Insertion Time of Random Walk Cuckoo Hashing below the Peeling Threshold

Stefan Walzer
Universität zu Köln, Germany

Abstract

Most hash tables have an insertion time of $\mathcal{O}(1)$, often qualified as "expected" and/or "amortised". While insertions into cuckoo hash tables indeed seem to take $\mathcal{O}(1)$ expected time in practice, only polylogarithmic guarantees are proven in all but the simplest of practically relevant cases. Given the widespread use of cuckoo hashing to implement compact dictionaries and Bloom filter alternatives, closing this gap is an important open problem for theoreticians.

In this paper, we show that random walk insertions into cuckoo hash tables take $\mathcal{O}(1)$ expected amortised time when any number $k \geq 3$ of hash functions is used and the load factor is below the corresponding *peeling threshold* (e.g. ≈ 0.81 for $k = 3$). To our knowledge, this is the first meaningful guarantee for constant time insertion for cuckoo hashing that works for $k \in \{3, \ldots, 9\}$.

In addition to being useful in its own right, we hope that our key-centred analysis method can be a stepping stone on the path to the true end goal: $\mathcal{O}(1)$ time insertions for all load factors below the *load threshold* (e.g. ≈ 0.91 for $k = 3$).

2012 ACM Subject Classification Theory of computation → Bloom filters and hashing; Theory of computation → Design and analysis of algorithms; Mathematics of computing → Random graphs

Keywords and phrases Cuckoo Hashing, Random Walk, Random Hypergraph, Peeling, Cores

Digital Object Identifier 10.4230/LIPIcs.ESA.2022.87

Related Version *Full Version*: `https://arxiv.org/abs/2202.05546`

Funding *Stefan Walzer*: DFG grant WA 5025/1-1.

Acknowledgements I would like to thank Martin Dietzfelbinger for providing several useful comments that helped with improving the presentation of this paper as well as an anonymous reviewer who gave useful feedback regarding the technical argument.

1 Introduction

Cuckoo Hashing Basics. Cuckoo hashing is an elegant approach for constructing compact and efficient dictionaries that has spawned both a rich landscape of theoretical results and popular practical applications. Briefly, each key x in a set of m keys is assigned k positions $h_1(x), \ldots, h_k(x)$ in an array of $n \geq m$ buckets via hash functions $h_1, \ldots, h_k$. Each bucket can hold at most ℓ keys and the challenge is to choose for each key one of its assigned buckets while respecting bucket capacities. We follow many previous works in assuming $k \geq 2$ and $\ell \geq 1$ to be constants and $h_1, \ldots, h_k$ to be uniformly random functions (but see e.g. [1, 2, 3] for works pursuing cuckoo hashing with explicit hash families).

Since the term cuckoo hashing was coined for $(k, \ell) = (2, 1)$ [27] and then generalised to $k \geq 3$ [12] and $\ell \geq 2$ [7], a major focus of theory papers has been to determine the load thresholds $c^*_{k,\ell}$, which are constants such that for a load factor $\frac{m}{n} < c^*_{k,\ell} - \varepsilon$ a placement of all keys exists with high probability (whp, defined in this paper as probability $1 - n^{-\Omega(1)}$) and for $\frac{m}{n} > c^*_{k,\ell} + \varepsilon$ a placement does not exist whp. This project has since been completed [11, 4, 6, 14, 13, 24] and we reproduce some thresholds in Table 1 for reference. Further research pursued cuckoo hashing variants with reduced failure probability [21], improved

load thresholds [7, 23, 30] or weaker randomness requirements for $h_1, \dots, h_k$ [1, 2, 3, 25, 22]. Moreover, cuckoo filters [9, 10, 8], which are Bloom filter alternatives based on cuckoo hashing, are now widely popular in the data base community.

Table 1 For each $2 \le k \le 7$ and $1 \le \ell \le 6$ a cell shows $c^{\triangle}_{k,\ell}/\ell$ (left) and $c^{*}_{k,\ell}/\ell$ (right), rounded to three decimal places. The thresholds are divided by ℓ to reflect the corresponding memory efficiency (populated space over allocated space).

$\ell \backslash k$	2	3	4	5	6	7
1	– /0.500	0.818/0.918	0.772/0.977	0.702/0.992	0.637/0.997	0.582/0.999
2	0.838/0.897	0.776/0.988	0.667/0.998	0.579/1	0.511/1	0.457/1
3	0.858/0.959	0.725/0.997	0.604/1	0.515/1	0.450/1	0.399/1
4	0.850/0.980	0.687/0.999	0.562/1	0.476/1	0.412/1	0.364/1
5	0.837/0.990	0.658/1	0.533/1	0.448/1	0.387/1	0.341/1
6	0.823/0.994	0.635/1	0.511/1	0.427/1	0.368/1	0.323/1

Cuckoo Insertions. An important concern in all variants of cuckoo hashing is how to insert a new key x into an existing data structure. If all buckets $h_1(x), \dots, h_k(x)$ are full, then one key that is currently placed in those buckets has to be moved out of the way into one of its alternative buckets, which might require additional relocations of keys. Two natural strategies for organising insertions have been proposed [12]. Breadth-first search (BFS) insertion systematically pursues all possibilities for relocating keys in parallel. Random walk (RW) insertion starts by optimistically placing x into bucket $h_i(x)$ for a uniformly random $i \in [k]$ (where in this paper $[a] := \{1, \dots, a\}$ for $a \in \mathbb{N}$), without first considering any of the $k-1$ other buckets. If $h_i(x)$ was already full, then a random key is evicted from $h_i(x)$ and is itself placed into one of its $k-1$ alternative buckets. This chain of evictions continues until a bucket with leftover space is reached (see Figure 1 for the case with $\ell = 1$).

```
1 Algorithm RW(x):
2     i_old ← ⊥
3     repeat
4         pick random i ∈ {h_1(x), ..., h_k(x)} \ {i_old}
5         swap(x, B[i])
6         i_old ← i
7     until x = ⊥
```

Figure 1 The random walk insertion algorithm for $\ell = 1$. The array B of buckets is initialised with $\perp$.

Experiments suggest that, regardless of k and ℓ, and for any load factor $\frac{m}{n} < c^{*}_{k,\ell} - \varepsilon$ where insertions still succeed whp, the expected insertion time is independent of n, hence "$\mathcal{O}(1)$" (neglecting dependence on the constants k, ℓ, ε), for both RW and BFS. Despite some partial success (see below), this claim has not been proven for any k and ℓ, neither for RW nor for BFS, with the exception of $(k, \ell) = (2, 1)$, which behaves very differently compared to other cases. A theoretical explanation for the good performance of cuckoo hashing in practical applications is therefore seriously lacking in this aspect. While this paper does not solve the problem, it puts a new kind of dent into it.

Contribution. Like most previous papers on cuckoo hashing insertions (an exception [18] is mentioned below) we focus on the case $\ell = 1$. Our analysis shows that RW insertions take $\mathcal{O}(1)$ expected amortised time for all $k \geq 3$, but it only works for $\frac{m}{n} < c^{\triangle}_{k,1} - \varepsilon$ where $c^{\triangle}_{k,\ell} < c^{*}_{k,\ell}$ is known as the *peeling threshold* or threshold for the occurrence of an $(\ell+1)$-core in a random k-uniform hypergraph [28, 26, 5, 19], see Table 1. Our analysis extends to a setting where the m insertions are carried out by m threads in parallel, each executing RW. We consider the worst case, where a (possibly adversarial) scheduler arbitrarily assigns the available computation time to threads that have not yet terminated. We only assume that the scheduler is oblivious of future random choices and that swaps are atomic, i.e. when several threads perform swaps concurrently, the effect is the same as executing these swaps in *some* sequential order (see e.g. [29, Sec. 2.4] for common parallel programming models).

▶ **Theorem 1.** *Let $k \in \mathbb{N}$ with $k \geq 3$ and $\varepsilon > 0$ be constants and $n, m \in \mathbb{N}$ with $\frac{m}{n} < c^{\triangle}_{k,1} - \varepsilon$.*

(i) *Conditioned on a high probability event, sequentially inserting m keys into a cuckoo hash table with n buckets of size 1 using RW takes $\mathcal{O}(n)$ steps in expectation.*

(ii) *The same applies if the m insertions are started in parallel with arbitrary scheduling, only assuming that swaps are atomic. In other words, the combined work is $\mathcal{O}(n)$.*

Related Work and Comparison. Table 2 summarises related work that we now discuss from left to right.

[20] is only included here to show that the case of *static* cuckoo hash tables is well understood with optimal results in all considered categories.

[12] offers a strong analysis of BFS. The only downside is that it does not work for some small k and does not reach all the way to $c^{*}_{k,1}$. These issues might be resolvable by modernising the proof (the values $c^{*}_{k,1}$ were not known at the time of writing). Note, however, that even full success on this front would not render an analysis of RW irrelevant as several authors, including [12], see significant *practical* benefits of RW over BFS.

[17, 15] propose and improve, respectively, an analysis of RW via graph expansion. It guarantees most desired properties, including a concentration bound on insertion times. The major downside is an only polylogarithmic bound on expected insertion time.

[16] are first to prove an $\mathcal{O}(1)$ bound on expected random walk insertion time. The proof extends to any load factor $1 - \varepsilon$ with $\varepsilon > 0$. There is a downside, however. Instead of using $k = \mathcal{O}(\log(1/\varepsilon))$ hash functions as would be required for the existence of a placement of all keys (and as are used by [12] in their BFS analysis), the authors use $k = \mathcal{O}(\log(1/\varepsilon)/\varepsilon)$ hash functions. To give an example, while $k = 3$ hash functions suffice for $\varepsilon = 0.2$ (because $80\% < c^{*}_{3,1} \approx 92\%$), the analysis of [16] requires $k \geq 50$ hash

Table 2 Guarantees offered by analyses on cuckoo table insertions. The motivation for the third line is that any load factor $< 50\%$ can be achieved with $k = 2$.

	[20]	[12]	[17, 15]	[16]	**new**
algorithm	offline construction	BFS	RW	RW′	RW
(expected amortised) insertion time	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\log^{\mathcal{O}(1)}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
least k for load factor $> 50\%$	3	≥ 10	3	≥ 12	3
supports load factor $1 - \varepsilon$ for large k	✓	✓	✓	✓	✗
supports load factor $c^{*}_{k,1} - \varepsilon$	✓	✗	✓	✗	✗
supports deletions	-	✓	✓	✗	✗

functions for that ε. Even if the analysis can be tuned to some degree (which seems probable), useful guarantees for practically relevant k would likely remain out of reach. We remark that they use a variant RW′ of RW where a key x searches all its buckets $h_1(x), \dots, h_k(x)$ for a free space and only moves to a random bucket if all are full.

[18], not shown in the table, considers $k = 2$ hash functions, buckets of size $\ell \geq 2$ and random walk insertion. The result resembles [16] in its merits and downsides: Expected insertion time of $\mathcal{O}(1)$ is supported at any load factor $1 - \varepsilon$, provided that ℓ is large enough; the value $\ell = \ell(\varepsilon)$ required in the analysis is exponentially larger than what is needed for the existence of a placement; and meaningful guarantees for small values of ℓ seem out of reach.

This paper is the first to guarantee constant time insertions into cuckoo hash tables using $k \in \{3, \dots, 9\}$ hash functions. Like [16] our proof does not consider deletions. The main downside is that our analysis only works for $\frac{m}{n} < c^{\triangle}_{k,1} - \varepsilon$. Paradoxically, this means that the load factor supported by our analysis decreases when more hash functions are used (indeed, $c^{\triangle}_{k,1} \to 0$ for $k \to \infty$) and the supremum of supported load factors is $c^{\triangle}_{3,1} \approx 0.818$ for $k = 3$ hash functions. We mention a potential avenue for overcoming this problem in the conclusion.

Technical Overview. A central idea in our approach is to not count the number of evictions caused by a single insertion operation but to take the perspective of a single key and count how often it moves in the course of all m insertion operations combined.

Our proof is inspired by a simple observation: If a key x is assigned a bucket $h_i(x)$ that is assigned to no other key, then x is safely out of the way of other keys as soon as it has been placed in $h_i(x)$. In expectation, this happens after x has moved k times. A constant fraction of keys is "harmless" in this way. Moreover, there are keys y that are assigned a bucket $h_i(y)$ that is assigned to no other key, *except* for some harmless keys. It seems plausible that y, too, can quickly find a home in $h_i(y)$ and is only expected to be evicted from it a few times until the harmless keys live up to their name.

A formalisation attempt goes like this: Let F be an injective placement function assigning to each key x a bucket $F(x) \in \{h_1(x), \dots, h_k(x)\}$ (such an F exists whp for $c < c^*_{k,\ell} - \varepsilon$). We say a key x *depends* on a key y if $F(x) \in \{h_1(y), \dots, h_k(y)\}$, i.e. if the position designated for x is admissible for y. Let $D(x)$ be the set of all keys that x depends on. Finally, let $\text{moves}(x)$ be the total number of times that x moves during the insertion of all keys. We then have

$$\mathbb{E}[\text{moves}(x)] \leq k + \sum_{y \in D(x)} \mathbb{E}[\text{moves}(y)]. \tag{1}$$

The "k" is due to x being first placed in $F(x)$ after k moves in expectation. It can then only be evicted from $F(x)$ by a key from $D(x)$. Each movement of a key in $D(x)$ has a chance of $\frac{1}{k}$ to evict x from $F(x)$, causing k more moves of x in expectation until x is back in $F(x)$. Hence each move of a key from $D(x)$ can cause at most one move of x in expectation, as (1) suggests. The claim is even true in a more general context we call the *random eviction process* where in each round an adversary choses the key y to be moved among all keys not currently placed in their designated location $F(y)$.

As a way of bounding $\mathbb{E}[\text{moves}(x)]$, Equation (1) is hopelessly circular at first, but it is useful for specific F. Indeed, assume that the configuration of keys and buckets is *peelable*, i.e. for every subset X' of keys there is a bucket b^* assigned to only one key $x^* \in X'$. In that case, we can iteratively construct F, always picking such a pair (x^*, b^*) uniformly at random, setting $F(x^*) = b^*$ and removing x^* from further consideration. This yields an

acyclic dependence relation and an acyclic *dependence graph* (the directed graph with one vertex for each key that has the dependence relation as its edge relation). We can then upper bound $\mathbb{E}[\text{moves}(x)]$ by a multiple of $\text{peel}_F(x)$, which is the number of paths in the dependence graph that start at x. Bounding the expected number of moves of all insertion operations combined by $\mathcal{O}(n)$ then amounts to bounding the total number of paths in the dependence graph by $\mathcal{O}(n)$.

The second part of our argument – contained in the full version of this paper – is intimately related to the analysis of 2-cores in random hypergraphs. We extend Janson and Luczak's "simple solution to the $(\ell+1)$-core problem" [19], which uses a random process embedded in continuous time where peeling is applied to the configuration model of a random hypergraph. We establish two guarantees concerning the peeling process.

- Firstly, the guarantee that during "early" rounds of peeling (when $\Omega(n)$ keys still remain) there are always $\Omega(n)$ candidate pairs (x^*, b^*) to choose from. Intuitively, this large number of choices for the peeling process makes it likely that the dependence graph becomes very "wide" with few long paths. For illustration (the formal argument works differently) assume the maximum path length is w with $w = \mathcal{O}(1)$. Since the indegree of the dependence graph is bounded by $k-1$ this gives a bound of $m \cdot (k-1)^w = \Theta(m)$ on the total number of paths as desired.
- Secondly, the technically demanding guarantee that in the "late" phase of peeling (when only $o(n)$ keys remain) almost all buckets have at most one remaining key assigned to them. Most steps of the peeling process will then not create further edges in the dependence graph. This implies that for each of the paths that already exist in the dependence graph less than one additional path is created in future rounds.

Outline. The rest of this paper is devoted to proving Theorem 1. We introduce two notable auxiliary concepts we call the *random eviction process* (REP) and *peeling numbers*. We reduce Theorem 1 to a claim about REP (Section 2) and reduce this claim to an upper bound on peeling numbers (Section 3). The remaining technical content is found in the full version of this paper. A deep dive into hypergraph peeling is required (full version Section 4). We then establish the upper bound on peeling numbers by counting paths in the dependence graph (full version Section 5).

2 Orientations, Peeling and the Random Eviction Process

In this section, we introduce the *random eviction process* (REP), which generalises sequential RW insertions, and formulate a claim on REP' that implies Theorem 1.

From Hashing to Hypergraphs. A well-subscribed model for cuckoo hashing involves hypergraph terminology. The set of buckets corresponds to the set V of vertices and each key x corresponds to the hyperedge $\{h_1(x), \dots, h_k(x)\}$ in the set E of hyperedges. The task of placing all keys then becomes the task of *orienting* $H = (V, E)$ as explained below.

Under the *simple uniform hashing assumption*, the distribution of $H = H_{n,m,k}$ is simple: Each of the km incidences of the m hyperedges are chosen independently and uniformly at random from V. Formally this means that hyperedges are multisets of size k, possibly containing multiple copies of the same vertex (though in expectation only $\mathcal{O}(1)$ do) and E is a multiset possibly containing identical hyperedges (though whp E does not). This issue complicates a few definitions but does not cause any real trouble.

```
1 Algorithm REP(V, E):
2     f ← {(e, ⊥) | e ∈ E} // i.e. f ≡ ⊥
3     while ∃e ∈ E : f(e) = ⊥ do
4         pick such an e arbitrarily
5         pick a random v ∈ e
6         if ∃e′ ≠ e : f(e′) = v then
7             f(e′) ← ⊥
8         f(e) ← v
```

```
1 Algorithm REP'(V, E, F):
2     f ← {(e, ⊥) | e ∈ E}
3     while ∃e ∈ E : f(e) ≠ F(e) do
4         pick such an e arbitrarily
5         pick a random v ∈ e
6         if ∃e′ ≠ e : f(e′) = v then
7             f(e′) ← ⊥
8         f(e) ← v
```

Figure 2 The random eviction process (REP) is a generalisation of sequential random walk insertion. A variant REP' only terminates when a specific target orientation $F : E \to V$ is reached. Changes are highlighted in bold.

Orientations and Peelings. A *partial orientation* of a hypergraph $H = (V, E)$ is a function $f : E \to V \cup \{\bot\}$ with $f(e) \in e \cup \{\bot\}$ for each $e \in E$ that is injective except for collisions on $\bot$. If $f(e) = \bot$ then we say that e is *unoriented*, otherwise e is *oriented* (to $f(e)$). We call f an *orientation* if all $e \in E$ are oriented.

We can try to construct an orientation F of H greedily by repeatedly selecting a vertex v of degree 1 arbitrarily as long as one such vertex exists, setting $F(e) = v$ for the unique hyperedge e incident to v, and removing e from H. We call the resulting partial orientation F a *peeling* of H. If H does not contain a subhypergraph of minimum degree at least 2 (i.e. when the 2-core of H is empty [26]) then F is an orientation and we say H is *peelable*. We call F a *random peeling* if the choice of v is made uniformly at random whenever there are several vertices of degree 1.

The Random Eviction Process. The *random eviction process* (REP), see Figure 2 (left), is run on a hypergraph $H = (V, E)$ and maintains a partial orientation f of H. The process continues in a sequence of rounds as long as unoriented hyperedges remain, possibly indefinitely. In each round, an unoriented hyperedge e is chosen and oriented to a random incident vertex. If a different hyperedge e' was oriented to that vertex, then this e' is *evicted*, i.e. becomes unoriented.

A variant of REP is the *random eviction process with target orientation* (REP'), see Figure 2 (right). It is run on a hypergraph H and an orientation F of H. REP' works just like REP, except that it terminates only when $f = F$ is reached, and in every round any hyperedge e with $f(e) \neq F(e)$ may be chosen. We claim:

▶ **Proposition 2.** *Let $k \in \mathbb{N}$ with $k \geq 3$ and $\varepsilon > 0$ be constants and $n, m \in \mathbb{N}$ with $\frac{m}{n} < c^{\triangle}_{k,1} - \varepsilon$. Conditioned on a high probability event, $H = H_{n,m,k}$ is peelable and the random peeling F of H satisfies the following. REP' with target orientation F and an arbitrary[1] policy for choosing e in line 4 terminates after $\mathcal{O}(n)$ rounds in expectation.*

Proposition 2 is proved in the following section. We now show that it implies Theorem 1.

[1] This allows these choices to be made *adversarially*. The adversary may know all about H and the state of the algorithm but cannot predict future random choices made in line 5.

Proof of Theorem 1. The case of m sequential insertions is equivalent to the case of m parallel insertions where the scheduler only assigns computation time to the thread of least index that has not yet terminated. It therefore suffices to prove **(ii)**, where the parallel case with *arbitrary* scheduling is considered.

We deal with m threads, each running RW, executed in an arbitrarily interleaved way. However, the only point where RW interacts with data visible to other threads is the swap, which is assumed to be atomic. A sufficiently general case is therefore one where the scheduler always picks an arbitrary thread that has not yet terminated and that thread is then allowed to run for one iteration of the loop. The correspondence between this process and REP should be clear: The scheduler's arbitrary choice of a thread implicitly chooses an unplaced key in that thread's local variable x, which is then placed into a random bucket, possibly evicting a different key. Likewise, REP arbitrarily chooses an unoriented hyperedge, which is then randomly oriented, possibly evicting another hyperedge.

For Proposition 2's claim on REP' to apply to RW, there are two differences to consider.

REP vs. REP'. Assume an adversary wants to *maximise* the expected running times of REP and REP' by making bad choices for e in line 4. Her job is *harder* for REP for two reasons: Firstly, the termination condition is strictly weaker, such that REP may terminate when REP' does not. Secondly, her choices for e are restricted to unoriented hyperedges, where REP' additionally permits oriented hyperedges with $f(e) \neq F(e)$.

Intuitively speaking, the relatively weaker adversary in REP means that the upper bound on expected running time in Proposition 2 carries over from REP' to REP. More formally, every policy P for line 4 of REP is also valid for REP' and under the natural coupling random coupling REP' with P takes always at least as long as REP with P.

REP vs. RW: i_{old}. In RW an evicted key is not allowed to immediately move back into the bucket i_{old} it was just evicted from. The intuition is that this avoids a needless back-and-forth that otherwise occurs in 1 out of every k evictions. However, the author is not aware of a simple proof that the use of i_{old} is an improvement. Instead, we will check that the relevant part of the argument (Lemma 3) works for both cases. ◀

3 Bounding the Number of Evictions using Peeling Numbers

We now introduce the concept of peeling numbers and bound the number of evictions occurring in REP' in terms of them. This proves Proposition 2 but leaves the task of bounding peeling numbers for the full version of this paper.

Direct Dependence and Numbers of Moves. Consider a peelable hypergraph $H = (V, E)$ and a peeling $F : E \to V$ of H. For $e \neq e' \in E$ we say that e *directly depends* on e' if $F(e) \in e'$. This implies that e is peeled after e', making the transitive closure of direct dependence an acyclic relation. We define $D(e) = D_F(e)$ as the set of all e' that e directly depends on, or more precisely: $D(e)$ is a multiset containing e' with the same multiplicity with which e' contains $F(e)$.

Now consider a run of REP' with target orientation F (and an arbitrary policy for line 4). For $e \in E$ let moves(e) be the number of times that e is selected in line 4 of REP' (this is one more than the number of times that e is evicted).

▶ **Lemma 3.** *For any $e \in E$ we have* $\mathbb{E}[\text{moves}(e)] \leq k + \sum_{e' \in D(e)} \mathbb{E}[\text{moves}(e')]$.

Proof. For clarity, we ignore complications that are due to multisets at first. Let m_1 be the number of times that e moves until $f(e) = F(e)$ holds for the first time. Whenever e is selected to be moved, the chance to select $f(e) = F(e)$ is $\frac{1}{k}$, so clearly $\mathbb{E}[m_1] = k$. Afterwards,

e may not be selected anymore until evicted. Only hyperedges in $D(e)$ can evict e from $F(e)$ and when selected they do so with probability $\frac{1}{k}$, causing another k moves of e in expectation. It follows that $\mathbb{E}[m_+] = \mathbb{E}[m_D]$ where $m_+ := \text{moves}(e) - m_1$ and where m_D is the number of times that a hyperedge from $D(e)$ moves *while* $f(e) = F(e)$. The claim now follows from $m_D \leq \sum_{e' \in D(e)} \text{moves}(e')$ and linearity of expectation.

When $D(e)$ is a multiset the argument can be adapted: Whenever a hyperedge e' moves that is contained in $D(e)$ with multiplicity $a > 1$ it has an increased chance of $\frac{a}{k}$ to move to $F(e)$. But this is reflected in our bound since $\mathbb{E}[\text{moves}(e')]$ is counted a times.

Let us now consider a variant of the claim that incorporates the "i_{old}" feature of RW as promised in the proof of Theorem 1. In particular, a hyperedge never moves into the position it was last evicted from. We now have $\mathbb{E}[m_1] < k$ because all moves after the first move have an improved chance of $\frac{1}{k-1}$ to select $F(e)$. To compare $\mathbb{E}[m_+]$ and $\mathbb{E}[m_D]$, we can distinguish two kinds of moves. Concerning moves *away from* $F(e)$, m_D counts the same or one more compared to m_+. All other moves have a chance of $\frac{1}{k-1}$ to end in $F(e)$ and contribute the same amount to $\mathbb{E}[m_+]$ and $\mathbb{E}[m_D]$ as before. The same adaptation to multisets applies. ◀

The Peeling Number. We define the peeling number of $e \in E$ recursively as

$$\text{peel}(e) = \text{peel}_F(e) := \sum_{e' \in D_F(e)} (1 + \text{peel}(e')). \tag{2}$$

Peeling numbers are well-defined by acyclicity of direct dependence, the base case being $\text{peel}(e) = 0$ for any e with $D(e) = \varnothing$. The idea is that $\text{peel}(e)$ counts the number of hyperedges that e directly *or indirectly* depends on, in other words, those hyperedges e' that must be peeled before e can be peeled. However, some e' may be counted multiple times. The relevance of peeling numbers lies in the following lemma.

▶ **Lemma 4.** *Let H be a peelable hypergraph with a peeling F. Let R be the number of rounds until REP' with target orientation F terminates. We have $\mathbb{E}[R] \leq k \cdot (m + \sum_{e \in E} \text{peel}_F(e))$.*

Proof. For a single $e \in E$ we have $\mathbb{E}[\text{moves}(e)] \leq k \cdot (1 + \text{peel}(e))$ because

$$\mathbb{E}[\text{moves}(e)] \overset{Lem.3}{\leq} k + \sum_{e' \in D(e)} \mathbb{E}[\text{moves}(e')] \overset{\text{Induction}}{\leq} k + \sum_{e' \in D(e)} k \cdot (1 + \text{peel}(e'))$$
$$\overset{\text{Eq.(2)}}{=} k + k \cdot \text{peel}(e) = k \cdot (1 + \text{peel}(e)).$$

Since the total number R of rounds of REP' is the sum of all moves we conclude

$$\mathbb{E}[R] = \mathbb{E}\Big[\sum_{e \in E} \text{moves}(e)\Big] \leq \sum_{e \in E} k \cdot (1 + \text{peel}(e)) = k \cdot (m + \sum_{e \in E} \text{peel}(e)).$$ ◀

The remaining technical challenge is to bound the sum of all peeling numbers:

▶ **Proposition 5.** *Let H be as in Proposition 2. There is a high probability event $\mathcal{E}$ such that, conditioned on $\mathcal{E}$, H is peelable and the peeling numbers with respect to the random peeling F of H satisfy*

$$\mathbb{E}\Big[\sum_{e \in E} \text{peel}_F(e) \mid \mathcal{E}\Big] = \mathcal{O}(n).$$

A prove is found in the full version of this paper (Section 5) and requires a detailed analysis of the peeling process (Section 4). We conclude this extended abstract with showing how Proposition 5 implies Proposition 2.

Proof of Proposition 2. We take the opportunity to clarify the structure of our probability space. There are three random experiments, performed in sequence: First, we pick a random hypergraph H. Second, if H is peelable, we pick a random peeling F of H and observe the peeling numbers. Last, we execute REP'(H, F) and observe which moves are made. Note that the high probability event $\mathcal{E}$ from Proposition 5 only relates to the first two steps (it does not relate to any moves). Lemma 4 only relates to the last step and does not require H and F to be random. For the number R of rounds of REP' we obtain:

$$\mathbb{E}[R \mid \mathcal{E}] = \mathbb{E}\big[\mathbb{E}[R \mid H, F, \mathcal{E}] \mid \mathcal{E}\big] = \mathbb{E}\big[\mathbb{E}[R \mid H, F] \mid \mathcal{E}\big] \overset{\text{Lem. 4}}{\leq} \mathbb{E}\Big[k \cdot (m + \sum_{e \in E} \text{peel}(e)) \mid \mathcal{E}\Big]$$
$$= km + k \cdot \mathbb{E}\Big[\sum_{e \in E} \text{peel}(e) \mid \mathcal{E}\Big] \overset{\text{Prop. 5}}{=} km + k \cdot \mathcal{O}(n) = \mathcal{O}(n). \quad \blacktriangleleft$$

4 Conclusion and Future Work

This paper proves $\mathcal{O}(1)$ expected amortised running times for random walk insertions into cuckoo hash tables and is the first to yield meaningful results for small values of k such as $k = 3$. Our proof strategy is to link the number of times that a key x moves to the number of times that certain other keys move, where these other keys all preceed x in the peeling process. The main technical challenge (addressed in the full version of this paper) was to extend an existing analysis of this peeling process in order to obtain stronger guarantees on its late stages when a sublinear number of keys remain. There are several ways in which our result might be strengthened.

- While amortisation is central to our argument, it seems unlikely to be required for the result itself. Indeed, the plausible claim that the expected time for inserting the i-th key is monotonically increasing in i already implies a non-amortised result.
- To make the result more relevant to practitioners, it is natural to pursue a generalisation to long sequences of insertions *and deletions* and to buckets of size $\ell \geq 2$. The author suspects that the given argument can be correspondingly extended with moderate technical complications.

  If deletions are allowed then it seems natural to partition the data structure's lifetime into time slices of εn operations such that the set S_t of all keys present at some point during time slice t yields a peelable configuration whp. One would hope to conclude that $\mathcal{O}(n)$ evictions occur during the time slice in expectation whp. However, the peeling number of a key is no longer sufficient for bounding its expected number of moves for the simple reason that the key itself might be inserted and deleted frequently within the time slice.
- The most important goal, however, is to obtain a result that works up to the load threshold (for all $c < c^*_{k,1} - \varepsilon$), not just up to the peeling threshold (for $c < c^{\triangle}_{k,1} - \varepsilon$). There is at least one reason for optimism, namely the recent discovery of a variant of cuckoo hashing that raises the peeling threshold to the load threshold [31]. Briefly, a key's k hashes are randomly distributed in a random window of γn consecutive buckets. The peeling threshold of this variant is equal to $c^*_{k,\ell} - \varepsilon$ where $\varepsilon(\gamma)$ can be made arbitrarily small. However, when using this variant, an analysis can no longer rely on the configuration model due to a lack of symmetry between the vertices, meaning that even if the general idea is still sound, the proof would have to use different methods.

Regardless of whether such improvements are achievable, we believe this paper to be a promising step forward in the ongoing project of retrofitting the widespread use of cuckoo hash tables and cuckoo filters with strong theoretical guarantees.

References

1 Anders Aamand, Mathias Bæk Tejs Knudsen, and Mikkel Thorup. Power of d choices with simple tabulation. In *45th ICALP*, volume 107 of *LIPIcs*, pages 5:1–5:14, 2018. doi:10.4230/LIPIcs.ICALP.2018.5.

2 Martin Aumüller, Martin Dietzfelbinger, and Philipp Woelfel. Explicit and efficient hash families suffice for cuckoo hashing with a stash. *Algorithmica*, 70(3):428–456, 2014. doi:10.1007/s00453-013-9840-x.

3 Michael A. Bender, Tsvi Kopelowitz, William Kuszmaul, Ely Porat, and Clifford Stein. Incremental edge orientation in forests. In *29th ESA*, volume 204 of *LIPIcs*, pages 12:1–12:18, 2021. doi:10.4230/LIPIcs.ESA.2021.12.

4 Julie Anne Cain, Peter Sanders, and Nicholas C. Wormald. The random graph threshold for k-orientiability and a fast algorithm for optimal multiple-choice allocation. In *Proc. 18th SODA*, pages 469–476, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283433.

5 Colin Cooper. The cores of random hypergraphs with a given degree sequence. *Random Struct. Algorithms*, 25(4):353–375, 2004. doi:10.1002/rsa.20040.

6 Martin Dietzfelbinger, Andreas Goerdt, Michael Mitzenmacher, Andrea Montanari, Rasmus Pagh, and Michael Rink. Tight thresholds for cuckoo hashing via XORSAT. In *Proc. 37th ICALP (1)*, pages 213–225, 2010. doi:10.1007/978-3-642-14165-2_19.

7 Martin Dietzfelbinger and Christoph Weidling. Balanced allocation and dictionaries with tightly packed constant size bins. *Theor. Comput. Sci.*, 380(1-2):47–68, 2007. doi:10.1016/j.tcs.2007.02.054.

8 David Eppstein. Cuckoo filter: Simplification and analysis. In *Proc. 15th SWAT*, pages 8:1–8:12, 2016. doi:10.4230/LIPIcs.SWAT.2016.8.

9 Bin Fan, David G. Andersen, and Michael Kaminsky. Cuckoo filter: Better than Bloom. *;login:*, 38(4), 2013. URL: https://www.usenix.org/publications/login/august-2013-volume-38-number-4/cuckoo-filter-better-bloom.

10 Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proc. 10th CoNEXT*, pages 75–88, 2014. doi:10.1145/2674005.2674994.

11 Daniel Fernholz and Vijaya Ramachandran. The k-orientability thresholds for $g_{n,p}$. In *Proc. 18th SODA*, pages 459–468, 2007. URL: http://dl.acm.org/citation.cfm?id=1283383.1283432.

12 Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. *Theory Comput. Syst.*, 38(2):229–248, 2005. doi:10.1007/s00224-004-1195-x.

13 Nikolaos Fountoulakis, Megha Khosla, and Konstantinos Panagiotou. The multiple-orientability thresholds for random hypergraphs. *Combinatorics, Probability & Computing*, 25(6):870–908, 2016. doi:10.1017/S0963548315000334.

14 Nikolaos Fountoulakis and Konstantinos Panagiotou. Sharp load thresholds for cuckoo hashing. *Random Struct. Algorithms*, 41(3):306–333, 2012. doi:10.1002/rsa.20426.

15 Nikolaos Fountoulakis, Konstantinos Panagiotou, and Angelika Steger. On the insertion time of cuckoo hashing. *SIAM J. Comput.*, 42(6):2156–2181, 2013. doi:10.1137/100797503.

16 Alan M. Frieze and Tony Johansson. On the insertion time of random walk cuckoo hashing. *Random Struct. Algorithms*, 54(4):721–729, 2019. doi:10.1002/rsa.20808.

17 Alan M. Frieze, Páll Melsted, and Michael Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM J. Comput.*, 40(2):291–308, 2011. doi:10.1137/090770928.

18 Alan M. Frieze and Samantha Petti. Balanced allocation through random walk. *Inf. Process. Lett.*, 131:39–43, 2018. doi:10.1016/j.ipl.2017.11.010.

19 Svante Janson and Malwina J. Luczak. A simple solution to the k-core problem. *Random Struct. Algorithms*, 30(1-2):50–62, 2007. doi:10.1002/rsa.20147.

20 Megha Khosla. Balls into bins made faster. In *Proc. 21st ESA*, pages 601–612, 2013. doi:10.1007/978-3-642-40450-4_51.

21 Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, 2009. doi:10.1137/080728743.

22 Mathieu Leconte. Double hashing thresholds via local weak convergence. In *Proc. 51st Allerton*, pages 131–137, 2013. doi:10.1109/Allerton.2013.6736515.

23 Eric Lehman and Rina Panigrahy. 3.5-way cuckoo hashing for the price of 2-and-a-bit. In *Proc. 17th ESA*, pages 671–681, 2009. doi:10.1007/978-3-642-04128-0_60.

24 Marc Lelarge. A new approach to the orientation of random hypergraphs. In *Proc. 23rd SODA*, pages 251–264. SIAM, 2012. doi:10.1137/1.9781611973099.23.

25 Michael Mitzenmacher and Justin Thaler. Peeling arguments and double hashing. In *Proc. 50th Allerton*, pages 1118–1125, 2012. doi:10.1109/Allerton.2012.6483344.

26 Michael Molloy. Cores in random hypergraphs and Boolean formulas. *Random Struct. Algorithms*, 27(1):124–135, 2005. doi:10.1002/rsa.20061.

27 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004. doi:10.1016/j.jalgor.2003.12.002.

28 Boris Pittel, Joel Spencer, and Nicholas C. Wormald. Sudden emergence of a giant k-core in a random graph. *J. Comb. Theory, Ser. B*, 67(1):111–151, 1996. doi:10.1006/jctb.1996.0036.

29 Peter Sanders, Kurt Mehlhorn, Martin Dietzfelbinger, and Roman Dementiev. *Sequential and Parallel Algorithms and Data Structures - The Basic Toolbox*. Springer, 2019. doi:10.1007/978-3-030-25209-0.

30 Stefan Walzer. Load thresholds for cuckoo hashing with overlapping blocks. In *Proc. 45th ICALP*, pages 102:1–102:10, 2018. doi:10.4230/LIPIcs.ICALP.2018.102.

31 Stefan Walzer. Peeling close to the orientability threshold: Spatial coupling in hashing-based data structures. In *Proc. 32nd SODA*, pages 2194–2211. SIAM, 2021. doi:10.1137/1.9781611976465.131.