2013 IEEE 54th Annual Symposium on Foundations of Computer Science

Simple Tabulation, Fast Expanders, Double Tabulation, and High Independence

Mikkel Thorup
Department of Computer Science
University of Copenhagen, Denmark
Email: mikkel2thorup@gmail.com

***Abstract*—Simple tabulation dates back to Zobrist in 1970 who used it for game playing programs. Keys are viewed as consisting of c characters from some alphabet Φ. We initialize c tables $h_0, \ldots, h_{c-1}$ mapping characters to random hash values. A key $x = (x_0, \ldots, x_{c-1})$ is hashed to $h_0[x_0] \oplus \cdots \oplus h_{c-1}[x_{c-1}]$, where $\oplus$ denotes bit-wise exclusive-or. The scheme is extremely fast when the character hash tables h_i are in cache. Simple tabulation hashing is not even 4-independent, but we show here that if we apply it twice, then we do get high independence. First we hash to some intermediate keys that are 6 times longer than the original keys, and then we hash the intermediate keys to the final hash values.**

The intermediate keys have $d = 6c$ characters from Φ. We can then view the hash function as a highly unbalanced bipartite graph with keys on one side, each with edges to d output characters on the other side. We show that this graph has nice expansion properties, and from that it follows that if we perform another level of simple tabulation on the intermediate keys, then the composition is a highly independent hash function. More precisely, the independence we get is $|\Phi|^{\Omega(1/c)}$. In our O-notation, we view both $|\Phi|$ and c is going to infinity, but with c much smaller than $|\Phi|$.

Our space is $O(c|\Phi|)$ and the hash function is evaluated in $O(c)$ time. Siegel [FOCS'89, SICOMP'04] has proved that with this space, if the hash function is evaluated in $o(c)$ time, then the independence can only be $o(c)$, so our evaluation time is best possible for $\Omega(c)$ independence—our independence is much higher if $c = |\Phi|^{o(1/c)}$.

Siegel used $O(c)^c$ evaluation time to get the same independence with similar space. Siegel's main focus was $c = O(1)$, but we are exponentially faster when $c = \omega(1)$.

Applying our scheme recursively, we can increase our independence to $|\Phi|^{\Omega(1)}$ with $o(c^{\log c})$ evaluation time. Compared with Siegel's scheme this is both faster and higher independence.

Siegel states about his scheme that it is "far too slow for any practical application". Our scheme is trivial to implement, and it does provide realistic implementations of 100-independent hashing for, say, 32-bit and 64-bit keys.

***Keywords*-hashing; independence; expanders;**

I. Introduction

Independent hashing: The concept of k-independent hashing was introduced by Wegman and Carter [1] at FOCS'79 and has been the cornerstone of our understanding of hash functions ever since. The hash functions map keys from some universe U to some range R of hash values. Formally, a family $\mathcal{H} = \{h \mid U \to R\}$ of hash functions is k-independent if (1) for any distinct keys $x_1, \ldots, x_k \in U$, the hash values $h(x_1), \ldots, h(x_k)$ are independent random variables when h is picked at random from $\mathcal{H}$; and (2) for any fixed x, $h(x)$ is uniformly distributed in R. By a k-independent hash function we refer to a function chosen at random from such a family. Often the family is only given implicitly as all possible choices some random parameters defining the function.

As the concept of independence is fundamental to probabilistic analysis, k-independent hash functions are both natural and powerful in algorithm analysis. They allow us to replace the heuristic assumption of truly random hash functions with real (implementable) hash functions that are still "independent enough" to yield provable performance guarantees. We are then left with the natural goal of understanding the independence required by algorithms. When first we have proved that k-independence suffices for a hashing-based randomized algorithm, then we are free to use *any* k-independent hash function.

Let U and R be the sets $U = [u] = \{0, \ldots, u-1\}$ and $R = [r] = \{0, \ldots, r-1\}$. The canonical construction of a k-independent family is a polynomial of degree $k-1$ over a prime field $\mathbb{Z}_p$ where $p \geq u$. The random parameters are the coefficients $a_0, \ldots, a_{k-1} \in \mathbb{Z}_p$. The hash function is then

$$h(x) = \Big(\big(a_{k-1}x^{k-1} + \cdots + a_1 x + a_0 \big) \bmod p \Big) \bmod r \quad (1)$$

For $p \gg r$, the hash function is statistically close to k-independent. One thing that makes polynomial hashing over $\mathbb{Z}_p$ slow for $\wp > 2^{32}$ is that each multiplication over $\mathbb{Z}_p$ translates into multiple 64-bit multiplications that due to discarded overflow can only do exact multiplication of 32-bit numbers. The "mod p" operation is very expensive in general, but [2] suggests using a Mersenne prime p such as $2^{61} - 1$ or $2^{89} - 1$, and then 'mod p' can be made very fast.

Word RAM model: We are assuming the word RAM model where the operations on words are those available in a standard programming language such as C [3]. A word defines the maximal unit we can operate on in constant time. For simplicity, we assume that each key or hash value fits in a single word. This implies that the time it takes to evaluate the degree $k-1$ polynomial from (1) is $O(k)$. The Random Access Memory (RAM) implies that we can create tables, accessing entries in constant time based on indices computed from key values. Such random access memory has

0272-5428/13 $26.00 © 2013 IEEE
DOI 10.1109/FOCS.2013.18

<table>
<tr><th colspan="4">Non-Constructive Cell-Probe Model</th></tr>
<tr><th>Space</th><th>Probes</th><th>Independence</th><th>Reference</th></tr>
<tr><td>$u^{1/c}$</td><td>-</td><td>$\leq u^{1/c}$</td><td>Trivial</td></tr>
<tr><td>$u^{1/c}$</td><td>$t < c$</td><td>$\leq t$</td><td>[5]</td></tr>
<tr><td>$u^{1/c}$</td><td>$O(c)$</td><td>$u^{\Omega(1/c)}$</td><td>[5]</td></tr>
<tr><th colspan="4">C-programmable Word RAM model</th></tr>
<tr><th>Space</th><th>Time</th><th>Independence</th><th>Reference</th></tr>
<tr><td>k</td><td>$O(k)$</td><td>k</td><td>Polynomial</td></tr>
<tr><td>u</td><td>1</td><td>u</td><td>Complete table</td></tr>
<tr><td>$u^{1/c}$</td><td>$O(c)^c$</td><td>$u^{\Omega(1/c^2)}$</td><td>[5]</td></tr>
<tr><td>$u^{1/c}$</td><td>$O(ck)$</td><td>k</td><td>[6], [7], [8]</td></tr>
<tr><td>$u^{1/c}$</td><td>$O(c)$</td><td>$u^{\Omega(1/c^2)}$</td><td>This paper</td></tr>
<tr><td>$u^{1/c}$</td><td>$O(c^{\lg c})$</td><td>$u^{\Omega(1/c)}$</td><td>This paper</td></tr>
</table>

Table I
HASHING WITH PREPROCESSED REPRESENTATION.

been assumed for hash tables since Dumey introduced them in 1956 [4].

Time-space trade-offs: To get faster hash functions, we implement them in two phases. First we have a preprocessing phase where we based on a random seed construct a representation of the hash function. We do not worry too much about the resources used constructing the representation, but we do worry about the space of the representation, measured in number of words. Next we have a typically deterministic query phase where we for a given key compute the hash value using the representation. Table I presents an overview of the results in this model that will be discussed here in the introduction. In our O-notation, we view both u and c as going to infinity, but c is much smaller than u.

In the case of polynomial hashing, the preprocessing just stores the coefficients $a_0, \ldots. a_k$ in k words. Unfortunately, to find the hash value of a key x, we have to access all k words in $O(k)$ time. Another extreme would be to store the hash values of all possible keys in one big table of size u. Then we can find the hash of a key in constant time by a single lookup.

There has been interesting work on representing a high degree polynomial for fast evaluation [9, Theorem 5.1]. For a degree $k-1$ polynomial over $\mathbb{Z}_p$, the evaluation time is $(\log k)^{O(1)}(\log p)^{1+o(1)}$. This avoids the linear dependence on k, but the factor $\log p \geq \log u$ is prohibitive for our purposes.

Simple tabulation hashing: In simple tabulation hashing, we generally view both keys and hash values as bit strings, so u and r are powers of two. Moreover, we view a key x as a vector of c characters $x_0, \ldots, x_{c-1}$ from the alphabet $\Phi = [u^{1/c}]$. *Simple tabulation* is defined in terms of c *character tables* $h_0, \ldots, h_{c-1} : \Phi \to R$. This induces a function $h : U \to R$ defined by

$$h(x) = \bigoplus_{i \in [c]} h_i(x_i) = h_0(x_0) \oplus \cdots \oplus h_{c-1}(x_{c-1}). \qquad (2)$$

Here $\oplus$ denotes bit-wise exclusive-or (xor). We call this *simple tabulation hashing* when the character tables are filled with random hash values from R. This is a well-known scheme dating back at least to Zobrist in 1970 [10] who used it for game playing programs. Simple tabulation hashing is only 3-independent even if all character tables are fully random.

In simple tabulation, the preprocessing phase fills the c character tables h_i. These may all be stored consecutively as a single 2D array $[c] \times \Phi \to R$ using $cu^{1/c}$ space. If we already have some randomly filled memory, then a simple tabulation hash function is defined in constant time, simply by placing the offset of the array in the random memory.

In the query phase, we find each $h_i(x_i)$ by a single lookup. We do only c lookups, and we only have a constant number of word operations per lookup, so each hash value is computed in $O(c)$ time. If Φ consists of 8-bit or 16-bit characters, then the character tables fit in fast cache. For 32-bit or 64-bit keys, simple tabulation is about 3 times faster than the 3-independent hashing obtained as in (1) by a degree 2-polynomial tuned for a Mersenne prime (see, e.g., experiments in [11], [8]). Also note that with simple tabulation, the cost of expanding the range R to longer bit-strings is minor in that we still only have to do c lookups. The added cost is only from storing and looking up longer bit-strings that have to be xor'd.

In [11] it was proved for many concrete applications that simple tabulation has far more power than its 3-independence suggests. However, to use simple tabulation in an application such as linear probing, one has to make a careful analysis to show that the dependence is not harmful to the application. This is not as attractive as the generic independence paradigm where any k-independent hash function can be used in any application for which k-independence suffices. According to Google Scholar, Siegel's [5] highly independent hashing has more than 150 citations (including those to the original conference version), but as he states, it is "far too slow for any practical application".

A. Results

In this paper we show that to get the same high independence as Siegel [5] efficiently, we just have to apply simple tabulation twice, and we get even higher independence with more applications. Our key is to show that simple tabulation, applied once, is likely to have some strong expander properties.

Unbalanced expanders by simple tabulation: To describe the result, we need some simple notation and terminology. Suppose $y \in \Psi^d$ is a vector of d characters from Ψ. We let y_j denote character j in y, so $y = (y_0, \ldots, y_{d-1})$. By a *position character* we mean a pair $(j, a) \in [d] \times \Psi$ consisting of a position and a character. The vector y is identified with the corresponding set $\{(j, y_j) | j \in [d]\}$ of position characters.

Consider a function $f : U \to \Psi^d$. It defines an unbalanced bipartite graph with the key set U on the left-hand side and the output position characters from $V = [d] \times \Psi$ on the right-hand side. A key $x \in U$ has d distinct neighbors; namely the d output position characters $(0, f(x)_0), \dots (d-1, f(x)_{d-1}) \in V$. Two keys x and y share a neighboring output position character if and only if $f(x)_j = f(y)_j$ for some j. We say a set $X \subseteq U$ has a *unique output position character* (j, a) if there is an $x \in X$ such that $f(x)_j = a$ and for all other $y \in X \setminus \{x\}$, $f(y)_j \neq a$. Our basic result is that if we consider random simple tabulation with 6 times more output than input characters, then every not too large set X has a unique output position character. This can be viewed as a weak expander property. As the number of output characters increases, we get the standard expansion property that X has $\Omega(d|X|)$ distinct output position characters (neighbors in the bipartite graph). The formal statement is as follows.

Theorem 1: Consider a simple tabulation function $h : \Phi^c \to \Psi^d$ where $d \geq 6c$ and where the character tables are fully random. Assume $c = |\Phi|^{o(1)}$ and $(c+d)^c = |\Psi|^{o(1)}$. Let $k = |\Psi|^{1/(5c)}$. With probability $1 - o(|\Phi|^2/|\Psi|^{d/(2c)})$,

(a) every key set $X \subseteq \Phi^c$ of size $|X| \leq k$ has at least one unique output position character.

Moreover, for any $\varepsilon \in (0, 1)$, with probability $1 - o(|\Phi|^2/|\Psi|^{\varepsilon d/(2c)})$,

(b) every key set $X \subseteq \Phi^c$ of size $|X| \leq k$ has more than $(1-\varepsilon)d|X|$ distinct output position characters.

The requirement that the character tables are fully random can be relaxed in the sense that we for (a) and (b) can use any $k \leq |\Psi|^{1/(5c)}$ such that all character tables are k-independent, and independent of each other.

Above we think of c and d as slow growing. Our construction is interesting also when d is constant, but then we cannot measure its effect with O-notation.

The assumptions $c = |\Phi|^{o(1)}$ and $(c+d)^c = |\Psi|^{o(1)}$ are not essential, but serve to give simple probability bounds for (a) and (b). As we shall see in Theorem 5, we can derive much better bounds for concrete cases.

Our work is orthogonal to the deep work on explicit expanders; for Theorem 1 relies on random values for the character tables. Also, when it comes to highly unbalanced expanders like in Theorem 1, the best explicit constructions [12] have logarithmic degrees. It would be very interesting if we could fill the character tables of Theorem 1 explicitly during preprocessing with an efficient deterministic algorithm. When done, we would enjoy the high speed of simple tabulation.

High independence by double tabulation: In this paper, we are mostly interested in the unique output position characters from (a). We say that a function $f : U \to \Phi^d$ is *k-unique,* or has *uniqueness k,* if every subset $X \subseteq U$ of size at most k has a unique output position character. Translating Lemma 2.6 in [5], we get

Lemma 2 (Siegel): Let $f : U \to \Psi^d$ be a k-unique function. Consider a random simple tabulation function $r : \Psi^d \to R$ where the character tables $r_j : \Psi \to R$, $j \in [d]$, are independent of each other, and where each r_j is k-independent. Then $r \circ f : U \to R$ is k-independent.

Suppose we have a concrete simple tabulation function $h : \Phi^c \to \Psi^d$ that satisfies (a) from Theorem 1. Then h is k-unique. We can now compose h with a random simple tabulation function $r : \Psi^d \to R$ from Lemma 2. The resulting function $r \circ h$ is a k-independent function from $U = \Phi^c$ to R. We call this composition *double tabulation.*

Note that if we want a new independent k-independent hash function, we can still use the same k-unique h as a universal constant. We only need to generate a new independent simple tabulation hash function $r' : \Psi^d \to R$, and use $r' \circ h : U \to R$ as the new k-independent hash function.

Unfortunately, we do not know of any efficient way of testing if the simple tabulation function h from Theorem 1 is k-unique. However, a random h is k-unique with some good probability. To emphasize that we only need k-uniqueness for a single universal $h : \Phi^c \to \Psi^d$, we say that it happens with *universal probability.*

Corollary 3: Let $u = |U|$ and assume $c^{c^2} = u^{o(1)}$. With universal probability $1 - o(1/u^{1/c})$, using space $o(u^{1/c})$, we get $u^{\Omega(1/c^2)}$-independent hashing from U to R in $O(c)$ time.

Proof: We use the above double tabulation. For simplicity, we assume that u is a power of a power of two. For the first simple tabulation function h from Theorem 1, we use $c' = 2^{\lceil \lg_2 c \rceil + 1}$ input characters from Φ and $d = 8c'$ output characters, also from Φ. The uniqueness we get is $k = |\Phi|^{1/(5c')} = |\Phi|^{\Omega(1/c)}$, and the error probability is $o((1/u^{1/c'})^{2-d/(2c')}) = o(1/u^{1/c})$. The second simple tabulation r from Lemma 2 has d input characters from Φ, so the total number of tables is $c'+d = O(c)$. This is also the number of lookups, and for each lookup, we do a constant number of operations on a constant number of words. The space is thus $O(cu^{1/c'}) = o(u^{1/c})$, and the evaluation time is $O(c)$. ■

Siegel [5] has proved that with space $u^{1/c}$ one needs evaluation time $\Omega(c)$ to get independence above c. The time bound in Corollary 3 is thus optimal for any higher independence. We note that the restriction $c^{c^2} = u^{o(1)}$ is equivalent to saying that the independence $u^{\Omega(1/c^2)}$ is more than polynomial in c.

Higher independence by recursive tabulation: With representation space $u^{1/c}$, the highest independence we can hope for is $u^{1/c}$. In Corollary 3 we only get independence $u^{\Omega(1/c^2)}$. We will show that we can get independence $u^{\Omega(1/c)}$ using recursive tabulation. This is where it is important that Theorem 1 allows different alphabets for input and output characters. The basic idea is to use output characters from $\Psi = [u^{1/2}]$, and recurse on them to prove:

Theorem 4: Let $u = |U|$ and assume $c^{c^2} = u^{o(1)}$. With universal probability $1 - o(1/u^{1/c})$, using space $o(u^{1/c})$, we can get $u^{\Omega(1/c)}$-independent hashing from U to R in $o(c^{\lg_2 c})$ time.

If we unravel the recursion (to be presented in Section IV), for some $D = o(c^{\lg c})$ and $k = u^{\Omega(1/c)}$, we get a function $f : U \to [u^{1/(2c)}]^D$ that is not k-unique, yet which yields k-independence if composed with a random simple tabulation function $r : [u^{1/(2c)}]^D \to R$. If follows from [8, Proposition 2] or [7, Theorem 3] that f has the property that some output position character appears an odd number of times.

Concrete parameters: Note that when dealing with n keys, it is fairly standard to use *universe reduction*, applying universal hashing into a domain of size $n^{2+\varepsilon}$, $\varepsilon = \Omega(1)$, hoping for no collisions. Starting from this domain, dividing into $c = 3$ characters brings us down to space $O(n^{2/3+\varepsilon})$ which may be very acceptable. Thus it is often reasonable to think of c as small.

Below we consider some concrete parameter choices yielding 100-independent hashing. This would have been prohibitively slow with the polynomial hashing from (1). With reference to Lemma 2, the challenge is to find a 100-unique function. The probabilities are based on careful calculations yielding much better bounds than those derived from the simple formula in Theorem 1 (a). We do not make any assumptions like $c = |\Phi|^{o(1)}$ and $(c + d)^c = |\Psi|^{o(1)}$.

Theorem 5: We consider a simple tabulation hash function $h : \Phi^c \to \Psi^d$. Assuming that the character tables h_i of h are fully random, or at least 100-independent, and independent of each other,

1) For 32-bit keys with $\Phi = \Psi = [2^{16}]$, $c = 2$, and $d = 20$, the probability that h is not 100-unique is bounded by 1.5×10^{-42}.
2) For 64-bit keys with $\Phi = \Psi = [2^{22}]$, $c = 3$, and $d = 24$, the probability that h is not 100-unique is bounded by 1.4×10^{-49}.
3) For 64-bit keys with $\Phi = [2^{16}]$, $\Psi = [2^{32}]$, $c = 4$, and $d = 14$, the probability that h is not 100-unique is bounded by 9.0×10^{-36}. The idea is to use triple tabulation, applying Case 1 to each of the 32-bit output characters.

Recall that we only need a single universal 100-unique function h for each set of parameters. Trusting some randomly filled memory to represent such a 100-unique function as in Theorem 5 is extremely safe.

B. Siegel's highly independent hashing

Siegel's study on hashing [5] considered the fundamental trade-offs between independence, representation space, and the time it takes to compute the hash of a key.

Lower bound: Siegel's lower bound [5, Theorem 3.1] is in Yao's [13] powerful cell probe model. To get clean bounds, he assumes that the domain of a word or cell is no bigger that of a single hash value. Trivially this means that we need at least k cells to get independence k.

The representation is an arbitrary function of the random seed. If the representation has s cells, an equivalent formulation is that the contents of the s cells follow an arbitrary distribution.

The querier is given the key. To compute the hash value, he can probe the cells of the representation. He is only charged for these cell probes. His next move is an arbitrary function of the key and the cells he has read so far: he can either pick a cell based on this information, or output the hash value.

Siegel shows that if the representation uses $u^{1/c}$ cells, and the query phase makes $t < c$ probes, then the hash function computed can be at most t-independent. His argument is very robust, e.g., with no change to the asymptotics, he can allow some quite substantial bias in the independence, look at average query time, etc.

Upper bounds: Siegel's framework for upper bounds is similar to what we already described, but simpler and in that he is not "position sensitive": Given a function $f : U \to \Psi^d$, he considers the unbalanced bipartite graph with the keys from U on the left-hand side, and output characters from Ψ on the right-hand side (on our right-hand side, we had the position output characters from $V = [d] \times \Psi$). A key $x \in U$ has the d neighbors $f(x)_0, \ldots, f(x)_{d-1}$ that may not all be distinct. He says that f is k-peelable (corresponding to k-unique) if every key set X of size at most k has a unique output character. Here $x, y \in X$ share an output character if $f(x)_i = f(y)_j$ even if $i \neq j$. He uses a *single* character table $r_0 : \Psi \to R$, and defines $r : \Psi^d \to R$ by

$$r(x) = \bigoplus_{j \in [d]} r_0(x_j). \tag{3}$$

Siegel proves [5, Lemma 2.6] that if f is k-peelable, and $r_0 : \Psi \to R$ is random, then $r \circ f$ is k-independent. Note that the space of r is independent of d since r_0 uses only a single character table taking space $|\Psi|$. It does, however, take d lookups to evaluate (3). The problem is to find the k-peelable function f.

Let $u = |U|$ and $u^{1/c} = |\Psi|$. For the existence of a k-peelable function, Siegel [5, Lemma 2.9] argues that a fully random $f : U \to \Psi^d$ is likely to be a good expander from U to Ψ if $d \geq 6c$. More precisely, with probability $1 - \widetilde{O}(1/u)$, for $k = u^{1/(2c)}$, he gets that every set X of size $|X| \leq k$ has more than $d|X|/2$ neighbors. He also notes [5, Lemma 2.8] that if X has more than $d|X|/2$ distinct neighbors, then some of them have to be unique, so f is also k-peelable.

Representing a fully random f would take space u, but existence is all that is needed for upper bounds in the abstract cell-probe model. We can simply use the unique lexicographically smallest k-peelable $F = \min\{f : U \to \Psi^d \mid f \text{ is } k\text{-peelable}\}$. The querier can identify F on-the-fly without any probes. The representation only needs to include the random r_0 which takes $u^{1/c}$ space. The hash $r(F(x))$ of a key x is computed with $d = O(c)$ probes to r_0, and the independence is $k = u^{1/(2c)}$. The number of probes is within a constant factor of the lower bound which says that with $u^{1/c}$, we need at least c probes for any independence above c.

To get an implementation on the word RAM [5, §2.2], Siegel makes a graph product based on a small random graph that can be stored in space $u^{1/c}$. Assuming that the random graph has sufficient expander properties, the product induces a $u^{\Omega(1/c^2)}$-peelable function $f : U \rightarrow \Psi^{O(c)^c}$. This leads to a $u^{\Omega(1/c^2)}$-independent hash function represented in $u^{1/c}$ space. Hash values are computed in $O(c)^c$ time. It should be noted that Siegel's focus was the case where $c = O(1)$, and then he does get $u^{\Omega(1)}$-independence in $O(1)$ time, but here we consider $c = \omega(1)$ in order to qualify the dependence on c.

The RAM implementation of Siegel should be compared with our bounds from Theorem 4: $u^{\Omega(1/c)}$-independent hashing using $o(u^{1/c})$ space, computing hash values in $o(c^{\lg c})$ time. Our independence is significantly higher—essentially as high as in his existential cell-probe construction—and we are almost exponentially faster. We should also compare with Corollary 3: $u^{\Omega(1/c^2)}$-independent hashing using $o(u^{1/c})$ space, computing hash values in $o(c)$ time. This is the same independence as an Siegel's RAM implementation, but with the optimal speed of his existential cell probe construction.

On the technical side, recall that Siegel's k-peelability is not position sensitive. This is only a minor technical issue, but being sensitive to positions does yield some extra structure. In particular, we do not expect the simple tabulation function from Theorem 1 to be k-peelable without the positions.

C. Other related work

Siegel states [5, Abstract] about his scheme that it is "far too slow for any practical application". This and the $O(c)^c$ evaluation time has lead researchers to seek simpler and faster schemes. Several works [6], [7], [8] have been focused on the case of smaller independence k. These works have all been position sensitive like ours. Fix $\Psi = [u^{1/c}]$. We are looking at functions $f : U \rightarrow \Psi^d$, to be composed with a simple tabulation hash function $r : \Psi^d \rightarrow R$. The evaluation time is $O(d)$, so we want d to be small.

Dietzfelbinger and Woelfel [6, §5] pick d 2-independent hash functions $f_0, \ldots, f_{d-1} : U \rightarrow \Psi$. This yields a function $f : U \rightarrow \Psi^d$ defined by $f(x) = (f_0(x), \ldots, f_{d-1}(x))$. Composing f with a random simple tabulation function $h : \Psi^d \rightarrow R$, they show that the result is close to k-independent if $d \gg kc$.

Thorup and Zhang [8] found an explicit deterministic construction of a k-unique f which also has better constants than the scheme from [6]. By Lemma 2, the resulting hash function is exactly k-independent. Simple tabulation is by itself 3-independent, but [8] is motivated by applications needing 4 and 5-independence. For $k = 5$ and $\Psi = [u^{1/c} + 1]$, [8] gets down to $d = 2c - 1$. For general k, using $\Psi = [u^{1/c}]$, [8] gets $d = (k-1)(c-1)+1$.

Klassen and Woelfel [7] focus mostly on $c = 2$, where for arbitrary k they get $d = (k+1)/2$. For general c, their bound is $d = \lceil 2\frac{c-1}{2c-1}(k-1)\rceil(c-1)+1$.

We note that the twisted tabulation in [14] has a similar flavor to the above schemes, but it does not yield independence above 3. The main target of [14] is to get strong Chernoff style bounds.

The above works [6], [7], [8] thus need $d = \Omega(kc)$ for independence k. This contrasts our Theorem 1 which gets $d = O(c)$ with independence $u^{\Omega(1/c)}$. Apart from the case $c = 2$, $k = 5$ from [8], our new scheme is probably also the easiest to implement, as we are just applying simple tabulation twice with different parameters.

There are also constructions aimed at providing good randomness for a single unknown set S of size n [15], [16]. In particular, Pagh and Pagh [16] have a two-part randomized construction of a constant time hash function h that uses $O(n)$ space so that for any given set S of size n, if Part 1 does not fail on S, then Part 2 makes h fully random on S. We have the same two-parts pattern in our double tabulation where Part 1 generates a random simple tabulation function that we hope to be k-unique on the whole universe, and Part 2 composes this function with another random simple tabulation function r. If Part 1 succeeds, the result is k-independent. A principal difference is that any concrete fixing of Part 1 from [16] fails for many sets S, so the success probability of Part 1 in [16] is not universal; otherwise this would have been an n-independent hash function. From a more practical perspective, often we only need, say, $\log n$-independence, and then double tabulation with universe reduction and small character tables in cache is much simpler and faster than [16]. In fact, [16] uses Siegel's [5] highly independent hash functions as a subroutine, and now we can instead use our double tabulation. Double tabulation fits very nicely with the other use of random tables in [16], making the whole construction of full randomness for a given set S quite simple. It should be noted that [6] have found a way of bypassing the need of [5] in [16]. However, our double tabulation is even simpler, and it replaces the use of [5] in all applications.

II. The basic analysis

The next two sections are devoted to the proof of Theorem 1. For now, we assume that all character tables are fully random, leaving the relaxation to k-independent character tables till the very end.

By an *input position character* we mean a value from $[c] \times \Phi$. Notationally, we can then view a key $x = (x_0, \ldots, x_{c-1})$ as the set of input position characters: $\{(0, x_0), \ldots, (c-1, x_{c-1})\}$. We can now specify h as a single table from input position characters $[c] \times \Phi$ to vectors $h(\alpha) \in \Psi^d$, that is, if $(a, i) = \alpha \in [c] \times \Phi$, then $h(\alpha) = h_i[a]$. This view induces a function h on arbitrary sets x of input position characters:

$$h(x) = \bigoplus_{\alpha \in x} h(\alpha). \tag{4}$$

Note that when x is the set corresponding to a key, (4) agrees with (2). We define an *output index* as a pair $(\alpha, j) \in ([c] \times \Phi) \times [j]$ indexing the individual output character $h(\alpha)_j$.

We want to show that the if we assign $h : [c] \times \Phi \to \Psi^d$ at random, then there is only a small probability that there exists a set $X \subseteq \Phi^c$, $|X| \le k \le |\Psi|^{1/(5c)}$, violating (a) or (b) in Theorem 1.

Efficient coding: To specify h, we have to specify a vector of d output characters from Ψ for each of the $c|\Phi|$ input position characters. Based on a violating set X, we will construct an efficient coding of some of the output characters. The number of such efficient codings will be much smaller than the number of ways we can assign the output characters coded. Efficient codings are therefore rarely possible, hence so are the violating sets.

Our coding will not describe the set X, and it is important that decoding can be done without any knowledge of X, except that $|X| \le k$. The coding starts by specifying a list L with some of the input position characters from the keys in X. We will now go through the input position characters $\alpha \in L$ in the order that they appear in L. For each α, we will specify the d output characters $h(\alpha)_j$, $j \in [d]$. Some of these output characters will be derivable from previously specified output characters, leading to a more efficient encoding:

Definition 6: We say the output character $h(\alpha)_j$ is *derivable* if there exist keys $x, y \in X$ such that:

- The symmetric difference $x \triangle y = \{(i, x_i), (i, y_i) \mid i \in [c], x_i \ne y_i\}$ of x and y is contained in L.
- α is last in L among the input position characters in $x \triangle y$.
- $h(x)_j = h(y)_j$, or equivalently, $h(x \triangle y)_j = 0$.

In our representation, we do not need to know the keys x and y. We only need to know the symmetric difference $A = x \triangle y \subseteq L$. We call (A, j) an *equation* as it represents the information that $h(A)_j = 0$. The output index specified by the equation (A, j) is the pair (α, j) where α is the last input position character from A in the list L. The equation derives the output character

$$h_j(\alpha) = \bigoplus \{h(\beta)_j \mid \beta \in A \setminus \{\alpha\}\}.$$

The input position characters β all precede α in L, so the output characters $h(\beta)_j$ have all been specified. We do not want more than one equation specifying the same output index.

When the list L of length ℓ is given, the set A can be picked in less than ℓ^{2c} ways, so the number of possible derivations is less than $\ell^{2c} d$. If $\ell^{2c} d \ll |\Psi|$, then this is a win. Indeed this is the case because $\ell \le kc \le c|\Psi|^{1/(5c)}$ and $(c + d)^c = |\Psi|^{o(1)}$. However, we will have to make a lot of derivations to make up for the fact that we first have to specify the ℓ input position characters in L. In Section III we will show that a violating set X implies the existence of a list L with many derivable output characters, e.g., a violation of (a) in Theorem 1 will yield $|L|\, d/(2c)$ derivable output characters.

Below, for a given parameter q, we study the probability $P^{\,q}$ of finding a list L of length at most kc with at least $q|L|$ derivable output characters. Below we will prove that

$$P^{\,q} = o(|\Phi|^2 / |\Psi|^q). \tag{5}$$

There may be much more than $q|L|$ output characters derivable from L. However, in our encoding, we also only store equations for exactly $\lceil q|L| \rceil$ of them.

Coding and decoding: To summarize, the exact components of our code are:

1) A list L of ℓ input position characters.
2) A set of M of $\lceil q\ell \rceil$ equations (A, j) where $A \subseteq L$ and $j \in [d]$. Let I be the set of output indices specified in these equations. The output indices should all be distinct, so $|I| = \lceil q\ell \rceil$.
3) A reduced table H that for each $(\alpha, j) \in (A \times [d]) \setminus I$, specifies the output character $h(\alpha)_j \in \Psi$.

Above, each component presumes that the previous components are known, so L is known when we specify M, and L, M, and hence I is known when we specify H. Together, this specifies L and $h|L$. The decoding of $h|L$ goes as follows. From L and M we compute the set I of output indices (α, j) specified by M. For all other output indices $(\alpha, j) \in L \times [d]$, we find the output character $h(\alpha)_j$ in H. To get the remaining output characters we run through the input position characters $\alpha \in L$ in the order they appear in L. For each α and $j \in [d]$, we check if $(\alpha, j) \in I$. If so, we take the corresponding equation $(A, j) \in M$, and set $h(\alpha)_j = h_j(A \setminus \{\alpha\})$.

Bounding the probabilities: Let the above coding be fixed, and consider a random simple tabulation function h. The probability that our coding matches $h(\alpha)_j$ for all output indices $(\alpha, j) \in L \times [d]$ is exactly $1/|\Psi|^{\ell d}$. A union bound over all possible codes will imply that none of them are likely to match a random h.

Let us first assume that ℓ is fixed, that is, we restrict our attention to codes where $|L| = \ell$. The number of choices for L is bounded as $\text{choices}_\ell(L) < (c|\Phi|)^\ell$. Let $\text{choices}_\ell^q(M)$ be the number of choices for M given L. We already saw that the number of possible equations is bounded by $\ell^{2c} d$. The number of ways we can pick $\lceil q\ell \rceil$ of these is trivially bounded as

$$\text{choices}_\ell^q(M) < (\ell^{2c} d)^{\lceil q\ell \rceil}.$$

Finally, we need to pick H with an output character for each output index in $(L \times [d]) \setminus I$. There are $\ell d - \lceil q\ell \rceil$

output characters to pick, leaving us $|\Psi|^{\ell d-\lceil q\ell\rceil}$ choices for H. All in all we have $\text{choices}_\ell(L)\cdot\text{choices}^q_\ell(M)\cdot|\Psi|^{\ell d-\lceil q\ell\rceil}$ possible codes with the given ℓ. By the union bound, the probability that any of them match a random h is

$$\begin{aligned}P^q_\ell &= \frac{\text{choices}_\ell(L)\cdot\text{choices}^q_\ell(M)\cdot|\Psi|^{\ell d-\lceil q\ell\rceil}}{|\Psi|^{\ell d}}\\ &= \frac{\text{choices}_\ell(L)\cdot\text{choices}^q_\ell(M)}{|\Psi|^{\lceil q\ell\rceil}} \qquad (6)\\ &< (c|\Phi|)^\ell\left(\frac{\ell^{2c}d}{|\Psi|}\right)^{\lceil q\ell\rceil} \le (c|\Phi|)^\ell\left(\frac{\ell^{2c}d}{|\Psi|}\right)^{q\ell} \qquad (7)\end{aligned}$$

Strictly speaking, the last inequality assumes $\frac{\ell^{2c}d}{|\Psi|}\le 1$. However, if $\frac{\ell^{2c}d}{|\Psi|}>1$, the whole bound is above 1, and hence a trivial upper bound on P^q_ℓ. Since $\ell\le ck\le c|\Psi|^{1/(5c)}$, we have

$$(c|\Phi|)^\ell\left(\frac{\ell^{2c}d}{|\Psi|}\right)^{q\ell}\le\left(|\Phi|/|\Psi|^{3q/5}c(c^{2c}d)^q\right)^\ell.$$

We will now use our assumptions $c=|\Phi|^{o(1)}$ and $(c+d)^c=|\Psi|^{o(1)}$. We can also assume that $|\Phi|^2\le|\Psi|^q$, for otherwise (5) is a trivial probability bound above 1. Hence $c=|\Phi|^{o(1)}=|\Psi|^{o(q)}$, so $c(c^{2c}d)^q=|\Psi|^{o(q)}$. Hence

$$P^q_\ell\le\left(|\Phi|/|\Psi|^{(3/5-o(1))\,q}\right)^\ell.$$

However, we must have $\ell\ge 2$, for otherwise there cannot be any equations. Therefore

$$P^q\le\sum_{\ell=2}^{ck}P^q_\ell=o(|\Phi|^2/|\Psi|^q).$$

This completes the proof of (5).

Finally, as stated in Theorem 1, we need to argue that we do not need the character tables of h to be fully random. For $k\le|\Psi|^{1/(5c)}$, it should suffice that the character tables are k-independents and independent of each other. The simple point is that the violating set X is of size $|X|\le k$, so it involves at most k input characters for each position, and L can only use these input characters. With k-independent hashing, the assignment of output characters to the input characters in L is completely random, so we do not need any changes to the above analysis.

III. Many derivable output characters.

The goal of this section is to prove that if there is a set X violating (a) or (b) in Theorem 1, then we can construct a list L with many derivable characters.

Theorem 7: Consider a simple tabulation function $h:\Phi^c\to\Psi^d$.

($\bar{a}$) If there is a key set X with no unique output position characters, then there is a list L with some of the input position characters from X so that at least $\frac{d}{2c}|L|$ of the output characters from L are derivable.

($\bar{b}$) If for some $\varepsilon\le 1$ there is a key set X with at most $(1-\varepsilon)d|X|$ distinct output position characters, then there is a list L with some of the input position characters from X so that at least $\frac{\varepsilon d}{2c}|L|$ of the output characters from L are derivable.

Proof that Theorem 7 implies Theorem 1: Before proving Theorem 7, we note that it trivially implies Theorem 1, for if there is a set X violating Theorem 1 (a), then X satisfies Theorem 7 ($\bar{a}$), so there is a list L with $\frac{d}{2c}|L|$ derivable characters. By (5) the probability of this event is $P^{d/(2c)}\le|\Phi|^2/|\Psi|^{d/(2c)}$. Likewise Theorem 1 (b) follows from Theorem 7 ($\bar{b}$). ■

Proof of Theorem 7: We assume that we have a set X satisfying the conditions of ($\bar{a}$) or ($\bar{b}$). For a uniform proof, if the condition of ($\bar{a}$) is true, we set $\varepsilon=1$, overruling a possibly smaller ε from ($\bar{b}$). Set $q=\frac{\varepsilon d}{2c}$. We will identify the list L so that at least $q|L|$ of the output characters from L are derivable.

Let $\alpha_1,\ldots,\alpha_{\ell^*}$ be the distinct input position characters from keys in X listed in order of decreasing frequency in X. Let n_i be the number of keys from X containing α_i. Then $n_1\ge n_2\ge\cdots\ge n_{\ell^*}$ and $\sum_{i=1}^{\ell^*}n_i=c|X|$.

Let $L_{\le\ell}$ be the prefix $\alpha_1,\ldots,\alpha_\ell$. The list L in the theorem will be $L_{\le\ell}$ for some $\ell\le\ell^*$. Let φ_ℓ be the number of new derivable output characters when α_ℓ is added to $L_{\le\ell-1}$ creating $L_{\le\ell}$. Then

$$\begin{aligned}\varphi_\ell=|\{j\in[d]\mid\exists x,y\in X:&\\ \alpha_\ell\in x\triangle y\subseteq L_{\le\ell},h(x)_j=h(y)_j\}|.&\end{aligned}$$

The list $L_{\le\ell}$ satisfies the theorem if $\sum_{i=1}^{\ell}\varphi_i\ge q\ell$. To prove that this is true for some $\ell\le\ell^*$, we study a related measure

$$\begin{aligned}\gamma_{\le\ell}=|\{(x,j)\in X\times[d]\mid\exists\, y\in X\setminus\{x\}:&\\ x\triangle y\subseteq L_{\le\ell},\ h(x)_j=h(y)_j\}|.&\end{aligned}$$

Then

$$\begin{aligned}\gamma_{\le\ell^*}=|\{(x,j)\in X\times[d]\mid\exists\, y\in X\setminus\{x\}:&\\ h(x)_j=h(y)_j\}|&\end{aligned}$$

counts with multiplicity the number of non-unique output characters from X.

Lemma 8: $\gamma_{\le\ell^*}\ge\varepsilon d|X|$.

Proof: Each key $x\in X$ has d output position characters, so with multiplicity, the total number of output position characters from X is $d|X|$. In case ($\bar{a}$) these are all non-unique and we have $\varepsilon=1$.

In case ($\bar{b}$) we have at most $(1-\varepsilon)d|X|$ distinct output characters from X. The number of unique output position characters must be smaller, so with multiplicity, the total number of non-unique output characters from X is bigger than $\varepsilon d|X|$. ■

The following key lemma relates the two measures:

Lemma 9: For $\ell=2,\ldots,\ell^*$,

$$\gamma_{\le\ell}-\gamma_{\le\ell-1}\le 2\varphi_\ell\,n_\ell. \qquad (8)$$

Proof: We want to count the pairs $(x, j) \in X \times [d]$ that are counted in $\gamma_{\leq \ell}$ but not in $\gamma_{\leq \ell-1}$. First we consider "α_ℓ-pairs" (x, j) where x contains α_ℓ and there is a "witnessing" key $y \in X$ not containing α_ℓ such that $x \triangle y \subseteq L_{\leq \ell}$ and $h(x)_j = h(y)_j$. We note that in this case (α_ℓ, j) is derivable, so j is counted in φ_ℓ. The number of α_ℓ-pairs (x, j) is thus bounded by $\varphi_\ell \, n_\ell$.

With the above x and y, we would also count the "witnessing" pair (y, j) if (y, j) is not already counted in $\gamma_{\leq \ell-1}$. Suppose we have another pair (z, j) witnessing (x, j). Thus $x \triangle y, x \triangle y \subseteq L_{\leq \ell}$ and $h(x)_j = h(y)_j = h(z)_j$. We want to show that $z \triangle y \subseteq L_{\leq \ell-1}$, hence that both (y, j) and (z, j) were already counted in $\gamma_{\leq \ell-1}$.

All input position characters in $y \triangle z$ come in pairs (i, y_i), (i, z_i), $y_i \neq z_i$. At least one of y_i and z_i is different from x_i. By symmetry, assume $y_i \neq x_i$. Then $(i, y_i), (i, x_i) \in y \triangle x \subseteq L_{\leq \ell}$. Therefore $(i, z_i) \in L_{\leq \ell}$ if $z_i = x_i$; but otherwise $z_i \neq x_i$ and $(i, z_i), (i, x_i) \in z \triangle x \subseteq L_{\leq \ell}$. In either case, we conclude that $(i, y_i), (i, z_i) \in L_{\leq \ell}$. But α_ℓ is in neither y nor z, so it follows that $(i, y_i), (i, z_i) \in L_{\leq \ell-1}$, hence that $y \triangle z \subseteq L_{\ell-1}$. We conclude that both (y, j) and (z, j) were counted in $\gamma_{\leq \ell-1}$, or conversely, that we for each α_ℓ-pair (x, i) have at most one witnessing pair (y, j) that is counted in $\gamma_{\leq \ell} - \gamma_{\leq \ell-1}$.

We conclude that the number of witnessing pairs is no bigger than the number of α_ℓ-pairs, hence that $\gamma_{\leq \ell} - \gamma_{\leq \ell-1}$ is at most $2\varphi_\ell \, n_\ell$. ■

By (8), for $\ell = 1, \ldots, \ell^*$,

$$\gamma_{\leq \ell} \leq 2 \sum_{i=1}^{\ell} \varphi_\ell \, n_\ell. \tag{9}$$

Recall that $L_{\leq \ell}$ satisfies the statement of the theorem if $\sum_{i=1}^{\ell} \varphi_i \geq q\ell$. Assume for a contradiction that there is a $q' < q$, such that for all $\ell = 1, \ldots, \ell^*$,

$$\sum_{i=1}^{\ell} \varphi_\ell \leq q'\ell. \tag{10}$$

The n_ℓ are decreasing, so the φ_ℓ values that satisfy (10) and maximize the sum in (9) are all equal to q'. Thus (9) and (10) implies that

$$\gamma_{\leq \ell} \leq 2 \sum_{i=1}^{\ell} \varphi_\ell \, n_\ell \leq 2 \sum_{i=1}^{\ell} q' n_i < 2q \sum_{i=1}^{\ell} n_i.$$

In particular, we get

$$\gamma_{\leq \ell^*} < 2q \sum_{i=1}^{\ell^*} n_i = 2q|X|c. \tag{11}$$

Since $q = \varepsilon d/(2c)$, this contradicts Lemma 8. Thus we conclude that there is an ℓ such that $L_{\leq \ell}$ satisfies the theorem. This completes the proof of Theorem 7, hence of Theorem 1. ■

IV. Higher independence with recursive tabulation

We will now use recursive tabulation to get the higher independence promised in Theorem 4:

> *Let $u = |U|$ and $c^{c^2} = u^{o(1)}$, With universal probability $1 - o(1/u^{1/c})$, using space $o(u^{1/c})$, we can get $u^{\Omega(1/c)}$-independent hashing from U to R in $o(c^{\lg_2 c})$ time.*

Proof of Theorem 4: For simplicity, we assume that u is a power of a power of two. Let $\ell = \lceil \lg_2 c \rceil + 1$, $c' = 2^\ell$, and $\Phi = [u^{1/c'}]$. The independence we aim for is $k = u^{1/(10c')} = u^{\Omega(1/c)}$.

Our construction is a recursion of depth ℓ. On level $i = 0, ..., \ell - 1$ of the recursion, the input key universe is $U_{(i)} = [u^{1/2^i}]$, and we want a k-independent hash functions $U_{(i)} \to R$. The set of input characters will always be $\Phi = [u^{1/c'}]$, so on level i, we have $c_{(i)} = c'/2^i$ input characters. We apply Theorem 1 with $d_{(i)} = 12c_{(i)}$ output characters from $\Psi_{(i)} = U_{(i+1)}$. With universal probability $1 - |\Phi|^2/\Psi_{(i)}^6$, Theorem 1 gives us a simple tabulation function $h_{(i)} : \Phi^{c_{(i)}} \to \Psi_{(i)}^{d_{(i)}}$ with uniqueness

$$|\Psi_{(i)}|^{1/(5c_{(i)})} = \left(u^{1/2^{i+1}}\right)^{1/(5(c'/2^i))} \geq u^{1/(10c')} = k,$$

as desired. To get k-independence from $U_{(i)}$ to R, as in Lemma 2, we compose $h_{(i)}$ with a simple tabulation function $r_{(i)} : \Psi_{(i)}^{d_{(i)}} \to R$ where the character tabulation functions $r_{(i),j} : \Psi_{(i)} \to R$ have to by k-independent and independent of each other. Here $\Psi_{(i)} = U_{(i+1)}$, and the $r_{(i),j}$ are constructed recursively. At the last recursive level, the output characters are from $\Psi_{(\ell-1)} = U_{(\ell)} = [u^{1/2^\ell}] = \Phi$. We will store an independent random character table for each of these output characters.

On each recursive level $i < \ell$, we can use the same universal k-unique simple tabulation function $h_{(i)} : \Phi^{c_{(i)}} \to \Psi_{(i)}^{d_{(i)}}$. However, on the bottom level, we need independent random character tables for all the output characters. The total number of output characters on the bottom level is

$$D = \prod_{i=0}^{\ell-1} d_{(i)} = \prod_{i=0}^{\ell-1} 12c'/2^i \leq O(\sqrt{c})^{\lg_2 c}.$$

Handling all of these, on the bottom level, we have a single large simple tabulation function $r : \Phi^D \to R$ where the D character tables are fully random tables supporting look-ups in constant time.

On recursive level $i < \ell$, the size of the intermediate domain $\Psi_{(i)}^{d_{(i)}}$ is $\left(u^{1/2^{i+1}}\right)^{12c'/2^i} = u^{6c'}$. The elements from this domain thus use $O(c)$ words. It follows that the space used by $h_{(i)}$ is $O(c_{(i)}|\Phi|c)$, and that its evaluation time from (2) is $O(c_{(i)}c) = O(c^2/2^i)$.

We only represent a single universal function $h_{(i)}$ on each level $i < \ell$, to the total space is clearly dominated by the

D tables on the bottom level. The total space is therefore $O(D|\Phi]) = O(\sqrt{c})^{\lg_2 c} u^{1/c'} = o(u^{1/c})$.

The evaluation time is actually dominated by the calls from level $\ell - 1$. Formally a recursive evaluation from the last recursive level $i \leq \ell$ takes time

$$\begin{aligned} &T(\ell) = O(1) \\ &T(i) = O(c^2/2^i) + d_{(i)} T(i+1) \text{ for } i = 0, \ldots, \ell - 1 \end{aligned}$$

Our evaluation time is thus $T(0) = O(cD) = o(c^{\lg c})$.

The probability that any of the universal $h_{(i)}$ is not k-unique is bounded by $\sum_{i=0}^{\ell-1} o(|\Phi|^2/\Psi_{(i)}^6) = o(|\Phi|^2/\Psi_{(\ell-1)}^6) = o(1/|\Phi|^4) = o(1/u^{1/c})$. ■

Let $k = u^{\Omega(1/c)}$ be the independence obtained in the above proof. Consider the $\ell - 1$ recursive levels. They compose into a function $f : U \rightarrow \Phi^D$, and we know that $r \circ f$ is k-independent. The interesting point here is that we do not expect f to be k-unique.

In [8, Proposition 2], or identically, [7, Theorem 3], is given an exact characterization of the functions $f : U \rightarrow \Phi^D$ that yield k-independence when composed with random simple tabulation hashing $h' : \Phi^D \rightarrow R$. The requirement is that every set X of size at most X has some output character appearing an odd number of times. Our f must satisfy this *k-odd* property.

V. Counting with care

Over the next two sections, we are now going tighten the analysis from Section II. In particular, this will allow us to derive the concrete values from Theorem 5 with no reference to asymptotics. As in Section II, we parameterize our analysis by the length ℓ of the list L of input characters. Later we will add up over relevant lengths $\ell \leq \ell^* = kc$. Using Theorem 7 we fix $q = \varepsilon d/(2c)$ with $\varepsilon = 1$ if we are only interested in uniqueness.

Removing order from the list: Our first improvement is to argue that we do not need store the order of the list L, i.e., we can just store L as a set. This immediately saves us a factor $\ell!$, that is, $\text{choices}_\ell(L) \leq (c|\Phi|)^\ell/\ell! < (ec|\Phi|/\ell)^\ell$.

With L only a set, an equation (A, j), $A \subseteq L$, $j \in [d]$ still has the same denotation that $h(A)_j = 0$. However, it was the ordering of L that determined the specified output index (α, j) with α being the last element from A in L. Changing the ordering of L thus changes the specified output indices. This may be OK as long as no two equations from M specify the same output index.

When L is only given as an unordered set and when we are further given a set M of equations, we implicitly assign L the lexicographically first order such that no two equations from M specify the same output index. This lexicographically first order replaces original order of L that we found in Section III. It redefines the set I of output indices specified by M, hence the set $(L \times [d]) \setminus I$ of output indices that have to be covered by the table H.

Equation count: We will now give a better bound on the number $\text{choices}_\ell(M)$ of possibilities for our set M of $\lceil q\ell \rceil$ equations. We know that our equations are of the form (A, j) where $A = x \triangle y \subseteq L$ for keys $x, y \in X$. More specifically, we have $A = x \triangle y = \{(x_i, i), (y_i, i) | i \in [c], x_i \neq y_i\}$. Let L_i be the set of input position characters from L in position i and let ℓ_i be their number. Let us assume for now that $\ell_i \geq 2$ for all $i \in [c]$. If this is not the case, we will later derive even better bounds with $c' < c$ active positions.

To describe A, for each $i \in [c]$, we pick either two elements from L_i or none. Since $\ell_i \geq 2$, this can be done in $\binom{\ell_i}{2} + 1 \leq \ell_i^2/2$ ways. The total number of ways we can pick A is thus

$$\text{choices}_{\ell,c}(A) \leq \prod_{i \in [c]} \ell_i^2/2 \leq ((\ell/c)^2/2)^c.$$

For an equation, we also need $j \in [d]$. We need $\lceil q\ell \rceil \geq \ell\varepsilon d/(2c)$ equations for M. We conclude that

$$\begin{aligned} \text{choices}_{\ell,c}(M) &\leq \binom{\text{choices}_{\ell,c}(A) \cdot d}{\lceil q\ell \rceil} \\ &\leq \left(\frac{e((\ell/c)^2/2)^c d}{\lceil q\ell \rceil} \right)^{\lceil q\ell \rceil} \\ &\leq \left(\frac{e((\ell/c)^2/2)^c d}{\ell\varepsilon d/(2c)} \right)^{\lceil q\ell \rceil} \\ &\leq \left(e(\ell/c)^{2c-1}/(\varepsilon 2^{c-1}) \right)^{\lceil q\ell \rceil}. \end{aligned}$$

Plugging our new bounds into (6), we get

$$\begin{aligned} P_{\ell,c} &\leq \frac{\text{choices}_\ell(L) \cdot \text{choices}_{\ell,c}(M)}{|\Psi|^{\lceil q\ell \rceil}} \\ &\leq (ec|\Phi|/\ell)^\ell \cdot \left(e(\ell/c)^{2c-1}/(\varepsilon 2^{c-1}|\Psi|) \right)^{\lceil q\ell \rceil} \\ &\leq \left((ec|\Phi|/\ell) \cdot \left(e(\ell/c)^{2c-1}/(\varepsilon 2^{c-1}|\Psi|) \right)^q \right)^\ell \\ &= \left((ec|\Phi|/\ell) \cdot \left(e(\ell/c)^{2c-1}/(\varepsilon 2^{c-1}|\Psi|) \right)^{\varepsilon d/(2c)} \right)^\ell. \end{aligned}$$

As with (7), we note that replacing the exponent $\lceil q\ell \rceil$ with $q\ell$ is valid whenever the probability bound is not bigger than 1. Above we assumed that L contained two characters in all c positions. In particular, this implies $\ell \geq 2c$. If this L contains less than two characters in some position, then that position has no effect. There are $\binom{c}{c'}$ ways that we can choose c' active positions, so the real bound we get for $\ell \leq \ell^*$ is:

$$\sum_{c' \leq c} \left(\binom{c}{c'} \sum_{\ell=2c'}^{\ell^*} P_{\ell,c'} \right)$$

The above bound may look messy, but it is easily evaluated by computer. For k-independence, we just need k-uniqueness, so $\varepsilon = 1$. As an example, with 32-bit keys, $c = 2$, $d = 20$, and $\ell^* = 32$, we get a total error probability less than 2.58×10^{-31}. This only allows us to rule out $k = \ell^*/c = 16$, but in the next section, we shall get up to $k = 100$ with an even better error probability.

VI. Coding keys

Trivially, we have $\ell \leq kc$ since kc is the total number of input position characters in the k at most keys. However, as ℓ approaches k, we can start considering a much more efficient encoding, for then, instead of encoding equations by the involved input position characters, we first encode the k keys, and then get a much cheaper encoding of the equations.

Our goal is to find efficient encodings of symmetric differences $x\triangle y \subseteq L$ where $x, y \in X$. We would like to use L to code all the keys in X. With that done, to describe $x\triangle y$, we just need to reference x and y. A small technical issue is that $x \in X$ may contain characters not in L. As in Section V, we assume that each position $i \in [c]$ is active with least two input position characters $(i, a) \in L$. Out of these, we pick a default position character (i, a_i). Now if key x contains $(i, x_i) \notin L$, we replace it with (i, a_i). Let x' the result of making these default replacements of all input position characters outside L. Then $x' \subseteq L$. Moreover, given any two keys $x, y \in X$, if $x\triangle y \subseteq L$, then $x'\triangle y' = x\triangle y$. Each key x' is now described with c characters from L, one for each position, and we get the maximum number of combinations if there are the same number in each position, so there are at most $(\ell/c)^c$ combinations.

Instead of just coding $X' = \{x' \mid x \in X\}$, for simplicity, we code a superset $Y' \supseteq X'$ with exactly k keys. The number of possible Y' is bounded by

$$\binom{(\ell/c)^c}{k} < \left(\frac{e(\ell/c)^c}{k}\right)^k.$$

An equation is now characterized by two keys from X' and an output position $j \in [d]$, leaving us $\binom{k}{2}d < k^2d/2$ possibilities. We can therefore pick $\lceil q\ell \rceil$ equations in less than

$$\binom{k^2d/2}{\lceil q\ell \rceil} < \left(\frac{e(k^2d/2)}{\lceil q\ell \rceil}\right)^{\lceil q\ell \rceil} \leq \left(\frac{e(k^2c)}{\varepsilon\ell}\right)^{\lceil q\ell \rceil}$$

ways. Our probability bound for a given ℓ is thus

$$\begin{aligned} Q^k_{\ell,c} &= (ec|\Phi|/\ell)^\ell \left(\frac{e(\ell/c)^c}{k}\right)^k \left(\frac{e(k^2c)}{\varepsilon\ell|\Psi|}\right)^{\lceil q\ell \rceil} \\ &\leq (ec|\Phi|/\ell)^\ell \left(\frac{e(\ell/c)^c}{k}\right)^k \left(\frac{e(k^2c)}{\varepsilon\ell|\Psi|}\right)^{\varepsilon d\ell/(2c)}. \end{aligned}$$

We are always free to use the best of our probability bounds, so with c active positions, the probability of getting a list L of size ℓ is bounded by $\min\{P_{\ell,c}, Q^k_{\ell,c}\}$. Allowing $c' \leq c$ active positions and considering all lengths $\ell = 2c', \ldots, kc'$, we get the overall probability bound

$$\sum_{c'=1}^{c}\left(\binom{c}{c'}\sum_{\ell=2c'}^{kc'}\min\{P_{\ell,c'}, Q^k_{\ell,c'}\}\right). \qquad (12)$$

Proof of Theorem 5: To prove Theorem 5, we used $\varepsilon = 1$ for uniqueness, and evaluated the sum (12) with the concrete parameters using a computer. ■

References

[1] M. N. Wegman and L. Carter, "New classes and applications of hash functions," *J. Comput. Syst. Sc.*, vol. 22, no. 3, pp. 265–279, 1981, announced at FOCS'79.

[2] L. Carter and M. N. Wegman, "Universal classes of hash functions," *J. Comput. Syst. Sc.*, vol. 18, no. 2, pp. 143–154, 1979, announced at STOC'77.

[3] B. Kernighan and D. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall, 1988.

[4] A. I. Dumey, "Indexing for rapid random access memory systems," *Computers and Automation*, vol. 5, no. 12, pp. 6–9, 1956.

[5] A. Siegel, "On universal classes of extremely random constant-time hash functions," *SIAM J. Comput.*, vol. 33, no. 3, pp. 505–543, 2004, announed at FOCS'89.

[6] M. Dietzfelbinger and P. Woelfel, "Almost random graphs with simple hash functions," in *Proc. 25th STOC*, 2003, pp. 629–638.

[7] T. Q. Klassen and P. Woelfel, "Independence of tabulation-based hash classes," in *Proc. 10th LATIN*, 2012, pp. 506–517.

[8] M. Thorup and Y. Zhang, "Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation," *SIAM J. Comput.*, vol. 41, no. 2, pp. 293–331, 2012, announced at SODA'04 and ALENEX'10.

[9] K. S. Kedlaya and C. Umans, "Fast polynomial factorization and modular composition," *SIAM J. Comput.*, vol. 40, no. 6, pp. 1767–1802, 2011.

[10] A. L. Zobrist, "A new hashing method with application for game playing," Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, Tech. Rep. 88, 1970.

[11] M. Pătraşcu and M. Thorup, "The power of simple tabulation hashing," *J. ACM*, vol. 59, no. 3, p. Article 14, 2012, announced at STOC'11.

[12] V. Guruswami, C. Umans, and S. P. Vadhan, "Unbalanced expanders and randomness extractors from Parvaresh–Vardy codes," *J. ACM*, vol. 56, no. 4, 2009, announced at CCC'07.

[13] A. C.-C. Yao, "Should tables be sorted?" *J. ACM*, vol. 28, no. 3, pp. 615–628, 1981, announced at FOCS'78.

[14] M. Pătraşcu and M. Thorup, "Twisted tabulation hashing," in *Proc. 24th SODA*, 2013, pp. 209–228.

[15] M. Dietzfelbinger and M. Rink, "Applications of a splitting trick," in *Proc. 36th ICALP*, 2009, pp. 354–365.

[16] A. Pagh and R. Pagh, "Uniform hashing in constant time and optimal space," *SIAM J. Comput.*, vol. 38, no. 1, pp. 85–96, 2008, announced at STOC'03.