University of North Florida
UNF Digital Commons

UNF Graduate Theses and Dissertations | Student Scholarship

2020

Sorting by Strip Moves and Strip Swaps

Chandrika Pandurang Rao
University of North Florida, chandrika.k.rao@gmail.com

Follow this and additional works at: https://digitalcommons.unf.edu/etd

 Part of the Other Computer Engineering Commons

Suggested Citation

Rao, Chandrika Pandurang, "Sorting by Strip Moves and Strip Swaps" (2020). *UNF Graduate Theses and Dissertations*. 982.
https://digitalcommons.unf.edu/etd/982

This Master's Thesis is brought to you for free and open access by the Student Scholarship at UNF Digital Commons. It has been accepted for inclusion in UNF Graduate Theses and Dissertations by an authorized administrator of UNF Digital Commons. For more information, please contact Digital Projects.
© 2020 All Rights Reserved

SORTING BY STRIP MOVES AND STRIP SWAPS

by

Chandrika Rao

A thesis submitted to the
School of Computing
in partial fulfilment of the requirements for the degree of

Master of Science in Computer and Information Sciences

UNIVERSITY OF NORTH FLORIDA
SCHOOL OF COMPUTING

December 2020

Copyright (©) 2020 by Chandrika Rao

All rights reserved. Reproduction in whole or in part in any form requires the prior written permission of Chandrika Rao or designated representative.

The thesis "Sorting by Strip Moves and Strip Swaps" submitted by Chandrika Rao in partial fulfillment of the requirements for the degree of Master of Science in Computing and Information Sciences has been

Approved by the thesis committee: Date:

Dr. Asai Asaithambi
Thesis Advisor and Committee Chairperson

Dr. Swapnoneel Roy
Committee Member

Dr. Xudong Liu
Committee Member

ACKNOWLEGEMENTS

I would like to offer my sincerest thanks and gratitude to my thesis advisors, Dr. Asai Asaithambi and Dr. Swapnoneel Roy for their continued support and mentoring throughout the development of this thesis and for the duration of my time at the University of North Florida. I am particularly grateful to Dr. Asaithambi for helping me explore this vast topic in an organized manner, for the detailed reviews of my thesis drafts, and for guidance on the use of LaTeX. I would also like to thank Dr. Xudong Liu for his remarkable comments and suggestions during my thesis prospectus and defense presentations.

My sincere thanks to Dr. Elfayoumy for assisting me with all my academic questions and concerns.

Most importantly, thank you Almighty for helping me achieve my dream and for everything else. And finally, the most special thanks to my parents and family for imparting the values and principles which have made me what I am today, and my husband for supporting me throughout our time together, especially during my studies. Thank you!

CONTENTS

FIGURES

TABLES

ABSTRACT

Genome rearrangement problems in computational biology [19, 29, 27] and zoning algorithms in optical character recognition [14, 4] have been modeled as combinatorial optimization problems related to the familiar problem of sorting, namely transforming arbitrary permutations to the identity permutation. The term *permutation* is used for an arbitrary arrangement of the integers $1, 2, \cdots, n$, and the term *identity permutation* for the arrangement of $1, 2, \cdots, n$ in increasing order. When a permutation is viewed as the string of integers from 1 through n, any substring in it that is also a substring in the identity permutation will be called a *strip*. The objective in the combinatorial optimization problems arising from the applications is to obtain the identity permutation from an arbitrary permutation in the least number of a particular chosen strip operation. Among the strip operations which have been investigated thus far in the literature are strip moves, transpositions, reversals, and block interchanges [16, 2, 25, 11, 34]. However, it is important to note that most of the existing research on sorting by strip operations has been focused on obtaining hardness results or designing approximation algorithms, with little work carried out thus far on the implementation of the proposed approximation algorithms. This research starts with implementing two existing algorithms [5, 34] and as the main contributions, provides two new algorithms for sorting by strip swaps: 1) A *greedy algorithm* in which each strip swap reduces the number of strips the most, and puts maximum strips in their correct positions; 2) Another algorithm that uses the strategy of bringing closest consecutive pairs together called the closest consecutive pair (CCP) algorithm. The approximation ratios for the implemented algorithms are also experimentally estimated.

CHAPTER 1

INTRODUCTION

Several problems in the field of comparative genomics have been modeled via combinatorial optimization problems related to the familiar problem of sorting, namely transforming an arbitrary permutation to the identity permutation [19, 29, 27]. We use the term *permutation* for an arbitrary arrangement of the integers $1, 2, \cdots, n$, and the term *identity permutation* for the arrangement of $1, 2, \cdots, n$ in increasing order. We are particularly interested in those permutations (viewed as strings) which may already contain substrings that are also substrings in the identity permutation. Such substrings will be called *strips*.[1] This means that the elements of a strip within an arbitrary permutation need not be considered individually when we try to obtain the identity permutation. Instead, sorting can be done by operations carried out on strips (entire substrings). The objective is to obtain the identity permutation from an arbitrary permutation in the least number of a particular chosen strip operation, which makes all of these problems combinatorial optimization problems, most of which have been proven as NP-hard. Among the strip operations which have been investigated thus far in the literature are strip moves, transpositions, reversals, block interchanges [16, 2, 25, 11] and strip exchanges [34, 35]. These problems have been named sorting by block moves or block sorting [25], sorting by transpositions [2], sorting by block interchanges [11] sorting by strip exchanges [34, 35], and others [26, 3].

It is important to note that most of the existing research on sorting with strip

[1] sometimes called *blocks*, but these terms will be differentiated later in the thesis.

operations has been focused on obtaining computational complexity results or designing approximation algorithms and establishing approximation ratios for them theoretically. Typical proofs of these results have been based on considering relationships between different strips and the positions where they occur within the permutations, while there has been little work carried out thus far on the implementation of the proposed approximation algorithms. We believe strongly that practical implementation is key to identifying such relationships between strips and their positions within permutations that might not have been theoretically possible before. We also believe implementation will provide means for discovering newer approaches and algorithms to tackle sorting by strip operations in general. With this in mind, this thesis research will make the following contributions: 1) implement two previously developed algorithms, known as *sorting by block deletions* and *sorting by strip swaps*; 2) implement an exact algorithm using brute-force search to find the minimum number of strip moves or swaps needed (for small-sized permutations); 3) implement a *greedy algorithm* newly developed in this thesis, for sorting by strip swaps; 4) implement an algorithm newly developed in this thesis, called the closest consecutive pair (CCP) algorithm; and 5) experimentally estimate the approximation ratios for all the implemented approximation algorithms for sorting by strip swaps using the exact algorithm as the reference. For ease of reference, we begin by introducing the terminology, along with some basic concepts and ideas that will be used in the rest of the thesis.

1.1 Strips in a Permutation

An arbitrary permutation of the integers $1, 2, \cdots, n$ is denoted by π, where each number occurs exactly once, and the identity permutation, in which the integers $1, 2, \cdots, n$ are in increasing order is denoted by id. As previously mentioned, substrings in π that are also substrings in id will be called strips. For example, in

the permutation π : 8 2 5 6 3 9 1 4 7 on nine elements, there are eight strips, and [5 6] is a strip containing more than a single element, all other strips containing one element each. The identity permutation 1 2 3 4 5 6 7 8 9 consists of the single strip. Figure 1.1 shows these two permutations with their strip structures.

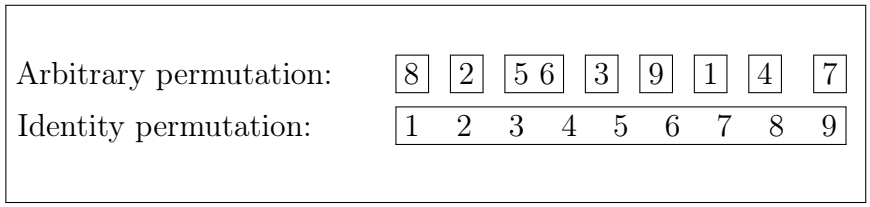

Figure 1.1: Strips in a Permutation

1.2 Sorting By Strip Moves

The operation of removing a strip from its current position and placing it elsewhere in the permutation is called a *strip move*. Consider the starting permutation 8 7 2 4 5 3 6 1 9 which consists of eight strips 8, 7, 2, 4 5, 3, 6, 1 and 9. In order to clearly show the strip structure, we enclose each strip as a separate box and write the permutation in the form 8 7 2 4 5 3 6 1 9. Upon (re)moving the strip 4 5 from its current position and placing it before the strip 6 in the above permutation, we obtain the permutation that can be shown with its new strip structure in the form 8 7 2 3 4 5 6 1 9. Note that the number of strips has been reduced from 8 to 5 after one strip move. Now, within this new permutation, moving the strip 2 3 4 5 6 to occupy the position before the strip 7 yields the permutation with the strip structure 8 2 3 4 5 6 7 1 9. Note that the second move has reduced the number of strips from 5 to 4. Next, within the most recently obtained permutation, moving the strip 8 to the position before the strip 9 yields 2 3 4 5 6 7 1 8 9 The third move has reduced the number of strips from 4 to 3. Finally, moving the strip 1 before the strip 2 3 4 5 6 7 yields 1 2 3 4 5 6 7 8 9, which is the identity permutation that

consists of a single strip. The above-described process of transforming an arbitrary permutation π consisting of several strips to the identity permutation *id* consisting of a single strip, through a sequence of strip moves is called *sorting by strip moves*. The sequence of strip moves is known as a *strip move schedule*. The schedule for the above example is shown in the Figure 1.2. The length of a strip move schedule is the number of strip moves in the schedule.

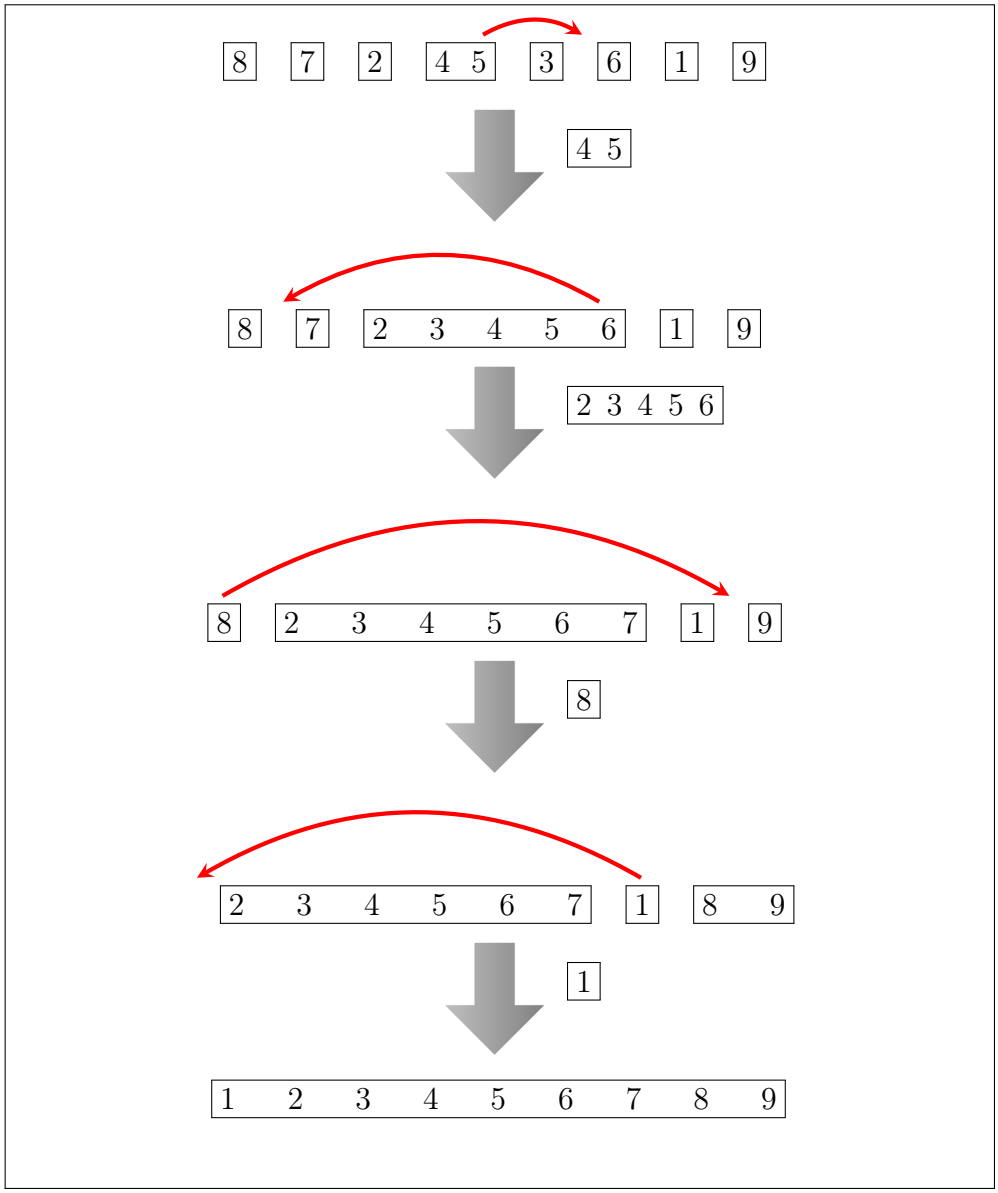

Figure 1.2: Strip Move Schedule for Sorting by Strip Moves

1.3 Sorting By Strip Swaps

The operation of interchanging the positions of two different strips in a permutation is called a *strip swap*. This process of transforming an arbitrary permutation π consisting of several strips to the identity permutation *id* consisting of a single strip, through a sequence of strip swaps is known as *sorting by strip swaps*. Also, the sequence of strip swaps is known as a *strip swap schedule*. Figure 1.3 illustrates a strip swap schedule for the same starting permutation, [8] [7] [2] [4 5] [3] [6] [1] [9] consisting of 8 strips, which we used in the previous section. The length of a strip swap schedule is the number of strip swaps in the schedule.

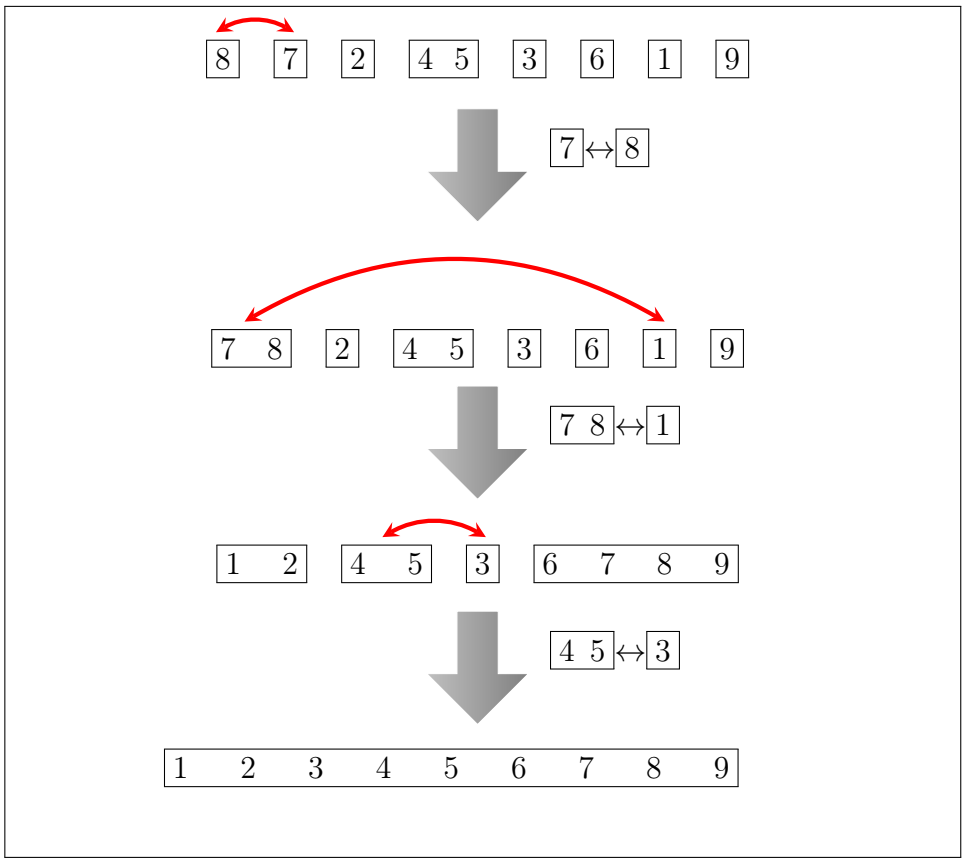

Figure 1.3: Strip Swap Schedule for Sorting by Strip Swaps

This thesis research began with the development of an implementation framework, which was then validated by testing and verifying the results of Bein et al. [4] which

presented a 2-Approximation algorithm for strip sorting. On the other hand, while implementing the algorithm of Roy et al. [34, 35] for sorting by strip swaps, it became evident that the algorithm does not take into account several scenarios that could arise in the strip structure. This experience led to the development of two new approaches to handle sorting by strip swaps, which constitute the main contributions of this thesis.

The remainder of the thesis document is organized as follows. Chapter 2 gives the background and motivation for this research as well as a summary of the previous work done. Chapter 3 discusses the existing approximation algorithms for sorting by strip moves and sorting by strip swaps. Chapter 4 introduces the two algorithms that have been newly developed in this thesis. These algorithms are named: 1) the Greedy algorithm for sorting by strip swaps; and 2) the Closest Consecutive Pairs (CCP) algorithm for sorting by strip swaps. Chapter 5 presents the results of the experiments conducted on the existing and new algorithms comparing their performances. Finally, Chapter 6 provides some directions for future research that could help expand the scope of this work.

CHAPTER 2

BACKGROUND

The underlying application domains that lead to the sorting problems investigated in this thesis are *comparative genomics* [15] and optical character recognition [14]. Of these, we explain the connection between our research and the ongoing research related to comparative genomics, a field of biological research. A genome is the entire DNA of a living organism, and a genome consists of chromosomes, which in turn are made up of genes. In comparative genomics, the genes of different species are compared to gain some insights as to how far they are separated genetically. Research on the genomes of different species shows that pairs of different species may have essentially the same set of genes, but the orderings of the genes in the two species may differ [18, 31]. For instance, the study by Palmer and Herbon indicates that cabbage and turnip are closely related to each other genetically, and many genes of the two species are 99–99.9 percent identical [32], while the orders in which the genes appear in the two species are different. Similarly, an article from the National Institute of General Medical Sciences [30], indicates that "all living things evolved from a common ancestor," and asserts that the human and mouse genomes are about 85 percent similar, containing the same set of genes but in different orders. This suggests that genome rearrangement events can be used to trace the evolutionary path among genomes. Genome rearrangement events refer to the mutations (alterations) that affect the orderings of the genes. Each study of genome rearrangements is actually solving a combinatorial optimization problem to find a sequence of rearrangements that transform one genome into another [33].

For modeling genome rearrangement problems, the order of genes in two arbitrary organisms is represented by permutations. Thus, given two permutations that represent the ordering of genes in the genomes of two different organisms, the basic task involved in understanding genome rearrangement problems is to find the shortest sequence of rearrangement operations that transforms one permutation into another.

The mutations leading to genome rearrangements occur in many common forms, which we refer to as *primitives*. Some well-defined primitives include: *deletion* (a part of a genome is lost) [21, 24]; *insertion* (a part is added) [37]; *duplication* [36]; *reversal* or *inversion* (a part is reversed) [1]; *transpositions* (a part is picked and placed elsewhere) [2]; *block interchanges* (where the positions of two parts are interchanged) [10]; *block moves* (a special kind of transposition) [26]; prefix reversals [13], and strip exchanges [35]. At times, a combination of more than one primitive is used to determine the distance. For instance, *transreversals* are combinations of transpositions and reversals [17]. Thus, the methodology to study genome rearrangements has traditionally amounted to using a single or a combination of well-defined primitives to transform one genome into another. The number of primitive steps needed to transform one genome into another is a measure of the evolutionary distance between the two species.

Computer scientists model the genome rearrangement problems based on the different primitives of reversals, transpositions, block moves, and block interchanges as corresponding combinatorial optimization sorting problems [1, 2, 10, 26]. The following assumptions are made in such models: i) genomes are considered as ordered sequences (or permutations) of genes or other segments; ii) the individual gene segments in the genomes may be represented by signed or unsigned integers as appropriate; iii) there is no duplication or deletion of segments when transforming from one genome to another; and iv) pairs of genomes which only differ in the

order in which segments occur are considered. Under these assumptions, genome rearrangement amounts to transforming a given permutation (starting species) to a target permutation (target species). Additionally, we think of the starting species as represented by an arbitrary permutation (may consist of signed or unsigned integers), and the target species as the sorted or the identity permutation (only consisting of unsigned integers). For instance, a segment of the human genome may be modeled as the permutation 4 6 1 7 2 3 5 8. In order to study the evolutionary distance between the human and the mouse, the corresponding segment in the mouse may be modeled as the permutation 1 2 3 4 5 6 7 8. Similarly, a segment of the genome of cabbage modeled permutation $1\ -5\ 4\ -3\ 2$ is transformed to the permutation 1 2 3 4 5 as the corresponding segment in the genome of turnip to study the evolutionary distance between the two species. Hence, genome rearrangement problems become sorting problems in which the goal is to sort in such a way that the number of these special primitive operations required is minimized. In other words, genome rearrangement problems translate to combinatorial optimization sorting problems.

Before we proceed to the next section which will provide an account of previous work related to sorting by the various primitives, we want to clarify the use of the term *block*, which has been used in the literature to mean two different things by different groups of researchers, as indicated in a footnote in the previous chapter. The term *block* has been used by some researchers to refer to *any* substring in the starting permutation [10, 23, 6, 2], and these works have focused on sorting by *block interchanges*. The same term, *block*, has also been used by other researchers to refer to *maximal substrings in the starting permutation that are also substrings in the identity permutation* [3, 25, 4, 26, 28, 5, 29, 38], and these works have focused on sorting by *block moves* and they also refer to the sorting problem as *block sorting*. At the same time, the term *strip* is also used by [27, 34, 35] to refer

to the maximal substrings in the starting permutation that are also substrings in the identity permutation. In this thesis, we will use the term *strip* to refer to the already-sorted maximal substrings in a permutation.

2.1 Previous work

Several studies relating to the design and analysis of algorithms for sorting by various primitives have been reported in the literature. Many of these combinatorial optimization problems have been proven to be NP-hard. Specifically, Caprara et al. [9] demonstrate that sorting by reversals is hard; Bulteau et al. prove that sorting by transpositions [7] and sorting by prefix reversals [8] are hard; Bein et al. [3] show that sorting by strip moves is hard. Finally, the computational complexity of sorting by strip swaps still remains an open question. This implies that there are no known efficient algorithms for solving the problem of sorting by these primitives, and it is quite unlikely that such an algorithm exists. However, sorting by block interchanges has been proven to be polynomially solvable, and an $O(n^2)$ algorithm exists for solving this problem [10].

Hence, the search for *approximation algorithms*, for sorting by the primitives of reversals, transpositions, and strip moves, has been an active area of research for many years now. In computer science and operations research, approximation algorithms are efficient algorithms that find approximate solutions to optimization problems (in particular NP-hard problems) with provable guarantees on the distance of the returned solution to the optimal one [40]. An algorithm A is said to be a ρ-approximation algorithm if,

$$\frac{\text{approximate solution obtained by the algorithm } A}{\text{the optimal solution}} \leq \rho. \tag{2.1}$$

A ρ-approximation algorithm guarantees that the approximate solution is no bigger

than ρ times the optimal solution.

In terms of previous work on approximation algorithms, Bafna et al. [1], in one of the earlier studies in the areas of genome rearrangements and sorting, describe an approximation algorithm for sorting by reversals. Hannenhalli et al. [16] study sorting of signed permutations by reversals, a problem that adequately models rearrangements in small genomes. Bafna et al. [2], in their work address the problem of sorting by transpositions. The first nontrivial approximation algorithm to the strip sorting problem is presented by Mahajan et al. [26], where they present a strip merging algorithm. Few years later, they also give a 2-approximation algorithm for sorting by strip moves [25]. Bein et al. [3] propose a 2-approximation algorithm based on finding an optimal sequence of absolute strip deletions [14, 20, 22]. Roy et al. [34] study the strip swaps, identify a new lower bound for the number of strip swaps, and design a 2-approximation algorithm for the problem.

2.2 Motivation

It is evident that a significant amount of theoretical research has been carried out during the past few decades in this area, along with the conceptualization and analysis of several approximation algorithms. Work related to validating the approximation algorithms by implementing the approximation algorithms proposed has been somewhat limited. To our best knowledge, Turlapaty [38] is the only work reported in the literature that is focused on implementation and experimental performance analysis of a few algorithms for sorting by strip moves. Turlapaty [38] also brings along a new greedy algorithm for sorting by strip moves. Our motivation for the research in this thesis is to take the theoretical research in computing to the next level to help computational biology researchers by providing the tools they may use in the future to study genome rearrangements. The research in this thesis builds on the goals of 1) validating the absolute block deletion algorithm

for sorting by strip moves [4]; 2) validating the 2-approximation algorithm of Roy et al. [34, 35] for sorting by strip swaps; and 3) developing new approximation algorithms for sorting by strip swaps.

2.3 Scope of the Present Work

Although several approximation algorithms for *primitive* operations on strips or blocks such as transpositions [2], block interchanges [10, 23], reversals or inversions [6], and strip moves [25, 26, 4], and strip swaps [34, 35] have been proposed and analyzed theoretically, little work has been carried out thus far in implementing, validating, and analyzing the proposed approximation algorithms. At this point, we would like to mention that all permutations we will consider in this thesis consist only of unsigned integers, even though signed integers may be relevant in appropriate contexts in computational biology research. Our focus in this thesis has been on two particular strip operations, namely, strip moves and strip swaps. An implementation framework for sorting by strip moves and sorting by strip swaps has been developed in the Python programming language. This framework was used to implement and analyze sorting by strip moves using the strip deletion algorithm of [4] and sorting by strip swaps [34], along with an exact brute-force algorithm that we have developed in order to determine the approximation ratios. Additionally, this thesis research has also led to the development of two new algorithms for sorting by strip swaps. The first algorithm strategically selects a swap at each step such that it minimizes the number of strips in the permutation and puts maximum elements in their correct positions. This strategy will be referred to as the greedy algorithm for the rest of the thesis. The second algorithm suggests a promising approach to select a good swap at every step to bring together the pair of consecutive strips that are closest. This approach will be referred to as the Closest Consecutive Pair (CCP) algorithm for the rest of the thesis.

CHAPTER 3

EXISTING ALGORITHMS

In this chapter, an existing algorithm for sorting by strip moves and an existing algorithm for sorting by strip swaps are described. The algorithm for sorting by strip moves, developed by Bein et al. [5], which we will call the *strip deletion algorithm*, is based on a sequence of strip deletions. The algorithm for sorting strip by swaps, developed by Roy et al. [34], which we will call the *cycle graph algorithm*, uses the idea of *cycle graphs*.

3.1 The Strip Deletion Algorithm for Sorting by Strip Moves

Bein et al. [5] provide a quadratic time, 2-approximation algorithm for sorting by strip moves. In their work, the term *block* represents a *strip*. Their method first finds the minimum length sequence of strip deletions to transform a list of distinct integers (permutation) into a monotone increasing sublist, and then derives a sequence of strip moves based on the sequence of strip deletions. For example, a minimum length strip deletion sequence for the list (2 4 1 5 3) is ($\boxed{1}$,$\boxed{4\ 5}$). In other words, deleting $\boxed{1}$ first, followed by the deletion of $\boxed{4\ 5}$ results in the monotone increasing list (2 3). Related to the problem of finding the minimum length strip deletion sequence for a list x of distinct integers is the problem of finding what is known as the *complete strip deletion sequence* for x. The complete strip deletion sequence corresponds to a sequence of strip deletions leading to an empty list. For example, the list (4 1 5) becomes the empty list when $\boxed{1}$ is deleted first and $\boxed{4\ 5}$ is deleted next. Thus, the complete deletion sequence for (4 1 5) is ($\boxed{1}$, $\boxed{4\ 5}$). Also,

note that for the list (2 4 1 5 3), the complete strip deletion sequence for (4 1 5) yields the monotone increasing list of (2 3). In general, if $x = (x_1\ x_2\ \cdots\ x_n)$ is a given list of distinct integers, then finding a minimum length strip deletion sequence that results in the monotone increasing sublist $(x_{i_1}\ x_{i_2}\ \cdots\ x_{i_k})$ corresponds to finding the complete strip deletion sequences for the sublists $(x_1\ x_2\ \cdots\ x_{i_1-1})$, $(x_{i_1+1}\ x_{i_1+2}\ \cdots\ x_{i_2-1}), \cdots, (x_{i_{k-1}+1}\ x_{i_{k-1}+2}\ \cdots\ x_{i_k-1})$. Based on this observation, sorting by strip deletions algorithm of Bein et al. [5] can be described as consisting of the following stages.

Stage-1: The minimum length of any complete strip deletion sequence. With the sublist $(x_i\ x_{i+1}\ \cdots\ x_j)$ denoted by $x_{i\cdots j}$, the minimum length of any complete strip deletion sequence for the sublist $x_{i\cdots j}$ is calculated as $t_{i,j}$ for $1 \le i \le j \le n$, where

$$t_{i,j} = \begin{cases} \min \begin{pmatrix} 1 + t_{i+1,j}, \\ t_{i+1,\ell-1} + t_{\ell,j} \end{pmatrix}, & \text{if } \exists \ell \in \{i+1, \cdots, j\} \text{ such that } x_\ell = x_i + 1; \\ 1 + t_{i+1,j}, & \text{otherwise.} \end{cases} \tag{3.1}$$

Suppose the given list is $x = (2\ 4\ 1\ 5\ 3)$. Then, as the first stage calculates the minimum length of any complete strip deletion sequence for the sublists $x_{i\cdots j}$ for $1 \le i \le j \le n$, it stores the results as $t_{i,j}$. For instance, corresponding to $i = 1$, the minimum lengths of any complete strip deletion sequence for the sublists $x_{1\cdots 1} = (2)$, $x_{1\cdots 2} = (2\ 4)$, $x_{1\cdots 3} = (2\ 4\ 1)$, $x_{1\cdots 4} = (2\ 4\ 1\ 5)$, and $x_{1\cdots 5} = (2\ 4\ 1\ 5\ 3)$ need to be calculated. It is clear that the minimum length of a complete strip deletion sequence for $x_{1\cdots 1} = (2)$ is 1 because it just contains the single strip 2. Similarly, the minimum length of a complete strip deletion sequence for $x_{1\cdots 2} = (2\ 4)$ is 2 and the minimum length of a complete strip deletion sequence for $x_{1\cdots 3} = (2\ 4\ 1)$ is 3. However, for the sublist $x_{1\cdots 4} = (2\ 4\ 1\ 5)$, the deletion of strip 1 results in the

formation of the strip (4 5), and hence the minimum length of a complete strip deletion sequence for $x_{1\cdots4}$ is also 3. Finally, for the sublist $x_{1..5} = (2\ 4\ 1\ 5\ 3)$, the deletion of strip (1), and the deletion of the strip (4 5) yield the final strip (2 3), which results in the minimum length of a complete strip deletion sequence for $x_{1\cdots5}$ equal to 3 as well. These are obtained using (3.1) in the strip deletion algorithm [5], which finally yields the matrix $[t_{i,j}]$ as shown below.

$$[t_{i,j}] = \begin{bmatrix} 1 & 2 & 3 & 3 & 3 \\ 0 & 1 & 2 & 2 & 3 \\ 0 & 0 & 1 & 2 & 3 \\ 0 & 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Stage-2: Finding the strip deletion sequence. With $t_{i,j}$, the the minimum length of any complete strip deletion sequence for the sublist $x_{i\cdots j}$ available from the first stage, the strip deletion sequence for the given list x can be obtained by first constructing a directed, weighted, acyclic graph G with the following properties: 1) there is one node in G for each $i \in \{0, 1, \cdots, n+1\}$, with $x_0 = 0$ and $x_{n+1} = n+1$; 2) there is an edge (i, j) from node i to node j if and only if $i < j$ and $x_i < x_j$ are both satisfied; and 3) the weight of this edge (i, j) is $t_{i+1,j-1}$. Then, stage-2 finds the minimum-weighted path $(0, i_1, i_2, \cdots, i_k, n+1)$ in G, whose vertices correspond to the monotone increasing sequence $(x_{i_1}, x_{i_2}, \cdots, x_{i_k})$.

Stage-3: Strip Sorting from the Strip Deletion Sequence. Before we proceed to describe the next stage, let us see what happens at the end of Stage-2 for our example permutation $x = (2\ 4\ 1\ 5\ 3)$. For this permutation, in Stage-2, the directed path of minimum weight from node 0 to node 6 (namely $n+1$ for $n = 5$) in the weighted, directed acyclic graph G needs to be determined. A sampling of the various directed paths from 0 to 6 with the corresponding weight for each path

is shown for reference in Figure 3.1, from which it is evident that the minimum weight path from 0 to $n+1$ is $(0, 2, 4, 5, 6)$ with a weight of 2. Also, the nodes(or strips) that are not present in the minimum weight path are $\boxed{1}$ and $\boxed{3}$, which provide the *complete block deletion sequence* A for this permutation as ($\boxed{1}$, $\boxed{3}$).

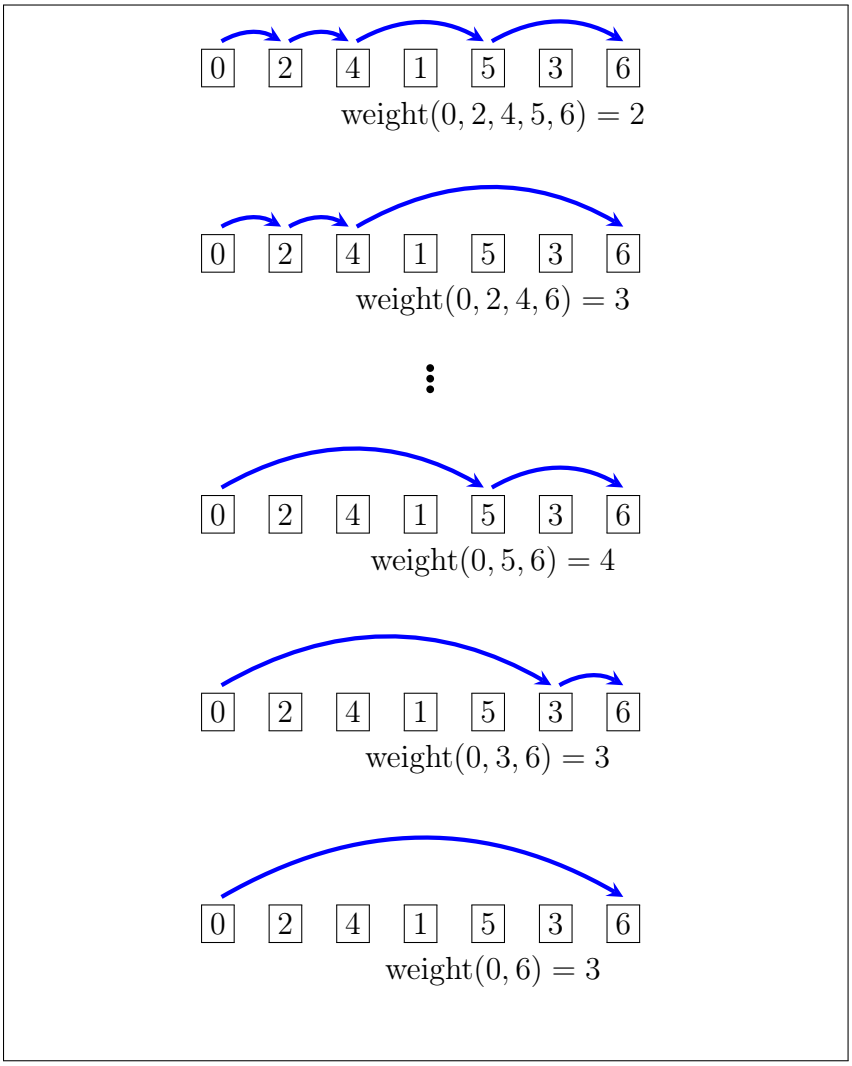

Figure 3.1: Weighted paths from 0 to 6 in $x = (0\ 2\ 4\ 1\ 5\ 3\ 6)$

Thus, from the first two stages, the minimum length strip deletion sequence A for the starting list x may be determined. Let the strips in the sequence be named $A_1, A_2, \cdots, A_m$. Then, Bein et al [5] provide the following algorithm for translating each member of the strip deletion sequence to a corresponding strip move.

Data: x: The input permutation; $A_1, A_2, \cdots, A_m$: strip deletion sequence
Result: Sorted permutation
Let $x^0 \leftarrow x$
for $t \leftarrow 1$ **to** m **do**
  Let B_t be the strip of x^{t-1} that contains $first(A_t)$.
  if $first(A_t) \neq first(B_t)$ **then**
    $x^t \leftarrow x^{t-1}$
  else
    Move B_t to the position after $first(B_t) - 1$
  end
end

Algorithm 1: Sorting by Strip Moves via Strip Deletions

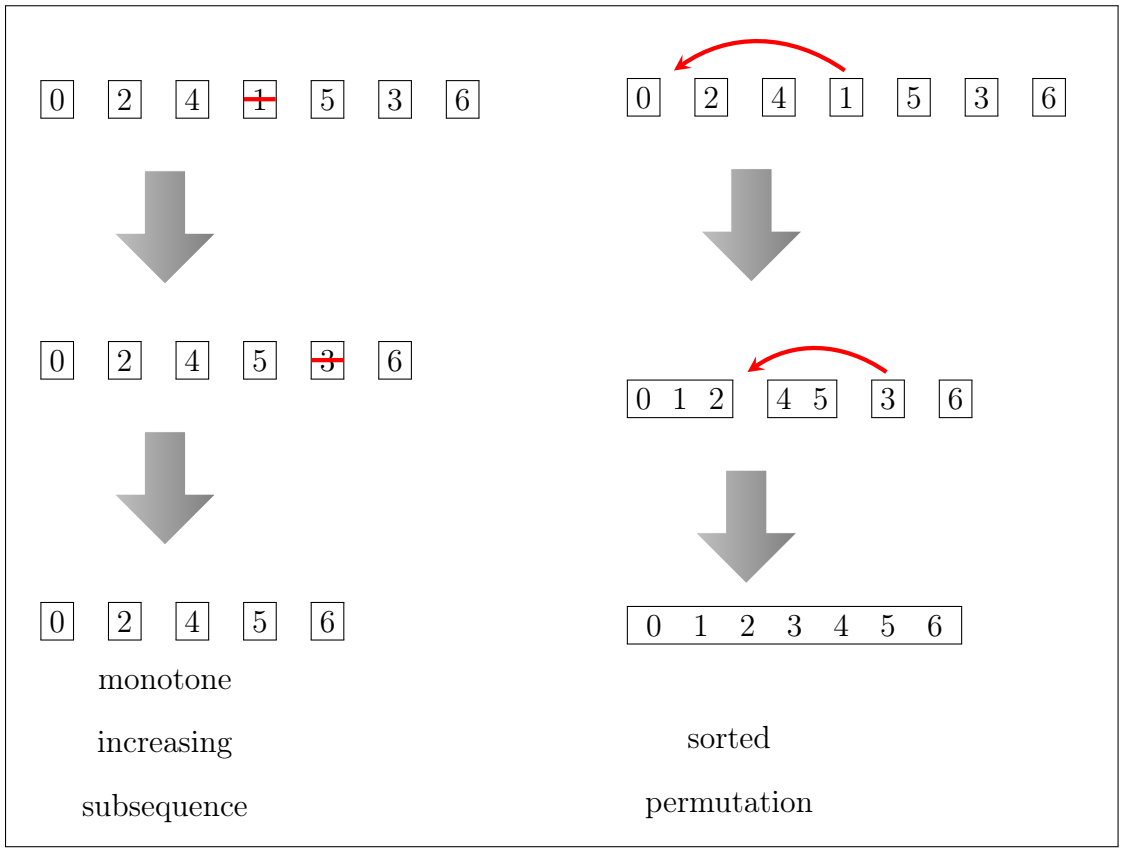

Figure 3.2: Sorting (2 4 1 5 3) by Strip Deletions

Shown in Figure 3.2 is the execution of Stage-3 of the strip deletion algorithm for the permutation (0 2 4 1 5 3 6) under consideration. The block deletion sequence for this permutation is ($\boxed{1}$,$\boxed{3}$). The deletion of $\boxed{1}$ corresponds to moving $\boxed{1}$ to the position after $\boxed{0}$, which yields the modified permutation (0 1 2 4 5 3 6). Next, deleting $\boxed{3}$ corresponds to the moving $\boxed{3}$ to the position after $\boxed{0\ 1\ 2}$, which yields the identity permutation.

3.2 The Cycle Graph Algorithm for Sorting by Strip Swaps

In the work of Roy et al. [34, 35], the problem of sorting by strip swaps is treated in a manner similar to the work of Christie [12]. Roy et al. [34, 35] present a new lower bound for the number of strip swaps needed to sort by strip swaps and also devise a 2-approximation algorithm using the concepts of cycle graphs [1], simple permutations [16], and 2-moves and 0-moves [2]. A strip swap operation involves interchanging the positions of two strips. Ordinarily the swaps are performed so that the operation results in several strips being combined to form longer strips. A sequence of strip swaps can then be used to transform an arbitrary permutation that may contain several strips to the identity permutation that is just a single strip. For ease of understanding, some of the terminology used by Roy et al. [34, 35] is included here as presented in [34]. We assume throughout this section that π is an arbitrary permutation of the integers $1, 2, \cdots, n$.

Cycle Graph. The *cycle graph of* π, denoted by $G(\pi)$, is a directed edge color graph with the vertex set $\{0, 1, 2, ..., n, n+1\}$ and the edge set defined as follows; for all $1 \leq i \leq n+1$, gray edges are directed from $i-1$ to i and black edges from π_i to π_{i-1}.

Alternating Cycles. An *alternating cycle* of a cycle graph is a cycle where each pair of adjacent edges are of different colors; the length of an alternating cycle is defined to be the number of black edges in the cycle. Also, an alternating cycle

with k black edges is referred to as a k-cycle.

As an example, the cycle graph for the permutation (3 2 1 4) is shown in Figure 3.3. From Figure 3.3, it can be seen that this cycle graph contains three alternating cycles, of which two are 2-cycles and one is a 1-cycle. These cycles are shown as Figure 3.4 for ease of reference later.

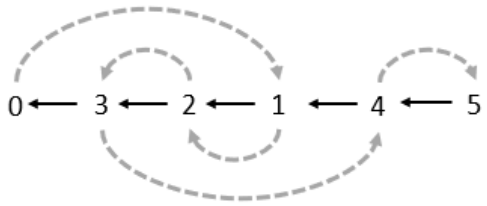

Figure 3.3: Cycle graph for (3 2 1 4) [2, 34]

$(0 \dashrightarrow 1 \rightarrow 2 \dashrightarrow 3 \rightarrow 0) \quad (1 \dashrightarrow 2 \rightarrow 3 \dashrightarrow 4 \rightarrow 1) \quad (4 \dashrightarrow 5 \rightarrow 4)$

Figure 3.4: Alternating cycles for (3 2 1 4) cycle graph [2, 34]

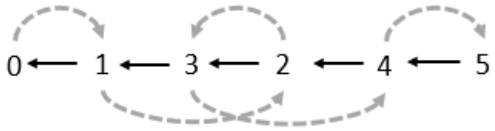

Figure 3.5: Cycle graph for (1 3 2 4) [2, 34]

As another example, the cycle graph for the permutation (1 3 2 4) is shown in Figure 3.5, from which it is evident that it contains three alternating cycles, of which two are 1-cycles and one is a 3-cycle. These cycles are shown in Figure 3.6.

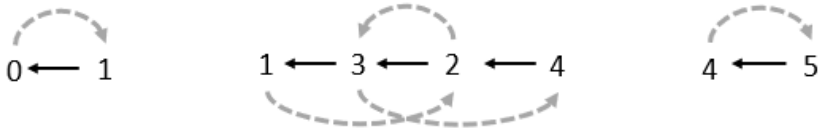

Figure 3.6: Alternating cycles for (1 3 2 4) cycle graph [2, 34]

Simple Permutation. A *simple permutation* π is a permutation whose cycle graph $G(\pi)$ does not contain any alternating cycle of length exceeding 3 [16, 34].

Thus, both (3 2 1 4) and (1 3 2 4) are simple permutations as their cycle graphs do not contain any cycles of length exceeding 3. A process known as the (g, b)-split is described in detail in Hennenhalli et al. [16] and Roy et al. [34] for transforming permutations whose cycle graphs contain alternating cycles of length four or more. Thus, Roy et al. [34, 35] describe an algorithm which focuses on eliminating 3-cycles and 2-cycles in a simple permutation π to transform it to the identity permutation.

Suppose $SSD(\pi)$ is the minimum number of strip swaps needed in order to obtain the identity permutation from π. Then, Christie et al. [12] show that

$$SSD(\pi) \geq (n + 1 - C(\pi))/2, \tag{3.2}$$

where $C(\pi)$ is the number of alternating cycles in the cycle graph of π. For example, since the cycle graph of the permutation π = (3 2 1 4) contains three alternating cycles, as shown in Figure 3.4, according to Christie et al. [12], at least one strip swap will be needed to get this permutation sorted. On the other hand, Roy et al. [34] show that

$$SSD(\pi) \geq (n + 1 - C_{\text{odd}}(\pi))/2, \tag{3.3}$$

where $C_{\text{odd}}(\pi)$ is the number of odd alternating cycles (alternating cycles with an odd number of black edges) in the cycle graph of π. Since the cycle graph of the permutation π = (3 2 1 4) contains only one odd alternating cycle, as shown in Figure 3.4, according to Roy et al. [34], at least two strip swaps will be needed to get this permutation sorted. Clearly, the lower bound (3.3) due to Roy et al. [34] is a tighter lower bound on the number of strip swaps when compared to the lower bound (3.2) due to Christie et al. [12]. Using the above ideas, Roy et al. [34] develop an algorithm that executes strip swaps motivated by the number of additional odd alternating cycles that a strip swap will introduce in the cycle graph

of the permutation being sorted.

A *0-move*, carried out on a 3-cycle, breaks it into one 2-cycle and one 1-cycle, thereby increasing the number of odd-cycles by 0 (no increase) in the cycle graph. This is illustrated in Figure 3.7.

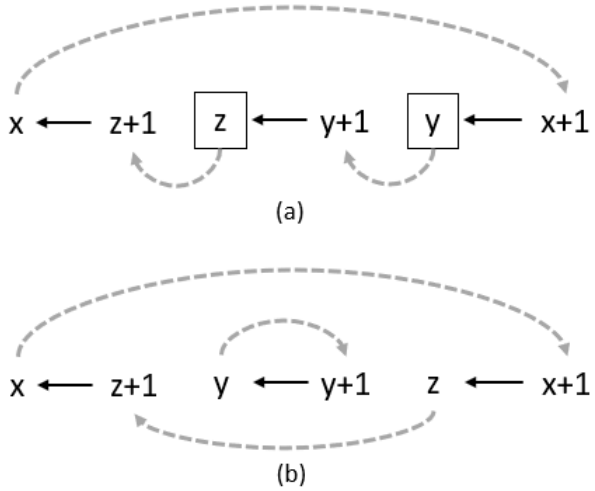

Figure 3.7: A 0-move leading to a 2-move [34]

A *2-move*, carried out on a 2-cycle, breaks it into two 1-cycles, thereby increasing the number of odd-cycles by 2. This is illustrated in Figure 3.8.

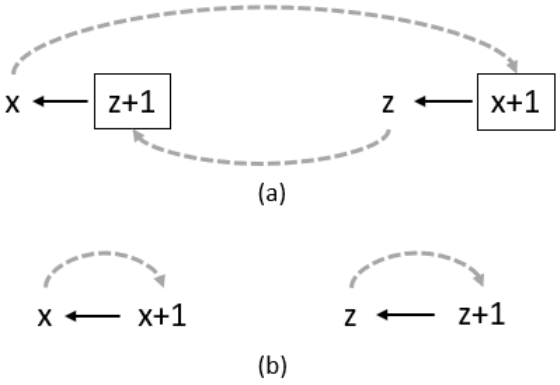

Figure 3.8: 2-move [34]

As seen in Figure 3.7, suppose a 3-cycle is represented as $(x, x+1, y, y+1, z, z+1, x)$, then swapping the strip y with the strip z results in the formation of the 2-cycle $(x, x+1, z, z+1, x)$ and the 1-cycle $(y, y+1, y)$. Thus, since a 3-cycle is split

into a 2-cycle and a 1-cycle, the number of odd cycles has not changed. Similarly, as seen in Figure 3.8, suppose a 2-cycle is represented as $(x, x+1, z, z+1, x)$, then swapping the strip $x+1$ with the strip $z+1$ results in the formation of the 1-cycle $(x, x+1)$ and the 1-cycle $(z, z+1)$. Thus, a 2-cycle (not an odd cycle), is split into two 1-cycles, leading to a net increase of two in the number of odd cycles. Note that from the cycle graph of the permutation (3 2 1 4) (see Figure 3.4), it is seen that it can be sorted upon making a 2-move. Similarly, the cycle graph of the permutation (1 3 2 4) (see Figure 3.6) shows that it can be sorted upon making a 0-move.

3.3 Findings from Implementation

We implemented the following algorithm of Roy et al. [34] which uses the concepts we have described in the previous section.

```
Data: π: The input permutation
Result: Sorted permutation
Construct the cycle graph G(π)
if π is not a simple permutation then
    Convert π to a simple permutation
end
while π ≠ id do
    while G(π) contains a 2-cycle do
        Perform a 2-move
    end
    while G(π) does not contain a 2-cycle do
        Perform a 0-move
        Perform a 2-move
    end
end
```

Algorithm 2: Sorting by Strip Swaps via Cycle Graphs

Our implementation experience confirmed our thoughts on the importance of implementation and validation of the algorithms. We discovered that while the cycle graph algorithm works for some permutations, but it fails on some others. It was

indeed surprising to note that the algorithm failed on the small-sized permutation (4 2 1 3), which however is not a simple permutation because it contains an alternating cycle of length that exceeds 3. In fact, as seen from Figure 3.9 and Figure 3.10, the cycle graph for (4 2 1 3) has just one alternating 5-cycle.

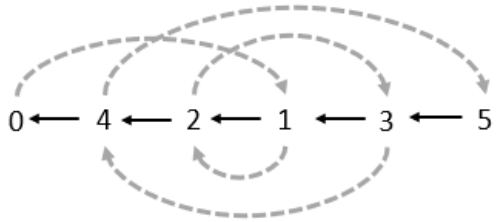

Figure 3.9: Cycle graph for (4 2 1 3) [2, 34]

$$(0 \dashrightarrow 1 \rightarrow 2 \dashrightarrow 3 \rightarrow 1 \dashrightarrow 2 \rightarrow 4 \dashrightarrow 5 \rightarrow 3 \dashrightarrow 4 \rightarrow 0)$$

Figure 3.10: Alternating cycles in the Cycle Graph of (4 2 1 3) [2, 34]

Using the (g, b)-split process [16, 34], the permutation (0 4 2 1 3 5) is converted into a simple permutation by inserting two additional appropriately chosen numbers in the original permutation, yielding the new, simple permutation (0 4 0.3 0.7 2 1 3). The cycle graph for this new simple permutation is shown in Figure 3.11.

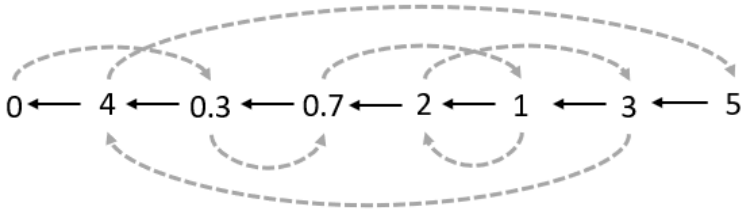

Figure 3.11: Cycle graph for (0 4 0.3 0.7 2 1 3) [2, 34]

The alternating cycles in the cycle graph of (0 4 0.3 0.7 2 1 3) are shown in Figure 3.12. It is clear that the new permutation is a simple permutation because

the lengths of all of the alternating cycles in the cycle graph of this permutation do not exceed 3.

$$(0 \dashrightarrow 0.3 \rightarrow 4 \dashrightarrow 5 \rightarrow 3 \dashrightarrow 4 \rightarrow 0) \qquad (0.3 \dashrightarrow 0.7 \rightarrow 0.3)$$
$$(0.7 \dashrightarrow 1 \rightarrow 2 \dashrightarrow 3 \rightarrow 1 \dashrightarrow 2 \rightarrow 0.7)$$

Figure 3.12: Alternating cycles for (0 4 0.3 0.7 2 1 3 5) cycle graph [2, 34]

According to Algorithm 2, the conversion of a non-simple permutation to a simple one is done once and only once, and the strip swaps are carried out within this simple permutation repeatedly until the identity permutation is obtained. However, for the simple permutation (0 4 0.3 0.7 2 1 3 5), in order to eliminate the 3-cycle $(0 \dashrightarrow 0.3 \rightarrow 4 \dashrightarrow 5 \rightarrow 3 \dashrightarrow 4 \rightarrow 0)$, a 0-move needs to be performed. This corresponds to swapping the elements 3 and 4. Performing this swap in the permutation yields (0 3 0.3 0.7 2 1 4 5). Next, Algorithm 2 requires a 2-move to be performed on (0 3 0.3 0.7 2 1 4 5). This corresponds to swapping the elements 0.3 and 4, which yields the permutation (0 3 4 0.7 2 1 0.3 5). Unfortunately, this newest permutation has a cycle graph which has an alternating cycle of length that exceeds 3, and hence fails to be a simple permutation.

Finally, we would like to point out that the tighter lower bound of Roy et al. [34] doesn't hold true either. For example, for the permutation (3 2 1 4), Figure 3.4 shows that there is only one odd cycle. Therefore, with $n = 4$ (the number of elements) and $C_{\text{odd}}(\pi) = 1$, the lower bound (3.3) indicates that at least two strip swaps are needed to sort (3 2 1 4). However, it is easy to see that with just one swap (of 1 and 3), the permutation could be sorted. Another permutation for which exactly the same scenario holds is (1 4 3 2), whose cycle graph contains only one odd cycle yielding a lower bound of 2, while it could be sorted with only one swap (or 2 and 4).

CHAPTER 4

NEW ALGORITHMS FOR SORTING BY STRIP SWAPS

In this chapter we present two new algorithms for sorting by strip swaps, which are the main contributions of this thesis. The first algorithm employs a greedy strategy, and we call it the *greedy algorithm*. The second algorithm tries to bring the closest consecutive pairs together, and we call this algorithm the *closest consecutive pair* (CCP) algorithm.

4.1 The Greedy Algorithm

The greedy algorithm selects the candidates for a strip swap at each step as the pair of strips that would result in the greatest reduction in the total number of strips at each step of advancing towards the identity permutation. In case there exists two of more such pairs that would reduce the number of strips by the same amount, the pair that would place a greater number of strips in their correct positions will be selected. It is important to emphasize here that the algorithm determines by how much the number of strips is reduced without actually performing the actual swaps.

In order to illustrate how the greedy algorithm works, let us consider sorting the permutation $\pi = (5\ 3\ 1\ 4\ 2)$ by strip swaps. Since there are 5 strips in the given permutation, the greedy algorithm will examine all possible $\binom{5}{2}$ pairs of strips as potential candidates for the initial swap. For ease of reference the pseudocode for the greedy algorithm is shown below.

Data: π_{init}: The input permutation
Result: s, the number of strip swaps needed with the greedy strategy
$s \leftarrow 0$
$\pi \leftarrow \pi_{\text{init}}$
while $\pi \neq id$ **do**
  Find (i, j) for which $\pi_i \leftrightarrow \pi_j$ will lead to the least number of strips
  For breaking ties, pick (i, j) which will maximize the number of correctly placed strips
  Swap π_i with π_j
  $s \leftarrow s + 1$
end

Algorithm 3: The Greedy Algorithm for Sorting by Strip Swaps

As indicated in the pseudocode above, the greedy algorithm will first try to find the pair of positions (i, j) that will reduce the number of strips the most if the strip at position i is swapped with the strip at position j. For clarity, we refer to the swaps in terms of π_i and π_j, instead of i and j in this explanation. Thus, for the permutation $\pi = (5\ 3\ 1\ 4\ 2)$, this would mean the swaps $(5 \leftrightarrow 3)$, $(5 \leftrightarrow 1)$, $(5 \leftrightarrow 4)$, $(5 \leftrightarrow 2)$, $(3 \leftrightarrow 1)$, $(3 \leftrightarrow 4)$, $(3 \leftrightarrow 2)$, $(1 \leftrightarrow 4)$, $(1 \leftrightarrow 2)$, and $(4 \leftrightarrow 2)$ need to be examined. As it turns out, two different swaps $(5 \leftrightarrow 2)$ and $(1 \leftrightarrow 4)$ will both reduce the number of strips by 2, while the other swaps reduce the number of strips by at most 1. When ties like this occur, the swap that puts the most number of strips in their correct positions, namely the positions at which they occur in the identity permutation. For the two swaps under consideration at this point, it is seen that the swap $(5 \leftrightarrow 2)$ places the strip $(4\ 5)$ in its correct position, while the swap $(1 \leftrightarrow 4)$ doesn't place any strip in its correct position. Thus, the swap $(5 \leftrightarrow 2)$ will be selected as the next swap by the algorithm. As a result of the first swap, the original permutation would have been transformed to the permutation $(\boxed{2\ 3}\ \boxed{1}\ \boxed{4\ 5})$. Thus, the possible swaps for the next iteration would be $(\boxed{2\ 3} \leftrightarrow 1)$, $(\boxed{2\ 3} \leftrightarrow \boxed{4\ 5})$ and $(1 \leftrightarrow \boxed{4\ 5})$. The greedy algorithm will select and carry out the swap $(\boxed{2\ 3} \leftrightarrow 1)$ as it results in the identity permutation.

4.2 The Closest Consecutive Pair Algorithm

The greedy strategy for sorting by strip swaps is based on the idea of reducing the number of strips by the maximum amount possible at each step. Another perspective of sorting by strip swaps is to consider it as a rearrangement problem with the goal of matching the position of the strips in any permutation π to their respective positions in the identity permutation *id*. The Closest Consecutive Pair (CCP) algorithm tries to achieve this goal as we will describe in this section.

The main idea behind the CCP algorithm is to associate a penalty with every strip in the permutation π on basis of how far it is from its corresponding position in the identity permutation *id*, and select a pair of strips to swap such that will put more strips in their correct place than any other swap. Also, since sorting by strip swaps is aimed at forming progressively longer strips from one step to the next, it is beneficial to take into account the penalties associated with the predecessor and successor strips also to determine a score for pairs of consecutive strips. This thought process led to the development of the CCP algorithm. The motivation for this design comes from the field of reinforced learning [39], where software agents are assigned cumulative rewards based on their advancement towards the optimal solution.

Before presenting the CCP algorithm in the form of pseudocode, we will illustrate how it works using the permutation $\pi = (1\ 4\ 3\ 2)$. To each strip in the permutation, we assign *positional penalty points* (PPP) based on how far the strip is from its correct position in the identity permutation. For example, in the permutation (1 4 3 2), the PPP of strip $\boxed{1}$ is zero since strip $\boxed{1}$ is in the same position in π as in *id*. Also, since the strip $\boxed{4}$ will need two hops to come to its expected position in *id*, the PPP of $\boxed{4}$ equals 2. This is illustrated in Figure 4.1.

Since the final goal in sorting by strip swaps is to arrive at the single strip repre-

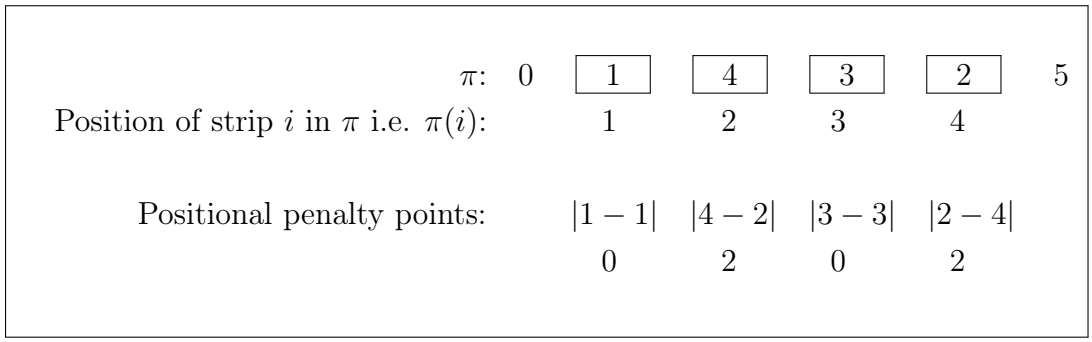

Figure 4.1: Positional penalty points

senting *id*, the CCP algorithm works towards increasing the sizes of existing strips at every step. Note that increasing the size of a particular strip will depend on where the strip's predecessor and successor are located within the permutation. Thus, the CCP algorithm associates a score for each strip as the sum of the PPP of the strip itself and the PPP of its predecessor and the PPP of its successor. For example, in the permutation (1 4 3 2), the score for the strip $\boxed{3}$ is obtained as PPP($\boxed{3}$)+PPP($\boxed{2}$)+PPP($\boxed{4}$), which equals 4. We will refer to this as *total penalty points*. Once the total penalty points are calculated for all of the strips in the permutation, the CCP algorithm proceeds to find a consecutive pair of strips that should be brought together. For the permutation of $1, 2, \cdots, n$, the the algorithm selects the pair $(i, i+1)$ for which the sum of the total penalty points of i and the total penalty points of $i+1$ is the least. Such a pair is called the *closest consecutive pair*, which is the reason why this algorithm has been named the CCP algorithm. Once such a pair is determined, the CCP algorithm proceeds to decide which strip swap will bring such a pair together, but we wish to first illustrate how the closest consecutive pair is determined for the example permutation (1 4 3 2).

As illustrated in Figure 4.2, for the example permutation (1 4 3 2), the closest consecutive pair is $(1, 2)$ with 4 accumulated penalty points.

Suppose the closest consecutive pair is $(i, i+1)$. Three scenarios need to be con-

π:	0	$\boxed{1}$	$\boxed{4}$	$\boxed{3}$	$\boxed{2}$ 5
Position of strip i in π i.e. $\pi(i)$:		1	2	3	4
Positional penalty points (PPP):		0	2	0	2
Predecessor PPP:		0	0	2	0
Successor PPP:		2	0	2	0
Total penalty points:		2	2	4	2
Consecutive pairs:		$(1,2)$		$(2,3)$	$(3,4)$
Consecutive pair points i.e. $d_i + d_{i+1}$:		4		6	6
$\min(d_i + d_{i+1})$:		4			
Closest Consecutive pair:		$(1,2)$			

Figure 4.2: Closest consecutive pair for (1 4 3 2)

sidered in terms of strips swaps that could bring strip i and strip $i+1$ together: 1) swapping strip i with strip $i+1$; 2) strip i may be swapped with the predecessor of $i+1$; and 3) strip $i+1$ may be swapped with the the successor of $i-1$. The CCP algorithm selects a swap among the above three possible swaps by using a greedy strategy at this point. The algorithm chooses the swap that reduces the most the number of strips in the permutation, and in case of ties, the algorithm chooses the strip that puts more strips in their correct positions. For the example permutation, the strips $\boxed{1}$ and $\boxed{2}$ may be brought together either by swapping the strips $\boxed{4}$ and $\boxed{2}$, or 2) by swapping the strips $\boxed{1}$ and $\boxed{3}$. The swap $(4 \leftrightarrow 2)$ will be selected as it yields a single strip that is the identity permutation. This is illustrated in Figure 4.3, following which we present the pseudocode for the CCP algorithm as Algorithm 4.

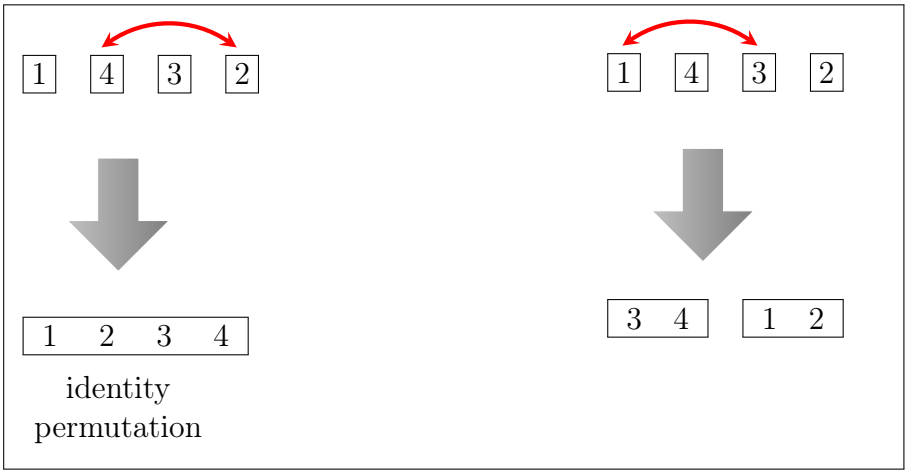

Figure 4.3: Sort (1 4 3 2) using the CCP algorithm

Data: π_{init}: The input permutation
Result: s, the number of strip swaps needed with the closest consecutive pair strategy

$s \leftarrow 0$
$\pi \leftarrow \pi_{\text{init}}$
while $\pi \neq id$ **do**
  /* $\pi(i) \leftarrow$ position of strip i in π */
  for $i \leftarrow 1$ to n **do**
    /* $d_i \leftarrow$ positional score + Successor score + Predecessor score */
    $d_i \leftarrow |i - \pi(i)| + |(i+1) - \pi(i+1)| + |(i-1) - \pi(i-1)|$
  end
  CCP $\leftarrow (i, i+1)$ from $\min(d_i + d_{i+1})$
  if greedy_winner(CCP) $= 1$ **then**
    CCP $\leftrightarrow$ CCP+1
  else if greedy_winner(CCP) $= 2$ **then**
    CCP $\leftrightarrow$ predecessor(CCP+1)
  else
    CCP+1 $\leftrightarrow$ successor(CCP-1)
end

Algorithm 4: Closest Consecutive Pair algorithm

CHAPTER 5

EXPERIMENTS AND RESULTS

In this chapter, we provide the details associated with the experiments we conducted for the purpose of testing and validating the existing algorithms as well as the new algorithms developed in this thesis. The chapter is divided in to four sections. In the first section, we present the idea of *kernel permutations*, a concept introduced by Mahajan et al. [26]. This section will serve as the justification for why we chose to perform our experiments with kernel permutations of size n, when n is the given number of strips in a permutation of an arbitrary size. In the second section, we describe the implementation framework we have developed in the Python programming language to represent permutations as strings of strips, as well as to carry out strip moves and strip swaps. In the third section, we summarize our findings from the experiments we conducted on the *strip deletion* algorithm. In the last section, we present the results from the experiments we conducted on the newly developed greedy and CCP algorithms.

5.1 Kernelized Permutations

Mahajan et al. [26] describe a process of reducing a permutation to a *kernel permutation*. Any permutation π can be reduced to its kernel counterpart $ker(\pi)$ by replacing the strips in it with their ranks in the permutation. The rank of a strip is decided by its position in the identity permutation. For example, Figure 5.1 shows the permutation $\pi = (7\ \underline{2\ 3}\ 8\ 1\ \underline{4\ 5\ 6})$, and its equivalent kernel permutation $ker(\pi) = (4\ 2\ 5\ 1\ 3)$.

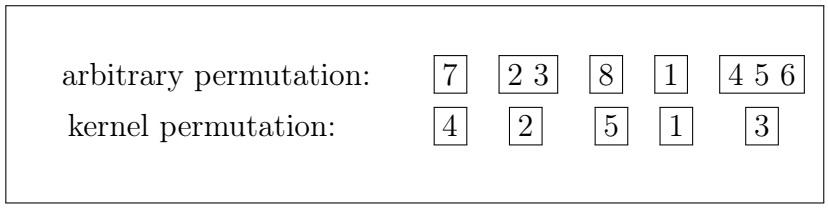

Figure 5.1: Kernel Permutation example

As seen in Figure 5.1, in kernel permutations each strip is of unit length. Mahajan et al. [26] have also demonstrated that sorting the permutation π by strip moves is equivalent to sorting its kernel $ker(\pi)$ by strip moves. Since a strip swap may be considered as a pair of strip moves, sorting π by strip swaps is also equivalent to sorting ker(π) by strip swaps. Thus, for permutations of size n, instead of considering all $n!$ permutations, it is sufficient to consider only those n-permutations which have n strips while evaluating the sorting algorithms considered in this thesis. We denote the number of kernelized permutations with n strips by K_n. Table 5.1 shows the values of K_n for $n = 5, 6, \cdots, 10$.

n	5	6	7	8	9	10
K_n	53	309	2119	16687	148329	1468457

Table 5.1: K_n – The number of Kernelized n-Permutations

Taking advantage of this observation, the experiments considered only kernel permutations so that it was possible to test permutations of larger sizes and reduce the number of permutations to consider for each specified n. It is important to note that the results obtained by testing with the K_n kernelized n-permutations will also hold for any r-permutation π with n strips, and $r \geq n$.

The experiments were carried out in the following phases:

1. All kernelized permutations of size $n = 5, 6, 7, 8$, and 9 were sorted using the block deletion algorithm. The number of strip moves needed by this algorithm, and the exact number of strip moves as determined using the

exact brute-force algorithm were used to obtain the approximation ratio corresponding to each permutation tested. Additionally, a random collection of 1000 kernelized 10-permutations were sorted and the approximation ratios for the block deletion algorithm, was determined for each of these 1000 permutations as well using the brute-force algorithm.

2. All kernelized permutations of size $n = 5, 6, 7, 8$, and 9 were sorted by strip swaps using the greedy algorithm, and the CCP algorithm. The number of strip swaps needed by these algorithms, and the the exact number of strip swaps as determined using an exact brute-force algorithm for sorting by strip swaps were used to obtain the approximation ratio corresponding to each permutation tested.

3. All kernelized 10-permutations were sorted using the greedy and CCP algorithms, but a lower bound (instead of the exact minimum number of swaps needed) was used to estimate the approximation ratios corresponding to each permutation tested.

4. A random collection of 100 kernelized n-permutations for $n = 12, 25$, and 50 were sorted using the greedy and CCP algorithms. This experiment was repeated 100 times and the averages of the results of these experiments were recorded.

5. The execution times for the greedy and the CCP algorithms were compared for all the experiments carried out.

5.2 Python Implementation Framework

An implementation framework has been developed in the Python programming language to run experiments on various algorithms for sorting with strip moves

and sorting by strip swaps. Along with the extensive use of the list and the dictionary data structures, the following modules and libraries were used as well in the framework.

1. Combinatoric iterators *permutations* and *combinations* from the *itertools* module

2. *consecutive_groups* function from *more_itertools* library to create strips in the permutations

3. *collections*, *heapq* and *queue* modules for creating and working with weighted directed acyclic graphs

4. *pool* object within *multiprocessing* package for parallel execution of the algorithms across multiple input values by distributing the input permutations across processes

5.3 Results for Sorting by Strip Moves via Strip Deletions

Suppose $bs(\pi)$ denotes the exact minimum number of strip moves needed to transform an arbitrary permutation π to the identity permutation; $bd(\pi)$ denotes the length of the shortest strip deletion sequence for π; and $sm(\pi)$ denotes the number of strip moves that the strip deletion algorithm of Bein et al. [5] requires to sort π. Then, our implementation of the strip deletion algorithm of Bein et al. [5] is aimed at validating the results reported. Specifically, according to Bein et al. [5]:

1. The minimum number of strip moves needed to sort an arbitrary permutation by strip deletions does not exceed the length of the shortest strip deletion sequence. Equivalently,

$$sm(\pi) \leq bd(\pi). \tag{5.1}$$

2. The strip deletion algorithm is a 2-approximation algorithm. Equivalently,

$$sm(\pi) \leq 2\ bs(\pi). \tag{5.2}$$

In order to verify (5.1), for each permutation that was sorted by strip moves via strip deletion, the length of the strip deletion sequence obtained in Stage-2 of Algorithm 1 was compared to the length of the strip move sequence in Stage-3. In order to verify (5.2), a brute-force algorithm that determines the exact value of $bs(\pi)$ for sorting by strip moves has been developed and implemented. The pseudocode for the brute-force algorithm is presented below as Algorithm 5.

Data: π_{init}: The input permutation
Result: m, the least number of strip moves to sort π_{init}
$m \leftarrow 0$
`/* Breadth First Search */`
$\pi \leftarrow \pi_{\text{init}}$
$frontier \leftarrow$ a FIFO queue with π as the only element
$explored \leftarrow$ an empty set
$ACTIONS \leftarrow$ all legal moves
while $\pi \neq id$ **do**
  $\pi \leftarrow$ POP(frontier)`/* chooses the shallowest permutation in frontier */`
  Add π to *explored*
  for each *move* in $ACTIONS$ **do**
    do the move, adding the resulting permutation to the frontier
      only if not in the frontier or explored set
  end
  $m \leftarrow m + 1$
end

Algorithm 5: Brute-Force Search for the least number of strip moves

For all kernelized n-permutations π with $n \in \{5, 6, 7, 8, 9\}$, and for all the 1000 randomly chosen 10-permutations, the inequality (5.1) was satisfied. In other words, the number of strip moves needed did not exceed the length of the shortest strip deletion sequence, as was proved in Bein et al. [5]. We further subdivided the results into two categories for each n, namely, those for which $sm(\pi) = bd(\pi)$,

and those for which $sm(\pi) < bd(\pi)$ were true. These results are summarized in Table 5.2.

n	5	6	7	8	9	10
$sm(x) = bd(x)$	53	309	2114	16596	144411	968
$sm(x) < bd(x)$	0	0	5	91	3918	32

Table 5.2: Sorting by Strip Moves via Strip Deletions Satisfies (5.1)

It is evident that $sm(\pi) \leq bd(\pi)$ is satisfied for all K_n kernelized n-permutations for $n \in \{5, 6, 7, 8, 9\}$, and for the 1000 randomly chosen 10-permutations. Additionally, for a small fraction of the permutations for each n, the number of strips moves was strictly smaller than the length of the shortest strip deletion sequence, or $sm(\pi) < bd(\pi)$; and whenever this happened, we observed that the number of strip moves needed was one less than the length of the shortest strip deletion sequence.

Next, in order to verify (5.2), we use the exact number of strip moves as determined from the brute-force algorithm to obtain the approximation ratio corresponding to each permutation tested. Table 5.3 shows how the K_n kernelized n-permutations are distributed across ranges of approximation ratios (ρ).

n	$\rho = 1$	$1 < \rho < 2$
5	100.00	–
6	99.68	0.32
7	98.63	1.37
8	96.81	3.19
9	92.19	7.81
10*	96.20	3.80

*1000 random 10-permutations

Table 5.3: Percent Distribution of Permutations Across Approximation Ratios (Strip Deletion for $n = 5, ..., 10$)

For instance, all of the kernelized 5-permutations were sorted optimally by the strip deletions algorithm. This means that the number of strip moves that the

strip deletions algorithm needed matched with the exact number of strip moves as determined by the brute-force algorithm. As another example, 92.19% of all the kernelized 9-permutations were sorted optimally, whereas the remaining 7.81% of the kernelized 9-permutations required no more than twice the optimal number of strip moves. It is evident from Table 5.3 that the approximation ratios for all permutations tested stayed under 2 as claimed in Bein et al.[5].

5.4 Results for Sorting by Strip Swaps Using the Greedy and CCP Algorithms

As the problem of sorting by strip swaps is still an open problem, there are currently no available algorithms, to our best knowledge, with known approximation ratios with which we could compare our experimental results. However, as was done with sorting by strip moves we have developed a brute-force algorithm for sorting by strip swaps as well. The pseudocode for this brute-force algorithm is shown below as Algorithm 6.

Data: π_{init}: The input permutation
Result: s, the least number of strip swaps to sort π_{init}
$s \leftarrow 0$
`/* Breadth First Search */`
$\pi \leftarrow \pi_{\text{init}}$
$frontier \leftarrow$ a FIFO queue with π as the only element
$explored \leftarrow$ an empty set
$ACTIONS \leftarrow$ all combinations of swaps between strips
while $\pi \neq id$ **do**
  $\pi \leftarrow$ POP(frontier)`/* chooses the shallowest permutation in frontier */`
  Add π to *explored*
  for each *swap* in $ACTIONS$ **do**
    do the swap, adding the resulting permutation to the frontier
      only if not in the frontier or explored set
  end
  $s \leftarrow s + 1$
end

Algorithm 6: Brute-Force Search for the least number of strip swaps

This algorithm is able to determine the exact least number of strip swaps needed by exploring all possible swaps. Thus, it is not feasible to use it for large-sized permutations. Lower bounds on the number of strip swaps have been given in Roy et al. [34] and Christie [12]. A trivial lower bound for the number of swaps needed to sort any permutation with n strips is known to be $(n-1)/4$ [34]. The lower bound given by Christie [12] to sort a permutation π with n strips is given by (3.2), which involves determining the number of alternating cycles in the cycle graph of π. For purposes of estimating the approximation ratios of the two newly developed algorithms for sorting by strip swaps, we have used this lower bound for large-sized permutations. The use of the lower bound instead of the exact number of strip swaps needed enabled us to experiment with our algorithms on permutations as large as those consisting of 50 strips. In the rest of this section, we present the results we obtained with the greedy and the CCP algorithms in two subsections. In the first subsection, results relating to estimating the approximation ratios for both the greedy and the CCP algorithms using the brute-force exact solution are presented. In the second subsection, results relating to estimating the approximation ratios for both the greedy and the CCP algorithms using the lower bound [12] are presented.

5.4.1 Approximation Ratios using the Optimal Brute-Force Solution

Table 5.4 shows how the K_n kernelized n-permutations are distributed across ranges of approximation ratios for both the greedy (GR) and the CCP algorithms.

Note that an approximation ratio of 1 means that the solution obtained by the algorithm matches with the optimal solution. Then, we see from Table 5.4 that the greedy algorithm sorts all kernelized 5-permutations with the least number of strip swaps, and the CCP algorithm sorts 98.1% of all kernelized 5-permutations with

n \ ρ	$\rho = 1$		$\rho \in (1, 1.5]$		$\rho \in (1.5, 2.0]$	
	CCP	GR	CCP	GR	CCP	GR
5	98.11	100.0	1.89	–	–	–
6	90.94	100.0	9.06	–	–	–
7	80.27	98.68	19.16	1.32	0.57	–
8	75.39	96.94	22.08	3.06	2.53	–
9	56.56	92.85	41.74	7.14	1.70	0.01

Table 5.4: Percent Distribution of Permutations Across Approximation Ratios (Strip swaps for $n = 5, ..., 9$)

the least number of strip swaps. As the size n increases the percentage of kernelized permutations on which both algorithms achieve the optimal solution decreases as expected. However, for $n \in \{5, 6, 7, 8, 9\}$, the greedy algorithm is able to sort over 99% of all kernelized n-permutations and the CCP algorithm is able to sort over 97% of all kernelized n-permutations using no more than 1.5 times the optimal number of strip swaps. We would also like to point out that the CCP algorithm required 2.33 times the optimal number of strip swaps on roughly 0.003% of all kernelized 9-permutations.

5.4.2 Approximation Ratios using the Lower Bound of Christie [12]

We experimented with the greedy (GR) and the CCP algorithms on large-sized permutations to investigate their scalability. However, it is not practically feasible to execute the brute-force algorithm to determine the exact number of strip swaps needed to sort permutations of size 10 or more because of time, cost, and resource constraints. Therefore, we used the lower bound given by (3.2) due to Christie [12]. In these experiments, we tested the algorithms on all kernelized 10-permutations as well as 100 randomly chosen kernelized n-permutations for $n = 12, 25$ and 50. As seen in Table 5.5, the greedy algorithm outperforms the CCP algorithm in terms of the approximation ratio. For at least 97.4% of the permutations the greedy algorithm achieves an approximation ratio not exceeding 1.5. On the other

hand, the CCP algorithm achieves an approximation ratio not exceeding 1.75 for at least 92.7% of the permutations. While the approximation ratios for the greedy algorithm range from 1 to 1.81, the approximation ratios for the CCP algorithm range from 1 to 2.33. However, it is important to note that only for about 0.035% of the permutations the approximation ratios achieved by the CCP algorithm ranged from 2.0 and 2.33.

ρ / n	$\rho = 1$		$\rho \in (1, 1.25]$		$\rho \in (1.25, 1.5]$		$\rho \in (1.5, 1.75]$		$\rho \in (1.75, 2.0]$	
	CCP	GR	CCP	GR	CCP	GR	CCP	GR	CCP	GR
10*	47.39	80.75	37.96	17.93	12.88	1.24	1.58	0.08	0.17	–
12	28.43	70.85	40.73	27.50	24.77	1.63	5.56	0.01	0.50	0.01
25	0.07	11.37	13.68	72.88	60.20	15.64	24.71	0.11	1.34	–
50	–	0.01	0.01	26.67	13.92	70.72	78.84	2.60	7.22	–

*for all kernelized 10-permutations

Table 5.5: Percent Distribution of Permutations Across Approximation Ratios (Strip Swaps for $n = 10, 12, 25, 50$)

5.4.3 Comparison of Execution Times

As mentioned previously, the greedy algorithm outperforms the CCP algorithm in achieving smaller approximation ratios. This means that the greedy algorithm is able to accomplish the sort using closer to optimal number of strip swaps when compared to the CCP algorithm for the permutation sets on which we tested both algorithms. On the other hand, the CCP algorithm takes much less time carrying out the sorts. Table 5.6 shows the execution times for the small-sized permutations ($n \in \{2, 3, \cdots, 9\}$). The times indicated in the tables are in milliseconds, and represent the average execution time over all kernelized n-permutations tested for each n.

n	2	3	4	5	6	7	8	9
CCP	0.82	0.67	0.64	0.66	0.76	0.87	0.97	1.09
Greedy	0.81	0.67	0.64	0.69	0.91	1.22	1.67	2.28

Table 5.6: Average Execution times in milliseconds for $n = 2, .., 9$

n	10	12	25	50
CCP	1.23	1.54	4.82	18.16
Greedy	3.07	5.28	73.49	1215.24

Table 5.7: Average Execution time in milliseconds for $n = 10, 12, 25, 50$

Next, as seen in Table 5.7, the execution times for the greedy algorithm continue to be considerably greater than the execution times for the CCP algorithm for the large-sized permutations ($n = 10, 12, 25, 50$) as well.

We conclude this section with a plot of the execution times for both the greedy and the CCP algorithms in the same graph for small-sized permutations ($n \in \{2, 3, \cdots, 9\}$ so that the advantage of the CCP approach as n increases is clear.

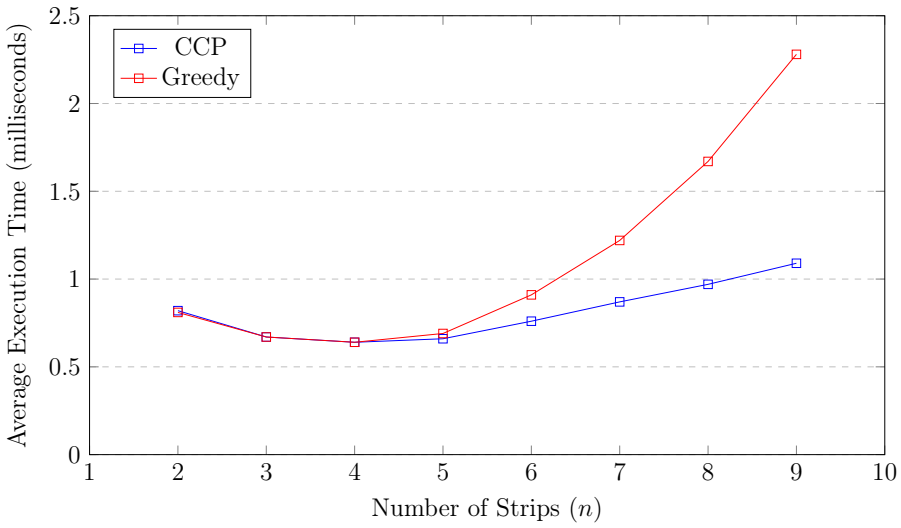

Figure 5.2: Execution Time For $n = 2, 3, \cdots, 9$

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

The research described in this thesis makes the following contributions:

1. For sorting by strip moves, a previously developed algorithm, known as the *strip deletions algorithm* [5] has been implemented and experimentally verified that it is a 2-approximation algorithm. For this purpose, an exact, brute-force algorithm was developed, implemented, and tested on small-sized permutations as part of the research in this thesis.

2. For sorting by strip swaps, through implementation of a previously developed algorithm [34], which is called the *cycle graph algorithm* in this thesis, it was discovered that

   - the cycle graph algorithm fails in a few scenarios that were not previously identified in the theoretical development of the algorithm [34]; and
   - the tighter lower bound for the number of strip swaps suggested in [34] does not hold true for certain permutations.

3. A new algorithm for sorting by strip swaps that uses a greedy strategy has been developed and implemented.

4. Another new algorithm for sorting by strip swaps that uses the strategy of

bringing closest consecutive pairs together has been developed and implemented.

5. Both the greedy algorithm and the closest consecutive pairs algorithm have been implemented and tested. For this purpose, an exact, brute-force algorithm, and another algorithm that determines the lower bound on the number of strip swaps using cycle graphs have been implemented.

6.2 Directions for Future Research

We see the potential for further work in this area from the following perspectives:

1. During the testing of the CCP algorithm, it was discovered that its approximation ratio exceeded 2 for a small number of kernelized, large-sized permutations (10, 12, 25, 50 initial strips). This offers the potential for further research on the CCP algorithm for possible theoretical developments and analyses.

2. As the CCP algorithm is new, it opens up several opportunities for further theoretical research with regards to the approximation ratio and its complexity.

3. As sorting problems using biologically-motivated primitives are modeled as combinatorial optimization problems, it may be worthwhile to investigate the possible use of machine learning approaches to devise new algorithms for these problems.

REFERENCES

[1] Bafna, V., and Pevzner, P. A. Genome rearrangements and sorting by reversals. SIAM Journal on Computing 25, 2 (1996), 272–289.

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

[3] Bein, W. W., Larmore, L. L., Latifi, S., and Sudborough, I. H. Block sorting is hard. International Journal of Foundations of Computer Science 14, 03 (2003), 425–437.

[4] Bein, W. W., Larmore, L. L., Morales, L., and Sudborough, I. H. A faster and simpler 2-approximation algorithm for block sorting. In International Symposium on Fundamentals of Computation Theory (2005), Springer, pp. 115–124.

[5] Bein, W. W., Larmore, L. L., Morales, L., and Sudborough, I. H. A quadratic time 2-approximation algorithm for block sorting. Theoretical Computer Science 410, 8-10 (2009), 711–717.

[6] Berman, P., Hannenhalli, S., and Karpinski, M. 1.375-approximation algorithm for sorting by reversals. Lecture notes in computer science (2002), 200–210.

[7] Bulteau, L., Fertin, G., and Rusu, I. Sorting by transpositions is difficult. SIAM Journal on Discrete Mathematics 26, 3 (2012), 1148–1180.

[8] Bulteau, L., Fertin, G., and Rusu, I. Pancake flipping is hard. Journal of Computer and System Sciences 81, 8 (2015), 1556–1574.

[9] Caprara, A. Sorting by reversals is difficult. In Proceedings of the first annual international conference on Computational molecular biology (1997), ACM, pp. 75–83.

[10] Christie, D. A. Sorting permutations by block-interchanges. Information Processing Letters 60, 4 (1996), 165–169.

[11] Christie, D. A. A 3/2-approximation algorithm for sorting by reversals. In SODA (1998), pp. 244–252.

[12] Christie, D. A. Genome rearrangement problems. PhD thesis, University of Glasgow, 1998.

[13] Gates, W. H., and Papadimitriou, C. H. Bounds for sorting by prefix reversal. Discrete mathematics 27, 1 (1979), 47–57.

[14] Gobi, R., Latifi, S., and Bein, W. Adaptive sorting algorithms for evaluation of automatic zoning employed in ocr devices. In Proceedings of the 2000 International Conference on Imaging Science, Systems, and Technology (2000), CSREA Press, pp. 253–259.

[15] Gu, Z., Wang, H., Nekrutenko, A., and Li, W.-H. Densities, length proportions, and other distributional features of repetitive sequences in the human genome estimated from 430 megabases of genomic sequence. Gene 259, 1-2 (2000), 81–88.

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

[17] Hartman, T., and Sharan, R. A 1.5-approximation algorithm for sorting by transpositions and transreversals. Journal of Computer and System Sciences 70, 3 (2005), 300–320.

[18] Hoot, S. B., and Palmer, J. D. Structural rearrangements, including parallel inversions, within the chloroplast genome of anemone and related genera. Journal of Molecular Evolution 38, 3 (1994), 274–281.

[19] Huang, J., and Roy, S. On sorting under special transpositions. In 2014 IEEE International Conference on Bioinformatics and Bioengineering (2014), IEEE, pp. 325–328.

[20] Kanai, J., Rice, S. V., Nartker, T. A., and Nagy, G. Automated evaluation of ocr zoning. IEEE Transactions on Pattern Analysis and Machine Intelligence 17, 1 (1995), 86–90.

[21] Kent, W. J., Baertsch, R., Hinrichs, A., Miller, W., and Haussler, D. Evolution's cauldron: duplication, deletion, and rearrangement in the mouse and human genomes. Proceedings of the National Academy of Sciences 100, 20 (2003), 11484–11489.

[22] Latifi, S. How can permutations be used in the evaluation of zoning algorithms? International journal of pattern recognition and artificial intelligence 10, 03 (1996), 223–237.

[23] Lin, Y. C., Lu, C. L., Chang, H.-Y., and Tang, C. Y. An efficient algorithm for sorting by block-interchanges and its application to the evolution of vibrio species. Journal of Computational Biology 12, 1 (2005), 102–112.

[24] Lindroos, H., Vinnere, O., Mira, A., Repsilber, D., Näslund, K., and Andersson, S. G. Genome rearrangements, deletions, and amplifica-

tions in the natural population of bartonella henselae. Journal of bacteriology 188, 21 (2006), 7426–7439.

[25] Mahajan, M., Rama, R., Raman, V., and Vijayakumar, S. Merging and sorting by block moves. In International Conference on Foundations of Software Technology and Theoretical Computer Science (2003), Springer, pp. 314–325.

[26] Mahajan, M., Rama, R., Raman, V., and Vijaykumar, S. Approximate block sorting. International Journal of Foundations of Computer Science 17, 02 (2006), 337–355.

[27] Mahajan, M., Rama, R., and Vijayakumar, S. Towards constructing optimal strip move sequences. In International Computing and Combinatorics Conference (2004), Springer, pp. 33–42.

[28] Mahajan, M., Rama, R., and Vijayakumar, S. Block sorting: A characterization and some heuristics. In Nordic Journal of Computing (2008).

[29] Narayanaswamy, N., and Roy, S. Block sorting is apx-hard. In International Conference on Algorithms and Complexity (2015), Springer, pp. 377–389.

[30] of General Medical Sciences, N. I. Studying genes.

[31] Palmer, J. D., and Herbon, L. A. Tricircular mitochondrial genomes of brassica and raphanus: reversal of repeat configurations by inversiton. Nucleic Acids Research 14, 24 (1986), 9755–9764.

[32] Palmer, J. D., and Herbon, L. A. Plant mitochondrial dna evolved rapidly in structure, but slowly in sequence. Journal of Molecular evolution 28, 1-2 (1988), 87–97.

[33] Pevzner, P. Computational molecular biology: an algorithmic approach. MIT press, 2000.

[34] Roy, S., and Thakur, A. K. Towards construction of optimal strip-exchanging moves. In 2007 IEEE 7th International Symposium on BioInformatics and BioEngineering (2007), IEEE, pp. 821–827.

[35] Roy, S., and Thakur, A. K. Approximate strip exchanging. International journal of computational biology and drug design 1, 1 (2008), 88–101.

[36] Sémon, M., and Wolfe, K. H. Consequences of genome duplication. Current opinion in genetics & development 17, 6 (2007), 505–512.

[37] Siguier, P., Filée, J., and Chandler, M. Insertion sequences in prokaryotic genomes. Current opinion in microbiology 9, 5 (2006), 526–531.

[38] Turlapaty, S. Implementation and performance comparison of some heuristic algorithms for block sorting, 2018.

[39] Wiki. Reinforcement learning.

[40] Williamson, D. P., and Shmoys, D. B. The design of approximation algorithms. Cambridge university press, 2011.

VITA

Chandrika Rao has a Bachelor of Science in Physics from the University of Mumbai, India. She expects to receive a Master of Science in Computer and Information Sciences with a concentration in Computer Science from the University of North Florida in Dec 2020. Dr. Asai Asaithambi and Dr. Swapnoneel Roy are serving as Chandrika's thesis advisors. Chandrika also has twelve years of experience in the software industry.

Chandrika's areas of research interest include Artificial Intelligence, Machine learning, Cybersecurity, and Internet of Things. She has co-authored the paper *Online Context-Adaptive Energy-Aware Security Allocation in Mobile Devices: A Tale of Two Algorithms*, which was published in the Proceedings of the International Conference on Distributed Computing and Internet Technology, 2020. After graduation with the Master's degree, Chandrika plans to pursue her career in the above fields.