CSEP 521 – Applied Algorithms

Greedy Algorithms

Reading:

- CLRS,
  - Chapter 17 (1st ed.)
  - Chapter 16 (2nd ed.)

Greedy Algorithms

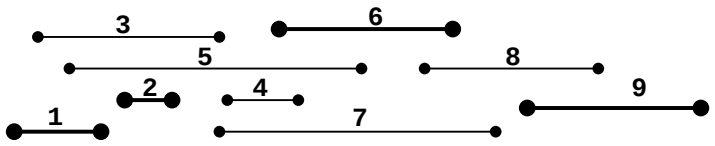
- A *greedy algorithm* always makes the choice that looks best at the moment
  - everyday example: Driving in heavy traffic
  - The hope: a locally optimal choice will lead to a globally optimal solution
  - For some problems, it works
- Dynamic programming can be overkill; greedy algorithms tend to be easier to code

Activity-Selection Problem

- Problem: get your money's worth out of a carnival
  - Buy a wristband that lets you onto any ride
  - Lots of rides, each starting and ending at different times
  - Your goal: ride as many rides as possible
    - Another, alternative goal that we don't solve here: maximize time spent on rides
- Welcome to the *activity selection problem*

Activity-Selection

- Formally:
  - Given a set $S = \{a_1, a_2, \ldots, a_n\}$ of n activities
    s_i = start time of activity i
    f_i = finish time of activity i
  - Find max-size subset A of non-conflicting activities



- Assume (w.l.o.g) that $f_1 \leq f_2 \leq \ldots \leq f_n$

Activity-Selection – A DP solution

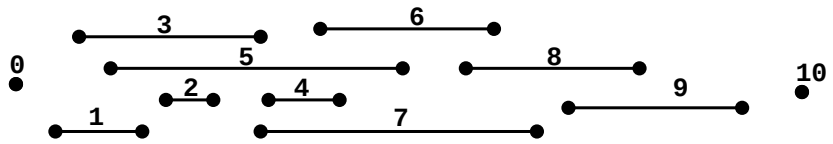
Define:

$S_{ij} = \{a_k \in S : f_i \le s_k < f_k \le s_j\}$

The subset of activities that can start after a_i finishes and finish before a_j starts.

Remark: we add 'dummy activities' a_0 with $f_0=0$

And a_{n+1} with $s_{n+1}=\infty$



Examples: $S_{2,9} = \{4,6,7\}$; $S_{1,8} = \{2,4\}$; $S_{0,10} = S$

Activity-Selection – A DP solution

Define:

$C[i,j]$= maximal number of activities from S_{ij} that can be selected.

$$C[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \varnothing \end{cases}$$

In words: if S_{ij} is not empty, then for any activity k in S_{ij} we check what is the best we can do if k is selected.

Based on this formula we can write a DP algorithm whose time complexity is $O(n^3)$

Activity-Selection – A DP solution

Define:

$C[i,j]$= maximal number of activities from S_{ij} that can be selected.

$$C[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \varnothing \\ \max_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \varnothing \end{cases}$$

In words: if S_{ij} is not empty, then for any activity k in S_{ij} we check what is the best we can do if k is selected.

Based on this formula we can write a DP algorithm whose time complexity is $O(n^3)$

Greedy Choice Property

- Activity selection problem also exhibits the *greedy choice* property:
  - Locally optimal choice $\Rightarrow$ globally optimal sol'n
- Theorem: if S is an activity selection instance sorted by finish time, then there exists an optimal solution $A \subseteq S$ such that $\{a_1\} \in A$
- Proof: Given an optimal solution B that does not contain $\{a_1\}$, replace the first activity in B with $\{a_1\}$. The resulting solution is feasible (why?), it has the same number of activities as B, and it includes $\{a_1\}$.

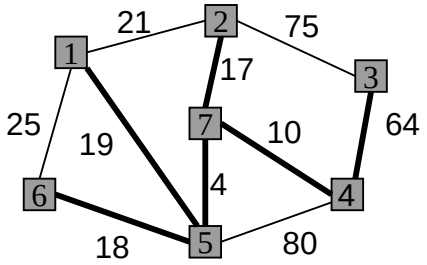
Activity Selection: A Greedy Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Time complexity: O(n log n)
- Intuition is even more simple:
  - Always pick the earliest to finish ride available at the time.

Minimum Spanning Tree

Each edge has a cost.

Goal: a minimal-cost subset of edges that will keep the graph connected.



Price of this tree = 18+19+4+10+17+64

Kruskal's Greedy Algorithm

```
Sort the edges by increasing cost;
Initialize T to be empty;
For each edge e chosen in increasing order do
    if adding e does not form a cycle then
        add e to T
```

In lecture 2 we saw that this is a possible executing of the blue/red rules.
Indeed, when proving these rules, we used the 'greedy choice property'.

Proof of Kruskal using Greedy-Choice Property

- Claim: for any $k \geq 0$, after Kruskal's algorithm inserts k edges to T, there exists an MST that includes all these edges.
- Proof: By induction on k (just a special case of the general proof we had for the blue/red rules)

The Coupon Collector Problem

•There are n different brands of cereal, each comes with a single coupon for a discount on another box of cereal, of another brand.

•You can use multiple coupons when purchasing a new box, up to getting the new box free, but you never get money back.

•You want to buy one box of each brand, for as little money as possible.

The Coupon Collector Problem

Example:

•Trix costs 2$ and has a 50c coupon for Cheerios.

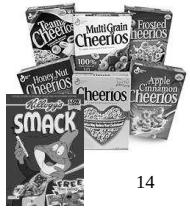
•Cheerios costs 3$ and has a 30c coupon for Smack.

•Smack costs 3$ and has a 50c coupon for Cheerios

Order 1: Trix, Cheerios, Smack
cost = 2$+(3$-50c)+(3$-30c) = 7.20$

Order 2: Cheerios, Trix, Smack
cost = 3$+2$+3$-30c = 7.70$

Order 3: Smack, Trix, Cheerios
cost = 3$+2$+3$-(50+50)c = 7$



The Coupon Collector Problem

In general:

Input: a list of n brands, each associated with a price, a value of the enclosed coupon, and the brand for which the coupon gives a discount.

Output: The optimal order.

- Inefficient algorithm: Check all n! possible orders.
- Can be solved by dynamic programming.
- We will see a simple optimal greedy algorithm.

A Greedy Algorithm for the Coupon Collector Problem

•If there is a brand with no coupon for it, buy a box of one such brand.

•If not, then every brand has exactly one coupon for it (why?). Let the real value of a coupon be the minimum of its value and the price of the brand it is for. Then we buy the brand where the coupon for it has the smallest real value.

•Let X be the brand for which we have a new coupon. Set the price of brand X to be maximum (old price minus coupon value, 0).

•Solve recursively (repeat for remaining brands).

A Greedy Algorithm for the Coupon Collector Problem

Proof: We show that there exists an optimal order that agrees with the order produced by the algorithm.

Formally, let i be the brand which is selected first, then there exists an optimal order in which i is first.

Since we continue recursively for the rest of the brands, we conclude that there exists an optimal order that is identical to the greedy order.

We distinguish between two cases

Case 1: We select a brand with no coupon for it.
Case 2: We select a brand where the coupon for it has the smallest real value.

A Greedy Algorithm for the Coupon Collector Problem

Case 1: We select a brand, X, with no coupon for it .

•Let S be any order in which X is not first. Let S' be the order where we buy X first, and then buy the other brands in the same order as in S.

•Since X had no coupon, we pay the same price for X in both orders (full price). For all of the other brands, the price in S' is at most the price in S (the price of the brand we got a coupon for by buying X may have dropped).

•Therefore, the total cost of S' is at most that of S.

A Greedy Algorithm for the Coupon Collector Problem

Case 2: We select a brand, X, where the coupon for it has the smallest real value.

Consider a directed graph with edges from each brand to the brand it has coupons for.

Claim: This graph must be a union of disjoint cycles .

Proof: There are no vertices with in-degree=0 (since we are not in case 1).

Each vertex has out-degree=1 (one coupon per box)

→ Each vertex has in-degree=1

→ Only cycles, and must be disjoint.

A Greedy Algorithm for the Coupon Collector Problem

Case 2 (cont.): Let S be any order that does not buy X first, and define S' as first buying all the brands in X's cycle in order starting from X and then buying the rest as in S.

We show that the total cost of S' is not larger than the cost of S.

- No brand outside of X's cycle has changed price.

-Let's compare the prices of the brands in X's cycle in S vs. S'.

-Let Y be the first brand in X's cycle bought in S.

A Greedy Algorithm for the Coupon Collector Problem

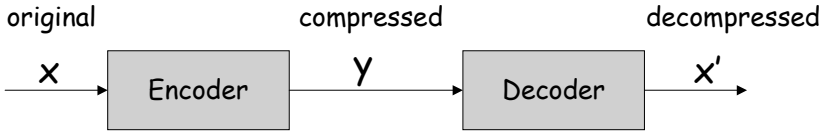
Case 2 (cont'):

For brands in the cycle except X and Y: in S' we pay full price minus value of coupon. In S we pay at least that amount (since there is only one coupon).

The price of X in S' may have increased over that in S by the value of its coupon, but the price of Y has decreased: in S, we pay full price, in S' we deduct the real value of Y' coupon. Since X had the smallest real coupon value, this means the total price for S' is at most that for S.

Data Compression

Basic concepts



- Lossless compression $x=x'$
  - Also called entropy coding, reversible coding.
- Lossy compression $x \neq x'$
  - Also called irreversible coding.
- Compression ratio = $|x|/|y|$
  - $|x|$ is number of bits in x.

Why Compress?

- Conserve storage space
- Reduce time for transmission
  - Faster to encode, send, then decode than to send the original
- Progressive transmission
  - Some compression techniques allow us to send the most important bits first so we can get a low resolution version of some data before getting the high fidelity version
- Reduce computation
  - Use less data to achieve an approximate answer

Lossless Compression

- Data is not lost - the original is really needed.
  - text compression
  - compression of computer binaries to fit on a floppy
- Compression ratio typically no better than 4:1 for lossless compression.
- Major techniques include
  - Huffman coding
  - Arithmetic coding
  - Dictionary techniques (Ziv, Lempel 1977,1978)
  - Sequitur (Nevill-Manning, Witten 1996)
  - Standards - Morse code, Braille, Unix compress, gzip, zip, GIF, JBIG, JPEG

Lossy Compression

- Data is lost, but not too much.
  - audio
  - video
  - still images, medical images, photographs
- Compression ratios of 10:1 often yield quite high fidelity results.
- Major techniques include
  - Vector Quantization
  - Wavelets
  - Transforms
  - Standards - JPEG, MPEG

Huffman Coding (1951)

- Uses frequencies of symbols in a string to build a variable rate prefix-free code.
  - Each symbol is mapped to a binary string.
  - More frequent symbols have shorter codes.
  - No code is a prefix of another.

Frequency-based Codes

Suppose we have a 100,000 character data file that we wish to store compactly.

The file contains only the characters a-f with the following frequencies.

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
Fixed-len	000	001	010	011	100	101

We are looking for a binary code.

In a fixed-length code, we need 3 bits per character. The total length of the code is 300,000 bits.

Variable-length Codes

Can we reduce the total size by using variable-length code?

We limit ourselves to prefix-free codes: no codeword is a prefix of another codeword.

This guarantees a simple one-pass decoding. Each code string has a unique parse, no need of 'end of character' sign.

Example: Consider the non-prefix-free code

$C(a)=0$, $C(b)=01$, $C(c)=10$.

The string 010 is a decoding of both 'ac' and 'ba'

Variable-length Codes

Can we reduce the total size by using variable-length code?

Yes!

In our example:

	a	b	c	d	e	f
Frequency	45	13	12	16	9	5
var-length	0	101	100	111	1100	1100

The total code length is

45k·1 + 13k·3 + 12k·3 + 16k·3 + 9k·4 + 5k·4 =224,000.

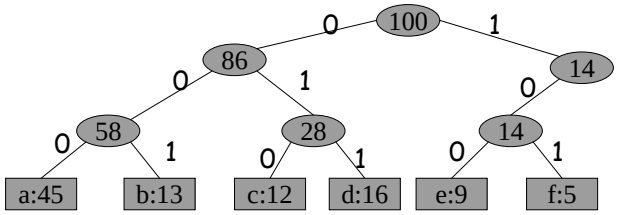
Improving by ~25% the 300,000 we had before.

Tree Representation of a binary-code

Every binary prefix-free code can be represented by a binary tree.

Each left-edge is marked 0, each right-edge is marked 1.

Each leaf represents a character. The path to the leaf determines the decoding of the character.



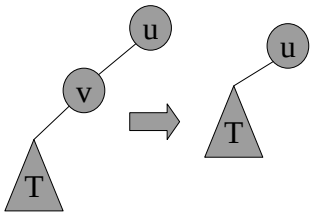
Example 1: The tree representing the fixed-length code

Each internal node is marked with the sum of weights of the leaves in its sub-tree.

Tree Representation of a Binary-code

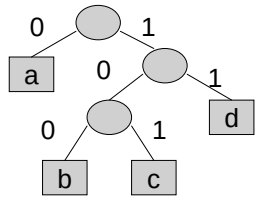
A simple observation: An optimal code is always represented by a full binary tree (each internal node has two children).

Proof: If we have a node with a singe child, we can improve the coding:



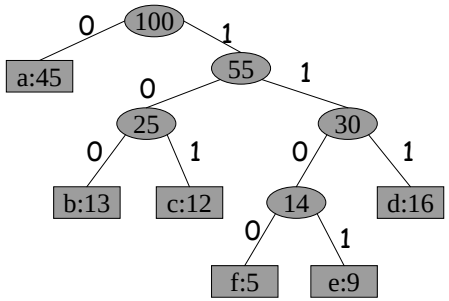
Encoding and Decoding

- Encoding: send the code, then the encoded data
  - x = aabddcaa
  - c(x) = a0,b100,c101,d11,0010011111010 0
- Decoding: Build the Huffman tree, then use it to decode.




```
repeat
  start at root of tree
    repeat
    if node is not a leaf
      if read bit = 1 then go right
      else go left
    report leaf
```

Tree Representation of a Binary-code



Example 2: The tree representing the optimal variable-length code

Given a tree T, For each character $a \in C$:

f(a) – the frequency of a in the file.

$d_T(a)$ – the depth of a in the file (= length of codeword for a)
The cost of the tree = $B(T) = \Sigma_a f(a)d_T(a)$

Constructing an Huffman Code

Huffman code – an optimal prefix-free code constructed by a greedy algorithm.

We build the tree in a bottom-up manner:

Starting with |C| isolated trees, we perform a sequence of |C|-1 merging operations to create the final tree.

Data Structure: A priority queue Q, keyed on f (the frequencies).

The objects in the queue are the trees. Whenever we merge two trees, the frequency of the merged tree is the sum of the frequencies of the merged trees.

Constructing an Huffman Code

Huffman(C)

•Init: |C| isolated vertices. Each vertex is a root of a tree of size 1. Each vertex gets a value $p(v_i)$= the frequency of the i^{th} character.

•Repeat |C|-1 times:

Add a new vertex v, whose children are the two tree-roots u_1,u_2 with minimal p value.

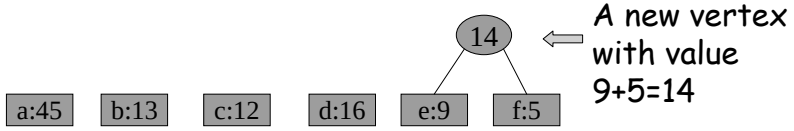
Determine $p(v)=p(u_1)+p(u_2)$

Constructing an Huffman Code - example

•Init: |C| isolated vertices. Each vertex is a root of a tree of size 1. Each vertex gets a value $p(v_i)$= the frequency of the i^{th} character.

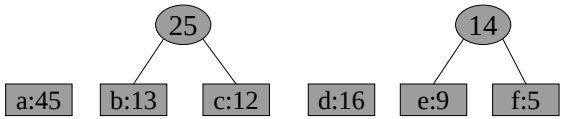
a:45 b:13 c:12 d:16 e:9 f:5

•Step 1: e and f have minimal value.



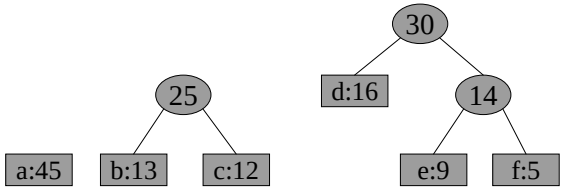
Constructing an Huffman Code – example (2)

•Step 2: b and c have minimal value.



A new vertex with value 12+13=25

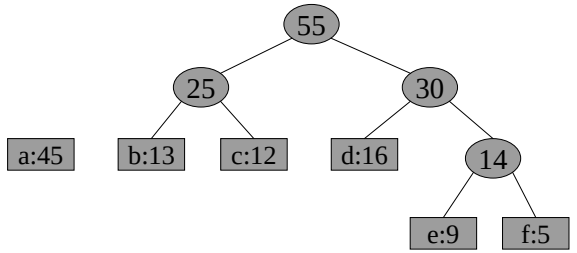
•Step 3: d and '14' have minimal value.



A new vertex with value 14+16=30

Constructing an Huffman Code – example (3)

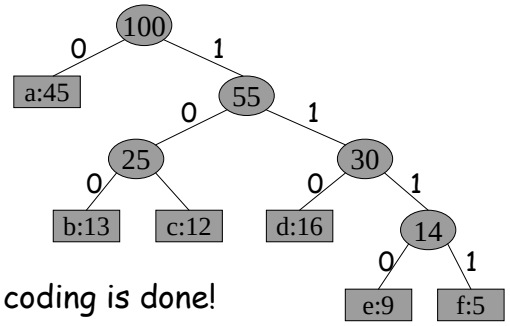
•Step 4: '25' and '30' have minimal value.



A new vertex with value 25+30=55

Constructing an Huffman Code – example (4)

•Step 5: '45' and '55' have minimal value.



A new vertex with value 45+55=100

All the characters are connected.

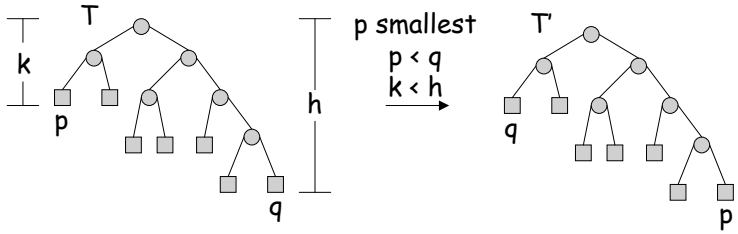
The coding is done!

Optimality of Huffman Code

•We show that Huffman code is optimal by showing that Huffman algorithm follows 3 optimality principles:

Optimality Principle 1

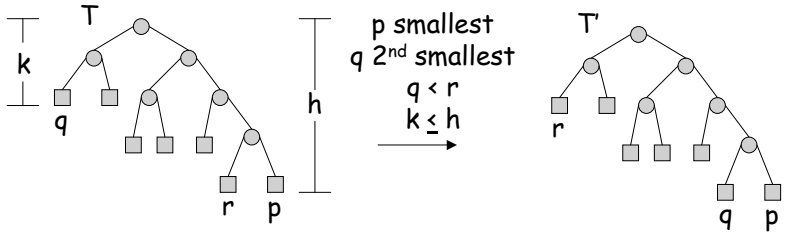
- In an optimal tree a lowest probability symbol has maximum distance from the root.
  - If not exchanging a lowest probability symbol with any deeper one will lower the cost.



$C(T') = C(T) + hp - hq + kq - kp = C(T) - (h-k)(q-p) < C(T)$

Optimality Principle 2

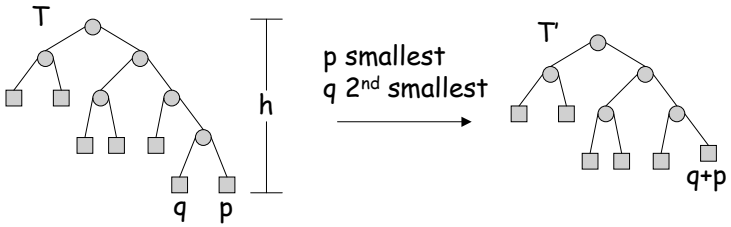
- The second lowest probability is a sibling of the smallest in some optimal tree.
  - If not, we can move it there not raising the cost.



$C(T') = C(T) + hq - hr + kr - kq = C(T) - (h-k)(r-q) \leq C(T)$

Optimality Principle 3

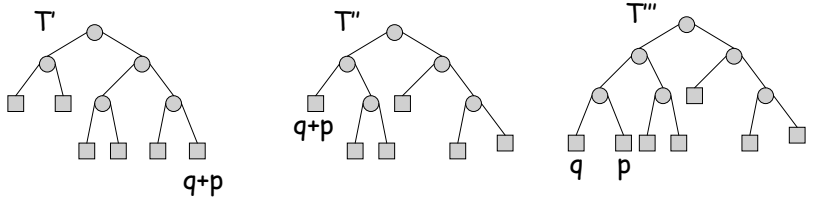
- Assuming we have an optimal tree T whose two lowest probability symbols are siblings at maximum depth, they can be replaced by a new symbol whose probability is the sum of their probabilities.
  - The resulting tree is optimal for the new symbol set.



$C(T') = C(T) + (h-1)(p+q) - hp - hq = C(T) - (p+q)$

Optimality Principle 3 (cont')

- If T' was not optimal then we could find a lower cost tree T''. This will lead to a lower cost tree T''' for the original alphabet.



$C(T''') = C(T'') + p + q < C(T') + p + q = C(T)$

which is a contradiction

Optimality of Huffman's Algorithm

- By the optimality principles, Huffman algorithm is an optimal greedy algorithm: In each stage, the tree that is built by Huffman is a subtree of some optimal tree.

  'the greedy choice property'

- In particular, the final tree is optimal.