15-853:Algorithms in the Real World

Announcements:

Projects:

- Enter your team information in the Google Sheet by today (Nov. 8)
- Share the proposal and related papers in the shared Google Drive by Monday (Nov. 11)
- Project reports due on Dec 3 2:30pm
- Project presentations are in class on Dec 3 and 5

15-853:Algorithms in the Real World

Announcements:

Project report:

- We will provide a style file with a format next week:
  - 5 page, single column
  - Appendices (might not read them)
  - References (no limit)
- Write carefully so that it is understandable. This carries weight.
- Same format even for surveys: you need to distill what you read, compare across papers and bring out the commonalities and differences, etc.

15-853:Algorithms in the Real World

Announcements:

Projects:

- Ian looking for partners:
  - Project on coded computation
  - <quick description of coded computation>

15-853:Algorithms in the Real World

Announcements:

Homeworks:

There will be one homework assignment next week on hashing and cryptography module.

No homework assignments after the next one. Focus on project.

15-853:Algorithms in the Real World

Hashing:

Concentration bounds

Load balancing: balls and bins

Hash functions (cont.)

First a quick recap of what we have learnt in hashing so far.

Recall: Hashing

Concrete running application for this module: **dictionary**.

Setting:

- A large universe of keys (e.g., set of all strings of certain length): denoted by **U**
- The actual dictionary **S** (subset of U)
- Let |S| = N (typically N << |U|)

Operations:

- add(x): add a key x
- query(q): is key q there?
- delete(x): remove the key x

Recall: Hashing

“....**with high probability** there are not too many collisions among elements of S”

- We will assume a family of hash functions H.
- **When it is time to hash S, we choose a random function h ∈H**

Recall: Hashing: Desired properties

Let [M] = {0, 1, ..., M-1}

We design a hash function h: U -> [M]

1. Small probability of distinct keys colliding:
   1. If $x \neq y \in S$, $P[h(x) = h(y)]$ is “small”
2. Small range, i.e., small M so that the hash table is small
3. Small number of bits to store h
4. h is easy to compute

Recall: Ideal Hash Function

Perfectly random hash function:

For each $x \in S$, $h(x)$ =a uniformly random location in $[M]$

Properties:

- Low collision probability: $P[h(x) = h(y)] = 1/M$ for any $x \neq y$
- Even conditioned on hashed values for any other subset A of S, for any element $x \in S$, $h(x)$ is still uniformly random over $[M]$

Recall: Universal Hash functions

Captures the basic property of non-collision.

Due to Carter and Wegman (1979)

Definition: A family H of hash functions mapping U to [M] is universal if for any $x \neq y \in U$,

$$P[h(x) = h(y)] \leq 1/M$$

Note: Must hold for every pair of distinct x and $y \in U$.

Recall: Addressing collisions in hash table

One of the main applications of hash functions is in hash tables (for dictionary data structures)

Handling collisions:

Closed addressing

Each location maintains some other data structure

One approach: "**separate chaining**"

Each location in the table stores a **linked list** with all the elements mapped to that location.

Look up time = length of the linked list

To understand lookup time, we need to study the number of many collisions.

Recall: Addressing collisions in hash table

Let C(x) be the number of other elements mapped to the value where x is mapped to.

E[C(x)] = (N-1)/M

Hence if we use M = N = |S|,

lookups take **constant time in expectation**.

Let C = total number of collisions

E[C] $= \binom{N}{2} 1/M$

Recall: Addressing collisions in hash table

Suppose we choose M >= N^2

P[there exists a collision] = ½

⇒ Can easily find a **collision free hash table**!

⇒ Constant lookup time for all elements! (worst-case guarantee)

But this is large a space requirement.

(Space measured in terms of number of keys)

Can we do better? O(N)? (while providing worst-case guarantee?)

Recall: Perfect hashing

Handling collisions via "**two-level hashing**"

- First level hash table has size O(N)
- Each location in the hash table performs a collision-free hashing

Let C(i) = number of elements mapped to location i in the first level table

For the second level table, use C(i)^2 as the table size at location i. (We know that for this size, we can find a collision-free hash function)

Collision-free and O(N) table space!

Recall: k-wise independent hash functions

In addition to universality, certain independence properties of hash functions are useful in analysis of algorithms

Definition. A family H of hash functions mapping U to [M] is called k-wise-independent if for any k distinct keys

$x_1, x_2, \ldots x_k$ and any k values $\alpha_1, \alpha_2, \ldots, \alpha_k$

we have

$$P\left(h(x_1) = \alpha_1 \cap h(x_2) = \alpha_2 \cap \ldots \cap h(x_k) = \alpha_k\right) \le \frac{1}{M^k}$$

Case for k=2 is called “pairwise independent.

Recall Constructions: 2-wise independent

Construction 1 (variant of random matrix multiplication):

Let A be a m x u matrix with uniformly random binary entries.

Let b be a m-bit vector with uniformly random binary entries.

$$h(x) := Ax + b$$

where the arithmetic is modulo 2.

Claim. This family of hash functions is 2-wise independent.

Recall Constructions: 2-wise independent

Construction 3 (Using finite fields)

Consider $GF(2^u)$

Pick two random numbers $a, b \in GF(2^u)$. For any $x \in U$, define
$h(x) := ax + b$

where the calculations are done over the field GF(2u).

2-wise independent.

Recall Constructions: k-wise independent

Construction 4 (k-wise independence using finite fields):

Q: Any ideas based on the previous construction?

Hint: Going to higher degree polynomial instead of linear.

Consider GF(2^u).

Pick k random numbers $a_0, a_1, \ldots, a_{k-1} \in GF(2^u)$

$$h(x) = a_0 + a_1 x + \cdots + a_{k-1} x^{k-1}$$

where the calculations are done over the field GF(2u).

Similar proof as before.

Recall: Other approaches to collision handling

Open addressing:

No separate structures

All keys stored in a single array

Linear probing:

When inserting x and h(x) is occupied, look for the smallest index i such that (h(x) + 1) mod M is free, and store h(x) there.

When querying for q, look at h(q) and scan linearly until you find q or an empty space.

Other probe sequences:

Using a step-size

Quadratic probing

Cuckoo hashing

Another open addressing hashing method.

Invented by Pagh and Rodler (2004).

Take a table T of size M = O(N).

Take two hash functions h1, h2: U -> [M] from hash family H.

Let H be a fully-random

(O(log N)-wise independence suffices).

There are different variants of insertion and we will analyze a particular one.

Cuckoo hashing

Insertion:

When an element x is inserted, if either T[h1(x)] or T[h2(x)] is empty, put the element x in that location.

If not bump out the element (say y) in either of these locations and put x in.

When an element gets bumped out, place it in the other possible location. If that is empty then done. If not, bump the element in that location and place y there.

If any element relocated more than once then rehash everything.

Query/delete:

An element x will be either in T[h1(x)] or T[h2(x)].

O(1) operations

Cuckoo hashing

Theorem. The expected time to perform an insert operation is O(1) if M >= 4N.

Proof sketch.

Assume completely random hash functions (ideal).

For analysis we will use "cuckoo graph" G

- M vertices corresponding to hashtable locations
- Edges correspond to the items to be inserted.
  - For all x in S, e_x=(h1(x),h2(x)) will be in the edge set
- Bucket of x, B(x) = set of nodes of G reachable from h1(x) or h2(x)
  - Connected component of G with edge e_x

Cuckoo hashing

Proof sketch (cont.):

Q: What is the relationship between the #vertices and #edges in any of the connected components of G for the requirement of no collision?

#vertices >= #edges (since #locations >= #items since no collisions allowed)

Q: If adding an edge violates this property, what does it lead to?

Rehash

E[Insertion time for x] = E[|B(x)|]

Goal: To show E[|B(x)|] <= O(1)

Cuckoo hashing

Proof sketch (cont.):

Goal: To show E[|B(x)|] <= O(1)

$$E[|B(x)|] = \sum_{\substack{y \in S \\ y \neq x}} P[e_y \in B(x)]$$

$$\leq N \, P[e_y \in B(x)]$$

Sufficient to show $P[e_y \in B(x)] \leq O\left(\frac{1}{xn}\right)$

Cuckoo hashing

Proof sketch (cont.):

Goal: To show $P[e_y \in B(x)] \leq O\left(\frac{1}{M}\right)$

Lemma. For any i, j in [M],

P[there exists a path of length ℓ between i and j in the cuckoo graph] $\leq \frac{1}{2^{\ell} M}$

Proof. For ℓ = 1, P[edge i between j]

$$= P[\exists y \text{ s.t. } e_y \text{ exists in } G]$$

$$\leq N \cdot \frac{2}{M^2}$$

$\leftarrow p[(h_1(y) = i \cap h_2(y) = j) \cup (h_2(y) = i \cap h_1(y) = j)]$

$$= \frac{1}{2} \cdot \frac{1}{M}$$

Then induction on ℓ. (Exercise)

Cuckoo hashing

Proof sketch (cont.):

Goal: To show $P[e_y \in B(x)] \leq O\left(\frac{1}{M}\right)$

Proof. Using the Lemma,

$$P[e_y \in B(x)] \leq \sum_{l \geq 1} \frac{1}{2^l M}$$

$$= O\left(\frac{1}{M}\right)$$

- This proof for Cuckoo hashing is by Rasmus Pagh and a very nice explanation of this proof can be found at: http://www.cs.toronto.edu/~wgeorge/csc265/2013/10/17/tutorial-5-cuckoo-hashing.html
- A different proof can be found at:

Cuckoo hashing: occupancy rate

One of the key metrics for hash tables is the "occupancy rate".

Corresponds to the space overhead needed

With M >= 4N we have only 25% occupancy!

Can we do better?

Turns out that you can get close to 50% occupancy, but better than 50% causes the linear-time bounds to fail.

If one uses d hash functions instead of 2?

With d = 3, experimentally > 90% occupancy with linear-time bounds.

Put more items in a location (say, 2 to 4 items) in each location?

Experimental conjectures on better occupancy.

Cuckoo hashing

On independence property of the hash functions used:

O(log N)-wise independence suffices.

But these are expensive to compute and store.

6-wise independent hash functions insufficient to get the failure probability low enough (i.e., 1-1/N) to get whp results (Cohen and Kane 2009).

Simple tabulation hashing has been shown to give pretty good performance (Patrascu and Thorup 2012)

Application: Bloom filter

Representing a dictionary with far fewer bits when only need membership query.

Possible if we:

Allow to make mistakes on membership queries

No deletions

Data structure: “Bloom filter” [Bloom 1970]

- Only false positives; no false negatives
  - may report that a key is present when it is not
- Very useful for “filtering out”: scenario where most keys will not belong to the dictionary (|S| << |U|).
  - E.g: malicious/blocked websites in web browser
- If the answer is “Yes” then you can use a slow data structure

Bloom filter

Space efficient data structure for *approximate* membership queries.

- Keep an array T of M bits
    - initially all entries are zero.
- k hash functions: $h_1, h_2, .., h_k: U \rightarrow [M]$
    - Assume completely random hash functions for analysis

Adding a key:

- To add a key $x \in S \subseteq U$, set bits $T[h_1(x)], T[h_2(x)], ..., T[h_k(x)]$ to 1

Bloom filter

Membership query:

- For a query for key $x \in U$: check if all the entries $T[h_i(x)]$ are set to 1
- If so, answer Yes else answer No.

Q: Why no false negatives?

If an item x is present, then corresponding bits will be set.

Q: Why false positives?

Other elements could have set the same bits.

Let's analyze the probability of false positives.

Bloom filter

A false positive for a query occurs when all k bits in T corresponding to a query is set.

Let p = probability that a bit in T is not set

p = $\left(1 - \frac{1}{M}\right)^{kN}$

This about how to simplify this expression.

We will continue from here in the next lecture.