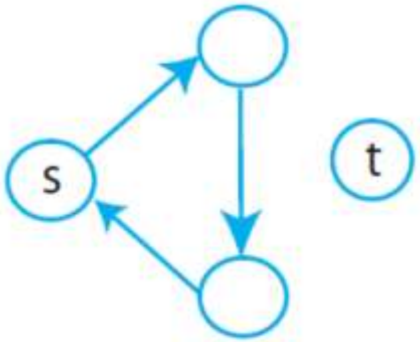
Lecture 06
Dynamic programming II

Spring 2023

Zhihua Jiang

Shortest Paths

- Recursive formulation:
  $\delta(s, v) = \min\{w(u, v) + \delta(s, u) \,\big|\, (u, v) \in E\}$
- Memoized DP algorithm: takes infinite time if cycles!
  in some sense necessary to handle negative cycles



- works for directed acyclic graphs in $O(V + E)$
  effectively DFS/topological sort + Bellman-Ford round rolled into a single recursion

* Subproblem dependency should be acyclic

- more subproblems remove cyclic dependence:
  $\delta_k(s, v)$ = shortest $s \to v$ path $\boxed{\text{using } \leq k \text{ edges}}$

- recurrence:

$$
\begin{aligned}
\delta_k(s, v) &= \min\{\delta_{k-1}(s, u) + w(u, v) \mid (u, v) \in E\} \\
\delta_0(s, v) &= \infty \text{ for } s \neq v \text{ (base case)} \\
\delta_k(s, s) &= 0 \text{ for any } k \text{ (base case, if no negative cycles)}
\end{aligned}
$$

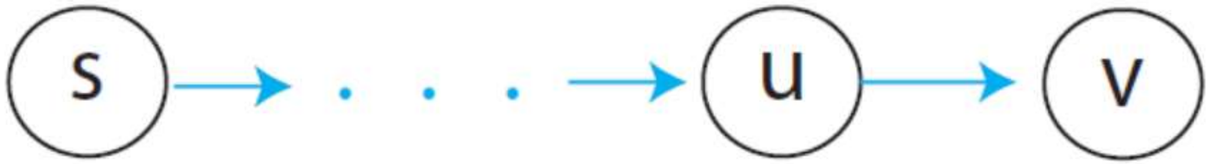
- $\underline{\text{Goal}}$: $\delta(s,v) = \delta_{|V|-1}(s,v)$ (if no negative cycles)
- memoize
- time: $\underbrace{\#\text{ subproblems}}_{|V|\cdot|V|} \cdot \underbrace{\text{time/subproblem}}_{O(v)} = 0(V^3)$
- actually $\Theta(\text{indegree}(v))$ for $\delta_k(s,v)$
- $\implies$ time $= \Theta(V \sum_{v \in V} \text{indegree}(v)) = \Theta(VE)$

BELLMAN-FORD!

Guessing

How to design recurrence

- want shortest $s \to v$ path

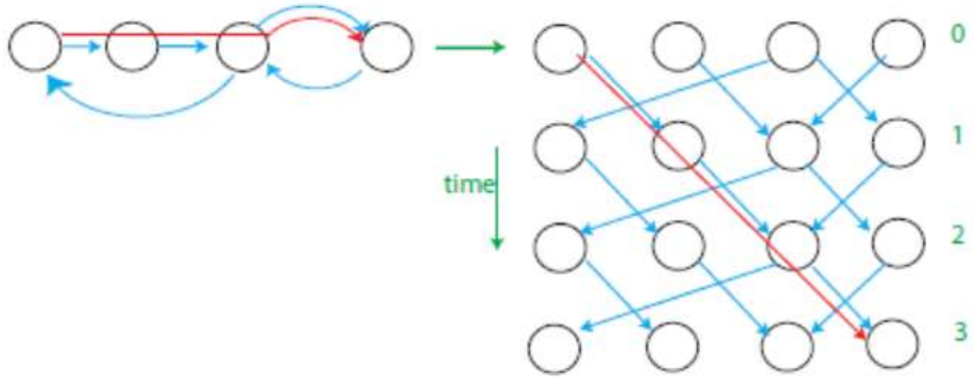


- what is the last edge in path?

- guess it is (u, v)
- path is $\underbrace{\text{shortest } s \to u \text{ path}}_{\text{by optimal substructure}} + \text{edge } (u, v)$
- cost is $\underbrace{\delta_{k-1}(s, u)}_{\text{another subproblem}} + w(u, v)$
- to find best guess, try all ($|V|$ choices) and use best
- * key: small (polynomial) # possible guesses per subproblem — typically this dominates time/subproblem

* DP $\approx$ recursion + memoization + guessing

DAG view



- like replicating graph to represent time
- converting shortest paths in graph → shortest paths in DAG

* DP ≈ shortest paths in some DAG

5 Easy Steps to Dynamic Programming

1. define subproblems — count # subproblems
2. guess (part of solution) — count # choices
3. relate subproblem solutions — compute time/subproblem
4. recurse + memoize — time = time/subproblem · # sub-problems

   OR build DP table bottom-up
   check subproblems acyclic/topological order
5. solve original problem: = a subproblem
   OR by combining subproblem solutions $\Longrightarrow$ extra time

Examples:	Fibonacci	Shortest Paths
subprobs:	F_k	$\delta_k(s,v)$ for $v \in V$, $0 \leq k < \lvert V \rvert$
	for $1 \leq k \leq n$	$=$ min $s \to v$ path using $\leq k$ edges
# subprobs:	n	V^2
guess:	nothing	edge into v (if any)
# choices:	1	indegree$(v) + 1$
recurrence:	$F_k = F_{k-1}$	$\delta_k(s,v) = \min\{\delta_{k-1}(s,u) + w(u,v)$
	$+F_{k-2}$	$\mid (u,v) \in E\}$
time/subpr:	$\Theta(1)$	$\Theta(1 + \text{indegree}(v))$
topo. order:	for $k = 1, \ldots, n$	for $k = 0, 1, \ldots \lvert V \rvert - 1$ for $v \in V$
total time:	$\Theta(n)$	$\Theta(VE)$
orig. prob.:	F_n	$\delta_{\lvert V \rvert - 1}(s,v)$ for $v \in V$
extra time:	$\Theta(1)$	$\Theta(V)$

Independent sets in trees

- Dependent set: subset of nodes $S \subset V$, and there are no edges between them
- Finding the largest independent set in a graph is intractable
- However, it can be solved in linear time when the graph is a tree, using dynamic programming
- Algorithm:
  - Start by rooting the tree at any node r. Each node defines a subtree.
  - The goal is $I(r)$:

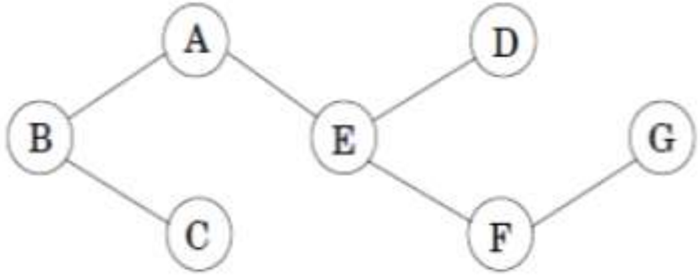
    $I(u)$ = size of largest independent set of subtree hanging from u
  - If know $I(w)$ for all descendants w of u, then compute $I(u)$:

$$I(u) = \max\{1 + \sum_{grandchild} I(gc), \quad \sum_{child} I(c)\}$$

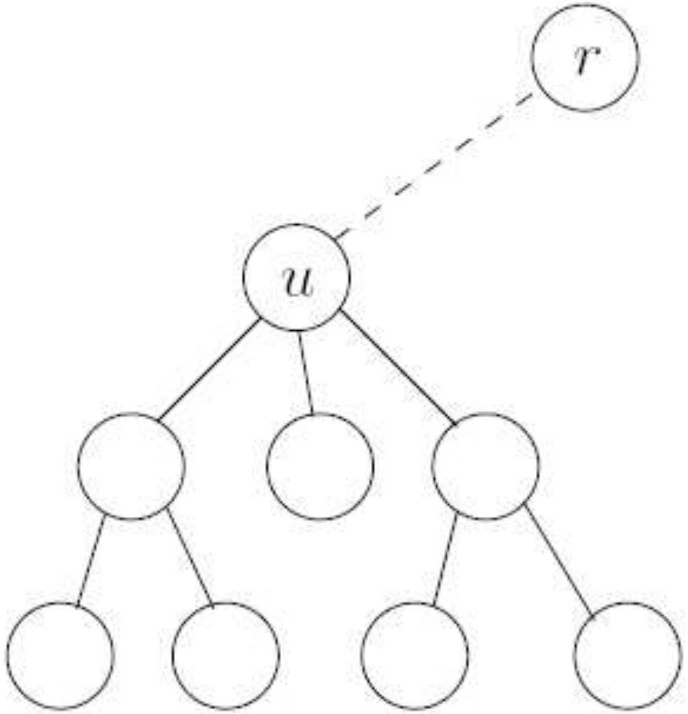
Independent sets in trees

- The number of subproblems: $O(|V|)$
- The running time: $O(|V|+|E|)$

$$I(u) = \max\{1 + \sum_{grandchild} I(gc), \quad \sum_{child} I(c)\}$$



$I(G)=1$
$I(D)=1$
$I(F)=\max\{\underline{1},1\}=1$
$I(E)=\max\{\underline{1+1},1+1\}=2$
$I(C)=1$
$I(A)=\max\{\underline{1+2},2\}=3$
$I(B)=\max\{1+2,\underline{3+1}\}=4$



Exercise 1

A subsequence is *palindromic* if it is the same whether read left to right or right to left. For instance, the sequence

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is *not* palindromic). Devise an algorithm that takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

Subproblems: Define variables $L(i, j)$ for all $1 \leq i \leq j \leq n$ so that, in the course of the algorithm, each $L(i, j)$ is assigned the length of the longest palindromic subsequence of string $x[i, \cdots, j]$.

Algorithm and Recursion: The recursion will then be:

$$L(i, j) = \max\{L(i+1, j), L(i, j-1), L(i+1, j-1) + \text{equal}(x_i, x_j)\}$$

where equal(a, b) is 1 if a and b are the same character and is 0 otherwise, The initialization is the following:

$$\forall i, 1 \leq i \leq n \ , \qquad L(i, i) = 0$$
$$\forall i, 1 \leq i \leq n-1 \ , \quad L(i, i+1) = \text{equal}(x_i, \ x_{i+1} \ )$$

For s=1 to n-1
    for i=1 to n-s
        j=i+s

Correctness and Running Time: Consider the longest palindromic subsequence s of $x[i, \cdots, j]$ and focus on the elements x_i and x_j. There are then three possible cases:

- If both x_i and x_j are in s then they must be equal and $L(i, j) = L(i+1, j-1) + \text{equal}(x_i, x_j)$
- If x_i is not a part of s, then $L(i, j) = L(i+1, j)$.
- If x_j is not a part of s, then $L(i, j) = L(i, j-1)$.

Hence, the recursion handles all possible cases correctly. The running time of this algorithm is $O(n^2)$, as there are $O(n^2)$ subproblems and each takes $O(1)$ time to evaluate according to our recursion.

i

j

	A	C	G	T	G	T	C	A	A	A	A	T	C	G
A	0													
C	0	0												
G	0	0	0				(i, j-1)	(i+1, j-1)						
T	0	0	0	0			**(i,j)**	(i+1, j)						
G	1	1	1	0	0									
T		1	1	1	0	0								
C			1	1	0	0	0							
A				1	0	0	0	0						
A					1	1	1	1	0					
A						1	1	1	1	0				
A							2	2	1	1	0			
T								2	1	1	0	0		
C									1	1	0	0	0	
G										1	0	0	0	0

I

		A	C	G	T	G	T	C	A	A	A	A	T	C	G
J	A	0													
	C	0	0												
	G	0	0	0											
	T	0	0	0	0										
	G	1	1	1	0	0									
	T	1	1	1	1	0	0								
	C	2	2	1	1	0	0	0							
	A	3	2	1	1	0	0	0	0						
	A	3	2	1	1	1	1	1	1	0					
	A	3	2	1	1	1	1	1	1	1	0				
	A	3	2	2	2	2	2	2	2	1	1	0			
	T	3	3	3	3	3	3	2	2	1	1	0	0		
	C	4	4	3	3	3	3	3	2	1	1	0	0	0	
	G	4	4	4	4	4	3	3	2	1	1	0	0	0	0

Exercise 2

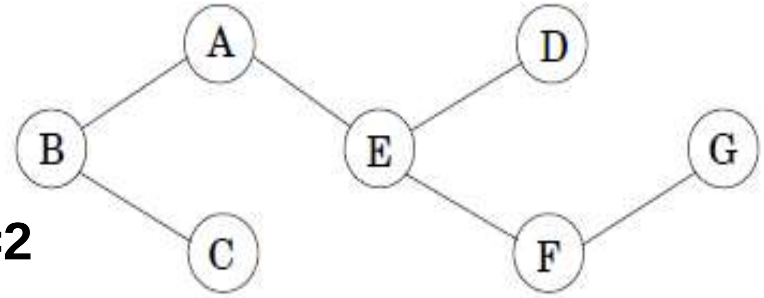
A *vertex cover* of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ that includes at least one endpoint of every edge in E. Give a linear-time algorithm for the following task.

Input: An undirected tree $T = (V, E)$.

Output: The size of the smallest vertex cover of T.

For instance, in the following tree, possible vertex covers include $\{A, B, C, D, E, F, G\}$ and $\{A, C, D, F\}$ but not $\{C, E, F\}$. The smallest vertex cover has size 3: $\{B, E, G\}$.

V(G)=0
V(D)=0
V(F)=min{1,1+0}=1
V(E)=min{2+0,1+1}=2
V(C)=0
V(A)=min{1+1,1+2}=2
V(B)=min{2+2,1+2}=3



Best cover: {B, E, G} or {B, E, F}

$$V(i) = \min\{\#child + \sum V(grandchild), 1 + \sum V(child)\}$$

The subproblem $V(u)$ will be defined to be the size of the minimum vertex cover for the subtree rooted at node u. We have $V(u) = 0$ if u is a leaf, as the subtree rooted at u has no edges to cover. The crucial observation is that if a vertex cover does not use a node it has to use all its neighboring nodes. Hence, for any internal node i

$$V(i) = \min\left\{ \sum_{j:(i,j)\in E} \left(1 + \sum_{k:(j,k)\in E} V(k)\right), 1 + \sum_{j:(i,j)\in E} V(j) \right\}$$

The algorithm can then solve all the subproblems in order of decreasing depth in the tree and output $V(n)$. The running time is linear in n because while calculating $V(i)$ for all i we look at most at $2 * |E| = O(n)$ edges in total.

$$V(i) = \min\{\#child + \sum V(grandchild), 1 + \sum V(child)\}$$

Exercise 4

You are given a string of n characters $s[1 \ldots n]$, which you believe to be a corrupted text document in which all punctuation has vanished (so that it looks something like "itwasthebestoftimes..."). You wish to reconstruct the document using a dictionary, which is available in the form of a Boolean function $\texttt{dict}(\cdot)$: for any string w,

$$\texttt{dict}(w) = \begin{cases} \texttt{true} & \text{if } w \text{ is a valid word} \\ \texttt{false} & \text{otherwise}. \end{cases}$$

(a) Give a dynamic programming algorithm that determines whether the string $s[\cdot]$ can be reconstituted as a sequence of valid words. The running time should be at most $O(n^2)$, assuming calls to `dict` take unit time.

(b) In the event that the string is valid, make your algorithm output the corresponding sequence of words.

a) *Subproblems:* Define an array of subproblems $S(i)$ for $0 \leq i \leq n$ where $S(i)$ is 1 if $s[1 \cdots i]$ is a sequence of valid words and is 0 otherwise.

Algorithm and Recursion: It is sufficient to initialize $S(0) = 1$ and update the values $S(i)$ in ascending order according to the recursion

$$S(i) = \max_{0 \leq j < i} \{S(j) : \texttt{dict}(s[j+1 \cdots i]) = \texttt{true}\}$$

Then, the string s can be reconstructed as a sequence of valid words if and only if $S(n) = 1$.

Correctness and Running Time: Consider $s[1 \cdots i]$. If it is a sequence of valid words, there is a last word $s[j \cdots i]$, which is valid, and such that $S(j) = 1$ and the update will cause $S(i)$ to be set to 1. Otherwise, for any valid word $S[j \cdots i]$, $S(j)$ must be 0 and $S(i)$ will also be set to 0. This runs in time $O(n^2)$ as there are n subproblems, each of which takes time $O(n)$ to be updated with the solution obtained from smaller subproblems.

b) Every time a $S(i)$ is updated to 1 keep track of the previous item $S(j)$ which caused the update of $S(i)$ because $s[j+1 \cdots i]$ was a valid word. At termination, if $S(n) = 1$, trace back the series of updates to recover the partition in words. This only adds a constant amount of work at each subproblem and a $O(n)$ time pass over the array at the end. Hence, the running time remains $O(n^2)$.

HW2-1

A *contiguous subsequence* of a list S is a subsequence made up of consecutive elements of S. For instance, if S is

$$5, 15, -30, 10, -5, 40, 10,$$

then $15, -30, 10$ is a contiguous subsequence but $5, 15, 40$ is not. Give a linear-time algorithm for the following task:

> *Input:* A list of numbers, $a_1, a_2, \ldots, a_n$.
>
> *Output:* The contiguous subsequence of maximum sum (a subsequence of length zero has sum zero).

For the preceding example, the answer would be $10, -5, 40, 10$, with a sum of 55.

(*Hint:* For each $j \in \{1, 2, \ldots, n\}$, consider contiguous subsequences ending exactly at position j.)

HW2-2

Assume $aa=ab=bb=b$, $ac=bc=ca=a$, $ba=cb=cc=c$ on the set $A=\{a, b, c\}$. Given a string $x = x_1x_2...x_n$, design a dynamic programming algorithm to check whether there is a computational order such that the final result is a.

For example,

$x=bbbba \Rightarrow$ Yes. $(b(bb))(ba) = (bb)(ba) = b(ba) = bc = a$

$x=bca \Rightarrow$ No. $(bc)a = aa = b, b(ca) = ba = c$