A Faster Algorithm for Cuckoo Insertion and Bipartite Matching in Large Graphs

Avishek Anand · Megha Khosla

Abstract Hash tables are ubiquitous in computer science for their use in storing large sets of data. Cuckoo hashing is an efficient technique of creating large hash tables with high space utilization and offers a guaranteed constant access time. We are given n locations and m items. Each item has to be placed in one of the $k \geq 2$ locations chosen by k random hash functions. By allowing more than one choice for a single item, cuckoo hashing resembles multiple choice allocations schemes. In addition it supports dynamically changing the location of an item among its possible locations. We propose and analyze an insertion algorithm for cuckoo hashing that runs in *linear time* with high probability and in expectation. Previous work on total allocation time has analysed breadth first search, and it was shown to be linear only in *expectation*. Our algorithm finds an assignment (with probability 1) whenever it exists. In contrast, the other known insertion method also called *random walk insertion* may run indefinitely even for a solvable instance. We present experimental results comparing the performance of our algorithm with the random walk method.

As a corollary we obtain a linear time algorithm(with high probability and in expectation) for finding perfect matchings in a special class of sparse random bipartite graphs. We also perform experiments for finding maximum matchings in general large bipartite graphs and report an order of magnitude improvement in the running time as compared to Hopkraft Karp matching algorithm.

F. Author
first address
Tel.: +123-45-678910
Fax: +123-45-678910
E-mail: fauthor@example.com

S. Author
second address

1 Introduction

In computer science, hash table [2] is a data structure that maps items (keys) to locations (values) using a hash function. More precisely, given a universe U of items and a hash table H of size $n \in \mathbb{N}$, a hash function $h : U \to \{1, 2, \ldots, n\}$ maps the items from U to the n positions on the table. Ideally, the hash function should assign to each possible item to a unique location, but this ideal is rarely achievable in practice. Two or more items could be mapped to the same location resulting in a collision. In this work we deal with a collision resolution technique known as *cuckoo hashing*. Cuckoo hashing was first proposed by Pagh and Rodler in [14]. We are interested in a generalization of the original idea (see [4]) where we are given a table with n locations, and we assume each location can hold a single item. Each item chooses randomly $k \geq 3$ locations (using k random hash functions) and has to be placed in one of them. Formally speaking we are given $k \geq 3$ hash functions $h_1, ..., h_k$ that each maps an element $x \in U$ to a position in the table H. Moreover we assume that $h_1, ..., h_k$ are truly independent and random hash functions. We refer the reader to [13,3] (and references therein) for justification of this idealized assumption. Other variations of cuckoo hashing are considered in for example [1,11].

Cuckoo hashing resembles multiple choice allocations schemes in the sense that it allows more than one choice for a single item. In addition it supports dynamically changing the location of an item among its possible locations during insertion. The insertion procedure in cuckoo hashing goes as follows. Assume that p items have been inserted, each of them having made their k random choices on the hash table, and we are about to insert the $p+1$st item. This item selects its k random locations from the hash table and is assigned to one of them. But this location might already be occupied by a previously inserted item. In that case, the previous item is kicked out and is assigned to one of the other $k - 1$ selected locations. In turn, this position might be occupied by another item, which is kicked out and goes to one of the remaining $k - 1$ chosen locations. This process may be repeated indefinitely or until a free loction is found.

We model cuckoo hashing by a directed graph $G = (V, E)$ such that the set of vertices $V = \{v_1, v_2, \ldots, v_n\}$ corresponds to locations on the hash table. We say a vertex is *occupied* if there is an item assigned to the corresponding location, otherwise it is *free*. Let $\mathcal{I}$ be the set of m items. We represent each item $x \in \mathcal{I}$ as a tuple of its k chosen vertices (locations), for example, $x = (v_{x_1}, v_{x_2}, \ldots, v_{x_k})$. A directed edge $e = (v_i, v_j) \in E$ if and only if there exists an item $y \in \mathcal{I}$ so that the following two conditions hold, (i) $v_i, v_j \in y$, and (ii) v_i is occupied by y. Note that a vertex with outdegree 0 is a free vertex. We denote the set of free vertices by F and the distance of any vertex v from some vertex in F by $d(v, F)$. Since G represents an allocation we call G an *allocation graph*.

Now assume that in the cuckoo insertion procedure, at some instance an item z arrives such that all its k choices are occupied. Let $v_j \in z$ be the vertex chosen to place item z. The following are the main observations.

1. The necessary condition for item z to be successfully inserted at v_{z_j} is the existence of a path from v_{z_j} to F. This condition remains satisfied as long as some allocation is possible.
2. The procedure will stop in the minimum number of steps if for all $v_{z_i} \in z$ the distance $d(v_{z_j}, F) \leq d(v_{z_i}, F)$.

With respect to our first observation, a natural question would be the following. We are given a set of m items and n locations such that each item picks $k \geq 3$ locations at random. Is it possible to place each of the items into one of their chosen locations such that each location holds at most one item? From [12,?,7] we know that there exists a critical size $c_k^* n$ such that if $m < c_k^* n$ then such an allocation is possible with high probability, otherwise this is not the case.

Theorem 1 *For integers $k \geq 3$ let ξ^* be the unique solution of the equation*

$$k = \frac{\xi(1 - e^{-\xi})}{1 - e^{-\xi} - \xi e^{-\xi}}. \tag{1}$$

Let $c_k^* = \frac{\xi^*}{k(1 - e^{-\xi^*})^{k-1}}$. Then

$$\mathbb{P}\left(\text{allocation of } m = \lfloor cn \rfloor \text{ items to } n \text{ locations is possible}\right) \stackrel{(n \to \infty)}{=} \begin{cases} 0, & \text{if } c > c_k^* \\ 1, & \text{if } c < c_k^* \end{cases}. \tag{2}$$

The proof of the above theorem is non-constructive, i.e., it does not give us an algorithm to find such an allocation. In this work we deal with the algorithmic issues and propose an algorithm which takes linear time with high probability to find the optimal allocation.

Our second observation suggests that the insertion time in the cuckoo hashing depends on the selection of the location, which we make for each assignment, from among the k possible locations. One can in principle use breadth first search to always make assignments over the shortest path (in the allocation graph). But this method is inefficient and expensive to perform for each item. One can also select uniformly at random a location from the available locations. This resembles a random walk on the locations of the table and is called the *random walk insertion*. In [6,8] the authors analyzed the random walk insertion method and gave a polylogarithmic bound (with high probability) on the maximum insertion time, i.e., the maximum time it can take to insert a single item.

1.1 More on Related Work

The allocation problem in cuckoo hashing can also be phrased in terms of *orientation of graphs* or more generally *orientations of k-uniform hypergraphs*. The n locations are represented as vertices and each of the m items form an edge with its k-vertices representing the k random choices of the item. In fact, this is a random (multi)hypergraph $H_{n,m,k}^*$ (or random (multi)graph $G_{n,m}^*$ for $k = 2$) with n vertices and m edges where each edge is drawn uniformly at random (with replacement) from the set of all k-multisubsets of the vertex set. An ℓ-orientation of a graph then amounts to a mapping of each edge to one of its vertices such that no vertex receives more than ℓ edges. ℓ is also called the maximum load capacity. In our algorithm, we focus on $\ell = 1$. Here, we give an overview of existing work for general ℓ for completeness.

For the case $k = 2$, several allocation algorithms and their analysis are closely connected to the cores of the associated graph. The ℓ core of a graph is the maximum vertex induced subgraph with minimum degree at least ℓ. For example, Czumaj and Stemann [**?**] gave a linear time algorithm achieving maximum load $O(m/n)$ based on computation of all cores. The main idea was to repeatedly choose a vertex v with minimum degree and remove it from the graph, and assigning all its incident edges (balls) to vertex (bin) v.

Cain, Sanders, and Wormald [**?**] used a variation of the above approach and gave a linear time algorithm for computing an optimal allocation (asymptotically almost surely). Their algorithm first guesses the optimal load among the two likely values values ($\lceil m/n \rceil$ or $\lceil m/n \rceil + 1$). The algorithm starts with load value say $\ell = \lceil m/n \rceil$. Each time a vertex with degree at most ℓ and its incident edges are assigned to v. The above rule also called the mindegree rule will first reduce the graph to its $\ell + 1$ core. After that some edge (u, v) is picked according to some priority rule and assigned to one of its vertices. Again the mindegree rule is applied with respect to some conditions. In case the algorithm fails it is repeated after incrementing the load value.

Fernholz and Ramachandran [**?**] used a different approach of dealing with the vertices with degree greater than the maximum load. Their algorithm also called *excess degree reduction* (EDR) always chooses a vertex with minimum degree, d. If $d < \ell$ then this vertex is assigned all its incident edges and is removed from the graph. In case $d > 2\ell$ the algorithm fails. Otherwise, EDR replaces $d - \ell$ paths of the form (u, v, w) by bypass edges (u, w) and then orients all remaining edges ($\leq \ell$) incident to v towards v.
Optimal allocations can also be computed in polynomial time using maximum flow computations and with high probability achieve a maximum load of $\lceil m/n \rceil$ or $\lceil m/n \rceil + 1$ [**?**].

Notations. Throughout the paper we use the following notations. We denote the set of integers $\{1, 2, \ldots, n\}$ by $[n]$. Let $V = \{v_1, v_2, \ldots, v_n\}$ be the set of n vertices representing the locations of the hash table. For an allocation graph $G = (V, E)$ and any two vertices $v_i, v_j \in V$, the shortest distance between v_i and v_j is denoted by $d(v_i, v_j)$. We denote the set of free vertices by F. We

denote the shortest distance of a vertex $v_i \in V$ to any set of vertices say S by $d(v_i, S)$ which is defined as

$$d(v_i, S) := \min_{v_j \in S} d(v_i, v_j).$$

We use R to denote the set of vertices furthest from F, i.e.,

$$R := \{v_i \in V | d(v_i, F) \geq \max_{v_j \in V} d(v_j, F)\}.$$

For some integer $t \in [n]$ and the subset of vertex set $V' \subset V$ let $N_t(v_i)$ and $N_t(V')$ denote the set of vertices at distance at most t from the vertex $v_i \in V$ and the set V'. Mathematically,

$$N_t(v_i) := \{v_j \in V \mid d(v_i, v_j) \leq t\}$$

and

$$N_t(V') := \{v_j \in V \mid d(v_i, V') \leq t\}.$$

1.2 Our Contribution

Our aim here is to minimize the total insertion time in cuckoo hashing, thereby minimizing the total time required to construct the hash table. We propose a deterministic strategy of how to select a vertex for placing an item when all its choices are occupied. We assign to each vertex $v_i \in V$ an integer label, $L(v_i)$. Initially all vertices have 0 as their labels. Note that at this stage, for all $j \in [n]$, $L(v_j) = d(v_j, F)$, i.e., the labels of all vertices represent their shortest distances from F. When an item x appears, it chooses the vertex with the least label from among its k choices. If the vertex is free, the item is placed on it. Otherwise, the previous item is kicked out . The label of the location is then updated and set to one more than the minimum label of the remaining $k - 1$ choices of the item x. The kicked out item chooses the location with minimum label from its k choices and the above procedure is repeated till an empty location is found. Note that to maintain the labels of the vertices as their shortest distances from F we would require to update labels of the neighbors of the affected vertex and the labels of their neighbors and so on. This corresponds to performing a breadth first search (bfs) starting from the affected vertex. We avoid the bfs and perform only local updations. Therefore, we also call our method as *local search allocation*.

Previous work [4] on total allocation time has analysed breadth first search, and it was shown to be linear only in *expectation*. The local search allocation method requires linear time with probability $1 - o(1)$ and in expectation to find an allocation. We now state our main result.

Theorem 2 *Let $k \geq 3$. For any fixed $\varepsilon > 0$, set $m = (1 - \varepsilon)c_k^* n$. Assume that each of the m items chooses k random locations (using k random hash functions) from a table with n locations. Then for some $\zeta > 0$ with probability $1 - n^{-\zeta}$, LSA finds an allocation of these items (such that no location holds*

more than one item) in time O(n). Moreover the expected running time of LSA is also O(n).

We prove the above theorem in two steps. First we show that the algorithm is correct and finds an allocation in polynomial time. To this end we prove that, at any instance, label of a vertex is at most its distance from the set of free vertices. Therefore, no vertex can have a label greater than n. This would imply that the algorithm could not run indefinitely and would stop after making at most n changes at each location. We then show that the local search insertion method will find an allocation in a time proportional to the sum of distances of the n vertices from F (in the resulting allocation graph). We then complete the proof by showing that (i) if for some $\varepsilon > 0$, $m = (1 - \varepsilon)c_k^*$ items are placed in n locations using k random hash functions for each item then the corresponding allocation graph has two special structural properties with probability $1 - o(1)$, and (ii) if the allocation graph has these two properties, then the sum of distances of its vertices from F is linear in n. In the next section we give a formal description of our algorithm and its analysis.

Cuckoo Hashing and Random Hypergraphs

As already mentioned we can model the allocation problem in cuckoo hashing as a hypergraph. Each location can be viewed as a vertex and each item as an edge. The k vertices of each edge represent its k-random choices. In fact, this is a random hypergraph with n vertices and m edges where each edge is drawn uniformly at random (with replacement) from the set of all k-multisubsets of the vertex set. Therefore, a proper allocation of items is possible if and only if the corresponding hypergraph is 1-*orientable*, i.e., if there is an assignment of each edge $e \in E$ to one of its vertices $v \in e$ such that each vertex is assigned at most one edge. We denote a random (multi)hypergraph with n vertices and m edges by $H_{n,m,k}$. We define the *density* of a hypergraph as the ratio of the number of edges to the number to its vertices.

We will need the following results from [6] about the expansion properties of a random hypergraph. In the analysis of LSA we would see that these properties help us to infer that the allocation graph expands considerably and the maximum label of any vertex there is $O(\log n)$.

Theorem 3 *Let for any fixed $\varepsilon > 0$, $m = (1 - \varepsilon)c_k^* n$. Then there exists a $\delta = \delta(\varepsilon, k)$ such that any subhypergraph of $H_{n,m,k}$ has density at most $(1 - \delta)$ with probability $1 - O(1/n)$.*

The proof of the following lemma is similar to that in [6]. The parameters here are adjusted to our requirements; so we present the proof for completeness.

Lemma 1 *Let $m < c_k^* n$ and $\alpha < 1/(k-1)$. Then for every integer s such that $1 \leq s \leq \alpha n$, there exists a constant $\zeta > 0$ such that the following holds with probability $1 - n^{-\zeta}$. The number of vertices spanned by any set of edges of size s in $H_{n,m,k}$ is greater than $\left(k - 1 - \frac{\log(k-1)e^k}{\log \frac{1}{\alpha(k-1)}} \right) s$.*

Proof Recall that each edge in $H_{n,m,k}$ is a multiset of size k. Therefore, the probability that an edge of $H_{n,m,k}$ is contained completely in a subset of size t of the vertex set is given by $\frac{t^k}{n^k}$. Thus the expected number of sets of edges of size s that span at most t vertices is at most $\binom{m}{s}\binom{n}{t}\left(\frac{t^k}{n^k}\right)^s$. Note that by the following approximation for factorials for positive integer a

$$\left(\frac{a}{e}\right)^a \sqrt{2\pi a} \le a! \le \left(\frac{a}{e}\right)^a e\sqrt{a},$$

we obtain for $0 < b < a$

$$\binom{a}{b} = \frac{a!}{b!(a-b)!} \le \frac{\left(\frac{a}{e}\right)^a e\sqrt{a}}{\left(\frac{b}{e}\right)^b \left(\frac{a-b}{e}\right)^{a-b} \sqrt{2\pi b}\sqrt{2\pi(a-b)}} = \frac{e}{2\pi} \cdot \left(1 - \frac{b}{a}\right)^{-(a-b+1/2)} \left(\frac{a}{b}\right)^b$$

$$< \frac{\exp\left(1 + b + \frac{b}{2a} + \frac{b^2}{2a^2} - \frac{b^3}{a^2}\right)}{2\pi} \left(\frac{a}{b}\right)^b < \frac{\exp\left(1 + \frac{b}{2a} - \frac{b^3}{2a^2}\right)}{2\pi} \left(\frac{ae}{b}\right)^b$$

$$< \frac{\exp(1.5)}{2\pi} \left(\frac{ae}{b}\right)^b < \left(\frac{ae}{b}\right)^b.$$

Using the above bounds for $m < c_k^* n$ and setting $t = (k-1-\delta_s)s$ we obtain

$$\binom{m}{s}\binom{n}{t}\left(\frac{t}{n}\right)^{ks} < \left(\frac{nc_k^* e}{s}\right)^s \left(\frac{ne}{t}\right)^t \cdot \left(\frac{t}{n}\right)^{ks} < \left(\frac{nc_k^* e}{s}\right)^s \left(\frac{ne}{t}\right)^t \cdot \left(\frac{t}{n}\right)^{ks}$$

$$= \left(\frac{nc_k^*}{s}\right)^s \left(\frac{n}{t}\right)^{t-ks} e^{t+s} = \left(\frac{nc_k^* e^{k-\delta_s}}{s}\right)^s \left(\frac{n}{(k-1-\delta_s)s}\right)^{-(1+\delta_s)s}$$

$$< \left(\frac{nc_k^*}{s}\right)^s \left(\frac{n}{(k-1)s}\right)^{-(1+\delta_s)s} e^{ks}$$

$$= \left(\left(\frac{n}{(k-1)s}\right)^{-\delta_s} \cdot (k-1)e^k c_k^*\right)^s.$$

Moreover from [5] we know that $c_k^* < 1$. Let β be such that $(1+\beta)c_k^* = 1$. Setting $\delta_s = \log(k-1)e^k / \log \frac{n}{s(k-1)}$ we obtain

$$\left(\left(\frac{n}{(k-1)s}\right)^{-\delta_s} \cdot (k-1)e^k c_k^*\right)^s = (1+\beta)^{-s}.$$

Therefore, for $\delta_s = 1 + \ln_{k-1} e^k / \ln_{k-1} \frac{n}{s} - 1$ and $\alpha < 1/(k-1)$, the probability that there exists a set of edges of size s, where $n^{1/2} \le s \le \alpha n$, spanning at most $(k-1-\delta_s)s$ vertices is $O((1+\beta)^{-n^{1/2}}) = e^{-O(n^{1/2})}$

Note that for $n^{1/2} \le s \le \alpha n$, $\delta_s < 1 + \log_{k-1} e^k / (\log_{k-1} \frac{1}{\alpha} - 1)$. For the case $1 \le s < n^{1/2}$, we substitute $\delta_s = \log(k-1)e^k / (\log \frac{1}{\alpha(k-1)} - 1)$. Then the expected number of sets of edges of size s spanning at most $(k-1-\delta_s)s$ vertices is at most

$$\left(\left(\frac{(k-1)n^{1/2}}{n}\right)^{\frac{\log(k-1)e^k}{\log \frac{1}{\alpha(k-1)} - 1}} \cdot ((k-1)e^k)\right)^s.$$

Therefore for large n the probability that there exists a set of edges of size $1 \leq s < n^{1/2}$ spanning at most $\left(k - 1 - \frac{\log(k-1)e^k}{\log \frac{1}{\alpha(k-1)}} \right) s$ vertices is at most $o(n^{-1/2})$, which completes the proof .

2 Local Search Insertion and its Analysis

Assume that we are given items in an online fashion, i.e., each item chooses its k random locations whenever it appears. Moreover, items appear in an arbitrary order. The insertion using local search method goes as follows. For each vertex $v \in V$ we maintain a label. Initially each vertex is assigned a label 0. To assign an item x at time t we select one of its chosen vertices v such that its label is minimum and assign x to v. We assign a new label to v which is one more than the minimum label of the remaining $k - 1$ choices of x. However, v might have already been occupied by a previously assigned item i'. In that case we kick out y and repeat the above procedure. Let $\mathbf{L} = \{L(v_1), \ldots, L(v_n)\}$ and $\mathbf{T} = \{T(v_1), \ldots, T(v_n)\}$ where $L(v_i)$ denotes the label of vertex v_i and $T(v_i)$ denotes the item assigned to vertex v_i. We initialize $\mathbf{L}$ with all 0s , i.e., all vertices are free. We then use Algorithm 1 to assign an arbitrary item when it appears. In the next subsection we first prove the correctness of the algorithm,

---
Algorithm 1 AssignItem $(x, \mathbf{L}, \mathbf{T})$

1: Choose a vertex v among the k choices of x with minimum label $L(v)$.
2: **if** $(L(v) >= n - 1)$ **then**
3:     **EXIT**   ▷**Allocation does not exist**
4: **else**
5:     $L(v) \leftarrow 1 + \min\left(L(u) | u \neq v \text{ and } u \in x\right)$
6:     **if** $(T(v) \neq \emptyset)$ **then**
7:         $y \leftarrow T(v)$ ▷**Move that replaces an item**
8:         $T(v) \leftarrow x$
9:         **CALL** AssignBall$(y, \mathbf{L}, \mathbf{T})$
10:     **else**
11:         $T(v) \leftarrow x$ ▷**Move that places an item**

---

i.e, it finds an allocation in a finite number of steps whenever an allocation exists. We show that the algorithm takes a maximum of $O(n^2)$ time before it obtains an mapping for each item. We then proceed to give a stronger bound on the running time.

2.1 Labels and the Shortest Distances

We need some additional notation. In what follows a *move* denotes either placing an item in a free vertex or replacing a previously allocated item. Let M be the total number of moves performed by the algorithm. For $p \in [M]$ we use $L_p(v)$ to denote the label of vertex v at the end of the pth move.

Similarly we use F_p to denote the set of free vertices at the end of pth move. The corresponding allocation graph is denoted as $G_p = (V, E_p)$. We need the following proposition.

Proposition 1 *For all $p \in [M]$ and all $v \in V$, the shortest distance of v to F_p is at least the label of v, i.e., $d(v, F_p) \geq L_p(v)$.*

Proof We first note that the label of a free vertex always remain 0, i.e.,

$$\forall p \in [M], \forall w \in F_p, \qquad L_p(w) = 0. \tag{3}$$

We will now show that throughout the algorithm the label of a vertex is at most one more than the label of any of its immediate neighbors (neighbors at distance 1). More precisely,

$$\forall p \in [M], \forall (u, v) \in E_p, \qquad L_p(u) \leq L_p(v) + 1. \tag{4}$$

We prove (4) by induction on the number of moves performed by the algorithm. Initially when no ball has appeared all vertices have 0 as their labels. When the first ball is assigned, i.e., there is a single vertex say u such that $L_1(u) = 1$. Clearly, (4) holds after the first move. Assume that (4) holds after p moves.

For the $(p+1)$th move let $w \in V$ be some vertex which is assigned a ball x. Consider an edge $(u, v) \in E_p$ such that $u \neq w$ and $v \neq w$. Note that the labels of all vertices $v \in V \setminus w$ remain unchanged in the $(p + 1)$th move. Therefore by induction hypothesis, (4) is true for all edges which does not contain w. By Step 2 of Algorithm 1 the new label of w is one more than the minimum of the labels of its $k - 1$ neighbors, i.e,

$$L_{p+1}(w) = \min_{w' \in x \setminus w} L_{p+1}(w') + 1.$$

Therefore (4) holds for all edges originating from w. Now consider a vertex $u \in V$ such that $(u, w) \in E_p$. Now by induction hypothesis we have $L_{p+1}(u) = L_p(u) \leq L_p(w) + 1$. Note that the vertex w was chosen because it had the minimum label among the k possible choices for the ball x, i.e.,

$$L_p(w) \leq \min_{w' \in x} L_p(w') = \min_{w' \in x \setminus w} L_{p+1}(w') < L_{p+1}(w).$$

We therefore obtain $L_{p+1}(u) \leq L_p(w) + 1 < L_{p+1}(w) + 1$, thereby completing the induction step. We can now combine (3) and (4) to obtain the desired result. To see this, consider a vertex v at distance $s < n$ to a free vertex $f \in F_p$ such that s is also the shortest distance from v to F_p. By iteratively applying (4) we obtain $L_p(v) \leq s + L_p(f) = d(v, F_p)$, which completes the proof.

We know that whenever the algorithm visits a vertex, it increases its label by at least 1. Trivially the maximum distance of a vertex from a free vertex is $n - 1$ (if an allocation exists), and so is the maximum label. Therefore the algorithm will stop in at most $n(n - 1)$ steps, i.e., after visiting each vertex at most $n - 1$ times, which implies that the algorithm is correct and finds an allocation in $O(n^2)$ time. In the following we show that the total running time is proportional to the sum of labels of the n vertices.

Lemma 2 *Let* $\mathbf{L}^*$ *be the array of labels of the vertices after all items have been allocated using Algorithm 1. Then the total time required to find an allocation is* $O(\sum_{v \in V} L^*(v))$.

Proof Now each invocation of Algorithm 1 increases the label of the chosen vertex by at least 1. Therefore, if a vertex has a label ℓ at the end of the algorithm then it has been selected (for any move during the allocation process) at most ℓ times. Now the given number of items can be allocated in a time proportional to the number of steps required to obtain the array $\mathbf{L}^*$ (when the initial set consisted of all zeros) and hence is $O(\sum_{v \in V} L^*(v))$.

For notational convenience let $F := F_M$ and $G := G_M$ denote the set of free vertices and the allocation graph (respectively) at the end of the algorithm. By Proposition 1 we know that for each $v \in V$, $L^*(v) \leq d(v, F)$. Moreover, by Step 2 of Algorithm 1 the maximum value of a label is n. Thus the total sum of labels of all vertices is bounded as follows.

$$\sum_{v_i \in V} L^*(v_i)) \leq \min \left(\sum_{v_i \in V} d(v, F), n^2 \right).$$

So our aim now is to bound the shortest distances such that the sum of these is linear in the size of G. We accomplish this in the following section.

2.2 Bounding the Distances

To compute the desired sum, i.e., $\sum_{v_i \in V} d(v, F)$, we study the structure of the allocation graph. The following lemma states that, with probability $1 - o(1)$, a fraction of the vertices in the allocation graph are at a constant distance to the set of free vertices, F. This would imply that the contribution for the above sum made by these vertices is $O(n)$.

Lemma 3 *For any fixed* $\varepsilon > 0$, *let* $m = (1 - \varepsilon)c_k^* n$ *balls are assigned to* n *bins using* k *random choices for each ball. Then the corresponding allocation graph* $G = (V, E)$ *satisfies the following with probability* $1 - O(1/n)$: *for every* $\alpha > 0$ *there exist* $C = C(\alpha, \varepsilon) > 0$ *and a set* $S \subseteq V$ *of size at least* $(1 - \alpha)n$ *such that every vertex* $v \in S$ *satisfies* $d(v, F) \leq C$.

Proof We perform the following stripping process on G. We start with G and in each step remove all its free vertices and the edges they are contained in. Note that by removing the edges, we have removed the balls placed on the corresponding vertices, thereby creating a new set of free vertices. For step i of the stripping process, we denote the set of vertices by V_i and the set of free vertices by F_i and let G_i be the corresponding allocation graph. The number of occupied vertices in G_i is then equal to $|V_i| - |F_i|$. As each vertex holds at most one ball, the number of remaining balls is $|V_i| - |F_i|$.

Let $H = (V, E')$ be a k-uniform hypergraph with n vertices representing the locations and m edges representing the items. Each edge consists of k

vertices or k choices of the ball. Note that the number of occupied vertices in G is equal to the number of edges in H. Similarly G_i corresponds to a subgraph in H induced on the vertex set V_i. Let us denote it by $H[V_i]$. The number of occupied vertices in G_i, i.e. $|V_i| - |F_i|$, then is the number of edges in $H[V_i]$. By Theorem 3, with probability $1 - o(1)$ we have $|F_i| \geq \delta|V_i|$. Also by the stripping process we obtain $|V_{i+1}| = |V_i| - |F_i|$. We can therefore conclude that, with probability $1 - o(1)$, $|V_{i+1}| \leq (1 - \delta)|V_i|$. Therefore, after $t \geq 1$ iterations of removing free vertices we obtain $|V_t| \leq (1 - \delta)^t|V|$. We can now choose $t = \lceil \ln_{(1-\delta)} \alpha \rceil$ to deduce that $|V_t| < \alpha|V|$. We complete the proof by substituting $S = V \setminus V_t \geq (1 - \alpha)n$ and $C = \lceil \log_{(1-\delta)} \alpha \rceil$.

We remark that the above lemma has already been proved in [6] (in the hypergraph setting). A similar result has also been proved in [7] (in the bipartite matching setting) for $k \geq 8$. With respect to an allocation graph recall that we denote the set of vertices furthest from F by R. Also for an integer s, $N_s(R)$ denotes the set of vertices at distance at most s from R. The next lemma states that the neighborhood of R expands suitably with high probability. We remark that the estimate, for expansion factor, presented here is not the best possible but nevertheless suffices for our analysis.

Lemma 4 *For any fixed $\varepsilon > 0$, let $m = (1 - \varepsilon)c_k^* n$ balls are assigned to n bins using k random choices for each ball and $G = (V, E)$ be the corresponding allocation graph. Then for any $0 < \alpha < \frac{1}{k-1}$ and every integer s such that $1 \leq |N_s(R)| \leq \alpha n$, there exists a constant $\zeta > 0$ such that G satisfies the following with probability $1 - n^{-\zeta}$.*

$$|N_s(R)| > \left(k - 1 - \frac{\log e^k(k-1)}{\log \frac{1}{\alpha(k-1)}} \right) |N_{s-1}(R)|.$$

Proof Recall that in the allocation graph G, R is the set of vertices furthest from the set of free vertices. The set of vertices at distance at most s from R is denoted by $N_s(R)$. Note that each occupied vertex in G holds one ball. By construction of the allocation graph $N_s(R)$ is the set of vertices representing the choices of balls placed on vertices in $N_{s-1}(R)$. In the hypergraph setting where each ball corresponds to an edge, $|N_s(R)|$ is the number of vertices spanned by the set of edges of size $|N_{s-1}(R)|$. We can now use Lemma 1 to obtain the desired result.

We define $\mu := \log e^k(k-1)/\log(-\alpha(k-1))$. For some fixed $\gamma > 0$ we set

$$\alpha := \exp\left(\frac{-k}{k-2-\gamma} \right) (k-1)^{-1-\frac{-1}{k-2-\gamma}}, \tag{5}$$

which implies that $\mu = k - 2 - \gamma$.

The following corollary follows from the above two lemmas.

Corollary 1 *With high probability, the maximum label of any vertex in the allocation graph is $O(\log n)$.*

Proof Set α as in (5). Let d be the shortest distance of vertices in R to S. Then by Lemma 4 with high probability,

$$|N_d(R)| > \left(k - 1 - \frac{\log e^k(k-1)}{\log \frac{1}{\alpha(k-1)}} \right) |N_{d-1}(R)| = (1+\gamma)^d |R|,$$

which implies that $d < \log_{1+\gamma} \alpha n$. Note that the shortest distance of vertices in S to F is a constant $C(\alpha, \delta)$ for δ defined in Lemma 3. Moreover, by Proposition 1 the label of any vertex is upper bounded by its distance to the set of free vertices. Therefore, the label of any vertex v is such that $L(v) = O(\log_{1+\gamma} \alpha n)$.

We now prove our main theorem.

Proof (Proof of Theorem 2) Set α as in (5). Then by Lemma 3, with probability $1 - O(1/n)$, there exists a $C = C(\alpha, \varepsilon)$ and a set S such that $|S| \geq (1-\alpha)n$ and every vertex $v \in S$ satisfies $d(v, F) \leq C$. Let $T + 1$ be the maximum of the distances of vertices in R to S, i.e.,

$$T = \max_{v \in R} d(v, S) - 1.$$

Clearly the number of vertices at distance at most T from R is at most αn, i.e., $|N_T(R)| \leq \alpha n$. Moreover for all $t < T$, $|N_t(R)| < |N_T(R)|$. Then by Lemma 4, for all $t \leq T$ the following holds with high probability,

$$|N_{t+1}(R)| > (k - 1 - \delta) |N_t(R)|.$$

One can check that for $\gamma > 0$ and α as chosen above, $\delta < k - 2 - \gamma$. The total distance of all vertices from F is then given by

$$D = \sum_{v \in N_T(R)} d(v, F) + \sum_{v \in S} d(v, F).$$

As every vertex in S is at a constant distance from F, we obtain $\sum_{v \in S} d(v, F) = O(n)$. Note that for every $i > 0$, $|N_i(R)| - |N_{i-1}(R)|$ is the number of vertices at distance i from R. Therefore,

$$\sum_{v \in N_T(R)} d(v, F) = (T + C)|N_0(R)| + \sum_{i=1}^{T} (T + C - i)(|N_i(R)| - |N_{i-1}(R)|)$$

$$= (T + C)|N_0(R)| + \sum_{i=1}^{T} (T - i)(|N_i(R)| - |N_{i-1}(R)|) + C \sum_{i=1}^{T} (|N_i(R)| - |N_{i-1}(R)|)$$

$$= (T + C)|N_0(R)| + \sum_{i=1}^{T} (T - i)(|N_i(R)| - |N_{i-1}(R)|) + C(|N_T(R)| - |N_0(R)|)$$

$$= \sum_{i=1}^{T} \left((T - i)(|N_i(R)| - |N_{i-1}(R)|) + |N_0(R)| \right) + C \cdot |N_T(R)| = \sum_{i=0}^{T-1} |N_i(R)| + O(n).$$

Now with high probability, we have $|N_{T-j}(R)| < \frac{|N_T(R)|}{(k-1-\delta)^j}$. Therefore,

$$\sum_{i=0}^{T-1} |N_i(R))| < |N_T(R)| \sum_{j=1}^{T} \frac{1}{(k-1-\delta)^j} < |N_T(R)| \sum_{j=1}^{T} \frac{1}{(1+\gamma)^j} = O(n),$$

which completes the first part of the proof of Theorem 2. To bound the expected time, observe that for i such that $|N_i(R))| < n^{1/2}$,

$$E\left(\sum_i |N_i(R)|\right) < n^{1/2} \cdot \frac{(n^{1/2}+1)}{2} = O(n)$$

. For all other i using lemmas **??**

$$\sum_i |N_i(R)| < |N_T(R)| \sum_i \frac{1}{(k-1-\delta)^i}$$

We obtain the following corollary about maximum matchings in left regular random bipartite graphs. Recall that a bipartite graph $G = (L \cup R; E)$ is k-left regular if each vertex $v \in L$ has exactly k neighbors in R.

Corollary 2 *For $k \geq 3$ and c_k^* as defined in Theorem 1, let $G = (L \cup R; E)$ be a random k-left regular bipartite graph such that $|L|/|R| < c_k^*$. The local search allocation method obtains a maximum cardinality matching in G in time $O(|R|)$ with probability $1 - o(1)$.*

Proof We assign label 0 to each of the vertices in R initially. Each vertex in L can be considered as a ball and let R be the set of bins. The k random choices for $v \in L$ (ball) are the k random neighbors of v. We can now find a matching for each $v \in L$ by using Algorithm 1.

3 Experiments

In this section we discuss the performance of our proposed LSA algorithm on real-world large datasets. The rationale of our evaluation is two-fold. First, qualitatively establish the effectiveness of LSA in terms of abstract cost measures. Second, we would want to validate the performance of LSA in terms of wall-clock times on large real-world bi-partite graphs.

3.1 Performance on Random Graphs

We present some simulations to compare the performance of local search allocation with the random walk method which (to the best of our knowledge) is currently the state-of-art method and so far considered to be the fastest algorithm for the case $k \geq 3$. We recall that in the random walk method we choose a bin at random from among the k possible bins to place the ball. If the bin is not free, the previous ball is moved out. The moved out ball again

chooses a random bin from among its choices and the procedure goes on till an empty bin is found. In our experiments we consider $n \in [10^5, 5 \times 10^6]$ balls and $\lfloor cn \rfloor$ bins. The k random bins are chosen when the ball appears. All random numbers in our simulations are generated by *MT19937* generator of GNU Scientific Library [9].

Recall that a move is either placing an item at a free location or replacing it with other item. In Figure 1 we give a comparison of the total number of moves (averaged over 100 random instances) performed by local search and random walk methods for $k = 3$ and $k = 4$. Figure 2 compares the maximum number of moves (averaged over 100 random instances) for a single insertion performed by local search and random walk methods. Figure 3 shows a comparison when the number of balls are fixed and density (ratio of number of balls to that of bins) approaches the threshold density. Note that the time required to obtain an allocation by random walk or local search methods is directly proportional to the number of moves performed.
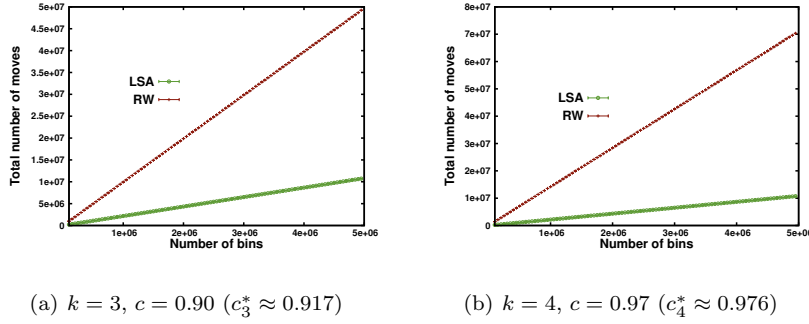

(a) $k = 3$, $c = 0.90$ ($c_3^* \approx 0.917$)   (b) $k = 4$, $c = 0.97$ ($c_4^* \approx 0.976$)

Fig. 1 Comparison of total number of moves performed by local search and random walk methods.

(a) $k = 3$, $c = 0.90$ ($c_3^* \approx 0.917$).   (b) $k = 4$, $c = 0.97$ ($c_4^* \approx 0.976$).
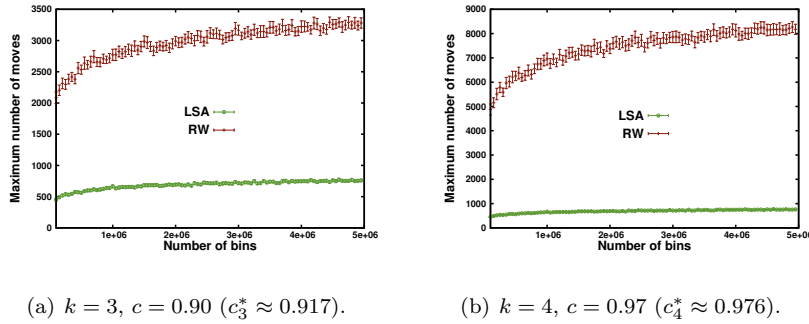
Fig. 2 Comparison of maximum number of moves performed by local search and random walk methods

(a) $k = 3$, $c \leq 0.915$ ($c_3^* \approx 0.917$)   (b) $k = 3$, $c \leq 0.915$ ($c_3^* \approx 0.917$)
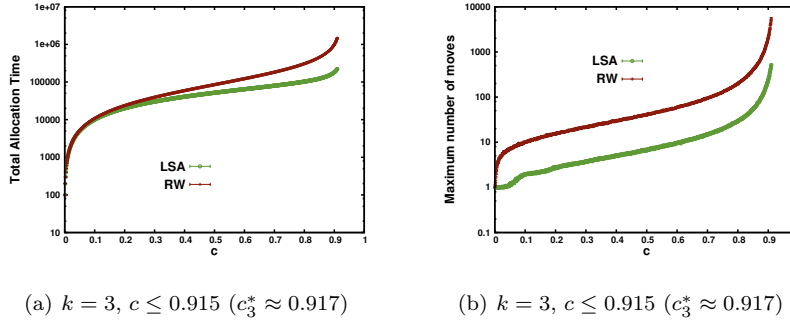
Fig. 3 Comparison of total number of moves and maximum number of moves (for fixed number of locations, $n = 10^5$) performed by local search and random walk methods when density c approaches c_k^*.
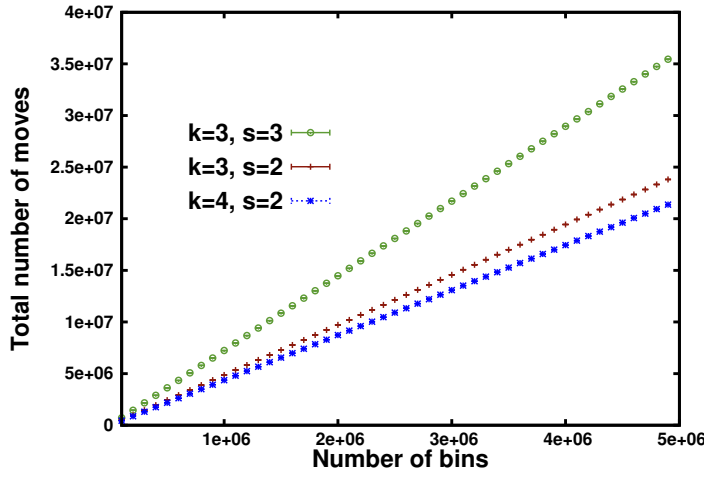

Fig. 4 Total number of moves for the case where bin capacities (maximum load, s) is greater than 1.

We remark that local search allocation has some additional cost, i.e., the extra space required to store the labels. Though this space is $O(n)$, local search allocation is still useful for the applications where the size of objects (representing the balls) to be allocated is much larger than the labels which are integers. Moreover, with high probability, the maximum label of any vertex is $O(\log n)$. Many integer compression methods [**?**] have been proposed for compressing small integers and can be potentially useful in our setting for further optimizations. Also in most of the load balancing problems, the speed of finding an assignment is a much desired and the most important requirement.

We also consider the case when each bin can hold more than one ball. To adapt LSA for this setting we make a small change, i.e., the label of a vertex (bin) stays 0 until it is fully filled. Algorithm **??** gives the modified procedure for the general bin capacities. Here BALLS(v) gives the number of balls already placed in v. Let the bin capacity or maximum load allowed be s. Figure 4 suggests that the total number of moves are linear in the number of bins for the cases $k = 3, 4$ where the maximum bin capacity is greater than 1.

	Limit on Moves	Wall-clock times	Result Size
LSA	1	12	1029449
	2	12	1080006
	4	12	1082199
	5	16	1082214
	10	15	1082214
	50	15	1082214
	100	15	1082214
	1000	15	1082214
	10000	27	1082214
	100000	136	1082214
	n	1887	1082214
Hopcroft-Karp		12605	1082214

Table 1 Performance of LSA on Delicious dataset. Time is measured in seconds.

3.2 Performance on Real-world graphs

Next, we compare our runtime performance to the optimal algorithm proposed by Hopcroft et al. [10]. In this experiment we want to study the effect of number of allowable moves on (a) the actual wall-clock times , (b) the result quality in terms of the size, or number of edges, of the final matching produced (refer Figure 1). We selected the following representative realworld datasets for our experiments:

– Delicious dataset : The Delicious dataset spans nine years from 2003 to 2011 and contain about 340 mio. bookmarks, 119 mio. unique URLs, 15 mio. tags and 2 mio. users [15]. Each bookmarked URL is time stamped and tagged with word descriptors. The nodes in one of the sets are URLs and in the other are its corresponding bookmarks.
– Yahoo Ad dataset : Computational advertising used bi-partite graph matching algorithms for its ad placement decisions.

We first observe that the optimal result in the Delicious dataset, i.e. 1082214, is already obtained when the limit on the allowable moves is only five and in 12 seconds. Interestingly, as we increase the limit on the allowable moves, the runtime does not change showing that only a small of defections are sufficient to arrive at an optimal result. However, at higher limits, indeed other

permutations are explored (in this case unsuccesfully) resulting in increased runtimes. In any case when the limit is set to n, that would guarantee optimality, we still perform an order of magnitude faster than the optimal algorithm of Hopcroft-Karp.

References

1. Y. Arbitman, M. Naor, and G. Segev. De-amortized cuckoo hashing: Provable worst-case performance and experimental results. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, pages 107–118, 2009.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
3. M. Dietzfelbinger and U. Schellbach. On risks of using cuckoo hashing with simple universal hash classes. In *Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '09, pages 795–804, 2009.
4. D. Fotakis, R. Pagh, P. Sanders, and P. Spirakis. Space efficient hash tables with worst case constant access time. In *STACS '03*, volume 2607 of *Lecture Notes in Computer Science*, pages 271–282. 2003.
5. N. Fountoulakis and K. Panagiotou. Sharp load thresholds for cuckoo hashing. *Random Structures & Algorithms*, 41(3):306–333, 2012.
6. N. Fountoulakis, K. Panagiotou, and A. Steger. On the insertion time of cuckoo hashing. *CoRR*, abs/1006.1231, 2010.
7. A. Frieze and P. Melsted. Maximum matchings in random bipartite graphs and the space utilization of cuckoo hash tables. *Random Structures & Algorithms*, 41(3):334–364, 2012.
8. A. Frieze, P. Melsted, and M. Mitzenmacher. An analysis of random-walk cuckoo hashing. *SIAM Journal on Computing*, 40(2):291–308, 2011.
9. M. Galassi, J. Davies, J. Theiler, B. Gough, G. Jungman, M. Booth, and F. Rossi. Gnu scientific library reference manual. *URL:http://www. gnu. org/software/gsl*, 2003.
10. John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.
11. A. Kirsch, M. Mitzenmacher, and U. Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM J. Comput.*, 39(4):1543–1561, December 2009.
12. M. Lelarge. A new approach to the orientation of random hypergraphs. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '12, pages 251–264, 2012.
13. M. Mitzenmacher and S. Vadhan. Why simple hash functions work: exploiting the entropy in a data stream. In *Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '08, pages 746–755, 2008.
14. R. Pagh and F. F. Rodler. Cuckoo hashing. In *ESA '01*, pages 121–133, 2001.
15. Arkaitz Zubiaga, Victor Fresno, Raquel Martinez, and Alberto Perez Garcia-Plaza. Harnessing folksonomies to produce a social classification of resources. *IEEE transactions on knowledge and data engineering*, 25(8):1801–1813, 2013.