/2 − ⌈span/2⌉ + 1)
       case d_a ≥ span and |d_b| < span
            do π ← RightHops(π, a, ⌈span/2⌉ − 1)
               π ← LeftHops(π, b, (|d_b| + 1)/2)
               π ← RightHops(π, a, (d_a − 1)/2 − ⌈span/2⌉ + 1)
       case d_a ≥ span and |d_b| ≥ span
            do π ← RightHops(π, a, (d_a − 1)/2)
               π ← RightSkip(π, a)   ▷ no need to increment dist
               π ← LeftHops(π, b, (|d_b| − 1)/2)
return (π, dist)
```

Figure 6.10: Algorithm InsertDiffSign.

BitonicSort(π). Minimum sorting by short block-moves for woven bitonic permutations.

INPUT: A woven bitonic permutation π.
OUTPUT: The size of the shortest sorting sequence of block-moves.

```
D ← dec_π   ▷ choose an appropriate decomposition for π
dist ← 0
while |D| > 0
        do switch case EvenExists(π, D)
                        do a ← EvenElement(π, D)
                           d_a ← InsertDistance(π, D, a)
                           (π, d) ← InsertEven(π, a, d_a)
                           D ← RemoveElement(D, a)
                   case |D| = 1
                        do a ← OneElement(D)
                           d_a ← InsertDistance(π, D, a)
                           (π, d) ← InsertOdd(π, a, d_a)
                           D ← RemoveElement(D, a)
                   case |D| = 2 and DiffSigns(π, D)
                        do (a, b) ← TwoElements(D)
                           d_a ← InsertDistance(π, D, a)
                           d_b ← InsertDistance(π, D, b)
                           (π, d) ← InsertDiffSign(π, a, d_a, b, d_b)
                           D ← RemoveElement(D, a)
                           D ← RemoveElement(D, b)
                   otherwise   ▷ default
                        do (a, b) ← SameSignElements(π, D)
                           d_a ← InsertDistance(π, D, a)
                           d_b ← InsertDistance(π, D, b)
                           (π, d) ← InsertSameSign(π, a, d_a, b, d_b)
                           D ← RemoveElement(D, a)
                           D ← RemoveElement(D, b)
            dist ← dist + d
return dist
```

Figure 6.11: Algorithm BitonicSort.

the increasing component of π. The loop terminates when $|D| = 0$, equivalently, when π is sorted. In the loop, if an element with even insert distance exists in D (lines 4–8), then Algorithm InsertEven is called that inserts that element using correcting hops. If only elements with odd insert distances remain, then, when possible (lines 21–27), two elements whose insert distances have the same sign are selected from D and then InsertSameSign is called. There are cases when such elements do not exist in D such as when D has only one element. In this case (lines 9–13), that element is selected and Algorithm InsertOdd is called. Another case is when D contains exactly two elements whose insert distances differ in sign (lines 14–20). This is handled by a call to Algorithm InsertDiffSign. We complete the discussion of Algorithm BitonicSort with a description of the other functions used by this algorithm, particularly those that deal with the sequence D. Assume that π is a woven bitonic permutation and that D is its decreasing component.

InsertDistance(π, D, a)**.** Given π, D, and an element a from D, this function returns the insert distance of a in π. Computing the insert distance for a single element takes $O(n)$ time since this requires a linear scan of inc_π.

EvenExists(π, D)**.** Given π and D, this function returns **true** if an element from D with even insert distance exists in π; it returns **false** otherwise.

DiffSigns(π, D)**.** Given π and D, this function returns **true** if the insert distances of the two elements that D contains differ in sign; it returns **false** otherwise.

EvenElement(π, D)**.** Given π and D, this function returns an element in D that has even insert distance.

OneElement(D)**.** This function returns the single remaining element in D.

TwoElements(D)**.** This function returns the two remaining elements in D.

SameSignElements(π, D)**.** Given π and D, this function returns two elements from D whose insert distances have the same sign. Observe that if $|D| > 2$, such elements always exist.

RemoveElement(D, a)**.** This function returns the sequence that results after removing the element a from D.

All of the above functions take $O(n)$ time (Algorithms OneElement and TwoElements take $O(1)$ time) and generally requires a linear scan of the elements in π or D.

Theorem 6.9 *Let π be a woven bitonic permutation. Algorithm* BitonicSort *returns $MinSort_{Bk^3}(\pi)$, the length of an optimal sorting sequence of short block-moves for π, in $O(n^2)$ time.*

Proof: By Lemma 6.2, it suffices to show that at most one skip is performed in the sort and that all block-moves are correcting block-moves. The only possibilities where a skip is performed are with Algorithm InsertOdd and Algorithm InsertDiffSign. Algorithm InsertOdd is called only when a single element remains. Algorithm InsertDiffSign, on the other hand, is called only when two elements remain. Clearly, such a call may occur only on the last iteration in the loop of Algorithm BitonicSort, since $|D| = 0$ after the call. This means that at most one of these algorithms is called and it follows that at most one skip is performed by the sort.

It remains to show that all the block-moves are correcting block-moves. Every iteration in the loop of Algorithm BitonicSort inserts an element from the decreasing component D of π to its corresponding increasing component, after which both π and D are updated. The woven bitonic property of the updated permutation is therefore retained by Lemma 6.7. As a result, all hops and skips in the algorithms are justified by Lemmas 6.3 and 6.8 and are therefore correcting hops and skips.

Since all except perhaps one of the block-moves are correcting hops, all arcs in the permutation graph $\mathcal{G}_\pi^p = (\mathcal{V}_\pi^p, \mathcal{A}_\pi^p)$ except perhaps one is paired with another as a correcting hop. Thus, the algorithm returns $\lceil |\mathcal{A}_\pi^p|/2 \rceil$, the length of an optimal solution (Lemma 6.2).

There are $O(n)$ elements that need to be inserted in the increasing component of π so the loop in Algorithm BitonicSort is executed $O(n)$ times. Each iteration involves a constant number of operations on D: the computation of insert distance for at most two elements, and the actual insertion of the elements. Recall that the operations on D as well as the computation of insert distance require $O(n)$ time. Also, inserting an element requires $O(n)$ (at most $n/2$) block-moves (calls to RightHops, LeftHops, RightSkip, and LeftSkip). It follows that the algorithm runs in $O(n^2)$ time. □

In hindsight, the short block-move distance $MinSort_{Bk^3}(\pi)$ for woven bitonic permutations can be computed by constructing $\mathcal{G}_\pi^p$ and then determining $\lceil |\mathcal{A}_\pi^p|/2 \rceil$, similarly an $O(n^2)$ process. However, $\mathcal{A}_\pi^p$ does not immediately exhibit the actual sorting sequence since an appropriate matching of arcs in $\mathcal{A}_\pi^p$ needs to be determined first. We show in the next section how this is not a straightforward task and that the structure of the permutation is essential in guaranteeing that a matching

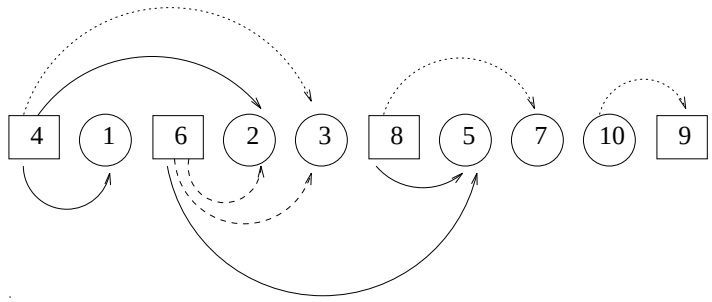

Figure 6.12: A maximum matching of feasible arc-pairs.

does yield the sequence. Algorithm BITONICSORT, on the other hand, can be easily revised so that it returns the actual sorting sequence, instead of just the short block-move distance.

6.6 Woven Double-Strip Permutations

Section 6.3 suggests the following strategy for solving $MinSort_{Bk^3}$. Obtain a maximum matching M in the arc graph $\mathcal{G}_\pi^a$ (feasible arc-pairs from $\mathcal{G}_\pi^p$), and then derive the set D_π of arc-pairs from M and single arcs from the set U of unmatched vertices in $\mathcal{G}_\pi^a$. The set D_π may represent a potential optimal sequence of short block-moves. For example, consider the woven double-strip permutation

$$\pi = \boxed{\mathbf{4}}\ \mathbf{1}\ \boxed{\mathbf{6}}\ \mathbf{2}\ \mathbf{3}\ \boxed{\mathbf{8}}\ \mathbf{5}\ \mathbf{7}\ \mathbf{10}\ \boxed{\mathbf{9}},$$

where $\mathcal{G}_\pi^p$ is shown in Figure 6.12. In the figure, the vertices are drawn using circles and squares to distinguish the two increasing components. The figure also illustrates a maximum matching of arcs. The unmatched arcs are drawn in dotted lines, while the arc-pairs are drawn in regular lines or dashed lines. The set D_π in this example is

$$\{(4,1)-(4,2),(4,3),(6,2)-(6,3),(6,5)-(8,5),(8,7),(10,9)\}.$$

The difficulty is that the sequence derived from D_π might not be realizable because of the possibility that a hop corresponding to an arc-pair may be disabled by an earlier block-move. In this section, we show that this can never be the case for woven double-strip permutations.

We begin by refining our notion of the set D_π. The set D_π may as well be described as a set of *potential* block-moves specified by value. That is, for each element in D_π:

- If the element is the single arc (a, b), specify this as the potential skip $B\langle \mathbf{a}, \mathbf{b} \rangle$;
- If the element is the pair of arcs $(a, b) - (a, c)$, specify this as the potential hop $B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c} \rangle$; and,
- If the element is the pair of arcs $(b, a) - (c, a)$, specify this as the potential hop $B\langle \mathbf{b}\ \mathbf{c}, \mathbf{a} \rangle$.

In the above example,

$$D_\pi = \{B\langle \mathbf{4}, \mathbf{1}\ \mathbf{2} \rangle, B\langle \mathbf{4}, \mathbf{3} \rangle, B\langle \mathbf{6}, \mathbf{2}\ \mathbf{3} \rangle, B\langle \mathbf{6}\ \mathbf{8}, \mathbf{5} \rangle, B\langle \mathbf{8}, \mathbf{7} \rangle, B\langle \mathbf{10}, \mathbf{9} \rangle\}.$$

Now the goal is to transform the set D_π, if possible, into a realizable sorting sequence, which we formalize next. Consider the permutation π and the potential block-move B. The block-move B is an *actual* block-move with respect to π:

- If B is the skip $B\langle \pi(i), \pi(j) \rangle$ and $j = i + 1$; or,
- If B is the hop $B\langle \pi(i), \pi(j)\ \pi(k) \rangle$ and $j = i + 1, k = i + 2$; or,
- If B is the hop $B\langle \pi(i)\ \pi(j), \pi(k) \rangle$ and $j = i + 1, k = i + 2$.

Observe that an actual block-move can be immediately applied to π; that is, the product $\pi \cdot B$ is valid. In our example, the potential block-move $B\langle \mathbf{6}, \mathbf{2}\ \mathbf{3} \rangle$ is an actual block-move with respect to π, while $B\langle \mathbf{4}, \mathbf{3} \rangle$ is not. The product $\pi \cdot B_1 \cdot B_2 \cdots B_k$ is a *realizable product* if each B_i is an actual block-move with respect to $\pi^{i-1} = \pi \cdot B_1 \cdot B_2 \cdots B_{i-1}$; the sequence $B_1, B_2, \ldots, B_k$ is then called a *realizable sequence*. The set D_π of potential block-moves is *realizable* if its elements can compose a realizable sequence.

We show in this section that for woven double-strip permutations, it can be guaranteed that the set D_π derived from M and U is realizable. Woven double-strip permutations have special characteristics, one of which is stated in the following lemma.

Lemma 6.10 *Let π be a woven double-strip permutation and let β be a correcting block-move. Then $\pi' = \pi \cdot \beta$ is a woven double-strip permutation.*

Proof: Since π is a woven double-strip permutation, a 2-decomposition $\{inc_\pi^1, inc_\pi^2\}$ for π exists. The block-move β preserves the correct relative order of the elements in π so that the same decomposition applies to $\pi' = \pi \cdot \beta$. It follows that π' is a woven double-strip permutation. $\square$

The permutation graph for a woven double-strip permutation has the following property: all arcs (a, b) are such that $\mathbf{a}$ and $\mathbf{b}$ are from different increasing components. A feasible arc-pair $(a, b) - (a, c)$ or $(b, a) - (c, a)$ therefore has the element $\mathbf{a}$ in one component and the elements $\mathbf{b}$ and $\mathbf{c}$ in the other. Also, $\mathbf{b}$ and $\mathbf{c}$ have to appear consecutively in the component that they are in, as the following lemma states.

Lemma 6.11 *Let π be a permutation and let $(a, b) - (a, c)$ or $(b, a) - (c, a)$ be a feasible arc-pair in $\mathcal{G}_\pi^p$. If inc_π^1 is the component that contains $\mathbf{b}$ and $\mathbf{c}$, then $\mathbf{b}$ and $\mathbf{c}$ occur consecutively in inc_π^1. If inc_π^2 is the component that contains $\mathbf{b}$ and $\mathbf{c}$, then, $\mathbf{b}$ and $\mathbf{c}$ occur consecutively in inc_π^2.*

Proof: Suppose $\mathbf{b}$ and $\mathbf{c}$ do not appear consecutively in inc_π^1 (or inc_π^2). Then, there exists an element $\mathbf{x}$, where $\mathbf{b} < \mathbf{x} < \mathbf{c}$, that is an obstacle between $\mathbf{b}$ and $\mathbf{c}$ (because the sequence that contains these elements is increasing). This implies that the arc-pair is not feasible, a contradiction. $\square$

As discussed in Section 6.3, a feasible arc-pair $(a, b) - (a, c)$ or $(b, a) - (c, a)$ may be disabled; that is, it is possible that there is a sequence of correcting block-moves that, when applied to π will cause an obstacle to occur between elements $\mathbf{b}$ and $\mathbf{c}$, thereby rendering the arc-pair an infeasible hop. The following lemma shows that this is not the case for woven double-strip permutations.

Lemma 6.12 *Let π be a woven double-strip permutation. For a given feasible arc-pair in $\mathcal{G}_\pi^p$, there is no sequence of correcting short block-moves that, when applied to π, will cause the arc-pair to be disabled.*

Proof: Let $(a, b) - (a, c)$ be a feasible arc-pair. Without loss of generality, let inc_π^1 be the component that contains $\mathbf{a}$ and let inc_π^2 be the component that contains $\mathbf{b}$ and $\mathbf{c}$. Since (a, b) and (a, c) are arcs in $\mathcal{G}_\pi^p$, the relationships $\mathbf{a} > \mathbf{b}$ and $\mathbf{a} > \mathbf{c}$ hold, by definition. We show that an element $\mathbf{x}$ satisfying $\mathbf{b} < \mathbf{x} < \mathbf{c}$ can never occur between element $\mathbf{b}$ and $\mathbf{c}$ as a result of other correcting block-moves. Such an element $\mathbf{x}$ is not in inc_π^2 as $\mathbf{b}$ and $\mathbf{c}$ are consecutive in this component (Lemma 6.11). Also, $\mathbf{x}$ must be less than $\mathbf{a}$ (by transitivity) and should therefore occur to the left of $\mathbf{a}$ in inc_π^1. As a result, a sequence of correcting block-moves will never place $\mathbf{x}$ to the right of $\mathbf{a}$. However, elements $\mathbf{b}$ and $\mathbf{c}$ are to the right of $\mathbf{a}$ in π so $\mathbf{x}$ can never occur between $\mathbf{b}$ and $\mathbf{c}$ unless the hop $B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c}\rangle$ that corresponds to arc-pair $(a, b) - (a, c)$, has already been applied to the permutation.

GETACTUAL(π, D_π)**.** Extracting an actual block-move.

INPUT: A woven double-strip permutation π and a set D_π of potential block-moves derived from a matching of $\mathcal{G}_\pi^a$.
OUTPUT: An actual block-move from D_π.

```
1   t ← max{a|(a,b) is an arc in G_π^p}   ▷ get the rightmost arc-tail
2   found ← false
3   while not found
4         do let π(p) = t
5            let h_1 = π(p+1)   ▷ element to t's right is an arc-head
6            B ← ArcSearch(D_π, (t, h_1))
7            switch case B = B⟨t, h_1⟩   ▷ an actual skip
8                        do found ← true
9                   case B = B⟨t, h_1 h_2⟩   ▷ a right hop
10                       do found ← true   ▷ an actual hop because h_2 = π(p+2)
11                  case B = B⟨t' t, h_1⟩   ▷ a left hop
12                       do let π(p') = t'
13                          if p' = p − 1
14                            then found ← true   ▷ found an actual left hop
15                            else  t ← t'
16  return B
```

Figure 6.13: Algorithm GETACTUAL.

A symmetric argument applies to the other form for an arc-pair, $(b, a) - (c, a)$. The lemma follows. □

Since disabling can be prevented as long as correcting short block-moves are performed, all that remains to be devised is a way to obtain an actual block-move from D_π.

Lemma 6.13 *Let π be a woven double-strip permutation. Let M be a maximum matching of the arc graph $\mathcal{G}_\pi^a$ and let U be the set of unmatched vertices in $\mathcal{G}_\pi^a$. Furthermore, let D_π be the set of potential block-moves derived from M and U. Then, D_π contains an actual block-move.*

Proof: Algorithm GETACTUAL, shown in Figure 6.13, details a method of extracting an actual block-move given a permutation π and the set D_π. The algorithm first obtains the element t that is the tail of an arc in $\mathcal{G}_\pi^p$ and is as large as possible (line 1). It can be verified that this is equivalent

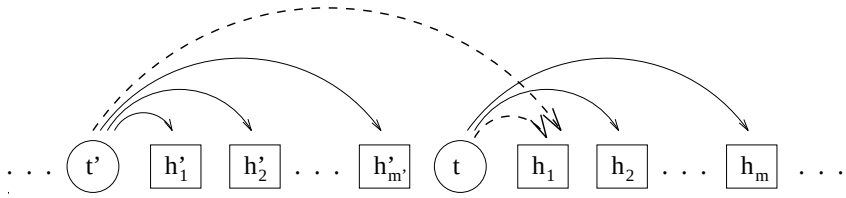

Figure 6.14: When a hop $B\langle t'\ t, h_1\rangle$ is not an actual block-move.

to the arc-tail that appears rightmost in π. Without loss of generality, let inc^1_π be the increasing component that contains t. Consider the elements $h_1, h_2, \ldots h_m$ such that (t, h_i) is an arc. There are at least one of these elements, and such elements are all in inc^2_π, the other increasing component. Also, these elements have to be contiguous in π and occur to the immediate right of t, for otherwise arc-tails that occur to the right of t would exist. Consider h_1, the element to the immediate right of t (line 5); (t, h_1) is an arc in $\mathcal{G}^p_\pi$ and the algorithm calls a procedure ArcSearch that returns the potential block-move B in D_π that contains this arc (line 6). We show that B is either an actual skip, an actual right hop, or a left hop (not necessarily actual). If B is a skip, that is, $B = B\langle t, h_1\rangle$, then it is an actual block-move as t is immediately next to h_1 and the search is complete (lines 7–8). If B is some right hop $B\langle t, h_1\ h_j\rangle$, then $h_j = h_2$, the element on the immediate right of h_1 by Lemma 6.11 and B is an actual block-move (lines 9–10). The last possibility is that B is some left hop $B\langle t'\ t, h_1\rangle$ (line 14). If t' occurs on the immediate left of t then B is an actual block-move (line 13–14). Otherwise, we note that t' and t are consecutive elements in inc^1_π (Lemma 6.11) and that the elements between them in π, say, $h'_1, h'_2, \ldots h'_{m'}$, where $m \geq 1$, are in inc^2_π, precisely the component that contains h_1. In addition, since these elements are to the left of h_1 and (t', h_1) is an arc, (t', h'_i) is an arc for each h'_i. Figure 6.14 illustrates this situation. In the figure, the elements in circles are those in component inc^1_π while the elements in squares are those in inc^2_π; the pair of arcs in dashed lines is the hop $B\langle t'\ t, h_1\rangle$. Consider t' and observe that it possesses a property similar to the property t had at the beginning of this search. More precisely, the arc (t', h'_1) is part of a block-move that is either an actual skip, an actual hop $B\langle t', h'_1\ h_j\rangle$, where $h_j = h'_2$ (h_j cannot be h_1 since (t', h_1) is already part of the hop $B\langle t, t'\ h'_1\rangle$), or some left hop. The process can thus be repeated by setting t to t' (line 15). The **while** loop of the algorithm (lines 3–15) therefore eventually finds an actual block-move since the permutation does not go infinitely to the left. □

MATCHSORT(π)**.** Minimum sorting by short block-moves using arc matching.

INPUT: A woven double-strip permutation π.
OUTPUT: The shortest sorting sequence of short block-moves, $B_1, B_2, \ldots, B_k$.

```
1   (M, U) ← a maximum matching in G_π^a   ▷ U represents the unmatched vertices
2   D_π ← potential block-moves derived from M and U
3   k ← 0
4   while |D_π| ≠ 0
5         do k ← k + 1
6            B_k ← GETACTUAL(π, D_π)
7            π ← π · B_k
8            D_π ← D_π − B_k
9   return B_1, B_2, ..., B_k
```

Figure 6.15: Algorithm MATCHSORT.

We now analyze the time complexity of Algorithm GETACTUAL. Determining the rightmost arc-tail t (line 1) can be done in $O(n)$ time. It is not necessary to inspect all the arcs. Instead, since we now know that the element to t's immediate right is an arc-head, we simply search for the rightmost position p in π where $\pi(p) > \pi(p+1)$. The **while** loop is performed $O(n)$ times in the worst case, assuming the algorithm always finds a non-actual hop (line 15). The loop requires finding the potential block-move containing the arc (t, h_1). Assuming D_π is a stored as an array of block-moves sorted by its arcs, this takes $O(\log n)$ time. As a result, Algorithm GETACTUAL runs in $O(n \log n)$ time.

We now present our main algorithm. Algorithm MATCHSORT, shown in Figure 6.15, returns an optimal sorting sequence of short block-moves given a woven double-strip permutation π. The algorithm first determines a maximum matching M of $\mathcal{G}_\pi^a$ (line 1) and then derives the set D_π of potential block-moves from M and the set of unmatched vertices U in $\mathcal{G}_\pi^a$ (line 2). Then, a **while** loop (lines 4–8) repeatedly selects an actual block-move B from D_π, applies B to π, and then removes B from D_π. The loop terminates when D_π is empty, after which, the sequence of block-moves applied is returned (line 9). We prove correctness of this algorithm in the following theorem.

Theorem 6.14 *Let π be a woven double-strip permutation. Algorithm* MATCHSORT *computes an optimal sorting sequence of short block-moves for π in $O(n^5)$ time.*

Proof: By Lemma 6.10, applying correcting block-moves to π does not destroy its woven double-strip property so that all the lemmas in this section apply at every iteration. In particular, Lemma 6.13 guarantees that there always exists an actual block-move from a set D_π derived from a maximum matching of $\mathcal{G}^a_\pi$. We note that at every iteration, the permutation is transformed to a new permutation $\pi' = \pi \cdot B$, for some $B \in D_\pi$. A naive algorithm would be to compute at every iteration the graph $\mathcal{G}^a_\pi$, a maximum matching M, and the set D_π. This is not necessary because by Lemma 6.12, no pair of arcs are rendered infeasible by a correcting block-move so that $\mathcal{G}^a_{\pi'}$ is simply $\mathcal{G}^a_\pi$ minus the arcs associated with B. It follows that some $M' \subseteq M$ is a maximum matching of $\mathcal{G}^a_{\pi'}$, and that $D_{\pi'} = D_\pi - B$. This justifies computing $\mathcal{G}^a_\pi$, M, and D_π once before entering the loop and then removing the applied block-move B from D_π at every iteration. When the loop terminates, all arcs from the permutation graph are eliminated, or, equivalently, π is sorted. Since a block-move is removed from D_π at every iteration, the length of the sorting sequence returned is $|M| + |U|$, the number of potential block-moves in D_π before entering the loop. By Lemma 6.4, this is an optimal sorting sequence.

The **while** loop in the algorithm is performed $|D_\pi| = O(n^2)$ times. The computation therefore takes $O(n^3 \log n)$ time considering that Algorithm GetActual runs in $O(n \log n)$ time. However, line 1 of the algorithm involves obtaining a maximum matching in a graph which takes $O(v^{2.5})$ [29], where v is the number of vertices in the graph. Noting that $v = O(n^2)$, matching takes $O(n^5)$ time and, as a result, dominates the running time of the algorithm. □

To illustrate how Algorithm MatchSort and Algorithm GetActual works, consider again, the example in Figure 6.12. Algorithm MatchSort returns the following sorting sequence:

$$B\langle \mathbf{10}, \mathbf{9} \rangle, B\langle \mathbf{6}, \mathbf{2\ 3} \rangle, B\langle \mathbf{6\ 8}, \mathbf{5} \rangle, B\langle \mathbf{8}, \mathbf{7} \rangle, B\langle \mathbf{4}, \mathbf{1\ 2} \rangle, B\langle \mathbf{4}, \mathbf{3} \rangle.$$

Figure 6.16 illustrates the permutation graphs for the intermediate permutations during the sort. The second block-move in the sequence, $B\langle \mathbf{6}, \mathbf{2\ 3} \rangle$, is a result of two iterations in Algorithm GetActual.

It is important to note that Algorithm MatchSort is guaranteed to work only for woven double-strip permutations. Specifically, the lemmas given in this section has to hold and this is not necessarily the case for arbitrary permutations. Nevertheless, we present a matching-based heuristic that applies to general permutations. Instead of starting with a matching and then obtaining short block-moves from that matching, we repeatedly select (actual) short block-moves in a permutation

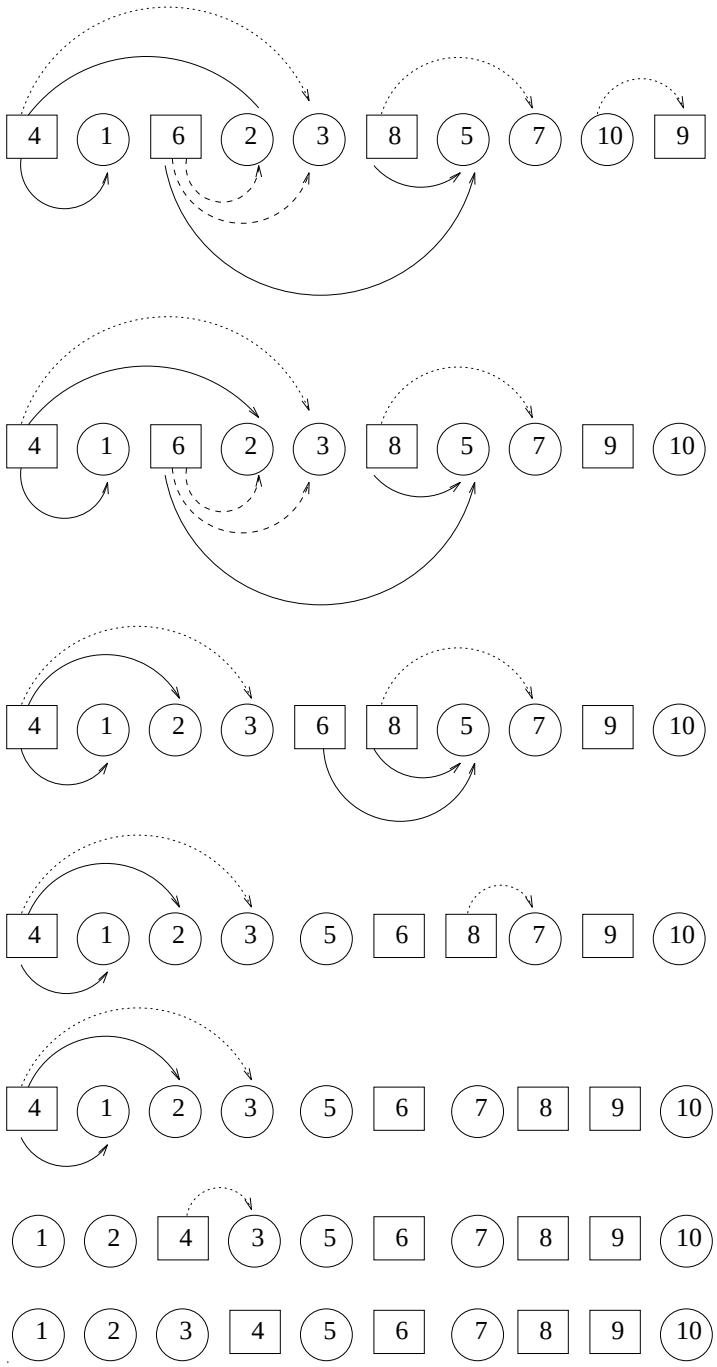

Figure 6.16: Sorting a woven double-strip permutation.

GreedyMatching(π) Sorting by short block-moves based on resulting matchings.

INPUT: A permutation π.
OUTPUT: The length of a sorting sequence of block-moves for π.

```
1   dist ← 0
2   while π ≠ ι
3         do if there are correcting hops for π
4               then β ← the correcting hop β such that
5                        G^a_{π·β} has the largest maximum matching
6               else  β ← the correcting skip β such that
7                        G^a_{π·β} has the largest maximum matching
8            π ← π · β
9            dist ← dist + 1
10  return dist
```

Figure 6.17: Algorithm GreedyMatching.

that yield a better matching. This way, we do not have to deal with potential and actual block-moves. Algorithm GreedyMatching, shown in Figure 6.17, is an algorithm that greedily chooses a correcting short block-move based on *resulting* maximum matchings. In a **while** loop (lines 2–9), the algorithm first selects, from all correcting hops, the one that yields a permutation that has the largest maximum matching in its arc graph (lines 3–5). If correcting hops do not exist, the algorithm chooses from correcting skips using the same criterion (lines 6–7). Theorem 5.1 motivates us to limit the selection to correcting short block-moves. The algorithm prefers hops over skips when there is a choice since the intention is to maximize hops (arc-pairs).

Theorem 6.15 *For arbitrary permutations, Algorithm* GreedyMatching *runs in $O(n^8)$ time.*

Proof: There are $O(n)$ possible correcting hops or skips to choose from so that a maximum matching is determined $O(n)$ times at every iteration. An arbitrary permutation may have $O(n^2)$ vertices in its corresponding arc graph. Thus, matching takes takes $O(n^5)$ time. This implies a time complexity of $O(n^6)$ time at every iteration. There are $O(n^2)$ iterations so that the resulting time complexity is $O(n^8)$. □

In practice, it is likely that the actual time complexity is better since there may, for instance,

be fewer choices for correcting block-moves at particular iterations. The computation of maximum matchings dominate this algorithm. By computing a matching for π before entering the loop, and then using it to compute the maximum matchings of resulting permutations, it is possible that computation time is significantly reduced. Care needs to be taken here since applying a block-move causes the removal of other edges in the resulting arc graph. In Chapter 9, we evaluate experimentally how well this algorithm performs and compare it against other heuristics.

6.7 Short Block-Move Diameter

In this section, we determine $Diam_{Bk^3}(n)$, the short block-move diameter of S_n. In determining this value, we seek a permutation π that is farthest from the identity permutation. A candidate for such a permutation is the decreasing permutation $\boldsymbol{\delta}$ such as $\boldsymbol{\delta}[8] = \mathbf{8\ 7\ 6\ 5\ 4\ 3\ 2\ 1}$. Recall that $\boldsymbol{\delta}$ is a woven bitonic permutation. By Theorem 6.9, Algorithm BitonicSort therefore applies to $\boldsymbol{\delta}$ and returns $MinSort_{Bk^3}(\boldsymbol{\delta}) = \lceil |\mathcal{A}^p_{\boldsymbol{\delta}}|/2 \rceil$. The permutation graph $\mathcal{G}^p_{\boldsymbol{\delta}}$ has an arc for every pair of elements in $\boldsymbol{\delta}$, for a total of $\binom{n}{2}$ arcs. Hence, the length of an optimal sorting sequence for $\boldsymbol{\delta}$ is roughly half this value. More precisely, we have this lower bound.

Lemma 6.16 *The short block-move diameter of S_n satisfies the inequality:*

$$Diam_{Bk^3}(n) \geq \left\lceil \binom{n}{2}/2 \right\rceil .$$

It remains to devise an algorithm that sorts any permutation π with at most this number of short block-moves. A naive algorithm to sort π uses *left insertion*: repeatedly position the smallest unpositioned element using hops to the left plus a skip if necessary. We refine this algorithm by reducing the number of skips performed in the sort. Define the *in-degree* of an element $\mathbf{a}$ in a permutation π, denoted by $indegree(\pi, \mathbf{a})$, as the number of elements that are greater than $\mathbf{a}$ and occur to its left. In the permutation graph $\mathcal{G}^p_\pi$, this is equivalent to the number of arcs whose head is the vertex a. Observe that if $\mathbf{a}$ is the smallest unpositioned element and $d = indegree(\pi, \mathbf{a})$ is even and nonzero, then $\mathbf{a}$ is left d-pending and Lemma 6.3 applies. Calling InsertEven$(\pi, \mathbf{a}, -d)$ described in Section 6.5 thus results in a permutation that reduces the in-degree of $\mathbf{a}$ to 0. If d is odd, then we want to find another element with odd in-degree and attempt to fix parities using Algorithm InsertSameSign also described in Section 6.5.

LEFTSORT(π)**.** Sorting by short block-moves by left insertion.

INPUT: A permutation π.
OUTPUT: The size of a sorting sequence of short block-moves.

```
a_1 ← 1, a_2 ← 2
dist ← 0
while π ≠ ι
        do let π(p_1) = a_1, π(p_2) = a_2
           d_1 ← InDegree(π, a_1)
           d_2 ← InDegree(π, a_2)
           if d_1 is even
             then (π, d) ← InsertEven(π, a_1, −d_1)
                  a_1 ← a_2, a_2 ← a_1 + 1
             else  switch case a_2 = n + 1 or p_1 < p_2
                                do (π, d) ← InsertOdd(π, a_1, −d_1)
                                   a_1 ← a_2, a_2 ← a_1 + 1
                           case p_2 < p_1 and d_2 is even
                                do (π, d) ← InsertEven(π, a_2, −d_2)
                                   a_2 ← a_2 + 1
                           case p_2 < p_1 and d_2 is odd
                                do (π, d) ← InsertSameSign (π, a_1, −d_1, a_2, −d_2)
                                   a_1 ← a_2 + 1, a_2 ← a_1 + 1
           dist ← dist + d
return dist
```

Figure 6.18: Algorithm LEFTSORT.

Consider Algorithm LeftSort, shown in Figure 6.18. Given an unsorted permutation π, it returns the size of a (not necessarily optimal) sorting sequence. It considers elements in pairs and performs mostly hops during the sort. The variables a_1 and a_2 hold the two elements that are currently being processed. In general, a_1 and a_2 are the two smallest unpositioned elements in π and are updated to be the next elements as soon as any of their in-degrees become zero. The algorithm begins by setting a_1 and a_2 to the two smallest elements in the permutation, **1** and **2** (line 1). Then, a **while** loop (lines 3–25) repeatedly positions one or both of a_1 and a_2 in π depending on certain conditions. First, note that at every iteration, p_1 and p_2 represent the elements' positions in π (line 4) while d_1 and d_2 represent their in-degrees determined by calls to a function InDegree (lines 5–6). Depending on the degree parities and positions of the elements, one of the following cases results.

Case 1. a_1 has even in-degree (lines 7–9): The element a_1 is left d_1-pending where d_1 is even so we perform hops to the left (InsertEven). After which, a_1 gets the value of a_2 and a_2 gets the next smallest element.

Case 2. a_1 has odd in-degree (lines 10–18): Here, the algorithm considers element a_1 with element a_2 in case the in-degree parity of a_1 can be made even. We have the following subcases:

- **Case 2a.** a_1 is the last unpositioned element or $p_1 < p_2$ (lines 10–12): As a_1 is left d_1-pending, where d_1 is odd, the algorithm performs hops and then spends a skip in this case for a_1 (InsertOdd). After which, a_1 and a_2 are updated.
- **Case 2b.** a_2 has even in-degree and $p_2 < p_1$ (lines 13–15): The element a_2 is left d_2-pending where d_2 is even so left hops are performed (InsertEven) and then a_2 is updated to be the next smallest element.
- **Case 2c.** a_2 has odd in-degree and $p_2 < p_1$ (lines 16–18): The element a_1 and a_2 have odd in-degrees and are pending elements going in the same direction. Algorithm InsertSameSign therefore applies and performs only correcting hops to position these elements. Finally, a_1 and a_2 are updated to be the next elements.

Consider sorting the permutation $\pi =$ **2 8 6 4 3 9 1 5 10 7 11**. Figure 6.7 illustrates how π is sorted by Algorithm LeftSort. The figure indicates the case that applies in each stage; the values for a_1 and a_2 are shown in boxes.

Table 6.2: Sorting by left insertion.

Stage	π	Case
(a)	$\boxed{\mathbf{2}}$ **8 6 4 3 9** $\boxed{\mathbf{1}}$ **5 10 7 11**	1
(b)	**1** $\boxed{\mathbf{2}}$ **8 6 4** $\boxed{\mathbf{3}}$ **9 5 10 7 11**	2b
(c)	**1 2 8 6** $\boxed{\mathbf{4}}$ $\boxed{\mathbf{3}}$ **9 5 10 7 11**	2b
(d)	**1 2 4 8 6** $\boxed{\mathbf{3}}$ **9** $\boxed{\mathbf{5}}$ **10 7 11**	2a
(e)	**1 2 3 4 8** $\boxed{\mathbf{6}}$ **9** $\boxed{\mathbf{5}}$ **10 7 11**	2c
(f)	**1 2 3 4 5 6** $\boxed{\mathbf{8}}$ **9 10** $\boxed{\mathbf{7}}$ **11**	2a
	1 2 3 4 5 6 7 8 9 10 11	

Although Algorithm LeftSort does not necessarily sort a permutation optimally, it does sort using only correcting short block-moves. In the next theorem, we bound the number of short block-moves it performs. Notice that, in general, hops are used in the sort. Only in Case 2a is a skip performed (through Algorithm InsertOdd), that is, when a_1 is the last element (Stage (f) in the example) or a_2 is to the right of a_1 (Stage (d)). The latter condition means a_1 and a_2 are already in their correct relative order in π (observe that $a_1 < a_2$). We use this fact to obtain an upper bound for the diameter.

Theorem 6.17 *The short block-move diameter of S_n satisfies*

$$Diam_{Bk^3}(n) = \left\lceil \binom{n}{2}/2 \right\rceil.$$

Proof: Let s be the number of skips performed by Algorithm LeftSort on some permutation π. Since each of these skips is a result of Case 2a in the algorithm, elements already in their correct relative order are revealed per skip (a_1 and a_2 in the algorithm), except perhaps for the last skip. Thus, there are at least $s-1$ pairs of elements that are *not* connected by an arc in the permutation graph of π. Since there are at most $\binom{n}{2}$ arcs in any permutation graph,

$$|\mathcal{A}^p_\pi| \leq \binom{n}{2} - (s-1).$$

Let h be the number of hops performed by the algorithm. Each skip corresponds to a single arc so that a total of $|\mathcal{A}^p_\pi| - s$ arcs remain for hops. A hop corresponds to two arcs so that:

$$h = (|\mathcal{A}^p_\pi| - s)/2 \leq \left(\binom{n}{2} - 2s + 1\right)/2 = \left(\binom{n}{2} + 1\right)/2 - s.$$

The short-block move diameter is at most the total number of hops and skips performed by the algorithm:

$$\begin{aligned} h+s &\leq \left(\binom{n}{2}+1\right)/2 - s + s \\ &= \left(\binom{n}{2}+1\right)/2 \\ &\leq \left\lceil \binom{n}{2}/2 \right\rceil . \end{aligned}$$

This value is also a lower bound for the short block-move diameter as shown in Lemma 6.16. We conclude that this value is exactly the short block-move diameter for length-n permutations. □

Although useful for determining short block-move diameter, Algorithm LeftSort is not a practical algorithm for sorting a permutation as it will often compute a non-optimal sorting sequence. However, it suggests a heuristic based on degrees of elements in a permutation graph. Algorithm DegreeSort, shown in Figure 6.19, is a refinement on Algorithm LeftSort. It considers the elements of the permutation in pairs and positions these elements depending on the parities of their in-degrees and out-degrees. An *out-degree* of an element has an analogous definition: $outdegree(\pi, \mathbf{a})$ is the number of elements less than $\mathbf{a}$ that occur to its right in π. Whereas LeftSort considers pairs of elements that represent the two smallest unpositioned elements in the permutation, DegreeSort considers the maximum elements as well. Two pairs are considered at every iteration of the **while** loop (lines 2–24): the two smallest unpositioned elements stored in variables a_1 and a_2, and the two largest unpositioned elements stored in variables b_1 and b_2. One or two of these four elements are positioned based on their in-degrees (for a_1 and a_2) or out-degrees (for b_1 and b_2). The positions and degrees of these elements that these variables hold are computed at every iteration (lines 3–7). The functions InDegree$(\pi, \mathbf{a})$ and OutDegree$(\pi, \mathbf{b})$ return the in-degree and out-degree of an element, respectively. An even in-degree for a_1 (the minimum unpositioned element) or an even out-degree for b_2 (the maximum unpositioned element) means that the element can be positioned using only correcting hops by Lemma 6.3 (lines 8–13), as in Case 1 in the discussion of LeftSort. If a_2, the next smallest unpositioned element, is $n+1$, then a_1 is the last unpositioned element and is inserted regardless of its parity (14–16). Similarly, if b_1, the next largest unpositioned element, is -1, then b_2 is the last unpositioned element (lines 17–19). The next cases (lines 20–22) are presented separately in Figure 6.20 due to page length constraints; these are the cases where both a_1 and b_2

DEGREESORT(π)**.** Sorting by short block-moves based on element degrees.

INPUT: A permutation π.
OUTPUT: The size of a sorting sequence of short block-moves.

```
1   a_1 ← 1, a_2 ← 2, b_1 ← n − 1, b_2 ← n
2   dist ← 0
3   while π ≠ ι
4         do let π(p_1) = a_1, π(p_2) = a_2, π(q_1) = b_1, π(q_2) = b_2
5            d_1 ← InDegree(π, a_1)
6            d_2 ← InDegree(π, a_2)
7            e_1 ← OutDegree(π, b_1)
8            e_2 ← OutDegree(π, b_2)
9            switch case d_1 is even
10                   do (π, d) ← InsertEven(π, a_1, −d_1)
11                      a_1 ← a_2, a_2 ← a_1 + 1
12                 case e_2 is even
13                   do (π, d) ← InsertEven(π, b_2, e_2)
14                      b_2 ← b_1, b_1 ← b_1 − 1
15                 case a_2 = n + 1
16                   do (π, d) ← InsertOdd(π, a_1, −d_1)
17                      a_1 ← a_2, a_2 ← a_1 + 1
18                 case b_1 = 0
19                   do (π, d) ← InsertOdd(π, b_2, e_2)
20                      b_2 ← b_1, b_1 ← b_2 − 1
21                 otherwise   ▷ d_1 and e_2 are odd
22                   do (π, d, a_1, a_2, b_1, b_2) ←
23                         BothOdd(π, a_1, p_1, d_1, a_2, p_2, d_2, b_1, q_1, e_1, b_2, q_2, e_2)
24           dist ← dist + d
25  return dist
```

Figure 6.19: Algorithm DEGREESORT.

have odd degree.

If a_2 has even degree and can be positioned without crossing element a_1, then it is inserted in the permutation (lines 1–3); similarly if b_1 has even degree and can be positioned without crossing element b_2, then it is inserted (lines 4–6). These two cases correspond to Case 2b in the discussion of LeftSort. The next two cases in the algorithm (lines 7–12), on the other hand, correspond to Case 2c in the same discussion. These are the cases where two elements with odd degrees going in the same direction are detected that could correct each other's degree parity. The remaining cases (lines 13–25) represent the situation where a_1 and b_2 have odd degree and both pairs (a_1, a_2) and (b_1, b_2) have their elements already in their correct relative order. It is possible that a_2 could correct the in-degree parity of a_1: if the elements are an odd distance apart and all elements between them are greater than a_2. If this condition is satisfied, a call to InsertEven causes a_2 to occur to the immediate right of a_1 and then a single correcting right hop shifts the parity of a_1 (lines 13–17). A symmetric discussion holds for the possibility where b_1 corrects the out-degree parity of b_2 (lines 18–22). If none of the above possibilities occur, the algorithm spends a skip and positions a_1 (lines 23–25).

Algorithm DegreeSort is in fact a greedy algorithm that selects correcting hops and spends a skip only when absolutely necessary given the information on the two pairs of elements it is currently processing. We evaluate this heuristic and compare it with others in Chapter 9. The following theorem states the time complexity of Algorithm DegreeSort.

Theorem 6.18 *Algorithm* DegreeSort *runs in $O(n^2)$ time.*

Proof: The while loop in the algorithm performs $O(n)$ iterations since an element is always positioned at every iteration. Each iteration performs $O(n)$ correcting block-moves regardless of which case holds. The computations of degrees and positions always performed at the beginning of the loop are easily done within $O(n)$ time as these require straightforward scans of the permutation (in fact, $O(1)$ suffices if the degrees and positions are computed before entering the loop and then updated at every iteration). It follows that the algorithm runs in $O(n^2)$ time. □

BothOdd($\pi, a_1, p_1, d_1, a_2, p_2, d_2, b_1, q_1, e_1, b_2, q_2, e_2$) Used by Algorithm DegreeSort.

```
switch case p_2 < p_1 and d_2 is even
        do (π, d) ← InsertEven(π, a_2, −d_2)
           a_2 ← a_2 + 1
    case q_2 < q_1 and e_1 is even
        do (π, d) ← InsertEven(π, b_1, e_1)
           b_1 ← b_1 − 1
    case p_2 < p_1 and d_2 is odd
        do (π, d) ← InsertSameSign(π, a_1, −d_1, a_2, −d_2)
           a_1 ← a_2 + 1, a_2 ← a_1 + 1
    case q_2 < q_1 and e_1 is odd
        do (π, d) ← InsertSameSign(π, b_1, e_1, b_2, e_2)
           b_2 ← b_1 − 1, b_1 ← b_2 − 1
    case p_2 − p_1 is odd and a_2 is left (p_2 − p_1 − 1)-pending
        do (π, d) ← InsertEven(π, a_2, −(p_2 − p_1 − 1))
           π ← π · β(p_1 − 1, p_1, p_1 + 2)
           d ← d + 1
           ▷ a_1 and a_2 remain unchanged
    case q_2 − q_1 is odd and b_1 is right q_2 − q_1 − 1)-pending
        do (π, d) ← InsertEven(π, b_1, q_2 − q_1 − 1)
           π ← π · β(q_2 − 1, q_2 + 1, q_2 + 2)
           d ← d + 1
           ▷ a_1 and a_2 remain unchanged
    otherwise
        do (π, d) ← InsertOdd(π, a_1, −d_1)
           a_1 ← a_2, a_2 ← a_1 + 1
return (π, d, a_1, a_2, b_1, b_2)
```

Figure 6.20: Subcases for Algorithm DegreeSort.

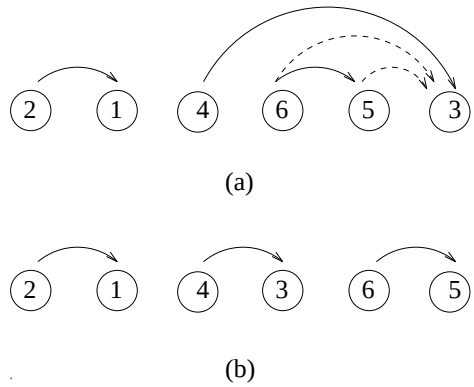

Figure 6.21: Lone arcs in a permutation graph.

6.8 A $\frac{4}{3}$-Approximation Algorithm

In this section, we present an algorithm for $MinSort_{Bk^3}$ that guarantees, for any permutation, a sorting sequence whose length is within $\frac{4}{3}$ of the optimal. As in the algorithms in the previous sections, the general strategy is to perform as many correcting hops as possible and avoid performing skips. Consider the permutation graph $\mathcal{G}_\pi^p$ of a given permutation π. A *lone arc* (a, b) in $\mathcal{G}_\pi^p$ is an arc that is not compatible with any other arc (see Figure 6.21, for an example). Observe that lone arcs have to correspond to a skip in a sorting sequence (under Theorem 5.1), since correcting hops correspond to feasible (hence compatible) arc-pairs. The number of lone arcs in a graph thus gives us a lower bound on the number of correcting skips that *must* be performed when sorting a permutation and a slightly tighter lower bound for the short block-move distance of π.

Lemma 6.19 *Let π be a permutation and let r be the number of lone arcs in $\mathcal{G}_\pi^p$. The short block-move distance of π satisfies the inequality:*

$$MinSort_{Bk^3}(\pi) \geq \lceil (|\mathcal{A}_\pi^p| - r)/2 \rceil + r.$$

Recall that potential block-moves are block-moves specified by the arcs of $\mathcal{G}_\pi^p$ but are not necessarily actual block-moves (immediately applicable to π—see Section 6.6). We call the potential skip that corresponds to a lone arc a *lone skip*.

Lemma 6.20 *Let π be a permutation and let B be a lone skip in π. Then B is an actual block-move.*

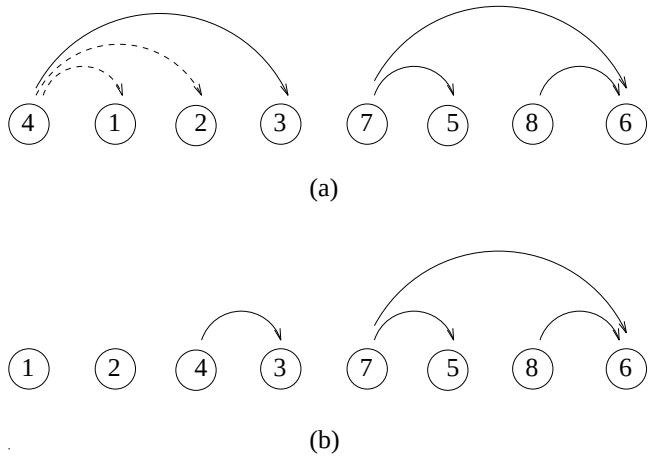

Figure 6.22: Finding desirable correcting hops.

Proof: Let $B = B\langle \mathbf{a}, \mathbf{b} \rangle$ and let $\mathbf{a} = \pi(i)$ and $\mathbf{b} = \pi(j)$. We want to show that $j = i + 1$, that is, $\mathbf{a}$ and $\mathbf{b}$ are immediately next to each other in π. Suppose this is not the case. Then there exists an element $\mathbf{x}$ between $\mathbf{a}$ and $\mathbf{b}$. Since (a, b) is an arc in $\mathcal{G}_\pi^p$, either the arc (a, x) or the arc (x, b) is an arc in $\mathcal{G}_\pi^p$. This results in a contradiction because (a, b) is a lone arc so that no other arcs exist whose head is $\mathbf{b}$ or whose tail is $\mathbf{a}$. The lemma follows. $\square$

Consider an algorithm that sorts π using correcting block-moves. Such an algorithm may *introduce* lone arcs. Let $\pi = \mathbf{2\ 1\ 4\ 6\ 5\ 3}$. The permutation graph of π is given in Figure 6.21(a). Observe that $(2, 1)$ is a lone arc for this permutation. However, upon the application of the correcting hop $B = B\langle \mathbf{6\ 5}, \mathbf{3} \rangle$ (shown in dashed arcs), two additional lone arcs, $(4, 3)$ and $(6, 5)$, result. The permutation graph for $\pi \cdot B$ is given in Figure 6.21(b). This implies three more block-moves (lone skips) to complete the sort. Of course, an optimal sorting sequence for this permutation applies the two correcting hops $B\langle \mathbf{6}, \mathbf{5\ 3} \rangle$ and $B\langle \mathbf{4\ 5}, \mathbf{3} \rangle$, and then the lone skip $B\langle \mathbf{2}, \mathbf{1} \rangle$.

We would prefer to apply a hop that does not introduce lone arcs. This is sometimes unavoidable as shown in Figure 6.22. Here, the permutation $\pi = \mathbf{4\ 1\ 2\ 3\ 7\ 5\ 8\ 6}$ allows only one correcting hop $B\langle \mathbf{4}, \mathbf{1\ 2} \rangle$ (shown in dashed arcs) and its application introduces the lone arc $(4, 3)$ in the permutation. A compromise would be to always perform a correcting hop that introduces at most one lone arc. In this way, the resulting sorting sequence performs at least as many hops as there are lone skips (not counting the necessary ones implied by the lone arcs that were originally in the permutation

graph). There is another problem to address that is also exhibited in Figure 6.22. After applying the only correcting hop possible, no correcting hops remain (Figure 6.22(b)). Define a *mushroom* in a permutation π as a subsequence of contiguous elements $\mathbf{a}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}$ in π, that satisfies $\mathbf{c} > \mathbf{a} > \mathbf{d} > \mathbf{b}$. It is called such because of the way the three arcs associated with the subsequence can be drawn. We call the arc (a, d) the *dome* of the mushroom, and the arcs (a, b) and (c, d) its *shades*. In Figure 6.22(a), the subsequence **7 5 8 6** comprises a mushroom. Such subsequences in a permutation are crucial to the analysis of our algorithm, as we shall see later. For now, observe that the correcting skip $B\langle \mathbf{a}, \mathbf{b}\rangle$ and then the correcting hop $B\langle \mathbf{a}\ \mathbf{c}, \mathbf{d}\rangle$ places the elements in a mushroom in their correct relative order.

Our algorithm, which we describe later in this section, is a while loop that distinguishes between three cases:

1. The permutation contains lone skips.
2. The permutation contains correcting hops.
3. The permutation contains a mushroom.

We argue that at least one of these cases hold. It suffices to show that the third case holds when neither of the first two cases hold. That is, that at least one mushroom exists when an unsorted permutation contains no lone skips or correcting hops. We first prove the existence of part of that mushroom. An *adjacent inversion* in π is a pair of elements $\{\mathbf{a}, \mathbf{b}\}$ such that $\mathbf{a} > \mathbf{b}$ and $\mathbf{a}$ occurs to the immediate left of $\mathbf{b}$.

Lemma 6.21 *An unsorted permutation π contains at least one adjacent inversion.*

Proof: Let $\mathbf{a} = \pi(i)$ be the largest unpositioned element in π. We claim that the element to its immediate right, $\mathbf{b} = \pi(i+1)$, is less than $\mathbf{a}$. Suppose this is not the case. Then, $\mathbf{b} > \mathbf{a}$ and is thus in its correct position; that is, $i+1 = \mathbf{b}$ so that $\mathbf{a} < i+1$. The element $\mathbf{i}$ is not in its correct position since $\mathbf{a}$ occupies its position. Hence, $i < \mathbf{a}$. This implies $i < \mathbf{a} < i+1$, which cannot happen, as $\mathbf{a}$ is an integer. Thus, $\mathbf{a} > \mathbf{b}$ and the pair of elements $\mathbf{a}$ and $\mathbf{b}$ is an adjacent inversion in π. The lemma follows. □

We locate a mushroom in a permutation by considering its adjacent inversions.

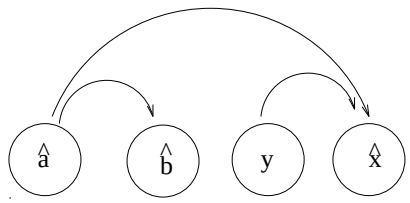

Figure 6.23: Finding a mushroom in a permutation.

Lemma 6.22 *An unsorted permutation containing no lone skips or correcting hops contains a mushroom.*

Proof: Let π be an unsorted permutation. By Lemma 6.21, π contains at least one adjacent inversion. Let $\mathbf{a} = \pi(i)$ and $\mathbf{b} = \pi(i+1)$ be one such inversion. Now, this implies (a, b) is an arc in $\mathcal{G}_\pi^a$. Since (a, b) is not a lone arc, there exists an arc (a, x) or an arc (x, b) for some element $\mathbf{x} = \pi(j)$. We are interested in all elements $\mathbf{x}$ that satisfy this property. Define $span((a, b), \mathbf{x})$ for any element $\mathbf{x} \neq \mathbf{a}, \mathbf{b}$ as follows:

$$span((a,b),\mathbf{x}) = \begin{cases} j-(i+1) & \text{if } (a,x) \in \mathcal{A}_\pi^p \\ i-j & \text{if } (x,b) \in \mathcal{A}_\pi^p \\ \infty & \text{if neither arc exists.} \end{cases}$$

Define $minspan(\mathbf{a}, \mathbf{b})$ as follows:

$$minspan(\mathbf{a},\mathbf{b}) = \min_{\mathbf{x} \neq \mathbf{a},\mathbf{b}} span((a,b),\mathbf{x}).$$

That is, we seek the element $\mathbf{x}$ that yields the smallest span possible with the adjacent inversion $\{\mathbf{a}, \mathbf{b}\}$. We call $\mathbf{x}$ the *closest connected neighbor* of the inversion. Finally, among all adjacent inversions in π, let $\{\hat{a}, \hat{b}\}$ be the inversion that yields the smallest minimum span and let $\hat{\mathbf{x}}$ be its closest connected neighbor.

Either $(\hat{a}, \hat{x})$ or $(\hat{x}, \hat{b})$ is an arc in $\mathcal{G}_\pi^p$. We consider the first case and the second case follows by symmetry. We claim that the elements $\hat{\mathbf{a}}$, $\hat{\mathbf{b}}$, and $\hat{\mathbf{x}}$ form part of a mushroom in π. Specifically, the arc $(\hat{a}, \hat{x})$ is the dome of this mushroom and the arc $(\hat{a}, \hat{b})$ is one of its shades. Since no correcting hops exist in π, there exists at least one element between $\hat{\mathbf{b}}$ and $\hat{\mathbf{x}}$. We show that *exactly one* such element $\mathbf{y}$ exists and that this element forms with $\hat{\mathbf{x}}$ the other shade $(y, \hat{x})$ of the mushroom, as illustrated in Figure 6.23. First, notice that any such element $\mathbf{y}$ between $\hat{\mathbf{b}}$ and $\hat{\mathbf{x}}$ must be adjacent

with either $\hat{\mathbf{a}}$ or $\hat{\mathbf{x}}$ because of the arc $(\hat{a}, \hat{x})$. Next, notice that the element $\mathbf{y}$ cannot be adjacent with $\hat{\mathbf{a}}$ because this would mean $\mathbf{y}$, not $\hat{\mathbf{x}}$, is the closest connected neighbor of $\{\hat{a}, \hat{b}\}$. Thus, $(y, \hat{x}) \in \mathcal{A}_\pi^p$ for any $\mathbf{y}$ between $\hat{\mathbf{b}}$ and $\hat{\mathbf{x}}$. It remains to show that only one such $\mathbf{y}$ exists. Suppose two or more elements occur between $\hat{\mathbf{b}}$ and $\hat{\mathbf{x}}$. Then a correcting hop occurs between $\hat{\mathbf{x}}$ and the two elements to its immediate left, contrary to the assumptions of the lemma. We conclude that $\mathbf{y}$ is unique and that $\hat{\mathbf{a}}\ \hat{\mathbf{b}}\ \mathbf{y}\ \hat{\mathbf{x}}$ is a subsequence of contiguous elements. To show that this comprises a mushroom, we want $\mathbf{y} > \hat{\mathbf{a}} > \hat{\mathbf{x}} > \hat{\mathbf{b}}$. All relationships have been shown in the above proof except for $\hat{\mathbf{x}} > \hat{\mathbf{b}}$. This is easily shown because if this were not the case, then the correcting hop $B\langle \hat{\mathbf{b}}\ \mathbf{y}, \hat{\mathbf{x}} \rangle$ would exist. The lemma follows. □

Recall that our general strategy is to avoid skips. One way to do so is to prevent, whenever possible, the introduction of lone arcs. A lone arc is *pre-existing* if it is present in the permutation graph of π before some algorithm begins to sort π. Skips that correspond to pre-existing lone arcs are called *required skips*. The goal is to show that every non-required skip performed during the sort is amortized by a hop. To do this, we need to precisely identify when a lone arc is introduced by a block-move.

Since a lone arc (a, b) is not compatible with other arcs, it should be clear that $outdegree(\pi, \mathbf{a}) = 1$ and $indegree(\pi, \mathbf{b}) = 1$. In addition, $indegree(\pi, \mathbf{a}) = 0$ and $outdegree(\pi, \mathbf{b}) = 0$, since otherwise there are other arcs whose heads or tails are these vertices and we would have $outdegree(\pi, \mathbf{a}) > 1$, or $indegree(\pi, \mathbf{a}) > 1$. An arc (a, b) is a *right hanging arc* if it satisfies $indegree(\pi, \mathbf{a}) = 0$, $outdegree(\pi, \mathbf{a}) > 1$, $indegree(\pi, \mathbf{b}) = 1$, and $outdegree(\pi, \mathbf{b}) = 0$; it is a *left hanging arc* if it satisfies $indegree(\pi, \mathbf{a}) = 0$, $outdegree(\pi, \mathbf{a}) = 1$, $indegree(\pi, \mathbf{b}) > 1$, and $outdegree(\pi, \mathbf{b}) = 0$. Hanging arcs are in danger of becoming lone arcs upon application of some block-move. In a right hanging arc (a, b), $\mathbf{a}$ is called a *right critical element* and $\mathbf{b}$ is called a *a right leaf element*; in a left hanging arc (a, b), $\mathbf{b}$ is called a *left critical element* and $\mathbf{a}$ is called a *a left leaf element*. Clearly, any block-move that introduces a lone arc must affect the degree of a critical element and cause it to be immediately next to its corresponding leaf element. The following statements are directly implied by the above definitions.

1. A critical element can not be a leaf element, and vice versa.

2. A right critical element can not be a left critical leaf element, and vice versa.

3. A right leaf element can not be a left leaf element, and vice versa.

A final observation is that, although a critical element may be associated with several leaf elements, only one such leaf element is relevant when analyzing whether a particular block-move introduces a lone arc.

The following lemmas indicate when and how many lone arcs are introduced by lone skips, correcting hops, and block-moves that eliminate a mushroom.

Lemma 6.23 *Applying a lone skip to a permutation does not introduce a lone arc.*

Proof: Let π be a permutation with a lone skip $B\langle \mathbf{a}, \mathbf{b} \rangle$. The arc (a, b) is a lone arc and hence is not a hanging arc; this implies that **a** and **b** are non-critical elements. Since applying B to π affects only the degrees of the elements **a** and **b**, it does not introduce a lone arc. □

Lemma 6.24 *For a permutation π containing correcting hops, there exists a correcting hop that, when applied to π, introduces at most one lone arc.*

Proof: Let B be a correcting hop in π whose application introduces at least two lone arcs. We assume that B is a right hop; the case where B is a left hop follows by symmetry. The objective is to identify an alternate correcting hop that introduces at most one lone arc. Let $B = B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c} \rangle$. Since B introduces at least 2 lone arcs, at least two of the following must hold:

(1) The element **a** is right critical.

(2) The element **b** is left critical.

(3) The element **c** is left critical.

Observe that if element **c** is left critical, its corresponding left leaf is **b** so that **b** cannot be left critical. This means at most two lone arcs may be introduced by the block-move and this can occur only when either statements **(1)** and **(2)** hold or when statements **(1)** and **(3)** hold.

Suppose statements **(1)** and **(2)** hold. Figure 6.24 illustrates part of the corresponding permutation graph. Element **x** in the figure is the left leaf that corresponds to the left critical element **b**. Element **y**, on the other hand, is the right leaf that corresponds to the right critical element **a**. Note that the elements **x**, **a**, **b**, **c**, and **y** have to be contiguous in the permutation and that the right hop

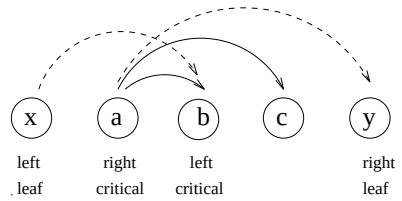

Figure 6.24: The case where $B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c}\rangle$ introduces lone arcs (x, b) and (a, y).

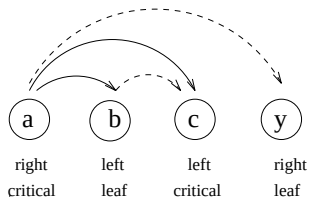

Figure 6.25: The case where $B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c}\rangle$ introduces lone arcs (b, c) and (a, y).

$B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c}\rangle$ does introduce two lone arcs, specifically, (x, b) and (a, y). Now, consider the correcting hop $B' = B\langle \mathbf{x}\ \mathbf{a}, \mathbf{b}\rangle$. Upon application of B', the elements **a** and **y** are not adjacent in the resulting permutation so (a, y) does not occur as a lone arc in $\pi \cdot B'$. In addition, as **x** is a left leaf, it cannot cause the introduction of a lone arc (it is not right critical). This means that a lone arc could be introduced only through the left critical element **b**. Thus, B' is a correcting hop that introduces at most one lone arc.

For the second possibility, suppose statements **(1)** and **(3)** hold. Figure 6.25 illustrates part of the corresponding permutation graph. Again, element **y** in the figure is the right leaf that corresponds to the right critical element **a**. Recall that **b** is the left leaf that corresponds to the left critical element **c**. Note that the elements **a**, **b**, **c**, and **y** have to be contiguous in the permutation and that the right hop $B\langle \mathbf{a}, \mathbf{b}\ \mathbf{c}\rangle$ does introduce two lone arcs, specifically, (b, c) and (a, y). Now, consider the correcting hop $B' = B\langle \mathbf{a}\ \mathbf{b}, \mathbf{c}\rangle$. Upon application of B', the elements **a** and **y** are not adjacent in the resulting permutation so (a, y) does not occur as a lone arc in $\pi \cdot B'$. In addition, **b** is a left leaf so it cannot cause the introduction of a lone arc (it is not right critical). This means that a lone arc could be introduced only through the left critical element **c**. Thus, B' is a correcting hop that introduces at most one lone arc. The lemma follows. □

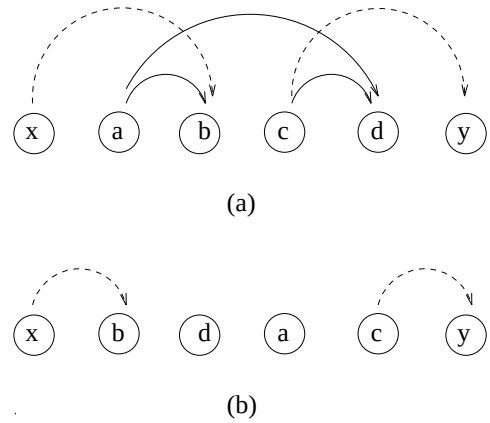

Figure 6.26: Arcs between mushroom elements.

Lemma 6.25 *Let π be an unsorted permutation containing no lone skips or correcting hops. Let* $\mathbf{a\ b\ c\ d}$ *be a mushroom in π. Applying the block-moves $B\langle \mathbf{a}, \mathbf{b}\rangle$ and $B\langle \mathbf{a\ c}, \mathbf{d}\rangle$ does not introduce lone arcs.*

Proof: Let $\pi' = \pi \cdot B\langle \mathbf{a}, \mathbf{b}\rangle \cdot B\langle \mathbf{a\ c}, \mathbf{d}\rangle$. The four elements of the mushroom in π still occur contiguously in π'; specifically, they occur as the contiguous subsequence $\mathbf{b\ d\ a\ c}$. In addition, no arcs exist between them in $\mathcal{G}^p_{\pi'}$ since the block-moves have eliminated the arcs implied by the mushroom, as Figure 6.26 illustrates. In the figure, there are other elements besides $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$. We explain these elements next.

Suppose lone arcs exist in π'. Since no arcs remain between the elements $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$, the only possible lone arcs are the arc (x, b) for some element $\mathbf{x}$ to the immediate left of $\mathbf{b}$ in π' or the arc (c, y) for some element $\mathbf{y}$ to the immediate right of $\mathbf{c}$ in π' (See Figure 6.26(b)). Neither of these possibilities can occur because this would mean that $\mathbf{b}$ is left critical in π with $\mathbf{x}$ as its left leaf ((x, b) is a left hanging arc) or that $\mathbf{c}$ is right critical in π with $\mathbf{y}$ as its right leaf ((c, y) is a right hanging arc), as illustrated in Figure 6.26(a). Both cases imply the existence of a correcting hop in π ($B\langle \mathbf{x\ a}, \mathbf{b}\rangle$ or ($B\langle \mathbf{c}, \mathbf{d\ y}\rangle$), which contradicts the assumptions of the lemma. It follows that no lone arcs are introduced by the block-moves. □

We are now ready to describe our algorithm. Algorithm GreedyHops, shown in Figure 6.27, takes as input a permutation π and returns the length of a sorting sequence for π. The **while** loop in

GreedyHops(π) Sorting by short block-moves based on greedy hop choices.

INPUT: A permutation π.
OUTPUT: The length of a sorting sequence of block-moves for π.

```
dist ← 0
while π ≠ ι
      do switch case there are lone skips in π
                     do let β be one such skip
                         π ← π · β
                         dist ← dist + 1
                case there are correcting hops for π
                     do let β be the hop that introduces the fewest lone arcs
                         π ← π · β
                         dist ← dist + 1
                otherwise
                     do let π(i), π(i + 1), π(i + 2), π(i + 3) be a mushroom in π
                         π ← π · β⟨i, i + 1, i + 2⟩
                         π ← π · β⟨i + 1, i + 3, i + 4⟩
                         dist ← dist + 2
return dist
```

Figure 6.27: Algorithm GreedyHops.

lines 2–16 repeatedly applies block-moves based on the existence of lone skips or correcting hops in the permutation. If π contains a lone skip, then one such skip is selected and applied (lines 3–6). If it does not contain any lone skips and it contains at least one correcting hop, then the correcting hop that introduces the fewest lone arcs is selected and applied (lines 7–10). Otherwise, we search for a mushroom in the permutation graph of π and apply two block-moves that remove the corresponding arcs (lines 11–16).

Theorem 6.26 *Algorithm* GreedyHops *sorts a permutation π with a sequence of short block-moves whose length is within $\frac{4}{3}$ of the optimal, in $O(n^3)$ time.*

Proof: Let π be a permutation. Lemma 6.22 guarantees that every iteration of the **while** loop in the algorithm performs correcting block-moves and thus eventually sorts π. Let r be the number of pre-existing lone arcs in $\mathcal{G}_\pi^p$. Clearly, the algorithm first eliminates these lone arcs so that it performs r required skips, initially. Any other lone arcs eliminated by the algorithm will have to be introduced by other block-moves performed during the sort. By Lemmas 6.23 and 6.25, lone arcs will not be introduced by the block-moves in lines 5, 13, and 14 of the algorithm. In fact, by Lemma 6.24, at most one lone arc is possibly introduced whenever the block-move applied is the one indicated in line 9 of the algorithm, a correcting hop. This means that every non-required lone skip performed in line 5 is amortized by some correcting hop in line 9. Also, the correcting skip in line 13 is amortized by the correcting hop in line 14 (the mushroom case). Let s and h be the number of non-required skips and correcting hops performed by the algorithm. By the above discussion, we have $h \geq s$. Noting that skips and hops eliminate 1 or 2 arcs from the graph $\mathcal{G}_\pi^p$, respectively, the number of arcs in this graph is

$$|\mathcal{A}_\pi^p| = 2h + s + r.$$

By Lemma 6.19,

$$\begin{aligned} MinSort_{Bk^3}(\pi) &\geq \lceil (|\mathcal{A}_\pi^p| - r)/2 \rceil + r \\ &= (2h + s)/2 + r \\ &= h + s/2 + r. \end{aligned}$$

To compute the performance guarantee of Algorithm GreedyHops, let $GreedyHops(\pi)$ denote the number of block-moves performed by algorithm. By definition,

$$GreedyHops(\pi) = h + s + r.$$

If $s = 0$, then $GreedyHops(\pi) = MinSort_{Bk^3}(\pi)$; that is, GreedyHops returns an optimal solution. Otherwise, we have these inequalities on the approximation ratio:

$$\begin{aligned}
\frac{GreedyHops(\pi)}{MinSort_{Bk^3}(\pi)} &\leq \frac{h+s+r}{h+s/2+r} \\
&\leq \frac{h+s}{h+s/2} \\
&\leq \frac{s+s}{s+s/2} \quad \text{since } h \geq s > 0 \\
&= \frac{4}{3}.
\end{aligned}$$

For the time complexity, the **while** loop in the algorithm performs $O(n^2)$ iterations. Each iteration requires at worst a linear scan of the elements since all it requires is consider every skip (first case) or every hop (second case) or every possible subsequence of 4 contiguous elements (third case). It follows that the algorithm runs in $O(n^3)$ time. □

6.9 Short Generators

The techniques in this chapter allow us to derive immediate results for the minimum sorting and diameter problems for short generators. Recall that short generators, described by the predicate Gn^3, permute the elements of any block of length 3 in a permutation. Apart from short block-moves, the only other short generator is a 3-reversal. By Theorem 5.2, it suffices to consider only correcting short generators when seeking an optimal sorting sequence. This means that the permutation graph model in this chapter remains appropriate for $MinSort_{Gn^3}$. A correcting 3-reversal corrects three inversions, or, equivalently, eliminates 3 arcs from the permutation graph $\mathcal{G}_\pi^p = (\mathcal{V}_\pi^p, \mathcal{A}_\pi^p)$. As this is the largest number of arcs a short generator can eliminate, we arrive at the following lower and upper bounds, which we state without proof.

Theorem 6.27 *The short generator distance of a permutation π satisfies the inequality:*

$$\lceil |\mathcal{A}_\pi^p|/3 \rceil \leq MinSort_{Gn^3}(\pi) \leq |\mathcal{A}_\pi^p|.$$

The above theorem implies an algorithm with a trivial performance guarantee of 3. Now, Algorithm GreedyHops in the previous section, even though meant for sorting by short block-moves,

provides a tighter upper bound for short generator distance. In fact, the same algorithm can be used to prove an approximation guarantee better than 3.

Theorem 6.28 *Algorithm* GreedyHops *sorts a permutation π with a sequence of short generators whose length is within twice the optimal.*

Proof: A similar analysis applies, noting that this time, 3-reversals are allowed so that

$$\begin{aligned} MinSort_{Gn^3}(\pi) &\geq \lceil (|\mathcal{A}_\pi^p| - r)/3 \rceil + r \\ &\geq (2h+s)/3 + r. \end{aligned}$$

That is, the arcs in $\mathcal{A}_\pi^p$ that are not lone arcs can at best be eliminated by a 3-reversal. Recall that

$$GreedyHops(\pi) = h + s + r.$$

If $s = 0$ and $h = 0$, then $GreedyHops(\pi) = MinSort_{Gn^3}(\pi)$. If $s = 0$ and $h > 0$ we have

$$\begin{aligned} \frac{GreedyHops(\pi)}{MinSort_{Gn^3}(\pi)} &\leq \frac{h+r}{(2h/3)+r} \\ &\leq \frac{h}{(2h/3)} \\ &= \frac{3}{2}. \end{aligned}$$

However, if $s > 0$, we arrive at the following approximation bound.

$$\begin{aligned} \frac{GreedyHops(\pi)}{MinSort_{Gn^3}(\pi)} &\leq \frac{h+s+r}{(2h+s)/3+r} \\ &\leq \frac{h+s}{(2h+s)/3} \\ &\leq \frac{s+s}{(2s+s)/3} \quad \text{since } h \geq s \\ &= 2. \end{aligned}$$

□

Finally, bounds for short generator diameter are also immediately available.

Theorem 6.29 *The short generator diameter of S_n satisfies the inequality:*

$$\left\lceil \binom{n}{2}/3 \right\rceil \leq Diam_{Gn^3}(n) \leq \left\lceil \binom{n}{2}/2 \right\rceil.$$

Proof: The lower bound is from Theorem 6.27 and the the fact that $|\mathcal{A}^p_{\boldsymbol{\delta}}| = \binom{n}{2}$. The upper bound is the maximum possible value returned by Algorithm LeftSort in Section 6.7. $\square$

Chapter 7

Block-Moves and the LIS

In this chapter, we look into the potential of the longest increasing subsequence (LIS) of a permutation as a basis for minimum sorting. Sorting by block-moves and its bounded variants are of particular interest. For instance, Guyer, Heath, and Vergara [16] show experimentally that, for the currently unresolved problem of sorting by unbounded block-moves, the heuristic based on longest increasing subsequences appears to produce near-optimal results.

In Section 7.1, we study $MinSort_{Bk^{1,n-1}}$, sorting by *single element insertions*. In this problem, a block-move is allowed if the length of one of blocks moved is 1. We provide a polynomial-time algorithm for $MinSort_{Bk^{1,n-1}}$ that is based on an LIS of the permutation. In Section 7.2, we show how the notions of insert position and insert distance described in the previous chapter relate to this algorithm. Section 7.3 uses insert positions and distances to show that, when seeking an optimal solution, single element insertions can always be chosen such that inversions are never introduced. This statement, in turn, motivates Section 7.4, where we devise an LIS-based heuristic for $MinSort_{Bk^3}$, a problem that is in fact a bounded variant of sorting by single element insertions.

7.1 Single Element Insertions

Single element insertions ($Bk^{1,n-1}$) are just block-moves that reposition a single element anywhere in the permutation. The following theorem shows that sorting by such block-moves is solvable in polynomial time.

SINGLEINSERTSORT(π). Sorting by single element re-insertions.

INPUT: A permutation π.
OUTPUT: The shortest sorting sequence of single element re-insertions.

```
1   L ← the longest increasing subsequence of π
2   D ← the set of elements in π not in L
3   k ← 0
4   while |D| ≠ 0
5         do a ← an element in D
6            k ← k + 1
7            β_k ← the block-move β that inserts a in the LIS
8            π ← π · β_k
9            D ← D − a
10  return β_1, β_2, ..., β_k
```

Figure 7.1: Algorithm SINGLEINSERTSORT.

Theorem 7.1 *$MinSort_{Bk^{1,n-1}}$ is solvable in $O(n^2)$ time.*

Proof: Let π be a permutation and consider an LIS in π. Observe that a block-move from $Bk^{1,n-1}[n]$ may increase the length of the LIS in the permutation by at most one. The sorted permutation has an LIS of length n. Therefore, a straightforward algorithm that optimally sorts π using generators from $Bk^{1,n-1}[n]$ simply inserts all the elements not in the LIS in their proper positions. Algorithm SINGLEINSERTSORT given in Figure 7.1 returns an optimal sorting sequence of single element insertions for a permutation π. The algorithm is dominated not by the computation of the LIS (line 1) but by the insertion of elements (lines 7 and 8). Determining a block-move that inserts an element and then applying that block-move to the permutation takes $O(n)$ time[1] (reminiscent of the familiar *insertion-sort* algorithm). Since there are $O(n)$ insertions, the algorithm takes $O(n^2)$ time. □

The above theorem implies that

$$MinSort_{Bk^{1,n-1}}(\pi) = n - |LIS(\pi)|,$$

[1] A more sophisticated process (the discussion of which is beyond the scope of this dissertation) involves AVL trees [1] and reduces this step to $O(\log n)$ time. This yields a resulting time complexity of $O(n \log n)$.

where $|LIS(\pi)|$ is the length of an LIS in π. If all that is required is this value, then only the computation of the LIS is necessary, which takes $O(n \log\log n)$ time [24]. Also, $|LIS(\boldsymbol{\delta}[n])| = 1$, the shortest possible, so that $MinSort_{Bk^{1,n-1}}(\boldsymbol{\delta}[n]) = n - 1$, giving us the following diameter result.

Theorem 7.2 *The single element insertion diameter of S_n satisfies*

$$Diam_{Bk^{1,n-1}}(n) = n - 1.$$

7.2 $MinSort_{Bk^{1,n-1}}$ and Insert Positions

We have not prescribed in the previous section how the indices for each β_i are determined (line 7 of the algorithm). Of course, a straightforward scan of both π and the LIS is sufficient to obtain the necessary indices to specify the block-move that inserts an element in the LIS. However, an alternate method is to use the notion of insert positions and insert distances as defined in Section 6.5. For convenience, we redefine them here in the context of a longest increasing subsequence. First, we obtain a 2-decomposition of π such that the first component L_π is an LIS and the second component D_π comprises the rest of the elements in π. Suppose that an LIS is

$$L_\pi = \pi(i_1)\ \pi(i_2) \ldots \pi(i_r).$$

For an element $\mathbf{a} = \pi(p)$ in D_π, we compare the position p with the positions $i_1, i_2, \ldots, i_r$; that is, we find the position i_x such that

$$i_1 < i_2 \cdots < i_x < p < i_{x+1} < \cdots < i_r.$$

The insert position $insertpos(\pi, D_\pi, \mathbf{a})$ is then determined as follows.

- If $\pi(i_x + 1) < \pi(p)$ then $insertpos(\pi, D_\pi, \mathbf{a}) = i_y$ where

$$\pi(i_1) < \pi(i_2) < \cdots < \pi(i_y) < \pi(p) < \pi(i_{y+1}) < \cdots < \pi(i_r);$$

- if $\pi(p) < \pi(i_x)$ then $insertpos(\pi, D_\pi, \mathbf{a}) = i_y$ where

$$\pi(i_1) < \pi(i_2) < \cdots < \pi(i_{y-1}) < \pi(p) < \pi(i_y) < \cdots < \pi(i_r).$$

The insert distance $insertdist(\pi, D_\pi, \mathbf{a})$, on the other hand, is computed as $insertpos(\pi, D_\pi, \mathbf{a}) - p$. This value is positive in the first case given above where $\mathbf{a}$ needs to move to the right, and negative

in the second case where **a** needs to move to the left. From insert positions and distances, we can derive the appropriate indices for a block-move β that inserts **a** in the LIS. More precisely, in the permutation π, let $\mathbf{a} = \pi(i)$ be an element such that $p = insertpos(\pi, D_\pi, \mathbf{a})$ and $d = insertdist(\pi, D_\pi, \mathbf{a})$. The block-move β for the element **a** is

- $\beta\langle i, i+1, p+1\rangle$, if $d > 0$;
- $\beta\langle p, i, i+1\rangle$, if $d < 0$.

For example, consider the permutation

$$\mathbf{3\ 12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{4\ 13}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \mathbf{6}.$$

whose LIS elements are shown in boxes. The insert positions and distances of the elements not in the LIS are given below.

Element	Insert Position	Insert Distance
3	4	3
12	12	10
4	5	-4
13	12	2
6	6	-7

Inserting $\pi(9) = \mathbf{4}$ in the LIS is performed by the block-move $\beta\langle 5, 9, 10\rangle$.

Insert positions and distances are dynamic so that, in an algorithm, the above table would need to be maintained as block-moves are applied to π. Initially computing such a table takes $O(n^2)$ time as it entails a scan of the permutation per element. Maintaining the table, on the other hand requires $O(n)$ time per block-move applied (inserting an element may for example cause all of the insert distances and positions to change by 1).

There is therefore little computational benefit in using such a table in Algorithm SingleInsertSort in lieu of just scanning the permutation and the LIS. However, insert distances allow us to prove a property for $MinSort_{Bk^{1,n-1}}$ (Section 7.3). This property, in turn, suggests a corresponding heuristic for $MinSort_{Bk^3}$ (Section 7.4).

7.3 $MinSort_{Bk^{1,n-1}}$ and Relative Order

For $MinSort_{Bk^{1,n-1}}$, we show that, in determining an optimal solution, single element insertions can always be chosen such that they never introduce inversions. Consider lines 5 and 7 of Algorithm

SingleInsertSort. An element and then a corresponding block-move is selected. The proof of the following theorem suggests how this element can be chosen.

Theorem 7.3 *For an unsorted permutation π, there exists a correcting block-move $\beta \in Bk^{1,n-1}[n]$ that lengthens an LIS of π.*

Proof: Let L_π be an LIS and let D_π the set of elements in π that is not in L_π. We select an element $\mathbf{a}$ in D_π as follows

(1) If there exists at least one element with positive insert distance, then, from among such elements, select the element $\mathbf{a}$ that appears rightmost in the permutation.

(2) Otherwise, all elements have negative insert distances. Select the element $\mathbf{a}$ in D_π such that it appears leftmost in the permutation.

We claim that the single element insertion β that inserts $\mathbf{a}$ in the LIS of π is a correcting block-move. Let $\mathbf{a} = \pi(i)$ and let p_a and d_a be the insert position and insert distance of $\mathbf{a}$, respectively.

If **(1)** holds, then the block-move β satisfies $\beta = \beta\langle i, i+1, p_a+1\rangle$. It suffices to show that the elements to the immediate right of $\mathbf{a}$,

$$\pi(i+1)\ \pi(i+2)\ \ldots\ \pi(p_a),$$

are all less than $\mathbf{a}$ (that is, $\mathbf{a}$ is right d_a-pending). Suppose there exists an element $\mathbf{b} = \pi(j) > \mathbf{a}$ among these elements. The element $\mathbf{b}$ cannot be in L_π (the LIS) since the definition of insert position requires that $\mathbf{b} > \mathbf{a}$. Thus, $\mathbf{b}$ is in D_π. Now, since $\mathbf{b} > \mathbf{a}$, its insert position is $p_b \geq p_a$. But $j < p_a$, so the insert distance of $\mathbf{b}$ is $p_b - q$ and therefore positive. A contradiction results since $\mathbf{a}$ is the rightmost element with positive insert distance and $\mathbf{b}$ occurs to the right of $\mathbf{a}$. It follows that β is a correcting block-move.

If **(2)** holds, then the block-move β satisfies $\beta = \beta\langle p_a, i, i+1\rangle$. It suffices to show that the elements to the immediate left of $\mathbf{a}$,

$$\pi(p_a)\ \pi(p_a+1)\ \ldots \pi(i-1),$$

are all greater than $\mathbf{a}$ (that is, $\mathbf{a}$ is left $|d_a|$-pending). An argument symmetric to **(1)** then holds. It follows that β is a correcting block-move. The theorem follows. $\square$

7.4 Sorting by Short Block-Moves Using the LIS

Sorting by short block-moves ($MinSort_{Bk^3}$) is just single element insertion, where an element may not be inserted more than two positions away from its original position. In this section, we use Theorem 7.3 to devise an LIS-based heuristic for $MinSort_{Bk^3}$.

Consider an unsorted permutation π and let L_π be an LIS in π. Let D_π be the elements in π that are not in L_π. An element $\mathbf{a} \in D_\pi$ is *LIS-ready* if the block-move β that inserts $\mathbf{a}$ in the LIS is a correcting block-move. Theorem 7.3 implies that there is at least one such element. For example, in the permutation

$$\pi = \mathbf{3\ 12}\ \boxed{\mathbf{1}}\ \boxed{\mathbf{2}}\ \boxed{\mathbf{5}}\ \boxed{\mathbf{7}}\ \boxed{\mathbf{8}}\ \boxed{\mathbf{9}}\ \mathbf{4\ 13\ 14}\ \boxed{\mathbf{10}}\ \boxed{\mathbf{11}}\ \mathbf{6},$$

where the elements of L_π are shown in boxes, $D_\pi = \{\mathbf{3}, \mathbf{4}, \mathbf{6}, \mathbf{12}, \mathbf{13}, \mathbf{14}\}$. The elements **3**, **4**, and **14** in π are LIS-ready while **6**, **12** and **13** are not. Let $\mathbf{a}$ and $\mathbf{b}$ be elements in D_π that are in their correct relative order in π and suppose $\mathbf{a}$ is not LIS-ready. The element $\mathbf{a}$ is *blocked* by $\mathbf{b}$ if the block-move that inserts $\mathbf{a}$ in the LIS places $\mathbf{a}$ and $\mathbf{b}$ out of order; the element $\mathbf{b}$ is called a *blocker* of $\mathbf{a}$. In the above example, **6** is blocked by **4**, **12** is blocked by the elements **13** and **14**, and **13** is blocked by **14**. Again, suppose that $\mathbf{a}$ is an element that is not LIS-ready. The element $\mathbf{a}$ is a *waiting* element if it has exactly one blocker $\mathbf{b}$ and $\mathbf{b}$ is LIS-ready; in this case, $\mathbf{a}$ is an *exclusive dependent* of $\mathbf{b}$. For example, elements **6** and **13** are waiting while **12** is not. The element **6** is an exclusive dependent of **4** and the element **13** is an exclusive dependent of **14**.

We are now ready to describe our algorithm. Algorithm LISSort, shown in Figure 7.2, takes as input a permutation π and returns the length of a sorting sequence of short block-moves. The algorithm is similar to BitonicSort in Section 6.5 in that it bases its block-move selection on parities and directions of the insert distances. First, the algorithm computes an LIS L_π (line 1) and D_π, the set of elements not in L_π (line 2). Then, in a **while** loop (lines 4–22), it distinguishes between several cases. If D_π contains an LIS-ready element with even insert distance, then InsertEven applies (lines 8–10). Otherwise, if two LIS-ready elements in D_π exist that have odd insert distances and go in the same direction, then InsertSameSign applies (lines 11–13). Otherwise, if D_π contains an LIS-ready element with an exclusive dependent with odd insert distance, then InsertSameSign also applies (lines 13–16). Otherwise, a skip is spent; InsertOdd is called with the LIS-ready element that is farthest from its insert position (lines 18–21). The condition in the last case maximizes the chance of shifting the parities of other insert distances.

LISSort(π)**.** Sorting by short block-moves based on the LIS.

INPUT: A permutation π.
OUTPUT: The length of a sorting sequence of short block-moves.

```
L_π ← the longest increasing subsequence of π
D_π ← the set of elements in π not in L
dist ← 0
while π ≠ ι
        do a_1, a_2, ..., a_ℓ ← the LIS-ready elements in D_π
            let p_1, p_2, ..., p_ℓ be the positions of a_1, a_2, ..., a_ℓ
            let d_1, d_2, ..., d_ℓ be the insert distances of a_1, a_2, ..., a_ℓ
            switch case there exists an a_i such that d_i is even
                        do (π, d) ← InsertEven(π, a_i, d_i)
                        D_π ← D_π − a_i
                    case there exists a_i, a_j such that p_i > p_j and d_i d_j > 0
                        do (π, d) ← InsertSameSign(π, a_i, d_i, a_j, d_j)
                        D_π ← D_π − a_i − a_j
                    case there exists an a_i with exclusive dependent b
                                such that insert the distance e of b is odd
                        do (π, d) ← InsertSameSign(π, a_i, d_i, b, e)
                        D_π ← D_π − a_i − b
                    otherwise
                        let a_i be the element such that |d_i| is maximum
                        do (π, d) ← InsertOdd(π, a_i, d_i)
                        D_π ← D_π − a_i
            dist ← dist + d
return dist
```

Figure 7.2: Algorithm LISSort.

Theorem 7.4 *Algorithm* LISSort *sorts a permutation* π *in* $O(n^3)$.

Proof: Every iteration inserts at least one element from D_π in the LIS, so the algorithm eventually sorts π. For the time complexity, the initial computations of insert positions and distances are performed in $O(n^2)$ time outside the **while** loop. As there are $O(n)$ elements to be inserted, the loop performs $O(n)$ iterations. The actual insertion of an element takes $O(n)$ short block-moves. The maintenance of the insert positions and distances take $O(n)$ time. However, identifying LIS-ready elements and its exclusive dependents takes $O(n^2)$ time since we need to perform an $O(n)$ test for $O(n)$ elements at every iteration. It follows that the algorithm runs in $O(n^3)$ time. □

In Chapter 9, we provide an example where this heuristic fails. We also evaluate the heuristic and compare it to others.

Chapter 8

Short Swaps (Short Reversals)

In this chapter, we present results for these short swap problems: $MinSort_{Sw^3}$ and $Diam_{Sw^3}$. Note that short swaps and short reversals are identical. In Section 8.1, we devise an algorithm that greedily selects a short swap that causes the largest reduction in inversions. We then establish distance and diameter bounds based on this algorithm. Section 8.2 introduces the notion of a vector diagram, the model we use in the next sections. In Section 8.3, an algorithm for sorting by short swaps is devised based on this model, and bounds for short swap distance are derived. Section 8.4 determines when the lower bound for short swap distance is achieved. In Section 8.5, bounds for short swap diameter are derived. Finally, in Section 8.6, we extend the results obtained for sorting by short swaps to sorting by ℓ-bounded swaps.

8.1 Correcting Inversions

In this section, we devise an algorithm that computes short swap distance based on inversions.

Theorem 5.3 guarantees that correcting swaps are sufficient when determining an optimal sorting sequence of short swaps. A correcting swap reduces the number of inversions in a permutation by either 1 or 3. A reduction of 3 inversions is possible only through a 3-swap. Define a *triple* in a permutation as a decreasing subsequence of three contiguous elements. Observe that the 3-swap that swaps the first and third elements of a triple removes 3 inversions in the permutation. Let $triples(\pi)$ be the set of triples in a permutation π (there can be at most $n-2$ of them).

Algorithm GreedyInversions, shown in Figure 8.1, repeatedly selects a swap that corrects the

GreedyInversions(π) Sorting by short swaps based on inversions corrected.

INPUT: A permutation π.
OUTPUT: A sorting sequence of short swaps for π

```
1   dist ← 0
2   while π ≠ ι
3         do if |triples(π)| > 0   ▷ if π contains triples (see text)
4               then ▷ select an appropriate triple and a corresponding swap
5                    a b c ← be the triple in triples(π)  such that
6                              |triples(π · S⟨a, c⟩)| is maximum
7                    σ ← S⟨a, c⟩
8               else  let Sw³[n]* ⊆ Sw³[n] be the set of correcting swaps for π
9                    σ ← the swap σ ∈ Sw³[n]* such that |triples(π · σ)| is maximum
10            π ← π · σ
11            dist ← dist + 1
12  return dist
```

Figure 8.1: Algorithm GreedyInversions.

most inversions. Since, in general, we would like as many triples as possible in the permutation, we break ties by selecting the swap that results in the permutation with the most triples. The algorithm therefore first checks for $|triples(\pi)| > 0$ to find out whether short swaps that remove 3 inversions exist (line 3). If this is the case, then a triple **a b c** from $triples(\pi)$ is selected such that swapping **a** and **c** results in a permutation that contains the most triples (lines 4–7). Otherwise, a swap is chosen from among the correcting swaps, particularly the one that results in the most triples (line 8–9).

Theorem 8.1 *Algorithm* GreedyInversions *sorts a permutation in $O(n^3)$ time.*

Proof: The permutation π is eventually sorted by the algorithm since the number of inversions in π strictly reduces at every iteration. For the time complexity, there are $O(n^2)$ inversions in a permutation which implies $O(n^2)$ iterations for the **while** loop. Each iteration, in turn, distinguishes between $O(n)$ swaps. The value $|triples(\pi \cdot \sigma)|$ for each σ can be computed in constant time if $triples(\pi)$ is available. Thus, $triples(\pi)$ can be computed in $O(n)$ time at the beginning of every iteration (and is in fact used in the test for the **while** loop) so that the body of the loop runs in $O(n)$ time. It follows that Algorithm GreedyInversions runs in $O(n^3)$ time. □

In Chapter 9, we implement Algorithm GreedyInversions and evaluate its performance. The following straightforward bounds for short swap distance apply to the above algorithm. Recall that the number of arcs in permutation graph $\mathcal{G}_\pi^p = (\mathcal{V}_\pi^p, \mathcal{A}_\pi^p)$ is exactly the number of inversions in π.

Theorem 8.2 *The short swap distance of a permutation π satisfies the inequality:*

$$\lceil |\mathcal{A}_\pi^p|/3 \rceil \leq MinSort_{Sw^3}(\pi) \leq |\mathcal{A}_\pi^p|.$$

The following corollary is a direct result of the above inequality.

Corollary 8.3 *Algorithm* GreedyInversions *sorts a permutation with a sequence of short swaps whose length is within 3 times the optimal.*

Inversions also provide us with a lower bound for short swap diameter.

Theorem 8.4 *The short swap diameter of S_n satisfies the inequality:*

$$Diam_{Sw^3}(n) \geq \left\lceil \binom{n}{2}/3 \right\rceil.$$

Proof: Recall that the decreasing permutation $\boldsymbol{\delta}$ has $\binom{n}{2}$ inversions, the maximum possible. By Theorem 8.2, $\lceil \binom{n}{2}/3 \rceil$ swaps are necessary to sort $\boldsymbol{\delta}$. The theorem follows. □

8.2 Vector Diagrams

The previous section uses inversions as a measure of the amount of work necessary to sort a permutation. In this section, we use a different measure. An element in an unsorted permutation has to "travel" to its correct position as it is sorted and short swaps limit the distance that an element can travel in a single step.

Let π be a permutation π of length n. The *destination graph* of π is the directed graph $\mathcal{G}_\pi^d = (\mathcal{V}_\pi^d, \mathcal{A}_\pi^d)$ where $\mathcal{V}_\pi^d = \{\pi(1), \pi(2), \ldots, \pi(n)\}$ and $(\pi(i), \pi(j)) \in \mathcal{A}_\pi^d$ whenever $\pi(i)$ is unpositioned and $\pi(i) = j$. Figure 8.2 illustrates the destination graph for the permutation **3 4 6 1 5 2**. An arc in $\mathcal{G}_\pi^d$ indicates where an unpositioned element goes to in the sorted permutation. The arcs in Figure 8.2 are labeled with the distance of an element from its destination which is $|\pi(i) - i|$. An alternate representation that includes this measure is a *vector diagram* V_π of π. In the vector diagram V_π,

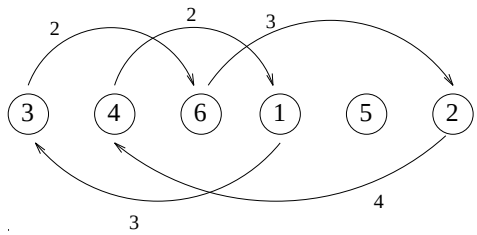

Figure 8.2: The destination graph of a permutation.

Figure 8.3: Vector diagrams for (a) $\pi = \mathbf{3\ 4\ 6\ 1\ 5\ 2}$, and (b) $\pi \cdot S\langle \mathbf{6}, \mathbf{1} \rangle$.

there is a *vector* $v_\pi(\mathbf{a})$ of length $|\pi(i) - i|$ for each element $\mathbf{a} = \pi(i)$. If $\mathbf{a}$ is an unpositioned element, its vector $v_\pi(\mathbf{a})$ has a direction indicated by the sign of $\pi(i) - i$. The vector $v_\pi(\mathbf{a})$ is a *right vector* if $\pi(i) - i > 0$ while it is a *left vector* if $\pi(i) - i < 0$. If $\mathbf{a}$ is in its correct position, $v_\pi(\mathbf{a})$ is a *zero vector*. In a vector diagram V_π, only non-zero vectors are drawn and they are drawn with the permutation π such that the endpoint of a vector is aligned with its corresponding element; that is, if $\mathbf{a} = \pi(i)$, then the vector $v_\pi(\mathbf{a})$ is drawn such that it begins at position i and ends at position $\mathbf{a}$. Figure 8.3(a) illustrates V_π for $\pi = \mathbf{3\ 4\ 6\ 1\ 5\ 2}$. In the figure, π has 5 non-zero vectors consisting of 3 right vectors and 2 left vectors. We denote the length of a vector $v_\pi(\mathbf{a})$ by $|v_\pi(\mathbf{a})|$ and the total length of the vectors in V_π by $|V_\pi|$. In Figure 8.3(a), for example, $|v_\pi(\mathbf{6})| = 3$ and $|V_\pi| = 14$.

For $\pi = \mathbf{3\ 4\ 6\ 1\ 5\ 2}$, consider the swap $S\langle \mathbf{6}, \mathbf{1} \rangle$. The vector diagram for $\pi \cdot S\langle \mathbf{6}, \mathbf{1} \rangle = \mathbf{3\ 4\ 1\ 6\ 5\ 2}$ is shown in Figure 8.3(b). Comparing this with the vector diagram in Figure 8.3(a) indicates a possible effect a swap makes on the vector diagram of a permutation: two vectors are shortened (in this case, $v_\pi(\mathbf{6})$ and $v_\pi(\mathbf{1})$). This observation suggests monitoring the lengths of these vectors as a sequence of swaps is applied to π. By dividing the initial vectors into *segments*, we may assign each

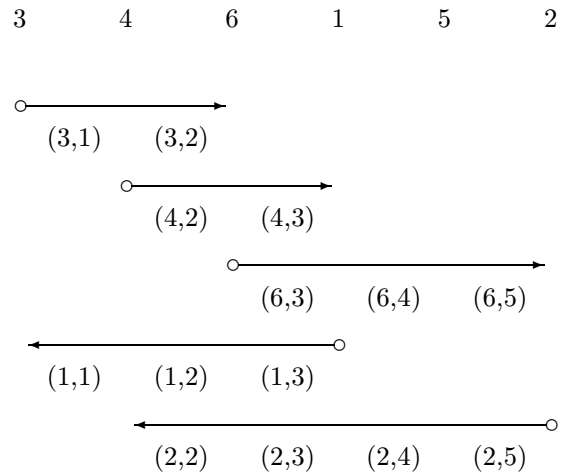

Figure 8.4: A vector diagram with labeled segments.

swap to the vector segments it eliminates. Note that the vector diagram of an identity permutation has a total vector length of zero meaning a sequence of swaps should eventually eliminate all the segments of the initial vectors.

We refine the vector diagram by identifying and labeling the segments of a vector. For each vector $v_\pi(\mathbf{a})$ of length k, there are k vector segments that are labeled

$$(\mathbf{a}, p), (\mathbf{a}, p+1), \ldots, (\mathbf{a}, p+k-1).$$

The coordinate q in the vector segment $(\mathbf{a}, q)$ indicates the position (between 1 and $n-1$) of the segment assuming the vector is aligned with the permutation. The value of p above depends on i, where $\mathbf{a} = \pi(i)$, and on the direction of the vector; that is,

$$p = \begin{cases} i & \text{if } v_\pi(\mathbf{a}) \text{ is a right vector,} \\ i-k & \text{if } v_\pi(\mathbf{a}) \text{ is a left vector.} \end{cases}$$

Figure 8.4 illustrates the labeling of the vector segments for $\pi = \mathbf{3\ 4\ 6\ 1\ 5\ 2}$. With this refined vector diagram, the swap $S\langle \mathbf{6}, \mathbf{1} \rangle$ corresponds to the elimination of vector segments $(\mathbf{6}, 3)$ and $(\mathbf{1}, 3)$.

Of course, not all swaps correspond to the elimination of vector segments. However, Theorem 5.6 implies that in seeking an optimal set of swaps that sort a permutation, one may consider only correcting swaps. That is, it may be assumed that a swap $\sigma\langle i, j \rangle$ on a permutation π satisfies the

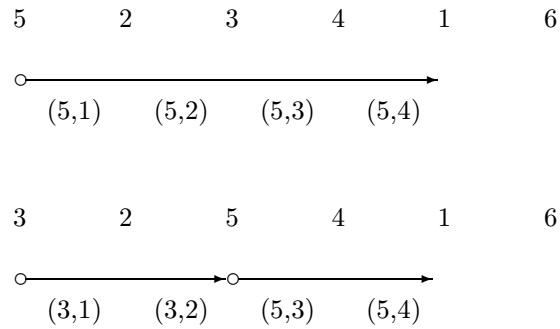

Figure 8.5: A cut operation (swap is $S\langle \mathbf{5}, \mathbf{3} \rangle$).

conditions $i < j$ and $\pi(i) > \pi(j)$. Under this assumption, the effect of a swap on the vector diagram of a permutation is now limited to four cases, enumerated as follows.

1. A short swap $\sigma\langle i, j \rangle$ is a *cancellation* if $\pi(j) \leq i$ and $j \leq \pi(i)$. Segments from vectors that go in opposite directions are eliminated, as in the example in Figure 8.3.

2. A short swap $\sigma\langle i, j \rangle$ is a *cut* if $\pi(j) = j$ or $\pi(i) = i$. Here, a vector is shortened and a new vector is added. In Figure 8.5, the swap $S\langle \mathbf{5}, \mathbf{3} \rangle$ applied to the permutation **5 2 3 4 1 6** causes $v_\pi(\mathbf{5})$ to be shortened but a new vector $v_\pi(\mathbf{3})$ of length 2 results.

3. A short swap $\sigma\langle i, j \rangle$ is a *switch* if $\pi(j) < i$ and $\pi(i) < i$ or $\pi(j) > j$ and $\pi(i) > j$. In this case, one vector is shortened and the other is lengthened. Figure 8.6 shows an example of a switch. Here, the swap $S\langle \mathbf{4}, \mathbf{1} \rangle$ is applied to the permutation **6 2 5 3 4 1**. Notice that $v_\pi(\mathbf{4})$ and $v_\pi(\mathbf{1})$ are lengthened and shortened, respectively.

4. A short swap $\sigma\langle i, i+2 \rangle$ is a *flip* if if either $\pi(i)$ or $\pi(i+2)$ equals $i+1$ (the correct position of one of the elements occurs between position i and j). Notice that a flip *must* be a 3-swap. In this case, a vector is shortened by two segments and a vector of length 1 flips in direction. In Figure 8.7, the permutation **6 4 2 3 5 1** has the vector $v_\pi(\mathbf{2})$ of length 1 going opposite vector $v_\pi(\mathbf{6})$. Once the swap $S\langle \mathbf{6}, \mathbf{2} \rangle$ ($\sigma\langle 1, 3 \rangle$) is performed, $v_\pi(\mathbf{6})$ is shortened and $v_\pi(\mathbf{2})$ changes in direction.

It can be verified, by considering the cases where the correct positions of elements $\pi(i)$ and $\pi(j)$ are,

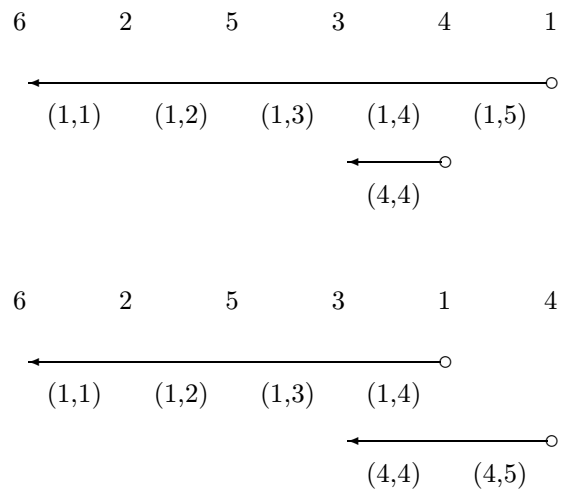

Figure 8.6: A switch operation (swap is $S\langle \mathbf{4}, \mathbf{1} \rangle$).

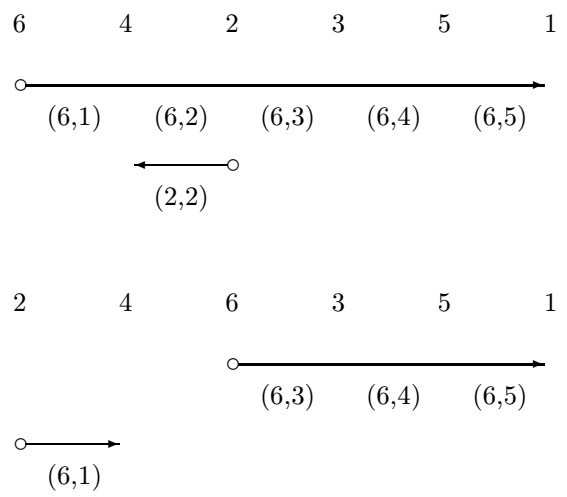

Figure 8.7: A flip operation (swap is $\sigma\langle 1, 3 \rangle$).

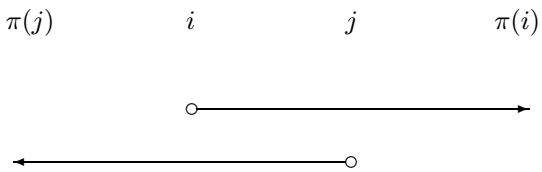

Figure 8.8: Vector-opposite elements $\pi(i)$ and $\pi(j)$.

that these are the only possibilities for correcting short swaps. Observe that the total vector length $|V_\pi|$ never increases upon application of any of these types of correcting short swaps.

In the next section, we devise an algorithm that optimally sorts a permutation by short swaps that are cancellations and cuts, the first two types indicated above. In Chapter 9, we give examples of permutations where the other two types (switches and flips) are necessary in an optimal sequence.

8.3 Bounds for Short Swap Distance

Our objective is to reduce $|V_\pi|$, the total number of vector segments for a permutation, by as much as possible. A cancellation, in particular, causes a reduction of 2 or 4 segments. In this section, we present an algorithm that allows us at least an average net decrease of 2 vector segments per short swap performed.

Two elements $\mathbf{a} = \pi(i)$ and $\mathbf{b} = \pi(j)$ in π, with $i < j$, are *vector-opposite* if their vectors $v_\pi(\mathbf{a})$ and $v_\pi(\mathbf{b})$ in V_π are such that $|v_\pi(\mathbf{a})| \geq j - i$, $|v_\pi(\mathbf{b})| \geq j - i$, and $v_\pi(\mathbf{a})$ and $v_\pi(\mathbf{b})$ differ in direction. For example, in the permutation **3 4 6 1 2 5**, the elements **6** and **1** are vector-opposite elements as illustrated by $v_\pi(\mathbf{6})$ and $v_\pi(\mathbf{1})$ in Figure 8.4. Notice that **4** and **2** are not vector-opposite even if their vectors go in opposite directions because $v_\pi(\mathbf{4})$ is shorter than the distance between **4** and **2**. An alternative characterization for vector-opposite elements $\pi(i)$ and $\pi(j)$ is when $\pi(j) \leq i < j \leq \pi(i)$, as illustrated in Figure 8.8. A cancellation, in particular, swaps two vector-opposite elements. Finally, it is sometimes useful to specify the distance between vector-opposite elements so we define *m-vector-opposite* elements as vector-opposite elements $\pi(i)$ and $\pi(j)$ where $m = j - i$.

Lemma 8.5 *Let π be an unsorted permutation. Then, π contains at least one pair of vector-opposite elements.*

Proof: Let $\mathbf{a}_r = \pi(i_r)$ be the largest unpositioned element in π. We first show by contradiction that $v_\pi(\mathbf{a}_r)$ is a right vector. Suppose $v_\pi(\mathbf{a}_r)$ is a left vector. Then, $\pi(i_r) - i_r < 0$ so that $i_r > \pi(i_r) = \mathbf{a}_r$. We find that $\mathbf{i}_r$ is a larger unpositioned element, a contradiction.

Now since there is at least one right vector in V_π, there exists a rightmost right vector in V_π, that is, a right vector $v_\pi(\mathbf{a})$ where $\mathbf{a} = \pi(i)$ and i is as large as possible.

The element $\mathbf{b}_\ell = \pi(a)$ is unpositioned since it occupies $\mathbf{a}$'s correct position. The vector $v_\pi(\mathbf{b}_\ell)$ is therefore a left vector as it occurs to the right of $v_\pi(\mathbf{a})$, the rightmost right vector.

Consider the positions $i+1, i+2, \ldots, a$ and the elements that occupy them: $\pi(i+1), \pi(i+2), \ldots, \mathbf{b}_\ell$. At least one of these elements ($\mathbf{b}_\ell$) corresponds to a left vector. Select the leftmost left vector from these elements; that is, let $v_\pi(\mathbf{b})$ be the left vector where $\mathbf{b} = \pi(j)$, $i+1 \leq j \leq a$, and j is as small as possible.

We claim that $\mathbf{a}$ and $\mathbf{b}$ are vector-opposite elements. Since $j \leq \mathbf{a} = \pi(i)$, all that remains to be shown is that $\mathbf{b} = \pi(j) \leq i$. In words, we need to show that the correct position of element $\mathbf{b}$ does not occur to the right of position i. To arrive at a contradiction, suppose this was the case. Then the element occupying position $\mathbf{b}$, say, $\mathbf{c} = \pi(b)$ is unpositioned and therefore corresponds to either a right or left vector $v_\pi(\mathbf{c})$. It is not a right vector since it occurs on the right of $v_\pi(\mathbf{a})$, the rightmost right vector. It is not a left vector since it occurs on the left of $v_\pi(\mathbf{b})$, the leftmost left vector from a set that includes $v_\pi(\mathbf{c})$. A contradiction results since we have found an unpositioned element that does not have a corresponding vector. The lemma follows. □

Algorithm FindOpposite, shown in Figure 8.9, returns the positions of a pair of vector-opposite elements in an unsorted permutation π. First, the element $\pi(i)$ with the rightmost right vector is located (lines 1-3). Then, the element $\pi(j)$ with the leftmost left vector among the elements to the right of $\pi(i)$ is located (lines 4–6). Finally, the algorithm returns i and j, the positions of these elements (line 7). As the algorithm involves two straightforward linear scans of the permutation, it should be clear that it runs in $O(n)$ time.

Detecting vector-opposite elements is useful because swapping these elements yields a predictable effect on V_π and thus a predictable decrease in $|V_\pi|$.

FINDOPPOSITE(π). Finding two vector opposite elements in a permutation.

INPUT: An unsorted permutation π.
OUTPUT: Positions i, j such that $\pi(j) \leq i < j \leq \pi(i)$.

```
i ← n
while π(i) ≤ i   ▷ find rightmost right vector
        do i ← i − 1
j ← i + 1
while π(j) = j   ▷ find leftmost left vector to its right
        do j ← j + 1
return (i, j)
```

Figure 8.9: Algorithm FINDOPPOSITE.

Lemma 8.6 *Let π be an unsorted permutation and let $\mathbf{a}$ and $\mathbf{b}$ be m-vector-opposite elements in π. Let $\pi' = \pi \cdot S\langle \mathbf{a}, \mathbf{b} \rangle$. Then, for all elements $\mathbf{c}$ in π,*

$$|v_\pi(\mathbf{c})| - |v_{\pi'}(\mathbf{c})| = \begin{cases} m & \text{if } \mathbf{c} \in \{\mathbf{a}, \mathbf{b}\} \\ 0 & \text{otherwise.} \end{cases}$$

Proof: Let $\mathbf{a} = \pi(i)$, $\mathbf{b} = \pi(j)$ so that $m = j - i$. Since $\mathbf{a}$ and $\mathbf{b}$ are vector-opposite elements, then $\mathbf{b} = \pi(j) \leq i < j < \pi(i) = \mathbf{a}$, where $|v_\pi(\mathbf{a})| = |a - i| = a - i$ and $|v_\pi(\mathbf{b})| = |b - j| = j - b$. Swapping $\mathbf{a}$ and $\mathbf{b}$ causes $\pi'(j) = \mathbf{a}$ and $\pi'(i) = \mathbf{b}$. In $V_{\pi'}$, their vector lengths are now given by $|v_{\pi'}(\mathbf{a})| = a - j$ and $|v_{\pi'}(\mathbf{b})| = i - b$. Thus,

$$|v_\pi(\mathbf{a})| - |v_{\pi'}(\mathbf{a})| = (a - i) - (a - j) = j - i = m,$$

and

$$|v_\pi(\mathbf{b})| - |v_{\pi'}(\mathbf{b})| = (j - a) - (i - b) = j - i = m.$$

Finally, noting that $\pi'(k) = \pi(k)$ for all $k \neq i, j$, the lengths of the other vectors remain unaffected; that is, $|v_\pi(\mathbf{c})| - |v_{\pi'}(\mathbf{c})| = 0$, for all $\mathbf{c} \notin \{\mathbf{a}, \mathbf{b}\}$. □

Lemma 8.7 *Let π be an unsorted permutation and let $\mathbf{a}$ and $\mathbf{b}$ be m-vector-opposite elements in π. Let $\pi' = \pi \cdot S\langle \mathbf{a}, \mathbf{b} \rangle$. Then, $|V_\pi| - |V_{\pi'}| = 2m$.*

Proof: Let $\mathbf{a} = \pi(i)$, $\mathbf{b} = \pi(j)$ so that $m = j - i$. By Lemma 8.6, the vectors associated with $\mathbf{a}$ and $\mathbf{b}$ both decrease in length by m and the length of the other vectors remain the same so that

$$\begin{aligned} |V_\pi| - |V_{\pi'}| &= |v_\pi(\mathbf{a})| - |v_{\pi'}(\mathbf{a})| + |v_\pi(\mathbf{b})| - |v_{\pi'}(\mathbf{b})| \\ &= 2m. \end{aligned}$$

□

If $m \leq 2$, then a single short swap, a cancellation, is sufficient to perform the swap. Otherwise, a series of short swaps is necessary. The following proposition summarizes how this series of swaps may be obtained.

Proposition 8.8 *Let π be a permutation and let $\pi(i)$ and $\pi(j)$ be elements in π, with $i < j$. Let $r_1, r_2, \ldots, r_k$ be distinct positions between i and j, that is, $i < r_1 < r_2 \cdots < r_k < j$. Then,*

$$\pi \cdot \sigma\langle i, r_1\rangle \cdot \sigma\langle r_1, r_2\rangle \cdots \sigma\langle r_{k-1}, r_k\rangle \cdot \sigma\langle r_k, j\rangle \cdot \sigma\langle r_{k-1}, r_k\rangle \cdots \sigma\langle r_1, r_2\rangle \cdot \sigma\langle i, r_1\rangle = \pi \cdot \sigma\langle i, j\rangle.$$

If $r_1 - i \leq 2$, $j - r_k \leq 2$ and $r_\ell - r_{\ell-1} \leq 2$ for all ℓ where $1 < \ell \leq k$, then the swaps specified in the above proposition are all short swaps. In addition, if $\pi(i)$ and $\pi(j)$ are vector-opposite elements obtained by Algorithm FindOpposite, it can be verified that the first $k-1$ short swaps performed are all cuts and the last k short swaps are all cancellations. Algorithm LongSwap, shown in Figure 8.10, performs such a series of short swaps on a permutation to swap two elements.

Lemma 8.9 *Let π be a permutation and let $\pi(i)$ and $\pi(j)$ be m-vector-opposite elements in π. Then, Algorithm* LongSwap *swaps elements $\pi(i)$ and $\pi(j)$ using $m-1$ short swaps if m is even, and m short swaps if m is odd. That is,* LongSwap(π, i, j) *returns (π', d), where $\pi' = \pi \cdot \sigma\langle i, j\rangle$, and*

$$d = \begin{cases} m-1 & \text{if } m \text{ is even} \\ m & \text{if } m \text{ is odd.} \end{cases}$$

Proof: It is clear that the algorithm performs only short swaps by inspecting lines 4, 8, and 12. First suppose m is even. Then the **while** loop in lines 3–6 performs the following $m/2 - 1$ short swaps

$$\sigma\langle i, i+2\rangle, \sigma\langle i+2, i+4\rangle, \ldots, \sigma\langle j-4, j-2\rangle.$$

The **if** statement in lines 7–10 is skipped as m is even and then the **while** loop in lines 11–14 performs the following $m/2$ short swaps

$$\sigma\langle j-2, j\rangle, \sigma\langle j-4, j-2\rangle, \ldots, \sigma\langle i+2, i+4\rangle, \sigma\langle i, i+2\rangle.$$

LongSwap(π, i, j)**.** Swapping two elements using short swaps.

INPUT: A permutation π and positions i, j, with $i < j$.
OUTPUT: The permutation $\pi \cdot \sigma\langle i, j\rangle$ and the number of short swaps performed.

```
k ← i
d ← 0
while k < j − 2
        do π ← π · σ⟨k, k + 2⟩
           k ← k + 2
           d ← d + 1
if j − i is odd   ▷ k = j − 1; extra 2-swap required
   then π ← π · σ⟨k, k + 1⟩
          k ← k − 2
          d ← d + 1
while k ≥ i
        do π ← π · σ⟨k, k + 2⟩
           k ← k − 2
           d ← d + 1
return (π, d)
```

Figure 8.10: Algorithm LongSwap.

VectorSort(π)**.** Sorting by short swaps based on vector-opposite elements.

INPUT: A permutation π.
OUTPUT: The length of a sorting sequence of short swaps.

```
1   dist ← 0
2   while π ≠ ι
3         do (i, j) ← FindOpposite(π)
4            (π, d) ← LongSwap(π, i, j)
5            dist ← dist + d
6   return dist
```

Figure 8.11: Algorithm VectorSort.

It can be verified that the result is a series of swaps described in Proposition 8.8. Hence, the resulting permutation is $\pi' = \pi \cdot \sigma\langle i, j\rangle$ and the number of swaps performed is $d = m/2 - 1 + m/2 = m - 1$.

Now suppose m is odd. Then the **while** loop in lines 3–6 performs the following $(m-1)/2 + 1$ short swaps

$$\sigma\langle i, i+2\rangle, \sigma\langle i+2, i+4\rangle, \ldots, \sigma\langle j-5, j-3\rangle, \sigma\langle j-3, j-1\rangle.$$

At this point, k is $j - 1$ in the algorithm and the **if** statement in lines 7–10 performs the swap $\sigma\langle j-1, j\rangle$. Then, the **while** loop in lines 11–14 performs the following $(m-1)/2$ short swaps

$$\sigma\langle j-3, j-1\rangle, \sigma\langle j-5, j-3\rangle, \ldots, \sigma\langle i+2, i+4\rangle, \sigma\langle i, i+2\rangle.$$

Again, it can be verified that the result is a series of swaps described in Proposition 8.8 so that the resulting permutation is $\pi' = \pi \cdot \sigma\langle i, j\rangle$. This time, the number of swaps performed is $d = (m-1)/2 + 1 + (m-1)/2 = m$. The lemma follows. □

A straightforward algorithm to sort a permutation using short swaps is to repeatedly find vector-opposite elements and then swap the elements using LongSwap. Algorithm VectorSort, shown in Figure 8.11, performs exactly this process. A **while** loop (lines 2–5) that terminates when π is sorted calls FindOpposite to locate vector-opposite elements (line 3) and then calls LongSwap to swap these elements.

Lemma 8.10 *Algorithm* VectorSort *sorts any given permutation π in $O(n^3)$ time.*

Proof: To show that the algorithm sorts π, it suffices to show that the loop terminates. By Lemma 8.5, vector-opposite elements always exist in an unsorted permutation π (line 3). The call to LongSwap (line 4) indeed swaps such elements by Lemma 8.9. Finally, by Lemma 8.7, the swap causes a strict decrease in $|V_\pi|$. The length $|V_\pi|$ therefore eventually becomes zero, precisely when $\pi = \boldsymbol{\iota}$, so it follows that the loop terminates.

For the time complexity, the loop in Algorithm VectorSort takes $O(n^2)$ iterations since there are $O(n^2)$ vector-opposite elements. Algorithm FindOpposite runs in $O(n)$ time so the calls to this procedure costs $O(n^3)$ time. Each call to Algorithm LongSwap, on the other hand, eliminates $2m$ vectors in $O(m)$ time. The cost in this case is bounded by the total length of the vectors rather than the number of iterations performed in the main loop. Noting that each vector is at most of length n, the total vector length is $O(n^2)$ meaning a cost of $O(n^2)$ time for the calls to Algorithm LongSwap. It follows that the algorithm runs in $O(n^3)$ time. □

The time complexity of $O(n^3)$ can be improved to $O(n^2)$ as follows. By supplying more information to FindOpposite, the costs of calls to this procedure can be significantly reduced. Recall that FindOpposite performs two scans of the permutation, one for each vector of the vector-opposite elements returned. By observing that a rightmost right vector remains a rightmost vector until it is eliminated from V_π, it need not be searched again if the vector has not been eliminated. Thus, the scan for the rightmost vector needs to be performed only $O(n)$ times. In addition, the total cost of scans for the leftmost left vector for the same right vector is bounded by the length of the right vector, also $O(n)$. The total cost for all calls to FindOpposite with this refinement is thus $O(n^2)$; consequently, Algorithm VectorSort runs in $O(n^2)$ time.

We now provide upper and lower bounds based on the result returned by Algorithm VectorSort.

Theorem 8.11 *The short swap distance of a permutation π satisfies the inequality:*

$$|V_\pi|/4 \leq MinSort_{Sw^3}(\pi) \leq |V_\pi|/2.$$

Proof: Given two m-vector-opposite elements, it takes, in the worst case, m swaps to eliminate $2m$ vector segments. This occurs when m is odd (Lemma 8.7) and implies that it takes, on the average, 1/2 swaps for each vector segment eliminated. Assuming that the worst case occurs at every iteration, that is, that FindOpposite always finds m-vector-opposite elements for some odd

GreedyVectors(π). Sorting by short swaps based on distances between vector-opposite elements.

INPUT: A permutation π.
OUTPUT: The length of a sorting sequence of short swaps.

```
dist ← 0
while π ≠ ι
      do if there are even vector-opposite elements in π
            then (i, j) ← positions of d-vector-opposite elements where
                          d is even and as small as possible
            else (i, j) ← positions of d-vector-opposite elements where
                          d is as small as possible
         (π, d) ← LongSwap(π, i, j)
         dist ← dist + d
return dist
```

Figure 8.12: Algorithm GreedyVectors.

m, then $MinSort_{Sw^3}(\pi) \leq |V_\pi|/2$.

For the lower bound, consider the best case at every iteration. This would occur when m is even (Lemma 8.7); in one iteration, it would take $m-1$ swaps to decrease $|V_\pi|$ by $2m$, or, $2m/(m-1)$ vector segments per swap. Clearly, the best case is when $m = 2$, resulting in an average decrease in $|V_\pi|$ of 4 vector segments per swap or 1/4 swaps per vector segment. It follows that $MinSort_{Sw^3}(\pi) \geq |V_\pi|/4$.

□

The following is a direct result of the above theorem.

Corollary 8.12 *Algorithm* VectorSort *sorts a permutation with a sequence of short swaps whose length is within twice the optimal.*

Contrast this with Algorithm GreedyInversions which provides an approximation guarantee of 3 times the optimal.

A refinement on Algorithm VectorSort is Algorithm GreedyVectors, shown in Figure 8.12. In the proof for Theorem 8.11, we find that it is advantageous to locate 2-vector-opposite elements as this implies a swap that causes the best average decrease in vector segments. Define even (odd) vector-opposite elements as m-vector-opposite elements where m is even (odd). In the absence of 2-

vector-opposite elements, locating even vector-opposite elements where d is as small as possible is still useful as swapping them causes a relatively better average decrease in vector segments. For example, 4-vector-opposite elements result in a decrease of 8 segments in 3 swaps, resulting in an average of 8/3 segments per swap; on the other hand, 6-vector-opposite elements result in a decrease of 12 vector segments in 5 swaps—an average of 12/5 segments per swap. Odd vector-opposite elements cause the same average decrease of 2 vectors per swap, regardless of the value of m. Swapping such elements should be delayed as long as possible. A greedy strategy would then be to find the closest pair of even vector-opposite elements (lines 3–5) and then call LongSwap (line 8) to perform the short swaps (as we did with VectorSort). If there are no even vector-opposite elements, the algorithm then selects odd vector-opposite elements, particularly the closest ones (lines 6–7), so that vector elimination is minimized in hope of possible elimination through even vector-opposite elements in succeeding iterations.

Theorem 8.13 *Algorithm* GreedyVectors *sorts a permutation with a sequence of short swaps whose length is within twice the optimal, in $O(n^4)$ time.*

Proof: The approximation guarantee follows from the proof of Theorem 8.11 since GreedyVectors is different from VectorSort only in its method of selecting vector-opposite elements. For the time complexity, $O(n^2)$ vector-opposite elements encountered implies $O(n^2)$ iterations of the **while** loop. At each iteration, we need to select from among $O(n^2)$ vector-opposite elements. It follows that the algorithm runs in $O(n^4)$ time. □

8.4 Best Decrease in Vector Length

Define a *lucky permutation* π as a permutation where $MinSort_{Sw^3}(\pi) = |V_\pi|/4$, the lower bound given in Theorem 8.11. A *best cancellation* is a 3-swap that causes the removal of 4 vector segments in V_π. For lucky permutations, each short swap in an optimal sorting sequence is a best cancellation. In this section, we show that we can recognize lucky permutations and sort them optimally in polynomial time. In particular, we show that Algorithm GreedyVectors in the previous section determines an optimal solution in this case. In addition, we show how its time complexity can be improved if our intention is to determine whether we can sort a permutation optimally using only best cancellations.

First, observe that a best cancellation swaps a pair of 2-vector-opposite elements. The following describes the relationship between a best cancellation and *any* pair of 2-vector-opposite elements in a permutation.

Lemma 8.14 *Let π be a permutation, let σ be a best cancellation, and let $\pi' = \pi \cdot \sigma$. Let* $\mathbf{a}$ *and* $\mathbf{b}$ *be 2-vector-opposite elements in π. Exactly one of the following is true:*

- $|v_\pi(\mathbf{a})| - |v_{\pi'}(\mathbf{a})| = |v_\pi(\mathbf{b})| - |v_{\pi'}(\mathbf{b})| = 2$*, or,*
- $\mathbf{a}$ *and* $\mathbf{b}$ *remain 2-vector-opposite in π'.*

Proof: As σ is a best cancellation and therefore a 3-swap, let $\sigma = \sigma\langle i, i+2\rangle$. Without loss of generality, we assume that $\mathbf{a}$ occurs to the left of $\mathbf{b}$ in π. Consider the situation where the sets of elements $\{\mathbf{a}, \mathbf{b}\}$ and $\{\pi(i), \pi(i+2)\}$ are not disjoint. Noting that $\mathbf{a}$ and $\mathbf{b}$ are 2 positions apart in π, three cases result:

Case 1. $\mathbf{a} = \pi(i)$ and $\mathbf{b} = \pi(i+2)$: This is precisely the case where Lemma 8.6 applies and the vectors $v_\pi(\mathbf{a})$ and $v_\pi(\mathbf{b})$ in V_π are shortened by the swap. By Lemma 8.6, $|v_\pi(\mathbf{a})| - |v_{\pi'}(\mathbf{a})| = |v_\pi(\mathbf{b})| - |v_{\pi'}(\mathbf{b})| = 2$. Furthermore, σ swaps $\mathbf{a}$ and $\mathbf{b}$ so they are no longer vector-opposite in π'.

Case 2. $\mathbf{a} = \pi(i-2)$ and $\mathbf{b} = \pi(i)$: Here, $\pi'(i+2) = \mathbf{b}$ so that $v_{\pi'}(\mathbf{b})$ is longer than $v_\pi(\mathbf{b})$, a contradiction since this would mean that σ is not a best cancellation (a best cancellation shortens two vectors and leaves other vectors unchanged by Lemma 8.6).

Case 3. $\mathbf{a} = \pi(i+2)$ and $\mathbf{b} = \pi(i+4)$: Here, $\pi'(i) = \mathbf{a}$ so that $v_{\pi'}(\mathbf{a})$ is longer than $v_\pi(\mathbf{a})$, a contradiction similar to Case 2.

The above implies that only Case 1 may occur and the result of the lemma follows in the case where $\{\mathbf{a}, \mathbf{b}\}$ and $\{\pi(i), \pi(i+2)\}$ are not disjoint. It remains to consider the situation where these pairs are disjoint. Here, the swap $\sigma\langle i, i+2\rangle$ preserves the original positions of $\mathbf{a}$ in $\mathbf{b}$ so that their positions in π are exactly their positions in π'. The elements therefore remain 2-vector-opposite in π' and the lemma follows. □

Recall that Algorithm GreedyVectors selects 2-vector-opposite elements if they exist. If there are several pairs of 2-vector-opposite elements in the permutation, we may enforce that it selects

the leftmost such pair. In fact, if our intention is to simply determine whether the permutation can be sorted using cancellations, then we can update lines 4–7 of the algorithm to read as follows.

```
i ← 1
found ← false
while i ≤ n − 2 and not found
      do if π(i) ≥ i + 2 and π(i + 2) ≤ i
            then found ← true
            else  i ← i + 1
if found
   then j ← i + 2
   else  (i, j) ← FindOpposite(π)
```

That is, first consider all pairs of positions from left to right where a best cancellation can occur. (there are $O(n)$ of these). If no best cancellations are possible, then select any pair of vector-opposite elements.

We show that it suffices to repeatedly select the leftmost pair of 2-vector-opposite elements when determining an optimal sorting sequence. Define a *leftmost best cancellation* as the best cancellation that swaps the leftmost pair of 2-vector-opposite elements in a permutation, if it exists.

Lemma 8.15 *Let π be a lucky permutation, and let $\sigma_1, \sigma_2, \ldots, \sigma_k$ be an optimal sequence of short swaps that sorts π. Suppose $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}$ are all leftmost best cancellations and σ_i is not. Then, there exists an alternate optimal sorting sequence of short swaps that begins with $\sigma_1, \sigma_2, \ldots, \sigma_{i-1}, \sigma'_i$ where σ'_i is a leftmost best cancellation.*

Proof: Since π is a lucky permutation, the swaps $\sigma_1, \sigma_2, \ldots, \sigma_k$ are all best cancellations. Let

$$\pi^r = \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_r$$

denote the permutation after the first r best cancellations have been applied to π and let $\pi^0 = \pi$. Since π is a lucky permutation, there exists at least one pair of 2-vector-opposite elements in π^{i-1}. Let $\mathbf{a}$ and $\mathbf{b}$ be the leftmost 2-vector-opposite elements in π^{i-1}. By assumption, the swap $\sigma_i \neq S\langle \mathbf{a}, \mathbf{b} \rangle$. However, since the sequence eventually sorts π, there is a subsequent swap $\sigma_j = S\langle \mathbf{a}, \mathbf{b} \rangle$, that shortens the vectors associated with elements $\mathbf{a}$ and $\mathbf{b}$, by Lemma 8.14. Also, by the same lemma, none of the swaps $\sigma_i, \sigma_{i+1}, \ldots, \sigma_{j-1}$ involves the elements $\mathbf{a}$ and $\mathbf{b}$; that is, none of their indices are the positions a and b in π^{i-1}. Now consider the product

$$\hat{\pi}^j = \pi^{i-1} \cdot S\langle \mathbf{a}, \mathbf{b} \rangle \cdot \sigma_i \cdot \sigma_{i+1} \cdots \sigma_{j-1}$$

In this product, $S\langle \mathbf{a}, \mathbf{b} \rangle$ is a leftmost best cancellation. Also, observing that $\pi^{i-1} \cdot S\langle \mathbf{a}, \mathbf{b} \rangle$ is just π^{i-1} with $\mathbf{a}$ and $\mathbf{b}$ switched, then $\hat{\pi}^j$ is just π^{j-1} with $\mathbf{a}$ and $\mathbf{b}$ switched. Set $\sigma_i' = S\langle \mathbf{a}, \mathbf{b} \rangle$ in the product $\hat{\pi}^j$ so that,

$$\begin{aligned}
\hat{\pi}^j &= \pi^{i-1} \cdot \sigma_i' \cdot \sigma_i \cdot \sigma_{i+1} \cdots \sigma_{j-1} \\
&= \pi^{j-1} \cdot S\langle \mathbf{a}, \mathbf{b} \rangle \\
&= \pi^{j-1} \cdot \sigma_j \\
&= \pi^j
\end{aligned}$$

Finally,

$$\begin{aligned}
\pi^k &= \pi^j \cdot \sigma_{j+1} \cdots \sigma_k \\
&= \hat{\pi}^j \cdot \sigma_{j+1} \cdots \sigma_k \\
&= \pi^{i-1} \cdot \sigma_i' \cdot \sigma_i \cdot \sigma_{i+1} \cdots \sigma_{j-1} \cdot \sigma_{j+1} \ldots \sigma_k \\
&= \pi \cdot \sigma_1 \cdot \sigma_2 \cdots \sigma_{i-1} \cdot \sigma_i' \cdot \sigma_i \cdot \sigma_{i+1} \cdots \sigma_{j-1} \cdot \sigma_{j+1} \ldots \sigma_k,
\end{aligned}$$

implying an alternate sorting sequence of the same length where the first i short swaps are leftmost best cancellations. □

Lemma 8.16 *For a permutation π, there exists an optimal sorting sequence of short swaps such that each swap is a leftmost best cancellation.*

Proof: We convert a given optimal sequence to an alternate sorting sequence by repeatedly applying the transformation process given in Lemma 8.15. By one application of the lemma, the resulting sorting sequence contains a longer prefix of leftmost best cancellations. Since the sequence remains optimal at each step, this conversion process terminates with an optimal sorting sequence of short swaps, all of which are leftmost best cancellations. □

Theorem 8.17 *Let π be a permutation. There is an $O(n^3)$ algorithm to determine whether π is a lucky permutation ($MinSort_{Sw^3}(\pi) = |V_\pi|/4$).*

Proof: By Lemma 8.16, repeatedly finding the leftmost best cancellation is sufficient when determining an optimal solution for π as long as π is a lucky permutation. The revision we specified for Algorithm GreedyVectors always finds the leftmost pair of 2-vector-opposite elements if they exist and this corresponds to the leftmost best cancellation. Thus, if π is a lucky permutation, then

the call to LongSwap always applies a leftmost best cancellation and returns $|V_\pi|/4$. Conversely, if Algorithm GreedyVectors returns $|V_\pi|/4$, then π is a lucky permutation, by definition. Therefore, to determine whether π is a lucky permutation, it suffices to test whether the algorithm returns $|V_\pi|/4$.

The revision made for Algorithm GreedyVectors results in a time complexity of $O(n^3)$ since the search for best cancellations requires $O(n)$ time. □

8.5 Bounds for Short Swap Diameter

In this section, we provide bounds for the short swap diameter of the symmetric group S_n. The bounds we obtain first are based on total vector length. We begin with a statement on the maximum possible value for $|V_\pi|$.

Lemma 8.18 *For any permutation π, the total length of the vectors in V_π satisfies the inequality:*

$$|V_\pi| \leq \left\lfloor \frac{n^2}{2} \right\rfloor.$$

Proof: The total length of the vectors in V_π is given by

$$|V_\pi| = \sum_{i=1}^{n} |\pi(i) - i|.$$

We distinguish right vectors from left vectors. Let R be the set of positions i in π such that $v_\pi(\pi(i))$ is a right vector in V_π. The length $|V_\pi|$ is now given by

$$\begin{aligned} |V_\pi| &= \sum_{i\in R}(\pi(i) - i) + \sum_{i\notin R}(i - \pi(i)) \\ &= \sum_{i\in R}\pi(i) + \sum_{i\in R} -i + \sum_{i\notin R} i + \sum_{i\notin R} -\pi(i) \\ &= \sum_{i\in R}\pi(i) + \sum_{i\notin R} i + \sum_{i\in R} -i + \sum_{i\notin R} -\pi(i). \end{aligned}$$

Noting that π is bijective, the above expression sums $2n$ terms such that each of $\{1, 2, \ldots n\}$ appears twice. In addition, the sum has n positive terms and n negative terms. The sum could thus be rewritten as $|V_\pi| = A - B$, where

$$A = \sum_{i\in R}\pi(i) + \sum_{i\notin R} i \quad \text{and} \quad B = \sum_{i\in R} i + \sum_{i\notin R}\pi(i).$$

The intention is to find an upper bound for $|V_\pi|$ so we want A to be as large as possible while B is as small as possible. We accomplish this by setting A to be the sum of the last n elements in the sorted list **1 1 2 2** ... **n n** and B to be the sum of the first n elements.

If n is even,

$$A_{max} = 2 \sum_{i=n/2+1}^{n} i$$

and

$$B_{min} = 2 \sum_{i=1}^{n/2} i.$$

so that

$$\begin{aligned}
|V_\pi| &\leq A_{max} - B_{min} \\
&= 2 \sum_{i=n/2+1}^{n} i - 2 \sum_{i=1}^{n/2} i \\
&= 2 \sum_{i=1}^{n/2} (n/2 + i) - 2 \sum_{i=1}^{n/2} i \\
&= 2 \sum_{i=1}^{n/2} n/2 + 2 \sum_{i=1}^{n/2} i - 2 \sum_{i=1}^{n/2} i \\
&= 2 \sum_{i=1}^{n/2} n/2 \\
&= n^2/2 \\
&= \left\lfloor \frac{n^2}{2} \right\rfloor.
\end{aligned}$$

If n is odd,

$$A_{max} = ((n-1)/2 + 1) + 2 \sum_{i=(n-1)/2+2}^{n} i$$

and

$$B_{min} = ((n-1)/2 + 1) + 2 \sum_{i=1}^{(n-1)/2} i.$$

so that

$$|V_\pi| \leq A_{max} - B_{min}$$

$$
\begin{aligned}
&= ((n-1)/2+1) + 2 \sum_{i=(n-1)/2+2}^{n} i - ((n-1)/2+1) - 2 \sum_{i=1}^{(n-1)/2} i \\
&= 2 \sum_{i=(n-1)/2+2}^{n} i - 2 \sum_{i=1}^{(n-1)/2} i \\
&= 2 \sum_{i=1}^{(n-1)/2} ((n-1)/2+1) + 2 \sum_{i=1}^{(n-1)/2} i - 2 \sum_{i=1}^{(n-1)/2} i \\
&= 2 \sum_{i=1}^{(n-1)/2} ((n-1)/2+1) \\
&= 2 \sum_{i=1}^{(n-1)/2} (n+1)/2 \\
&= (n^2-1)/2 \\
&= \left\lfloor \frac{n^2}{2} \right\rfloor .
\end{aligned}
$$

□

For example, the decreasing permutation $\boldsymbol{\delta}[7] = \mathbf{7\ 6\ 5\ 4\ 3\ 2\ 1}$ has $V_{\boldsymbol{\delta}}$ whose vectors have lengths that are given by

$$
\begin{aligned}
&|v_{\boldsymbol{\delta}}(\mathbf{7})| = 7-1, \quad |v_{\boldsymbol{\delta}}(\mathbf{6})| = 6-2, \quad |v_{\boldsymbol{\delta}}(\mathbf{5})| = 5-3, \quad |v_{\boldsymbol{\delta}}(\mathbf{4})| = 4-4, \\
&|v_{\boldsymbol{\delta}}(\mathbf{3})| = 5-3, \quad |v_{\boldsymbol{\delta}}(\mathbf{2})| = 6-2, \quad |v_{\boldsymbol{\delta}}(\mathbf{1})| = 7-1.
\end{aligned}
$$

and $|V_{\boldsymbol{\delta}}|$ is computed as

$$
\begin{aligned}
|V_{\boldsymbol{\delta}}| &= (7+7+6+6+5+5+4) - (4+3+3+2+2+1+1) \\
&= \left\lfloor \frac{7^2}{2} \right\rfloor = 24.
\end{aligned}
$$

Theorem 8.19 *The short swap diameter of S_n satisfies the inequality:*

$$Diam_{Sw^3}(n) \leq \left\lfloor \frac{n^2}{4} \right\rfloor .$$

Proof: Lemma 8.18 gives us a worst-case total vector length of $\left\lfloor \frac{n^2}{2} \right\rfloor$. From the upper bound we obtained for short swap distance in Theorem 8.11, we have,

$$Diam_{Sw^3}(n) \leq \left\lfloor \frac{n^2}{2} \right\rfloor /2.$$

Table 8.1: Short swap diameters for small n.

n	2	3	4	5	6	7	8	9	10	11
$Diam_{Sw^3}(n)$	1	2	4	5	7	10	14	16	19	23

Noting that when n is odd, its square is $1 \pmod 4$, so the theorem follows. □

Theorem 8.4 in Section 8.1 provides us with a lower bound so that we have

$$\left\lceil \binom{n}{2}/3 \right\rceil \leq Diam_{Sw^3}(n) \leq \left\lfloor \frac{n^2}{2} \right\rfloor /2.$$

Table 8.1 shows $Diam_{Sw^3}(n)$ for small values of n. These where obtained by exhaustively computing the short swap distances of each permutation in S_n (see Section 9.1).

8.6 ℓ-Bounded Swaps

Recall that ℓ-bounded swaps generalize short swaps. In particular, the vector segment diagram still applies. Observe that a k-swap can remove at most $2(k-1)$ vector segments. Thus, for $MinSort_{Sw^\ell}$, a best cancellation is one that removes $2(\ell-1)$ vector segments. The proof of the following theorem is a straightforward generalization of Theorem 8.11.

Theorem 8.20 *The ℓ-bounded swap distance of a permutation π satisfies the inequality:*

$$\frac{|V_\pi|}{2(\ell-1)} \leq MinSort_{Sw^\ell}(\pi) \leq \frac{|V_\pi|}{2}.$$

The upper bound in the above result is discouraging, since it does not involve ℓ. However, it is necessary (at least for greedy algorithms based on vector-opposite elements) since it is possible that only m-vector-opposite elements such that $m = 1$ are found during the sort. This results in an average net decrease of 2 vector segments eliminated per swap. Consider an algorithm for $MinSort_{Sw^\ell}$ that greedily selects the swap that causes the best net decrease in vector segments. When there exists m-vector-opposite elements such that $\ell - 1$ divides m, such an algorithm favors swapping these elements. Otherwise, the algorithm selects, from among the vector-opposite elements, the swap that causes the best net decrease. When $\ell = 3$, the selection is trivial since m being odd always implies an average net decrease of 2. Let $longswapcount(\ell, m)$ denote the number of

ℓ-bounded swaps necessary to eliminate the corresponding vector segments of m-vector-opposite elements. Using Proposition 8.8, we have

$$longswapcount(\ell, m) = \begin{cases} 2(\frac{m}{l-1}) - 1 & \text{if } l-1 \text{ divides } m \\ 2\left\lfloor \frac{m}{l-1} \right\rfloor + 1 & \text{otherwise.} \end{cases}$$

A greedy choice is then the ℓ-bounded swap that maximizes $longswapcount(\ell, m)/2m$.

Finally, corresponding lucky permutations under $MinSort_{Sw^\ell}$ are classes of permutations that can be both detected and solved. The proof of the following theorem is a straightforward generalization of Theorem 8.17.

Theorem 8.21 *Let π be a permutation. There is an $O(n^3)$ algorithm to determine whether*

$$MinSort_{Sw^\ell}(\pi) = \frac{|V_\pi|}{2(\ell - 1)}.$$

Chapter 9

Experimental Results

In this chapter, we evaluate experimentally the models used in the previous two chapters. In Section 9.1, we present exact but exhaustive algorithms for sorting by generators. The algorithms are implemented for $MinSort_{Bk^3}$ and $MinSort_{Sw^3}$ and are used for permutations of relatively short lengths to compare to the solutions obtained from our heuristics. Section 9.2 outlines the framework we use for conducting the experiments and obtaining the results in the next two sections. Finally, in Sections 9.3 and 9.4, we analyze $MinSort_{Bk^3}$ and $MinSort_{Sw^3}$ using this framework. In these sections, we consider the heuristics presented in the previous chapters and provide examples of permutations where these heuristics fail. We then implement these heuristics to determine how well they perform in practice.

9.1 Exact Algorithms

In this section, we present algorithms that compute the exact P distance of a permutation. Our first algorithm computes the P distances of *all* permutations of a given length. Algorithm SortAll, shown in Figure 9.1, takes as input an integer n and returns the P distance of all permutations of length n. The algorithm uses an array T of $n!$ elements each with two fields, *dist* and *link*, initialized to ∞ and **nil** (lines 2–5). Each permutation π of size n corresponds to an element in the array. We assume that P is indexed by S_n, the set of length-n permutations (in our actual implementation, we order the permutations in lexicographic order, map the permutations to integers, and then use integers as indices into the array). During the execution of Algorithm SortAll, the *dist* field of

SortAll(n) $MinSort_P$ for all permutations of length n.

INPUT: An integer n.
OUTPUT: A *P-generator tree* T, of all length-n permutations rooted at $\boldsymbol{\iota}$, such that the length of the path from any node π to $\boldsymbol{\iota}$ represents the P distance of π.

```
▷ T is an array of {dist and link} indexed by S_n (see text)
▷ First, initialize the T array
for each π ∈ S_n
    do T[π].dist ← ∞
       T[π].link ← nil
T[ι].dist ← 0
for k ← 0 to bound − 1   ▷ bound is some diameter bound appropriate for P
    do for each π where T[π].dist = k
            do ▷ determine all permutations accessible from π
               for each γ ∈ P[n]
                   do π′ ← π · γ
                      if T[π′].dist = ∞
                        then T[π′].dist ← k + 1
                             T[π′].link ← π
return T
```

Figure 9.1: Algorithm SortAll.

$T[\pi]$, for all permutations $\pi \in S_n$, contains the P distance of π. More precisely, a *P-generator tree* is returned (line 15) where the identity permutation is the root node and the *link* field of a node $T[\pi]$ is a pointer to its parent node. The unique path from node $T[\pi]$ to the root represents a transformation of π to the identity permutation using the minimum number of allowable generators. $T[\pi].dist$ is exactly the length of this path.

The algorithm sets the distances and links of the nodes as follows. After initialization, $T[\boldsymbol{\iota}].dist$ is set to 0 (line 6). Then, a **for** loop (lines 7–14) iteratively sets to $k+1$ all the *dist* fields of all unvisited nodes accessible from nodes whose *dist* field is k. Let $T[\pi]$ and $T[\pi']$ be nodes such that $T[\pi].dist = k$ and $\pi' = \pi \cdot \gamma$ for some generator $\gamma \in P[n]$. Then, $P[\pi'].dist$ is set to $k+1$ and $P[\pi'].link$ is set to π. The loop is executed *bound* times where *bound* is an upper bound on $Diam_P(n)$.

This algorithm is in fact a single-source shortest-paths algorithm for the Cayley graph $\mathcal{G}^c(P, n)$. The single source is the identity permutation $\boldsymbol{\iota}$ and the distances between the nodes are all 1. As a result, it suffices to simply look for all nodes with $dist = \infty$ that can be reached from some node with $dist = k$. A breadth-first search could take the place of the **for** loop in the algorithm.

The transformation represented by the path from a node to the root is not typically the only way to optimally sort π. The program determines only one such way to sort. This is because $P[\pi'].dist$ is set once a permutation π (with distance k) and a generator $\gamma \in P[n]$ has been encountered with $\pi' = \pi \cdot \gamma$. The distance is not reset even when the same condition applies later in the execution. A useful enhancement for future work is to implement *link* as a list that stores all possible links.

Algorithm SortAll runs in $\Theta(p(n)n!)$ where $p(n)$ represents the cost of applying each generator in $P[n]$ to a permutation. All permutations π (there are $n!$ of these) are eventually visited and all permutations accessible from π are tested for each π. For example, if $P = Bk^3$, there are $2n-3$ short block-moves in $Bk^3[n]$ and applying a short block-move to a permutation takes constant time. Thus, SortAll runs in $\Theta((n+1)!)$ time for short block-moves; a similar analysis applies when $P = Sw^3$. For $P = Rv$, on the other hand, applying an m-reversal to a permutation requires $\Theta(m)$ time. There are $\Theta(n^2)$ reversals in $Rv[n]$ such that $m = \Theta(n)$. This implies a time complexity of $\Theta(n^3)$ for applying each reversal to a permutation and a resulting time complexity of $\Theta(n^3 n!)$ for SortAll.

Algorithm SortAll is practical only for small n, in particular, when $n \leq 10$. Later in this section, we present a branch-and-bound algorithm that can be implemented for relatively larger

SortAllQualify(n) $MinSort_P$ for all permutations of length n.

INPUT: An integer n.
OUTPUT: A *P-generator tree* T of all length-n permutations qualified under some routine Check that tests a conjecture.

```
1   ▷ T is an array of {dist, link, and mark} indexed by S_n
2   for each π ∈ S_n
3       do T[π].dist ← ∞
4          T[π].link ← nil
5          T[π].mark ← false
6   T[ι].dist ← 0
7   T[ι].mark ← true
8   for k ← 0 to bound − 1
9       do for each π where T[π].dist = k
10             do for each γ ∈ P[n]
11                    do π′ ← π · γ
12                       if T[π′].dist = ∞
13                         then T[π′].dist ← k + 1
14                              T[π′].link ← π
15                       if P[π′].dist = k + 1
16                         then if P[π′].mark = false and Check(π′, γ^−1)
17                                 then P[π′].mark ← true
18                                      P[π′].link ← π
19  return T
```

Figure 9.2: Algorithm SortAllQualify.

values of n.

A useful variant of Algorithm SortAll is shown in Figure 9.2. Algorithm SortAllQualify is similar in structure to SortAll, so it takes as input an integer n and returns a P-generator tree T. However, it returns a *qualified* tree T based on some Check routine. Check performs an acceptability test on a permutation and a generator—that is, P is further restricted so that the algorithm may "prefer" one generator over another. This way, a conjecture on how a generator is chosen can be tested. An additional field *mark* is set to **true** when a link that passes the Check constraint occurs (lines 16–18). The mark fields are all initialized to **false** (line 5) so that, effectively, all those nodes whose *mark* fields remain **false** upon completion are counterexamples to the conjecture. As an example, consider the (proven) conjecture for short swaps (see Theorem

BranchAndBound(π) Sorting by generators using branch and bound.

INPUT: A permutation π.
OUTPUT: An optimal sequence of generators from $P[n]$ that sorts π.

```
GLOBAL bound, current[1..n], best[1..n]
(bound, best) ← UpperBound(π)
Search(π, 0)
return (bound, best)
```

Search(π, *depth*) Recursive search routine used by BranchAndBound.

```
if π = ι
  then ▷ depth < bound
       bound ← depth
       best ← current
  else for each γ ∈ P[n]
           do if LowerBound(π · γ) + depth + 1 < bound
              then current[depth + 1] ← γ
                   Search(π · γ, depth + 1)
```

Figure 9.3: Algorithm BranchAndBound.

5.6) where we state that only correcting swaps are necessary. The check routine would then be as follows:

$$\textsc{Check}(\pi, \sigma\langle i, j\rangle) \qquad \{\textbf{return } \pi(i) > \pi(j)\}.$$

Algorithm SortAllQualify is used by Heath and Vergara [20] for testing conjectures on sorting by reversals.

Algorithms SortAll and SortAllQualify require a huge amount of memory, particularly because they return the P distances of all permutations in S_n. Our next algorithm, shown in Figure 9.3, computes the P distance for a single permutation and could thus be used for larger n. Algorithm BranchAndBound is patterned after the branch and bound algorithm for sorting by reversals presented by Kececioglu and Sankoff [27]. The algorithm assumes that two routines, UpperBound and LowerBound, are available. These routines depend on P and return upper and lower bounds, respectively, for the P distance of a permutation; in addition, UpperBound returns a particular

sorting sequence (whose length is the upper bound). In the algorithm, the global variables *bound* and *best* hold the best distance and sorting sequence computed by the algorithm so far, respectively. Initially, these variables contain the result returned by UpperBound (line 2). The algorithm then invokes a Search routine (line 3) that recursively applies to the permutation all generators that may lead to an optimal solution. Upon completion of the search routine, *bound* and *best* contain an optimal solution and are returned by Algorithm BranchAndBound (line 4).

The procedure Search considers all generators $\gamma \in P[n]$ except those that could not possibly lead to an optimal solution for π based on LowerBound($\pi \cdot \gamma$) (lines 5–8 of Search). Whenever the identity permutation $\boldsymbol{\iota}$ is encountered with a resulting number of generators smaller than the current solution (lines 1–2), then the solution is updated (lines 3–4).

The time complexity of Algorithm BranchAndBound is exponential in the worst case and is highly dependent on the input permutation and the quality of the results returned by UpperBound and LowerBound. We have implemented this algorithm for short block-moves and short swaps and have been able to optimally sort permutations with length ≤ 10 or with expected distances ≤ 10.

9.2 Experimental Framework

In the last two sections of this chapter, heuristics for $MinSort_{Bk^3}$ and $MinSort_{Sw^3}$ are evaluated. In this section, the general framework used in investigating these two problems is described.

For each of these problems, we implement the heuristics presented in Chapters 6, 7, and 8. We first present examples of permutations where each of the heuristics fail either by implementing Algorithm SortAllQualify in Section 9.1, or by comparing the results of Algorithm SortAll (also in Section 9.1) with those of the heuristics.

The heuristics are then tested for permutations of different lengths. The lengths n fall into the following classes:

- *Small:* $n < 10$ (specifically, $n \in \{5, 6, 7, 8, 9\}$).
- *Medium:* $10 \leq n \leq 50$ (specifically, $n \in \{10, 20, 30, 40, 50\}$).
- *Large:* $50 < n \leq 200$ (specifically, $n \in \{100, 150, 200\}$).

For the *Small* case, we test the heuristics for all ($n!$) permutations of the given length and then contrast these results with the exact results given by Algorithm SortAll. For the *Medium* and *Large* cases, the heuristics are tested for permutations of the given lengths and the results are compared with each other, since exact results are not always available. Whenever possible, exact results are obtained using BranchAndBound and comparisons similar to the *Small* case are made.

For the *Medium* and *Large* cases, permutations are generated randomly. Two methods of generating these permutations are employed. The first method involves randomly selecting numbers from $\{\mathbf{1}, \mathbf{2}, \ldots, \mathbf{n}\}$ in sequence to form a permutation of length n. It has been shown that permutations of this type tend to yield worst-case P distances, particularly when P describes reversals [6]. We use a second method to provide permutations that are not necessarily worst-case. For a given integer m, we obtain a test permutation by randomly applying m generators to the identity permutation. This way, the P distance is at most m. In our experiments, we use the following values for m: $\lfloor\sqrt{n}\rfloor$, n, and $\lfloor n^2/4\rfloor$, We generate 200 test permutations for each experiment in the *Medium* and *Large* case, broken down as follows: 50 permutations generated randomly using the first method and 50 permutations for each m-value using the second method.

Our results are tabulated as follows. For the *Small* case, for each heuristic, we determine the average ratio between the computed distance and the exact distance and count the number of permutations where the correct P distance is obtained. For the *Medium* case, when $n = 10$ or $m = \sqrt{n}$, exact results are made available by the branch and bound algorithms so the same comparisons are made. For the rest of the medium-sized permutations and for the *Large* case, we determine the average distance computed by each heuristic. In addition, particularly for the *Large* case, we distinguish between the different permutation "types" ($m = \lfloor\sqrt{n}\rfloor$, $m = n$, $m = \lfloor n^2/4\rfloor$, and random). All the detailed tabulations are in the appendix.

The machine used in these experiments is a DEC Alpha 255/233 running Digital UNIX V4.0. All heuristics were implemented in C/C++. The timings for the experiments are shown in the appendix. All but two of the heuristics implemented took an average of less than one second per permutation for all the different permutation lengths and types. For short block-moves, only GreedyMatching (an $O(n^8)$ algorithm) posted average computation times longer than one second. In this case, the algorithm was not tested for some permutations; the longest time recorded was around 20 minutes per permutation, for random permutations of length 50 (see Table A.10). On the other hand, for short swaps, GreedyInversions posted an average of 20 seconds for random permutations of length

200 (see Table B.10).

9.3 Sorting by Short Block-Moves

We implement four heuristics for this problem: Algorithm GreedyMatching (Section 6.6), Algorithm DegreeSort (Section 6.7), Algorithm GreedyHops (Section 6.8), and Algorithm LISSort (Section 7.4).

First, we list the conjectures associated with each heuristic and provide examples of permutations where the heuristics fail. For GreedyMatching, we begin with a counterexample for a conjecture based on the model employed in Section 6.6.

Conjecture 9.1 *For a permutation π, there exists a realizable sorting sequence of short block-moves from the set of potential block-moves derived from any maximum matching of $\mathcal{G}_\pi^a$.*

Counterexample: **2 4 6 1 7 5 8 3**

The above is a permutation whose arc graph contains 10 vertices. Furthermore, the graph has a perfect matching, which incorrectly suggests a short block-move distance of 5. The set of potential block-moves implied by a matching is

$$\{B\langle \mathbf{2\ 4}, \mathbf{1}\rangle, B\langle \mathbf{4\ 6}, \mathbf{3}\rangle, B\langle \mathbf{6}, \mathbf{1\ 5}\rangle, B\langle \mathbf{7}, \mathbf{5\ 3}\rangle, B\langle \mathbf{5}, \mathbf{8\ 3}\rangle\}.$$

It can be verified that this set of block-moves (and other sets based on alternate perfect matchings) cannot be realized. An optimal solution is shown below (short block-move distance is 6).

$$\begin{aligned}
\mathbf{2\ 4\ 6\ 1\ 7\ 5\ 8\ 3} &\rightarrow \mathbf{2\ 1\ 4\ 6\ 7\ 5\ 8\ 3}\\
&\rightarrow \mathbf{1\ 2\ 4\ 6\ 7\ 5\ 8\ 3}\\
&\rightarrow \mathbf{1\ 2\ 4\ 5\ 6\ 7\ 8\ 3}\\
&\rightarrow \mathbf{1\ 2\ 4\ 5\ 6\ 3\ 7\ 8}\\
&\rightarrow \mathbf{1\ 2\ 4\ 3\ 5\ 6\ 7\ 8}\\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}
\end{aligned}$$

Now, suppose a product $\pi \cdot \beta$ results in the above counterexample. Then, a greedy algorithm based on *resulting* matchings such as GreedyMatching may fail for the permutation π.

Conjecture 9.2 GreedyMatching *always computes an optimal solution.*

Counterexample: **2 4 6 1 8 7 5 3**

GreedyMatching performs the following first step, resulting in the previous counterexample.

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

GreedyMatching then returns a distance of 7. The actual short block-move distance is 6 as exhibited by the solution given below.

$$\begin{aligned}
\mathbf{2\ 4\ 6\ 1\ 8\ 7\ 5\ 3} &\rightarrow \mathbf{2\ 4\ 6\ 1\ 5\ 8\ 7\ 3}\\
&\rightarrow \mathbf{2\ 4\ 1\ 5\ 6\ 8\ 7\ 3}\\
&\rightarrow \mathbf{1\ 2\ 4\ 5\ 6\ 8\ 7\ 3}\\
&\rightarrow \mathbf{1\ 2\ 4\ 5\ 6\ 7\ 3\ 8}\\
&\rightarrow \mathbf{1\ 2\ 4\ 5\ 3\ 6\ 7\ 8}\\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8}
\end{aligned}$$

Conjecture 9.3 DegreeSort *always computes an optimal solution.*

Counterexample: **4 1 3 2 9 6 5 8 7**

In this example, the element **9** has even out-degree so it is moved to the left, resulting in

$$\begin{aligned}
\mathbf{4\ 1\ 3\ 2\ 9\ 6\ 5\ 8\ 7} &\rightarrow \mathbf{4\ 1\ 3\ 2\ 6\ 5\ 9\ 8\ 7}\\
&\rightarrow \mathbf{4\ 1\ 3\ 2\ 6\ 5\ 8\ 7\ 9}
\end{aligned}$$

At this point, **1**, **2**, **6**, and **8** are the 2 minimum and 2 maximum unpositioned elements. In addition, their degrees are all odd and they are in their correct relative order. DegreeSort therefore spends a skip. However, the optimal solution for this permutation consists of all correcting hops, as follows.

$$\begin{aligned}
\mathbf{4\ 1\ 3\ 2\ 9\ 6\ 5\ 8\ 7} &\rightarrow \mathbf{1\ 3\ 4\ 2\ 9\ 6\ 5\ 8\ 7}\\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 9\ 6\ 5\ 8\ 7}\\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 9\ 6\ 8\ 7}\\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6\ 8\ 9\ 7}\\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9}
\end{aligned}$$

Conjecture 9.4 GreedyHops *always computes an optimal solution.*

Counterexample: **2 4 6 5 1 3**

Here, there are no lone skips in the permutation and the correcting hop $B\langle \mathbf{6}, \mathbf{5\ 1} \rangle$ does not introduce lone skips. Yet, starting with this block-move causes 2 lone skips later in the sort as the following sorting sequence of length 5 illustrates.

$$\begin{aligned} \mathbf{2\ 4\ 6\ 5\ 1\ 3} &\rightarrow \mathbf{2\ 4\ 5\ 1\ 6\ 3} \\ &\rightarrow \mathbf{2\ 1\ 4\ 5\ 6\ 3} \\ &\rightarrow \mathbf{1\ 2\ 4\ 5\ 6\ 3} \\ &\rightarrow \mathbf{1\ 2\ 4\ 3\ 5\ 6} \\ &\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6} \end{aligned}$$

An optimal solution (short block-move distance is 4) follows.

$$\begin{aligned} \mathbf{2\ 4\ 6\ 5\ 1\ 3} &\rightarrow \mathbf{2\ 4\ 1\ 6\ 5\ 3} \\ &\rightarrow \mathbf{1\ 2\ 4\ 6\ 5\ 3} \\ &\rightarrow \mathbf{1\ 2\ 4\ 6\ 5\ 3} \\ &\rightarrow \mathbf{1\ 2\ 4\ 5\ 3\ 6} \\ &\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6} \end{aligned}$$

Conjecture 9.5 LISSort *always computes an optimal solution.*

Counterexample: **5 2 1 4 3**

The above permutation has an LIS of length 2, namely **1 3**, and the element **5** has even insert distance. However, positioning this element first results in the following sorting sequence.

$$\begin{aligned} \mathbf{5\ 2\ 1\ 4\ 3} &\rightarrow \mathbf{2\ 1\ 5\ 4\ 3} \\ &\rightarrow \mathbf{2\ 1\ 4\ 3\ 5} \\ &\rightarrow \mathbf{1\ 2\ 4\ 3\ 5} \\ &\rightarrow \mathbf{1\ 2\ 3\ 4\ 5} \end{aligned}$$

An optimal solution (short block-move distance is 3) is as follows.

$$\begin{aligned} \mathbf{5\ 2\ 1\ 4\ 3} &\rightarrow \mathbf{1\ 5\ 2\ 4\ 3} \\ &\rightarrow \mathbf{1\ 2\ 4\ 5\ 3} \\ &\rightarrow \mathbf{1\ 2\ 3\ 4\ 5} \end{aligned}$$

We now present the results of our experiments. Table 9.1 shows the number of times the algorithms compute the correct distances for *all* small-sized permutations (those of length 5, 6, 7, 8, and 9). For $n \leq 7$, GreedyMatching computed all the distances correctly; for lengths 8 and 9, it returned the correct distances for more than 99% of permutations. For the other three algorithms, there is a consistent decrease in percentage of correct computations as the permutation length increases. GreedyHops appears to be the better of the three returning correct distances at least 98% of the time for lengths ≤ 7. When $n = 9$, the differences in performance between these three algorithms are most significant, showing percentages of 86.2%, 88.9%, and 93.3%, for DegreeSort, LISSort, and GreedyHops, respectively. In terms of closeness of approximation, measured as the average ratio between computed and correct distances, all four algorithms performed satisfactorily showing ratios that are very close to 1 for small-sized permutations, as shown in Table 9.2.

Results for length-10 permutations, shown in Table 9.3, are consistent with the *Small* case. The table distinguishes between the four different types of test permutations; recall that the algorithms were run with 50 permutations of each type. GreedyMatching returned correct computations for all the test permutations. The other three algorithms, on the other hand, exhibited the same general ranking in terms of performance: GreedyHops returned the largest number of correct computations while DegreeSort returned the fewest. This experiment also exhibits the relationship between expected distance (m) and the correctness of the algorithms. Particularly for DegreeSort and LISSort, larger expected distances imply less correct computations. The table containing the corresponding approximation ratios for this experiment is shown in the appendix (Table A.3).

For other medium-sized permutations, we first observe that when $m = \sqrt{n}$, all algorithms return optimal or near-optimal results. All four algorithms except for DegreeSort returned correct computations for all the permutations; DegreeSort returned an incorrect solution only for one permutation. The appendix shows the tables containing the percentages and corresponding approximation ratios for this experiment (Tables A.1 and A.2).

Table 9.1: Number (and percentage) of correct Bk^3 distance computations (*Small*).

Length n	$n!$	GREEDY-MATCHING	DEGREE-SORT	GREEDY-HOPS	LISSORT
5	120	120 (100.0%)	119 (99.2%)	120 (100.0%)	119 (99.2%)
6	720	720 (100.0%)	693 (96.3%)	707 (98.2%)	694 (96.4%)
7	5040	5040 (100.0%)	4708 (93.4%)	4939 (98.0%)	4743 (94.1%)
8	40320	40238 (99.8%)	36113 (89.6%)	38198 (94.7%)	36748 (91.1%)
9	362880	361898 (99.7%)	312690 (86.2%)	338554 (93.3%)	322463 (88.9%)

Table 9.2: Average ratio between computed and correct Bk^3 distance (*Small*).

Length	GREEDY-MATCHING	DEGREE-SORT	GREEDY-HOPS	LISSORT
5	1.000	1.003	1.000	1.003
6	1.000	1.010	1.004	1.011
7	1.000	1.013	1.004	1.013
8	1.000	1.016	1.008	1.015
9	1.000	1.017	1.008	1.015

Table 9.4 presents the data for all medium-sized and large-sized permutations. The table represents all 200 test permutations for each length. Here, the average distances computed by the algorithms seem to indicate a similar pattern. That is, that the ranking exhibited by the algorithms according to performance is as follows: GREEDYMATCHING, GREEDYHOPS, LISSORT, and DEGREESORT. The results for GREEDYMATCHING with large-sized permutations are unavailable particularly for the third and fourth types as computations for these cases took too long (greater than one hour per permutation). Table 9.5 and Table 9.6 present the results for $n = 50$ and $n = 200$, respectively, broken down into the four different types of permutations (the tables containing similar breakdowns for the other lengths are shown in the appendix). In Table 9.5, we find that the difference between the computed distances between GREEDYMATCHING (the best algorithm) and DEGREESORT (the worst) appears more significant when $m = n$ and $m = \lfloor n^2/4 \rfloor$, less significant when $m = \sqrt{n}$ and for random permutations. In Table 9.6, we find that GREEDYHOPS performs significantly better than DEGREESORT and LISSORT for random permutations.

In summary, Algorithm GREEDYMATCHING performs very well in practice in that it often produces the correct results at least for permutations of size ≤ 50. In addition, whenever exact results

Table 9.3: Number (and percentage) of correct Bk^3 distance computations ($n = 10$).

m	GREEDY-MATCHING	DEGREE-SORT	GREEDY-HOPS	LISSORT
3	50 (100%)	49 (98%)	50 (100%)	50 (100%)
10	50 (100%)	42 (84%)	48 (96%)	47 (94%)
25	50 (100%)	43 (86%)	44 (88%)	46 (92%)
Random	50 (100%)	41 (82%)	48 (96%)	43 (86%)

Table 9.4: Average computed Bk^3 distance (*Medium* and *Large*).

Length	GREEDY-MATCHING	DEGREE-SORT	GREEDY-HOPS	LISSORT
10	6.57	6.69	6.62	6.64
20	20.96	21.45	21.16	21.36
30	42.58	43.44	42.89	43.20
40	70.67	71.89	71.19	71.17
50	106.50	108.22	107.03	107.36
100	—	395.46	392.53	394.47
150	—	851.77	846.87	851.35
200	—	1471.33	1464.20	1474.69

are available, we find that its overall approximation ratio is very close to 1. For medium-sized and large-sized permutations, where a basis for determining correctness is not generally available, the algorithm faired well against the other three algorithms. One disadvantage of GREEDYMATCHING is the fact that it is based on determining maximum matchings so its running time is dependent on this task. GREEDYMATCHING is therefore recommendable particularly for small-sized permutations.

Algorithms DEGREESORT, GREEDYHOPS, and LISSORT, on the other hand, are more efficient even for large-sized permutations since the heuristics require limited linear scans of the permutation. However, their performance in terms of correctness and approximation ratio declines more significantly as permutation length increases (for *Small* permutations). GREEDYHOPS, the only algorithm with a proven approximation bound, appears to be the best among the three. The counterexamples given in this section for GREEDYHOPS are solved by DEGREESORT and LISSORT—this suggests a possible hybrid algorithm that may produce better results and would remain efficient.

Table 9.5: Average computed Bk^3 distance ($n = 50$).

m	GREEDY-MATCHING	DEGREE-SORT	GREEDY-HOPS	LISSORT
7	6.30	6.30	6.30	6.30
50	25.82	27.20	26.12	26.32
625	92.16	95.42	93.48	92.82
Random	301.70	303.98	302.20	304.00

Table 9.6: Average computed Bk^3 distance ($n = 200$).

m	GREEDY-MATCHING	DEGREE-SORT	GREEDY-HOPS	LISSORT
14	13.56	13.70	13.56	13.56
200	105.52	112.00	106.68	107.14
10000	—	814.14	802.42	793.96
Random	—	4945.46	4934.16	4984.12

9.4 Sorting by Short Swaps (Short Reversals)

We implement two heuristics for this problem: Algorithm GREEDYINVERSIONS (Section 8.1) and Algorithm GREEDYVECTORS (Section 8.3). We start with a discussion of conjectures and counterexamples associated with each heuristic.

The following is the conjecture associated with GREEDYVECTORS.

Conjecture 9.6 *For a permutation π, there exists an optimal sorting sequence of short swaps such that each swap maximally shortens total vector length.*

Counterexample: **3 4 6 1 5 2**

The greedy choice in this case is the swap $S\langle \mathbf{4}, \mathbf{1} \rangle$ since it eliminates 4 vector segments. This, however, leads to a non-optimal solution. An optimal solution (short swap distance is 4) is as follows.

$$\begin{aligned} \mathbf{3\ 4\ 6\ 1\ 5\ 2} &\rightarrow \mathbf{3\ 4\ 1\ 6\ 5\ 2} \\ &\rightarrow \mathbf{1\ 4\ 3\ 6\ 5\ 2} \\ &\rightarrow \mathbf{1\ 4\ 3\ 2\ 5\ 6} \\ &\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6} \end{aligned}$$

A related conjecture is the following, since GreedyVectors favors cancellations and cuts.

Conjecture 9.7 *For a permutation π, there exists an optimal sorting sequence of short swaps that does not contain switches or flips.*

The following counterexample requires a switch as the first swap in its optimal solution.

Counterexample: **6 2 5 3 4 1**

$$\begin{aligned}
\mathbf{6\ 2\ 5\ 3\ 4\ 1} &\rightarrow \mathbf{6\ 2\ 5\ 3\ 1\ 4} \\
&\rightarrow \mathbf{6\ 2\ 1\ 3\ 5\ 4} \\
&\rightarrow \mathbf{1\ 2\ 6\ 3\ 5\ 4} \\
&\rightarrow \mathbf{1\ 2\ 3\ 6\ 5\ 4} \\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6}
\end{aligned}$$

This next example, on the other hand, requires a flip.

Counterexample: **6 2 4 5 3 1**

$$\begin{aligned}
\mathbf{6\ 2\ 4\ 5\ 3\ 1} &\rightarrow \mathbf{6\ 2\ 4\ 1\ 3\ 5} \\
&\rightarrow \mathbf{6\ 2\ 1\ 4\ 3\ 5} \\
&\rightarrow \mathbf{1\ 2\ 6\ 4\ 3\ 5} \\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 6\ 5} \\
&\rightarrow \mathbf{1\ 2\ 3\ 4\ 5\ 6}
\end{aligned}$$

Finally, we have the corresponding conjecture for Algorithm GreedyInversions.

Conjecture 9.8 *For a permutation π, there exists an optimal sorting sequence of short swaps such that each swap corrects the maximum number of inversions possible.*

Counterexample: **6 2 4 5 3 1**

Here, **5 2 1** is a triple, but swapping **5** with **1** will not lead to an optimal solution. An optimal solution is:

Table 9.7: Number (and percentage) of correct Sw^3 distance computations (*Small*).

Length n	$n!$	GREEDY-INVERSIONS	GREEDY-VECTORS
5	120	115 (95.8%)	120 (100.0%)
6	720	652 (90.6%)	693 (96.3%)
7	5040	4161 (82.6%)	4553 (90.3%)
8	40320	29350 (72.8%)	32526 (80.7%)
9	362880	223789 (61.7%)	250240 (69.0%)

Table 9.8: Average ratio between computed and correct Sw^3 distance (*Small*).

Length	GREEDY-INVERSIONS	GREEDY-VECTORS
5	1.024	1.000
6	1.040	1.014
7	1.057	1.030
8	1.072	1.048
9	1.084	1.065

$$\begin{aligned} \mathbf{3\,6\,4\,5\,2\,1} &\rightarrow \mathbf{3\,6\,4\,2\,5\,1} \\ &\rightarrow \mathbf{3\,2\,4\,6\,5\,1} \\ &\rightarrow \mathbf{3\,2\,4\,1\,5\,6} \\ &\rightarrow \mathbf{3\,2\,1\,4\,5\,6} \\ &\rightarrow \mathbf{1\,2\,3\,4\,5\,6} \end{aligned}$$

The results of our experiments follow. Table 9.7 shows the number of times the algorithms compute the correct short swap distances for all permutations of length 5, 6, 7, 8, and 9. Table 9.8 shows the corresponding approximation ratios. GREEDYVECTORS performs relatively better both in terms of correctness and approximation ratio. However, we note a significant decline in performance in both measures for both algorithms as the permutation length increases.

Tables 9.9 and 9.10 show the results for $n = 10$, broken down into the four types of permutations. Although GREEDYVECTORS still performs relatively well, consistent with the results in the *Small* case, we find that with random permutations, the difference in performance is not that significant.

Table 9.9: Number (and percentage) of correct Sw^3 distance computations ($n = 10$).

m	GREEDY-INVERSIONS	GREEDY-VECTORS
3	50 (100%)	50 (100%)
10	47 (94%)	50 (100%)
25	40 (80%)	47 (94%)
Random	25 (50%)	26 (52%)

Table 9.10: Average ratio between computed and correct Sw^3 distance ($n = 10$).

m	GREEDY-INVERSIONS	GREEDY-VECTORS
3	1.000	1.000
10	1.020	1.000
25	1.043	1.013
Random	1.088	1.087

Incidentally, for these types of permutations, both algorithms performed poorly, returning correct computations for only about 50% the test permutations.

We ran a test similar to the previous section for medium-sized permutations of the first type, that is, when the expected distance is $m = \sqrt{n}$. We find that, in this case, both algorithms returned correct results for all the test permutations (see Tables B.1 and B.2 in the appendix).

Table 9.11 shows the average distance computed by the algorithms for all medium-sized and large-sized permutations; recall that correct distances are not available. We notice a result opposite to that of the *Small* case. GREEDYINVERSIONS now performs better than GREEDYVECTORS, in fact increasingly better as the permutation size increases.

Finally, we show in Table 9.12 the average distances computed by the algorithms for $n = 200$ broken down into permutation type (see appendix for other lengths). As one may expect, the difference in performance between the two algorithms is more significant when the expected distance is larger (for $m = \lfloor n^2/4 \rfloor$ and for random permutations).

To summarize, both GREEDYVECTORS and GREEDYINVERSIONS perform particularly well for small-sized permutations. There is a decline in correctness and approximation ratio as permutation length increases for these heuristics. However, compared with their theoretical approximation bounds

Table 9.11: Average computed Sw^3 distance (*Medium* and *Large*).

Length	GREEDY-INVERSIONS	GREEDY-VECTORS
10	7.37	7.27
20	22.95	23.30
30	44.03	45.00
40	71.84	74.63
50	106.95	111.33
100	380.67	400.06
150	817.59	859.71
200	1415.85	1481.78

Table 9.12: Average computed Sw^3 distance ($n = 200$).

m	GREEDY-INVERSIONS	GREEDY-VECTORS
14	13.72	13.68
200	128.64	126.40
10000	809.52	850.08
Random	4711.52	4936.96

(2 for GREEDYVECTORS and 3 for GREEDYINVERSIONS), the results are still much better than expected—at most 1.088 for permutations where the exact results are available. Finally, we note that GREEDYVECTORS returns better results than GREEDYINVERSIONS for small-sized permutations but that the opposite is the case for medium-sized and large-sized permutations, particularly when expected distance is large.

Chapter 10

Conclusions

An ideal system for a biologist studying genome rearrangements is one where the predicate P can be prescribed and the minimum sorting sequence for any permutation can be derived. Even more useful is one where weights can be assigned to particular rearrangements based on length or distance as suggested by Kececioglu and Sankoff [27] where the sorting sequence with minimum total weight is sought. In this situation, it is important to understand the effect of bounding the lengths of these rearrangements on the computational complexity of the problem. This dissertation has gained some progress in this respect.

We show (in Chapter 4) that for some predicates such as those that describe block-moves and reversals, bounded versions of the problem are at least as difficult computationally as the unbounded problems. This occurs precisely when the distance bound is a function of the length of the permutation. This, in turn, motivates investigating bounded problems where the distance bound is fixed in search of possibly easier $MinSort_P$ variants.

In Chapter 5, we find that for some bounded problems, a strong relationship exists between finding optimal sorting sequences and correcting the relative order of individual pairs of elements. This is not particularly present in unbounded problems except for $MinSort_{Sw}$ (for $MinSort_{Rv}$, for example, optimally sorting the permutation **2 3 4 5 6 1** has to involve a first step that destroys the relative order between several pairs of elements). The simplest among such problems that are yet to be resolved are those where the distance bound is exactly 3—short generators. Two specific problems where we have gained significant progress are sorting by short block-moves and sorting by

short swaps (short reversals).

These statements on correcting relative order allow us to develop graph models appropriate for $MinSort_{Bk^3}$ and $MinSort_{Sw^3}$ (or $MinSort_{Rv^3}$). Using the permutation graphs and arc graphs of Chapter 6, we determine $Diam_{Bk^3}(n)$ and solve $MinSort_{Bk^3}$ for several types of permutations such as woven bitonic and woven double-strip permutations. We also determine bounds for the short block-move distance of permutations based on these graph models. Empirically, we show that algorithms based on degrees in the permutation graph as well as the maximum matchings in the arc graphs perform well even with permutations not under these special types (see Section 9.3). In fact, one of the algorithms we devise based on some greedy hop selection computes a sorting sequence whose length is provably within $\frac{4}{3}$ of the optimal. The complexity of $MinSort_{Bk^3}$ for general permutations remains an open problem. For future research on this problem, we suggest the following directions. First, the algorithm devised for woven double-strip permutations is an $O(n^5)$ algorithm as it is dominated by the computation of a maximum matching. However, the graph from which the matching is obtained is a particular kind of graph and we speculate that matching is easier for such graphs. The resulting time complexity for sorting woven double-strip permutations may then be improved. In addition, studying the matching properties of arc graphs may lead to clues on how to handle the disabling of potential block-moves for arbitrary permutations. Another direction is to develop greedy hybrid algorithms from the heuristics that we have devised in this dissertation. We find that some counterexamples for a particular heuristic are solved by other heuristics—this motivates generalizing these heuristics to arrive at a more appropriate one.

Another promising heuristic for sorting by short block-moves is one based on an LIS of the permutation (see Chapter 7). This is motivated by the fact that another problem, sorting by single element insertion ($MinSort_{Bk^{1,n-1}}$), is solvable and the algorithm uses the LIS. $MinSort_{Bk^3}$ is a bounded variant of $MinSort_{Bk^{1,n-1}}$ so a corresponding LIS-based algorithm applies. We find that such an algorithm performs well experimentally. A future goal would be to pursue other bounded variants of $MinSort_{Bk^{1,n-1}}$, in particular, $MinSortBk^{1,\ell}$, for $\ell > 2$. LIS-based algorithms for such problems may prove as effective.

For $MinSort_{Sw^3}$, we find that the vector diagram model introduced in Chapter 8 is an appropriate basis for sorting algorithms. Good, though not tight, bounds on the short swap distance of a permutation and the short swap diameter of S_n are obtained based on the model. We also have an exact result; that is, we now know whether or not a permutation can be optimally sorted by

a sequence whose length equals the computed lower bound. Empirically, as shown in Section 9.4, the algorithm devised based on vectors work considerably better than expected (as implied by the theoretical approximation bounds) when tested with general permutations. An algorithm based on inversions (the permutation graph model) also proves effective particularly for large permutation lengths. $MinSort_{Bk^3}$ for general permutations remains an open problem. We speculate that fully understanding the effect of a short swap on vector-opposite elements is the key to resolving this problem. It appears, for instance, that a sorting algorithm should, if possible, always swap even vector-opposite elements. Although it is easy to identify such elements and the corresponding swaps, it is not immediately clear how to select the swaps so that even vector-opposite elements still exist later in the sort.

Determining the value of $Diam_{Sw^3}(n)$ is a particularly interesting open problem (see Section 8.5). The diameter for swaps is easily derived; the diameter for reversals is known [6]; and, a conjecture for the diameter of block-moves is available [5]. We have tabulated $Diam_{Sw^3}(n)$ for small n and have not been able to find a pattern to make a reasonable conjecture for the short swap diameter of S_n.

Finally, we find that corresponding results based on the models are available for two other $MinSort_P$ problems, sorting by short generators and sorting by ℓ-bounded swaps. Future research may embark on refining our models to more accurately represent these problems.

REFERENCES

[1] G. M. Adel'son-Vel'skiĭ and E. M. Landis, *An algorithm for the organization of information*, Soviet Mathematics Doklady, 3 (1962), pp. 1259–1263.

[2] M. Aigner and D. B. West, *Sorting by insertion of leading element*, Journal of Combinatorial Theory A, 45 (1987), pp. 306–309.

[3] S. B. Akers and B. Krishnamurthy, *A group-theoretic model for symmetric interconnection networks*, IEEE Transactions on Computers, 38 (1989), pp. 555–566.

[4] F. Annexstein, M. Baumslag, and A. L. Rosenberg, *Group action graphs and parallel architectures*, SIAM Journal on Computing, 19 (1990), pp. 544–569.

[5] V. Bafna and P. A. Pevzner, *Sorting permutations by transpositions*, in Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, 1995, pp. 614–623.

[6] ——, *Genome rearrangements and sorting by reversals*, SIAM Journal on Computing, 25 (1996), pp. 272–289.

[7] A. Caprara, *Sorting by reversals is difficult.* Submitted to the *SIAM Journal on Computing*, 1996.

[8] T. Chen and S. Skiena, *Sorting with fixed-length reversals*, Discrete Applied Mathematics, 71 (1997), pp. 269–295.

[9] D. A. Christie, *Sorting permutations by block-interchanges*, Information Processing Letters, 60 (1996), pp. 165–169.

[10] J. R. Driscoll and M. L. Furst, *Computing short generator sequences*, Information and Computation, 72 (1987), pp. 117–132.

[11] S. Even and O. Goldreich, *The minimum-length generator sequence is NP-hard*, Journal of Algorithms, 2 (1981), pp. 311–313.

[12] M. Farach, S. Kannan, and T. Warnow, *A robust model for finding optimal evolutionary trees*, Algorithmica, 13 (1995), pp. 155–179.

[13] W. M. Fitch and E. Margoliash, *Construction of phylogenetic trees*, Science, 155 (1967), pp. 279–284.

[14] N. Franklin, *Conservation of gene form but not sequence in the transcription antitermination determinants of bacteriophages λ, $\phi 21$, and $p22$*, Journal of Molecular Evolution, 181 (1985), pp. 75–84.

[15] W. H. Gates and C. H. Papadimitriou, *Bounds for sorting by prefix reversal*, Discrete Mathematics, 27 (1979), pp. 47–57.

[16] S. A. Guyer, L. S. Heath, and J. P. C. Vergara, *Subsequence and run heuristics for the sorting by transpositions problem.* Accepted to the Fourth DIMACS International Algorithm Implementation Challenge, 1995. To appear, 1997.

[17] E. Gyori and G. Turan, *Stack of pancakes*, Studia Scientarium Mathematicarum Hungarica, 13 (1978), pp. 133–137.

[18] M. Hall, *The Theory of Groups*, Chelsea Pub. Co., 1976.

[19] S. Hannenhalli and P. A. Pevzner, *Transforming cabbage into turnip (polynomial algorithm for sorting signed permutations by reversals)*, in Twenty-Seventh Annual ACM Symposium on the Theory of Computing, 1995, pp. 178–189.

[20] L. S. Heath and J. P. C. Vergara, *Some experiments on the sorting by reversals problem*, Tech. Rep. TR 95-16, Department of Computer Science, Virginia Polytechnic Institute and State University, 1995.

[21] M. H. Heydari, *The Pancake Problem*, PhD thesis, University of Texas, Dallas, 1993.

[22] M. H. Heydari and I. H. Sudborough, *On sorting by prefix reversals and the diameter of pancake networks*, in Parallel Architectures and Their Efficient Use, Lecture Notes in Computer Science, vol. 678, 1992.

[23] M. R. Jerrum, *The complexity of finding minimum-length generator sequences*, Theoretical Computer Science, 36 (1985), pp. 265–289.

[24] D. B. Johnson, *A priority queue in which initialization and queue operations take $O(\log\log D)$ time*, Mathematical Systems Theory, 15 (1982), pp. 295–309.

[25] H. Kaplan, R. Shamir, and R. Tarjan, *Faster and simpler algorithm for sorting signed permutations by reversals.* To appear in *Proceedings of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1997.

[26] J. Kececioglu and D. Sankoff, *Efficient bounds for oriented chromosome inversion distance*, in Fifth Annual Symposium on Combinatorial Pattern Matching, Lecture Notes in Computer Science, vol. 807, 1994, pp. 307–325.

[27] ——, *Exact and approximation algorithms for sorting by reversals with application to genome rearrangement*, Algorithmica, 13 (1995), pp. 180–210.

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

[29] P. A. Peterson and M. C. Loui, *The general maximum matching algorithm of Micali and Vazirani*, Algorithmica, 3 (1988), pp. 511–533.

[30] A. Pneuli, A. Lempel, and S. Even, *Transitive orientation of graphs and identification of permutation graphs*, Canadian Journal of Math, 23 (1971), pp. 160–175.

[31] D. Rotem and J. Urrutia, *Permutation graphs*, Networks, 12 (1982), pp. 429–437.

Appendix A

Tables: Short Block-Moves

This appendix presents all other tabulations made for the experiments for sorting by short block-moves. Tables A.1 and A.2 show the number of correct computations and average ratios for all short block-move heuristics with medium-sized permutations whose expected distance is $m = \sqrt{n}$. Table A.3 shows the average approximation ratios for permutations of length 10 broken down into the four different permutation types. Tables A.4, A.5, A.6, A.7, A.8, and A.9 show the average computed distances for lengths 10, 20, 30, 40, 100, and 150, respectively, for each heuristic, broken down into the different permutation types. Finally, Tables A.10, A.11, A.12, and A.13 present the average running times for the four heuristics; in these tables, ϵ means that the average running time is less than 0.05 seconds.

Table A.1: Number (and percentage) of correct Bk^3 distance computations ($m = \sqrt{n}$).

Length	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
10	50 (100%)	49 (98%)	50 (100%)	50 (100%)
20	50 (100%)	49 (98%)	50 (100%)	50 (100%)
30	50 (100%)	49 (98%)	50 (100%)	50 (100%)
40	50 (100%)	49 (98%)	50 (100%)	50 (100%)
50	50 (100%)	50 (100%)	50 (100%)	50 (100%)

Table A.2: Average ratio between computed and correct Bk^3 distance ($m = \sqrt{n}$).

Length	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
10	1.000	1.007	1.000	1.000
20	1.000	1.007	1.000	1.000
30	1.000	1.004	1.000	1.000
40	1.000	1.005	1.000	1.000
50	1.000	1.000	1.000	1.000

Table A.3: Average ratio between computed and correct Bk^3 distance ($n = 10$).

m	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
3	1.000	1.007	1.000	1.000
10	1.000	1.033	1.009	1.012
25	1.000	1.018	1.017	1.011
Random	1.000	1.019	1.006	1.019

Table A.4: Average computed Bk^3 distance ($n = 10$).

m	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
3	2.36	2.38	2.36	2.36
10	4.80	4.96	4.84	4.86
25	7.88	8.02	8.00	7.96
Random	11.22	11.40	11.26	11.40

Table A.5: Average computed Bk^3 distance ($n = 20$).

m	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
4	3.48	3.50	3.48	3.48
20	10.36	10.84	10.50	10.54
100	22.14	22.98	22.52	22.52
Random	47.88	48.48	48.16	48.92

Table A.6: Average computed Bk^3 distance ($n = 30$).

m	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
5	4.34	4.36	4.34	4.34
30	15.36	16.02	15.50	15.50
225	42.72	44.42	43.50	43.26
Random	107.88	108.94	108.22	109.72

Table A.7: Average computed Bk^3 distance ($n = 40$).

m	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
6	5.12	5.14	5.12	5.12
40	21.32	22.24	21.62	21.74
400	67.36	69.78	68.60	67.92
Random	188.86	190.38	189.42	189.92

Table A.8: Average computed Bk^3 distance ($n = 100$).

m	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
10	9.36	9.42	9.36	9.38
100	51.92	54.98	52.44	52.86
2500	—	284.14	279.32	276.52
Random	—	1233.32	1229.00	1239.12

Table A.9: Average computed Bk^3 distance ($n = 150$).

m	Greedy-Matching	Degree-Sort	Greedy-Hops	LISSort
12	11.54	11.62	11.54	11.54
150	78.62	83.22	79.38	80.14
5625	—	526.36	518.36	509.88
Random	—	2785.90	2778.18	2803.82

Table A.10: Timings (in seconds/permutation) for GreedyMatching.

Length	$m = \sqrt{n}$	$m = n$	$m = \lfloor n^2/4 \rfloor$	Random
10	ϵ	ϵ	ϵ	ϵ
20	ϵ	ϵ	0.30	2.88
30	ϵ	0.10	2.12	37.06
40	ϵ	0.28	8.86	238.90
50	ϵ	0.60	25.40	1229.50
100	0.20	7.10	—	—
150	0.64	32.42	—	—
200	1.48	105.28	—	—

Table A.11: Timings (in seconds/permutation) for DegreeSort.

Length	$m = \sqrt{n}$	$m = n$	$m = \lfloor n^2/4 \rfloor$	Random
10	ϵ	ϵ	ϵ	ϵ
20	ϵ	ϵ	ϵ	ϵ
30	ϵ	ϵ	ϵ	ϵ
40	ϵ	ϵ	ϵ	ϵ
50	ϵ	ϵ	ϵ	ϵ
100	ϵ	ϵ	ϵ	ϵ
150	ϵ	ϵ	ϵ	ϵ
200	ϵ	ϵ	ϵ	ϵ

Table A.12: Timings (in seconds/permutation) for LISSort.

Length	$m = \sqrt{n}$	$m = n$	$m = \lfloor n^2/4 \rfloor$	Random
10	ϵ	ϵ	ϵ	ϵ
20	ϵ	ϵ	ϵ	ϵ
30	ϵ	ϵ	ϵ	ϵ
40	ϵ	ϵ	ϵ	ϵ
50	ϵ	ϵ	ϵ	ϵ
100	ϵ	0.06	0.10	0.10
150	ϵ	0.20	0.28	0.30
200	0.10	0.44	0.62	0.62

Table A.13: Timings (in seconds/permutation) for GreedyHops.

Length	$m = \sqrt{n}$	$m = n$	$m = \lfloor n^2/4 \rfloor$	Random
10	ϵ	ϵ	ϵ	ϵ
20	ϵ	ϵ	ϵ	ϵ
30	ϵ	ϵ	ϵ	ϵ
40	ϵ	ϵ	ϵ	ϵ
50	ϵ	ϵ	ϵ	ϵ
100	ϵ	ϵ	ϵ	0.12
150	ϵ	ϵ	0.08	0.38
200	ϵ	ϵ	0.18	0.88

Appendix B

Tables: Short Swaps

This appendix presents all other tabulations made for the experiments for sorting by short swaps. Tables B.1 and B.2 show the number of correct computations and average ratios for all short swap heuristics with medium-sized permutations whose expected distance is $m = \sqrt{n}$. Tables B.3, B.4, B.5, B.6, B.7, B.8, and B.9 show the average computed distances for lengths 10, 20, 30, 40, 50, 100, and 150, respectively, for each heuristic, broken down into the different permutation types. Finally, Tables B.10 and B.11 present the average running times for the two heuristics devised for this problem (in these tables, ϵ means that the average running time is less than 0.05 seconds).

Table B.1: Number (and percentage) of correct Sw^3 distance computations ($m = \sqrt{n}$).

Length	GREEDY-INVERSIONS	GREEDY-VECTORS
10	50 (100%)	50 (100%)
20	50 (100%)	50 (100%)
30	50 (100%)	50 (100%)
40	50 (100%)	50 (100%)
50	50 (100%)	50 (100%)

Table B.2: Average ratio between computed and correct Sw^3 distance ($m = \sqrt{n}$).

Length	GREEDY-INVERSIONS	GREEDY-VECTORS
10	1.000	1.000
20	1.000	1.000
30	1.000	1.000
40	1.000	1.000
50	1.000	1.000

Table B.3: Average computed Sw^3 distance ($n = 10$).

m	Greedy-Inversions	Greedy-Vectors
3	2.72	2.72
10	6.12	6.00
25	8.12	7.84
Random	12.52	12.52

Table B.4: Average computed Sw^3 distance ($n = 20$).

m	Greedy-Inversions	Greedy-Vectors
4	3.76	3.76
20	12.40	12.28
100	27.08	26.88
Random	48.54	50.26

Table B.5: Average computed Sw^3 distance ($n = 30$).

m	Greedy-Inversions	Greedy-Vectors
5	4.48	4.48
30	18.84	18.36
225	46.40	46.32
Random	106.40	110.84

Table B.6: Average computed Sw^3 distance ($n = 40$).

m	Greedy-Inversions	Greedy-Vectors
6	5.60	5.60
40	25.24	24.84
400	72.84	74.52
Random	183.68	193.56

Table B.7: Average computed Sw^3 distance ($n = 50$).

m	GREEDY-INVERSIONS	GREEDY-VECTORS
7	6.60	6.60
50	32.12	31.76
625	99.56	101.24
Random	289.52	305.72

Table B.8: Average computed Sw^3 distance ($n = 100$).

m	GREEDY-INVERSIONS	GREEDY-VECTORS
10	9.68	9.64
100	63.96	63.12
2500	285.84	295.56
Random	1163.20	1231.92

Table B.9: Average computed Sw^3 distance ($n = 150$).

m	GREEDY-INVERSIONS	GREEDY-VECTORS
12	11.32	11.32
150	97.56	96.08
5625	524.56	546.32
Random	2636.90	2785.10

Table B.10: Timings (in seconds/permutation) for GreedyInversions.

Length	$m = \sqrt{n}$	$m = n$	$m = \lfloor n^2/4 \rfloor$	Random
10	ϵ	ϵ	ϵ	ϵ
20	ϵ	ϵ	ϵ	ϵ
30	ϵ	ϵ	ϵ	ϵ
40	ϵ	ϵ	ϵ	ϵ
50	ϵ	ϵ	ϵ	0.10
100	ϵ	0.06	0.34	1.34
150	ϵ	0.18	1.40	6.38
200	ϵ	0.40	3.80	19.74

Table B.11: Timings (in seconds/permutation) for GreedyVectors.

Length	$m = \sqrt{n}$	$m = n$	$m = \lfloor n^2/4 \rfloor$	Random
10	ϵ	ϵ	ϵ	ϵ
20	ϵ	ϵ	ϵ	ϵ
30	ϵ	ϵ	ϵ	ϵ
40	ϵ	ϵ	ϵ	ϵ
50	ϵ	ϵ	ϵ	ϵ
100	ϵ	ϵ	ϵ	ϵ
150	ϵ	0.08	0.12	0.14
200	ϵ	0.18	0.24	0.34

Index

VITA

John Paul C. Vergara was born in Manila, Philippines on October 27, 1965. He obtained his B.S. in Mathematics and Computer Science from the Ateneo de Manila University in 1986. He obtained his graduate degrees from Virginia Tech: an M.S. in Computer Science in 1990 and a Ph.D. in Computer Science in 1997. He works as a professor of Computer Science at the Ateneo de Manila University.