Objectives

- Clustering
- Encoding

Review: Our Problem Solving Process

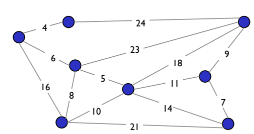
1. Understand/identify problem
   - Simplify as appropriate
2. Design a solution
3. Analyze
   - Correctness, efficiency
   - May need to go back to step 2 and try again
4. Implement
   - Within bounds shown in analysis

Review

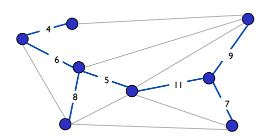
- What is a minimum spanning tree?
- What are some algorithms to find an MST?

Review: Minimum Spanning Tree

- Spanning tree: spans all nodes in graph
- Given a connected graph G = (V, E) with positive edge weights c_e, an MST is a subset of the edges $T \subseteq E$ such that T is a *spanning tree* whose **sum of edge weights** is *minimized*



G = (V, E)

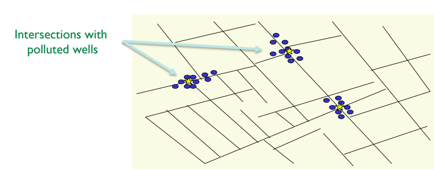


T, $\Sigma_{e \in T}\, c_e = 50$

Review: Greedy Algorithms

All three algorithms produce a MST

- Prim's algorithm. Start with some root node *s* and greedily grow a tree *T* from *s* outward. At each step, add the cheapest edge *e* to *T* that has exactly one endpoint in *T*.
  - Similar to Dijkstra's (but simpler)
- Kruskal's algorithm. Start with $T = \phi$. Consider edges in ascending order of cost. Insert edge *e* in *T* unless doing so would create a cycle.
- Reverse-Delete algorithm. Start with T = E. Consider edges in descending order of cost. Delete edge *e* from *T* unless doing so would disconnect *T*.



Outbreak of cholera deaths in London in 1850s.
Reference: Nina Mishra, HP Labs

CLUSTERING

Clustering

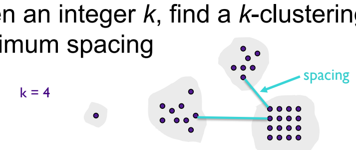
- Given a set *U* of *n* objects (or points) labeled $p_1, \ldots, p_n$, classify into coherent groups
  - Problem: Divide objects into clusters so that points in different clusters are far apart
    - Requires quantification of distance
- Applications
  - Routing in mobile ad hoc networks
  - Identify patterns in gene expression
  - Identifying patterns in web application use cases
    - Sets of URLs
  - Similarity searching in medical image databases

Clustering: Distance Function

- Numeric value specifying "closeness" of two objects
- Assume distance function satisfies several natural properties
  - $d(p_i, p_j) = 0$ iff $p_i = p_j$ (identity of indiscernibles)
  - $d(p_i, p_j) \geq 0$ (nonnegativity)
  - $d(p_i, p_j) = d(p_j, p_i)$ (symmetry)

Our Problem: k-Clustering of Maximum Spacing

- k-clustering. Divide objects into *k* non-empty groups
- Spacing. Min distance between any pair of points in different clusters
- k-clustering of maximum spacing. Given an integer *k*, find a *k*-clustering of maximum spacing



Ideas about solving?

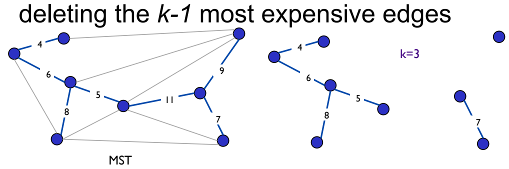
Greedy Clustering Algorithm

- Single-link *k*-clustering algorithm
  - Form a graph on the vertex set *U*, corresponding to *n* clusters
  - Find the closest pair of objects such that *each object is in a different cluster* and add an edge between them
  - Repeat *n-k* times until there are exactly *k* clusters

How is this related to the MST?

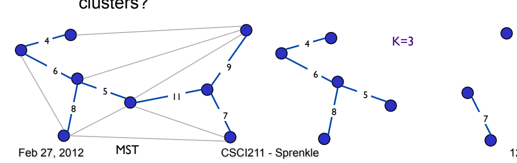
Greedy Clustering Algorithm

- Key observation. Same as Kruskal's algorithm
  - Except we stop when there are *k* connected components
- Remark. Equivalent to finding MST and deleting the *k-1* most expensive edges



Greedy Clustering Algorithm: Analysis

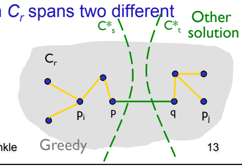
- Theorem. Let C denote the clustering $C_1, \ldots, C_k$ formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of *max spacing*.
- Pf Intuition:
  - What can we say about C's spacing?
    - Within clusters and between clusters
  - What if C isn't optimal?
    - What does that mean about C's clusters vs (optimal) C*'s clusters?



Greedy Clustering Algorithm: Analysis

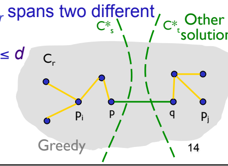
- Theorem. Let C denote the clustering $C_1, \ldots, C_k$ formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of ***maximum spacing***.
- Pf Sketch. Let C* denote some other clustering $C^*_1, \ldots, C^*_k$. C* and C must be different; otherwise we're done.
  - The spacing of *C* is length *d* of (k-1)st most expensive edge
  - Let p_i, p_j be in the same cluster in Greedy solution *C* (say C_r) but different clusters in other solution C*, say C^*_s and C^*_t
  - Some edge (*p*, *q*) on p_i-p_j path in C_r spans two different clusters in *C**

What do we know about (*p*, *q*)?



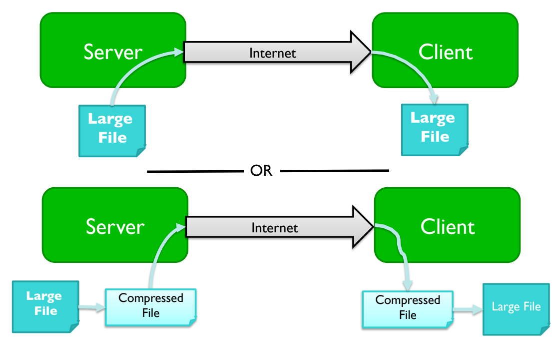
Greedy Clustering Algorithm: Analysis

- Theorem. Let C denote the clustering $C_1, \ldots, C_k$ formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of ***maximum spacing***.
- Pf. Let C* denote some other clustering $C^*_1, \ldots, C^*_k$. C* and C must be different; otherwise we're done.
  - The spacing of *C* is length *d* of (k-1)st most expensive edge
  - Let p_i, p_j be in the same cluster in *C* (say C_r) but different clusters in C*, say C^*_s and C^*_t
  - Some edge (*p*, *q*) on p_i-p_j path in C_r spans two different clusters in *C**
  - All edges on p_i-p_j path have length $\leq d$ since Kruskal chose them
  - Spacing of C* is at most $\leq$ d since *p* and *q* are in different clusters



IMPROVING TRANSMISSION SPEEDS

Which Is Better?

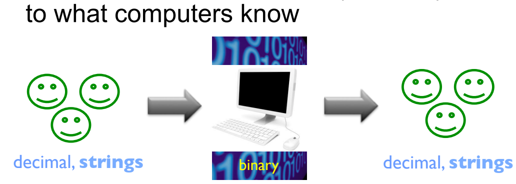


Discussion: Which Is Better?

- Depends on your metrics, compression time/amount
- Case 1 requires
  - More network resources
  - Less CPU time (server: compress; client: uncompress)
- Case 2 requires
  - Less network resources
  - More CPU time (client and server)
- Overall best
  - Depends on file size, network speed, compression time/amount
  - Bigger files → Case 2

Problem: Encoding

- Computers use bits: 0s and 1s
- Need to represent what we (humans) know to what computers know



- Map **symbol** → unique sequence of 0s and 1s
- Process is called *encoding*

Problem: Encoding

- Let's say we want to encode characters using 0s and 1s
  - Lower case letters (26)
  - Space
  - Punctuation (, . ? ! ')

What is the **least** number of bits we would we need to encode these characters?

Feb 27, 2012 CSCI211 - Sprenkle 19

Problem: Encoding Symbols

- 32 characters to encode
  - $\log_2(32)$ = 5 bits
  - Can't use fewer bits
- Examples:
  - a → 00000
  - b → 00001
- Actual mapping from character to encoding doesn't matter
  - Easier if have a way to compare …

Feb 27, 2012 CSCI211 - Sprenkle 20

For Long Strings of Characters…

- Do we need an average of 5 bits/character always?
- What if we could use *shorter encodings* for *frequently* used characters, like a, e, s, t?

Goal: Optimal encoding that takes advantage of *nonuniformity* of letter frequencies

- A fundamental problem for **data compression**
  - Represent data *as compactly as possible*

Feb 27, 2012 CSCI211 - Sprenkle 21

Example: Morse Code

- Used for encoding messages over telegraph
- Example of *variable-length encoding*

How are letters encoded?
How are letters differentiated?

Feb 27, 2012 CSCI211 - Sprenkle 22

Example: Morse Code

- Used for encoding messages over telegraph
- Example of *variable-length encoding*
- How are letters encoded?
  - Dots, dashes
  - Most frequent letters use shorter sequences
    - e → dot; t → dash; a → dot-dash
- How are letters differentiated?
  - Spaces in between letters
    - Otherwise, ambiguous

Feb 27, 2012 CSCI211 - Sprenkle 23

Ambiguity in Morse Code

- Encoding:
  - e → dot; t → dash; a → dot-dash
- Example: dot-dash-dot-dash could correspond to

Feb 27, 2012 CSCI211 - Sprenkle 24

Ambiguity in Morse Code

- Encoding:
  - e → dot; t → dash; a → dot-dash
- Example: dot-dash-dot-dash could correspond to
  - etet
  - aa
  - eta
  - aet

What's the problem?

Problem

- Ambiguity caused by encoding of one character is a *prefix* of encoding of another

Prefix Codes

- Problem: Encoding of one character is a *prefix* of encoding of another
- Solution: **Prefix Codes**: map letters to bit strings such that *no encoding is a prefix of any other*
  - Won't need artificial devices like spaces to separate characters
- Example encodings:
  - Verify that no encoding is a prefix of another
  - What is `0010000011101`?

```
a: 11     d: 10
b: 01     e: 000
c: 001
```


Optimal Prefix Codes

- For typical English messages, this set of prefix codes is **not** the *optimal* set

```
a: 11     d: 10
b: 01     e: 000
c: 001
```

- Why not?

Optimal Prefix Codes

- For typical English messages, this set of prefix codes is **not** the *optimal* set

```
a: 11     d: 10
b: 01     e: 000
c: 001
```

- Why not?
  - 'e' is more commonly used than other letters and should therefore have a shorter encoding

Optimal Prefix Codes

- **Goal**: minimize **Average number of Bits per Letter (ABL):**

  $\Sigma_{x \in S}$frequency of x * length of encoding of x

  For all characters in our alphabet
- f_x: frequency that letter x occurs
- $\gamma(x)$: encoding of x
  - $|\gamma(x)|$: length of encoding of x
- Minimize **ABL =** $\boxed{\Sigma_{x \in S} f_x \, |\gamma(x)|}$

Example: Calculating ABL

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```

- **ABL** = $\Sigma_{x \in S} f_x \, |\gamma(x)|$ = ?

handout

Example: Calculating ABL

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```

- **ABL** = $\Sigma_{x \in S} f_x \, |\gamma(x)|$ = ?
- = .32 * 2 + .25 * 2 + .20 * 3 + .18 * 2 + .05 * 2
- = 2.25

Consider a fixed-length encoding:
Is it a prefix code? What is its ABL?

Fixed-Length Encodings

- Is it a prefix code?
  - Yes. Always look at fixed number of characters
- What is its ABL?
  - ABL is the length of the encoding
- For 5 characters, ABL is 3
- Variable-length prefix code's ABL (2.25) is an improvement

Can We Improve the ABL?

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```


Can We Improve the ABL?

$f_a = .32$
$f_b = .25$
$f_c = .20$
$f_d = .18$
$f_e = .05$

```
a: 11
b: 01
c: 001
d: 10
e: 000
```

Swap these because *c* occurs more frequently than *d*.
Give *c* the shorter encoding

- **ABL** = $\Sigma_{x \in S} f_x \, |\gamma(x)|$ = 2.23

Problem Statement

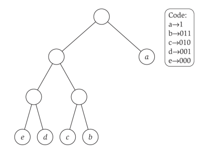
- Given an alphabet and a set of frequencies for the letters, produce optimal (most efficient) prefix code
  - Minimizes average # of bits per letter (ABL)

Approaches to Solution

- Brute force
  - ➢ Search space is complicated → all ways to map letters to bit strings that adhere to prefix code property
- Build towards greedy approach
  - ➢ Start: representing prefix codes
    - Given we know the codes, how do we represent them?

Binary Trees to Represent Prefix Codes

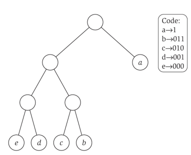
- Exposes structure better than list of mappings
  - ➢ Each leaf node is a letter
  - ➢ Follow path to the letter
    - Going left: 0
    - Going right: 1



Are these really prefix codes?
How could we show they weren't?

Binary Trees to Represent Prefix Codes

- Proof. If it weren't: a letter's encoding is a prefix of another letter
  - ➢ Letter is in the path of another letter
  - ➢ But, all letters are leaf nodes
    - Contradiction



Building the Binary Tree

- How do we build the binary tree for this mapping?

Code:
a→1
b→011
c→010
d→001
e→000

- Tree Rules:
  - ➢ Each leaf node is a letter
  - ➢ Follow path to the letter
    - Going left: 0
    - Going right: 1

Recursively Generate Tree

- All letters are in root node
- For all letters in node
  - ➢ If encoding begins with 0, letter belongs in left subtree
  - ➢ Otherwise (encoding begins with 1), letter belongs in right subtree
  - ➢ If last bit of encoding, make the letter a leaf node of that subtree
  - ➢ Shift encoding one bit
  - ➢ Process left and right children

Exam Feedback

- Median: 82; Average: 84
- Systematic write up
  - ➢ One student had subsections: Problem, Solution, Efficiency Analysis, Why it works

Assignments

- Wiki due Wed night
  - Beginning of Chapter 4 (before 4.1)
  - 4.1 – 4.6, excluding 4.3
- PS 5 due next Monday in class
  - Sheet says Friday