Lecture 22: Shortest Paths Cont.

- Quick Review of Previous Class

- Concept of Edge Relaxation

Relax(u, v)
 If $u.d + w(u,v) < v.d$ *Then*
 $v.d = u.d + w(u,v)$
 $v.p = u$

- Bellman-Ford Algorithm: relax all edges $V - 1$ times in arbitrary order $\Theta(VE)$.

- Shortest path in a *Directed Acyclic Graph*: relax all edges exactly once in *topological order* $\Theta(V + E)$.

- Algorithms work with negative weights.

- Shortest paths are not applicable for negative cycles.

Outline

- Single Source Shortest Path
  - Dijkstra Algorithm
- All-Pairs Shortest Paths
  - First DP Formulation
  - 2nd DP Formulation
  - Floyd-Warshall

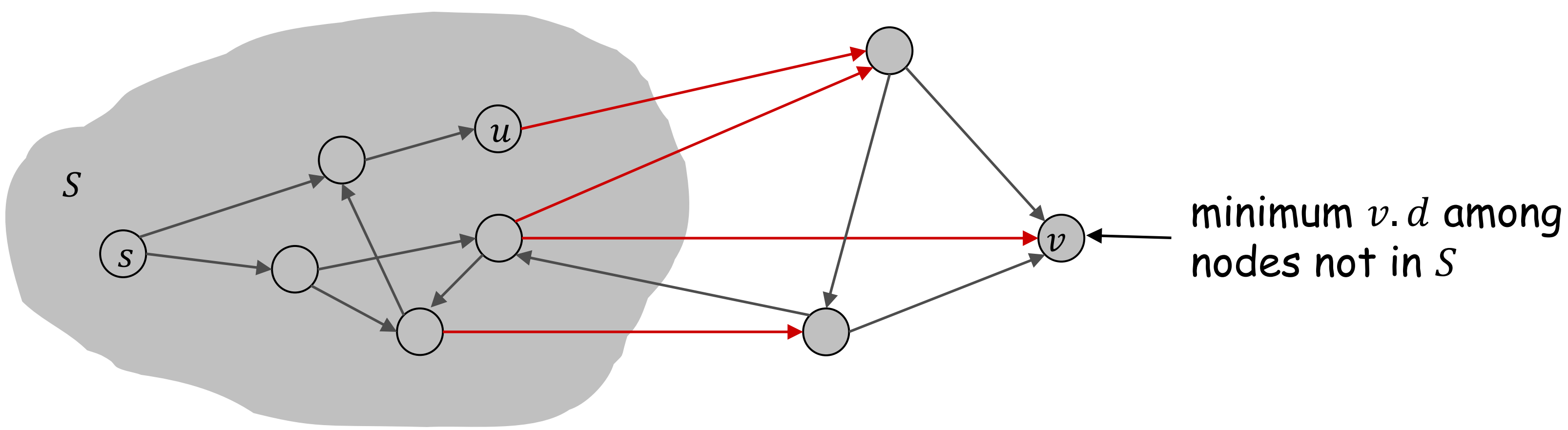
SPs in a graph with cycles and nonnegative weights

Dijkstra's algorithm.

- Maintain a set S of explored nodes.
  Initialize $S = \{s\},\ s.d = 0,\ v.d = \infty$.
  Assume we know, $\forall u \in S, u.d = \delta(s,u)$.

Key lemma: If all edges leaving S were already *relaxed*, let v be the vertex in $V - S$ with the minimum $v.d$. Then $v.d = \delta(s,v)$,

- This v can then be added to S, and process repeated.



Dijkstra's Algorithm

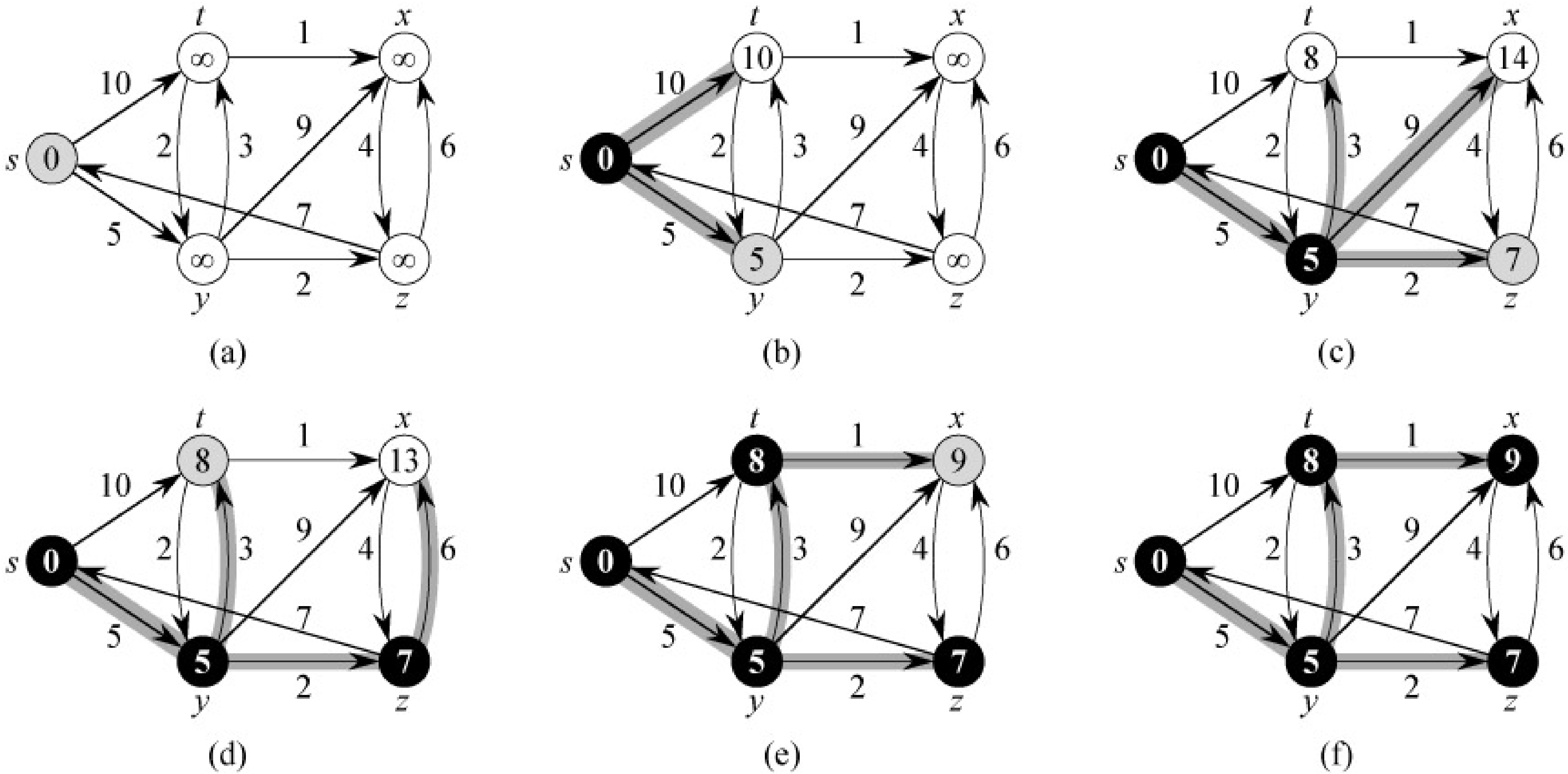
```
Dijkstra(G,s):
for each v ∈ V do
    v.d ← ∞, v.p ← nil, v.color ← white
s.d ← 0
insert all nodes into a min-heap Q with d as key
while Q ≠ ∅
    u ← Extract-Min(Q)
    u.color ← black
    for each v ∈ Adj[u] do  % relax all edges leaving v
        if v.color = white and u.d + w(u,v) < v.d then
            v.p ← u
            v.d ← u.d + w(u,v)
            Decrease-Key(Q,v,v.d)
```

Running time: $O(E \log V)$

- Very similar to Prim's algorithm

Analysis Assumption:
G is connected so $V = O(E)$.

Dijkstra's Algorithm: Example



Note: All the shortest paths found by Dijkstra's algorithm form a tree (shortest-path tree).

Dijkstra's Algorithm: Implementation

```
Dijkstra(G,s):
for each v ∈ V do
      v.d ← ∞, v.p ← nil, v.color ← white
s.d ← 0
Insert(Q,s,s.d)
while Q ≠ ∅
      u ← Extract-Min(Q)
      if u.color = black then continue
      output (u,u.d,u.p)
      u.color ← black
      for each v ∈ Adj[u] do   % relax all edges leaving v
            if v.color = white and u.d + w(u,v) < v.d then
                  v.p ← u
                  v.d ← u.d + w(u,v)
                  Insert(Q,v,v.d)
```

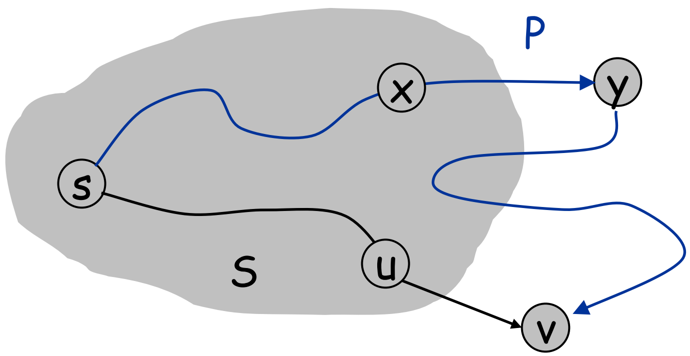
Running time: $O(E \log V)$

Dijkstra's Algorithm: Correctness

Lemma. Suppose $u.d = \delta(s,u)$ for all $u \in S$, and all edges leaving S have been relaxed. Then $v.d = \delta(s,v)$, where v is the vertex with minimum $v.d$ in $V - S$.

Pf. (by contradiction) (assume $v.d \neq \delta(s,v)$)

- Note that $v.d$ starts $= \infty$. Whenever $v.d$ is updated, it's because a path with distance $v.d$ was found. So always have $v.d \geq \delta(s,v)$.
  Thus if $v.d \neq \delta(s,v)$ then $v.d > \delta(s,v)$.



- Consider a shortest path P from s to v.

  - Suppose $x \to y$ is the first edge on P that takes P out of S.
  - Since $x \in S$, we have $x.d = \delta(s,x)$.

  - The edge $x \to y$ has been relaxed, so $y.d \leq x.d + w(x,y)$.
  - P is a shortest path => its subpath $(s, \ldots, x, y)$ must also be a shortest path,
    => $x.d + w(x,y) = \delta(s,y)$.

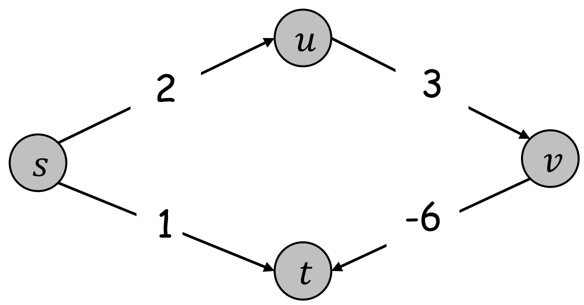
  - $\delta(s,y) \leq \delta(s,v)$, assuming nonnegative weights

=> $v.d > \delta(s,v) \geq \delta(s,y) = x.d + w(x,y) \geq y.d$,

contradicting fact that $v.d$ is the **smallest** in $V - S$.

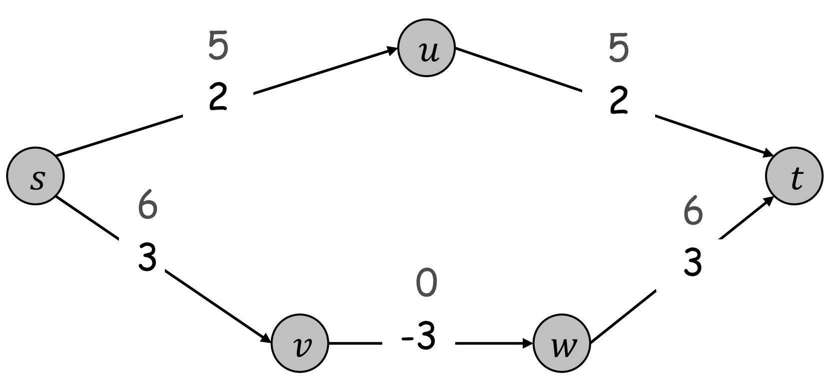
Dijkstra fails with Negative Weights

Example



Dijkstra would calculate $\delta(s,t) = 1$, but correct answer is $\delta(s,t) = -1$.

Re-weighting. Might think that this can be "fixed" by adding a constant to every edge weight. This doesn't work.



Add 3 to every weight. Dijkstra would find shortest s–t path is (s, u, v), but shortest s–t path in original graph is (s, v, w, t).

Exercise on Most Reliable Paths

Consider a directed graph corresponding to a communication network. Each edge (u, v) is associated with a reliability value $r(u, v)$, that represents the probability that the channel from from u to v will not fail. Assume that the edge probabilities are independent. Modify Dijkstra's algorithm to find the most reliable path between a node *s* and every other vertex.

Solution

Set $d[s] = 1$, and $d[u] = 0$ for all $u \neq s$

Insert all vertices in a max heap Q on $d[\cdot]$

While Q is not empty

  $u \coloneqq$ Extract-max(Q)

  For each edge (u, v) // v is in the adjacency list of u

    If $d[u] \cdot r(u, v) > d[v]$ // relax (u, v)

      $d[v] \coloneqq d[u] \cdot r(u, v)$

      Increase-key$(Q, v, d[v])$

      Set u to be the predecessor of v

A^* for s—t shortest path

We wish to find the shortest path between s and t.

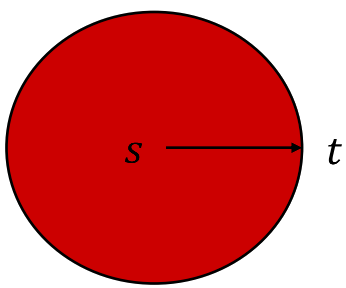
Assume that the weight of each edge (u, v) corresponds to the length of the road connecting them. Then, $\delta(u, t)$ between any node u and t, is their network distance. Let $E(u, t)$ be the Euclidean distance between u and t. Then, $E(u, t) \leq \delta(u, t)$.

When Dijkstra visits a node u, it inserts in the min heap $d[u]$, i.e., the current network distance from s. It extracts from the min heap the node u with min $d[u]$.

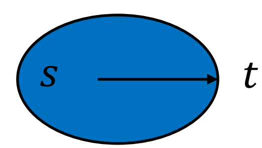
When A^*-search visits a node u, it inserts into the min heap $d[u] + E(u, t)$. It extracts from the heap the node u that minimizes $d[u] + E(u, t)$, i.e., guides search towards the destination. It terminates when we reach t.

A^* can be used with any function f provided that $f(u, t) \leq \delta(u, t)$. Faster than Dijkstra in practice, but asymptotically the same.

Dijkstra visited nodes



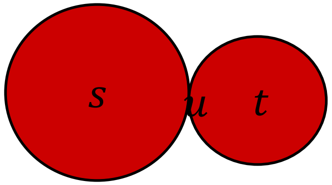
A^*-visited nodes



Other fast algorithms s–t shortest path

Bidirectional: start Dijkstra expansions from both s and t in parallel. When you find a common node u in both expansions, stop. The shortest path has distance: $\delta(s,u) + \delta(t,u)$.

Can also be combined with A^*.



Continuous monitoring of shortest path: the previous algorithms return a one-time path, assuming fixed edge weights. Real navigation systems monitor the traffic conditions and continuously update your path when traffic conditions change (e.g., accidents).

Many later algorithms for s–t paths (based on contraction hierarchies, partial materialization, landmarks etc) are much faster than Dijkstra in practice.

All-Pairs Shortest Paths

Input:

- Directed graph $G = (V, E)$.
- Weight $w(e) =$ length of edge e.

Output:

- $\delta(u, v)$, for all pairs of nodes u, v.
- A data structure from which the shortest path from u to v can be extracted efficiently, for any pair of nodes u, v
  - Note: Storing *all* shortest paths explicitly for all pairs requires $O(V^3)$ space.

Graph representation

- Assume adjacency matrix
  - $w(u, v)$ can be extracted in $O(1)$ time.
  - $w(u, u) = 0$, $w(u, v) = \infty$ if there is no edge from u to v.
- If the graph is stored in adjacency lists format, can convert to adjacency matrix in $O(V^2)$ time.

Using previous algorithms

When there are no negative cost edges

- Apply Dijkstra's algorithm to each vertex (as the source).
- Recall that Dijkstra algorithm runs in $O(E \log V)$
- This gives an $O(VE \log V)$-time algorithm
- If the graph is dense, this is $O(n^3 \log n)$.

When negative-weight edges are present

- The Bellman-Ford algorithm permits negative edges and solves the single-source shortest path problem in $O(VE)$ time
  - Run the B-F algorithm from each vertex.
- $O(V^2 E)$ time, which is $O(n^4)$ for dense graphs.

Dynamic Programming: Solution 1

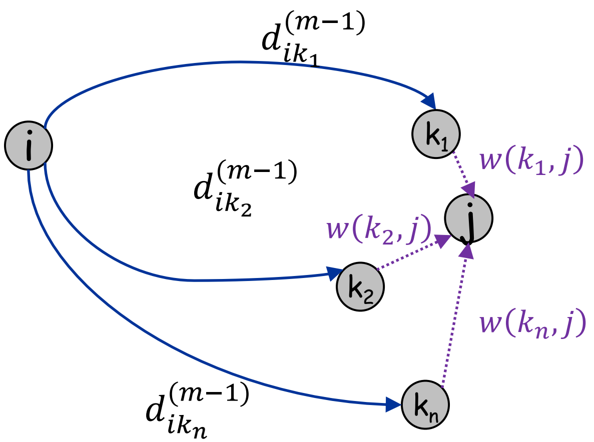
Def: $d_{ij}^{(m)}$ = length of the shortest path from i to j that contains at most m edges.

- Use $D^{(m)}$ to denote the matrix $\left[d_{ij}^{(m)}\right]$.

Recurrence:

For some k, let P' be the shortest path from i to k containing at most $m-1$ edges.
$length(P') = d_{ik}^{(m-1)}$

Then P' followed by j is a path from i to j containing at most m edges and has length $d_{ik}^{(m-1)} + w(k,j)$



$$d_{ij}^{(m)} = \min_{1\le k\le n}\{d_{ik}^{(m-1)} + w(k,j)\}$$

$$d_{ij}^{(1)} = w(i,j)$$

Solution 1: Algorithm

Def: $d_{ij}^{(m)}$ = length of the shortest path from i to j that contains at most m edges.

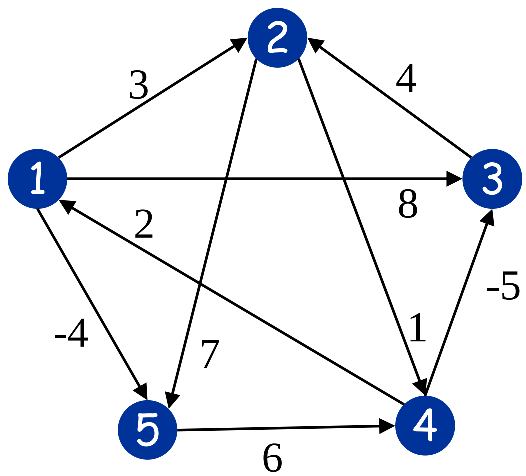
- Use $D^{(m)}$ to denote the matrix $\left[d_{ij}^{(m)}\right]$.
- Recurrence: $$d_{ij}^{(m)} = \min_{1 \le k \le n} \{d_{ik}^{(m-1)} + w(k,j)\}$$ $$d_{ij}^{(1)} = w(i,j)$$

Goal: $D^{(n-1)}$, since no shortest path can have more than $n-1$ edges

```
Slow-All-Pairs-Shortest-Paths(G):
d_ij^(1) = w(i,j) for all 1 ≤ i,j ≤ n
for m ← 2 to n − 1
    let D^(m) be a new n × n matrix
    for i ← 1 to n
        for j ← 1 to n
            d_ij^(m) ← ∞
            for k ← 1 to n
                if d_ik^(m−1) + w(k,j) < d_ij^(m) then d_ij^(m) ← d_ik^(m−1) + w(k,j)
return D^(n−1)
```

Analysis: $O(n^4)$ time, $O(n^3)$ space, can be improved to $O(n^2)$

Example of Solution 1



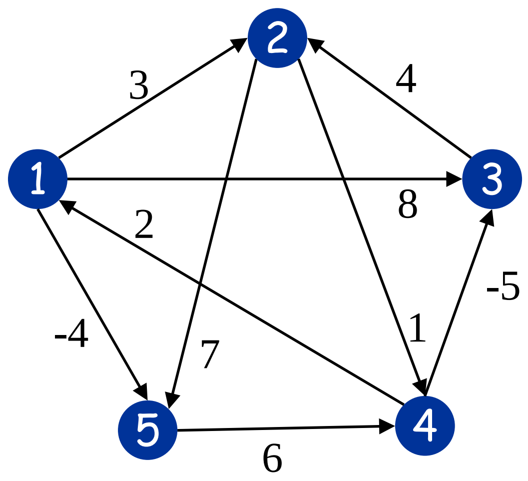
- Algorithm starts with $D^{(1)}$, initial edge lengths
- It then iteratively constructs $D^{(2)}$, $D^{(3)}$, $D^{(4)}$
- $D^{(4)}$ is the final solution, containing all shortest path lengths.

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

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

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

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

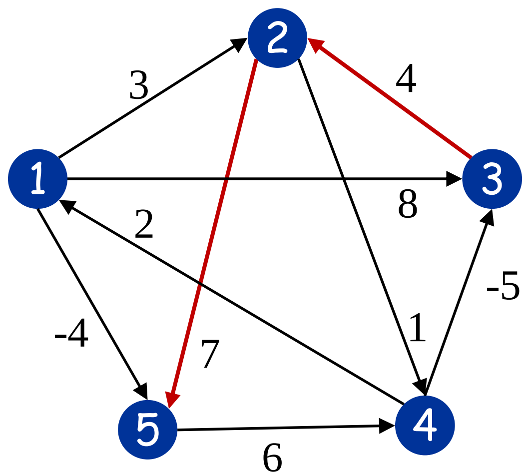


A Deeper Dive

Consider shortest path from 3 -> 5

$d^{(1)}(3,5) = \infty$

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



A Deeper Dive

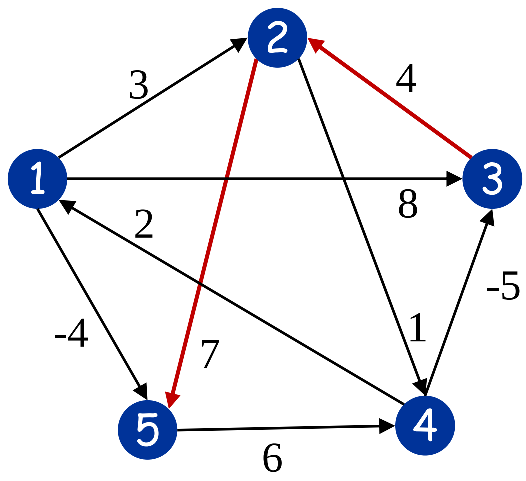
Consider shortest path from 3 -> 5

$$d^{(1)}(3,5) = \infty$$

$$d^{(2)}(3,5) = 11$$

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

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



A Deeper Dive

Consider shortest path from 3 -> 5

$d^{(1)}(3,5) = \infty$

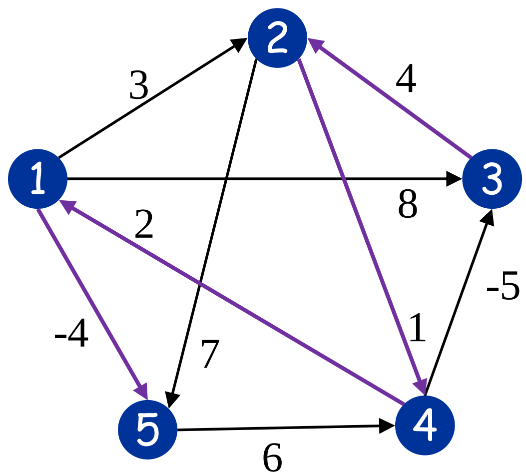
$d^{(2)}(3,5) = 11$

$d^{(3)}(3,5) = 11$

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

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

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



A Deeper Dive

Consider shortest path from 3 -> 5

$d^{(1)}(3{,}5) = \infty$ $\quad$ $d^{(3)}(3{,}5) = 11$

$d^{(2)}(3{,}5) = 11$ $\quad$ $d^{(4)}(3{,}5) = 3$

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

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

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

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

Dynamic Programming: Solution 2

Observation:

- To compute $d_{ij}^{(m)}$, instead of looking at the last stop before j, we look at the middle point.
- This cuts down the problem size by half.

New recurrence:

$$d_{ij}^{(2s)} = \min_{1 \le k \le n} \{d_{ik}^{(s)} + d_{kj}^{(s)}\}$$

Algorithm:

- Calculate $D^{(1)}, D^{(2)}, D^{(4)}, D^{(8)}, \ldots$
- Calculating each matrix takes $O(n^3)$ time: total time $= O(n^3 \log n)$.

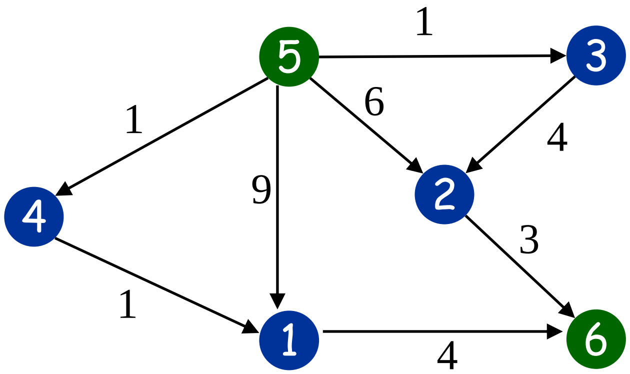
Q: This might overshoot $D^{(n-1)}$. Is algorithm still correct?

A: It's OK. $D^{(n')}$, $n' > n-1$, contains length of shortest paths with at most n' edges; it will not miss any shortest path with up to $n-1$ edges.

- Actually, $D^{(n')} = D^{(n-1)}$ for any $n' > n-1$, since no shortest path has more than $n-1$ edges.

Solution 3: Floyd-Warshall

Def: $d_{ij}^{(k)}$ = length of the shortest path from i to j that such that all intermediate vertices on the path (if any) are in the set $\{1,2,\dots,k\}$.



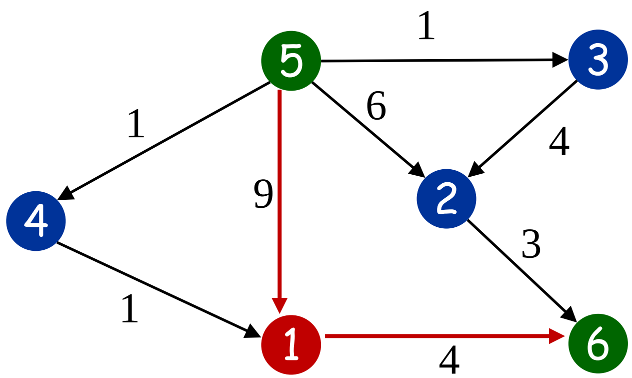
$d_{5,6}^{(0)} = \infty$ No Path

Initially: $d_{ij}^{(0)} = w(i,j)$

Goal: $D^{(n)}$

Solution 3: Floyd-Warshall

Def: $d_{ij}^{(k)}$ = length of the shortest path from i to j that such that all intermediate vertices on the path (if any) are in the set $\{1,2,\ldots,k\}$.



$d_{5,6}^{(0)} = \infty$ No Path

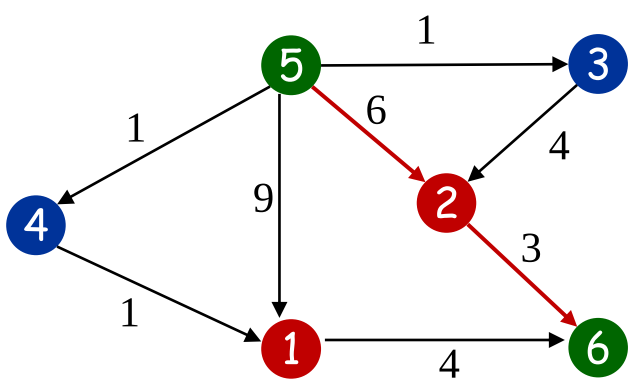
$d_{5,6}^{(1)} = 13$ (5 1 6)

Initially: $d_{ij}^{(0)} = w(i,j)$

Goal: $D^{(n)}$

Solution 3: Floyd-Warshall

Def: $d_{ij}^{(k)}$ = length of the shortest path from i to j that such that all intermediate vertices on the path (if any) are in the set $\{1,2,\ldots,k\}$.



$d_{5,6}^{(0)} = \infty$ No Path

$d_{5,6}^{(1)} = 13$ (5 1 6)

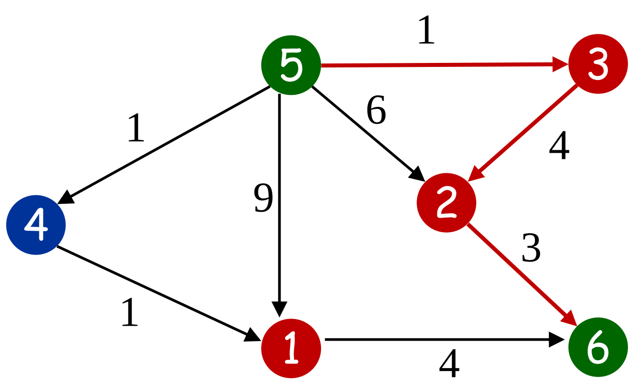
$d_{5,6}^{(2)} = 9$ (5 2 6)

Initially: $d_{ij}^{(0)} = w(i,j)$

Goal: $D^{(n)}$

Solution 3: Floyd-Warshall

Def: $d_{ij}^{(k)}$ = length of the shortest path from i to j that such that all intermediate vertices on the path (if any) are in the set $\{1,2,\dots,k\}$.



$d_{5,6}^{(0)} = \infty$ No Path

$d_{5,6}^{(1)} = 13$ (5 1 6)

$d_{5,6}^{(2)} = 9$ (5 2 6)

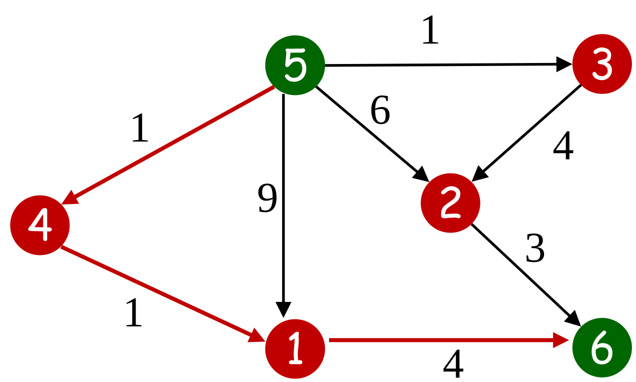
$d_{5,6}^{(3)} = 8$ (5 3 2 6)

Initially: $d_{ij}^{(0)} = w(i,j)$

Goal: $D^{(n)}$

Solution 3: Floyd-Warshall

Def: $d_{ij}^{(k)}$ = length of the shortest path from i to j that such that all intermediate vertices on the path (if any) are in the set $\{1,2,\ldots,k\}$.



$d_{5,6}^{(0)} = \infty$ No Path

$d_{5,6}^{(1)} = 13$ (5 1 6)

$d_{5,6}^{(2)} = 9$ (5 2 6)

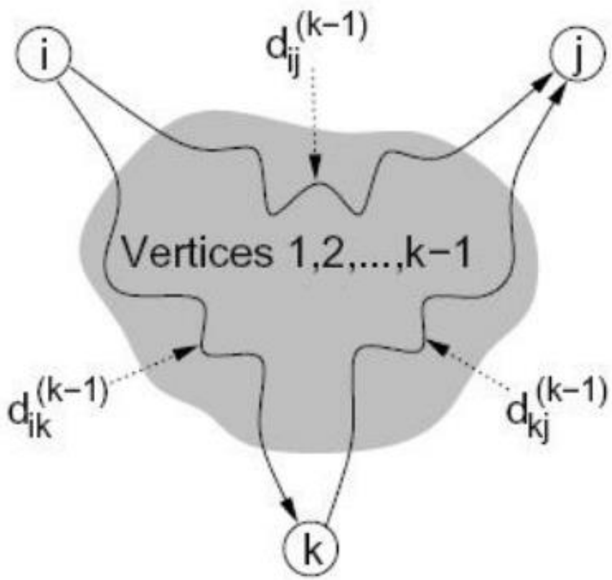
$d_{5,6}^{(3)} = 8$ (5 3 2 6)

$d_{5,6}^{(4)} = 6$ (5 4 1 6)

Initially: $d_{ij}^{(0)} = w(i,j)$

Goal: $D^{(n)}$

Recurrence



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

When computing $d_{ij}^{(k)}$, there are two cases:

- Case 1: k is not a vertex on the shortest path from i to j
  => then the path uses only vertices in $\{1,2,\dots,k-1\}$. $d_{ij}^{(k-1)}$
- Case 2: k is an intermediate node on the shortest path from i to j,
  => path can be split into shortest subpath from i to k and a subpath from k to j.
  Both subpaths use only vertices in $\{1,2,\dots,k-1\}$ $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$

Floyd-Warshall Algorithm

```
Floyd-Warshall(G):
d_ij^(0) = w(i,j) for all 1 ≤ i,j ≤ n
for k ← 1 to n
    let D^(k) be a new n × n matrix
    for i ← 1 to n
        for j ← 1 to n
            if d_ik^(k-1) + d_kj^(k-1) < d_ij^(k-1) then
                d_ij^(k) ← d_ik^(k-1) + d_kj^(k-1)
            else
                d_ij^(k) ← d_ij^(k-1)
return D^(n)
```

Analysis:

- $O(n^3)$ time
- $O(n^3)$ space, but can be improved to $O(n^2)$

Surprising discovery: If we just drop all the superscripts, i.e., the algorithm just uses one $n \times n$ array D, the algorithm still works! (why?)

Floyd-Warshall Algorithm: Final Version

Floyd-Warshall(G):
$d_{ij} = w(i,j)$ **and** $intermed[i,j] \leftarrow 0$ **for all** $1 \le i,j \le n$
for $k \leftarrow 1$ **to** n
    for $i \leftarrow 1$ **to** n
        for $j \leftarrow 1$ **to** n
            if $d_{ik} + d_{kj} < d_{ij}$ **then**
                $d_{ij} \leftarrow d_{ik} + d_{kj}$
                $intermed[i,j] \leftarrow k$
return D

Analysis:

- $O(n^3)$ time
- $O(n^2)$ space

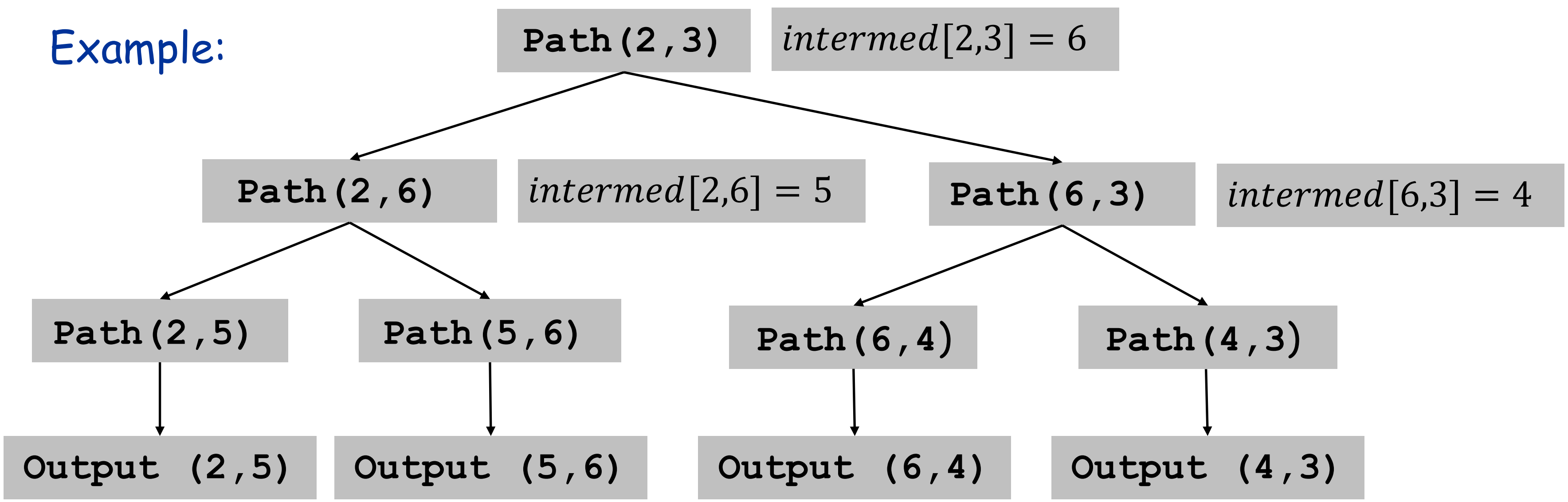
The $intermed[i,j]$ array records one intermediate node on the shortest path from i to j.

- It is nil if the shortest path does not pass any intermediate nodes.

Extracting Shortest Paths

```
Path(i,j):
if intermed[i,j] = nil then
    output (i,j)
else
    Path(i,intermed[i,j])
    Path(intermed[i,j],j)
```

Example:



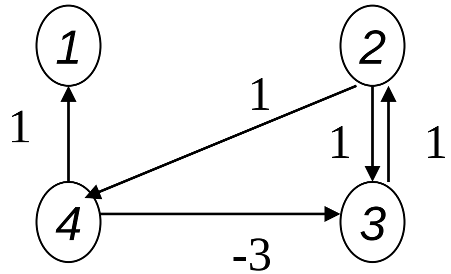
Running time: O(length of the shortest path)

Exercise on Detection of Negative Cycles

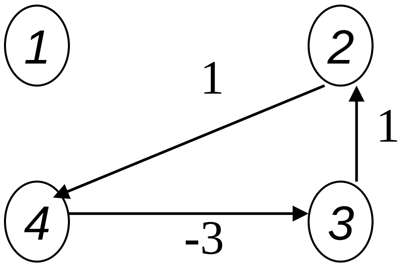
Given a directed weighted graph $G = (V, E)$, use Floyd-Warshall in order to find if a graph has negative cycles

- Assume that $w(i, i) = 0$, for each vertex i

graph G



Negative cycle



Solution for Negative Cycles

- Let's consider the smallest negative cycle C (i.e., the one involving the smallest number of vertices).
- Let k be the highest-numbered vertex in C, and let i be any other vertex in C.
- Then, $d_{i,i}^{(k)} = min\left\{d_{i,i}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,i}^{(k-1)}\right\} < 0$
- Therefore, as soon as we see $d_{i,i}^{(k)} < 0$ for any i, we conclude that there is a negative cycle and abort the algorithm.

Exercise on Transitive Closure

Given a directed unweighted graph $G = (V, E)$, we want to generate $G^* = (V, E^*)$, where $E^* = \{(i, j)\colon \text{ there is a path from } i \text{ to } j \text{ in } G\}$

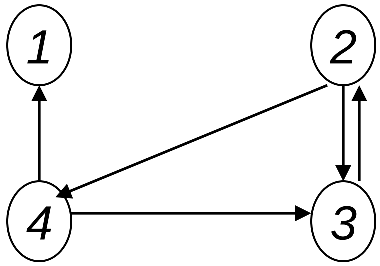
Input: an adjacency matrix A of G:

- $a(i,j) = 1$ if there is an edge from vertex i to j in G
- $a(i,j) = 0$ if there is no edge from vertex i to j in G

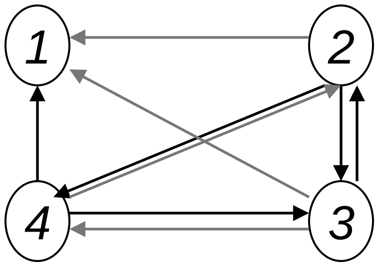
Output: an adjacency matrix A^* of G^*:

- $a^*(i,j) = 1$ if there is a path from vertex i to j in G
- $a^*(i,j) = 0$, otherwise

graph G



transitive closure G^*



Solution 1 on Transitive Closure

We first derive the weight matrix as follows

- $w(i,j) = 1$, if $a(i,j) = 1$
- $w(i,j) = \infty$, if $a(i,j) = 0$

Apply Floyd-Warshall and obtain shortest distance matrix $D^{(n)}$

- $d_{i,j}^{(n)}$ is the length of the shortest path from vertex i to j in G, in terms of the number of edges.

If $d_{i,j}^{(n)} < \infty$, set $a^*(i,j) = 1$

If $d_{i,j}^{(n)} = \infty$, set $a^*(i,j) = 0$

Solution 2 on Transitive Closure

Based on Boolean Operators

Define boolean matrix $T^{(0)} = A$

- $T_{i,j}^{(0)} = a(i,j)$

Optimal substructure

- $T_{i,j}^{(k)} = T_{i,j}^{(k-1)} \vee \left(T_{i,k}^{(k-1)} \wedge T_{k,j}^{(k-1)}\right)$
- $A^* = T^{(n)}$

Asymptotic complexity same as that of Floyd-Warshall $\Theta(n^3)$.

- However, more efficient in practice because boolean operators are faster than arithmetic operators
- Needs less space for the boolean matrix (instead of the distance matrix)