Towards the Efficient Generation of Gray Codes in the Bitprobe Model

by

Zachary Frenette

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2016

Abstract

We examine the problem of representing integers modulo L so that both increment and decrement operations can be performed efficiently. This problem is studied in the bitprobe model, where the complexity of the underlying problem is measured by the number of bit operations performed on the data structure [18]. In this thesis, we will primarily be interested in constructing space-optimal data structures. That is, we would like to use exactly n bits to represent integers modulo 2^n. Brodal et al. gave such a data structure, which requires $n-1$ bit reads and 3 bit writes, in the worst case, to perform increment and decrement operations [3]. We provide several improvements to their data structure. First, we give a data structure that requires $n-1$ bit reads and 2 bit writes, in the worst case, to perform increment and decrement operations. Then, we refine this result to obtain a data structure that requires $n-1$ bit reads and a single bit write to perform both operations. This disproves the conjecture that, when a space-optimal data structure uses only 1 bit write to perform these operations, then every bit in the data structure must be inspected in the worst case [10, 2].

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor J. Ian Munro, for his patience, knowledge, and support throughout the duration of my research. I would also like to thank Hicham for our many discussions, as well as Nicole for always lending an ear. Last but not least, I would like to thank Professor Naomi Nishimura and Professor Gordon Agnew for taking the time to read my thesis and providing feedback, when necessary.

Dedication

This thesis is dedicated to my family, for their endless support and encouragement.

Table of Contents

List of Figures

Glossary of Terms and Acronyms

BRGC	The acronym for Binary Reflected Gray Code.
Code	A counting sequence with space-efficiency $e = 1$.
Counter	A d-bit data structure that encodes integers modulo L, where $L \leq 2^d$.
Counting sequence	A cyclic sequence of L distinct d-bit integers.
DAT	The acronym for Decision Assignment Tree, which is used to describe the increment and decrement operations.
(d, e, r, w)-scheme	A description of the increment and decrement operations performed by a d-bit counter with space-efficiency e, which requires reading r bits and writing w bits in the worst case.
(d, r, w)-scheme	Abbreviated notation for a (d, e, r, w)-scheme with $e = 1$.
Gray code	A code induced by a (d, r, w)-scheme whenever $w = 1$.
Quasi-Gray code	A code induced by a (d, r, w)-scheme whenever $w \in \Theta(1)$.
Redundant counter	A counter with space-efficiency $e < 1$.
RPGC	The acronym for Recursive Partition Gray Code.
SBC	The acronym for Standard Binary Code.
Space-efficiency	The ratio $e = L/2^d$.
Space-optimal counter	A counter with space-efficiency $e = 1$.

Chapter 1

Introduction

We study the problem of designing data structures that efficiently represent integers modulo L. In particular, we would like to increment and decrement the integer using as few bit operations as possible. We analyze this problem in the bitprobe model, which was first introduced by Minsky and Papert in 1969 to discuss the average-case bitprobe complexity of the membership problem [18]. In this model, the cost of computation is measured purely by the number of bit operations performed on the data structure. That is, the number of bits read and the number of bits written are the characteristics that define the complexity of the underlying problem. In 1981, Yao generalized the bitprobe model to the cellprobe model [29], where the complexity of the underlying problem is measured by the number of reads and writes performed on w-bit cells. Although less powerful than the cellprobe model, the bitprobe model has been subjected to extensive study in computer science [8, 10, 17, 4]. In fact, many problems have been analyzed in the bitprobe model, including polynomial evaluation [11, 6], dynamic connectivity and dynamic partial sums [22], and the membership problem [23, 24, 27, 15, 12]. For an overview of other data structure problems in the bitprobe model, we refer the reader to the survey of Nicholson, Raman and Rao [19].

There are many applications which require data structures that can efficiently increment and decrement integers. For example, Pagh, Pagh and Rao used such a data structure to create a succinct representation of a dynamic multiset, allowing them to construct an optimal dynamic Bloom filter [21]. In another application, succinct representations of integers are used implicitly to create efficient representations of binomial queues – a notion first examined by Vuillemin [28]. Finally, by utilizing a redundant binary representation, Carlsson, Munro and Poblete devised an implicit binomial queue that allows constant time insertions in the worst case [5]. For other applications, we refer the reader to the works of Okasaki and Savage [20, 26].

1.1 Problem Definition and Notation

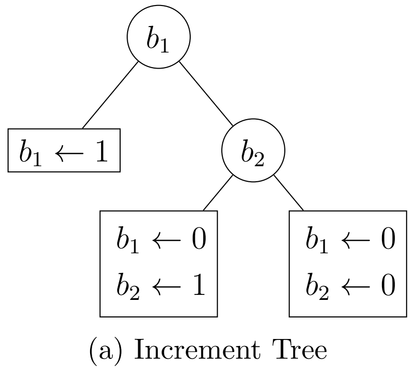
We define a counter as a d-bit data structure that encodes integers modulo L, where $L \leq 2^d$. In addition, we define a counting sequence $\mathcal{C} = \langle c_0, c_1, \ldots, c_{L-1} \rangle$ as a cyclic sequence of L distinct d-bit integers. For a fixed counting sequence, a counter must support the following two basic operations:

1. **Increment** modifies the state of the counter from c_i to c_{i+1}
2. **Decrement** modifies the state of the counter from c_i to c_{i-1}

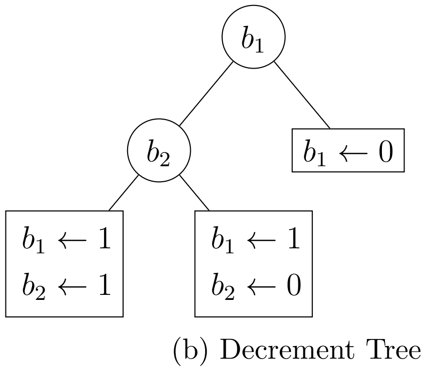
Note that both operations are performed modulo L. That is, incrementing c_{L-1} produces c_0, and likewise, decrementing c_0 produces c_{L-1}.

The ratio $e = L/2^d$ is called the space-efficiency of the counter. In particular, when $L = 2^d$, we say that the counter is space-optimal, and refer to the corresponding counting sequence as a code. On the other hand, when $L < 2^d$, we say that the counter is redundant. Note that in such a case, not all bit arrangements may correspond to valid integer values. When examining the overall efficiency of a counter, we will consider three different measures of complexity: the number of bit reads and writes performed in the worst case, the number of bit reads and writes performed on average, and the the space-efficiency of the counter. For this problem, the average number of bit reads is calculated by adding the total number of bits read during L increment (or decrement) operations, and then dividing this total by L. Likewise, the average number of bits written can be defined in an analogous fashion.

Given a d-bit counter with space-efficiency e, we define a (d, e, r, w)-scheme as a description of the increment and decrement operations, which are performed by reading r bits and writing w bits in the worst case. Whenever the space-efficiency of the counter is equal to 1, we will omit the second parameter for convenience. That is, we define a (d, r, w)-scheme as a (d, e, r, w)-scheme with $e = 1$. For example, the standard binary representation of integers (also called the Standard Binary Code) uses n bits to represent integers modulo 2^n. Since every bit needs to be read and written in the worst case, the Standard Binary Code gives an (n, n, n)-scheme. In this thesis, we will primarily be interested in a specific class of codes. In particular, we say that a (d, r, w)-scheme induces a Gray code $\mathcal{G}$ whenever $w = 1$ [13], and more generally, we call $\mathcal{G}$ a quasi-Gray code whenever $w \in \Theta(1)$ [2].



(a) Increment Tree



(b) Decrement Tree

Figure 1.1: DATs for incrementing and decrementing the SBC on 2-bit integers

1.2 Decision Assignment Tree Model

The algorithms for incrementing and decrementing integer counters will be presented in the Decision Assignment Tree (DAT) model. First introduced by Fredman [10], a DAT is a binary tree with the following characteristics. Each internal node is labelled by the location of a particular bit in the data structure, while each leaf node contains assignment statements that describe which bits in the data structure are updated. The increment (or decrement) algorithm begins by reading the bit specified by the label at the root node. Then depending on the value of the bit read, the algorithm moves to either the left child or the right child of that node. More precisely, if the algorithm reads a bit having the value 0, it will move to the left child of the current node. On the other hand, if the algorithm reads a bit having the value 1, it will move to the right child of the current node. This procedure is applied recursively until the algorithm arrives at a leaf node, where it finally performs the required updates to the data structure. Figure 1.1 shows how to generate the Standard Binary Code (SBC) on 2-bit integers of the form b_2b_1. In particular, the DAT on the left shows how to perform increment operations, while the DAT on the right shows how to perform decrement operations.

Analyzing the complexity of a scheme under this model can be done in a straightforward manner. In particular, the number of bits read in the worst case is determined by the height of the DAT, while the number of bits written in the worst case is indicated by the leaf with the largest number of assignment statements. To calculate the average number of bits read, we sum over the length of every root-to-leaf path in the DAT, multiplied by the fraction of times each path is used during the generation of the counting sequence. Similarly, to calculate the average number of bits written, we sum over the number of assignment statements in every leaf, multiplied by the fraction of times each leaf is reached during the generation of the counting sequence. For example, we can see that the increment tree in Figure 1.1 has a height of 2, and that the largest number of assignment statements in a single leaf is also 2. This allows us to conclude that the scheme performs 2 bit reads and 2 bit writes in the worst case. Furthermore, we observe that two of the four bit strings in the counting sequence require a single bit read, while the other two bit strings require both bits to be read. Likewise, the same observation can be made regarding the number of bits written. From this, we conclude that the scheme requires, on average, $\frac{2}{4} \cdot 1 + \frac{2}{4} \cdot 2 = 1.5$ bit reads and bit writes to perform an increment operation. The same analysis can be carried out on the decrement tree.

1.3 Thesis Outline

The remaining chapters of this thesis are organized as follows. In chapter 2, we survey some of the previous results pertaining to integer counters. In chapter 3, we explore several lower bounds regarding the number of bit reads required to perform increment and decrement operations. In chapter 4, we describe the details of an $(n, n-1, 3)$-scheme [3]. In chapter 5, we improve the write complexity of this scheme by reducing the number of bit writes required from 3 to 2. We then further improve this result by reducing the number of bit writes required down to 1. That is, we present an n-bit Gray code that requires, in the worst case, $n-1$ bit reads to perform increment and decrement operations. Lastly, we finish with some concluding remarks and then briefly discuss potential directions for future work.

Chapter 2

Survey of Previous Work

Succinct representations of integers have been extensively studied in computer science, and in this chapter, we survey some of the key results.

2.1 Space-Optimal Counters

The SBC uses n bits to represent integers modulo 2^n, resulting in an (n, n, n)-scheme that requires all n bits to be read and written in the worst case. However, the SBC requires only $2 - 2^{1-n}$ bit reads and bit writes on average to perform increment (or decrement) operations. In 1953, Frank Gray patented a code now known as the Binary Reflected Gray Code (BRGC) while working on methods for converting analog signals into digital signals [13, 14]. Unlike the SBC, the BRGC requires merely a single bit write to generate the next element in the code, and can be defined recursively as follows. Let $\mathcal{G}_n$ denote the BRGC on n bits, and let $\mathcal{G}_n^R$ denote the code $\mathcal{G}_n$ in reverse order. For $n = 1$, the BRGC is defined as $\mathcal{G}_1 = \langle 0, 1 \rangle$, while for $n \geq 2$, we have

$$\mathcal{G}_n = \langle 0\mathcal{G}_{n-1}, 1\mathcal{G}_{n-1}^R \rangle.$$

Given $b \in \{0, 1\}$, we use $b\mathcal{G}$ to denote a sequence where b is prefixed to every bit string in $\mathcal{G}$. Therefore, prefixing a 0 to every bit string in $\mathcal{G}_{n-1}$ generates the first half of $\mathcal{G}_n$, while prefixing a 1 to every bit string in $\mathcal{G}_{n-1}^R$ generates the second half of $\mathcal{G}_n$. An example of the BRGC for $3 \leq n \leq 5$ is provided in Figure 2.1. To perform an increment operation, we have two cases to consider. If the current state of the counter has even parity, then it suffices to flip the rightmost bit. On the other hand, if current state of the counter has odd

000	0000	1100	00000	01100	11000	10100
001	0001	1101	00001	01101	11001	10101
011	0011	1111	00011	01111	11011	10111
010	0010	1110	00010	01110	11010	10110
110	0110	1010	00110	01010	11110	10010
111	0111	1011	00111	01011	11111	10011
101	0101	1001	00101	01001	11101	10001
100	0100	1000	00100	01000	11100	10000

Figure 2.1: The BRGC on 3, 4, and 5 bits

parity, then we find the rightmost 1 and flip the bit to its left. Decrement operations can be performed in a similar manner. In particular, if the current state of the counter has odd parity, then it suffices to flip the rightmost bit. However, if current state of the counter has even parity, then we find the rightmost 1 and flip the bit to its left. Although the problem of generating the BRGC has been subjected to extensive study [7, 1, 14], Fredman showed that any DAT which induces $\mathcal{G}_n$ must read all n bits in the worst case [10]. Furthermore, since every bit must be inspected to determine the parity of a bit string, this procedure for incrementing and decrementing $\mathcal{G}_n$ requires an average of n bit reads.

In 2010, Bose et al. introduced a different Gray code called the Recursive Partition Gray Code (RPGC) [2]. By using a divide and conquer approach, they were able to create an n-bit counter that requires $O(\log n)$ bit reads on average to perform increment and decrement operations, even though generating the code requires n bit reads in the worst case. [1] Let $\mathcal{R}_n$ denote the RPGC on n bits. When $n = 1$, the RPGC is defined as $\mathcal{R}_1 = \langle 0, 1\rangle$, and when $n \geq 2$, the increment and decrement operations are defined in a mutually recursive manner. For simplicity, assume that n is a power of 2. To perform increment and decrement operations, the current state of the counter is divided into two substrings, X and Y, which we interpret as bit strings from the sequence $\mathcal{R}_{n/2}$. When performing an increment operation, we recursively increment X unless $X = Y$, at which point we recursively decrement Y. Similarly, when performing a decrement operation, we recursively decrement X unless $X = Y + 1$, at which point we recursively increment Y. Figure 2.2 provides an example of $\mathcal{R}_2$ and $\mathcal{R}_4$. In addition to this result, Bose et al. utilized the RPGC as a building block to construct DATs for other space-optimal counters [2]. They show that for sufficiently large n and any integer $c \geq 1$, there exists an (n, n, c)-scheme which generates a quasi-Gray code requiring, on average, no more than $O(\log^{(2c-1)} n)$ bit reads to perform increment operations. Furthermore, by picking $c \in O(\log^* n)$, they construct a DAT for

[1]Logarithms are taken to the base 2.

00	0000	1010	1111	0101
01	0010	1011	1101	0100
11	0110	0011	1001	1100
10	1110	0111	0001	1000

Figure 2.2: The RPGC on 2 and 4 bits

an $(n, n, O(\log^* n))$-scheme that reads at most 17 bits on average to perform increment operations. Although these two results have incredibly efficient increment operations, we note that they do not efficiently support decrement operations.

All of the counters seen so far require n bit reads in the worst case. However, through an exhaustive search, Brodal et al. were able to find a $(4, 3, 2)$-scheme, and then use it to construct an $(n, n-1, 3)$-scheme which supports both increment and decrement operations [3]. Furthermore, their construction uses the RPGC to represent the lower-order bits of the data structure, allowing their counter to read, on average, no more than $O(\log n)$ bits. The results described in this section are summarized in Figure 2.3.

2.2 Redundant Counters

By utilizing redundancy, it is often possible to improve the number of bit reads required by a scheme. In particular, by using $O(n)$ bits of extra space, Fredman provided a data structure that requires only $O(\log n)$ bit reads and a single bit write to perform increment operations [10]. Rahman and Munro later improved this result by giving an $(n + 1, 1/2, \log n + 4, 4)$-scheme that supports both increment and decrement operations [25]. Their data structure divides the counter into three blocks. The first block is a status bit that indicates whether a delayed bit write is required, while the other two blocks are used to store a variation of the BRGC [16, 25]. These two blocks are of size $\log n$ and $n - \log n$ respectively. When performing increment (or decrement) operations, the writes required in the smaller block are applied immediately. However, the writes required in the larger block are applied during subsequent operations, at which point the status bit is used to keep track of this information.

Similar ideas were used by Brodal et al. to give an $(n + 1, 1/2, \log n + 2, 3)$-scheme that supports both increment and decrement operations [3]. However, in their data structure, the smaller block stores the RPGC while the larger block stores the SBC. To perform increment operations, the smaller block is incremented until it overflows, at which point the status bit is used to indicate a delayed propagation of the carry. During subsequent increment operations, the value of the smaller block is used as a pointer to a particular bit within the larger block, implicitly indicating the location of the current carry. In addition to this result, a trade-off is shown between the space-efficiency of the scheme and the number of bit reads required in the worst case. In particular, for an arbitrary integer $1 \leq t \leq n - \log n - 1$, a variety of data structures parametrized by t are given, each with a space-efficiency close to one [3]. For counters with efficient average-case complexity, Bose et al. give an $(n + t \log n, 1 - O(n^{-t}), O(t \log n), 2c + 1)$-scheme for arbitrary parameters $c \geq 1$ and $t > 0$ [2]. In addition to having a space-efficiency close to one, their scheme requires, on average, no more than $O(\log^{(2c)} n)$ bit reads.

Prior to these results, this problem was studied by Frandsen, Miltersen and Skyum within the context of dynamic language membership problems [9]. In particular, they give an $(n + \log n, 2/n - O(2^{-n}), \log n + 1, \log n + 1)$-scheme supporting both increment and decrement operations. However, as n becomes larger, the space-efficiency of their scheme approaches zero. The results of this section are summarized in Figure 2.4.

Figure 2.3: Summary of previous results for space-optimal counters

Dimension	Bits read		Bits written	Inc. & Dec.	Reference
	Worst Case	Average Case	Worst Case		
n	n	$2-2^{1-n}$	n	Y	SBC
n	n	n	1	Y	Gray [13]
n	n	$6\log n$	1	Y	Bose et al. [2]
n	n	$O(\log^{(2c-1)} n)$	c	N	
n	n	17	$O(\log^* n)$	N	
4	3	3	2	Y	Brodal et al. [3]
n	$n-1$	$O(\log n)$	3	Y	

Figure 2.4: Summary of previous results for redundant counters

Dimension	Efficiency	Bits read		Bits written	Inc. & Dec.	Reference
		Worst Case	Average Case	Worst Case		
$O(n)$	$1/2^{O(n)}$	$O(\log n)$	$O(\log n)$	1	N	Fredman [10]
$n+\log n$	$2/n-O(2^{-n})$	$\log n+1$	3	$\log n+1$	Y	Frandsen et al. [9]
$n+t\log n$	$1-O(n^{-t})$	$O(t\log n)$	$O(\log^{(2c)} n)$	$2c+1$	N	Bose et al. [2]
$n+1$	$1/2$	$\log n+4$	$O(1)$	4	Y	Rahman et al. [25]
$n+t$	$\geq 1-1/2^t$	$\log n+t+1$	$O(\log\log n)$	3	N	Brodal et al. [3]
$n+t$	$\geq 1-1/2^t$	$\log n+t+2$	$O(\log\log n)$	2	N	
$n+t$	$\geq 1-1/2^t$	$\log n+t+3$	$O(\log\log n)$	1	N	
$n+t$	$\geq 1-1/2^t$	$\log n+t+2$	$O(\log\log n)$	3	Y	
$n+t$	$\geq 1-1/2^t$	$\log n+t+3$	$O(\log\log n)$	2	Y	

Chapter 3

Lower Bounds

In this chapter, we explore several lower bounds regarding the number of bit reads required to perform increment and decrement operations. In particular, to generate integers modulo 2^n, it can be shown that $r \in \Omega(\log n)$ whenever $e = 1$ or $w = 1$ [10, 19]. Furthermore, we will show that, on average, no space-optimal counter can perform better than the SBC. That is, any (n, r, w)-scheme requires at least $2 - 2^{1-n}$ bit reads on average to perform increment or decrement operations.

3.1 Worst-Case Lower Bounds

For this first lower bound, information-theoretic arguments are used to show that any $(d, e, r, 1)$-scheme requires at least a logarithmic number of bit reads in the worst case [10]. In particular, to obtain a bound on the height the corresponding DAT, it suffices to obtain a bound on the number of leaves in the tree. This notion is formalized with the following theorem.

Theorem 3.1 ([10]). *Every $(d, e, r, 1)$-scheme that generates integers modulo L requires at least $r \geq \log\log L + 1$ bit reads in the worst case to perform increment (or decrement) operations.*

Proof. Let $\mathcal{T}$ be a DAT which performs increment operations for a $(d, e, r, 1)$-scheme. Since $\mathcal{T}$ generates integers modulo L, at least $\log L$ of the d bits in the data structure need to be modified. In particular, each of these bits need to be modified at least twice: once when being set to 1, and once when being set to 0. Therefore, since leaves can only contain one assignment statement, the number of leaves in $\mathcal{T}$ must be at least $2\log L$. Furthermore, since a DAT of height r has at most 2^r leaves, it follows that $2\log L \leq 2^r$. Hence, we have $r \geq \log\log L + 1$, as required. □

To facilitate the discussion of the next lower bound, we begin by defining some notation. Let $\mathcal{T}$ be a DAT for a (d, e, r, w)-scheme and let l be a leaf node of $\mathcal{T}$. First, we define W_l to be the set of bits that appear in the assignment statements of l. Similarly, let R_l denote the set of bits that must be read in order to reach l. The following lemma shows that, in a space-optimal counter, we can only update bits that have been previously read. In particular, if there exists a leaf l such that $W_l \not\subseteq R_l$, then $\mathcal{T}$ cannot possibly induce a valid code. Moreover, this lemma will be the crux of the argument used in the proof of Theorem 3.3.

Lemma 3.2. *Let $\mathcal{T}$ be a DAT which performs increment (or decrement) operations for an n-bit space-optimal counter. Then for any leaf node l of $\mathcal{T}$, we have $W_l \subseteq R_l$.*

Proof. Assume by way of contradiction that there exists a leaf l and a bit b such that $b \in W_l$ but $b \notin R_l$. Since $b \notin R_l$, there must exist a pair of integers $X = x_n x_{n-1} \ldots x_1$ and $Y = y_n y_{n-1} \ldots y_1$ that reach l when they are incremented. Moreover, this pair of integers differs only at the position indicated by b. As a result, X and Y must be incremented to the same integer, which contradicts the definition of a space-optimal counter. □

Theorem 3.3 ([19])**.** *Every (n, r, w)-scheme requires at least $r \geq \log n + 1$ bit reads in the worst case to perform increment (or decrement) operations.*

Proof. Assume by way of contradiction that there exists an (n, r, w)-scheme that induces a code $\mathcal{C}$ such that $r \leq \log n$, and let $\mathcal{T}$ denote the corresponding DAT which performs increment operations. Since $\mathcal{T}$ has height r, it follows that $\mathcal{T}$ must have at most $2^r - 1 \leq 2^{\log n} - 1 = n - 1$ internal nodes. Thus, there exists a bit b in the data structure that is never read during any of the increment operations. Since the space-efficiency of the counter is equal to one, there must exist a leaf l such that $b \in W_l$ but $b \notin R_l$. Consequently, we have $W_l \not\subseteq R_l$, which, by Lemma 3.2, contradicts the fact that $\mathcal{C}$ is a code. As a result, it follows that $r \geq \log n + 1$, as required. □

3.2 Average-Case Lower Bounds

In this section, we prove a lower bound on the average number of bit reads required to perform increment (or decrement) operations on an n-bit code. In particular, we show that any (n, r, w)-scheme requires at least $\bar{r} \geq 2 - 2^{1-n}$ bit reads on average to increment an n-bit integer, which implies that the SBC is optimal in this regard. To show this, we will argue that any DAT achieving an average of $2 - 2^{1-n}$ bit reads must have a particular structure. More precisely, for every internal node in the levels 1 through $n - 1$, exactly one of its two children must be a leaf. Consequently, any deviation from this structure either induces an invalid code, or increases the number of bit reads performed. We will formalize this notion with the following theorem.

Theorem 3.4. *Every (n, r, w)-scheme requires at least $\bar{r} \geq 2 - 2^{1-n}$ bit reads on average to perform increment (or decrement) operations.*

Proof. Let $\mathcal{T}$ be a DAT for an (n, r, w)-scheme that requires an average of $\bar{r}$ bit reads to perform increment operations, and let z_1 denote the first bit inspected by the increment algorithm. Observe that each internal node in $\mathcal{T}$ must have two children, each of which is either a leaf node, or another internal node. Hence, when analyzing the underlying structure of $\mathcal{T}$, we have three cases to consider. First, suppose that both children of the root are leaf nodes. Then, by Lemma 3.2, $\mathcal{T}$ cannot induce a valid code for any value of n larger than one. On the other hand, suppose that both children of the root are internal nodes. In this case, the increment algorithm will read a second bit, regardless of the value of z_1. Hence, we have $\bar{r} \geq 2 \geq 2 - 2^{1-n}$. In the third case, the algorithm will only read another bit whenever $z_1 = b_1$, for some $b_1 \in \{0, 1\}$. That is, when $z_1 = b_1$, the algorithm will probe a second bit, and when $z_1 = 1 - b_1$, the algorithm performs the required bit writes to the data structure. In this case, the expected number of bit reads is bounded by $\bar{r} \geq \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 2 = \frac{3}{2}$, which is the only case where we can hope to do better than $2 - 2^{1-n}$.

We now apply this argument inductively. Suppose that the internal node at level $k \leq n - 1$ reads bit z_k. We again have three cases to consider. In the first case, when both children are leaf nodes, at most $k \leq n - 1$ of the n bits are ever read. Hence, by Lemma 3.2, $\mathcal{T}$ cannot induce a valid code. In the second case, if both children are internal nodes, then $\bar{r}$ becomes too large. In particular, the leaf at depth $1 \leq j \leq k - 1$ is reached 2^{n-j} times, while the internal node at level k is reached 2^{n-k+1} times. Therefore, by using the identity $\sum_{j=1}^{k-1} j/2^j = 2 - (k + 1)2^{1-k}$, we have

$$\begin{aligned} \bar{r} &\geq \sum_{j=1}^{k-1} j/2^j + (k + 1)2^{1-k} \\ &= 2 - (k + 1)2^{1-k} + (k + 1)2^{1-k} \\ &= 2. \end{aligned}$$

In the third case, the algorithm will only read another bit whenever $z_k = b_k$, for some $b_k \in \{0, 1\}$. That is, when $z_k = b_k$, the algorithm will probe a $(k + 1)$-th bit, and when $z_k = 1 - b_k$, the algorithm performs the required bit writes to the data structure. In this case, the expected number of bit reads is bounded by

$$\begin{aligned}\bar{r} &\geq \sum_{j=1}^{k-1} j/2^j + k2^{-k} + (k+1)2^{-k} \\ &= 2 - (k+1)2^{1-k} + k2^{-k} + (k+1)2^{-k} \\ &= 2 - 2^{-k}.\end{aligned}$$

Finally, when $k = n$, there are no new bits for the algorithm to read. Therefore, both children of the nth internal node must be leaf nodes, and so, the average number of bit reads required does not increase. Hence, we have $\bar{r} \geq 2 - 2^{1-n}$, as required. □

Chapter 4

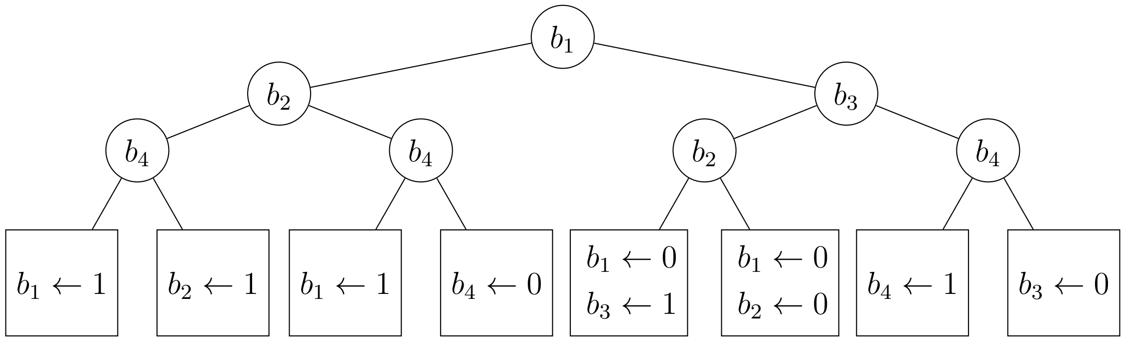
Efficiently Generating Quasi-Gray Codes

In 2014, Brodal et al. presented the first space-optimal counter that, in the worst case, requires fewer than n bit reads to perform increment and decrement operations [3]. Since their result forms the basis of our contributions in this thesis, we cover it extensively in this chapter. In particular, by utilizing a $(4, 3, 2)$-scheme, they devised an $(n, n-1, 3)$-scheme for any $n \geq 5$. Their $(4, 3, 2)$-scheme was obtained through an exhaustive search. Assuming the counter is of the form $b_4b_3b_2b_1$, the increment and decrement trees are shown in Figure 4.1. We will denote these trees by $\mathcal{T}_{(4,3,2)}$ and $\mathcal{T}'_{(4,3,2)}$ respectively. For example, to increment the counter from 0001 to 0100, the algorithm reads the bits b_1, b_3 and b_2, and then updates the bits $b_1 \leftarrow 0$ and $b_3 \leftarrow 1$. Likewise, to decrement the counter from 0001 to 0000, the algorithm reads the bits b_1, b_3 and b_4, and then updates the bit $b_1 \leftarrow 0$.

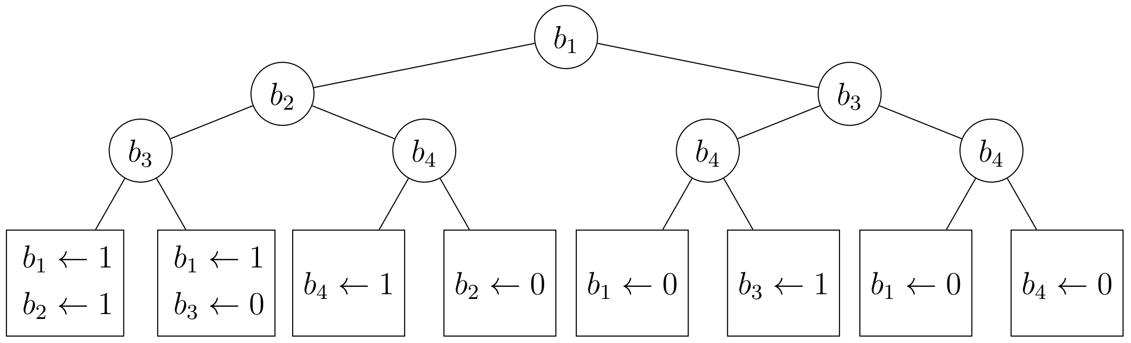
Theorem 4.1 ([3])**.** *There is a* $(4, 3, 2)$*-scheme that supports both increment and decrement operations.*

For larger values of n, the representation of the counter is divided into two blocks $B_{(4,3,2)}$ and $B_{\mathcal{G}}$ of size 4 and $n - 4$ respectively. In particular, $B_{(4,3,2)}$ uses the $(4, 3, 2)$-scheme outlined in Theorem 4.1, while $B_{\mathcal{G}}$ uses the representation of a Gray code. This structure is illustrated in Figure 4.2. To perform increment operations, we first increment $B_{\mathcal{G}}$ and then check if it represents 0. If it does, then we also increment $B_{(4,3,2)}$. Similarly, to perform decrement operations, we decrement $B_{\mathcal{G}}$, and if $B_{\mathcal{G}}$ was 0 before being decremented, then we also decrement $B_{(4,3,2)}$. We will formalize this notion with the following theorem.

Theorem 4.2 ([3])**.** *For* $n \geq 5$*, there exists an* $(n, n-1, 3)$*-scheme that supports both increment and decrement operations.*



(a) Increment Tree



(b) Decrement Tree

Figure 4.1: Increment and decrement trees for a $(4, 3, 2)$-scheme

Proof. Let $\mathcal{G}$ be an $(n-4)$-bit Gray code, and let $\mathcal{T}_{\mathcal{G}}$ and $\mathcal{T}'_{\mathcal{G}}$ denote the corresponding increment and decrement trees. Without loss of generality, we assume that every leaf in these two trees has depth $n-4$. Let l be the leaf in $\mathcal{T}_{\mathcal{G}}$ that corresponds to the integer that appears immediately before the string of all 0's. In addition, let $b_k \leftarrow 0$ denote the lone assignment statement in l. To construct the increment tree, which we denote by $\mathcal{T}$, we begin by taking $\mathcal{T}_{\mathcal{G}}$ and replacing l by a copy of $\mathcal{T}_{(4,3,2)}$. Moreover, we add the assignment statement $b_k \leftarrow 0$ to each of the leaves in $\mathcal{T}_{(4,3,2)}$. Therefore, $\mathcal{T}$ will initially update the first $n-4$ bits through 2^{n-4} states, according to the rules of $\mathcal{T}_{\mathcal{G}}$. However, when performing this final increment operation, $\mathcal{T}$ will also update the last 4 bits of the data structure, according to the rules of $\mathcal{T}_{(4,3,2)}$.

Performing decrement operations can be done in an analogous manner. Let l' be the leaf in $\mathcal{T}'_{\mathcal{G}}$ that corresponds to the string of all 0's. To construct the decrement tree, which we denote by $\mathcal{T}'$, we take $\mathcal{T}'_{\mathcal{G}}$ and replace l' by a copy of $\mathcal{T}'_{(4,3,2)}$. Then, like in the construction

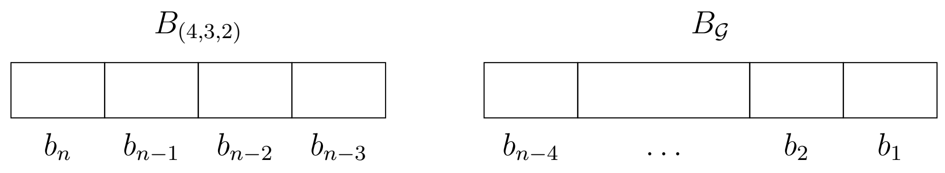


Figure 4.2: Structure of an $(n, n-1, 3)$-scheme

of $\mathcal{T}$, we add the assignment statement $b_k \leftarrow 1$ to each of the leaves in $\mathcal{T}'_{(4,3,2)}$. Therefore, $\mathcal{T}'$ will update the first $n-4$ bits according to the rules of $\mathcal{T}'_{\mathcal{G}}$. Moreover, when the first $n-4$ bits are all 0's, $\mathcal{T}'$ will also update the last 4 bits of the data structure, according to the rules of $\mathcal{T}'_{(4,3,2)}$. In terms of efficiency, both $\mathcal{T}$ and $\mathcal{T}'$ have height $n-1$. Furthermore, both DATs require 3 bit writes in the worst case: one when updating $B_{\mathcal{G}}$, and up to two more when updating $B_{(4,3,2)}$. Therefore, $\mathcal{T}$ and $\mathcal{T}'$ induce an $(n, n-1, 3)$-scheme, as required. $\square$

Note that, in our construction, $\mathcal{T}$ and $\mathcal{T}'$ require at least $n-4$ bit reads to perform increment and decrement operations. However, by choosing $\mathcal{G}$ to be the RPGC, a careful analysis will show that, on average, the increment and decrement operations can be performed with $O(\log n)$ bit inspections [2, 3].

Chapter 5

Efficiently Generating Gray Codes

In this chapter, we explore methods for reducing the write complexity of the $(n, n-1, 3)$-scheme presented in the previous chapter. First, we describe a method for generalizing existing schemes to larger values of n, without increasing the number of bit writes required. This result, in conjunction with Theorem 4.1, will allow us to construct an $(n, n-1, 2)$-scheme for all $n \geq 4$. We then improve this result by further reducing the number of bit writes required. By constructing an $(n, n-1, 1)$-scheme for all $n \geq 5$, we provide the first Gray code that requires, in the worst case, fewer than n bit reads to perform increment and decrement operations. Finally, we also summarize the results of our exhaustive search for schemes on small values of n.

5.1 Constructing an $(n, n-1, 2)$-Scheme

Let $\mathcal{G}$ be an n-bit Gray code and let π be a permutation of $[n]$. [1] We use the notation $\pi(\mathcal{G})$ to denote the sequence obtained by applying π to every element in $\mathcal{G}$. That is, for every element $X = x_n x_{n-1} \ldots x_1$ in $\mathcal{G}$, we let $\pi(X) = x_{\pi(n)} x_{\pi(n-1)} \ldots x_{\pi(1)}$. Similarly, for any bijection $\varphi : \{0,1\}^n \to \{0,1\}^n$, we use the notation $\varphi(\mathcal{G})$ to denote the sequence obtained by applying φ to each element in $\mathcal{G}$. Lastly, we use the notation $\oplus$ to denote the exclusive-or operator. The following lemmas will lead the reader to the main result of this section, which is Theorem 5.4.

Lemma 5.1. *Let $\mathcal{G}$ be an n-bit Gray code and let π be a permutation of $[n]$. Then $\pi(\mathcal{G})$ is also an n-bit Gray code.*

[1] The notation $[n]$ is used to denote the set $\{1, \ldots, n\}$.

Proof. Let $X = x_n x_{n-1} \dots x_1$ and $Y = y_n y_{n-1} \dots y_1$ be two consecutive elements in $\mathcal{G}$. Observe that $x_i = y_i$ implies that $x_{\pi(i)} = y_{\pi(i)}$. Likewise, $x_i \neq y_i$ implies that $x_{\pi(i)} \neq y_{\pi(i)}$. Hence $\pi(X)$ and $\pi(Y)$ have Hamming distance one, and so, $\pi(\mathcal{G})$ is an n-bit Gray code. $\square$

Lemma 5.2. *Let $\mathcal{G}$ be an n-bit Gray code, let $A = a_n a_{n-1} \dots a_1$ be any n-bit integer, and let $\varphi_A(Z) = Z \oplus A$. Then $\varphi_A(\mathcal{G})$ is also an n-bit Gray code.*

Proof. Let $X = x_n x_{n-1} \dots x_1$ and $Y = y_n y_{n-1} \dots y_1$ be two consecutive elements in $\mathcal{G}$. Observe that $x_i = y_i$ implies that $x_i \oplus a_i = y_i \oplus a_i$. Likewise, $x_i \neq y_i$ implies that $x_i \oplus a_i \neq y_i \oplus a_i$. Hence $\varphi_A(X)$ and $\varphi_A(Y)$ have Hamming distance one. Furthermore, since φ_A is bijective, $\varphi_A(\mathcal{G})$ contains 2^n distinct elements. Therefore, $\varphi_A(\mathcal{G})$ is an n-bit Gray code. $\square$

Lemma 5.3. *Let S and T be two distinct n-bit integers that have Hamming distance one. Then there exists an n-bit Gray code $\mathcal{G}_{ST}$ such that S and T appear consecutively in $\mathcal{G}_{ST}$.*

Proof. Let $\mathcal{G}$ be any n-bit Gray code, and let X and Y denote two consecutive elements in $\mathcal{G}$. Suppose that X and Y differ at position j and that S and T differ at position i. We will use $\mathcal{G}$ to construct $\mathcal{G}_{ST}$ as follows. First, we define a permutation π of $[n]$ such that $\pi(i) = j$ and $\pi(j) = i$. In addition, let $A = \pi(X) \oplus S$ and let $\varphi_A(Z) = Z \oplus A$. We claim that $\mathcal{G}_{ST} = \varphi_A(\pi(\mathcal{G}))$ is a Gray code satisfying the desired properties. In particular, by Lemmas 5.1 and 5.2, $\mathcal{G}_{ST}$ is indeed a Gray code, and so it suffices to verify that S and T appear consecutively in $\mathcal{G}_{ST}$. Since X and Y are consecutive elements in $\mathcal{G}$, we have

$$\begin{aligned} \varphi_A(\pi(X)) &= \pi(X) \oplus A \\ &= \pi(X) \oplus (\pi(X) \oplus S) \\ &= S \end{aligned}$$

and

$$\begin{aligned} \varphi_A(\pi(Y)) &= \pi(Y) \oplus A \\ &= \pi(Y) \oplus (\pi(X) \oplus S) \\ &= T \end{aligned}$$

are consecutive elements in $\mathcal{G}_{ST}$. Note that the second equation follows from the fact that $\pi(Y) \oplus \pi(X)$ is the n bit integer containing a single 1 at position j and 0's everywhere else. Therefore, $\mathcal{G}_{ST}$ is a Gray code that satisfies the desired properties. $\square$

Let $\mathcal{C}$ be an n-bit code and let $\mathcal{T}_\mathcal{C}$ be a DAT which generates $\mathcal{C}$ by using r bit reads and w bit writes. Moreover, suppose that there exists a leaf l in $\mathcal{T}_\mathcal{C}$ such that $|W_l| = 1$. We

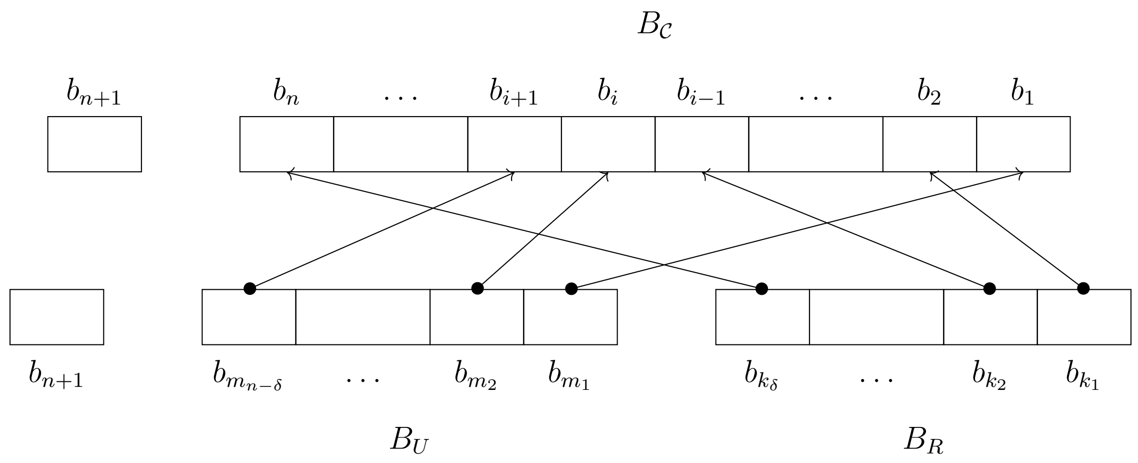


Figure 5.1: Conceptualization of Theorem 5.4

will now show that by using $\mathcal{C}$ and $\mathcal{T}_{\mathcal{C}}$ as building blocks, we can construct an $(n+1)$-bit code that can be generated by a DAT requiring $r+1$ bit reads and w bit writes. The representation of this $(n+1)$-bit code is divided into two blocks. The first block is a single bit b_{n+1}, while the second block, which we denote by $B_{\mathcal{C}}$, is used to store $\mathcal{C}$. Conceptually, it will also help to further divide $B_{\mathcal{C}}$ into two subblocks: B_R and B_U. In particular, B_R contains the $\delta \leq r$ bits that appear in R_l, while B_U contains the remaining $n-\delta$ bits. By utilizing this structure, which is illustrated in Figure 5.1, we can generate our $(n+1)$-bit code as follows.

Theorem 5.4. *Let $\mathcal{C}$ be an n-bit code and let $\mathcal{T}_{\mathcal{C}}$ be a DAT which, in the worst case, requires r bit reads and w bit writes to generate $\mathcal{C}$. If there exists a leaf l in $\mathcal{T}_{\mathcal{C}}$ such that $|W_l| = 1$, then there exists a DAT which, in the worst case, requires $r+1$ bit reads and w bit writes to generate an $(n+1)$-bit code.*

Proof. Let $T = t_\delta t_{\delta-1} \dots t_1$ denote the string that corresponds to the values of the bits in B_R when l is reached. Likewise, let $S = s_\delta s_{\delta-1} \dots s_1$ denote the string that corresponds to the values of the bits in B_R, immediately after applying the assignment statement in l. Note that S and T have Hamming distance one since $|W_l| = 1$. Therefore, by Lemma 5.3, there exists a δ-bit Gray code $\mathcal{G}_{ST}$ such that S and T appear consecutively in $\mathcal{G}_{ST}$. We will use the notation $\mathcal{T}_{ST}$ to denote the corresponding DAT. Moreover, without loss of generality, we assume that every leaf in $\mathcal{T}_{ST}$ has depth δ.

Now we construct a DAT, which is denoted by $\mathcal{T}$, that generates the desired $(n+1)$-bit code. First, we label the root node of $\mathcal{T}$ by b_{n+1}. Next, we define the left subtree of

$\mathcal{T}$ to be $\mathcal{T}_{\mathcal{C}}$, with the modification that l contains the assignment statement $b_{n+1} \leftarrow 1$. Finally, we define the right subtree of $\mathcal{T}$ to be $\mathcal{T}_{ST}$, with the modification that the leaf corresponding to S contains the assignment statement $b_{n+1} \leftarrow 0$. Therefore, to generate this $(n+1)$-bit code, we begin by reading b_{n+1}. If the value of b_{n+1} is 0, we increment B_C according to the rules of $\mathcal{T}_{\mathcal{C}}$, unless the values of the bits in B_R are equal to T. In this case, the algorithm performs the update $b_{n+1} \leftarrow 1$. On the other hand, if the value of b_{n+1} is 1, we increment B_R according the the rules of $\mathcal{T}_{ST}$. In particular, the algorithm continues to cycle through the elements of $\mathcal{G}_{ST}$ until the values of the bits in B_R are equal S. At this point, the algorithm performs the update $b_{n+1} \leftarrow 0$. Note that this occurs $2^{n-\delta}$ times, with each occurrence having a different value stored in B_U. Therefore, $\mathcal{T}$ generates $2^{n-\delta}2^{\delta} = 2^n$ distinct integers when b_{n+1} has a value of 1. Moreover, when b_{n+1} has a value of 0, $\mathcal{T}$ generates the 2^n distinct integers in $\mathcal{C}$. As a result, this construction induces a code with a total of 2^{n+1} distinct integers. In terms of efficiency, since $\delta \leq r$, the height of $\mathcal{T}$ is equal to $r+1$. Since the right subtree of $\mathcal{T}$ generates a Gray code, the maximum number of writes required remains w. $\square$

In order to generate a code that supports both increment and decrement operations, we introduce the following definitions. Let $\mathcal{T}$ and $\mathcal{T}'$ denote the increment and decrement trees for an (n, r, w)-scheme. Moreover, let $\mathcal{Z}_l$ denote the set of integers that reach l when incremented (or decremented). We say that the pair of leaves (l, l') is symmetric if the following two conditions hold:

- l is a leaf in $\mathcal{T}$ and l' is a leaf in $\mathcal{T}'$
- there exists a bijection $\psi : \mathcal{Z}_l \to \mathcal{Z}_{l'}$ such that $\psi(X) = Y$ if and only if X appears immediately before Y in the corresponding code

By utilizing this symmetry between leaves, we obtain the following theorem.

Theorem 5.5. *For $n \geq 4$, there exists an $(n, n-1, 2)$-scheme that supports both increment and decrement operations.*

Proof. When $n = 4$, the claim is established in Theorem 4.1. For larger values of n, we will need to inductively apply Theorem 5.4. Let $\mathcal{T}_{n-1}$ and $\mathcal{T}'_{n-1}$ denote the increment and decrement trees for an $(n-1, n-2, 2)$-scheme. To support both operations, we require a symmetric pair of leaves (l, l') such that $|W_l| = |W_{l'}| = 1$. Observe that when $n = 5$, the 4th and 3rd leaves in Figure 4.1 satisfy this requirement. We begin by applying Theorem 5.4 on $\mathcal{T}_{n-1}$ and l to obtain the increment tree for an $(n, n-1, 2)$-scheme, which is denoted by $\mathcal{T}_n$. Furthermore, let $\mathcal{G}_{ST}$ denote the δ-bit Gray code used in the construction of $\mathcal{T}_n$, and let $\mathcal{G}_{ST}^R$ denote the code $\mathcal{G}_{ST}$ in reverse order. Then, to construct the corresponding decrement tree $\mathcal{T}'_n$, it suffices to apply Theorem 5.4 using l' as the chosen leaf and $\mathcal{G}_{ST}^R$ as the chosen δ-bit Gray code. Moreover, this procedure for constructing $\mathcal{T}_n$ and $\mathcal{T}'_n$ guarantees the existence of another symmetric pair of leaves that can be used during the next application of Theorem 5.4. This follows from the fact that the bits of $\mathcal{G}_{ST}$ and $\mathcal{G}_{ST}^R$ are always read in the right subtree of $\mathcal{T}_n$ and $\mathcal{T}'_n$ respectively. Hence, $\mathcal{T}_n$ and $\mathcal{T}'_n$ induce an $(n, n-1, 2)$-scheme, as required. □

5.2 Constructing an $(n, n-1, 1)$-Scheme

In this section, we show how to transform the $(4, 3, 2)$-scheme, which is outlined in Figure 4.1, into a $(5, 4, 1)$-scheme. The main idea is to replace the 2-bit write transitions by multiple 1-bit write transitions. For example, consider the 6th leaf from the left in the increment tree. Currently, we have

$$(0011, 1011) \xrightarrow{b_1, b_2} (0000, 1000).$$

Here, values above the arrow denote the bits that are updated during the corresponding increment operation. Observe that since our scheme is space-optimal, we cannot directly replace this transition by

$$(0011, 1011) \xrightarrow{b_1} (0010, 1010) \xrightarrow{b_2} (0000, 1000).$$

This follows from the fact that these transitions, in conjunction with the original ones, would require both 0011 and 1010 to be incremented to 0010. In order to resolve this issue, we simply need to add an extra bit b_5, which instead gives the following transitions

$$(00011, 01011) \xrightarrow{b_5} (10011, 11011) \xrightarrow{b_1} (10010, 11010) \xrightarrow{b_2} (10000, 11000) \xrightarrow{b_5} (00000, 01000).$$

We can now apply the same idea to the 5th leaf in the increment tree. In particular, the resulting transitions are

$$\begin{aligned}(00001, 01001) &\xrightarrow{b_5} (10001, 11001) \xrightarrow{b_3} (10101, 11101) \xrightarrow{b_2} (10111, 11111) \\ &\xrightarrow{b_1} (10110, 11110) \xrightarrow{b_2} (10100, 11100) \xrightarrow{b_5} (00100, 01100).\end{aligned}$$

Note that two additional transitions are required in order to ensure that the counting sequence has no redundancy. Moreover, to support decrement operations, we simply need to reverse the transitions outlined above. The resulting increment and decrement trees for this $(5, 4, 1)$-scheme are presented in Figure 5.2.

Theorem 5.6. *There is a* $(5, 4, 1)$*-scheme that supports both increment and decrement operations.*

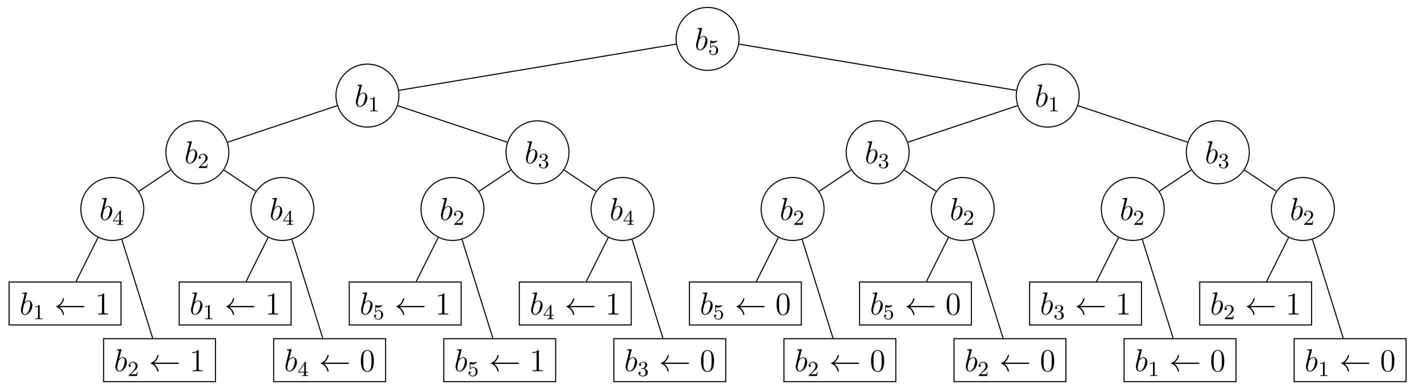
For larger values of n, we can apply the same proof strategy as Theorem 5.5.

Theorem 5.7. *For* $n \geq 5$*, there exists an* $(n, n-1, 1)$*-scheme that supports both increment and decrement operations.*

Proof. When $n = 5$, the claim is established in Theorem 5.6. For larger values of n, we will need to inductively apply Theorem 5.4. Let $\mathcal{T}_{n-1}$ and $\mathcal{T}'_{n-1}$ denote the increment and decrement trees for an $(n-1, n-2, 1)$-scheme. To support both operations, we require a symmetric pair of leaves (l, l') such that $|W_l| = |W_{l'}| = 1$. Observe that when $n = 6$, the 4th and 3rd leaves in Figure 5.2 satisfy this requirement. We begin by applying Theorem 5.4 on $\mathcal{T}_{n-1}$ and l to obtain the increment tree for an $(n, n-1, 1)$-scheme, which is denoted by $\mathcal{T}_n$. Furthermore, let $\mathcal{G}_{ST}$ denote the δ-bit Gray code used in the construction of $\mathcal{T}_n$, and let $\mathcal{G}^R_{ST}$ denote the code $\mathcal{G}_{ST}$ in reverse order. Then, to construct the corresponding decrement tree $\mathcal{T}'_n$, it suffices to apply Theorem 5.4 using l' as the chosen leaf and $\mathcal{G}^R_{ST}$ as the chosen δ-bit Gray code. Moreover, this procedure for constructing $\mathcal{T}_n$ and $\mathcal{T}'_n$ guarantees the existence of another symmetric pair of leaves that can be used during the next application of Theorem 5.4. This follows from the fact that the bits of $\mathcal{G}_{ST}$ and $\mathcal{G}^R_{ST}$ are always read in the right subtree of $\mathcal{T}_n$ and $\mathcal{T}'_n$ respectively. Hence, $\mathcal{T}_n$ and $\mathcal{T}'_n$ induce an $(n, n-1, 1)$-scheme, as required. $\square$

5.3 Exhaustive Search Results

In this section, we summarize the results of our exhaustive search for schemes on small values of n. In particular, these results are shown in Figure 5.3. We use the notation '+' to denote that a scheme with the corresponding values of n, r and w exists. Likewise, we use the notation '$-$' to denote that no scheme exists for the specified parameters. Moreover, we use '?' to denote that the existence of a (n, r, w)-scheme remains unknown, as the state space was much too large to search entirely. Observe that, by Lemma 3.2, we require that $W_l \subseteq R_l$ for every leaf l in a DAT. Therefore, we can automatically place a '$-$' for schemes where $w > r$. In addition, the existence of a $(6, 4, w)$-scheme remains unknown when $1 \leq w \leq 4$. Although we conjecture that such a scheme does not exist, we do believe that an $(n, n-2, w)$-scheme exists for larger values of n.


b_5
b_1 b_1
b_2 b_3 b_3 b_3
b_4 b_4 b_2 b_4 b_2 b_2 b_2 b_2
$b_1 \leftarrow 1$ $b_2 \leftarrow 1$ $b_1 \leftarrow 1$ $b_4 \leftarrow 0$ $b_5 \leftarrow 1$ $b_5 \leftarrow 1$ $b_4 \leftarrow 1$ $b_3 \leftarrow 0$ $b_5 \leftarrow 0$ $b_2 \leftarrow 0$ $b_5 \leftarrow 0$ $b_2 \leftarrow 0$ $b_3 \leftarrow 1$ $b_1 \leftarrow 0$ $b_2 \leftarrow 1$ $b_1 \leftarrow 0$
(a) Increment Tree

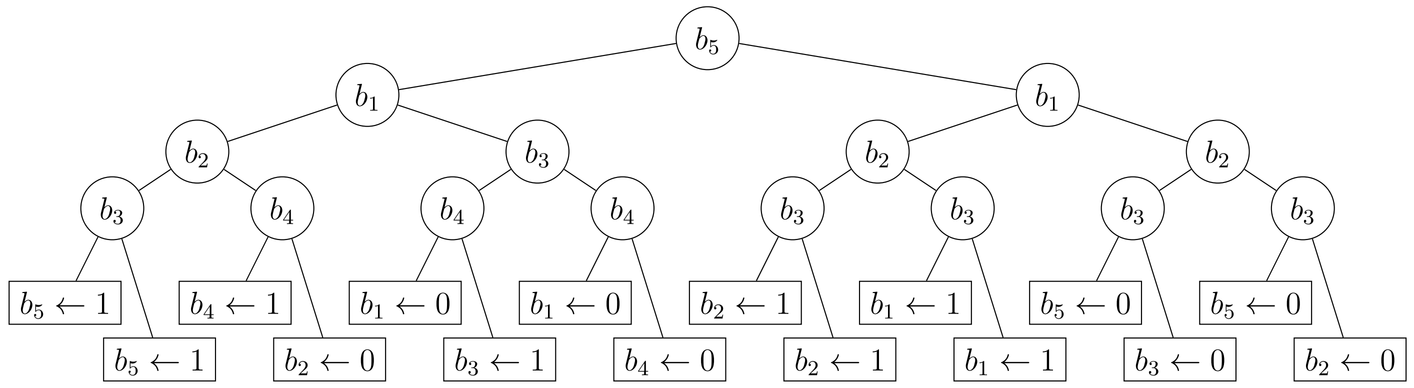

b_5
b_1 b_1
b_2 b_3 b_2 b_2
b_3 b_4 b_4 b_4 b_3 b_3 b_3 b_3
$b_5 \leftarrow 1$ $b_5 \leftarrow 1$ $b_4 \leftarrow 1$ $b_2 \leftarrow 0$ $b_1 \leftarrow 0$ $b_3 \leftarrow 1$ $b_1 \leftarrow 0$ $b_4 \leftarrow 0$ $b_2 \leftarrow 1$ $b_2 \leftarrow 1$ $b_1 \leftarrow 1$ $b_1 \leftarrow 1$ $b_5 \leftarrow 0$ $b_3 \leftarrow 0$ $b_5 \leftarrow 0$ $b_2 \leftarrow 0$
(b) Decrement Tree

Figure 5.2: Increment and decrement trees for a $(5, 4, 1)$-scheme

Figure 5.3: Exhaustive search results for $4 \leq n \leq 6$

$$\begin{array}{cc|cccc|}
 & & \multicolumn{4}{c}{r} \\
 & & 1 & 2 & 3 & 4 \\ \hline
 & 1 & - & - & - & + \\
 & 2 & - & - & + & + \\
w & 3 & - & - & + & + \\
 & 4 & - & - & - & + \\ \hline
\end{array}
\qquad
\begin{array}{cc|ccccc|}
 & & \multicolumn{5}{c}{r} \\
 & & 1 & 2 & 3 & 4 & 5 \\ \hline
 & 1 & - & - & - & + & + \\
 & 2 & - & - & - & + & + \\
w & 3 & - & - & - & + & + \\
 & 4 & - & - & - & + & + \\
 & 5 & - & - & - & - & + \\ \hline
\end{array}
\qquad
\begin{array}{cc|cccccc|}
 & & \multicolumn{6}{c}{r} \\
 & & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline
 & 1 & - & - & - & ? & + & + \\
 & 2 & - & - & - & ? & + & + \\
 & 3 & - & - & - & ? & + & + \\
w & 4 & - & - & - & ? & + & + \\
 & 5 & - & - & - & - & + & + \\
 & 6 & - & - & - & - & - & + \\ \hline
\end{array}$$

Chapter 6

Conclusions and Future Work

In this thesis, we first surveyed several of the previous results pertaining to integer counters. We then explored lower bounds regarding the number of bit reads required to perform increment and decrement operations. In particular, it was shown that $r \in \Omega(\log \log L)$ bit reads are necessary to generate integers modulo L, whenever $e = 1$ or $w = 1$ [10, 19]. Moreover, we argued that, on average, no (n, r, w)-scheme can require fewer than $2 - 2^{1-n}$ bit reads to perform increment and decrement operations, implying that the SBC is optimal in this regard. Then, we examined a space-optimal counter that requires, in the worst case, $n - 1$ bit reads and 3 bit writes to perform increment and decrement operations [3]. This counter was constructed from a $(4, 3, 2)$-scheme, which was found through an exhaustive search. Finally, we presented a technique for extending the $(4, 3, 2)$-scheme mentioned above, without having to increase the number of bit writes required. That is, for all $n \geq 4$, we established the existence of an $(n, n - 1, 2)$-scheme which supports increment and decrement operations. Furthermore, we showed how transform the above-mentioned $(4, 3, 2)$-scheme into a $(5, 4, 1)$-scheme. Then, by using this $(5, 4, 1)$-scheme, we designed an $(n, n - 1, 1)$-scheme for all $n \geq 5$. In other words, we constructed an n-bit Gray code that requires, in the worst case, $n - 1$ bit reads to perform increment and decrement operations. This construction disproves the conjecture that a DAT which generates an n-bit Gray code must have height n [10, 2].

In terms of future work, there are many avenues that one could explore, and so we conclude with a list of open problems.

1. Improve the read complexity of space-optimal counters. In particular, space-optimal counters are only known to exist for values of $r \geq n - 1$, though we conjecture that $(n, n - c, w)$-schemes exist, for any constant $c > 0$.

2. The RPGC is an n-bit code that requires, on average, $O(\log n)$ bit reads to perform increment and decrement operations [2]. Is it possible to design a Gray code that reads, on average, fewer bits than the RPGC?

3. The $(n, n-1, 1)$-scheme devised in chapter 5 requires, on average, $n-1$ bit reads to perform increment and decrement operations. Is there a Gray code that reads $n-1$ bits in the worst case, but only $O(\log n)$ bits on average?

4. Improve the lower bounds for space-optimal counters. In particular, to generate integers modulo 2^n, at least $\log n + 1$ bit reads are necessary. Can this lower bound be improved? Likewise, can one obtain an $\omega(1)$ lower bound for the average-case read complexity of quasi-Gray codes?

5. Several of the schemes surveyed in chapter 2 can only perform increment operations. Can these schemes be modified to also support decrement operations?

References

[1] James R. Bitner, Gideon Ehrlich, and Edward M. Reingold. Efficient generation of the binary reflected gray code and its applications. *Communications of the ACM*, 19(9):517 – 521, 1976.

[2] Prosenjit Bose, Paz Carmi, Dana Jansens, Anil Maheshwari, Pat Morin, and Michiel Smid. Improved methods for generating quasi-gray codes. In *Proceedings of the 12th Scandinavian Conference on Algorithm Theory*, SWAT '10, pages 224 – 235. Springer-Verlag, 2010.

[3] Gerth Stølting Brodal, Mark Greve, Vineet Pandey, and Srinivasa Rao Satti. Integer representations towards efficient counting in the bit probe model. *Journal of Discrete Algorithms*, 26:34 – 44, 2014.

[4] Harry Buhrman, Peter Bro Miltersen, Jaikumar Radhakrishnan, and Srinivasan Venkatesh. Are bitvectors optimal? *SIAM Journal on Computing*, 31(6):1723 – 1744, 2002.

[5] Svante Carlsson, J. Ian Munro, and Patricio V. Poblete. An implicit binomial queue with constant insertion time. In *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, SWAT '88, pages 1 – 13. Springer-Verlag, 1988.

[6] Victor Chen, Elena Grigorescu, and Ronald de Wolf. Efficient and error-correcting data structures for membership and polynomial evaluation. In *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science*, STACS '10, pages 203 – 214, 2010.

[7] Martin Cohn and Shimon Even. A gray code counter. *IEEE Transactions on Computers*, 18(7):662 – 664, 1969.

[8] Peter Elias and Richard A. Flower. The complexity of some simple retrieval problems. *J. ACM*, 22(3):367 – 379, 1975.

[9] Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *J. ACM*, 44(2):257 – 271, 1997.

[10] Michael L. Fredman. Observations on the complexity of generating quasi-gray codes. *SIAM Journal on Computing*, 7(2):134 – 146, 1978.

[11] Anna Gál and Peter Bro Miltersen. The cell probe complexity of succinct data structures. *Theoretical Computer Science*, 379(3):405 – 417, 2007.

[12] Mohit Garg and Jaikumar Radhakrishnan. Set membership with a few bit probes. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 776 – 784, 2015.

[13] Frank Gray. Pulse code communications. U.S. Patent 2632058, 1953.

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

[15] Moshe Lewenstein, J. Ian Munro, Patrick K. Nicholson, and Venkatesh Raman. Improved explicit data structures in the bitprobe model. In *Algorithms - ESA 2014*, volume 8737, pages 630 – 641. Springer Berlin Heidelberg, 2014.

[16] Harold M. Lucal. Arithmetic operations for digital computers using a modified reflected binary code. *IEEE Trans. Comput.*, pages 449 – 458, 1959.

[17] Peter Bro Miltersen. The bit probe complexity measure revisited. In *Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '93, pages 662 – 671. Springer-Verlag, 1993.

[18] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry.* MIT Press, 1969.

[19] Patrick K. Nicholson, Venkatesh Raman, and S. Srinivasa Rao. A survey of data structures in the bitprobe model. In *Space-Efficient Data Structures, Streams, and Algorithms*, volume 8066, pages 303 – 318. Springer Berlin Heidelberg, 2013.

[20] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[21] Anna Pagh, Rasmus Pagh, and S. Srinivasa Rao. An optimal bloom filter replacement. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pages 823 – 829, 2005.

[22] Mihai Pătraşcu and Corina E. Tarniţă. On dynamic bit-probe complexity. *Theoretical Computer Science*, 380(1-2):127 – 142, 2007.

[23] Jaikumar Radhakrishnan, Venkatesh Raman, and S. Srinivasa Rao. Explicit deterministic constructions for membership in the bitprobe model. In *Algorithms - ESA 2001*, volume 2161, pages 290 – 299. Springer Berlin Heidelberg, 2001.

[24] Jaikumar Radhakrishnan, Smit Shah, and Saswata Shannigrahi. Data structures for storing small sets in the bitprobe model. In *Algorithms - ESA 2010*, volume 6347, pages 159 – 170. Springer-Verlag, 2010.

[25] M. Ziaur Rahman and J. Ian Munro. Integer representation and counting in the bit probe model. *Algorithmica*, 56(1):105 – 127, 2010.

[26] Carla Savage. A survey of combinatorial gray codes. *SIAM Review*, 39(4):605–629, 1997.

[27] Emanuele Viola. Bit-probe lower bounds for succinct data structures. *SIAM Journal on Computing*, 41(6):1593 – 1604, 2012.

[28] Jean Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM*, 21(4):309 – 315, 1978.

[29] Andrew C. Yao. Should tables be sorted? *J. ACM*, 28(3):615 – 628, 1981.