Low Complexity Two-Dimensional Weight-Constrained Codes

Erik Ordentlich and Ron M. Roth

***Abstract*—Two low complexity coding techniques are described for mapping arbitrary data to and from $m \times n$ binary arrays in which the Hamming weight of each row (respectively, column) is at most $n/2$ (respectively, $m/2$). One technique is based on flipping rows and columns of an arbitrary binary array until the Hamming weight constraint is satisfied in all rows and columns, and the other is based on a certain explicitly constructed "antipodal" matching between layers of the Boolean lattice. Both codes have a redundancy of roughly $m+n$ and may have applications in next generation resistive memory technologies.**

***Index Terms*—Boolean lattice, resistive memory, two-dimensional coding, weight-constrained codes.**

I. Introduction

Let $\mathbb{B}$ be the subset $\{0,1\}$ of the set of integers $\mathbb{Z}$, and for any positive integer ℓ, let $\langle \ell \rangle$ denote the integer set $\{1,2,\ldots,\ell\}$. For a word $\boldsymbol{x} = (x_i)_{i \in \langle \ell \rangle}$ in $\mathbb{B}^\ell$, denote by $\mathsf{w}(\boldsymbol{x})$ the (Hamming) weight of $\boldsymbol{x}$ (equivalently, $\mathsf{w}(\boldsymbol{x}) = \sum_{i \in \langle \ell \rangle} x_i$). Consider the set $\mathcal{A}_{m \times n}$ of all $m \times n$ arrays $A = (A_{i,j})_{i,j}$ over $\mathbb{B}$ such that the weight of each row is at most $n/2$ and the weight of each column is at most $m/2$; namely, for every $i \in \langle m \rangle$ and $j \in \langle n \rangle$,

$$\mathsf{w}\left((A_{i,h})_{h \in \langle n \rangle}\right) \le n/2 \quad \text{and} \quad \mathsf{w}\left((A_{k,j})_{k \in \langle m \rangle}\right) \le m/2 \,.$$

We are interested in the problem of efficiently encoding and decoding arbitrary data to and from (a subset of) $\mathcal{A}_{m \times n}$. Following the usual formal definitions, a code for this problem consists of a subset $\mathbb{C} \subseteq \mathcal{A}_{m \times n}$, an encoder (one-to-one mapping) $E : \langle |\mathbb{C}| \rangle \to \mathbb{C}$, and a decoder $D : \mathbb{C} \to \langle |\mathbb{C}| \rangle$, such that $D(E(u)) = u$ for all $u \in \langle |\mathbb{C}| \rangle$. Of interest are codes for which $E(\cdot)$ and $D(\cdot)$ can be computed with low complexity and $\mathbb{C}$ is as large as possible. Obviously, $|\mathbb{C}| \le |\mathcal{A}_{m \times n}| < 2^{mn}$, and we shall refer to the gap $mn - \log_2 |\mathbb{C}|$ as the redundancy of the code.

Efficient schemes for encoding and decoding data to and from binary arrays with respective row and column weights *precisely* $n/2$ and $m/2$ (or any other uniform fraction of n and m) have been described previously in [1],[2]. In this paper, we take advantage of the more relaxed constraint (i.e., weights *at most* $n/2$ and $m/2$, respectively) and present new codes that have lower encoding and decoding complexity and lower redundancy than the schemes of [1],[2]. In particular, the codes of [1],[2] have redundancies that are proportional to $(m+n)\log(mn)$ and have super-linear encoding and decoding complexity; on the other hand, the present codes have redundancies of roughly $m+n$, linear decoding complexity, and linear encoding complexity in the case of one of the codes. One can compare the achieved redundancy to the best possible redundancy for this constraint, namely, $mn - \log_2 |\mathcal{A}_{m \times n}|$, which, for $m = n$, is shown to be approximately $1.42515n$ in [3].

Our first code is presented in Section III and is based on a bit-flipping procedure and is relatively simple to explain and verify. Its encoding complexity, however, is yet to be tightly characterized (we do conjecture it to be considerably lower than the corresponding encoders of [1],[2]). The second code is more involved and is presented in Section IV, which makes up the bulk of this paper. We shall refer to the first code as the *iterative flipping code* and to the second as the *antipodal matching code*, where the names are derived from key algorithmic components of the respective codes. A summary of the attributes of the codes is as follows. The iterative flipping code has a redundancy of $m+n-1$ for all m and n, linear decoding complexity, and a bound of $O(mn(m+n))$ register operations on the worst-case encoding complexity. We conjecture that the true worst-case encoding complexity is $c(m,n)mn$ where $c(m,n)$ is slowly growing in m and n (e.g., logarithmically) and would thus be slightly super-linear in the number of encoded bits, which is $(m-1)(n-1)$ for $m \times n$ arrays. The antipodal matching code, on the other hand, is best suited for the case where the number of columns (say) n is even, and for such n has a redundancy of $m+n$ and linear encoding and decoding complexity (for odd n, the redundancy is $m+n+1$).

II. Motivation

Although our primary purpose here is to study the purely information theoretic problem of efficiently coding into $\mathcal{A}_{m \times n}$, this and similar constraints and the associated codes may be applicable, as presented in [5], to limiting parasitic current in next generation memory technologies based on crossbar arrays of resistive devices, such as memristors (see, e.g., [4]). As depicted in Figure 1, bits are stored in these types of memories by "programming" the resistance values of individual devices to high and low states, which can be accomplished by the application of suitable positive and negative voltages to the row and column of a selected device (circled in the figure) and zero voltage to all other rows and columns. Notice that

E. Ordentlich is with Hewlett–Packard Laboratories, Palo Alto, CA 94304, USA, email: erik.ordentlich@hp.com.

R.M. Roth is with the Computer Science Department, Technion, Haifa, Israel, email: ronny@cs.technion.ac.il. His was carried out in part while visiting Hewlett–Packard Laboratories, Palo Alto, CA 94304, USA.

This work was supported in part by the US Government Nano-Enabled Technology Initiative. Portions of this work were presented at the 2nd Non-volatile Memories Workshop, UC San Diego, La Jolla, CA (March 2011) and at the 2011 International Symposium on Information Theory, St. Petersburg, Russia (July 2011).

Copyright © 2011 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.

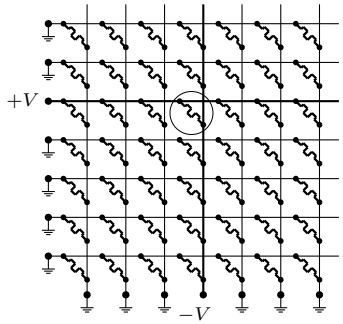

Fig. 1. Crossbar of resistive memory devices with a selected device undergoing programming.

the devices in the same row and column as the selected device being programmed experience half of the programming voltage drop across their terminals. These devices are thus termed half-selected devices. The devices are engineered so that they do not undergo resistance changes at half of the programming voltage. Nevertheless, the half-selected devices, particularly those in low resistance states, will collectively induce a parasitic current that, in the worst case (e.g., all half-selected devices are in low resistance states), increases linearly with the dimensions of the array. This imposes a limit on the maximum size of crossbar arrays, which, in turn, limits planar data densities in memories comprised of such crossbars, particularly in the multi-layered architectures proposed in [4].

The proposed codes might help with this problem in the following manner. Assuming that the dominant sources of the parasitic current are indeed the half-selected devices in *low* resistance states and that 1's are mapped to such states (with 0 's mapped to high resistance states), imposing the above weight constraint on stored data using the above codes could potentially mitigate the worst-case half-select current by roughly a factor two. This would allow for larger arrays, with correspondingly more stored data (up to the redundancy of the code), for a given total current limit, or for reduced current for a given array size or given data density.

III. Iterative flipping code

As mentioned in Section I, the iterative flipping code encodes $(m-1)(n-1)$ information bits into $m \times n$ arrays belonging to $\mathcal{A}_{m\times n}$. Encoding proceeds as in Figure 2.

The encoding procedure is guaranteed to terminate since each row or column flip strictly reduces the total number of 1's in the array.[1] Since the procedure terminates only when there is no row and column with weight larger than $n/2$ and $m/2$, respectively, the final array must belong to $\mathcal{A}_{m\times n}$.

The $(m-1) \times (n-1)$ subarray of information bits U arranged in Step 1 in Figure 2 can be decoded as follows from the encoded array A.

Proposition 3.1: If U and A are respectively the information bit array and the encoded array from the algorithm of Figure 2, then $U_{i,j} \equiv A_{i,j} + A_{i,n} + A_{m,j} + A_{m,n} \pmod 2$.

[1]The procedure can be shown to correspond to a deterministic schedule for the zero-temperature Glauber dynamics [6] in a certain Ising model having $m+n$ states corresponding to the flip status of each row and column and with interactions determined by the initial array.

Input: Sequence u of $(m-1)(n-1)$ information bits.

1) Arrange u as an $(m-1)\times(n-1)$ array U.
2) Extend U into an $m\times n$ array A by adding an mth row and nth column of 0's.
3) If A has any rows with weight larger than $n/2$ then flip all 1's to 0's and all 0's to 1's in these rows.
4) If A has any columns with weight larger than $m/2$ then flip all 1's to 0's and 0's to 1's in these columns.
5) If flips happened in **Step 4** then go to **Step 3**, otherwise terminate and output the final array.

Output: $m \times n$ binary array A.

Fig. 2. Encoding algorithm for iterative flipping code.

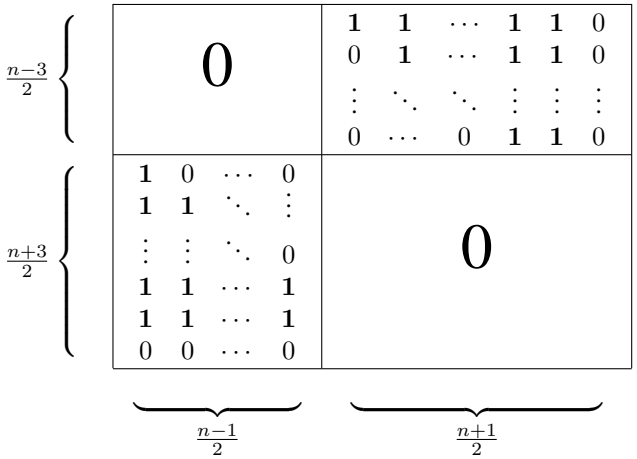

Fig. 3. Example of a sequence of $n \times n$ arrays, n odd, that requires $\Omega(n)$ row–column flips to encode.

Proof: Any column or row flip involving the array locations $(i,j),(i,n),(m,j),(m,n)$ during encoding flips precisely two of the corresponding array bits, thereby preserving the modulo-2 sum of these 4 bits. The claim follows since the initial modulo-2 sum is simply $U_{i,j}$, as the other three bits are initialized to 0. ∎

Thus, the number of bit operations required for decoding is linear in the number of decoded bits. The true complexity of encoding, on the other hand, is less clear. An upper bound of $O(mn)$ row–column flips (equivalently, $O(mn(m+n))$ bit flips) and iterations between rows and columns follows readily from the above noted fact that each row–column flip strictly reduces the number of 1's in the array.

Conversely, in Figure 3 we exhibit a sequence of binary $n \times n$ arrays that requires $\Omega(n)$ row–column iterations under the encoding algorithm of Figure 2. In the figure, we assume n odd, but the construction extends readily to n even as well. In reference to the figure, notice that the number of 1's in the first (from the left) column is $(n+1)/2$, and that this column is the only constraint violating row or column in the array. The encoder will thus flip this column, resulting in $(n-3)/2$ new 1's in the first column of the $((n-3)/2) \times ((n-1)/2)$ upper left block. This, in turn, induces a constraint violation in the first row, and thus a flip of this row (and only this row). The resulting partial row of new 1's in the upper left block, in turn, induces a constraint violation in the second column

(and only this column). The 1's in the array are arranged so that this process continues with each column flip triggering the flipping of precisely one new row and vice versa, until the first $(n-1)/2$ columns and $(n-3)/2$ rows are all flipped. The total number of row–column flips required for encoding is thus at least $n-2$. We conjecture that the true worst-case complexity of the encoder of Figure 2 is much closer to $O(m+n)$ row–column flips than $O(mn)$.

IV. Antipodal matching code

The highly sequential nature and uncertain complexity of the iterative flipping encoder motivates the consideration of alternative schemes. In this section, we describe the antipodal matching code which has the attributes noted in Section I. The preceding iterative flipping code essentially consists of repeatedly applying to selected rows and columns the simple one-to-one mapping between binary words that flips all 0's to 1's and 1's to 0's. While this mapping serves to reduce the number of 1's in the array and, hence eliminate all constraint violations in the long run, it has the drawback that it may create new constraint violations in some columns (rows) when applied to a given row (column). This explains the need for at least $\Omega(\min\{m,n\})$ (possibly more) row–column flip iterations (as in Figure 3). The code considered in this section is based on antipodal matchings, a different class of one-to-one mappings on binary words, that avoid this drawback. We formally specify their properties in the next subsection. In Subsection IV-B, we present the construction of the 2D antipodal matching code based on a general antipodal matching. A specific, efficiently computable antipodal matching is then presented in Subsection IV-C, together with a proof that it is one-to-one. This matching can be computed in linear time.

A. Antipodal matchings

Fix ℓ to be a positive integer and, for two words $\boldsymbol{x} = (x_j)_{j\in\langle\ell\rangle}$ and $\boldsymbol{y} = (y_j)_{j\in\langle\ell\rangle}$ in $\mathbb{B}^\ell$, we write $\boldsymbol{y} \le \boldsymbol{x}$ if $y_j = 1$ implies $x_j = 1$ for each index $j \in \langle\ell\rangle$. An *antipodal matching* ϕ is a mapping from $\mathbb{B}^\ell$ to itself with the following properties holding for every $\boldsymbol{x} \in \mathbb{B}^\ell$:

(P1) $\mathsf{w}(\phi(\boldsymbol{x})) = \ell - \mathsf{w}(\boldsymbol{x})$.

(P2) If $\mathsf{w}(\boldsymbol{x}) \ge \ell/2$ then $\phi(\boldsymbol{x}) \le \boldsymbol{x}$ (note that $\mathsf{w}(\phi(\boldsymbol{x})) \le \ell/2$ by property (P1)).

(P3) $\phi(\phi(\boldsymbol{x})) = \boldsymbol{x}$.

From these properties we see that an antipodal matching can be decomposed into a collection of bijective mappings $\varphi = \varphi_w$ from $\mathcal{L}_w = \{\boldsymbol{x} \in \mathbb{B}^\ell : \mathsf{w}(\boldsymbol{x}) = w\}$ to $\mathcal{L}_{\ell-w} = \{\boldsymbol{x} \in \mathbb{B}^\ell : \mathsf{w}(\boldsymbol{x}) = \ell-w\}$, $w = 0, 1, 2, \ldots, \ell$, where each mapping φ_w satisfies the covering property (P2) and $\varphi_{\ell-w} = \varphi_w^{-1}$. It is straightforward to see that when $w > \ell/2$, the mapping φ_w defines a perfect matching in the bipartite graph $\mathcal{G}_w = (\mathcal{L}_w : \mathcal{L}_{\ell-w}, \mathcal{E}_w)$ with a vertex set $\mathcal{L}_w \cup \mathcal{L}_{\ell-w}$ and edge set

$$\mathcal{E}_w = \left\{(\boldsymbol{x}, \boldsymbol{y}) \in \mathcal{L}_w \times \mathcal{L}_{\ell-w} \ : \ \boldsymbol{y} \le \boldsymbol{x}\right\}.$$

Since $\mathcal{G}_w$ is a regular balanced bipartite graph (the degree of each vertex is $\binom{w}{\ell-w}$), the existence of such constituent mappings φ_w, which will also be referred to as antipodal matchings, is guaranteed by Hall's theorem [7, pp. 217–218].

Input: For n even, arbitrary sequence u of $(m-1)(n-1)-1$ bits.

1) Precede the sequence u with a 0 and arrange the resulting $(m-1)(n-1)$ bits into an $(m-1)\times(n-1)$ binary array U (thus, the added 0 is at position $(1,1)$ in U).
2) Extend U into an $m \times n$ array A by adding an mth row and nth column of 0's.
3) For each row of A that has weight $n/2$ or more, flip all 1's in the row to 0's and all 0's to 1's.
4) For all $j \in \langle n-1\rangle$ do:
   If $\mathsf{w}(A_{\langle m-1\rangle,j}) > m/2$ then do the following:
   a) replace $A_{\langle m-1\rangle,j}$ with $\phi\big(A_{\langle m-1\rangle,j}\big)$ (= the antipodal matching applied to the word formed by the first $m-1$ bits in the jth column);
   b) set $A_{m,j} \leftarrow 1$.
5) If $\mathsf{w}(A_{m,\langle n\rangle}) > n/2$ then do the following:
   a) replace $A_{m,\langle n-1\rangle}$ by $\phi\big(A_{m,\langle n-1\rangle}\big)$;
   b) set $A_{m,n} \leftarrow 1$.
6) If $\mathsf{w}(A_{\langle 2:m\rangle,n}) > m/2$ then do the following:
   a) replace $A_{\langle 2:m\rangle,n}$ with $\phi\big(A_{\langle 2:m\rangle,n}\big)$;
   b) set $A_{1,n} \leftarrow 1$;
   Otherwise set $A_{1,n} \leftarrow 0$.

Output: The resulting $m \times n$ array A.

Fig. 4. Encoding algorithm for antipodal matching code.

In Subsections IV-C and IV-D, we present efficiently computable (linear complexity) antipodal matchings φ_w for all word lengths ℓ and all $w \in \{0, 1, \ldots, \ell\}$, which then collectively constitute an efficient antipodal matching in the sense of properties (P1)–(P3).

B. 2D antipodal matching code from antipodal matchings

There are many ways to build a 2D weight-constrained code out of antipodal matchings and here we describe a particular scheme. As mentioned, the key idea behind the resulting antipodal matching code is that the underlying antipodal matching ϕ permits iteration-free encoding by not creating new constraint violations. We note that the specific scheme we describe does make (non-essential) use of bit-flipping, but in a single, non-iterative and parallelizable step, unlike in the iterative flipping code.

The encoding and decoding algorithms for the antipodal matching code are depicted in Figures 4 and 5, for the case when the number of columns n is even (see Remark 4.1 below for the case when n is odd). In these figures, $A_{\langle h:i\rangle,j}$ stands for the column word formed by the $i-h+1$ entries $A_{h,j}, A_{h+1,j}, \ldots, A_{i,j}$ of an array A, and $A_{\langle i\rangle,j}$ stands for $A_{\langle 1:i\rangle,j}$ (namely, the column word formed by the first i entries of the jth column of A). The respective notation $A_{i,\langle j\rangle}$ is used for (partial) rows.

Proposition 4.1: The output array A of the encoder of Figure 4 belongs to $\mathcal{A}_{m\times n}$.

Proof: We can see that after Step 3, not only do all rows of A have weight at most $n/2$, but the number of 1's among the first $n-1$ entries in each row (in particular, in the first row) is at most $(n/2)-1$. Additionally, right after Step 4a, the weight of $A_{\langle m-1\rangle,j}$ (namely, the number of 1's among the first $m-1$ entries of the jth column) is less than $m-1-(m/2) =$

Input: $m \times n$ array A.

1) If $A_{1,n} = 1$ then replace $A_{\langle 2:m\rangle,n}$ with $\phi\big(A_{\langle 2:m\rangle,n}\big)$.
2) If $A_{m,n} = 1$ then replace $A_{m,\langle n-1\rangle}$ with $\phi\big(A_{m,\langle n-1\rangle}\big)$.
3) For all $j \in \langle n{-}1\rangle$ do:
   If $A_{m,j} = 1$ then replace $A_{\langle m-1\rangle,j}$ with $\phi\big(A_{\langle m-1\rangle,j}\big)$.
4) Set $A_{1,n} \leftarrow A_{1,1}$.
5) For all $(i,j) \in (\langle m{-}1\rangle \times \langle n{-}1\rangle) \setminus \{(1,1)\}$ do: set $U_{i,j} \in \{0,1\}$ so that $U_{i,j} \equiv A_{i,j} + A_{i,n} \pmod 2$.

Output: $(m{-}1)(n{-}1) - 1$ information bits in U.

Fig. 5. Decoding algorithm for antipodal matching code.

$(m/2)-1$ for any j for which this step is applied. This means that right after Step 4, the weight of the jth column, $A_{\langle m\rangle,j}$, is at most $m/2$ for every $j < n$, even if the last entry, $A_{m,j}$, becomes 1 in Step 4b.

Further, since by design the antipodal matching applied in Step 4a does not introduce 1's where there were previously 0's, after Step 4, each of the first $m{-}1$ rows in A will continue to have weight at most $n/2$ with a strictly smaller weight among the first $n{-}1$ entries in each row.

Similarly, the application of the antipodal matching in Step 5a does not increase the weight in any of the first $n{-}1$ columns in A. And, since Step 5a results in an mth row having weight less than $n{-}1-(n/2) = (n/2)-1$, the weight of the mth row right after Step 5 is at most $n/2$, even if the last entry, $A_{m,n}$, becomes 1 in Step 5b.

By virtually the same reasoning, it follows that Step 6 results in the nth column having weight at most $m/2$, and in any row still having weight at most $n/2$. Note that the latter applies also to the first row, even if Step 6b is executed, as the number of 1's among the first $n{-}1$ entries in that row is at most $(n/2){-}1$. ∎

Proposition 4.2: The $(m{-}1)(n{-}1)-1$ information bits in U computed in Decoding Step 5 in Figure 5 coincide with the corresponding input bit array entries created in Encoding Step 1 in Figure 4.

Proof: Clearly, Decoding Step 1 correctly recovers the last $m{-}1$ entries of the nth column as of just after Encoding Step 5.

Next, Decoding Step 2 recovers the first $n{-}1$ entries of the mth row as of just after Encoding Step 4. A 1 in any of these entries of this row indicates that the antipodal matching was applied to the preceding entries of the corresponding column in Encoding Step 4a. Thus, the application of the antipodal matching in Decoding Step 3, by property (P3) of such a mapping, recovers the preceding entries of each column as of just after Encoding Step 3.

So, after Decoding Step 3, we have recovered the first $m{-}1$ rows of A as of just after Encoding Step 3, except for the entry $A_{1,n}$ (which was modified in Encoding Step 6).[2] Since both $A_{1,1}$ and $A_{1,n}$ are initialized to 0 in Encoding Steps 1 and 2, these entries remain equal even after a possible flip of the first row in Encoding Step 3. It follows that Decoding Step 4 correctly recovers the entry $A_{1,n}$ as of just after Encoding Step 3.

We now need to determine which rows were flipped in Encoding Step 3 so that they can be flipped back. The entries in the last column now carry this information, and the flipping is carried out in Decoding Step 5. ∎

It is easy to see from Figures 4 and 5 that encoding and decoding of the antipodal matching code involves the application of at most $n{+}1$ antipodal matchings along with another $O(mn)$ increment-decrement-compare operations over integer registers that are $O(1) + \log_2(\max\{m,n\})$ bits long (henceforth "operations" shall have this meaning). In the next subsection, we present a specific antipodal matching that, in turn, can be computed in a number of operations that grows linearly in the length of the word (as well as requiring only $O(1)$ registers beyond the word). The overall antipodal matching code will thus have encoding and decoding complexity of $O(mn)$ operations, which is linear in the number of encoded bits.

Remark 4.1: In Encoding Step 3 in Figure 4, a row is flipped also when its weight is *equal* to $n/2$: we need this in order to guarantee that Encoding Step 6b does not increase the weight of the first row beyond $n/2$ (for all other rows, it suffices to carry out the flipping in Encoding Step 3 only when the row weight is *greater* than $n/2$). In fact, it is the effect of Encoding Step 6b on the weight of the first row which requires us to assume that n is even. Thus, we can accommodate odd values of n by forcing one of the information bits in the first row to be 0 (thereby increasing the overall redundancy to $m{+}n{+}1$), and applying a flipping of the first row in Encoding Step 3 only to the remaining entries in that row, whenever the row weight is $(n{-}1)/2$ or more. ∎

C. An efficient antipodal matching

It will be convenient to present our efficient antipodal matching in terms of words over the bipolar alphabet $\Phi = \{\text{"+"}, \text{"−"}\}$. Recalling the discussion in Subsection IV-A, formally, we will define the constituent mappings φ' on bipolar words and obtain the mapping φ on binary words via a pre- and post- application of the symbol-wise mapping $b(\cdot)$ which maps 0 to "−" and 1 to "+", and its inverse $b^{-1}(\cdot)$.

The elements of Φ will be regarded as integers (for the purpose of addition and subtraction), where "+" and "−" stand for 1 and -1, respectively. The sum of entries of a word $\boldsymbol{x} \in \Phi^\ell$ will be denoted by $\mathcal{S}_{\boldsymbol{x}}$ (it is easy to see that $\mathcal{S}_{\boldsymbol{x}}$ and ℓ have the same parity: they are both even or both odd).

For a positive integer ℓ and $s \in \{\ell{-}2w : w = 0, 1, \ldots, \ell\}$, define

$$\mathcal{C}(\ell, s) = \{\boldsymbol{x} \in \Phi^\ell \,:\, \mathcal{S}_{\boldsymbol{x}} = s\} \,. \tag{1}$$

For any $s > 0$ in the above range, the two sets $\mathcal{C}(\ell, s)$ and $\mathcal{C}(\ell, -s)$ are referred to as *antipodal layers* in the Boolean lattice [8] (under the mapping $b(\cdot)$, these two sets correspond to the two partition elements of the vertex set of the bipartite graph $\mathcal{G}_w$ defined in Subsection IV-A).

[2] For each row except the first, the last entry in the row is used to indicate whether the contents of the row has been replaced through flipping (by Encoding Step 3) or antipodal matching (by Encoding Step 5). It is only the first row where the *first* entry, $A_{1,1}$, is actually used to record a flipping; the last entry, $A_{1,n}$, in that row is used in Encoding Step 6b to record a replacement of the contents of the last *column*.

Reformulating the discussion in Subsection IV-A in terms of the alphabet Φ, for each positive s in the above range, an antipodal matching is a bijective mapping $\varphi' : \mathcal{C}(\ell, s) \to \mathcal{C}(\ell, -s)$. In addition, for every $\boldsymbol{x} = (x_j)_{j\in\langle\ell\rangle}$ in Φ^ℓ, the image $\boldsymbol{y} = (y_j)_{j\in\langle\ell\rangle} = \varphi'(\boldsymbol{x})$ can have $y_j =$ "+" only if $x_j =$ "+", for any entry index j.

We are interested here in the problem of finding antipodal matchings which can be computed efficiently. This problem has been studied for the special case of $s = 1$ (and ℓ odd), primarily in the context of attempting to solve the long-standing problem as to whether there exists a Hamiltonian cycle in the bipartite sub-graph that is induced by the middle levels, $\mathcal{C}(\ell, 1)$ and $\mathcal{C}(\ell, -1)$, of the Boolean lattice; see [9], [10], [11]. We present here efficient antipodal matchings for all s. Our construction can be seen as a generalization of one of the matchings in [11] (yet we point out that some stronger results that are shown in [11] for $s = 1$, do not seem to generalize for larger s). We note that our construction can also be inferred from known explicit minimal partitions of $\mathbb{B}^\ell$ into symmetric chains [9], [14], [15]; our exposition here however will be self-contained.

We borrow some of our notation from [11]. Hereafter, we fix ℓ to be a positive integer and let $\mathbb{Z}_\ell = \mathbb{Z}/\ell\mathbb{Z}$ be the ring of integer residues modulo ℓ. For distinct i and j in $\mathbb{Z}_\ell$, we denote by $[i, j)$ the subset $\{i, i+1, i+2, \ldots, j-1\}$ of $\mathbb{Z}_\ell$ (where addition and subtraction are taken modulo ℓ: namely, to obtain the elements of $[i, j)$ one starts with i and iteratively adds the unity element of $\mathbb{Z}_\ell$ until one reaches $j-1$). For $i \in \mathbb{Z}_\ell$, we formally define $[i, i)$ to be the whole set $\mathbb{Z}_\ell$. The notation $(i, j]$ and (i, j) will stand for the subsets $[i+1, j+1)$ and $[i, j) \setminus \{i\}$, respectively (thus, $(i, i) = \mathbb{Z}_\ell \setminus \{i\}$ and $(i, i+1) = \emptyset$). For any $i, j \in \mathbb{Z}_\ell$ and $k \in (i, j)$ we have $[i, k) \cup [k, j) = [i, j)$ and, in particular (when $i = j$), $[i, k) \cup [k, i) = \mathbb{Z}_\ell$.

We will adopt the convention that entries of words in Φ^ℓ are indexed by $\mathbb{Z}_\ell$. For a word $\boldsymbol{x} \in \Phi^\ell$ and indexes $i, j \in \mathbb{Z}_\ell$, we denote by $\mathcal{S}_{\boldsymbol{x}}[i, j)$ the sum of the entries of $\boldsymbol{x}$ that are indexed by $[i, j)$. Notation such as $\mathcal{S}_{\boldsymbol{x}}(i, j]$ and $\mathcal{S}_{\boldsymbol{x}}(i, j)$ will have its obvious meaning (where $\mathcal{S}_{\boldsymbol{x}}(i, i+1)$ is defined as 0).

Given a word $\boldsymbol{x} \in \Phi^\ell$, denote by $\mathcal{P}_{\boldsymbol{x}}$ the subset of $\mathbb{Z}_\ell$ which consists of all *minimal* indexes i in the sense that

$$\mathcal{S}_{\boldsymbol{x}}[i, j) > 0 \quad \text{for all } j \in \mathbb{Z}_\ell \ . \tag{2}$$

Obviously, $\mathcal{P}_{\boldsymbol{x}} \neq \emptyset$ only if $\mathcal{S}_{\boldsymbol{x}} > 0$, and $i \in \mathcal{P}_{\boldsymbol{x}}$ only if $x_i x_{i+1} =$ "++".

Example 4.1: Consider the following word of length $\ell = 11$ over Φ:

$$\begin{array}{rcccccccccccc} & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{x} & = & + & - & - & + & - & \boxed{+} & + & - & \boxed{+} & \boxed{+} & + \end{array}$$

Here $\mathcal{S}_{\boldsymbol{x}} = 3$ and $\mathcal{P}_{\boldsymbol{x}} = \{5, 8, 9\}$. The entries marked by boxes are those that are indexed by $\mathcal{P}_{\boldsymbol{x}}$. ■

The next lemma, which specifies the size of $\mathcal{P}_{\boldsymbol{x}}$, is a special case of a result known as the Cycle Lemma [12] (see also [13] and references therein). Our derivation below will imply, in passing, a proof for this lemma, which we will include for the sake of completeness.

Lemma 4.3 (Dvoretzky and Motzkin [12]): Let $\boldsymbol{x} \in \Phi^\ell$ be such that $\mathcal{S}_{\boldsymbol{x}} > 0$. Then $|\mathcal{P}_{\boldsymbol{x}}| = \mathcal{S}_{\boldsymbol{x}}$.

In analogy with the definition of $\mathcal{P}_{\boldsymbol{x}}$, we have the following definition which suits words $\boldsymbol{y} \in \Phi^\ell$ with $\mathcal{S}_{\boldsymbol{y}} < 0$: we define $\mathcal{M}_{\boldsymbol{y}}$ to be the subset of $\mathbb{Z}_\ell$ which consists of all indexes i such that $\mathcal{S}_{\boldsymbol{y}}(j, i] < 0$ for all $j \in \mathbb{Z}_\ell$ (note that here, unlike (2), each index interval $(j, i]$ over which the sum is taken extends to the *left* of i).

Example 4.2: For the following word over Φ

$$\begin{array}{rcccccccccccc} & & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \boldsymbol{y} & = & + & - & - & + & - & \boxed{-} & + & - & \boxed{-} & \boxed{-} & + \end{array}$$

one has $\mathcal{S}_{\boldsymbol{y}} = -3$ and $\mathcal{M}_{\boldsymbol{y}} = \{5, 8, 9\}$. ■

For a word $\boldsymbol{x} = (x_j)_{j\in\mathbb{Z}_\ell}$ over Φ, we define its *conjugate* to be the word $\boldsymbol{x}^* = (-x_{-j})_{j\in\mathbb{Z}_\ell}$. Clearly, $(\boldsymbol{x}^*)^* = \boldsymbol{x}$, and it is also easy to see that $\mathcal{S}_{\boldsymbol{x}^*} = -\mathcal{S}_{\boldsymbol{x}}$ and that $\mathcal{M}_{\boldsymbol{x}^*} = -\mathcal{P}_{\boldsymbol{x}} = \{-t : t \in \mathcal{P}_{\boldsymbol{x}}\}$. We can state the following counterpart of Lemma 4.3 simply by replacing each word with its conjugate.

Lemma 4.4: Let $\boldsymbol{y} \in \Phi^\ell$ be such that $\mathcal{S}_{\boldsymbol{y}} < 0$. Then $|\mathcal{M}_{\boldsymbol{y}}| = -\mathcal{S}_{\boldsymbol{y}}$.

Let s be a positive integer in $\{\ell - 2w : w = 0, 1, 2, \ldots\}$ and recall the definition of the sets $\mathcal{C}(\ell, \pm s)$ from (1). Let the mapping $\mathsf{E}_s : \mathcal{C}(\ell, s) \to \mathcal{C}(\ell, -s)$ be defined as follows: for every $\boldsymbol{x} = (x_j)_{j\in\mathbb{Z}_\ell}$ in $\mathcal{C}(\ell, s)$, the entries of $\boldsymbol{y} = (y_j)_{j\in\mathbb{Z}_\ell} = \mathsf{E}_s(\boldsymbol{x})$ are given by

$$y_j = \begin{cases} -x_j = \text{"}-\text{"} & \text{if } j \in \mathcal{P}_{\boldsymbol{x}} \\ x_j & \text{otherwise} \end{cases} \ .$$

Note that from Lemma 4.3 we have $\mathcal{S}_{\boldsymbol{y}} = \mathcal{S}_{\boldsymbol{x}} - 2|\mathcal{P}_{\boldsymbol{x}}| = -\mathcal{S}_{\boldsymbol{x}} = -s$, so every image of E_s is indeed an element of $\mathcal{C}(\ell, -s)$. In addition, it is straightforward to see that y_j can be "+" only if x_j is, for every $j \in \mathbb{Z}_\ell$.

In a similar manner, we can also define the mapping $\mathsf{D}_s : \mathcal{C}(\ell, -s) \to \mathcal{C}(\ell, s)$ that maps every word $\boldsymbol{y} \in \mathcal{C}(\ell, -s)$ to a word $\boldsymbol{x} = \mathsf{D}_s(\boldsymbol{y})$, where

$$x_j = \begin{cases} -y_j = \text{"}+\text{"} & \text{if } j \in \mathcal{M}_{\boldsymbol{y}} \\ y_j & \text{otherwise} \end{cases} \ .$$

The remaining part of this subsection is devoted to showing that the mapping E_s is an antipodal matching. We start with the following lemma.

Lemma 4.5: Given $\boldsymbol{x} \in \Phi^\ell$, suppose that t and t' are distinct elements in $\mathcal{P}_{\boldsymbol{x}}$ such that $\mathcal{S}_{\boldsymbol{x}}[t, t') > 1$. Then $(t, t') \cap \mathcal{P}_{\boldsymbol{x}} \neq \emptyset$.

Proof: Consider the function $f : (t, t'] \to \mathbb{Z}$ which is defined by

$$f(i) = \mathcal{S}_{\boldsymbol{x}}[t, i) \quad \text{for every } i \in (t, t'] \ .$$

Note that since $t \in \mathcal{P}_{\boldsymbol{x}}$ then $f(i) > 0$ for every $i \in (t, t']$. Moreover, the function f is "stepwise continuous" in that it satisfies

$$|f(i+1) - f(i)| = 1 \quad \text{for every } i \in (t, t') \ ;$$

as such, this function takes all integer values in the range $\{1, 2, \ldots, f(t')\}$, where $f(t') = \mathcal{S}_{\boldsymbol{x}}[t, t') > 1$. In particular, there exists $k \in (t, t')$ such that $f(k) = 1$. Without loss of generality we can assume in addition that $f(j) > 1$ for every $j \in (k, t')$, namely, k is the "rightmost" element i among those in (t, t') that satisfy $f(i) = 1$.

Next, we show that $k \in \mathcal{P}_{\boldsymbol{x}}$, namely, $\mathcal{S}_{\boldsymbol{x}}[k, j) > 0$ for every $j \in \mathbb{Z}_\ell$. We do this by distinguishing between the following two ranges of j.

- $j \in (k, t']$. Here $k \in (t, j)$ and, so,
$$\mathcal{S}_{\boldsymbol{x}}[k, j) = \mathcal{S}_{\boldsymbol{x}}[t, j) - \mathcal{S}_{\boldsymbol{x}}[t, k) = \underbrace{f(j)}_{>1} - \underbrace{f(k)}_{1} > 0 \ .$$
- $j \in (t', k]$. Here $t' \in (k, j)$ and we get
$$\begin{aligned} \mathcal{S}_{\boldsymbol{x}}[k, j) &= \mathcal{S}_{\boldsymbol{x}}[k, t') + \mathcal{S}_{\boldsymbol{x}}[t', j) \\ &= (\mathcal{S}_{\boldsymbol{x}}[t, t') - \mathcal{S}_{\boldsymbol{x}}[t, k)) + \mathcal{S}_{\boldsymbol{x}}[t', j) \\ &= \underbrace{f(t')}_{>1} - \underbrace{f(k)}_{1} + \underbrace{\mathcal{S}_{\boldsymbol{x}}[t', j)}_{>0} > 0 \ , \end{aligned}$$
where the inequality $\mathcal{S}_{\boldsymbol{x}}[t', j) > 0$ follows from the assumption that $t' \in \mathcal{P}_{\boldsymbol{x}}$.

This completes the proof. ∎

The next proposition presents a useful characterization of $\mathcal{P}_{\boldsymbol{x}}$.

Proposition 4.6: Given $\boldsymbol{x} \in \Phi^\ell$, let $t_0, t_1, t_2, \ldots, t_{s-1}$ be s (> 0) distinct elements of $\mathbb{Z}_\ell$ with their subscripts assigned so that the following s subsets form a partition of $\mathbb{Z}_\ell$:
$$[t_0, t_1), [t_1, t_2), \ldots, [t_{s-1}, t_s)$$
(where $t_s = t_0$). The following two conditions are equivalent.

(i) $\mathcal{P}_{\boldsymbol{x}} = \{t_0, t_1, \ldots, t_{s-1}\}$.

(ii) For each $h = 0, 1, \ldots, s-1$: $\mathcal{S}_{\boldsymbol{x}}[t_h, t_{h+1}) = 1$ and $\mathcal{S}_{\boldsymbol{x}}[t_h, j) > 0$ for $j \in (t_h, t_{h+1})$.

Proof: (i) ⇒ (ii). Suppose that condition (i) holds. Then clearly $\mathcal{S}_{\boldsymbol{x}}[t_h, j) > 0$ for every $h = 0, 1, \ldots, s-1$ and $j \in \mathbb{Z}_\ell$. Thus, it remains to show that $\mathcal{S}_{\boldsymbol{x}}[t_h, t_{h+1}) \leq 1$. Yet this follows from Lemma 4.5 by observing that the assumed assignment of subscripts implies that
$$(t_h, t_{h+1}) \cap \mathcal{P}_{\boldsymbol{x}} = \emptyset \quad \text{for } h = 0, 1, \ldots, s-1 \ .$$

(ii) ⇒ (i). Suppose that condition (ii) holds. We show that $t_0 \in \mathcal{P}_{\boldsymbol{x}}$; the proof that every other element t_h is in $\mathcal{P}_{\boldsymbol{x}}$ will then follow simply by shifting the subscripts cyclically. Given $j \in \mathbb{Z}_\ell$, let r be the unique subscript for which $j \in (t_r, t_{r+1}]$. Then
$$\mathcal{S}_{\boldsymbol{x}}[t_0, j) = \Big(\sum_{h=0}^{r-1} \underbrace{\mathcal{S}_{\boldsymbol{x}}[t_h, t_{h+1})}_{1}\Big) + \underbrace{\mathcal{S}_{\boldsymbol{x}}[t_r, j)}_{>0} > r \geq 0 \ .$$
It remains to show that there exists no $t' \in \mathcal{P}_{\boldsymbol{x}}$ other than $t_0, t_1, \ldots, t_{s-1}$. Indeed, if there were such a t', then, for the unique subscript r for which $t' \in (t_r, t_{r+1})$, we would have
$$\mathcal{S}_{\boldsymbol{x}}[t_r, t_{r+1}) = \underbrace{\mathcal{S}_{\boldsymbol{x}}[t_r, t')}_{>0} + \underbrace{\mathcal{S}_{\boldsymbol{x}}[t', t_{r+1})}_{>0} > 1 \ ,$$
thereby contradicting condition (ii). ∎

One can easily obtain Lemma 4.3 from Proposition 4.6, as follows.

Proof of Lemma 4.3: Using the notation of Proposition 4.6 and writing $\mathcal{P}_{\boldsymbol{x}} = \{t_0, t_1, \ldots, t_{s-1}\}$, we get from the proposition that
$$|\mathcal{P}_{\boldsymbol{x}}| = s = \sum_{h=0}^{s-1} \underbrace{\mathcal{S}_{\boldsymbol{x}}[t_h, t_{h+1})}_{1} = \mathcal{S}_{\boldsymbol{x}}[t_0, t_0) = \mathcal{S}_{\boldsymbol{x}} \ ,$$
as claimed. ∎

We now reach the main result of this subsection.

Proposition 4.7: For every positive $s \in \{\ell - 2w : w = 0, 1, 2, \ldots\}$ and every $\boldsymbol{x} \in \mathcal{C}(\ell, s)$,
$$\mathsf{D}_s(\mathsf{E}_s(\boldsymbol{x})) = \boldsymbol{x} \ .$$
Namely, the mappings $\mathsf{E}_s : \mathcal{C}(\ell, s) \to \mathcal{C}(\ell, -s)$ and $\mathsf{D}_s : \mathcal{C}(\ell, -s) \to \mathcal{C}(\ell, s)$ are bijective.

Proof: Fix an $s = \ell - 2w$ and let $\boldsymbol{x} = (x_j)_{j \in \mathbb{Z}_\ell}$ be in $\mathcal{C}(\ell, s)$. We show that the respective image $\boldsymbol{y} = (y_j)_{j \in \mathbb{Z}_\ell} = \mathsf{E}_s(\boldsymbol{x})$ satisfies $\mathcal{M}_{\boldsymbol{y}} = \mathcal{P}_{\boldsymbol{x}}$ which, in turn, yields the desired result.

Write $\mathcal{P}_{\boldsymbol{x}} = \{t_0, t_1, \ldots, t_{s-1}\}$, where the subscripts h of t_h are as in Proposition 4.6. Now fix some $h \in \{0, 1, \ldots, s-1\}$, and recall that $y_j = x_j$ for every $j \in (t_h, t_{h+1})$ and $y_{t_{h+1}} = -x_{t_h} =$ "$-$". By Proposition 4.6 we have $\mathcal{S}_{\boldsymbol{x}}[t_h, t_{h+1}) = 1$ and, so,
$$\mathcal{S}_{\boldsymbol{y}}(t_h, t_{h+1}] = \underbrace{\mathcal{S}_{\boldsymbol{x}}[t_h, t_{h+1})}_{1} - \underbrace{x_{t_h}}_{1} + \underbrace{y_{t_{h+1}}}_{-1} = -1 \ .$$
Equivalently,
$$\mathcal{S}_{\boldsymbol{y}^*}[-t_{h+1}, -t_h) = 1 \ . \tag{3}$$
In addition, for every $j \in (t_h, t_{h+1})$ we have
$$\begin{aligned} \mathcal{S}_{\boldsymbol{y}}(j, t_{h+1}] &= \mathcal{S}_{\boldsymbol{y}}(t_h, t_{h+1}] - \mathcal{S}_{\boldsymbol{y}}(t_h, j] \\ &= -1 - \mathcal{S}_{\boldsymbol{y}}(t_h, j] = -1 - \mathcal{S}_{\boldsymbol{x}}(t_h, j] \\ &= -1 - \mathcal{S}_{\boldsymbol{x}}[t_h, j{+}1) + x_{t_h} = -\mathcal{S}_{\boldsymbol{x}}[t_h, j{+}1) \\ &< 0 \ . \end{aligned}$$
Equivalently,
$$\mathcal{S}_{\boldsymbol{y}^*}[-t_{h+1}, i) > 0 \quad \text{for every } i \in (-t_{h+1}, -t_h) \ . \tag{4}$$
Hence, by applying Proposition 4.6 to $\boldsymbol{y}^*$ we get from (3)–(4) that $-t_0, -t_1, \ldots, -t_{s-1} \in \mathcal{P}_{\boldsymbol{y}^*}$ i.e., $\mathcal{P}_{\boldsymbol{x}} \subseteq -\mathcal{P}_{\boldsymbol{y}^*} = \mathcal{M}_{\boldsymbol{y}}$. Equality then follows from the fact that $\mathcal{P}_{\boldsymbol{x}}$ and $\mathcal{M}_{\boldsymbol{y}}$ are both of the same size s. ∎

We can thus set the constituent mappings φ' to E_s for $s > 0$ and to D_s for $s < 0$ (φ' is necessarily the identity for $s = 0$, ℓ even).

The time complexity of computing E_s and D_s is dictated by the complexity of computing the set $\mathcal{P}_{\boldsymbol{x}}$ of minimal indexes of a given word $\boldsymbol{x} \in \Phi^\ell$. This set can be found naively by checking the condition (2) for each i in $\mathbb{Z}_\ell$, yet such a procedure runs in time that is quadratic in ℓ. The first proof in [13] for the Cycle Lemma implies the following alternate method for computing the set $\mathcal{P}_{\boldsymbol{x}}$. Two copies of the word $\boldsymbol{x}$ are pushed into a stack, bit by bit; each time that a "$-$" is pushed on top of a "$+$", both are popped; the bits of the second copy of $\boldsymbol{x}$ that remain in the stack then correspond to the elements of $\mathcal{P}_{\boldsymbol{x}}$. This procedure has linear time complexity (in ℓ), yet it requires space—additionally to the input and output space—that grows at least linearly with ℓ in order to keep track of the positions (in $\boldsymbol{x}$) of the bits that are in the stack. The algorithm that we present in the next subsection computes $\mathcal{P}_{\boldsymbol{x}}$ in time that is linear in ℓ and space that is logarithmic in ℓ.

D. Fast computation of minimal indexes

Figure 6 presents our algorithm for computing the set $\mathcal{P}_{\boldsymbol{x}}$ for any given word $\boldsymbol{x} \in \Phi^\ell$ with $s = \mathcal{S}_{\boldsymbol{x}} > 0$.

Input: $\boldsymbol{x} = (x_j)_{j\in\mathbb{Z}_\ell}$ over Φ with $\mathcal{S}_{\boldsymbol{x}} > 0$.
Data structures: $i, t \in \mathbb{Z}_\ell$; $\sigma, \mu \in \mathbb{Z}$; $\mathcal{P} \subseteq \mathbb{Z}_\ell$.

```
i = 0; σ = μ = 0; P = ∅;
do {
    if (σ ≤ μ) {
        μ = σ;
        t = i;
    }
    σ += x_i;
    i++;
} while (i ≠ 0);
h = σ;
while (h > 0) {
    if (σ ≤ h) {
        P = P ∪ {t};
        h--;
    }
    t--;
    σ -= x_t;
}
```

Output: Set $\mathcal{P}$.

Fig. 6. Algorithm for computing $\mathcal{P}_{\boldsymbol{x}}$.

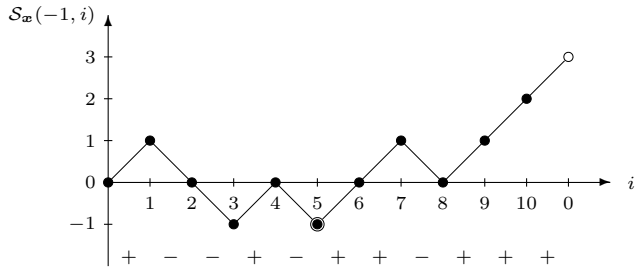

Fig. 7. Values of $\mathcal{S}_{\boldsymbol{x}}(-1,i)$ for the word in Example 4.1.

In the first "do–while" loop of the algorithm, an element $t = t_0$ is found which is the "rightmost" among the elements i that minimize $\mathcal{S}_{\boldsymbol{x}}(-1,i)$ over all $i \in \mathbb{Z}_\ell$; namely, $\mathcal{S}_{\boldsymbol{x}}(-1,t_0) < \mathcal{S}_{\boldsymbol{x}}(-1,j)$ for $j \in (t_0,0)$ and $\mathcal{S}_{\boldsymbol{x}}(-1,t_0) \le \mathcal{S}_{\boldsymbol{x}}(-1,j)$ for $j \in (-1,t_0)$ (we use the notation $(-1,\cdot)$ instead of $[0,\cdot)$ to cover correctly also the case where $t_0 = 0$). Note that at the end of the first loop, the value of the variable σ equals s $(= \mathcal{S}_{\boldsymbol{x}})$. It is easy to see that t_0 is an element of $\mathcal{P}_{\boldsymbol{x}}$: for $j \in (t_0,0)$ we have

$$\mathcal{S}_{\boldsymbol{x}}[t_0,j) = \mathcal{S}_{\boldsymbol{x}}(-1,j) - \mathcal{S}_{\boldsymbol{x}}(-1,t_0) > 0\ ,$$

and for $j \in (-1,t_0)$,

$$\begin{aligned}\mathcal{S}_{\boldsymbol{x}}[t_0,j) &= \mathcal{S}_{\boldsymbol{x}} - \mathcal{S}_{\boldsymbol{x}}[j,t_0)\\ &= \underbrace{\mathcal{S}_{\boldsymbol{x}}}_{>0} - \underbrace{(\mathcal{S}_{\boldsymbol{x}}(-1,t_0) - \mathcal{S}_{\boldsymbol{x}}(-1,j))}_{\le 0} > 0\ .\end{aligned}$$

For example, for the word $\boldsymbol{x}$ in Example 4.1, one gets the values of $\mathcal{S}_{\boldsymbol{x}}(-1,i)$ as shown in Figure 7; in this case $t_0 = 5$.

The second "while" loop iterates over $t \in \mathbb{Z}_\ell$ "backwards," starting with $t = t_0$, and the variable σ at the beginning of iteration t equals $\sigma(t) = \mathcal{S}_{\boldsymbol{x}}[t_0,t)$. When the "if" condition in that loop is met, the algorithm inserts the respective value, t_h, of t into $\mathcal{P}$, for $h = s, s-1, \ldots, 1$, where the first element to be inserted is[3] $t_s = t_0$. The stepwise continuity of $i \mapsto \sigma(i)$ guarantees that the loop indeed terminates, since $\sigma(i)$ takes all the integer values from $\sigma(t_0) = s$ down to $\sigma(t_0{+}1) = 1$. For example, for the word $\boldsymbol{x}$ in Example 4.1, one gets the values of $\sigma(i)$ as shown in Figure 8, and the values that are inserted into $\mathcal{P}$ are $t_3 = 5$, $t_2 = 9$, and $t_1 = 8$ (which are the abscissas of the three highlighted points on the piecewise-linear curve in Figure 8; these are precisely the points that can be "seen" from the right looking left, without being obstructed by the curve).

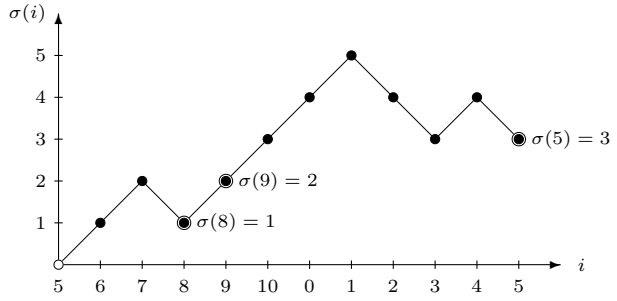

Fig. 8. Values of $\sigma(i)$ for the word in Example 4.1.

The next two properties can be easily verified by induction on $h = s{-}1, s{-}2, \ldots, 1$.

- $\sigma(t_h) = h$.
- $\sigma(t_h) < \sigma(j)$ for every $j \in (t_h, t_{h+1})$.

Thus, $\mathcal{S}_{\boldsymbol{x}}[t_h,t_{h+1}) = \sigma(t_{h+1}) - \sigma(t_h) = 1$ and $\mathcal{S}_{\boldsymbol{x}}[t_h,j) = \sigma(j) - \sigma(t_h) > 0$ for $j \in (t_h,t_{h+1})$. We can therefore conclude from Proposition 4.6 that $\mathcal{P} \subseteq \mathcal{P}_{\boldsymbol{x}}$; since both $\mathcal{P}$ and $\mathcal{P}_{\boldsymbol{x}}$ have the same size $s = \mathcal{S}_{\boldsymbol{x}}$, they must be equal.

The algorithm in Figure 6 has time complexity of $O(\ell)$ operations over integers in the range $\{0, \pm1, \pm2, \ldots, \pm\ell\}$, where the operations are either additions of ±1 or comparisons. The algorithm can be seen as a simplified version of a known method [16] for computing the sequence of minima seen in sliding windows of length ℓ of a sequence of length 2ℓ, where we take into account that the sequence is stepwise continuous.

ACKNOWLEDGMENT

We wish to thank Farzad Parvaresh, Gilberto Ribeiro, Gadiel Seroussi, and Pascal Vontobel for helpful discussions. We also thank the anonymous referees for their constructive comments, in particular for pointing out the relevance of the Cycle Lemma to our work.

REFERENCES

[1] R. Talyansky, T. Etzion, and R.M. Roth, "Efficient code constructions for certain two-dimensional constraints,"" *IEEE Trans. Inform. Theory*, 45 (1999), 794–799.

[2] E. Ordentlich and R.M. Roth, "Two-dimensional weight-constrained codes through enumeration bounds," *IEEE Trans. Inform. Theory,* 46 (2000), 1292–1301.

[3] It can be verified that the *second* element to be inserted into $\mathcal{P}$, namely t_{s-1}, has to be in $(t_0,0)$. This observation can be used to accelerate the second "while" loop, yet for clarity, we have elected to present the algorithm in its current form.

[3] E. Ordentlich, F. Parvaresh, and R.M. Roth, "Asymptotic enumeration of binary matrices with bounded row and column weights," in *Proc. IEEE Int'l. Symp. Inform. Theory,* St. Petersburg, Russia (July 2011), 116–120.

[4] D.B. Strukov and R.S. Williams, "Four-dimensional address topology for circuits with stacked multilayer crossbar arrays," *Proc. Nat'l. Acad. Sci.*, 106 (2009), 20155–20158.

[5] E. Ordentlich, G.M. Ribeiro, R.M. Roth, G. Seroussi, and P.O. Vontobel, "Coding for limiting current in memristor crossbar memories," *2nd Annual Non-Volatile Memories Workshop,*, UCSD, La Jolla, CA, March 2011. Presentation slides are accessible at: `http://nvmw.ucsd.edu/2011/`

[6] R.J. Glauber, "Time-dependent statistics of the Ising model," *J. Mat. Phys.,* 4 (1963), 294–307.

[7] N.L. Biggs, *Discrete Mathematics,* Oxford University Press, Oxford, 1985.

[8] G. Hurlbert, "The antipodal layers problem," *Discrete Math.,* 128 (1994), 237–245.

[9] M. Aigner, "Lexicographic matching in Boolean algebras," *J. Comb. Theory B,* 14 (1973), 187–194.

[10] D.A. Duffus, H.A. Kierstead, and H.S. Snevily, "An explicit 1-factorization in the middle of the Boolean lattice," *J. Comb. Theory A,* 65 (1994), 334–342.

[11] H.A. Kierstead and W.T. Trotter, "Explicit matchings in the middle levels of the Boolean lattice," *Order,* 5 (1988), 163–171.

[12] A. Dvoretzky and Th. Motzkin, "A problem of arrangemnts," *Duke Math. J.,* 14 (1947), 305–313.

[13] N. Dershowitz and S. Zaks, "The Cycle Lemma and some applications," *Europ. J. Combinatorics,* 11 (1990), 35–40.

[14] N. de Bruijn, C. Tengbergen, and D. Kruyswijk, "On the set of divisors of a number," *Nieuw Arch. Wiskunde,* 23 (1951), 191–193.

[15] C. Greene and D.J. Kleitman, "Strong versions of Sperner's Theorem," *J. Comb. Th. A,* 20 (1976), 80–88.

[16] D. Lemire, "Streaming maximum-minimum filter using no more than three comparisons per element," *Nordic Journal of Computing,* 13 (2006), 328–339.

Erik Ordentlich (S'92–M'96–SM'06–F'11) received the S.B. and S.M. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, MA, in 1990, and the Ph.D. degree, also in electrical engineering, from Stanford University, Stanford, CA, in 1996.

He is a Senior Research Scientist in the Information Theory Research Group at Hewlett-Packard Laboratories, Palo Alto, CA. He has been with Hewlett-Packard Laboratories since 1996, with the exception of a period in 1999–2002, when he was with iCompression, Inc., Santa Clara, CA. His work has addressed multiple topics in signal processing and information theory. He is a coinventor on numerous U.S. Patents and contributed technology to the ISO JPEG 2000 image compression standard.

Dr. Ordentlich was a corecipient of the 2006 IEEE Joint Communications–Information Theory Paper Award and is a member of Phi Beta Kappa and Tau Beta Pi. He served as Associate Editor for Source Coding for the IEEE TRANSACTIONS ON INFORMATION THEORY from 2007–2010.

Ron M. Roth (M'88–SM'97–F'03) received the B.Sc. degree in computer engineering, the M.Sc. in electrical engineering and the D.Sc. in computer science from Technion—Israel Institute of Technology, Haifa, Israel, in 1980, 1984 and 1988, respectively. Since 1988 he has been with the Computer Science Department at Technion, where he now holds the General Yaakov Dori Chair in Engineering. During the academic years 1989–91 he was a Visiting Scientist at IBM Research Division, Almaden Research Center, San Jose, California, and during 1996–97, 2004–05, and 2011–2012 he was on sabbatical leave at Hewlett–Packard Laboratories, Palo Alto, California. He is the author of the book *Introduction to Coding Theory*, published by Cambridge University Press in 2006. Dr. Roth was an associate editor for coding theory in IEEE TRANSACTIONS ON INFORMATION THEORY from 1998 till 2001, and he is now serving as an associate editor in *SIAM Journal on Discrete Mathematics*. His research interests include coding theory, information theory, and their application to the theory of complexity.