CS 261: Graduate Data Structures

Week 3: Sets

David Eppstein
University of California, Irvine

Spring Quarter, 2021

Sets

Example

Depth-first search example again:

```
def DFS(s,G):
    visited = set()     # already-processed vertices

    def recurse(v):     # call for each vertex we find
        visited.add(v)  # remember we've found it
        for w in G[v]:  # look for more in neighbors
            if w not in visited:
                recurse(w)
```

We need a data structure to represent the visited set

Operations: new set, add element, test membership
Neighbors `G[v]` might also be a set, iterated over

Many other operations not used here, for example: remove element

Sets in Python

New empty set: `set()`
New set from iterator: `set(L)`

Add or remove element: `S.add(x)`, `S.remove(x)`

Union: `S & T`
Intersection: `S | T`
Asymmetric difference (elements in one but not the other): `S - T`
Symmetric difference (elements in exactly one of two sets): `S ^ T`

Subset and equality tests: `S < T`, `S <= T`, `S == T`

Membership testing: `x in S`, `x not in S`

List elements: `for x in S`

Not built into Python until version 2.4
(2004, ten years after Python 1.0 released)

Sets in Java

Main interface: java.util.Set

(doesn't implement sets, just describes their API)

Implementations include HashSet
(more or less the same as Python sets)

...and EnumSet
(for sets of elements from enumerated lists of keywords)

Combining sets using one-element operations

Example: set intersection of two sets A and B

1. Swap if necessary so A is the smaller set
2. Make output set C
3. For each element x of A:

    If x is also in B:

        Add x to C
4. Return C

Number of one-element operations = O(size of smaller set)

Other set operations may need # operations = O(total size)

Sets from hash tables

Used by Python set and Java HashSet

Set = the keys of a hash table

Ignore the values
or use a special flag value as the value for each key

All operations take expected time $O(1)$ per element

Space for a set with n elements: $O(n)$ words of memory
(where a word = enough storage to point to a single object)

Bitmaps

Representing sets as numbers

Useful when the set elements are, or can be easily converted to, small non-negative integers $0, 1, 2, \ldots$

(Example: Java EnumSet)

Main idea: Represent the set $S = \{x, y, z, \ldots\}$

as the number $s = 2^x + 2^y + 2^z$

Binary representation of s: 1 in positions $x, y, z, \ldots$, 0 elsewhere

Example: The number 222, in binary, is

$11011110_2 = 2^7 + 2^6 + 2^4 + 2^3 + 2^2 + 2^1$.

It represents the set $\{1, 2, 3, 4, 6, 7\}$.

Implementation for small universes

When a single set fits into a single word of storage
(all elements are integers in range $[0, 31]$ or $[0, 63]$):

empty set:
```
0
```

set with one element x:
```
1<<x
```

add x to S:
```
S |= 1<<x
```

remove x from S:
```
S &= ~(1<<x)
```

test membership:
```
if S & (1<<x)
```

test if $A \subset B$:
```
(A &~ B) == 0
```

intersection:
```
A & B
```

union:
```
A | B
```

asymmetric difference:
```
A &~ B
```

symmetric difference:
```
A ^ B
```

Iterating over the elements, in order

Recall how binary numbers S and $S-1$ differ:
Convert low-order 1 to 0, lower 0's to 1's

Smallest element of S, as a one-element set: `S &~ (S-1)`

Repeatedly find this one-element set, convert it into an element, and remove it until the whole set is empty

```
set2element = {1<<x: x for x in range(64)}

def elements(S):
    while S:
        yield set2element[S &~ (S-1)]
        S &= S-1
```

Larger ranges of elements

For max element $N \geq 64$ this all still works but is less efficient

Better: Store array of $N/64$ words, each 64 bits

Individual-element operations: only look at one word

Whole-set operations: look at all words

Iterate elements: Can also maintain recursive set of nonempty words to find them more quickly

Analysis

Individual-element operations: $O(1)$, same as hash table

Whole-set operations: $O(N)$ (where N is max element value), worse than $O(n)$ of hash table (where n is set size)

But in practice when this works it is much faster, more compact!

Two reasons:

- No hash functions, no random memory access
- Whole-set operations operate on 64 elements at a time, giving a factor-64 speedup: same O-notation, but huge in practice

Filters

Main idea of filters

Represent n-element sets using only $O(n)$ bits

Better than hash tables, $O(n)$ words

Better than bitmaps, $O(N)$ bits where $N = \max$ element

What do we have to pay to get this savings?

Answers are approximate

If $x \in S$, filter will always say that $x \in S$
(cannot have "false negatives")

But if $x \notin S$, it might incorrectly say $x \in S$
(can have "false positives")

False positive rate

Choose a random x that is not in your set S

What is the probability that your filter incorrectly says $x \in S$?

Called the "false positive rate"

We want it to be small, so we will use ε as notation

Typically known when we initialize filter structure,
used to determine its structural parameters

Often (but not always) ok to assume constant, e.g. $\varepsilon = 0.1$

When are filters useful?

If processing non-members is easier and you expect many of them

Filter can be small enough to fit in cache $\Rightarrow$ fast

Use slower exact set data structure to check matched elements

Few false positives $\Rightarrow$ few unnecessary calls to exact structure

If memory is limited and some false positives are harmless

Example: Access control for private internet server

Use filter on firewall to only allow whitelisted clients through

Firewall needs only small memory for filter

Server can handle smaller volume of non-clients that get through

Comparison of filters: Bloom filter

Bloom, CACM 1970; ≈ 25k other publications

(More details later this week.)

Widely implemented, practical

Storage: $1.44 n \log_2 \frac{1}{\varepsilon}$ bits
larger than optimal by the 1.44 factor

Membership testing: $O(1/\epsilon)$ time

Can add but not remove elements

Comparison of filters: Cuckoo filter

Fan et al, CoNEXT '14; $\approx$ 500 other publications

(More details later this week.)

Implemented and practical,
better in practice than Bloom

Storage: $(1 + o(1))n \log_2 \frac{1}{\varepsilon}$ bits, optimal!

Membership testing: $O(1)$ time
(with good locality of reference: works well with cache)

Can add and remove elements

Storage bound requires $\epsilon = o(1)$
bigger sets need to have smaller false positive rates

(Some sources exaggerate this requirement by saying that
"in theory, Cuckoo filters do not work")

Comparison of filters: Xor filter

Graf and Lemire, JEA 2020, only one publication

For details, see `https://r-libre.teluq.ca/1857/`

Implemented and practical,
better in practice than Bloom
often better than cuckoo

Storage: $(1 + o(1))n \log_2 \frac{1}{\varepsilon}$ bits, optimal!

Membership testing: $O(1)$ time

Can handle constant error rates, unlike cuckoo

Cannot handle additions or removals

Bloom filters

Main idea of Bloom filters

Two parameters, N and k, to be chosen later

Store a table B of N bits, initially all zero

Construct k hash functions $h_1(x), \ldots h_k(x)$

To add x to the set, set its bits to one:

$$B[h_1(x)] = B[h_2(x)] = \cdots = B[h_k(x)] = 1$$

To test membership, check that all bits are one:

```
for i = 1, 2, ... k:
    if B[h_i(x)] = 0:
        return False
return True
```

B is just the bitmap representation of the set of hashes of elements!

Example of Bloom filter

Suppose $N = 9$ and $k = 3$ with hash functions mapping
$a \to 0, 3, 4$; $b \to 1, 5, 7$; $c \to 2, 3, 5$; $d \to 1, 4, 8$; $e \to 0, 3, 5$

Initially $B = b_8 b_7 b_6\ b_5 b_4 b_3\ b_2 b_1 b_0 = 000\,000\,000$

Add a, setting bits $0, 3, 4$: $\quad B = 000\,011\,001$

Add b, setting bits $1, 5, 7$: $\quad B = 010\,111\,011$

Add c, setting bits $2, 3, 5$: $\quad B = 010\,111\,111$

Test membership for d: $b_1 = b_4 = 1$, $b_8 = 0 \Rightarrow$ return False

Test membership for e: $b_0 = b_3 = b_5 = 1 \Rightarrow$ return True
This is a false positive!

Bloom filter analysis

Let f be the fraction of bits that are one $\Rightarrow$
(by random hash assumption) false positive rate $\varepsilon = f^k$

Can't use Chernoff bound (bits are not independent of each other)
but related Azuma–Hoeffding inequality $\Rightarrow f \approx E[f]$
Linearity of expectation $\Rightarrow E[f] = \Pr[\text{any given bit is one}]$

$$
\begin{aligned}
\Pr[\text{bit is } 1] &= 1 - \Pr[\text{same bit is } 0] \\
&= 1 - \Pr[\text{all hashes of elements miss that bit}] \\
&= 1 - \left(1 - \tfrac{1}{N}\right)^{kn} \\
&= 1 - \left(\left(1 - \tfrac{1}{N}\right)^{N}\right)^{kn/N} \\
&\approx 1 - \left(\tfrac{1}{e}\right)^{kn/N}
\end{aligned}
$$

Bloom filter analysis (continued)

Simplifying assumptions: Suppose we already know N

Let's try plugging fractional values of k into the calculation (even though in the actual data structure it must be an integer)

What choice of k gives the best false positive rate ε?

Turns out to be: k that makes fraction of ones be $f = 1/2$

(Can prove by calculus, but intuitive reason: because then the Bloom filter has the highest possible information content)

$$f = \frac{1}{2} \quad \Rightarrow \quad 1 - \left(\tfrac{1}{e}\right)^{kn/N} = \frac{1}{2} \quad \Rightarrow \quad N = \frac{kn}{\log 2}$$

With $f = 1/2$, $\varepsilon = 1/2^k$ giving $k = \log_2 \frac{1}{\varepsilon}$ and $N = \dfrac{n \log_2 1/\varepsilon}{\log 2}$

Bloom filter summary

For sets of size n, with desired false positive rate ε:

Choose number of hash functions $k \approx \log_2 \frac{1}{\varepsilon}$

Choose bit array size $N \approx \dfrac{n \log_2 1/\varepsilon}{\log 2} \approx 1.44 n \log_2 \dfrac{1}{\varepsilon}$

Store bitmap set of hashes of elements

Additions and membership tests take time $O(k)$,
which is $O(1)$ for $\varepsilon = \text{constant}$

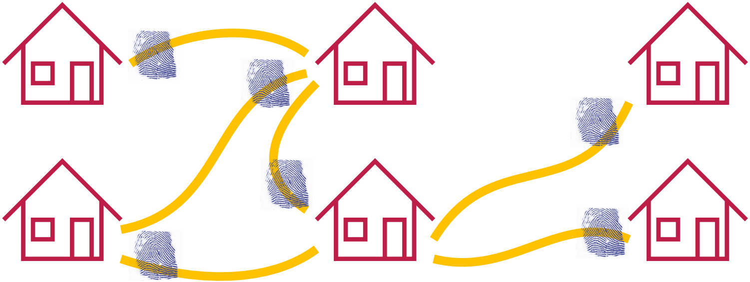
Can't remove any element because we don't know which of its bits are shared with other elements and which are used only by it

Cuckoo filters

Main idea

Use a hash function f to compute a short
"fingerprint" $f(x)$ for each element x

Store fingerprints, not key-value pairs, in a cuckoo hash table
(each fingerprint can go in one of two possible home cells)



Saves space because fingerprints use fewer bits than full elements

Basic operations

Test if x is in set:

Check whether either of the two cells for x contains $f(x)$

False positive:

Some other element collides with x in both location and fingerprint

Insert x:

(Allowing > 1 fingerprint/cell to get load factor near one)

Add fingerprint $f(x)$ to home cell for x

If fingerprints overflow, insert recursively to second home cells

Delete x:

Remove fingerprint from one of its two homes

Difficulties

When we move a fingerprint $f(x)$ to its other cell,
we don't know which element x generated it
$\Rightarrow$ compute new cell using only current cell and $f(x)$

Fingerprints in any one cell can only go to a small number of other cells (as many as the number of different fingerprints)
$\Rightarrow$ the two cells for x cannot be chosen independently

Cuckoo hashing analysis depends on independence of pairs of cells
$\Rightarrow$ we need to prove that this works (all fingerprints can be inserted) all over again, without using independence

How to find the two homes for a fingerprint

Original version:

Choose three hash functions h_1, h_2, and f

Map each element x to fingerprint $f(x)$
with two homes $h_1(x)$ and $(h_1(x) \text{ xor } h_2(f(x)))$

When we see fingerprint f in cell with index i
its other home cell has index $(i \text{ xor } h_2(f))$

We don't need to know the x that generated it!

Works well in practice (up to same load factor as cuckoo hash)

No mathematical proof that it works!

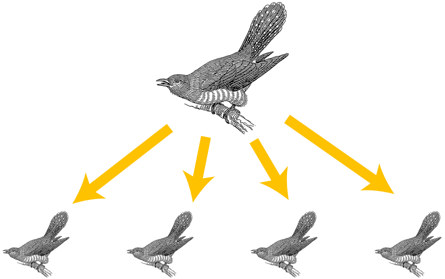
How to find the two homes for a fingerprint

Simplified version [Eppstein, SWAT 2016]:

Choose two hash functions h_1 and f

Map x to fingerprint $f(x)$ with homes $h_1(x)$ and $(h_1(x) \text{ xor } f(x))$

Effectively partitions big cuckoo hash table into many smaller ones, within which pairs of home cells are chosen independently



Can reuse random-graph analysis from cuckoo hashing!

How much space do we need?

Assume k bits per fingerprint, then

$$\begin{aligned}\Pr[\text{false positive}] &\le (\#\text{ elements that could collide}) \times \Pr[\text{collision}] \\ &= n \times \Pr[\text{same } h_1(x)] \times \Pr[\text{same } f(x)] \\ &= n \times O\left(\frac{1}{n}\right) \times \frac{1}{\#\text{ fingerprints}} \\ &= O\left(\frac{1}{2^k}\right).\end{aligned}$$

Invert this: false positive rate ε needs $k = \log_2 \frac{1}{\varepsilon} + O(1)$

Insertion analysis needs k to be nonconstant ($\epsilon = o(1)$)
$\Rightarrow$ can replace $+O(1)$ in formula for k by $\times(1 + o(1))$

Cuckoo load factor near one $\Rightarrow$ multiply space by $(1 + o(1))$

So for false positive rate $\varepsilon = o(1)$, need $(1 + o(1))n \log_2 \frac{1}{\varepsilon}$ bits

Disjointness

Disjoint set query problem

Data: a family of sets S_i

N = how many sets in the family

k = max size of any set in the family

Typical assumption: k is much smaller than N

Problem: Construct a data structure for the family to quickly answer, given query set T, whether $\exists i$ with S_i disjoint from T

Naïve solution (no data structure):

Compare T to each S_i, total query time $O(Nk)$

The real problem: Can we do better than the naïve solution?

CNF satisfiability

Disjointness can be used to solve the following problem:

Given a Boolean formula in conjunctive normal form, can we assign True/False to its variables so the whole formula becomes true?

Term: a variable or its negation

Clause: set of terms connected by Boolean or ("disjunction")

CNF: set of clauses connected by Boolean and ("conjunction")

Example: $(A \vee B \vee C) \wedge (\neg A \vee \neg B) \wedge (\neg A \vee \neg C) \wedge (\neg B \vee \neg C)$

We have to make ≥ 1 term true in every clause

E.g. set A: true, B: false, C: false

Using disjointness for satisfiability

Given CNF formula with n variables, m clauses:

1. Split variables into subsets A, B of size $\approx n/2$
2. For each truth assignment x_i of variables in A, make a set X_i of the clauses it does not satisfy
3. For each truth assignment y_j of variables in B, make a set Y_j of the clauses it does not satisfy
4. Look for a disjoint pair of sets X_i, Y_j

Number of sets $N = O(2^{n/2})$, set size $k \leq m$

See: Ryan Williams, "A new algorithm for optimal constraint satisfaction and its implications", Theor. Comp. Sci. 2005, §5.1, `https://people.csail.mit.edu/rrw/2-csp-final.pdf`

Strong exponential time hypothesis

Naïve algorithm for CNF satisfiability:
Try all 2^n truth assignments

We don't know anything significantly faster than this!

"Strong exponential time hypothesis":
There isn't anything significantly faster than this
For all $\varepsilon > 0$, not possible in time $(2-\varepsilon)^n m^{O(1)}$

Standard to assume this in complexity theory
Unproven, would imply $P \neq NP$

Implications for disjointness

Suppose we could solve disjoint set problem with

Preprocessing time $N^{2-\delta}k^{O(1)}$

Query time $N^{1-\delta}k^{O(1)}$

(That is, significantly better than naïve method)

Then using this for satisfiability would give time

$(2^{n/2})^{2-\delta}k^{O(1)} = (2-\epsilon)^n m^{O(1)}$

For some $\epsilon > 0$ whenever $\delta > 0$

SETH $\Rightarrow$ this cannot happen!

So either no better-than-naïve disjointness structure exists, or (if we find one), SETH is incorrect

Summary

Summary

- Set operations and their implementation in Python and Java
- How to combine sets using single-element operations
- Exact representations of sets using hash tables
- Exact representations of sets using bitmaps
- Filters: approximate representations of sets
- False positives versus false negatives
- Bloom filters and cuckoo filters
- Nonexistence of good data structures for disjointness