Objectives

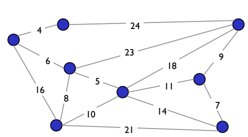
- Clustering
- Encoding

Review

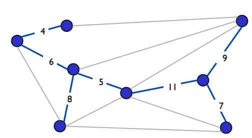
- What is a minimum spanning tree?
- What are some efficient algorithms to find an MST?
- What two properties did we prove that would helps us to prove that these algorithms are correct?

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

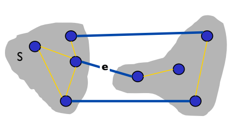
Review: Greedy Algorithms

All three algorithms produce a MST

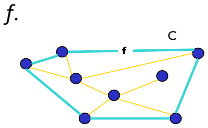
- Prim's algorithm. Start with some root node *s* and greedily grow a tree *T* from *s* outward. At each step, add the cheapest edge *e* to *T* that has exactly one endpoint in *T*.
  - Similar to Dijkstra's (but simpler)
- Kruskal's algorithm. Start with $T = \phi$. Consider edges in ascending order 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*.

Summary of What Just Proved

- Simplifying assumption: All edge costs c_e are distinct
  - ➜ MST is unique
- Cut property. Let *S* be any subset of nodes, and let *e* be the min cost edge with exactly one endpoint in *S*. Then MST contains *e*.
- Cycle property. Let *C* be any cycle, and let *f* be the max cost edge belonging to *C*. Then MST does not contain *f*.



Cut Property: e is in MST



Cycle Property: f is **not** in MST

Kruskal's Algorithm [1956]

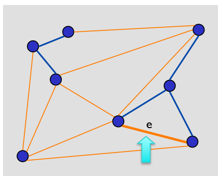
- Start with $T = \phi$
- Consider edges in *ascending order of cost*
- Insert edge *e* in T unless doing so would create a cycle
  - Add edge as long as it is "compatible"

How can we prove algorithm's correctness?

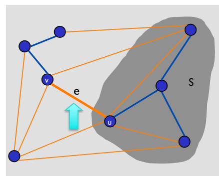
Kruskal's Algorithm: Proof of Correctness

What is tricky about implementing Kruskal's algorithm?

- Consider edges in ascending order of weight
- Case 1: If adding *e* to T creates a cycle, discard *e* according to ***cycle property*** (*e* must be max weight)
- Case 2: Otherwise, insert *e* = (u, v) into T according to ***cut property*** where *S* = set of nodes in u's *connected component*



Case 1



Case 2

Implementing Kruskal's Algorithm

What is tricky about implementing Kruskal's algorithm?

How do we know when adding an edge will create a cycle?
- What are the properties of a graph/its nodes when adding an edge will create a cycle?

UNION-FIND DATA STRUCTURE

Union-Find Data Structure

- Keeps track of a graph as edges are added
  - Cannot handle when edges are deleted
- Maintains disjoint sets
  - E.g., graph's connected components
- Operations:
  - `Find(u)`: returns name of set containing u
    - How utilized to see if two nodes are in the same set?
    - Goal implementation: **O(log n)**
  - `Union(A, B)`: merge sets A and B into one set
    - Goal implementation: **O(log n)**

Best darn Union-Find Data Structure

Implementing Kruskal's Algorithm

- Using the **union-find** data structure
  - Build set T of edges in the MST
  - Maintain set for each connected component

Implementation?

```
Sort edges weights so that c1 ≤ c2 ≤ ... ≤ cm
T = {}
foreach (u ∈ V) make a set containing singleton u

for i = 1 to m
   (u,v) = ei
   if (u and v are in different sets)
      T = T ∪ {ei}
      merge the sets containing u and v
return T
```

are u and v in different connected components?

merge two components

Implementing Kruskal's Algorithm

- Using the **union-find** data structure
  - Build set T of edges in the MST
  - Maintain set for each connected component

Costs?

```
Sort edge weights so that c1 ≤ c2 ≤ ... ≤ cm
T = {}
foreach (u ∈ V) make a set containing singleton u

for i = 1 to m
   (u,v) = ei
   if ( Find(u) != Find(v) )
      T = T ∪ {ei}
      Union( Find(u), Find(v) )
return T
```

are u and v in different connected components?

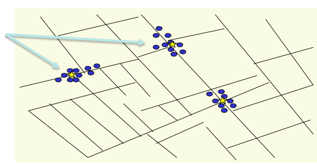
merge two components

Implementing Kruskal's Algorithm

- Using best implementation of union-find
  - Sorting: O(m log n) ⟵ $m \le n^2 \Rightarrow \log m$ is O(log n)
  - Union-find: O(m α (m, n)) — α (m, n): essentially a constant
  - O(m log n)

```
Sort edge weights so that c1 ≤ c2 ≤ ... ≤ cm      O(m logn)
T = {}
foreach (u ∈ V) make a set containing singleton u

for i = 1 to m   O(m)   are u and v in different connected components?
   (u,v) = ei
   if ( Find(u) != Find(v) )   O(logn)
      T = T ∪ {ei}
      Union( Find(u), Find(v) )   O(logn)
return T
                         merge two components
```

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

CLUSTERING

Clustering

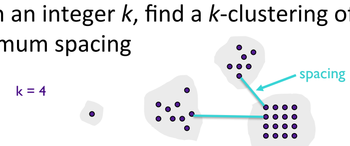
- Given a set *U* of *n* objects (or points) labeled $p_1, \ldots, p_n$, classify into coherent groups
  - Problem: Divide objects into clusters so that points in different clusters are far apart
    - Requires quantification of distance
- Applications
  - Routing in mobile ad hoc networks
  - Identify 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)
  - $d(p_i, p_j) \ge 0$ (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?

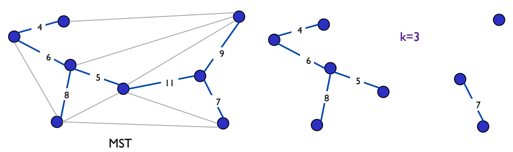
Greedy Clustering Algorithm

- 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?

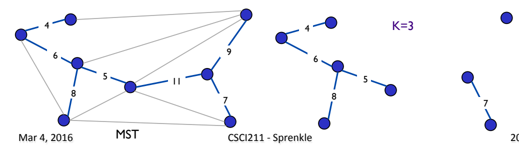
Greedy Clustering Algorithm

- 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

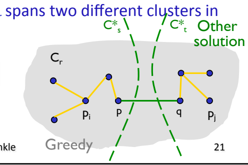
- Theorem. Let C denote the clustering $C_1, \ldots, C_k$ formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of *max spacing*.
- Pf Intuition:
  - What can we say about C's spacing?
    - Within clusters and between clusters
  - What if C isn't optimal?
    - What does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



Greedy Clustering Algorithm: Analysis

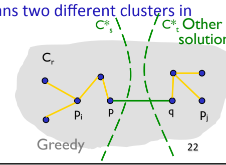
- Theorem. Let C denote the clustering $C_1, \ldots, C_k$ formed by deleting the *k-1* most expensive edges of a MST. C is a *k*-clustering of *maximum spacing*.
- Pf Sketch. Let C* denote some other clustering $C^*_1, \ldots, C^*_k$. C* and C must be different; otherwise we're done.
  - The spacing 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*)?



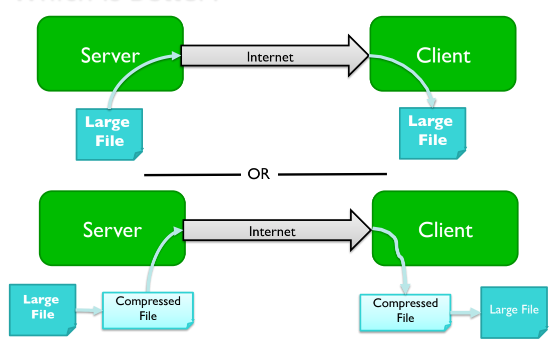
Greedy Clustering Algorithm: Analysis

- 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 ≤ *d* since Kruskal chose them
  - Spacing of C* is at most ≤ d since *p* and *q* are in different clusters



IMPROVING TRANSMISSION SPEEDS

Which Is Better?

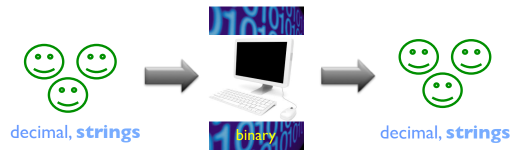


Discussion: Which Is Better?

- 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?

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 …

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*

Example: Morse Code

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

How are letters encoded?
How are letters differentiated?

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
    - adds one more character to each letter

Ambiguity in Morse Code

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

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 cause of the ambiguity?

Problem

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

Prefix Codes

- Problem: Encoding of one character being a *prefix* of encoding of another → ambiguity
- 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
```


Assignments

- Wiki due Mon night
  - 4.5-4.8
- PS 6 due next Friday in class