The Power of Simple Tabulation Hashing*

Mihai Pătraşcu
AT&T Labs
Florham Park, NJ
mip@alum.mit.edu

Mikkel Thorup
AT&T Labs
Florham Park, NJ
mthorup@research.att.com

ABSTRACT

Randomized algorithms are often enjoyed for their simplicity, but the hash functions used to yield the desired theoretical guarantees are often neither simple nor practical. Here we show that the simplest possible tabulation hashing provides unexpectedly strong guarantees.

The scheme itself dates back to Carter and Wegman (STOC'77). Keys are viewed as consisting of c characters. We initialize c tables $T_1, \ldots, T_c$ mapping characters to random hash codes. A key $x = (x_1, \ldots, x_c)$ is hashed to $T_1[x_1] \oplus \cdots \oplus T_c[x_c]$, where $\oplus$ denotes xor.

While this scheme is not even 4-independent, we show that it provides many of the guarantees that are normally obtained via higher independence, e.g., Chernoff-type concentration, min-wise hashing for estimating set intersection, and cuckoo hashing.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures

General Terms

Algorithms, Performance, Theory

Keywords

Tabulation Hashing, Linear Probing, Cuckoo Hashing, Minwise Independence, Concentration Bounds, Independence

1. INTRODUCTION

An important target of the analysis of algorithms is to determine whether there exist *practical* schemes, which enjoy mathematical *guarantees* on performance.

Hashing and hash tables are one of the most common inner loops in real-world computation, and are even built-in "unit cost" operations in high level programming languages that offer associative arrays. Often, these inner loops dominate the overall computation time. Knuth gave birth to the analysis of algorithms in 1963 [12] when he analyzed linear probing, the most popular practical implementation of hash tables. Assuming a perfectly random hash function, he bounded the expected number of probes. However, we do not have perfectly random hash functions. The approach of algorithms analysis is to understand when simple and practical hash functions work well. The most popular multiplication-based hashing schemes maintain the $O(1)$ running times when the sequence of operations has sufficient randomness [13]. However, they fail badly even for very simple input structures like an interval of consecutive keys [15, 17, 23], giving linear probing an undeserved reputation of being non-robust.

On the other hand, the approach of algorithm design (which may still have a strong element of analysis) is to construct (more complicated) hash functions providing the desired mathematical properties. This is usually done in the influential k-independence paradigm of Wegman and Carter [24]. It is known that 5-independence is sufficient [15] and necessary [17] for linear probing. Then one can use the best available implementation of 5-independent hash functions, the tabulation-based method of [22, 23].

Here we analyze simple tabulation hashing. This scheme views a key x as a vector of c characters $x_1, \ldots, x_c$. For each character position, we initialize a totally random table T_i, and then use the hash function

$$h(x) = T_1[x_1] \oplus \cdots \oplus T_c[x_c].$$

This is a well-known scheme dating back at least to Wegman and Carter [24]. From a practical view-point, tables T_i can be small enough to fit in fast cache, and the function is probably the easiest to implement beyond the bare multiplication. However, the scheme is only 3-independent, and was therefore assumed to have weak mathematical properties. We note that if the keys are drawn from a universe of size u, and hash values are machine words, the space required is $O(cu^{1/c})$ words. The idea is to make this fit in fast cache. We also note that the hash values are bit strings, so when we hash into bins, the number of bins is generally understood to be a power of two.

The challenge in analyzing simple tabulation is the significant dependence between keys. Nevertheless, we show that the scheme works in some of the most important randomized algorithms, including linear probing and several instances when $\Omega(\lg n)$-independence was previously needed. We confirm our findings by experiments: simple tabulation is competitive with just one 64-bit multiplication, and the hidden constants in the analysis appear to be very acceptable in practice.

*A full version of this paper is available as arXiv:1011.5200.

STOC'11, June 6–8, 2011, San Jose, California, USA.

In many cases, our analysis gives the first provably good *implementation* of an algorithm which matches the algorithm's conceptual simplicity if one ignores hashing.

Desirable properties. We will focus on the following popular properties of truly random hash functions.

- The worst-case query time of chaining is $O(\lg n/\lg\lg n)$ with high probability (w.h.p.). More generally, when distributing balls into bins, the bin load obeys Chernoff bounds.
- Linear probing runs in expected $O(1)$ time per operation. Variance and all constant moments are also $O(1)$.
- Cuckoo hashing: Given two tables of size $m \geq (1+\varepsilon)n$, it is possible to place a ball in one of two randomly chosen locations without *any* collision, with probability $1 - O(\frac{1}{n})$.
- Given two sets A, B, we have $\Pr_h[\min h(A) = \min h(B)] = \frac{|A\cap B|}{|A\cup B|}$. This can be used to quickly estimate the intersection of two sets, and follows from a property called *minwise independence*: for any $x \notin S$, $\Pr_h[x < \min h(S)] = \frac{1}{|S|+1}$.

As defined by Wegman and Carter [24] in 1977, a family $\mathcal{H} = \{h : [u] \to [m]\}$ of hash functions is k-independent if for any distinct $x_1, \ldots, x_k \in [u]$, the hash codes $h(x_1), \ldots, h(x_k)$ are independent random variables, and the hash code of any fixed x is uniformly distributed in $[m]$.

Chernoff bounds continue to work with high enough independence [18]; for instance, independence $\Theta(\frac{\lg n}{\lg\lg n})$ suffices for the bound on the maximum bin load. For linear probing, 5-independence is sufficient [15] and necessary [17]. For cuckoo hashing, $O(\lg n)$-independence suffices and at least 6-independence is needed [3]. While minwise independence cannot be achieved, one can achieve ε-minwise independence with the guarantee $(\forall)x \notin S, \Pr_h[x < \min h(S)] = \frac{1\pm\varepsilon}{|S|+1}$. For this, $\Theta(\lg\frac{1}{\varepsilon})$ independence is sufficient [10] and necessary [17]. (Note that the ε is a bias so it is a lower bound on how well set intersection can be approximated, with any number of independent experiments.)

The canonical construction of k-independent hash functions is a random degree $k-1$ polynomial in a prime field, which has small representation but $\Theta(k)$ evaluation time. Competitive implementations of polynomial hashing simulate arithmetic modulo Mersenne primes via bitwise operations. Even so, tabulation-based hashing with $O(u^{1/c})$ space and $O(ck)$ evaluation time is significantly faster [22]. The linear dependence on k is problematic, e.g., when $k \approx \lg n$.

Siegel [19] shows that a family with superconstant independence but $O(1)$ evaluation time requires $\Omega(u^{\varepsilon})$ space, i.e. it requires tabulation. He also gives a solution that uses $O(u^{1/c})$ space, $c^{O(c)}$ evaluation time, and achieves $u^{\Omega(1/c^2)}$ independence (which is superlogarithmic, at least asymptotically). The construction is non-uniform, assuming a certain small expander which gets used in a graph product. Dietzfelbinger and Rink [6] use universe splitting to obtain similar high independence with some quite different costs. Instead of being highly independent on the whole universe, their goal is to be highly independent on an unknown but fixed set S of size n. For some constant parameter γ, they tolerate an error probability of $n^{-\gamma}$. Assuming no error, their hash function is highly independent on S. The evaluation time is constant and the space is sublinear. For error probability $n^{-\gamma}$, each hash computation calls $O(\gamma)$ subroutines, each of which evaluates its own degree $O(\gamma)$ polynomial. The price for a lower error tolerance is therefore a slower hash function (even if we only count it as constant time in theory).

While polynomial hashing may perform better than its independence suggests, we have no positive example yet. On the tabulation front, we have one example of a good hash function that is not formally k-independent: cuckoo hashing works with an ad hoc hash function that combines space $O(n^{1/c})$ and polynomials of degree $O(c)$ [8].

1.1 Our results

Here we provide an analysis of simple tabulation showing that it has many of the desirable properties above. For most of our applications, we want to rule out certain obstructions with high probability. This follows immediately if certain events are independent, and the algorithms design approach is to pick a hash function guaranteeing this independence, usually in terms of a highly independent hash function.

Instead we here stick with simple tabulation with all its dependencies. This means that we have to struggle in each individual application to show that the dependencies are not fatal. However, from an implementation perspective, this is very attractive, leaving us with one simple and fast scheme for (almost) all our needs.

In all our results, we assume the number of characters is $c = O(1)$. The constants in our bounds will depend on c. Our results use a rather diverse set of techniques analyzing the table dependencies in different types of problems. For chaining and linear probing, we rely on some concentration results, which will also be used as a starting point for the analysis of min-wise hashing. Theoretically, the most interesting part is the analysis for cuckoo hashing, with a very intricate study of the random graph constructed by the two hash functions.

Chernoff bounds. We first show that simple tabulation preserves Chernoff-type concentration:

THEOREM 1. *Consider hashing n balls into $m \geq n^{1-1/(2c)}$ bins by simple tabulation. Let q be an additional* query ball*, and define X_q as the number of regular balls that hash into a bin chosen as a function of $h(q)$. Let $\mu = \mathbf{E}[X_q] = \frac{n}{m}$. The following probability bounds hold for any constant γ:*

$$(\forall)\delta \leq 1 : \Pr[|X_q - \mu| > \delta\mu] < 2e^{-\Omega(\delta^2\mu)} + m^{-\gamma} \tag{1}$$

$$(\forall)\delta = \Omega(1) : \Pr[X_q > (1+\delta)\mu] < (1+\delta)^{-\Omega((1+\delta)\mu)} + m^{-\gamma} \tag{2}$$

For any $m \leq n^{1-1/(2c)}$, every bin gets

$$n/m \pm O\left(\sqrt{n/m}\log^c n\right). \tag{3}$$

keys with probability $1 - n^{-\gamma}$.

Contrasting standard Chernoff bounds (see, e.g., [14]), Theorem 1 can only provide polynomially small probability, i.e. at least $n^{-\gamma}$ for any desired constant γ. In addition, the exponential dependence on μ in (1) and (2) is reduced by a constant which depends (exponentially) on the constants γ and c. It is possible to get some super polynomially small bounds with super constant γ but they are not as clean. An alternative way to understand the bound is that our tail bound depends exponentially on $\varepsilon\mu$, where ε decays to subconstant as we move more than inversely polynomially out in

the tail. Thus, our bounds are sufficient for any polynomially high probability guarantee. However, compared to the standard Chernoff bound, we would have to tolerate a constant factor more balls in a bin to get the same failure probability.

By the union bound (1) implies that with $m = \Theta(n)$ bins, no bin receives more than $O(\lg n / \lg \lg n)$ balls w.h.p. This is the first realistic hash function to achieve this fundamental property. Similarly, for linear probing with fill bounded below 1, (2) shows that the longest filled interval is of length $O(\log n)$ w.h.p.

Linear probing. Building on the above concentration bounds, we show that if the table size is $m = (1+\varepsilon)n$, then the expected time per operation is $O(1/\varepsilon^2)$, which asymptotically matches the bound of Knuth [12] for a truly random function. In particular, this compares positively with the $O(1/\varepsilon^{13/6})$ bound of [15] for 5-independent hashing.

Our proof is a combinatorial reduction that relates the performance of linear probing to concentration bounds. The results hold for any hash function with concentration similar to Theorem 1. To illustrate the generality of the approach, we also improve the $O(1/\varepsilon^{13/6})$ bound from [15] for 5-independent hashing to the optimal $O(1/\varepsilon^2)$. This was raised as an open problem in [15].

For simple tabulation, we get quite strong concentration results for the time per operation, e.g,, constant variance for constant ε. For contrast, with 5-independent hashing, the variance is only known to be $O(\log n)$ [15, 23].

Cuckoo hashing. In general, the cuckoo hashing algorithm fails iff the random bipartite graph induced by two hash functions contains a component with more vertices than edges. With truly random hashing, this happens with probability $\Theta(\frac{1}{n})$. Here we study the random graphs induced by simple tabulation, and obtain a rather unintuitive result: the optimal failure probability is inversely proportional to the *cube root* of the set size.

THEOREM 2. *Any set of n keys can be placed in two table of size $m = (1+\varepsilon)$ by cuckoo hashing and simple tabulation with probability $1 - O(n^{-1/3})$. There exist sets on which the failure probability is $\Omega(n^{-1/3})$.*

Thus, cuckoo hashing and simple tabulation are an excellent construction for a static dictionary. The dictionary can be built (in linear time) after trying $O(1)$ independent hash functions w.h.p., and later every query runs in constant worst-case time with two probes. We note that even though cuckoo hashing requires two independent hash functions, these essentially come for the cost of one in simple tabulation: the pair of hash codes can be stored consecutively, in the same cache line, making the running time comparable with evaluating just one hash function.

In the dynamic case, Theorem 2 implies that we expect $\Omega(n^{4/3})$ updates between failures requiring a complete rehash with new hash functions.

Our proof involves a complex understanding of the intricate, yet not fatal dependencies in simple tabulation. The proof is a (complicated) algorithm that assumes that cuckoo hashing has failed, and uses this knowledge to compress the random tables $T_1, \ldots, T_c$ below the entropy lower bound.

Using our techniques, it is also possible to show that if n balls are placed in $O(n)$ bins in an online fashion, choosing the least loaded bin at each time, the maximum load is $O(\lg \lg n)$ in expectation.

Minwise independence. In the full version, we show that simple tabulation is ε-minwise independent, for a vanishingly small ε (inversely polynomial in the set size). This would require $\Theta(\log n)$ independence by standard techniques.

THEOREM 3. *Consider a set S of $n = |S|$ keys and $q \notin S$. Then with h implemented by simple tabulation:*

$$\Pr[h(q) < \min h(S)] = \frac{1 \pm \varepsilon}{n}, \qquad \text{where } \varepsilon = O\left(\frac{\lg^2 n}{n^{1/c}}\right).$$

This can be used to estimate the size of set intersection by estimating:

$$\begin{aligned}\Pr[\min h(A) = \min h(B)] &= \sum_{x \in A \cap B} \Pr[x < \min h(A \cup B \setminus \{x\})] \\ &= \frac{|A \cap B|}{|A \cup B|} \cdot \left(1 \pm \widetilde{O}\left(\frac{1}{|A \cup B|^{1/c}}\right)\right).\end{aligned}$$

For good bounds on the probabilities, we would make multiple experiments with independent hash functions. An alternative based on a single hash function is that we for each set consider the k elements with the smallest hash values. We will also present concentration bounds for this alternative.

Fourth moment bounds. An alternative to Chernoff bounds in proving good concentration is to use bounded moments. In the full version of the paper, we analyze the 4^{th} moment of a bin's size when balls are placed into bins by simple tabulation. For a fixed bin, we show that the 4^{th} moment comes extremely close to that achieved by truly random hashing: it deviates by a factor of $1 + O(4^c/m)$, which is tiny except for a very large number of characters c. This would require 4-independence by standard arguments. This limited 4^{th} moment for a given bin was discovered independently by [1].

If we have a designated query ball q, and we are interested in the size of a bin chosen as a function of $h(q)$, the 4^{th} moment of simple tabulation is within a constant factor of that achieved by truly random hashing (on close inspection of the proof, that constant is at most 2). This would require 5-independence by standard techniques. (See [17] for a proof that 4-independence can fail quite badly when we want to bound the size of the bin in which q lands.) Our proof exploits an intriguing phenomenon that we identify in simple tabulation: in any fixed set of 5 keys, one of them has a hash code that is independent of the other four's hash codes.

Unlike our Chernoff-type bounds, the constants in the 4^{th} moment bounds can be analyzed quite easily, and are rather tame. Compelling applications of 4^{th} moment bounds were given by [11] and [21]. In [11], it was shown that any hash function with a good 4^{th} moment bound suffices for a nonrecursive version of quicksort, routing on the hypercube, etc. In [21], linear probing is shown to have constant expected performance if the hash function is a composition of universal hashing down to a domain of size $O(n)$, with a strong enough hash function on this small domain (i.e. any hash function with a good 4^{th} moment bound).

We will also use 4^{th} moment bounds to attain certain bounds of linear probing not covered by our Chernoff-type

bounds. In the case of small fill $\alpha = \frac{n}{m} = o(1)$, we use the 4^{th} moment bounds to show that the probability of a full hash location is $O(\alpha)$.

Pseudorandom numbers. The tables used in simple tabulation should be small to fit in the first level of cache. Thus, filling them with truly random numbers would not be difficult (e.g. in our experiments we use atmospheric noise from `random.org`). If the amount of randomness needs to be reduced further, we remark that all proofs continue to hold if the tables are filled by a $\Theta(\lg n)$-independent hash function (e.g. a polynomial with random coefficients).

With this modification, simple tabulation naturally lends itself to an implementation of a very efficient pseudorandom number generator. We can think of a pseudorandom generator as a hash function on range $[n]$, with the promise that each $h(i)$ is evaluated once, in the order of increasing i. To use simple tabulation, we break the universe into two, very lopsided characters: $[\frac{n}{R}] \times [R]$, for R chosen to be $\Theta(\lg n)$. Here the second coordinate is least significant, that is, (x, y) represents $xR + y$. During initialization, we fill $T_2[1\,..\,R]$ with R truly random numbers. The values of $T_1[1\,..\,n/R]$ are generated on the fly, by a polynomial of degree $\Theta(\lg n)$, whose coefficients were chosen randomly during initialization. Whenever we start a new row of the matrix, we can spend a relatively large amount of time to evaluate a polynomial to generate the next value r_1 which we store in a register. For the next R calls, we run sequentially through T_2, xoring each value with r_1 to provide a new pseudorandom number. With T_2 fitting in fast memory and scanned sequentially, this will be much faster than a single multiplication, and with R large, the amortized cost of generating r_1 is insignificant. The pseudorandom generator has all the interesting properties discussed above, including Chernoff-type concentration, minwise independence, and random graph properties.

Experimental evaluation. We performed an experimental evaluation of simple tabulation. Our implementation uses tables of 256 entries (i.e. using $c = 4$ characters for 32-bit data and $c = 8$ characters with 64-bit data). The time to evaluate the hash function turns out to be competitive with multiplication-based 2-independent functions, and significantly better than for hash functions with higher independence. We also evaluated simple tabulation in applications, in an effort to verify that the constants hidden in our analysis are not too large. Simple tabulation proved very robust and fast, both for linear probing and for cuckoo hashing.

Contents of extended abstract. We only have room to present the most essential results; namely the Chernoff style concentration from Theorem 1 and the results for Cuckoo hashing from Theorem 2.

Notation. We now introduce some notation that will be used throughout the proofs. We want to construct hash functions $h : [u] \to [m]$. We use simple tabulation with an alphabet of Σ and $c = O(1)$ characters. Thus, $u = \Sigma^c$ and $h(x_1, \ldots, x_c) = \bigoplus_{i=1}^{c} T_i[x_i]$. It is convenient to think of each hash code $T_i[x_i]$ as a fraction in $[0, 1)$ with large enough precision. We always assume m is a power of two, so an m-bit hash code is obtained by keeping only the most significant $\log_2 m$ bits in such a fraction. We always assume the table stores long enough hash codes, i.e. at least $\log_2 m$ bits.

Let $S \subset \Sigma^c$ be a set of $|S| = n$ keys, and let q be a query. We typically assume $q \notin S$, since the case $q \in S$ only involves trivial adjustments (for instance, when looking at the load of the bin $h(q)$, we have to add one when $q \in S$). Let $\pi(S, i)$ be the projection of S on the i-th coordinate, $\pi(S, i) = \{x_i \mid (\forall)x \in S\}$.

We define a *position-character* to be an element of $[c] \times \Sigma$. Then, the alphabets on each coordinate can be assumed to be disjoint: the first coordinate has alphabet $\{1\} \times \Sigma$, the second has alphabet $\{2\} \times \Sigma$, etc. Under this view, we can treat a key x as a *set* of q position-characters (on distinct positions). Furthermore, we can assume h is defined on position characters: $h((i, \alpha)) = T_i[\alpha]$. This definition is extended to keys (sets of position-characters) in the natural way $h(x) = \bigoplus_{\alpha \in x} h(\alpha)$.

When we say with high probability in r, we mean $1 - r^a$ for any desired constant a. Since $c = O(1)$, high probability in $|\Sigma|$ is also high probability in u. If we just say high probability, it is in n.

2. CONCENTRATION BOUNDS

This section proves Theorem 1, except branch (3) which is shown in the full version of the paper.

If n elements are hashed into $n^{1+\varepsilon}$ bins by a truly random hash function, the maximum load of any bin is $O(1)$ with high probability. First we show that simple tabulation preserves this guarantee. Building on this, we shows that the load of any fixed bin obeys Chernoff bounds. Finally we show that the Chernoff bound holds even for a bin chosen as a function of the query hash code, $h(q)$.

As stated in the introduction, the number of bins is always understood to be a power of two. This is because our hash values are xor'ed bit strings. If we want different numbers of bins we could view the hash values as fractions in the unit interval and divide the unit interval into subintervals. Translating our results to this setting is standard.

Hashing into Many Bins. The notion of peeling lies at the heart of most work in tabulation hashing. If a key from a set of keys contains one position-character that doesn't appear in the rest of the set, its hash code will be independent of the rest. Then, it can be "peeled" from the set, as its behavior matches that with truly random hashing. More formally, we say a set T of keys is peelable if we can arrange the keys of T in some order, such that each key contains a position-character that doesn't appear among the previous keys in the order.

LEMMA 4. *Suppose we hash $n \le m^{1-\varepsilon}$ keys into m bins, for some constant $\varepsilon > 0$. For any constant γ, all bins get less than $d = \min\left\{((1+\gamma)/\varepsilon)^c, 2^{(1+\gamma)/\varepsilon}\right\}$ keys with probability $\ge 1 - m^{-\gamma}$.*

PROOF. We will show that among any d elements, one can find a *peelable* subset of size $t \ge \max\{d^{1/c}, \lg d\}$. Then, a necessary condition for the maximum load of a bin to be at least d is that some bin contain t peelable elements. There are at most $\binom{n}{t} < n^t$ such sets. Since the hash codes of a peelable set are independent, the probability that a fixed set lands into a common bin is $1/m^{t-1}$. Thus, an upper bound on the probability that the maximum load is d can

be obtained: $n^t/m^{t-1} = m^{(1-\varepsilon)t}/m^{t-1} = m^{1-\varepsilon t}$. To obtain failure probability $m^{-\gamma}$, set $t = (1+\gamma)/\varepsilon$.

It remains to show that any set T of $|T| = d$ keys contains a large peelable subset. Since $T \subset \pi(T,1) \times \cdots \times \pi(T,c)$, it follows that there exists $i \in [c]$ with $|\pi(T,i)| \geq d^{1/c}$. Pick some element from T for every character value in $\pi(S,i)$; this is a peelable set of $t = d^{1/c}$ elements.

To prove $t \geq \log_2 d$, we proceed iteratively. Consider the coordinate giving the largest projection, $j = \arg\max_i |\pi(T,i)|$. As long as $|T| \geq 2$, $|\pi(T,j)| \geq 2$. Let α be the most popular value in T for the j-th character, and let $T^\star$ contain only elements with α on the j-th coordinate. We have $|T^\star| \geq |T|/|\pi(T,j)|$. In the peelable subset, we keep one element for every value in $\pi(T,j) \setminus \{\alpha\}$, and then recurse in $T^\star$ to obtain more elements. In each recursion step, we obtain $k \geq 1$ elements, at the cost of decreasing $\log_2 |T|$ by $\log_2(k+1)$. Thus, we obtain at least $\log_2 d$ elements overall. □

We note that, when the subset of keys of interest forms a combinatorial cube, the probabilistic analysis in the proof is sharp up to constant factors. In other words, the exponential dependence on c and γ is inherent.

Chernoff Bounds for a Fixed Bin. We study the number of keys ending up in a prespecified bin B. The analysis will define a total ording $\prec$ on the space of position-characters, $[c] \times \Sigma$. Then we will analyze the random process by fixing hash values of position-characters $h(\alpha)$ in the order $\prec$. The hash value of a key $x \in S$ becomes known when the position-character $\max_\prec x$ is fixed. For $\alpha \in [c] \times \Sigma$, we define the *group* $G_\alpha = \{x \in S \mid \alpha = \max_\prec x\}$, the set of keys for whom α is the last position-character to be fixed.

The intuition is that the contribution of each group G_α to the bin B is a random variable independent of the previous G_β's, since the elements G_α are shifted by a new hash code $h(\alpha)$. Thus, if we can bound the contribution of G_α by a constant, we can apply Chernoff bounds.

LEMMA 5. *There is an ordering $\prec$ such that the maximal group size is $\max_\alpha |G_\alpha| \leq n^{1-1/c}$.*

PROOF. We start with S being the set of all keys, and reduce S iteratively, by picking a position-character α as next in the order, and removing keys G_α from S. At each point in time, we pick the position-character α that would minimize $|G_\alpha|$. Note that, if we pick some α as next in the order, G_α will be the set of keys $x \in S$ which contain α and contain no other character that hasn't been fixed: $(\forall)\beta \in x \setminus \{\alpha\}, \beta \prec \alpha$.

We have to prove is that, as long as $S \neq \emptyset$, there exists α with $|G_\alpha| \leq |S|^{1-1/c}$. If some position i has $|\pi(S,i)| > |S|^{1/c}$, there must be some character α on position i which appears in less than $|S|^{1-1/c}$ keys; thus $|G_\alpha| \leq S^{1-1/c}$. Otherwise, $\pi(S,i) \leq |S|^{1/c}$ for all i. Then if we pick an arbitrary character α on some position i, have $|G_\alpha| \leq \prod_{j \neq i} |\pi(S,j)| \leq (|S|^{1/c})^{c-1} = |S|^{1-1/c}$. □

From now on assume the ordering $\prec$ has been fixed as in the lemma. This ordering partitions S into at most n non-empty groups, each containing at most $n^{1-1/c}$ keys. We say a group G_α is *d-bounded* if no bin contains more than d keys from G_α.

LEMMA 6. *Assume the number of bins is $m \geq n^{1-1/(2c)}$. For any constant γ, with probability $\geq 1 - m^{-\gamma}$, all groups are d-bounded where*

$$d = \min\left\{(2c(3+\gamma)^c, 2^{2c(3+\gamma)}\right\}$$

PROOF. Since $|G_\alpha| \leq n^{1-1/c} \leq m^{1-1/(2c)}$, by Lemma 4, we get that there are at most d keys from G_α in any bin with probability $1 - m^{-(2+\gamma)} \geq 1 - m^{-\gamma}/n$. The conclusion follows by union bound over the $\leq n$ groups. □

Henceforth, we assume that γ and d are fixed as in Lemma 6. Chernoff bounds (see [14, Theorem 4.1]) consider independent random variables $X_1, X_2, \cdots \in [0,d]$. Let $X = \sum_i X_i$, $\mu = \mathbf{E}[X]$, and $\delta > 0$, the bounds are:

$$\begin{aligned} \Pr[X \geq (1+\delta)\mu] &\leq \left(\frac{e^\delta}{(1+\delta)^{(1+\delta)}}\right)^{\mu/d} \\ \Pr[X \leq (1-\delta)\mu] &\leq \left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{\mu/d} \end{aligned} \tag{4}$$

Let X_α be the number of elements from G_α landing in the bin B. We are quite close to applying Chernoff bounds to the sequence X_α, which would imply the desired concentration around $\mu = \frac{n}{m}$. Two technical problems remain: X_α's are not d-bounded in the worst case, and they are not independent.

To address the first problem, we define the sequence of random variables $\hat{X}_\alpha$ as follows: if G_α is d-bounded, let $\hat{X}_\alpha = X_\alpha$; otherwise $\hat{X}_\alpha = |G_\alpha|/m$ is a constant. Observe that $\sum_\alpha \hat{X}_\alpha$ coincides with $\sum_\alpha X_\alpha$ if all groups are d-bounded, which happens with probability $1 - m^{-\gamma}$. Thus a probabilistic bound on $\sum_\alpha \hat{X}_\alpha$ is a bound on $\sum_\alpha X_\alpha$ up to an additive $m^{-\gamma}$ in the probability.

Finally, the $\hat{X}_\alpha$ variables are not independent: earlier position-character dictate how keys cluster in a later group. Fortunately (4) holds even if the distribution of each X_i is a function of $X_1, \ldots, X_{i-1}$, as long as the mean $\mathbf{E}[X_i \mid X_1, \ldots, X_{i-1}]$ is a fixed constant μ_i independent of $X_1, ..., X_{i-1}$. We claim that our means are fixed this way: regardless of the hash codes for $\beta < \alpha$, we will argue that $\mathbf{E}[\hat{X}_\alpha] = \mu_\alpha = |G_\alpha|/m$.

Observe that whether or not G_α is d-bounded is determined before $h(\alpha)$ is fixed in the order $\prec$. Indeed, α is the last position-character to be fixed for any key in G_α, so the hash codes of all keys in G_α have been fixed up to an xor with $h(\alpha)$. This final shift by $h(\alpha)$ is common to all the keys, so it cannot change whether or not two elements land together in a bin. Therefore, the choice of $h(\alpha)$ does not change if G_α is d-bounded.

After fixing all hash codes $\beta \prec \alpha$, we decide if G_α is d-bounded. If not, we set $\hat{X}_\alpha = |G_\alpha|/m$. Otherwise $\hat{X}_\alpha = X_\alpha$ is the number of elements we get in B when fixing $h(\alpha)$, and $h(\alpha)$ is a uniform random variable sending each element to B with probability $1/m$. Therefore $\mathbf{E}[\hat{X}_\alpha] = |G_\alpha|/m$. This completes the proof that the number of keys in bin B obeys Chernoff bounds from (4), which immediately imply (1) and (2) in Theorem 1.

The Load of a Query-Dependent Bin. When we are dealing with a special key q (a query), we may be interested in the load of a bin B_q, chosen as a function of the query's hash code, $h(q)$. We show that the above analysis also works

for the size of B_q, up to small constants. The critical change is to insist that the query position-characters come first in our ordering $\prec$:

LEMMA 7. *There is an ordering $\prec$ placing the characters of q first, in which the maximal group size is $2 \cdot n^{1-1/c}$.*

PROOF. After placing the characters of q at the beginning of the order, we use the same iterative construction as in Lemma 5. Each time we select the position-character α minimizing $|G_\alpha|$, place α next in the order $\prec$, and remove G_α from S. It suffices to prove that, as long as $S \neq \emptyset$, there exists a position-character $\alpha \notin q$ with $|G_\alpha| \leq 2 \cdot |S|^{1-1/c}$. Suppose in some position i, $|\pi(S,i)| > |S|^{1/c}$. Even if we exclude the query character q_i, there must be some character α on position i that appears in at most $|S|/(|\pi(S,i)|-1)$ keys. Since $S \neq \emptyset$, $|S|^{1/c} > 1$, so $|\pi(S,i)| \geq 2$. This means $|\pi(S,i)| - 1 \geq |S|^{1/c}/2$, so α appears in at most $2|S|^{1-1/c}$ keys. Otherwise, we have $\pi(S,i) \leq |S|^{1/c}$ for all i. Then, for any character α on position i, we have $|G_\alpha| \leq \prod_{j \neq i} |\pi(S,j)| \leq |S|^{1-1/c}$. □

The lemma guarantees that the first nonempty group contains the query alone, and all later groups have random shifts that are independent of the query hash code. We lost a factor two on the group size, which has no effect on our asymptotic analysis. In particular, all groups are d-bounded w.h.p. Letting X_α be the contribution of G_α to bin B_q, we see that the distribution of X_α is determined by the hash codes fixed previously (including the hash code of q, fixing the choice of the bin B_q). But $\mathbf{E}[X_\alpha] = |G_\alpha|/m$ holds irrespective of the previous choices. Thus, Chernoff bounds continue to apply to the size of B_q. This completes the proof of (1) and (2) in Theorem 1.

In Theorem 1 we limited ourselves to polynomially small error bounds $m^{-\gamma}$ for constant γ. However, we could also consider a super constant $\gamma = \omega(1)$ using the formula for d in Lemma 6. For the strongest error bounds, we would balance $m^{-\gamma}$ with the Chernoff bounds from (4). Such balanced error bounds would be messy, and we found it more appealing to elucidate the standard Chernoff-style behavior when dealing with polynomially small errors.

3. ANALYSIS OF CUCKOO HASHING

We begin with the negative side of our result:

OBSERVATION 8. *There exists a set S of n keys such that cuckoo hashing with simple tabulation hashing cannot place S into two tables of size $2n$ with probability $\Omega(n^{-1/3})$.*

PROOF. The hard instance is the 3-dimensional cube $[n^{1/3}]^3$. Here is a sufficient condition for cuckoo hashing to fail:

- there exist $a, b, c \in [n^{1/3}]^2$ with $h_0(a) = h_0(b) = h_0(c)$;
- there exist $x, y \in [n^{1/3}]$ with $h_1(x) = h_1(y)$.

If both happen, then the elements ax, ay, bx, by, cx, cy cannot be hashed. Indeed, on the left side $h_0(a) = h_0(b) = h_0(c)$ so they only occupy 2 positions. On the right side, $h_1(x) = h_1(y)$ so they only occupy 3 positions. In total they occupy $5 < 6$ positions.

The probability of 1. is asymptotically $(n^{2/3})^3/n^2 = \Omega(1)$. This is because tabulation (on two characters) is 3-independent. The probability of 2. is asymptotically $(n^{1/3})^2/n = \Omega(1/n^{1/3})$. So overall cuckoo hashing fails with probability $\Omega(n^{-1/3})$. □

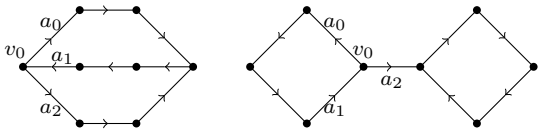

Figure 1: Minimal obstructions to cuckoo hashing.

Our positive result will effectively show that this is the worst possible instance: for any set S, the failure probability is $O(n^{-1/3})$.

The proof is an encoding argument. A tabulation hash function from $\Sigma^c \mapsto [m]$ has entropy $|\Sigma|^c \lg m$ bits; we have two random functions h_0 and h_1. If, under some event $\mathcal{E}$, one can encode the two hash functions h_0, h_1 using $\big(2|\Sigma|^c \lg m\big) - \gamma$ bits, it follows that $\Pr[\mathcal{E}] = O(2^{-\gamma})$. Letting $\mathcal{E}_S$ denote the event that cuckoo hashing fails on the set of keys S, we will demonstrate a saving of $\gamma = \frac{1}{3}\lg n - f(c,\varepsilon) = \frac{1}{3}\lg n - O(1)$ bits in the encoding. Note that we are analyzing simple tabulation on a *fixed* set of n keys, so both the encoder and the decoder know S.

We will consider various cases, and give algorithms for encoding some subset of the hash codes (we can afford $O(1)$ bits in the beginning of the encoding to say which case we are in). At the end, the encoder will always list all the remaining hash codes in order. If the algorithm chooses to encode k hash codes, it will use space at most $k \lg m - \frac{1}{3}\lg n + O(1)$ bits. That is, it will save $\frac{1}{3}\lg n - O(1)$ bits in the complete encoding of h_0 and h_1.

An easy way out. A *subkey* is a set of position-characters on distinct positions. If a is a subkey, we let $C(a) = \{x \in S \mid a \subseteq x\}$ be the set of "completions" of a to a valid key.

We first consider an easy way out: there subkeys a and b on the positions such that $|C(a)| \geq n^{2/3}, |C(b)| \geq n^{2/3}$, and $h_i(a) = h_i(b)$ for some $i \in \{0,1\}$. Then we can easily save $\frac{1}{3}\lg n - O(1)$ bits. First we write the set of positions of a and b, and the side of the collision ($c+1$ bits). There are at most $n^{1/3}$ subkeys on those positions that have $\geq n^{2/3}$ completions each, so we can write the identities of a and b using $\frac{1}{3}\lg n$ bits each. We write the hash codes h_i for all characters in $a \Delta b$ (the symmetric difference of a and b), skipping the last one, since it can be deduced from the collision. This uses $c + 1 + 2 \cdot \frac{1}{3}\lg n + (|a\Delta b| - 1)\lg m$ bits to encode $|a\Delta b|$ hash codes, so it saves $\frac{1}{3}\lg n - O(1)$ bits.

The rest of the proof assumes that there is no easy way out.

Walking Along an Obstruction. Consider the bipartite graph with m nodes on each side and n edges going from $h_0(x)$ to $h_1(x)$ for all $x \in S$. Remember that cuckoo hashing succeeds if and only if no component in this graph has more edges than nodes. Assuming cuckoo hashing failed, the encoder can find a subgraph with one of two possible obstructions: (1) a cycle with a chord; or (2) two cycles connected by a path (possibly a trivial path, i.e. the cycles simply share a vertex).

Let v_0 be a node of degree 3 in such an obstruction, and let its incident edges be a_0, a_1, a_2. The obstruction can be traversed by a walk that leaves v_0 on edge a_0, returns to v_0 on edge a_1, leaves again on a_2, and eventually meets itself. Other than visiting v_0 and the last node twice, no node or edge is repeated. See Figure 1.

Let $x_1, x_2, \ldots$ be the sequence of keys in the walk. The first key is $x_1 = a_0$. Technically, when the walk meets itself at the end, it is convenient to expand it with an extra key, namely the one it first used to get to the meeting point. This repeated key marks the end of the original walk, and we chose it so that it is not identical to the last original key. Let $x_{\le i} = \bigcup_{j \le i} x_j$ be the position-characters seen in keys up to x_i. Define $\hat{x}_i = x_i \setminus x_{<i}$ to be the position-characters of x_i not seen previously in the sequence. Let k be the first position such that $\hat{x}_{k+1} = \emptyset$. Such a k certainly exists, since the last key in our walk is a repeated key.

At a high level, the encoding algorithm will encode the hash codes of $\hat{x}_1, \ldots, \hat{x}_k$ in this order. Note that the obstruction, hence the sequence (x_i), depends on the hash functions h_0 and h_1. Thus, the decoder does not know the sequence, and it must also be written in the encoding.

For notational convenience, let $h_i = h_{i \bmod 2}$. This means that in our sequence x_i and x_{i+1} collide in their h_i hash code, that is $h_i(x_i) = h_i(x_{i+1})$. Formally, we define 3 subroutines:

ID(x): Write the identity of $x \in S$ in the encoding, which takes $\lg n$ bits.

HASHES(h_i, x_k): Write the hash codes h_i of the characters $\hat{x}_k$. This takes $|\hat{x}_k| \lg m$ bits.

COLL(x_i, x_{i+1}): Document the collision $h_i(x_i) = h_i(x_{i+1})$. We write all h_i hash codes of characters $\hat{x}_i \cup \hat{x}_{i+1}$ in some fixed order. The last hash code of $\hat{x}_i \Delta \hat{x}_{i+1}$ is redundant and will be omitted. Indeed, the decoder can compute this last hash code from the equality $h_i(x_i) = h_i(x_{i+1})$. Since $\hat{x}_{i+1} = x_{i+1} \setminus x_{\le i}$, $\hat{x}_{i+1} \setminus \hat{x}_i \neq \emptyset$, so there exists a hash code in $\hat{x}_i \Delta \hat{x}_{i+1}$. This subroutine uses $\big(|\hat{x}_i \cup \hat{x}_{i+1}| - 1\big) \lg m$ bits, saving $\lg m$ bits compared to the trivial alternative: HASHES(h_i, x_i); HASHES(h_i, x_{i+1}).

To decode the above information, the decoder will need enough context to synchronize with the coding stream. For instance, to decode COLL(x_i, x_{i+1}), one typically needs to know i, and the identities of x_i and x_{i+1}.

Our encoding begins with the value k, encoded with $O(\lg k)$ bits, which allows the decoder to know when to stop. The encoding proceeds with the output of the stream of operations:

$$\text{ID}(x_1);\text{HASHES}(h_0, x_1); \text{ID}(x_2); \text{COLL}(x_1, x_2);$$
$$\ldots \text{ ID}(x_k); \text{COLL}(x_k, x_{k-1}); \text{HASHES}(h_k, x_k)$$

We observe that for each $i > 1$, we save ε bits of entropy. Indeed, ID(x_i) uses $\lg n$ bits, but COLL(x_{i-1}, x_i) then saves $\lg m = \lg((1+\varepsilon)n) \ge \varepsilon + \lg n$ bits.

The trouble is ID(x_1), which has an upfront cost of $\lg n$ bits. We must devise algorithms that modify this stream of operations and save $\frac{4}{3} \lg n - O(1)$ bits, giving an overall saving of $\frac{1}{3} \lg n - O(1)$. (For intuition, observe that a saving that ignores the cost of ID(x_1) bounds the probability of an obstruction at some fixed vertex in the graph. This probability must be much smaller than $1/n$, so we can union bound over all vertices. In encoding terminology, this saving must be much more than $\lg n$ bits.)

We will use modifications to all types of operations. For instance, we will sometimes encode ID(x) with much less than $\lg n$ bits. At other times, we will be able to encode COLL(x_i, x_{i+1}) with the cost of $|\hat{x}_i \cup \hat{x}_{i+1}| - 2$ characters, saving $\lg n$ bits over the standard encoding.

Since we will make several such modifications, it is crucial to verify that they only touch distinct operations in the stream. Each modification to the stream will be announced at the beginning of the stream with a pointer taking $O(\lg k)$ bits. This way, the decoder knows when to apply the special algorithms. We note that terms of $O(\lg k)$ are negligible, since we are already saving εk bits by the basic encoding (ε bits per edge). For any k, $O(\lg k) \le \varepsilon k + f(c, \varepsilon) = k + O(1)$. Thus, if our overall saving is $\frac{1}{3} \lg n - O(\lg k) + \varepsilon k$, it achieves the stated bound of $\lg n - O(1)$.

Safe Savings. Remember that $\hat{x}_{k+1} = \emptyset$, which suggests that we can save a lot by local changes towards the end of the encoding. We have $x_{k+1} \subset x_{\le k}$, so $x_{k+1} \setminus x_{<k} \subseteq \hat{x}_k$. We will first treat the case when $x_{k+1} \setminus x_{<k}$ is a proper subset of $\hat{x}_k$ (including the empty subset). This is equivalent to $\hat{x}_k \not\subset x_{k+1}$.

LEMMA 9 (SAFE-STRONG). *If $\hat{x}_k \not\subset x_{k+1}$, we can save $\lg n - O(c \lg k)$ bits by changing* HASHES(x_k).

PROOF. We can encode ID(x_{k+1}) using $c \lg k$ extra bits, since it consists only of known characters from $x_{\le k}$. For each position $1 \ldots c$, it suffices to give the index of a previous x_i that contained the same position-character. Then, we will write all hash codes h_k for the characters in $\hat{x}_k$, except for some $\alpha \in \hat{x}_k \setminus x_{k+1}$. From $h_k(x_k) = h_k(x_{k+1})$, we have $h_k(\alpha) = h_k(x_k \setminus \{\alpha\}) \oplus h_k(x_{k+1})$. All quantities on the right hand side are known (in particular $\alpha \notin x_{k+1}$), so the decoder can compute $h_k(\alpha)$. □

It remains to treat the case when the last revealed characters of x_{k+1} are precisely $\hat{x}_k$: $\hat{x}_k \subset x_{k+1}$. That is, both x_k and x_{k+1} consist of $\hat{x}_k$ and some previously known characters. In this case, the collision $h_k(x_k) = h_k(x_{k+1})$ does not provide us any information, since it reduces to the trivial $h_k(\hat{x}_k) = h_k(\hat{x}_k)$. Assuming that we didn't take the "easy way out", we can still guarantee a more modest saving of $\frac{1}{3} \lg n$ bits:

LEMMA 10 (SAFE-WEAK). *Let K be the set of position-characters known before encoding* ID(x_i)*, and assume there is no easy way out. If $x_i \Delta x_{i+1} \subseteq x_{<i}$, then we can encode both* ID(x_i) *and* ID(x_{i+1}) *using a total of $\frac{2}{3} \lg n + O(c \lg |K|)$ bits.*

A typical case where we apply the lemma is $i = k$ and $K = x_{<k}$. If $\hat{x}_k \subset x_{k+1}$, we have $x_k \Delta x_{k+1} \subset K$. Thus, we can obtain ID(x_k) for roughly $\frac{2}{3} \lg n$ bits, which saves $\frac{1}{3} \lg n$ bits.

PROOF PROOF OF LEMMA 10. With $O(c \lg k)$ bits, we can code the subkeys $x_i \cap x_{<i}$ and $x_{i+1} \cap x_{<i}$. It remains to code $z = x_i \setminus x_{<i} = x_{i+1} \setminus x_{<i}$. Since z is common to both keys x_i and x_{i+1}, we have that $x_i \setminus z$ and $x_{i+1} \setminus z$ are subkeys on the same positions. With no easy way out and $h_i(x_i \setminus z) = h_i(x_{i+1} \setminus z)$, we must have $|C(x_i \setminus z)| \le n^{2/3}$ or $|C(x_{i+1} \setminus z)| \le n^{2/3}$. In the former case, we code z as a member of $C(x_i \setminus z)$ with $\lceil \frac{2}{3} \lg n \rceil$ bits; otherwise we code z as member of $C(x_{i+1} \setminus z)$. □

Piggybacking. Before moving forward, we present a general situation when we can save $\lg n$ bits by modifying a COLL(x_i, x_{i+1}) operation:

LEMMA 11. *We can save* $\lg n - O(\lg k)$ *bits by modifying* $\textsc{Coll}(x_i, x_{i+1})$ *if we have identified two (sub)keys* e *and* f *satisfying:*

$$h_i(e) = h_i(f);\ \ e\Delta f \subset x_{\le i+1};\ \ \emptyset \neq (e\Delta f)\backslash x_{<i} \neq (x_i \Delta x_{i+1})\backslash x_{<i}.$$

PROOF. In the typical encoding of $\textsc{Coll}(x_i, x_{i+1})$, we saved one redundant character from $h_i(x_i) = h_i(x_{i+1})$, which is an equation involving $(x_i \Delta x_{i+1}) \setminus x_{<i}$ and some known characters from $x_{<i}$. The lemma guarantees a second linearly independent equation over the characters $\hat{x}_i \cup \hat{x}_{i+1}$, so we can save a second redundant character.

Formally, let α be a position-character of $(e\Delta f) \setminus x_{<i}$, and β a position-character in $(x_i \Delta x_{i+1}) \setminus x_{<i}$ but outside $(e\Delta f)\setminus x_{<i}$. Note $\beta \neq \alpha$ and such a β exists by assumption. We write the h_i hash codes of position characters $(\hat{x}_i \cup \hat{x}_{i+1})\setminus \{\alpha, \beta\}$. The hash $h_i(\alpha)$ can be deduced since α is the last unknown in the equality $h_i(e\setminus f) = h_i(f\setminus e)$. The hash $h_i(\beta)$ can be deduced since it is the last unknown in the equality $h_i(x) = h_i(x_{i+1})$. □

While the safe saving ideas only require simple local modifications to the encoding, they achieve a weak saving of $\frac{1}{3}\lg n$ bits for the case $\hat{x}_k \subset x_{k+1}$. A crucial step in our proof is to obtain a saving of $\lg n$ bits for this case. We do this by one of the following two lemmas:

LEMMA 12 (ODD-SIZE SAVING). *Consider two edges* e, f *and an* $i \le k-2$ *satisfying:*

$$h_{i+1}(e) = h_{i+1}(f); \qquad e\backslash x_{\le i} \neq f\backslash x_{\le i}; \qquad e\backslash x_{\le i+1} = f\backslash x_{\le i+1}.$$

We can save $\lg n - O(c \lg k)$ *bits by changing* $\textsc{Coll}(x_{i+1}, x_{i+2})$.

PROOF. We apply Lemma 11 with the subkeys $\tilde{e} = e \setminus f$ and $\tilde{f} = f \setminus e$. We can identify these in $O(c \lg k)$ bits, since they only contain characters of $x_{\le i+1}$. Since e and f have different free characters before $\hat{x}_{i+1}$, but identical free characters afterward, it must be that $\tilde{e} \cup \tilde{f} \subset x_{i+1}$ by $\tilde{e} \cup \tilde{f} \not\subseteq x_{\le i}$. To show $(e\Delta f) \setminus x_{<i} \neq (x_{i+1}\Delta x_{i+2}) \setminus x_{\le i}$, remark that $\hat{x}_{i+2} \neq \emptyset$ and $\hat{x}_{i+2}$ cannot have characters of $\tilde{e} \cup \tilde{f}$. Thus, Lemma 11 applies. □

LEMMA 13 (PIGGYBACKING). *Consider two edges* e, f *and an* $i \le k-1$ *satisfying:*

$$h_i(e) = h_i(f); \qquad e\backslash x_{\le i} \neq f\backslash x_{\le i}; \qquad e\backslash x_{\le i+1} = f\backslash x_{\le i+1}.$$

We can encode $\textsc{Id}(e)$ *and* $\textsc{Id}(f)$ *using only* $O(c \lg k)$ *bits, after modifications to* $\textsc{Id}(x_i)$, $\textsc{Id}(x_{i+1})$, *and* $\textsc{Coll}(x_i, x_{i+1})$.

The proof of this lemma is more delicate, and is given below. The difference between the two lemmas is the parity (side in the bipartite graph) of the collision of x_i and x_{i+1} versus the collision of e and f. In the second result, we cannot actually save $\lg n$ bits, but we can encode $\textsc{Id}(e)$ and $\textsc{Id}(f)$ almost for free: we say e and f piggyback on the encodings of x_i and x_{i+1}.

Through a combination of the two lemmas, we can always achieve a saving $\lg n$ bits in the case $\hat{x}_k \subset x_{k+1}$, improving on the safe-weak bound:

LEMMA 14. *Assume* k *is minimal such that* $\hat{x}_k \subset x_{k+1}$. *We can save* $\lg n - O(c \lg k)$ *bits if we may modify any operations in the stream, up to those involving* x_{k+1}.

PROOF. We will choose $e = x_k$ and $f = x_{k+1}$. We have $e \setminus x_{<k} = f \setminus x_{<k} = \hat{x}_k$. On the other hand, $e \setminus x_1 \neq f \setminus x_1$ since x_1 only reveals one character per position. Thus there must be some $1 \le i < k-1$ where the transition happens: $e \setminus x_{\le i} \neq f \setminus x_{\le i}$ but $e \setminus x_{\le i+1} = f \setminus x_{\le i+1}$. If i has the opposite parity compared to k, Lemma 12 saves a $\lg n$ term. (Note that $i \le k-2$ as required by the lemma.)

If i has the same parity as k, Lemma 13 gives us $\textsc{Id}(x_k)$ at negligible cost. Then, we can remove the operation $\textsc{Id}(x_k)$ from the stream, and save $\lg n$ bits. (Again, note that $i \le k-2$ as required.) □

PROOF OF LEMMA 13. The lemma assumed $e\setminus x_{\le i} \neq f \setminus x_{\le i}$ but $e \setminus x_{\le i+1} = f \setminus x_{\le i+1}$. Therefore, $e\Delta f \subset x_{\le i+1}$ and $(e\Delta f) \cap \hat{x}_{i+1} \neq \emptyset$. Lemma 11 applies if we furthermore have $(e\Delta f)\setminus x_{<i} \neq (x_i \Delta x_{i+1})\setminus x_{<i}$. If the lemma applies, we have a saving of $\lg n$, so we can afford to encode $\textsc{Id}(e)$. Then $\textsc{Id}(f)$ can be encoded using $O(c \lg k)$ bits, since f differs from e only in position-characters from $x_{\le i+1}$.

If the lemma does not apply, we have a lot of structure on the keys. Let $y = \hat{x}_i \setminus (e \cup f)$ and $g = e \setminus x_{\le i+1} = f \setminus x_{\le i+1}$. We must have $y \subset x_{i+1}$, for otherwise $\hat{x}_i \setminus x_{i+1}$ contains an elements outside $e\Delta f$ and the lemma applies. We must also have $\hat{x}_{i+1} \subset e \cup f$.

We can write $\textsc{Id}(x_i)$, $\textsc{Id}(x_{i+1})$, $\textsc{Id}(e)$, and $\textsc{Id}(f)$ using $2\lg n + O(c \lg k)$ bits in total, as follows:

- the coordinates on which y and g appear, taking $2c$ bits.
- the value of y using Huffman coding. Specifically, we consider the projection of all n keys on the coordinates of y. In this distribution, y has frequency $\frac{C(y)}{n}$, so its Huffman code will use $\lg \frac{n}{C(y)} + O(1)$ bits.
- the value of g using Huffman coding. This uses $\lg \frac{n}{C(g)} + O(1)$ bits.
- if $C(y) \le C(g)$, we write x_i and x_{i+1}. Each of these requires $\lceil \log_2 C(y) \rceil$ bits, since $y \subset x_i, x_{i+1}$ and there are $C(y)$ completions of y to a full key. Using an additional $O(c \lg k)$ bits, we can write $e \cap x_{\le i+1}$ and $f \cap x_{\le i+1}$. Remember that we already encoded $g = e \setminus x_{\le i+1} = f \setminus x_{\le i+1}$, so the decoder can recover e and f.
- if $C(g) < C(y)$, we write e and f, each requiring $\lceil \log_2 C(g) \rceil$ bits. Since we know $y = \hat{x}_i \setminus (e \cup f)$, we can write x_i using $O(c \lg k)$ bits: write the old characters outside $\hat{x}_i$, and which positions of $e \cup f$ to reuse in $\hat{x}_i$. We showed $\hat{x}_{i+1} \subset e \cup f$, so we can also write x_{i+1} using $O(c \lg k)$.

Overall, the encoding uses space: $\lg \frac{n}{C(\xi)} + \lg \frac{n}{C(\hat{e}_{i+1})} + 2\lg \min\left\{C(\xi), C(\hat{e}_{i+1})\right\} + O(c \lg k) \le 2\lg n + O(c \lg k)$. □

Putting it Together. We now show how to obtain a saving of at least $\frac{4}{3}\lg n - O(c \lg k)$ bits by a careful combination of the above techniques. Recall that our starting point is three edges a_0, a_1, a_2 with $h_0(a_0) = h_0(a_1) = h_0(a_2)$. The walk $x_1, ..., x_{k+1}$ started with $x_1 = a_0$ and finished when $\hat{x}_{k+1} = \emptyset$. We will now involve the other starting edges a_1 and a_2. The analysis will split into many cases, each ended by a '$\diamond$'.

Case 1: One of a_1 and a_2 contains a free character. Let $j \in \{1, 2\}$ such that $a_j \not\subseteq x_{\le k}$. Let $y_1 = a_j$. We consider a walk $y_1, y_2, \ldots$ along the edges of the obstruction.Let $\hat{y}_i = y_i \setminus x_{\le k} \setminus y_{\le i}$ be the free characters of y_i

(which also takes all x_i's into consideration). We stop the walk the first time we observe $\hat{y}_{\ell+1} = \emptyset$. This must occur, since the graph is finite and there are no leaves (nodes of degree one) in the obstruction. Thus, at the latest the walk stops when it repeats an edge.

We use the standard encoding for the second walk:

$$\textsc{Id}(y_1);\textsc{Coll}(a_0, y_1); \textsc{Id}(y_2); \textsc{Coll}(y_2, y_1);$$
$$\ldots;\textsc{Id}(y_\ell); \textsc{Coll}(y_{\ell-1}, y_\ell); \textsc{Hashes}(h_\ell, y_\ell)$$

Note that every pair $\textsc{Id}(y_j), \textsc{Coll}(y_{j-1}, y_j)$ saves ε bits, including the initial $\textsc{Id}(y_1), \textsc{Coll}(a_0, y_1)$. To end the walk, we can use one of the safe savings of Lemmas 9 and 10. These give a saving of $\frac{1}{3}\lg n - O(c\lg(\ell + k))$ bits, by modifying only $\textsc{Hashes}(h_\ell, y_\ell)$ or $\textsc{Id}(y_\ell)$. These local changes cannot interfere with the first walk, so we can use any technique (including piggybacking) to save $\lg n - O(c \log k)$ bits from the first walk. We obtain a total saving of $\frac{4}{3}\lg n - O(1)$, as required. ◇

We are left with the situation $a_1 \cup a_2 \subseteq x_{\leq k}$. This includes the case when a_1 and a_2 are actual edges seen in the walk $x_1, \ldots, x_k$.

Let t_j be the first time a_j becomes known in the walk; that is, $a_j \not\subseteq x_{<t_j}$ but $a_j \subseteq x_{\leq t_j}$. By symmetry, we can assume $t_1 \leq t_2$. We begin with two simple cases.

Case 2: For some $j \in \{1, 2\}$, t_j is even and $t_j < k$. We will apply Lemma 11 and save $\lg n - O(c \lg k)$ bits by modifying $\textsc{Coll}(x_{t_j}, x_{t_j+1})$. Since $t_j < k$, this does not interact with safe savings at the end of the stream, so we get total saving of at least $\frac{4}{3}\lg n - O(c \lg k)$.

We apply Lemma 11 on the keys $e = a_0$ and $f = a_j$. We must first write $\textsc{Id}(a_j)$, which takes $O(c \lg k)$ bits given $x_{\leq k}$. We have $a_0 \cup a_j \subseteq x_{\leq t_j}$ by definition of t_j. Since $a_j \cap \hat{x}_{t_j} \neq \emptyset$ and $\hat{x}_{t_j+1} \cap (a_j \cup a_0) = \emptyset$, the lemma applies. ◇

Case 3: For some $j \in \{1, 2\}$, t_j is odd and $a_j \setminus x_{<t_j-1} \neq \hat{x}_{t_j-1}\Delta\hat{x}_{t_j}$. This assumption is exactly what we need to apply Lemma 11 with $e = a_0$ and $f = a_j$. Note that $h_0(e) = h_0(f)$ and t_j is odd, so the lemma modifies $\textsc{Coll}(x_{t_j-1}, x_{t_j})$. The lemma can be applied in conjunction with any safe saving, since the safe savings only require modifications to $\textsc{Id}(x_k)$ or $\textsc{Hashes}(h_k, x_k)$. ◇

We now deal with two cases when $t_1 = t_2$ (both being odd or even). These require a combination of piggybacking followed by safe-weak savings. Note that in the odd case, we may assume $a_1 \setminus x_{<t-1} = a_2 \setminus x_{<t-1} = \hat{x}_{t-1}\Delta\hat{x}_t$ (due to case 3 above), and in the even case we may assume $t_1 = t_2 = k$ (due to case 2 above).

Case 4: $t_1 = t_2 = t$ is odd and $a_1 \setminus x_{<t-1} = a_2 \setminus x_{<t-1} = \hat{x}_{t-1}\Delta\hat{x}_t$. We first get a_1 and a_2 by piggybacking or odd-side saving. Let i be the largest value such that $a_1 \setminus x_{\leq i} \neq a_2 \setminus x_{\leq i}$. Since $a_1 \setminus x_{<t-1} = a_2 \setminus x_{<t-1}$, we have $i \leq t-3$. The last key that piggybacking or odd-side saving can interfere with is x_{t-2}.

We will now use the safe-weak saving of Lemma 10 to encode $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$. The known characters are $K = x_{<t-1} \cup a_1 \cup a_2$, so $x_{t-1}\Delta x_t \subseteq K$. Lemma 10 codes both $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$ with $\frac{2}{3}\lg n + O(c \lg k)$ bits, which represents a saving of roughly $\frac{4}{3}\lg n$ over the original encoding of the two identities. We don't need any more savings from the rest of the walk after x_t. ◇

Case 5: $t_1 = t_2 = k$ is even. Thus, k is even and the last characters of a_1 and a_2 are only revealed by $\hat{x}_k$.

LEMMA 15. *We can save $2\lg n - O(c \lg k)$ bits by modifying $\textsc{Hashes}(h_k, x_k)$, unless both: (1) $a_1 \cap \hat{x}_k = a_2 \cap \hat{x}_k$; and (2)$\hat{x}_k \setminus x_{k+1}$ is the empty set or equal to $a_1 \cap \hat{x}_k$.*

PROOF. The h_0 hash codes of the following 3 subkeys are known from the hash codes in $x_{<k}$: $a_1 \cap \hat{x}_k$, $a_2 \cap \hat{x}_k$ (both because we know $h_0(a_0) = h_0(a_1) = h_0(a_2)$), and $\hat{x}_k \setminus x_{k+1}$ (since x_k and x_{k+1} collide). If two of these subsets are distinct and nonempty, we can choose two characters α and β from their symmetric difference. We can encode all characters of $\hat{x}_k$ except for α and β, whose hash codes can be deduced for free.

Since $a_j \cap \hat{x}_k \neq$ in the current case, the situations when we can find two distinct nonempty sets are: (1) $a_1 \cap \hat{x}_k \neq a_2 \cap \hat{x}_k$; or (2) $a_1 \cap \hat{x}_k = a_2 \cap \hat{x}_k$ but $\hat{x}_k \setminus x_{k+1}$ is nonempty and different from them. □

From now on assume the lemma fails. We can still save $\lg n$ bits by modifying $\textsc{Hashes}(h_k, x_k)$. We reveal all hash codes of $\hat{x}_k$, except for one position-character $\alpha \in a_1 \cap \hat{x}_k$. We then specify $\textsc{Id}(a_1)$, which takes $O(c \lg k)$ bits. The hash $h_0(\alpha)$ can then be deduced from $h_0(a_1) = h_0(a_0)$.

We will now apply piggybacking or odd-side saving to a_1 and a_2. Let i be the largest value with $a_1 \setminus x_{\leq i} \neq a_2 \setminus x_{\leq i}$. Note that $a_1 \setminus x_{<k} = a_2 \setminus x_{<k}$, so $i < k - 1$. If i is odd, Lemma 12 (odd-side saving) can save $\lg n$ bits by modifying $\textsc{Coll}(x_{i+1}, x_{i+2})$; this works since $i + 2 \leq k$. If i is even, Lemma 13 (piggybacking) can give use $\textsc{Id}(a)$ and $\textsc{Id}(b)$ at a negligible cost of $O(c \lg k)$ bits. This doesn't touch anything later than $\textsc{Id}(x_{i+1})$, where $i + 1 < k$.

When we arrive at $\textsc{Id}(x_k)$, we know the position characters $K = x_{<k} \cup a_1 \cup a_2$. This means that $x_k\Delta x_{k+1} \subseteq K$, because $\hat{x}_k \setminus x_{k+1}$ is either empty or a subset of a_1. Therefore, we can use weak-safe savings from Lemma 10 to code $\textsc{Id}(x_k)$ in just $\frac{1}{3}\lg n + O(c \lg k)$ bits. In total, we have save at least $\frac{4}{3}\lg n - O(c \lg k)$ bits. ◇

It remains to deal with distinct t_1, t_2, i.e. $t_1 < t_2 \leq k$. If one of the numbers is even, it must be $t_2 = k$, and then t_1 must be odd (due to case 2). By Case 3, if t_j is odd, we also know $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1}\Delta\hat{x}_{t_j}$. Since these cases need to deal with at least one odd t_j, the following lemma will be crucial:

LEMMA 16. *If $t_j \leq k$ is odd and $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1}\Delta\hat{x}_{t_j}$, we can code $\textsc{Id}(x_{t_j-1})$ and $\textsc{Id}(x_{t_j})$ with $\frac{3}{2}\lg n + O(c \lg k)$ bits in total.*

PROOF. Consider the subkey $y = \hat{x}_{t_j-1} \setminus x_{t_j}$. We first specify the positions of y using c bits. If $C(y) \geq \sqrt{n}$, there are at most $\sqrt{n}$ possible choices of y, so we can specify y with $\frac{1}{2}\lg n$ bits. We can also identify x_{t_j} with $\lg n$ bits. Then $\textsc{Id}(x_{t_j-1})$ requires $O(c \lg k)$ bits, since $x_{t_j-1} \subseteq y \cup x_{t_j} \cup x_{<t_j-1}$.

If $C(y) \leq \sqrt{n}$, we first specify $\textsc{Id}(x_{t_j-1})$ with $\lg n$ bits. This gives us the subkey $y \subseteq x_{t_j-1}$. Since $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1}\Delta\hat{x}_{t_j}$, it follows that $y \subset a_j$. Thus, we can write $\textsc{Id}(a_j)$ using $\lg C(y) \leq \lg \frac{1}{2}\lg n$ bits. Since $x_{t_j} \subseteq x_{\leq t_j-1} \cup a_j$, we get $\textsc{Id}(x_{t_j})$ for an additional $O(c \lg k)$ bits. □

Case 6: Both t_1 and t_2 are odd, $t_1 < t_2 < k$, and for all $j \in \{1, 2\}$, $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1}\Delta\hat{x}_{t_j}$. We apply Lemma 16 for both $j = 1$ and $j = 2$, and save $\lg n$ bits in coding $\textsc{Id}(x_{t_1-1})$, $\textsc{Id}(x_{t_1})$, $\textsc{Id}(x_{t_2-1})$, and $\textsc{Id}(x_{t_2})$. These are all distinct keys, because $t_1 < t_2$ and both are odd. Since $t_2 < k$, we can combine this with any safe saving. ◇

Case 7: $t_2 = k$ is even and $t_1 < k$ is odd with $a_1 \setminus x_{<t_1-1} = \hat{x}_{t_1-1}\Delta\hat{x}_{t_1}$. We apply Lemma 16 for $j = 1$, and save $\frac{1}{2}\lg n - O(c\lg k)$ bits in coding $\textsc{Id}(x_{t_1-1})$, $\textsc{Id}(x_{t_1})$. We also save $\lg n$ bits by modifying $\textsc{Hashes}(h_0, x_k)$. We reveal all hash codes of $\hat{x}_k$, except for one position-character $\alpha \in a_2 \cap \hat{x}_k$ (which is a nonempty set since $t_2 = k$). We then specify $\textsc{Id}(a_2)$, which takes $O(c\lg k)$ bits. The hash $h_0(\alpha)$ can then be deduced from $h_0(a_2) = h_0(a_0)$. ◇

Case 8: Both t_1 and t_2 are odd, $t_1 < t_2 = k$, and for all $j \in \{1, 2\}$, $a_j \setminus x_{<t_j-1} = \hat{x}_{t_j-1}\Delta\hat{x}_{t_j}$. To simplify notation, let $t_1 = t$. This case is the most difficult. If we can apply strong-safe saving as in Lemma 9, we save $\lg n$ by modifying $\textsc{Hashes}(h_k, x_k)$. We also save $\lg n$ by two applications of Lemma 16, coding $\textsc{Id}(x_{t-1})$, $\textsc{Id}(x_t)$, $\textsc{Id}(x_{k-1})$, and $\textsc{Id}(x_k)$. These don't interact since $t < k$ and both are odd.

The strong-safe saving fails if $\hat{x}_k \subset x_{k+1}$. We will attempt to piggyback for x_k and x_{k+1}. Let i be the largest value such that $x_k \setminus x_{\le i} \neq x_{k+1} \setminus x_{\le i}$. If i is even, we get an odd-side saving of $\lg n$ (Lemma 12). Since this does not affect any identities, we can still apply Lemma 16 to save $\frac{1}{2}\lg n$ on the identities $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$.

Now assume i is odd. We have real piggybacking, which may affect the coding of $\textsc{Id}(x_i)$, $\textsc{Id}(x_{i+1})$ and $\textsc{Id}(x_k)$. Since both i and t are odd, there is at most one common key between $\{x_i, x_{i+1}\}$ and $\{x_{t-1}, x_t\}$. We consider two cases:

- Suppose $x_{t-1} \notin \{x_i, x_{i+1}\}$. Let $y = \hat{x}_{t-1} \setminus x_t$. After piggybacking, which in particular encodes x_t, we can encode $\textsc{Id}(x_{t-1})$ in $\lg \frac{n}{C(y)} + O(c\lg k)$ bits. Indeed, we can write the positions of y with c bits and then the identity of y using Huffman coding for all subkeys on those positions. Finally the identity of x_{t-1} can be written in $O(c\lg k)$ bits, since $x_{t-1} \subset x_{<t-1} \cup y \cup x_t$.
- Suppose $x_t \notin \{x_i, x_{i+1}\}$. Let $y = \hat{x}_t \setminus x_{t-1}$. As above, we can write $\textsc{Id}(x_t)$ using $\lg \frac{n}{C(y)} + O(c\lg k)$ bits, after piggybacking.

If $C(y) \ge n^{1/3}$, we have obtained a total saving of $\frac{4}{3}\lg n - O(c\lg k)$: a logarithmic term for $\textsc{Id}(x_k)$ from piggybacking, and $\frac{1}{3}\lg n$ for $\textsc{Id}(x_{t-1})$ or $\textsc{Id}(x_t)$.

Now assume that $C(y) \le n^{1/3}$. In this case, we do *not* use piggybacking. Instead, we use a variation of Lemma 16 to encode $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$. First we code the one containing y with $\lg n$ bits. Since $a_1 \setminus x_{<t-1} = \hat{x}_{t-1}\Delta\hat{x}_t$, and therefore $y \subset a_1$, we have $y \subset a_1$. We code $\textsc{Id}(a_1)$ with $\lg C(y) \le \frac{1}{3}\lg n$ bits. We obtain the other key among x_{t-1} and x_t using $O(c\lg k)$ bits, since all its characters are known. Thus we have coded $\textsc{Id}(x_{t-1})$ and $\textsc{Id}(x_t)$ with $\frac{4}{3}\lg n + O(c\lg k)$ bits, for a saving of roughly $\frac{2}{3}\lg n$ bits.

Next we consider the coding of $\textsc{Id}(x_{k-1})$ and $\textsc{Id}(x_k)$. We know that $a_2 \setminus x_{<k-1} = \hat{x}_{k-1}\Delta\hat{x}_k$ and $\hat{x}_k \subset x_{k+1}$. Lemma 16 would guarantee a saving of $\frac{1}{2}\lg n$ bits. However, we will perform an analysis like above, obtaining a saving of $\frac{2}{3}\lg n$ bits.

Let $y = \hat{x}_{k-1} \setminus x_k$. First assume $C(y) \ge n^{1/3}$. We use the safe-weak saving of Lemma 10 to encode $\textsc{Id}(x_k)$ using $\frac{2}{3}\lg n$ bits. We then encode the subkey y using $\lg \frac{n}{C(y)} + O(c) \le \frac{2}{3}\lg n + O(c)$ bits, and finally x_{k-1} using $O(c\lg k)$ bits. This obtains both $\textsc{Id}(x_{k-1})$ and $\textsc{Id}(x_k)$ using $\frac{4}{3}\lg n + O(c\lg k)$ bits.

Now assume $C(y) \le n^{1/3}$. We first code $\textsc{Id}(x_{k-1})$ using $\lg n$ bits. This gives us y for the price of c bits. But $a_2 \setminus x_{<k-1} = \hat{x}_{k-1}\Delta\hat{x}_k$, so $y \subset a_2$, and we can code $\textsc{Id}(a_2)$ using $\lg C(y) \le \frac{1}{3}\lg n$ bits. Then $\textsc{Id}(x_k)$ can be coded with $O(c\lg k)$ bits. Again, we obtain both $\textsc{Id}(x_{k-1})$ and $\textsc{Id}(x_k)$ for the price of $\frac{4}{3}\lg n + O(c\lg k)$ bits. ◇

This completes our analysis of cuckoo hashing.

4. REFERENCES

[1] V. Braverman, K.-M. Chung, Z. Liu, M. Mitzenmacher, and R. Ostrovsky. AMS without 4-wise independence on product domains. In STACS'10, pages 119–130.

[2] L. Carter and M. N. Wegman. Universal classes of hash functions. *J. Comp. Sys. Sci.*, 18(2):143–154, 1979. See also STOC'77.

[3] J. S. Cohen and D. M. Kane. Bounds on the independence required for cuckoo hashing. Manuscript, 2009.

[4] M. Dietzfelbinger. Universal hashing and k-wise independent random variables via integer arithmetic without primes. In STACS'96, pages 569–580.

[5] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. *J. Alg.*, 25(1):19–51, 1997.

[6] M. Dietzfelbinger and M. Rink. Applications of a splitting trick. In ICALP'09, pages 354–365.

[7] M. Dietzfelbinger and U. Schellbach. On risks of using cuckoo hashing with simple universal hash classes. In SODA'09, pages 795–804.

[8] M. Dietzfelbinger and P. Woelfel. Almost random graphs with simple hash functions. In STOC'03, pp. 629–638.

[9] L. Gerber. An extension of Bernoulli's inequality. *Amer. Math. Monthly*, 75:875–876, 1968.

[10] P. Indyk. A small approximately min-wise independent family of hash functions. *J. Alg.*, 38(1):84–90, 2001. See also SODA'99.

[11] H. J. Karloff and P. Raghavan. Randomized algorithms and pseudorandom numbers. *JACM*, 40(3):454–476, 1993.

[12] D. E. Knuth. Notes on open addressing. Unpublished memorandum, 1963.

[13] M. Mitzenmacher and S. P. Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In SODA'08, pp. 746–755.

[14] R. Motwani and P. Raghavan. *Randomized algorithms*. Cambridge University Press, 1995.

[15] A. Pagh, R. Pagh, and M. Ružić. Linear probing with constant independence. *SIAM J. Comp.*, 39(3):1107–1120, 2009. See also STOC'07.

[16] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Alg.*, 51(2):122–144, 2004. See also ESA'01.

[17] M. Pătraşcu and M. Thorup. On the k-independence required by linear probing and minwise independence. In ICALP'10, pages 715–726.

[18] J. P. Schmidt, A. Siegel, and A. Srinivasan. Chernoff-Hoeffding bounds for applications with limited independence. *SIAM J. Discrete Math.*, 8(2):223–250, 1995. See also SODA'93.

[19] A. Siegel. On universal classes of extremely random constant-time hash functions. *SIAM J. Comp.*, 33(3):505–543, 2004. See also FOCS'89.

[20] M. Thorup. Even strongly universal hashing is pretty fast. In SODA'00, pages 496–497.

[21] M. Thorup. String hashing for linear probing. In SODA'09, pages 655–664.

[22] M. Thorup and Y. Zhang. Tabulation based 4-universal hashing with applications to second moment estimation. In SODA'04, pages 615–624.

[23] M. Thorup and Y. Zhang. Tabulation based 5-universal hashing and linear probing. In ALENEX'09.

[24] M. N. Wegman and L. Carter. New classes and applications of hash functions. *J. Comp. Sys. Sci.*, 22(3):265–279, 1981. See also FOCS'79.