On the Ehrenfeucht-Mycielski sequence

Grzegorz Herman* Michael Soltys†

January 16, 2009

Abstract

We introduce the inverted prefix tries (a variation of suffix tries) as a convenient formalism for stating and proving properties of the Ehrenfeucht-Mycielski sequence ([3]). We also prove an upper bound on the position in the sequence by which all strings of a given length will have appeared; our bound is given by the Ackermann function, which, in light of experimental data, may be a gross over-estimate. Still, it is the best explicitly known upper bound at the moment. Finally, we show how to compute the next bit in the sequence in a constant number of operations.

1 Introduction

In [3] Ehrenfeucht and Mycielski propose a pseudorandom[1] sequence, henceforth called the EM sequence. The sequence is computed starting with the single bit 0; the next bit of the sequence is computed by finding the largest suffix that occurs elsewhere, and then flipping the bit following the penultimate occurrence of this suffix. Ehrenfeucht and Mycielski conjectured that the ratio of zeros and ones approaches $\frac{1}{2}$; experimental data confirms this, and also shows that the convergence is very fast. However, a proof of this convergence is not known[2].

The EM sequence arises from the study of decision method used by all learning organisms. Suppose that we follow the procedure outlined in the above paragraph, but instead of flipping the bit following the penultimate occurrence of the sequence, we take it as is; this may be viewed as making our decision based on past experience, where we search for the event in the past that looked most like our current predicament. This method may seem very naïve, but [3] claim that more or less refined variants of this method are used by all learning organisms. From this point of view, if we flip the last bit (instead of taking it as is) the learning organism is "always wrong," and the resulting sequence is the most unpredictable one—it is *pseudorandom*.

On the other hand, from a string-algorithms point of view, the EM sequence is a new way of generating a disjunctive word (an infinite word that contains all finite words as substrings). This, together with the apparent gap between experimental and theoretical bounds on the appearance of all the words of a given length n, makes the EM sequence an interesting object of study.

*Computing and Software, McMaster University, grzegorz.herman@gmail.com

†Computing and Software, McMaster University, soltys@mcmaster.ca

[1]The sequence appears not to be statistically random, as experimental data suggests that it violates the law of iterated logarithm (as Mycielski remarked in a private communication).

[2]See [1] for progress on this conjecture.

Our contribution is three-fold. We present a convenient formalism for expressing and proving results about the EM sequence, namely the inverted prefix trie; this is done in section 3. The advantage of this formalism is that our proofs are significantly simpler than those of [4] and [5]. We provide an efficient algorithm for generating the sequence (not based on heuristics); given the data structure of an inverted prefix trie, we can generate the next bit in constant time; this is Theorem 4.5. Finally, we prove that the occurrence of all the strings of length k is bounded by the Ackermann function; this is Theorem 5.3. Our Ackermann bound is probably a gross over-estimate, as experimental data[3] indicates that this bound may be a single exponential in k.

The structural properties of the EM sequence that we present in section 3 have been shown already; our contribution in this section is to simplify them considerably thanks to the inverted prefix trie formalism. Proposition 4.2 matches Proposition 1 in [5]; Proposition 4.3 matches the Theorem of [3] and Lemma 1 in [5]; Corollary 4.4 matches Proposition 4.3 in [4] and Proposition 4 in [5]; Corollary 4.6 matches Proposition 4.2 in [4]; Proposition 4.7 encompasses Proposition 4.1 of [4] and Proposition 3 in [5]; Proposition 4.8 is a slightly stronger version of Theorem 4.6 in [4]; Corollary 4.10 matches Corollary 4.7 in [4]; Proposition 4.11 matches Corollary 4.8 in [4].

If we were to assume the result stated in [5, Lemma 3.5], we could immediately conclude an exponential upper bound, which is a vast improvement over the Ackermann function. However, the proof of [5, Lemma 3.5] relies on the preceding lemma of [5], whose proof in turn misses a case (in our framework, presented in section 4, this case consists in the weight 4 of a node being distributed 3:1 among both its children and followers). We do not see how to prove this missing case of [5], and so we believe that while the Ackermann bound is probably a gross over-estimate, it is the best legitimate bound at the moment.

2 Notation

Let us fix the **alphabet** $\Sigma = \{0, 1\}$. A **string** is a finite sequence of elements from Σ. We denote the **length** of s by $|s|$, as usual. A **substring** $s[i..j]$ (for $0 \leq i \leq j \leq |s|$) of s consists of those characters between (and including) positions $i + 1$ and j (in particular, $s[0..|s|] = s$ and $s[i..i] = \epsilon$). We will also write $s[i]$ as the shorthand for $s[i - 1..i]$ (i.e., the i-th character of s) and s^* for the **inverse** of s (i.e., s written "backwards"). **Prefixes** and **suffixes** of s are strings of the form $s[0..i]$ and $s[i..|s|]$ (for $0 \leq i \leq |s|$), respectively.

For any string $s \neq \epsilon$, **the EM sequence**[4] generated by seed s, and denoted by e_s, is defined as follows:

- for $0 < i \leq |s|$, $e_s[i] := s[i]$
- for $i \geq |s|$, $e_s[i + 1] := 1 - e_s[k + 1]$,
  where $e_s[j..k]$ is the last occurrence of the longest substring of $e_s[0..i - 1]$ which is a suffix of $e_s[0..i]$

[3] We have examined the first fifty million bits. The source code of the program is available on the second author's web page.

[4] The original sequence proposed by Ehrenfeucht and Mycielski in [3] is e_0.

3 Construction Algorithm

The first thing we are going to show about the EM sequence is how to generate it efficiently. The naïve algorithm that calculates the next bit directly from the definition, needs up to $O(n^2)$ operations[5] to generate the n-th bit. If some of the conjectures about the EM sequence are true (e.g., no substring of length k occurs twice before position $\Omega(2^k)$), then only $O(n \log n)$ operations are required. Searching for the longest matching suffix with the help of a border array (as used in the KMP pattern matching algorithm; see [2]) for the reverse string can push the complexity down to $O(n)$.

We now present an even better algorithm. An **inverted trie** (over Σ) is a finite, rooted tree, with edges labeled using elements of Σ, with every node having at most one outgoing edge for every label. In particular, for $\Sigma = \{0, 1\}$ it is a binary tree.

Every node u in the trie **represents** a string (denoted as $\hat{u}$) obtained by concatenating the labels on the path from u to the root. The trie itself represents the set of strings represented by all its nodes. Note that this set is closed under extracting suffixes, and that the root represents ϵ, the empty string.

An **inverted prefix trie (IPT)**[6] of a string s is the smallest inverted trie (i.e., an inverted trie with the least number of nodes) representing all prefixes of s. Since every substring of s is a suffix of some prefix, one can say that an IPT for s contains all substrings of s. For example, an IPT for $s = 01011$ is given in Figure 1.

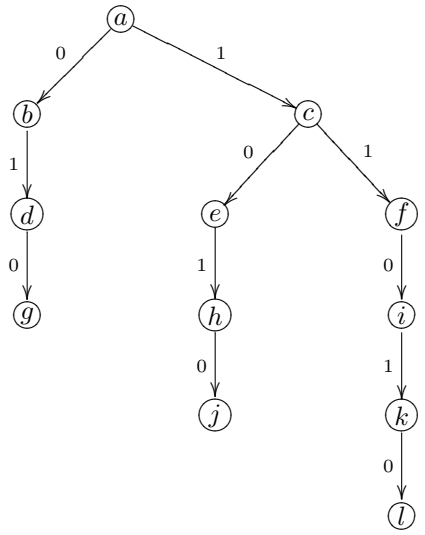

Figure 1: An inverted prefix trie (IPT) for $s = 01011$.

The prefixes of $s = 01011$ are $0, 01, 010, 0101, 01011$, and we can see that

[5]Following the common practice in string algorithm literature (see, for example, [6]), we treat operations on string indices as atomic (i.e., requiring constant time)—the actual time complexity ("measured" on a Turing Machine) is therefore higher by at least a factor of $\log n$.

[6]The more common formalism is that of **suffix tries**; we could use those and examine the string backwards instead. However, the EM sequence grows to the right, so IPTs are more intuitive for our application. See [6] for more background on suffix tries.

they can be obtained by traversing the tree in Figure 1 to the root starting at nodes b, e, g, j, l, respectively.

Given an s, we can now construct the sequence e_s as follows:

Algorithm 3.1 (Construction of e_s)**.**

1. set $w := \epsilon$
   let T be an inverted trie containing only the root
   label the root of T with 0
2. for $i = 1, 2, \ldots$:
3. if $i \leq |s|$, set $a := s[i]$
4. let $w := aw$
5. traverse T from the root, following the path described by w as far as possible, updating the labels of all visited nodes to be i
6. let k be the previous label of the last visited node
7. add a branch described by the remaining part of w below the last visited node, labeling all the nodes on that branch by i
8. output a as $e_s[i]$
9. set $a := 1 - e_s[k + 1]$

Proposition 3.2. The correctness of algorithm 3.1 follows from the following invariants, which hold after the i-th iteration:

- $w = e_s[0..i]^*$ (recall that w^* is the string w written backwards),
- the leaf of the added branch represents $e_s[0..i]$,
- T is the inverted prefix trie of $e_s[0..i]$,
- the label of any node u in T is the position of the end of the last occurrence of $\hat{u}$ in $e_s[0..i]$ (recall that $\hat{u}$ is the string obtained by reading all labels from u to the root),
- k is the position of the end of the longest substring of $e_s[0..i-1]$ which is also a suffix of $e_s[0..i]$.

The proof of the invariants in Proposition 3.2 is a straightforward induction on i, so we do not carry it out here.

How long does it take to generate the n-th bit of e_s? We need a number of operations proportional to the depth of T. If we keep T in the simple form described above, it is $\Theta(n)$ and we seem to have gained nothing over the KMP algorithm.

Given an IPT, we can perform the following space-saving operation on it[7]: given a node x with a single child y, and the edge (x, y) labeled with b, if x has a parent z (which is always the case unless x is the root), we delete y and prepend b to the label on (z, x). See Figure 2. We continue performing this action until there are no more nodes with a single child (other than possibly the root). We say that such a trie is **compressed**. See Figure 3 which is the compressed version of Figure 1. Note that in a compressed trie the edges are labeled by *strings* of characters.

In a compressed trie the cost will become bounded by the length of the matched suffix, which, from the experimental data, appears to be $O(\log n)$.

[7]This is the IPT counterpart of the regular suffix trie compression technique.

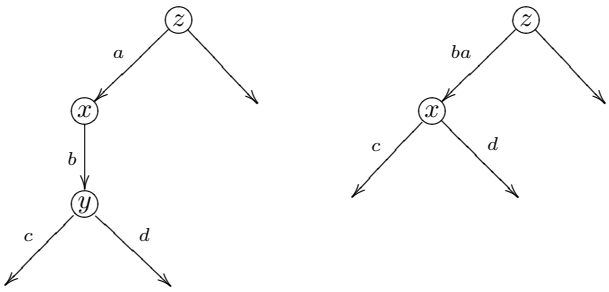

Figure 2: Eliminating nodes with an only child.

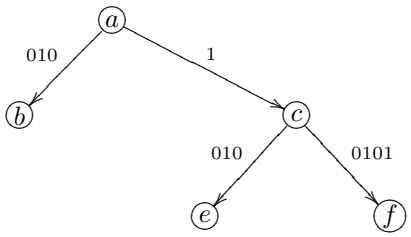

Figure 3: Compressed IPT for $s = 01011$ (shown in Figure 1).

But we are going to show in theorem 4.5 that we can do much better; a modification of algorithm 3.1 finds the n-th bit in only $O(1)$ operations (constant number of operations).

4 Properties of the Sequence

The inverted prefix trie provide an efficient method for generating the EM sequence, and exhibit, as it were, its internal structure. The results in this section will illustrate how well the IPTs captures this structure; the proofs are succinct and the formalism very handy.

Let us first introduce some useful terminology. Analogously to the usual tree notions of the **parent**, **0-child** and **1-child** of a node (in an inverted trie, representing the same string with a single bit removed or added at the beginning), we define the **precursor**, **0-follower** and **1-follower** of a node to be the nodes representing the same string with one bit removed or appended at the end. That is, given a node u, representing a string $\hat{u}$, its 0-follower is the node representing $\hat{u}0$, and its 1-follower is the node representing $\hat{u}1$. Note that in a tree the parent-children relationship is easy to see, while we may need to "jump" to a different part of the tree to find a precursor or follower. As usual, the **ancestor** and **descendant** relations are the reflexive and transitive closures of parent and child relations, respectively.

For example, in Figure 1, node c is the parent of node e and node f, node e is the 0-child of node c, and node f is the 1-child of node c. A precursor of node h ($\hat{h} = 101$) is node d ($\hat{d} = 10$). A 0-follower of node e ($\hat{e} = 01$) is node g ($\hat{g} = 010$) and a 1-follower of node j ($\hat{j} = 0101$) is node l ($\hat{l} = 01011$).

By "time i" we mean the moment after the i-th iteration of the loop in step 2 of algorithm 3.1. We call the last node visited in step 5 the **i-th insertion point**, and the leaf of the branch added in step 7—the **i-th pioneer** (it will eventually stop being a leaf, but we want to remember that it once was a leaf). Pioneers are precisely those nodes which are first to exist at their levels of the trie, and so precisely the nodes representing the prefixes of e_s.

The **weight** of a node (at time i) is the number of pioneers in its subtree—this is exactly the number of times the string represented by this node has appeared in the sequence. For example, in Figure 1 node c has three pioneers in its subtree: nodes e, j, l. Note that although node e is not a leaf, it is still a pioneer (as it was a leaf at time 2). Consequently, the weight of node c is 3, and indeed, the string 1 ($\hat{c} = 1$) occurs three times in the string $s = 01011$.

Using this new vocabulary we can now prove some interesting properties of the EM sequence[8].

Proposition 4.1. The following three statements are true:

1. the precursor of any node u appears before u does,
2. the $(i+1)$-st pioneer is a follower of the i-th,
3. if a node exists at time i, it has at least one follower at time $i+1$.

Proof. If $y = xa$ is the string inserted into the IPT at time $i+1$, then x was the string inserted at time i. Therefore, every node on the path from the root to the $(i+1)$-st pioneer (which represents y) has its precursor on the path to the i-th pioneer (which represents x).

For example, in Figure 1, the 4-th pioneer is node j, while the 3-rd pioneer is node g. Consequently, the precursors of nodes j, h, b, c (which are the nodes on the branch from the root to the 4-th pioneer) are the nodes g, d, b, respectively (which are the nodes on the branch from the root to the 3-rd pioneer).

All three claims in the proposition follow directly from this observation. □

Proposition 4.2. If a node u has weight 2 or more at time i, it has both followers by time $i+1$.

Proof. Consider the time j, where $j \leq i$, when the weight of u changes from 1 to 2. The j-th insertion point has to be a descendant of u. The next inserted bit is therefore different from the one that followed the first (which was the only previous) occurrence of $\hat{u}$. Thus u gains a new follower at time $j+1 \leq i+1$ and we have the proposition. □

Proposition 4.3. The IPT of e_s converges to the full binary tree.

Proof. Assume that there is a node whose left subtree remains bounded. Let u be the shallowest such node and let d bound the depth of its left subtree—refer to Figure 4.

Consider v, the precursor of u. Because of the way u was chosen, the left subtree of v is unbounded. Therefore we can find a node w in that subtree, of depth at least d and of unbounded weight. But one step after the weight of w reaches 2, by Proposition 4.2, it is going to have both followers. One of

[8] All proofs work directly for e_0 and e_1. Some technical modifications are required to take care of longer seeds, but we have decided to omit them for clarity.

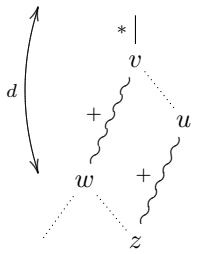

Figure 4: Proof of proposition 4.3. In this figure, and henceforth in all figures, we use solid and dotted lines to represent parent-child and precursor-follower relations, respectively. A label of "+" denotes a path with at least one edge, and "$*$" a possibly empty path. The squiggly line denotes descendants, i.e., not necessarily a child, but a grand-child, or a great-grand-child, etc.).

them (call it z) is inevitably in a left subtree of u, and of depth greater than d, yielding a contradiction.

A similar argument holds for the right subtree of u. This shows that every node in the tree eventually obtains a left and right child, and proves the statement of the proposition. □

Corollary 4.4. Every pioneer becomes an insertion point exactly twice, and every other node exactly once.

Proof. Pioneers are created as leaves (with no children), and all other nodes are created with a single child. Being an insertion point adds a single new child. According to Proposition 4.3, every node will eventually have exactly two children. Combining the above we reach the conclusion. □

Theorem 4.5. The n-th bit of the EM sequence can be computed in $O(1)$ operations, i.e., in constant time.

Proof. Let us now consider a different way of finding the $(i+1)$-st insertion point u than the one used in Algorithm 3.1.

Let w be the i-th insertion point; by the same observation that we used to prove Proposition 4.1, we know that every node on the path from the root to the $(i+1)$-st pioneer has its precursor on the path to the i-th one. Thus u is a follower of some ancestor (which we call v) of the i-th insertion point (which we already called w).

The weight of w at time i is at least 2, as it is an insertion point. If it is 3 or more, it had to be at least 2 at time $i-1$. But then by time i it would have both followers and we would have $v = w$. Otherwise, w has weight exactly 2 and, because of the way the bits are chosen, its appropriate follower is just about to be created. In this case v has to be the closest ancestor (an ancestor that is farthest from the root) of w with larger weight, which is precisely the first insertion point encountered while traversing the IPT upwards.

This suggests that if the compressed IPT representation was enriched by the follower and weight information, the next insertion point could be found in a

fixed number of operations[9]. Moreover, as weights of 3 or more are equivalent for the working of this algorithm, and the next bit can be found by looking at the missing follower of the next insertion point, no information on the path to the root would need to be updated. This yields an algorithm that generates the n-th bit of the EM sequence in $O(1)$ operations. □

Corollary 4.6. The following conditions are equivalent:

1. The depth of insertion increases between steps i and $i+1$.
2. The weight of the i-th insertion point (at time i) is at least 3.
3. The $i+1$-st insertion point is a follower of the i-th.

Proposition 4.7. The following conditions are equivalent:

1. The insertion depth attains a new record value at time $i+1$.
2. The i-th insertion point is a pioneer, matching the second time.
3. The $i+1$-st insertion point is a pioneer, matching the first time.

Proof. We prove three implications.

$[1 \implies 2]$ The insertion depth has to increase to attain a record value, which means that the i-th insertion point needs to have a weight of at least 3. But it cannot have former insertion points as descendants (because then the new depth would not be a record), so both its children have weight at most 1 each. Thus it has to be a pioneer and it has to be its second time to match.

$[2 \implies 3]$ Consider the two moments when a pioneer becomes an insertion point. The first time, the bit appended to the sequence is different than the one that followed the prefix represented by this pioneer. Therefore the second time it becomes an insertion point, a longer prefix (corresponding to the next pioneer) is going to match for the first time.

$[3 \implies 1]$ Assume that the $i+1$-st insertion depth d is not a record. Therefore there must have been at least d record-breaking insertions before (as by Corollary 4.6, the insertion depth can only increase by 1 at a time) Each of them had (by $[1 \implies 2]$) to follow the matching of a (different) pioneer, so each pioneer of depth up to d had to be matched. But then the $i+1$-st insertion point (being a pioneer of depth d) would have matched already, contradicting the assumption that it is its first time to match. □

We can thus see that the pioneers "regulate" the increase of the insertion depth. We can therefore divide the whole history of the inverted prefix trie into "rounds" during which the record insertion depth remains constant. These are exactly the periods between the first and second match of a pioneer.

Proposition 4.8. Once a node u becomes an insertion point before its proper ancestor v, no node deeper than v can match before v does.

[9]The precursor information analogues the suffix links of the suffix tree construction algorithms (see Section 5.2 of [6]). Followers can be managed simultaneously—whenever a node u is identified as a precursor of v, v can be linked as a follower of u.

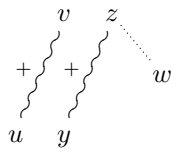

Figure 5: Proof of proposition 4.8

Proof. (refer to Figure 5) Assume to the contrary that u is the first node to break the above. We can see that the weight of u, when it is an insertion point, equals 2—otherwise some descendant of u (of which v is also a proper ancestor) would become an insertion point earlier, contradicting the way u was chosen. Thus we know that the next insertion point after u cannot lie deeper than v.

Now let w be the earliest (after u) insertion point coming both before and deeper than v. Note that, as the insertion depth can only increase by 1 at a time, w has to be exactly one level deeper than v. As the insertion depth has increased but not reached a new record value, the precursor of w (call it z) was the previous insertion point with weight 3, but was not a pioneer. But then its proper descendant y had to match before z did. It cannot have come before u, because then u would come between it and z (conflicting the choice of u). It cannot have come after u either, due to the choice of w. The resulting contradiction proves the claim. □

Combining the last two propositions we immediately get the following.

Corollary 4.9. At every threshold (moment between rounds), the insertion points form a connected fragment of the full binary tree, with respect to both parent-child and precursor-follower relationship (i.e., every ancestor and precursor of every insertion point has already become an insertion point itself)[10].

Corollary 4.10. A node cannot become an insertion point after its weight reaches 3.

Proof. When the weight of a node u reaches 3, it must have two insertion points as descendants. But then whichever of them came first, the other would have come before u does, contradicting Proposition 4.8. □

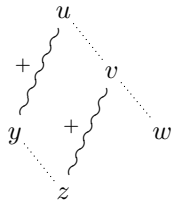

Figure 6: Proof of proposition 4.11

[10]Looking at the EM sequence, it means that at every threshold, if a string s has matched, all its substrings have matched as well, and they are not going to match again.

Proposition 4.11. The insertion depth cannot increase twice in a row.

Proof. Let u, v and w be the insertion points at times i, $i+1$ and $i+2$, and assume that the insertion depth increases both between u and v and between v and w. The weight of v at time $i+1$ has to be 3. It cannot be a pioneer, as then so would be u, but the insertion depth could not attain a record value at time $i+1$, as v would need to be an insertion point before. Thus there is a proper descendant z of v having weight 2 at time i. Its precursor y has to be a former insertion point, with weight 3 at time i. But, as it is also a descendant of u, this would force the weight of u to be 3 before it matched, which cannot happen. □

5 An Upper Bound

In Proposition 4.3 we have shown that every string appears in the EM sequence. We are now going to present an upper bound on when this happens.

Let $f(i,j)$ be the least position in e_0 by which every string of length i is guaranteed to occur j times. Let $g(i,j)$ be the least number l such that one step after any string s of length i has appeared l times, both $s0$ and $s1$ have appeared at least j times each. In terms of inverted prefix tries, $g(i,j)$ bounds the weight a node u of depth i can reach one step before both its followers reach weight j.

Lemma 5.1. The function $g(i,j)$ can be recursively bounded from above as follows:

$$g(i,1) = 2 \tag{1}$$

$$g(i,j+1) \leq f(i,g(i,j))^2. \tag{2}$$

Proof. (1) follows trivially from Proposition 4.2. To show (2), take a node u of depth i and consider the moment when it reaches weight $g(i,j)$ (and so, both its followers have weight j already). By definition of f, this has to happen no later than at time $f(i,g(i,j))$, and so the cardinality of u's subtree is bounded by $g(i,j)f(i,g(i,j))$.

Looking into the future of u's subtree, let us bound the number of times an insertion can happen at one of the currently existing nodes. Each of the pioneers (and there are $g(i,j)$ many of them) can become an insertion point at most twice, every other node at most once. Therefore when the weight of u exceeds $g(i,j)(f(i,g(i,j))+1)$, a node that does not currently exist will become an insertion point. It will immediately gain both followers, and each of them will contribute to the weight of the appropriate follower of u, thus forcing them both to have weight at least $j+1$, as required. From this discussion we obtain:

$$g(i,j+1) \leq g(i,j)(f(i,g(i,j))+1). \tag{3}$$

Also note that for all $i,j \geq 1$,

$$j+1 \leq 2^i + j \leq f(i,j). \tag{4}$$

We are now ready to prove the main result:

$$\begin{aligned} g(i,j+1) &\leq g(i,j)(f(i,g(i,j))+1) && \text{(by (3))} \\ &= g(i,j)+g(i,j)f(i,g(i,j)) \\ &\leq f(i,g(i,j))+g(i,j)f(i,g(i,j)) && \text{(by (4))} \\ &= f(i,g(i,j))(g(i,j)+1) \\ &\leq f(i,g(i,j)f(i,g(i,j)) && \text{(by (4) again)} \\ &= f(i,g(i,j))^2, \end{aligned}$$

which ends the proof. □

Lemma 5.2. The function $f(i,j)$ can be recursively bounded from above as follows:

$$f(1,j) \leq 2^{j+1} \tag{5}$$
$$f(i+1,j) \leq f(i,g(i,j))+1. \tag{6}$$

Proof. We first deal with (5). By time $2^j + j < 2^{j+1}$ a string of length j had to appear twice, and so the insertion depth must have reached j. But before that happens, the first j pioneers have to match[11] (and so have both followers), which guarantees at least j occurrences of both 0 and 1.

To show (6), consider a string ua ($|u| = i$). One step after u has appeared $g(i,j)$ many times (which is not later than at time $f(i,g(i,j))$), both $u0$ and $u1$ (and so ua, being one of them) must appear at least j times. □

Consider the Ackermann function $A : \omega \longrightarrow \omega$:

$$A(0,j) := j+1 \tag{7}$$
$$A(i+1,0) := A(i,1) \tag{8}$$
$$A(i+1,j+1) := A(i,A(i+1,j)). \tag{9}$$

Theorem 5.3. $f(i,j), g(i,j) \leq A(4i,j)$

Proof. The proof is by induction on i. More precisely, we prove the statement $\alpha(i)$, where

$$\alpha(i) := (\forall j \geq 1)[f(i,j) \leq A(4i,j) \ \wedge \ g(i,j) \leq A(4i,j)], \tag{10}$$

by induction on i.

For the basis case of $i = 1$, note that $f(1,j) < 2^{j+1}$ while $A(4,j) > A(3,j) = 2^{j+3} - 3$. Also $g(1,j) < f(1,g(1,j-1))^2 < 2^{2g(1,j-1)}$, so $g(1,j)$ is bounded by a stack of 4's of height j. On the other hand, $A(4,j)+3$ is a stack of 2's of height $j+3$. It is an easy proof by induction to show that $g(1,j) < A(4,j)$.

[11] See the discussion following Proposition 4.7

For the induction step, assume $\alpha(i)$ (i.e., (10)), and show $\alpha(i+1)$. We show first that $f(i+1,j) \leq A(4(i+1),j)$, for $j \geq 1$:

$$\begin{aligned}
f(i+1,j) &\leq f(i,g(i,j))+1 && \text{(by (6))}\\
&= A(4i,g(i,j))+1 && \text{(by (10), i.e., IH)}\\
&= A(4i,A(4i,j))+1 && \text{(by (10), i.e., IH)}\\
&\leq A(4i+3,A(4i+4,j-1)) && \text{(properties of Ackermann fn.)}\\
&= A(4i+4,j) && \text{(by (9))}\\
&= A(4(i+1),j). && (11)
\end{aligned}$$

We show second that $g(i+1,j) \leq A(4(i+1),j)$. When $j=1$, then $g(i+1,1) = 2 < A(4(i+1),1)$. For $j \geq 2$ we have:

$$\begin{aligned}
g(i+1,j) &\leq f(i,g(i,j-1))^2 && \text{(by (2))}\\
&\leq A(4i,g(i,j-1))^2 && \text{(by (10), i.e., IH)}\\
&\leq A(4i,A(4i,j-1))^2 && \text{(by (10), i.e., IH)}\\
&\leq A(4i+3,A(4i+4,j-1)) && \text{(properties of Ackermann fn.)}\\
&= A(4i+4,j) && \text{(by (9))}\\
&= A(4(i+1),j).
\end{aligned}$$

which finishes the proof. □

Thus we have shown that the least position in e_0 by which every string of length i is guaranteed to occur j times is bounded above by the Ackermann function $A(4i,j)$. In light of experimental data this seems to be a gross overestimate. Still, it is the best explicitly known upper bound at the moment.

6 Acknowledgments

We are very grateful to Jan Mycielski for many discussions and for feedback on this paper. We are also indebted to two anonymous referees for insightful comments that greatly improved the presentation of this paper.

References

[1] John C. Kieffer; W. Szpankowski "On the Ehrenfeucht-mycielski balance conjecture" *DMTCS Proceedings from the Conference on Analysis of Algorithms*, pages 19–28, 2007.

[2] Donald Knuth; James H. Morris, Jr; Vaughan Pratt "Fast pattern matching in strings" SIAM Journal on Computing, 6 (1977), vol. 2, pp. 323-350

[3] Andrzej Ehrenfeucht; Jan Mycielski "A Pseudorandom Sequence—How Random Is It?" American Mathematical Monthly, 99 (1992), pp. 373-375

[4] Terry R. McConnell "Laws of Large Numbers for Some Non-Repetitive Sequences" Technical report, Syracuse University, Department of Mathematics (1996)

[5] Klaus Sutner "The Ehrenfeucht-Mycielski sequence" Lecture Notes in Computer Science, 2759 (2003), pp. 282-293

[6] William Smyth "Computing Patterns in Strings" Addison Wesley; 1st edition (2003)