CS146: Data Structures and Algorithms
Lecture 19

DYNAMIC PROGRAMMING
ALL PAIRS SHORTEST SOURCE SHORTEST PATH (CH 25)
LONGEST COMMON SUBSEQUENCE (CH 15)

INSTRUCTOR: KATERINA POTIKA
CS SJSU

DP: Three-step method

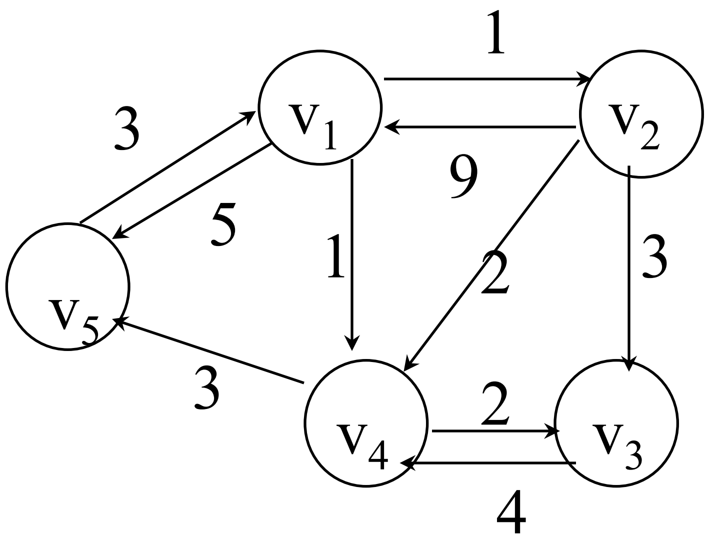
1. Define subproblems
2. Write down the recurrence that relates subproblems
3. Recognize and solve the base cases

All pairs shortest path (Ch 25)

- Input: A weighted graph (may contain negative edges but no negative cycles)
- Output: the shortest path between every pair of vertices of the graph
- *A representation*: a weight matrix where
  W(i,j)=0 if i=j.
  W(i,j)=∞ if there is no edge between i and j.
  W(i,j)="weight of edge"
- Note: we have shown principle of optimality applies to shortest path problems

The weight matrix and the graph

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0



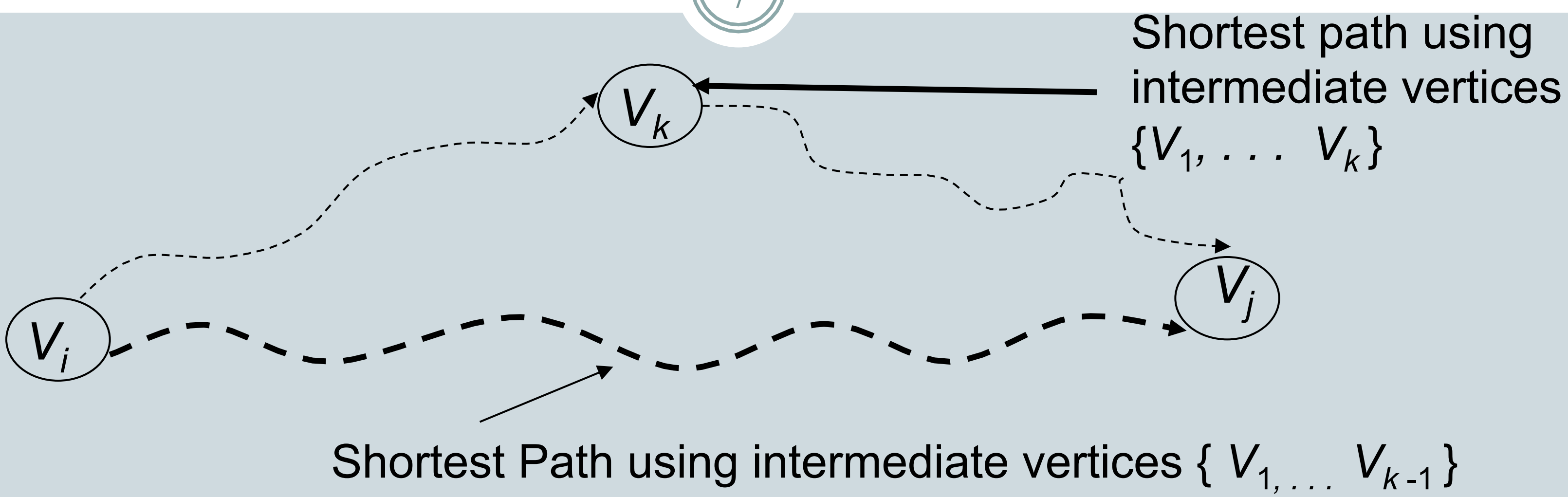
The subproblems

- How can we define the shortest distance $d_{i,j}$ in terms of "smaller" problems?
- One way is to restrict the paths to only include vertices from a *restricted* subset.
- Initially, the subset is empty.
- Then, it is incrementally increased until it includes all the vertices.

The subproblems

- Let $\boldsymbol{D^{(k)}[i,j]}$=weight of a shortest path from $\boldsymbol{v_i}$ to $\boldsymbol{v_j}$ using only vertices from $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices in the path
  - $\boldsymbol{D^{(0)}=W}$
  - $\boldsymbol{D^{(n)}=D}$ **which is the goal matrix**
- How do we compute $D^{(k)}$ from $D^{(k-1)}$?

The Recursive Definition



A shortest path from v_i to v_j restricted to using only vertices from $\{v_1,v_2,\ldots,v_k\}$ as intermediate vertices

- [*Case1*] does **not use** v_k. Then $D^{(k)}[i,j]= D^{(k-1)}[i,j]$
- [*Case2*] does **use** v_k. Then $D^{(k)}[i,j]= D^{(k-1)}[i,k]+ D^{(k-1)}[k,j]$.

The recursive definition

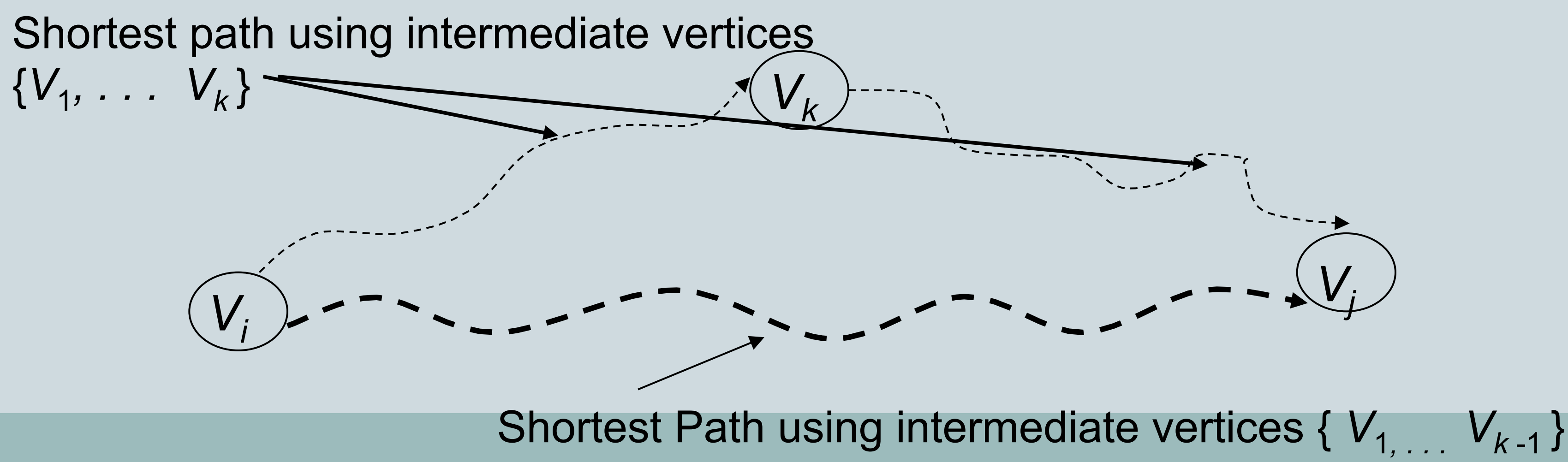
- Since

$$D^{(k)}[i,j] = D^{(k-1)}[i,j] \ \ \textbf{or}$$

$$D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j].$$

We conclude:

$$D^{(k)}[i,j] = \min\{\, D^{(k-1)}[i,j],\, D^{(k-1)}[i,k] + D^{(k-1)}[k,j] \,\}.$$



The pointer array Π

- Used to enable finding a shortest path (solution)
- Initially the array contains 0

- Each time that a shorter path from i to j is found the k that provided the minimum is saved (highest index node on the path from i to j)

- To print the intermediate nodes on the shortest path a recursive procedure that print the shortest paths from i and k, and from k to j can be used

Floyd's Algorithm Using (n+1) *D* matrices

FLOYD-WARSHALL(W, n)

$D^{(0)} \leftarrow W$

for $k \leftarrow 1$ **to** n

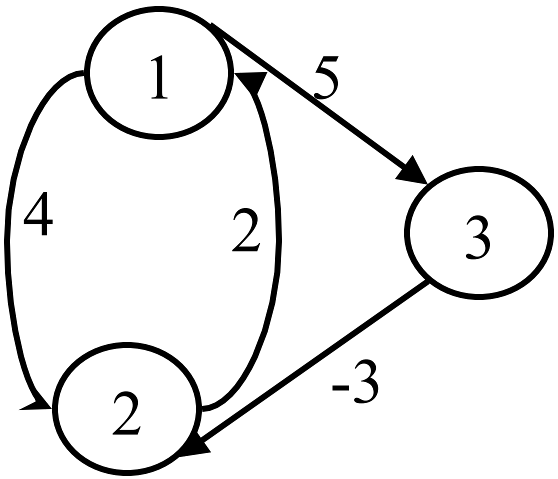
  do for $i \leftarrow 1$ **to** n

    do for $j \leftarrow 1$ **to** n

      do $d_{ij}^{(k)} \leftarrow \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$

return $D^{(n)}$

Example



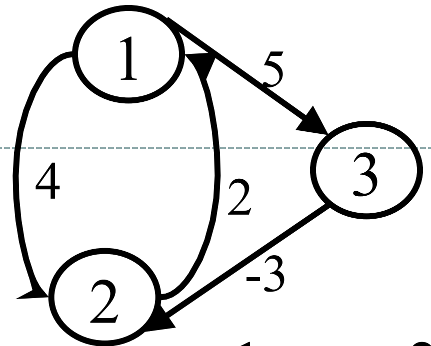
$W = D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$\Pi =$

	1	2	3
1	0	0	0
2	0	0	0
3	0	0	0

k = 1Vertex 1 can be intermediate node



$D^0 =$

	1	2	3
1	0	4	5
2	2	0	∞
3	∞	-3	0

$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

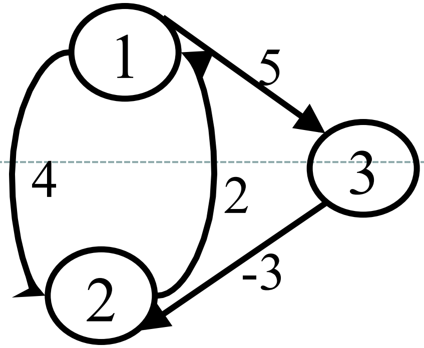
$$D^1[2,3] = \min(D^0[2,3], D^0[2,1]+D^0[1,3]) = \min (\propto, 7) = 7$$

$\Pi =$

	1	2	3
1	0	0	0
2	0	0	1
3	0	0	0

$$D^1[3,2] = \min(D^0[3,2], D^0[3,1]+D^0[1,2]) = \min (-3,\propto) = -3$$

k = 2Vertices 1, 2 can be intermediate



$D^1 =$

	1	2	3
1	0	4	5
2	2	0	7
3	∞	-3	0

$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

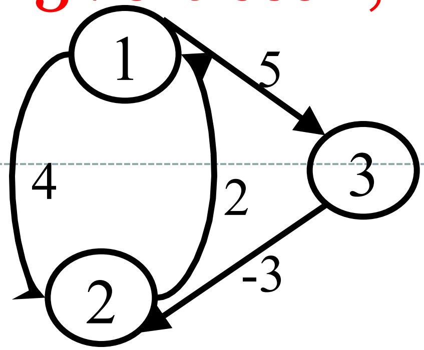
$D^2[1,3] = \min(D^1[1,3],$
$D^1[1,2]+D^1[2,3])$
$= \min (5, 4+7)$
$= 5$

$\Pi =$

	1	2	3
1	0	0	0
2	0	0	1
3	2	0	0

$D^2[3,1] = \min(D^1[3,1],$
$D^1[3,2]+D^1[2,1])$
$= \min (\propto, -3+2)$
$= -1$

k = 3Vertices 1, 2, 3 can be intermediate



$D^2 =$

	1	2	3
1	0	4	5
2	2	0	7
3	-1	-3	0

$D^3 =$

	1	2	3
1	0	2	5
2	2	0	7
3	-1	-3	0

$$D^3[1,2] = \min(D^2[1,2],\ D^2[1,3]+D^2[3,2])$$
$$= \min(4, 5+(-3))$$
$$= 2$$

$\Pi =$

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

$$D^3[2,1] = \min(D^2[2,1],\ D^2[2,3]+D^2[3,1])$$
$$= \min(2, 7+(-1))$$
$$= 2$$

Floyd's Algorithm: Using 2 D matrices


```
D  ← W   // initialize D array to W [ ]
Π ← 0     // initialize Π array to [0]
for k ← 1 to n
  //Computing D’ from D
  for i ← 1 to n
     for j ← 1 to n
        if (D[ i, j ] > D[ i, k ] + D[ k, j ] )
        then  D’[ i, j ] ← D[ i, k ] + D[ k, j ]
              Π [ i, j ] ← k;
        else D’[ i, j ] ← D[ i, j ]
Move D’ to D.
```

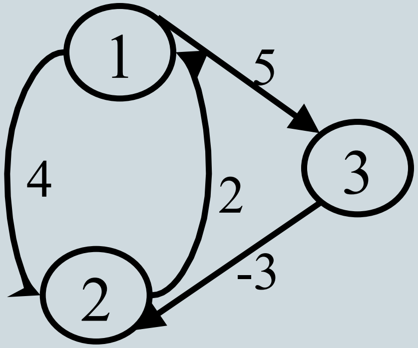
Printing shortest path from q to r


```
path(index q, r)
  if (Π[ q, r ]!=0)
           path(q, Π[q, r])
           println( “v”+ Π[q, r])
           path(Π[q, r], r)
             return;
  //no intermediate nodes
  else return
```

$\Pi =$

	1	2	3
1	0	3	0
2	0	0	1
3	2	0	0

Before calling path check D[q, r] < ∞, and print node q, after the call to path print node r



Longest Common Subsequence (LCS)

- A subsequence of a sequence/string *S* is obtained by deleting zero or more symbols from *S*. For example, the following are some subsequences of “president”: pred, sdn, predent. In other words, the letters of a subsequence of S appear in order in *S*, but they are not required to be consecutive.

- The longest common subsequence problem is to find a maximum length common subsequence between two sequences.
- Applications? Compare the DNA of two (or more) different organisms

DNA

A strand of DNA consists of a string of molecules called ***bases***, possible bases: adenine (A), guanine(G), cytosine(C), and thymine(T).

For example:

the DNA of one organism may be

S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA, and

the DNA of another organism may be

S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.

One reason to compare two strands of DNA is to determine how “similar” the two strands are

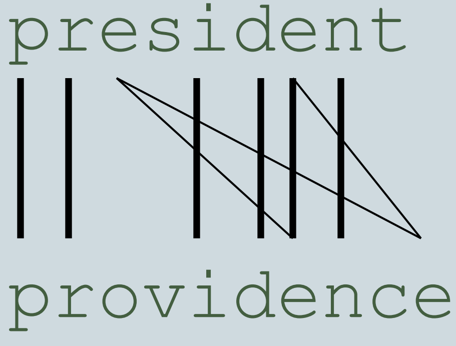
LCS

Example1

Sequence 1: president

Sequence 2: providence

Its LCS is priden.



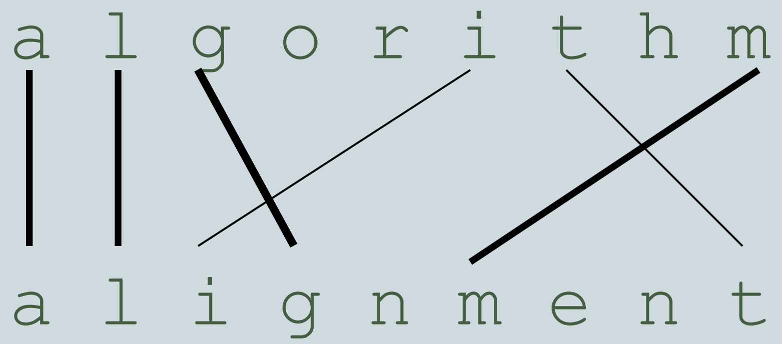
LCS

Example2

Sequence 1: algorithm

Sequence 2: alignment

One of its LCS is algm.



How to compute LCS?

- Let A=$x_1x_2 \ldots x_m$ and B=$y_1y_2 \ldots y_n$.
- $c[i, j]$: the length of an LCS between
  $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_j$
- With proper initializations, $c[i, j]$ can be computed as follows.

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

Algorithm for LCS

```
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
```

Running time: $\Theta(mn)$

Example of algorithm

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Constructing a LCS

Running time: $\Theta(m+n)$

PRINT-LCS(b, X, i, j)

if $i == 0$ or $j == 0$
    return
if $b[i, j] ==$ "↖"
    PRINT-LCS$(b, X, i-1, j-1)$
    print x_i
elseif $b[i, j] ==$ "↑"
    PRINT-LCS$(b, X, i-1, j)$
else PRINT-LCS$(b, X, i, j-1)$

Initial call PRINT-LCS$(b, X, X.length, Y.length)$

Example of LCS

i	*j* *0*	*1* *p*	*2* *r*	*3* *o*	*4* *v*	*5* *i*	*6* *d*	*7* *e*	*8* *n*	*9* *c*	*10* *e*
0	0	0	0	0	0	0	0	0	0	0	0
1 *p*	0	↖ 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1	← 1
2 *r*	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2	← 2
3 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3	← 3	← 3	↖ 3
4 *s*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3	↑ 3	↑ 3
5 *i*	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3	↑ 3	↑ 3	↑ 3	↑ 3
6 *d*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4	← 4	← 4	← 4	← 4
7 *e*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↖ 5	← 5	← 5	↖ 5
8 *n*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↖ 6	← 6	← 6
9 *t*	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 4	↑ 5	↑ 6	↑ 6	↑ 6

Output: *priden*