Noname manuscript No.
(will be inserted by the editor)

Explicit and Efficient Hash Families Suffice for Cuckoo Hashing with a Stash

Martin Aumüller · Martin Dietzfelbinger · Philipp Woelfel

the date of receipt and acceptance should be inserted later

Abstract It is shown that for cuckoo hashing with a stash as proposed by Kirsch, Mitzenmacher, and Wieder (2008) families of very simple hash functions can be used, maintaining the favorable performance guarantees: with stash size s the probability of a rehash is $O(1/n^{s+1})$, the evaluation time is $O(s)$, the amortized expected insertion time is $O(1)$, and the worst-case deletion time is $O(1)$. Instead of the full randomness needed for the analysis of Kirsch *et al.* and of Kutzelnigg (2010) (resp. $\Theta(\log n)$-wise independence for standard cuckoo hashing) the new approach even works with 2-wise independent hash families as building blocks. Both construction and analysis build upon the work of Dietzfelbinger and Woelfel (2003). The analysis, which can also be applied to the fully random case, utilizes a graph counting argument and is much simpler than previous proofs.

Keywords randomized algorithms · hash functions · cuckoo hashing · random graphs

The second author was supported in part by DFG grant DI 412/10-2. The last author was supported by a Discovery Grant from the National Sciences and Research Council of Canada (NSERC). A preliminary version of this paper appeared under the title "Explicit and Efficient Hash Functions Suffice for Cuckoo Hashing with a Stash" in *Proceedings of the 20th Annual European Symposium on Algorithms, Ljubljana, Slovenia, September 2012, Lecture Notes in Computer Science 7501, Springer 2012.*

Martin Aumüller · Martin Dietzfelbinger
Faculty of Computer Science and Automation, Ilmenau University of Technology,
98694 Ilmenau, Germany
E-mail: martin.aumueller@tu-ilmenau.de, martin.dietzfelbinger@tu-ilmenau.de

Philipp Woelfel
Department of Computer Science, University of Calgary,
Calgary, Alberta T2N 1N4, Canada
E-mail: woelfel@cpsc.ucalgary.ca

1 Introduction

Cuckoo hashing as proposed by Pagh and Rodler [14] is a popular implementation of a dictionary with guaranteed constant lookup time. To store a set S of n keys from a universe U (i.e., a finite set), cuckoo hashing utilizes two hash functions, $h_1, h_2 : U \to [m]$, where $m = (1+\varepsilon)n$, $\varepsilon > 0$. Each key $x \in S$ is stored in one of two hash tables of size m; either in the first table at location $h_1(x)$ or in the second one at location $h_2(x)$. The pair h_1, h_2 might not be suitable to accommodate S in these two tables. In this case, a *rehash* operation is necessary, which chooses a new pair h_1, h_2 and inserts all keys anew.

In their ESA 2008 paper [9], Kirsch, Mitzenmacher, and Wieder deplored the order of magnitude of the probability of a rehash, which is as large as $\Theta(1/n)$. They proposed adding a *stash*, an additional segment of storage that can hold up to s keys for some (constant) parameter s, and showed that this change reduces the rehash probability to $\Theta(1/n^{s+1})$. However, the analysis of Kirsch *et al.* requires the hash functions to be fully random. In the journal version [10] Kirsch *et al.* posed "proving the above bounds for explicit hash families that can be represented, sampled, and evaluated efficiently" as an open problem.

Our contribution. In this paper we generalize a hash family construction proposed by Dietzfelbinger and Woelfel [8] and show that the resulting hash functions have random properties strong enough to preserve the qualities of cuckoo hashing with a stash. The proof involves a new and simpler analysis of this hashing scheme, which also works in the fully random case. The hash functions we propose have a very simple structure: they combine functions from $O(1)$-wise independent families[1] with a few tables of size $n^\delta, 0 < \delta < 1$, where δ is a constant, with random entries from $[m] = \{0, \ldots, m-1\}$. A certain choice of parameters in the construction of our hash functions for stash capacity s leads to the following attractive performance characteristics: the description of a hash function pair (h_1, h_2) consists of a table with $\sqrt{n}$ entries from $[m]^2$ and $2s+6$ functions from 2-wise independent classes. To evaluate $h_1(x)$ and $h_2(x)$ for $x \in U$, we must evaluate these $2s+6$ functions, read $2s+4$ table entries, and carry out $4s+8$ additions modulo m. Our main result implies for these hash functions and for any set $S \subseteq U$ of n keys that with probability $1 - O(1/n^{s+1})$ S can be accommodated according to the cuckoo hashing rules. Furthermore, we show that deletions run in worst-case constant time and insertions need only amortized expected constant time.

Cuckoo hashing with a stash and weak hash functions. In [10,12] it was noticed that for the analysis of cuckoo hashing with a stash of size s the properties of the so-called *cuckoo graph* $G(S, h_1, h_2)$ are central. Assume a set S and hash functions h_1 and h_2 with range $[m]$ are given. The associated cuckoo graph $G(S, h_1, h_2)$ is the bipartite multigraph whose two node sets are

[1] κ-wise independent families of hash functions are defined in Section 2.

copies of $[m]$ and whose edge set contains the n pairs $(h_1(x), h_2(x))$, for $x \in S$. It is known that a single parameter of $G = G(S, h_1, h_2)$ determines whether a stash of size s is sufficient to store S using (h_1, h_2), namely the *excess* $\mathrm{ex}(G)$, which is defined as the minimum number of edges one has to remove from G so that all connected components of the remaining graph are acyclic or unicyclic.

Lemma 1 ([10]) *The keys from S can be stored in the two tables and a stash of size s using (h_1, h_2) if and only if $\mathrm{ex}(G(S, h_1, h_2)) \leq s$.*

For the convenience of the reader, a proof is given in Appendix A.

Kirsch *et al.* [10] showed that with probability $1 - O(1/n^{s+1})$ a random bipartite graph with $2m = 2(1+\varepsilon)n$ nodes and n edges has excess at most s. Their proof uses sophisticated tools such as Poissonization and Markov chain coupling. This result generalizes the analysis of standard cuckoo hashing [14] with no stash, in which the rehash probability is $\Theta(1/n)$. Kutzelnigg [12] refined the analysis of [10] in order to determine the constant factor in the asymptotic bound of the rehash probability. His proof uses generating functions and differential recurrence equations. Both approaches inherently require that the hash functions h_1 and h_2 used in the algorithm are fully random.

Recently, Pătraşcu and Thorup [15] showed that using simple tabulation hash functions for running cuckoo hashing leads to a rehash probability of $\Theta(1/n^{1/3})$, which is tight. The counterexample of [15], which yields the bound of $\Theta(1/n^{1/3})$, can be generalized to the stash case.[2] Thus, these hash functions do not benefit from using a stash.

Our main contribution is a new analysis that shows that explicit and efficient hash families are sufficient to obtain the $O(1/n^{s+1})$ bound on the rehash probability. We build upon the work of Dietzfelbinger and Woelfel [8]. For standard cuckoo hashing, they proposed hash functions of the form $h_i(x) = \big(f_i(x) + z^{(i)}[g(x)]\big) \bmod m$, for $x \in U$, and $i \in \{1, 2\}$, where f_i and g are from $2k$-wise independent classes with range $[m]$ and $[\ell]$, resp., and $z^{(1)}, z^{(2)} \in [m]^\ell$ are random vectors. They showed that with such hash functions the rehash probability is $O(1/n + n/\ell^k)$. Their proof has parts (i) and (ii). Part (i) already appeared in [3] and [14]: The rehash probability is bounded by the sum, taken over all minimal excess-1 graphs H of different sizes and all subsets T of S, of the probability that $G(T, h_1, h_2)$ is isomorphic to H. In Section 4 of this paper we demonstrate that for h_1 and h_2 fully random a similar counting approach also works for minimal excess-$(s+1)$ graphs, whose presence in $G(S, h_1, h_2)$ determines whether a rehash is needed when a stash of size s is used. As in [14], this analysis also works for $O((s+1)\log n)$-wise independent families.

Part (ii) of the analysis in [8] is a little more subtle. It shows that for each key set S of size n there is a part B_S^{conn} of the probability space given by (h_1, h_2) such that $\Pr(B_S^{\mathrm{conn}}) = O(n/\ell^k)$ and in $\overline{B_S^{\mathrm{conn}}}$ the hash functions act fully randomly on $T \subseteq S$ as long as $G(T, h_1, h_2)$ is connected. In Section 4 we show how this argument can be adapted to the situation with a stash, using excess core graphs in place of the connected subgraphs. Woelfel [19]

[2] Personal communication with Mikkel Thorup, 2012.

already demonstrated by applying functions in [8] to balanced allocation that the approach has more general potential to it.

A comment on the "full randomness assumption" and work relating to it seems in order. It is often quoted as an empirical observation that weaker hash functions like κ-wise independent families will behave almost like random functions. Mitzenmacher and Vadhan [13] showed that if the key set S has a certain kind of entropy then 2-wise independent hash functions will behave similar to fully random ones. However, as demonstrated in [7], there are situations where cuckoo hashing fails for a standard 2-wise independent family and even a random set S (which is "too dense" in U). The rather general "split-and-share" approach of [6] makes it possible to justify the full randomness assumption for many situations involving hash functions, including cuckoo hashing (with a stash) and further variants. However, for practical application this method is less attractive, since space consumption and failure probability are negatively affected by splitting the key set into "chunks" and treating these separately.

Organization of this paper Section 2 introduces our hash functions and proves their randomness properties. Section 3 provides the basic setup of our analysis. In Section 4, we will analyze the rehash probability of cuckoo hashing with a stash when we use our class of hash functions. Subsequently, in Section 5 resp. Section 6 we will consider the performance of the insert resp. delete operation when running cuckoo hashing with a stash. Finally, Section 7 gives an experimental evaluation of different constructions of our hash class.

2 Basics

Let U (the "universe") be a finite set. A mapping from U to $[r]$ is a *hash function with range* $[r]$. For an integer $\kappa \geq 2$, a set $\mathcal{H}$ of hash functions with range $[r]$ is called a *κ-wise independent* hash family if for arbitrary distinct keys $x_1, \ldots, x_\kappa \in U$ and for arbitrary $j_1, \ldots, j_\kappa \in [r]$ we have $\Pr_{h\in\mathcal{H}}\big(h(x_1) = j_1 \wedge \ldots \wedge h(x_\kappa) = j_\kappa\big) = 1/r^\kappa$. The classical κ-wise independent hash family construction is based on polynomials of degree $\kappa - 1$ over a finite field [18]. More efficient hash function evaluation can be achieved with tabulation-based constructions [8,16,17,11]. Throughout this paper, $\mathcal{H}_r^\kappa$ denotes an arbitrary κ-wise independent hash family with domain U and range $[r]$.

We combine κ-wise independent classes with lookups in tables of size ℓ in order to obtain pairs of hash functions from U to $[m]$:

Definition 1 Let $c \geq 1$ and $\kappa \geq 2$. For integers m, $\ell \geq 1$, and given $f_1, f_2 \in \mathcal{H}_m^\kappa$, $g_1, \ldots, g_c \in \mathcal{H}_\ell^\kappa$, and vectors $z_j^{(i)} \in [m]^\ell$, $1 \leq j \leq c$, for $i \in \{1,2\}$, let $(h_1, h_2) = (h_1, h_2)\langle f_1, f_2, g_1, \ldots, g_c, z_1^{(1)}, \ldots, z_c^{(1)}, z_1^{(2)}, \ldots, z_c^{(2)}\rangle$, where

$$h_i(x) = \Big(f_i(x) + \textstyle\sum_{1\leq j\leq c} z_j^{(i)}[g_j(x)]\Big) \bmod m, \text{ for } x \in U, i \in \{1,2\}.$$

Let $\mathcal{Z}_{\ell,m}^{\kappa,c}$ be the family of all these pairs (h_1, h_2) of hash functions.

While this is not reflected in the notation, we consider (h_1, h_2) as a structure from which the components $g_1, \ldots, g_c$ and $f_i, z_1^{(i)}, \ldots, z_c^{(i)}$, $i \in \{1, 2\}$, can be read off again. It is family $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{2k,c}$, for some $k \geq 1$, made into a probability space by the uniform distribution, that we will study in the following. We usually assume that c and k are fixed and that m and ℓ are known.

2.1 Basic Facts

We start with some basic observations concerning the effects of compression properties in the "g-part" of (h_1, h_2), extending similar statements in [8].

Definition 2 For $T \subseteq U$, define the random variable d_T, the "deficiency" of (h_1, h_2) with respect to T, by $d_T((h_1, h_2)) = |T| - \max\{k, |g_1(T)|, \ldots, |g_c(T)|\}$. *(*Note*: d_T depends only on the g_j-components of (h_1, h_2).)* Further, define
(i) bad_T as the event that $d_T > k$;
(ii) $good_T$ as $\overline{bad_T}$, i. e., the event that $d_T \leq k$;
(iii) $crit_T$ as the event that $d_T = k$.
Hash function pairs (h_1, h_2) in these events are called *"T-bad"*, *"T-good"*, and *"T-critical"*, resp.

Lemma 2 *Assume $k \geq 1$ and $c \geq 1$. For $T \subseteq U$, the following holds*:

(a) $\Pr(\text{bad}_T \cup \text{crit}_T) \leq \left(|T|^{2k}/\ell^k\right)^c$.
(b) *Conditioned on* good_T*, the pairs $(h_1(x), h_2(x))$, $x \in T$, are distributed uniformly and independently in $[r]^2$. The same is true if we condition on* crit_T.

Proof (a) Assume $|T| \geq 2k$ (otherwise the events bad_T and crit_T cannot occur). Since $g_1, \ldots, g_c$ are independent, it suffices to show that for a function g chosen randomly from $\mathcal{H}_\ell^{2k}$ we have $\Pr(|T| - |g(T)| \geq k) \leq |T|^{2k}/\ell^k$.

We first argue that if $|T| - |g(T)| \geq k$ then there is a subset T' of T with $|T'| = 2k$ and $|g(T')| \leq k$. Initialize T' as T. Repeat the following as long as $|T'| > 2k$: (i) if there exists a key $x \in T'$ such that $g(x) \neq g(y)$ for all $y \in T' \setminus \{x\}$, remove x from T'; (ii) otherwise, remove any key. Clearly, this process terminates with $|T'| = 2k$. It also maintains the invariant $|T'| - |g(T')| \geq k$: In case (i) $|T'| - |g(T')|$ remains unchanged. In case (ii) before the key is removed from T' we have $|g(T')| \leq |T'|/2$ and thus $|T'| - |g(T')| \geq |T'|/2 > k$.

Now fix a subset T' of T of size $2k$ that satisfies $|g(T')| \leq k$. The preimages $g^{-1}(u)$, $u \in g(T')$, partition T' into $k' := |g(T')| \leq k$ classes such that g is constant on each class. Since g is chosen from a $2k$-wise independent class, the probability that g is constant on all classes of a given partition of T' into classes $C_1, \ldots, C_{k'}$, with $k' \leq k$, is exactly $\prod_{i=1}^{k'} \ell^{-(|C_i|-1)} = \ell^{-(2k-k')} \leq \ell^{-k}$.

Finally, we bound $\Pr(|g(T)| \leq |T| - k)$. There are $\binom{|T|}{2k}$ subsets T' of T of size $2k$. Every partition of such a set T' into $k' \leq k$ classes can be represented by a permutation of T' with k' cycles, where each cycle contains the elements

from one class. Hence, there are at most $(2k)!$ such partitions. This yields:

$$\Pr(|T| - |g(T)| \geq k) \leq \binom{|T|}{2k} \cdot (2k)! \cdot \frac{1}{\ell^k} \leq \frac{|T|^{2k}}{\ell^k}. \tag{1}$$

(b) If $|T| \leq 2k$, then h_1 and h_2 are fully random on T simply because f_1 and f_2 are $2k$-wise independent. So suppose $|T| > 2k$. Fix an arbitrary g-part of (h_1, h_2) so that good_T occurs, i.e., $\max\{k, |g_1(T)|, \ldots, |g_c(T)|\} \geq |T| - k$. Let $j_0 \in \{1, \ldots, c\}$ be such that $|g_{j_0}(T)| \geq |T| - k$. Arbitrarily fix all values in the tables $z_j^{(i)}$ with $j \neq j_0$ and $i \in \{1, 2\}$. Let T^* be the set of keys in T colliding with other keys in T under g_{j_0}. Then $|T^*| \leq 2k$. Choose the values $z_{j_0}^{(i)}[g_{j_0}(x)]$ for all $x \in T^*$ and $i \in \{1, 2\}$ at random. Furthermore, choose f_1 and f_2 at random from the $2k$-wise independent family $\mathcal{H}_r^{2k}$. This determines $h_1(x)$ and $h_2(x)$, $x \in T^*$, as fully random values. Furthermore, the function g_{j_0} maps the keys $x \in T - T^*$ to distinct entries of the vectors $z_{j_0}^{(i)}$ that were not fixed before. Thus, the hash function values $h_1(x), h_2(x)$, $x \in T - T^*$, are distributed fully randomly as well and are independent of those with $x \in T^*$—the proof for conditioning on crit_T is the same. □

We remark that the prove of part (b) is still possible if we use 2-universal hash families for the g_j functions. A family $\mathcal{H}$ of hash functions with range R is 2-*universal* if for each pair $x, y \in U$, $x \neq y$, and h chosen at random from $\mathcal{H}$ we have $\Pr(h(x) = h(y)) \leq 2/|R|$. For constructions, see, e. g., [2,5]. These constructions are in general more efficient than those for k-wise independence. In Section 7, we will experimentally evaluate the impact of using such hash families.

3 Graph Properties, Basic Setup and a Graph Counting Argument

A node with degree 1 in a graph is called a *leaf*; an edge incident with a leaf is called a *leaf edge*. An edge is called a *cycle edge* if removing it does not disconnect any two nodes. A graph is called *leafless* if it has no leaves. For $m \in \mathbb{N}$ let $\mathcal{G}_m$ denote the set of all bipartite (multi-)graphs with vertex set $[m]$ on each side of the bipartition. A set $\mathcal{A} \subseteq \mathcal{G}_m$ is called a *graph property*. For example, $\mathcal{A}$ could be the set of graphs in $\mathcal{G}_m$ that have excess larger than s.

Definition 3 Let $\mathcal{A}$ be a graph property, let $S \subseteq U$, and let $T \subseteq S$.

(a) $\mathcal{A}_T \subseteq \mathcal{Z}$ denotes the event that $G(T, h_1, h_2)$ has property $\mathcal{A}$ (i. e., that $G(T, h_1, h_2) \in \mathcal{A}$).
(b) $B_S^{\mathcal{A}} \subseteq \mathcal{Z}$ denotes the event that $\exists T \subseteq S \colon \mathcal{A}_T \cap \text{bad}_T$ (see Definition 2).

In the following, our main objective is to bound the probability $\Pr(\exists T \subseteq S \colon \mathcal{A}_T)$ for graph properties $\mathcal{A}$ which are important for our analysis.

For the next lemma we need the following definition.

Definition 4 For $T \subseteq U$, a graph property $\mathcal{A}$, and (h_1^*, h_2^*) a pair of fully random hash functions let $p_T^{\mathcal{A}} = \Pr(G(T, h_1^*, h_2^*) \in \mathcal{A})$.

Lemma 3 *For an arbitrary graph property $\mathcal{A}$ we have*

$$\Pr(\exists T \subseteq S\colon \mathcal{A}_T) \leq \Pr(B_S^{\mathcal{A}}) + \sum_{T \subseteq S} p_T^{\mathcal{A}}. \tag{2}$$

Proof $\Pr(\exists T \subseteq S\colon \mathcal{A}_T) \leq \Pr(B_S^{\mathcal{A}}) + \Pr((\exists T \subseteq S : \mathcal{A}_T) \cap \overline{B_S^{\mathcal{A}}})$, and

$$\sum_{T \subseteq S} \Pr(\mathcal{A}_T \cap \overline{B_S^{\mathcal{A}}}) \underset{\text{(i)}}{\leq} \sum_{T \subseteq S} \Pr(\mathcal{A}_T \cap \text{good}_T) \leq \sum_{T \subseteq S} \Pr(\mathcal{A}_T \mid \text{good}_T) \underset{\text{(ii)}}{=} \sum_{T \subseteq S} p_T^{\mathcal{A}},$$

where (i) holds by the definition of $B_S^{\mathcal{A}}$, and (ii) holds by Lemma 2(b). □

This lemma encapsulates our overall strategy for bounding $\Pr(\exists T \subseteq S\colon \mathcal{A}_T)$. When bounding this probability, we will use the following graph counting approach.

First, let us recall a standard notion from graph theory (already used in [8]; *cf.* Appendix A.1): The *cyclomatic number* $\gamma(G)$ of a graph G is the smallest number of edges one has to remove from G to obtain a graph with no cycles. Also, let $\zeta(G)$ denote the number of connected components of G (ignoring isolated vertices).

Lemma 4 *Let $N(t, \ell, \gamma, \zeta)$ be the number of non-isomorphic (multi-)graphs with ζ connected components and cyclomatic number γ that have t edges, ℓ of which are leaf edges. Then $N(t, \ell, \gamma, \zeta) = t^{O(\ell+\gamma+\zeta)}$.*

Proof We will prove this lemma in three steps:

1. $N(t, \ell, 0, 1) = t^{O(\ell)}$
2. $N(t, \ell, \gamma, 1) = t^{O(\ell+\gamma)}$
3. $N(t, \ell, \gamma, \zeta) = t^{O(\ell+\gamma+\zeta)}$

Part 1. We first consider the case $\gamma = 0$ and $\zeta = 1$, thus we consider trees. For $\ell = 2$, the tree is a path of length t. We refer to this tree with G_2 (the index refers to the number of leaf edges in the graph). For $\ell = 3, \ldots, \ell$, G_i is constructed using G_{i-1} by taking a new path of length $t_i \geq 1$ such that $t_2 + \cdots + t_i \leq t - (\ell - i)$ and identify one endpoint of the path with a vertex in G_{i-1}. The length of the last path is uniquely determined by $t_\ell = t - t_2 - \cdots - t_{\ell-1}$. There are fewer than $t^{\ell-2}$ choices for picking these lengths. Furthermore, there are at most $t^{\ell-2} < t^{O(\ell)}$ choices for the inner vertex a new path is connected to. It follows

$$N(t, \ell, 0, 1) < t^{O(\ell)}.$$

Part 2. Now we consider the case where $\gamma \geq 1$, $\ell \geq 0$ and $\zeta = 1$, i. e. connected graphs with cycles. In this case, removing γ cycle edges yields a tree. There are fewer than $t^{2\gamma} = t^{O(\gamma)}$ choices for the endpoints of these edges and the remaining tree has at most $\ell + 2\gamma$ leaf edges. Thus,

$$N(t, \ell, \gamma, 1) < t^{O(\gamma)} \cdot N(t - \gamma, \ell + 2\gamma, 0, 1) < t^{O(\gamma)} \cdot t^{O(\ell+\gamma)} = t^{O(\gamma+\ell)}.$$

Part 3. Now we consider the general case. Each graph G with cyclomatic number γ, ζ connected components, $t-\ell$ non-leaf edges, and ℓ leaf edges can be obtained from some connected graph G' with cyclomatic number γ, $t-\ell+\zeta-1$ non-leaf edges, and ℓ leaf edges by removing $\zeta-1$ non-leaf, non-cycle edges. There are no more than $(t-\ell+\zeta-1)^{\zeta-1}$ ways for choosing the edges to be removed. This implies, using [8, Lemma 2]:

$$\begin{aligned} N(t,\ell,\gamma,\zeta) &\leq N(t+\zeta-1,\ell,\gamma,1)\cdot(t-\ell+\zeta-1)^{\zeta-1} \\ &\leq (t+\zeta)^{O(\ell+\gamma)}\cdot(t+\zeta)^{\zeta} = (t+\zeta)^{O(\ell+\gamma+\zeta)} = t^{O(\ell+\gamma+\zeta)}. \qquad \square \end{aligned}$$

We will now consider the rehash probability in cuckoo hashing with a stash.

4 The Rehash Probability in Cuckoo Hashing With a Stash

In this section we prove the desired bound on the rehash probability of cuckoo hashing with a stash when functions from $\mathcal{Z}$ are used. We focus on the question whether the pair (h_1, h_2) allows storing key set S in the two tables with a stash of size s. A detailed discussion of the insertion procedure will follow in the next section.

Lemma 1 suggests that we should focus on graphs having excess at least $s+1$ to bound the rehash probability. We start by identifying minimal graphs with excess $s+1$.

Definition 5 An *excess-$(s+1)$ core graph* is a leafless graph G with excess exactly $s+1$ in which all connected components have at least two cycles. By $\mathsf{CS}^{(s+1)}$ we denote the set of all excess-$(s+1)$ core graphs in $\mathcal{G}_m$.

Figure 1 gives an example of a cuckoo graph and an excess-3 core structure.

Lemma 5 *Let G be an arbitrary graph from $\mathcal{G}_m$ with excess at least $s+1$. Then G contains an excess-$(s+1)$ core graph as a subgraph.*

Proof We repeatedly remove edges from G. First we remove cycle edges until the excess is exactly $s+1$. Then we remove components that are trees or unicyclic. Finally we remove leaf edges one by one until the remaining graph is leafless. $\square$

We are now ready to state our main theorem.

Theorem 1 *Let $\varepsilon > 0$ and $0 < \delta < 1$, let $s \geq 0$ and $k \geq 1$ be given. Assume $c \geq (s+2)/(\delta k)$. For $n \geq 1$ consider $m \geq (1+\varepsilon)n$ and $\ell = n^{\delta}$. Let $S \subseteq U$ with $|S| = n$. Then for (h_1, h_2) chosen at random from $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{2k,c}$ the following holds:*

$$\Pr(\mathrm{ex}(G(S,h_1,h_2)) \geq s+1) = O(1/n^{s+1}).$$

In view of Lemma 3 and Lemma 5, we can write

$$\begin{aligned} \Pr(\mathrm{ex}(G(S,h_1,h_2)) \geq s+1) &\leq \Pr(\exists T \subseteq S : \mathsf{CS}_T^{(s+1)}) \\ &\leq \Pr(B_S^{\mathsf{CS}^{(s+1)}}) + \sum_{T\subseteq S} p_T^{\mathsf{CS}^{(s+1)}}. \end{aligned} \tag{3}$$

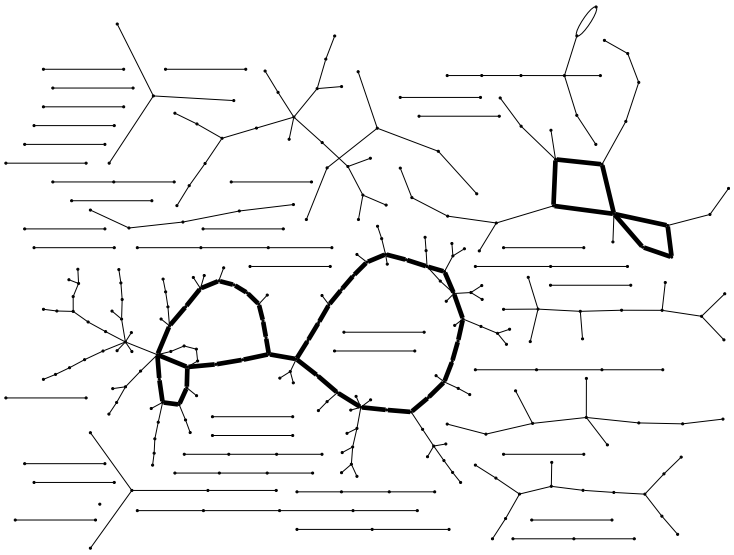

Fig. 1 An example for a cuckoo graph which contains an excess-3 core structure (bold edges). This subgraph indicates that a stash of size at most 2 does not suffice to encompass the key set.

We will consider these two summands independently. We remark that the following calculations also give an alternative, simpler proof of [9, Theorem 2.1] for the fully random case, even if the effort needed to prove Lemma 4 is taken into account.

Lemma 6 *Let $\varepsilon > 0$. Let $S \subseteq U$ with $|S| = n$ and $s \geq 0$ be given. Set $m = (1+\varepsilon)n$. Then*

$$\sum_{T \subseteq S} p_T^{\mathsf{CS}^{(s+1)}} = O(1/n^{s+1}).$$

Proof We start by counting (unlabeled) excess-$(s+1)$ core graphs with t edges. A connected component C of such a graph G with cyclomatic number $\gamma(C)$ (which is at least 2) contributes $\gamma(C) - 1$ to the excess of G. This means that if G has $\zeta = \zeta(G)$ components, then $s + 1 = \gamma(G) - \zeta$ and $\zeta \leq s + 1$, and hence $\gamma = \gamma(G) \leq 2(s+1)$. Using Lemma 4, there are at most $N(t, 0, \gamma, \zeta) = t^{O(\gamma+\zeta)} = t^{O(s)}$ such graphs G. If from each component C of such a graph G we remove $\gamma(C) - 1$ cycle edges, we get unicyclic components, which have as many nodes as edges. This implies that G has $t - (s+1)$ nodes.

Now fix a bipartite (unlabeled) excess-$(s+1)$ core graph G with t edges and ζ components, and let $T \subseteq U$ with $|T| = t$ be given. There are $2^\zeta \leq 2^{s+1}$ ways of assigning the $t - s - 1$ nodes to the two sides of the bipartition, and then at most m^{t-s-1} ways of assigning labels from $[m]$ to the nodes. Thus, the number of bipartite graphs with property $\mathsf{CS}^{(s+1)}$, where each node is labeled with one side of the bipartition and an element of $[m]$, and where the t edges are labeled with distinct elements of T is smaller than $t! \cdot 2^{s+1} \cdot m^{t-s-1} \cdot t^{O(s)}$.

Now if G with such a labeling is fixed, and we choose t edges from $[m]^2$ uniformly at random, the probability that all edges $(h_1(x), h_2(x))$, $x \in T$, match the labeling is $1/m^{2t}$. For constant s, this yields the following bound:

$$\sum_{T\subseteq S} p_T^{\mathsf{CS}^{(s+1)}} \leq \sum_{s+3\leq t\leq n} \binom{n}{t} \frac{2^{s+1}\cdot n^{t-s-1}\cdot t!\cdot t^{O(s)}}{m^{2t}} \leq \frac{2^{s+1}}{n^{s+1}} \cdot \sum_{s+3\leq t\leq n} \frac{n^t\cdot t^{O(1)}}{m^t}$$
$$= O\Big(\frac{1}{n^{s+1}}\Big)\cdot \sum_{s+3\leq t\leq n} \frac{t^{O(1)}}{(1+\varepsilon)^t} = O\Big(\frac{1}{n^{s+1}}\Big). \qquad \square$$

We will now bound $\Pr(B_S^{\mathsf{CS}^{(s+1)}})$.

Lemma 7 *Let $\varepsilon > 0$, let $s \geq 0$, let $c, k \geq 1$, let $S \subseteq U$ with $|S| = n$, and let $m = (1+\varepsilon)n$. Assume (h_1, h_2) is chosen at random from $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{2k,c}$. Then*

$$\Pr(B_S^{\mathsf{CS}^{(s+1)}}) = O\big(n/\ell^{ck}\big).$$

To prove this bound, we need the following auxiliary graph property. A graph from $\mathcal{G}_m$ belongs to $\mathsf{LCY}^{(s+1)}$ if it has the following four properties:

1. at most one connected component of G contains leaves;
2. the number $\zeta(G)$ of connected components is bounded by $s+1$;
3. if present, the leaf component of G contains at most 2 leaf edges;
4. the cyclomatic number $\gamma(G)$ is bounded by $2(s+1)$.

Lemma 8 *If $T \subseteq U$ and (h_1, h_2) is from $\mathcal{Z}$ such that $G(T, h_1, h_2) \in \mathsf{CS}^{(s+1)}$ and (h_1, h_2) is T-bad, then there exists a subset T' of T such that $G(T', h_1, h_2) \in \mathsf{LCY}^{(s+1)}$ and (h_1, h_2) is T'-critical.*

Proof Fix T and (h_1, h_2) as in the assumption. Then $d_T((h_1, h_2)) > k$ and $G(T, h_1, h_2) \in \mathsf{CS}^{(s+1)}$. This means that $G(T, h_1, h_2) \in \mathsf{LCY}^{(s+1)}$ as well. To see this, we have to check that $\gamma(G(T, h_1, h_2)) \leq 2(s+1)$. Each component C in $G(T, h_1, h_2)$ with cyclomatic number $\gamma(C)$ contributes $\gamma(C) - 1$ to the excess of the graph. Since there are at most $s+1$ connected components, the cyclomatic number of this graph is at most $2(s+1)$.

Initialize T' as T. As just noted, we then have $d_{T'}((h_1, h_2)) > k$ and $G(T', h_1, h_2) \in \mathsf{LCY}^{(s+1)}$. We will remove ("peel") edges from $G(T', h_1, h_2)$. Of course, by "removing edge $(h_1(x), h_2(x))$ from $G(T', h_1, h_2)$" we mean removing x from T'. Since $d_{T'}((h_1, h_2))$ can decrease by at most 1 when an edge is removed, we finally reach a situation where $d_{T'}((h_1, h_2)) = k$, i. e., (h_1, h_2) is T'-critical. Our goal is to preserve $G(T', h_1, h_2) \in \mathsf{LCY}^{(s+1)}$ in each step.

We use the following iterative process to remove edges. We disregard isolated vertices in the whole process. Initialize C as an arbitrary connected component of $G(T', h_1, h_2)$. Whenever we remove an edge from C, we remove this edge from $G(T', h_1, h_2)$ as well.

While $d_{T'}((h_1, h_2)) > k$, repeat the following steps. If C contains a leaf edge, remove a leaf edge. Otherwise, remove an arbitrary cycle edge from C. If

we removed the last edge of C, arbitrarily choose a new connected component and call it C.

We claim that this process maintains the property $G(T', h_1, h_2) \in \mathsf{LCY}^{(s+1)}$ in each step. To see this, note that initially $G(T', h_1, h_2)$ does not contain leaf edges. Removing a cycle edge yields at most 2 leaf edges. Removing a leaf edge does not increase the number of leaf edges. Furthermore, removing these two types of edges does not destroy connectivity—disregarding isolated vertices. We conclude that eventually the described process stops and that the resulting graph has property $\mathsf{LCY}^{(s+1)}$; also, $d_{T'}((h_1, h_2))$ is T'-critical. □

What have we achieved? By Lemma 8, we can bound $\Pr(B_S^{\mathsf{CS}^{(s+1)}})$ by just adding, over all $T' \subseteq S$, the probabilities $\Pr(\mathsf{LCY}_{T'}^{(s+1)} \cap \mathrm{crit}_{T'})$, that means, the terms $\Pr(\mathsf{LCY}_{T'}^{(s+1)} \mid \mathrm{crit}_{T'}) \cdot \Pr(\mathrm{crit}_{T'})$. Lemma 2(a) takes care of the second factor. By Lemma 2(b), we may assume that (h_1, h_2) acts fully random on T' for the first factor. The next lemma estimates this factor, using the notation from Section 3.

Lemma 9 *Let $T \subseteq U, |T| = t$, and $s \geq 0$. Then*

$$p_T^{\mathsf{LCY}^{(s+1)}} \leq t! \cdot t^{O(1)}/m^{t-1}.$$

Proof By Lemma 4, there are at most $t^{O(s)} = t^{O(1)}$ ways to choose a bipartite graph G in $\mathsf{LCY}^{(s+1)}$ with t edges. Graph G cannot have more than $t+1$ nodes, since cyclic components have at most as many nodes as edges, and in the single leaf component, if present, the number of nodes is at most one bigger than the number of edges. In each component of G, there are two ways to assign the vertices to the two sides of the bipartition. After such an assignment is fixed, there are at most m^{t+1} ways to label the vertices with elements of $[m]$, and there are $t!$ ways to label the edges of G with the keys in T. Assume now such labels have been chosen for G. Draw t edges $(h_1^*(x), h_2^*(x))$ from $[m]^2$ uniformly at random. The probability that they exactly fit the labeling of nodes and edges of G is $1/m^{2t}$. Thus, $p_T^{\mathsf{LCY}^{(s+1)}} \leq m^{t+1} \cdot t! \cdot t^{O(1)}/m^{2t} = t! \cdot t^{O(1)}/m^{t-1}$. □

We can now prove Lemma 7.

Proof (of Lemma 7) By Lemma 8, and using the union bound, we get

$$\begin{aligned}\Pr(B_S^{\mathsf{CS}^{(s+1)}}) &= \Pr(\exists T \subseteq S : \mathsf{CS}_T^{(s+1)} \cap \mathrm{bad}_T) \leq \Pr(\exists T' \subseteq S : \mathsf{LCY}_{T'}^{(s+1)} \cap \mathrm{crit}_{T'}) \\ &\leq \sum_{T' \subseteq S} \Pr(\mathsf{LCY}_{T'}^{(s+1)} \mid \mathrm{crit}_{T'}) \cdot \Pr(\mathrm{crit}_{T'}) =: \rho_S.\end{aligned}$$

By Lemma 2(b), given the event that (h_1, h_2) is T'-critical, (h_1, h_2) acts fully random on T'. Using Lemma 9 and Lemma 2(a), this yields:

$$\Pr(\mathsf{LCY}_{T'}^{(s+1)} \mid \mathrm{crit}_{T'}) \cdot \Pr(\mathrm{crit}_{T'}) \leq (|T'|! \cdot |T'|^{O(1)}/m^{|T'|-1}) \cdot (|T'|^2/\ell)^{ck}.$$

Summing up, collecting sets T' of equal size together, and using that ck is constant, we obtain

$$\rho_S \leq \sum_{2k \leq t \leq n} \binom{n}{t} \cdot \frac{t! \cdot t^{O(1)}}{m^{t-1}} \cdot \left(\frac{t^2}{\ell}\right)^{ck} \leq \frac{n}{\ell^{ck}} \cdot \sum_{2k \leq t \leq n} \frac{t^{O(1)}}{(1+\varepsilon)^{t-1}} = O\Big(\frac{n}{\ell^{ck}}\Big). \qquad \square$$

We can now put everything together to prove Theorem 1.

Proof (of Theorem 1) By inequality (3), the probability that the excess of $G(S, h_1, h_2)$ is at least $s+1$ is at most

$$\Pr(\exists T \subseteq S \colon \mathsf{CS}_T^{(s+1)}) \leq \Pr(B_S^{\mathsf{CS}^{(s+1)}}) + \sum_{T \subseteq S} p_T^{\mathsf{CS}^{(s+1)}}.$$

By plugging in the values from Lemma 6 and Lemma 7, and using $\ell = n^\delta$ and $c \geq (s+2)/(k\delta)$, we calculate

$$\Pr(\mathrm{ex}(G(S, h_1, h_2)) \geq s+1) = O(n/\ell^{ck}) + O(1/n^{s+1}) = O(1/n^{s+1}). \qquad \square$$

In conclusion, the rehash probability of cuckoo hashing with a stash does not increase if we use hash functions from $\mathcal{Z}$ instead of fully random hash functions.

5 Insertions in Cuckoo Hashing with a Stash

We consider here the obvious generalization of the insertion procedure in standard cuckoo hashing [14]. It assumes that a procedure *rehash* is given that will choose two new hash functions and insert all keys anew. The parameter *maxloop* is used for avoiding infinite loops. (When using cuckoo hashing with a stash of size s, we will see that it suffices to set $\mathit{maxloop} = \Theta((s+2)\log n)$. For analysis purposes, larger values of *maxloop* are considered as well.) Empty table cells contain the value **nil**. The operation *swap* exchanges the contents of two variables.

Algorithm 1 (Insertion in a cuckoo table with a stash)

```
procedure stashInsert(x: key)
(1)   if lookup(x) = true then return;
(2)   nestless := x;
(3)   i := 1;
(4)   repeat maxloop times
(5)       swap(nestless, T_i[h_i(nestless)]);
(6)       if nestless = nil then return;
(7)       i := 3 − i;
(8)   if stash is not yet full
(9)       then add nestless to stash
(10)      else rehash.
```

As long as it is not finished, the procedure maintains a "nestless" key (in `nestless`) and the current index $i \in \{1, 2\}$ (in `i`) of the table where this key

is to be placed. When a new key x is to be inserted, it is declared "nestless" and i is set to 1. As long as there is a nestless key x, but at most for *maxloop* iterations ("rounds"), the following is repeated: Assume x is nestless and the current index is i. Then x is placed in position $h_i(x)$ in table T_i. If this position is empty, the procedure terminates; if it contains a key x', that key gets evicted to make room for x, is declared nestless, and i is changed to the other value $3 - i$. If the loop does not terminate within *maxloop* rounds, the key that is currently nestless gets stored in the stash. If this causes the stash to overflow, a *rehash* is carried out. (This may be realized by collecting all keys from tables and stash as well as the nestless key, choosing a new pair (h_1, h_2) of hash functions, and calling the insertion procedure for all keys.)

5.1 Complete Insertion Loops and the Excess

We first look at the behavior of certain variants of the insertion procedure, which we call *complete*, that exhibit the following behavior when x is inserted: (i) if with *maxloop* set to infinity the loop were to run forever, then this is noticed and at some point the currently nestless key is put in the stash; (ii) otherwise the loop is left to run until the nestless key is stored in an empty cell. It is not hard to see (*cf.* [3]) that one obtains a complete variant from Algorithm 1 if one chooses *maxloop* as some number larger than $2|S| + 3$.

Proposition 1 ([10,12]) *If inserting the keys of S by some complete insertion procedure places s keys in the stash, then $s = \mathrm{ex}(G(S)) = \mathrm{ex}(G(S, h_1, h_2))$.*

Proof "$\geq$": After the insertion is complete, all keys from S are stored in the two tables and the stash. Lemma 1 implies that $s \geq \mathrm{ex}(G(S))$.
"$\leq$": For this, we use induction on the size of S. If $S = \emptyset$, excess and stash size are both 0. Now assume as induction hypothesis that set S has been inserted, that the set of keys placed in the stash is T, and that $|T| = s \leq \mathrm{ex}(G(S))$. Let $S' = S - T$. We insert a new key y from $U - S$.
Case 1: The insertion procedure finds that y can be accommodated without using the stash.—The stash size remains s, and $s \leq \mathrm{ex}(G(S)) \leq \mathrm{ex}(G(S \cup \{y\}))$.
Case 2: The complete insertion procedure notices that the loop were to run forever and places some key in the stash.—By the properties of the complete insertion loop for standard cuckoo hashing as explored in [3] we know that $G(S' \cup \{y\})$ must contain a connected component that is neither acyclic nor unicyclic. Since $\mathrm{ex}(G(S')) = 0$, it must be edge $(h_1(y), h_2(y))$ that makes the difference. This means that each endpoint of $(h_1(y), h_2(y))$ lies in some cyclic component of $G(S')$. Now $G(S')$ is a subgraph of $G(S)$, so the same is true in $G(S)$.

For the next argument, we make use of the result that the excess of a graph is the number of cyclic connected components of this graph substracted from its cyclomatic number.[3] Consider two cases when changing from S to $S \cup \{y\}$:

[3] This result appeared in [9] and [12]. We give an alternative proof in Lemma 13 in Appendix A.

If the endpoints of $(h_1(x), h_2(x))$ lie in two different cyclic components of $G(S)$, then the number of cyclic components decreases by 1, hence the excess increases by 1; if they lie in one and the same cyclic component, then the cyclomatic number increases by 1, and the excess increases by 1 as well. In both cases we get that $s + 1 \leq \mathrm{ex}(G(S)) + 1 = \mathrm{ex}(G(S \cup \{y\}))$. □

5.2 Standard Insertion and Hash Class $\mathcal{Z}$

It turns out that by choosing $maxloop = \Theta((s + 2) \log n)$ in Algorithm 1 we can make sure that with probability of $O(1/n^{s+1})$ no rehash is necessary, even in the case that we use hash functions from $\mathcal{Z}$ instead of fully random hash functions. Note that if the stash has size 0, then Algorithm 1 is exactly the insertion procedure of standard cuckoo hashing from [14].

Theorem 2 *Let $\varepsilon > 0$ and $0 < \delta < 1$, let $s \geq 0$ and $k \geq 1$ be given. Assume $c \geq (s+2)/(\delta k)$. For $n \geq 1$ consider $m \geq (1+\varepsilon)n$ and $\ell = n^\delta$. Let $S \subseteq U$ with $|S| = n$. Let (h_1, h_2) be chosen at random from $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{2k,c}$. Assume that the keys from S are inserted sequentially into a cuckoo table with a stash of size s using Algorithm 1. Then the following holds:*

(i) *If we choose $maxloop = \alpha(s + 2) \log n$ for a suitable constant $\alpha > 0$ then*

$$\Pr(\textit{the stash of size } s \textit{ overflows}) = O(1/n^{s+1}).$$

(ii) *Assume that S is stored in the cuckoo table with stash size s. If a new key y is inserted by Algorithm 1, then the expected number of steps needed for this insertion is $O(1)$.*

To prove this theorem, we make use of the following fact that already appeared in [14] (in a different terminology).

Fact 3 *Let $S = \{x_1, \ldots, x_n\}$, with the keys listed in the order in which they are inserted, and let $S_j = \{x_1, \ldots, x_j\}$, for $1 \leq j \leq n$. Assume that the insertion procedure for x_j needs t or more rounds.*

Then there exists a path $u_0, u_1, \ldots, u_p, u_{p+1}$ in $G(S_j)$ with $p = \lceil t/3 \rceil$, where u_0 corresponds to either $T_1[h_1(x_j)]$ or $T_2[h_2(x_j)]$ and $u_0, \ldots, u_p$ are distinct nodes.

In view of this fact, we let P^t contain all graphs from $\mathcal{G}_m$ that form a simple path of length $\lceil t/3 \rceil$ (disregarding isolated vertices). We can now calculate

$$\Pr(\text{an insertion of a key from } S \text{ needs at least } t \text{ rounds}) \leq \Pr(\exists T \subseteq S \colon \mathsf{P}_T^t)$$

Using (2), we can write this probability as

$$\Pr(\exists T \subseteq S \colon \mathsf{P}_T^t) \leq \Pr(B_S^{\mathsf{P}^t}) + \sum_{T \subseteq S} p_T^{\mathsf{P}^t} \tag{4}$$

We consider these two summands separately.

Lemma 10 *Let $\varepsilon > 0$, let $s \geq 0$, let $c, k, t \geq 1$, let $S \subseteq U$ with $|S| = n$, and let $m = (1+\varepsilon)n$. Assume (h_1, h_2) is chosen at random from $\mathcal{Z} = \mathcal{Z}_{\ell,m}^{2k,c}$. Then*

$$\Pr(B_S^{\mathsf{P}^t}) = O\big(n/\ell^{ck}\big).$$

Proof Assume that $B_S^{\mathsf{P}^t}$ occurs. Fix $T \subseteq S$ and (h_1, h_2) such that we have $G(T, h_1, h_2) \in \mathsf{P}^t$ and bad_T. Thus, $G(T, h_1, h_2)$ forms a simple path and $d_T((h_1, h_2)) > k$. Since $\mathsf{P}^t \subseteq \mathsf{LCY}^{(1)}$, we also have $G(T, h_1, h_2) \in \mathsf{LCY}^{(1)}$. As in the proof of Lemma 8, we will now remove edges from this graph one by one.

Initialize T' as T. While $d_{T'}((h_1, h_2)) > k$, keep removing leaf edges $(h_1(x), h_2(x))$ from $G(T', h_1, h_2)$, i.e., remove x from T' . This process stops when $d_{T'}((h_1, h_2)) = k$ and hence $\mathrm{crit}_{T'}$ occurs. Moreover, $G(T', h_1, h_2)$ still forms a simple path. We conclude:

$$\Pr(B_S^{\mathsf{P}^t}) = \Pr(\exists T \subseteq S \colon \mathsf{P}_T^t \cap \mathrm{bad}_T) \leq \Pr(\exists T' \subseteq S \colon \mathsf{LCY}_{T'}^{(1)} \cap \mathrm{crit}_{T'}) = O(n/\ell^{ck}),$$

where the last step follows from calculations that can be found in the proof of Lemma 7. □

Now we consider the second summand in (4). Bounding a sum like this in a random graph was central in the original analysis of cuckoo hashing, see [14, Section 2.3].

Lemma 11 *Let $\varepsilon > 0$. Let $S \subseteq U$ with $|S| = n$. Set $m = (1+\varepsilon)n$ and $t = 3(s+2)\lceil \log_{1+\varepsilon} n \rceil$. Then*

$$\sum_{T \subseteq S} p_T^{\mathsf{P}^t} = O(1/n^{s+1}).$$

Proof A simple path $u_0, u_1, \ldots, u_k$ of length $k = \lceil t/3 \rceil$ has $k+1$ vertices. First, fix $i \in \{1, 2\}$ such that u_0 belongs to table T_i. Once this is fixed, there are $\binom{m}{k+1}$ possibilities to choose the vertices on this path. There are $k!$ many ways to assign the keys from T to the edges of the simple path. Fix such a simple path p. The probability that a pair of fully random hash functions yields p on the keys from T is $1/m^{2k}$. We calculate:

$$\sum_{T \subseteq S} p_T^{\mathsf{P}^t} = \sum_{\substack{T \subseteq S \\ |T| = k}} p_T^{\mathsf{P}^t} = \binom{n}{k} \cdot 2 \cdot \binom{m}{k+1} \cdot k! \cdot 1/m^{2k} \leq \frac{2m}{(1+\varepsilon)^k} = \frac{2m}{(1+\varepsilon)^{\lceil t/3 \rceil}}.$$

Plugging in $t = 3(s+2)\lceil \log_{1+\varepsilon} n \rceil$ shows the lemma. □

We can now prove the main theorem of this section:

Proof (of Theorem 2) Let $maxloop = 3(s+2)\lceil \log_{1+\varepsilon} n \rceil$.

Part (i): Applying Lemma 10 and Lemma 11 to (4), we get

$$\Pr(\exists T \subseteq S \colon \mathsf{P}_T^{maxloop}) = O(n/\ell^{ck}) + O(1/n^{s+1}).$$

Using $\ell = n^\delta$ and $c \geq (s+2)/(\delta k)$ shows that this probability is bounded by $O(1/n^{s+1})$.

Part (ii): Assume that S is the set of keys stored in the cuckoo table with stash size s and let $y \in U \setminus S$ be the key that is to be inserted. Let the random variable L denote the number of rounds needed for this insertion. First, assume that y triggers no rehash, ignoring the contribution of such an event to the total insertion time. Furthermore, assume that $B_S^{\mathsf{P}^{maxloop}}$ does not happen. The contribution of the event $B_S^{\mathsf{P}^{maxloop}}$ to the expected overall insertion time is at most $\Pr(B_S^{\mathsf{P}^{maxloop}}) \cdot \mathit{maxloop} = O((\log n)/n^{s+1}) = o(1)$.

To calculate the probability that at least t rounds are needed to store y, we again use Fact 3. As in the proof of Theorem 2, Part (ii), there must exist a simple path $u_0, \ldots, u_k$ in $G(S, h_1, h_2)$. Using that u_0 either corresponds to $T_1[h_1(y)]$ or $T_2[h_2(y)]$, and using the same line of argument as above, we calculate

$$\begin{aligned}
\mathrm{E}(L) &\leq \sum_{t \geq 1} \Pr(\text{at least } t \text{ rounds are needed to store } y) \\
&\leq \sum_{t \geq 1} \frac{2}{(1+\varepsilon)^{\lceil t/3 \rceil}} \\
&= O(1).
\end{aligned}$$

Now assume that the insertion of y triggers a rehash event. The insertion stops after *maxloop* rounds and the rehash operation is invoked; new hash functions are chosen and the data structure is build anew. First, assume that all insertions are successful, which happens with probability p_{succ}. As above, each such insertion takes expected constant time. Thus, the expected running time R_{succ} in this case is in $O(n)$. Now assume that an insertion is not successful and again a rehash event is triggered. All key insertions prior to the last insertion were successful and ran in expected constant time. The insertion that invoked the rehash operation took *maxloop* rounds. Thus, the expected time is $R_{\text{unsucc}} = O(n) + O(\mathit{maxloop}) = O(n)$. Now, the rehash is triggered, new hash functions are chosen, and the data structure is build anew. Thus, for the total expected time R of a rehash event we have:

$$R \leq p_{\text{succ}} \cdot R_{\text{succ}} + (1 - p_{\text{succ}}) \cdot (R_{\text{unsucc}} + R).$$

Solving for R we get:

$$\begin{aligned}
R &\leq R_{\text{succ}} + \frac{1 - p_{\text{succ}}}{p_{\text{succ}}} \cdot R_{\text{unsucc}} \\
&= O(n) + O(1/n^{s+1}) \cdot O(n) = O(n).
\end{aligned}$$

We conclude that the contribution of a rehash event to the expected insertion time of y is $O(1/n^{s+1}) \cdot O(n) = O(1)$. □

In conclusion, using hash class $\mathcal{Z}$ preserves the performance of the insertion procedure in cuckoo hashing with a stash.

6 Deletions in Cuckoo Hashing with a Stash

In this section, we consider deletions in cuckoo hashing with a stash. We review two methods proposed by Kirsch *et al.* [10] and Arbitman [1], respectively. For each method we determine whether it works with our class of hash functions. Whenever we refer to the *maxloop* value, we mean the value used in Theorem 2.

Deletions are simple in standard cuckoo hashing: To remove a key x, we have to check whether x resides in $T_1[h_1(x)]$ or $T_2[h_2(x)]$ and remove it from the specific cell, if present. With a stash, we have to check the whole stash for the key if it is not stored in either T_1 or T_2. Additionally, if a key is removed from a table cell it might be possible that a key x from the stash can be reinserted into the two tables. If this is the case, we say that x *unnecessarily* resides in the stash. The naïve approach to avoid keys unnecessarily residing in the stash is to reinsert all stash keys upon each deletion. However, this yields a non-constant worst-case time for deletions.

Postponing the reinsertion of all stash keys yields the following three negative properties:

(i) The stash might overflow although the keys can be stored in the data structure with the current hash functions.
(ii) The lookup of keys unnecessarily residing in the stash takes longer.
(iii) An unsuccessful lookup operation unnecessarily has to check stash keys.

In light of (i), prior to a rehash operation one should try to reinsert all stash keys. Unfortunately, this does not avoid the effects of (ii) and (iii). The methods of [1,10] distribute the work needed for the reinsertion of all stash keys to operations immediately following a delete operation. In this way, deletions preserve their worst-case constant time, while stash keys are reinserted over time. Now we review each method in detail:

1. Kirsch *et al.* [10] propose reinserting all stash keys at the beginning of the insert operation following a delete operation. Each such insertion takes time $\mathit{maxloop} = O(\log n)$ in the worst-case. However, by Theorem 1 we know that the stash is empty with probability $1 - O(1/n)$. Thus, the contribution of the time needed to reinsert all stash keys to the expected insertion time is $O((\log n)/n) = o(1)$ using our class of hash functions.
2. In his Master's Thesis, Arbitman [1] suggests distributing the work of reinserting all stash keys to several operations following a delete operation in slices of constant cost. This is done by splitting up a single reinsertion into a logarithmic number of operations and conducting constant work in each such operation. We slightly modify the approach in [1] to avoid implementing a cycle detection mechanism. The stash is implemented as a double-ended queue. Additionally, we have a counter `cnt`, initialized as 0. New elements are always added to the tail of the queue.
   The idea is as follows: After each $\log n$ operations on the data structure, we take the first element from the stash and try to insert it into the hash table for L steps, where L is some arbitrary constant. If the key is successfully inserted, `cnt` is reset to 0 and we are done. Otherwise, let y be the nestless

key after L rounds of the insertion algorithm. If $\texttt{cnt} + L > \mathit{maxloop}$ then y is inserted at the tail of the stash and `cnt` is reset to 0. Otherwise, y is put at the head of the stash and `cnt` is incremented by L. In addition, we store in a bit `table` the table y got evicted from. The subsequent reinsertion of y will then begin at the other table.

Obviously, the approach of Arbitman adds only a constant overhead to some operations. However, it is not clear that all keys unnecessarily residing in the stash are moved back to the hash table in this way. We will only sketch the proof that this is indeed true with high probability. In fact, if a key x is unnecessarily in the stash then with high probability after at most $O(\log^2 n)$ operations this key is successfully reinserted into the hash table. We ignore the influence of operations that occur between these $O(\log^2 n)$ operations. The details of the proof can be found in the work of Arbitman [1].

A fixed element of the stash is after at most $O(\log^2 n)$ operations the first element in the stash. Assume that the head of the stash together with the keys stored in the two hash tables forms a connected component with more than one cycle in the cuckoo graph. Then this is a key that belongs to the stash and there is nothing to show, because the insertion fails. Now assume that the first element is a key x that could be stored in the hash table. If the insertion succeeds then again there is nothing to show. So, let y be the key that is moved to the back of the stash after $\lceil \mathit{maxloop}/L \rceil$ insertion trials (indicating that the insertion of x was not successful). As in the previous section, the cuckoo graph contains a simple path of length at least $\mathit{maxloop}/3$ in this case. However, we have shown in the previous section that the probability of such a long path to exist is at most $O(1/n^{s+1})$, using our class of hash functions and setting $\mathit{maxloop} = \Theta((s+2)\log n)$. Thus, the probability that a key is not reinserted into the hash tables—although it would be possible—is at most $O(1/n^{s+1})$. A union bound over the constant number of keys residing in the stash shows that there exists no such key in the stash with probability $O(1/n^{s+1})$.

In conclusion, we support deletions in cuckoo hashing with a stash in worst-case constant time. Our hash class supports two variants of dealing with keys unnecessarily residing in the stash, and guarantees the same performance as when we would use fully random hash functions in both cases.

7 Experimental Evaluation of our Hash Functions

In view of the main results of this paper, a hash function from $\mathcal{Z}^{2k,c}_{\ell,m}$ guarantees the same performance as a fully random hash function when we choose $\ell = n^\delta$ and $c \geq (s+2)/(k\delta)$ (*cf.* Theorem 1 and Theorem 2). We have three main parameters:

1. The size ℓ of the random tables.
2. The degree of independence $2k$ required for the functions f_1, f_2 and the functions g_j.
3. The number c of random tables and functions g_j.

In the following, we present experiments for creating a cuckoo hashing data structure that contains n unsigned 32-bit integers. We fix $\ell = \sqrt{n}$.

Following Kirsch *et al.* [9], we wish for a failure probability of $O(1/n^4)$ and consequently use a stash size of 3. The parameters c and k must then satisfy $c \geq 10/k$. We evaluate the following three constructions:

1. **Low Space Usage, Few Hash Functions**. We set $c = 1$ and $k = 10$. Thus, a hash function pair from this hash class contains three 20-wise independent hash functions f_1, f_2, g_1 and two tables of size $\sqrt{n}$ filled with random values. To implement a 20-wise independent hash class, we used polynomials of degree 19 over some prime field. More specifically, we fixed the prime to be the Mersenne prime $p = 2^{48} - 1$ and applied the so-called CW-Trick of Carter and Wegman [2]. This approach is described in more detail in [17].
2. **Tabulation-based Hash Functions**. We used the 5-wise independent tabulation class of Thorup and Zhang [17]. Due to the construction of our hash class, f_1, f_2 and the g_j's must have an even level of independence. For $2k = 4$ we have $c \geq 5$. A hash function pair from this class uses seven 5-wise independent hash functions $f_1, f_2, g_1, \ldots, g_5$ and ten tables filled with random values.
3. **Universal Hashing**. The fastest hash functions we can use as building blocks for our hash functions are 2-universal. We used the multiplication-shift scheme [5]

$$h_a(x) = (ax \bmod 2^{32}) \operatorname{div} 2^{32-\ell_{\text{out}}},$$

which in 32-bit arithmetic can be implemented as

$$h_a(x) = (ax) \gg (32 - \ell_{\text{out}}),$$

where we set ℓ_{out} to the smallest integer such that $2^{\ell_{\text{out}}} \geq \sqrt{n}$ and used $2^{\ell_{\text{out}}}$ as the size of our random tables. In this case, we can set $k = 1$ and get $c = 10$. A hash functions from this class has two 2-wise independent hash functions f_1, f_2, ten 2-universal hash functions $g_1, \ldots, g_{10}$ and twenty tables filled with random values.

In the following, we call these constructions *construction 1*, *construction 2*, and *construction 3*, respectively.

We observe that apart from the description lengths of these constructions, the difference in evaluation time is not clear. While constructions 2 and 3 use faster hash functions as building blocks than construction 1, they need more evaluations of such functions to calculate a single hash value.

Our experiments were carried out on a 6-core Intel Xeon E5645 at 2.4 GHz with 48 GB Ram running Ubuntu 12.04 with kernel version 3.2.0. The implementation was written in C++. For generating random numbers and measuring time, we used the boost library.[4] The source code was compiled with *gcc* using the *-O2* optimization flag.[5]

[4] `http://www.boost.org`

[5] Source code available at: `http://eiche.theoinf.tu-ilmenau.de/ch-stash/`

n	1	2	3	TZ
5 000	1.76 ms	1.48 ms	0.62 ms	0.25 ms
10 000	3.52 ms	2.97 ms	1.21 ms	0.54 ms
50 000	18.02 ms	15.44 ms	6.80 ms	3.06 ms
100 000	36.64 ms	31.57 ms	14.72 ms	6.46 ms
500 000	188.10 ms	164.66 ms	86.98 ms	38.61 ms
1 000 000	383.68 ms	340.48 ms	202.47 ms	84.22 ms

Table 1 Measured time in milliseconds to construct a cuckoo hashing data structure with n keys and table size $m = 1.05 \cdot n$. The table shows results for the first, second, and third construction, resp. Additionally, we ran the experiments using hash functions from Thorup and Zhang's Tabulation class [17]. Table entries are the median of the construction times over 10 000 tries for each parameter setting.

We considered the following scenario for setting up a cuckoo hashing data structure: For $\varepsilon = 0.05$ and a random key sequence of unsigned 32-bit integers of length $n \in \{5\,000, 10\,000, 50\,000, 100\,000, 500\,000, 1\,000\,000\}$, we set up three hash function pairs from $\mathcal{Z}$ at random according to the parameter choices from above. The tables of the cuckoo hashing data structure have $m = (1 + \varepsilon)n$ cells each. We insert all keys from the sequence of keys. For each choice of parameters, we repeat this experiment 10 000 times.

Table 1 shows the construction time for fixed $\varepsilon = 0.05$ and different values of n. As we can see, the third construction (using 2-universal hash functions) presents the fastest way to construct a cuckoo hashing data structure using our class of hash functions, although it uses the most evaluations of f and g functions. It is by a factor of 1.68 faster than construction 2. Construction two is by a factor of 1.13 faster than construction 1, which represents the slowest way of constructing the data structure. In general, the construction times using our fastest construction are about 2.5 times slower than if we used tabulation hashing of Thorup and Zhang for h_1 and h_2 directly, without having guaranteed bounds on the rehash probability.

All three hash function constructions showed the same behavior with respect to the average resp. maximum number of rounds during an insertion for fixed n and ε.

For $\varepsilon = 0.025$ there occured rehash events for the parameter choices $n \in \{5\,000, 10\,000, 50\,000\}$. For larger values of n and ε, no rehash events occured. Table 2 shows an example for $\varepsilon = 0.025$ and $n = 50\,000$. The number of rehashes do not significantly differ when using a complete insertion procedure in contrast to the standard insertion procedure with $\mathit{maxloop} = \Theta((s+2)\log n)$.

Concluding Remarks

We presented a family of efficient hash functions and showed that it exhibits sufficiently strong random properties to run cuckoo hashing with a stash, preserving the favorable performance guarantees of this hashing scheme. It remains open whether generalized cuckoo hashing can be run with efficient hash families.

construction	$s = 0$	$s = 1$	$s = 2$	$s = 3$	rehashes
1	9 619	326	46	8	1
2	9 615	333	40	8	4
3	9 600	332	55	9	4
fully random	9 574	357	52	13	4

Table 2 Stash sizes s for $n = 50\,000$ and $\varepsilon = 0.025$ and 10 000 runs. Construction 1 triggered one rehash event over 10 000 insertions, constructions 2 and 3 both required four rehash operations. In all cases, the keys could be accommodated in the cuckoo hashing data structure after the rehash. For completeness, we ran the experiments using fully random hash functions. The results are shown in the last row.

Acknowledgements

We thank Pascal Klaue for implementing the algorithms and carrying out the experiments presented in Section 7.

References

1. Arbitman, Y.: Efficient dictionary data structures based on cuckoo hashing (2010)
2. Carter, L., Wegman, M.N.: Universal classes of hash functions. *J. Comput. Syst. Sci.* **18**(2), 143–154 (1979)
3. Devroye, L., Morin, P.: Cuckoo hashing: Further analysis. *Inf. Process. Lett.* **86**(4), 215–219 (2003)
4. Diestel, R.: *Graph Theory.* Springer (2005)
5. Dietzfelbinger, M., Hagerup, T., Katajainen, J., Penttonen, M.: A reliable randomized algorithm for the closest-pair problem. *J. Algorithms* **25**(1), 19–51 (1997)
6. Dietzfelbinger, M., Rink, M.: Applications of a splitting trick. In: Proc. 36th ICALP (1), LNCS 5555, pp. 354–365. Springer (2009)
7. Dietzfelbinger, M., Schellbach, U.: On risks of using cuckoo hashing with simple universal hash classes. In: Proc. 20th SODA, pp. 795–804 (2009)
8. Dietzfelbinger, M., Woelfel, P.: Almost random graphs with simple hash functions. In: Proc. 35th STOC, pp. 629–638. New York, NY, USA (2003). DOI http://doi.acm.org/10.1145/780542.780634
9. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: Cuckoo hashing with a stash. In: Proc. 16th ESA 2008, LNCS 5193, pp. 611–622. Springer (2008)
10. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.* **39**(4), 1543–1561 (2009)
11. Klassen, T.Q., Woelfel, P.: Independence of tabulation-based hash classes. In: Proc. 10th LATIN 2012, LNCS 7256, pp. 506–517. Springer (2012)
12. Kutzelnigg, R.: A further analysis of cuckoo hashing with a stash and random graphs of excess r. *Discr. Math. and Theoret. Comput. Sci.* **12**(3), 81–102 (2010)
13. Mitzenmacher, M., Vadhan, S.P.: Why simple hash functions work: exploiting the entropy in a data stream. In: Proc. 19th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 746–755 (2008)
14. Pagh, R., Rodler, F.F.: Cuckoo hashing. *J. Algorithms* **51**(2), 122–144 (2004)
15. Pătraşcu, M., Thorup, M.: The power of simple tabulation hashing. In: Proc. 43rd ACM Symp. on Theory of Computing (STOC), pp. 1–10 (2011)
16. Thorup, M., Zhang, Y.: Tabulation based 4-universal hashing with applications to second moment estimation. In: Proc. 15th Ann. ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 615–624 (2004)
17. Thorup, M., Zhang, Y.: Tabulation-based 5-independent hashing with applications to linear probing and second moment estimation. SIAM J. Comput. **41**(2), 293–331 (2012)
18. Wegman, M.N., Carter, L.: New hash functions and their use in authentication and set equality. In: J. Comput. System Sci., 22, pp. 265–279 (1981)

19. Woelfel, P.: Asymmetric balanced allocation with simple hash functions. In: Proc. 17th ACM-SIAM Symp. on Discrete Algorithms (SODA), pp. 424–433 (2006)

A Excess, Stash Size, and Insertions

In this supplementary section, provided for the convenience of the reader, we clarify the connection between stash size needed and the excess $\mathrm{ex}(G(S, h_1, h_2))$ of the cuckoo graph $G(S, h_1, h_2)$ as well as the role of insertion procedures. In particular, we prove Lemma 1. The central statements of this section can also be found in [10, 12].

A.1 The Excess of a Graph

For G a graph, $\zeta(G)$ denotes the number of connected components of G. The cyclomatic number $\gamma(G)$, technically defined as "the dimension of the cycle space of G", can be characterized by the following basic formula [4]:

$$\gamma(G) = m - n + \zeta(G), \tag{5}$$

for n the number of nodes and m the number of edges of G. Note that acyclic graphs are characterized by the equation $n = m + \zeta(G)$ and hence by the equation $\gamma(G) = 0$. Using (5), two helpful ways of viewing $\gamma(G)$ are easy to prove.

Lemma 12 (a) *If we remove edges from G sequentially, in an arbitrary order, and the resulting graph is acyclic, then $\gamma(G)$ is the number of removed cycle edges—edges that are on a cycle when removed.*

(b) *$\gamma(G)$ is the minimum number of edges one has to remove from G such that the resulting graph is acyclic.*

Proof Assume a subgraph G' of G (with all n nodes) has $m' > 0$ edges. If we remove one edge e' from G' to obtain G'', we have, using (5) twice:

$$\gamma(G'') = (m' - 1) - n + \zeta(G'') = \gamma(G') - (1 - (\zeta(G'') - \zeta(G'))).$$

We observe:

- If e' is a cycle edge in G', then $\zeta(G'') = \zeta(G')$, and hence $\gamma(G'') = \gamma(G') - 1$.
- If e' is not a cycle edge, then $\zeta(G'') = \zeta(G') + 1$, and hence $\gamma(G'') = \gamma(G')$.

We prove (a): By what we just observed, to reduce the cyclomatic number from $\gamma(G)$ to 0 the number of rounds in which an edge is removed that is on a cycle must be $\gamma(G)$. Now we prove (b): Think of the edges as being removed sequentially. Again, by our observation, in order to reduce the cyclomatic number from $\gamma(G)$ to 0 by removing as few edges as possible we should never remove an edge that is not on a cycle. In this way we remove exactly $\gamma(G)$ (cycle) edges. □

We have defined the excess $\mathrm{ex}(G)$ of a graph G as the minimum number of edges one has to remove from G so that the remaining subgraph has only acyclic and unicyclic components. In [12] the characterization of this quantity given next was used as a definition; the same idea was used in [10] (without giving it a name).

For G a graph, let $\zeta_{\mathrm{cyc}}(G)$ denote the number of cyclic components of G.

Lemma 13 *In all graphs G the equation $\mathrm{ex}(G) = \gamma(G) - \zeta_{\mathrm{cyc}}(G)$ is satisfied.*

Proof Assume G has n nodes and m edges.
"$\leq$": Starting with G, we iteratively remove *cycle* edges until each cyclic component has only one cycle left. The number of edges removed is at least $\mathrm{ex}(G)$. Call the resulting graph G'. Removing one cycle edge from each of the $\zeta_{\mathrm{cyc}}(G)$ cyclic components of G' will yield an acyclic graph. Lemma 12(a) tells us that together exactly $\gamma(G)$ edges have been removed; hence $\gamma(G) \geq \mathrm{ex}(G) + \zeta_{\mathrm{cyc}}(G)$.
"$\geq$": Choose a set E^+ of $\mathrm{ex}(G)$ edges in G such that removing these edges leaves a graph G' with only acyclic and unicyclic components. Now imagine that the edges in E^+ are removed one by one in an arbitrary order. Let β denote the number of edges in E^+ that are on a cycle when removed; the other $\mathrm{ex}(G) - \beta$ many were non-cycle edges when removed. Removing one cycle edge from each cyclic component of G' will leave an acyclic graph. Counting the number of cycle edges we removed altogether, and applying Lemma 12(a) again, we see that $\gamma(G) = \beta + \zeta_{\mathrm{cyc}}(G')$. Since removing a non-cycle edge from a graph can increase the number of cyclic components by at most 1, we have that $\zeta_{\mathrm{cyc}}(G') \leq \zeta_{\mathrm{cyc}}(G) + (\mathrm{ex}(G) - \beta)$. Combining the inequalities yields $\gamma(G) \leq \zeta_{\mathrm{cyc}}(G) + \mathrm{ex}(G)$. □

A.2 The Excess of the Cuckoo Graph and the Stash Size

The purpose of this section is to prove Lemma 1, which we recall here. We assume that h_1 and h_2 are given, and write $G(S)$ for $G(S, h_1, h_2)$, for $S \subseteq U$.

Lemma 1 ([10]) *The keys from S can be stored in the two tables and a stash of size s using (h_1, h_2) if and only if $\mathrm{ex}(G(S)) \leq s$.*

Proof "$\Rightarrow$": Assume T is a subset of S of size at most s such that all keys from $S' = S - T$ can be stored in the two tables. Then all components of $G(S')$ must be acyclic or unicyclic. (Assume C is a component with $\gamma(C) > 1$. Then by (5) the number of edges (keys) in C would be strictly larger than the number of nodes (table positions), which is impossible.) Since $G(S')$ is obtained from $G(S)$ by removing the edges $(h_1(x), h_2(x))$, $x \in T$, we get $\mathrm{ex}(G(S)) \leq s$.
"$\Leftarrow$": Assume $\mathrm{ex}(G(S)) \leq s$. Choose a subset T of S of size $\mathrm{ex}(G(S))$ such that $G(S - T)$ has only acyclic and unicyclic components. From what is known about the behaviour of standard cuckoo hashing, we can store $S' = S - T$ in the two tables using h_1 and h_2 (e. g., see [3, Sect. 4]). (This can even be proved directly. If one of the nodes touched by an edge $(h_1(x), h_2(x))$, $x \in S'$, has degree 1, we place x in the corresponding cell. Iterating this, we can place all keys excepting those that belong to cycle edges. Since $G(S')$ has only acyclic and unicyclic components, the cycle edges form isolated simple cycles, and clearly the keys that belong to such a cycle can be placed in the corresponding cells.) By assumption, the keys from T fit into the stash. □