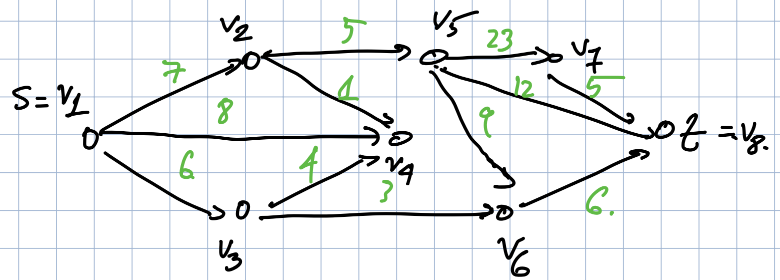
Notes on Dynamic Programming. (How to Think About Algorithms).

Shortest Weighted Path within a Directed Leveled Graph

$G(V, E, w)$, $s, t \in V$. where G is weighted directed. layered graph.



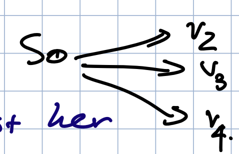
Easiest to assume that an edge can go from v_i to v_j only if $i < j$.

Postcondition An s-t path of min cost (defined as the sum of edge weights)

(A) Suppose there's a stranger and a good friend of yours that you can ask them questions. For now, suppose we can trust the stranger.

(1) Design a question and the list of possible answers for the stranger.

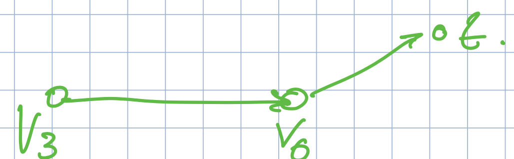
e.g. "Which edge should we take first to form an optimal path to t?
Assuming the out-degree of a node is at most d, the stranger can give at most possibly one out of d answers. (eg. $\{v_2, v_3, v_4\}$).



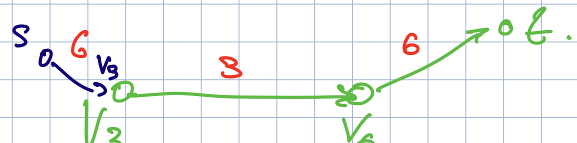
(2) Stranger answers v_3. (Assume we trust her for now).

Now we ask our friend: what's shortest path from v_3 to t?

She answers.



(3) We combine the answers.



$6st = 6 + 3 + 6 = 15.$

④ "TRUST ISSUES" to STRANGER.

Ask our friend best PATHS to t from v_2, v_3, v_4.

⑤ BASIC CODE STRUCTURE. Algo(G(V, E, w), s, t)

if $s = t$.

return $\langle \phi, 0 \rangle$.

else

for each of the edges $s \to v_k$

$\langle$optSubSol, optSubCost$\rangle$ = Algo(G, v_k, t).

$optSol_k = \langle s, v_k \rangle + optSubSol.$

$optCost_k = w_{\langle s, v_k \rangle} + optSubCost.$

$k^*_{min} = \arg\min \; optCost_k.$

return $\langle optSol_{k^*_{min}}, optCost_{k^*_{min}} \rangle$.

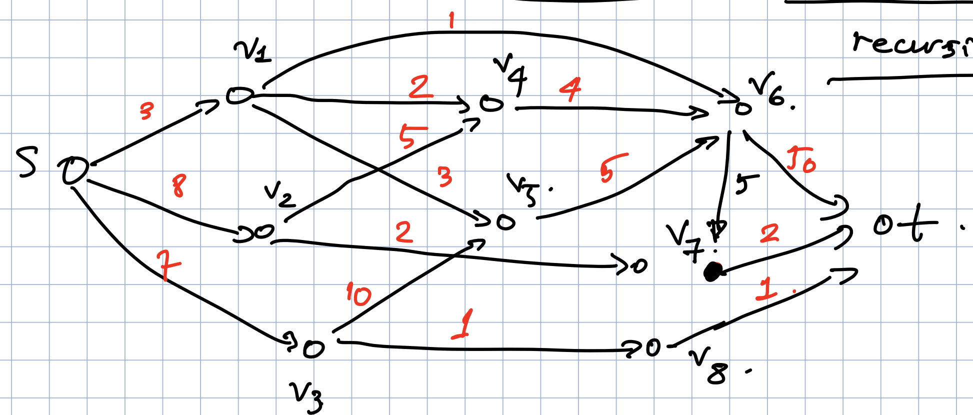
RECURRENCES

$$\text{LeveledGraphSol}(G, s, t) = \min_{k: s \to v_k} (s, v_k) + \text{LeveledGraphSol}(G, v_k, t).$$

$$\text{LeveledGraphCost}(G, s, t) = \min_{k: s \to v_k} w(s, v_k) + \text{LeveledGraphCost}(G, v_k, t).$$

However, implementing the recursion as above (naïvely) leads to exponential run-time. (explain why)

DP Avoids Redundancies. Instead of recursive calls to solve subinstances (memoization + Tabulation + reversing the recursive backtracking)



① Table indexed by subinstances

opt Sol [0...n] where opt Sol [i] will store the best path from v_i to t.

opt Cost [0...n]

② Solutions from subsolutions

$$\text{opt Cost}[i] = \min_k \Big[w(v_i, v_k) + \text{opt Cost}[v_k] \Big]$$

③ $k > i$ this suggests the order in which to fill the order.

LABELED GRAPH (G, s, t).

begin

opt Sol [n] ← φ.
opt Cost [n] ← 0

let's assume $v_n = t$.
$v_0 = s$.

for i ← n−1 to 0. ← "reverse" the reverse backtracking

for each of the edges. (v_i, v_k).

tmp Sol [k] ← (v_i, v_k) + tmp Sol [k]

tmp Cost [k] ← $w(v_i, v_k)$ + tmp.Cost [k]

$k^* \leftarrow \arg\min_k \; tmp.Cost[k]$ // break ties ARBITRARILY.

$optSol[i] \leftarrow tmpSol[k^*]$

$optCost[i] \leftarrow tmpCost[k^*]$.

return $(optSol[0], optCost[0])$.

RUNTIME : $O(nd)$

SPACE : $O(n)$

CLRS Longest-Common Subsequence PROBLEM.

$S_1 =$ A C C G T A T C G C A

$S_2 =$ C A G A T G C T A A C.

How similar?

INSTANCE $X = \langle x_1, \dots, x_n \rangle$, $Y = \langle y_1, \dots, y_m \rangle$ two sequences

Subsequence. $Z = \langle z_1, z_2, \dots, z_\ell \rangle$ is a subsequence of X is a subset of X taken according to X's order.

e.g. $X = \langle B, D, C, A, B, A \rangle$.

$Z = \langle B, A, B, A \rangle$. is a subseq.

Output longest-common subsequence of X, Y of MAX length.

Notice that a greedy algorithm can fail.

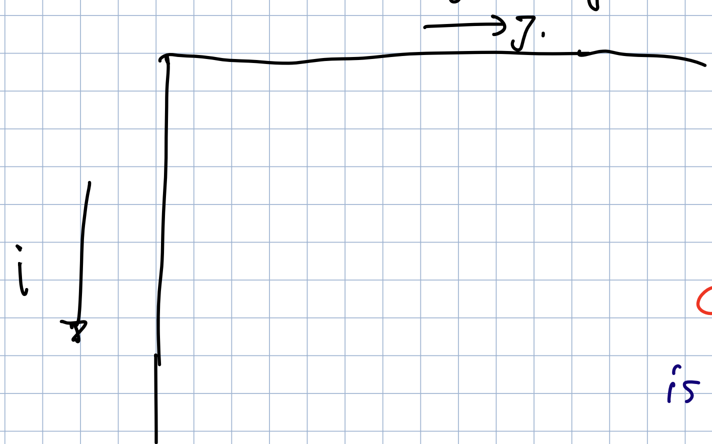
e.g., $X = \langle A, B, C, D \rangle$, $Y = \langle B, C, D, A \rangle$ committing to matching the two As is a mistake.

Question for stranger. let $Z = (z_1, \dots, z_k)$ be an LCS.

$x_n \neq z_k$? $y_m \neq z_k$? or $x_n = y_m = z_k$?

The set of subinstances are of the form.

$\{ \langle x_1, \dots, x_i \rangle, \langle y_1, \dots, y_j \rangle \}$ $\quad 0 \le i \le n, \; 0 \le j \le m$



Notice that if we fill the table according to i+j. this corresponds to. in order of diagonals

is that what we want here?

Theorem.

(1) if $x_n = y_m$, then $z_k = x_n = y_m$ and $Z_{k-1} = \langle z_1, \dots, z_{k-1} \rangle$ is an LCS of X_{n-1}, Y_{m-1}.

(2) if $x_n \neq y_m$, then $z_k \neq x_n$ implies. Z is an. LCS of X_{n-1}, Y.

(3) if $x_n \neq y_m$, then $z_k \neq y_m$ implies. Z is an. LCS of X, Y_{m-1}

Recursive solution

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0. \\ c[i-1,j-1]+1 & \text{if } i,j>0 \text{ AND } x_i = y_j. \\ \max\left(c[i,j-1],\ c[i-1,j]\right) & \text{if } i,j>0,\ x_i \neq y_j. \end{cases}$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑0	↑0	↑0	↖1	←1	↖1
2	B	0	↖1	←1	←1	↑1	↖2	←2
3	C	0	↑1	↑1	↖2	←2	↑2	↑2
4	B	0	↖1	↑1	↑2	↑2	↖3	←3
5	D	0	↑1	↖2	↑2	↑2	↑3	↑3
6	A	0	↑1	↑2	↑2	↖3	↑3	↖4
7	B	0	↖1	↑2	↑2	↑3	↖4	↑4

Figure 15.8 The c and b tables computed by LCS-LENGTH on the sequences $X = \langle A, B, C, B, D, A, B \rangle$ and $Y = \langle B, D, C, A, B, A \rangle$. The square in row i and column j contains the value of $c[i, j]$ and the appropriate arrow for the value of $b[i, j]$. The entry 4 in $c[7, 6]$—the lower right-hand corner of the table—is the length of an LCS $\langle B, C, B, A \rangle$ of X and Y. For $i, j > 0$, entry $c[i, j]$ depends only on whether $x_i = y_j$ and the values in entries $c[i-1, j]$, $c[i, j-1]$, and $c[i-1, j-1]$, which are computed before $c[i, j]$. To reconstruct the elements of an LCS, follow the $b[i, j]$ arrows from the lower right-hand corner; the sequence is shaded. Each "↖" on the shaded sequence corresponds to an entry (highlighted) for which $x_i = y_j$ is a member of an LCS.

(The pseudocode and the example ABOVE ARE from CLRS).

```
LCS-LENGTH(X, Y)
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```