DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 17 | Part 1

Today's Lecture

Massive Sets

- You’ve collected 1 billion tweets.[1]
- **Goal**: given the text of a new tweet, is it already in the data set?

[1]This is about two days of activity.

Membership Queries

- We want to perform a **membership query** on a collection of strings.
- Hash tables support $\Theta(1)$ membership queries.
- **Idea**: so let's use a hash table (Python: `set`).

Problem: Memory

- How much memory would a `set` of 1 billion strings require?

- Assume average string has 100 ASCII characters.

  (8 bits per char)×(100 chars)×1 billion = 100 gigabytes

- That's way too large to fit in memory!

Today's Lecture

- **Goal**: fast membership queries on massive data sets.
- Today's answer: **Bloom filters**.

DSC 190

DATA STRUCTURES & ALGORITHMS

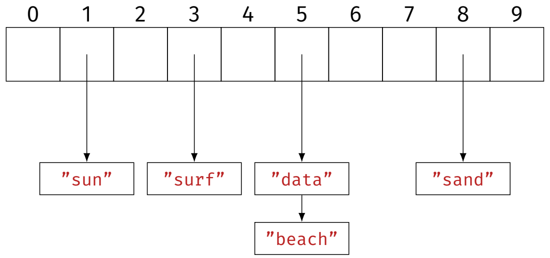
Lecture 17 | Part 2

Bit Arrays

The Challenge

- We want to perform membership queries on a massive collection (too large to fit in memory).
- We want to remember which elements are in the collection...
- ...*without* actually storing all of the elements.
- From hash tables to Bloom filters in 3 steps.

First Stop: Hash Tables



s	hash(s)
"surf"	3
"sand"	8
"data"	5
"sun"	1
"beach"	5

Memory Usage

- **Problem**: we’re storing all of the elements.
- Why? To resolve collisions.
- **Fix**: ignore collisions.

Second Stop: Hashing Into Bit Arrays

0	1	2	3	4	5	6	7	8	9
0	1	0	1	0	1	0	0	1	0

s	hash(s)
"surf"	3
"sand"	8
"data"	5
"sun"	1
"beach"	5

- Use a bit array `arr` of size c.
- **Insertion**: Set `arr[hash(x)] = 1`.
- **Query**: Check if `arr[hash(x)] = 1`.

Second Stop: Hashing Into Bit Arrays

0	1	2	3	4	5	6	7	8	9
`0`	`1`	`0`	`1`	`0`	`1`	`0`	`0`	`1`	`0`

`s`	`hash(s)`
`"surf"`	3
`"sand"`	8
`"data"`	5
`"sun"`	1
`"beach"`	5

- Use a bit array `arr` of size c.
- **Insertion**: Set `arr[hash(x)] = 1`.
- **Query**: Check if `arr[hash(x)] = 1`.
- Can be **wrong**!

False Positives

0	1	2	3	4	5	6	7	8	9
0	1	0	1	0	1	0	0	1	0

s	hash(s)
"surf"	3
"sand"	8
"data"	5
"sun"	1
"beach"	5

- Query can return **false positives**.
- e.g., `hash("ucsd") == 3`
- Cannot return false negatives.

Memory Usage

- Requires c bits, where c is size of the bit array.
- False positive rate depends on c.
    - c is small → more collisions → more errors
    - c is large → fewer collisions → fewer errors
- **Tradeoff**: get more accuracy at cost of memory.

False Positive Rate

- What is the probability of a false positive?
- Suppose there are c buckets, and we've inserted n elements so far.
- We query an object x that we haven't seen before.
- False positive ⇔ `arr[hash(x)] == 1`.

False Positive Rate

- Assume `hash` assigns bucket uniformly at random.
  - If $x \neq y$ then, $\mathbb{P}(\texttt{hash(x)} = \texttt{hash(y)}) = 1/c$
- Prob. that first element does not collide with x: $1 - 1/c$.
- Prob. that first two do not collide: $(1 - 1/c)^2$.
- Prob. that all n elements do not collide: $(1 - 1/c)^n$.

False Positive Rate

- Hint: for large z, $(1 - 1/z)^z \approx \frac{1}{e}$
- So the probability of no collision is:

$$(1 - 1/c)^n = [(1 - 1/c)^c]^{n/c} \approx e^{-n/c}$$

- This is the probability of no false positive.
- Probability of false positive upon querying x: $\approx 1 - e^{-n/c}$

False Positive Rate

- For fixed query, probability of false positive: $\approx 1 - e^{-n/c}$.
    - n: number of elements stored
    - c: size of array (number of bits)

- Randomness is over choice of hash function.
    - Once hash function is fixed, the result is always the same.

Fixing False Positive Rate

- Suppose we'll tolerate false positive rate of ε.
- Assume that we'll store around n elements.
- We can choose c:

$$1 - e^{-n/c} = \varepsilon \quad \Longrightarrow \quad c = -\frac{n}{\ln(1 - \varepsilon)}$$

Example

- Suppose we want ≤ 1% error.
- Previous slide says our bit array needs to be 100 times larger than number of elements stored.[2]
- Memory when $n = 10^9$: 1 billion bits ×100 = 12.5 GB.
- Can we do better?

[2]We could have guessed this, huh?

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 17 | Part 3

Bloom Filters

Wasted Space

- Suppose we want ≤ 1% error.
- Our bit array needs to be 100 times larger than number of elements stored.
- That’s a lot of **wasted space**!

Third Stop: Multiple Hashing

- **Idea**: use several smaller bit arrays, each with own hash function.

Third Stop: Multiple Hashing

0	1	2	3	4	5	6	7	8	9
0	1	0	1	0	1	0	0	1	0

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	1	1	0	1

s	hash_1(s)	hash_2(s)
"surf"	3	7
"sand"	8	7
"data"	5	4
"sun"	1	9
"beach"	5	6

- Use *k* bit arrays of size *c*, each with own independent hash function.
- **Insertion**: Set
  `arr_1[hash_1(x)] = 1`,
  `arr_2[hash_2(x)] = 1`,
  ...,
  `arr_k[hash_k(x)] = 1`.

Third Stop: Multiple Hashing

0	1	2	3	4	5	6	7	8	9
0	1	0	1	0	1	0	0	1	0

0	1	2	3	4	5	6	7	8	9
0	0	0	0	1	0	1	1	0	1

s	hash_1(s)	hash_2(s)
"surf"	3	7
"sand"	8	7
"data"	5	4
"sun"	1	9
"beach"	5	6

- Use *k* bit arrays of size *c*, each with own independent hash function.
- **Query**: Return `True` if **all** of
  ```
  arr_1[hash_1(x)] = 1,
  arr_2[hash_2(x)] = 1,
  ...,
  arr_k[hash_k(x)] = 1.
  ```
- Example:
  ```
  hash_1("hello") == 3,
  hash_2("hello") == 2
  ```

Intuition

- False positive occurs only if false positive in **all** tables.
- This is pretty unlikely.
- If false positive rate in one table is small (but not tiny), probability false positive in all tables is still tiny.

More Formally

- Probability of false positive in first table:
  $\approx 1 - e^{-n/c}$.

- Probability of false positive in all k tables:
  $\approx (1 - e^{-n/c})^k$.

- Example: if $c = 4n$ and $k = 3$, error rate is $\approx 1\%$.

- Uses only $12 \times n$ bits, as opposed to $100 \times n$ from before.

Last Stop: Bloom Filters

- How many different bit arrays do we use? (What is k?)
- How large should they be? (What is c?)
- **Bloom filters**: use k hash functions, but only one medium-sized array.

Last Stop: Bloom Filters

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	1	0	1	0	1	1	0	1	1	0	0	0	1	0	1	0	1	0	0	0

`s`	`hash_1(s)`	`hash_2(s)`
`"surf"`	13	17
`"sand"`	8	6
`"data"`	15	1
`"sun"`	1	3
`"beach"`	5	9

- Use one bit arrays of size *c*, but *k* hash functions.
- **Insertion**: Set `arr[hash_1(x)] = 1`, `arr[hash_2(x)] = 1`, ..., `arr[hash_k(x)] = 1`.

Last Stop: Bloom Filters

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
0	1	0	1	0	1	1	0	1	1	0	0	0	1	0	1	0	1	0	0	0

s	hash_1(s)	hash_2(s)
"surf"	13	17
"sand"	8	6
"data"	15	1
"sun"	1	3
"beach"	5	9

- Use one bit arrays of size *c*, but *k* hash functions.
- **Query**: Return `True` if **all** of `arr[hash_1(x)] = 1`, `arr[hash_2(x)] = 1`, …, `arr[hash_k(x)] = 1`.
- Example: `hash_1("hello") == 3`, `hash_2("hello") == 2`

Intuition

- Multiple hashing allows bit arrays to be smaller.
- Even more efficient: let them share memory.
- “Overlaps” are just collisions; we can handle them.

False Positive Rate

- Consider querying new, unseen object `x`.
- We'll look at k bits.
    - `arr[hash_1(x)]`, ..., `arr[hash_k(x)]`.
- Fix one bit. What is the chance that it is already one?

False Positive Rate

- Probability of bit being zero after first element inserted: $(1 - 1/c)^k$
- After second element inserted: $(1 - 1/c)^{2k}$
- After all n elements inserted: $(1 - 1/c)^{nk}$
- And:

$$(1 - 1/c)^{nk} = [(1 - 1/c)^c]^{nk/c} \approx e^{-nk/c}$$

False Positive Rate

- Probability of bit being **one** after *n* elements inserted:

$$1 - e^{-nk/c}$$

- For a false positive, all *k* bits (for each hash function) need to be one.

- Assuming independence,[3] probability of false positive:

$$(1 - e^{-nk/c})^k$$

[3]Only true approximately. If this bit was set, some other bit was not.

Minimizing False Positives

- For a fixed n and c, the number of hash functions k which minimizes the false positive rate is

$$k = \frac{c}{n} \ln 2$$

- Plugging this into the error rate:

$$\varepsilon = (1 - e^{-nk/c})^k \quad \Longrightarrow \quad \ln \varepsilon = -\frac{c}{n}(\ln 2)^2$$

- If we fix ε, then $c = -n \ln \varepsilon / (\ln 2)^2$

Summary: Designing Bloom Filters

- Suppose we wish to store n elements with ε false positive rate.
- Allocate a bit array with $c = -n \ln \varepsilon / (\ln 2)^2$ bits.
- Pick $k = \frac{c}{n} \ln 2$ hash functions.

Example

- Let $n = 10^9$, $\varepsilon = 0.01$.
- We need $c \approx 9.5n \rightarrow 10n$ bits = 1.25 GB.
- We choose $k = \frac{9.5n}{n} \ln 2 \rightarrow 7$ hash functions.

DSC 190

DATA STRUCTURES & ALGORITHMS

Lecture 17 | Part 4

Bloom Filters in Practice

Applications

- A cool data structure.
- Most useful when data is huge or memory is small.

Application #1

- De-duplicate 1 billion strings, each about 100 bytes.

- Memory required for `set`: 100 gigabytes.

- Instead:
    - Loop through data, reading one string at a time.
    - If not in Bloom filter, write it to file.

- With 1% error rate, takes 1.25 GB.

Application #2

- A ***k*-mer** is a substring of length k in a DNA sequence:

  "GATTACATATAGGTGTCGA"

- Useful: does a long string have a given k-mer?

- There are a *massive* number of possible k-mers.
  - 4^k, to be precise.
  - Example: there are over 10^{18} 30-mers.

- Slide window of size k over sequence, store each substring in Bloom filter.

Application #2

- Human genome is a 725 Megabyte string, 2.9 billion characters.
- To store all k-mers, each character stored k times.
- Storing 30-mers in `set` would take 30 × 725 MB ≈ 22 GB.
- By “forgetting” the actual strings, Bloom filter (1% false positive) takes

2.9 billion bits ≈ 360 megabytes

Application #3

- Suppose you have a massive database on disk.
- Querying the database will take a while, since it has to go to disk.
- Build a Bloom filter, keep in memory.
    - If Bloom filter says x not in database, don’t perform query.
    - Otherwise, perform DB query.
- Speeds up time of “misses”.

Limits

- Bloom filters are useful in certain circumstances.

- But they have disadvantages:
  - Need good idea of size, *n*, ahead of time.
  - There are false positives.
  - The elements are not stored (can't iterate over them).

- Often a `set` does just fine, with some care.

Example

- Suppose you have 1 billion tweets.
- Want to de-duplicate them by **tweet ID** (64 bit number).
- Total size: 8 gigabytes.
- I have 4 GB RAM. Should I use a Bloom filter?

De-duplication Strategy

- Design a hash function that maps each tweet ID to $\{1, \ldots, 8\}$.
- Loop through tweet IDs one-at-a-time, hash, write to file:

  `hash(x) == 3` → write to `data_3.txt`
- Read in each file, one-at-a-time, de-duplicate with `set`, write to `output.txt`