Greedy and Speedy: New Gray Code Algorithms for Generating Signed Permutations

by

Yuan (Friedrich) Qiu

Professor Aaron Williams, Advisor

A thesis submitted in partial fulfillment
of the requirements for the
Degree of Bachelor of Arts with Honors
in Computer Science

Williams College
Williamstown, Massachusetts

July 13, 2024

Contents

List of Figures 4

List of Tables 6

1 Introduction 8
- 1.1 Gray Codes . . . 8
  - 1.1.1 Flip Graphs . . . 9
  - 1.1.2 Binary Reflected Gray Code . . . 10
  - 1.1.3 Plain Change Order . . . 13
  - 1.1.4 Additional Gray Codes . . . 16
- 1.2 Combinatorial Generation . . . 17
  - 1.2.1 Lexicographic Order . . . 18
  - 1.2.2 Loopless Algorithms . . . 18
  - 1.2.3 Application: Exact Algorithms . . . 19
- 1.3 New Results . . . 19
- 1.4 Outline . . . 20

2 The Greedy Gray Code Algorithm 21
- 2.1 The Greedy Gray Code Algorithm . . . 21
  - 2.1.1 Backtracking . . . 22
- 2.2 Greedy Implementation of BRGC . . . 23
- 2.3 Greedy Interpretation of Plain Changes . . . 24
- 2.4 Greedy Interpretation of Other Gray Codes . . . 24
- 2.5 Failure of the Greedy Algorithm . . . 25

3 Loopless Change Sequence Algorithms 27
- 3.1 Binary and Decimal Ruler Sequences . . . 27
- 3.2 Ruler Sequences . . . 29
- 3.3 Signed Ruler Sequences . . . 29
- 3.4 Loopless BRGC and Plain Changes . . . 30

4 Greedy Gray Codes for Signed Permutations 32
- 4.1 Two-Sided Ribbons: Twists and Swaps . . . 33
  - 4.1.1 Twist Properties . . . 33
- 4.2 Restrictions on Operations . . . 34
- 4.3 Experiments with 1-Twists, 2-Twists and Swaps . . . 34
- 4.4 Selected Proofs . . . 35

5 Change Sequences for Signed Permutations **42**
5.1 Signed Change Sequences for Experiments 42
5.2 Loopless Algorithms 43
5.3 Implementation of Loopless Algorithms 43

6 Spanning Tree Gray Codes **45**
6.1 Matroids 46
6.1.1 Greedy Basis-Exchange Gray Codes 46
6.2 Generating Spanning Trees 47
6.2.1 Contraction and Deletion 47
6.2.2 Spanning Trees of Complete Graphs 48
6.3 A Greedy Revolving Door Algorithm 49
6.4 Experimental Results 51

7 Summary **54**

8 Bibliography **56**

A Selected Implementations **61**

List of Figures

1.1 Wikipedia's word ladder from `head` to `tail` [64]. Consecutive words differ in one letter, with the underlined letter changing to create the next word. 8
1.2 Is there a Gray code for the set of all 4-letter Scrabble™ words? 8
1.3 A subgraph of the word graph $\mathcal{W}_4$. The purple path is the word ladder from `head` to `tail` from Figure 1.1, while the gold path provides a shorter solution. Nodes `evil` and `zebu` have degree zero in the graph. In other words, no other 4-letter Scrabble™ word can be obtained by changing one of their letters. 10
1.4 The vertices of the n-dimensional cube are the n-bit binary strings B_n, and the edges connect those that differ in a single bit. 11
1.5 Illustrating how the binary reflected Gray code has a single bridge transition between its two subcubes. Note that the left copy of $\mathcal{H}_3$ in (b) has been reflected left-to-right relative to (a) in order to simplify the drawing. 12
1.6 An n-bit rotary encoder for $n = 4$. It contains n rings and 2^n sectors, with each sector having a different binary pattern $b_n b_{n-1} \cdots b_1 \in B_n$. Successive sectors follow the binary reflected Gray code $\mathsf{brgc}(n)$ in clockwise order (e.g., $b_4 b_3 b_2 b_1 = 0000, 0001, 0011, \ldots, 1000$) with white/black for 0/1. If the n-bit sensor straddles two sectors, then the Gray code property ensures that only one bit has an ambiguous value, and that the state will be read as being in either of the straddling sectors. Image adapted from [22]. 13
1.7 The vertices of the permutohedron $\mathcal{P}_n$ are the permutations S_n, and the edges connect those that differ by an adjacent transposition (i.e., swap). 14
1.8 The lexicographic ordering of the words found in Figure 1.1. Note that consecutive words in this order differ in up to three letters. 18

2.1 Binary reflected Gray code using indistinct two-sided ribbons for $n = 4$. 23
2.2 Plain changes $\mathsf{plain}(n)$ using distinct one-sided ribbons for $n = 4$. 24

3.1 Illustrating rulers that given rise to the decimal ruler sequence $1, 1, 1, 1, 1, 1, 1, 1, 1, 2, \ldots$ (top) and binary ruler sequence $1, 2, 1, 3, 1, 2, 1, 4, \ldots$ (bottom). 29

4.1 Twisted plain changes $\mathsf{twisted}(4)$ (experiment 8) up to its 25th entry. 32
4.2 Two-sided ribbons with distinct positive (i.e., glossy) and negative (i.e., matte) sides running in parallel. A k-twist reverses the order of k neighboring ribbons and turns each of them over. This is shown in (a) for $k = 1$ (which is also known as a flip), and in (b) for $k = 2$. Note that swap is an unsigned 2-twist, meaning that the relative order of two neighboring ribbons are reversed, but neither of the ribbons are turned over. 33
4.3 Illustrating the zig-zag pattern that extends experiment 11 from $n = 3$ to $n = 4$. (The total length of the $n = 4$ order is $n! \cdot 2^n = 24 \cdot 16 = 384$.) 40
4.4 The start of the order for Experiment 7 when $n = 4$ is $1234, 123\overline{4}, \ldots$. 41

4.5 The start of the order for Experiment 8 when $n = 4$ is $1234, 12\overline{43}, \ldots$. 41

6.1 Don Kunth's question on a "simple" revolving door algorithm for listing all spanning trees of K_n 45
6.2 A graph G for illustrating concepts in Feussner's enumeration of spanning trees . . . 48
6.3 $G \setminus e_1$, with e_1 deleted from G 48
6.4 G_{e_1}, with e_1 contracted to a vertex 48
6.5 T_4, Spanning trees of the complete graph K_4. 49
6.6 Complete graph K_4 with edges ordered according to increasing first and increasing second vertices. 51
6.7 Consequent Gray code of spanning trees. 51
6.8 Complete graph K_4 with edges ordered according to increasing first and decreasing second vertices. 51
6.9 Consequent Gray code of spanning trees. 51

A.1 `twisted.py` [68] twisted(n) (Experiment 8). It can be adapted to generate our other Gray codes by modifying the signed ruler `bases` and the `fns` that are applied for each entry in the associated signed ruler sequence. Similarly, a streamlined version (without lambda functions) can be made by replacing `fns[change]` with specific calls to the `flip` and `twist` functions. 63
A.2 `signedPerm8.cpp` [46] generates twisted(n) (Experiment 8). 64
A.3 `signedPerm.cpp` [46] generates all 12 Gray codes. 69

List of Tables

3.1 The first fifteen entries of the binary ruler sequence A007814 and its incremented version A001511. Each entry is explained by the value of n's largest divisor that is a power of two. 28

3.2 The relationship between the (incremented) binary ruler sequence A001511 and our two familiar orders of binary strings: lexicographic and binary reflected Gray code. Note that each entry in the sequence provides the number of bits that change in lexicographic order, and the specific bit that changes in BRGC order when the strings are indexed as $b_n b_{n-1} \cdots b_1$. For example, the middle entry of 4 in A001511 corresponds to the transition from 0111 to 1000 in lexicographic order (i.e., 4 bits changed) and from 0100 to 1100 in the BRGC (i.e., bit b_4 changed in $b_4 b_3 b_2 b_1$). 28

4.1 Experiments on greedy algorithms for Gray codes of signed permutations. For example, Experiment 1 prioritizes 1-twisting the values $n, n-1, \ldots, 1$ and if none of these operations succeed, then it prioritizes swapping the values $n, n-1, \ldots, 1$. On the other hand, Experiment 3 prioritizes 1-twisting value n, then swapping value n, then 1-twisting value $n-1$, then swapping value $n-1$, and so on in a zigzag or interlaced fashion. 35

5.1 Experiments on greedy algorithms and the bases of their corresponding signed ruler sequences. (Note that the bases are listed in reverse in our programs in the appendix.) 43

6.1 Gray code **3a**. 52

6.2 Gray code **4a**. 53

Abstract

A *Gray code* is an ordering of objects in which consecutive objects are similar to each other. These orders are similar in some ways to a type of puzzle known as a *word ladder*. A word ladder puzzle starts with specified first and last words, and the goal is to create a list of words in which consecutive words differ in a single letter. Given `head` and `tail`, a potential solution is `head`, `heal`, `teal`, `tell`, `tall`, `tail`. In contrast, Gray codes are typically focused on mathematical objects of a particular size and type. For example, the *binary reflected Gray code* orders the 2^n binary strings with n bits in such a way that consecutive strings differ in a single bit: $00\overline{0}, 0\overline{0}1, 01\overline{1}, \overline{0}10, 11\overline{0}, 1\overline{1}1, 10\overline{1}, 100$ is the order for $n = 4$. Similarly, *plain changes* orders the $n!$ permutations of $[n] = \{1, 2, \ldots, n\}$ so that consecutive permutations differ by a swap: $1\overleftrightarrow{23}, \overleftrightarrow{13}2, 3\overleftrightarrow{12}, \overleftrightarrow{32}1, 2\overleftrightarrow{31}, 213$ is the order for $n = 3$. These two orders are also examples of greedy Gray codes, meaning that they can be constructed by greedy algorithms. In particular, the binary reflected Gray code greedily complements the rightmost bit that gives a new binary string, while plain changes greedily swaps the largest value that gives a new permutation. The two orders can also be generated by efficient *loopless algorithms*, meaning that consecutive objects are created in worst-case $\mathcal{O}(1)$-time.

In this thesis, we consider Gray codes for the *signed permutations* of $[n]$. These objects are permutations except that each value is independently given a positive or negative sign. There are $2^n \cdot n!$ such objects for a given value of n. More specifically, each signed permutation can be seen as the product of a permutation of $[n]$ and an n-bit binary string, where the ith bit provides the sign of the value i. We provide 12 different Gray codes for the signed permutations of $[n]$. Each of the orders can be constructed greedily through different prioritizations of swaps, 1-twists and 2-twists. Furthermore, these algorithms produce elegant and memorable patterns that pay homage to both the binary reflected Gray code and plain changes. Selected proofs of correctness are presented. Furthermore, we provide loopless algorithms for all 12 Gray codes. These implementations are based on new signed variants of integer sequences known as ruler sequences that we refer to as signed ruler sequences. Finally, we discuss recent greedy basis-exchange Gray codes for matroids by Merino et. al, including Gray codes for the spanning trees of complete graph K_n. We demonstrate that one of these Gray codes is cyclic, and consider the goal of generating it efficiently.

Chapter 1

Introduction

This thesis is focused on the development of simple and efficient algorithms for generating signed permutations in Gray code order. This chapter introduces the background concepts of Gray codes, combinatorial generation, and signed permutations. New results and an outline are provided at the end of the chapter.

1.1 Gray Codes

Informally, a Gray code orders of a set of objects so that the next object always differs from the previous object by some small amount. To introduce this idea, it is helpful to consider a type of puzzle that was invented by Lewis Carroll on Christmas 1877 [8, 25]. A *word ladder*[1] is populated by a first and last word of the same length, and the player must complete the list in such a way that successive words differ in one letter. Word ladders illustrate the notions of "an ordering" and "small amount" used in Gray codes. On the other hand, Gray codes typically focus on sets that contain every object of a particular type and size, and the first and last objects are not specified. For instance, the set of all 4-letter Scrabble™ words would be a natural choice for "a set of objects" in a Gray code, and any of these 4,030 words [3] could be chosen to be first or last. These ideas are illustrated in Figures 1.1 and 1.2.

`head`, `heal`, `teal`, `tell`, `tall`, `tail`

Figure 1.1: Wikipedia's word ladder from `head` to `tail` [64]. Consecutive words differ in one letter, with the underlined letter changing to create the next word.

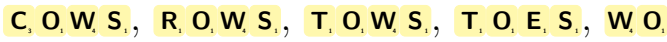


Figure 1.2: Is there a Gray code for the set of all 4-letter Scrabble™ words?

There are various ways to define Gray codes. We consider a set of objects and a set of operations, and we say that a *Gray code* is an ordering of the objects in which each successive object can be

[1] These puzzles (also called *doublets* and *word-links*) are often limited to Scrabble™ words.

obtained through a single application of one of the operations to the previous object. A formalization of this idea appears in Definition 1.

Definition 1. *Let $A = \{a_1, a_2, ..., a_m\}$ be a set of objects, $F = \{f_1, f_2, \ldots, f_k\}$ be a set of operations, and σ be a permutation of $\{1, 2, \ldots, m\}$ that will be used to order the objects. If $a_{\sigma_1}, a_{\sigma_2}, \ldots, a_{\sigma_m}$ has the property that $\exists i, f_i(a_{\sigma_j}) = a_{\sigma_{j+1}}$ for $1 \leq j < m$, then the ordering is a* Gray code*. More specifically, it is a* Gray code of A using the operations F*. If additionally $\exists i, f_i(a_{\sigma_m}) = a_{\sigma_1}$, then the ordering is a* cyclic Gray code*.*

Note that the operations are functions acting on the objects, and we often have that $f_i : A \to A$ for all i. However, there are cases when the range is not always A. For example, suppose that the objects are the spanning trees of a graph (as in Chapter 6), and the operation $t_{e,f}$ toggles the inclusion of edges e and f. In other words, if T is a set of edges forming a spanning tree, then $T' = t_{e,f}(T) = T \oplus \{e, f\}$ where $\oplus$ denotes symmetric difference (i.e., xor). Note that the resulting set T' is not always a spanning tree (even if $|T \cap \{e, f\}| = 1$). If we really wanted the range of each function to be spanning trees, then we could further specify that $t_{e,f}(T) = T$ whenever $T \oplus \{e, f\}$ is not a spanning tree. Instead we use the terms *valid* and *invalid* when referring to objects, as well as operations applied to certain objects, when necessary.

Valid operations are also known as *changes* or *flips*. For example, we can say that a cyclic Gray code is a Gray code in which the last object can be transformed into the first object by a flip. When discussing English words, we will assume that the flips are single letter changes, although different operations can also be considered. The term *minimal change order* also refers to our notion of a Gray code.

1.1.1 Flip Graphs

It can be helpful to visualize Gray codes (and word ladders) using graphs. The underlying *flip graph* (or *change graph*) $\mathcal{G}(A, F)$ contains one node for each object in A, and edges connect two objects that differ by a flip in F. If the operations are closed under inversions (i.e., if $f \in F$, then $f^{-1} \in F$), then we consider $\mathcal{G}(A, F)$ to be an undirected graph, and otherwise it is a directed graph.

For example, our discussion in Section 1.1 can be modeled by the *word graph* $\mathcal{W}_n$. This undirected graph contains one node for each n-letter word, and its edges connect any two words that differ in a single letter. In particular, the word graph $\mathcal{W}_4$ contains 4,030 nodes, each of which is a 4-letter word.

- A word ladder is a path between two specified words of length n in $\mathcal{W}_n$.
- Gray codes of n-letter words correspond to Hamilton paths in $\mathcal{W}_n$, and cyclic Gray codes correspond to Hamilton cycles in $\mathcal{W}_n$.

Figure 1.3 shows the (unique) shortest path between `head` and `tail` in $\mathcal{W}_4$. It also includes longer solutions to the associated word ladder puzzle, including one that visits a word (`veal`) that would concern our college's mascot.

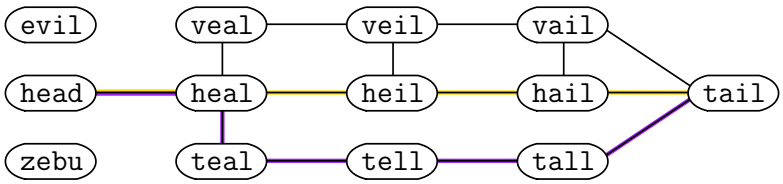


Figure 1.3: A subgraph of the word graph $\mathcal{W}_4$. The purple path is the word ladder from `head` to `tail` from Figure 1.1, while the gold path provides a shorter solution. Nodes `evil` and `zebu` have degree zero in the graph. In other words, no other 4-letter Scrabble™ word can be obtained by changing one of their letters.

Figure 1.3 also shows that some nodes in $\mathcal{W}_4$ have *degree* (i.e., number of neighbors) equal to zero. This means that the graph is disconnected, and it is impossible to solve some word ladder problems. It also means that there is no Gray code for 4-letter words, so the answer the question in Figure 1.2 is no.

Next we will look at two of the most important (and well-known) Gray codes.

1.1.2 Binary Reflected Gray Code

The modern history of Gray codes begins with the *binary reflected Gray code*, which appeared in multiple patents from Bell Labs in the 1940s. The term *reflected binary code* was used by Frank Gray in his 1947 patent application, in which he states that the order "may be built up from the conventional binary code by a sort of reflection process" [23] as will be explained below. The eponymous *Gray code* also demonstrates Stigler's law [62], as the same order appeared earlier in a 1941 patent application by George R. Stibitz, another Bell Lab researcher. For additional historical remarks from the 1970s see [19] and [26].

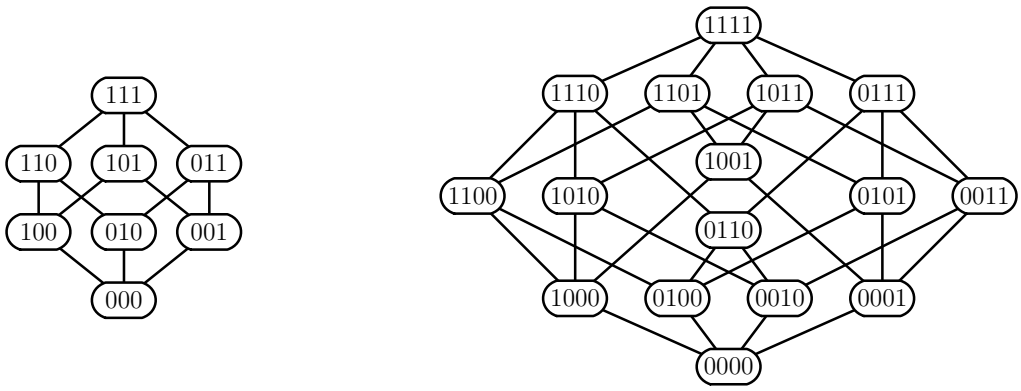
Regardless of its name or origins, the objects in question are the n-bit binary strings, which we denote as B_n. For example, $B_2 = \{00, 01, 10, 11\}$ and in general, $|B_n| = 2^n$. The operations are individual *bit-flips*, meaning that a single bit is complemented from 0 to 1, or vice versa. Given these objects and operations, the underlying flip graph is the *n-dimensional cube* $\mathcal{H}_n$, as illustrated in Figure 1.4.

We index the bits in a binary string $\mathbf{b} \in B_n$ as $\mathbf{b} = b_n b_{n-1} \cdots b_1$ (i.e., right-to-left starting with index 1). The $\cdot$ operation denotes concatenation and is frequently omitted (i.e., $0 \cdots 1 = 01$). The set B_n can then be created from the set B_{n-1} as follows (where color is added for emphasis).

$$B_n = \{0 \cdot \mathbf{b} \mid \mathbf{b} \in B_{n-1}\} \cup \{1 \cdot \mathbf{b} \mid \mathbf{b} \in B_{n-1}\} \tag{1.1}$$

In other words, we can create B_n by taking two copies of each $\mathbf{b} \in B_{n-1}$, and prefixing 0 to one copy, and prefixing 1 to the other copy.

As Gray mentioned, and as its name implies, the binary reflected Gray code can be constructed using reflection. More specifically, the second copy of the order on $n-1$ bits is *reflected* (i.e., the



(a) $\mathcal{H}_3$ (b) $\mathcal{H}_4$

Figure 1.4: The vertices of the n-dimensional cube are the n-bit binary strings B_n, and the edges connect those that differ in a single bit.

last string goes first and the first string goes last). The order is formally defined as

$$\mathsf{brgc}(n) = 0 \cdot \mathsf{brgc}(n{-}1),\ 1 \cdot \mathsf{reflect}(\mathsf{brgc}(n{-}1)) \tag{1.2}$$

where $\mathsf{brgc}(1) = 0, 1$ provides the base case. In this formula (1.2), the $0\cdot$ denotes prefixing 0 to the front of every string in $\mathsf{brgc}(n{-}1)$, and similarly, $1\cdot$ denotes prefixing 1 to the front of every string in $\mathsf{reflect}(\mathsf{brgc}(n{-}1))$. As mentioned above, the $\mathsf{reflect}$ operation reorders $\mathsf{brgc}(n{-}1)$ so that its last string is first, and its first string is last. Finally, the comma in the middle is used to join the two suborders into a single order. The application of this recursive formula is illustrated below in (1.3) and (1.4).

Note that this definition means that the most significant digit is flipped only once when transitioning between the two copies of $\mathsf{brgc}(n-1)$. To illustrate this process, $\mathsf{brgc}(3)$ is listed as the following:

$$\mathsf{brgc}(3) = 000, 001, 011, 010, 110, 111, 101, 100. \tag{1.3}$$

To generate the $n = 4$ BRGC, we use two copies of the above list and prefix 0 and 1 to the two copies, respectively, and the second copy is reflected:

$$\begin{aligned}\mathsf{brgc}(4) = {} & 0000, 0001, 0011, 0010, 0110, 0111, 0101, 0100, \\ & 1100, 1101, 1111, 1110, 1010, 1011, 1001, 1000.\end{aligned} \tag{1.4}$$

We refer to the recursion used in (1.2) as *global recursion* because it operates on the suborders $\mathsf{brgc}(n-1)$ as individual units, rather than considering their individual objects. In other words, it doesn't split up the $\mathsf{brgc}(n-1)$ suborders, so they appear in $\mathsf{brgc}(n)$ (although the second copy is reflected).

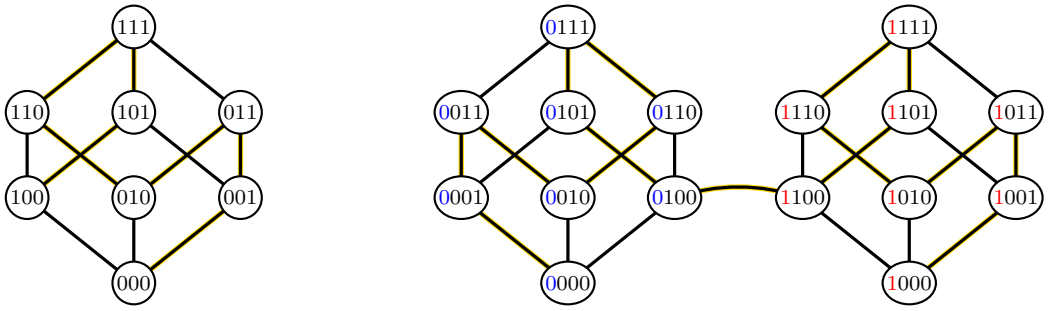
To fully understand why (1.4) is indeed a Gray code for B_4 using bit-flips, we can make several simple observations. First, note that (1.4) contains each object in B_4 exactly once, and more generally, this holds for all n by (1.1) and (1.2). Second, note that every transition within the first

half of brgc(4) is a single bit-flip due to the fact that this is true for brgc(3), and similarly, this is true for the second half of brgc(4) since reflecting brgc(3) does not change this property. The same argument holds for arbitrary n. Third, note that the transition from the last string in the first half of brgc(4) to the first string in the second half of brgc(4) is the transition 0100, 1100, which is a bit-flip of the most significant digit. To ensure that this property holds for all n we state two simple remarks that follow easily by (1.2) and induction on n. These remarks use the convention that exponentiation denotes repeated concatenation. For example, the last string in (1.4) is $10^3 = 1000$.

Remark 1. *The first string in* brgc(n) *is* 0^n.

Remark 2. *The last string in* brgc(n) *is* 10^{n-1}.

These remarks ensure that the first half of brgc(n) ends with 010^{n-2}, and the second half of brgc(n) begins with 110^{n-2} (keeping in mind that the first string in reflect(brgc($n-1$)) is the last string in brgc($n-1$)). As previously mentioned, this transition is the only time in the order that the leftmost bit is changed, and Figure 1.5 highlights this bridge graphically for $n = 4$. Remarks 1–2 also imply that brgc(n) is a cyclic Gray code.



(a) brgc(3) in $\mathcal{H}_3$ (b) brgc(4) in $\mathcal{H}_4$

Figure 1.5: Illustrating how the binary reflected Gray code has a single bridge transition between its two subcubes. Note that the left copy of $\mathcal{H}_3$ in (b) has been reflected left-to-right relative to (a) in order to simplify the drawing.

Outside of this thesis, the term *Gray code* often refers exclusively to bit-flip orders of n-bit binary strings. In fact, the term sometimes refers only to the specific order discovered by Stibitz, Gray, and others.

Application: Rotary Encoders

Binary Gray codes using bit-flips, including the binary reflected Gray code, are used in a large number of applications in a variety of fields. One such example is a *rotary encoder*. These electromechanical devices convert the angular position of a shaft into a digital signal. For example, when you rotate the volume dial in your car, the stereo converts its rotational position into a digital value. This is done by placing n concentric rings of magnetic (or optical) material around the back of the dial to represent the least significant bit b_1 (outer ring) to most significant bit b_n (inner ring). The area is partitioned into 2^n sectors or wedges, each of which is patterned radially with a distinct

n-bit pattern. An n-bit sensor, which does not spin, is situated radially behind the dial, and it detects which of the 2^n sectors, or states, it is aligned with. A potential problem arises whenever the dial is rotated in such a way that two sectors are straddling the sensor. If k different bits change between the binary patterns on the two neighboring sectors, then the sensor could read the state in 2^k different ways. By using a cyclic Gray code like $\mathsf{brgc}(n)$, we ensure that $k = 1$ bits differ between neighboring sectors, thereby ensuring that the state will be read in $2^1 = 2$ possible ways (i.e., as one of the two sectors straddling the sensor).

The encoding used in a rotary encoder appears in Figure 1.6 for $n = 4$. Besides being functional, the diagram also gives a birds-eye view of $\mathsf{brgc}(4)$, and this type of visualization is now widely used to illustrate other types of Gray codes [43]. Commercially available rotary encoders often have at least 10 rings and over a thousand states [1].

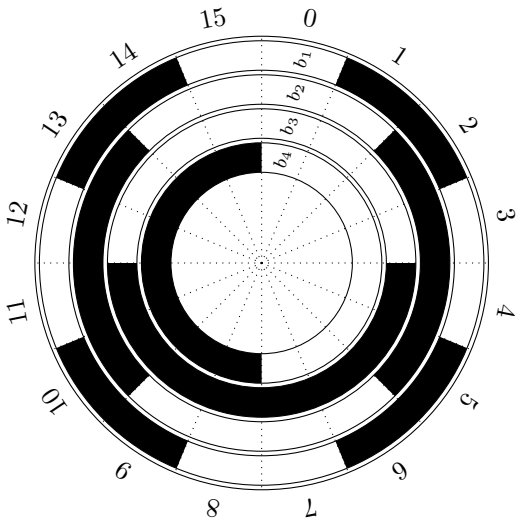


Figure 1.6: An n-bit rotary encoder for $n = 4$. It contains n rings and 2^n sectors, with each sector having a different binary pattern $b_n b_{n-1} \cdots b_1 \in B_n$. Successive sectors follow the binary reflected Gray code $\mathsf{brgc}(n)$ in clockwise order (e.g., $b_4 b_3 b_2 b_1 = 0000, 0001, 0011, \ldots, 1000$) with white/black for 0/1. If the n-bit sensor straddles two sectors, then the Gray code property ensures that only one bit has an ambiguous value, and that the state will be read as being in either of the straddling sectors. Image adapted from [22].

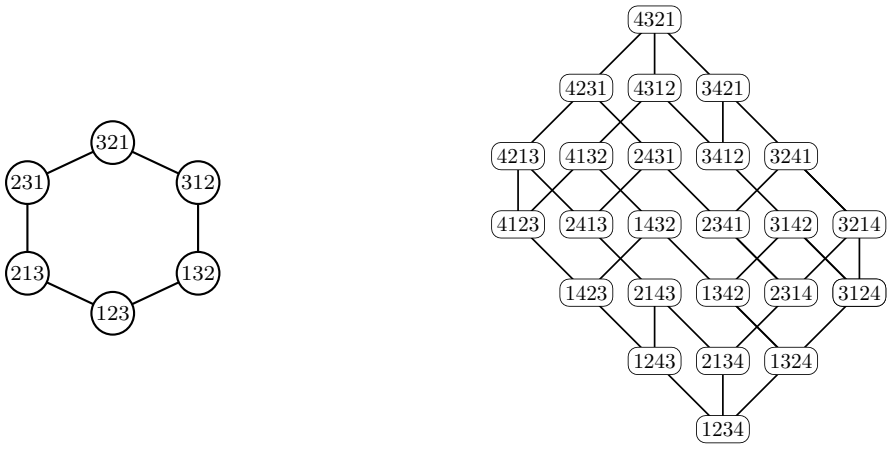
1.1.3 Plain Change Order

Perhaps the most well-known Gray code after the binary reflected Gray code is plain changes. This order is not based on n-bit binary strings and the bit-flip operation, but rather on permutations and swaps.

A *permutation* of the set $S = \{s_1, s_2, ..., s_n\}$ can be loosely defined as an arrangement of elements in s_n into a linear order. For example, if $S = \{a, b, c, d\}$, then the permutations of S (written in one-line notation) include $abcd, abdc, cabd, dacb$ and so on. In particular, we let S_n be the set of permutations of $[n] = \{1, 2, \ldots, n\}$. For example, $S_3 = \{123, 132, 213, 231, 312, 321\}$ and in general,

$|S_n| = n!$.

A *transposition* on a permutation can be defined as interchanging the position of two elements in the permutation. For example, we can swap 1 and 4 in the permutation $1234 \in S_n$ to obtain $4231 \in S_n$. An *adjacent-transition* is a transposition in which the two positions are consecutive, or equivalently, the transposed elements are next to each other in the permutation. Adjacent-transpositions are also known as *swaps*. Given these objects and operations, the underlying flip graph is the *permutohedron* $\mathcal{P}_n$, as illustrated in Figure 1.7.



(a) $\mathcal{P}_3$ (b) $\mathcal{P}_4$

Figure 1.7: The vertices of the permutohedron $\mathcal{P}_n$ are the permutations S_n, and the edges connect those that differ by an adjacent transposition (i.e., swap).

We index permutations in the same way that we index binary strings. That is, if $\mathbf{p} \in S_n$, then $\mathbf{p} = p_n p_{n-1} \dots p_1$. The set S_n can be created from the set S_{n-1} as follows (where color is added for emphasis).

$$\begin{aligned} S_n = \{ \ & n \cdot p_{n-1}p_{n-2}p_{n-3}\cdots p_1 \mid \mathbf{p} \in S_{n-1}\} \cup \\ & \{p_{n-1} \cdot n \cdot p_{n-2}p_{n-3}\cdots p_1 \mid \mathbf{p} \in S_{n-1}\} \cup \\ & \{p_{n-1}p_{n-2} \cdot n \cdot p_{n-3}\cdots p_1 \mid \mathbf{p} \in S_{n-1}\} \cup \\ & \dots \\ & \{p_{n-1}p_{n-2}p_{n-3}\cdots p_1 \cdot n \mid \mathbf{p} \in S_{n-1}\} \end{aligned} \tag{1.5}$$

In other words, we can create S_n by taking n copies of each $\mathbf{p} \in S_{n-1}$, and inserting the symbol n into each of the n possible positions. This is done in (1.5) by splitting each $\mathbf{p} \in S_{n-1}$ string into two substrings in all n ways.

Notice that the column of strings in (1.5) differ by swaps for a fixed $\mathbf{p} \in S_{n-1}$. For example, the strings in the first row and second row differ by a swap of the values n and $n-1$. The strings in the second and third row then differ by a swap of the values n and $n-2$ swap, and so on. We'll refer

to this left-to-right motion as *zagging* n through $\mathbf{p}$. Similarly, the reverse right-to-left motion (i.e., obtained by reading (1.5) from bottom to top) is *zigging* n through $\mathbf{p}$.

Plain changes is created by alternately zigging and zagging n through successive permutations in the order of S_{n-1}. We let zig_n and zag_n give the length n lists that repeatedly swap n to the left or right as described above. The order is formally defined as

$$\begin{aligned}\mathsf{plain}(n) = \mathsf{zig}_n(\mathbf{p_1}),\ \mathsf{zag}_n(\mathbf{p_2}),&\\ \mathsf{zig}_n(\mathbf{p_3}),\ \mathsf{zag}_n(\mathbf{p_4}),&\\ \mathsf{zig}_n(\mathbf{p_5}),\ \mathsf{zag}_n(\mathbf{p_6}),&\\ \ldots,&\\ \mathsf{zig}_n(\mathbf{p_{(n-1)!-1}}), \mathsf{zag}_n(\mathbf{p_{(n-1)!}})&\end{aligned} \tag{1.6}$$

where $\mathsf{plain}(1) = 1$ and $\mathsf{plain}(2) = 12, 21$ are base cases, and $\mathsf{plain}(n-1) = \mathbf{p_1}, \mathbf{p_2}, \ldots, \mathbf{p_{(n-1)!}}$.[2] We refer to the recursion used in (1.6) as *local recursion* because it operates on the individual objects within the suborder $\mathsf{plain}(n-1)$. Note that two bases were used as the end of (1.6) assumes that $(n-1)!$ is even.

To illustrate this process, the plain change order for $n = 3$ is given below.

$$\mathsf{plain}(3) = 123, 132, 312, 321, 231, 213. \tag{1.7}$$

To generate plain changes for $n = 4$ we zig and zag n through each successive permutation as follows.

$$\begin{aligned}\mathsf{plain}(4) &= \mathsf{zig}_4(123),\ \mathsf{zag}_4(132),\\ &\quad\ \mathsf{zig}_4(312),\ \mathsf{zag}_4(321),\\ &\quad\ \mathsf{zig}_4(312),\ \mathsf{zag}_4(321),\\ &= 1234, 1243, 1423, 4123,\ \ 4132, 1432, 1342, 1324, &(1.8)\\ &= 3124, 3142, 3412, 4312,\ \ 4321, 3421, 3241, 3214, &(1.9)\\ &= 2314, 2341, 2431, 4231,\ \ 2134, 2413, 2143, 2134 &(1.10)\end{aligned}$$

To fully understand why (1.8) is indeed a Gray code for S_4 using swaps, we can make several simple observations. First, note that (1.8) contains each object in $\mathsf{plain}(4)$ exactly once, and more generally, this holds for all n by (1.5) and (1.6). Second, note that every transition within an individual zig and zag sublist of S_4 is a swap, and this holds for arbitrary n. Third, note that the transition from the last string in zig sublist to the first string in the next zag sublist has the form $4\mathbf{p}_i$, $4\mathbf{p}_{i+1}$, which is a swap due to 4 being on the left side and $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ differing by a swap in (1.7). Similarly, the transition from the last string in zag sublist to the first string in the next zig sublist has the form $\mathbf{p}_i4$, $\mathbf{p}_{i+1}4$, which is a swap due to 4 being on the right side and $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ differing by a swap in (1.7). The same result holds for arbitrary n.

[2] Bold is used in each $\mathbf{p_i}$ to denote that it is the i^{th} permutation in the suborder, whereas the lack of bold would indicate that p_i is a specific symbol in a permutation.

Values 1 and 2 are swapped only once in $\mathsf{plain}(n)$. An additional swap makes the order cyclic. This is true by the following remarks which use induction on n.

Remark 3. *The first permutation in* $\mathsf{plain}(n)$ *is* $1234\cdots n$.

Remark 4. *The last permutation in* $\mathsf{plain}(n)$ *is* $2134\cdots n$.

It may be tempting to dismiss plain changes as trivial mathematics. To counter such an urge, consider the following anecdote. In 1964, Ron Graham and A. J. Goldstein attempted to generate permutations by swaps. They were unaware of plain changes, and their result [21] is nowhere near as elegant[3]. In other words, plain changes is not always obvious even for brilliant mathematicians. On the other hand, it is simple enough to have been rediscovered many times, as we will see next.

Application: Bell Ringing

Plain changes was discovered by bell ringers in the 1600s [13] (see Knuth [32]), and it has been used continually by bell ringers (especially English bell ringers) until today. The permutation was used a sequence for ringing n church bells, and an *extent* rings all $n!$ sequences in succession. Due to inertia, a bell's order cannot change arbitrarily from one sequence to the next, but swaps are feasible. (i.e., one bell moves one position earlier in the sequence, while another moves one position later. Bell-ringers moved on to more complex patterns with multiple swaps between permutations [4], but the dawn of efficient computing led plain changes to be rediscovered multiple times under the *Steinhaus-Johnson-Trotter algorithm* moniker [59] [30] [63]. Regardless of its name, Sedgewick referred to the order as "perhaps the most prominent permutation enumeration algorithm" in a survey from 1977 [58].

1.1.4 Additional Gray Codes

While the binary reflected Gray code and plain changes are two mainstream Gray codes that have widespread applications, there are many other Gray codes that are interesting to explore. These include Gray codes for other combinatorial objects (e.g., combinations, partitions, graphs, and so on).

In Section 2.4 we will see two additional Gray codes for permutations, including one that generalizes to colored permutations. A *colored permutation* is a permutation in which each value is assigned one of c colors. Standard permutations are obtained when $c = 1$, and signed permutations are obtained when $c = 2$ (i.e., the positive and negative signs are interpreted as two different colors). There exist other generalizations of permutations including *k-permutations* (where only $k \leq n$ values from $[n]$ are included) and *multiset permutations* (where the set $[n]$ is replaced by a multiset). There are well-known Gray codes for these generalizations (e.g., see [29] for k-permutations and [65] for multiset permutations). In contrast, Gray codes for signed permutations have not been studied in as much detail, with [34] providing one such example.

[3] An internal Bell Labs report [20] contains an implementation of [21], but the authors have not been able to obtain a copy of it.

Additional Gray codes also exist for binary strings. In particular, a *Beckett-Gray code* is based on Samuel Beckett's play *Quad* [5]. When designing this play, Beckett attempted to create $2^4 = 16$ different scenes with the property that successive scenes differed by having one cast member enter the stage or leave the stage. If this were the only constraint, then the binary reflected Gray code brgc(4) would have sufficed. This is because each subset of cast members can be represented by a binary string $b_1b_2b_3b_4$ (with $b_i = 1$ indicating that cast member i is on-stage) and successive scenes would differ in a single bit. However, Beckett also wanted to have an additional property: the only cast member who can exit the scene is the member who had been on stage the longest amount of consecutive prior scenes. Later research showed that these restricted Gray codes exist for $n = 2, 5, 6, 7, 8$ and do not exist for $n = 3, 4$ (including *Quad*). For further information on Beckett-Gray codes see [57], [11], [9].

For a wider introduction to Gray codes in general, we recommend the classic survey by Savage [52], as well as a more recent survey by Mütze [42]. We also note that questions on the existence of Gray codes are often NP-complete when limited to a subset of objects that are provided as part of the input [40].

1.2 Combinatorial Generation

A common task in computer programming is to generate every instance of a particular combinatorial object. This task arises for many different reasons, including testing. For example, one may wish to generate all n-bit binary strings in order to test a (simulated) circuit with n inputs. Alternatively, one may wish to generate all permutations of $[n]$ in order to test if a student's implementation of a sorting algorithm is correct.

The area of *combinatorial generation* is focused on these tasks. In combinatorial generation, the baseline goal is to generate each object exactly once. Typically there are exponentially many instances of the combinatorial object in question (e.g., 2^n binary strings or $n!$ permutations). For this reason, we want to create each object once in memory, without putting storing every object simultaneously in a list. When writing pseudocode we use the term *visit* to signify that the next object is available for the application to use. Many programming languages have iterators and generators to facilitate this particular approach. When working with other languages, generation algorithms often allow applications to provide a `visit` function, which is then called whenever the next object is available.

When evaluating the efficiency of a combinatorial generation algorithm, we focus on the amount of time and memory required to create the next object. For example, an algorithm that has *polynomial-delay* means that it is always able to create the next object in polynomial-time with respect to the size of the object. When working with simpler combinatorial objects, the goal is often $O(1)$-time delay, either in a worst-case or amortized sense. These algorithms are known as *loopless* and *constant amortized time (CAT)* algorithms, respectively.

1.2.1 Lexicographic Order

As Ruskey explains in *Combinatorial Generation* [48], humans have been making exhaustive lists of various kinds for thousands of years. More often than not, these lists have been in some variation of alphabetic or numeric order. More generally, we use the term *lexicographic order* to refer to orders that are applied symbol-by-symbol from left to right. More precisely, lexicographic order generalizes alphabetical order to any totally ordered set $P = p_1, p_2, ..., p_n$. Assuming a and b are two different words created with elements in P with the same length, then $a < b$ if and only if there exists i such that $a_i < b_i$. Figure 1.8 illustrates lexicographic order for the words found in the word ladder at the beginning of this chapter.

head, heal, tail, tall, teal, tell

Figure 1.8: The lexicographic ordering of the words found in Figure 1.1. Note that consecutive words in this order differ in up to three letters.

Lexicographic order is the most commonly used order in combinatorial generation. While the order is straightforward and easy to understand, it is usually not the most efficient order for working with simple combinatorial objects. This is because there is no limit on how much two consecutive objects in the order can differ. For example, the binary string 01^{n-1} is followed by 10^{n-1} (i.e., all bits change) in lexicographic order. Similarly, the permutation $1nn{-}1\cdots2$ is followed by $2134\cdots n$ (i.e., all symbols change) in lexicographic order. As a result, lexicographic order algorithms typically require $\otimes(n)$-time to create the next object.

1.2.2 Loopless Algorithms

A combinatorial generation algorithm is *loopless* if it generates each successive object in worst-case O(1)-time [15]. This term comes from the fact that these algorithms typically have no inner loops. These algorithms are also known as *loop-free* algorithms.

At first the goal of a loopless algorithm may seem impossible. If the size of the combinatorial object is parameterized by n, then how can successive objects be created in $O(1)$-time? The trick is that only one object needs to be stored in memory, and the generation algorithm can simply *change* this object to effectively create the next one. In other words, the speed of the algorithm is limited not by the size of the objects, but by the size of the changes. If the algorithm follows a Gray code order, and the algorithm stores the object in a suitable data structure for the Gray code's changes, then a loopless algorithm becomes possible.

It is notable that there are loopless algorithms to generate both the binary reflected Gray code and plain change order [15]. In both cases, the algorithms are based on generating the changes made between successive objects, as discussed in more detail in Chapter 3.

A weaker goal is to generate each successive object in amortized O(1)-time. These algorithms are called *constant amortized time* (CAT) algorithms. In these algorithms it often takes constant time to create the next object, but sometimes it takes longer. Depending on the combinatorial object, it may also be possible for a CAT algorithm to use lexicographic order. For example, this is

possible for binary strings, since an average of two bits change when creating the next binary string in lexicographic order. CAT algorithms are weaker than loopless algorithms in a theoretical sense, but they are often just as fast or faster in practice.

1.2.3 Application: Exact Algorithms

Combinatorial generation is central to many applications beyond testing, such as exact algorithms to NP-complete problems. In these problems Gray code algorithms can be particularly useful. For example, a traveling salesman problem on n cities can be solved by generating all $n!$ permutations of $[n]$, with each member of S_n providing a possible route through the cities (e.g., $p_1p_2\cdots p_n \in S_n$ represents the route $p_1 \to p_2 \to \cdots \to p_n$). Plain changes is advantageous because successive routes differ in at most three segments (e.g., swapping p_ip_{i+1} to $p_{i+1}p_i$ replaces segment $p_i \to p_{i+1}$ with $p_i \to p_{i+2}$) [28]. Thus, the distance of each successive route can be *updated* in constant time.

Now consider a variant of the renowned Travel Salesman Problem (TSP) involving trains, where each of the n stations can be entered/exited in one of two orientations (e.g., the train may travel along the station's eastbound or westbound track). Note that the time taken to travel from one station to another depends on these orientations. As a result, there are $2^n \cdot n!$ possible routes and they correspond to objects that we will refer to as signed permutations.

1.3 New Results

This thesis provides new Gray codes for signed permutations, along with simple loopless algorithms for generating them.

Informally, a signed permutation is the product of a binary string and a permutation. More formally, a *signed permutation* of $[n]$ is a permutation of $[n]$ in which every symbol is appended a $\pm$ sign in the front. Let $S_n^{\pm}$ be the set of signed permutations of $[n]$. For example, the signed permutations for $n = 2$ appear below.

$$S_2^{\pm} = \{+1+2,\ +1-2,\ -1+2,\ -1-2,\ +2+1,\ +2-1,\ -2+1,\ -2-1\} \tag{1.11}$$

More generally, $|S_n^{\pm}| = 2^n n!$.

We visualize signed permutations using two-sided ribbons. A two-sided ribbon is polished only on one face, and we model a signed permutation with n two-sided ribbons running in parallel. One of our results is a "twist Gray code" for signed permutations: each successive signed permutation in is created by a 1-twist (turn over one ribbon) or a 2-twist (turn over two consecutive ribbons and reverse their order). More specifically, the order is created by greedily 2-twisting the largest possible value (ignoring the signs) followed by 1-twisting the largest possible value. These ideas are further explained in the coming chapters, with visualizations of the order appearing at the start of Chapter 4 (see Figures 4.1 and 4.2). A conference paper discussing this particular result was presented at the LATIN 2024 conference [47].

This thesis explains how our new twist Gray code order was discovered, and how it fits into the

existing literature on Gray codes and combinatorial generation. More broadly, we create 12 new Gray code algorithms for efficiently generating signed permutations. Each of these results is inspired by the binary reflected Gray code and plain changes.

1.4 Outline

In Section 1.1, we defined the binary reflected Gray code and plain changes using global and local recursion, respectively. Instead it is possible to generate these orders using simple greedy algorithms. The greedy Gray code algorithm is investigated in Chapter 2.

Loopless algorithms for generating the binary reflected Gray code and plain changes are based on describing and generating their change sequences. In both cases, the change sequences follow a type of pattern known as a ruler sequence. Ruler sequences and loopless algorithms for generating them are provided in Chapter 3.

Chapter 4 discusses our experiments on greedily generating Gray codes of signed permutations. We investigate 20 different potential algorithms, and find empirical evidence that 12 of them work. Then we provide proofs for three representative algorithms.

Chapter 5 explains how our new Gray codes can be generated by loopless algorithms using various signed ruler sequences. Python and C++ implementation of our loopless algorithms appear in the appendix.

Chapter 6 concludes with further greedy experiments involving Gray codes for spanning trees.

Chapter 2

The Greedy Gray Code Algorithm

This chapter introduces the greedy Gray code algorithm and compares it to backtracking. Greedy reinterpretations of several classic Gray codes are then discussed, including the binary reflected Gray code (BRGC) and plain change order. Rulesets that do not produce proper Gray codes are also covered.

2.1 The Greedy Gray Code Algorithm

Historically, the majority of Gray codes have been initially discovered and defined in a recursive manner [48]. We prefer another approach that proceeds iteratively. It has provided simpler interpretations of existing Gray code constructions and facilitated the discovery of new orders (including our new Gray codes for signed permutations that will be discussed in Chapter 4).

Let $A = \{a_1, a_2, ..., a_m\}$ be a set of m objects, and F be a set of k operations. The *greedy Gray code algorithm* [66] attempts to create a Gray code one object at time based on two parameters: a start object $s \in A$, and a prioritized list of the operations $\langle f_1, f_2, \cdots, f_k \rangle$. At each step, the algorithm extends the current order to a new object by applying an operation to the most recently created object. The modification with highest possible priority (i.e., the operation with the smallest index) that creates a new valid object is used, and the resulting object is added to the end of the current order. If none of the modifications create a new object, then the algorithm terminates. Pseudocode appears in Algorithm 1. We use $\langle\rangle$ to enclose ordered structures, with $+$ to extend them.

This algorithm is not meant to be efficient. More specifically, it remembers every previous object, so it uses exponential memory when it generates exponentially many objects. However, it has two benefits. First, it can provide simpler descriptions of Gray codes that were previously defined using recursion. Second, it provides an easy way to search for simple new Gray codes. To contextualize the second point, we compare the algorithm to backtracking.

Algorithm 1 The greedy Gray code algorithm. The algorithm attempts to build a Gray code for objects A using operations F. The input is a start object $s \in A$ and a prioritization of the operations $\langle f_1, f_2, \ldots, f_k \rangle$. The algorithm is not guaranteed to find a suitable Gray code, even if one exists starting from s.

```
1:  procedure GreedyGray(s, A, ⟨f_1, f_2, ..., f_k⟩)
2:      G = ⟨s⟩                          ▷ Initialize the Gray code order to the start object
3:      a = s                            ▷ Set the most recently added object to be a
4:      for i = 1, 2, ..., k do          ▷ There are k operations to consider
5:          b = f_i(a)                   ▷ Apply the ith operation to create b
6:          if b ∈ A & b ∉ G then        ▷ Check if b is both valid and new
7:              G = G + b                ▷ Extend the order to include b
8:              a = b                    ▷ Update the most recent object a
9:              i = 1                    ▷ Reset the loop counter
10:     return G                         ▷ Return the order that was created
```

2.1.1 Backtracking

Backtracking provides another way to search for Gray codes. This approach also builds an order one object at a time, but there is a critical difference. When no modifications can be used to extend the current order, the most recently created object is removed, and the algorithm goes back to considering alternate modifications for the second most recently created object. In other words, when backtracking reaches a dead end, it goes backward to consider other choices. The greedy algorithm never does this, so it is like backtracking without any backtracking. Pseudocode appears in Algorithm 2, along with a discussion of how it differs from a recursive version of the greedy Gray code algorithm.

Algorithm 2 Backtracking algorithm for Gray codes of objects A starting from $s \in A$ using operations F. The parameters are the most recent object s and the partial Gray code G built thus far. The initial call should omit G and it will be set using the default value $G = \langle s \rangle$. From the initial call, the algorithm is guaranteed to return a Gray code starting from s if one exists. We assume that the set of objects A, and a prioritization of the operations $\langle f_1, f_2, \ldots, f_k \rangle$ are globally available. To convert this algorithm to a recursive version of Algorithm 1, simply delete line 8 (i.e., always **return** G'), which removes all backtracking; one difference is that unsuccessful searches return the empty list $\langle\rangle$ here.

```
1:  procedure backtrack(s, G = ⟨s⟩)
2:      if |G| == |A| then               ▷ Check if every object has been reached
3:          return G
4:      for i = 1, 2, ..., k do
5:          b = f_i(s)                   ▷ Apply the ith operation to create b
6:          if b ∈ A & b ∉ G then        ▷ Check if b is both valid and new
7:              G' = backtrack(b, G + b)
8:              if |G'| == |A| then
9:                  return G'
10:     return ⟨⟩
```

At this point, the purpose of the greedy Gray code algorithm may seem unclear. After all, the backtracking algorithm is a strict improvement over the greedy algorithm in terms of being able

to find a Gray code. While this observation is true, it also misses the point. When backtracking succeeds, the resulting Gray code order can be quite complicated. On the other hand, when the greedy approach succeeds, it is quite likely that the resulting order will be simple. As discussed in Chapter 4, this is particularly helpful when searching for new Gray codes that have the potential to be generated efficiently.

Next we discuss greedy reinterpretations of several classic Gray codes. These reinterpretations were first observed in [66].

2.2 Greedy Implementation of BRGC

As discussed in the section 1.1.2, BRGC orders n-bit binary strings by *bit-flips*, or successive strings differ in one bit. Previous research revealed that BRGC can be generated starting at the all-0s string by greedily flipping the rightmost possible bit [66]. For example, the order for $n = 4$, $\mathsf{brgc}(4)$, begins as follows, where overlines denote the bit that is flipped to create the next binary string,

$$\mathsf{brgc}(4) = 000\overline{0}, 00\overline{0}1, 001\overline{1}, 0010, \ldots. \tag{2.1}$$

To continue (2.1) we consider bit-flips in 0010 from right to left. We can't flip the right bit since $001\overline{0} = 0011$ is already in the list. Similarly, $00\overline{1}0 = 0000$ is also in the list. However, $0\overline{0}10 = 0110$ is not in the list, so this becomes the next string. Algorithm 3 provides pseudocode for this algorithm, with a Python implementation the Appendix.

The full BRGC order for $n = 4$ is visualized in Figure 2.1 using two-sided ribbons, where each bit-flip is a 1-twist of the corresponding ribbon. Note that the order is *cyclic*, as the last and first strings differ by flipping the first bit.

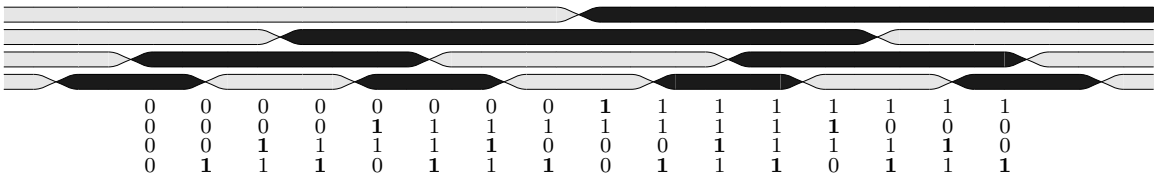


Figure 2.1: Binary reflected Gray code using indistinct two-sided ribbons for $n = 4$.

Remark 5. *If the BRGC is started in* $100\cdots00$ *(the last string of the original BRGC) instead and generated with the same greedy rule, then the resulting sequence is exactly the reversed sequence of BRGC.*

Remark 6. *Through applying the same greedy rule starting at any binary string, a Gray code is generated. In particular, the final string is equal to the first string but with the first bit complemented,*

Algorithm 3 Greedy algorithm for generating binary reflected Gray code $\mathsf{brgc}(n)$.

```
1:  procedure brgc(n)                          ▷ Binary strings are visited in brgc(n) order
2:      π ← 00···00                            ▷ Starting binary string s = π ∈ B_n
3:      visit(π)                               ▷ Visit π for the first and only time
4:      S = {π}                                ▷ Add π to the visited set
5:      T = {f_n, f_{n−1}, ··· , f_1}          ▷ Set prioritization of operations T
6:      i ← 1                                  ▷ 1-based index into T; T[1] = f_n will flip the n-th digit
7:      while i ≤ n do                         ▷ Index i iterates through the n flips in T
8:          π′ ← T[i](π)                       ▷ Apply the i^th highest priority flip to create π′
9:          if π′ ∉ S then                     ▷ Check if π′ is a new binary string
10:             π ← π′                         ▷ Update the current binary string π
11:             visit(π)                       ▷ Visit π for the first and only time
12:             S = S ∪ {π}                    ▷ Add π to the visited set
13:             i ← 1                          ▷ Reset the 1-based index into T
14:         else
15:             i ← i + 1                      ▷ If π′ ∈ S, then consider the next flip
```

2.3 Greedy Interpretation of Plain Changes

A straightforward greedy realization of plain change order on the set $\{1, 2, ..., n\}$ also exists: start at $12\cdots n$ and then swap the largest possible digit with its adjacent digit [66]. Such a definition may seem to be ambiguous at a first glance: should we swap a symbol to the left or the right? Nevertheless, it turns out that the largest possible digit (the digit to be swapped) is either in the leftmost or rightmost position, or the opposite swap recreates a previous permutation. A proof of this simple fact is included in [66].

Figure 2.2: Plain changes $\mathsf{plain}(n)$ using distinct one-sided ribbons for $n = 4$.

Remark 7. *If the plain change is started in* $2134\cdots$ *(the last string of the original plain change order) instead and generated with the same greedy rule, then the resulting sequence is exactly the reversed sequence of plain change.*

Algorithm 3 illustrates the structure of a greedy algorithm that generates plain changes.

2.4 Greedy Interpretation of Other Gray Codes

We have seen that the two most familiar Gray codes, the BRGC and plain changes, can be interpreted using the lens of greedy algorithms in a relatively unequivocal way. Many other classic Gray codes have simple greedy algorithms as well. Here we mention two additional Gray codes for permutations.

In Zaks' order [70], the shortest possible prefix is reversed that gives a result that has not already been created. When discussing this order, each permutation is modeled as a stack of pancakes. For

Algorithm 4 Greedy algorithm for generating plain changes $\mathsf{plain}(n)$.

```
1:  procedure plain(n)                          ▷ Permutations are visited in plain(n) order
2:      π ← 12···n                              ▷ Starting permutation s = π ∈ S_n
3:      visit(π)                                ▷ Visit π for the first and only time
4:      S = {π}                                 ▷ Add π to the visited set
5:      T = {←t_n, →t_n, ←t_{n-1}, →t_{n-1}, ··· , ←t_2, →t_2}   ▷ Set order of operations T
6:      i ← 1                                   ▷ 1-based index into T; T[1] = ←t_n will swap n left
7:      while i ≤ 2n − 2 do                     ▷ Index i iterates through the 2n − 2 swaps in T
8:          π′ ← T[i](π)                        ▷ Apply the i^th highest priority swap to create π′
9:          if π′ ∉ S then                      ▷ Check if π′ is a new permutation
10:             π ← π′                          ▷ Update the current permutation π
11:             visit(π)                        ▷ Visit π for the first and only time
12:             S = S ∪ {π}                     ▷ Add π to the visited set
13:             i ← 1                           ▷ Reset the 1-based index into T
14:         else
15:             i ← i + 1                       ▷ If π′ ∈ S, then consider the next swap
```

example, if  represents 1234, then reversing the prefix of length three corresponds to flipping the top three pancakes to give  or $\overleftrightarrow{1234} = 3214$. Zaks defined the prefix-reversal Gray code using recursion in 1984, before it was reinterpreted greedily by Williams in 2013 [66]. Unknown to those authors at the time, Klügel had discovered the same order by 1796 [27]. The same greedy strategy also works for signed permutations (see Chapter 4). Here the model is a stack of 'burnt' pancakes (i.e., each pancake has a burnt side and an unburnt side) and the operations are sign-complementing prefix-reversals [53]. For example, the signed permutation $-1+2+3+4$ is modeled by  , and becomes $-3-2+1+4$  after a flip of length three. More broadly, the minimum length flip strategy works for colored permutations [6]; that paper also discusses the history of the permutation order.

Corbett's order [10] instead uses prefix-rotations, which can be understood as moving one symbol into the first position, or equivalently, as a prefix being rotated one position to the right. When discussing this order, each permutation is modeled as marbles on a ramp. For example, if  represents 1234, then a prefix-rotation of length four will cause the fourth marble to be picked up (allowing the other marbles to roll down) and moved to the top to give  or $\overleftarrow{1234} = 4123$ (where the arrow shows the movement of one value). This order was also defined recursively before being reinterpreted greedily. Here the greedy interpretation is somewhat subtle as the prefix-rotations are prioritized not by minimum length or maximum length, but instead by lengths $n, 2, n-1, 3, \ldots$. In other words, longest and shortest prefix-rotations are interleaved by extremity.

2.5 Failure of the Greedy Algorithm

When discussing Corbett's order in the previous section, the reader may have wondered why the Gray code wasn't defined to instead prioritize the prefix-rotations from shortest to longest, or longest to shortest. One compelling reason is that these prioritizations don't work! For example, suppose that we prioritize prefix-rotations of length $2, 3, \ldots, n$ which is equivalent to moving the leftmost

symbol into the first position that creates a new permutation. If we run the greedy Gray code algorithm starting from the identity permutation 1234 for $n = 4$, then the algorithm creates the following order.

$$\overleftarrow{1}234, \overleftarrow{2134}, \overleftarrow{3}214, \overleftarrow{2314}, \overleftarrow{4}231, \overleftarrow{2431}, \overleftarrow{3}241, 2341. \tag{2.2}$$

Note that we reach a stalemate at 2341: there is no symbol that can be moved into the first position. More specifically, $\overleftarrow{2}341 = 3241$, $\overleftarrow{234}1 = 4231$, and $\overleftarrow{2341} = 1234$ all appear earlier in the order. Therefore, this ruleset fails to generate a Gray code for $n = 4$ starting from 1234. Furthermore, it fails for any starting permutations when $n = 4$. This is because prefix-rotations do not consider the values in the current permutation, so the choice of the starting permutations does not affect whether the algorithm will be successful or not. More broadly, the minimum-length prefix-rotation ruleset fails to create Gray codes for larger n.

Remark 8. *By greedily moving the leftmost symbol of permutations of* $[n]$ *(*$n \geq 4$*) to the first position (also known as prefix-rotation), Algorithm 1 will halt prematurely and fails to generate a Gray code.*

We complete this section by discussing another failure result that is more closely related to the results of this thesis.

Remark 9. *By greedily swapping the rightmost pair of adjacent symbols of permutations of* $[n]$ *(*$n \geq 4$*), Algorithm 1 will halt prematurely and fails to generate a Gray code.*

For example, consider $n = 4$. If we start with 1234 and greedily swap the rightmost adjacent symbols, then the algorithm creates the following order.

$$\begin{aligned} &12\overleftrightarrow{34}, 1\overleftrightarrow{24}3, 14\overleftrightarrow{23}, 1\overleftrightarrow{43}2, 13\overleftrightarrow{42}, \overleftrightarrow{13}24, 31\overleftrightarrow{24}, 3\overleftrightarrow{14}2, 34\overleftrightarrow{12}, 3\overleftrightarrow{42}1, 32\overleftrightarrow{41}, \overleftrightarrow{32}14, \\ &23\overleftrightarrow{14}, 2\overleftrightarrow{34}1, 24\overleftrightarrow{31}, 2\overleftrightarrow{41}3, 21\overleftrightarrow{43}, 2134. \end{aligned} \tag{2.3}$$

Again the algorithm terminates before all $4! = 24$ permutations have been created. More specifically, our greedy ruleset fails to proceed from 2134, since every swap gives a permutation that already exists in the order.

Chapter 3

Loopless Change Sequence Algorithms

Although greedy algorithms for generating Gray codes are straightforward to understand and implement, they are also highly inefficient in terms of computational complexity. During a single iteration we may need to iterate through all k operations to create a new object, which results in an worst-case $\Omega(k)$-time per step. In addition, every past object needs to be stored in order to avoid transitions to previous objects. If each object has size n and the set has cardinality m, then this leads to $\Omega(nm)$-space, which is often exponential. Fortunately, greedy Gray codes can often be generated directly without storing previous objects.

This chapter provides loopless *history-free* algorithms for the binary reflected Gray code and plain changes. Both algorithms generate the *change sequence*, which is the sequence of changes to apply to create the Gray code. These change sequences turn out to be previously studied sequences known as ruler sequences. Furthermore, existing algorithms can generate each successive entry in these sequences in worst-case $\mathcal{O}(1)$-time. We also introduce the first general definition for signed ruler sequences, which will provide the basis for our new loopless Gray code algorithms for signed permutations in Chapter 5.

3.1 Binary and Decimal Ruler Sequences

The Online Encyclopedia of Integer Sequences (OEIS) [44] defines the *binary ruler sequence* as the "exponent of the highest power of 2 dividing n". The sequence is also known as the *binary carry sequence*, and it is indexed as entry A007814 in the OEIS. There is also an entry in the OEIS for the sequence with every entry incremented by one: A001511. The start of these sequences appear in Table 3.1 along with the corresponding values of n and thier largest power of two divisors.

The binary ruler sequence is closely related to both the lexicographic order of binary strings, and the binary reflected Gray code. More specifically, each entry of sequence provides the number of bits that changed in lexicographic order, as well as the individual bit index that changed in the

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
divisor	1	2	1	4	1	2	1	8	1	2	1	4	1	2	1
power	2^0	2^1	2^0	2^2	2^0	2^1	2^0	2^3	2^0	2^1	2^0	2^2	2^0	2^1	2^0
A007814	0	1	0	2	0	1	0	3	0	1	0	2	0	1	0
A001511	1	2	1	3	1	2	1	4	1	2	1	3	1	2	1

Table 3.1: The first fifteen entries of the binary ruler sequence A007814 and its incremented version A001511. Each entry is explained by the value of n's largest divisor that is a power of two.

BRGC. At first, the relationship between these three concepts may seem surprising. However, with a little bit of reflection[1] it follows quite naturally from the recursive definition the binary reflected Gray code in (1.2). More specifically, the reflection in this formula ensures that only one bit changes in Gray code. If we omit this reflection, then the formula instead generates lexicographic order, and the transition between the two sublists involves a rollover of all of the bits. These observations are illustrated in Table 3.2.

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A001511		1	2	1	3	1	2	1	4	1	2	1	3	1	2	1
lex	0000	0001	0010	0011	0100	0101	0110	0111	1000	1001	1010	1011	1100	1101	1110	1111
brgc	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000

Table 3.2: The relationship between the (incremented) binary ruler sequence A001511 and our two familiar orders of binary strings: lexicographic and binary reflected Gray code. Note that each entry in the sequence provides the number of bits that change in lexicographic order, and the specific bit that changes in BRGC order when the strings are indexed as $b_n b_{n-1} \cdots b_1$. For example, the middle entry of 4 in A001511 corresponds to the transition from 0111 to 1000 in lexicographic order (i.e., 4 bits changed) and from 0100 to 1100 in the BRGC (i.e., bit b_4 changed in $b_4 b_3 b_2 b_1$).

The relationship between A001511 and the binary reflected Gray code is particular useful when we want to efficiently generate the latter. This is because each successive entry in the sequence can be generated in worst-case $\mathcal{O}(1)$-time. Therefore, we can create a loopless algorithm for the BRGC by generating its change sequence and applying these changes. The details of how to do this are discussed in Section 3.4.

The binary ruler sequence can be understood as using bases $(2, 2, \ldots, 2)$. Similarly, the *decimal ruler sequence* uses bases $(10, 10, \ldots, 10)$, and it is from this sequence where ruler sequences get their name. More specifically, the (decimal) ruler sequence is named after the height of the tick marks on a typical one meter rulers, where there are ticks for millimeters (mm), centimeters (cm) and decimeters (dm). This is illustrated in Figure 3.1.

The next section considers ruler sequences with arbitrary bases. These sequences are fundamental for understanding and generating many types of Gray codes, including those introduced in Chapter 2. In particular, the *factorial bases* $(1, 2, \ldots, n)$ and $(n, n-1, \ldots, 1)$ are often associated with permutation Gray codes [18] [32].

[1] Pun intended!

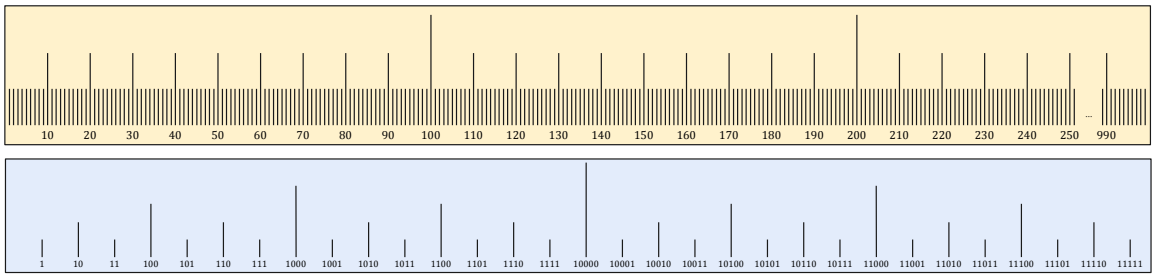


Figure 3.1: Illustrating rulers that given rise to the decimal ruler sequence $1, 1, 1, 1, 1, 1, 1, 1, 1, 2, \ldots$ (top) and binary ruler sequence $1, 2, 1, 3, 1, 2, 1, 4, \ldots$ (bottom).

3.2 Ruler Sequences

The *ruler sequence* with bases $b_n, b_{n-1}, \ldots, b_1$ can be recursively defined as following:

$$\mathsf{ruler}(b_1) = 1^{b_1-1} = \overbrace{1, 1, \ldots, 1}^{b_1-1 \text{ times}} \tag{3.1}$$

$$\mathsf{ruler}(b_n, b_{n-1}, \ldots, b_1) = (s, n)^{b_n-1}, s \text{ where } s = \mathsf{ruler}(b_{n-1}, b_{n-2}, \ldots, b_1) \tag{3.2}$$

Note that here commas denote the concatenation of join sequences, and exponents denotes times of repetitions. That is, x^y denotes y consecutive copies of x. There are b_n copies of the previous ruler subsequence s, and the sequence ends with s.

For example, $\mathsf{ruler}(4, 3)$ = 1,1,2,1,1,2,1,1,2,1,1 since $\mathsf{ruler}(3) = 1, 1$. The length of the ruler sequence is $(\prod_{i=1}^{n} b_i) - 1$. In other words, each base b_i causes the pattern to be repeated b_i times, with individual copies of i interspersed between them.

Ruler sequences are used to create *mixed-radix words* $W_{b_n,b_{n-1},\ldots,b_1}$, where $w_n \cdots w_2 w_1 \in W_{b_n,b_{n-1},\ldots,b_1}$ if $0 \leq w_i < b_i$ for $1 \leq i \leq n$. The number of these words is $|W_{b_n,b_{n-1},\ldots,b_1}| = \prod_{i=1}^{n} b_i = |\mathsf{ruler}(b_n, b_{n-1}, \ldots, b_1)| + 1$.

When mixed-radix words are written in lexicographic order, the ruler sequence is its *change sequence*. Each ruler sequence entry is the position that we perform "operations" on mixed radix words: they can be prefi rotations, prefix reversal, swapping, flipping, etc.

As previously mentioned, the *binary ruler sequence* $\mathsf{ruler}(2, 2, \ldots, 2)$ (OEIS A001511 [44]) gives the length of suffix that needs to be flipped when listing the binary numbers $011 \cdots 1$ to $100 \cdots 0$ in lexicographic order (or, in simpler terms, counting from smallest to largest). The case of binary ruler sequence with length 3 is shown below:

$$00\overline{0}, 0\overline{01}, 01\overline{0}, \overline{011}, 10\overline{0}, 1\overline{01}, 11\overline{0}, 111 \text{ since } \mathsf{ruler}(2, 2, 2) = 1, 2, 1, 3, 1, 2, 1$$

3.3 Signed Ruler Sequences

Here we introduce a general definition of a signed ruler sequence. Such sequences have previously been considered in special cases, including a signed version of the binary ruler sequence A164677. However, we are unaware of any previous general definition.

Our *signed ruler sequence* (ruler±) is a variation of ruler sequence. More specifically, it is equal to the ruler sequence for the same base, but some of the entries are negative. We define it in a similar way as ruler sequence except that every second copy of the previous ruler subsequence is both reversed and sign complemented. When considering this definition, it is important to note that standard ruler sequences are palindromes, so reversing various subsequences does not change them. The placement of negative signs in our signed sequences will prevent signed ruler sequences from being palindromes, so the reversal becomes relevant. (This is also a potential reason why these sequences have not formally been defined or popularized in the literature.) In the following formal definition, negative number $-x$ will be denoted as $\overline{x}$ for notational simplicity, and reversal of a sequence (or subsequence) s will be denoted as s^R.

$$\mathsf{ruler}\pm(b_1) = 1^{b_1-1} = \overbrace{1,1,\ldots,1}^{b_1-1 \text{ times}} \tag{3.3}$$

$$\mathsf{ruler}\pm(b_n,b_{n-1},\ldots,b_1) = \begin{cases} (s, n, \overline{s}^R, n)^{(b_n-1)/2}, s & \text{if } b_n \text{ is odd} \\ (s, n, \overline{s}^R, n)^{(b_n-2)/2}, s, n, \overline{s} & \text{if } b_n \text{ is even} \end{cases} \tag{3.4}$$

It is notable that the subsequence s is repeated b_n times in the $n > 1$ case, similar as in ruler sequences. However, every second subsequence is reversed and complemented. For example,

$$\begin{aligned} \mathsf{ruler}\pm(2,3) &= 1, 2, \text{-}1, 2, 1 \\ \mathsf{ruler}\pm(2,3,4) &= 1, 2, \text{-}1, 2, 1, 4, \underline{\text{-}1, \text{-}2, 1, \text{-}2, \text{-}1}, 4, 1, 2, \text{-}1, 2, 1, 4, \underline{\text{-}1, \text{-}2, 1, \text{-}2, \text{-}1} \end{aligned} \tag{3.5}$$

where the underline subsequences in $\mathsf{ruler}\pm(2,3,4)$ are complemented and reversed versions of $\mathsf{ruler}(2,3)$ in (3.5). The length of the signed ruler sequence is the same as the unsigned ruler sequence, $(\prod_{i=1}^{n} b_i) - 1$.

Signed ruler sequences provide more information for some Gray code changes. For example, the signed binary ruler sequence entries provide not just the index i of the bit that changes, but also whether the bit b_i changes from 0 to 1 (entry $+i$) or from 1 to 0 (entry $-i$). Similarly, the signed downstairs factorial ruler sequence provides the symbol to swap and the direction to swap it in when generating plain changes. These details are discussed in the next section.

3.4 Loopless BRGC and Plain Changes

This section shows how we can generate BRGC and plain changes via ruler sequence in a *loopless* way. When creating BRGC, we will use the binary ruler sequence $\mathsf{ruler}(2,2,\ldots,2)$ (n copies). In this case, each ruler sequence entry is the position (counting from least to most significant digit) that is flipped to create the next word. For instance, we can use $\mathsf{ruler}(2,2,2) = 1,2,1,3,1,2,1$ to generate BRGC for $n = 3$:

$$000, 001, 011, 010, 110, 111, 101, 100. \tag{3.6}$$

Furthermore, if we use the signed version $\mathsf{ruler}\pm(2,2,2) = 1, 2, \text{-}1, 3, 1, \text{-}2, \text{-}1$ then the signs tell us the direction of each bit change. By computing each entry of the (signed) ruler sequence in worst-case $\mathcal{O}(1)$-time, we can update the associated binary string in the binary reflected Gray code in worst-case $\mathcal{O}(1)$-time, thereby completing a loopless algorithm.

For plain change order, we will use the signed ruler sequence $\mathsf{ruler}\pm(n, n-1, ..., 2, 1)$ (also referred as the *downstairs ruler sequence*). Here the absolute value of each ruler sequence entry is the digit to be swapped, and the sign indicates which direction to swap ($+$ is to the left and $-$ is to the right). To illustrate, for $n = 3$ we use $\mathsf{ruler}\pm(3, 2, 1) = +3, +3, +2, -3, -3$:

$$12\overset{\leftarrow}{3}, 1\overset{\leftarrow}{3}2, 31\overset{\leftarrow}{2}, \overset{\rightarrow}{3}21, 2\overset{\rightarrow}{3}1, 213. \tag{3.7}$$

By computing each entry of $\mathsf{ruler}\pm(n, n-1, ..., 2, 1)$ in worst-case $\mathcal{O}(1)$-time, we can update the associated permutation in plain changes in worst-case $\mathcal{O}(1)$-time, thereby completing a loopless algorithm. Note that in this case we must also store and update the inverse of the current permutation, since otherwise we cannot identify the location of the particular symbol to swap in worst-case $\mathcal{O}(1)$-time.

Algorithm 5 Generating BRGC and plain changes using ruler sequences. BRGC uses the ruler sequence $\mathsf{ruler}(2, 2, \ldots, 2)$ (n copies), while plain change order uses the signed ruler sequence $\mathsf{ruler}\pm(n, n-1, ..., 2, 1)$. Focus pointers are stored in $\mathbf{f}$. $\mathbf{d}$ resembles the directions of change. $\mathbf{flip}(j)$ resembles flipping the j-th symbol in the binary string. $\mathbf{swap}(j)$ resembles swapping j to the left when $j > 0$, and swapping j to the right when $j < 0$. The overall algorithm is loopless if each function runs in worst-case $\mathcal{O}(1)$-time.

```
1:  procedure LooplessBRGC(s)
2:      a_1 a_2 ··· a_n     ← 0 0 ··· 0
3:      f_1 f_2 ··· f_{n+1} ← 1 2 ··· n+1
4:
5:      visit(s)
6:      while f_1 ≤ n do
7:          j ← f_1
8:          f_1 ← 1
9:          a_j ← a_j + 1
10:         s ← flip[j](s)
11:         visit(j, s)
12:         if a_j = 1 then
13:             a_j ← 0
14:             f_j ← f_{j+1}
15:             f_{j+1} ← j + 1
```

```
1:  procedure LooplessPlainChange(s)
2:      a_1 a_2 ··· a_n     ← 0 0 ··· 0
3:      f_1 f_2 ··· f_{n+1} ← 1 2 ··· n+1
4:      d_1 d_2 ··· d_n     ← 1 1 ··· 1
5:      visit(s)
6:      while f_1 ≤ n do
7:          j ← f_1
8:          f_1 ← 1
9:          a_j ← a_j + d_j
10:         s ← swap[d_j · j](s)
11:         visit(d_j · j, s)
12:         if a_j ∈ {0, j − 1} then
13:             d_j ← −d_j
14:             f_j ← f_{j+1}
15:             f_{j+1} ← j + 1
```

Chapter 4

Greedy Gray Codes for Signed Permutations

In this chapter, we attempt to create Gray codes for signed permutations using the greedy Gray code algorithm from Chapter 2. In that chapter, we saw that Gray codes of permutations are often inspired by different physical models of the permutation. More specifically, certain operations are more natural when considering a specific physical model. For example, prefix-reversals are natural when visualizing stacks of pancakes  , while prefix-rotations are natural when visualizing marbles on a ramp  . With this in mind, we start by selecting a physical model for signed permutations: two-sided ribbons. This model suggests the operations that we will consider for our Gray codes. The operations include the types of twists and swaps that someone would perform while braiding.

We experimented with 20 different rulesets. Each ruleset uses two types of operations selected from flips (1-twists), twists (2-twists), and swaps (unsigned 2-twists). Empirically, we found that a dozen of our rulesets worked up to $n = 8$. We analyzed each order and identified three distinct types of correctness proofs.

Aesthetically, we found Experiment 8 to be particularly pleasing both visually and mathematically. The start of the $n = 4$ order is visualized in Figure 4.1. The familiar zig-zag pattern from in plain changes (see Figure 2.2) is prominent, except that the symbols are complemented (i.e., the ribbons are turned over) along each pass. We named the order *twisted plain changes* in [47].

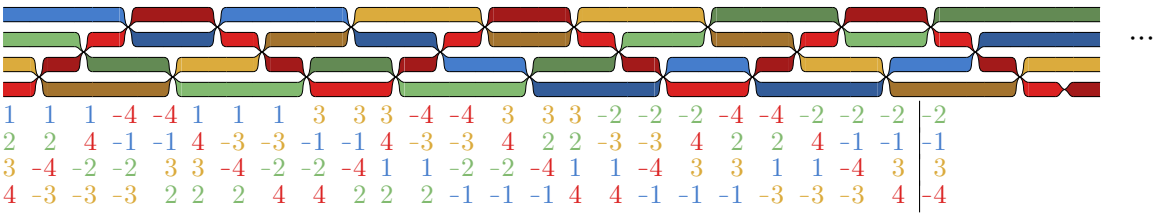
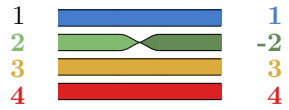


Figure 4.1: Twisted plain changes twisted(4) (experiment 8) up to its 25th entry.

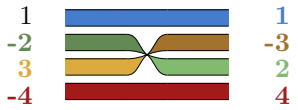
4.1 Two-Sided Ribbons: Twists and Swaps

Now we introduce our physical model for signed permutations. A *two-sided ribbon* appears to be polished on one side and matte on the other side[1] and we model a n-bit signed permutation using n two-sided ribbons in parallel.

Given this physical model, a basic operation that can be performed is a twist. More specifically, a *k-twist* turns over k neighboring ribbons and reverses their order, as visualized in Figure 4.2 for $k = 1, 2$. 1-twists are also referred as *flips* as only one ribbon is turned over. A twist performs a *complementing substring reversal*, or simply a *reversal* [24], on the signed permutation.



(a) The 1-twist changes $1\,2\,3\,4$ into $1\,\overline{2}\,3\,4$.



(b) The 2-twist changes $1\,\overline{2}\,3\,\overline{4}$ into $1\,\overline{3}\,2\,\overline{4}$.

Figure 4.2: Two-sided ribbons with distinct positive (i.e., glossy) and negative (i.e., matte) sides running in parallel. A k-twist reverses the order of k neighboring ribbons and turns each of them over. This is shown in (a) for $k = 1$ (which is also known as a flip), and in (b) for $k = 2$. Note that swap is an unsigned 2-twist, meaning that the relative order of two neighboring ribbons are reversed, but neither of the ribbons are turned over.

4.1.1 Twist Properties

Now we consider some basic properties of twists on signed permutations. These properties will be useful when we consider our greedy experiments.

Recall that $S_n^{\pm}$ is the set of signed permutations over $[n]$ (see Section 1.3). A k-twist is said to have *even length* or *odd length* depending on the parity of k. The following remark states that the number of negative symbols in a signed permutation changes by an even amount when applying an even-length twist. In general, this is due to the fact each symbol in the twist switches from positive to negative or vice versa, and an even number of ± 1 values is even. This point will be useful when we consider the types of twists that can be used in a twist Gray code in Section 4.2.

Remark 10. *Suppose that $\alpha \in S_n^{\pm}$ and $\beta \in S_n^{\pm}$ differ by a k-twist for some even k. Then the number of negative symbols in α and β have the same parity.*

The following remark will also be useful later in this chapter.

Remark 11. *Two signed permutations $\pi, \pi' \in S_n^{\pm}$ differ by a k-twist starting at index i, if and only if, their negations $-\pi, -\pi' \in S_n^{\pm}$ differ by a k-twist starting at index i.*

Another basic operation on two-sided ribbons is a swap. A *swap* exchanges two neighboring ribbons (without turning either ribbon over). Note that this operation is exactly the same as swap in unsigned permutations (in contrast, twists are more or less exclusive for signed permutations).

[1] This type of ribbon is referred as *single face* as only one side is polished.

4.2 Restrictions on Operations

Before considering our greedy experiments, it is helpful to rule out certain combinations of operations that cannot create a Gray code. For example, note that 2-twists and swaps do not change the parity of the number of negative symbols in a signed permutation. More specifically, swaps do not change the sign of any symbol, and 2-twists change the sign of two symbols, thus changing the number of negative symbols by -2 or 0 or $+2$. For this reason, it is impossible to create a Gray code for $S_n^{\pm}$ using only swaps and 2-twists, and for this reason we will not consider experiments using pairs of these operations. The following lemma provides the same statement for even-length twists.

Lemma 1. *If L is a twist Gray code of $S_n^{\pm}$ for some $n \geq 1$, then at least one pair of consecutive signed permutations in L differ by a k-twist for some odd k.*

Proof. Note that $\alpha = 123\cdots n \in S_n^{\pm}$ cannot be transformed into $\beta = \overline{1}23\cdots n \in S_n^{\pm}$ via any sequences of even-length twists. This is because α has an even number of negative symbols, β has an odd number of negative symbols, and by Remark 10 every even-length twist does not change the parity of the number negative symbols. □

However, as the next section 4.3 indicates, 1-twists and 2-twists are sufficient when used together, and there exist other combinations of two operations that are capable of generating signed permutations.

4.3 Experiments with 1-Twists, 2-Twists and Swaps

Although any k-twist are potential candidates of components of greedy algorithm for generating twist Gray codes, we only consider 1-twists, 2-twists and swaps in our experiments on searching for such greedy algorithms.

We list our experiments in Table 4.1. Each experiment is composed of 2 greedy rules, and the rule order indicates how the two rules are prioritized. For example, in experiment 1, we attempt to 1-twist the largest digit (absolute value) in the signed permutation. If this generate a permutation we have already generated, then we 1-twist the second largest digit. If all 1-twists fail to create a new signed permutation, we swap the largest digit with its neighboring digit. By default we prioritize swapping to the left over to the right; nevertheless, if the target digit is on the rightmost position, we can only swap to the left; similarly if the target digit is on the leftmost position we can only swap to the right. For another example, in experiment 4, we attempt to swap the largest digit with its neighboring digit. If that is not possible, we attempt to 1-twist the largest digit. Then we attempt to swap the second largest digit (thus, in a "zig-zag" prioritization).

From our experiment results we notice that all experiments involving swapping or 2-twisting the rightmost symbol fail to generate a proper Gray code of signed permutations. This is due to the fact that "swapping the rightmost" cannot generate a Gray code under the context of simple permutations (recall 9). In contrast, we observe that both flipping the rightmost symbol or flipping the largest symbol can create algorithms that generate Gray codes of signed permutations.

Experiment No.	Type 1	Type 2	Type order
1	1-twist largest	swap largest	after
2	swap largest	1-twist largest	after
3	1-twist largest	swap largest	zigzag
4	swap largest	1-twist largest	zigzag
5	1-twist rightmost	swap largest	after
6	swap largest	1-twist rightmost	after
7	1-twist largest	2-twist largest	after
8	2-twist largest	1-twist largest	after
9	1-twist rightmost	2-twist largest	after
10	2-twist largest	1-twist rightmost	after
11	1-twist largest	2-twist largest	zigzag
12	2-twist largest	1-twist largest	zigzag
13	1-twist largest	swap rightmost	after
14	swap rightmost	1-twist largest	after
15	1-twist largest	swap rightmost	zigzag
16	swap rightmost	1-twist largest	zigzag
17	1-twist largest	2-twist rightmost	after
18	2-twist rightmost	1-twist largest	after
19	1-twist largest	2-twist rightmost	zigzag
20	2-twist rightmost	1-twist largest	zigzag

Table 4.1: Experiments on greedy algorithms for Gray codes of signed permutations. For example, Experiment 1 prioritizes 1-twisting the values $n, n-1, \ldots, 1$ and if none of these operations succeed, then it prioritizes swapping the values $n, n-1, \ldots, 1$. On the other hand, Experiment 3 prioritizes 1-twisting value n, then swapping value n, then 1-twisting value $n-1$, then swapping value $n-1$, and so on in a zigzag or interlaced fashion.

We tested these 20 experiments on $n = 3, 4, 5, 6, 7, 8$. Experiments 1-12 successfully yielded all signed permutations of the corresponding length without repetitions. Experiments 13-20 terminated prematurely (i.e. did not generate all signed permutations). It is notable that experiments 13-20 all contain swaps or 2-twists on the rightmost digit. We conjecture that this type of rule will cause the greedy algorithm to halt prematurely, just as rightmost swaps were shown to fail for permutations in Section 2.5.

4.4 Selected Proofs

When analyzing the data from our 12 successful experiments, we found that there were three distinct types of patterns that arose.

- When prioritizing 1-twists before other operations, we found that the Gray codes partitioned into blocks of length 2^n. Each block has the same underlying permutation, and the signs are complemented in all possible ways. In other words, the blocks traverse a subgraph in the flip graph that is isomorphic to an n-dimensional cube (see Figure 1.4) but with each vertex being prescribed the same permutation. We illustrate this type of pattern by considering Experiment 7.

- When prioritizing 2-twists or swaps before other operations, we found that the Gray codes partitioned into blocks of length $n!$. Each block traverses a subgraph in the flip graph that is isomorphic to an n-dimensional permutohedron (see Figure 1.7). However, when 2-twists are used, the signs of each permutation are perturbed in a manner that requires some consideration. We illustrate this type of pattern by considering Experiment 8.

- When prioritizing operations in a zigzag or interlaced fashion, we found that the Gray codes partitioned into blocks of length n or $2n$. Each block involves zigging or zagging or zigzagging the value $\pm n$ through successive signed permutations (or a modified versions of them) in the order for $S^{\pm}_{n-1}$. In other words, the blocks utilize a type of local recursion found in the definition of plain changes found in Section 1.1.3. We illustrate this type of pattern by considering Experiment 11.

Now we provide sample proofs and illustrations for the validity of Experiments 7, 8 and 11.

Algorithm 6 Greedy algorithm for generating twisted plain changes $\mathsf{twisted}(n, T)$.

```
1:  procedure Twisted(n, T)          ▷ Signed permutations are visited in twisted(n) order
2:      π ← +1 +2 ··· +n             ▷ Starting signed permutation s = π ∈ S_n^±
3:      visit(π)                     ▷ Visit π for the first and only time
4:      S = {π}                      ▷ Add π to the visited set
5:      i ← 1                        ▷ 1-based index into T; T[1] = ←t_n will 2-twist n left
6:      while i ≤ 3n − 2 do          ▷ Index i iterates through the 3n − 2 twists in T
7:          π′ ← T[i](π)             ▷ Apply the i^th highest priority twist to create π′
8:          if π′ ∉ S then           ▷ Check if π′ is a new signed permutation
9:              π ← π′               ▷ Update the current signed permutation π
10:             visit(π)             ▷ Visit π for the first and only time
11:             S = S ∪ {π}          ▷ Add π to the visited set
12:             i ← 1                ▷ Reset the 1-based index into T
13:         else
14:             i ← i + 1            ▷ If π′ ∈ S, then consider the next twist
```

Theorem 1. *Experiment 7 visits a twist Gray code of signed permutations. That is,* $\mathsf{twisted}(n, T)$ *where* $T = \mathsf{t}_n, \ldots, \mathsf{t}_2, \mathsf{t}_1, \overleftarrow{\mathsf{t}_n}, \overrightarrow{\mathsf{t}_n}, \overleftarrow{\mathsf{t}_{n-1}}, \overrightarrow{\mathsf{t}_{n-1}}, \ldots, \overleftarrow{\mathsf{t}_2}, \overrightarrow{\mathsf{t}_2}$ *orders* $S^{\pm}_n$.

Proof. Since 1-twists are prioritized before 2-twists, the algorithm proceeds in the manner as BRGC Gray codes (with negative signs acting as the role of 1's in BRGC) and preserves the absolute value of numbers at each position of the permutation until a 2-twist changes them. Consequently, it generates sequences of 2^n signed permutations using 1-twists until a single 2-twist is required. It is notable that per the patterns of BRGC, in each block of 2^n signed permutation, the last permutation always differs from the first by changing the sign of 1. We should emphasize that since our prioritization is based on *values* rather than positions, this BRGC structure will flip 1 rather than the value of the first position.

As a result, we are able to traverse through all permutations resulting from a given order of absolute values of 1 to n. We also know that we are able to reach all order of absolute values through 2-twists from plain change order, which are just swaps when only acting on absolute values.

Hence the algorithm traverses through all $2^n \cdot n!$ signed permutations. More specifically, the order generated by the algorithm appears in Figure 4.4.

□

Theorem 2. *Experiment 8 visits a twist Gray code of signed permutations. That is,* $\mathsf{twisted}(n, T)$ *where* $T = \overleftarrow{\mathsf{t}_n}, \overrightarrow{\mathsf{t}_n}, \overleftarrow{\mathsf{t}_{n-1}}, \overrightarrow{\mathsf{t}_{n-1}}, \ldots, \overleftarrow{\mathsf{t}_2}, \overrightarrow{\mathsf{t}_2}, \mathsf{t}_n, \ldots, \mathsf{t}_2, \mathsf{t}_1$ *orders* $S_n^{\pm}$.

Proof. Since 2-twists are prioritized before 1-twists, the algorithm proceeds in the same manner as plain changes, except for the signs of the visited objects. As a result, it generates sequences of $n!$ signed permutations using 2-twists until a single 1-twist is required. One caveat is that the first signed permutation in a sequence alternates between having the underlying permutation of $1234\cdots n$ or $2134\cdots n$. This is due to the fact that plain changes starts at $1234\cdots n$ and ends at $2134\cdots n$ and swaps 12 to 21 one time. As a result, 12 will be inverted while traversing every second sequence of length $n!$, and these traversals will be done in reflected plain changes order by Remark 7. The order generated by the algorithm is illustrated in in Figure 4.5. □

Note that the proof of Theorem 2 can be largely used for proving the validity of other greedy algorithms that prioritize swaps or 2-twists over 1-twists via some minor changes, including Experiment 2, 6, 8, 10. These algorithms execute in the sequence of plain changes when 2-twists or swaps are used until 1-twists are required. Hence they all appear to be a traversal through 2^n plain changes blocks.

Finally we attempt to prove the validity of algorithms that involve zig-zagging of 2-twists/swaps and 1-twists. Such algorithms are much harder to understand compared with the rest, which either traverse through plain change blocks or BRGC blocks.

Theorem 3. *Experiment 11 visits a twist Gray code of signed permutations. That is,* $\mathsf{twisted}(n, T)$ *where* $T = \mathsf{t}_n, \overleftarrow{\mathsf{t}_n}, \overrightarrow{\mathsf{t}_n}, \mathsf{t}_{n-1}, \overleftarrow{\mathsf{t}_{n-1}}, \overrightarrow{\mathsf{t}_{n-1}}, \ldots, \mathsf{t}_2, \overleftarrow{\mathsf{t}_2}, \overrightarrow{\mathsf{t}_2}, \mathsf{t}_1$ *orders* $S_n^{\pm}$.

Definition 2. *If* $\pi = p_1 p_2 \cdots p_{n-1} \in S_{n-1}^{\pm}$ *and* $m \in \{-n, n\}$*, then each of the following lists contains* $2n$ *distinct signed permutations in* $S_n^{\pm}$.

$$
\begin{aligned}
\pm\mathsf{zig}(\pi, m) = {} & p_1\, p_2 \cdots p_{n-2}\, p_{n-1} \quad m, \\
& p_1\, p_2 \cdots p_{n-2}\, p_{n-1} \quad \overline{m}, \\
& p_1\, p_2 \cdots p_{n-2} \quad m \quad \overline{p_{n-1}}, \\
& p_1\, p_2 \cdots p_{n-2} \quad \overline{m} \quad \overline{p_{n-1}}, \\
& \vdots \quad \vdots \quad \vdots \quad \vdots \quad \vdots \\
& p_1\, m\, \overline{p_2} \quad \cdots \quad \overline{p_{n-2}}\, \overline{p_{n-1}}, \\
& p_1\, \overline{m}\, \overline{p_2} \quad \cdots \quad \overline{p_{n-2}}\, \overline{p_{n-1}}, \\
& m\, \overline{p_1}\, \overline{p_2} \quad \cdots \quad \overline{p_{n-2}}\, \overline{p_{n-1}}, \\
& \overline{m}\, \overline{p_1}\, \overline{p_2} \quad \cdots \quad \overline{p_{n-2}}\, \overline{p_{n-1}}
\end{aligned}
$$

$$\begin{aligned}
\pm\mathsf{zag}(m,\pi) = {} & m\,p_1\,p_2 \;\cdots\; p_{n-2}\,p_{n-1},\\
& \overline{m}\,p_1\,p_2 \;\cdots\; p_{n-2}\,p_{n-1},\\
& \overline{p_1}\,m\,p_2 \;\cdots\; p_{n-2}\,p_{n-1},\\
& \overline{p_1}\,\overline{m}\,p_2 \;\cdots\; p_{n-2}\,p_{n-1},\\
& \vdots\;\vdots\;\vdots\;\;\vdots\;\;\;\vdots\\
& \overline{p_1}\,\overline{p_2}\cdots\overline{p_{n-2}}\;\;m\;\;p_{n-1},\\
& \overline{p_1}\,\overline{p_2}\cdots\overline{p_{n-2}}\;\;\overline{m}\;\;p_{n-1},\\
& \overline{p_1}\,\overline{p_2}\cdots\overline{p_{n-2}}\,\overline{p_{n-1}}\;\;m,\\
& \overline{p_1}\,\overline{p_2}\cdots\overline{p_{n-2}}\,\overline{p_{n-1}}\;\;\overline{m}
\end{aligned}$$

Remark 12. *Let $\pi \in S_{n-1}^{\pm}$ be a signed permutation over $[n-1]$. Then*

$$\pm\mathsf{zig}(\pi,n) = \mathsf{reflect}(\pm\mathsf{zag}(-n,-\pi)) \tag{4.1}$$

In other words, zigging n through a signed permutation gives the reflected list of zagging $-n$ through the negative version of the signed permutation. In particular, the two lists contain the same signed permutations...

Lemma 2. *The set*

$$A = \{\pm\mathsf{zig}(\pi,n) \mid \pi \in S_{n-1}^{\pm}\} \tag{4.2}$$

exactly covers all permutations in $S_n^{\pm}$. In other words, zigging n through each signed permutation of $[n-1]$ gives every signed permutation of $[n]$ exactly once.

Proof. Note that each signed permutation $\pi \in S_{n-1}^{\pm}$ corresponds to $2n$ different signed permutations in $\pm\mathsf{zig}(\pi,n)$. Furthermore, we see that all those permutations are mutually different: all signed permutations in $\pm\mathsf{zig}(\pi,n)$ keep the relative order of numbers in π, so $\pm\mathsf{zig}(\pi,n)$ and $\pm\mathsf{zig}(\epsilon,n)$ have no common elements if $\pi \neq \epsilon$. Hence $|A| = 2n \cdot 2^{n-1}(n-1)! = 2^n \cdot n! = |S_n^{\pm}|$, and $A = S_n^{\pm}$. □

Definition 3. *The $\pm$-zig-zag order $\pm\mathsf{ZigZag}(n)$ is inductively as*

$$\begin{aligned}
\pm\mathsf{ZigZag}(n) = {} & \pm\mathsf{zig}(\pi_1,n), \pm\mathsf{zag}(-n,-\pi_2),\\
& \pm\,\mathsf{zig}(\pi_3,n), \pm\mathsf{zag}(-n,-\pi_4),\\
& \ldots,\\
& \pm\,\mathsf{zig}(\pi_{m-1},n), \pm\mathsf{zag}(-n,-\pi_m)
\end{aligned} \tag{4.3}$$

where $\pm\mathsf{ZigZag}(n-1) = \pi_1, \pi_2, \ldots, \pi_m$ with base case $\pm\mathsf{ZigZag}(1) = 1, \overline{1}$.

Theorem 4. *$\pm\mathsf{ZigZag}(n)$ is a k-twist Gray code of $S_n^{\pm}$ for $k \in \{1,2\}$.*

Proof. First note that $\pm\mathsf{ZigZag}(n)$ contains the correct number of strings. This is because $\pm\mathsf{ZigZag}(n)$ consists of one sublist per string in $\pm\mathsf{ZigZag}(n-1)$ by Definition 3, and each sublist has length $2n$ by Definition 2. Thus, the lengths of these lists follow the recurrence

$$|\pm\,\mathsf{ZigZag}(n)| = 2n \cdot |\pm\,\mathsf{ZigZag}(n-1)| \text{ with } |\pm\,\mathsf{ZigZag}(1)| = 2 \tag{4.4}$$

so $|\pm\,\mathsf{ZigZag}(n)| = 2^n n! = |S_n^{\pm}|$ as desired. Furthermore, no signed permutation appears twice in $\pm\mathsf{ZigZag}(n)$. This can be seen by applying Remark 12 and reflecting every second sublist. More specifically, we can rewrite the contents of $\pm\mathsf{ZigZag}(n)$ as follows

$$\begin{aligned}
&\pm\,\mathsf{zig}(\pi_1, n), \pm\mathsf{zig}(\pi_2, n),\\
&\pm\,\mathsf{zig}(\pi_3, n), \pm\mathsf{zig}(\pi_4, n),\\
&\ldots,\\
&\pm\,\mathsf{zig}(\pi_{m-1}, n), \pm\mathsf{zig}(\pi_m, n)
\end{aligned}$$

since each sublist $\pm\mathsf{zag}(-n, -\pi_{2k})$ contains the same set of strings as $\pm\mathsf{zig}(\pi_{2k}, n)$. As there are no duplicates in the above list, there are no duplicates in $\pm\mathsf{ZigZag}(n)$.

Now we prove that successive signed permutations in $\pm\mathsf{ZigZag}(n)$ differ by a 1-twist or a 2-twist via induction on n. When $n = 1$, we have $\pm\mathsf{ZigZag}(1) = 1, \overline{1}$ so the only pair of successive signed permutations differ by a 1-twist. Suppose that successive signed permutations in $\pm\mathsf{ZigZag}(n)$ differ by a 1-twist or a 2-twist for some $n = k$. Now consider $\pm\mathsf{ZigZag}(k+1)$. Note that each successive signed permutations within an individual sublist of the form $\pm\mathsf{zig}(\pi_i, n)$ differ by a 2-twist or 1-twist involving n or $-n$, and this is true for sublists of the form $\pm\mathsf{zag}(-n, -\pi_i)$ as well. Therefore, we need only consider successive permutations spanning two such sublists. There are two cases to consider.

- When i is odd, we need to consider the transition from $\pm\mathsf{zig}(\pi_i, n)$ to $\pm\mathsf{zag}(-n, -\pi_{i+1})$. The last signed permutation in $\pm\mathsf{zig}(\pi_i, n)$ is $-n - \pi_i$ and the first signed permutation in $\pm\mathsf{zag}(-n, -\pi_{i+1})$ is $-n - \pi_{i+1}$. By induction, π_i and π_{i+1} differ by a 1-twist or 2-twist, so $-\pi_i$ and $-\pi_{i+1}$ also differ by a 1-twist or 2-twist by Remark 11, and hence, so too do $-n - \pi_i$ and $-n - \pi_{i+1}$.

- When i is even, we need to consider the transition from $\pm\mathsf{zag}(-n, -\pi_i)$ to $\pm\mathsf{zig}(\pi_{i+1}, n)$. The last signed permutation in $\pm\mathsf{zag}(-n, -\pi_i)$ is $\pi_i n$ and the first signed permutation in $\pm\mathsf{zig}(\pi_{i+1}, n)$ is $\pi_{i+1} n$. By induction, π_i and π_{i+1} differ by a 1-twist or 2-twist, so too do $\pi_i n$ and $\pi_{i+1} n$.

□

Now we show that the Gray code is cyclic.

Remark 13. $\mathsf{first}(\pm\mathsf{ZigZag}(n)) = 123\cdots n$ *and* $\mathsf{last}(\pm\mathsf{ZigZag}(n)) = \overline{1}23\cdots n$.

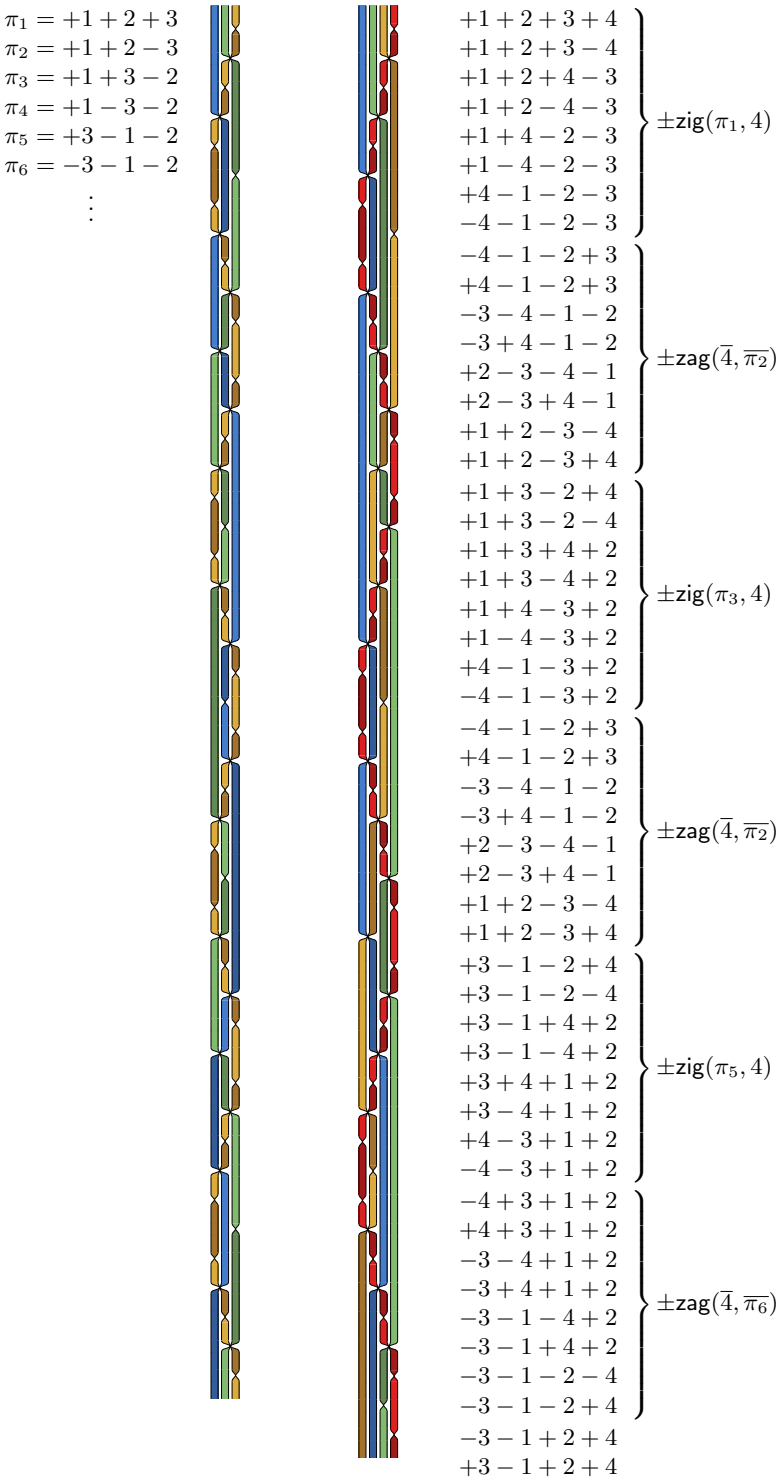


Figure 4.3: Illustrating the zig-zag pattern that extends experiment 11 from $n = 3$ to $n = 4$. (The total length of the $n = 4$ order is $n! \cdot 2^n = 24 \cdot 16 = 384$.)

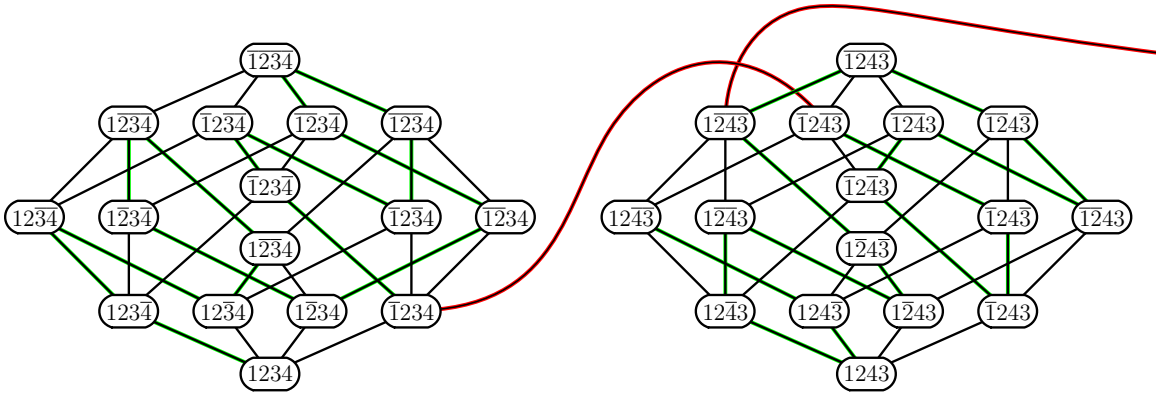


Figure 4.4: The start of the order for Experiment 7 when $n = 4$ is $1234, 123\overline{4}, \ldots$.

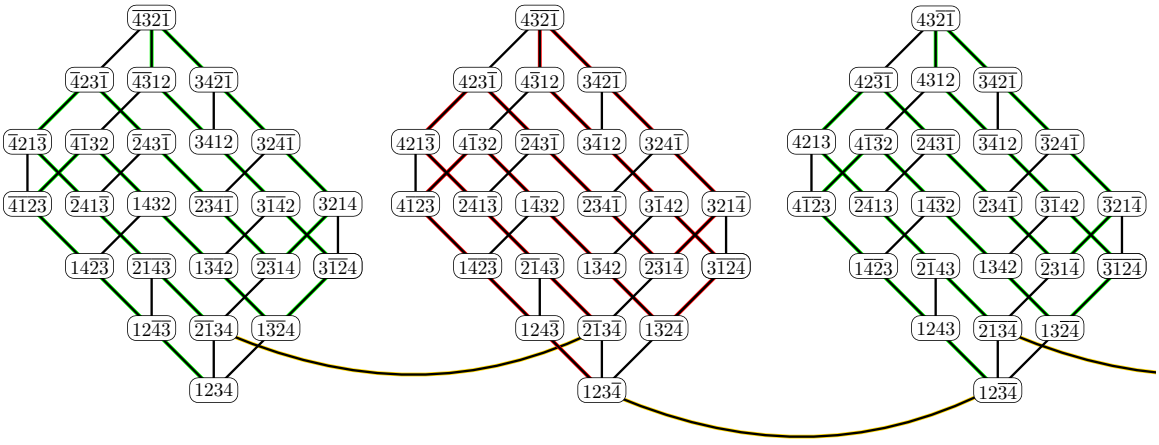


Figure 4.5: The start of the order for Experiment 8 when $n = 4$ is $1234, 12\overline{43}, \ldots$.

Chapter 5

Loopless Change Sequence Algorithms for Signed Permutations

While our greedy algorithms for creating signed permutation Gray codes in Section 4.3 have simple descriptions, these descriptions do not translate directly into efficient algorithms. This is due to the fact that the greedy Gray code algorithm explicitly stores its *full history* (i.e., every previously generated object) in order to decide which object to generate next. Fortunately, Gray codes obtained with the greedy algorithm can often be generated directly using *history-free* implementations that do not need to store the previous objects. Previous examples of this include prefix-reversal (or 'flip') Gray codes for permutations [66, 16][1], signed permutations [53, 54], and colored permutations [7, 6], as well as basis-exchange Gray codes for matroids [39] which will be discussed in Chapter 6.

In this chapter we provide loopless history-free implementations for all 12 of our signed permutation Gray codes from Chapter 4. In each case, we will see that the Gray code's change sequence can be described using signed ruler sequences that were introduced in Section 3.3. The loopless implementations are then based on generating each successive entry in the associated signed ruler sequence in worst-case $\mathcal{O}(1)$-time.

5.1 Signed Change Sequences for Experiments

In light of the inefficiency of greedy algorithms for generating Gray codes for signed permutations, we seek other types of algorithms that are able to generate them without using exponential space and $\mathcal{O}(n)$ time for each object. Luckily, greedy Gray codes can often be generated without remembering previous objects. We utilize signed ruler sequences as introduced in Section 3.3 to *looplessly* generate the Gray codes. We should note there are also loopless algorithms for non-greedy Gray codes with ruler sequences [28] [18].

[1] [16] was published under pseudonyms inspired by *Harry Dweighter* ("harried waiter") [31].

We experimented on the 12 algorithms proposed in Section 4.3 that are generated by signed ruler sequences instead. The signed ruler sequences utilized in each algorithm is shown in Table 5.1.

Experiment No.	Ruler Sequence Bases
1	1, 2, ..., n-1, n, 2, 2, ..., 2, 2
2	2, 2, ..., 2, 2, 1, 2, ..., n-1, n
3	1, 2, 2, 2, 3, 2, ..., n-1, 2, n, 2
4	2, 1, 2, 2, 2, 3, ..., 2, n-1, 2, n
5	1, 2, ..., n-1, n, 2, 2, ..., 2, 2
6	2, 2, ..., 2, 2, 1, 2, ..., n-1, n
7	1, 2, ..., n-1, n, 2, 2, ..., 2, 2
8	2, 2, ..., 2, 2, 1, 2, ..., n-1, n
9	1, 2, ..., n-1, n, 2, 2, ..., 2, 2
10	2, 2, ..., 2, 2, 1, 2, ..., n-1, n
11	1, 2, 2, 2, 3, 2, ..., n-1, 2, n, 2
12	2, 1, 2, 2, 2, 3, ..., 2, n-1, 2, n

Table 5.1: Experiments on greedy algorithms and the bases of their corresponding signed ruler sequences. (Note that the bases are listed in reverse in our programs in the appendix.)

We implement the changes incurring in the signed permutation sequence in the following way:

Lemma 3. *The change sequence for any greedy algorithm* twisted(n, T) *that prioritizes operation 1 over operation 2 as specified in Section 4.1 can be determined from its corresponding signed ruler sequence in Table 5.1 in the following way:* $+j$ *and* $-j$ *respectively performs operation 1 on value* $n-j+1$ *to the left and right for* $1 \leq |j| \leq n$*;* $+j$ *and* $-j$ *respectively performs operation 2 on value* $2n-j+1$ *down and up for* $n < |j| \leq 2n$*.*

5.2 Loopless Algorithms

Algorithm 7 contains procedures for generating Gray codes. Their changes follow a signed ruler sequence with any bases **b**. The start object is **s** and the change functions are in **fns**. The ruler sequence is generated one entry at a time, and the current object is updated and visited accordingly. More specifically, if j is the next entry, then **fns**$[j]$ is applied to **s** to create the next object. The pseudocode is adapted from Knuth's loopless reflected mixed-radix Gray code Algorithm H [32].

5.3 Implementation of Loopless Algorithms

In order to implement of Algorithm 7, we generate each successive entry in the signed ruler sequence that corresponds to a specific set of prioritization rules. We then utilize Lemma 3 to determine which symbol to change and how to change the signed permutation. Note that in order to keep our operations in $\mathcal{O}(1)$ time, we need to keep track of the position of each symbol in the underlying permutation. In other words, we need to store the inverse of the underlying permutation in an additional array. Furthermore, each operation should change both the signed permutation and the underlying permutation's inverse array.

Algorithm 7 Generating Gray codes using ruler sequences with bases $\mathbf{b}$. The **fns** modify object $\mathbf{s}$ and are indexed by the sequence. For example, if $\mathbf{b} = 3, 2$ then $\mathsf{RulerGrayCode}\pm(\mathbf{b})$ visits its $\mathsf{ruler}\pm(2,3) = 1, 1, 2, -1, -1$ alongside a Gray code that starts $\mathbf{s}$ and applies **fns** with indices $1, 1, 2, -1, -1$. The signed version also generates the reflected mixed-radix Gray code $\mathsf{mix}(\mathbf{b})$ in $\mathbf{a}$, with the $\mathbf{d}$ values providing ± 1 directions of change. So in the previous example the mixed-radix words $\overline{0}0, \overline{1}0, 2\overline{0}, \underline{2}1, \underline{1}1, 10$ are generated in $\mathbf{a}$. Focus pointers are stored in $\mathbf{f}$. The overall algorithm is loopless if each function runs in worst-case $\mathcal{O}(1)$-time. Note that the indexing is reversed with respect to Section 3.2 with $\mathbf{b} = b_1, b_2, \ldots, b_n$.

```
1:  procedure RulerGrayCode(b, s, fns)
2:      a_1 a_2 ··· a_n     ← 0 0 ··· 0
3:      f_1 f_2 ··· f_{n+1} ← 1 2 ··· n+1
4:      b_1 b_2 ··· b_n ← 2 2 ··· 2
5:      visit(s)
6:      while f_1 ≤ n do
7:          j ← f_1
8:          f_1 ← 1
9:          a_j ← a_j + 1
10:         s ← fns[j](s)
11:         visit(j, s)
12:         if a_j = b_j − 1 then
13:             a_j ← 0
14:             f_j ← f_{j+1}
15:             f_{j+1} ← j + 1
```

```
1:  procedure RulerGrayCode±(b, s, fns)
2:      a_1 a_2 ··· a_n     ← 0 0 ··· 0
3:      f_1 f_2 ··· f_{n+1} ← 1 2 ··· n+1
4:      d_1 d_2 ··· d_n     ← 1 1 ··· 1
5:      visit(s)
6:      while f_1 ≤ n do
7:          j ← f_1
8:          f_1 ← 1
9:          a_j ← a_j + d_j
10:         s ← fns[d_j · j](s)
11:         visit(d_j · j, s)
12:         if a_j ∈ {0, b_j − 1} then
13:             d_j ← −d_j
14:             f_j ← f_{j+1}
15:             f_{j+1} ← j + 1
```

Both Python and C++ implementations are included in Appendix. Since Python is known to integrate lambda functions with abundant functionality, which are essential for implementing the changes from signed ruler sequences, the Python code appears to be more concise. On the contrary, C++ has only introduced lambda features starting from C++11, and corresponding functionality is still limited. Consequently, complicated functional types have to be injected. For details refer to the Appendix.

Chapter 6

Spanning Tree Gray Codes

This chapter extends our previous work on Gray codes from relatively elementary combinatorial objects (i.e., binary strings, permutations, and signed permutations) to much more varied and elusive combinatorial objects.

We start by describing a recently discovered greedy Gray code algorithm for generating the bases of any matroid [39]. This algorithm is incredibly broad and flexible, but its output can be difficult to understand. One reason is that the changes made by the algorithm do not seem to follow a simple pattern. More specifically, the change sequences do not appear to be ruler sequences (or signed ruler sequences), so the approach to creating loopless algorithms seen in Chapter 5 cannot be used for the algorithms from [39], at least not in general.

We then consider several specializations of the generic algorithm for a special case of matroid bases: the spanning trees of the complete graph K_n. Don Knuth is especially interested in obtaining a simple Gray code for this special case, and he includes it as an open problem (with difficulty rating 46/50) in *The Art of Computer Programming* [33]. The problem is shown in Figure 6.1. (The term "revolving door" refers to a Gray code in which one edge is removed and one edge is added, and this is a property that is guaranteed by [39].)

> **101.** [*46*] Is there a simple revolving-door way to list all n^{n-2} spanning trees of the complete graph K_n? (The order produced by Algorithm S is quite complicated.)

Figure 6.1: Don Kunth's question on a "simple" revolving door algorithm for listing all spanning trees of K_n

Although he does not explicitly state it in his open problem, one could ultimately hope for a loopless Gray code algorithm. While we do not provide such an algorithm here, we do find that certain choices lead to outputs that may be easier to understand than others. In particular, we identify a set of choices that leads to a cyclic Gray code for $n \leq 9$. We hope that future work will give an algorithm that can claim Knuth's trophy.

6.1 Matroids

Matroids [45] generalize the notion of independence found in various areas of mathematics, including graph theory and linear algebra. More specifically, a *matroid* is a pair (E, I) where E is a finite set of elements, and I is a subset of its power set $\mathcal{P}(T)$ (i.e., I is a set of subsets of E). Each member of I (i.e., one of the selected subsets of E) is said to be *independent* and overall the independent subsets must satisfy the following conditions:

1. $\emptyset \in I$.
2. If $A \in I$ and $B \subseteq A$, then $B \in I$.
3. If $A, B \in I$ and $B \subsetneq A$, then $\exists x \in A \backslash B$ with $B \cup \{x\} \in I$.

These three conditions ensure that maximal independent sets can be generated by a simple and flexible greedy algorithm starting from $\emptyset$:

Repeatedly add any element that maintains independence.
Stop when no more elements can be added.

Less obviously, every maximal independent set has the same size in a given matroid. In other words, all maximal independent sets are maximum independent sets. Each maximum independent set is called a *basis* and collectively they are called the *bases*. Specific matroids (E, I) include the following.

- In graph theory, the set of edges in a graph form E and a subset of edges is in I if it forms a forest (i.e., there is no cycle). The maximum independent sets are the spanning trees of the graph.
- In abstract algebra, the set of all vectors in a vector space form E, and a subset of vectors is in I if it is linearly independent. The maximum independents sets form a basis of the vector space.

6.1.1 Greedy Basis-Exchange Gray Codes

Recently, a flexible approach for generating the bases of any matroid was discovered [39] (see also [41]). The bases are generated so that successive bases differ by a *basis exchange*. This means that a single element is removed and a single element is added to create the next basis. In other words, the approach generates *basis exchange Gray codes* for any matroid. For example, in the case of spanning trees, the Gray codes have the property that a single edge is removed and a single edge is added in order to create the next spanning tree. These Gray codes are also known as *revolving door Gray codes* for spanning trees.

Fittingly, the newly discovered approach is a simple and flexible greedy algorithm[1]. Given a matroid (E, I) the algorithm can be summarized as follows.

- Choose an initial basis $B_0 \in I$.

[1] This means that there are simple and flexible greedy algorithms for generating one basis of a matroid, or every basis of a matroid.

- Choose an ordering of the elements $\langle e_1, e_2, \ldots, e_m \rangle$ where $|E| = m$ and $E = \{e_1, e_2, \ldots, e_m\}$.
- Start the Gray code at B_0.
- From the most recently added basis perform any basis exchange on elements e_i and e_j that (a) minimizes $\max(i, j)$, and (b) results in a new basis to add to the Gray code. If no such choice of i and j exists, then stop.

Amazingly, this approach always works. That is, it always terminates with a Gray code containing every basis of the matroid.

The approach is flexible because it can create many different Gray codes for a given matroid. More specifically, the algorithm can start from any basis, and it can use any ordering (or *prioritization*) of the elements in E. Furthermore, each exchange can offer a choice: the largest-indexed element is fixed (i.e., it is completely determined by the previous steps of the algorithm) but amongst the valid choices the smaller-indexed element can be chosen arbitrarily. To create a specific Gray code, one must choose the initial basis, the ordering of the elements, and a process for breaking ties on smaller-indexed elements. The tiebreaker rules considered in [39] include minimizing or maximizing the smaller-indexed elements.

There are other well-known methods for efficiently generating the bases of a matroid in non-Gray code orders. For example, see the reverse search approach in [2].

6.2 Generating Spanning Trees

Spanning trees are fundamental in many algorithm fields, including path algorithms, network algorithms, etc. Efficient methods of generating all spanning trees of a given graph have been the focus of many researchers. Here we discuss two well-known methods and the special case of generating spanning trees of the complete graph.

6.2.1 Contraction and Deletion

Wilhelm Feussner developed a systematic way to enumerate the spanning trees of any graphs G in 1902 [17]: If edge $e = (u, v) \in G$, then it is either included in G or not. If it is included, then we can contract e into a single vertex and consider the spanning tree of the resulting graph G_e. If it is not included, then we simply consider the spanning tree of $G \setminus e$ which is G with edge e removed. Hence, we arrive at the following recursive relationship regrading $S(G)$, the set of all spanning trees of G:

$$S(G) = \{T \cup \{e\} \mid T \in S(G_e)\} \cup S(G \setminus e) \tag{6.1}$$

Another way of stating (6.1) is that the spanning trees of G are obtained by generating the spanning trees of G's contraction minor and deletion minor on edge e. Also note that (6.1) does not explicitly generate a Gray code order of the spanning trees. However, it could be possible to

create a Gray code by carefully choosing which edge to perform the minor on, and whether to do contraction or deletion first.

Malcolm Smith developed an elegant "revolving-door" (i.e., edge exchange) Gray code algorithm for the spanning trees of G based on 6.1 (see Algorithm S in [33]). For G containing n vertices, a "near tree" $\{e_1, e_2, \cdots, e_{n-2}\}$ that is a set of $n-2$ edges with no cycles is determined. Recursively consider all spanning trees of G_{e_1}, and each of these trees appended with e_1 is a spanning tree of G. After listing them, the algorithm is continued by using the last spanning tree of G_{e_1} as the new "near tree" and searching for the spanning trees of $G \setminus e_1$. Note that before the procedure starts, $G \setminus e_1$ must be verified to be connected; if not then switch for a different edge. Figure 6.2 - 6.4 shows an example of how deletion and contraction are implemented.

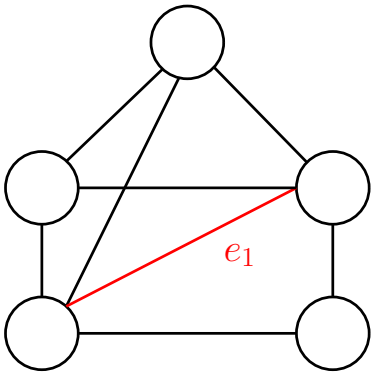


Figure 6.2: A graph G for illustrating concepts in Feussner's enumeration of spanning trees

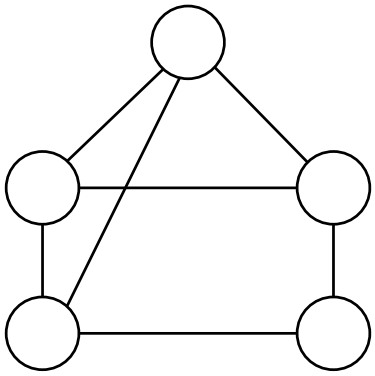


Figure 6.3: $G \setminus e_1$, with e_1 deleted from G

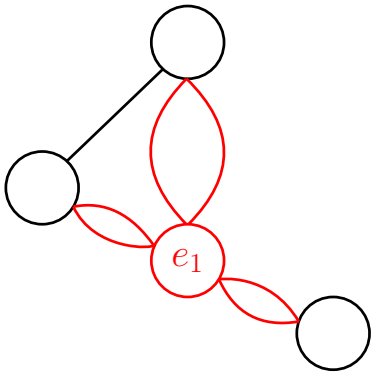


Figure 6.4: G_{e_1}, with e_1 contracted to a vertex

6.2.2 Spanning Trees of Complete Graphs

The *complete graph* K_n has n vertices and an edge between every pair of vertices. Notably, K_n has n^{n-2} different spanning trees, and the set containing them are referred as T_n. Figure 6.5 illustrates T_4, all $4^2 = 16$ spanning trees of complete graph K_4.

Though Smith's algorithm is fairly easy to understand and implement, the resulting Gray code generally appears to be extremely chaotic and difficult to interpret. One would hope that the inherent symmetry demonstrated by complete graphs would lead to Smith's algorithm producing neat and interpretable Gray codes when they are applied on K_n. However, this does not seem to be the case. More broadly, searching for a neat Gray code of the n^{n-2} spanning trees of K_n remains an extraordinarily challenging task, as Knuth designated the question in exercise of *Art of Programming*, Vol. 4A with a difficulty rating of 46/50 [32] (see Figure 6.1).

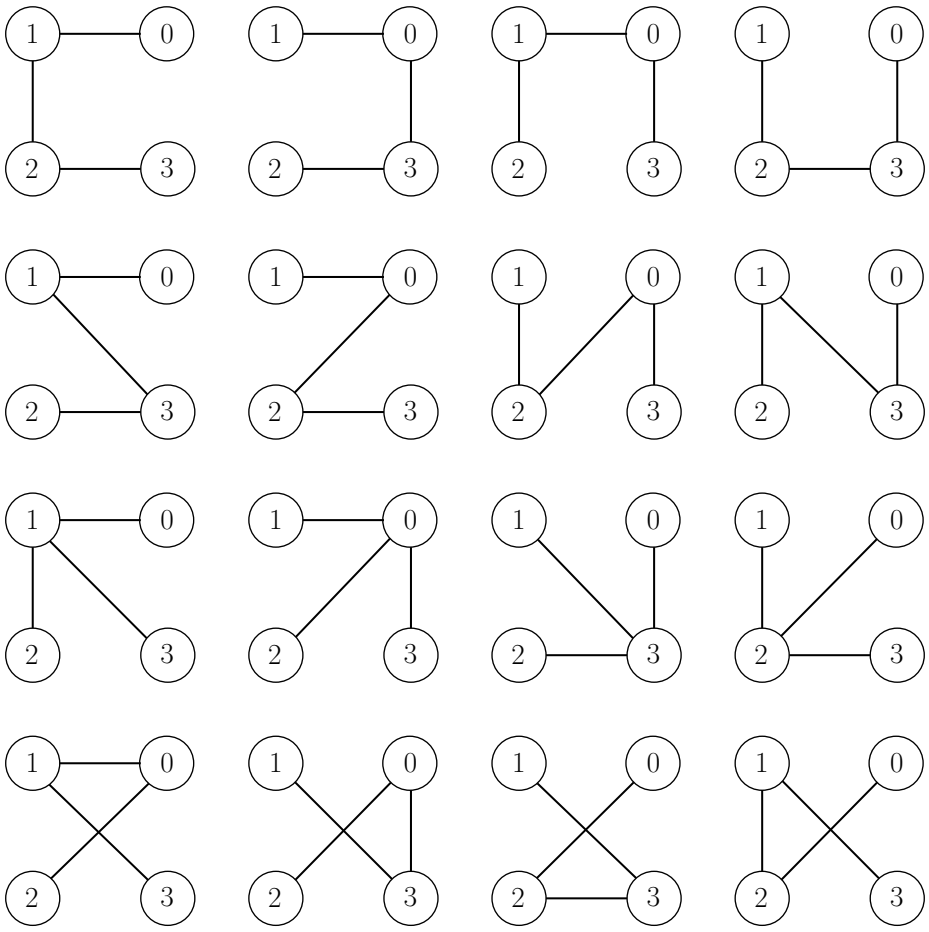


Figure 6.5: T_4, Spanning trees of the complete graph K_4.

6.3 A Greedy Revolving Door Algorithm

As in Chapter 1, we saw traditional, recursive definitions of the BRGC and plain change order, we revealed that those common examples of Gray code can be defined in a more straightforward way by using a Gray code algorithm in Chapter 2. As we will discover in this section, a similar situation also applies for spanning trees for complete graph K_n. In the last section we discussed a Gray code for spanning trees by Malcolm Smith using via contraction and deletion. Recent work [39] has uncovered a more simple and flexible greedy algorithms.

Since the complete graph will never be disconnected if a single edge is deleted, the "revolving door" algorithm could be substantially simplified. The thought is to specify a starting spanning tree **s**, and each time delete an edge from **s** then insert another edge. The deletion and insertion will follow a certain set of prioritization rules, thus converting the procedure into a greedy algorithm.

Algorithm 8 displays our implementation of the flexible greedy algorithm for generating Gray codes of spanning trees. The parameters of this algorithm include the initial spanning tree, the order of edges in which they are deleted or inserted in the outer loop (i.e. how to order the elements of the

Algorithm 8 Greedy algorithm for generating Gray codes of spanning trees $\mathsf{spanning}(n, \mathbf{first}, \mathbf{O}, \mathbf{T}())$. By default vertices of the complete graph are numbered $0, 1, \cdots, n-1$, and edges are stored by their two vertices. The order starts with the spanning tree **first**. **O** illustrates the priority that edges are deleted or inserted. $\mathbf{T}(\mathbf{s}, \mathbf{i})$ is the tiebreaker rule that determines the order of edges to select in the inner loop, which in our case is defined either in the increasing order: $\mathbf{T}(\mathbf{s}, \mathbf{i}) = 1, 2, \cdots, i$ or the decreasing order: $\mathbf{T}(\mathbf{s}, \mathbf{i}) = i, i-1, \cdots, 1$.

```
1:  procedure spanning(n, first, O, T())
2:      s ← first                                  ▷ Starting spanning tree s = first
3:      S ← {s}                                    ▷ Add s to the visited set
4:      visit(s)                                   ▷ Visit s for the first and only time
5:      i ← 1                                      ▷ 1-based index into O
6:      for i ≤ size(O) do                         ▷ Index i iterates through all no. of edges in O
7:          for j ∈ T(s, i) do                     ▷ Index j iterates through all edges in T(s, i)
8:              e1 ← O[i]
9:              e2 ← O[j]
10:             if e1 ∈ s & e2 ∉ s then            ▷ Check if only one of e1, e2 is in s
11:                 s' ← (s\e1) ∪ e2               ▷ Delete one edge and insert the other
12:             else if e2 ∈ s & e1 ∉ s then
13:                 s' ← (s\e2) ∪ e1
14:             else
15:                 continue
16:             if s' ∈ Tn & s' ∉ S then           ▷ Check if s' is a new spanning tree
17:                 s ← s'                         ▷ Update the current spanning tree s
18:                 visit(s)                       ▷ Visit s for the first and only time
19:                 S ← S ∪ s                      ▷ Add s to the visited set
20:                 i ← 1                          ▷ Reset i
```

matroid as well as the first basis in the order), and the order of edges to consider in the inner loop (tiebreaker). In our algorithms, the tiebreaker is either increasing or decreasing order, but technically other orders may be considered and the order may even be dependent on the choice in the outer loop [39]. As a result, the greedy algorithm is highly flexible and can generate a large number of different Gray codes. Unlike the signed permutation Gray codes, we do not have a clear understanding of these Gray codes. In order to obtain "neat" Gray codes, we are especially interested in rulesets which produce either a cyclic Gray code or an elegant ending spanning tree given a simple starting spanning tree. The goal is to interpret one or several spanning tree Gray codes so that they can be generated efficiently, similar as the Gray codes for signed permutations we generated in Chapter 5.

Figure 6.6 - 6.9 displays two examples of Algorithm 8. Note that in both examples, we use $(0,1),(0,2),(0,3)$ as the starting spanning tree of K_4; Figure 6.6 and Figure 6.8 displays the order of edges **O** in the two cases; and the first case uses the decreasing order as tiebreaker, while the second case uses the increasing tiebreaker. The resulting Gray codes are also referred as Gray code $1b$ and $2a$ in the next section.

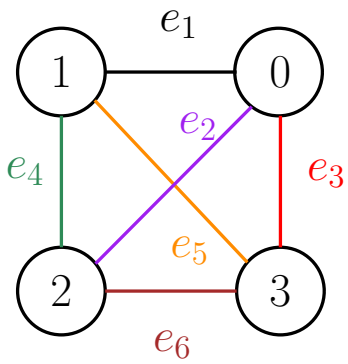


Figure 6.6: Complete graph K_4 with edges ordered according to increasing first and increasing second vertices.

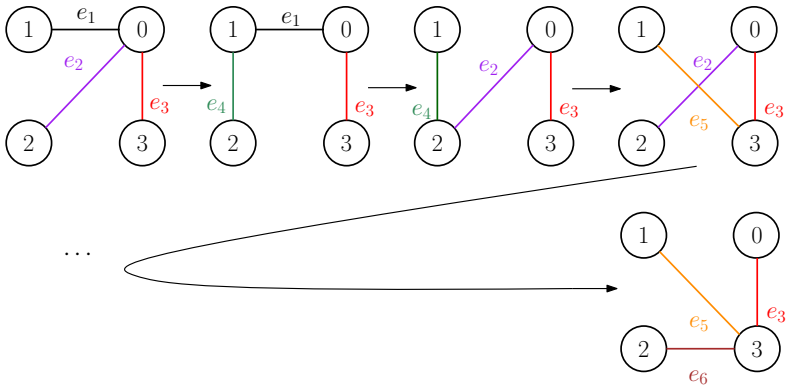


Figure 6.7: Consequent Gray code of spanning trees.

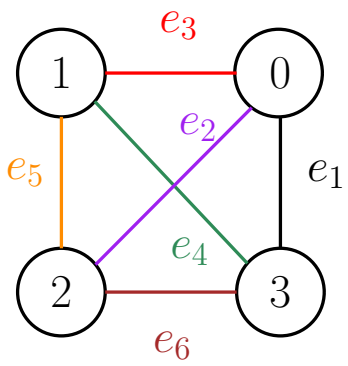


Figure 6.8: Complete graph K_4 with edges ordered according to increasing first and decreasing second vertices.

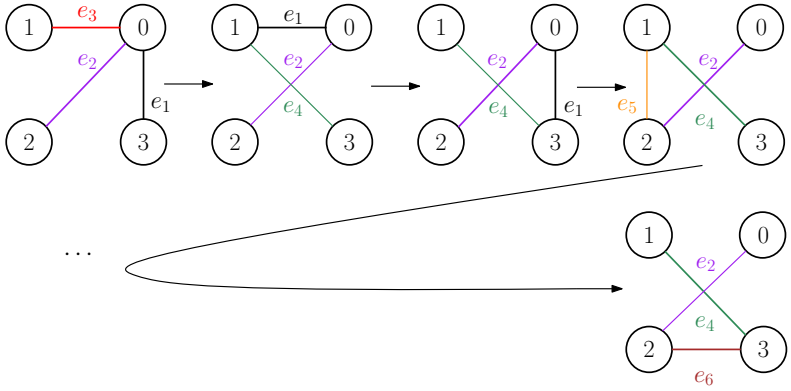


Figure 6.9: Consequent Gray code of spanning trees.

6.4 Experimental Results

[39] proves that the greedy algorithm for Gray codes of spanning trees of K_n is valid for all choices of edge labels, first spanning tree, and tiebreaker rules. Nevertheless, there is no guarantee that the resulting order is simpler, or that it can be generated efficiently. It isn't even clear how we could identify that an order is "simple". In this section, we adopt the idea that an order is hopefully simple if its final spanning tree is predictable. We run several experiments to check the final spanning tree based on various combinations of choices.

We nominate spanning trees containing all edges with distance 1 as the starting spanning tree for all our experiments, which are paths $(0,1),(1,2),\cdots,(n-2,n-1)$.

We focused our experiments on 4 different ordering of edges (**O**) combined with the 2 natural tiebreaker rules (increasing and decreasing order), thus producing 8 different Gray codes. The

ordering in O is listed as following:

Order 1: Arrange vertices of each edge in lexicographic order $(0,1) < (0,2) < (0,3) < \cdots < (0,n-1) < (1,2) < (1,3) < \cdots < (1,n-1) < \cdots < (n-2,n-1)$.

Order 2: Arrange the first vertex of each edge in increasing order and second vertex in decreasing order: $(0,n-1) < (0,n-2) < (0,n-3) < \cdots < (0,1) < (1,n-1) < (1,n-2) < \cdots < (1,2) < \cdots < (n-2,n-1)$.

Order 3: Arrange vertices in the order of increasing "distance" between vertices and then the first vertex in increasing order $(0,n-1) < (0,n-2) < (1,n-1) < (0,n-3) < (1,n-2) < (2,n-1) < \cdots < (0,1) < (1,2) < (2,3) < \cdots < (n-2,n-1)$.

Order 4: Arrange vertices in the order of decreasing "distance" between vertices and then the first vertex in increasing order $(0,1) < (1,2) < (2,3) < \cdots < (n-2,n-1) < (0,2) < (1,3) < \cdots < (0,n-1)$.

Designating the increasing tiebreaker as **a** and decreasing tiebreaker as **b**, the 8 different Gray codes are numbered as **1a**, **1b**, **2a**, $\cdots$, **4b**. After inspection with regards to the standards mentioned at the end of Section 6.3, we found Gray code $3a$ and $4a$ to be especially promising candidates:

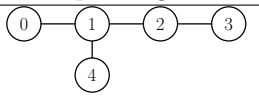
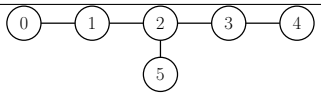
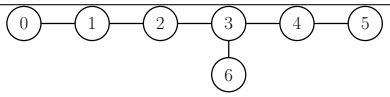
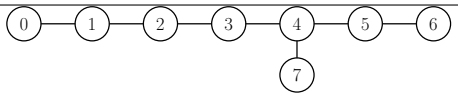
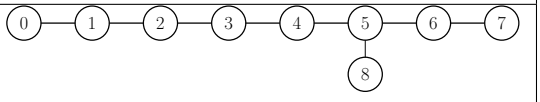
n	Last spanning tree
5	
6	
7	
8	
9	

Table 6.1: Gray code **3a**.

Note that both Gray codes use the distance between vertices of edges (i.e. the difference between the numbers on the vertices). From Table 6.1 we observe that in all cases, the Gray code **3a** is cyclic since the last spanning tree always differs from the first spanning tree from replace the edge of the largest distance $(0,n-1)$ with an edge of the shortest distance $(n-2,n-1)$. Such an Gray code is considered as a valid candidate for our goal. From Table 6.2, we observe that Gray code **3a** is non-cyclic, albeit seemingly transforming the starting spanning tree into another path attached with several branches. We speculate that some sort of more formulatable relationship between the first and last spanning tree may exist, but more meticulous research on the Gray code is required.

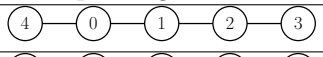
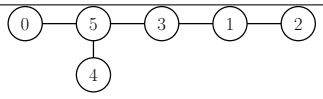
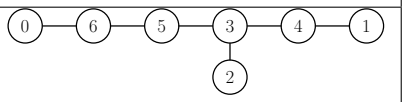
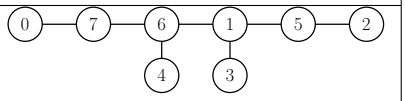
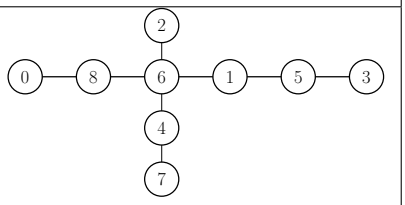
n	Last spanning tree
5	
6	
7	
8	
9	

Table 6.2: Gray code **4a**.

Though we nominated two candidates for Knuth's problem on Gray codes of complete graph K_n, we still do not understand how to implement these Gray codes looplessly, or if there exist loopless algorithms that generate these Gray codes. This is an area for future work.

Chapter 7

Summary

This thesis began by considering orderings of objects so that consecutive objects are close to each other. This includes Lewis Carroll's notion of a word ladder puzzle, as well as the general notion of a Gray code. We considered the two most celebrated Gray codes, the binary reflected Gray code for binary strings and plain changes for permutations. We described how these orders can be constructed recursively, and visualized them as Hamilton paths in their underlying flip graphs. Then we saw simpler greedy descriptions of these orders, and how to generate them with loopless algorithms using associated ruler sequences. Our focus then changed to creating Gray codes for signed permutations.

We modeled signed permutations using two-sided ribbons, which helped motivate twist operations. Inspired by the greedy algorithms for the binary reflected Gray code and plain changes, we experimented with a variety of greedy algorithms for signed permutations using twists, flips, and swaps. We found a dozen such algorithms that worked empirically, and considered why they worked in general. We provided three representative proofs of correctness, which involved looking at the resulting orders in blocks of length $n!$ (following permutohedra), 2^n (following cubes), or $2n$ (following zig-zags). Using the efficient loopless algorithms for the binary reflected Gray code and plain changes as guidelines, we developed loopless generation algorithms for our new Gray codes using signed ruler sequences. In other words, we accomplished our goal of creating greedy and speedy Gray code algorithms for signed permutations.

Finally, we investigated new greedy Gray codes results for matroid bases [39]. These algorithms are flexible and can provide a wide variety of specific Gray codes for various combinatorial objects. However, they are not yet well understood in the sense that they can be generated by simple efficient algorithms. We specifically considered tuning these algorithms for the spanning trees of the complete graph, for which there is an associated open 46/50 difficulty problem posted by Knuth. We considered two specific sets of parameters, and found that one set appears to generate a cyclic Gray code, which is a result that was not anticipated in [39]. This is a promising direction that we hope to investigate in the future.

For additional future work we mention the problem of creating successor rules for our Gray codes. Given a Gray code, a *successor rule* is a function that efficiently maps each object to the

next object. Note that these rules derive all of the necessary information from the current object, and do not use any alternate information or sequence to determine the next change. In particular, our algorithms in Chapter 5 do not employ successor rules, since the next change is provided by a signed ruler sequence.

It is worth noting that some Gray codes have been developed by creating a successor rule from scratch. In other words, the first description of the Gray code was a successor rule. This includes *cool-lex orders*, which were investigated by Paul Lapey ('22) [35] for Łukasiewicz words [37] and ordered trees [36][1] as well as the *sigma-tau order* for permutations (see [67] [55] [56] [38]). However, it is more common for Gray codes to be developed in another manner (i.e., recursively or greedily) before the successor rule is derived. To add hope to this ambition, we mention that a successor rule was developed for the greedy Gray code of colored permutations using flips (sign-incrementing prefix-reversals) [7, 6].

[1] Cool-lex orders exist for other objects including combinations [51], binary strings [60, 61], balanced parentheses [50], k-ary trees [14], Catalan squares [12], and bubble languages [49] [69].

Chapter 8

Bibliography

[1] Rotary encoder - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Rotary_encoder. [Accessed 11-05-2024].

[2] Avis, D., and Fukuda, K. Reverse search for enumeration. *Discrete applied mathematics 65*, 1-3 (1996), 21–46.

[3] Barton, M. *redbo/scrabble*. 2015.

[4] Batchelor, M., and Henle, J. The mathematical art of change ringing, 2022.

[5] Brits, B. Ritual, code, and matheme in Samuel Beckett's Quad. *Journal of Modern Literature 40*, 4 (2017), 122–133.

[6] Cameron, B., Sawada, J., Therese, W., and Williams, A. Hamiltonicity of k-sided pancake networks with fixed-spin: Efficient generation, ranking, and optimality. *Algorithmica 85*, 3 (2023), 717–744.

[7] Cameron, B., Sawada, J., and Williams, A. A Hamilton cycle in the k-sided pancake network. In *Combinatorial Algorithms: 32nd International Workshop, IWOCA 2021, Ottawa, ON, Canada, July 5–7, 2021, Proceedings 32* (2021), Springer, pp. 137–151.

[8] Carroll, L. *Doublets-A Word-Puzzle*. Read Books Ltd, 2016.

[9] Cooke, M., North, C., Dewar, M., and Stevens, B. A note on Beckett-Gray codes and the relationship of Gray codes to data structures. *arXiv preprint arXiv:1608.06001* (2016).

[10] Corbett, P. F. Rotator graphs: An efficient topology for point-to-point multiprocessor networks. *IEEE Trans Parallel Distrib Syst. 3*, 5 (1992), 622–626.

[11] Dewar, M., and Stevens, B. *Ordering block designs: Gray codes, universal cycles and configuration orderings*. Springer Science & Business Media, 2012.

[12] Downing, E., Einstein, S., Hartung, E., and Williams, A. Catalan squares and staircases: Relayering and repositioning Gray codes. In *Proceedings of the 35th Canadian Conference on Computational Geometry, CCCG* (2023).

[13] Duckworth, R., and Stedman, F. *Tintinnalogia: Or, The Art of Ringing*. Kingsmead Reprints, 1970.

[14] Durocher, S., Li, P. C., Mondal, D., Ruskey, F., and Williams, A. Cool-lex order and k-ary catalan structures. *Journal of Discrete Algorithms 16* (2012), 287–307.

[15] Ehrlich, G. Loopless algorithms for generating permutations, combinations, and other combinatorial configurations. *J. ACM 20*, 3 (1973), 500–513.

[16] Essed, H., and Therese, W. The harassed waitress problem. In *Fun with Algorithms: 7th International Conference, FUN 2014, Lipari Island, Sicily, Italy, July 1-3, 2014. Proceedings 7* (2014), Springer, pp. 325–339.

[17] Feussner, W. Ueber stromverzweigung in netzförmigen leitern. *Annalen der Physik 314*, 13 (1902), 1304–1329.

[18] Ganapathi, P., and Chowdhury, R. A unified framework to discover permutation generation algorithms. *The Computer Journal 66*, 3 (2023), 603–614.

[19] Gardner, M. Curious properties of the Gray code and how it can be used to solve puzzles. *Scientific American 227*, 2 (1972), 106.

[20] Goldstein, A. A computer oriented algorithm for generating permutations. *Bell Telephone Laboratories (Internal Memorandum MM-64-1271-3), Murray Hill, NJ* (1964).

[21] Goldstein, A., and Graham, R. Sequential generation by transposition of all the arrangements of n symbols. *Bell Telephone Laboratories (Internal Memorandum MM-64-1271-3 / MM-64-1213-12), Murray Hill, NJ* (1964), 14.

[22] Gondor, G. Absolute Rotary Encoder — tikz.net. https://tikz.net/absolute-rotary-encoder/. [Accessed 11-05-2024].

[23] Gray, F. Pulse code communication. *United States Patent Number 2632058* (1953).

[24] Hannenhalli, S., and Pevzner, P. A. Transforming cabbage into turnip: polynomial algorithm for sorting signed permutations by reversals. In *Proc. of the 27th Annual ACM Symposium on Theory of Computing (STOC 95)* (1995), ACM, pp. 178–189.

[25] Haunsperger, D., and Kennedy, S. *The Edge of the Universe: Celebrating Ten Years of Math Horizons*, vol. 48. MAA, 2006.

[26] Heath, F. Origins of the binary code. *Scientific American 227*, 2 (1972), 76–83.

[27] Hindenburg, C. F. *Sammlung combinatorisch-analytischer Abhandlungen*, vol. 1. ben Gerhard Fleischer dem Jungern, 1796.

[28] Holroyd, A. E., Ruskey, F., and Williams, A. Shorthand universal cycles for permutations. *Algorithmica 64* (2012), 215–245.

[29] Jackson, B. W. Universal cycles of k-subsets and k-permutations. *Discrete mathematics 117*, 1-3 (1993), 141–150.

[30] Johnson, S. M. Generation of permutations by adjacent transposition. *Mathematics of computation 17*, 83 (1963), 282–285.

[31] Kleitman, D., Kramer, E., Conway, J., Bell, S., and Dweighter, H. Elementary problems: E2564-e2569. *The American Mathematical Monthly 82*, 10 (1975), 1009.

[32] Knuth, D. E. *The Art of Computer Programming, Volume 4A, Fascicle 2: Generating All Tuples and Permutations*. Addison-Wesley, 2005.

[33] Knuth, D. E. *Art of Computer Programming, Volume 4A, Fascicle 4, The: Generating All Trees–History of Combinatorial Generation*. Addison-Wesley Professional, 2013.

[34] Korsh, J., LaFollette, P., and Lipschutz, S. A loopless implementation of a Gray code for signed permutations. *Publications de l'Institut Mathematique 89*, 103 (2011), 37–47.

[35] Lapey, P. Cooler than cool: Cool-lex order for generating new combinatorial objects. Bachelor's thesis, Williams College, 2022.

[36] Lapey, P., and Williams, A. Pop & push: Ordered tree iteration in $o(1)$-time. In *33rd International Symposium on Algorithms and Computation (ISAAC 2022)* (2022), Schloss-Dagstuhl-Leibniz Zentrum für Informatik.

[37] Lapey, P. W., and Williams, A. A shift Gray code for fixed-content Łukasiewicz words. In *International Workshop on Combinatorial Algorithms* (2022), Springer, pp. 383–397.

[38] Liptak, Z., Masillo, F., Navarro, G., and Williams, A. Constant time and space updates for the sigma-tau problem. In *Proc. of the 30th International Symposium on String Processing and Information Retrieval (SPIRE 2023)* (2023), Springer, p. 6 pages.

[39] Merino, A., Mutze, T., and Williams, A. All your bases are belong to us: Listing all bases of a matroid by greedy exchanges. In *11th International Conference on Fun with Algorithms (FUN 2022)* (2022), vol. 226, Schloss Dagstuhl—Leibniz-Zentrum für Informatik, p. 22.

[40] Merino, A., Namrata, and Williams, A. On the hardness of Gray code problems for combinatorial objects. In *International Conference and Workshops on Algorithms and Computation* (2024), Springer, pp. 103–117.

[41] Merino Figueroa, A. I. Combinatorial generation: greedy approaches and symmetry.

[42] Mütze, T. Combinatorial Gray codes-an updated survey. *arXiv preprint arXiv:2202.01280* (2022).

[43] Mütze, T., Sawada, J., and Williams, A. combos.org. http://www.combos.org/. [Accessed 11-05-2024].

[44] OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2023. Published electronically at http://oeis.org.

[45] Oxley, J. G. *Matroid theory*, vol. 3. Oxford University Press, USA, 2006.

[46] Qiu, Y. Signedperm. https://github.com/friedrichq2002/SignedPerm, 2024.

[47] Qiu, Y., and Williams, A. Generating signed permutations by twisting two-sided ribbons. In *Latin American Symposium on Theoretical Informatics* (2024), Springer, pp. 114–129.

[48] Ruskey, F. Combinatorial generation. *Preliminary working draft. University of Victoria, Victoria, BC, Canada 11* (2003), 20.

[49] Ruskey, F., Sawada, J., and Williams, A. Binary bubble languages and cool-lex order. *Journal of Combinatorial Theory, Series A 119*, 1 (2012), 155–169.

[50] Ruskey, F., and Williams, A. Generating balanced parentheses and binary trees by prefix shifts. In *CATS* (2008), vol. 8, Citeseer, p. 140.

[51] Ruskey, F., and Williams, A. The coolest way to generate combinations. *Discrete Mathematics 309*, 17 (2009), 5305–5320.

[52] Savage, C. A survey of combinatorial Gray codes. *SIAM review 39*, 4 (1997), 605–629.

[53] Sawada, J., and Williams, A. Greedy flipping of pancakes and burnt pancakes. *Discrete Applied Mathematics 210* (2016), 61–74.

[54] Sawada, J., and Williams, A. Successor rules for flipping pancakes and burnt pancakes. *Theoretical Computer Science 609* (2016), 60–75.

[55] Sawada, J., and Williams, A. A Hamilton path for the sigma-tau problem. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms* (2018), SIAM, pp. 568–575.

[56] Sawada, J., and Williams, A. Solving the sigma-tau problem. *ACM Transactions on Algorithms (TALG) 16*, 1 (2019), 1–17.

[57] Sawada, J., and Wong, D. C.-H. A fast algorithm to generate Beckett-Gray codes. *Electronic Notes in Discrete Mathematics 29* (2007), 571–577.

[58] Sedgewick, R. Permutation generation methods. *ACM Computing Surveys (CSUR) 9*, 2 (1977), 137–164.

[59] Steinhaus, H. *One hundred problems in elementary mathematics*. Courier Corporation, 1979.

[60] Stevens, B., and Williams, A. The coolest order of binary strings. In *Fun with Algorithms: 6th International Conference, FUN 2012, Venice, Italy, June 4-6, 2012. Proceedings 6* (2012), Springer, pp. 322–333.

[61] Stevens, B., and Williams, A. The coolest way to generate binary strings. *Theory of Computing Systems 54* (2014), 551–577.

[62] Stigler, S. M. Stigler's law of eponymy. *Transactions of the New York academy of sciences 39*, 1 Series II (1980), 147–157.

[63] Trotter, H. F. Algorithm 115: perm. *Communications of the ACM 5*, 8 (1962), 434–435.

[64] Wikipedia. Word ladder — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Word%20ladder&oldid=1214579348, 2024. [Online; accessed 10-May-2024].

[65] Williams, A. Loopless generation of multiset permutations using a constant number of variables by prefix shifts. In *Proceedings of the twentieth annual ACM-SIAM symposium on discrete algorithms* (2009), SIAM, pp. 987–996.

[66] Williams, A. The greedy Gray code algorithm. In *Algorithms and Data Structures: 13th International Symposium, WADS 2013, London, ON, Canada, August 12-14, 2013. Proceedings 13* (2013), Springer, pp. 525–536.

[67] Williams, A. Hamiltonicity of the Cayley digraph on the symmetric group generated by $\sigma = (1\,2\,\ldots\,n)$ and $\tau = (1\,2)$. *arXiv preprint arXiv:1307.2549* (2013).

[68] Williams, A. signed-plain-changes. https://gitlab.com/combinatronics/signed-plain-changes, 2024.

[69] Williams, A. M. *Shift Gray codes*. PhD thesis, 2009.

[70] Zaks, S. A new algorithm for generation of permutations. *BIT Numerical Mathematics 24*, 2 (1984), 196–204.

Appendix A

Selected Implementations

This appendix includes Python and C++ programs for generating the Gray codes discussed in this thesis. The programs are also available online [68, 46].

The first program is written in Python 3. It is a loopless implementation of our signed plain change order twisted(n) (Experiment 8), but it can be adjusted to generate any of the other 11 Gray codes. It generates the signed ruler sequence ruler±$(n, n-1, \ldots, 2, 1, 2, 2, \ldots, 2)$, with each entry selecting the 2-twist or 1-twist (i.e., flip) to apply. Some implementation notes are below.

- Negative indices give right-to-left access in Python, so the ruler entry `-1` selects the last function `fns[-1] = twist(p,q,n,1)` (i.e., 2-twist n right).
- The slice notation `[-1::-1]` reverses a list. Also note that the order of indices is reversed from Chapter 3.
- The `v=v` default values in the lambda functions are for proper binding.

The second program is written in C++. It provides a loopless implementation of our signed plain change order twisted(n) (Experiment 8). More specifically, it is a streamlined version of the third program. This program also illustrates the use of a `visit` function. The user of the generation program (i.e., the application) can provide their own `visit` function that will be called every time a new object is created. In particular, a sample `visit` function called `visit_print` is provided in [46], and it simply outputs each successive signed permutation. More broadly, the `visit` function is called with the specific change that is made to create the next signed permutation. In this way, the user of the program does not need to scan for the location of the change, so they can update certain associated information in worst-case $\mathcal{O}(1)$-time based on the operation that was performed. For more information, see the discussion of exact algorithms in Section 1.2.3, or Section 1.2 of [28] for the efficient evaluation of TSP problems using permutation Gray codes.

The third program is written in C++. It provides loopless implementations of all 12 Gray codes created in this thesis. The signed ruler sequences are generated in `loopless_signed_perm` with a similar way as in the Python program. The signed permutations are changed in correspondence with the signed ruler sequences, and for each of the 12 Gray codes, corresponding functions representing

operations and bases are passed in as variables. Standalone compilable versions of this program as well as the third program are available in [46].

- `base_type` indicates bases of ruler sequence: 0 is `two_dec(n)`$= (2, 2, \ldots, 2, n, n{-}1, \ldots, 2, 1)$, 1 is `dec_two(n)`$= (n, n{-}1, \ldots, 1, 2, 2, \ldots, 2, 2)$, 2 is `two_dec_zigzag(n)`$= (2, n, 2, n{-}1, \ldots, 2, 2, 2, 1)$, 3 is `dec_two_zigzag(n)`$= (n, 2, n-1, 2, \ldots, 2, 2, 1, 2)$.
- `algorithm` is a lambda function that takes change from ruler sequence and returns the symbol to be changed and the operation type, which are individually defined by the 12 Gray codes.
- `visit` is a user-defined function that is called for each object and change. It can be written to perform printing, debugging, etc.

```python
# Flip sign of value v in signed permutation p with unsigned inverse q
def flip(p, q, v):  # with 1-based indexing, p[0] and q[0] are ignored.
    p[q[v]] = -p[q[v]]
    return p, q

# 2-twists value v to the left / right using delta = -1 / delta = 1
def twist(p, q, v, delta):  # with 1-based indexing into both p and q.
    pos = q[v]           # Use inverse to get the position of value v.
    u = abs(p[pos+delta]) # Get value to the left or right of value v.
    p[pos], p[pos+delta] = -p[pos+delta], -p[pos]     # Twist u and v.
    q[v], q[u] = pos+delta, pos             # Update unsigned inverse.
    return p, q  # Return signed permutation and its unsigned inverse.

# Generate each signed permutation in worst-case O(1)-time.
def twisted(n):
  m = 2*n-1  # The mixed-radix bases are n, n-1, ..., 2, 1, 2,..., 2.
  bases = tuple(range(n,1,-1)) + (2,) * n     # but the 1 is omitted.
  word = [0] * m            # The mixed-radix word is initially 0^m.
  dirs = [1] * m            # Direction of change for digits in word.
  focus = list(range(m+1))  # Focus pointers select digits to change.
  flips =   [lambda p,q,v=v: flip(p,q,v)     for v in range(n,0,-1)]
  twistsL = [lambda p,q,v=v: twist(p,q,v,-1) for v in range(n,1,-1)]
  twistsR = [lambda p,q,v=v: twist(p,q,v, 1) for v in range(n,1,-1)]
  fns = [None] + twistsL + flips + flips[-1::-1] + twistsR[-1::-1]
  p = [None] + list(range(1,n+1))  # To use 1-based indexing we set
  q = [None] + list(range(1,n+1))  # and ignore p[0] = q[0] = None.
  yield p[1:]  # Pause the function and return signed permutation p.
  while focus[0] < m: # Continue if the digit to change is in word.
    index = focus[0]  # The index of the digit to change in word.
    focus[0] = 0      # Reset the first focus pointer.
    word[index] += dirs[index]  # Adjust the digit using its direction.
    change = dirs[index] * (index+1)  # Note: change can be negative.
    if word[index] == 0 or word[index] == bases[index]-1:  # If the
      focus[index] = focus[index+1] # mixed-radix word's digit is at
      focus[index+1] = index+1      # its min or max value, then update
      dirs[index] = -dirs[index]    # focus pointers, change direction.
    p, q = fns[change](p, q)  # Apply twist or flip encoded by change.
    yield p[1:]

# Demonstrating the use of our twisted function for n = 4.
for p in twisted(4): print(p)  # Print all 2^n n! signed permutations.
```

Figure A.1: `twisted.py` [68] twisted(n) (Experiment 8). It can be adapted to generate our other Gray codes by modifying the signed ruler `bases` and the `fns` that are applied for each entry in the associated signed ruler sequence. Similarly, a streamlined version (without lambda functions) can be made by replacing `fns[change]` with specific calls to the `flip` and `twist` functions.

```
void signed_perm_8(int n, function<void(vector<int>, int, string)> visit) {
  vector<int> cur_perm(n);
  vector<int> inverse(n + 1);
  iota(cur_perm.begin(), cur_perm.end(), 1);
  iota(inverse.begin(), inverse.end(), -1);
  vector<int> bases; // bases = [n,n-1,...,2,1,2,2,...,2,2]
  for (int i = n; i > 0; i--) bases.push_back(i);
  for (int i = 0; i < n; i++) bases.push_back(2);
  vector<int> word(bases.size(), 0);
  vector<int> dir(bases.size(), 1);
  vector<int> focus(bases.size() + 1);
  iota(focus.begin(), focus.end(), 0);
  while (focus[0] < bases.size()) {
    int index = focus[0];
    focus[0] = 0;
    word[index] += dir[index];
    int change = dir[index] * (index + 1);
    if (word[index] == 0 || word[index] == bases[index] - 1) {
      dir[index] = -dir[index];
      int extra = (index < bases.size()-1 && bases[index+1] == 1 ?1:0);
      focus[index] = focus[index + extra + 1];
      focus[index + extra + 1] = index + extra + 1;
    }
    int val = -1; int change_type = -1;
    if (abs(change) > n) { val = 2*n+1 - abs(change); change_type = 0; }
    else { val = n+1 - abs(change); change_type = (change > 0 ?1:2);}
    int pos = inverse[abs(val)];
    switch (change_type){
    case 0: {
      visit(cur_perm, val, "t1"); // flip sign of value
      cur_perm[pos] *= -1;
      break;}
    case 1: {
      visit(cur_perm, val, "t2l"); // twist value to the left
      int other = cur_perm[pos - 1];
      swap(cur_perm[pos - 1], cur_perm[pos]);
      cur_perm[pos] *= -1;
      cur_perm[pos - 1] *= -1;
      inverse[abs(val)] = pos - 1;
      inverse[abs(other)] = pos;
      break;}
    case 2: {
      visit(cur_perm, val, "t2r"); // twist value to the right
      int other = cur_perm[pos + 1];
      swap(cur_perm[pos + 1], cur_perm[pos]);
      cur_perm[pos] *= -1;
      cur_perm[pos + 1] *= -1;
      inverse[abs(val)] = pos + 1;
      inverse[abs(other)] = pos;
      break;}
    }
  }
  visit(cur_perm, 0, "o"); // the last permutation has no change
}
```

Figure A.2: `signedPerm8.cpp` [46] generates twisted(n) (Experiment 8).

```
vector<int> two_dec(int n){
  vector<int> ans(n, 2);
  for (int i = n; i > 0; i--) ans.push_back(i);
  return ans;
}

vector<int> dec_two(int n){
  vector<int> ans = {};
  for (int i = n; i > 0; i--) ans.push_back(i);
  for (int i = 0; i < n; i++) ans.push_back(2);
  return ans;
}

vector<int> two_dec_zigzag(int n){
  vector<int> ans = {};
  for (int i = n; i > 0; i--){
    ans.push_back(2);
    ans.push_back(i);
  }
  return ans;
}

vector<int> dec_two_zigzag(int n){
  vector<int> ans = {};
  for (int i = n; i > 0; i--){
    ans.push_back(i);
    ans.push_back(2);
  }
  return ans;
}

void loopless_signed_perm(int n, int base_type,
  function<pair<int, int>(int)> algorithm,
  function<void(vector<int>, int, string)> visit){
  vector<int> cur_perm(n);
  vector<int> inverse(n+1);
  iota(cur_perm.begin(), cur_perm.end(), 1);
  iota(inverse.begin(), inverse.end(), -1);
  vector<int> bases;
  switch(base_type){
    case 0:
      bases = two_dec(n);
      break;
```

```
    case 1:
      bases = dec_two(n);
      break;
    case 2:
      bases = two_dec_zigzag(n);
      break;
    case 3:
      bases = dec_two_zigzag(n);
      break;
  }
  vector<int> word(bases.size(), 0);
  vector<int> dir(bases.size(), 1);
  vector<int> focus(bases.size() + 1);
  iota(focus.begin(), focus.end(), 0);
  int first = (bases[0] > 1 ? 0 : 1);
  while(focus[first] < bases.size()){
    int index = focus[first];
    focus[first] = first;
    word[index] += dir[index];
    int change = dir[index] * (index + 1);
    if (word[index] == 0 || word[index] == bases[index] - 1){
      dir[index] = -dir[index];
      int extra = (index < bases.size() - 1 && bases[index+1] == 1 ?1:0);
      focus[index] = focus[index + extra + 1];
      focus[index + extra + 1] = index + extra + 1;
    }
    pair<int, int> op = algorithm(change);
    int val = op.first;
    int pos = inverse[abs(val)];
    switch (op.second){
      case 0: {
        visit(cur_perm, val, "sl"); // swap value to the left
        int other = cur_perm[pos - 1];
        swap(cur_perm[pos - 1], cur_perm[pos]);
        inverse[abs(val)] = pos - 1;
        inverse[abs(other)] = pos;
        break;
      }
      case 1: {
        visit(cur_perm, val, "sr"); // swap value to the right
        int other = cur_perm[pos + 1];
        swap(cur_perm[pos + 1], cur_perm[pos]);
        inverse[abs(val)] = pos + 1;
        inverse[abs(other)] = pos;
```

```
        break;
      }
      case 2: {
        visit(cur_perm, cur_perm[pos], "t1"); // flip sign at index
        pos = abs(val)-1;
        cur_perm[pos] *= -1;
        break;
      }
      case 3: {
        visit(cur_perm, val, "t1"); // flip sign of value
        cur_perm[pos] *= -1;
        break;
      }
      case 4:  {
        visit(cur_perm, val, "t2l"); // twist value to the left
        int other = cur_perm[pos - 1];
        swap(cur_perm[pos - 1], cur_perm[pos]);
        cur_perm[pos] *= -1;
        cur_perm[pos - 1] *= -1;
        inverse[abs(val)] = pos - 1;
        inverse[abs(other)] = pos;
        break;
      }
      case 5:  {
        visit(cur_perm, val, "t2r"); // twist value to the right
        int other = cur_perm[pos + 1];
        swap(cur_perm[pos + 1], cur_perm[pos]);
        cur_perm[pos] *= -1;
        cur_perm[pos + 1] *= -1;
        inverse[abs(val)] = pos + 1;
        inverse[abs(other)] = pos;
        break;
      }
    }
  }
  visit(cur_perm, 0, "o"); // the last permutation has no change
}

int main(){
  int n;
  scanf("%d", &n);

  auto algorithm1 = [n](int change){
    if(abs(change) <= n) return make_pair(n+1-abs(change), 3);
```

```
  else return make_pair(2*n+1-abs(change), (change > 0 ? 0 : 1));
};
loopless_signed_perm(n, 0, algorithm1, visit_print);

auto algorithm2 = [n](int change){
  if (abs(change) > n) return make_pair(2*n+1-abs(change), 3);
  else return make_pair(n+1-abs(change), (change > 0 ? 0 : 1));
};
loopless_signed_perm(n, 1, algorithm2, visit_print);

auto algorithm3 = [n](int change){
  if (abs(change)%2 == 1) return make_pair(n+1-(abs(change)+1)/2, 3);
  else return make_pair(n+1-(abs(change)+1)/2, (change > 0 ? 0 : 1));
};
loopless_signed_perm(n, 2, algorithm3, visit_print);

auto algorithm4 = [n](int change){
  if (abs(change)%2 == 0) return make_pair(n+1-(abs(change)+1)/2, 3);
  else return make_pair(n+1-(abs(change)+1)/2, (change > 0 ? 0 : 1));
};
loopless_signed_perm(n, 3, algorithm4, visit_print);

auto algorithm5 = [n](int change){
  if (abs(change) <= n) return make_pair(n+1-abs(change), 2);
  else return make_pair(2*n+1-abs(change), (change > 0 ? 0 : 1));
};
loopless_signed_perm(n, 0, algorithm5, visit_print);

auto algorithm6 = [n](int change){
  if (abs(change) > n) return make_pair(2*n+1-abs(change), 2);
  else return make_pair(n+1-abs(change), (change > 0 ? 0 : 1));
};
loopless_signed_perm(n, 1, algorithm6, visit_print);

auto algorithm7 = [n](int change){
  if (abs(change) <= n) return make_pair(n+1-abs(change), 3);
  else return make_pair(2*n+1-abs(change), (change > 0 ? 4 : 5));
};
loopless_signed_perm(n, 0, algorithm7, visit_print);

auto algorithm8 = [n](int change){
  if (abs(change) > n) return make_pair(2*n+1-abs(change), 3);
  else return make_pair(n+1-abs(change), (change > 0 ? 4 : 5));
};
```

```
  loopless_signed_perm(n, 1, algorithm8, visit_print);

  auto algorithm9 = [n](int change){
    if (abs(change) <= n) return make_pair(n+1-abs(change), 2);
    else return make_pair(2*n+1-abs(change), (change > 0 ? 4 : 5));
  };
  loopless_signed_perm(n, 0, algorithm9, visit_print);

  auto algorithm10 = [n](int change){
    if (abs(change) > n) return make_pair(2*n+1-abs(change), 2);
    else return make_pair(n+1-abs(change), (change > 0 ? 4 : 5));
  };
  loopless_signed_perm(n, 1, algorithm10, visit_print);

  auto algorithm11 = [n](int change){
    if (abs(change)%2 == 1) return make_pair(n+1-(abs(change)+1)/2, 3);
    else return make_pair(n+1-(abs(change)+1)/2, (change > 0 ? 4 : 5));
  };
  loopless_signed_perm(n, 2, algorithm11, visit_print);

  auto algorithm12 = [n](int change){
    if (abs(change)%2 == 0) return make_pair(n+1-(abs(change)+1)/2, 3);
    else return make_pair(n+1-(abs(change)+1)/2, (change > 0 ? 4 : 5));
  };
  loopless_signed_perm(n, 3, algorithm12, visit_print);
  return 0;
}
```

Figure A.3: `signedPerm.cpp` [46] generates all 12 Gray codes.