Week 3 HoffMan coding + Dynamic Progamming

How we encode Alphabet?

10/17/2018

⇒ how to represent alphabet in binary?

- Straight forward idea: Use 5-bit binary string. (32) [A fixed lenghth code]
  ⇒ Kind of ASCII code idea
- Is there more efficient way?
  ⇒ Yes. If some charactors of alphabet are much more frequent then others, use the "variable-length" code.

What is "fix length" and "variable-length"?

Example: Suppose $\Sigma = \{A, B, C, D\}$ ~~Fixed length~~

Fix length: {00, 01, 10, 11}

Variable length: {0, 01, 10, 1} ⇒ Less data, right?

wait! too Ambiguous e.g. 001 can be AB or AAD

⇒ with variable-length codes, not clear where one charactor ends + the next one begins.

Solution : Prefix-free codes

Every pair $i, j \in \Sigma$, neither of the encodings $f(i), f(j)$ is a prefix of the other.

e.g. $\Sigma = \{\underset{A}{0}, \underset{B}{10}, \underset{C}{110}, \underset{D}{111}\}$

$\Rightarrow$ cannot make other alphabet by the combination of rest of the alphabets.

$\Rightarrow$ data stream can be decoded even there is no space in between.

Example:

Σ	useage	fixed length	variable length
A	60%	00	0
B	25%	01	10
C	10%	10	110
D	5%	11	111

Fixed-length encoding : 2 bit / char

Variable-lengh encoding : 1.55 bit/char → Cheaper

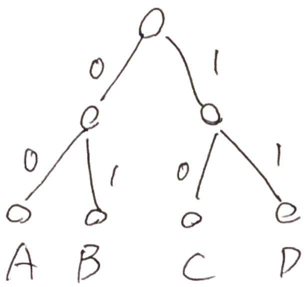
$\Rightarrow$ Inorder to do this, we have to figure out the best binary prefix-free encoding for a given set of char frequency.

$\Rightarrow$ How we can do this?

Think binary code as a binary trees

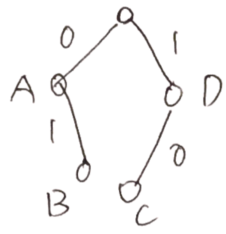
E.g. $\Sigma = \{A, B, C, D\}$

Fixed-length



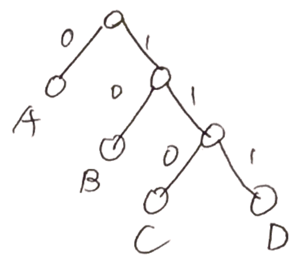
$\{A=00, B=01, C=10, D=11\}$

Defected - varible



$\{A=0, B=01, C=10, D=1\}$

Good - prefix-free



$\{A=0, B=10, C=110, D=111\}$

⇒ Readable.

General

- left child edges ⇔ "0", right child edges ⇔ "1".
- for each $i \in \Sigma$, exactly one node labled "i". (unique)
- Encoding of $i \in \Sigma$ ⇔ bits along path from the root to the node.
- Prefix-free ⇔ labelled nodes = the leaves (No label on middle)

Decoding is easy!

⇒ follow the path from root until you hit the leaf.

Note : encoding length of $i \in \Sigma$ = depth of i in tree.

How to make the tree?

Input: Probability p_i for each char $i \in \Sigma$

Notation: If T = tree with leaves $\Longleftrightarrow$ Symbols of Σ

then $\underbrace{L(T)}_{\text{Average encoding length}} = \sum_{i \in \Sigma} p_i \cdot [\text{depth of } i \text{ in } T]$

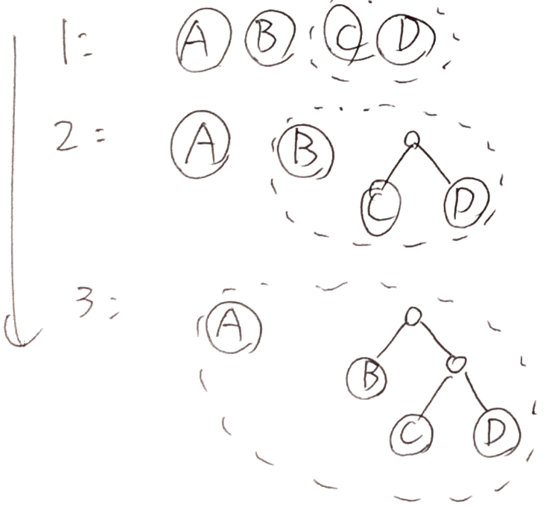
e.g. $60\% \times 1 + 25\% \times 2 + 10\% \times 3 + 5\% \times 3$
$= 1.55$ bit/char

Output: A binary tree T minimizing the average encoding length $L(T)$.

$\rightarrow$ Hoffman coding algorithm.

Natual but suboptimal idea $\Rightarrow$ top-down / divide & conquer.

Optimal idea: Bottom up using successive mergers.

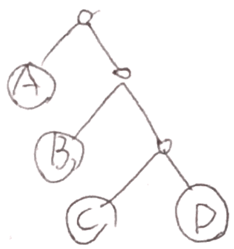


- merger will make non-label node and fuse current tree and new node.

$\Rightarrow$ But what is the greedy factor?

which pair of symbol I should merge?

Observation: Final encoding length of $i \in \Sigma$ = # of merges.



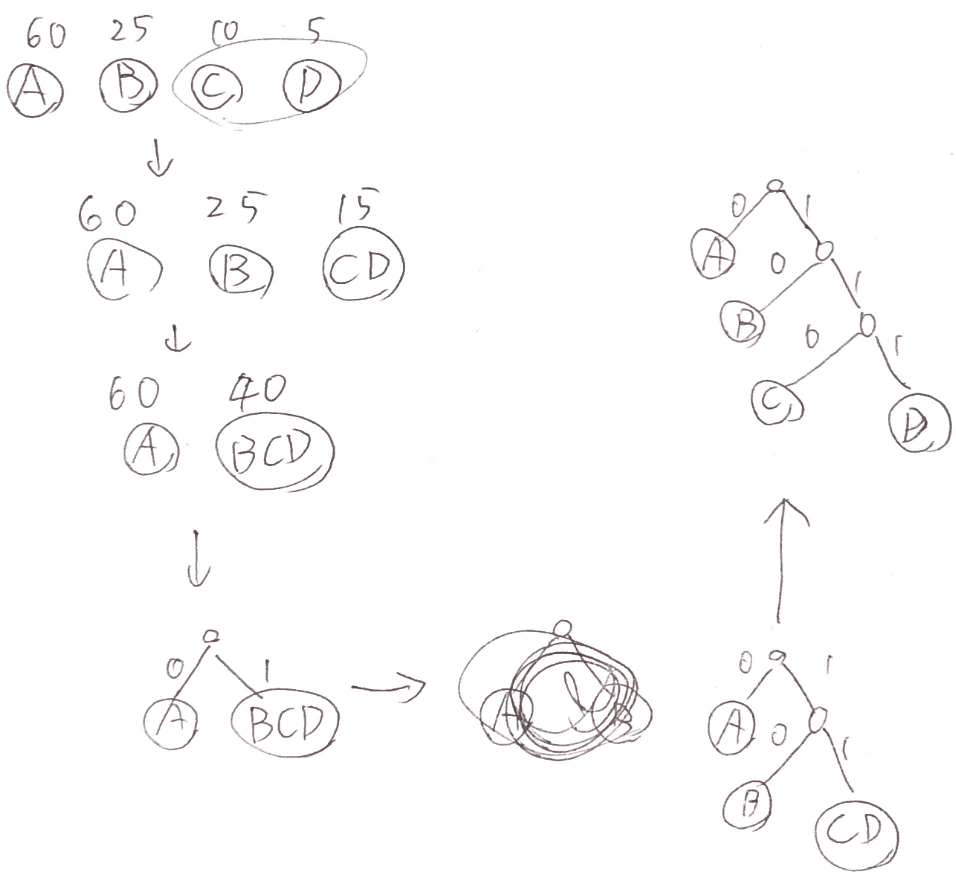
Depth = 3
Merge# = 3 } same

Step by step

1: Choose two least frequencies symbles a & B and merge.

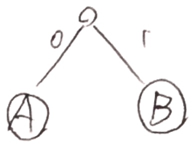
2: Replace the symbols a, b by a new "meta-symble" ab. and set prob a+b

e.g



Hoffman's algorithm

If $|\Sigma| = 2$ return



Let $a, b \in \Sigma$ have the smallest frequencies.

Let $\Sigma' = \Sigma$ with a, b replaced by new symbol ab.

Define $P_{ab} = P_a + P_b$

Recursively compute T' (for the alphabet Σ')

Extend T' (with leas $\leftrightarrow \Sigma'$) to a tree T with leaves $\leftrightarrow \Sigma$ by splitting leaf ab into two leaves a anb b.

Return T

Bigger Example

Input:

Char:	A	B	C	D	E	F
weight:	3	2	6	8	2	6

Step 1: merge B and E. Make connection ~~promise~~ between B and E for later.

A	BE	C	D	F
3	4	6	8	6

Step 2: merge A and BE

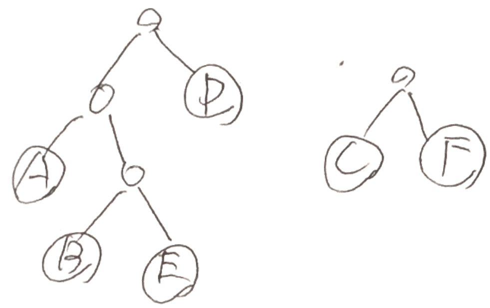
ABE	C	D	F
7	6	8	6

step3: merge C and F

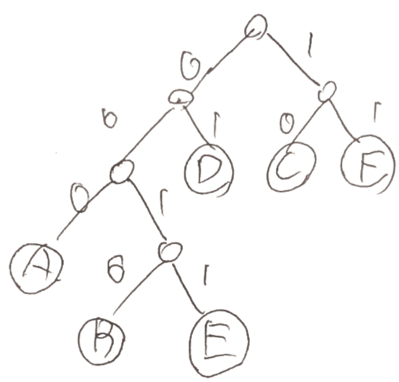
ABE CF D
7 ~~8~~ 12 8

Step4: merge ABE and D

ABDE CF
15 12



last Step5: merge ABDE and CF



A: 000 D: 01
B: 0010 E: 10
C: 0011 F: 11

Running time

Naive: $O(n^2)$

Optimal: Use Heap! Sorted by frequencies

- After extracting the two smallest nodes, Re-insert the new meta-node with new frequency.

$\Rightarrow$ $O(n \log n)$ implemetation.

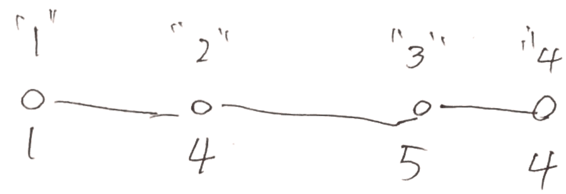
Also, Sorting + $O(n)$ is possible.

Dynamic Programming 10/18/2018 ①

~~One~~ ⇒ Feel counter intuitive at beginning but once you get use to it, fairy easy.

~~Promte~~
Problem

A path graph $G=(V,E)$ with nonnegative weights on vertices.

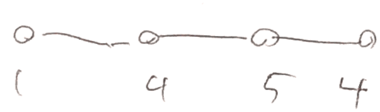


Desired output: Subset of nonadjacent vertices.
- an independent set - of maximum total weight
e.g. ("1","3"), ("1","4"), ("2","4")

Try the algorithm we learned

- Brute force $\Rightarrow O(n^2)$
- Greedy algorithm ⇒ Iteratively choose the max-weight node not adjacent to any previously chosen node

e.g.

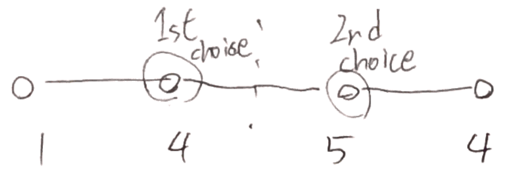


correct answer is
$4+4=8$

but greedy will start from "5, and choose 1 next. Hence
$5+1=6$

- Divide & conquer

Idea: Recursively compute the max-weight in 1st half and 2nd half, and combine the solutions.



Issue: What if recursive sub-solutions conflict?
⇒ Not clear how to quickly fix.
⇒ Even we can, it will be $O(n^2)$.

⇒ Dynamic Programming can solve in $O(n)$.

Time to find the optimized way of doing it.

How to find optimal solution?
⇒ Critical step is to: Reason about structure of an optimal solution.
[In term of optimal solutions of smaller subproblem]

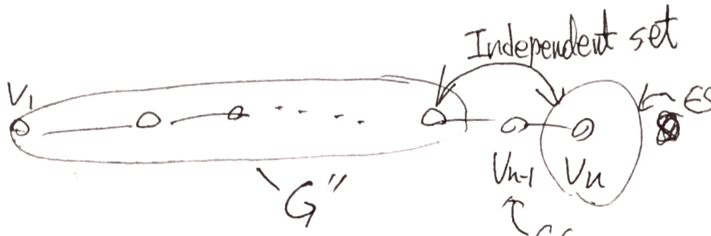
Why we need this?
⇒ To narrow down the set of candidates for the optimal solution; can search through the small set using brute force search.

Notation: Let $S \subseteq V$ be a max-weight independent set (IS), Let V_n = last vertex of path.

Case 1: Suppose $V_n \notin S$, let $G' = G$ with V_n deleted.

Note: - S also an IS of G'.
- S must be a max-weight IS of G' - if S^* was better, it would also be better than S in G [Contradiction]

Case 2: Suppose V_n is part of $\in S$



Note: Previous vertex $V_{n-1} \notin S$, Let $G'' = G$ with V_n, V_{n-1} deleted.
[By definition of an IS]

Note: $S - \{V_n\}$ is an IS of G''.

Note: Must in fact be a max-weight IS of G'' - if S^* is better than S in G'' then S^* and $\cup \{V_n\}$ is better than S in G.
[Contradiction]

Upshot: a max-weight IS must be either
(i) a max-weight IS of G'
(ii) V_n + a max-weight IS of G''.

Corollary: If we know whether or not V_n was in the max-weight IS, could recursively compute the max-weight IS of G' or G'' and be done.

Crazy idea: Try both possibilities + return the better solution.

~~Who~~ Wait! that sounds like a brute force!

$\Rightarrow$ Yes. It is! But the trick is to eliminate the redundancy.

AND RUN IN ~~Θ~~ LINEAR TIME!

Proposed Algorithm

- recursively compute S_1 = max-wt IS of G'.
- recursively compute S_2 = max-wt IS of G''.
- return S_1 or $S_2 \cup [V_n]$, whichever is better.

$\Rightarrow$ Always correct.

$\Rightarrow$ Takes exponential time.

$\Rightarrow$ However, how many distinct subproblems ever got solved by this algorithm?

$\Rightarrow \Theta(n) \longrightarrow$ Only 1 for each "Prefix" of the graph.
[Recursion only plucks nodes off from the right]

Obvious fix: The first time you solve the subproblem, cache its solution in a global table for $O(1)$-time lookup later on.
[Memoization]

Even better: Reformulate as a bottom-up iterative algorithm.
Let G_i = 1st i vertices of G.

Plan: Populate array A left to right with $A[i]$ = Value of max-wt IS of G_i.

Initialization: $A[0]=0$, $A[1]=w_1$

Mainloop: For $i=2,3,4,\dots,n$:

$$A[i]=\max\{\underbrace{A[i-1]}_{\text{Case 1: max-wt IS of } G_{i-1}},\ \underbrace{A[i-2]+w_i}_{\text{Case 2: Max-wt IS of } G_{i-2}+\{v_n\}}\}$$

Runtime: $O(n)$

Correctness: same as recursive version

Is this method really return the optimal solution?

⇒ Understood that DP can construct the table. But how we can get the optimal solution from that table? (Easy to get the optimal value from the table because it will be ~~all the~~ always at last.)

⇒ Change the table that it can also keep the max-wt IS

⇒ Correct but not ideal!

(Don't waste the space and time)

⇒ Just reconstruct the optimal solution from the optimal value.

Is that even possible!?

We know that a vertex v_i belongs to a max-wt IS of G_i $\iff$ $\underbrace{w_i + \text{max-wt IS of } G_{i-2} \geq \text{max-wt IS of } G_{i-1}}_{\text{Follows from correctness of our algorithm.}}$

A Reconstruction Algorithm

Let A = filled-in array :

0	4	4	7	...	...	...	184
0	1	2	3				n

$\xleftarrow{\text{scan}}$

- Let $S = \emptyset$
- while $i \geq 1$ [Scan through array from right to left]
    - If $A[i-1] \geq A[i-2] + w_i$ [i.e. Case 1 wins]
        - decrease i by 1
    - else [i.e. case 2 wins]
        - add v_i to S, decrease i by 2
- return S

Claim: [by induction + our case analysis] final outputs is a max-wt IS of G.

running time : $O(n)$

Just ~~scan~~ scan through

So, what is Dynamic programing?

Fact: Our WIS algorithm is a dynamic programming algo.

Key ingredients of the dynamic programming

① Identify a small number of subproblems
[e.g. compute the max-wt IS of G_i, for $i = 0, 1, 2 \cdots n$]

② Can quickly + correctly solve "larger" subproblems given the solutions to "smaller subproblems"
[Usually via a ~~recursive~~ recurrence such as $A[i] = \max\{A[i-1], A[i-2] \ldots$ fu...

⇒ Identifying a suitable collection of subproblem is the key to solve the problem.

⇒ "Current subproblem's solution is easy to get from the smallest subproblem's solution" ⇒ Important

③ After solving all subproblems, can quickly compute the find solution
[Usually, it's just the answer of the "biggest" subproblem]

Why Dynamic Programming

⇒ Bellman's sarcastic way of ~~for~~ naming to hide the fact that he was working as a mathematition in RAND Instutitute!