Lakehead
UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

March 8, 2023 (Lecture 13)

Shortest-Paths Problem

In the shortest-paths problem, we are given a weighted, directed graph $G = (V, E)$, with weight function $w : E \rightarrow R$. The weight $w(p)$ of $p = \langle v_0, v_1, \ldots, v_k \rangle$ is the sum of the weights of its constituent edges:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i).$$

The shortest-path weight δ(u, v) from u to v is defined as

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{if there is a path from } u \text{ to } v, \\ \infty & \text{otherwise} \end{cases}$$

A shortest path from vertex u to vertex v is defined as any path p with weight w(p) = δ(u, v).

Single Source Shortest Paths

For the shortest-path problem, we may consider single-source (or single destination, respectively) shortest path which finds a shortest path from a given source to each vertex (or to a given destination from each vertex, respectively).

We also can consider single-pair shortest path which finds a shortest path from a source vertex v to a vertex u. However, all the known algorithms for single-pair shortest path have the same worst-case asymptotic running time as the best single-source algorithms. So we mainly consider the single-source short path problem.

Shortest Path (Part II)

Dijkstra's Algorithm

Dijkstra's algorithm solves the single-source shortest-paths problem on a weighted, directed graph G=(V; E) for the case in which all edge weights are nonnegative.

We assume that w(u,v)>= 0 for each edge (u,v) in E. As we shall see, with a good implementation, the running time of Dijkstra's algorithm is lower than that of the Bellman-Ford a algorithm.

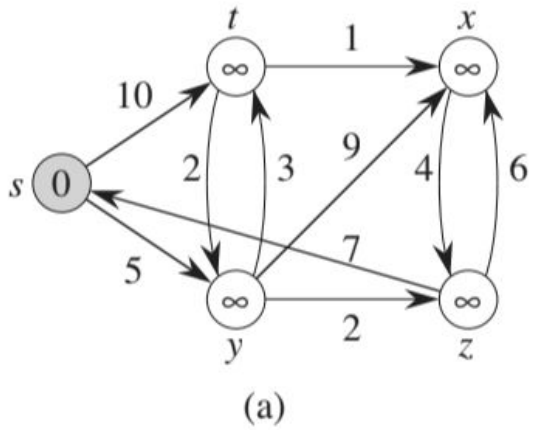
The algorithm maintains a set S of vertices whose final shortest path weights from the source s have already been determined. It uses a min-priority queue Q keyed by their d value.

Dijkstra's Algorithm

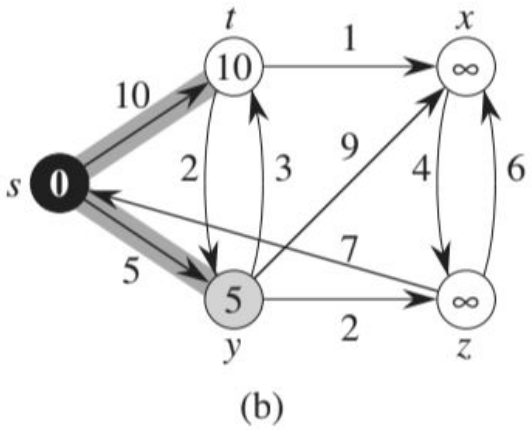
```
DIJKSTRA(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  S = ∅
3  Q = G.V
4  while Q ≠ ∅
5      u = EXTRACT-MIN(Q)
6      S = S ∪ {u}
7      for each vertex v ∈ G.Adj[u]
8          RELAX(u, v, w)
```

```
BELLMAN-FORD(G, w, s)
1  INITIALIZE-SINGLE-SOURCE(G, s)
2  for i = 1 to |G.V| − 1
3      for each edge (u, v) ∈ G.E
4          RELAX(u, v, w)
5  for each edge (u, v) ∈ G.E
6      if v.d > u.d + w(u, v)
7          return FALSE
8  return TRUE
```

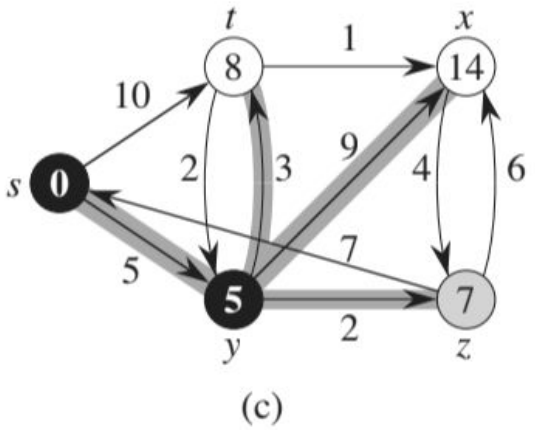
Dijkstra's Algorithm (Example)



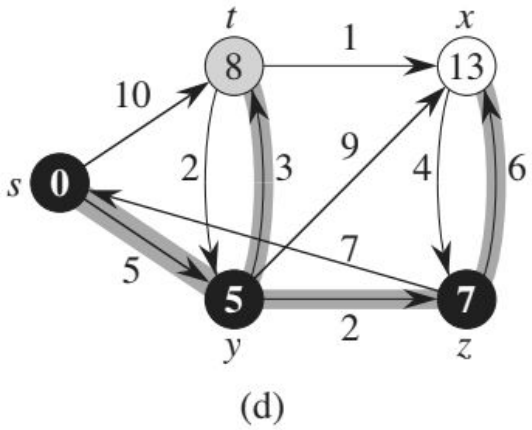
(a)



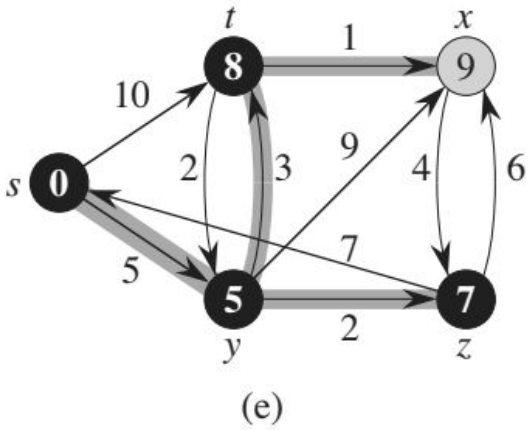
(b)



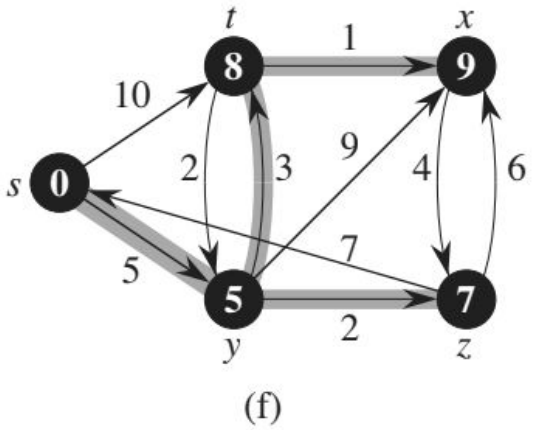
(c)



(d)

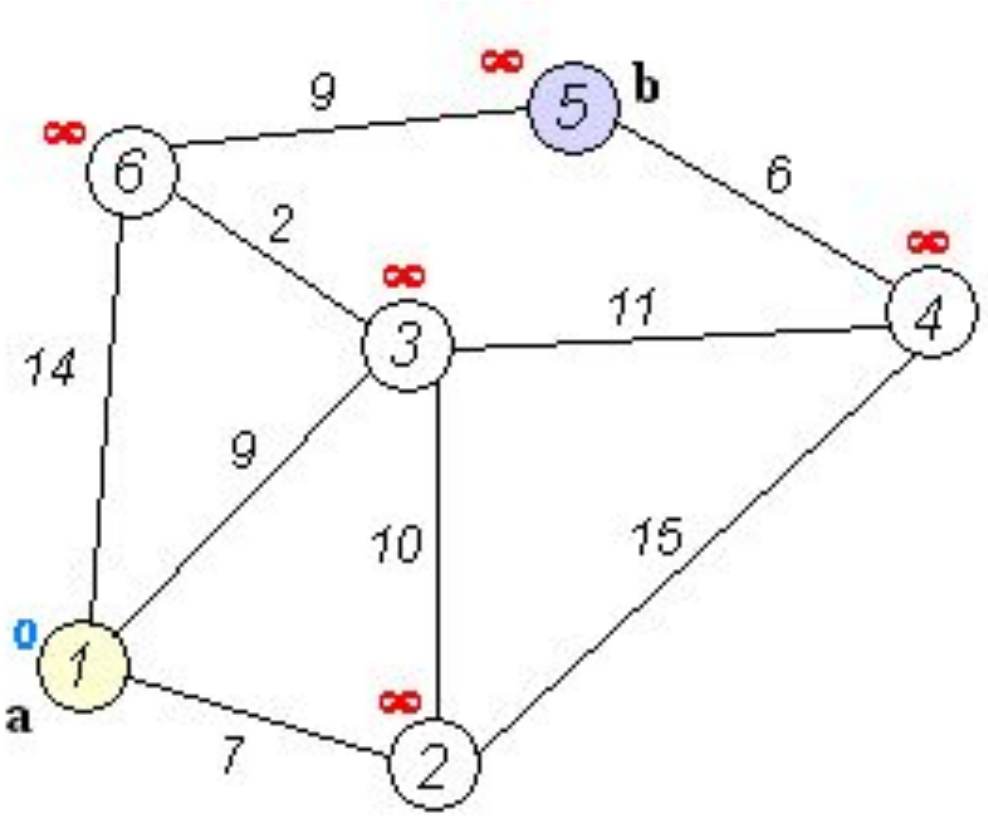


(e)



(f)

Dijkstra's Algorithm (Example)



Dijkstra's Algorithm (Correctness)

Theorem 24.6 [Correctness of Dijkstra's algorithm]

Dijkstra's algorithm, run on a non-negative weighted directed graph G with a source s, terminates with $u.d = \delta(s, u)$ for all vertices $u \in G.V$.

Proof.

We claim that at the start of each iteration of the while loop, $v.d = \delta(s, v)$ for each $v \in S$.

Initially, $S = \varnothing$, so the claim is true. Assume that the claim is not always true and let u be the first vertex for which $u.d \neq \delta(s, u)$ when it is added to S.

Dijkstra's Algorithm (Correctness)

We must have $u \neq s$ because s is the first vertex added to S and $s.d = \delta(s, s) = 0$.

Because $u \neq s$, $S \neq \varnothing$ when u is added to S. There must be some path from s to u otherwise $u.d = \delta(s, u) = \infty$. So there is a shortest path p from s to u. Prior to adding u to S, p connects $s \in S$ and $v \in V - S$. We consider the first vertex y along p such that $y \in V - S$.

Let $x \in S$ be y's predecessor along p. We can decompose path p into $s \overset{p1}{\rightsquigarrow} x \rightarrow y \overset{p2}{\rightsquigarrow} u$. Because the path $s \overset{p1}{\rightsquigarrow} x \rightarrow y$ is the shortest path from s to y and $x.d = \delta(s, x)$, edge (x, y) was relaxed when x was added to S and $y.d = \delta(s, y)$ by convergence property. So we have

Dijkstra's Algorithm (Correctness)

So we have

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \end{aligned}$$

But both u and y were in V − S when u was chosen in line 5, we have u.d ≤ y.d. Therefore we have

$$y.d = \delta(s, y) = \delta(s, u) = u.d.$$

Therefore, our claim is always true. □

Analysis

So we have

$$\begin{aligned} y.d &= \delta(s, y) \\ &\leq \delta(s, u) \\ &\leq u.d \end{aligned}$$

But both u and y were in V − S when u was chosen in line 5, we have u.d ≤ y.d. Therefore we have

$$y.d = \delta(s, y) = \delta(s, u) = u.d.$$

Therefore, our claim is always true. □

All Pairs Shortest Paths

All Pairs Shortest Paths

Now we consider the problem of finding shortest paths between all pairs of vertices in a graph. Suppose we are given a weighted, directed graph G = (V, E) with a weight function w : E $\longrightarrow$ R that maps edges to real-valued weights. We wish to find, for every pair of vertices u, v $\in$ V , a shortest (least-weight) path from u to v, where the weight of a path is the sum of the weights of its constituent edges. We typically want the output in tabular form: the entry in u's row and v's column should be the weight of a shortest path from u to v. We can run Dijkstra's algorithm or Bellman-ford algorithm for each of the vertices, but we want to find more efficient algorithms.

All Pairs Shortest Paths

We will use an adjacency-matrix representation of a graph instead of adjacency-list representation. For convenience, we assume that the vertices are numbered 1, 2, . . . , |V|, and the matrix representation of the directed graph is $W = (w_{ij})$, where

$$w_{ij} = \begin{cases} 0 & \text{if } i = j, \\ \text{the weight of edge}(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E, \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

We allow negative-weight edges, but the input graph contains no negative-weight cycle.

A DP Method

Since when we compute all pairs shortest paths there will be a lot of repeated computations, we consider to use dynamic programming. To do that, we first need to characterize the structure of an optimal solution. Suppose that we represent the graph by an adjacency matrix $W = (w_{ij})$. Consider a shortest path p from vertex i to vertex j , and suppose that p contains at most m edges. Assuming that there are no negative-weight cycles, m is finite. If i = j, then p has weight 0 and no edges. If vertices i and j are distinct, then we decompose path p into $i \overset{p'}{\rightsquigarrow} k \rightarrow j$, where path p' now contains at most $m - 1$ edges. By Lemma 1 (in Lecture 12), p' is a shortest path from i to k, and so $\delta(i, j) = \delta(i, k) + w_{kj}$

A DP Method

Next we consider recursive solution to the problem. So we define $l_{ij}(m)$ be the minimum weight of any path from vertex i to vertex j that contains at most m edges. When m = 0, we have

$$l_{ij}(0) = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j. \end{cases}$$

For $m \geq 1$, we compute $l_{ij}(m)$ as the minimum of $l_{ij}(m-1)$ (the weight of a shortest path from i to j consisting of at most $m - 1$ edges) and the minimum weight of any path from i to j consisting of at most m edges, obtained by looking at all possible predecessors k of j.

A DP Method

Thus, we recursively define

$$l_{ij}(m) = min\{l_{ij}(m-1), min_{1\le k\le n}(l_{ik}(m-1) + w_{kj})\}$$
$$= min_{1\le k\le n}(l_{ik}(m-1) + w_{kj})$$

The latter equality follows since when $k = j$, $w_{kj} = 0$.
For any pair of vertices i and j for which $\delta(i, j) < \infty$, there is a shortest path from vertex i to vertex j that is simple and contains $m \le n - 1$ edges. Therefore

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

In general, we have

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

A DP Method

Taking as our input the matrix $W = (w_{ij})$, we now compute a series of matrices $L(1), L(2), \ldots L(n-1)$, where for $m = 1, 2, \ldots, n-1$, $L(m) = (l_{ij}(m))$. The final matrix $L(n-1)$ contains the actual shortest-path weights. Observe that $l_{ij}(1) = w_{ij}$ for all vertices $i, j \in V$, so $L(1) = W$. The heart of the algorithm is the following procedure, which, given matrices $L(m-1)$ and W, returns the matrix $L(m)$.

A DP Method

```
EXTEND-SHORTEST-PATHS(L, W)
1  n = L.rows
2  let L' = (l'_ij) be a new n × n matrix
3  for i = 1 to n
4      for j = 1 to n
5          l'_ij = ∞
6          for k = 1 to n
7              l'_ij = min(l'_ij, l_ik + w_kj)
8  return L'
```

A DP Method

It is easy to see that the running time for this procedure is $\Theta(n^3)$. We can use the following procedure to compute L (n-1), which as running time as $\Theta(n^4)$.

```
SLOW-ALL-PAIRS-SHORTEST-PATHS(W)
1  n = W.rows
2  L^(1) = W
3  for m = 2 to n - 1
4       let L^(m) be a new n × n matrix
5       L^(m) = EXTEND-SHORTEST-PATHS(L^(m-1), W)
6  return L^(n-1)
```

A DP Method

To improve the algorithm, we can reconsider the recursive formula on Page 19. Recall that $l_{ij}(k) = l_{ij}(m)$ for k > m if there is a shortest path with m edges from vertex i to vertex j. We have

$$
\begin{aligned}
l_{ij}(1) &= w_{ij} \\
l_{ij}(2) &= min_{1 \le k \le n}\{l_{ik}(1) + l_{kj}(1)\} \\
l_{ij}(4) &= min_{1 \le k \le n}\{l_{ik}(2) + l_{kj}(2)\} \\
&\dots \\
l_{ij}(2m) &= min_{1 \le k \le n}\{l_{ik}(m) + l_{kj}(m)\}
\end{aligned}
$$

A DP Method

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

In the above procedure, the while loop runs $\lceil \lg(n-1) \rceil$ times. Since the running time for Extend-Shortest-Paths is $\Theta(n^3)$, the running time for Faster-All-Pairs-Shortest-Paths is $\Theta(n^3 \lg n)$.

The Floyd-Warshall Algorithm

In the Floyd-Warshall algorithm, we characterize the structure of a shortest path differently from how we characterized it in previous section.

The Floyd-Warshall algorithm considers the intermediate vertices of a shortest path, where an intermediate vertex of a simple path $p = \langle v_1, v_2, \ldots, v_l \rangle$ is any vertex of p other than v_1 or v_l , that is, any vertex in the set $\{v_2, \ldots, v_{l-1}\}$.

The Floyd-Warshall Algorithm

As before, we assume that the vertices of G are V = {1, 2, . . . , n}.

Let us consider a subset {1, 2, . . . , k} of vertices for some k. For any pair of vertices i, j ∈ V , consider all paths from i to j whose intermediate vertices are all drawn from {1, 2, . . . , k}, and let p be a minimum-weight path from among them. (Path p is simple.)

The Floyd-Warshall algorithm exploits a relationship between path p and shortest paths from i to j with all intermediate vertices in the set {1, 2, . . . , k − 1}. The relationship depends on whether or not k is an intermediate vertex of path p.

The Floyd-Warshall Algorithm

- If k is not an intermediate vertex of path p, then all intermediate vertices of path p are in the set {1, 2, . . . , k − 1}. Thus, a shortest path from vertex i to vertex j with all intermediate vertices in the set {1, 2, . . . , k − 1} is also a shortest path from i to j with all intermediate vertices in the set {1, 2, . . . , k}.
- If k is an intermediate vertex of path p, then we decompose p into i (p1) $\rightsquigarrow$ k p2 $\rightsquigarrow$ j, By Lemma 4.4.1, p1 is a shortest path from i to k with all intermediate vertices in the set {1, 2, . . . , k − 1}. Similarly, p2 is a shortest path from vertex k to vertex j with all intermediate vertices in the set {1, 2, . . . , k − 1}.

The Floyd-Warshall Algorithm

Let $d_{ij}^{(k)}$ be the weight of a shortest path from vertex i to vertex j for which all intermediate vertices are in the set {1, 2, . . . , k}. When k = 0, a path from vertex i to vertex j with no intermediate vertex numbered higher than 0 has no intermediate vertices at all. Such a path has at most one edge, and hence $d_{ij}^{(0)} = w_{ij}$. Following the above discussion, we define $d_{ij}^{(k)}$ recursively by

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{if } k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & \text{if } k \geq 1 \end{cases}$$

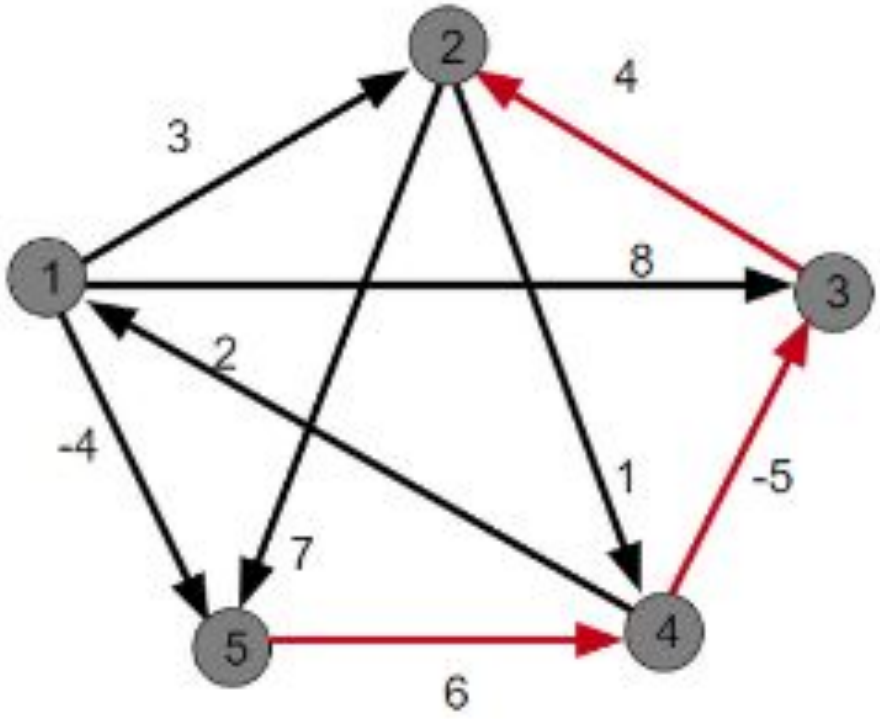
Because for any path, all intermediate vertices are in the set {1, 2, . . . , n}, the matrix $D(n) = d_{ij}^{(n)}$ gives the final answer:

$$d_{ij}^{(n)} = \delta(i, j) \text{ for all } i, j \in V$$

The Floyd-Warshall Algorithm

```
FLOYD-WARSHALL(W)
1  n = W.rows
2  D^(0) = W
3  for k = 1 to n
4      let D^(k) = (d_ij^(k)) be a new n × n matrix
5      for i = 1 to n
6          for j = 1 to n
7              d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
8  return D^(n)
```

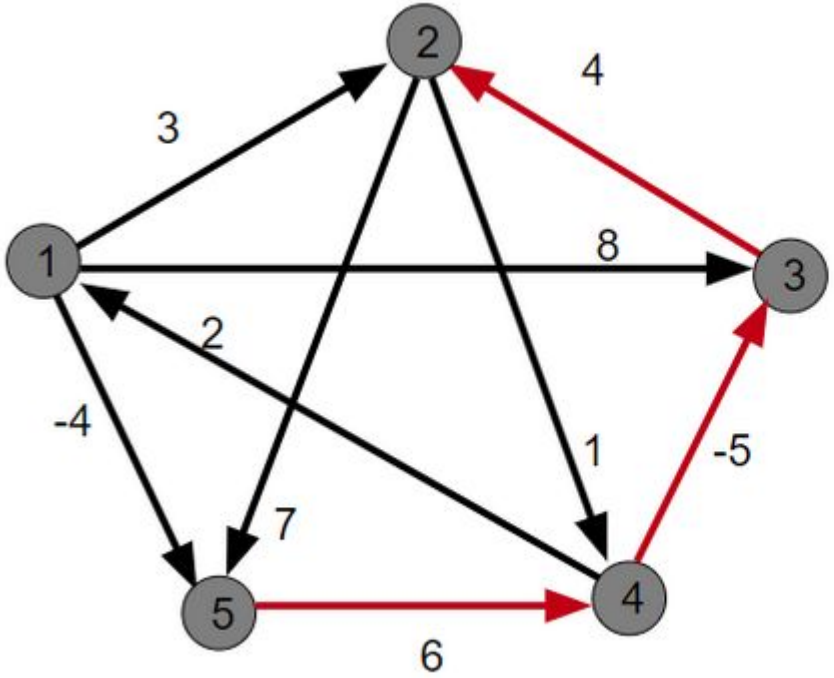
The Floyd-Warshall Algorithm



W

	1	2	3	4	5
1	0	3	8	∞	-4
2	∞	0	∞	1	7
3	∞	4	0	∞	∞
4	2	∞	-5	0	∞
5	∞	∞	∞	6	0

The Floyd-Warshall Algorithm



D

	1	2	3	4	5
1	0	1	-3	2	-4
2	3	0	-4	1	-1
3	7	4	0	5	3
4	2	-1	-5	0	-2
5	8	5	1	6	0

The Floyd-Warshall Algorithm

The running time of the Floyd-Warshall algorithm is determined by the triply nested for loops.

Because each execution of line 7 takes O(1) time, the algorithm runs in time $\Theta(n^3)$.

As the previous dynamic program, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small.

Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

The Floyd-Warshall Algorithm

Now we consider how to construct a shortest path.

We need to define a predecessor matrix $\Pi = (\pi_{ij})$, where π_{ij} is NIL if either i = j or there is no path from i to j, otherwise π_{ij} is the predecessor of j on some shortest path from i. To obtain Π, we compute a sequence of matrices $\Pi(0), \Pi(1), \ldots, \Pi(n)$, where $\Pi = \Pi(n)$ and we define $\pi_{ij}^{(k)}$ as the predecessor of vertex j on a shortest path from j on a shortest path from vertex i with all intermediate vertices in the set {1, 2, . . . , k}. Then we have

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

The Floyd-Warshall Algorithm

For k ≥ 1, if we take the path i ⇝ k ⇝ j, where k ≠ j, , then the predecessor of j we choose is the same as the predecessor of j we chose on a shortest path from k with all intermediate vertices in the set {1, 2, . . . , k − 1}. Otherwise, we choose the same predecessor of j that we chose on a shortest path from i with all intermediate vertices in the set {1, 2, . . . , k − 1}.

Formally, for k ≥ 1,

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & \text{if } d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)}, \\ \pi_{kj}^{(k-1)} & \text{if } d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)}. \end{cases}$$

The Floyd-Warshall Algorithm

For each vertex $i \in V$, define the predecessor subgraph of G for i as $G_{\pi,i} = (V_{\pi,i}, E_{\pi,i})$, where

$$V_{\pi,i} = \{j \in V : \pi_{ij} \neq \text{NIL}\} \cup \{i\} \text{ and } E_{\pi,i} = \{(\pi_{ij}, j) : j \in V_{\pi,i} - \{i\}\}.$$

If $G_{\pi,i}$ is a shortest-paths tree, then we can use the following procedure to print a shortest path from vertex i to vertex j.

The Floyd-Warshall Algorithm

```
Print-All-Pairs-Path(Π, i, j).
1:  if i == j then
2:      print i
3:  else if π_ij == NIL then
4:      print "no path from i to j exists"
5:  else
6:      Print-All-Pairs-Shortest-Path(Π, i, π_ij )
7:      print j
```

For the Π from the Floyd-Warshall algorithm, it can be proved that $G_{\pi,i}$ is a shortest path tree with root i.

Transitive Closure

Now we introduce the transitive closure of a directed graph $G = (E, V)$, which is a graph $G^* = (V, E^*)$, where
$E^* = \{(i, j) : \text{there is a path from vertex } i \text{ to vertex } j \text{ in } G\}$.
One way to compute the transitive closure of a graph in $\Theta(n^3)$ time is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$.

There is another, similar way to compute the transitive closure of G in $\Theta(n^3)$ time that can save time and space in practice. This method substitutes the logical operations $\vee$ (logical OR) and $\wedge$ (logical AND) for the arithmetic operations min and + in the Floyd-Warshall algorithm.

Transitive Closure

For i, j, k = 1, 2, . . . , n, we define $t_{ij}^{(k)}$ to be 1 if there exists a path in graph G from vertex i to vertex j with all intermediate vertices in the set {1, 2, . . . , k}, and 0 otherwise. We construct the transitive closure $G^* = (V, E^*)$ by putting edge (i, j) into E^* if and only if $t_{ij}^{(n)} = 1$. A recursive definition of $t_{ij}^{(k)}$, analogous to recurrence on Slide 31, is

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

and for k ≥ 1,

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

Transitive Closure

We compute the matrices T (k) = ($t_{ij}^{(k)}$) in order of increasing k

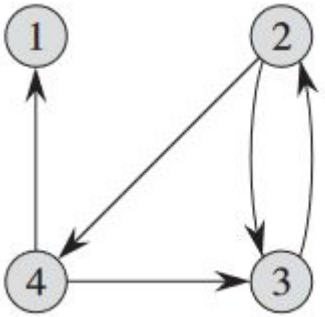
```
TRANSITIVE-CLOSURE(G)
 1  n = |G.V|
 2  let T^(0) = (t_ij^(0)) be a new n × n matrix
 3  for i = 1 to n
 4      for j = 1 to n
 5          if i == j or (i, j) ∈ G.E
 6              t_ij^(0) = 1
 7          else t_ij^(0) = 0
 8  for k = 1 to n
 9      let T^(k) = (t_ij^(k)) be a new n × n matrix
10      for i = 1 to n
11          for j = 1 to n
12              t_ij^(k) = t_ij^(k−1) ∨ (t_ik^(k−1) ∧ t_kj^(k−1))
13  return T^(n)
```

Transitive Closure

The above procedure also runs in $\Theta(n^3)$ time. But on some computers, logical operations on single-bit values execute faster than arithmetic operations on integer words of data.

Moreover, because the direct transitive-closure algorithm uses only boolean values rather than integer values, its space requirement is less than the Floyd-Warshall algorithm's by a factor corresponding to the size of a word of computer storage.

Transitive Closure



$$T^{(0)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(1)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 \end{pmatrix}$$

$$T^{(2)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{pmatrix} \quad T^{(3)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix} \quad T^{(4)} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \end{pmatrix}$$

After Class

- After class: Part VI 24.1 and 24.2