CS 224: Advanced Algorithms Spring 2017

Lecture 5 — February 7, 2017

Prof. Jelani Nelson *Scribe: Rohit Agrawal*

1 Wrapping up approximate membership

Recall that the approximate membership problem for a set $S \subset [u]$ of size n entails answering queries of the form "is $x \in S$", subject to the constraint that if $x \in S$ the algorithm always returns yes, and if $x \notin S$ the algorithm returns yes with probability at most ϵ.

Last week we saw the Bloom filter [Blo70] which used space $O\big(n \lg(1/\epsilon)\big)$ bits and update/query time $O\big(\lg(1/\epsilon)\big)$. This leads to the natural question, **are these bounds optimal?**

This was partially answered in the affirmative by Carter, Floyd, Gill, Markowsky, and Wegman [CFG$^+$78], who showed that static approximate membership requires $\Omega\big(n \lg(1/\epsilon)\big)$ bits of space. They also gave a different data structure that used space $n \lg(1/\epsilon) + O(n)$ bits. This result was improved and partially dynamized by Arbitman, Naor, and Segev [ANS10] who gave another implementation matching the space of [CFG$^+$78], but also supporting insertion and $O(1)$ update/query. This result was extended to deletions by Pagh, Segev, and Wieder [PSW13] at the cost of achieving amortized and expected time bounds rather than worst-case.

2 Dynamic Dictionary and Cuckoo Hashing

The dynamic dictionary problem (without deletions) asks us to maintain the set $\mathcal{S} = S \times V$ with $S \subseteq [u]$ a set of keys, subject to the following operations:

- `insert`(x, v): add (x, v) to $\mathcal{S}$, removing any other mapping of x.
- `query`(x): returns the most recent v stored with a call to `insert`(x, v).

Cuckoo hashing, introduced by Pagh and Rodler [PR04], solves this problem. The structure stores two random hash functions $h, g : [u] \to [m]$ where $m = c \cdot n$ (with $c = 4$ sufficient), and an array $A[1 \ldots m]$ that stores items. The structure will maintain the invariant that the entry x will always be stored in either $A[h(x)]$ or $A[g(x)]$, so that queries will take constant time.

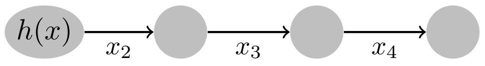
Hence, the main procedure of interest is insertion. Upon a call to `insert`(x, v), we consider four cases:

- $A[h(x)]$ is empty. This case is trivial, and we simply put (x, v) in $A[h(x)]$.
- In the other three cases, $A[h(x)]$ is already full. In all cases, we will set $A[h(x)] = (x, v)$ and place the old element in $A[h(x)]$ (suppose it is (y, z)) in the position given by the other hash function of y. For example, if $h(x) = h(y)$, we will move (y, z) to $A[g(y)]$, and if $h(x) = g(y)$

we will move (y, z) to $A[h(y)]$. As the new location for y might also be occupied, we iterate this process.

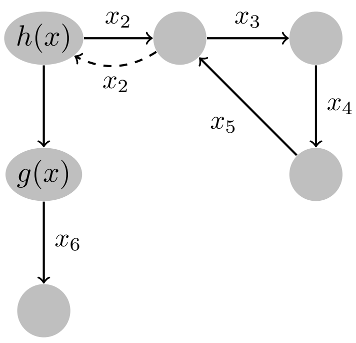
There are three different ways this iteration can go.

- There are no cycles, and the shifts form a path.



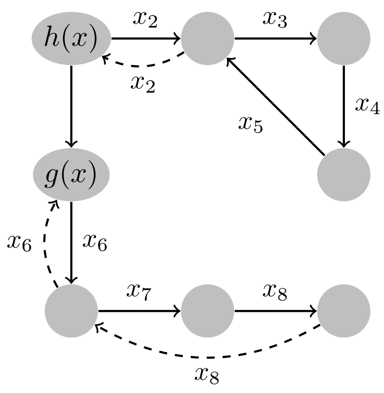
  In this case, we shift around some items but eventually find an empty slot.

- There is a cycle formed by shifting around the positions starting with $h(x)$, so we try again with $g(x)$ and succeed since it forms a path.



  In this case, we shift around some items but eventually find an empty slot.

- There is a cycle formed by shifting around the positions starting with $h(x)$, so we try again with $g(x)$ and find another cycle.



  In this case, we shift around some items indefinitely, and *never* find an empty slot! To address this, if we encounter a double cycle like in this case, we abandon the current Cuckoo hash, choose two new random hash functions, and re-insert all the items into that data structure. We will show that this happens so rarely we can afford to do such an expensive operation.

There is one additional subtlety involved in insertion: we keep a counter of number of shuffles, and if we ever reach $50 \lg n$, we give up on the current structure and try again as in the last case of a double cycle.

2.1 Runtime analysis

It remains to do an analysis of the expected time to do an insertion. Let T be the random variable representing the time to do an insertion. Let P_k be an indicator for the random variable that the shuffling from the insertion forms a path (case 2 above), and furthermore that the path is of length at least k. Similarly, let C_k be the indicator for the random variable that the shuffling from the insertion forms a single cycle (case 3 above), and furthermore that the total number of distinct edges traversed is at least k. Similarly, let D_k be the indicator for the random variable that the shuffling from the insertion forms a double cycle (case 4 above), and furthermore that the total number of distinct edges traversed is at least k.

With this terminology, we can bound

$$\mathbb{E}[T] \leq \mathbb{E}\left[\sum_{k=1}^{\infty} P_k + \sum_{k=1}^{\infty} C_k\right] + n \cdot \mathbb{E}[T] \cdot \left[\sum_{k=1}^{\infty} \mathbb{P}(D_k = 1) + \mathbb{P}(\text{more than } 50 \lg n \text{ shuffles})\right]$$

where the last term includes giving up and rebuilding. If α is the probability of rebuilding, then rearranging gives

$$\mathbb{E}[T] \leq \frac{1}{1 - \alpha \cdot n}\left[\sum_{k=1}^{\infty} \mathbb{E}[P_k] + \sum_{k=1}^{\infty} \mathbb{E}[C_k]\right].$$

We will show that the denominator is $1 - o(1)$ and the term in brackets is $O(1)$ so that $\mathbb{E}[T] = O(1)$.

Let's bound $\mathbb{P}(P_k = 1)$ for each k. Fix a particular path (i.e. hash locations/vertices $y_1, y_2, \ldots, y_{k+1}$) and edges $x_2, x_3, \ldots, x_{k+1}$. The probability that $h(x) = y_1$ is m^{-1}, and for each edge x_i the probability that $\{g(x_i), h(x_i)\} = \{y_{i-1}, y_i\}$ is at most $2m^{-2}$ (one copy of m^{-2} for each ordering of (g, h) and (h, g)). Hence, the probability of the particular path is at most $m^{-1} \cdot (2m^{-2})^k$. There are at most m^{k+1} possible choices of vertices and n^k choices of edges, so a union bound implies the probability of seeing any cycle of length k is

$$\mathbb{E}[P_k] = \mathbb{P}(P_k = 1) \leq \frac{1}{m}\left(\frac{2}{m^2}\right)^k \cdot m^{k+1} \cdot n^k = \left(\frac{2}{c}\right)^k$$

as $m = c \cdot n$. In particular, if $c = 4$, we have that

$$\mathbb{E}\left[\sum_{k=1}^{\infty} P_k\right] \leq \sum_{k=1}^{\infty}\left(\frac{2}{4}\right)^k = 1.$$

What about $\mathbb{P}(C_k = 1)$? This is mostly analogous, and is left as an exercise. Hint: it is possible to categorize the edges depending on whether the edge occurs before or after reaching the cycle. At least one of these subpaths has length at least $k/2$, and a similar analysis as above holds.

What about $\mathbb{P}(D_k = 1)$? A double cycle with k distinct edges has $k - 1$ distinct vertices. Fix these $k - 1$ vertices and k edges. The probability that $h(x)$ and $g(x)$ are correct is at most m^{-2}, and for each of the other $k - 1$ edges the probability is at most $2m^{-2}$. Hence, since there are at most m^{k-1} possible vertex choices and n^{k-1} possible other edge choices, by a union bound the probability of any double cycle of length k is at most

$$\mathbb{P}(D_k = 1) \leq \frac{1}{m^2} \cdot \left(\frac{2}{m^2}\right)^{k-1} \cdot m^{k-1} \cdot n^{k-1} = \left(\frac{2}{c}\right)^k \cdot \frac{1}{n^2}$$

so that

$$\sum_{k=1}^{\infty} \mathbb{P}(D_k = 1) = \Theta\left(\frac{1}{n^2}\right).$$

Finally, what is the probability of reaching $50 \lg n$ shuffles? This is analogous to the analysis of P_k and C_k analysis with the single large term $k^* = 50 \lg n$. It is an exercise to fill out the details, but again the probability will be $(2c^{-1})^{\Theta(\lg n)} = o(1)$.

Hence, we have that

$$\mathbb{E}[T] \le \frac{1}{1 - \alpha \cdot n}\left[\sum_{k=1}^{\infty} \mathbb{E}[P_k] + \sum_{k=1}^{\infty} \mathbb{E}[C_k]\right] \le \frac{1}{1 - o(1)}[O(1) + O(1)] = O(1)$$

as desired.

Note that this analysis assumed that h and g were random hash functions. It is an exercise to show that if h and g are drawn from a $\Theta(\lg n)$-wise independent family, then expected time per insertion is still $\Theta(1)$. It is not known if this much independence is necessary, but partial progress was made by Cohen and Kane [CK09] who constructed a 5-wise independent family which does not give expected constant time.

3 Approximate static dictionary

Consider the problem of *approximate static dictionary*, which requires maintaining $S \times \{0,1\}^r$ for $S \subseteq [u]$ a set of keys and $\{0,1\}^r$ the set of values, subject to:

- `query`(x): Returns the associated r-bit string if $x \in S$, and returns an arbitrary string if $x \notin S$.

This differs from non-approximate static dictionary as the behavior when $x \notin S$ can be arbitrary rather than reporting that $x \notin S$. Known solutions to non-approximate static dictionary (for example, using 2-level perfect hashing as covered in CS125) have parameters $O\big(n \cdot (r + \lg u)\big)$ bits of memory and $O(1)$ query time.

The use of approximation allows better space bounds, as in the **Bloomier filters** of Chazelle, Kilian, Rubinfeld, and Tal [CKRT04] which solve approximate static dictionary with an r-bit string using $O(nr)$ bits and $O(1)$ query time. The name invokes the Bloom filter, which can be thought of as an approximate static dictionary with $r = 1$ with the value being an indicator of whether the element is in the set.

We will present a different data structure based on Cuckoo hashing that matches the same bounds.

First, define the *Cuckoo graph* associated to a Cuckoo hash as the graph G with m vertices, labelled by locations in the array A, and n (undirected) edges labelled by the elements in S, with an edge $\big(h(x), g(x)\big)$ for each $x \in S$.

Our data structure now works as follows, for preprocessing, store the set S in a Cuckoo hash. If the associated Cuckoo graph G is not a forest, abandon the Cuckoo hash and try again (remember that the Cuckoo hash is randomized). Repeat until the Cuckoo graph is acyclic (we will show this takes

$O(1)$ attempts in expectation). Now, we have a Cuckoo graph G which is a forest. For each tree in G, consider an arbitrary node as the root, and associate to it the r-bit zero vector. Recall that edges in the Cuckoo graph are labelled by indices of A, and edges by elements in S. For each vertex v in the tree with edge x to its parent, associate $A[parent(v)] \oplus key(x)$ to v. This is illustrated in Figure 1.

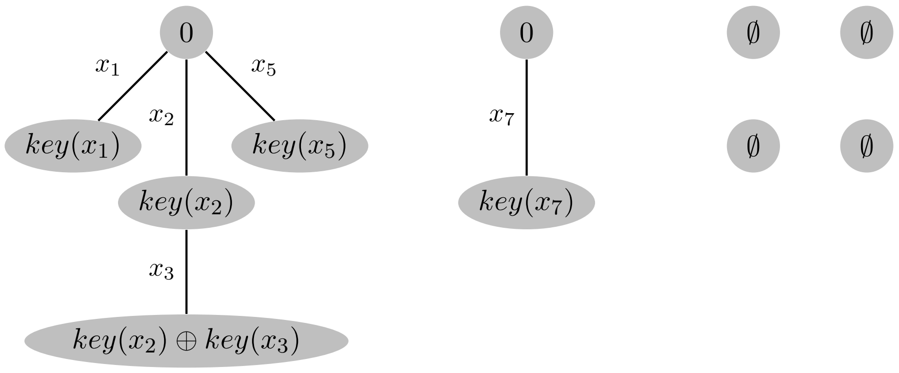


Figure 1: Cuckoo graph and labelling

Queries in the graph are then answered by taking the xor of the associated values of the nodes $h(x)$ and $g(x)$ in G, and hence can be done in constant time. We don't need to store isolated vertices or the keys themselves, so (ignoring the hash functions h and g), the total space used is r bits per each of at most $2n$ nodes, for total space $O(n \cdot r)$ bits as desired. Note that this analysis ignored the $O(n \lg u)$ bits needed to store the fully independent hash functions, it will be an exercise to show that this can be done in a space efficient manner using a $\Theta(\lg n)$-independent family.

All that remains is to show that in expectation, we only need $O(1)$ iterations to find a Cuckoo hash with acyclic Cuckoo graph. To do this, we need to bound the probability that the Cuckoo graph contains a cycle. There are two types of cycles: cycles of length 1 in which $h(x) = g(x)$ for some x, and cycles of length at least 2. For fixed x and value y, the probability that $h(x) = g(x) = y$ is m^{-2}, so by a union bound over the n values of x and m values of y, the probability of a single element cycle is at most $n \cdot m \cdot m^{-2} = c^{-1}$ as $m = c \cdot n$. For longer cycles of length $t \geq 2$, by fixing the vertices $y_1, y_2, \ldots, y_t$ and edges $x_1, x_2, \ldots, x_t$, the probability of each edge being in the appropriate place is at most $2m^{-2}$ (one copy of m^{-2} for each order of (g, h) and (h, g)). Hence, by a union bound over the at most m^t ordered vertices and n^t edge labels, the probability of a cycle of length t is at most $(2m^{-2})^t \cdot m^t \cdot n^t = (2c^{-1})^t$. Hence, the probability there is a cycle is at most

$$\mathbb{P}(\exists \text{ a cycle}) \leq \frac{1}{c} + \sum_{t=2}^{\infty} \left(\frac{2}{c}\right)^t = \frac{1}{c} + \frac{(2/c)^2}{1 - 2/c}$$

which if $c = 4$ is equal to $3/4$. Hence, the probability there is no cycle is at least $1/4$, and thus the expected number of tries to get an acyclic Cuckoo graph is at most $(1/4)^{-1} = 4 = O(1)$ as desired.

Note that this proof used full independence, as mentioned before it will be an exercise to reduce this to using $\Theta(\lg n)$-wise independence.

References

[ANS10] Yuriy Arbitman, Moni Naor, and Gil Segev. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 787–796, Oct 2010.

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[CFG$^+$78] Larry Carter, Robert Floyd, John Gill, George Markowsky, and Mark Wegman. Exact and approximate membership testers. In *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, STOC '78, pages 59–65, New York, NY, USA, 1978. ACM.

[CK09] Jeffrey Cohen and Daniel M. Kane. Bounds on the independence required for cuckoo hashing. Manuscript submitted to ACM Transactions on Algorithms, 2009.

[CKRT04] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The Bloomier filter: An efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, pages 30–39, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, May 2004.

[PSW13] Rasmus Pagh, Gil Segev, and Udi Wieder. How to approximate a set without knowing its size in advance. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 80–89, Oct 2013.