CS 261: Data Structures

Week 3: Sets

Lecture 3a: Hashing and bitmaps

David Eppstein
University of California, Irvine

Spring Quarter, 2024

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

Set size

Set size

Hashing-based data structures typically also allow you to ask how many elements are in the set

E.g. python len(S)

(Stored with hash table; needed for load factor calculation)

What about bitmap-based sets?

Example application

In chess programming, 64-bit bitmaps are also used for sets of chessboard squares

For example:

- Set of squares with white pieces on them
- Set of squares attacked by black pieces

Called "bitboards"

Evaluation of position may include weighted sums of set sizes

Naive solutions

Loop through all indexes; add one when element is present: time $O(N)$

Loop through elements of set, adding one for each: time $O(n)$ where $n =$ size

Build lookup table of number of nonzeros in a block of b bits; loop through blocks of a set, adding table value to total: time $O(N)$ but with smaller constant factor

Hardware and library support

Sometimes this is available in hardware

E.g. for Intel architecture in the SSE4 extensions (since 2007/2008, depending on chipset): POPCNT

Can be accessed by software library routines

- C++ numerics library: std::popcount
- Java integer.bitCount
- Python (3.10): int.bit_count

Bit-parallel addition

Start with a word of 1-bit blocks that you want to add (the initial set).
Then repeat: given a word of 2^i-bit blocks, mask off the blocks in even positions, mask off the blocks in odd positions, shift to align the blocks with each other and add $\Rightarrow$ 2^{i+1}-bit blocks

```
m1  = 0x5555555555555555      x = (x & m1 ) + ((x >>  1) & m1 )
m2  = 0x3333333333333333      x = (x & m2 ) + ((x >>  2) & m2 )
m4  = 0x0f0f0f0f0f0f0f0f      x = (x & m4 ) + ((x >>  4) & m4 )
m8  = 0x00ff00ff00ff00ff      x = (x & m8 ) + ((x >>  8) & m8 )
m16 = 0x0000ffff0000ffff      x = (x & m16) + ((x >> 16) & m16)
m32 = 0x00000000ffffffff      x = (x & m32) + ((x >> 32) & m32)
```

Time $O(\log N)$

With multiplication

If you have numbers in the range $0..i$ stored in 2^i-bit blocks, you can combine them into a single block by multiplying by a constant whose block structure is $1, 1, \ldots$ $\Rightarrow$ can grow blocks more quickly instead of only doubling their size

In theory this gives $O(\log^* N)$,
height of tower of powers of two, $2^{2^{2\cdots}} = N$

In practice better to combine with addition, $O(\log \log N)$:

```
x = (x & m1 ) + ((x >>  1) & m1 )
x = (x & m2 ) + ((x >>  2) & m2 )
x = (x & m4 ) + ((x >>  4) & m4 )
return (x * 0x0101010101010101) >> 56
```

Disjointness

Overview

Data: a family of many small sets

Goal: given a query set, find a data set as far as possible from it

More specifically: find a data set that is disjoint from the query
(this means they have no elements in common)

Without even using a data structure for the data, we can answer
queries in time linear in the size of the data set

We will see evidence that:
no data structure can perform significantly better

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

To test whether T is disjoint from any set S_i:

- Build an exact set data structure for T
- For each set S_i, loop through its elements checking that they do not belong to T
- If we ever find an element in T, move on to the next set
- If we get through all the elements of S_i, then S_i is disjoint from T

If there are N sets S_i, each of size $\leq k$, total time is $O(|T| + Nk)$.

Dominant term in the time bound: N

Using disjointness for satisfiability

The satisfiability 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")

CNF example

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

One possible solution: set A: true, B: false, C: false

Why?

Satisfiability is NP-complete; completeness means that it provides a unifying language in which many other hard computational problems can be formulated

Modern satisfiability solvers are very powerful and can be used in practice to solve many of these problems, even though theoretically they are hard

We would like to understand the theoretical complexity of this problem

Faster disjoint sets would lead to faster theoretical solutions

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. Build a data structure of all the X_i's and query each Y_j to find disjoint 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`

Analysis of this method

The number of sets in our data structure (truth assignments to A) is $N = O(2^{n/2})$

Each one is a subset of clauses (the clauses it does not satisfy): $k \leq m$

The number of queries (truth assignments to B) is also $O(2^{n/2})$

With query time $\approx N$ we get total time $\approx 2^n$

But better disjointness queries would give faster solutions!

The exponential time hypothesis

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