Detour: Bloom Filters

Say we want to store a set \( S \subseteq [n] \), \( |S| = m \ll n \).

Recall hash tables let us do this using \( O(m \log n) \) bits of space, and depending on the implementation, supported insert, delete, and lookup operations in expected or worst-case, constant time with high probability.

(Also note that these hash tables allowed enumeration of all items in \( S \) in time \( O(m) \) and also allowed associating keys with values).

We will now see a method for approximately representing \( S \) using just \( O(m) \) bits. This method supports only lookup and insert operations and does not support associating keys and values.

Bloom Filter:

- Let \( k \) be a constant (say 5).
- Maintain \( M \) bits, initialized to 0. View the bit as \( k \) tables, each covering \( \frac{m}{k} \) bits.
- Let \( h_1, \ldots, h_k : \mathbb{N} \rightarrow [m/k] \) be random hash functions.
- Insert \( (x) \): For \( i = 1 \ldots k \)
  \[ T_i[h_i(x)] = 1 \]
- Look-up \( (x) \):
  If \( T_i[h_i(x)] = 1 \) for all \( i \in [k] \), output 1. Else output 0.

Insert and lookups require \( k \) hash and \( k \) bit reads.

Claim 1: If \( x \in S \), then \( \Pr \left[ \text{look-up}(x) \right] = 1 \). “No fake negatives.”

Proof: Obvious from definition of insert and look-up operations.

Claim 2: If \( x \notin S \), then
\[
\Pr_{x \in [n]} \left[ \text{look-up}(x) \right] = \left( \frac{1 - \frac{1}{m}}{m} \right)^k 
\]
\[
\approx \left( 1 - e^{-m \cdot m/k} \right)^k, 
\]

“False positive are rare.”

Example: Consider a set \( S \) of 100,000 common passwords, which are 7 characters long on average. TH3 requires 700,000 bytes to store exactly. They might be compressible to 300,000 bytes or so, but then insert and lookups might take a long time.

Instead, keep a 100,000-byte Bloom Filter consisting of \( k = 5 \) tables, each with 169,000 bits. Then false positive probability is \( \approx 0.06 \% \) and inserts and lookups take 5 hash evaluations.
Sparse Recovery in Data Streams via IBLTs

Consider a turnstile stream \( \sigma = \langle (a_1, f_1), \ldots, (a_m, f_m) \rangle \) under the promise that at the end of the stream, \( f_i = 0 \) for all but at most \( N \) items i.

E.g., flows through a router.

- **Goal:** List all items with non-zero frequency, using space \( O(N \log n) \) in bits.

- We will use this next lecture to build algorithm, for more complicated problems.

For simplicity, assume that at end of stream no item has frequency more than 1.

**IBLT algorithm:**

- Maintain \( M \) cells, where each cell stores a count and a keysum \( a_j \) initialized to 0.
- Let \( h_1, \ldots, h_r : [n] \rightarrow M \) be random hash functions.

- While processing update \( (a_i, f_i) \):
  - For \( j = 1, \ldots, r \):
    - Add \( f_i \cdot a_i \) to keysum field of cell \( h_j(a_i) \)
    - Add \( f_i \) to count for cell \( h_j(a_i) \)
Listing Algorithm:

• Call a cell "pure" if exactly one item with non-zero frequency hashes to it.
• Under assumption that all items have frequency ≤ 1, can check if a cell is
  pure simply by looking whether count = 1. If so, can identify the item hashing to
  it by looking at Key run field.
• While there exists a pure cell:
  • Output the unique item 𝑖: fichio hashing to the cell.
  • Let 𝑠 = count of pure cell (assumed to be 1 by )
  • Call Insert (𝑖, 𝑠).

• In case where frequency may be > 1, can still identify pure cells and the
  unique item hashing to it (with high probability, using fingerprint 3).
  • Details later, (Actually on Problem Set #3).

Key Question: How many cells are required to ensure the listing algorithm
successfully recovers an item with non-zero frequency?

**Relating Algorithms and Their Analysis**

• A hypergraph is just like a standard graph (vertices and edges
  connecting them) except edges may contain more than 2 vertices.
  • A hypergraph is r-uniform if each edge contains r-vertices.

• An IBLT naturally defines a corresponding hypergraph G
  • IBLT cells ↔ vertices of G
  • Items in IBLT ↔ hyperedges of G
  • G is r-uniform if IBLT uses r hash functions

Note: If IBLT uses truly random
  hash functions, then G is a
  random r-uniform hypergraph
  with 𝑀 nodes and 𝑁 edges.
The Lifting Algorithm for the IBLT corresponds to running the following algorithm on $G$.

- While there exists a vertex $v$ in $G$ of degree $\leq 2$
  - Remove $v$ and all of its incident edges

The remaining graph when the above algorithm gets stuck is called the $2$-core of $G$.

- Lifting succeeds when $2$-core is empty.

- Example in case of a simple graph:

In simple graphs, the $2$-core is empty if and only if $G$ does not contain a cycle.

In hypergraphs, the $2$-core is a more complicated object.
Consider a random r-uniform hypergraph $G$ with $M$ nodes and $N = c \cdot M$ edges.

- i.e., each edge has $r$ vertices, chosen uniformly at random from $[N]$.

Known Fact: Appearance of a non-empty $K$-core obeys a sharp threshold.

- For some constant $C_{K,r}$, when $N < C_{K,r} \cdot M$ by a constant factor, the $K$-core is empty with probability $1 - o(1)$.
- When $N \geq C_{K,r} \cdot M$ by a constant factor, the $K$-core is non-empty with probability $1 - o(1)$.

Implication: to successfully lift a set of size $N$ with probability $1 - o(1)$, the IMET needs $\frac{N}{C_{K,r}}$ cells.

E.g., $C_{2,3} \approx 0.819$, $C_{3,4} \approx 0.772$, $C_{3,3} \approx 1.553$

In general, $C_{K,r} = \min_{x > 0} x \sum_{j=0}^{K-1} \frac{x^j}{j!} \left( 1 - e^{-x} \right)^{K-1}$. 