On the Poset Structure of Floating Codes

Mochan Shrestha  
Department of Computer Science  
Wayne State University  
Detroit, MI 48202  
mochan@wayne.edu

Lihao Xu  
Department of Computer Science  
Wayne State University  
Detroit, MI 48202  
lihao@cs.wayne.edu

Abstract—Flash memory is a non-volatile data storage technology that stores data by trapping charge and can be reused by freeing the trapped charge with an internal erase operation. When flash memory cells are erased, there is a considerable negative impact on the longevity and performance of the device. To defer and minimize these erasures, the data can be encoded with a floating code that is able to store variable updates as cell increments. A \((n, q, k)\) floating code uses an array of \(n\) cells with \(q\) levels to store \(k\) binary variables. In this paper, we investigate the poset (partially ordered sets) structures derived from the various states of the \(n\) cells and \(k\) variables. These posets have fundamentally different structures that make designing floating codes a challenge, most notably their structure of their vertex covers. Based on the poset structure, we present an algorithm for constructing the floating code and prove that our algorithm does produce a valid floating code.

I. INTRODUCTION

Flash memory is a widely used non-volatile storage technology that operates by trapping charges inside a floating gate. An array of millions to billions of flash memory cells can be put on a chip to enable storage of large amounts of data. Additionally, the cells can be used in a variety of different ways so that data storage can be optimized with different applications in mind.

Charge can be added incrementally to a cell but to release the stored charge, an erase operation has to be performed. Repeated erasures break down the cell insulation and thus, after erasing a cell certain number of times, it is no longer able to hold charge and store data. Additionally, erasing can only happen simultaneously on a large group of cells called a block and any valid data in the block needs to be copied somewhere else. On top of that, compared to other flash memory operations, the erase operation is a few orders of magnitude slower. Thus, erasing has to be avoided as much as possible in order to increase the longevity and performance of flash memory devices.

To reduce and defer block erasures, a coding scheme called floating codes have been proposed. In this coding scheme, an array of variables and its updates is stored as various states of flash memory cells and variable updates are encoded as cell increments. Thus, each time data is updated it does not require erasing the flash memory cells but only requires cells to be incremented.

In this paper, we explore floating codes in terms of posets (partially ordered sets) where each update of a cell or variable provides a natural ordering for a poset. The cell increments produce a cell poset and the variable updates produce a variable poset, which have fundamentally different poset structures. In Section III, we describe the properties of the posets and the fundamental structural differences between the posets. Mainly, we show that two cell state vectors can only have one cover while variable state vectors have no such restriction. This property underlies the fundamental challenge to constructing floating codes because each variable state vector has to be represented by multiple cell state vectors.

In Section IV, we give a construction for a \(t_1\)-optimal floating code. We call a code \(t_1\)-optimal for \(l = 2\) if it can do \((n-k+1)(q-1)\) variable updates with a single cell increment (the fundamental limit of single cell increments possible). We first give an algorithm for the floating cell construction and then prove that the algorithm produces a valid floating code.

The construction we present in this paper is a novel way of creating floating codes. For arbitrary \(n, q, k\) parameters with \(l = 2\) (binary variables), the code’s rewriting capability is closer to the bound that any other known floating code. The deficiency of the code is \(O(kq)\), the best possible deficiency achievable. The previously best known floating code for arbitrary \(n, k, q\) was \(O(qk \log k)\) [19], [13]. In addition, our floating code construction achieves optimality for single cell increments portion of the optimality bound given in [12].

A. Related Work

Floating codes were first introduced in [12] as a generalization of the WOM model [14], and an optimality bound and constructions of optimal codes for arbitrary \(n, q, k = 2\) and \(l = 2\) were given. These floating codes were generalized for arbitrary \(k\) in [16] with a deficiency of \(O(k^2q)\) and later improved upon in [19], [13] to a deficiency of \(O(qk \log k)\). Floating codes for \(n, q, k\) and \(l = 2\) called indexed codes were given in [7] which are asymptotically optimal but imposes some restrictions on \(n, q, k\). Further bounds and more example codes for some values of \(n, q, k\) were given in [6]. Floating codes called covering codes for \(l > 2\) were given in [7]. Less restrictive forms of floating codes were studied in [2], [3].

A more general problem of coding for flash memory has been studied in [9], [5], [8], [10] and have led to related codes like rank modulation [11], buffer codes [19], [6] and WOM codes [17], [18], [1].
II. FLOATING CODES

A. Flash Memory Architecture

Data is stored in flash memory cells where each cell can be at \( q \) different levels \((0, 1, \ldots, q - 1)\). Each cell can be incremented up to level \( q - 1 \) but then cannot be decremented. The only possible way to reuse these cells, once the data stored in them are no longer needed, is to erase a block comprising of hundreds of thousands of these cells. After a block erasure, all the cells in the block have level 0.

B. Flash Memory Representation

Given \( n \) cells which have \( q \) levels, we call the vector

\[
[c_0, c_1, \ldots, c_{n-1}]
\]

the cell state vector and \( C^{n,q} \) the space of all state space vectors of length \( n \) and \( q \) levels.

Given \( c = [c_0, c_1, \ldots, c_{n-1}] \), \( \hat{c} = [\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{n-1}] \in C^{n,q} \) we say that \( c \leq \hat{c} \) if \( \forall i \in \{0, 1, \ldots, n - 1\}, c_i \leq \hat{c}_i \), i.e. at each co-ordinate of \( c \), the value at that co-ordinate is less than that of the value at the co-ordinate at \( \hat{c} \). With the above relation, the set of cell-state vectors forms a partially ordered set (poset) (the partially ordered set is of the product topology of \( n \) products of the ordered space \( \{0, 1, \ldots, q - 1\} \)).

By our assumption, the smallest change that can be made to a cell state vector is by incrementing the value of a single cell. The states that are possible after \( \gamma \) increments will be called the \( \gamma \)th generation. The cell poset and its generations are shown in Figure 1a for \( n = 3 \) and \( q = 3 \).

Let \( G \) be the function that gives the generation of the cell state vector \( G : C^{n,q} \to \mathbb{Z}^+ \) which is the weight of the cell state vector

\[
w(c) = G(c) = \sum_{i=0}^{n-1} c_i
\]

C. Data Variable Representation

Given \( k \) variables which can store \( l \) values (i.e. any value from the set \( \{0, 1, \ldots, l - 1\} \)) and we call the vector

\[
[v_0, v_1, \ldots, v_{k-1}]
\]

the variable vector and \( V^{k,l} \) the space of all variable vectors of length \( k \) each containing \( l \) values.

We assume that at each update step just one variable gets updated. Thus, if \( v = [v_0, v_1, \ldots, v_{k-1}] \) was updated to the variable vector \( \hat{v} = [\hat{v}_0, \hat{v}_1, \ldots, \hat{v}_{k-1}] \) and the updated variable was for variable \( i \), then \( \forall j \in \{0, 1, \ldots, k\} \setminus \{i\} \),

\[
v_j = \hat{v}_j
\]

As with the notions of generations with cell state vectors, we can impose a natural poset on the variable state vectors as well. We call the set of variable state vectors available after \( \gamma \) updates to be the \( \gamma \)th generation. However, several generations will share the same variable state vectors and so we distinguish between them by indexing to the number of variable updates.

We assume that we start with the all zero vector in the 0th generation and each subsequent generation updates only one variable. The 1st generation has variable state vectors that are non-zero in one location.

D. Mapping Functions

A floating code is essentially a mapping between a given variable state poset and a given cell state poset. In this mapping, a cell state vector represents some variable state vector. Since the same variable state vectors are repeated in many generations, the new variable state vector is indexed to the generation in the variable poset. When a variable is updated, to reflect the new variable vector, a new cell state vector higher than the current one is used to represent the new updated variable state variable.

1) Decode and Encode Mapping: The decode mapping \( D \) maps cell state vectors to variable state vectors, i.e.

\[
D : C^{n,q} \to V^{k,l}
\]

The mapping need not be injective but must be surjective. It needs to be surjective as every possible variable state vector must be represented by a cell state vector. It may not injective because multiple cell state vectors might be decoded as the same variable vector. The decode mapping function \( D \) might also not be a function because variable state vectors of different generations might be mapped to the same cell state vector.

We call the inverse of this mapping the encode mapping

\[
E : V^{k,l} \to C^{n,q}
\]

which gives us the cell state vectors that store a particular variable state vector. Note that this mapping need not be surjective since there could be cell state vectors that are unused.
2) Update Function: Given the current cell state vector $c$ and desired variable state vector $v$, the update function 
\[ U : C^{n,q} \times V^{k,l} \rightarrow C^{n,q} \]
gives the cell state vector $\hat{c}$ that reflects the desired variable state vector. However, the cell state vectors can be updated such that the new cell state vector is greater, i.e., 
\[ [\hat{c}_0, \hat{c}_1, \ldots, \hat{c}_{n-1}] > [c_0, c_1, \ldots, c_{n-1}] \]. If the above is not possible, then the cell is unusable until it has been erased.

In the poset form, the update function is canonical from the decode and encode mapping. Given the current cell state vector $c$, the update function finds the cell state $\hat{c}$ vector that contains the variable state vector and that $c < \hat{c}$, and that can be reached by the least number of updates.

3) Optimality: Since the cell poset allows for $n(q-1)$ updates, the largest possible number of variable updates is $n(q-1)$ and serves as a loose upper bound.

Let $t$ be the greatest number of updates possible before a cell becomes unusable and requires erasure. From above, 
\[ t \leq n(q-1) \]
We can further tighten the bound observing that each cell state vector must allow for $k(l-1)$ variable updates. To do the update in one cell increment, we must have $k(l-1)$ outgoing edges in the cell state vector. All the updates might be possible when a cell state vector has $n$ cells and when $n \geq k(l-1)$. However, if any of the cells reach state $q-1$, further updating of the cell is not possible and we lose an outgoing edge. Thus, from here, we can see that only $(n-k(l-1)+1)(q-1)$ variable updates are actually possible with a single cell increment because after $(n-k(l-1)+1)(q-1)$ updates, the number of cells that can be updatable is less than $k(l-1)$. Then, we then have to update using more than one increment and if we use two increments for each update, we can do further \( \left\lfloor \frac{k(l-1)-1}{2} \right\rfloor \) increments.

If $n < k(l-1)-1$, we have to start off doing each variable update with more than one cell state increment. There are \( n(q-1) \) cell state increments possible and each variable increment using two cell state updates, we have \( \left\lfloor \frac{n(q-1)}{2} \right\rfloor \) updates possible.

This above is the upper bound as given in [12]
\[
t \leq \begin{cases} 
\left( n-k(l-1)+1 \right)(q-1) & n > k(l-1) - 1 \\
\left\lfloor \frac{k(l-1)-1}{2} \right\rfloor & n < k(l-1) - 1 
\end{cases}
\] (1)

As defined in [16], the deficiency $\delta$ is
\[ \delta = n(q-1) - t \]
and from the definition of the deficiency we have that $\delta > 0$. Thus, finding the least upper bound on $t$ is finding the greatest lower bound on $\delta$. For binary variables ($l=2$), the best known bound on deficiency is $O(k(q-1))$ [13], derived from the upper bound for $t$ given in Equation 1. For $l>2$, the lower bound for the deficiency is $O(kq(l-1))$ (assuming $n > k(l-1) - 1$).

4) $t_1$ Optimality: As given in Equation 1, the number of variable updates possible is determined first by unit increments and then, by multiple increments. Let $t_1$ be the number of possible variable updates with a unit increment. We here assume $n > k(l-1) + 1$ otherwise $t_1 = 0$. We call a floating code $t_1$-optimal if
\[ t = n-k(l-1)+1)(q-1) \]
and when $k = 2$, $t = (n-k+1)(q-1)$. Basically, a floating code is a $t_1$-optimal code when it is possible to update variables using single cell increments until we reach the generation which has cell state vectors which doesn’t have enough updatable cells. A $t_1$-optimal code also has deficiency $O(k(q-1)(l-1))$ as $\delta = n(q-1)-t = k(l-1)(q-1)$.

For a $t_1$-optimal code to be a fully optimal code, we also have to be able to update variables in the last \( [k(l-1)-1](q-1) \) generations using two generations in the cell state poset for one variable update.

III. PROPERTIES OF THE POSETS

A. Cell State Poset

Let $\gamma$ denote the generation, i.e. $\gamma$ updates or increments have so far been made. Let $S_\gamma$ denote the set of cell state vectors that are of the $\gamma$th generation or the set of all cell state vectors possible after $\gamma$ increments.

The all zero cell state vector is the only vector in generation 0. The last generation has all the all-$s$ vector and that can be reached after $n(q-1)$ updates. Thus, there are $n(q-1)+1$ possible generations.

1) Counts of States Per Generation: Let $N_s(n, q, \gamma)$ denote the number of states in the cell state poset at generation $\gamma$ where cell state vectors are $n$ cells wide and each cell can take a maximum value of $q-1$. Since the total must sum up to all the states of the poset, we have
\[ \sum_{\gamma=0}^{nq-1} N_s(n, q, \gamma) = q^n \]
We would like to know the number of states in each generation. We look at an recursive definition. For the base case we have that
\[ N_s(1, q, \gamma) = \begin{cases} 1 & 0 \leq \gamma < q \\ 0 & \text{otherwise} \end{cases} \]
since if $n = 1$, each generation can only have 1 state and the recursive equation we have
\[ N_s(k, q, \gamma) = \sum_{i=0}^{q-1} N_s(k-1, q, \gamma-i) \]
where we look at cell state vectors of length $k-1$ and count all the possible vectors that sum to $\gamma$ if we added all cells from values of 0 to $q-1$.

This distribution follows the distributions of “s” [4] and for large values of $n$, it can be approximated by a normal distribution [4].
2) Properties of the Cell State Vectors: We first define the notion of a vertex cover.

**Definition III.1.** Let $c_1$ be a vertex in a poset $P$. We say the vertex $c_2 \in P$ covers $c_1$ ($c_1 \sqsubseteq c_2$) if

1) $c_1 < c_2$

2) $\forall v_3$ such that $c_1 < v_3$, we have $c_2 < v_3$ or that $c_2 \parallel v_3$, i.e., they are incomparable.

In our case, the cover of a cell state vector is the set of vectors that are in immediate next generation that can be reached after a single update.

**Theorem III.2.** Let $c_1$ and $c_2$ be two cell state vectors in the same generation, i.e., $w(c_1) = w(c_2)$. Then, at most one cell state vector covers both $c_1$ and $c_2$.

**Proof.** Suppose $c_3 = [c_0, c_1, \ldots, c_{n-1}]$ covers both $c_1$ and $c_2$ and that $c_1$ and $c_2$ were incremented at $i$ and $j$ to reach $c_3$ respectively. Then, $c_1 = [c_0, c_1, \ldots, c_{i-1}, c_{i-1}, c_{i+1}, \ldots, c_{n-1}]$ and $c_2 = [c_0, c_j, \ldots, c_{j-1}, c_{j-1}, c_{j+1}, \ldots, c_{n-1}]$. Let $c_4$ be the cell state vector obtained by incrementing $c_1$ at location $k$ where $k \neq i$. Then, $c_4$ covers $c_1$ but can never cover $c_2$ because $c_{4,i} = c_{i} - 1$ and $c_{2,i} = c_1$ and $c_{2,i} > c_{4,i}$. Thus, any other cell state vector that covers $c_1$ cannot cover $c_2$ and thus, $c_1$ and $c_2$ have just one cover. \qed

**Lemma III.3.** Let $c_1$ and $c_2$ be two cell state vectors in the same generation. Then, $c_1$ and $c_2$ have a common cover $c_3$ if and only if they must vary by $l$ at two locations.

We next define the root which is simply the opposite of a cover. If $c_1$ covers $c_2$, then $c_2$ is a root of $c_1$. We use this to show that if two vertices share a cover, then they must also share a root.

**Definition III.4.** Let $c_1$ and $c_2$ be vertices in the poset $P$. We say that $c_1$ is a root of $c_2$ if

1) $c_1 < c_2$

2) $\forall v_3 \in P$ such that $c_3 < c_2$, we either have $c_3 < c_1$ or that $c_3 \parallel c_1$ ($c_3$ or $c_1$ are incomparable).

**Theorem III.5.** Let $c_1$ and $c_2$ be cell state vectors in the same generation. Suppose that $c_1$ and $c_2$ share a common cover $\hat{c}$ ($c_1 \sqsubseteq \hat{c}$ and $c_2 \sqsubseteq \hat{c}$). Then, there must be a common root $r$ between $c_1$ and $c_2$. ($r \sqsubseteq c_1$ and $r \sqsubseteq c_2$). Furthermore, there can only be at most one such root.

**Theorem III.6.** Let $c_1$ and $c_2$ be cell state vectors in the same generation. Then, $c_1$ and $c_2$ have a common cover if and only if they have a common root.

**IV. Constructing Floating Codes**

Given a cell state poset $P$ and a variable state poset $V$, a floating code $F$ can be constructed by finding a decode mapping $D$.

The problems and challenging aspects of finding a decode mapping $D$ are the following:

1) the cell poset and variable poset have different shapes; the cell state poset is “round” in shape and the variable state poset “rectangular with a triangular bottom”

2) as from lemma III.2 and III.7, the structure of the covers are different; two cell state vectors can only have at most one cover whereas two variable state vectors can have more than one cover. For $l = 2$, it can have two covers from Theorem III.7.

**A. $t_1$-Optimal Floating Code for $l = 2$**

We next present the construction of $t_1$ optimal floating code for $l = 2$ and arbitrary $n,q$, and $k$. When $l = 2$, the variables are binary. By the definition of $t_1$ optimality for $l = 2$, we should be able to do $(n-k+1)(q-1)$ variable updates using single cell increments.

The construction is done using a recursive method where we build a $(n,q,k)$ floating code using $(n-1,q,k-1)$ floating code. Thus, we only need to define the $(n,q,1)$ floating code and then rest of the floating codes can be derived from there.

To illustrate the construction of the code, we start with two examples, $(3,3,2)$ and $(4,3,3)$ floating codes. We then present the recursive algorithm and then discuss the proof that the construction algorithm creates a valid floating code. Since we are only investigating the $t_1$ optimality, we are only interested in the first $(n-k+1)(q+1)$ generations. We will only look that portion of the poset and ignore for now the remaining $(k-1)(q-1)$ generations which would require multiple cell increments.

**B. Example Constructions**

a) $(3,3,2)$ Code: Figure 2 shows the $(3,2,2)$ floating code construction where Figure 2a shows the floating code poset only (without the cell state vectors that are not used). Figure 2b shows the floating code in the full cell state poset. The different colored vertices in the code represent sub-posets that are made from $(2,3,1)$ floating code.

The red-colored sub-poset $P_1$ in Figure 2 is the $(2,3,1)$ floating code but with a co-ordinate added to the end of both the cell and variable state vectors and set to 0. The blue sub-poset $P_2$ is $P_1$ but with $(0,0,1)$ added to all the cell state vectors in $P_1$, $(0,1)$ added to all the variable state vectors in $P_1$ and the top generation taken out. All the other sub-posets $P_3$, $P_4$, $P_5$ illustrated by the different colors are derived from $P_1$ by adding different roots to $P_1$. 
b) (4,3,3) Code: We construct the (4,3,3) floating code using the (3,3,2) floating code $F_1$ we constructed above. The red sub-poset $P_1$ in Figure 3 is $F_1$ with a co-ordinate added to the end of the cell state vectors and variable state vectors of $F_1$ and set to 0. The blue sub-poset $P_2$ is $P_1$ with the root vector $(0,0,0,1)$ added to the cell state vectors and $(0,0,1)$ added to the variable state vectors in $P_1$.

The interesting thing to observe is how $P_1$ and $P_2$ are connected together. Since $P_2$ starts one generation above $P_1$, we draw the edge from the vertices in the $i$th generation in $P_1$ to the $i+1$st generation of $P_2$. The number of vertices are equal since they are both from the $i$th generation of $P_1$ and we draw the vertices in the same order as they appear so there is no ambiguity. We will prove why this is always possible in Section IV-D.

The other thing to look at is the generation of the posets $P_2, P_3, \ldots, P_6$. They are all constructed from $P_1$ by adding a root $c_r$ to all the cell state vertices and $c_v$ to all the variable state vertices. The $c_r$ just alternates between $(0,0,1)$ and $(0,0,0)$. The method to choose the sequence of $c_r$ is described in IV-C.

C. Construction Algorithm

Next we describe the construction algorithm for generating floating code for arbitrary $n$, $q$, $k$ and $l = 2$. Here, the input is $n,q,k$ and the output is a floating code. The floating code is represented as an array of arrays where each generation is an array of vector vertices.

The outline of the algorithm is given in Algorithm 1. The floating code is generated recursively and to create a floating code of $(n,q,k)$ we use a floating code for $(n-1,q,k-1)$. Eventually, when we reach $k=1$ in the recursive algorithm, which means that there is only one variable, we can easily generate the floating code for $(\hat{n},q,1)$ for $\hat{n} = n-k+1$. We describe the procedure for generating $(\hat{n},q,1)$ floating codes in the function Create_1_Floating_Code in Section IV-C1.

After creating the $(n-1,q,k-1)$ floating code, we create the sub-poset $P_1$ by adding a new co-ordinate at the end of the cell state vector and variable state vector and marking them all as zeros. After that, we generate the sequence of cell state vector roots that we are going to use to generate the sub-posets. We are going to generate $(n-k+1)(q-1)$ sub-posets and need as many roots. Note, the variable state vector root sequence are zeros everywhere and alternate between 0 and 1 in the last co-ordinate.

\begin{algorithm}
\caption{Create $(n,q,k,2)$ $t_1$-optimal floating code}
\begin{algorithmic}[1]
\Function{Create}(n,q,k)
\State $F = \text{CreateFloatingCode}(n,q)$
\If{$k = 1$}
\State $F = \text{Create}_1\_\text{Floating\_Code}(n,q)$
\Else
\State $F_1 = \text{CreateFloatingCode}(n-1,q,k-1)$
\State $P_1 = \text{Add\_Zero\_Coordinates}(F_1)$
\State $F = P_1$
\State $R = \text{Generate\_Roots}(P_1)$
\For{$i = 1$ to $(n-k+1)(q-1)$}
\State $P_{i+1} = \text{Create\_Subposet}(R[i],i)$
\State $F = \text{Join\_Vertices}(F,P_{i+1})$
\EndFor
\EndIf
\EndFunction
\end{algorithmic}
\end{algorithm}

We build the rest of the floating code $F$ by iteratively generating the sub-posets $P_i$ from the roots generated above and then join the vertices of $P_i$ to the floating code. Once we have created $(n-k+1)(q-1)$ sub-posets and joined them all together, we have the complete floating code.

Next, let’s look at the functions in Algorithm 1 closely. In the entire floating code construction algorithm, the only variable is how we construct the $(\hat{n},q,1)$ floating code. This determines the roots and hence $P_1, P_2, \ldots$ and so, the final floating code. Given $(n,q)$ there are many ways to create a $(n,q,1)$ floating code but we will create the floating code in the method given below. This allows the floating code to be valid which we prove in Section IV-D.

1) Create_1_Floating_Code: We describe the algorithm for creating an $(n,q,1)$ floating code. We only have to store one binary variable so the vector state variable just alternates between a single value of 0 and 1. The number of generations is $n(q-1) + 1$ and each generation has just one cell state vector. We start by letting all the vectors be zero in the first generation and then continue by adding a single value to the left-most co-ordinate until it reaches $q-1$. Then, we
move to the next left-most co-ordinate. So, the sequence of
cell state vectors will be

\[(0, 0, 0, \ldots, 0), (1, 0, 0, \ldots, 0), \ldots, (q - 1, 0, 0, \ldots, 0),\]
\[(q - 1, 1, 0, \ldots, 0), \ldots, (q - 1, q - 1, 0, \ldots, 0),\]
\[\vdots\]
\[(q - 1, q - 1, \ldots, 1), \ldots, (q - 1, q - 1, \ldots, q - 1)\]
The vector state variable is just a sequence of 0s and 1s where
the first generation is 0. For the \(i\)th generation, it is 0 if \(i\) is
even and 1 if it is odd (here assuming generation starts at 0).

2) Add_Zero_Coordinates: After we have generated
a \((n - 1, q, k - 1)\) floating code, we need to add one more co-
ordinates to the end of the cell state and vector state variables
to make it into a \((n, q, k)\) code. The value for the added co-
ordinate is 0 and this gives us \(P_i\), the first sub-poset of the
floating code. We will generate the rest of the sub-posets from
\(P_i\). We also generate the floating code by attaching the sub-
posets to \(P_i\) and make a copy of \(P_i\) to \(F\) for it.

3) Generate_Roots: For \((n, q, k)\) floating code, we
have to generate \((n - k + 1)(q - 1)\) sub-posets and join them
together. So, we have to generate \((n - k + 1)(q - 1)\) roots to
generate each of the sub-posets. The roots should have weight
from 1, 2, \ldots, \((n - k + 1)(q - 1)\) and each should be a cell
increment of the previous one.

The root chosen to generate a sub-poset must be chosen so
that the vertices in the generated sub-poset do not occur in any
previous sub-posets. In other words, sub-posets do not share
vertices. We will prove that the algorithm to generate the roots
enforces this property in Section IV-D2. For example, in \(P_i\) we
have chosen the last co-ordinate of the cell state variables to be
0. If our root was 1 in the last co-ordinate, i.e., \((0, 0, \ldots, 0, 1)\),
then the vertices generated from this root would not have occurred
in the previous sub-posets. Similarly, \((0, 0, \ldots, 0, 2)\)
is another root and we can go on to \((0, 0, \ldots, 0, q - 1)\).
However, we need \((n - k + 1)(q - 1)\) roots and the above
is just \((q - 1)\) roots. Thus, we need to find \((n - k + 1)\) such co-
ordinates.

The algorithm to generate the roots relies on the algorithm
to find \((n - k + 1)\) co-ordinates called root locations which
we discuss below. For each of the co-ordinates, we generate
\(q - 1\) roots giving the total of the required \((n - k - 1)(q - 1)\)
roots. For each root location, we start with 1 and then generate
successive roots until we reach \(q - 1\) at that location. The root
locations used are largest co-ordinate first and then to lower
co-ordinates.

Let \(z_1, z_2, \ldots, z_{n-k+1}\) be the root locations in order of co-
ordinate. By our construction, \(z_{n-k+1} = n\) because we added
a co-ordinate at the \(n\)th location and set it to 0. Thus, the
sequence of generated roots will be

\[(0, \ldots, 0, 1), \ldots, (0, \ldots, 0, q - 1),\]
\[\ldots, (1_{z_{n-k-2}}, \ldots, q - 1), \ldots, (q - 1, z_{n-k-2}, \ldots, q - 1),\]
\[\vdots\]
\[\ldots, (q - 1, z_{n-k-2}, \ldots, q - 1)\]

In other words, let \(i = \alpha(q - 1) + j\) for \(0 \leq \alpha < n - k - 1\)
and \(0 < j < q\). Then,

\[r_{i, \beta} = \begin{cases}
0 & \beta < z_{n-k-\alpha} \\
\beta = z_{n-k-\alpha} \\
q - 1 & \beta > z_{n-k-\alpha} \\
0 & \beta \neq z_\eta \text{ for } \eta = 1, \ldots, n - k + 1
\end{cases}\]

Algorithm 2 Find \(n - k + 1\) root locations from \(P_i\)

1: function \(L = \text{FIND_ROOT_LOCATIONS}(P_i)\)
2: \(S = 0\)
3: for \(i = 1\) to \(q\) do
4: \hspace{1em} for each cell state vector \(v\) in generation \(i\) do
5: \hspace{2em} \(S = S + v\)
6: \hspace{1em} end for
7: end for
8: \(L = \text{find}(S = 0)\)

To find the root locations, we take the sum of all the cell
state vectors in the first \(q\) generations and the locations that
are zero serve as the root locations as given in Algorithm 2.
We give the details of why the algorithm works in the section
IV-D1.

4) Create_Subposet: We take the poset \(P_i\) and add the
given root to each cell state variable in the sub-poset. With
\(i = 1, \ldots, (n - k - 1)(q - 1)\), we set the last co-ordinate of
the variable state vector to 0 or 1 depending on if \( i \) is odd or even. For even \( i \), we set to 0 and for odd \( i \) we set to 1. Lastly, we take out the top \( i \) generations after the addition to get the sub-poset \( P_{i+1} \).

5) \textit{Join_Vertices:} Finally we join the vertices of the newly created poset \( P_i \) to the floating code. The root of the vertex \( P_i \) is joined to the root of the vertex \( P_{i-1} \) by creating an edge between the root of \( P_{i-1} \) and \( P_i \). Thus, all the vertices of \( P_i \) are one generation higher than the vertices of \( P_{i-1} \).

We call two vertices \( c_{i-1} \in P_{i-1} \) and \( c_i \in P_i \) corresponding vertices if they are from the same vertex in \( P_i \). In other words, \( c_{i-1} = v + r_{i-1} \) and \( c_i = v + r_i \) for some \( v \in P_1 \). Thus, \( c_{i-1} \) and \( c_i \) have the same positions within the sub-posets \( P_{i-1} \) and \( P_i \).

For connecting the rest of the vertices, just like the root, we join the corresponding vertices of \( P_{i-1} \) to \( P_i \) for each generation. This is best illustrated in the example constructions by black edges between different sub-posets and also Figure 4 which shows joining of the roots and corresponding vertices. The edges are connected in the order they appear in each generation from one sub-poset to the other. Each vertex in the sub-poset now connects to one other corresponding vertex and thus, the number of outgoing edges of the vertices has increased from \( k-1 \) to \( k \) for all the vertices. Note that each sub-poset \( P_i \) has \( i \) generations from \( P_0 + r_i \) removed from the top and so, the top generation of \( P_i \) does not have any outgoing edges.

The final sub-poset only has the root and thus no outgoing edges. After the final sub-poset is joined, we have the full floating code with each vertex has \( k \) outgoing edges.

D. Proofs for the Algorithm

The algorithm given above assumes a certain structure in the poset exists but we have to prove that they do for arbitrary values of \( n,q,k \). We also have to prove that after all the structures are constructed, we have a valid floating code by showing that each variable update can be satisfied with a single cell increment.

The structures that we have to verify are:

1) In the algorithm for finding the root locations, we need to prove that we will always find \((n-k+1)\) zero locations if we sum up all the cell state vectors in the first \( q \) generations of \( P_1 \).

2) When we generate the sub-posets from the root vectors, we have to guarantee that each cell state vector in the sub-poset does not exist in any other sub-poset.

3) When joining the sub-posets, the edges exist between the two corresponding vertices in the cell poset.

When the above have been proved, we can verify that the sub-posets \( P_1, P_2, \ldots, P_{(n-k+1)(q-1)+1} \) are disjoint (in terms of cell state vectors). Each vertex has \( k \) outgoing edges that are in the cell poset. After that, we have to prove that the above poset (a sub-poset of the \((n,q)\) cell poset) is a floating code by satisfying the property that for each cell state vector that represents a variable state vector, its outgoing edges lead to other cell state vectors that represent all the possible variable updates. When the above is proved, we can guarantee a valid floating code. For detailed proofs, please see [15].

1) Root Locations: The lemma below says that there are \( n-k+1 \) zeros when we sum the first \( q \) generations of the sub-poset \( P_1 \).

**Lemma IV.1.** Let \( F \) be a \((n-1,q,k-1)\) floating code as in the above algorithm and let \( P_1 \) be the poset created by adding one co-ordinate at the end of the cell state vector and assigning that co-ordinate 0 throughout the poset. The sum of all the cell state posets in the first \( q \) generations of \( P_1 \) has zeros at \( n-k+1 \) co-ordinates.

**Proof:** We will prove this by induction on \( k \), the number of variables. For the base case, let \( k = 2 \) and that \( P_1^2 \) be the sub-poset made from a \((n-1,q,1)\) floating code \( F \) by adding a co-ordinate to the end and assigning it 0. Then, \( P_1^2 \) has \( n \) co-ordinates and is a sub-poset of the floating code \((n,q,2)\). For the first \( q \) generations, we only update the first location and thus, the sum of the first \( q \) generations would have zeros in all the co-ordinates except the first. For \( F \), that is \( n-2 \) and for \( P_1 \) that is \( n-1 = n-2+1 \) which satisfies the case for \( k = 2 \).

Now, let us assume that \( P_1^k \), a sub-poset of \((n,q,k)\) floating code, generated by adding a co-ordinate to a \((n-1,q,k-1)\) floating code has the property that summing the first \( q \) generations of \( P_1^k \) has zeros in \( n-k+1 \) locations.

Consider \( P_1^{k+1} \) generated from the \((n,q,k)\) floating code \( F_k \) which is a sub-poset of \((n+1,q,k+1)\) floating code. The first \( q \) generations of the \((n,q,k)\) code is made from \( P_1^k \) and \( q-1 \) other posets derived from by adding a root. We used the last co-ordinate for the first \( q-1 \) root values. Thus, if we added all the vertices in the first \( q \) generations, we would have one less zero location because we have added root vectors that are non-zero in the last co-ordinate. Thus, the number of zeros of the first \( q \) generations of \( F_k \) is \( n-k \). When we create \( P_1^{k+1} \), we add a new zero to co-ordinate and increase the number of zeros. Thus, we have \( n-k+1 \) zeros in \( P_1^{k+1} \). Since, \( n-k+1 = (n+1) - (k+1) + 1 \), this is also true for \( k+1 \). Thus, by induction, it is true for all values of \((n,q,k)\).

2) Disjoint Sub-Posets: We next prove that the sub-posets generated do not have a vertex in common. This comes from the way we constructed the sub-posets. From Lemma IV.1, we have that there are \( n-k+1 \) zero co-ordinates. The roots are constructed so that these zeros are filled from right to left while these co-ordinates are filled up left to right in the floating code as we go higher up in the generations. From the construction, there never is a case when these two meet and so, two sub-posets cannot have the same cell state vector.

**Lemma IV.2.** Let \( P_1, P_2, \ldots, P_{(n-k+1)(q-1)+1} \) be the sub-posets as generated above. They do not have a cell state vector (vertex) in common.

3) Existence of the Edges for Joining Posets: We have said in the construction algorithm that we can draw edges between the poset \( P_i \) and \( P_{i+1} \) but haven’t proven that such edges exist
the way we have described it. The next lemma states that these edges must exist.

**Lemma IV.3.** Let \( r_i \) be a root and \( r_{i+1} \) be the next root by doing a cell increment. Let \( P_i \) be the sub-poset generated by \( r_i \) and \( P_{i+1} \) be the sub-poset generated by \( r_{i+1} \), done by adding the root to the sub-poset \( P_i \). We connect an edge between \( r_i \) and \( r_{i+1} \). Then, there must be an edge between a vertex in \( P_i \) to a vertex at \( P_{i+1} \). Furthermore, let \( v \in P_i \) and if \( v + r_{i+1} \in P_{i+1} \), then there is an edge between \( v + r_i \in P_i \) and \( v + r_{i+1} \in P_{i+1} \) (i.e. there is an edge between two corresponding vertices in the sub-posets \( P_i \) and \( P_{i+1} \)).

4) **Validity of the Floating Code:** Having established the structural properties of the floating code, we have to verify the final step that this is indeed a valid floating code. For each cell state vector, we have to be able to update each of the \( k \) variables by going up one generation in the cell state poset.

**Theorem IV.4.** The poset \( F \) generated as above is a valid floating code.

**Proof:** We have shown that the floating code generated when \( k = 1 \) is valid. Now, let us assume that the floating for \((n, q, k)\) is valid. We will show that the floating code for \((n+1, q, k+1)\) is also valid and prove by induction.

Since the \((n+1, q, k+1)\) floating code is made up sub-posets made from the \((n, q, k)\) floating code, and by assumption the \((n, q, k)\) code is valid, the first \( k \) variables can be updated on the \((n+1, q, k+1)\) code. We just have to prove that the \( k+1 \)-st variable update is possible.

For each cell state vector we construct, we join it to the next sub-poset where the variable state vector is exactly the same except the last variable alternates between 0 and 1, as described above. This allows for the update of the \( k+1 \)-st variable between 0 and 1 and thus, we are able to update all \( k+1 \) variables at each cell state poset. Therefore, the poset generated by the above algorithm is a valid floating code.

V. **Conclusions and Future Work**

In this paper, we have presented a poset view of floating codes and explored the properties of the cell and variable set posets in terms of their covers and roots. Next, we gave an algorithm for constructing a floating code that is \( t_1 \)-optimal (optimal for single cell increments and with \( O(kq) \) deficiency) for arbitrary \( n, q, k \) and \( l = 2 \) and proved that the algorithm indeed produces a floating code that maps \((n-k+1)(q-1)\) variable update to a single cell increment.

In the future, we want to extend it to arbitrary \( l \) and also extend the \( t_1 \)-optimal construction to a fully optimal construction by exploring multiple cell increments for variable updates. We would also like to give a decoding function that doesn’t require the construction of the entire poset structure that would be useful for large values of the parameters.

**References**


