

Greedy Algorithms: MSTs, Clustering, and Huffman Coding

CMSC 451, Summer 2009

Reminder

- Homework 1 due at the beginning of class on Monday morning. 9:40 is too late.
- See me at office hours if you have questions!
- Email is also fine, but I might not respond as quickly as you'd like...

Where were we...?

- Minimum spanning tree
- 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.
 - Need a data structure to manage the connected components

Union-Find Data Structure

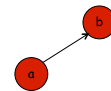
MakeUnionFind($S=\{a,b\}$)

Create singleton trees for all items in the set



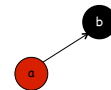
Union($A=\{a\}, B=\{b\}$)

Merge two connected components by creating a pointer from the root of the smaller tree to the root of the larger tree. Store the size of its tree with each root.



Find(a)

Traverse up the tree until finding the root. The name of the root is the name of the tree.



Union-Find Data Structure: Analysis

MakeUnionFind($S=\{a,b\}$)

Create singleton trees for all items in the set

Time analysis

$O(n)$

Union($A=\{a\}, B=\{b\}$)

Merge two connected components by creating a pointer from the root of the smaller tree to the root of the larger tree. Store the size of its tree with each root.

$O(1)$

$O(\log n)$: Takes time on the order of the number of times the name changed. Since the larger set keeps its name, a name change implies that the set at least doubled. It can be of size at most n .

Find(a)

Traverse up the tree until finding the root. The name of the root is the name of the tree.

Kruskal's Algorithm: Time Analysis

```

Kruskal(G, c) {
  Sort edge weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .
   $T \leftarrow \emptyset$ 
  foreach ( $u \in V$ )
    make a set containing singleton  $u$ 
  for  $i = 1$  to  $m$ 
    ( $u, v$ ) =  $e_i$ 
    if ( $u$  and  $v$  are in different sets) {
       $T \leftarrow T \cup \{e_i\}$ 
      merge the sets containing  $u$  and  $v$ 
    }
  return  $T$ 
}

```

Use the **union-find** data structure.

- Sort: $O(m \log n)$ time (since $m=O(n^2)$, $\log m = O(\log n^2) = O(2 \log n)$)
- For all nodes: $O(n)$ total time for MakeUnionFind
- For each edge: $O(\log n)$ time for Find, $O(1)$ time for Union
- Total: $O(m \log n)$ time

Kruskal's Algorithm

Find the MST and maintain the union-find structure.

```

Kruskal(G, c) {
  Sort edge weights so that  $c_1 \leq c_2 \leq \dots \leq c_m$ .
   $T \leftarrow \emptyset$ 
  foreach ( $u \in V$ )
    make a set containing singleton  $u$ 
  for  $i = 1$  to  $m$ 
    ( $u, v$ ) =  $e_i$ 
    if ( $u$  and  $v$  are in different sets) {
       $T \leftarrow T \cup \{e_i\}$ 
      merge the sets containing  $u$  and  $v$ 
    }
  return  $T$ 
}
    
```

Lexicographic Tiebreaking

To remove the assumption that all edge costs are distinct: perturb all edge costs by tiny amounts 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.

e.g., if all edge costs are integers, perturbing cost of edge e_i by $1/n^2$

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

```

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

Clustering

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

Clustering

Clustering. Given a set U of n objects labeled p_1, \dots, 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

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)

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.

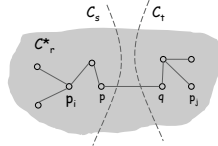
Key observation. This procedure is precisely Kruskal's algorithm (except we stop when there are k connected components).

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, \dots, 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, \dots, 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. ■



Huffman Codes

These lecture slides are supplied by Mathijs de Weerd

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 ?

Suppose frequencies are known in a text with 16 of letters:

$f_e=0.4, f_s=0.2, f_k=0.2, f_l=0.1, f_u=0.1$

Q. What is the size of the encoded text?

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 ?

▲ "leuk"

Suppose frequencies are known in a text with 16 of letters:

$f_e=0.4, f_s=0.2, f_k=0.2, f_l=0.1, f_u=0.1$

Q. What is the size of the encoded text?

A. $2*f_e + 2*f_s + 3*f_k + 2*f_l + 3*f_u = 2.36$

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:

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

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

Suppose we model a code in a binary tree...

19

Representing Prefix Codes using Binary Trees

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

Q. What is distinctive about the tree of a prefix code?

20

Representing Prefix Codes using Binary Trees

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

Q. What is distinctive about the tree of a prefix code?
 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 .

21

Representing Prefix Codes using Binary Trees

Q. What is the meaning of 111010001111101000 ?

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

22

Representing Prefix Codes using Binary Trees

Q. What is the meaning of 111010001111101000 ?
 A. "simpel"

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

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

23

Representing Prefix Codes using Binary Trees

Q. What is the meaning of 111010001111101000 ?
 A. "simpel"

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

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**.

24

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.

25

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.

26

Optimal Prefix Codes: False Start

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

27

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] $f_e=0.32, f_k=0.25, f_l=0.20, f_i=0.18, f_u=0.05$

28

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.

29

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 w in S' with  $f_w=f_y+f_z$ 
    T' = Huffman(S')
    T = add two children y and z to leaf w from T'
    return T
  }
}
    
```

Build a tree for:
 Alphabet: {a,e,k,l,u}
 Frequencies: $f_e=0.32, f_k=0.25, f_l=0.20, f_i=0.18, f_u=0.05$

30

Huffman Coding Analysis

We didn't get to the following slides. You are not responsible for their specific content, however, since you should generally know how analyze algorithms, it might be useful to review them.

31

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?

32

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 $O(n \log n)$

33

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.

34

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.

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

35

Huffman Encoding: Greedy Analysis

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

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

36

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)

37

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.

38

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 .
- Similar 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.

39