Anchor Points Algorithms for Hamming and Edit Distance

Foto Afrati[†*] afrati@softlab.ece.ntua.gr
Anish Das Sarma anish.dassarma@gmail.com
Anand Rajaraman datawocky@gmail.com
Pokey Rule[‡] pokey@stanford.edu
Semih Salihoglu[‡] semih@cs.stanford.edu
Jeffrey Ullman[‡] ullman@gmail.com

[†]National Technical University of Athens, [‡]Stanford University

ABSTRACT

Algorithms for computing similarity joins in MapReduce were offered in [2]. Similarity joins ask to find input pairs that are within a certain distance d according to some distance measure. Here we explore the "anchor-points algorithm" of [2]. We continue looking at Hamming distance, and show that the method of that paper can be improved; in particular, if we want to find strings within Hamming distance d, and anchor points are chosen so that every possible input is within Hamming distance k of some anchor point, then it is sufficient to send each input to all anchor points within distance $(d/2)+k$, rather than $d+k$ as was suggested in the earlier paper. This improves on the communication cost of the MapReduce algorithm, i.e., reduces the amount of data transmitted among machines. Further, the same holds for edit distance, provided inputs all have the same length n and either the length of all anchor points is $n-k$ or the length of all anchor points is $n+k$. We then explore the problem of finding small sets of anchor points for edit distance, which also provides an improvement on the communication cost. We give a close-to-optimal technique to extend anchor sets (called "covering codes") from the $k=1$ case to any k. We then give small covering codes that use either a single deletion or a single insertion, or – in one algorithm – two deletions. Discovering covering codes for edit distance is important in its own right, since very little work is known.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Distributed databases, Parallel databases*

[*]This work was supported by the project Handling Uncertainty in Data Intensive Applications, co-financed by the European Union (European Social Fund) and Greek national funds, through the Operational Program âĂIJEducation and Lifelong LearningâĂİ, under the program THALES.

General Terms

Theory

1. INTRODUCTION

Fuzzy or *similarity* joins is the problem of finding pairs of strings from a given corpus that are within a certain distance from each other according to some distance measure. Computing fuzzy joins efficiently and at scale in distributed systems is important for many applications, such as collaborative filtering for recommendation systems on large consumer data [10], entity recognition among labeled records in the web [13], clustering large-scale genetics data [9], and many others. Reference [2] introduced the *anchor-points* algorithm for computing fuzzy joins in the MapReduce system [8] under Hamming distance. This algorithm is based on finding a set of strings, called the *anchor points*, with the property that all strings in the input corpus have small distance to some anchor point. The problem of finding a set of anchor points for Hamming distance has been studied under the term "covering codes."[1] In this paper we first improve the *anchor-points* algorithm from [2] for Hamming distance. We then describe anchor-points algorithms for edit distance and show the existence and explicit construction of nontrivial edit-distance covering codes.

The specific problem considered in [2] is the following: Given a set of input strings of fixed length n over some alphabet, find pairs of strings that are within Hamming distance d, i.e., differ in at most d positions. The anchor-points algorithm described there uses a set A of anchor-point strings such that all strings of length n are within distance d of some anchor point in A. The algorithm operates by creating one reducer for each anchor point. The mappers send each string w to the reducer for each anchor point at Hamming distance at most $2d$ from w. Each reducer then searches for strings at distance up to d from each other, among the strings it has received. While not always the best algorithm, [2] showed that for some inputs and parameters, anchor-points is the best among known algorithms.

In this paper, we improve on this approach in three ways.

1. We generalize the algorithm and decouple the desired Hamming distance d from the maximum distance k between any string and its nearest anchor point.

2. We show that it is possible to reduce the radius $2d$ used in the algorithm from [2] to $3d/2$ and still find all

[1]We thank George Varghese for pointing out the term for this concept.

pairs of input strings at distance up to d.

3. We give a construction for finding near-optimal sets of anchor points, rather than relying on the nonconstructive existence proof in [2].

In addition, we describe anchor-points algorithms for edit distance. We focus on the case when all input strings are of fixed length n and we want to find all pairs of strings u and w that are at edit distance $2d$, i.e., u can be turned into w by a combination of d insertions and d deletions. It turns out to be more difficult to construct sets of anchor points for strings at a fixed edit distance than within a fixed Hamming distance. However, we describe an explicit construction of a set of anchor points for edit distance 2 that is within a constant factor of the best possible. This construction can be used to find sets of anchor points for any edit distance, in a manner similar to the way we construct sets of anchor points for arbitrary Hamming distance, as hinted at in item (3) above.

1.1 Related Work

A number of recent works have explored MapReduce algorithms for fuzzy joins– finding all pairs of elements from some input set that are within a similarity threshold. Usually, the notion of similarity is that the two elements are within distance d according to some distance measure. [12] tries to identify similar records based on the Jaccard similarity of sets, using the length/prefix-based methods of [5], combined with the positional and suffix filtering techniques of [14], and then parallelizes these techniques using MapReduce. [4] shows improvements over [12] by using two MapReduce jobs rather than one. [11] gives multiround algorithms for fuzzy join.

There is a significant literature regarding sets of anchor points for Hamming distance; these sets are called "covering codes." We mention some sources and related work in the next section.

2. COVERING CODES

A *covering code* for length n and distance k is a set C of strings of length n over an alphabet of size a such that every string of length n is within Hamming distance k of some member of C. The question of how small a covering code for n and k can be is a hard combinatorial problem that has been resolved only for small n, k, and a [1, 6]. A modification of the problem called "asymmetric" covering codes has been considered for the binary alphabet [7, 3]. An "asymmetric" binary covering code covers every bit string w of length n by changing at most k 1's of w to 0's. As for covering codes in the original formulation, lower and upper bounds on the sizes of asymmetric covering codes are known only for small values of n and k.

EXAMPLE 2.1. *In this example we assume the binary alphabet $\{0, 1\}$ where a, the alphabet size, is 2. For $k = 1$, the Hamming code itself provides a covering code of size 2^{n-m} if $n = 2^m - 1$. It is easy to show that this size is the best possible, since the Hamming code is* perfect*; that is, every bit string of length n is covered by exactly one codeword. Hamming codes exist only when the length n is one less than a power of 2.*

As another example, there is a covering code of size 2 for $n = 5$ and $k = 2$: $\{00000, 11111\}$. That is, any bit string of length 5 either has at most two 1's, in which case it is distance at most 2 from 00000, or it has at most two 0's, in which case it is at distance 2 or less from 11111. This code also happens to be perfect; each string of length 5 is covered by exactly one of the two strings.

Unfortunately, sometimes there is no perfect covering code. For instance, for $n = 6$ and $k = 3$, the all-0's and all-1's strings again form a covering code of size 2. It is easy to see that there is no covering code of size 1, so two codewords is the smallest possible size for a code. However, in this case, the strings with three 0's and three 1's are covered by both codewords.

2.1 Constructing Covering Codes for Larger Distances by Cross Product

Although we cannot offer a general formula for the size of the smallest covering code for n, k and alphabet of size a, we can give a construction that is not too far from what is possible. We start with the smallest possible covering code for length n/k and distance 1 over the given alphabet and extend it as follows.

THEOREM 2.2. *If C is any covering code for length n/k and Hamming distance 1 over an alphabet of size a, then $C' = C^k$ is a covering code for n and Hamming distance k over the same alphabet.*

PROOF. Given a string w of length n, write $w = w_1 w_2 \cdots w_k$, where each w_i is of length n/k. We can change at most one position of each w_i to get a string x_i in the covering code C. The concatenation of $x_1 x_2 \cdots x_k$ is a string in C'. □

EXAMPLE 2.3. *Let $n = 28$, $k = 4$, and $a = 2$. Then $n/k = 7 = 2^3 - 1$, so $m = 3$. There is a Hamming code of length $2^3 - 1 = 7$, with $2^{(n/k)-m} = 16$ members. Thus, there is a covering code for $n = 28$ and $k = 4$ with $16^4 = 2^{16}$ members. That is, fraction $2^{-km} = 2^{-12}$, or $1/4096$ of the 2^{28} bit strings of length 28 is in the covering code constructed by Theorem 2.2. In comparison, the lower bound, which is not necessarily attainable, states that one in $\sum_{i=0}^{4} \binom{28}{i}$, or 1 in 24,158 of the binary strings of length 28 must be in any covering code for $n = 28$ and $k = 4$.*

3. AN IMPROVED ANCHOR-POINTS ALGORITHM

Suppose w and x are two bit strings of length n, and the Hamming distance between them is d. Assume for convenience that d is even. Let y be any string at distance $d/2$ from both w and x. There is at least one such y, since we can find it by starting with w, choosing any $d/2$ bits where w and x disagree, and flipping those bits in w to agree with x.

EXAMPLE 3.1. *Let $w = 01010$ and $x = 11000$. Then $d = 2$, since w and x differ in only their first and fourth bits. There are two possible y's. Each is obtainable by starting with w and flipping either the first or fourth bits. That is, one possible y is 11010 and another is 01000.*

The observation above proves the following theorem.

THEOREM 3.2. *If C is a covering code for n and k, then any two bit strings that are within distance d are within $k + d/2$ distance from some member of C.*

PROOF. Let w and x be the two strings at distance d. As above, we may find y at distance $d/2$ from both w and x. Since C is a covering code, there is a member of C, say z, at distance at most k from y. By the triangle inequality, w and x are each within $k + d/2$ distance from z. □

Let C be a covering code for n and k. The improved anchor-points algorithm using C to find pairs of bit strings at distance d works as follows. As before, there is one reducer for each member of the set C of anchor points. The mappers operate as follows. For any input string w, find all the anchor points at Hamming distance at most $k+d/2$ from w and send w to the reducer for each such anchor point. The reducers find all pairs of received bit strings that are at distance up to d. As in [2], the reducers can avoid emitting a pair more than once by checking, for each pair found, that there is no lexicographically earlier anchor point that is distance at most $k + d/2$ from both strings. The proof that all pairs of distance d are found in this way follows from Theorem 3.2.

Recall from Section 1 that the algorithm from [2] picks a covering code of distance d and sends every string w to the anchor points that are at distance up to $2d$ from w. In the new algorithm, if we pick the same covering code, i.e. pick $k = d$, we improve over the algorithm in [2] by sending each input string w to all anchor points within $3d/2$ radius. This reduces the overall communication of the algorithm from $O\big(IB(2d)/B(d)\big)$ to $O\big(IB(3d/2)/B(d)\big)$, where I is the number of input strings (all of length n), and $B(r)$ is the "ball of radius r": the number of strings that can be obtained by flipping at most r bits from a given string and is equal to $\sum_{i=0}^{r} \binom{n}{i}$. With some algebra, it can be shown that the ratio of the communication used by the two algorithms to find pairs of strings within Hamming distantce d is at most $(2d/n)^{d/2}$, which is tiny when n is much larger than d, as it normally is.

Another way to view the improvement of the new algorithm is the following. By incurring the same communication that the algorithm from [2] incurs for finding strings within Hamming distance d, the new algorithm can find strings at distance up to $2d$ (i.e. by picking $k = d$ and sending every string to anchor points within distance $k + 2d/2 = 2d$).

4. COVERING CODES FOR EDIT DISTANCE

We can use some of the Hamming-distance ideas to develop an anchor-points algorithm for edit distance. However, with edit distance, we can cover strings by using insertions, deletions, or a combination of these. We shall focus on covering codes that cover strings of a fixed length, using only insertions or only deletions, so the covering code itself has strings of a fixed length.

DEFINITION 4.1. *(Insertion-k Covering Code): A set C of strings of length $n + k$ is an insertion-k covering code for length n, distance k, and alphabet Γ if for every string w of length n over Γ we can insert k characters from Γ into w and produce some string in C. Equivalently, for every w of length n we can find some string x in C such that it is possible to delete k positions from x and produce w. We say that x* covers *w in this case.*

DEFINITION 4.2. *(Deletion-k Covering Code): Similarly, we say a set C of strings of length $n - k$ is a* deletion-k covering code *for length n, distance k, and alphabet Γ if for every string w of length n over Γ we can delete k positions from w and produce some string in C. Again, we say that x covers w if so.*

Throughout our analyses we assume that $|\Gamma| = a$ and w.l.o.g. the letters in Γ are the integers from 0 to $(a - 1)$. Finding covering codes for edit distance is harder than for Hamming distance, since there is no convenient "perfect" code like the Hamming codes to build from. One tricky aspect of working with edit distance is that certain deletions and insertions have the same effect. For instance, deleting from any of the three middle positions of 01110 yields 0110. When we want to develop a covering code, this phenomenon actually works against us. For example, if we want a deletion code for $n = 5$, $k = 1$, and the binary alphabet, then 00000 requires us to have 0000 in the code, since every deletion of one position from 00000 yields 0000. Likewise, the code must have 1111; there are no options.

4.1 Elementary Lower Bounds

There are simple arguments that say a covering code cannot be too small; these are obtained by giving an upper bound on the number of strings one codeword can cover. For example, [2] shows that a string of length $n - 1$ over an alphabet of size a yields exactly $n(a - 1) + 1$ strings by a single insertion. That observation gives us a lower bound on the size of a deletion-1 code for strings of length n. Such a code must contain at least

$$\frac{a^n}{n(a-1)+1}$$

strings.

Different strings of length $n + 1$ can cover different numbers of strings of length n by single deletions. The number of strings covered is the number of *runs* in the string, where a run is a maximal sequence of identical symbols. For example, we observed above that the string 00000, which has only one run, can cover only one string, 0000, by a single deletion. Surely, a string of length $n + 1$ can have no more than $n + 1$ runs. Thus, an insertion-1 code for strings of length n must have at least $a^n/(n + 1)$ strings.

We can get a better bound by observing that strings with r runs can only cover, by single deletions, strings with between $r-2$ and r runs. Thus, an insertion-1 code must have strings of almost all numbers of runs. However, the detailed bound involves complex formulas and is not more than a factor of two better than the simple $a^n/(n + 1)$ bound.

4.2 Summary of Results

Our results are summarized in Table 1. In the table and the rest of the paper, we specify code sizes as fractions of the number of strings of length n. For example, the $a/(n+1)$-size insertion-1 code of the first row of Table 1 contains $a/(n+1)$ fraction of all strings of length n (or exactly $aa^n/(n + 1) = a^{n+1}/(n + 1)$ codewords).

Section 5 begins by summarizing our proof strategy for explicitly constructing covering codes. In Section 5.1, we describe our explicit construction of insertion-1 covering codes. In Section 5.2 and Section 5.3 we give explicit constructions of deletion codes for distances 1 and 2, that are of size $O(1/a^2)$ and $O(1/a^3)$, respectively.

Finally, in Section 6, we prove the existence of $O(\log(n)/n)$-size deletion-1 codes—a major improvement over

Insertion/Deletion	Size	Explicit/Existence
insertion-1	$a/(n+1)$	explicit
deletion-1	$O(\log(n)/n)$ for $\frac{n}{\log(n)} \geq 48a$	existence
deletion-1	$O(1/a^2)$ for $n \geq 3a\log(a)$	explicit
deletion-2	$O(1/a^3)$ for $n \geq \frac{a}{2} + \log(a)$	explicit

Table 1: Summary of Edit Distance Covering Codes.

our result from Section 5.2 for long strings. However, note that the existential upper bound we offer is greater by a factor of $O(a \log n)$ than the lower bound from Section 4.1.

Just as we did for Hamming distance in Section 2.1, we can take the cross product of a covering code C with itself several times to get a covering code for longer strings with a larger distance. This construction is not usually optimal, but cannot be too far from optimal. This construction can then be used in our anchor points algorithm from Section 3, but now for finding strings of length n at edit distance d.

5. EXPLICIT CONSTRUCTION OF EDIT-DISTANCE COVERING CODES

Let $w = w_n w_{n-1} \cdots w_1$ be a string of length n over an alphabet Γ of size a, and let C be the edit-distance covering code we are constructing. We first outline the general recipe we use to construct C:

1. **Sum value**: Assign each string w a *sum* value $\mathbf{sum}(w)$, the value of applying some function to w, treating each of its positions w_i as an integer (recall we assume the symbols of the alphabet are integers $0, 1, \ldots, a-1$).
2. **Modulus value**: Pick an appropriate integer c and let $\mathbf{score}(w) = \mathbf{sum}(w) \mod c$.
3. **Residues**: Pick one or more residues modulo c. Put into C all strings of appropriate length (e.g $n+1$ for insertion-1 codes or $n-1$ for deletion-1 codes), whose score values are equal to one of the residues.

We then count the strings in C and prove that C covers all strings of length n. In some cases, we do not cover all strings with C. Rather, we show that the number of strings not covered (called *outliers*) is small compared to the size of C. We can then argue that by adding one codeword into C for each outlier string, we can construct an extended code C' that covers all strings and that has the same asymptotic size as C. We can find the outliers by going through each code word $c \in C$ and finding all strings of length n that c covers. This operation can be done in $n * |C|$ time for insertion codes, and $a * n * |C|$ for deletion codes. Afterwards we can go through all strings of length n in a^n time to find the outliers. Note that if we let $N = a^n$ be the set of all strings of length n, this entire construction takes $O(aNlogN)$ time. Notice that $O(aN \log N)$ is much less than the brute-force way of finding a code, even though it is exponential in the length of the strings. The obvious way to find a code would be to look at all 2^N subsets of strings of length n, smallest first, and test each to see if the subset covers all strings.

5.1 Insertion-1 Covering Codes

We follow the recipe above to construct an insertion-1 covering code:

- **Sum value**: $\mathbf{sum}(w) = \Sigma_{i=1}^{n} w_i \times i$
- **Modulus value**: $c = (n+1) \times (a-1)$
- **Residues**: Any $a-1$ consecutive residues, $\{(i \mod c), (i+1 \mod c), \ldots, (i+(a-2) \mod c)\}$. For example, if $a = 4$ and $n = 5$, then $c = 18$, we can pick the three consecutive residues $2, 3, 4$ or $17, 0, 1$.

Before we prove that the code we constructed covers every string of length n, we give an example:

EXAMPLE 5.1. *Let $a = 4$, $n = 5$, and assume we pick* 8, 9, *and* 10 *as our residues. Then our code consists of all strings of length 6, whose score values equal* 8, 9, *or* 10. *Consider the string* 23010. *Then we can insert* 0 *between the fourth and fifth digits (*3 *and* 2*), and produce* 203010, *which is a codeword since its sum value is* 26 *and score value is* 8. *Similarly consider the string of all zeros:* 00000. *We can insert* 3 *between the second and third digits, and produce* 000300, *which also is a codeword as it has a score of 9.*

It is not a coincidence that we were able to take a string w of length five and generate a codeword by inserting a 0 or a 3 into w. As we prove momentarily, our code has the property that every string w of length n is covered by inserting one of the symbols 0 or $a-1$ somewhere in w.

Consider a string w of length n. Let $sumX_j$, and $scoreX_j$, for $j = n+1, \ldots, 1$ be the sum and score values, respectively, of the string that is constructed by adding 0 to the left of w_{j-1}. If $j = 1$, we add 0 at the right end. Similarly, let $sumY_j$ and $scoreY_j$ be the sum and score values, respectively, of the string constructed by adding $(a-1)$ to the left of w_{j-1}, or at the right end if $j = 1$. For example, for the string 23010, $sumX_3$ is the sum value of the string 230010 (the second 0 is the one inserted) and is equal to 29. $scoreX_3$ is then 29 $\mod 18 = 11$. Similarly, $sumY_1$ is the sum value of the string 230103 and is equal to 33, and $scoreY_1$ is 33 $\mod 18 = 15$.

LEMMA 5.2. *(i) $sumY_{n+1} - sumX_{n+1} = (n+1)(a-1)$ (ii) $sumY_1 - sumX_1 = (a-1)$.*

PROOF. (*i*) Let $u = (a-1)w_n \cdots w_1$ and $v = 0w_n \cdots w_1$. u and v differ only in the $(n+1)$st digit. Therefore the difference between $\mathbf{sum}(u)$ and $\mathbf{sum}(v)$ is exactly $(n+1) \times (a-1)$.

(*ii*) Let $z = w_n \cdots w_1(a-1)$ and $t = w_n \cdots w_1 0$. z and t differ only in the first digit. Therefore the difference between $\mathbf{sum}(z)$ and $\mathbf{sum}(t)$ is exactly $a-1$. □

Consider the sequences $sumX_{n+1}, sumX_n, \ldots, sumX_1$ and $sumY_{n+1}, sumY_n, \ldots, sumY_1$ of the sum values produced by inserting a 0 and $(a-1)$ to the left of each digit in w, respectively. We can visualize these sequences as two walkers, an X walker and a Y walker, taking an n-step walk on the number line. Figure 1 shows the walk for the string 23010. In the figure, the top labels of the lines are the sum values and bottom labels are the score values. Note

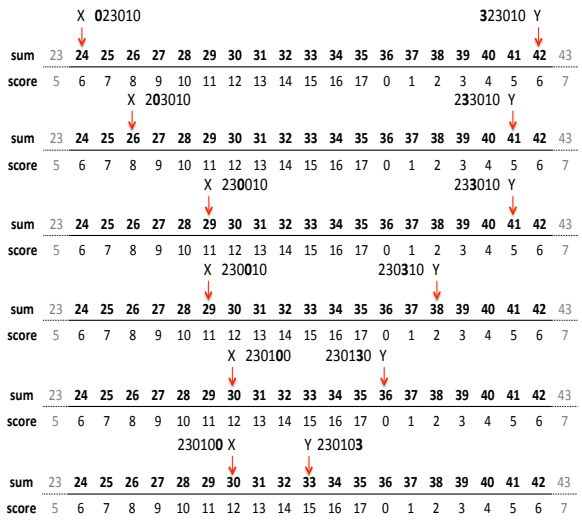

Figure 1: Simulation of insertions of symbols 0 and $(a-1)$ into strings as two walkers.

that the X (Y) walker being on a position with a particular sum value s and score value r corresponds to constructing a string of length six from 23010 by a single insertion of 0 $\big($(a-1)$\big)$ with sum value s and score value r. We know from Lemma 5.2 that $sumY_{n+1} - sumX_{n+1} = (n+1)(a-1)$ and $sumY_1 - sumX_1 = (a-1)$: the walkers start $(n+1)(a-1)$ and finish exactly $(a-1)$ positions away from each other. We will next prove that the walkers always walk in opposite directions in steps of size at most $a-1$.

LEMMA 5.3. *$sumX_j - sumX_{j+1} = i$ and $sumY_j - sumY_{j+1} = -(a-1-i)$, for some $i \in 0, \ldots, a-1$.*

PROOF. Let w_{j+1} be i. Then

$$sumX_j = \mathbf{sum}(w_n \ldots w_{j+2} i 0 w_j \ldots w_1)$$

$$sumX_{j+1} = \mathbf{sum}(w_n \ldots w_{j+2} 0 i w_j \ldots w_1)$$

Notice that the inputs to the sum functions differ only in the $(j+1)$st and $(j+2)$nd digits. Subtracting one from another, $sumX_j - sumX_{j+1} = i(j+2) - i(j+1) = i$. Similarly

$$sumY_j = \mathbf{score}(w_n \ldots w_{j+2} i (a-1) w_j \ldots w_1)$$

$$sumY_{j+1} = \mathbf{score}(w_n \ldots w_{j+2} (a-1) i w_j \ldots w_1)$$

Therefore,

$$sumY_j - sumY_{j+1} = i(j+2) + (a-1)(j+1) - [(a-1)(j+2) + i(j+1)] = -(a-1-i)$$

□

In other words, the sum values are always increasing for walker X and decreasing for walker Y. Moreover, the sum values differ by $\leq (a-1)$ for each walker and cumulatively they travel a distance of $(a-1)$. In Figure 1, this can be visualized as two walkers at two ends of a line walking towards each other synchronously, and at each step, if walker X moves i amount to the right, walker Y moves $(a-1-i)$ amount to the left.

THEOREM 5.4. *Fix any $(a-1)$ consecutive residues*

$$R = \{i \mod c, i+1 \mod c, \ldots, \big(i+(a-2)\big) \mod c\}$$

where $c = (n+1)(a-1)$. The code C constructed by taking all strings of length $n+1$ whose score values are in R covers all strings of length n by a single insertion.

PROOF. Again consider any string w of length n and the corresponding X and Y walkers for it. We know from Lemma 5.2 that the walkers starts exactly $(n+1)(a-1)$ sum values away. Therefore the score values of the numbers between their initial posititions cover exactly a full residue cycle of modulo $c = (n+1)(a-1)$. We also know that they walk in opposite directions (Lemma 5.3) and finish the walk exactly $(a-1)$ sum values away (Lemma 5.2). Since the step sizes of the walkers is $\leq (a-1)$ (Lemma 5.3) neither of the walkers can skip over all the $(a-1)$ consecutive residues in R in a single step, which implies that at least one of the walkers must step on one of the residues in R. In other words we can insert 0 or $(a-1)$ into some position j of w and generate a codeword. □

COROLLARY 5.5. *We can construct an $a/(n+1)$ size insertion-1 covering code C for strings of length n.*

PROOF. Let C_j be the code we construct by selecting the $(a-1)$ residues between $j(a-1)$ and $(j+1)(a-1)$, for $j \in 0, \ldots, n$. Note that C_j's are disjoint, and every string of length $n+1$ belongs to one C_j. We have $n+1$ disjoint codes and their union has size a^{n+1} (all strings of length $n+1$). Therefore one of the codes must contain at most $a^{n+1}/n+1$ strings and is an $a/n+1$-size code. □

5.2 O(1/a²)-size Deletion-1 Covering Codes

We next use our recipe for explicitly constructing codes to construct an $O(1/a^2)$ size deletion-1 code, for large enough n.

- **Sum value**: $\mathbf{sum}(w) = \Sigma_{i=1}^{n} w_i$. That is, the sum value of w is the sum of the integer values of its digits.
- **Modulus value**: $c = a$
- **Residues**: 0

This code covers nearly all strings of length n. Consider a string w of length n. Let $\mathbf{score}(w) = i$. If w has any occurrence of the symbol i, delete it, and you get a codeword. Thus, our code covers all strings that contain their modulus. To make it a covering code, we take any string that is not covered, remove its first digit, and add it to the code. Then any string w of length n will either be covered by the original code, or it will be covered by the codeword that we added specifically for it.

To determine the size of our code, we first observe that induction on n shows that there are a^{n-1} strings of length n with score r for each residue $r \in \{0, \ldots, a-1\}$. Thus, in particular, there are a^{n-2} strings of length $n-1$ with score 0, making the original code a $1/a^2$-size code. We show that the number of strings of length n that are missing their modulus is $O(1/a^2)$. To do so, we exhibit a bound on the size of the set S of strings that are missing at least one symbol, which certainly contains every string that is missing its modulus. Observe that $S = \cup_i S_i$, where S_i is the set of strings of length n that do not contain symbol i. By the union bound, we have that $|S| \leq \sum_i |S_i|$, and thus it suffices to show that each $|S_i|$ represents an $O(1/a^3)$ fraction of the strings

of length n. The number of strings that do not contain the symbol i is exactly $(a-1)^n$ which is exactly $(1-1/a)^n$ fraction of all strings. This quantity is at most $e^{-n/a}$ and is bounded above by $1/a^3$ for $n \geq 3a\log(a)$, proving the following result:

THEOREM 5.6. *For $n \geq 3a\log(a)$, there is an $O(a^{n-2})$-size deletion-1 code.*

5.3 O(1/a³)-size Deletion-2 Covering Code For Shorter Strings

For our deletion-2 code we use the following scheme.

- **Sum value**: $\mathbf{sum}(w) = \Sigma_{i=1}^{n} w_i$, as in Section 5.2
- **Modulus value**: $c = a$
- **Residues**: 0

Suppose we have a string x of length n and $\mathbf{score}(x) = i$. We need to find a pair of positions of x that sum to i modulo a and delete them both. To start, we assume that a is even; the situation for odd a is very similar and we will discuss it in the end. We can group the integers from 0 to $a-1$ into pairs that sum to i modulo a. There is a special case where for some integer j, we have $2j = i \mod a$. In that case, there are two such integers j, and we group those two integers into one group.

EXAMPLE 5.7. *Let $a = 6$. Figure 2 shows the pairs that sum to i modulo 6:*

i				
0:	0-0	1-5	2-4	3-3
1:	0-1	2-5	3-4	
2:	0-2	1-1	3-5	4-4
3:	0-3	1-2	4-5	
4:	0-4	1-3	2-2	5-5
5:	0-5	1-4	2-3	

Figure 2: Pairs that sum to i modulo 6

So, for example, if $i = 1$, then the three groups are $\{0,1\}, \{2,5\}$, and $\{3,4\}$. If $i = 2$, then the three groups are $\{0,2\}, \{1,4\}$, and $\{3,5\}$. Note that 1+1 and 4+4 are both equal to 2 mod 6*, so we put them into one group.*

In general, if a is even, then the pairs that sum to 0 modulo a are $0+0$, $\frac{a}{2}+\frac{a}{2}$, $1+(a-1)$, $2+(a-2)$, $3+(a-3)$, and so on, until $(\frac{a}{2}-1)+(\frac{a}{2}+1)$. If we want the pairs that sum to i, where i is even, then we add $i/2$ to every integer in this list of pairs. The integers $i/2$ and $(a+i)/2$, when added to themselves, make i modulo a, while the other $\frac{a}{2}-1$ pairs of two different integers also sum to i modulo a.

If we want the pairs of integers that sum to 1 modulo a, we note that these are $0+1$, $2+(a-1)$, $3+(a-2)$, and so on, until $(\frac{a}{2})+(\frac{a}{2}+1)$. That is, there are $\frac{a}{2}$ pairs of distinct integers. If we want to find the pairs that sum to i, for odd i, then we add $(i-1)/2$ to each of the integers, and again we get $\frac{a}{2}$ pairs of distinct integers.

The important point is that regardless of the desired sum i, we can divide the integers modulo a into $\frac{a}{2}$ groups. Each group either consists of two distinct integers that sum to i modulo a or consist of the two integers that, when added to themselves, yield i modulo a.

If there are k positions in the string holding members of the same group, then the probability is at least $1-2^{-(k-1)}$ that these positions hold two symbols that sum to i modulo a. First, look at groups composed of two different values that sum to i modulo a, such as $\{3,5\}$ for $a = 6$ and $i = 2$. All positions belonging to the group are independent (assuming we have chosen a string x randomly). So each position after the first has probability $1/2$ of disagreeing with the first. That is, the probability that all k positions hold the same symbol is $2^{-(k-1)}$.

For a group that is composed of two symbols each of which, added to itself makes i, such as the group $\{1,4\}$ for $a = 6$ and $i = 2$, then the situation is even better. If $k = 2$, the probability is $1/2$ that the two positions for that group are the same, but if $k \geq 3$, then we are certain to find two positions that sum to i modulo a.

If the length of x is n, then there are at least $n-(a/2)$ positions of x that are *not* the first in their group. Thus, the probability that we are unable to find any pair of positions of x that sum to i modulo a is at most $2^{n-(a/2)}$. If n is bigger than $a/2+\log(a)$, then the number of outliers is at most $1/a$ of the total number of strings of length $n-2$. Thus, we can expand C to include one codeword for each outlier, proving the following result:

THEOREM 5.8. *For $n \geq \frac{a}{2}+\log(a)$, there is an $O(a^{n-3})$-size deletion-2 code.*

6. EXISTENCE OF O(log(n)/n)-SIZE DELETION-1 COVERING CODES

We next show that for sufficiently long strings there are deletion-1 covering codes that are much smaller than the $O(1/a^2)$-size code from Section 5.2. The proof of the existence of such codes is much more involved than our previous constructions. Instead of showing the existence of an edit-distance-1 covering code directly, we convert the strings of length n and alphabet size a into binary strings of lengths $\leq n$. We then show the existence of a Hamming-distance-1 covering code H for the converted binary strings and convert H into a deletion-1 covering code C for the original strings.

We begin with a roadmap and proof outline. All the terminology we use in the outline, e.g. "run patterns", "bits of runs", or "safe bits" will be defined in the specified sections.

1. Convert each string w of length n to its *run pattern,* $\mathbf{runs}(w)$ (Section 6.1).
2. Convert run patterns of w to a binary string, which we refer to as *bits of runs of w* (Section 6.2).
3. Partition the strings of length n into two groups based on the number of *safe bits* their bits of runs have: LS (*low-safe-bit*) and HS (*high-safe-bit*). The strings in LS will be the first set of outlier strings for the final code we construct and will be covered separately at the end of our construction (Section 6.3).
4. Construct a deletion-1 code C that covers all but $1/n$ fraction of the strings in HS. The remaining $1/n$ fraction of the strings in HS will be the second set of outlier strings. We will construct C from a Hamming-Distance-1 code H for binary strings, which covers the bits of runs of the strings in HS on their safe bits (Section 6.4).

5. For each outlier string s, put into C the string that we get by removing the first symbol of s, and construct a deletion-1 covering code (Section 6.6).

6. Count the number of outliers and the total number of strings in C. (Section 6.6).

6.1 Step 1: Run Patterns

We view strings as sequences of *runs* – consecutive positions that hold the same character. The *length* of a run is the number of consecutive positions that hold the same character. A *run pattern* (or just "pattern") is a list of positive integers. Every string w of length n corresponds to exactly one pattern P, which is the list of positive integers, the ith of which is the length of the ith run of w. We denote this pattern P by $\mathbf{runs}(w)$. Note that the run pattern of a string has the same length as the number of runs in that string.

EXAMPLE 6.1. *String $w = 002111100$ consists of four runs, 00, 2, 1111, 00, in that order. The lengths of these runs are 2, 1, 4, and 2, respectively, so* $\mathbf{runs}(w) = [2, 1, 4, 2]$.

6.2 Step 2: Converting Run Patterns into Binary Strings

For the second step of our proof, we need to convert run patterns into bit strings of the same length. Define $\mathbf{bits}(P)$ to be the bit string whose ith position holds 0 if the ith integer on the list P is even, and holds 1 if the ith integer is odd.

EXAMPLE 6.2. *If $w = 002111100$, then*

$$\mathbf{runs}(w) = [2, 1, 4, 2]$$

and $\mathbf{bits}\big(\mathbf{runs}(w)\big) = \mathbf{bits}([2, 1, 4, 2]) = 0100$.

6.3 Step 3: Partitioning Strings Based on Safe Bit Counts

Deletion of a symbol from a string w in general can generate a string v with a shorter run pattern, and hence $\mathbf{bits}\big(\mathbf{runs}(v)\big)$ can be shorter than $\mathbf{bits}\big(\mathbf{runs}(w)\big)$. For example, deletion of the symbol 2 from 00211100, whose run pattern is $[2, 1, 3, 2]$, generates 00111100, whose run pattern is $[2, 4, 2]$. However, if we delete a symbol from w that belongs to a run of length 2 or more, we will get a string v with the following properties:

- $|\mathbf{bits}\big(\mathbf{runs}(v)\big)| = |\mathbf{bits}\big(\mathbf{runs}(w)\big)|$; v has the same number of runs and hence bits of runs as w.
- $\mathbf{bits}\big(\mathbf{runs}(v)\big)$ and $\mathbf{bits}\big(\mathbf{runs}(w)\big)$ differ in exactly one bit and hence have Hamming distance 1. The bit in which they differ corresponds to the run from which we removed the symbol.

EXAMPLE 6.3. *If we remove one of the 1's in*

$$w = 002111100$$

we get $v = 00211100$. $\mathbf{bits}\big(\mathbf{runs}(v)\big) = 0110$*, which is at Hamming distance one from* $\mathbf{bits}\big(\mathbf{runs}(w)\big) = 0100$. *Note that because we removed a symbol from the third run of w, the two bit strings differ in the third bit.*

We call a bit in $\mathbf{bits}\big(\mathbf{runs}(w)\big)$ a *safe bit* for w, if it corresponds to a run of length ≥ 2. Consider again the string $w = 002111100$ as an example. Every 0 in $\mathbf{bits}\big(\mathbf{runs}(w)\big) = 0100$, is safe, and the bit 1, which corresponds to the second run of w is unsafe because it corresponds to a run of length 1. Different strings have different numbers of safe bits. For example, a string composed of an alternating sequence of different symbols, such as 212121 has no safe bits, since it has no runs of length ≥ 2.

We partition the set of strings we want to cover into two groups based on the number of safe bits they have. Let LS (for low safe-bit strings) be the set of strings of length n that have fewer than $n/6a$ safe bits. Similarly, let HS (for high safe bit strings) be the set of strings with at least $n/6a$ safe bits. Furthermore, we partition HS into $HS_1, \ldots, HS_n$, where HS_i is the set of high safe bit strings with i runs.

We finish this section with a key definition. Consider a Hamming covering code H that covers all bit strings of length i and a string w with i runs. We say that H *covers w on a safe bit*, if there is a codeword $h \in H$, such that:

1. h and $\mathbf{bits}\big(\mathbf{runs}(w)\big)$ are at Hamming distance 1, and
2. The bit on which h and $\mathbf{bits}\big(\mathbf{runs}(w)\big)$ differ corresponds to a safe bit of w.

We note that two strings w_1 and w_2 can have the same bits of runs, yet a Hamming covering code can cover only one of them on a safe bit.

EXAMPLE 6.4. *Let $w_1 = 22111300$ and $w_2 = 33022211$. The bits-of-runs for both strings is* 0110*. Consider a Hamming covering code H containing the string* 0010*, which is at Hamming distance 1 from* 0110*. Then H covers both w_1 and w_2 on the second bit from left, which is a safe bit of w_1 but not w_2. However, if there is no other string in H that covers* 0110*, then H covers w_1 but not w_2 on a safe bit.*

In the next section, we will construct an edit covering code C that covers all but $1/n$ fraction of all strings in HS, using Hamming covering codes that cover the bits of runs of strings in HS on safe bits.

6.4 Step 4: Constructing a Deletion-1 Code Covering (1-1/n) Fraction of HS

We start this section by explaining how we can take a Hamming covering code and turn it into a deletion-1 code (not necessarily a covering code). Let HCC_i be any covering code for Hamming distance 1 and bit strings of length i. We construct a particular deletion-1 code $\mathrm{EC}_{r=i}$ from HCC_i as follows.

$$\mathrm{EC}_{r=i} = \{x \mid \mathbf{bits}\big(\mathbf{runs}(x)\big) \in \mathrm{HCC}_i\}$$

That is, we put into $\mathrm{EC}_{r=i}$ all strings of length $n-1$, whose bits of runs is in HCC_i.

In the rest of this section, we first state three key lemmas. Lemmas 6.5 and 6.6 are proved in Section 6.5. Then we prove, using the lemmas, that we can build an $O\big(\log(n)/n\big)$-size deletion-1 code C that covers all but $1/n$ fraction of strings in HS. In Section 6.6, we will expand C by codewords that cover the all the strings in HS not covered and the strings in LS and construct a deletion-1 covering code.

LEMMA 6.5. *Let X be a subset of the strings in HS_i. Suppose there exists a Hamming covering code* HCC_i *for bit strings of length i, such that $|\mathrm{HCC}_i| = m$. Then there exists a set* $\mathrm{EC}_{r=i}$ *of strings of length $n-1$, such that the following is true.*

1. $|\mathrm{EC}_{r=i}| \le m/2^{i-1}$ *fraction of the strings of length* $n-1$ *with* i *runs.*

2. $\mathrm{EC}_{r=i}$ *covers at least* $nm/12a2^i$ *fraction of all stings in* X *on their safe bits.*

We defer the proof until Section 6.5. At a high level, this lemma says that if we have a small Hamming covering code HCC_i for bit strings of length i and a subset X of strings in HS_i, we can construct a small size deletion-1 code $\mathrm{EC}_{r=i}$ that covers an important fraction of the strings in X. Our next lemma says that such small size Hamming covering codes indeed exist.

LEMMA 6.6. *There is an* HCC_i *code with at most* $2^{i+1}/i$ *codewords. Put another way, there is a code* HCC_i *with at most fraction* $2/i$ *of the binary strings of length* i*.*

Again, the proof is deferred to Section 6.5. We next state an immediate corollary to Lemmas 6.5 and 6.6.

COROLLARY 6.7. *For any* i*, with* $1 \le i < n$*, there is a deletion-1 code* $\mathrm{EC}_{r=i}$ *of strings of length* $n-1$*, such that*

1. $|\mathrm{EC}_{r=i}| \le 4/i$ *fraction of the strings of length* $n-1$ *with* i *runs, and*

2. $\mathrm{EC}_{r=i}$ *covers at least*

$$\frac{n2^{i+1}}{12ai2^{i+1}} = n/12ai \ge 1/12a$$

fraction of all strings in HS_i*.*

PROOF. The corollary follows from substituting $2^{i+1}/i$ from Lemma 6.6 for m in Lemma 6.5. □

Finally, we need the following lemma to count the number of strings in the deletion-1 code we construct in Theorem 6.9.

LEMMA 6.8. *The number of strings of length* $n-1$ *over an alphabet of size* a*, with* i *runs, is* $a(a-1)^{i-1}\binom{n-2}{i-1}$*.*

PROOF. Imagine a string of length $n-1$ with $i-1$ "fenceposts" separating the runs. A string of length $n-1$ may thus be viewed as $n-1$ "regular" symbols and $i-1$ fenceposts. However, there are some constraints on where the fenceposts appear. A fencepost cannot occupy the last position, and each fencepost must be preceded by a regular symbol. Thus, we can think of the string and fenceposts as $i-1$ pairs consisting of a regular symbol followed by a fencepost, $n-i-1$ regular symbols that are not at the end and not followed by a fencepost, and finally, a regular symbol at the end. The number of arrangements of the $i-1$ pairs and $n-i-1$ regular symbols is $\binom{n-2}{i-1}$. The factor $a(a-1)^{i-1}$ is justified by the fact that the first run can be any of the a symbols of the alphabet, and each of the $i-1$ succeeding runs may be any of the a symbols except for the symbol that is used for the previous run. □

We can now prove that we can construct a $O(\frac{\log(n)}{n})$-size deletion-1 code that covers all but $\le \frac{1}{n}$ fraction of the strings in HS.

THEOREM 6.9. *There is an* $O(\frac{\log(n)}{n})$*-size deletion-1 code* C *that covers* $1-\frac{1}{n}$ *fraction of the strings in* HS*.*

PROOF. For each i, $1 \le i < n$, we construct a deletion-1 code $\mathrm{EC}_{r=i}$ as follows: We let $X = HS_i$ and using Corollary 6.7, find a deletion-1 code $\mathrm{EC}_{(r=i),1}$ that covers at least fraction $1/12a$ of X_i, and contains at most fraction $4/i$ of the strings of length $n-1$ with i runs. Then, we remove the covered strings from X and find an $\mathrm{EC}_{(r=i),2}$ that covers at least fraction $1/12a$ of the remaining X, and is of size at most fraction $4/i$ of the strings of length $n-1$ with i runs. We repeat this construction $\log_{\frac{12a}{12a-1}}(n)$ times, to construct $\mathrm{EC}_{(r=i),j}$ for $j = 3, 4, \ldots$. We then take the union all $\mathrm{EC}_{(r=i),j}$'s and construct $\mathrm{EC}_{r=i}$ which

1. contains $4\log_{\frac{12a}{12a-1}}(n)/i$ fraction of all strings of length $n-1$ with i runs, and

2. covers $1-\frac{1}{n}$ fraction of the strings in HS_i.

Let $C = \cup_i \mathrm{EC}_{r=i}$. By construction, C covers $1-\frac{1}{n}$ of all strings in HS. That is, each $EC_{r=i}$ covers $1-\frac{1}{n}$ fraction of all strings in HS_i, and $HS = \cup HS_i$. We only have to prove that C is a $O(\frac{\log(n)}{n})$-size code: i.e., it contains $O(\frac{\log(n)}{n})$ fraction of all strings of length n.

By Lemma 6.8, the number of strings of length $n-1$ with i runs is $a(a-1)^{i-1}\binom{n-2}{i-1}$. We know that each $EC_{r=i}$ contains $4\log_{\frac{12a}{12a-1}}(n)/i$ fraction of those strings. When we sum over i, we get an upper bound on the size of C:

$$4\log_{\frac{12a}{12a-1}}(n)\sum_{i=1}^{n-1} a(a-1)^{i-1}\binom{n-2}{i-1}\frac{1}{i}$$

Now, expand the combinatorial function in factorials:

$$4\log_{\frac{12a}{12a-1}}(n)\sum_{i=1}^{n-1} a(a-1)^{i-1}\frac{(n-2)!}{(i-1)!(n-1-i)!}\frac{1}{i}$$

Multiply by $(n-1)/(n-1)$, and group the factor i with $(i-1)!$ to get:

$$4\log_{\frac{12a}{12a-1}}(n)\sum_{i=1}^{n-1} \frac{a(a-1)^{i-1}}{n-1}\frac{(n-1)!}{i!(n-1-i)!}$$

Next, observe that the factorials give exactly $\binom{n-1}{i}$. Move all the factors that do not involve i outside the summation to get

$$\frac{4a\log_{\frac{12a}{12a-1}}(n)}{(n-1)(a-1)}\sum_{i=1}^{n-1}(a-1)^i\binom{n-1}{i} \qquad (1)$$

The summation is all the terms in the expansion of

$$[1+(a-1)]^{n-1}$$

with the exception of the first and last terms — those for $i=0$. Thus, a good upper bound on Equation 1 is

$$a^{n-1}\frac{4a\log_{\frac{12a}{12a-1}}(n)}{(n-1)(a-1)} \qquad (2)$$

The factor $\log_{\frac{12a}{12a-1}}(n)$ is approximately $(12a-1)\log(n)$:

$$\log_{\frac{12a}{12a-1}}(n) = \log_{1+\frac{1}{12a-1}}(n) = \log_{1+\frac{1}{12a-1}}(e)\log(n)$$

$\log(1+\epsilon)$ is approximately ϵ for small values of ϵ. Therefore,

$$\log_{1+\frac{1}{12a-1}}(e) = \frac{1}{\log(1+\frac{1}{12a-1})} \approx \frac{1}{\frac{1}{12a-1}} = 12a-1$$

Substituting $(12a-1)\log(n)$ for $\log_{\frac{12a}{12a-1}}(n)$ in Equation 2, we get:

$$|C| = O(\frac{a^n \log(n)}{n})$$

□

We will next prove Lemmas 6.5 and 6.6. Finally, in Section 6.6 we will show that for sufficiently large n, the number of outliers that we have to add to C is less than fraction $1/n$ fraction of all strings of length n , which will prove the existence of $O(\frac{\log(n)}{n})$-size deletion-1 covering codes.

6.5 Proof of Lemmas

Recall Lemma 6.5 states that given a Hamming covering code HCC_i for bit strings of length i of size m and given a set X of strings with enough safe bits ($\geq \frac{n}{6a}$) that we want to cover, we can find a deletion-1 code $\text{EC}_{r=i}$ that contains $\frac{m}{2^{i-1}}$ fraction of the strings of length $n-1$ with i runs and that covers a large fraction ($\frac{nm}{6a2^i}$) of the strings in X. Our strategy is to generate a large number of covering codes from HCC_i and calculate the average number of strings they cover from X. We can then argue that at least one choice is average or above. We first introduce *affine Hamming codes*.

6.5.1 Affine Codes

Suppose we start with some fixed Hamming covering code $H = \text{HCC}_i$. For any bit string x of length i, the *affine code* $H_x = H \oplus x$ is the set of strings that are formed by taking the bitwise modulo-2 sum of x and any string w in H.

EXAMPLE 6.10. *Suppose $i = 4$ and*

$$H = \{0000, 0111, 1011, 1101, 1110\}$$

We leave it to the reader to verify that every string of length four is covered by H. There are sixteen ways we can construct an affine code from H; some of these codes will be the same, however. We can construct H itself by choosing $x = 0000$. That is, $H_{0000} = H$. If we choose $x = 0011$, we get $H_{0011} = \{0011, 0100, 1000, 1101\}$, and so on.

Some useful facts about the collection of affine codes is the following.

LEMMA 6.11. *If H is a Hamming covering code for strings of length i, then so is H_x for any string x of length i.*

PROOF. Let w and x be strings of length i. We need to show that w is covered by some string in H_x. We know that $w \oplus x$ is covered by some string y in H. That is, y and $w \oplus x$ differ in at most one bit. Then $y \oplus x$ is in H_x. We claim that w and $y \oplus x$ differ in at most one bit, and therefore w is covered by H_x.

Consider any bit j in which y and $w \oplus x$ agree; there will be at least $i-1$ such bits. Let w_j, x_j, and y_j be the jth bits of w, x, and y, respectively. Then we are given that $y_j = w_j \oplus x_j$. If we add $y_j \oplus w_j$ to both sides modulo 2 we get $y_j \oplus y_j \oplus w_j = w_j \oplus x_j \oplus y_i \oplus w_j$. Since $\oplus$ is associative and commutative, and $z \oplus z = 0$ for any z, it follows that $w_j = y_j \oplus x_j$. Therefore, w and $y \oplus x$ differ in at most one bit, and w is covered by H_x. □

LEMMA 6.12. *Suppose H is a Hamming covering code with m members, for strings of length i. Then among all the affine codes H_x, each string of length i appears exactly m times.*

PROOF. The string w appears in H_x if and only if $w = y \oplus x$ for some y in H. But $w = y \oplus x$ if and only if $x = w \oplus y$ (the argument is the same as that given for bits in Lemma 6.11). Thus, w is in one affine code H_x for every member y of H. Therefore, w is in exactly m affine codes. □

We are now ready to prove Lemma 6.5.

6.5.2 Proof of Lemma 6.5

Let HCC_i be a Hamming covering code for bit strings of length n, and let B_i be the set of strings of length $n-1$ with i runs. Consider a randomly picked affine code HCC_x of HCC_i out of the 2^i possible affine codes. By Lemma 6.12, for each string w in B_i, there are exactly m affine codes of HCC_i for which w is in the code. By linearity of expectations, the expected number of strings of B_i in H_x is $m|B_i|/2^i$. By Markov's inequality, the probability that the number of strings from B_i in HCC_x is greater than twice the expectation is $\leq 1/2$, which implies:

$$Pr(\#\text{ of strings from } B_i \in \text{HCC}_x \leq 2m|B_i|/2^i) > \frac{1}{2} \quad (3)$$

Now consider the set X and a string $x \in X$. Recall that strings in X have more than $\frac{n}{6a}$ safe bits. Let

$$b = \mathbf{bits}\big(\mathbf{runs}(x)\big) = b_i, \ldots b_1$$

Let b_j be a safe bit for x. Then there are exactly m affine codes HCC_y of HCC_i, such that HCC_y covers b by flipping b_j. That is because by Lemma 6.12, there are exactly m affine codes of HCC_i that contain the string $b' = b_i, \ldots, b_{j+1}, \neg b_j, \ldots, b_1$—the string b with b_j flipped. Since there are at least $\frac{n}{6a}$ safe bits in b for x, there are at least $\frac{mn}{6a}$ affine codes, whose generated deletion-1 covering code will cover x. Therefore, expected number of strings that a randomly picked affine code H_x will cover from X is $\frac{|X|mn}{6a}$. Again by Markov's inequality, the probability that a random H_x covers fewer than $\frac{|X|mn}{12a}$ strings from X is $\leq \frac{1}{2}$, which implies:

$$Pr(\text{HCC}_x \text{ covers} > \frac{|X|mn}{12a} \text{ strings from } X) > \frac{1}{2} \quad (4)$$

By equation 3, the probability that a randomly picked HCC_x contains less that $2m|B_i|/2^i$ fraction of the strings of length $n-1$ with i runs is $> \frac{1}{2}$. By equation 4, the probability that a random HCC_x covers more than $|X|mn/12a$ strings from X is $> \frac{1}{2}$. Then, there must exist one HCC_x for which both conditions hold, completing the proof of the lemma.

We next prove Lemma 6.6.

6.5.3 Proof of Lemma 6.6

Let i be in the range $2^{r-1}-1 < i \leq 2^r - 1$ for some integer r. There is a Hamming code C for strings of length $2^{r-1}-1$, and it is a perfect code, so it is also a Hamming covering code for that length. Take the cross product of $C \times \{0,1\}^{i-2^{r-1}+1}$; call the resulting code C'. That is, expand the code C by appending all possible bit strings to each of the codeword, to make strings of length i rather than length $2^{r-1}-1$.

We claim that C' is a covering code for strings of length i. In proof, let w be a string of length i. Since C is a covering code, we know we can find a member x of C such that the first $2^{r-1}-1$ bits of w differ from x in at most one bit. Extend x with the last $i - 2^{r-1} + 1$ bits of w. We now have

a codeword of C', and that codeword differs from w in at most one bit.

6.6 Steps 5 and 6: Existence of O(log(n)/n)-size Deletion-1 Codes

We are now ready to complete our proof that $O(log(n)/n)$-size deletion-1 codes exist. So far, we partitioned strings of length n into LS, those $< \frac{n}{6a}$ safe bits, and HS, those with $> \frac{b}{6a}$. We then showed in Theorem 6.9 that we can cover all but $\leq \frac{1}{n}$ fraction of the strings in HS with a $O(log(n)/n)$-size code C. The two groups of outliers to C are: (1) the $\leq \frac{1}{n}$ fraction of strings in HS that c does not cover; and (2) the strings in LS. Notice that the size of the strings in (1) is $\leq \frac{1}{n}$ fraction of all strings of length n, since HS is a subset of all strings of length n. Our next lemma states that for large enough n, the size of LS is also $\leq \frac{1}{n}$ of all strings of length n.

LEMMA 6.13. *For n such that $n/\log(n) \geq 24a$, $|LS| \leq \frac{1}{n}$ fraction of all strings of length n.*

PROOF. Instead of counting $|LS|$ directly, we will count another set LSP for low "special" letter strings which contains LS. Divide a string w of length n into chunks of three: $w = w_1w_2w_3|w_4w_5w_6|\ldots|w_{n-2}w_{n-1}w_n$. For simplicity, we assume n is divisible by 3. Call w_{3j}, last letter of a chunk, for $j \in 1,\ldots,\frac{n}{3}$ a *special* letter if the following two conditions hold:

1. w_{3j} equals w_{3j-1} (the symbol to its left)
2. w_{3j} is different from w_{3j-2} (the symbol two positions to its left)

In other words, the letter has to be in a position congruent to 0 mod (3) and be the second letter in a run of length ≥ 2. For example, if $w = 231|10\mathbf{0}|034$, the 0 in position six (bolded) is the only special letter. Notice that the 1 at position four is the second letter in a run of length 2. However, it is not a special letter because it is not in position congruent to 0 mod 3. Let LSP be the set of strings with less than $\frac{n}{6a}$ special letters.

We first show that LSP contains LS. Consider a string $w \in LS$. Then w has $< \frac{n}{6a}$ runs of length ≥ 2. Then it has $< \frac{n}{6a}$ letters that satisfy conditions (1) and (2) above. Therefore it must have $< \frac{n}{6a}$ letters that satisfy conditions (1) and (2) and are also in a position congruent to 0 mod (3). Therefore w must also be in LSP.

We complete the proof by showing that for large n, the size of LSP is very small. Consider a procedure that generates strings of length n by generating n independent letters. We look at the generated string in chunks of three. Let $Y_1,\ldots,Y_{n/3}$ be random variables, such that $Y_i = 1$ if the last letter of the ith chunk is special and 0 otherwise. $\Pr(Y_i = 1) = \frac{a-1}{a} \times \frac{1}{a} = \frac{a-1}{a^2}$. This is because for Y_i to assume a value of 1: (1) the first letter can be anything; (2) the second letter has to be different from the first letter: a probability of $\frac{a-1}{a}$; and (3) the third letter has to equal to the second letter: a probability of $\frac{1}{a}$. And notice that Y_i are independent of each other because the value that Y_i takes only depends on the three bits produced for chunk i. By linearity of expectation, number of special bits in a random string is $\frac{n(a-1)}{3a^2} \geq \frac{n}{6a}$.

Let $Z = \Sigma_i Y_i$. By Chernoff bounds, $\Pr(Z < n/12a) < e^{-n/48a}$ which is less than $1/n$ for $n/\log(n) > 48a$. Since $|SSL| > |SL|$, for $n/\log(n) > 48a$ $|SL| < 1/n$ of all strings of length n completing our proof. □

We can now formally prove that $O(\log(n)/n)$-size deletion-1 codes exist for long enough strings.

THEOREM 6.14. *There exists a $O(\log(n)/n)$-size deletion-1 code for strings of length n when $n/\log(n) > 48a$.*

PROOF. We start with the $O(\log(n)/n)$-size code C from Theorem 6.9. For each uncovered string w in HS and LS, we put one codeword into C covering w, for example by deleting the first symbol of w, and produce a deletion-1 covering code. The number of uncovered strings in HS is $\leq \frac{1}{n}$ fraction of all strings of length n. Similarly, by Lemma 6.13, the size of LS is $\leq \frac{1}{n}$ fraction of all strings of length n for $n/\log(n) > 48a$. Therefore expanding C does not affect its asymptotic size of $O(\log(n)/n)$, for $n/\log(n) > 48a$. □

7. CONCLUSIONS

We have explored the design of anchor-points algorithms for solving fuzzy join problems using MapReduce. In addition to improving the efficiency of the approach when similarity is based on Hamming distance, we tackled the problem of finding good anchor-point sets, or "covering codes," for edit distance. We showed that finding covering codes for single insertions or deletions is sufficient to get good, although not optimal, covering codes for larger numbers of insertions or deletions. We use a number of constructions to get concrete codes, using several strategies where a small code covering almost all strings is constructed, and then augmented to capture the remaining strings. We also gave an existence proof for single-deletion covering codes that is within a factor $O(a \log n)$ of optimal, for any string length n and alphabet of size a. A number of challenging open questions remain:

1. Can the existence of smaller codes for single insertions or deletions be proved? Alternatively, can the lower bounds suggested in Section 4.1 be improved?
2. Can we find better constructions than those given here for explicit codes, even for special cases, such as small alphabets or long strings?
3. Can we extend the covering-code idea to other interesting distance measures, such as Jaccard distance for sets?

8. REFERENCES

[1] Covering codes. `http://en.wikipedia.org/wiki/Covering_code`.

[2] F. N. Afrati, A. D. Sarma, D. Menestrina, A. G. Parameswaran, and J. D. Ullman. Fuzzy Joins Using MapReduce. In *ICDE*, 2012.

[3] D. Applegate, E. M. Rains, and N. J. A. Sloane. On Asymmetric Coverings and Covering Numbers. *Journal on Combinatorial Designs*, 11(3), 2003.

[4] R. Baraglia, G. D. F. Morales, and C. Lucchese. Document Similarity Self-Join with MapReduce. In *ICDM*, 2010.

[5] S. Chaudhuri, V. Ganti, and R. Kaushik. A Primitive Operator for Similarity Joins in Data Cleaning. In *ICDE*, 2006.

[6] G. Cohen. *Covering codes*. North-Holland Mathematical Library, V. 54. Elsevier Science & Technology Books, 1997.

[7] J. N. Cooper, R. B. Ellis, and A. B. Kahng. Asymmetric Binary Covering Codes. *Journal on Combinatorial Theory, Series A*, 100(2), 2002.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[9] L. Milanesi, M. Muselli, and P. Arrigo. Hamming-Clustering Method for Signals Prediction in 5' and 3' Regions of Eukaryotic Genes. *Computer Applications in Bioscience*, 12(5), 1996.

[10] I. Schwab, A. Kobsa, and I. Koychev. Learning User Interests through Positive Examples Using Content Analysis and Collaborative Filtering. *User Modeling and User-adapted Interaction*, 14(5), 2004.

[11] Y. N. Silva and S. Pearson. Exploiting Database Similarity Joins for Metric Spaces. *PVLDB*, 5(12), 2012.

[12] R. Vernica, M. J. Carey, and C. Li. Efficient Parallel Set-similarity Joins Using MapReduce. In *SIGMOD*, 2010.

[13] C. Whitelaw, A. Kehlenbeck, N. Petrovic, and L. Ungar. Web-scale Named Entity Recognition. In *CIKM*, 2008.

[14] C. Xiao, W. Wang, X. Lin, and J. X. Yu. Efficient Similarity Joins for Near Duplicate Detection. In *WWW*, 2008.