Improved Methods For Generating Quasi-Gray Codes[⋆]

Prosenjit Bose,[1] Paz Carmi,[2] Dana Jansens,[1] Anil Maheshwari,[1] Pat Morin,[1] and Michiel Smid[1]

[1] Carleton University, Ottawa ON, Canada
[2] Ben-Gurion University of the Negev

Abstract. Consider a sequence of bit strings of length d, such that each string differs from the next in a constant number of bits. We call this sequence a quasi-Gray code. We examine the problem of efficiently generating such codes, by considering the number of bits read and written at each generating step, the average number of bits read while generating the entire code, and the number of strings generated in the code. Our results give a trade-off between these constraints, and present algorithms that do less work on average than previous results, and that increase the number of bit strings generated.

Key words: Gray codes, quasi-Gray codes, decision trees, counting, combinatorial generation

1 Introduction

We are interested in efficiently generating a sequence of bit strings. The class of bit strings we wish to generate are cyclic quasi-Gray codes. A *Gray code* [3] is a sequence of bit strings, such that any two consecutive strings differ in exactly one bit. We use the term *quasi-Gray code* [2] to refer to a sequence of bit strings where any two consecutive strings differ in at most c bits, where c is a constant defined for the code. A Gray code (quasi-Gray code) is called *cyclic* if the first and last generated bit strings also differ in at most 1 bit (c bits).

We say a bit string that contains d bits has *dimension* d, and are interested in efficient algorithms to generate a sequence of bit strings that form a quasi-Gray code of dimension d. After generating a bit string, we say the algorithm's data structure corresponds exactly to the generated bit string, and it's *state* is the bit string itself. In this way, we restrict an algorithm's data structure to using exactly d bits. At each step, the input to the algorithm will be the previously generated bit string, which is the algorithm's previous state. The output will be a new bit string that corresponds to the state of the algorithm's data structure.

The number of consecutive unique bit strings generated is equal to the number of consecutive unique states for the generating data structure, and we call this value L, the *length* of the generated code. Clearly $L \leq 2^d$. We define the *space*

[⋆] Research supported by NSERC.

efficiency of an algorithm as the ratio $L/2^d$, that is, the fraction of bit strings generated out of all possible bit strings given the dimension of the strings. When the space efficiency is 1 we call the data structure *space-optimal*, as it generates all possible bit strings. When $L < 2^d$ the structure is non-space-optimal.

We are concerned with the efficiency of our algorithms in the following ways. First, we would like to know how many bits the algorithm must read in the worst case in order to make the appropriate changes in the input string and generate the next bit string in the code. Second, we would like to know how many bits must change in the worst case to reach the successor string in the code, which must be a constant value to be considered a quasi-Gray code. Third, we examine the average number of bits read at each generating step. And last, we would like our algorithms to be as space efficient as possible, ideally generating as many bit strings as their dimension allows, with $L = 2^d$. Our results give a trade-off between these different goals.

Our decision to limit the algorithm's data structure to exactly d bits differs from previous work, where the data structure could use more bits than the strings it generated [2, 5]. To compare previous results to our own, we consider the extra bits in their data structure to be a part of their generated bit strings. This gives a more precise view of the space efficiency of an algorithm.

Each generated bit string of dimension d has a distinct totally ordered rank in the generated code. Given a string of rank k in a code of length L, where $0 \leq k < L$, we want to generate the bit string of rank $(k + 1) \mod L$.

We work within the bit probe model [4, 5], counting the average-case and the worst-case number of bits read and written for each bit string generated. We use the Decision Assignment Tree (DAT) model [2] to construct algorithms for generating quasi-Gray codes and describe the algorithms' behaviour, as well as to discuss upper and lower bounds.

We use a notation for the iterated log function of the form $\log^{(c)} n$ where c is a non-negative whole number, and is always surrounded by brackets to differentiate it from an exponent. The value of the function is defined as follows. When $c = 0$, $\log^{(c)} n = n$. If $c > 0$, then $\log^{(c)}(n) = \log^{(c-1)}(\log(n))$. We define the function $\log^* n$ to be equal to the smallest non-negative value of c such that $\log^{(c)} n \leq 1$. Throughout, the base of the log function is assumed to be 2 unless stated otherwise.

1.1 Results Summary

Our results, as well as previous results, are summarized in Table 1.

First, we present some space-optimal algorithms. Although our space-optimal algorithms read a small number of bits in the average case, they all read d bits in the worst case. In Section 3.1, we describe the Recursive Partition Gray Code (RPGC) algorithm, which generates a Gray code of dimension d while reading on average no more than $O(\log d)$ bits. This improves the average number of bits read for a space-optimal Gray code from d to $O(\log d)$. In Section 3.2, we use the RPGC to construct a DAT that generates a quasi-Gray code while reducing the average number of bits read. We then apply this technique iteratively in

	Space	Bits Read		Bits Written	
Dimension	Efficiency	Average	Worst-Case	Worst-Case	Reference
d	1	$2 - 2^{1-d}$	d	d	folklore
d	1	d	d	1	[2, 3]
d	1	$O(\log d)$	d	1	Theorem 2
d	1	$O(\log^{(2c-1)} d)$	d	c	Theorem 3
d	1	17	d	$O(\log^* d)$	Corollary 1
$n+1$	$1/2$	$O(1)$	$\log n + 4$	4	[5]
$n + \log n$	$O(n^{-1})$	3	$\log n + 1$	$\log n + 1$	[6]
$n + \log n + 1$	$1/2 + O(n^{-1})$	4	$\log n + 1$	$\log n + 1$	[1]
$n + O(t \log n)$	$1 - O(n^{-t})$	$O(1)$	$O(t \log n)$	$O(t \log n)$	Theorem 4
$n + O(t \log n)$	$1 - O(n^{-t})$	$O(t \log n)$	$O(t \log n)$	3	Theorem 5
$n + O(t \log n)$	$1 - O(n^{-t})$	$O(\log^{(2c)} n)$	$O(t \log n)$	$2c + 1$	Theorem 6

Table 1: Summary of results. When "Worst-Case Bits Written" is a constant then the resulting code is a quasi-Gray code, and when it is 1, the code is a Gray code. $c \in \mathbb{Z}$ and t are constants greater than 0.

Section 3.3 to create a d-dimensional DAT that reads on average $O(\log^{(2c-1)} d)$ bits, and writes at most c bits, for any constant $c \geq 1$. In section 3.4 we create a d-dimensional space-optimal DAT that reads at most 17 bits on average, and writes at most $O(\log^* d)$ bits. This reduces the average number of bits read to $O(1)$ for a space-optimal code, but increases the number of bits written to be slightly more than a constant.

Next, we consider quasi-Gray codes that are not space-optimal, but achieve space efficiency arbitrarily close to 1, and that read $O(\log d)$ bits in the worst case. In Section 3.5 we construct a DAT of dimension $d = n + O(t \log n)$ that reads and writes $O(t \log n)$ bits in the worst case, $O(1)$ on average, and has space efficiency $1 - O(n^{-t})$, for a constant $t > 0$. This improves the space efficiency dramatically of previous results where the worst-case number of bits written is $O(\log n)$. By combining a Gray code with this result, we produce a DAT of dimension $d = n + O(t \log n)$ that reads $O(t \log n)$ bits on average and in the worst case, but writes at most 3 bits. This reduces the worst-case number of bits written from $O(\log n)$ to $O(1)$. We then combine results from Section 3.3 to produce a DAT of dimension $d = n + O(t \log n)$ that reads on average $O(\log^{(2c)} n)$ bits, and writes at most $2c + 1$ bits, for any constant $c \geq 1$. This reduces the average number of bits read from $O(t \log d)$ to $O(\log \log d)$ when writing the same number of bits, and for each extra bits written, the average is further reduced by a logarithmic factor.

2 Decision Assignment Trees

In the Decision Assignment Tree (DAT) model, an algorithm is described as a binary tree. We use the DAT model to analyze algorithms for generating bit strings in a quasi-Gray code. We say that a DAT which reads and generates bit strings of length d has *dimension* d. Further, we refer to the bit string that the DAT reads and modifies as the *state* of the DAT between executions. Generally the initial state for a DAT will be the bit string 000...0.

Let T be a DAT of dimension d. Each internal node of T is labeled with a value $0 \leq i \leq d-1$ that represents reading bit i of the input bit string. The algorithm starts at the root of T, and reads the bit with which that node is labeled. Then it moves to a left or right child of that node, depending on whether the bit read was a 0 or a 1, respectively. This repeats recursively until a leaf node in the tree is reached. Each leaf node of T represents a subset of states where the bits read along the path to the leaf are in a fixed state. Each leaf node contains rules that describe which bits to update to generate the next bit string in the code. And each rule must set a single fixed bit directly to 0 or 1.

Under this model, we can measure the number of bits read to generate a bit string by the depth of the tree's leaves. We may use the average depth, weighted by the number of states in the generated code that reach each leaf, to describe the average number of bits read, and the tree's height to measure the worst-case number of bits read. The number of bits written can be measured by counting the rules in each leaf of the tree. Average and worst-case values for these can be found similarly. A trivial DAT, such as iterating through the standard binary representations of 0 to $2^d - 1$, in the worst case, will require reading and writing all d bits to generate the next bit string, but it may also read and write as few as one bit when the least-significant bit changes. On average, it reads and writes $2 - 2^{1-d}$ bits. Meanwhile, it is possible to create a DAT [2] that generates the Binary Reflected Gray Code [3]. This DAT would always write exactly one bit, but requires reading all d bits to generate the next bit string. This is because the least-siginificant bit is flipped if and only if the parity is even, which can only be determined by reading all d bits.

To generate a Gray code of dimension d with length $L = 2^d$, Fredman [2] shows that any DAT will require reading $\Omega(\log d)$ bits for some bit string. Fredman conjectures that for a Gray code of dimension d with $L = 2^d$, any DAT must read all d bits to generate at least one bit string in the code. That is, any DAT generating the code must have height d. This remains an open problem.[3]

3 Efficient generation of quasi-Gray codes

In this section we will address how to efficiently generate cyclic quasi-Gray codes of dimension d. First we present DATs that read up to d bits in the worst case, but read fewer bits on average. Then we present our lazy counters that read at

[3] In [5] the authors claim to have proven this conjecture true for "small" d by exhaustive search.

most $O(\log d)$ bits in the worst case and also read fewer bits on average, but with slightly reduced space-efficiency.

3.1 Recursive Partition Gray Code (RPGC)

We show a method for generating a cyclic Gray code of dimension d that requires reading an average of $6 \log d$ bits to generate each successive bit string. First, assume that d is a power of two for simplicity. We use both an increment and decrement operation to generate the gray code, where these operations generate the next and previous bit strings in the code, respectively. Both increment and decrement operations are defined recursively, and they make use of each other. Pseudocode for these operations is provided in Algorithm 1 and 2.

Algorithm 1: RecurIncrement

```
Input: $b[]$, an array of $n$ bits
if $n = 1$ then
    if $b[0] = 1$ then $b[0] \leftarrow 0$;
    else $b[0] \leftarrow 1$;
else
    Let $A = b[0...n/2 - 1]$;
    Let $B = b[n/2...n - 1]$;
    if $A = B$ then
        $RecurDecrement(B)$;
    else
        $RecurIncrement(A)$;
    end
end
```

Algorithm 2: RecurDecrement

```
Input: $b[]$, an array of $n$ bits
if $n = 1$ then
    if $b[0] = 1$ then $b[0] \leftarrow 0$;
    else $b[0] \leftarrow 1$;
else
    Let $A = b[0...n/2 - 1]$;
    Let $B = b[n/2...n - 1]$;
    if $A = B + 1$ then
        $RecurIncrement(B)$;
    else
        $RecurDecrement(A)$;
    end
end
```

To perform an increment, we partition the bit string of dimension d into two substrings, A and B, each of dimension $d/2$. We then recursively increment A unless $A = B$, that is, unless the bits in A are in the same state as the bits in B, at which point we recursively decrement B.

To perform a decrement, we again partition a bit string of dimension d into two substrings, A and B, each of dimension $d/2$. We then recursively decrement A unless $A = B + 1$, that is, the bits of A are in the same state as the bits of B would be after they were incremented, at which time we recursively increment B instead.

Theorem 1. *Let $d \geq 2$ be a power of two. There exists a space-optimal DAT that generates a Gray code of dimension d, where generating the next bit string requires reading on average no more than $4 \log d$ bits. In the worst case, d bits are read, and only 1 bit is written.*

Proof. The length L of the code is equal to the number of steps it takes to reach the initial state again. Because B is decremented if and only if $A = B$, then for each decrement of B, A will always be incremented $2^{d/2}-1$ times in order to reach the state $A = B$ again. Thus the total number of states is the number of times B is decremented, plus the number of times A is incremented and $L = 2^{d/2} + (2^{d/2})(2^{d/2}-1) = 2^d$. After L steps, the algorithm will output the bit string that was its initial input, creating a cyclic Gray code.

Since the RPGC has length $L = 2^d$, it is space-optimal, and the algorithm will be executed once for each possible bit string of dimension d. As such, we bound the average number of bits read by studying the expected number of bits read given a random bit string of dimension d. The proof is by induction on d. For the base case $d = 2$, in the worst case we read at most 2 bits, so the average bits read is at most $2 \le 4\log d$. Then we assume it is true for all random bit strings $X \in \{0,1\}^{d/2}$.

We define $|X|$ to denote the dimension of the bit string X. Let $C(A,B)$ be the number of bits read to determine whether or not $A = B$, where A and B are bit strings and $|A| = |B|$. Let $I(X)$ be the number of bits read to increment the bit string X. Let $D(X)$ be the number of bits read to decrement the bit string X. Note that since we are working in the DAT model, we read any bit at most one time, and $D(X) \le |X|$.

To finish the proof, we need to show that $\mathrm{E}[I(X)] \le 4\log d$, when $X \in \{0,1\}^d$ is a random bit strings. We can determine the expected value of $C(A,B)$ as follows. $C(A,B)$ must read two bits at a time, one from each of A and B, and compare them, only until it finds a pair that differs. Given two random bit strings, the probability that bit i is the first bit that differs between the two strings is $1/2^i$. If the two strings differ in bit i, then the function will read exactly i bits in each string. If $|A| = |B| = n$, then the expected value of $C(A,B)$ is $\mathrm{E}[C(A,B)] = 2\sum_{i=1}^{n} i/2^i = (2^{n+1}-n-2)/2^{n-1} = 4-(n+2)/2^{n-1}$.

Let $X = AB$, and $|A| = |B| = d/2$. Then $|X| = d$. For a predicate P, we define $\mathbf{1}_P$ to be the indicator random variable whose value is 1 when P is true, and 0 otherwise. Note that $I(A)$ is independent of $\mathbf{1}_{A=B}$ and $\mathbf{1}_{A\neq B}$. This is because the relation between A and B has no effect on the distribution of A (which remains uniform over $\{0,1\}^{d/2}$).

The `RecurIncrement` operation only performs one increment or decrement action, depending on the condition $A = B$, thus the expected number of bits read by $I(X)$ is $\mathrm{E}[I(X)] = \mathrm{E}[C(A,B)] + \mathrm{E}[\mathbf{1}_{A=B}D(B)] + \mathrm{E}[\mathbf{1}_{A\neq B}I(A)] \le 4-(n+2)/2^{n-1} + (1/2^{d/2})(d/2) + (1-1/2^{d/2})\mathrm{E}[I(A)] \le 4-(d/2+4)/2^{d/2} + 4\log(d/2) \le 4\log d$, as required.

The RPGC algorithm must be modified slightly to handle cases where d is not a power of two. We prove the following result in the full version of this paper.

Theorem 2. *Let $d \ge 2$. There exists a space-optimal DAT that generates a Gray code of dimension d, where generating the next bit string requires reading on average no more than $6\log d$ bits. In the worst case, d bits are read, and only 1 bit is written.*

3.2 Composite quasi-Gray code construction

Lemma 1. *Let $d \geq 1$, $r \geq 3$, $w \geq 1$ be integers. Assume we have a space-optimal DAT for a quasi-Gray code of dimension d such that the following holds: Given a bit string of length d, generating the next bit string in the quasi-Gray code requires reading no more than r bits on average, and writing at most w bits in the worst case.*

Then there is a space-optimal DAT for a quasi-Gray code of dimension $d + \lceil \log r \rceil$, where generating each bit string requires reading at most $6 \log \lceil \log r \rceil + 3$ bits on average, and writing at most $w + 1$ bits. That is, the average number of bits read decreases from r to $O(\log \log r)$, while the worst-case number of bits written increases by 1.

Proof. We are given a DAT A that generates a quasi-Gray code of dimension d. We construct a DAT B for the RPGC of dimension d', as described in Section 3.1. B will read $O(\log d')$ bits on average, and writes only 1 bit in the worst case.

We construct a new DAT using A and B. The combined DAT generates bit strings of dimension $d + d'$. The last d' bits of the combined code, when updated, will cycle through the quasi-Gray code generated by B. The first d bits, when updated, will cycle through the code generated by A.

The DAT initially moves the last d' bits through $2^{d'}$ states according to the rules of B. When it leaves this final state, to generate the initial bit string of B again, the DAT also moves the first d bits to their next state according to the rules of A. During each generating step, the last d' bits are read and moved to their next state in the code generated by the rules of B, and checked to see if they have reached their initial position, which requires $6 \log d' + 2$ bits to be read on average and 1 bit to be written. However, the first d bits are only read and written when the last d' bits cycle back to their initial state - once for every $2^{d'}$ bit strings generated by the combined DAT.

If we let $d' = \lceil \log r \rceil$, then the RPGC B has dimension at least 2. Let r' be the average number of bits read by the d'-code. Then the average number of bits read by the combined quasi-Gray code is no more than $r' + 2 + r/2^{d'} \leq 6 \log d' + 2 + r/2^{\lceil \log r \rceil} \leq 6 \log \lceil \log r \rceil + 3$.

The number of bits written, in the worst case, is the number of bits written in DAT A and in DAT B together, which is at most $w + 1$.

3.3 RPGC-Composite quasi-Gray Code

We are able to use the RPGC from Theorem 2 with our Composite quasi-Gray code from Lemma 1 to construct a space-optimal DAT that generates a quasi-Gray code. By applying Lemma 1 to the RPGC, and then repeatedly applying it $c - 1$ more times to the resulting DAT, we create a DAT that generates a quasi-Gray code while reading on average no more than $6 \log^{(2c-1)} d + 14$ bits, and writing at most c bits to generate each bit string, for any constant $c \geq 1$.

Theorem 3. *Given integers d and $c \geq 1$, such that $\log^{(2c-1)} d \geq 14$. There exists a DAT of dimension d that generates a quasi-Gray code of length $L = 2^d$,* *where generating the next bit string requires reading on average no more than* $6\log^{(2c-1)} d + 14$ *bits and writing in the worst case at most* c *bits.*

3.4 Reading a constant average number of bits

From Theorem 3, by taking c to be $O(\log^* d)$, it immediately follows that we can create a space-optimal DAT that reads a constant number of bits on average. This is a trade off, as the DAT requires writing at most $O(\log^* d)$ in the worst case, meaning the code generated by this DAT is no longer a quasi-Gray code.

Corollary 1. *For any integer* $d > 24 + 2^{16}$*, there exists a space-optimal DAT of dimension* d *that reads at most* 17 *bits on average, and writes no more than* $\lfloor(\log^* d + 3)/2\rfloor$ *bits in the worst case.*

Proof. Let $d > 2^{16}$, and $c = \lfloor(\log^* d - 3)/2\rfloor$. Then $c \geq 1$ and $\log^{(2c-1)} d \geq 14$ and it follows from Theorem 3 that there exists a space-optimal DAT of dimension d that reads on average at most $6\log^{(\log^* d-4)} d + 14 \leq 6 \cdot 2^{16} + 14$ bits, and writes at most $\lfloor(\log^* d - 1)/2\rfloor$ bits.

Let $d > 19 + 2^{16}$, and $m = d - 19 > 2^{16}$ Use the DAT from our previous statement with Lemma 1, setting $r = 6 \cdot 2^{16} + 14$ and $w = \lfloor(\log^* d - 1)/2\rfloor$. Then there exists a DAT of dimension $m + \lceil\log r\rceil = m + 19 = d$, that reads on average at most $6\log\lceil\log r\rceil + 3 \leq 29$ bits and writes no more than $\lfloor(\log^* d + 1)/2\rfloor$ bits.

If we apply this same technique again, we create a DAT of dimension $d > 24 + 2^{16}$ that reads on average at most 17 bits and writes no more than $\lfloor(\log^* d + 3)/2\rfloor$ bits.

3.5 Lazy counters

A lazy counter is a structure for generating a sequence of bit strings. In the first n bits, it counts through the standard binary representations of 0 to $2^n - 1$. However, this can require updating up to n bits, so an additional data structure is added to slow down these updates, making it so that each successive state requires fewer bit changes to be reached. We present a few known lazy counters, and then improve upon them, using our results to generate quasi-Gray codes.

Frandsen *et al.* [6] describe a lazy counter of dimension d that reads and writes at most $O(\log n) \leq O(\log d)$ bits for an increment operation. The algorithm uses $d = n + \log n$ bits, where the first n are referred to as b, and the last $\log n$ are referred to as i. A state in this counter is the concatenation of b and i, thus each state generates a bit string of dimension d. In the initial state, all bits in b and i are set to 0. The counter then moves through $2^{n+1} - 2$ states before cycling back to the initial state, generating a cyclic code.

The bits of b move through the standard binary numbers. However, moving from one such number to the next may require writing as many as n bits. The value in i is a pointer into b. For a standard binary encoding, the algorithm to move from one number to the next is as follows: starting at the right-most (least significant) bit, for each 1 bit, flip it to a 0 and move left. When a 0 bit is found, flip it to a 1 and stop. Thus the number of bit flips required to reach the next standard binary number is equal to one plus the position of the right-most 0. This counter simply uses i as a pointer into b such that it can flip a single 1 to a 0 each increment step until i points to a 0, at which point it flips the 0 to a 1, resets i to 0, and b has then reached the next standard binary number. The pseudocode is given in Algorithm 3.

Algorithm 3: LazyIncrement [6]

```
Input: $b[]$: an array of $n$ bits; $i$: an integer of $\log n$ bits
if $b[i] = 1$ then
    $b[i] \leftarrow 0$;
    $i \leftarrow i + 1$;
else
    $b[i] \leftarrow 1$;
    $i \leftarrow 0$;
end
```

Lemma 2. *[6] There exists a DAT of dimension $d = n + \log n$, using the* `LazyIncrement` *algorithm, that generates $2^n - 1$ of a possible $n2^{n-1}$ bit strings, where in the limit $n \to \infty$ the space efficiency drops to 0. The DAT reads and writes in the worst case $\log n + 1$ bits to generate each successive bit string, and on average reads and writes 3 bits.*

An observation by Brodal [1] leads to a dramatic improvement in space efficiency over the previous algorithm by adding a single bit to the counter. This extra bit allows for the $\log n$ bits in i to spin through all their possible values, thus making better use of the bits and generating more bit strings with them. The variables b and i are unchanged from the counter in Lemma 2, and k is a single bit, making the counter have dimension $d = n + \log n + 1$. The pseudocode is given in Algorithm 4.

Algorithm 4: SpinIncrement [1]

```
Input: $b[]$: an array of $n$ bits; $i$: an integer of $\log n$ bits; $k$: a single bit
if $k = 0$ then
    $i \leftarrow i + 1$ ; // spin $i$
    if $i = 0$ then $k \leftarrow 1$ ; // the value of $i$ has rolled over
else
    LazyIncrement($b[]$, $i$) ; // really increment the counter
    if $i = 0$ then $k \leftarrow 0$;
end
```

Lemma 3. *[1] There exists a DAT of dimension $d = n + \log n + 1$, using the* ***SpinIncrement*** *algorithm, that generates $(n+1)(2^n - 1)$ of a possible $2n2^n$ bit strings, where in the limit $n \to \infty$ the space efficiency converges to $1/2$. The DAT reads and writes in the worst case $\log n + 2$ bits to generate each successive bit string, and on average reads at most 4 bits.*

By generalizing the dimension of k, we are able to make the counter even more space efficient while keeping its $O(\log n) = O(\log d)$ worst-case bound for bits written and read. Let k be a bit array of dimension g, where $1 \leq g \leq t \log n$ and $t > 0$. Then for a counter of dimension $d = n + \log n + g$, the new algorithm is given by Algorithm 5.

Algorithm 5: DoubleSpinIncrement

```
Input: $b[]$: an array of $n$ bits; $i$: an integer of $\log n$ bits; $k$: an integer of $g$ bits
1 if $k < 2^g - 1$ then
2     $i \leftarrow i + 1$ ; // happens in $(2^g - 1)/2^g$ of the cases
3     if $i = 0$ then  $k \leftarrow k + 1$;
4 else
5     LazyIncrement($b[]$, $i$) ; // do a real increment
6     if $i = 0$ then  $k \leftarrow 0$;
7 end
```

Theorem 4. *There exists a DAT of dimension $d = n + \log n + g$, where $1 \leq g \leq t \log n$ and $t > 0$, with space efficiency $1 - O(2^{-g})$. The DAT, using the* ***DoubleSpinIncrement*** *algorithm, reads and writes in the worst case $g + \log n + 1$ bits to generate each successive bit string, and on average reads and writes $O(1)$ bits.*

Proof. This counter generates an additional $2^g - 1$ states for each time it spins through the possible values of i. Thus the number of bit strings generated is $(2^g - 1)n(2^n - 1) + 2^n - 1$. Given the dimension of the counter, the possible number of bit strings generated is $n2^n2^g$. When $g = 1$, we have exactly the same counter as given by Lemma 3. If $g > O(\log n)$, the worst-case number of bits read would increase. When $g = t \log n$, the space efficiency of this counter is $\dfrac{(n^{t+1} - n)(2^n - 1) + 2^n - 1}{n^{t+1}2^n} = 1 - O(n^{-t})$. Thus as n gets large, this counter becomes more space efficient, and is space-optimal as $n \to \infty$.

In the worst case, this counter reads and writes every bit in i and k, and a single bit in b, thus $g + \log n + 1 \leq (t+1)\log n + 1$ bits. On average, the counter now reads and writes $O(1)$ bits. This follows from a similar argument to that made for Lemma 3, where each line modified in the algorithm still reads on average $O(1)$ bits.

Rahman and Munro [5] present a counter that reads at most $\log n + 4$ bits and writes at most 4 bits to perform an increment or decrement operation. The counter uses $n+1$ bits to count through 2^n states, and has space efficiency $2^n/2^{n+1} = 1/2$. Compared to `DoubleSpinIncrement`, their counter writes fewer bits per generating step, but is less space efficient. By modifying our lazy counter to use Gray codes internally, the worst-case number of bits read remains asymptotically equivalent to the counter by Rahman and Munro, and the average number of bits we read increases. We are able to write a smaller constant number of bits per increment and retain a space efficiency arbitrarily close to 1.

We modify our counter in Theorem 4 to make i and k hold a cyclic Gray code instead of a standard binary number. The BRGC is a suitable Gray code for this purpose. Let $rank(j)$ be a function that returns the rank of the bit string j in the BRGC, and $next(j)$ be a function that returns a bit string k where $rank(k) = rank(j) + 1$. Algorithm 6 provides a lazy counter of dimension $d = n + \log n + g$, where $1 \leq g \leq t \log n$ and $t > 0$, that writes at most 3 bits, reads at most $g + \log n + 1$ bits to generate the next state, and has space efficiency arbitrarily close to 1.

Algorithm 6: WineIncrement[4]

```
Input: b[]: an array of n bits; i: a Gray code of log n bits; k: a Gray code of g
       bits
if $k \neq 100...00$ then
    $i \leftarrow$ next($i$) ; // happens in $(2^g - 1)/2^g$ of the cases
    if $i = 0$ then  $k \leftarrow$ next($k$);
else
    if $b[rank(i)] = 1$ then
        $b$[rank($i$)] $\leftarrow 0$;
        $i \leftarrow$ next($i$);
        if $i = 0$ then  $k \leftarrow 0$ ; // wraps around to the initial state
    else
        $b$[rank($i$)] $\leftarrow 1$;
        $k \leftarrow 0$ ; // resets $k$ to 0
    end
end
```

Theorem 5. *There exists a DAT of dimension $d = n + \log n + g$, where $1 \leq g \leq t \log n$ and $t > 0$, with space efficiency $1 - O(2^{-g})$. The DAT, using the `WineIncrement` algorithm, reads in the worst case $g + \log n + 1$ bits and writes in the worst case 3 bits to generate each successive bit string, and on average reads at most $O(\log n + g)$ bits.*

[4] The name `WineIncrement` comes from the Caribbean dance known as Wineing, a dance that is centered on rotating your hips with the music. The dance is pervasive in Caribbean culture, and has been popularized elsewhere through songs and music videos such as Alison Hinds' "Roll It Gal", Destra Garcia's "I Dare You", and Fay-Ann Lyons' "Start Wineing".

While the previous counter reads at most $g + \log n + 1$ bits in the worst case, its average number of bits read is also $O(\log n)$. Using the quasi-Gray code counter from Theorem 3, we are able to bring the average number of bits read down as well. The worst case number of bits read remains $g + \log n + 1$, but on average, we only read at most $12 \log^{(2c)} n + O(1)$ bits, for any $c \geq 1$.

The algorithm does not need to change from its fourth iteration for these modifications. We simply make i a quasi-Gray code from Theorem 3 of dimension $\log n$ and k a similar quasi-Gray code of dimension $g \leq t \log n$, for a $t > 0$.

Theorem 6. *Let n be such that $\log^{(2c)} n \geq 14$ and g be such that $\log^{(2c-1)} g \geq 14$, for $1 \leq g \leq t \log n$ and $t > 0$. Then for any $c \geq 1$, there exists a DAT of dimension $d = n + \log n + g$ bits, using the `WineIncrement` algorithm, with space efficiency $1 - O(2^{-g})$, that reads in the worst case $g + \log n + 1$ bits, writes in the worst case $2c + 1$ bits, and reads on average no more than $12 \log^{(2c)} n + O(1)$ bits.*

4 Conclusion

We have shown in this paper how to generate a Gray code, while reading siginificantly fewer bits on average than previously known algorithms, and how to efficiently generate a quasi-Gray code with the same worst-case performance and improved space efficiency. Our results give a tradeoff between space-optimality, and the worst-case number of bits written. This trade-off highlights the initial problem which motivated this work: a lower bound on the number of bits read in the worst case, for a space-optimal Gray code. After many hours and months spent on this problem, we are yet to find a tighter result than the $\Omega(\log d)$ bound shown by Fredman [2] (when d is more than a small constant). Our Recursive Partition Gray Code does provide a counter-example to any efforts to show a best-case bound of more than $\Omega(\log d)$, and our hope is that this work will contribute to a better understanding of the problem, and eventually, a tighter lower bound in the case of generating a space-optimal Gray code.

Acknowledgment: The fifth author would like to thank Peter Bro Miltersen for introducing us to this class of problems.

References

1. Gerth Stølting Brodal. personal communication, 2009.
2. Michael L. Fredman. Observations on the complexity of generating quasi-gray codes. *Siam Journal of Computing*, 7(2):134–146, 1978.
3. Frank Gray. Pulse code communications. *U.S. Patent 2632058*, 1953.
4. Marvin Minsky and Seymour Papert. *Perceptrons.* MIT Press, Cambridge, Mass., 1969.
5. M. Ziaur Rahman and J. Ian Munro. Integer representation and counting in the bit probe model. *Algorithmica*, December 2008.
6. Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *J. ACM*, 44(2):257–271, 1997.