Lecture 24: Compression
How do we store big data?
How do we represent data?

• So far:
  – So that it is easy to manipulate/process!

• Another consideration:
  – short

Today’s goal!
How can we compress data?

• Bad news: In general, we can’t … not even by one bit:
  – To represent every m-bit string with at most (m-1) bits: each m-bit string should be represented by a different \((\leq m-1)\)-bit string
  – There are \(2^m\) m-bit strings,
  – But there are only \(2^m-1\) different \((\leq m-1)\)-bit strings!
What if we know *something more* about our data?

• Some examples:

  – Frequency of characters, certain sequences differ
    • E.g. Some English letters are uncommon, so use larger strings to encode them

  – Data represents certain type of object – take advantage of characteristics of the object
    • Picture, movie, music
    • Set
    • Graph
Data compression: two paradigms

• **Lossless:**
  – Can get back the original sequence perfectly
    • Run-length encoding
    • Huffman-coding
    • Lempel-Ziv

• **Lossy:**
  – Keep enough information for the task, but not everything
    • Pictures, video, music, ....: .jpg, .mpg, .mp3
    • Wavelets
    • Bloom filters
Lossless compression: a few words
Lossless compression: a few buzzwords and very high level ideas
Run length encoding

• Given bits:
  00000011100011000000000000

• Encode by lengths of successive runs:
  6 0’s, 3 1’s, 3 0’s, 2 1’s, 9 0’s
  (Actually, need only store vector 6,3,3,2,9)

Great for fax machines!
Also used in jpeg
Huffman coding

• Variable length encoding –
  – Use shortest strings to encode most frequent letters
    • E.g. Use 3 bits to encode “a” and “e”, 4 bits for “f”, “s”, “m” and 5 bits to encode “z”, “x”
  – Need prefix-free property
    • Bad if $a \to 00$, $b \to 111$ and $z \to 00111$
      Is “00111” decoded to “ab” or “z”?

See CLRS Chapter 16.3 for more details!
Lempel-Ziv

- Lempel Ziv
  - Use pointers to previous places where saw the substring
    - i.e., “copy the string starting at location 10 for the next 20 letters” --- works well if cheaper to represent (10,20) than the values of 20 letters.
  - Many variants (LZ77, LZ78, LZW...)

Compressing graphs

• Adjacency matrix: $n^2$ bits
• Adjacency list: $m \log n$ bits

• In general, can’t do much better
  – There are $2^\frac{n^2}{2}$ many graphs, each needing a distinct representation, so need $\frac{n^2}{2}$ bits
  – Can also give counting argument for graphs with $m$ edges
Compressing Web graphs

• Many nodes have similar links
  – Can express u’s node as “copy v’s links and make the following modifications”

• Destinations of links exhibit locality
  – Can use small integers to express destinations relative to source of link
  – e.g. grid graph + short edges:
    • nodes are pairs (x,y)
    • (x,y) connected to (x ± 1, y ± 1)
    • (x,y) also connected to (x+d,y+e) for constant d,e – can represent edge by log d + log e bits instead of 2 log n bits.
Lossy compression
Images/Movies/...

• Store via Fourier representation
  – Do we really care about the “high order Fourier coefficients”?
  – Drop undetectable information....
Lossy compression: storing sets
Bloom filters

- Data structure for representing a set to support membership queries
  (+) Save lots of space
  (-) false positives
Bloom filter

- Goal: Maintain set \( S = \{x_1, x_2, ... x_n\} \) of \( n \) elements

- Bloom filter:
  - array of \( m \) bits
  - Given \( k \) independent hash functions \( h_1, ..., h_k \) mapping \( x_i \)'s to \( \{1, ..., m\} \)
  - Storage Algorithm:
    - initially all \( m \) bits are 0
    - For each \( x_i \in S \), for each \( 1 \leq j \leq k \), set bit \( h_j(x_i) \) to 1
  - On query \( y \):
    - For each \( 1 \leq j \leq k \), check if bit \( h_j(x_i) \) is 1
    - If yes for all \( j \), output “in \( S \)”
    - else output “not in \( S \)”
A nice property:

- union of two sets represented by Bloom filters of size $m$ (same hash functions)
  - Take “or” of the bits
Applications of Bloom Filters: Dictionaries

• Unix spell-checker:
  – Store dictionary in BF
  – While spell-checking, look up each work in BF
    • False positive causes you to ignore misspelled word

• Unsuitable passwords:
  – Store dictionary + words of edit distance 1 in BF
  – Don’t allow passwords that seem to be in BF
    • Might not allow you to use a perfectly good password
Other applications:

- Databases: speed up semi-join
- Distributed caching: find which (if any) of cooperating caches holds a web page
- P2P networks: locate objects
- Network routing

- And many many many many more...
Compression

• Lot’s more!
  – Ways to compress
  – Applications
    • Better storage
    • Cheaper communication
    • ...
    • Maybe surprisingly:
      – Connected to learnability, randomness