Improved Methods For Generating Quasi-Gray Codes

Dana Jansens

October 16, 2009

Abstract

Consider a sequence of bit strings of length d, such that each string differs from the next in a constant number of bits, and the same holds for the first and last strings in the sequence. We call this sequence a cyclic 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 show a trade-off between these four constraints, and present algorithms which do less work on average than previous results, while also increasing the number of strings generated. Previous work has required at least one extra bit in the algorithms, such that at most $1/2$ of the possible bit strings are generated. Our results do not require an extra bit, and generate a factor of $1 - O(d^{-t})$, for a constant t, of the possible bit strings.

1 Introduction

We are interested in efficiently generating a set of bit strings. The class of bit strings we wish to generate are cyclic quasi-Gray codes. A Gray code [Gra53] is a sequence of bit strings, such that any two consecutive strings differ in exactly one bit. We use the term quasi-Gray code 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. The term quasi-Gray code first appeared in a paper by Fredman [Fre78], with a slightly different, but

comparable definition. Lastly, a quasi-Gray code is called *cyclic* if the first and last generated bit strings also differ in at most 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 refer to the state of all bits within the algorithm's data structure to be its *state*. We restrict an algorithm's data structure to using exactly d bits. In this way the data structure's state corresponds exactly to the generated bit strings. At each step, the input to the algorithm will be the previously generated bit string. 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 $L \leq 2^d$. When $L = 2^d$ we call the data structure space-optimal, as it generates all possible bit strings. When $L < 2^d$ the structure is non-space-optimal, a technique which can be used to improve the time required to generate each consecutive bit string.

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, in order to 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 should be a constant value. Third, we examine and improve 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 show a trade-off between these different goals, where one must be sacrificed to improve another, as demonstrated in our results.

Our decision to limit the algorithm's data structure to exactly d bits differs from previous work, where the data structure could use more or fewer bits than the strings it generated [Fre78, RM08]. To compare their results to our own, we consider the extra bits in their data structure to be a part of their generated bit strings. This gives, in our opinion, 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, we want to support the following three operations:

- *next* generates the bit string of rank $(k+1) \mod L$
- *previous* generates the bit string of rank $(k-1) \mod L$
- *rank* returns k

We work within the bit probe model [MP69, RM08], 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 [Fre78] to construct algorithms for generating quasi-Gray codes and describe the algorithms' behaviour, as well as to discuss lower bounds.

1.1 Definitions

We introduce a notation for the iterated log function which is 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 $\log^{(c)}(n) = \log^{(c-1)}(\log(n))$. That is, $\log^{(2)} n = \log(\log n)$. Throughout, the base of the log function is assumed to be 2 unless stated otherwise.

1.2 Results Summary

In Section 3.1, we use a DAT that generates a standard Gray code to construct a new DAT that generates a quasi-Gray code while reducing the average number of bits read. We then apply this technique recursively in Section 3.2 to create a d-dimensional DAT which reads worst-case d bits, but reads on average only $\log^{(c)} d+3$ bits, and writes at most $c+1$ bits, for any constant $c \geq 0$, to generate a bit string in the code. These DATs are all space-optimal.

In section 3.3 we extend this result to create an d-dimensional DAT which reads worst-case all d bits, while reading at most 3 bits on average, and writing at most $\log^* n + 1$ bits to generate each bit string, while also being space-optimal.

In Section 3.4 we construct a $d = n+O(\log n)$-dimensional DAT that reads and writes in the worst case $O(\log n)$ bits, and on average $O(\log n)$ bits. We do this while keeping the algorithm space-optimal for large d. This reduces the worst-case bits read dramatically. By combining a simple Gray code with this result, we are able to produce a DAT of dimension $d = n+O(\log n)$ that reads $O(\log n)$ bits on average and in the worst case, but writes at most 3

Thm	Dimension	WC Read	WC Write	Avg Read	Space Effic
3.4	d	d	$c+1$	$\log^{(c)} d + 3$	1
3.5	d	d	$\log^* d + 1$	3	1
3.8	$n + O(t \log n)$	$O(t \log n)$	$O(t \log n)$	$O(1)$	$1 - O(n^{-t})$
3.9	$n + O(t \log n)$	$O(t \log n)$	3	$O(t \log n)$	$1 - O(n^{-t})$
3.10	$n + O(t \log n)$	$O(t \log n)$	$2c + 3$	$O(\log^{(c+1)} n)$	$1 - O(n^{-t})$

Table 1: Summary of new results

bits to generate each bit string, thus producing a quasi-Gray code, while remaining space-optimal for large d. We then combine results from Section 3.2 to produce a $d = n + O(\log n)$-dimensional DAT that reads worst case $O(\log n)$ bits, reads on average $O(\log^{(c+1)} n)$ bits, and writes at most $2c + 3$ bits, for any constant $c \geq 0$. This improves the average work to generate each bit string in a quasi-Gray code, while also improving the space-efficiency compared to previously known results.

These results are summarized in Table 1. "WC Read" stands for bits read in the worst case, "Avg Read" stands for bits read on average to generate each successive bit string, and "Space Effic" refers to the space efficiency. "Thm" refers to the theorem, corollary, or lemma in which the result appears. When "WC Write" is a constant then the resulting code is a quasi-Gray code.

2 Model

2.1 Decision Assignment Tree

We use the Decision Assignment Tree (DAT) model to describe and analyze algorithms for generating bit strings in a quasi-Gray code. In the DAT model, an algorithm is described as a binary tree. 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 the DAT reads and modifies as the *state* of the DAT. Generally the initial bit string for a quasi-Gray code of dimension d, and thus the initial state of its generating DAT, is the bit string $\{0\}^d$. Each internal node of the tree is labeled with a single fixed position within the input bit string, and represents reading that bit.

Let T be a DAT of dimension d. 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 if the bit read was a 0 or a 1. This repeats recursively until a leaf node in the tree is reached.

Each leaf of T represents a subset of states for the DAT, where the bits read along the path to the leaf are in a fixed state. If the path to a leaf does not read every bit in the input string, it may represent more than one state for the DAT, or be responsible for changing the bits for generating more than one bit string in the code. In this case, the leaf will be reached multiple times while generating all bit strings in the code.

More formally, a state S_i for a DAT is represented by a single leaf L_i. When the DAT is in state S_i, traversing the DAT while reading the current state will lead to the leaf L_i. It is possible for two different states S_i and S_j to share the same leaf L_m if they both cause the same bit positions to be read and those positions share all the same values. In this case it is required that the bits which were not read on the path to the leaf L_m must be in a different state each time traversing the DAT reaches L_m.

Each leaf node contains rules that describe which bits to update to generate the next bit string in the code. The update rules are constrained in the following ways:

1. Each rule must set a single fixed bit directly to 0 or to 1.

2. The rules together must change at least one bit.

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 paired to a 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.

2.2 Next and previous operations

The *next* operation takes a bit string of dimension d as input, and modifies it to become the next bit string in a quasi Gray-code. This operation does not necessarily require reading all d bits of the current string, and a non-trivial lower bound remains unknown. A trivial DAT, such as iterating through the two's-complement binary representations of 0 to 2^d, 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 requires at most two bits to be read and updated. Meanwhile, it is possible to create a DAT that generates the Binary Reflected Gray Code, as described in Section 2.4. This DAT would always write exactly one bit, but requires reading all d bits to generate each bit string. A tradeoff exists between these two, and by writing more bits, it is possible to reduce the number of bits read.

To generate a Gray code of dimension d with length $L = 2^d$ strings, it is known that any DAT will require reading at least $\Omega(\log d)$ bits for some bit string. This bound is included in [Fre78]. Fredman also conjectured that for a Gray code of dimension d with $L = 2^d$, any DAT will have to 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[1].

For any quasi-Gray code of dimension d, it is always possible to construct a DAT that generates the same bit strings in reverse order. It is clear that the average and worst case number of bits read or written to generate each bit string will be the same as generating the bit strings in the original order. We can call this symmetric operation the *previous* operation for the original code, which can be performed by a separate DAT of dimension d.

2.3 Assembling DATs

Decision Assignment Trees can be assembled by joining together other DATs. We present here observations based on this.

Lemma 2.1. *Given two DATs, L and R, each for a binary code of dimension d with 2^d states. The L and R trees may be joined together, under a new root node, to create a DAT of dimension $d + 1$.*

Proof. We join the L tree and R tree together by adding a new root node, and making L and R its left and right children respectively. The subtrees L and R each read and write d bits. We assign them to the same bits, 0 to d, and the root node to the $d+1$-*th* bit. We assume w.l.o.g. that reading a 0 at the root node means to move to the root of the L subtree, while 1 means to move to the root of the R subtree. Assume that the $d + 1$-*th* bit is initially set to 0. If it is 1, then swap L and R in what follows.

[1] In [RM08] the authors claim to have proven this conjecture true for "small" d by exhaustive search.

It is clear that L remains a valid Decision Assignment Tree of dimension d. And because it never changes the $d{+}1$-*th* bit, it will never cause the R subtree to be used. Thus our initial construction is a valid Decision Assignment Tree of dimension $d+1$ that counts through only 2^d states.

Let L_A be the first state of L and L_Z be the last state. If L is cyclic, then any two states such that L_A immediately follows L_Z in the code is valid. Similarly, let R_A and R_Z be the first and last states of R. We modify the construction to join the code generated by L to the code generated by R, producing a new code of dimension $d+1$:

1. Make the update rules of L_Z change the counter to state R_A, and change the $d+1$-*th* bit to 1.

2. Make the update rules of R_Z change the counter to state L_A, and change the $d+1$-*th* bit to 0.

Because the L subtree is able to count through 2^d states, it will take $2^d - 1$ steps to go from L_A to L_Z. Likewise for the R subtree. Within each of these steps, the $d+1$-*th* bit is not changed, and so the subtree is able to operate correctly. After generating $2^d - 1$ bit strings in L, the $d+1$-*th* bit is changed, and the state is changed to R_A. This makes 2^d successive bit strings, generated by the L subtree. Now the same argument holds for the R subtree, which will generate $2^d - 1$ bit strings within its own subtree, and then move to L_A, completing a full cycle through all 2^{d+1} possible bit strings. Thus, we have a cyclic binary code of length $d+1$. □

2.4 Generating the BRGC

The standard Binary Reflected Gray Code (or BRGC) was introduced by Gray in 1953 [Gra53]. The code's structure is defined recursively. For a single bit, the code is simply 0 then 1. To create a code of $d+1$ bits, take the code of d bits: first place the 2^d bit strings of the dimension d Gray code in order. Place a 0 at the left of each of the bit strings. Then repeat the same 2^d bit strings in reverse order, placing a 1 at the left of each one. Figure 1 shows the BRGC for up to three bits. Note that the last bit string in the code differs from the first in a single bit, making the code a cyclic Gray code.

The BRGC can be generated by a DAT, and can be used as a tool for creating more complex quasi-Gray codes.

0	0 0	0 0 0
1	0 1	0 0 1
	1 1	0 1 1
	1 0	0 1 0
		1 1 0
		1 1 1
		1 0 1
		1 0 0

Figure 1: The standard Binary Reflected Gray Code for 1, 2, and 3 dimensions

Lemma 2.2. *The Binary Reflected Gray Code of dimension d can be generated by a DAT, which requires reading d bits and writing at most* 1 *bits to generate each successive bit string.*

Proof. For a Binary Reflected Gray Code of dimension 1, create a DAT with height 2. The root node reads the 0-*th* bit (counting from the right side). If the root reads a 0, move to its left child, if it reads a 1, move to its right child. The left child changes bit 0 to 1 and the right child changes bit 1 to be 0. This generates the cyclical BRGC of dimension 1.

Let L be a DAT for the BRGC of dimension d, and Let R be a DAT which generates the same bit strings as L in reverse order. Then, by Lemma 2.1, we can use L and R to construct a new DAT of dimension $d+1$. We choose L_A to be the bit string 0^d and L_Z to be the bit string 10^{d-1}. Since the BRGC is cyclic, L_A is the state which follows L_Z. We choose R_A to be the state 10^{d-1} and R_Z to be 0^d.

Because $R_Z = L_A$ and $L_Z = R_A$, no bits needs to be changed to move between them. Thus, in the combined DAT, only $d+1$-*th* bit needs to change in order to generate L_A from R_Z or R_A from L_Z, and the DAT is able to move between subtrees with only one bit change.

Further, the first 2^d states will correspond to a dimension d BRGC, with a 0 in the $d+1$-*th* bit. The second 2^d states will correspond to a dimension d BRGC in reverse order, with a 1 in the $d+1$-*th* bit. This is precisely the definition of the BRGC of dimension $d+1$. Thus, we are able to construct a DAT that generates the BRGC of any dimension. □

3 Efficient generation of quasi-Gray codes

In this section we will address how to efficiently generate quasi-Gray codes of dimension d. We examine efficiency in terms of the number of bits read and written in the worst case to generate each successive bit string, the number of bits read on average to generate the entire code, and the space efficiency, or ratio of the number of bit strings generated by the DAT to the number possible, 2^d. Note that the codes we generate are all cyclic. First we present DATs which read up to d bits in the worst-case, but read fewer bits on average. Then we present our lazy counters which read at most $O(\log d)$ bits in the worst-case, while also reading fewer bits on average.

3.1 Composite Gray code construction

Lemma 3.1. *Given integer constants $d > 0$, $k > 0$, $p > 0$. Assume we have a DAT for a quasi-Gray code of dimension d, that generates $L = 2^d$ bit strings, 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 k bits on average, and writing at most p bits in the worst case.*

Then there is a DAT for a quasi-Gray code of dimension $d+\lceil \log k \rceil$, which generates all $2^{d+\lceil \log k \rceil}$ bit strings, and where generating each bit string from the previous one requires reading no more than $\log k + 2$ bits on average, and writing at most $p + 1$ bits. That is, the number of bits read decreases from k to $O(\log k)$ on average, while increasing the number of bits written by 1 in the worst case.

Proof. We are given a DAT A that generates a quasi-Gray code of dimension d. The DAT, A, requires reading on average no more than k bits, and writing at most p bits to generate each bit string. We are also given a DAT B for the Binary Reflected Gray Code of dimension d', such as described in Section 2.4. The DAT, B, requires reading all d' bits for every transition, and writing only 1 bit in the worst case.

We construct a new DAT from the two DATs of dimension d and d'. 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 last state, to generate the initial bit string

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, which requires d' bits to be read and 1 bit to be written. However, the first d bits are only read and written when the last m' bits cycle back to their initial state - once for every $2^{d'}$ bit strings generated by the combined DAT.

If we let $d' = \lceil \log k \rceil$, then the average read cost of the combined quasi-Gray code to generate each bit string in the sequence is:

$$\begin{aligned}
\text{average bits read} &= (\text{cost of } d'\text{-code}) + \frac{(\text{average cost of } d\text{-code})}{\text{number of states between } d\text{-code changes}} \\
&\leq d' + \frac{k}{2^{d'}} \\
&= \lceil \log k \rceil + \frac{k}{2^{\lceil \log k \rceil}} \\
&\leq \lceil \log k \rceil + 1 \\
&\leq \log k + 2
\end{aligned}$$

The number of writes in the worst case, is the number of bits that must change to update both the code contained in the first d bits, and the code contained in the last d' bits. Thus the number of bits written per increment operation is:

$$\begin{aligned}
\text{worst case bits written} &= (\text{cost of } d\text{-code update}) + (\text{cost of } d'\text{-code update}) \\
&\leq p + 1
\end{aligned}$$

□

3.2 Reading fewer bits, Writing a constant number

This new DAT can then be used to construct an even more efficient DAT for generating a quasi-Gray code. We use it to construct a space-optimal DAT that generates such a code while reading on average no more than $\log^{(c)} d + 3$ bits, and never writing more than $c + 1$ bits to generate a bit string, for any constant $c \geq 0$.

Lemma 3.2. *Given an n such that $\log^{(c)} n \geq 4$, for some $c \geq 0$. Then for any $m \geq \frac{n}{2}$, $\log^{(c)} m \geq \frac{1}{2} \log^{(c)} n$.*

Proof. For the base case $c = 0$, it is true by the assumption on m. Then assuming it is true for c, proof by induction for $c + 1$ follows:

$$\log^{(c+1)} m = \log(\log^{(c)} m) \tag{1}$$

$$\geq \log(\frac{1}{2} \log^{(c)} n) \text{ (by inductive hypothesis)} \tag{2}$$

$$= \log^{(c+1)} n - 1 \geq \frac{1}{2} \log^{(c+1)} n \tag{3}$$

Step (3) above comes from the fact that $x - 1 \geq x/2$ if and only if $x \geq 2$. In this case, $x = \log^{(c+1)}$ and the following is true under our initial assumptions for n: $\log^{(c)} n \geq 4 \iff \log(\log^{(c)} n) \geq 2 \iff \log^{(c+1)} n \geq 2$. Thus $\log^{(c)} m \geq \frac{1}{2} \log^{(c)} n$ for all $c \geq 0$. □

Lemma 3.3. *Given an n such that $\log^{(c+1)} n \geq 4$, for some $c \geq 0$. Let $m = n - \lceil \log(\log^{(c)} n + 3) \rceil$. Then we can say the following:*

1. $\log(\log^{(c)} n + 3) \leq 1 + \log^{(c+1)} n$
2. $\log^{(c)} m \geq 4$

Proof. By our choice of n, we know that $\log^{(c)} n \geq 16$, and thus $n \geq 16$. Proof of the first statement follows from this fact:

$$\log(\log^{(c)} n + 3) \leq \log(\log^{(c)} n + \log^{(c)} n) = \log(2 \log^{(c)} n) = 1 + \log^{(c+1)} n \tag{4}$$

Proof of the second statement follows from Lemma 3.2. However to use it we must show its assumptions are satisfied. As we have already seen, $\log^{(c)} n \geq 16 \geq 4$, and our choice of c has the same range. So we must show that $m \geq n/2$. The inequality in step (8) is satisfied when $n \geq 16$, which we have already shown.

$$m = n - \lceil \log(\log^{(c)} n + 3) \rceil \tag{5}$$

$$\geq n - (1 + \log(\log^{(c)} n + 3)) \tag{6}$$

$$\geq n - (2 + \log^{(c+1)} n) \text{ (from (4))} \tag{7}$$

$$\geq n - (2 + \log n) \geq n/2 \tag{8}$$

This shows that the assumptions of Lemma 3.3 also satisfy the assumptions of Lemma 3.2. Therefore, from Lemma 3.2 we know $\log^{(c)} m \geq \frac{1}{2} \log^{(c)} n$, which completes the proof of the second statement of Lemma 3.3:

$$\log^{(c)} m \geq \frac{1}{2} \log^{(c)} n \geq \frac{1}{2} 16 \geq 4$$

□

Theorem 3.4. *Given a d such that $\log^{(c)} d \geq 4$, for a constant $c \geq 0$. There is a DAT of dimension d, which generates a quasi-Gray code of length $L = 2^d$, where generating the next bit string requires reading no more than $\log^{(c)} d + 3$ bits on average, and writing at most $c + 1$ bits.*

Proof. The proof is by induction. Let $c = 0$. In this case, a standard Binary Reflected Gray Code satisfies the requirements. It requires reading $n \leq \log^{(0)} n + 3 = n + 3$ bits and writes 1 bit each step.

Then let $c \geq 0$, and assume that the lemma is true for c. We will show it is true for $c + 1$.

Let d be such that $\log^{(c+1)} d \geq 4$. We define $m = d - \lceil \log(\log^{(c)} n + 3) \rceil$. By Lemma 3.3 we know $\log^{(c)} d \geq 4$. Therefore, we know by our inductive hypothesis, there exists a DAT M of dimension m that requires reading on average no more than $\log^{(c)} m + 3$ bits, and writing at most $c + 1$ bits for each transition.

We note that $\log^{(c)} m + 3 \leq \log^{(c)} d + 3$ since $m \leq d$, and use the construction from Lemma 3.1 to perform the inductive step and construct a new DAT. Using M with Lemma 3.1, we set $k = \log^{(c)} d + 3$ and $p = c + 1$. This gives us a new DAT of dimension $m + \lceil \log k \rceil = m + \lceil \log(\log^{(c)} d + 3) \rceil = d$.

Our new DAT-code requires reading, on average, no more than the following number of bits for each transition:

$$\begin{aligned} (\text{average bits read}) = \log k + 2 &= \log(\log^{(c)} d + 3) + 2 \\ &\leq \log^{(c+1)} d + 3 \ \textit{(from Lemma 3.3)} \end{aligned}$$

From Lemma 3.1, in the worst case, our DAT writes at most most $p+1 = (c + 1) + 1$ bits to generate the next bit string.

Thus, there exists a DAT for any $c \geq 0$, when $\log^{(c)} d \geq 4$, which generates all 2^d bit strings of dimension d, and requires reading on average no more than $\log^{(c)} d + 3$ bits, and writing at most $c + 1$ bits to generate each bit string from the previous. □

3.3 Reading a constant average number of bits, Writing almost a constant amount

We have shown in Section 3.2 that it is possible to construct a quasi-Gray code which requires reading, on average, at most $O(\log^{(c)} d)$ bits, and never requires writing more than $c + 1$ bits, for any constant $c \geq 0$. This allows reading a small number of bits while writing only a constant amount.

From Theorem 3.4, it immediately follows that we can create a DAT which generates all 2^d bit strings of dimension d, for which each generating step requires reading a constant number of bits on average. This is a trade off, as the DAT requires writing at most $O(\log^* n)$ in the worst case, meaning the code generated by this DAT is not a quasi-Gray code.

Corollary 3.5. *Let $d > 0$, $d \in \mathbb{Z}$ and $c = \log^* d - 3$. Then there exists a space-optimal DAT of dimension d which has the following properties for each generating step:*

1. *For any $d \geq 5$, the DAT reads no more than 11 bits on average to generate the next bit string in the code, and writes no more than $\log^* d - 2$ bits in the worst case.*

2. *For any $d \geq 9$, the DAT reads no more than 5 bits on average to generate the next bit string in the code, and writes no more than $\log^* d - 1$ bits in the worst case.*

3. *For any $d \geq 12$, the DAT reads no more than 4 bits on average to generate the next bit string in the code, and writes no more than $\log^* d$ bits in the worst case.*

4. *For any $d \geq 12$, the DAT reads no more than 3 bits on average to generate the next bit string in the code, and writes no more than $\log^* d + 1$ bits in the worst case.*

Proof. Let $d \geq 5$, and $c = \log^* d - 3$. Then $c \geq 0$ and $\log^{(c)} d \geq 4$. Then it follows from Theorem 3.4 that there exists a space-optimal DAT of dimension d, which has the following properties for each generating step: In the average case, it requires reading no more than $\log^{(\log^* d - 3)} d + 3 \leq 11$ bits. And in the worst case, it requires writing at most $\log^* d - 2$ bits.

Let $d \geq 9$, and $m = d - 4 \geq 5$. From our previous statement, we can construct a DAT of dimension m, which requires reading at most 11 bits on

average, and writing no more than $\log^* m - 2 \leq \log^* d - 2$ bits in the worst case. By using this DAT with Lemma 3.1, there exists a DAT of dimension $m + \lceil \log k \rceil = m + \lceil \log 11 \rceil = m + 4 = d$, which requires reading no more than $1 + \lceil \log 11 \rceil = 5$ bits to generate the next bit string in the code, and writes no more than $\log^* d - 1$ bits in the worst case.

Let $d \geq 12$, and $m = d - 3 \geq 9$. From our previous statement and Lemma 3.1, there exists a DAT of dimension $m + \lceil \log 5 \rceil = m + 3 = d$, which requires reading no more than $1 + \lceil \log 5 \rceil = 4$ bits on average to generate the next bit string in the code, and writes no more than $\log^* d$ bits in the worst case.

Let $d \geq 14$, and $m = d - 2 \geq 12$. From our previous statement and Lemma 3.1, there exists a DAT of length $m + \lceil \log 4 \rceil = m + 2 = d$, which requires reading no more than $1 + \lceil \log 4 \rceil = 3$ bits on average to generate the next bit string in the code, and writes no more than $\log^* d + 1$ bits in the worst case. □

3.4 Lazy counters

A lazy counter is a structure for generating a quasi-Gray code. In the first n bits, it counts through the two's-complement 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 only a constant number of bit changes to be reached. Thus the combined data structure generates a quasi-Gray code. We present a few known lazy counters, and then improve upon them.

Frandsen *et al* [SFMS97] 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 a bit string which combines both b and i together, 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 quasi-Gray code.

The bits of b move through the standard two's-complement binary numbers. However, moving from one such number to the next may require writing as many as n bits. The bits in i are a pointer into b. For a two's-complement 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 two's-complement number is equal to 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 two's-complement binary number. The algorithm is stated as follows:

Algorithm 1: LazyIncrement

Input: $b[]$: an array of n bits; i: an integer of $\log n$ bits

```
1 if b[i] = 1 then
2     b[i] ← 0;
3     i ← i + 1;
4 else
5     b[i] ← 1;
6     i ← 0;
7 end
```

The maximum number of bit strings generated by this counter is $2^d = n2^n$, however it actually generates significantly less.

The bits of b move through each two's-complement binary number, but with additional states in between, such that each state in differs by a single bit in b. Thus, the number of states between two of these numbers is equal to the number of the bits that need to change, and this is equal to the distance to the right-most 0 bit in b.

The right-most bit in b is 0 for half of the two's-complement numbers. For the other half, there is a 0 to the left of it half of the time. If the right-most bit is position 1, then bit j contains the right-most 0 exactly $\frac{2^n}{2^j}$ times, over all two's-complement numbers which fit in n bits. Meanwhile, the bits in i make it possible to have these transitional states, but don't provide any additional states beyond them. Thus we can count the number of total states in the counter. Bit j is the right-most 0 in b for $\frac{2^n}{2^j}$ numbers, and each time it is, it requires j bit flips to reach the next number. Additionally, when there are no 0 bits in b at all, n bit flips are required to reach the next number, which returns b back to all zeros. Thus the total number of bit strings generated by this lazy counter, for dimension $d = n + \log n$ is:

$$\begin{aligned}\text{bit strings generated} &= \sum_{j=1}^{n} j\frac{2^n}{2^j} + n \\ &= 2^n \sum_{j=1}^{n} \frac{j}{2^j} + n \\ &= 2^{n+1} - 2\end{aligned}$$

The space efficiency of this counter, or the ratio of bit strings generated to the number of possible strings is $\frac{2^{n+1}-2}{2^{n+\log n}} = \frac{2^n - 1}{n2^{n-1}}$. For large values of n, $\lim_{n\to\infty} \frac{2^n - 1}{n2^{n-1}} = 0$. That is, the counter is non-space-optimal, and the space efficiency of the counter grows worse as its dimension grows larger.

To generate each successive bit string, the worst-case number of bits read or written by this counter is the cost to increment an integer of dimension $\log n$, plus the single bit which changes in b. Thus the counter never reads nor writes more than $\log n + 1$ bits. On average, to increment an integer (using the standard two's-complement representation) requires reading and writing at most 2 bits, while only a single bit is ever read or written in b at a time. Thus the average cost to generate a bit string in this counter is 3.

The `LazyIncrement` algorithm presented above can be constructed as DAT. First construct a DAT that reads the value of i. This tree reads all $\log n$ bits of i and has n leaves, one for each value of i. The leaf which represents $i = c$ will then read bit k of b. If the bit was a 0, it is changed to a 1, and all of the bits in i are set to 0. If the bit was a 1, then the bit is changed to a 0 and i is changed to represent $c + 1$.

Lemma 3.6. *There exists a DAT of dimension $d = n + \log n$ which generates $\frac{2^n - 1}{n2^{n-1}}$ 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[2] (unpublished) 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

[2]Gerth Stølting Brodal

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 3.6, and k is a single bit, making the counter have dimension $d = n + \log n + 1$:

Algorithm 2: SpinIncrement

```
Input: b[]: an array of n bits; i: an integer of log n bits; k: a single bit
1  if k = 0 then
2      i ← i + 1 ; // spin i
3      if i = 0 then
4          k ← 1 ; // the value of i has rolled over
5      end
6  else
7      LazyIncrement(b[], i) ; // really increment the counter
8      if i = 0 then
9          k ← 0;
10     end
11 end
```

This new counter spins through i every time a bit in b is changed to a 1. This happens exactly half of the times that b is changed, which occurs each time the previous `increment` function is called. Spinning through all values for i adds exactly $2^{\log n} = n$ states. Thus the total number of states for this improved lazy counter is $n(2^n - 1) + 2^n - 1 = (n + 1)(2^n - 1)$.

The space efficiency of this improved counter is $\frac{(n+1)(2^n-1)}{2n2^n}$. The limit as n gets large is $\lim_{n\to\infty} \frac{(n+1)(2^n-1)}{2n2^n} = \frac{1}{2}$. That is, when the counter has large dimension d, approximately half of the 2^d possible bit strings are generated.

The worst-case number of bits read or written by this counter is one more than the counter in 3.6. The only cases added are where i and k are changed, which together to not exceed that bound. However, in the case where b and i are changed, k may also be read and changed, and so the counter may read and write at most $\log n + 2$ bits to generate the next bit string. Checking the value of k requires reading a single bit, incrementing i requires on average to read and write at most 2 bits. Comparing i for equality to the fixed bit string $0^{\log n}$, once for each possible bit pattern in i, requires reading on average 2 bits. From these observations, it follows that the average number of bits read

and written by this counter, to generate the next bit string, does not exceed 4.

The `SpinIncrement` algorithm can be constructed as DAT using the DAT from Lemma 3.6. Add a new root node which reads k. When $k = 0$, go to its left child, which is the root of a subtree identical to the DAT for `LazyIncrement`. When $k = 1$, go to its right child. The right child is a subtree which reads all the bits of i. For all leaf nodes of this subtree, i is incremented. And when i was equal to $n - 1$, k is also set to 0.

Lemma 3.7. *There exists a DAT of dimension $d = n + \log n$ which generates $\frac{(n+1)(2^n-1)}{2n2^n}$ bit strings, where in the limit $n \to \infty$ the space efficiency becomes $\frac{1}{2}$. The DAT reads and writes in the worst case $\log n + 2$ bits to generate each successive bit string, and on average reads and writes 4 bits.*

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

Algorithm 3: 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 ← i + 1 ; // happens in (2^g − 1)/2^g of the cases
3      if i = 0 then
4          k ← k + 1;
5      end
6  else
7      LazyIncrement(b[], i) ; // do a real increment
8      if i = 0 then
9          k ← 0;
10     end
11 end
```

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 3.7. 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 $\frac{n(2^g - 1)(2^n - 1) + 2^n - 1}{n2^n2^g} = \frac{(n^{t+1} - n)(2^n - 1) + 2^n - 1}{n^{t+1}2^n} = 1 - O(\frac{1}{n^t})$. Thus as n gets large, this counter becomes more space efficient, and is space-optimal in the limit $n \to \infty$.

In the worst case, this counter reads and writes every bit in i and k, and a single bit in b, thus $g + \log n + 1 \leq (t+1) \log n + 1$. On average, the counter now reads and writes $O(1)$ bits. This follows from a similar argument to that made for Lemma 3.7, where each line added to the algorithm also reads on average $O(1)$ bits.

The `DoubleSpinIncrement` can also be constructed as DAT by building on the DAT from Lemma 3.7. Replace the root node with a subtree which reads all the bits of k and has n^c leaf nodes. For leaf nodes that read a value in k less than $n^c - 1$, the leaf node becomes the root of a subtree similar to the right child of the root node in the DAT for `SpinIncrement`. These subtrees are modified in that k is incremented instead of set to 0 when i was equal to $n - 1$. The one leaf node where $k = n^c - 1$ becomes the root of a subtree identical to the left child of the root node in the DAT for `SpinIncrement`.

Theorem 3.8. *There exists a DAT of dimension $d = n + \log n + g$, where $1 \leq g \leq t \log n$, for $t > 0$, with space efficiency $1 - O(n^{-t})$. The DAT 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.*

Rahman and Munro give a counter in [RM08] which 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 so generates $\frac{2^n}{2^{n+1}} = \frac{1}{2}$ of all the possible bit strings. This counter is more efficient in terms of bits modified, while being less space efficient, than our third lazy counter. By modifying our lazy counter to use Gray codes internally, we are able to make the work performed asymptotically equivalent to the counter by Rahman and Munro, while writing a smaller constant number of bits per increment, and retaining our superior space efficiency.

We modify our counter in Theorem 3.8 to make i and k hold a cyclic Gray code instead of a standard two's complement number. The BRGC is a suitable Gray code for this purpose, and so we will use it. Given a function

$next(j)$ that takes a bit string j of rank r in the BRGC and returns the bit string of rank $r+1$, and a function $rank(j)$ that returns the rank value of the bit string j in the BRGC. The following algorithm provides a lazy counter of dimension $d = n + \log n + g$, for $1 \leq g \leq O(\log n)$, which writes at most 3 bits, and reads at most $g + \log n + 1$ bits to generate the next state, and is space-optimal when n goes to infinity.

Algorithm 4: WineIncrement

Input: $b[]$: an array of n bits; i: a Gray code of $\log n$ bits; k: a Gray code of g bits

```
1  if k ≠ 100...00 then
2      i ← next(i) ; // happens in (2^g − 1)/2^g of the cases
3      if i = 0 then
4          k ← next(k);
5      end
6  else
       // do a real increment
7      if b[rank(i)] = 1 then
8          b[rank(i)] ← 0;
9          i ← next(i);
10         if i = 0 then
11             k ← 0 ; // wraps around to the initial state
12         end
13     else
14         b[rank(i)] ← 1;
15         k ← 0 ; // resets k to 0
16     end
17 end
```

The number of states used by this counter would be the same as our third lazy counter, except that we are unable to reset i to 0 when a bit in b is flipped to 1. Instead, we leave i as it is and observe that it does not significantly reduce the space efficiency of the counter.

Note that when a bit in b is flipped to a 1:

- $rank(i)$ points to index of the right-most 1 in b, counting from the least-significant bit.
- a bit in b flips to 1 iff the bit sequence b is entering the next state of the

standard two's complement encoding. That is, b counts from 0 to 2^n-1 as a two's complement binary number, with extra states in between. The extra states all come from steps where a bit in b flips to 0.

Based on these observations we can sum up all the values of i that occur when a bit in b flips to 1 as $2^n \sum_{i=1}^{n} \frac{i}{2^i} = 2^{n+1} - n - 2$. When i has a value greater than 0, the number of states lost compared to our third lazy counter is exactly the difference between i and 0. Thus, over all increment steps, this summation describes the total number of states lost compared to our third lazy counter. Therefore the total number of states used is $n(2^g-1)(2^n-1)+(2^n-1)-2^{n+1}+n+2$. The number of bits, and possible states, is unchanged from the third lazy counter. So the space efficiency of this counter as n grows large becomes:

$$\begin{aligned}
& \lim_{n\to\infty} \frac{n(2^g-1)(2^n-1)+2^n-2^{n+1}+n+1}{2^{n+\log n+g}} \\
= & \lim_{n\to\infty} \frac{n(2^g-1)(2^n-1)+2^n-2^{n+1}+n+1}{n2^g2^n} \\
= & \lim_{n\to\infty} \frac{n2^g2^n-n2^n-n2^g+2n-2^n+1}{n2^n2^g} \\
= & 1+\lim_{n\to\infty} \frac{2n-n2^n-n2^g-2^n+1}{n2^n2^g} \\
= & 1-O(2^{-g})
\end{aligned}$$

When $g = 1$, this is $1/2$, as in our third lazy counter. When g is an increasing function of n, then $\lim_{n\to\infty} g = \infty$ and the space efficiency converges to 1, that is the counter remains space-optimal in the limit.

Thus, when $g = t \log n$, for $t > 0$, our fourth iteration gives a counter that is more space efficient than previously known counters, with a constant number of bits written for each increment. The counter reads $O(t \log n)$ bits per increment as before, and writes at most 3 bits per increment operation, one in each of b, i, and k.

Theorem 3.9. *There exists a DAT of dimension $d = n + \log n + g$, where $1 \le g \le t \log n$, for $t > 0$, with space efficiency $1 - O(2^{-g})$. The DAT 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 $O(t \log n)$ bits.

While the previous counter reads at most $g + \log n + 1$ bits in the worst case, it's average number of bits read is also $O(\log n)$. Using our Gray code counter from Theorem 3.4, we are able to bring the average number of bits read down as well. The worst case number of bits remains $g + \log n + 1$, but on average, we only need to read at most $2 \log^{(c+1)} n + O(1)$ bits, for any $c \geq 0$.

The algorithm does not need to change from its fourth iteration for these modifications. We simply make i a quasi-Gray code counter from Theorem 3.4 of dimension $\log n$ and k a similar quasi-Gray code counter of dimension $g = t \log n$, for a $t > 0$.

We determine the average number of bits read in the previous algorithm for each line of the `increment` function. Line 1 is `false` once for each bit in b that is flipped to a 1. Bits in b flip to a 1 exactly half of the bit-flips overall in b, which is $2^n - 1$ times. In this case, the entire string of k must be read to determine its inequality, which has dimension g. When k is not equal to 10...00 however, fewer bits need to be read on average. Half of the time, only a single bit needs to be read. One quarter of the time, two bits must be read, and so on. Each time line 1 begins to return `false`, it stays `false` for $(2^g - 1)(2^{\log n})$ increment operations, as k and i count through their states, during with each bit string in k is seen once for each value in i. Thus the the average number of bits read for line 1, R_1, is:

$$
\begin{aligned}
R_1 &\leq \frac{(2^n-1)2^{\log n}\sum_{i=1}^{g} i\frac{2^g}{2^i} + (2^n-1)g}{n(2^g-1)(2^n-1)+2^n-2^{n+1}+n+1} \\
&= \frac{(2^n-1)n2^g\sum_{i=1}^{g}\frac{i}{2^i} + (2^n-1)g}{n(2^g-1)(2^n-1)-2^n+n+1} \\
&\leq \frac{(2^n-1)n2^{g+1}}{n(2^g-1)(2^n-1)+2^n-2^{n+1}+n+1} \\
&= \frac{(2^n-1)n2^{g+1}}{n(2^g-1)(2^n-1)-2^n+n+1} \\
&\leq \frac{(2^n-1)n2^{g+1}}{n(2^g-1)(2^n-1)} \\
&= \frac{2^{g+1}}{(2^g-1)} \\
&= 2\frac{2^g-1+1}{2^g-1} \\
&= 2(1+\frac{1}{2^g-1}) \\
&= O(1)
\end{aligned}
$$

Line 2 increments our counter from Theorem 3.4, and so reads on average $R_2 = \log^{(c+1)} n + 3$ bits.

Line 3 checks if i is equal to a specific value. This check is done consecutively over all values of i. Thus half the time, only one bit needs to be checked, a quarter of the time, two bits, and so on. The average number of bits read for this line is then $\sum_{i=1}^{\log n} i\frac{n}{2^i}$, each time it is reached. Once the check in line 1 returns false, it does so once for each value of k other than 10..00. Thus this line is executed $(2^g-1)(2^n-1)$ times. The average number of bits read by line 3 is:

$$
\begin{aligned}
R_3 &\leq (\sum_{i=1}^{\log n} i\frac{n}{2^i})\frac{2^g(2^n-1)}{n(2^g-1)(2^n-1)+2^n-2^{n+1}+n+1} \\
&\leq \frac{2n2^g(2^n-1)}{n(2^g-1)(2^n-1)-2^n+n+1} \\
&= \frac{n2^{g+1}(2^n-1)}{n(2^g-1)(2^n-1)-2^n} \\
&= \frac{2^{g+1}}{(2^g-1)-\frac{2^n}{n(2^n-1)}} \\
&\leq \frac{2^{g+1}}{(2^g-1)-1} \\
&= 2\frac{(2^g-2)+2}{(2^g-2)} \\
&= 2+\frac{4}{(2^g-1)} \\
&= O(1)
\end{aligned}
$$

Line 4 is similar to line 2, but with a slightly larger counter. It reads on average $R_4 = \log^{(c)}(t\log n)+3 = \log^{(c)} t + \log^{(c+1)} n + 3$ bits, for $c \geq 1$.

Line 7 reads a single bit in b, and must read all $\log n$ bits in i in order to determine its rank. The line of code is executed $2^{n+1}-2$ times over the entire sequence of transitions for the counter as a whole. Thus the average number of bits read is:

$$
\begin{aligned}
R_7 &= \frac{(1+\log n)(2^{n+1}-2)}{n(2^g-1)(2^n-1)+2^n-2^{n+1}+n+1} \\
&= \frac{2(2^n-1)+2(2^n-1)\log n}{n(2^g-1)(2^n-1)-2^n+n+1} \\
&\le \frac{2(2^n-1)+2(2^n-1)\log n}{n(2^g-1)(2^n-1)-2^n} \\
&= \frac{2+2\log n}{n(2^g-1)-1-\frac{1}{2^n-1}} \\
&\le \frac{2+2\log n}{n(2^g-1)-2} \\
&\le 2+\frac{2\log n-2n(2^g-1)}{n(2^g-1)} \\
&\le 2+\frac{2\log n}{n(2^g-1)} \\
&= O(1)
\end{aligned}
$$

Line 8 reads all $\log n$ bits of i, similarly to line 7, and is executed no more times than is line 7. Thus the average number of bits read for line 8 is $R_8 \le O(1)$.

Line 9 has the average cost as line 2 per execution, and is executed fewer times over all generated states, thus $R_9 \le R_2$ bits.

Line 10 has an average cost per execution which is the same as line 3, but is executed fewer times, at most $2^{n+1}-2$ times. Thus its average cost $R_{10} \le O(1)$.

Line 11 does not need to read any bits, as the state of k is already known from line 1.

Line 14 is similar to line 8, and does not read more bits on average than line 7. Its average cost is $R_{14} \le 7$.

Line 15, like line 11, does not need to read any bits.

The total average number of bits read can be determined by the summation $\sum_{i=1}^{15} R_i$. It is clear that this average is at most $2\log^{(c+1)} n + O(1)$. The number of bits written in the worst case remains constant, though it grows slightly to $2(c+1)+1 = 2c+3$. Thus with $c=1$, we can read on average $O(\log\log n)$ bits, write at most 5, and read in the worst case $(t+1)\log n + 1$

bits. This is accomplished while keeping the space efficiency, as in the previous counter, such that the ratio of bit patterns used to those possible goes to 1 as n grows large.

The `WineIncrement` algorithm can be constructed as a DAT as well, as long as the quasi-Gray codes used in i and k can be generated with a DAT. This is done by using the DAT structures for i and k inside the tree from Theorem 3.8. For this DAT, we are using a quasi-Gray code instead of standard integers for our counting variables. While this changes the description of the resulting DAT, the actual structure does not change dramatically. First, replace the root node with a subtree T_k which reads all the bits of k and has n^c leaf nodes, labeled from 1 to n^c. Let leaf j be reached when k has rank j in its quasi-Gray code. The highest rank leaf will read the last bit string from the code in k, and it becomes the root of another subtree, $T_{i_{n^c}}$. This subtree corresponds to the main `else` clause of the algorithm. The subtree reads i, and has n leaves, one for each possible state of i. Again, let leaf j be reached when i has rank j in its quasi-Gray code. Then, in the leaf node j of $T_{i_{n^c}}$, the j-th bit of b is read. If the bit was a 0, it is changed to a 1, and k is moved ahead one state to 0. If the bit was a 1, then the bit is changed to a 0 and i is changed to represent the state of rank $j+1$ in its quasi-Gray code, and lastly if j is $n-1$, that is this leaf represents the highest rank for the quasi-Gray code in i, additionally move k ahead one state to return it to 0.

Other leaf nodes of T_k also become roots of subtrees T_{i_u} for $1 \leq u < n^c$, where T_{i_u} is rooted at the u-th child of T_k, and is reached when k has rank u in its quasi-Gray code. These subtrees represent the many cases where the main `if` clause is true in the algorithm. The subtrees each read all of i and have n leaf nodes, one for each possible state of i. Each leaf node has a rule to modify i such that its rank in its quasi-Gray code increases one. Additionally the n-th leaf of each subtree T_{i_u}, has a rule to modify k to represent the state of rank $u+1$ in its quasi-Gray code.

Theorem 3.10. *There exists a DAT of dimension $d = n + \log n + g$, where $1 \leq g \leq t \log n$, for $t > 0$, with space efficiency $1 - O(2^{-g})$. If $c \geq 1$, the DAT reads in the worst case $g + \log n + 1$ bits and writes in the worst case $2c + 3$ bits to generate each successive bit string, and on average reads at most $2 \log^{(c+1)} n + O(1)$ bits.*

References

[Fre78] Michael L. Fredman. Observations on the complexity of generating quasi-gray codes. *Siam Journal of Computing*, 7(2):134–146, 1978.

[Gra53] Frank Gray. Pulse code communications. *U.S. Patent 2632058*, 1953.

[MP69] Marvin Minsky and Seymour Papert. *Perceptrons*. MIT Press, Cambridge, Mass., 1969.

[RM08] M. Ziaur Rahman and J. Ian Munro. Integer representation and counting in the bit probe model. *Algorithmica*, December 2008.

[SFMS97] Gudmund Skovbjerg Frandsen, Peter Bro Miltersen, and Sven Skyum. Dynamic word problems. *J. ACM*, 44(2):257–271, 1997.