AALBORG UNIVERSITY

Advanced Algorithms

Lecture 2
All-Pairs Shortest Paths

Bin Yang
byang@cs.aau.dk

Center for Data-intensive Systems

ILO of Lecture 2

- Dynamic programming
  - All-pairs shortest paths
    - To understand the adjacency matrix and the distance/predecessor matrix, which are the representations of the input and output of most of the all-pairs shortest-path algorithms.
    - To understand how the dynamic programming principles play out in the repeated squaring and Floyd-Warshall algorithm.
    - Understand the definition of transitive closure of a directed graph.
  - Activity selection
    - Top-down with memoization.
    - Bottom-up, examples.

All-pairs shortest paths

- Finding shortest paths between all pairs of vertices in a graph.

- Why the problem is useful?
  - E.g., Google Maps, FlexDanmark.

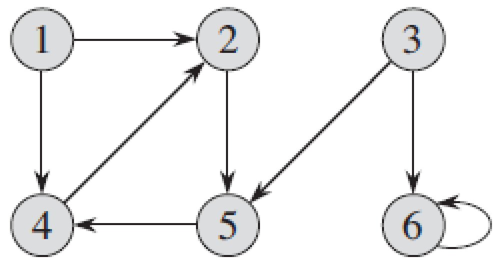
- How to solve the problem efficiently?
  - Repeatedly run one-to-all shortest paths |V| times.
  - Repeated squaring
  - The Floyd-Warshall algorithm

Agenda

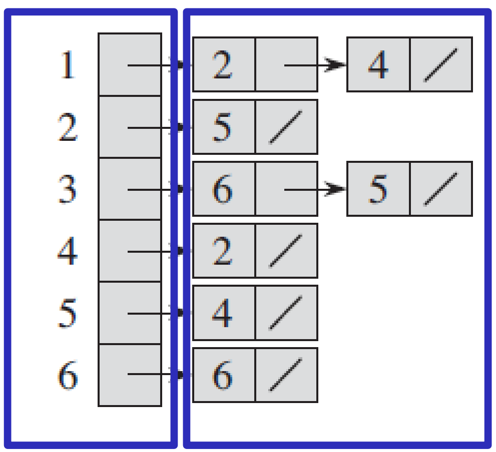
- Recall one-to-all shortest paths
- All-pairs shortest paths
- Repeated squaring algorithm
- Floyd-Warshall algorithm
- Transitive closure of a directed graph
- Activity selection (if time permits)

How to represent a graph?

- Adjacency list vs. adjacency matrix



Space for the array: $\Theta(|V|)$



Space for the linked lists: $\Theta(|E|)$

Total space: $\Theta(|V|+|E|)$

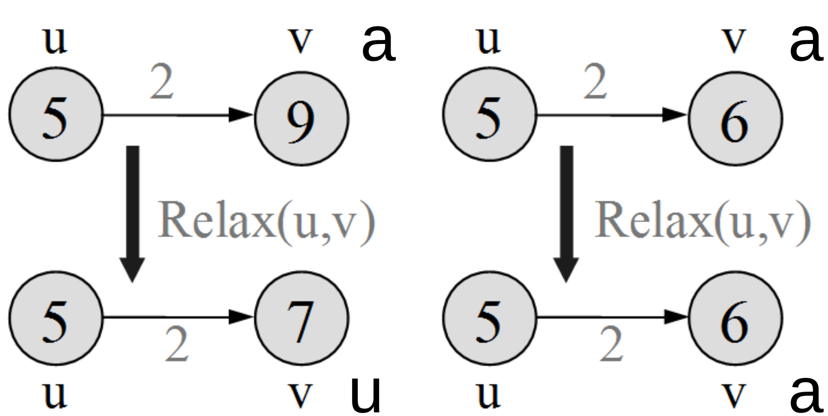
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Space: $\Theta(|V|^2)$

- How to represent edge weights for a weighted graph?

Relaxation

- For each vertex *v* in the graph, we maintain *v*.***d***
  - *v*.***d*** is the estimated distance of the shortest path from *s;*
  - *v*.***d*** is initialized to ∞ at the beginning.
- Relaxing an edge (*u, v*) means testing whether we can improve the shortest path distance to *v* found so far by going through *u.*




```
Relax (u,v,G)
if v.d() > u.d()+G.w(u,v) then
    v.setd(u.d()+G.w(u,v))
    v.setparent(u)
```

Dijkstra's algorithm

```
Dijkstra(G,s)
for each vertex u ∈ G.V()
    u.setd(∞)
    u.setparent(NIL)
s.setd(0)
S ←∅              // Set S is used to explain the algorithm
Q.init(G.V())    // Q is a priority queue ADT
while not Q.isEmpty()
    u ← Q.extractMin()
    S ←S ∪ {u}
    for each v ∈ u.adjacent() do
        Relax(u, v, G)
        Q.modifyKey(v)
```

Initialize all vertices: Θ(|V|)

Initialize Q: O(|V|)

|V| times of Q.extractMin(): |V| * O(lg|V|)

|E| times of edge relax: |E|

|E| times of Q.modifyKey(): |E| * O(lg|V|)

A single Dijkstra's alg needs:
O((|V|+|E|)*lg|V|)
O(|E|*lg|V|)

All-pairs shortest paths:
|V| times of Dijkstra's alg
needs: O(|V||E|lg|V|)

Bellman-Ford

```
Bellman-Ford(G,s)
for each vertex u ∈ G.V()
    u.setd(∞)
    u.setparent(NIL)
s.setd(0)
for i ← 1 to |G.V()|-1 do
    for each edge (u,v) ∈ G.E() do
        Relax (u,v,G)
for each edge (u,v) ∈ G.E() do
    if v.d() > u.d() + G.w(u,v) then
        return false
return true
```

Initialize all vertices: $\Theta(|V|)$

Keep relaxing edges: $\Theta(|V|*|E|)$

Check negative cycles: $O(|E|)$

A single Bellman-Ford: $O(|V|*|E|)$

All-pairs shortest paths: $|V|$ times of Bellman-Ford needs: $O(|V|^2|E|)$

Agenda

- Recall one-to-all shortest paths
- All-pairs shortest paths
- Repeated squaring algorithm
- Floyd-Warshall algorithm
- Transitive closure of a directed graph
- Activity selection

All-pairs shortest path

- Let n = |V|
- Input:
  - Adjacency matrix **W**=(w_{ij}) is an n by n matrix, where w_{ij}
    - 0, if i=j;
    - The weight of directed edge (i, j), if i ≠ j *and* (i, j) is in E.
    - ∞, if i ≠ j *and* (i, j) is not in E.
- Output:
  - Distance matrix ***D***=(d_{ij}) is an n by n matrix, where d_{ij}
    - δ(i, j): the weight of a shortest path from vertex i to vertex j.
  - Predecessor matrix **P**=(p_{ij}) is an n by n matrix, where p_{ij}
    - Nil, if i=j *or* there is no path from vertex i to vertex j.
    - The predecessor of j on a shortest path from i.
  - The i-th row of this matrix encodes the shortest-path tree with root i.
    - One-to-all shortest path finding identifies a row of the predecessor matrix.

Agenda

- Recall one-to-all shortest paths
- All-pairs shortest paths
- Repeated squaring algorithm
- Floyd-Warshall algorithm
- Transitive closure of a directed graph
- Activity selection

Sub-problems

- $l^{(m)}_{ij}$: the minimum weight of any path from vertex i to vertex j that contains at most ***m*** edges.
- When m=0, there is a shortest path from i to j with no edges if and only if i=j.

$$l_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j \ , \\ \infty & \text{if } i \neq j \ . \end{cases} \qquad L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

- For m>0, $l^{(m)}_{ij}$ is the minimum of the following

$$l_{ij}^{(m)} = \min_{1 \le k \le n} \left\{ l_{ik}^{(m-1)} + w_{kj} \right\} .$$

  - n is the number of vertices
  - $l^{(m)}_{ij}$ is the shortest path from i to j using at most m-1 edges
  - Extends the shortest paths computed so far (i.e., $l^{(m-1)}_{ik}$) by one more edge (w_{kj}).
- Optimal sub-structure?
- What m will give the correct shortest path from i to j?

Sub-problems

- $L^{(n-1)}_{ij}$ is the final distance matrix.
  - Path $p=<v_1, v_2, \ldots, v_l>$ is ***simple*** if all vertices are distinct.
  - A shortest path from i to j is ***simple*** and can have at most n-1 edges

$$d_{ij} = \delta(i, j) = l_{ij}^{(n-1)} = l_{ij}^{(n)} = l_{ij}^{(n+1)} = \cdots .$$

  - Why?
    - Recall that n is the number of vertices.
    - This property is also used in Bellman-Ford algorithm.

- Naïve divide and conquer? Overlapping sub-problems?

- Dynamic programming: which order has to be used to compute the solutions to sub-problems?
  - Increasing m from 0 to n-1, i.e., bottom-up.

Algorithm

EXTEND-SHORTEST-PATHS(L, W)

$n = L.rows$
let $L' = (l'_{ij})$ be a new $n \times n$ matrix
for $i = 1$ **to** n
  for $j = 1$ **to** n
    $l'_{ij} = \infty$
    for $k = 1$ **to** n
      $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

Extends the shortest paths computed so far by one more edge.

$$\begin{array}{ccc} L' & & L \\ l_{ij}^{(m)} & = & \min_{1 \le k \le n} \{l_{ik}^{(m-1)} + w_{kj}\} \; . \end{array}$$

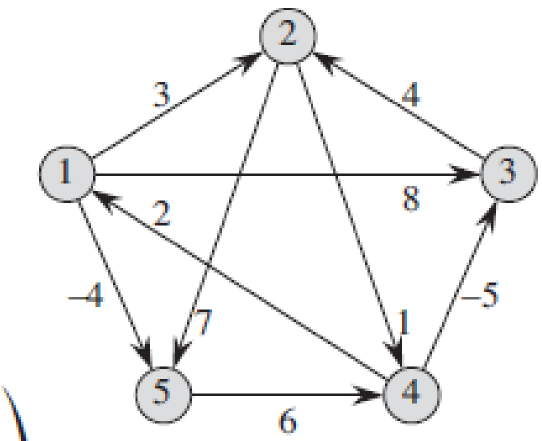
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

$n = W.rows$
$L^{(1)} = W$
for $m = 2$ **to** $n - 1$
  let $L^{(m)}$ be a new $n \times n$ matrix
  $L^{(m)} =$ EXTEND-SHORTEST-PATHS$(L^{(m-1)}, W)$
return $L^{(n-1)}$

Extends n-1 times in total.

Example

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix} \qquad W = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \qquad L^{(3)} = \begin{pmatrix} 0 & 3 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

From 1 to 2 and 2 to 4, so 3+1=4

From 1 to 5 and 5 to 4, so -4+6=2

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{2} & -4 \\ \boxed{3} & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix} \qquad L^{(4)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Extends n-1=5-1=4 times in total.

Run-time

EXTEND-SHORTEST-PATHS(L, W)

$n = L.rows$
let $L' = (l'_{ij})$ be a new $n \times n$ matrix
for $i = 1$ **to** n
    for $j = 1$ **to** n
        $l'_{ij} = \infty$
        for $k = 1$ **to** n
            $l'_{ij} = \min(l'_{ij}, l_{ik} + w_{kj})$
return L'

Is this an efficient algorithm?

Three level of loops.
Each takes n iterations.
Thus, $\Theta(n^3)$

SLOW-ALL-PAIRS-SHORTEST-PATHS(W)

$n = W.rows$
$L^{(1)} = W$
for $m = 2$ **to** $n - 1$
    let $L^{(m)}$ be a new $n \times n$ matrix
    $L^{(m)} =$ EXTEND-SHORTEST-PATHS$(L^{(m-1)}, W)$
return $L^{(n-1)}$

n-1 times of $\Theta(n^3)$
Thus, $\Theta(n^4)$

Improvement – Repeated Squaring

- At step m, instead of extending the shortest paths computed so far by one more edge to get $L^{(m+1)}$, we extend m more edges to get $L^{(2m)}$.
  - $L^{(1)}, L^{(2)}, L^{(3)}, L^{(4)}, L^{(5)}, \ldots, L^{(n-1)}$
  - $L^{(1)}, L^{(2)}, L^{(4)}, L^{(8)}, L^{(16)}, \ldots, L^{(x)}$, where $x = 2^{\lceil \lg(n-1) \rceil}$
    - E.g., if n = 36. n-1 = 35, $\lceil \lg(n-1) \rceil = \lceil 5.2 \rceil = 6$, $2^6 = 64$
    - 35 vs 6 times

```
FASTER-ALL-PAIRS-SHORTEST-PATHS(W)
n = W.rows
L^(1) = W
m = 1
while m < n - 1
    let L^(2m) be a new n × n matrix
    L^(2m) = EXTEND-SHORTEST-PATHS(L^(m), L^(m))
    m = 2m
return L^(m)
```

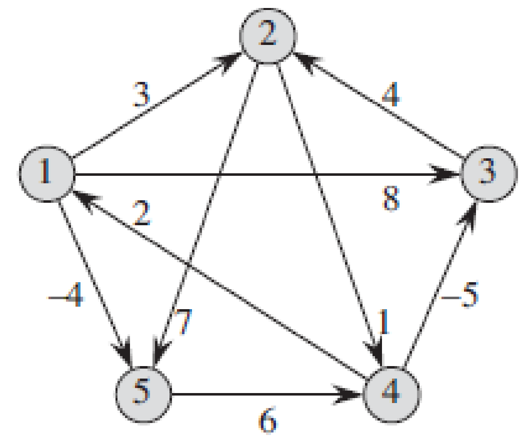
lgn times of $\Theta(n^3)$

Thus, $\Theta(n^3 \lg n)$

Example

$$L^{(0)} = \begin{pmatrix} 0 & \infty & \infty & \cdots & \infty \\ \infty & 0 & \infty & \cdots & \infty \\ \infty & \infty & 0 & \cdots & \infty \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \infty & \infty & \infty & \cdots & 0 \end{pmatrix}$$

$$W= \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$



$$L^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

No need to compute $L^{(3)}$

$$L^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 2 & -4 \\ 3 & 0 & -4 & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & \infty & 1 & 6 & 0 \end{pmatrix}$$

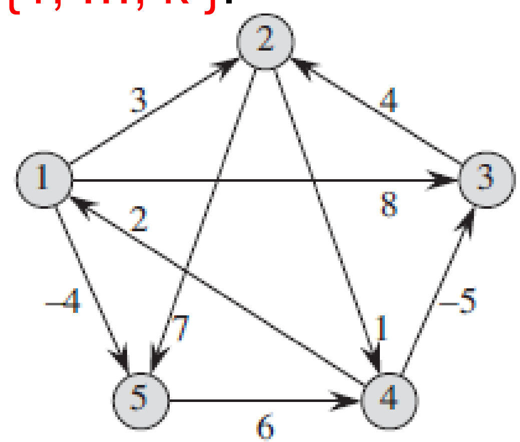
$$L^{(4)} = \begin{pmatrix} \boxed{0} & \boxed{1} & \boxed{-3} & \boxed{2} & \boxed{-4} \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix}$$

Agenda

- Recall one-to-all shortest paths
- All-pairs shortest paths
- Repeated squaring algorithm
- Floyd-Warshall algorithm
- Transitive closure of a directed graph
- Activity selection

The Floyd-Warshall Algorithm

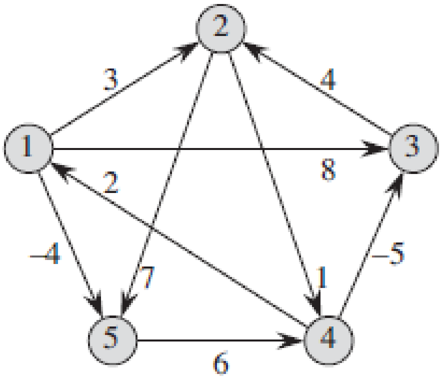
- **Intermediate vertex** of a simple path
  - Path p=$<v_1, v_2, \ldots, v_l>$ is ***simple*** if all vertices are distinct.
  - An intermediate vertex is any vertex of p ***other than*** the source vertex v_1 and the destination vertex v_l.
- Sub-problems
  - $d^{(k)}(i,j)$: minimum weight of a path where the only intermediate vertices (not i or j) allowed are from the set {1, …, k }.
  - $d^{(1)}(1, 3) = 8$, <1,3>, {1}
  - $d^{(1)}(1, 4) = \infty$, no path, {1}
  - $d^{(2)}(1, 3) = 8$, <1,3>, {1, 2}
  - $d^{(2)}(1, 4) = 4$, <1, 2, 4>, {1, 2}
  - $d^{(4)}(1, 3) = -1$, <1, 2, 4, 3>, {1, 2, 3, 4}
  - $d^{(4)}(1, 4) = 4$, <1, 2, 4>, {1, 2, 3, 4}



- Floyd-Warshall algorithm uses $d^{(k)}(i,j)$ as a sub-problem.
  - $d^{(n)}(i,j)$ is the solution to the original problem of all-pairs shortest paths.

Solving sub-problems

- Let p be the shortest path from i to j containing only the intermediate vertices from the set $\{1, \ldots, k\}$.
  - If vertex k **is not** in p then a shortest path with intermediate vertices in $\{1, \ldots, k\text{-}1\}$ is also a shortest path with intermediate vertices in $\{1, \ldots, k\}$.
  - $d^{(k)}(i, j) = d^{(k-1)}(i, j)$
  - E.g.: $d^{(2)}(1, 3) = 8$, <1,3>, {1, 2}
    - $d^{(2)}(1, 3) = d^{(1)}(1, 3) = 8$, since the shortest path p=<1, 3> does not contain 2 as an intermediate vertex.



  - If vertex k **is** an intermediate vertex in p, then we break down p into p_1 (from i to k) and p_2 (from k to j), where p_1 and p_2 are shortest paths with intermediate vertices in $\{1, \ldots, k\text{-}1\}$.
  - $d^{(k)}(i, j) = d^{(k-1)}(i, k) + d^{(k-1)}(k, j)$
  - E.g.: $d^{(4)}(1, 3) = -1$, <1, 2, ***4***, 3>, {1, 2, 3, 4}.
    - $d^{(3)}(1, 4) + d^{(3)}(4, 3) = 4 + (-5) = -1$

      <1, 2, ***4***> <***4***, 3>

Recurrence

- The trivial sub-problems
  - $d^{(0)}(i, j) = w_{ij}$
- Recurrence

k **is** an intermediate vertex in *p*

$$d^{(k)}(i,j) = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(\boxed{d^{(k-1)}(i,j)}, \boxed{d^{(k-1)}(i,k) + d^{(k-1)}(k,j)}\right) & \text{if } k \geqslant 1 \end{cases}$$

k **is not** an intermediate vertex in *p*

- Optimal substructure and overlapping sub-problems?
- DP: Which order has to be used to compute the solutions to sub-problems?
  - Increasing k from 0 to n bottom up.

The Floyd-Warshall Algorithm

```
Floyd-Warshall(W[1..n][1..n])
D ← W      // D(0)
for k ← 1 to n do // compute D(k)
    for i ←1 to n do
        for j ←1 to n do
            if D[i][k] + D[k][j] < D[i][j] then
               D[i][j] ← D[i][k] + D[k][j]
return D
```

$$d^{(k)}(i,j)=\begin{cases} w_{ij} & \text{if } k=0 \\ \min\left(\boxed{d^{(k-1)}(i,j)},\ \boxed{d^{(k-1)}(i,k)+d^{(k-1)}(k,j)}\right) & \text{if } k \geqslant 1 \end{cases}$$

Predecessor matrix

- How do we compute the predecessor matrix?
  - Initialization:

$$p^{(0)}(i,j)=\begin{cases} nil & \text{if } i=j \text{ or } w_{ij}=\infty \\ i & \text{if } i\neq j \text{ and } w_{ij}<\infty \end{cases}$$

  - Updating:

```
Floyd-Warshall(W[1..n][1..n])
D ← W      // D^(0)
for k ← 1 to n do // compute D^(k)
    for i ←1 to n do
        for j ←1 to n do
            if D[i][k] + D[k][j] < D[i][j] then
                D[i][j] ←D[i][k] + D[k][j]
                P[i][j] ←P[k][j]
return D
```

D[i][1]

D[1][j]

$$D^{(0)} = \begin{pmatrix} \boxed{0} & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \boxed{\infty} \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\begin{pmatrix} NIL & 1 & 1 & NIL & \boxed{1} \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & NIL & 4 & NIL & NIL \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

k=1

D[4][1]+D[1][5]<D[4][5]

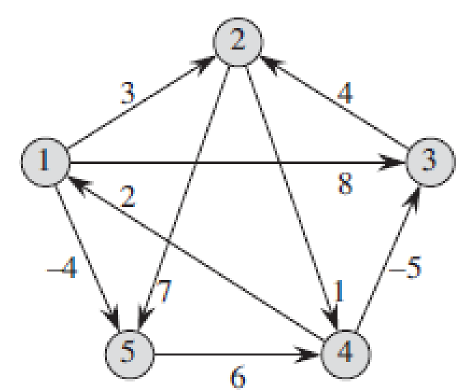
$2+(-4) < \infty$

P[4][5] ← P[1][5]

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & \boxed{-2} \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

$$\begin{pmatrix} NIL & 1 & 1 & NIL & 1 \\ NIL & NIL & NIL & 2 & 2 \\ NIL & 3 & NIL & NIL & NIL \\ 4 & 1 & 4 & NIL & \boxed{1} \\ NIL & NIL & NIL & 5 & NIL \end{pmatrix}$$

```
Floyd-Warshall(W[1..n][1..n])
D ← W       // D(0)
for k ←1 to n do // compute D(k)
    for i ←1 to n do
        for j ←1 to n do
            if D[i][k] + D[k][j] < D[i][j] then
                D[i][j] ←D[i][k] + D[k][j]
                P[i][j] ←P[k][j]
return D
```



$$
\begin{array}{cc}
 & \begin{array}{ccccc} & \text{D[i][2]} & & & \end{array} \\
\text{D[2][j]} & \\
D^{(1)} = & \begin{pmatrix} 0 & 3 & 8 & \boxed{\infty} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}
\end{array}
\qquad
\begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & \boxed{2} & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}
$$

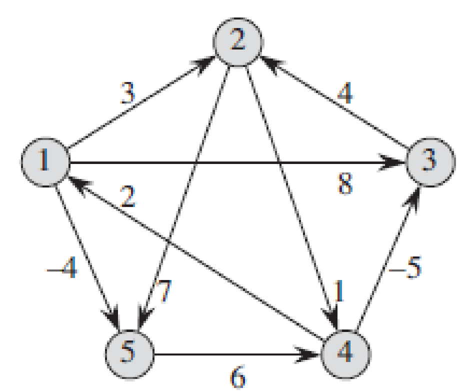
k=2

D[1][2]+D[2][4]<D[1][4]
3+1 < ∞

P[1][4] ← P[2][4]

$$
D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & \boxed{4} & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}
\qquad
\begin{pmatrix} \text{NIL} & 1 & 1 & \boxed{2} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}
$$

```
Floyd-Warshall(W[1..n][1..n])
D ← W      // D(0)
for k ←1 to n do // compute D(k)
    for i ←1 to n do
        for j ←1 to n do
            if D[i][k] + D[k][j] < D[i][j] then
                D[i][j] ←D[i][k] + D[k][j]
                P[i][j] ←P[k][j]
return D
```



Run-time

```
Floyd-Warshall(W[1..n][1..n])
D ← W        // D^(0)
for k ← 1 to n do // compute D^(k)
    for i ←1 to n do
        for j ←1 to n do
            if D[i][k] + D[k][j] < D[i][j] then
                D[i][j] ←D[i][k] + D[k][j]
                P[i][j] ←P[k][j]
return D
```

Three level of loops.
Each takes n iterations.
Thus, $\Theta(n^3)$

Summary

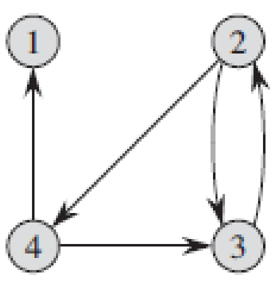
- For a graph with non-negative weights
  - Run |V|=n times of Dijkstra: O(n|E|lgn)
  - In the worst case: $|E|=|V|^2 = n^2$ then, $O(n^3 lgn)$
- For a graph with negative weights
  - Run |V|=n times of Bellman-Ford: $O(n^2|E|)$
  - Worst case: $|E|=|V|^2 = n^2$, $O(n^4)$.
- Repeated squaring $\Theta(n^3 \lg n)$
- Floyd-Warshall ***$\Theta(n^3)$***

Agenda

- Recall one-to-all shortest paths
- All-pairs shortest paths
- Repeated squaring algorithm
- Floyd-Warshall algorithm
- Transitive closure of a directed graph
- Activity selection

Transitive closure of a directed graph

- Given a graph G=(V, E), we may wish to find out whether there is a path in the graph from i to j for all vertex pairs.
  - Indicates reachability from every pair of (i, j).
  - E.g., whether I can go from i to j or whether i is a friend of j.
- Transitive closure of G is defined as G*=(V, E*)
  - E*={(i,j): there is a path from vertex i to vertex j in G}



E*={(1,1),
(2,1), (2,2), (2,3), (2,4),
(3,1), (3,2), (3,3), (3,4),
(4,1), (4,2), (4,3), (4,4)}

Algorithm

- Using Floyd-Warshall to identify transitive closure
  - Assigne weight of 1 to each edge of E.
  - Run the Floyd-Warshall algorithm.
  - If $d^{(n)}(i, j) < n$, there is path from vertex i to vertex j so that it should be included in E*.
  - Otherwise $d^{(n)}(i, j) = \infty$, there is no path so that it is not in E*.

An alternative algorithm

- The same asymptotic run time, but can save time and space in practice.
- Substitutes the logical OR and logical AND for arithmetic min and plus in the Floyd-Warshall algorithm.

$$t_{ij}^{(0)} = \begin{cases} 0 & \text{if } i \neq j \text{ and } (i, j) \notin E , \\ 1 & \text{if } i = j \text{ or } (i, j) \in E , \end{cases}$$

$$t_{ij}^{(k)} = t_{ij}^{(k-1)} \vee \left(t_{ik}^{(k-1)} \wedge t_{kj}^{(k-1)}\right)$$

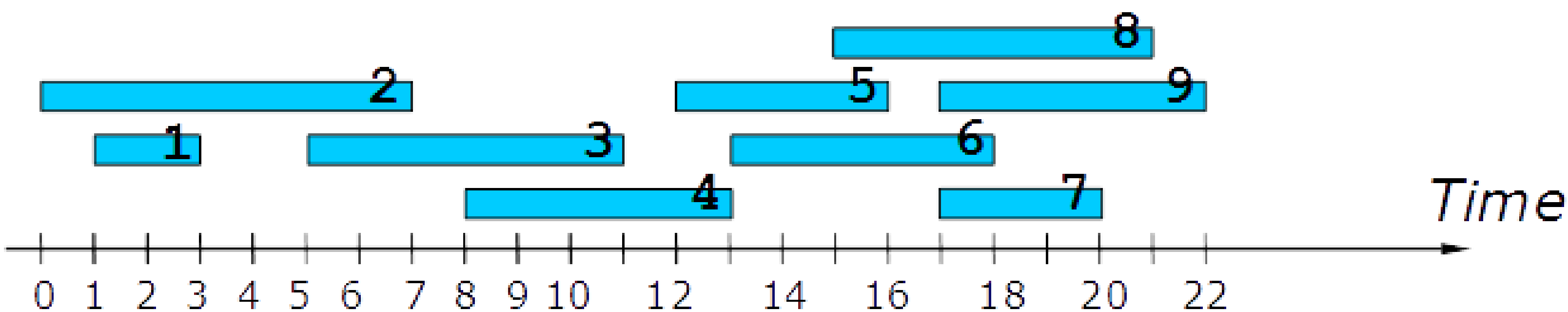
$$\min\left(d^{(k-1)}(i,j), d^{(k-1)}(i,k) + d^{(k-1)}(k,j)\right)$$

Agenda

- Recall one-to-all shortest paths
- All-pairs shortest paths
- Repeated squaring algorithm
- Floyd-Warshall algorithm
- Transitive closure of a directed graph
- Activity selection

Activity Selection

- Input:
  - A set of *n* activities each with start and end times: s_i and f_i. The i-th activity lasts during the period $[s_i, f_i)$.
- Output:
  - The ***largest*** subset of mutually *compatible* activities.
  - Activities are compatible if their intervals do not intersect.



- *Activities 1 and 2 are not compatible.*
- *Activities 2 and 4 are compatible.*

Some definitions

- Sort activities on the end time.
  - Introduce also "sentinel" activities a_0 and a_{n+1}.

0	i	1	2	3	4	5	6	7	8	9	10	11	12
-100	s_i	1	3	0	5	3	5	6	8	8	2	12	100
-100	f_i	4	5	6	7	9	9	10	11	12	14	16	100

- $S_{i,j}$: a set of activities that start after activity a_i finishes and that finish before activity a_j starts.
  - $S_{2,11=}$ $\{a_4, a_6, a_7, a_8, a_9\}$ according to interval $[a_2.e=5, a_{11}.s=12)$.
  - $S_{0,12=}$ $\{a_1, a_2, a_3, a_4, a_5 , a_6, a_7, a_8, a_9 , a_{10} , a_{11}\}$
- $M_{i,j}$: a maximum set of mutually compatible activities in $S_{i,j}$.
- $C_{i,j}$: the cardinality of $M_{i,j}$
- Activity Selection: identify $C_{0,n+1}$

Optimal substructure

- Choose an activity a_k in $S_{i,j,}$ which splits $S_{i,j}$ into $S_{i,k}$ and $S_{k,j}$

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

  - $S_{2,11=}$ {a_4, a_6, a_7, a_8, a_9}
  - a_8, $S_{2,8=}$ {a_4} $S_{8,11=}$ {}
- The maximum number of compatible activities in $S_{i,j}$ *is the maximum of the sum of the following over all possible* a_k
  - *maximum number of* compatible activities in $S_{i,k,}$ i.e., $C_{i,k}$
  - *maximum number of* compatible activities in $S_{k,j}$, i.e., $C_{k,j}$
  - *1, i.e.,* a_k *itself*
- *Trivial sub-problems: 0 if* $S_{i,k}$ is empty.

$$c[i,j]=\begin{cases}0 & \text{if } S_{ij}=\emptyset\,,\\ \max\limits_{a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\emptyset\,.\end{cases}$$

Algorithm, Top down with memoization

```
ActivitySel1m(A, i, j)
if c[i,j] == ∞ then
    c[i,j] = 0
    for k = i+1 to j-1 do
        if not overlaps(A[k], A[i]) and
           not overlaps(A[k], A[j]) then
           sol = ActivitySel1m(A,i,k) +
                 ActivitySel1m(A,k,j) + 1
           if sol > c[i,j] then c[i,j] = sol
return c[i,j]
```

test if a_k is in S_{ij}

$$c[i,j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max\limits_{a_k \in S_{ij}} \{c[i,k] + c[k,j] + 1\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

Activity Selection: identify $C_{0,n+1}$

We should call ActivitySel1m(A, 0, n+1)

Algorithm, bottom-up

	0	1	2	3	4
0	0	0			
1	x	0	0		
2	x	x	0	0	
3	x	x	x	0	0
4	x	x	x	x	0

Only the cells on the upper right part of the diagonal.

The cells on the diagonal and the cells on one-step to the right side of the diagonal are with 0.

$c[0,2]=\max_k\{c[0,k]+c[k,2]+1\}$, k is in [1, 1]
$c[1,3]$ = .., $c[2,4]$ =..

$c[0,3]=\max_k\{c[0,k]+c[k,3]+1\}$, where k is in[1,2]
$c[1,4]$ = …

$c[0,4]=\max_k\{c[0,k]+c[k,4]+1\}$, where k is in[1,3].

Do n-2 iterations in total.
For the i-th iteration, compute the cells that are i+1 steps from the diagonal to the right side, where i=1…n-2.

$$c[i,j]=\begin{cases}0 & \text{if } S_{ij}=\emptyset, \\ \max\limits_{a_k\in S_{ij}}\{c[i,k]+c[k,j]+1\} & \text{if } S_{ij}\neq\emptyset.\end{cases}$$

Note that k must be in [i+1, j-1], but not every integer in [i+1, j-1].

a1	a2	a3
1	2	4
3	5	6

a0	a1	a2	a3	a4
0	1	2	4	10
0	3	5	6	10

	0	1	2	3	4
0	0	0	0	1(a1)	2 (a1,a3)
1	X	0	0	0	1(a3)
2	X	X	0	0	0
3	X	X	X	0	0
4	X	X	X	X	0

c[0,3]=max{c[0,1]+c[1,3]+1}, where k is in[1,2] , only a1 is in $S_{0,3}$
c[1,4] = max{c[1,3]+c[3,4]+1}, where k is in[2,3] , only a3 is in $S_{1,4}$

c[0,2]=0, k is in [1, 1], but a1 is not in $S_{0,2}$
c[1,3] =0, k is in [2, 2] but a2 is not in $S_{1,3}$
c[2,4] =0, k is in [3, 3] but a3 is not in $S_{2,4}$

c[0,4]=max{c[0,1]+c[1,4]+1,
c[0,2]+c[2,4]+1,
c[0,3]+c[3,4]+1}, where k is in[1,3], a1, a2
a3 are in $S_{0,4}$.

Algorithm, bottom-up

- How to write the pseudo code for the bottom-up algorithm?
  - It is quite similar to the Matrix-Chain-Order algorithm shown on p. 375, CLRS.

- What is the run-time?
  - Exercise 15.2-5 gives you some hint.

- What is the space used?

- The first exercise of this lecture.

A different sub-problem formulation

- Alternative way of thinking about it – ***binary choice*** :
- Sort activities on the start time (have “sentinel” activity *A* [*n* +1] after all the other activities)
- Let ***next(i)*** = min {k | k > i ***and*** notOverlaps(A[i], A[k])}
- The sub-problem is then to schedule all the activities starting with the *i-th activity* and after.
- C[i] denotes the maximum number of compatible activities in the set of activities $\{a_i, a_{i+1}, \ldots, a_n\}$.

$$c[i] = \begin{cases} 0 & \text{if } i > n, \\ \max\left(\boxed{1 + c[next(i)]}, \boxed{c[i+1]}\right) & \text{otherwise.} \end{cases}$$

The maximum set includes the i-th activity.

The maximum set does not include the i-th activity.

- Activity Selection: identify c[1]
- *What is the run time and space used* ? Exercise 2.

ILO of Lecture 2

- Dynamic programming
  - All-pairs shortest paths
    - To understand the adjacency matrix and the predecessor matrix, which are the representations of the input and output of most of the all-pairs shortest-path algorithms.
    - To understand how the dynamic programming principles play out in the repeated squaring and Floyd-Warshall algorithm.
    - Understand the definition of transitive closure of a directed graph.
  - Activity selection
    - Top-down with memoization.
    - Bottom-up, examples.
    - An alternative sub-problem formulation.

Lecture 3

- Flow network
  - to understand the formalisms of flow networks and flows;
  - to understand the Ford-Fulkerson method and why it works;
  - to understand the Edmonds-Karp algorithm and to be able to analyze its worst-case running time;
  - to be able to apply the Ford and Fulkerson method to solve the maximum-bipartite-matching problem.