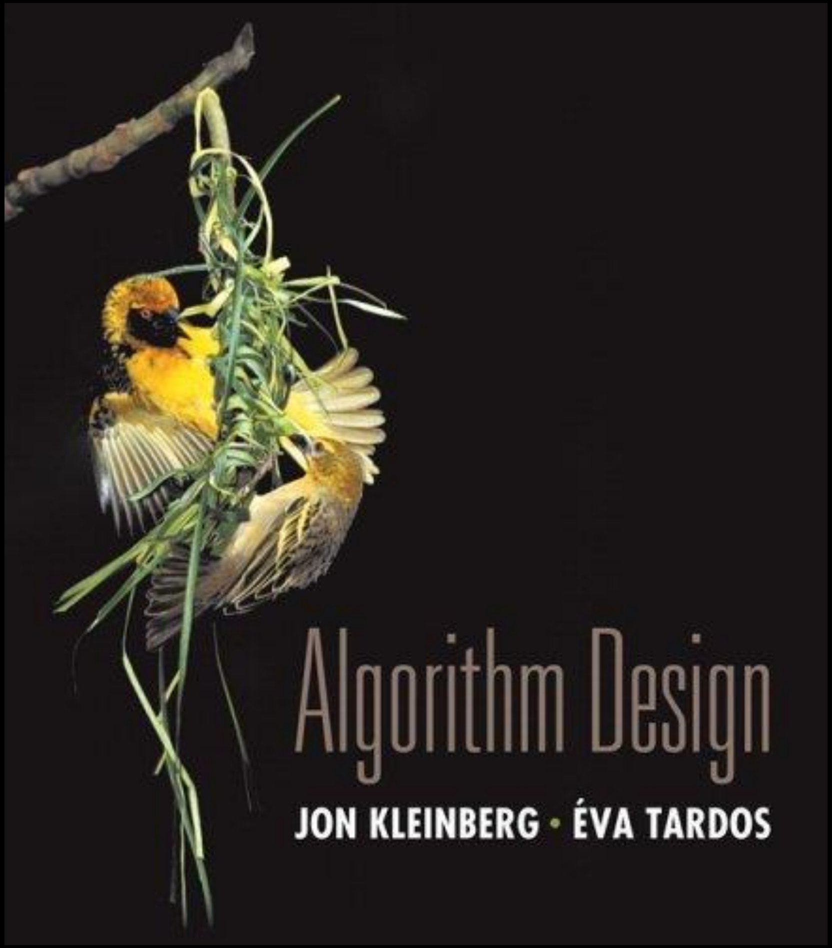


Chapter 4

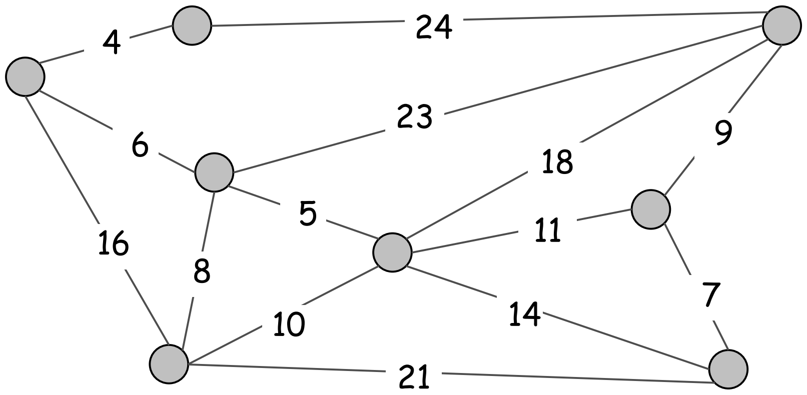
Greedy Algorithms : Part II

CS 350 Winter 2018

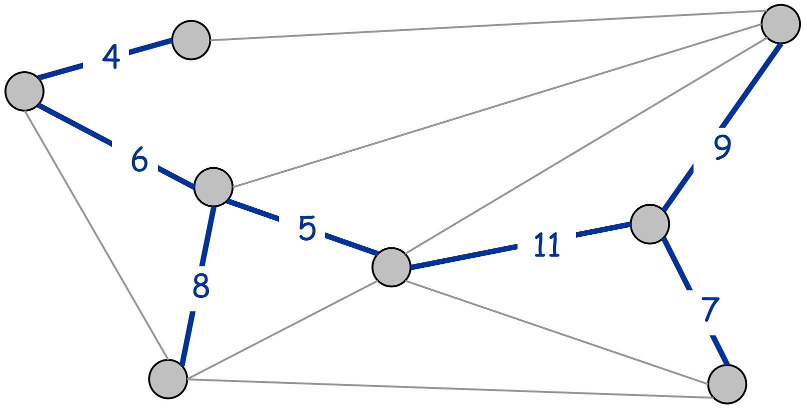
4.5 Minimum Spanning Tree

Minimum Spanning Tree

Minimum spanning tree. Given a connected graph $G = (V, E)$ with real-valued 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 ("spanning" means that the tree encompasses all vertices in G.)



$G = (V, E)$



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

Cayley's Theorem. There are n^{n-2} spanning trees of K_n (up to node labeling).

$\uparrow$ can't solve by brute force

Applications

MST is fundamental problem with diverse applications.

- Network design.
  - telephone, electrical, hydraulic, TV cable, computer, road
- Approximation algorithms for NP-hard problems.
  - traveling salesperson problem, Steiner tree
- Indirect applications.
  - max bottleneck paths
  - LDPC codes for error correction
  - image registration with Renyi entropy
  - learning salient features for real-time face verification
  - reducing data storage in sequencing amino acids in a protein
  - model locality of particle interactions in turbulent fluid flows
  - autoconfig protocol for Ethernet bridging to avoid cycles in a network
- Cluster analysis.

Greedy Algorithms

Kruskal's algorithm. Start with T = ϕ. 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.

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.

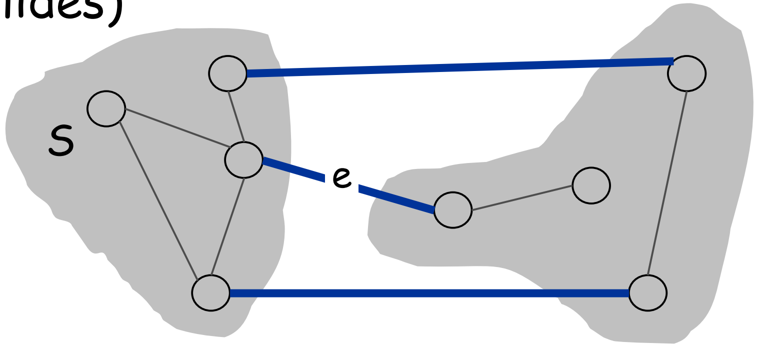
Remark. All three algorithms produce an MST.

Greedy Algorithms

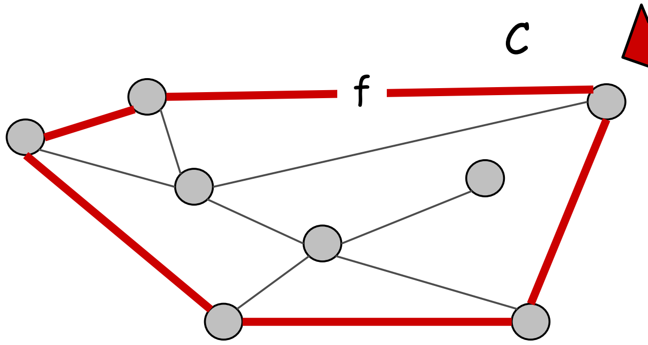
Simplifying assumption. All edge costs c_e are distinct (makes math cleaner, but it doesn't change the general proof drastically).

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST contains e. (Why? Proof later in slides)

Cycle property. Let C be any cycle, and let f be the max cost edge belonging to C. Then the MST does not contain f. (Why? Proof later in slides)



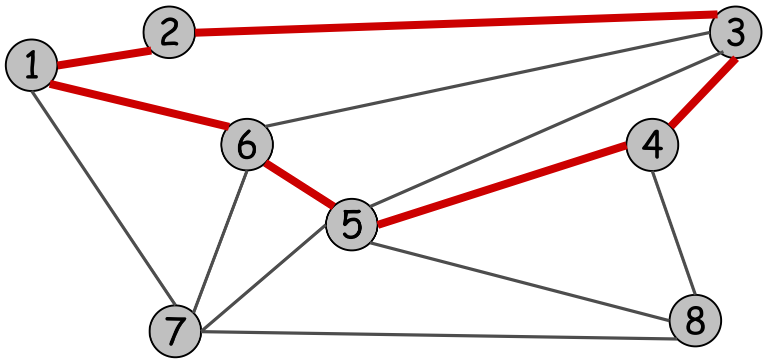
e is in the MST



f is not in the MST

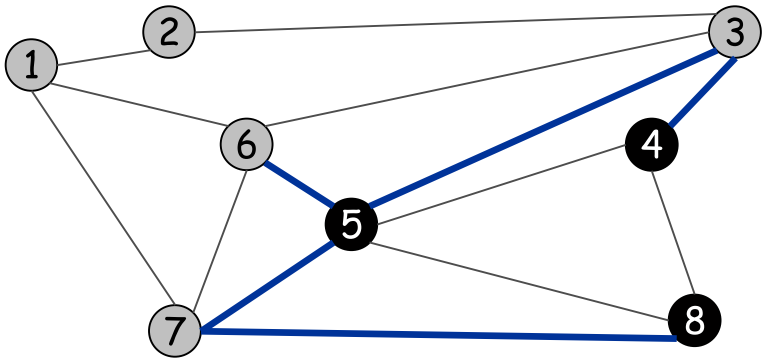
Cycles and Cuts

Cycle. Set of edges the form a-b, b-c, c-d, …, y-z, z-a.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1

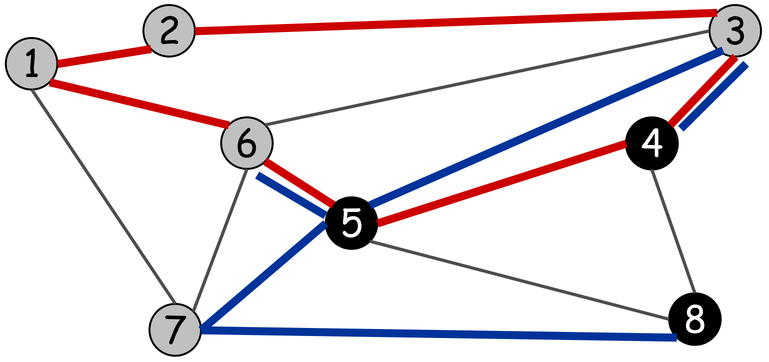
Cutset. A cut is a subset of nodes S. The corresponding cutset D is the subset of edges with exactly one endpoint in S.



Cut S = { 4, 5, 8 }
Cutset D = 5-6, 5-7, 3-4, 3-5, 7-8

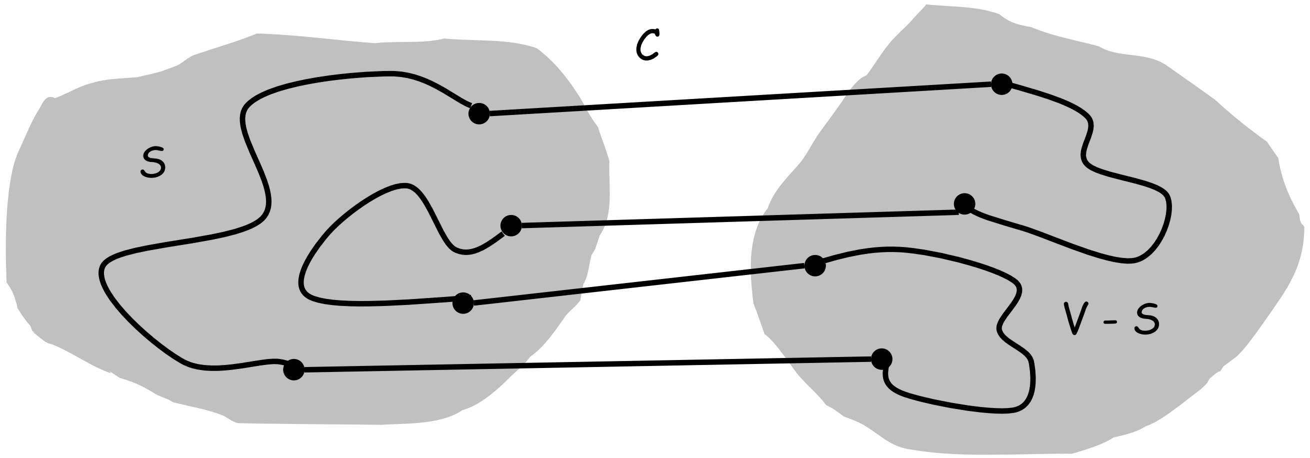
Cycle-Cut Intersection

Claim. A cycle and a cutset intersect in an even number of edges.



Cycle C = 1-2, 2-3, 3-4, 4-5, 5-6, 6-1
Cutset D = 3-4, 3-5, 5-6, 5-7, 7-8
Intersection = 3-4, 5-6

Pf. (by picture)



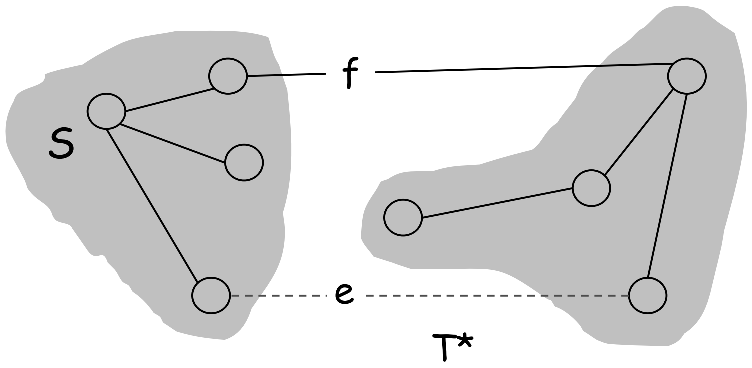
Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Cut property. Let S be any subset of nodes, and let e be the min cost edge with exactly one endpoint in S. Then the MST T* contains e.

Pf. (exchange argument, contradiction)

- Suppose e does not belong to T*, and let's see what happens.
- Adding e to T* creates a cycle C in T* (why?).
- Edge e is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say f, that is in both C and D.
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree (why?).
- Since $c_e < c_f$, cost(T') < cost(T*).
- This is a contradiction. ▪



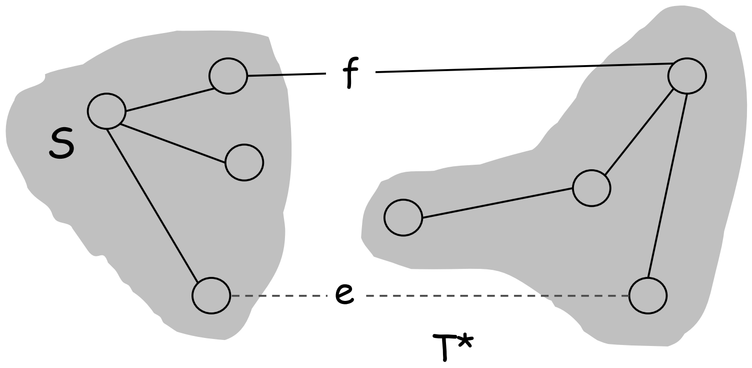
Greedy Algorithms

Simplifying assumption. All edge costs c_e are distinct.

Cycle property. Let C be any cycle in G, and let f be the max cost edge belonging to C. Then the MST T* does not contain f.

Pf. (exchange argument, contradiction)

- Suppose f belongs to T*, and let's see what happens.
- Deleting f from T* creates a cut S in T* (why?).
- Edge f is both in the cycle C and in the cutset D corresponding to S $\Rightarrow$ there exists another edge, say e, that is in both C and D.
- $T' = T^* \cup \{ e \} - \{ f \}$ is also a spanning tree.
- Since $c_e < c_f$, $\text{cost}(T') < \text{cost}(T^*)$.
- This is a contradiction. ▪

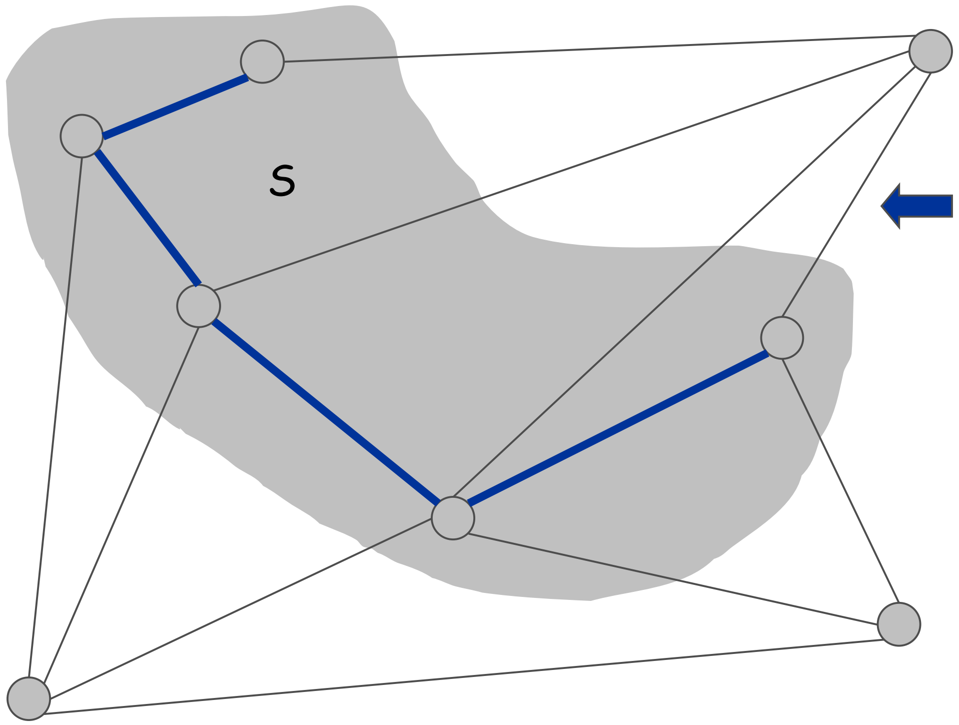


Prim's Algorithm: Proof of Correctness

Prim's algorithm. [Jarník 1930, Dijkstra 1957, Prim 1959]

- Initialize S = any node.
- Apply cut property to S.
- Add min cost edge in cutset corresponding to S to T, and add one new explored node u to S.

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.



Implementation: Prim's Algorithm

Implementation. Use a priority queue ala Dijkstra.

- Maintain set of explored nodes S.
- For each unexplored node v, maintain attachment cost a[v] = cost of cheapest edge v to a node in S.
- $O(n^2)$ with an array; $O(m \log n)$ with a binary heap.

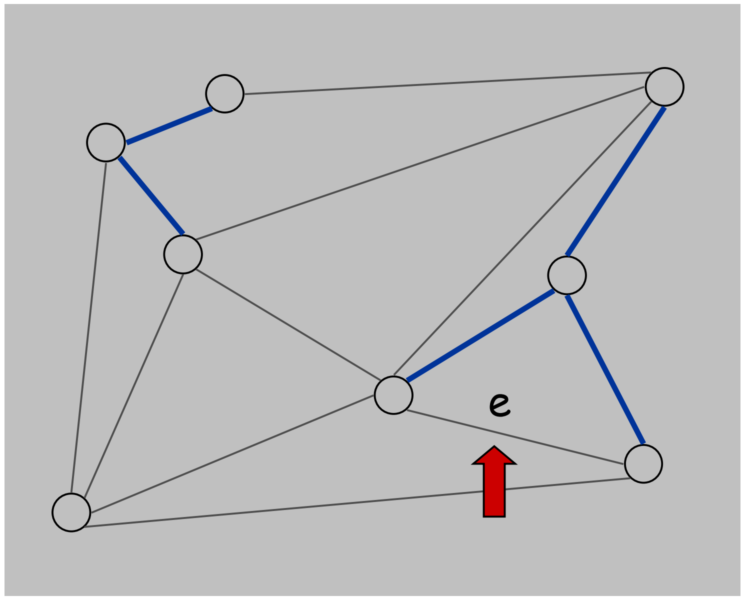
```
Prim(G, c) {
   foreach (v ∈ V) a[v] ← ∞
   Initialize an empty priority queue Q
   foreach (v ∈ V) insert v onto Q
   Initialize set of explored nodes S ← ϕ

   while (Q is not empty) {
      u ← delete min element from Q
      S ← S ∪ { u }
      foreach (edge e = (u, v) incident to u)
          if ((v ∉ S) and ($c_e$ < a[v]))
             decrease priority a[v] to $c_e$
}
```

Kruskal's Algorithm: Proof of Correctness

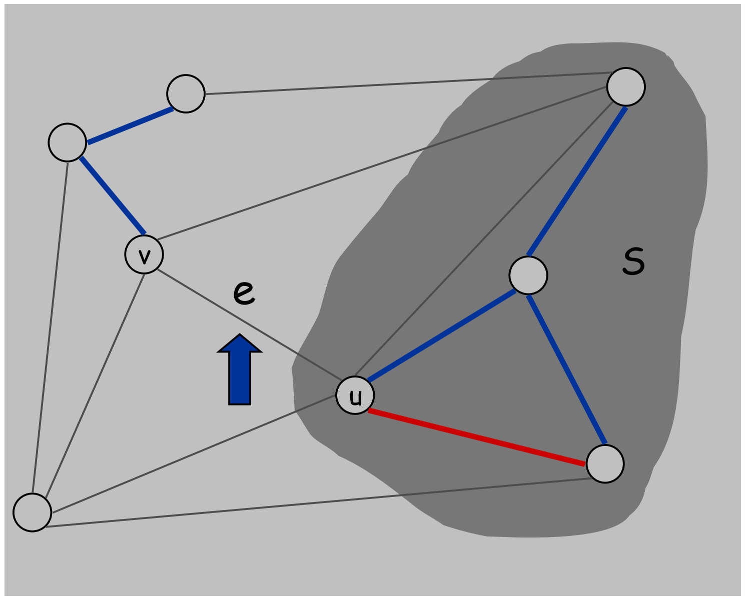
Kruskal's algorithm. [Kruskal, 1956]

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

Kruskal's algorithm. Start with T = ϕ. Consider edges in ascending order of cost. Insert edge e in T unless doing so would create a cycle.



Case 1



Case 2

Aside: Union-Find Data Structure

Union-Find Data Structure. We can maintain disjoint sets (e.g. components of a graph) with the union-find data structure; in particular, it is used to handle the effects of adding edges (not deletions).

(*) Find(u) operation returns the name of set containing node u.

(*) We can then easily test if two nodes are in the same component by simply checking Find(u)==Find(v).

(*) If we add edge (u,v) to the graph, we first test if u and v are already in the same connected component. If they are not, then Union(Find(u),Find(v)) can be used to merge the two components into one.

(*) Operations: (1) MakeUnionFind(S) (initializes structure for no edges); (2) Find(u) (can run in O(log n)); (3) Union(A,B) (can run in O(log n).

(*) We can instantiate U-F data structure with an array: "Component": Component[s] returns the name of set containing s.

Implementation: Kruskal's Algorithm

Implementation. Use the union-find data structure.

- Build set T of edges in the MST.
- Maintain set for each connected component.
- $O(m \log n)$ for sorting and $O(m\, \underbrace{\alpha\,(m, n)}_{\text{essentially a constant}})$ for union-find.

$m \le n^2 \Rightarrow \log m$ is $O(\log n)$

- Demo: https://visualgo.net/en/mst

```
Kruskal(G, c) {
   Sort edges weights so that $c_1 \le c_2 \le ... \le c_m$.
   T ← φ

   foreach (u ∈ V) make a set containing singleton u

   for i = 1 to m        are u and v in different connected components?
      (u,v) = $e_i$
      if (u and v are in different sets) {
         T ← T ∪ {$e_i$}
         merge the sets containing u and v
      }                   merge two components
   return T
}
```

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts (i.e. ε)to break any ties.

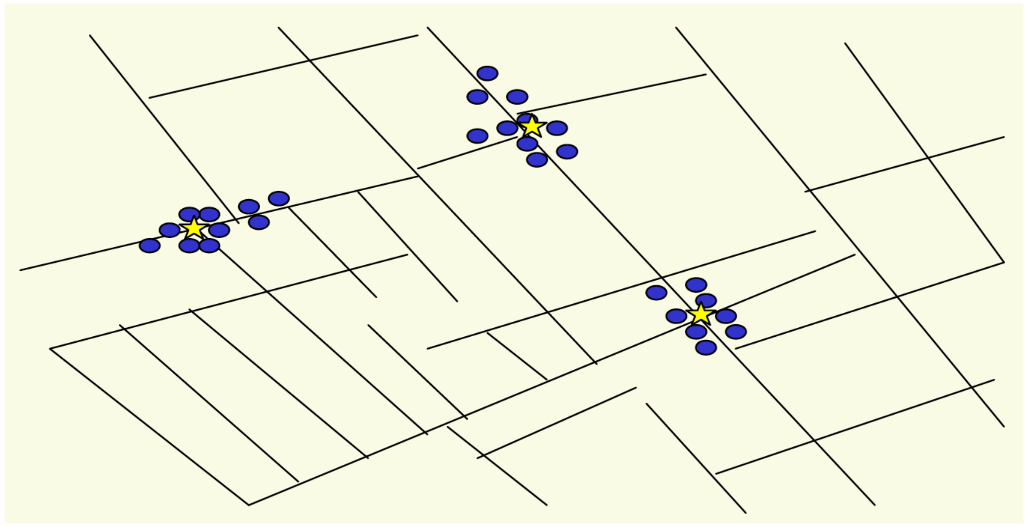
Impact. Kruskal and Prim only interact with costs via pairwise comparisons. If perturbations are sufficiently small, MST with perturbed costs is MST with original costs.

$\uparrow$ e.g., if all edge costs are integers, perturbing cost of edge e_i by i / n^2

Implementation. Can handle arbitrarily small perturbations implicitly by breaking ties lexicographically, according to index.

```
boolean less(i, j) {
   if      (cost(e_i) < cost(e_j)) return true
   else if (cost(e_i) > cost(e_j)) return false
   else if (i < j)                  return true
   else                             return false
}
```

4.7 Clustering



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

Clustering

Clustering. Given a set U of n objects labeled $p_1, \ldots, p_n$, classify into coherent groups.

photos, documents. micro-organisms

Distance function. Numeric value specifying "closeness" of two objects.

number of corresponding pixels whose intensities differ by some threshold

Fundamental problem. Divide into clusters so that points in different clusters are far apart.

- Routing in mobile ad hoc networks.
- Identify patterns in gene expression.
- Document categorization for web search.
- Similarity searching in medical image databases
- Skycat: cluster 10^9 sky objects into stars, quasars, galaxies.

Clustering of Maximum Spacing

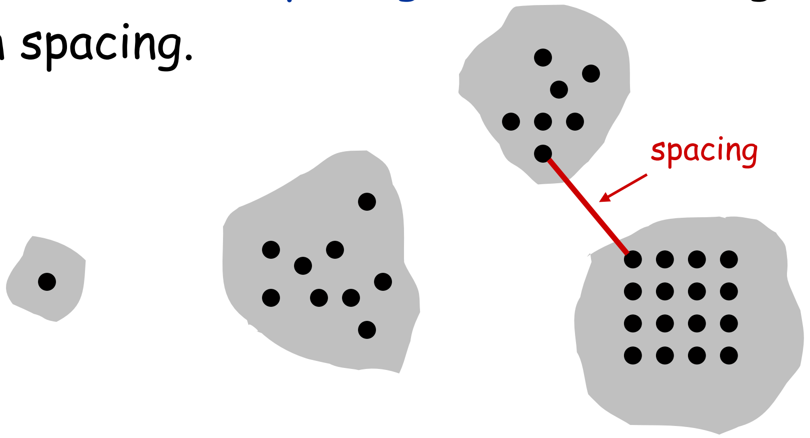
k-clustering. Divide objects into k non-empty groups.

Distance function. Assume it 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)

(*) Common Clustering Criterion:
Spacing. Min distance between any pair of points in different clusters.

Clustering of maximum spacing. Given an integer k, find a k-clustering of maximum spacing.



k = 4

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.

Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

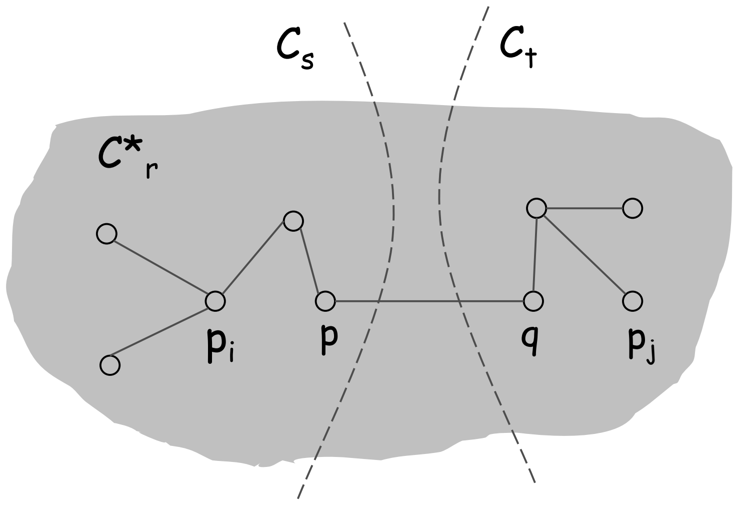
Remark. Equivalent to finding an MST and deleting the k-1 most expensive edges.

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. Let C denote some other clustering $C_1, \ldots, C_k$.

- The spacing of C^* is the length d^* of the (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 $\leq d^*$ since p and q are in different clusters. ▪

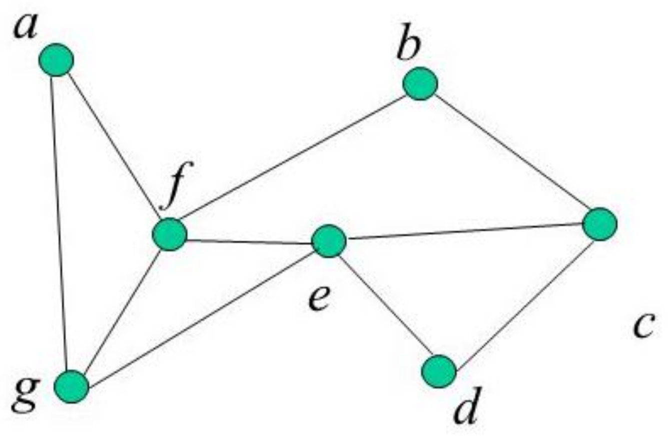


(Before Huffman codes) Aside: Extended HW Discussion

HW: 3.8

A few preliminaries. The distance dist(u,v) between two nodes in a graph is defined as the shortest path length between u and v (here length is determined by the number of edges in a path).

The diameter dim(G) of a graph is the maximum distance between any pair of nodes.



Distance(f,c) : 2
Distance(g,c): 2
Distance(a,c): 3

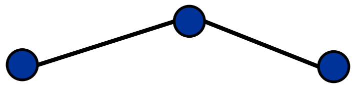
diameter: 3

HW: 3.8

A few preliminaries. Define the average pairwise distance, apd(G) to be the average, over all ${}_nC_2$ sets of two distinct nodes u and v, of the distance between u and v:

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u,v) \right] / \binom{n}{2}$$

Ex. Consider G: on three nodes with two edges {u,v} and {v,w}.

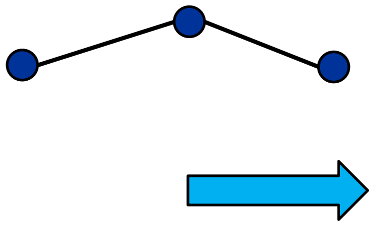


HW: 3.8

A few preliminaries. Define the average pairwise distance, apd(G) to be the average, over all ${}_nC_2$ sets of two distinct nodes u and v, of the distance between u and v:

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u,v) \right] / \binom{n}{2}$$

Ex. Consider G: on three nodes with two edges {u,v} and {v,w}.

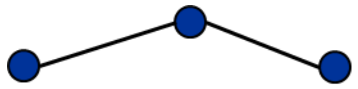


$$diam(G) = 2$$

$$apd(G) = \left[dist(u,v) + dist(u,w) + dist(v,v) \right] / 3 = 4/3$$

HW: 3.8

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u,v) \right] / \binom{n}{2}$$



$$\Longrightarrow \quad \begin{aligned} & diam(G) = 2 \\ & apd(G) = [dist(u,v) + dist(u,w) + dist(v,v)] / 3 = 4/3 \end{aligned}$$

Notice that diam(G) and apd(G) are relatively close to one another.

Claim: There exists a positive number c so that for all connected graphs G, it is the case that:

$$\frac{diam(G)}{apd(G)} \leq c$$

HW: 3.8

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u,v) \right] / \binom{n}{2}$$

Claim: There exists a positive number c so that for all connected graphs G, it is the case that:

$$\frac{diam(G)}{apd(G)} \le c$$

Is this true in general? If not, perhaps we construct a graph with a large diameter and a small apd value.

What type of graph has a very large diameter?

HW: 3.8

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u,v) \right] / \binom{n}{2}$$

Claim: There exists a positive number c so that for all connected graphs G, it is the case that:

$$\frac{diam(G)}{apd(G)} \le c$$

Is this true in general? If not, perhaps we construct a graph with a large diameter and a small apd value.

What type of graph has a very large diameter? A Path!

What type of graph has a very small apd?

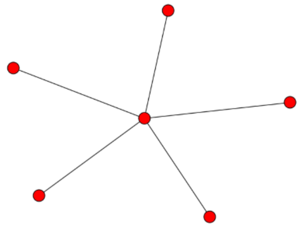
HW: 3.8

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u,v) \right] / \binom{n}{2}$$

Claim: There exists a positive number c so that for all connected graphs G, it is the case that:

$$\frac{diam(G)}{apd(G)} \leq c$$

Is this true in general? If not, perhaps we construct a graph with a large diameter and a small apd value.



What type of graph has a very large diameter? A Path!

What type of graph has a very small apd? One example: A graph with one shared vertex and many edges emanating from it (called a “star graph”).

HW: 3.8

$$apd(G) = \left[\sum_{\{u,v\} \subseteq V} dist(u,v) \right] / \binom{n}{2}$$

Claim: There exists a positive number c so that for all connected graphs G, it is the case that:

$$\frac{diam(G)}{apd(G)} \leq c$$

Hint: Consider a path with a star affixed to it (at, say one endpoint). Now consider the ratio of diam(G)/apd(G). You should be able to show with this example that the ratio can be arbitrarily large.

HW: 4.1

Let G be an arbitrary connected, undirected graph with distinct cost c(e) on every edge. Suppose e* is the cheapest edge in G; that is $c(e^*) < c(e)$ for every edge e different from e*. Then there is a MST of G that contains edge e*.

Thoughts?

HW: 4.4

Subsequence detection.

Give an algorithm that takes two sequences of events – S’ of length m and S of length n, each possibly containing an event more than once – and decide in time O(m+n) whether S’ is a subsequence of S.

HW: 4.4

Subsequence detection.

Give an algorithm that takes two sequences of events – S' of length m and S of length n, each possibly containing an event more than once – and decide in time O(m+n) whether S' is a subsequence of S.

Idea: Use a greedy algorithm that finds the first event in S that is the same as s_1' (the first event in S'); then find the first event after this that is the same as s_2', etc.; one can show this runs in O(m+n) time. Note: need to prove correctness…use induction on size of S.

HW: 4.8

Suppose that you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique MST.

Where to start?

HW: 4.8

Suppose that you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique MST.

Pf. (Proof by contradiction). Suppose there exist two MSTs for G, T and T*.

Then there is some edge e in T that is not in T* (by distinction b/w T and T*).

HW: 4.8

Suppose that you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique MST.

Pf. (Proof by contradiction). Suppose there exist two MSTs for G, T and T*.

Then there is some edge e in T that is not in T* (by distinction b/w T and T*).

Now add edge e to T* (recognize this pattern?).

What can we claim now about the graph T*+e?

HW: 4.8

Suppose that you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique MST.

Pf. (Proof by contradiction). Suppose there exist two MSTs for G, T and T*.

Then there is some edge e in T that is not in T* (by distinction b/w T and T*).

Now add edge e to T* (recognize this pattern?).

What can we claim now about the graph T*+e? It has a cycle.

Consider the edge e' in this cycle with maximum cost (guaranteed to exist by distinct edge cost assumption). Now use the cycle property proved in lecture…

HW: 4.8

Suppose that you are given a connected graph G, with edge costs that are all distinct. Prove that G has a unique MST.

Pf. (Proof by contradiction). Suppose there exist two MSTs for G, T and T*.

Then there is some edge e in T that is not in T* (by distinction b/w T and T*).

Now add edge e to T* (recognize this pattern?).

What can we claim now about the graph T*+e? It has a cycle.

Consider the edge e' in this cycle with maximum cost (guaranteed to exist by distinct edge cost assumption). Now use the cycle property proved in lecture…

HW: 4.21

Let us say that a graph G=(V,E) is a near-tree if it is connected and has at most n+8 edges, where n=|V|.

Give an algorithm with running time O(n) that takes a near-tree G with costs on its edges, and returns a MST of G. You may assume the edge costs are distinct.

HW: 4.21

Let us say that a graph G=(V,E) is a near-tree if it is connected and has at most n+8 edges, where n=|V|.

Give an algorithm with running time O(n) that takes a near-tree G with costs on its edges, and returns a MST of G. You may assume the edge costs are distinct.

Q: What's a good approach to detect cycles?

HW: 4.21

Let us say that a graph $G=(V,E)$ is a near-tree if it is connected and has at most $n+8$ edges, where $n=|V|$.

Give an algorithm with running time $O(n)$ that takes a near-tree G with costs on its edges, and returns a MST of G. You may assume the edge costs are distinct.

Q: What's a good approach to detect cycles?

Idea: Use BFS + cycle property. How?

HW: 4.21

Let us say that a graph $G=(V,E)$ is a near-tree if it is connected and has at most $n+8$ edges, where $n=|V|$.

Give an algorithm with running time $O(n)$ that takes a near-tree G with costs on its edges, and returns a MST of G. You may assume the edge costs are distinct.

Q: What's a good approach to detect cycles?

Idea: Use BFS + cycle property. How?

Note: cycle property will guarantee MST of G.

HW: 4.21

Let us say that a graph G=(V,E) is a near-tree if it is connected and has at most n+8 edges, where n=|V|.

Give an algorithm with running time O(n) that takes a near-tree G with costs on its edges, and returns a MST of G. You may assume the edge costs are distinct.

Q: What's a good approach to detect cycles?

Idea: Use BFS + cycle property. How?

Note: cycle property will guarantee MST of G.

HW: 4.29

Given a list of n natural numbers: $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G=(V,E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (G is assumed to be a simple graph).

HW: 4.29

Given a list of n natural numbers: $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph $G=(V,E)$ whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (G is assumed to be a simple graph).

<u>A few comments</u>.

Not every list of such numbers corresponds with an undirected simple graph.

Example: 1,1,1,0 does not have such a corresponding graph. Why not?

HW: 4.29

Given a list of n natural numbers: $d_1,d_2,\ldots,d_n$, show how to decide in polynomial time whether there exists an undirected graph $G=(V,E)$ whose node degrees are precisely the numbers $d_1,d_2,\ldots,d_n$. (G is assumed to be a simple graph).

A few comments.

Not every list of such numbers corresponds with an undirected simple graph.

Example: 1,1,1,0 does not have such a corresponding graph. Why not?

When a list of degrees does have a graph representation we say the degree list is graphical.

HW: 4.29

Given a list of n natural numbers: $d_1, d_2, \ldots, d_n$, show how to decide in polynomial time whether there exists an undirected graph G=(V,E) whose node degrees are precisely the numbers $d_1, d_2, \ldots, d_n$. (G is assumed to be a simple graph).

Proof idea: Use recursion/induction.

For simplicity, assume all of the degree values are greater than zero and arranged in decreasing order:

$$d_1 \geq d_2 \geq \ldots \geq d_n > 0 \qquad L = \{d_1, d_2, \ldots, d_n\}$$

HW: 4.29

Given a list of n natural numbers: $d_1,d_2,\dots,d_n$, show how to decide in polynomial time whether there exists an undirected graph G=(V,E) whose node degrees are precisely the numbers $d_1,d_2,\dots,d_n$. (G is assumed to be a simple graph).

Proof idea: Use recursion/induction.

For simplicity, assume all of the degree values are greater than zero and arranged in decreasing order:

$$d_1 \geq d_2 \geq \dots \geq d_n > 0 \qquad L = \{d_1, d_2, \dots, d_n\}$$

(*) Now, if you delete the vertex of maximum degree from this list (say $d_1=\Delta$), then we have a new (smaller) graphical list. Why?

HW: 4.29

Proof idea: Use recursion/induction.

For simplicity, assume all of the degree values are greater than zero and arranged in decreasing order:

$$d_1 \geq d_2 \geq \ldots \geq d_n > 0 \qquad L = \{d_1, d_2, \ldots, d_n\}$$

(*) Now, if you delete the vertex of maximum degree from this list (say $d_1=\Delta$), then we have a new (smaller) graphical list. Why?

$$L^* = \{d_2 - 1, d_3 - 1, \ldots, d_{\Delta+1} - 1, d_{\Delta+2}, \ldots, d_n\}$$

This is akin to deleting the largest degree vertex from a graph and all of its neighbors. This should be enough to help you solve the problem in full.

4.8 Huffman Codes

Data Compression

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?

Q. How do we know when the next symbol begins?

Ex. c(a) = 01   What is 0101?
c(b) = 010
c(e) = 1

Data Compression

Q. Given a text that uses 32 symbols (26 different letters, space, and some punctuation characters), how can we encode this text in bits?
A. We can encode 2^5 different symbols using a fixed length of 5 bits per symbol. This is called fixed length encoding.

Q. Some symbols (e, t, a, o, i, n) are used far more often than others. How can we use this to reduce our encoding?
A. Encode these characters with fewer bits, and the others with more bits.

Q. How do we know when the next symbol begins?
A. Use a separation symbol (like the pause in Morse), or make sure that there is no ambiguity by ensuring that no code is a prefix of another one.

Ex. c(a) = 01 What is 0101?
c(b) = 010
c(e) = 1

Prefix Codes

Definition. A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x,y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

Ex. $c(a) = 11$
$c(e) = 01$
$c(k) = 001$
$c(l) = 10$
$c(u) = 000$

Q. What is the meaning of 1001000001 ?

Prefix Codes

Definition. A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x, y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000

Q. What is the meaning of 1001000001 ?

A. "leuk"

Prefix Codes

Definition. A prefix code for a set S is a function c that maps each $x \in S$ to 1s and 0s in such a way that for $x,y \in S$, $x \neq y$, $c(x)$ is not a prefix of $c(y)$.

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000

Q. What is the meaning of 1001000001 ?
A. "leuk"

Suppose frequencies are known in a text of 1G:
$f_a=0.4$, $f_e=0.2$, $f_k=0.2$, $f_l=0.1$, $f_u=0.1$

Q. What is the size of the encoded text?
A. $2*f_a + 2*f_e + 3*f_k + 2*f_l + 4*f_u = 2.4G$

Optimal Prefix Codes

Definition. The average bits per letter of a prefix code c is the sum over all symbols of its frequency times the number of bits of its encoding:

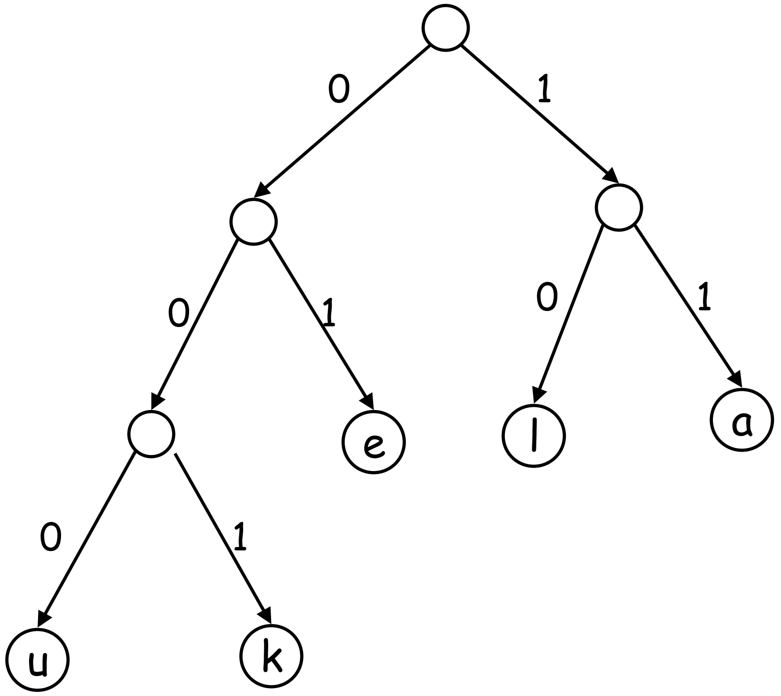
$$ABL(c) = \sum_{x \in S} f_x \cdot |c(x)|$$

We would like to find a prefix code that is has the lowest possible average bits per letter.

Suppose we model a code in a binary tree…

Representing Prefix Codes using Binary Trees

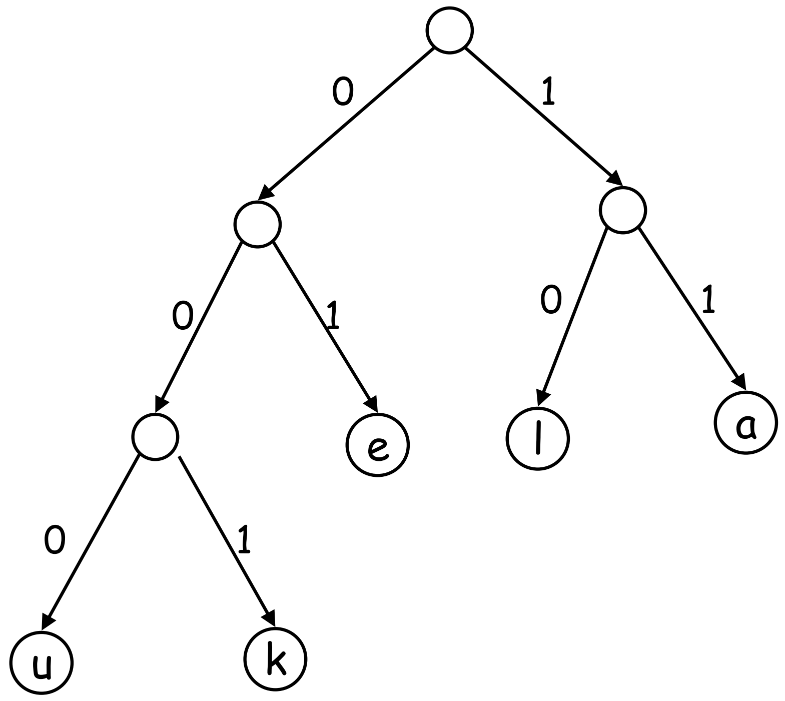
Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000



Q. What does the tree of a prefix code look like?

Representing Prefix Codes using Binary Trees

Ex. c(a) = 11
c(e) = 01
c(k) = 001
c(l) = 10
c(u) = 000

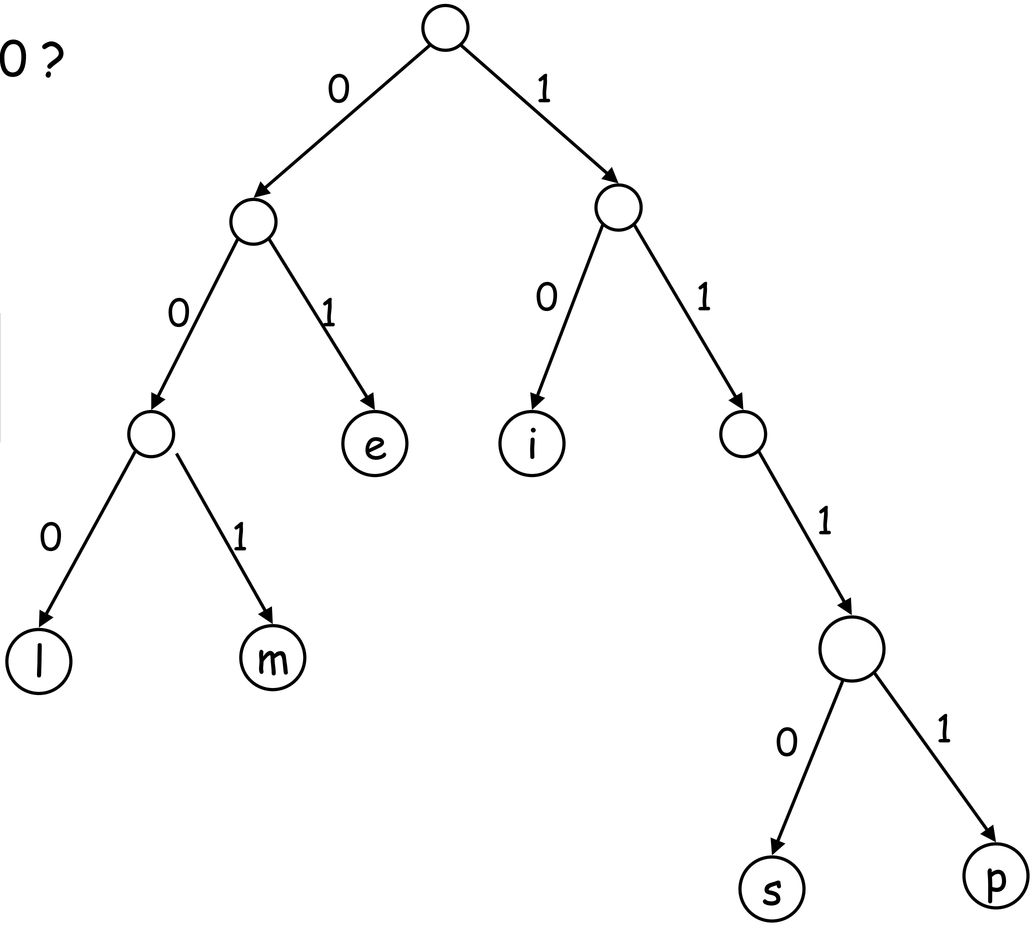


Q. Whatdoes the tree of a prefix code look like?
A. Only the leaves have a label.
Pf. An encoding of x is a prefix of an encoding of y if and only if the path of x is a prefix of the path of y.

Representing Prefix Codes using Binary Trees

Q. What is the meaning of
111010001111101000 ?

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$

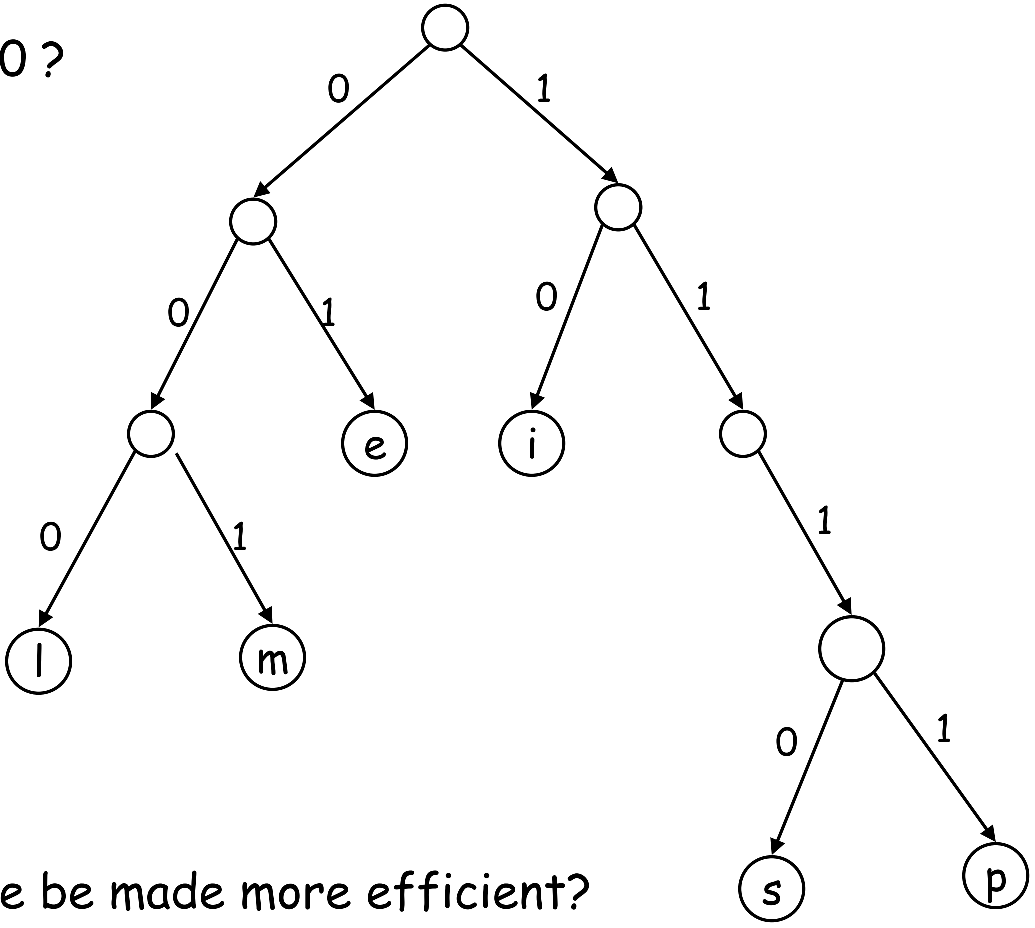


Representing Prefix Codes using Binary Trees

Q. What is the meaning of
111010001111101000 ?

A. "simpel"

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$



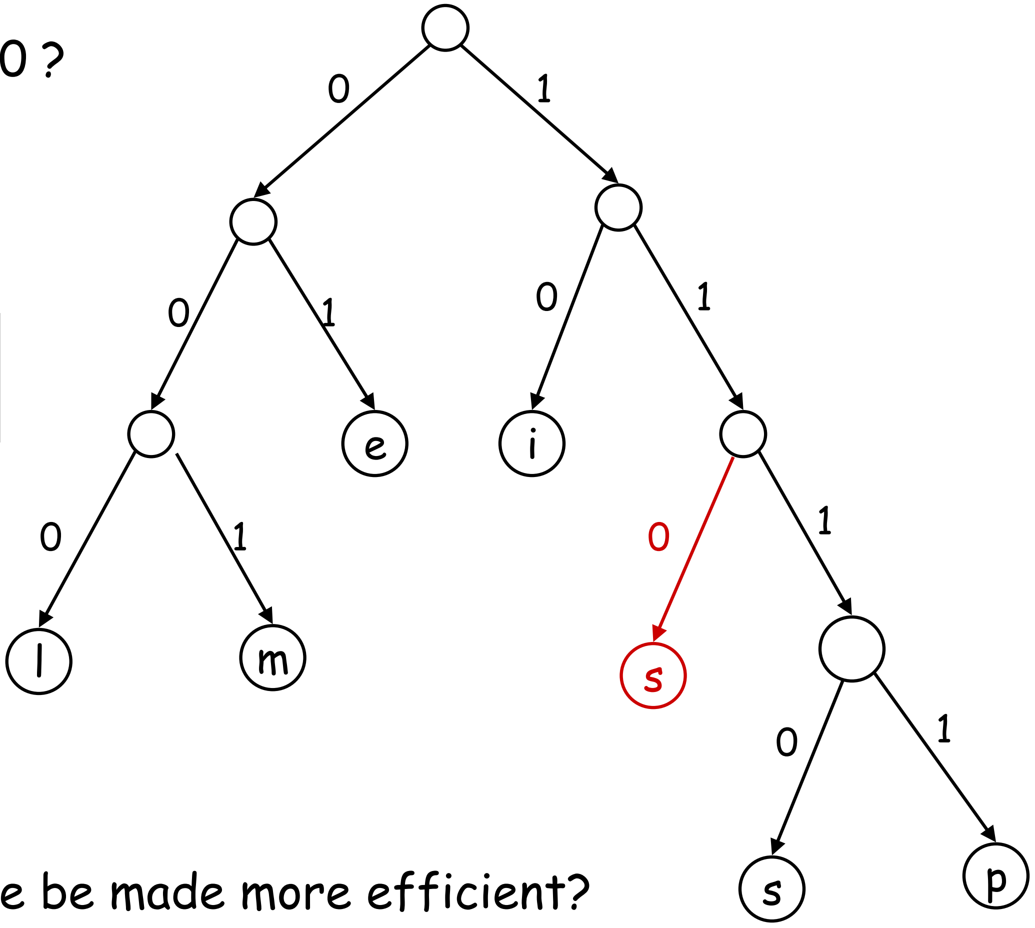
Q. How can this prefix code be made more efficient?

Representing Prefix Codes using Binary Trees

Q. What is the meaning of
111010001111101000 ?

A. "simpel"

$$ABL(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x)$$



Q. How can this prefix code be made more efficient?

A. Change encoding of p and s to a shorter one.

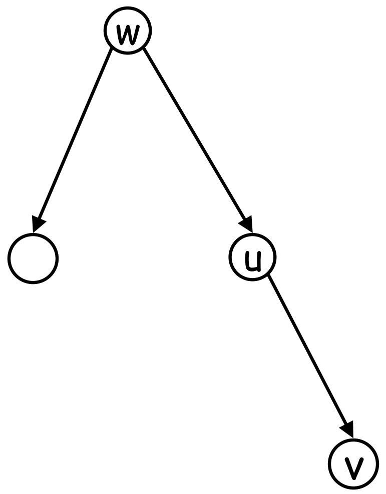
This tree is now full (i.e. each node that is not a leaf has two children).

Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.
Pf.

Suggestions?



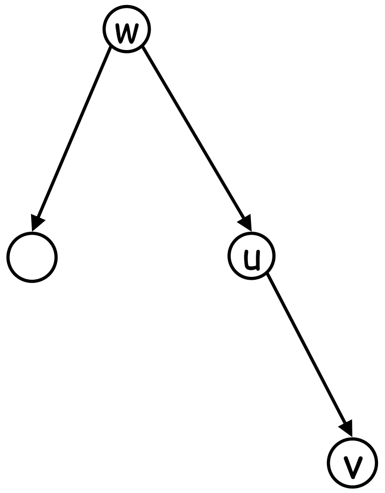
Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.

Pf. (by contradiction)

- Suppose T is binary tree of optimal prefix code and is not full.
- This means there is a node u (viz., a non-leaf) with only one child v.

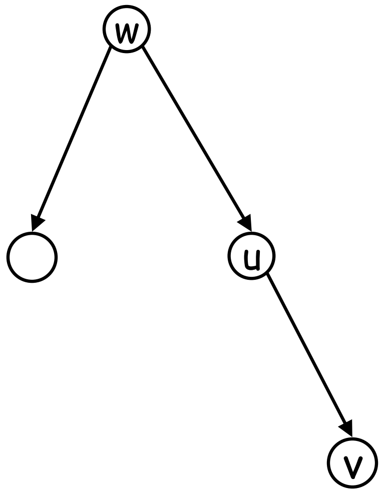


Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.
Pf. (by contradiction)

- Suppose T is binary tree of optimal prefix code and is not full.
- This means there is a node u with only one child v.
- Case 1: u is the root; delete u and use v as the root

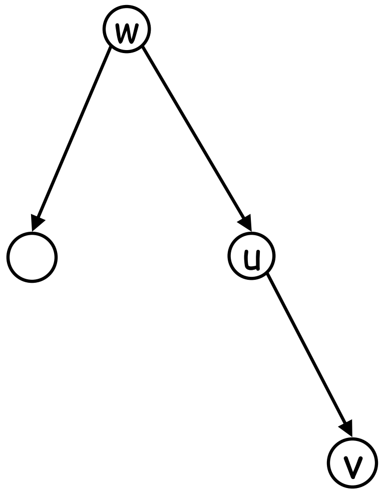


Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.
Pf. (by contradiction)

- Suppose T is binary tree of optimal prefix code and is not full.
- This means there is a node u with only one child v.
- Case 1: u is the root; delete u and use v as the root
- Case 2: u is not the root
  - let w be the parent of u
  - delete u and make v be a child of w in place of u



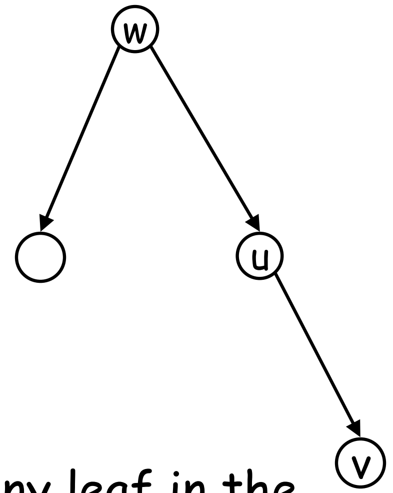
Representing Prefix Codes using Binary Trees

Definition. A tree is full if every node that is not a leaf has two children.

Claim. The binary tree corresponding to the optimal prefix code is full.
Pf. (by contradiction)

- Suppose T is binary tree of optimal prefix code and is not full.
- This means there is a node u with only one child v.
- Case 1: u is the root; delete u and use v as the root

- Case 2: u is not the root
  - let w be the parent of u
  - delete u and make v be a child of w in place of u



- In both cases the number of bits needed to encode any leaf in the subtree of v is decreased. The rest of the tree is not affected.
- Clearly this new tree T' has a smaller ABL than T. Contradiction.

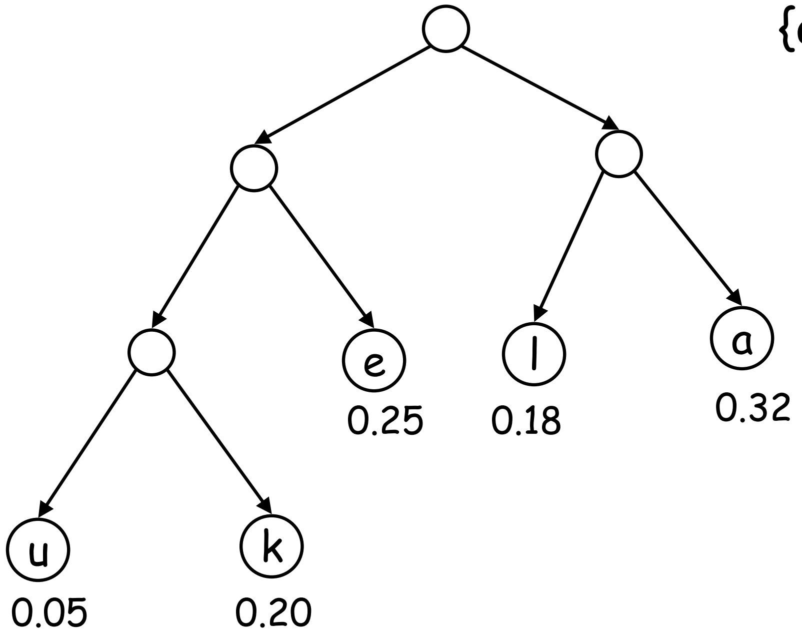
Optimal Prefix Codes: False Start

Q. Where in the tree of an optimal prefix code should letters be placed with a high frequency?

Optimal Prefix Codes: False Start

Q. Where in the tree of an optimal prefix code should letters be placed with a high frequency?
A. Near the top.

Greedy template. Create tree top-down, split S into two sets S_1 and S_2 with (almost) equal frequencies. Recursively build tree for S_1 and S_2.
[Shannon-Fano, 1949] f_a=0.32, f_e=0.25, f_k=0.20, f_l=0.18, f_u=0.05

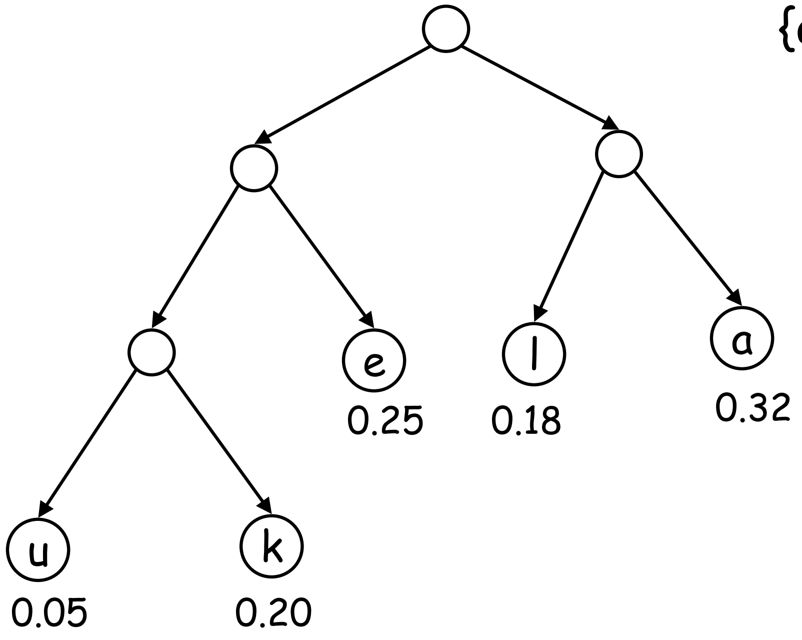


{a,l} , {e,k,u}

Optimal Prefix Codes: False Start

Q. Where in the tree of an optimal prefix code should letters be placed with a high frequency?
A. Near the top.

Greedy template. Create tree top-down, split S into two sets S_1 and S_2 with (almost) equal frequencies. Recursively build tree for S_1 and S_2.
[Shannon-Fano, 1949] $f_a=0.32$, $f_e=0.25$, $f_k=0.20$, $f_l=0.18$, $f_u=0.05$



{a,l} , {e,k,u} -> {a,l} , {e,{k,u}}

Optimal Prefix Codes: Huffman Encoding

Observation. Lowest frequency items should be at the lowest level in tree of optimal prefix code.

Observation. For $n > 1$, the lowest level always contains at least two leaves.

Observation. The order in which items appear in a level does not matter.

Optimal Prefix Codes: Huffman Encoding

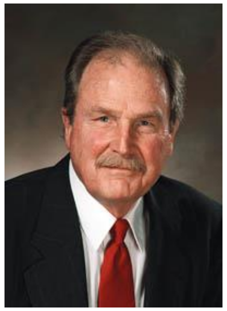
Observation. Lowest frequency items should be at the lowest level in tree of optimal prefix code.

Observation. For $n > 1$, the lowest level always contains at least two leaves.

Observation. The order in which items appear in a level does not matter.

Claim. There is an optimal prefix code with tree T* where the two lowest-frequency letters are assigned to leaves that are siblings in T*.

Greedy template. [Huffman, 1952] Create tree bottom-up.
Make two leaves for two lowest-frequency letters y and z.
Recursively build tree for the rest using a meta-letter for yz.



Optimal Prefix Codes: Huffman Encoding

```
Huffman(S) {
   if |S|=2 {
      return tree with root and 2 leaves
   } else {
      let y and z be lowest-frequency letters in S
      S′ = S
      remove y and z from S′
      insert new letter ω in S′ with $f_\omega=f_y+f_z$
      T′ = Huffman(S′)
      T = add two children y and z to leaf ω from T′
      return T
   }
}
```

Demo:
http://www.ida.liu.se/opendsa/OpenDSA/Books/TDDD86F17/html/Huffman.html#

Optimal Prefix Codes: Huffman Encoding

```
Huffman(S) {
   if |S|=2 {
      return tree with root and 2 leaves
   } else {
      let y and z be lowest-frequency letters in S
      S′ = S
      remove y and z from S′
      insert new letter ω in S′ with $f_\omega = f_y + f_z$
      T′ = Huffman(S′)
      T = add two children y and z to leaf ω from T′
      return T
   }
}
```

Q. What is the time complexity?

Optimal Prefix Codes: Huffman Encoding

```
Huffman(S) {
   if |S|=2 {
      return tree with root and 2 leaves
   } else {
      let y and z be lowest-frequency letters in S
      S' = S
      remove y and z from S'
      insert new letter ω in S' with $f_ω=f_y+f_z$
      T' = Huffman(S')
      T = add two children y and z to leaf ω from T'
      return T
   }
}
```

Q. What is the time complexity?

A. $T(n) = T(n-1) + O(n)$

so $O(n^2)$

Q. How to implement finding lowest-frequency letters efficiently?

A. Use priority queue for S: $T(n) = T(n-1) + O(\log n)$ so $O(n \log n)$

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Pf. by induction, based on optimality of T' (y and z removed, ω added) (see next page)

Claim. $ABL(T')=ABL(T)-f_{\omega}$
Pf.

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Pf. by induction, based on optimality of T' (y and z removed, ω added) (see next page)

Claim. ABL(T')=ABL(T)-f_ω
Pf.

$$\begin{aligned}
\text{ABL}(T) &= \sum_{x\in S} f_x \cdot \text{depth}_T(x) \\
&= f_y \cdot \text{depth}_T(y) + f_z \cdot \text{depth}_T(z) + \sum_{x\in S, x\neq y,z} f_x \cdot \text{depth}_T(x) \\
&= (f_y + f_z)\cdot(1+\text{depth}_T(\omega)) + \sum_{x\in S, x\neq y,z} f_x \cdot \text{depth}_T(x) \\
&= f_\omega \cdot (1+\text{depth}_T(\omega)) + \sum_{x\in S, x\neq y,z} f_x \cdot \text{depth}_T(x) \\
&= f_\omega + \sum_{x\in S'} f_x \cdot \text{depth}_{T'}(x) \\
&= f_\omega + \text{ABL}(T')
\end{aligned}$$

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

Pf. (by induction over n=|S|)

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.
Pf. (by induction over n=|S|)
Base: For n=2 there is no shorter code than root and two leaves.
Hypothesis: Suppose Huffman tree T' for S' of size n-1 with ω instead of y and z is optimal.
Step: (by contradiction)

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

Pf. (by induction)

Base: For n=2 there is no shorter code than root and two leaves.

Hypothesis: Suppose Huffman tree T' for S' of size n-1 with ω instead of y and z is optimal. (IH)

Step: (by contradiction)

- *Idea of proof:*
  - Suppose other tree Z of size n is better.
  - Delete lowest frequency items y and z from Z creating Z'
  - Z' cannot be better than T' by IH.

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

Pf. (by induction)

Base: For n=2 there is no shorter code than root and two leaves.

Hypothesis: Suppose Huffman tree T' for S' with ω instead of y and z is optimal. (IH)

Step: (by contradiction)

- Suppose Huffman tree T for S is not optimal.
- So there is some tree Z such that ABL(Z) < ABL(T).
- Then there is also a tree Z for which leaves y and z exist that are siblings and have the lowest frequency (see observation).

Huffman Encoding: Greedy Analysis

Claim. Huffman code for S achieves the minimum ABL of any prefix code.

Pf. (by induction)

Base: For n=2 there is no shorter code than root and two leaves.

Hypothesis: Suppose Huffman tree T' for S' with ω instead of y and z is optimal. (IH)

Step: (by contradiction)

- Suppose Huffman tree T for S is not optimal.
- So there is some tree Z such that ABL(Z) < ABL(T).
- Then there is also a tree Z for which leaves y and z exist that are siblings and have the lowest frequency (see observation).
- Let Z' be Z with y and z deleted, and their former parent labeled ω.
- Similarly T' is derived from S' in our algorithm.
- We know that $ABL(Z')=ABL(Z)-f_\omega$, as well as $ABL(T')=ABL(T)-f_{\omega}$.
- But also ABL(Z) < ABL(T), so ABL(Z') < ABL(T').
- Contradiction with IH.