Graphs - Shortest Paths

Fundamental Algorithms

Rodica Potolea, Camelia Lemnaru and Ciprian Oprișa

Technical University of Cluj-Napoca
Computer Science Department

Agenda

1. Single Source Shortest Paths
   - Dijkstra's Algorithm
   - The Bellman-Ford Algorithm
   - SSSP in DAGs
2. All-Pairs Shortest Paths
   - The Floyd-Warshall Algorithm
   - Transitive closure

Agenda

1. Single Source Shortest Paths
   - Dijkstra's Algorithm
   - The Bellman-Ford Algorithm
   - SSSP in DAGs
2. All-Pairs Shortest Paths
   - The Floyd-Warshall Algorithm
   - Transitive closure

Shortest Paths problem definition

- we are given a weighted graph $G = (V, E)$, $w : E \to \mathbb{R}$
- a **path** p is a sequence of vertices connected by edges $p =< v_0, v_1, \ldots v_k >$
- the **weight of a path** $w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$
- the **shortest-path weight** from u to v

$$\delta(u, v) = \begin{cases} \min\{w(p)\} & \text{if } \exists p =< u, \ldots, v > \\ \infty & \text{otherwise} \end{cases}$$

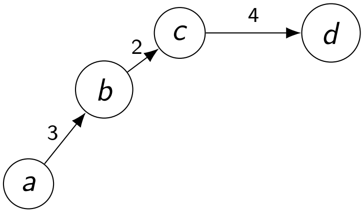
- a **shortest path** from u to v is any path p with $w(p) = \delta(u, v)$

Optimal substructure of a shortest path

Lemma

Given a weighted graph $G = (V, E)$ with $w : E \to \mathbb{R}$, let $p =< v_0, v_1, \dots v_k >$ be a shortest path from v_0 to v_k. For any i, j, $0 \le i \le j \le k$, the subpath $p_{ij} =< v_i, v_{i+1}, \dots v_j >$ is a shortest path from v_i to v_j.

- if a path is a shortest path, all subpaths are shortest paths

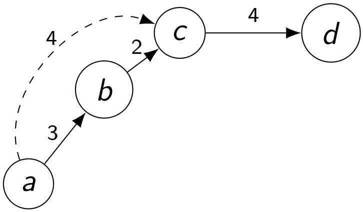


Optimal substructure of a shortest path

Lemma

Given a weighted graph $G = (V, E)$ with $w : E \to \mathbb{R}$, let $p =< v_0, v_1, \ldots v_k >$ be a shortest path from v_0 to v_k.
For any i, j, $0 \le i \le j \le k$, the subpath $p_{ij} =< v_i, v_{i+1}, \ldots v_j >$ is a shortest path from v_i to v_j.

- if a path is a shortest path, all subpaths are shortest paths



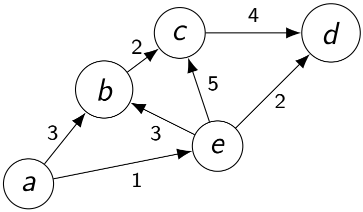
If a shorter subpath existed, the path $v_0 \rightsquigarrow v_k$ would use it.

Optimal substructure of a shortest path

Lemma

Given a weighted graph $G = (V, E)$ with $w : E \to \mathbb{R}$, let $p =< v_0, v_1, \dots v_k >$ be a shortest path from v_0 to v_k. For any i, j, $0 \le i \le j \le k$, the subpath $p_{ij} =< v_i, v_{i+1}, \dots v_j >$ is a shortest path from v_i to v_j.

- if a path is a shortest path, all subpaths are shortest paths
- the reverse is not true



$< a, b >$, $< b, c >$, $< c, d >$, $< a, b, c >$ and $< b, c, d >$ are all shortest paths, but $< a, b, c, d >$ is not.

Representing shortest paths

Add attributes d and π for each node v.

- $v.d$ - weight of the shortest path from source s to v
- $v.\pi$ - the parent/predecessors in the shortest path from s to v

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex $v \in G.V$

    $v.d = \infty$

    $v.\pi = \text{NIL}$

$s.d = 0$

Relaxation technique

- try to improve the shortest path found so far
- can we find a shorter path to v by passing through u?

RELAX(u, v, w)

```
1  if v.d > u.d + w(u, v)
2       v.d = u.d + w(u, v)
3       v.π = u
```

Relaxation technique

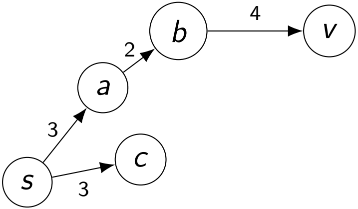
- try to improve the shortest path found so far
- can we find a shorter path to v by passing through u?

$\text{RELAX}(u, v, w)$

if $v.d > u.d + w(u, v)$

$\quad v.d = u.d + w(u, v)$

$\quad v.\pi = u$



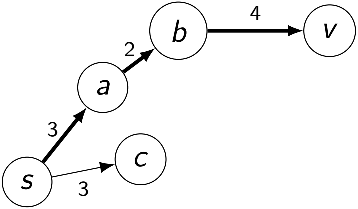
$v.d = ?$

Relaxation technique

- try to improve the shortest path found so far
- can we find a shorter path to v by passing through u?

$\textsc{Relax}(u, v, w)$

```
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```



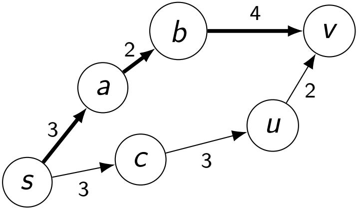
$v.d = 9$

Relaxation technique

- try to improve the shortest path found so far
- can we find a shorter path to v by passing through u?

$\text{RELAX}(u, v, w)$

$$
\begin{array}{ll}
1 & \textbf{if } v.d > u.d + w(u,v) \\
2 & \qquad v.d = u.d + w(u,v) \\
3 & \qquad v.\pi = u
\end{array}
$$



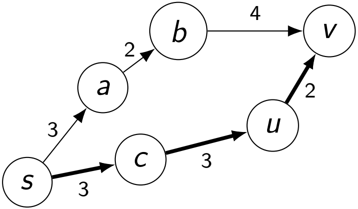
$v.d = 9 \qquad u.d = 6$

Relaxation technique

- try to improve the shortest path found so far
- can we find a shorter path to v by passing through u?

$\textsc{Relax}(u, v, w)$

```
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```



$\cancel{v.d = 9}$ $\quad u.d = 6$

$v.d = 6 + 2 = 8$

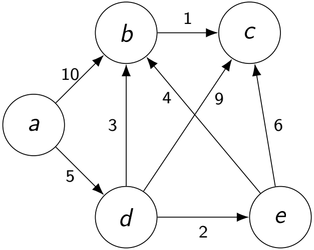
Dijkstra's Algorithm

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

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

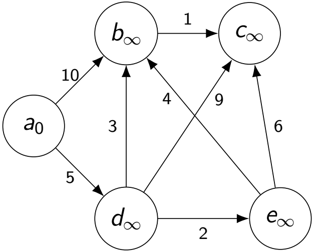
INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX(u, v, w)



Dijkstra's Algorithm

DIJKSTRA(G, w, s)

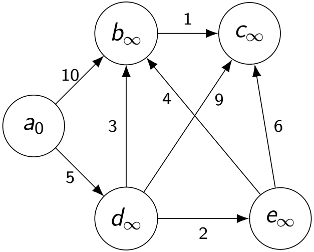
```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



	π	d
a	NIL	0
b	NIL	∞
c	NIL	∞
d	NIL	∞
e	NIL	∞

Dijkstra's Algorithm

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

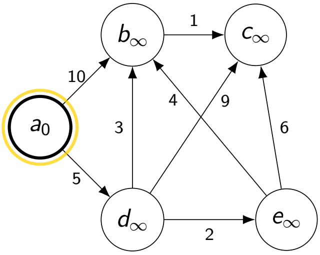


	π	d
a	NIL	0
b	NIL	∞
c	NIL	∞
d	NIL	∞
e	NIL	∞

Q

Dijkstra's Algorithm

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

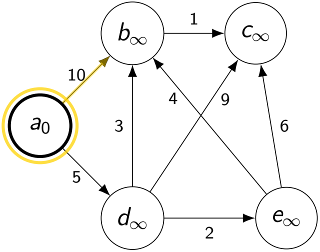


	π	d	
a	NIL	0	S
b	NIL	∞	Q
c	NIL	∞	
d	NIL	∞	
e	NIL	∞	

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

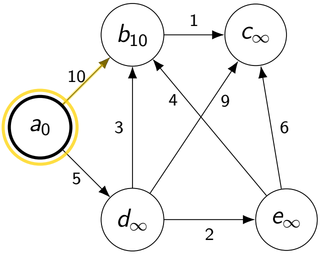
```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

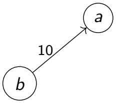


	π	d	
a	NIL	0	S
b	NIL	∞	Q
c	NIL	∞	
d	NIL	∞	
e	NIL	∞	

Dijkstra's Algorithm

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



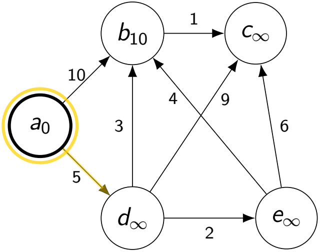


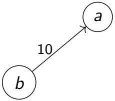
	π	d	
a	NIL	0	S
b	a	10	Q
c	NIL	∞	Q
d	NIL	∞	Q
e	NIL	∞	Q

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

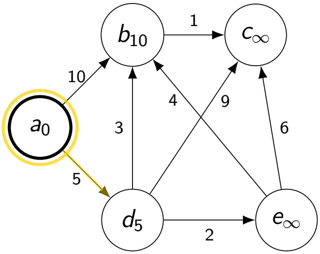


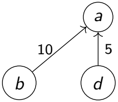


	π	d	
a	NIL	0	S
b	a	10	
c	NIL	∞	Q
d	NIL	∞	
e	NIL	∞	

Dijkstra's Algorithm

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

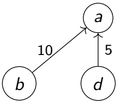


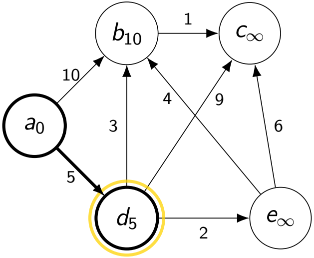


	π	d	
a	NIL	0	S
d	a	5	Q
b	a	10	Q
c	NIL	∞	Q
e	NIL	∞	Q

Dijkstra's Algorithm

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

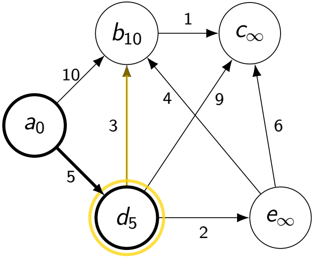


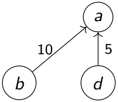


	π	d	
a	NIL	0	S
d	a	5	
b	a	10	Q
c	NIL	∞	
e	NIL	∞	

Dijkstra's Algorithm

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

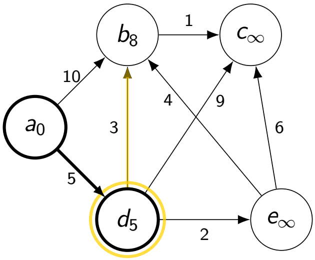


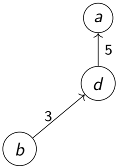


	π	d	
a	NIL	0	S
d	a	5	
b	a	10	Q
c	NIL	∞	
e	NIL	∞	

Dijkstra's Algorithm

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



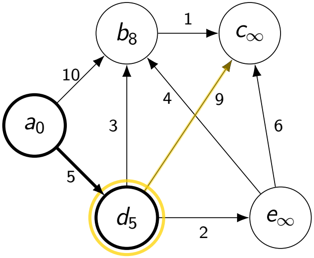


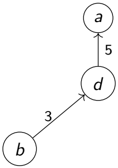
	π	d	
a	NIL	0	S
d	a	5	S
b	d	8	Q
c	NIL	∞	Q
e	NIL	∞	Q

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



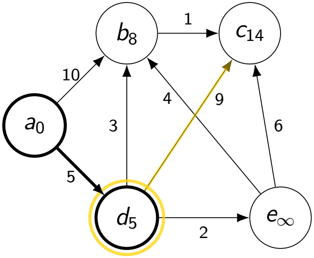


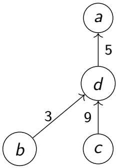
	π	d	
a	NIL	0	S
d	a	5	
b	d	8	Q
c	NIL	∞	
e	NIL	∞	

Dijkstra's Algorithm

$\textsc{Dijkstra}(G, w, s)$

```
Initialize-Single-Source(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = Extract-Min(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        Relax(u, v, w)
```



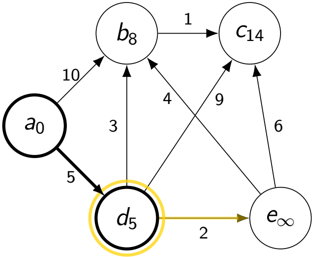


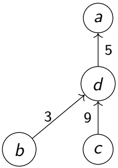
	π	d	
a	NIL	0	S
d	a	5	
b	d	8	
c	d	13	Q
e	NIL	∞	

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



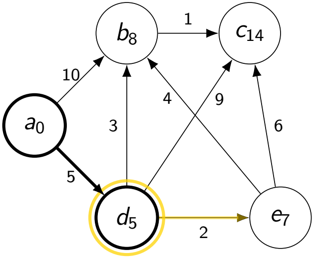


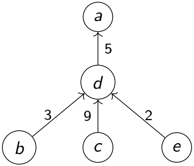
	π	d	
a	NIL	0	S
d	a	5	
b	d	8	Q
c	d	13	
e	NIL	∞	

Dijkstra's Algorithm

$\textsc{Dijkstra}(G, w, s)$

$\quad \textsc{Initialize-Single-Source}(G, s)$
$\quad S = \emptyset$
$\quad Q = G.V$
$\quad$ **while** $Q \neq \emptyset$
$\qquad u = \textsc{Extract-Min}(Q)$
$\qquad S = S \cup \{u\}$
$\qquad$ **for** each vertex $v \in G.Adj[u]$
$\qquad\quad \textsc{Relax}(u, v, w)$



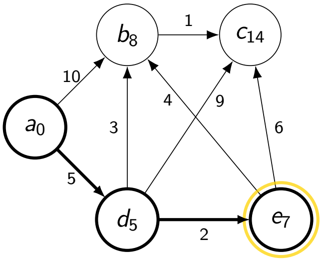


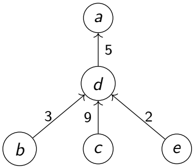
	π	d	
a	NIL	0	S
d	a	5	
e	d	7	Q
b	d	8	
c	d	14	

Dijkstra's Algorithm

$\text{DIJKSTRA}(G, w, s)$

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

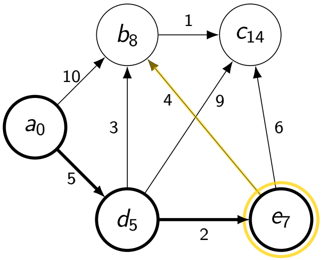


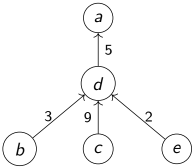


	π	d	
a	NIL	0	S
d	a	5	
e	d	7	
b	d	8	Q
c	d	14	

Dijkstra's Algorithm

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



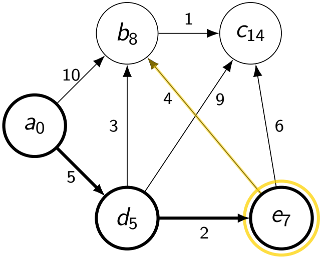


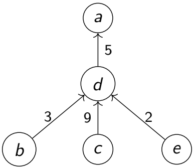
	π	d	
a	NIL	0	S
d	a	5	S
e	d	7	S
b	d	8	Q
c	d	14	Q

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



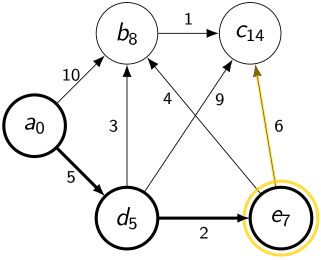


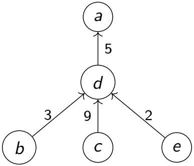
	π	d	
a	NIL	0	S
d	a	5	S
e	d	7	S
b	d	8	Q
c	d	14	Q

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



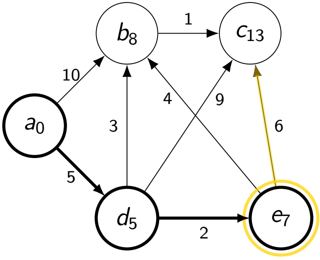


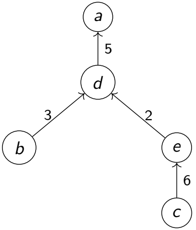
	π	d	
a	NIL	0	
d	a	5	S
e	d	7	
b	d	8	Q
c	d	14	

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX(u, v, w)



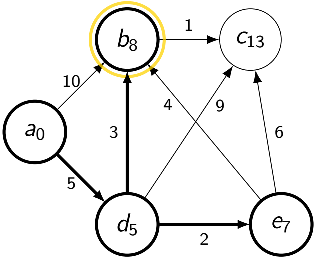


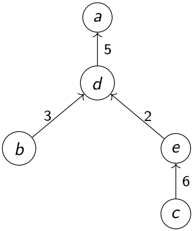
	π	d	
a	NIL	0	
d	a	5	S
e	d	7	
b	d	8	
c	e	13	Q

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



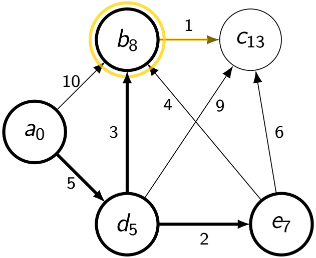


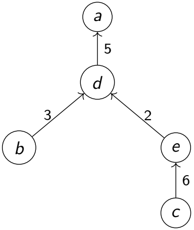
	π	d	
a	NIL	0	
d	a	5	S
e	d	7	
b	d	8	
c	e	13	Q

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

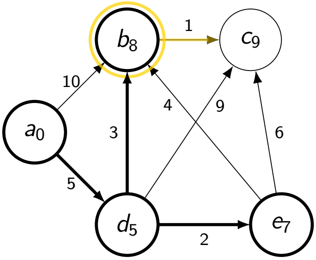


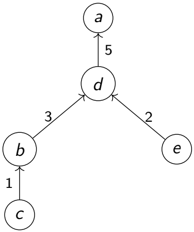


	π	d	
a	NIL	0	
d	a	5	S
e	d	7	
b	d	8	
c	e	13	Q

Dijkstra's Algorithm

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

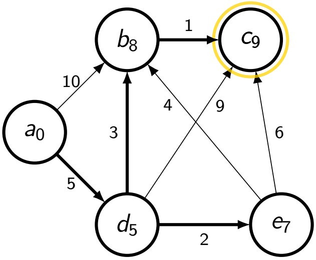


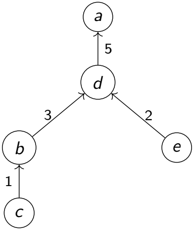


	π	d	
a	NIL	0	S
d	a	5	S
e	d	7	S
b	d	8	S
c	b	9	Q

Dijkstra's Algorithm

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





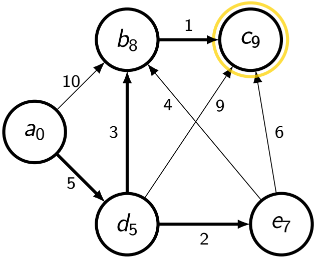
	π	d
a	NIL	0
d	a	5
e	d	7
b	d	8
c	b	9

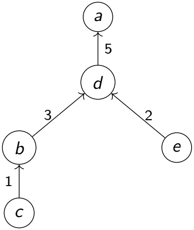
S

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
    $u =$ EXTRACT-MIN(Q)
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$
        RELAX(u, v, w)





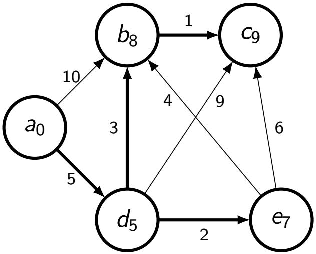
	π	d
a	NIL	0
d	a	5
e	d	7
b	d	8
c	b	9

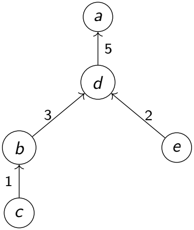
S

Dijkstra's Algorithm

DIJKSTRA(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```



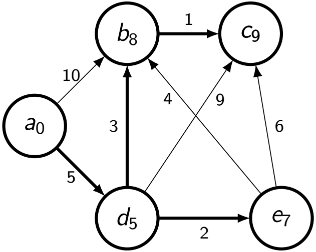


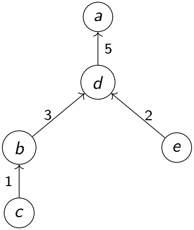
	π	d
a	NIL	0
d	a	5
e	d	7
b	d	8
c	b	9

S

Dijkstra's Algorithm

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





	π	d
a	NIL	0
d	a	5
e	d	7
b	d	8
c	b	9

S

Dijkstra's Algorithm - Complexity

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

Dijkstra's Algorithm - Complexity

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

regular array | binary heap | Fibonacci heap

Dijkstra's Algorithm - Complexity

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

$O(V)$

regular array

binary heap

Fibonacci heap

Dijkstra's Algorithm - Complexity

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

$O(V)$

$O(V)/O(V)/O(V)$

regular array   binary heap   Fibonacci heap

Dijkstra's Algorithm - Complexity

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
  $u =$ EXTRACT-MIN(Q)
  $S = S \cup \{u\}$
  for each vertex $v \in G.Adj[u]$
    RELAX(u, v, w)

$O(V)$

$O(V)/O(V)/O(V)$

$|V|$ steps

regular array   binary heap   Fibonacci heap

Dijkstra's Algorithm - Complexity

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

$O(V)$

$O(V) / O(V) / O(V)$

$|V|$ steps

$O(V) / O(\log V) / O(\log V)$

regular array   binary heap   Fibonacci heap

Dijkstra's Algorithm - Complexity

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s) — $O(V)$
$S = \emptyset$
$Q = G.V$ — $O(V)/O(V)/O(V)$
while $Q \neq \emptyset$ — $|V|$ steps
    $u =$ EXTRACT-MIN(Q) — $O(V)/O(\log V)/O(\log V)$
    $S = S \cup \{u\}$
    for each vertex $v \in G.Adj[u]$ — each neighbor for every vertex, so $|E|$ steps in total
        RELAX(u, v, w)

regular array | binary heap | Fibonacci heap

Dijkstra's Algorithm - Complexity

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

updating the distance → heap decrease key

$O(V)$

$O(V)/O(V)/O(V)$

$|V|$ steps

$O(V)/O(\log V)/O(\log V)$

each neighbor
for every vertex,
so $|E|$ steps in total

$O(1)/O(\log V)/O(1)$

regular array | binary heap | Fibonacci heap

Dijkstra's Algorithm - Complexity

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)
```

$O(V)$

$O(V)/O(V)/O(V)$

$|V|$ steps

$O(V)/O(\log V)/O(\log V)$

each neighbor for every vertex, so $|E|$ steps in total

$O(1)/O(\log V)/O(1)$

updating the distance → heap decrease key

regular array: $O(V \cdot V + E \cdot 1) = O(V^2)$

binary heap

Fibonacci heap

Dijkstra's Algorithm - Complexity

DIJKSTRA(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
  $u =$ EXTRACT-MIN(Q)
  $S = S \cup \{u\}$
  for each vertex $v \in G.Adj[u]$
    RELAX(u, v, w)

updating the distance $\rightarrow$ heap decrease key

$O(V)$

$O(V) / O(V) / O(V)$

$|V|$ steps

$O(V) / O(\log V) / O(\log V)$

each neighbor for every vertex, so $|E|$ steps in total

$O(1) / O(\log V) / O(1)$

regular array

$O(V \cdot V + E \cdot 1)$
$= O(V^2)$

binary heap

$O(V \log V + E \log V)$
$= O(E \log V)$

Fibonacci heap

Dijkstra's Algorithm - Complexity

$\text{DIJKSTRA}(G, w, s)$

$\text{INITIALIZE-SINGLE-SOURCE}(G, s)$
$S = \emptyset$
$Q = G.V$
while $Q \neq \emptyset$
  $u = \text{EXTRACT-MIN}(Q)$
  $S = S \cup \{u\}$
  for each vertex $v \in G.Adj[u]$
    $\text{RELAX}(u, v, w)$

updating the distance → heap decrease key

$O(V)$

$O(V)/O(V)/O(V)$

$|V|$ steps

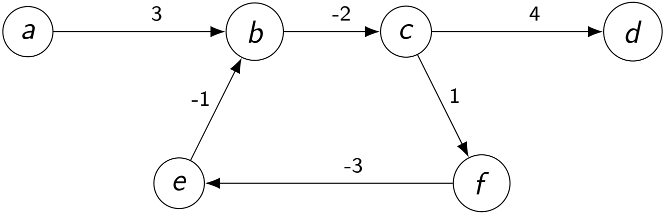
$O(V)/O(\log V)/O(\log V)$

each neighbor for every vertex, so $|E|$ steps in total

$O(1)/O(\log V)/O(1)$

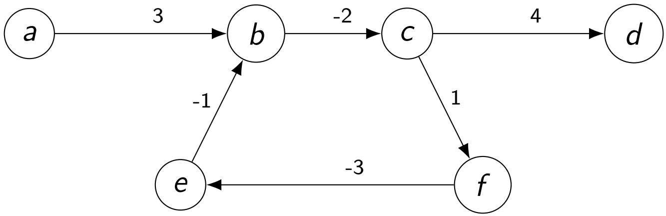
regular array	binary heap	Fibonacci heap
$O(V \cdot V + E \cdot 1) = O(V^2)$	$O(V \log V + E \log V) = O(E \log V)$	$O(V \log V + E)$

Negative-weight cycles



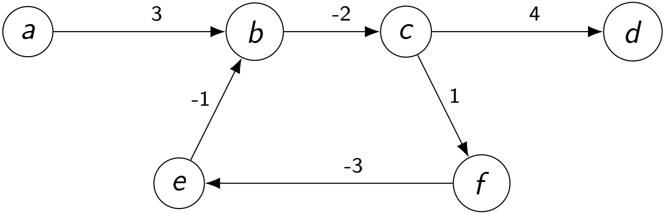
- what is the shortest path from a to d?

Negative-weight cycles



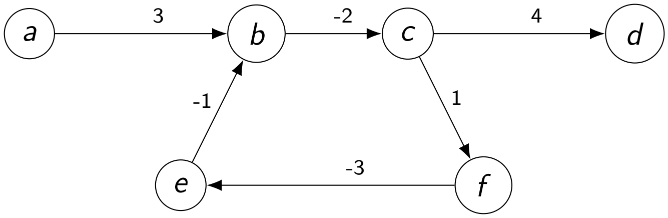
- what is the shortest path from a to d?
  - $w(< a, b, c, d >) = 5$

Negative-weight cycles



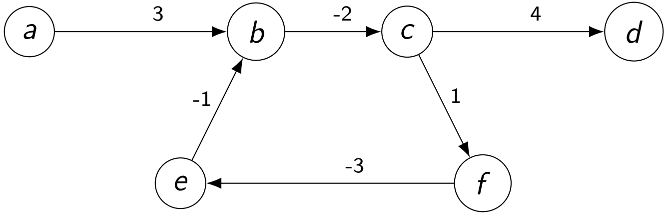
- what is the shortest path from a to d?
  - $w(< a, b, c, d >) = 5$
  - $w(< a, b, c, f, e, b, c, d >) = 2$

Negative-weight cycles



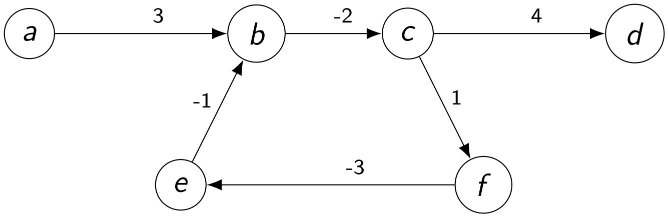
- what is the shortest path from a to d?
  - $w(< a, b, c, d >) = 5$
  - $w(< a, b, c, f, e, b, c, d >) = 2$
  - $w(< a, b, c, f, e, b, c, f, e, b, c, d >) = -1$

Negative-weight cycles



- what is the shortest path from a to d?
  - $w(< a, b, c, d >) = 5$
  - $w(< a, b, c, f, e, b, c, d >) = 2$
  - $w(< a, b, c, f, e, b, c, f, e, b, c, d >) = -1$
  - ...
  - $-\infty$

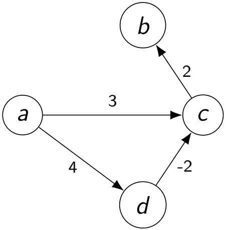
Negative-weight cycles



- what is the shortest path from a to d?
  - $w(< a, b, c, d >) = 5$
  - $w(< a, b, c, f, e, b, c, d >) = 2$
  - $w(< a, b, c, f, e, b, c, f, e, b, c, d >) = -1$
  - $\ldots$
  - $-\infty$
- the shortest path is not defined if we have negative-weight cycles

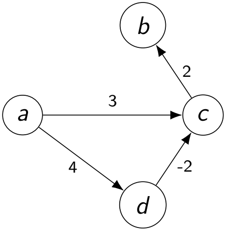
Negative-weight edges

- Dijkstra's algorithm doesn't work if we have negative edges

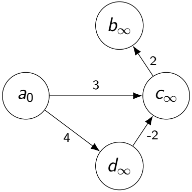


Negative-weight edges

- Dijkstra's algorithm doesn't work if we have negative edges

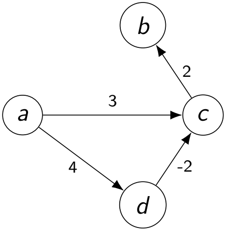


Dijkstra's solution

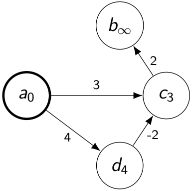


Negative-weight edges

- Dijkstra's algorithm doesn't work if we have negative edges

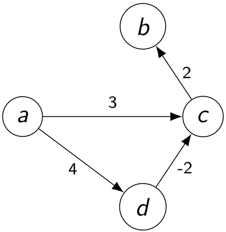


Dijkstra's solution

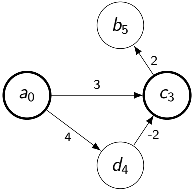


Negative-weight edges

- Dijkstra's algorithm doesn't work if we have negative edges

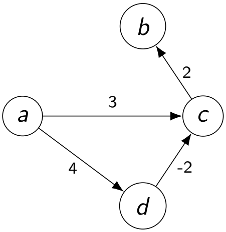


Dijkstra's solution

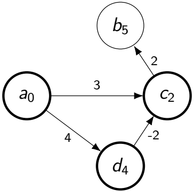


Negative-weight edges

- Dijkstra's algorithm doesn't work if we have negative edges

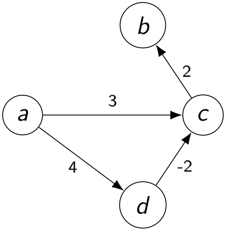


Dijkstra's solution

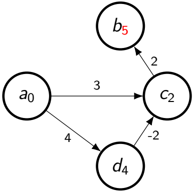


Negative-weight edges

- Dijkstra's algorithm doesn't work if we have negative edges

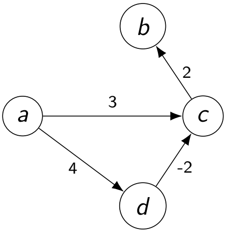


Dijkstra's solution

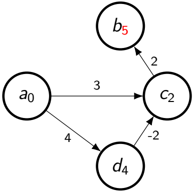


Negative-weight edges

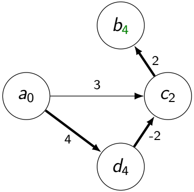
- Dijkstra's algorithm doesn't work if we have negative edges



Dijkstra's solution



Correct solution



Bellman-Ford - approach

- if we have negative edges, some edges need to be *"relaxed"* more than once

Bellman-Ford - approach

- if we have negative edges, some edges need to be *"relaxed"* more than once
- perform $|V| - 1$ steps
  - call $\textsc{Relax}$ on every edge

Bellman-Ford - approach

- if we have negative edges, some edges need to be *"relaxed"* more than once
- perform $|V| - 1$ steps
  - call Relax on every edge
- check for negative-weight cycles

Bellman-Ford - approach

- if we have negative edges, some edges need to be *"relaxed"* more than once
- perform $|V| - 1$ steps
  - call $\textsc{Relax}$ on every edge
- check for negative-weight cycles
  - if an edge can be *"relaxed"* even further it means that we have a negative-weight cycle

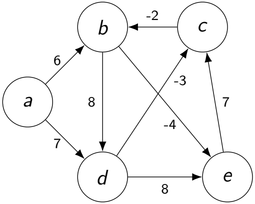
The Bellman-Ford Algorithm

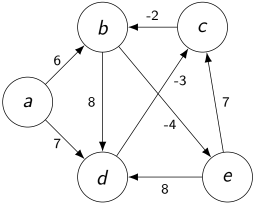
```
BELLMAN-FORD(G, w, s)
  INITIALIZE-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| - 1
      for each edge (u, v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
          return FALSE
  return TRUE
```

The Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
for $i = 1$ **to** $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)
for each edge $(u, v) \in G.E$
  if $v.d > u.d + w(u, v)$
    return FALSE
return TRUE

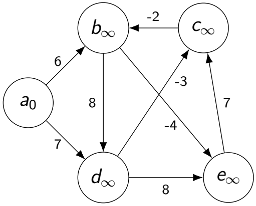


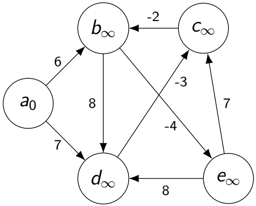


The Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

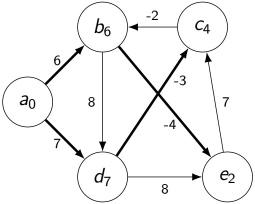




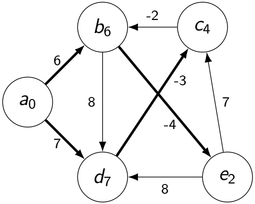
The Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
for $i = 1$ **to** $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX(u, v, w)
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE



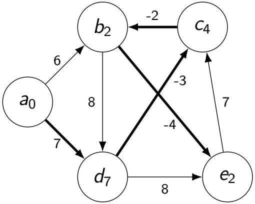
$i = 1$



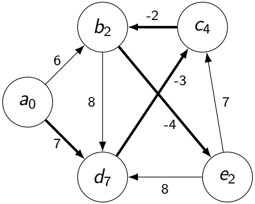
The Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

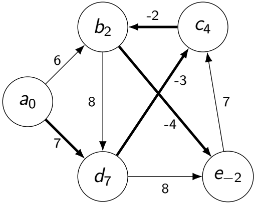


$i = 2$

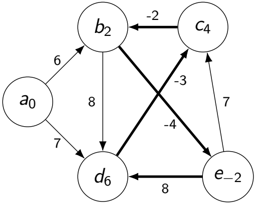


The Bellman-Ford Algorithm

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



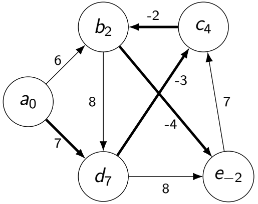
$i = 3$



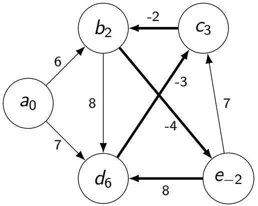
The Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1
    for each edge (u, v) ∈ G.E
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

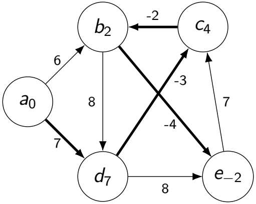


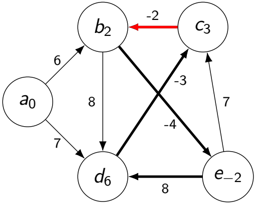
$i = 4$



The Bellman-Ford Algorithm

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





The Bellman-Ford Algorithm - Complexity

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

The Bellman-Ford Algorithm - Complexity

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

The Bellman-Ford Algorithm - Complexity

BELLMAN-FORD(G, w, s)

INITIALIZE-SINGLE-SOURCE(G, s)
for $i = 1$ **to** $|G.V| - 1$ → $O(V)$
    for each edge $(u, v) \in G.E$ → $\times O(E)$
        RELAX(u, v, w)
for each edge $(u, v) \in G.E$
    if $v.d > u.d + w(u, v)$
        return FALSE
return TRUE

The Bellman-Ford Algorithm - Complexity

BELLMAN-FORD(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| - 1                          O(V)
    for each edge (u, v) ∈ G.E                  × O(E)
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E          ⎫
    if v.d > u.d + w(u, v)          ⎬          O(E)
        return FALSE                ⎭
return TRUE
```

The Bellman-Ford Algorithm - Complexity

BELLMAN-FORD(G, w, s)

```
INITIALIZE-SINGLE-SOURCE(G, s)                  → O(V)
for i = 1 to |G.V| − 1                          → O(V)
    for each edge (u, v) ∈ G.E                  → ×O(E)
        RELAX(u, v, w)
for each edge (u, v) ∈ G.E       ⎫
    if v.d > u.d + w(u, v)       ⎬              → O(E)
        return FALSE             ⎭
return TRUE
```

Complexity: $O(V \cdot E)$

SSSP in DAGs - approach

- DAG = Directed Acyclic Graph
- no cycles $\rightarrow$ no negative-weight cycles
- the vertices can be topologically sorted
  - if $\exists u \rightsquigarrow v$, then u comes before v

SSSP in DAGs - approach

- DAG = Directed Acyclic Graph
- no cycles $\rightarrow$ no negative-weight cycles
- the vertices can be topologically sorted
  - if $\exists u \rightsquigarrow v$, then u comes before v
- a single pass over the vertices in topologically sorted order
  - relax each edge that leaves the vertex

SSSP in DAGs Algorithm

DAG-SHORTEST-PATHS(G, w, s)

```
topologically sort the vertices of G // see seminar
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

SSSP in DAGs Algorithm - Complexity

```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G // see seminar
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

SSSP in DAGs Algorithm - Complexity

```
DAG-SHORTEST-PATHS(G, w, s)
topologically sort the vertices of G // see seminar
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

$O(V + E)$

SSSP in DAGs Algorithm - Complexity

```
DAG-SHORTEST-PATHS(G, w, s)
    topologically sort the vertices of G // see seminar        → O(V + E)
    INITIALIZE-SINGLE-SOURCE(G, s)                              → O(V)
    for each vertex u, taken in topologically sorted order
        for each vertex v ∈ G.Adj[u]
            RELAX(u, v, w)
```

SSSP in DAGs Algorithm - Complexity

DAG-SHORTEST-PATHS(G, w, s)

```
topologically sort the vertices of G // see seminar          → O(V + E)
INITIALIZE-SINGLE-SOURCE(G, s)                                → O(V)
for each vertex u, taken in topologically sorted order        → |V| steps
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

SSSP in DAGs Algorithm - Complexity

DAG-SHORTEST-PATHS(G, w, s)

topologically sort the vertices of G // see seminar

INITIALIZE-SINGLE-SOURCE(G, s)

for each vertex u, taken in topologically sorted order

  for each vertex $v \in G.Adj[u]$

    RELAX(u, v, w)

$O(V + E)$

$O(V)$

$|V|$ steps

each neighbor
for every vertex,
so $|E|$ steps in total

SSSP in DAGs Algorithm - Complexity

DAG-SHORTEST-PATHS(G, w, s)

```
topologically sort the vertices of G // see seminar
INITIALIZE-SINGLE-SOURCE(G, s)
for each vertex u, taken in topologically sorted order
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

$O(V + E)$

$O(V)$

$|V|$ steps

each neighbor
for every vertex,
so $|E|$ steps in total

Complexity: $O(V + E + V + E) = O(V + E)$

Agenda

All-Pairs Shortest Paths

- compute the shortest path between every pair of vertices
  - the result is a matrix
- we assume that the vertices are numbered $1, 2, \ldots, n$
- the graph is given as an adjacency matrix with weights $W = (w_{ij})$

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

- simple approach: use SSSP on every node

Floyd-Warshall - Approach

- denote by $d_{ij}^{(k)}$ the weight of a shortest path from i to j for which all intermediary nodes are in the set $\{1, 2, \ldots, k\}$

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

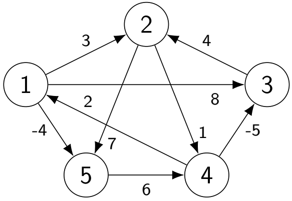
- dynamic programming
- $D^{(n)} = \left(d_{ij}^{(n)}\right)$ gives the final answer - $d_{ij}^{(n)} = \delta(i, j)$

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

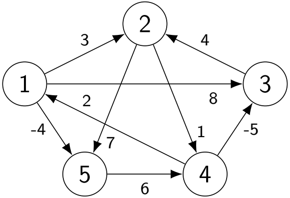
The Floyd-Warshall Algorithm



FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

The Floyd-Warshall Algorithm



FLOYD-WARSHALL(W)

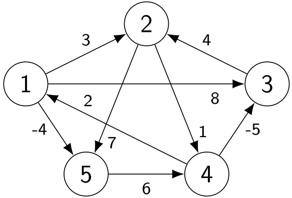
```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

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

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```



$k = 1$

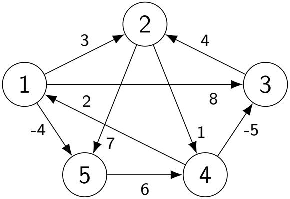
$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

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

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$



$k = 2$

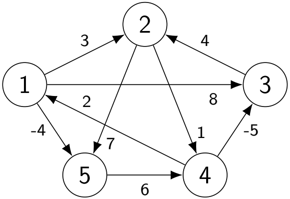
$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

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

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```



$k = 3$

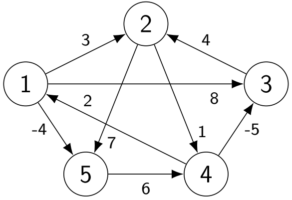
$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

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

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$



$k = 4$

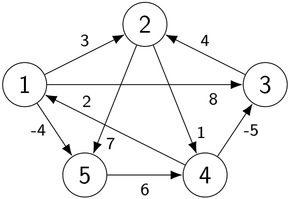
$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix}$$

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

The Floyd-Warshall Algorithm

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n
    for $j = 1$ **to** n
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

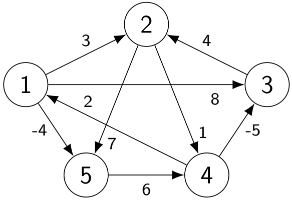


$k = 5$

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

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

The Floyd-Warshall Algorithm



FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

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

Floyd-Warshall - Time Complexity

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

Floyd-Warshall - Time Complexity

FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n                                        → O(n)
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

Floyd-Warshall - Time Complexity

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

$O(n)$
$\times O(n)$

Floyd-Warshall - Time Complexity

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n $\quad O(n)$
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n $\quad \times O(n)$
        for $j = 1$ **to** n $\quad \times O(n)$
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Floyd-Warshall - Time Complexity

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n $\rightarrow O(n)$
  let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
  for $i = 1$ **to** n $\rightarrow \times O(n)$
    for $j = 1$ **to** n $\rightarrow \times O(n)$
      $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Time Complexity: $O(n^3)$ or $O(V^3)$

Floyd-Warshall - Space Complexity

FLOYD-WARSHALL(W)

$n = W.rows$
$D^{(0)} = W$
for $k = 1$ **to** n
    let $D^{(k)} = \left(d_{ij}^{(k)}\right)$ be a new $n \times n$ matrix
    for $i = 1$ **to** n
        for $j = 1$ **to** n
            $d_{ij}^{(k)} = \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right)$
return $D^{(n)}$

Floyd-Warshall - Space Complexity

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

naive approach:
a new matrix for each k,
so $O(n \times n^2) = O(n^3)$

Floyd-Warshall - Space Complexity

naive approach:
a new matrix for each k,
so $O(n \times n^2) = O(n^3)$

FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

actually, at step k
we only need the matrix
from step $k-1$

Floyd-Warshall - Space Complexity

FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

naive approach:
a new matrix for each k,
so $O(n \times n^2) = O(n^3)$

actually, at step k
we only need the matrix
from step $k - 1$

in fact, we can work
on the same matrix

Floyd-Warshall - Space Complexity

FLOYD-WARSHALL(W)

```
n = W.rows
D^(0) = W
for k = 1 to n
    let D^(k) = (d_ij^(k)) be a new n × n matrix
    for i = 1 to n
        for j = 1 to n
            d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
return D^(n)
```

naive approach:
a new matrix for each k,
so $O(n \times n^2) = O(n^3)$

actually, at step k
we only need the matrix
from step $k - 1$

in fact, we can work
on the same matrix

Space Complexity: $O(n^2)$ or $O(V^2)$

Floyd-Warshall - Path Construction

- build a matrix $\Pi = (\pi_{ij})$, where π_{ij} is the parent of node j in the shortest path from i to j

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

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

- $\Pi = \Pi^{(n)}$

The Floyd-Warshall Algorithm with Path (1)

FLOYD-WARSHALL-2(W)

```
n = W.rows
for i = 1 to n
    for j = 1 to n
        d_ij = w_ij
        if i == j or w_ij == ∞
            π_ij = NIL
        else π_ij = i
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            if d_ij > d_ik + d_kj
                d_ij = d_ik + d_kj
                π_ij = π_kj
return D, Π
```

The Floyd-Warshall Algorithm with Path (2)

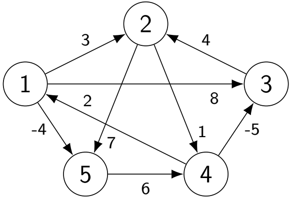
PATH-CONSTRUCTION(Π, i, j)

if $\pi_{ij} == \text{NIL}$
    return NIL
$path = \{j\}$
while $j \neq i$
    $j = \pi_{ij}$
    $path = \{j\} \cup path$
return *path*

The Floyd-Warshall Algorithm with Path (2)

PATH-CONSTRUCTION(Π, i, j)

if $\pi_{ij} == \text{NIL}$
    return NIL
path $= \{j\}$
while $j \neq i$
    $j = \pi_{ij}$
    path $= \{j\} \cup$ *path*
return *path*



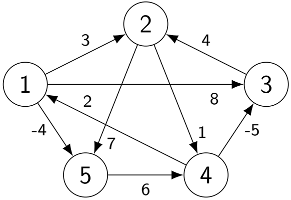
$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$i = 2; j = 5$

The Floyd-Warshall Algorithm with Path (2)

PATH-CONSTRUCTION(Π, i, j)

if $\pi_{ij} == \text{NIL}$
  return NIL
$path = \{j\}$
while $j \neq i$
  $j = \pi_{ij}$
  $path = \{j\} \cup path$
return $path$



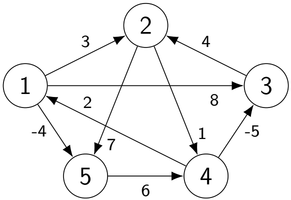
$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$i = 2; j = 5$
$path = \{ \quad 5 \}$

The Floyd-Warshall Algorithm with Path (2)

$\textsc{Path-Construction}(\Pi, i, j)$

```
if π_ij == NIL
    return NIL
path = {j}
while j ≠ i
    j = π_ij
    path = {j} ∪ path
return path
```

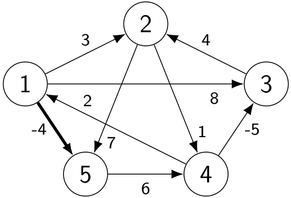


$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$i = 2;\ j = \cancel{5}\ 1$

$path = \{\ \ 5\ \}$

The Floyd-Warshall Algorithm with Path (2)



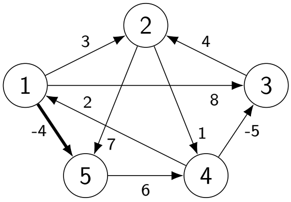
$\textsc{Path-Construction}(\Pi, i, j)$

if $\pi_{ij} == \text{NIL}$
    return NIL
$path = \{j\}$
while $j \neq i$
    $j = \pi_{ij}$
    $path = \{j\} \cup path$
return *path*

$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$i = 2;\ j = \cancel{5}\ 1$
$path = \{\ \ 1,\ 5\ \}$

The Floyd-Warshall Algorithm with Path (2)



PATH-CONSTRUCTION(Π, i, j)

```
if π_ij == NIL
      return NIL
path = {j}
while j ≠ i
      j = π_ij
      path = {j} ∪ path
return path
```

$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

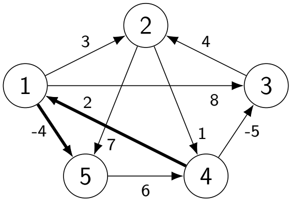
$i = 2; j = \cancel{5}\ \cancel{1}\ 4$

$path = \{\ \ 1, 5\ \}$

The Floyd-Warshall Algorithm with Path (2)

PATH-CONSTRUCTION(Π, i, j)

if $\pi_{ij} == \text{NIL}$
    return NIL
path $= \{j\}$
while $j \neq i$
    $j = \pi_{ij}$
    path $= \{j\} \cup$ *path*
return *path*



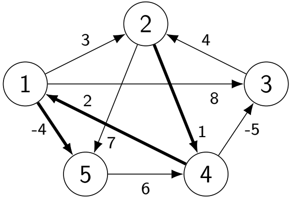
$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$i = 2;\ j = \cancel{5}\ \cancel{1}\ 4$
$path = \{\ 4, 1, 5\ \}$

The Floyd-Warshall Algorithm with Path (2)

PATH-CONSTRUCTION(Π, i, j)

```
if π_ij == NIL
    return NIL
path = {j}
while j ≠ i
    j = π_ij
    path = {j} ∪ path
return path
```



$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

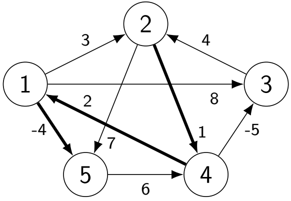
$i = 2; j = \cancel{5}\ \cancel{1}\ \cancel{4}\ 2$

$path = \{\ 4, 1, 5\ \}$

The Floyd-Warshall Algorithm with Path (2)

PATH-CONSTRUCTION(Π, i, j)

```
if π_ij == NIL
    return NIL
path = {j}
while j ≠ i
    j = π_ij
    path = {j} ∪ path
return path
```



$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

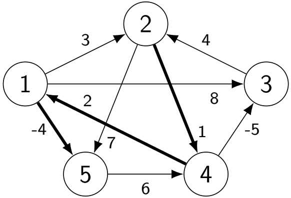
$i = 2;\ j = \cancel{5}\ \cancel{1}\ \cancel{4}\ 2$

$path = \{\ 2, 4, 1, 5\ \}$

The Floyd-Warshall Algorithm with Path (2)

PATH-CONSTRUCTION(Π, i, j)

```
if π_ij == NIL
      return NIL
path = {j}
while j ≠ i
      j = π_ij
      path = {j} ∪ path
return path
```



$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

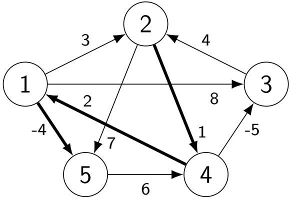
$i = 2; j = \cancel{5}\ \cancel{1}\ \cancel{4}\ 2$

$path = \{\ 2, 4, 1, 5\ \}$

The Floyd-Warshall Algorithm with Path (2)

PATH-CONSTRUCTION(Π, i, j)

```
if π_ij == NIL
    return NIL
path = {j}
while j ≠ i
    j = π_ij
    path = {j} ∪ path
return path
```



$$\Pi = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$i = 2;\ j = \cancel{5}\ \cancel{1}\ \cancel{4}\ 2$

$path = \{\ 2, 4, 1, 5\ \}$

Transitive closure

Definition

Given a directed graph $G = (V, E)$, with $V = \{1, 2, \ldots, n\}$, we define the **transitive closure** as $G = (V, E^\star)$, where $E^\star = \{(i, j) : \exists i \rightsquigarrow j\}$.

- we want to determine whether G contains a path from i to j for all pairs $i, j \in V$
- solution 1: run Floyd-Warshall with $w_{ij} = 1, \forall (i, j) \in E$ and check if $d_{ij} < \infty$
- solution 2: change arithmetical operations into logical operations

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

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

The transitive closure algorithm

TRANSITIVE-CLOSURE(G)

```
n = |G.V|
let T = (t_ij) be a new n × n matrix
for i = 1 to n
    for j = 1 to n
        if i == j or (i, j) ∈ G.E
            t_ij = 1
        else t_ij = 0
for k = 1 to n
    for i = 1 to n
        for j = 1 to n
            t_ij = t_ij ∨ (t_ik ∧ t_kj)
return T
```

Bibliography

- Cormen, Thomas H., et al., *"Introduction to algorithms."*, MIT press, 2009, cap. 24, cap. 25