Algorithms: SHORTEST PATHS AND PROBLEM SOLVING

Ola Svensson

EPFL School of Computer and Communication Sciences

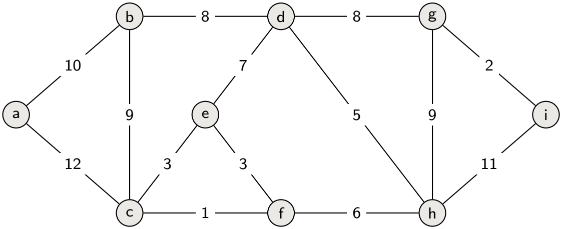
Lecture 21, 6.05.2025

Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

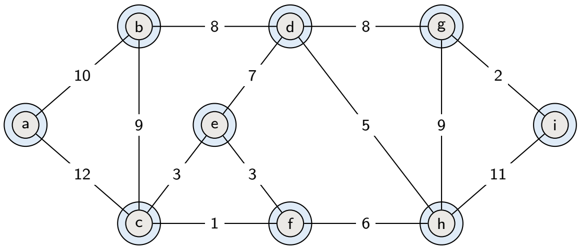


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

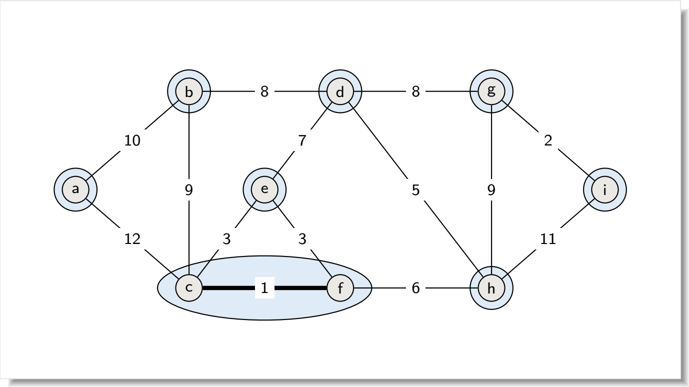


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

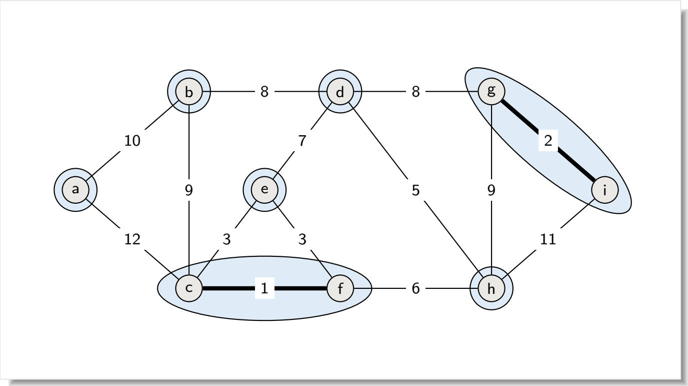


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

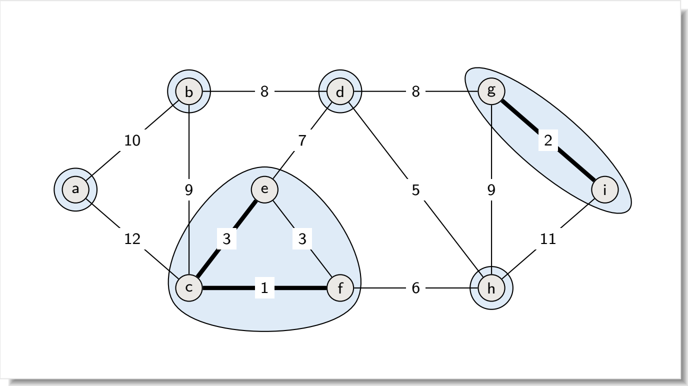


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

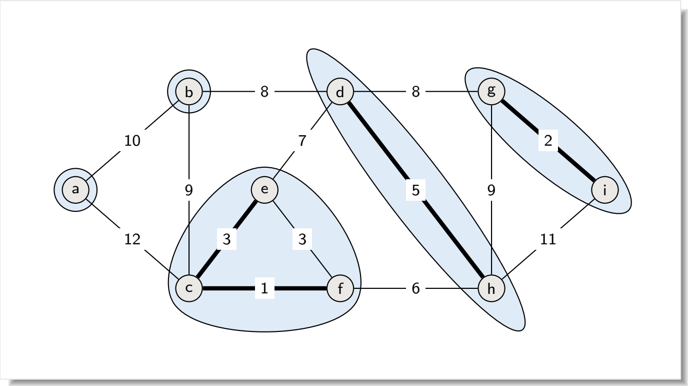


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

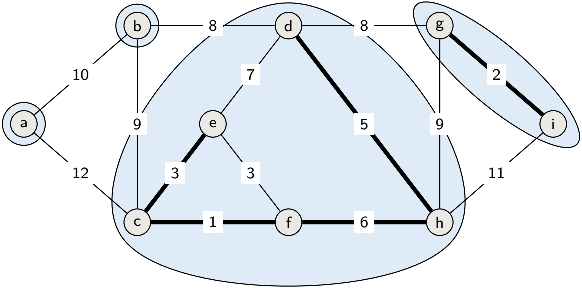


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

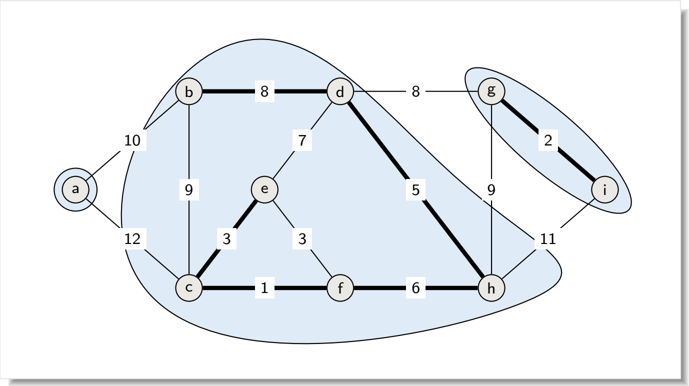


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

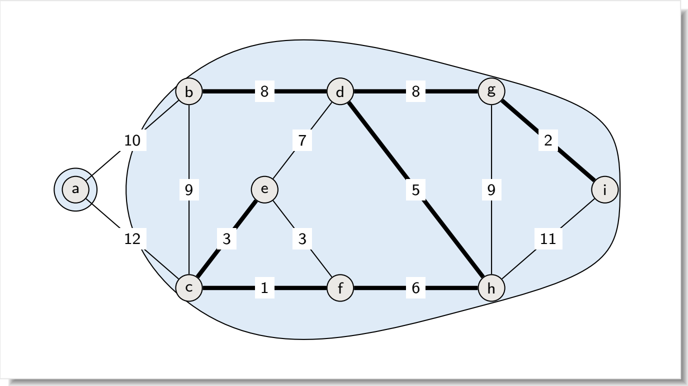


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

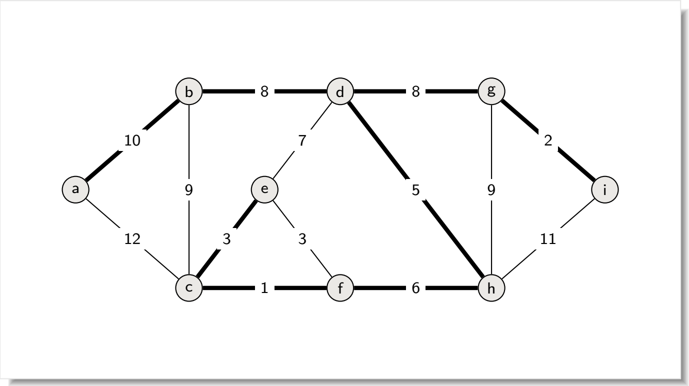


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle

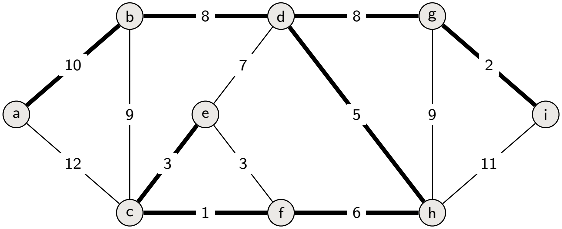


Kruskal's algorithm

Start from empty forest T

Greedily maintain forest T which will become MST at the end:

at each step add cheapest edge that does not create a cycle



Minimum spanning tree of weight $1 + 2 + 3 + 5 + 6 + 8 + 8 + 10 = 43$

Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

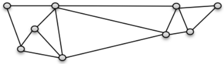
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

Base case: trivial

T is a union of singleton vertices



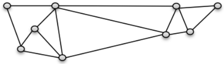
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

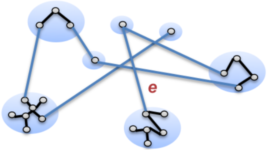
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,



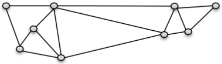
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

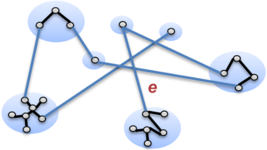
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,
2. Edge ***e*** is an edge of minimum weight that doesn't create a cycle



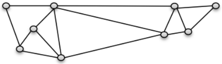
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

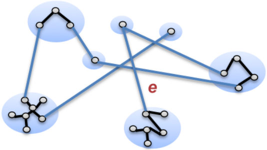
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,
2. Edge ***e*** is an edge of minimum weight that doesn't create a cycle
3. *Suppose* ***e*** creates a cycle with MST



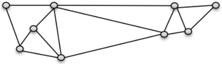
Why does it work?

Claim: T is always a sub-forest of a MST

Proof by induction on the number of components/edges in T

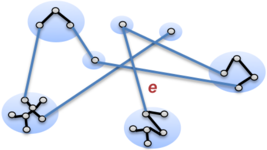
Base case: trivial

T is a union of singleton vertices



Inductive step:

1. By hypothesis, current ***T*** is a sub-forest of a MST,
2. Edge ***e*** is an edge of minimum weight that doesn't create a cycle
3. *Suppose* ***e*** creates a cycle with MST
4. Replace an edge (with larger weight) along this cycle by ***e***



An MST since weight did not increase!

Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

This is the same thing as checking whether its endpoints belong to the same component $\Rightarrow$ **use disjoint sets (union-find) data structure**

Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

This is the same thing as checking whether its endpoints belong to the same component $\Rightarrow$ **use disjoint sets (union-find) data structure**

Let the connected components denote sets

- Initially each singleton is a set
- When edge (u, v) is added to T, make union of the two connected components/sets

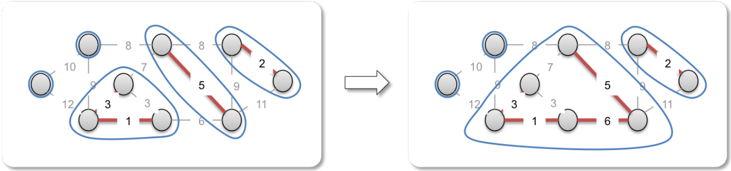
Implementation challenge

In each iteration, we need to check whether cheapest edge creates a cycle

This is the same thing as checking whether its endpoints belong to the same component ⇒ **use disjoint sets (union-find) data structure**

Let the connected components denote sets

- Initially each singleton is a set
- When edge (u, v) is added to T, make union of the two connected components/sets



Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A:

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop:

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V Make-Sets

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E :

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E : $O(E \lg E)$

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E : $O(E \lg E)$
- Second **for** loop:

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E : $O(E \lg E)$
- Second **for** loop: $O(E)$ FIND-SETS and UNIONS

Implementation and Analysis

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E : $O(E \lg E)$
- Second **for** loop: $O(E)$ FIND-SETS and UNIONS
- Total time: $O((V + E)\alpha(V)) + O(E \lg E) = O(E \lg E) = O(E \lg V)$

Implementation and Analysis

KRUSKAL(G, w)
  $A = \emptyset$
  for each vertex $v \in G.V$
    MAKE-SET(v)
  sort the edges of $G.E$ into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
    if FIND-SET$(u) \neq$ FIND-SET(v)
      $A = A \cup \{(u, v)\}$
      UNION(u, v)
  return A

- Initialize A: $O(1)$
- First **for** loop: V MAKE-SETS
- Sort E : $O(E \lg E)$
- Second **for** loop: $O(E)$ FIND-SETS and UNIONS
- Total time: $O((V+E)\alpha(V)) + O(E \lg E) = O(E \lg E) = O(E \lg V)$

  If edges already sorted time is $O(E\alpha(V))$ which is almost linear

Problem solving

A feedback edge set of a graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of a given edge-weighted graph.

Problem solving

A feedback edge set of a graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of a given edge-weighted graph.

Let $G = (V, E)$ be an arbitrary connected graph with weighted edges. Assume further that no two edges have the same weight.

1. Prove that for any partition of the vertices V into two subsets, the minimum-weight edge with one endpoint in each subset is in the minimum spanning tree of G.

Problem solving

A feedback edge set of a graph G is a subset F of the edges such that every cycle in G contains at least one edge in F. In other words, removing every edge in F makes the graph G acyclic. Describe and analyze a fast algorithm to compute the minimum weight feedback edge set of a given edge-weighted graph.

Let $G = (V, E)$ be an arbitrary connected graph with weighted edges. Assume further that no two edges have the same weight.

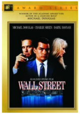
1. Prove that for any partition of the vertices V into two subsets, the minimum-weight edge with one endpoint in each subset is in the minimum spanning tree of G.
2. Prove that the maximum-weight edge in any cycle of G is not in the minimum spanning tree of G.

Summary

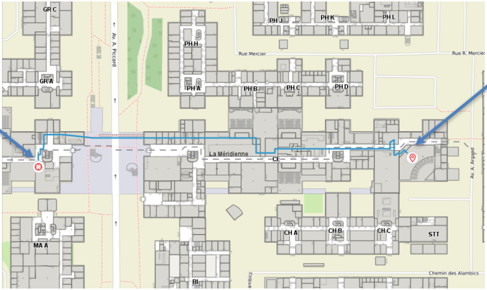
- Greedy is good (sometimes)
- Prim's algorithm

  Min-priority queue for implementation
- Kruskal's algortihm

  Union-Find for implementation
- Many applications



Satellite

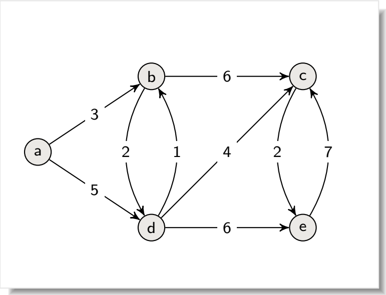


this class room

THE SHORTEST PATH PROBLEM

Shortest paths

Input: directed graph $G = (V, E)$, edge-weights $w(u, v)$ for $(u, v) \in E$

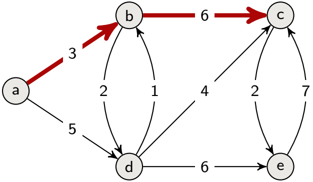


Shortest paths

Input: directed graph $G = (V, E)$, edge-weights $w(u, v)$ for $(u, v) \in E$

Weight of path $\langle v_0, v_1, \ldots, v_k \rangle$: $\sum_{i=1}^{k} w(v_{i-1}, v_i)$

Path of weight $3 + 6 = 9$

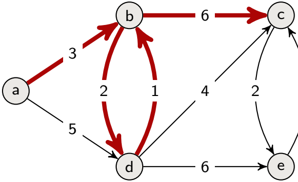


Shortest paths

Input: directed graph $G = (V, E)$, edge-weights $w(u, v)$ for $(u, v) \in E$

Weight of path $\langle v_0, v_1, \ldots, v_k \rangle$: $\sum_{i=1}^{k} w(v_{i-1}, v_i)$

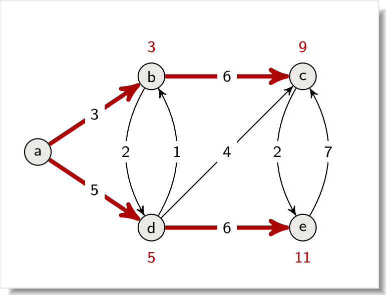
Path of weight $3 + 2 + 1 + 6 = 12$



Shortest paths

Input: directed graph $G = (V, E)$, edge-weights $w(u, v)$ for $(u, v) \in E$

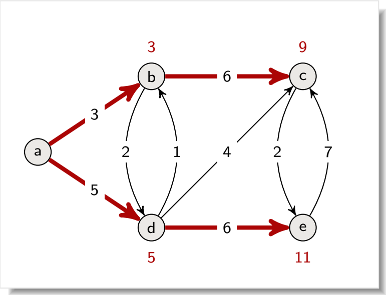
Shortest paths from a: (may have many solutions)

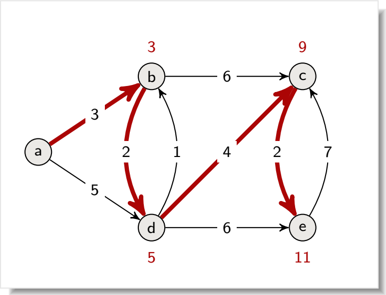


Shortest paths

Input: directed graph $G = (V, E)$, edge-weights $w(u, v)$ for $(u, v) \in E$

Shortest paths from a: (may have many solutions)





Problem variants

Problem variants

Single-source: Find shortest paths from **source** vertex to every vertex

Problem variants

Single-source: Find shortest paths from **source** vertex to every vertex

Single-destination: Find shortest paths to given **destination** vertex

Problem variants

Single-source: Find shortest paths from **source** vertex to every vertex

Single-destination: Find shortest paths to given **destination** vertex

Single-pair: Find shortest path from u to v

Problem variants

Single-source: Find shortest paths from **source** vertex to every vertex

Single-destination: Find shortest paths to given **destination** vertex

Single-pair: Find shortest path from u to v

All-pairs: Find shortest path from u to v for all pairs u, v of vertices

Problem variants

Single-source: Find shortest paths from **source** vertex to every vertex

Single-destination: Find shortest paths to given **destination** vertex

Can be solved by single-source by reversing edge directions

Single-pair: Find shortest path from u to v

All-pairs: Find shortest path from u to v for all pairs u, v of vertices

Problem variants

Single-source: Find shortest paths from **source** vertex to every vertex

Single-destination: Find shortest paths to given **destination** vertex

Can be solved by single-source by reversing edge directions

Single-pair: Find shortest path from u to v

No algorithm known that is better in worst case than solving single-source

All-pairs: Find shortest path from u to v for all pairs u, v of vertices

Problem variants

Single-source: Find shortest paths from **source** vertex to every vertex

Single-destination: Find shortest paths to given **destination** vertex

Can be solved by single-source by reversing edge directions

Single-pair: Find shortest path from u to v

No algorithm known that is better in worst case than solving single-source

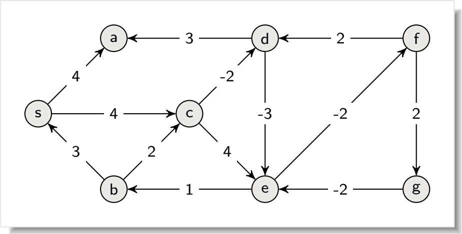
All-pairs: Find shortest path from u to v for all pairs u, v of vertices

Can be solved by solving single-source for each vertex. Better algorithms known

NEGATIVE WEIGHTS AND APPLICATIONS

Negative-weight edges

We will allow **negative weights**

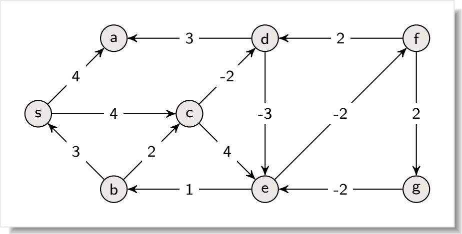


Negative-weight edges

We will allow **negative weights**

OK, as long as no negative-weight cycle is reachable from source:

- Then we can just keep going around it to have paths of length $-\infty$



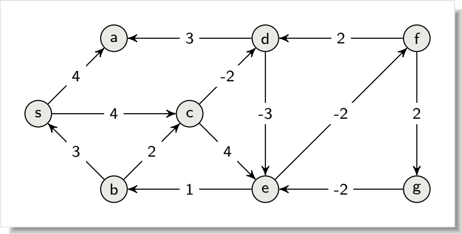
Negative-weight edges

We will allow **negative weights**

OK, as long as no negative-weight cycle is reachable from source:

- Then we can just keep going around it to have paths of length $-\infty$

Some algorithms only work with positive weights (Dijkstra's algorithm)



Why negative weights?

Example: Buying and selling currency

Input: table of exchange rates

	Euro	Dollar	RMB	CHF
Euro	1.0	1.3	8	1.2
Dollar	0.77	1.0	6.2	0.93
RMB	0.14	0.16	1.0	0.15
CHF	0.83	1.07	6.67	1.0

Why negative weights?

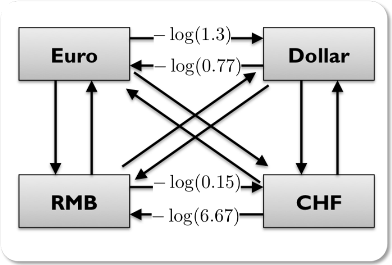
Example: Buying and selling currency

Input: table of exchange rates

	Euro	Dollar	RMB	CHF
Euro	1.0	1.3	8	1.2
Dollar	0.77	1.0	6.2	0.93
RMB	0.14	0.16	1.0	0.15
CHF	0.83	1.07	6.67	1.0

Graph:

- a vertex for each currency
- An edge of weight between two currencies of weight –log(rate)



Why negative weights?

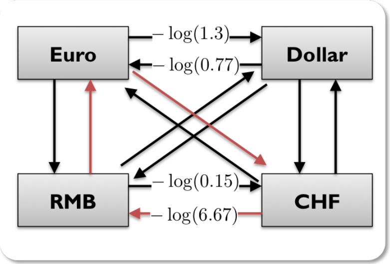
Example: Buying and selling currency

Input: table of exchange rates

	Euro	Dollar	RMB	CHF
Euro	1.0	1.3	8	1.2
Dollar	0.77	1.0	6.2	0.93
RMB	0.14	0.16	1.0	0.15
CHF	0.83	1.07	6.67	1.0

Graph:

- a vertex for each currency
- An edge of weight between two currencies of weight –log(rate)



Check for negative cycle

$$-\log(6.67) - \log(0.14) - \log(1.2) \approx -0.11$$

Why negative weights?

Example: Buying and selling currency

Strategy from negative cycle:

Start with 10 000 CHF

Convert CHF to RMB

Convert RMB to Euro

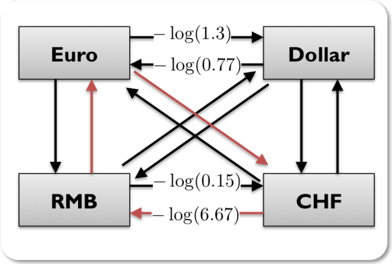
Convert Euro to CHF

Amount of CHF obtained:

$$10^{\log(6.67)+\log(0.14)+\log(1.2)} \cdot 10000$$

$\approx$ 13000

	Euro	Dollar	RMB	CHF
Euro	1.0	1.3	8	1.2
Dollar	0.77	1.0	6.2	0.93
RMB	0.14	0.16	1.0	0.15
CHF	0.83	1.07	6.67	1.0



Why negative weights?

Example: Buying and selling currency

Strategy from negative cycle:

Start with 10 000 CHF

Convert CHF to RMB

Convert RMB to Euro

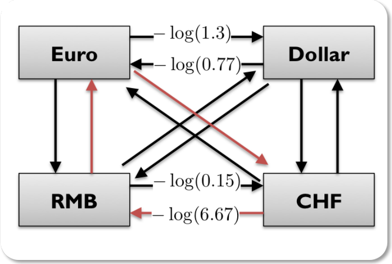
Convert Euro to CHF

Amount of CHF obtained:

$$10^{\log(6.67)+\log(0.14)+\log(1.2)} \cdot 10000$$

$\approx$ 13000

	Euro	Dollar	RMB	CHF
Euro	1.0	1.3	8	1.2
Dollar	0.77	1.0	6.2	0.93
RMB	0.14	0.16	1.0	0.15
CHF	0.83	1.07	6.67	1.0



BELLMAN-FORD ALGORITHM

Algorithm

Input: directed graph with edge weights, a source s, no negative cycles

Algorithm

Input: directed graph with edge weights, a source s, no negative cycles

For each vertex v keep track of

- $l(v) =$ current *upper estimate* of length of shortest path to v
- $\pi(v) =$ is the predecessor of v in this shortest path

Algorithm

Input: directed graph with edge weights, a source s, no negative cycles

For each vertex v keep track of

- $l(v) =$ current *upper estimate* of length of shortest path to v
- $\pi(v) =$ is the predecessor of v in this shortest path

Start by trivial initialization:

```
INIT-SINGLE-SOURCE(G, s)
  for each v ∈ G.V
      v.d = ∞
      v.π = NIL
  s.d = 0
```

Improving the shortest-path estimate

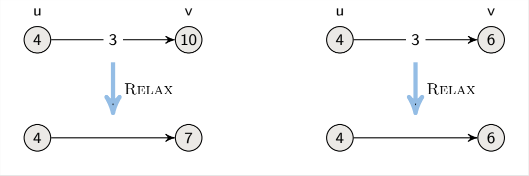
Can we improve the shortest path estimate for v by going through u and taking (u, v)?

Improving the shortest-path estimate

Can we improve the shortest path estimate for v by going through u and taking (u, v)?

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

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



Improving the shortest-path estimate

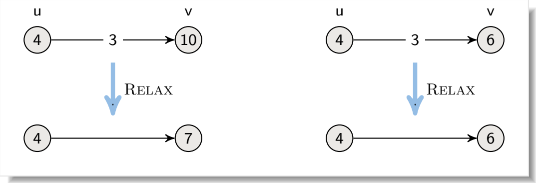
Can we improve the shortest path estimate for v by going through u and taking (u, v)?

RELAX(u, v, w)

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

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

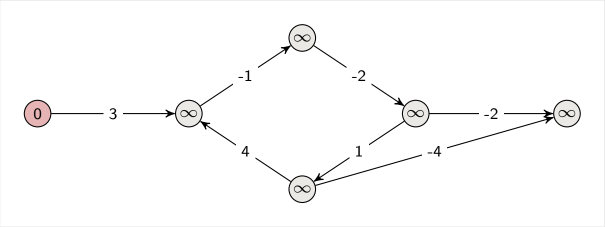
$\quad v.\pi = u$



Bellman-Ford updates shortest-path estimates iteratively by using RELAX

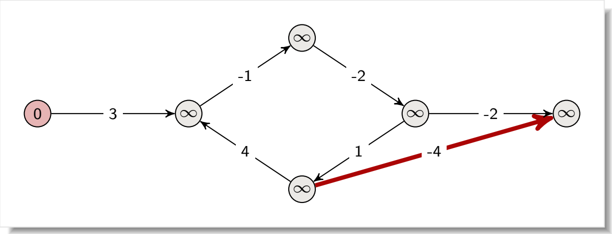
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

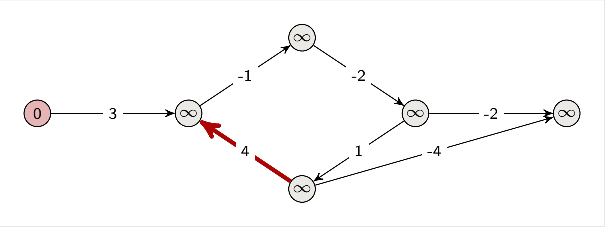
BELLMAN-FORD(G, w, s)
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

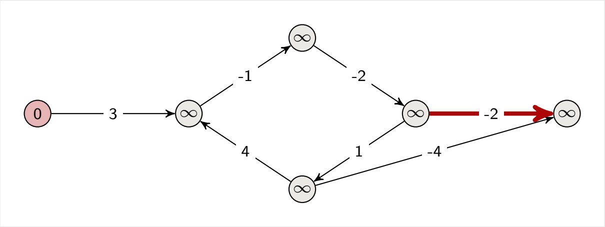
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

$\textsc{Bellman-Ford}(G, w, s)$

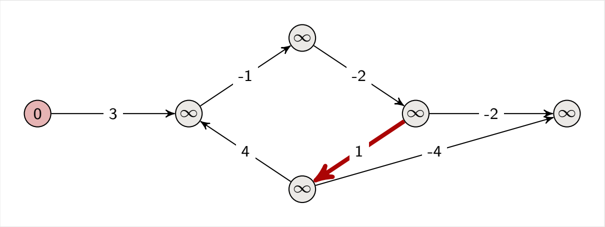
$\quad\textsc{Init-Single-Source}(G, s)$
$\quad$**for** $i = 1$ to $|G.V| - 1$
$\quad\quad$**for** each edge $(u, v) \in G.E$
$\quad\quad\quad\textsc{Relax}(u, v, w)$



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

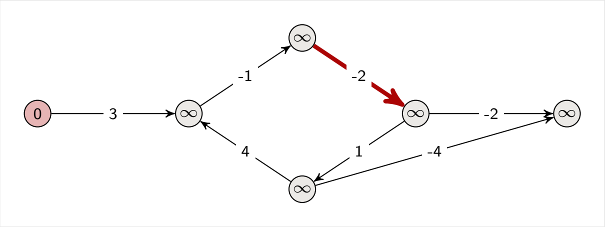
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

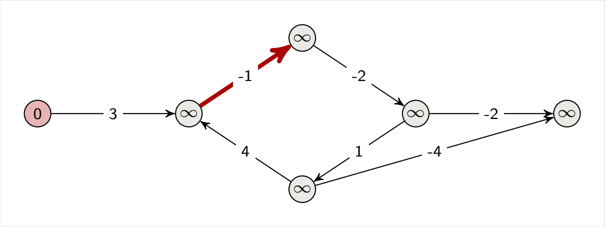
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

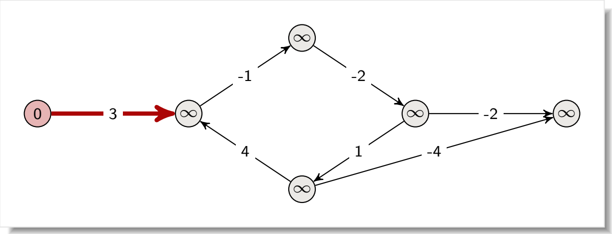
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)

for $i = 1$ to $|G.V| - 1$

  for each edge $(u, v) \in G.E$

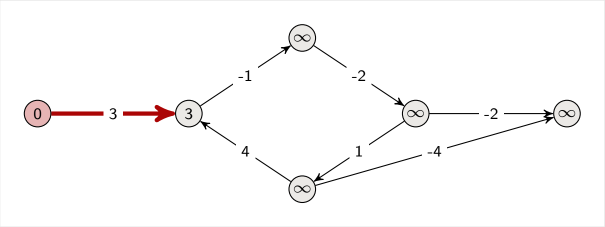
    RELAX(u, v, w)



Bellman-Ford Algorithm

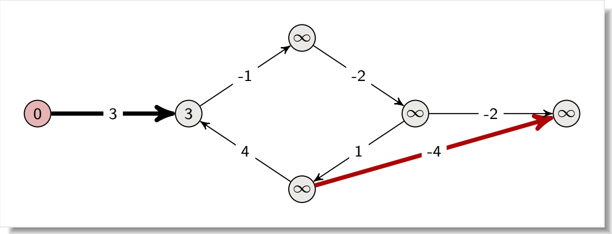
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

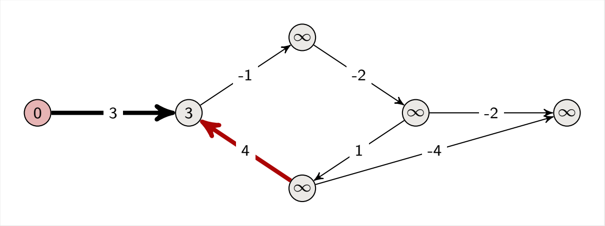
$\textsc{Bellman-Ford}(G, w, s)$
  $\textsc{Init-Single-Source}(G, s)$
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      $\textsc{Relax}(u, v, w)$



Bellman-Ford Algorithm

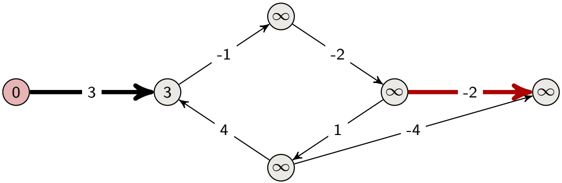
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

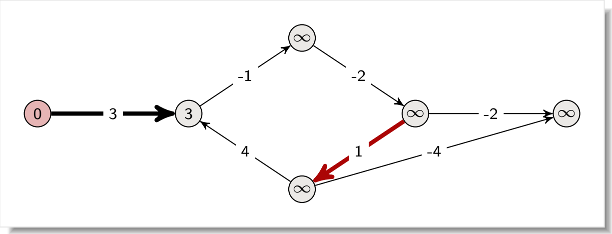
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

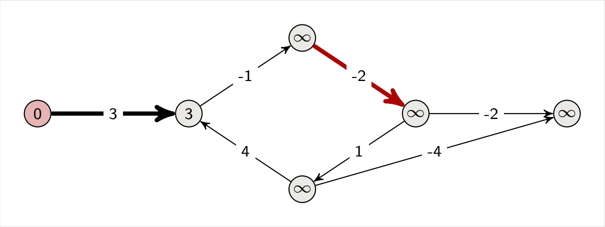
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)

for $i = 1$ to $|G.V| - 1$

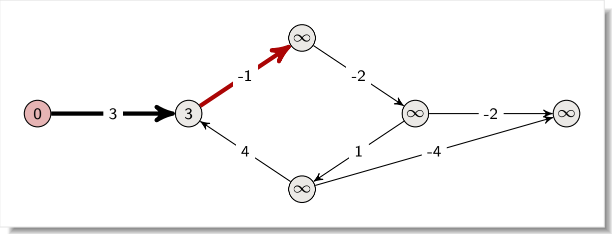
  for each edge $(u, v) \in G.E$

    RELAX(u, v, w)



Bellman-Ford Algorithm

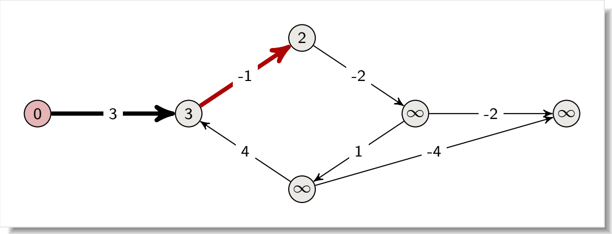
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

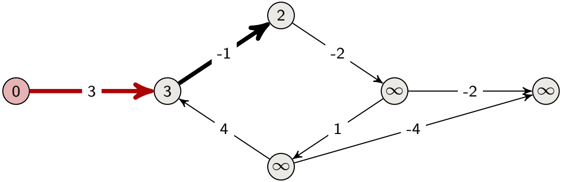
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



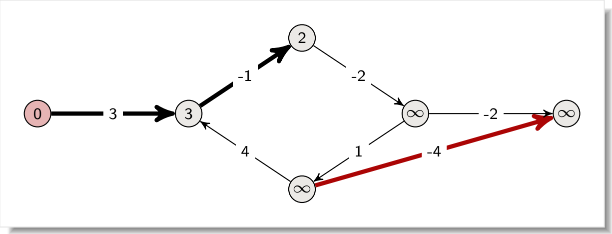
Bellman-Ford Algorithm

$\text{Bellman-Ford}(G, w, s)$
- $\text{Init-Single-Source}(G, s)$
- **for** $i = 1$ to $|G.V| - 1$
  - **for** each edge $(u, v) \in G.E$
    - $\text{Relax}(u, v, w)$



Bellman-Ford Algorithm

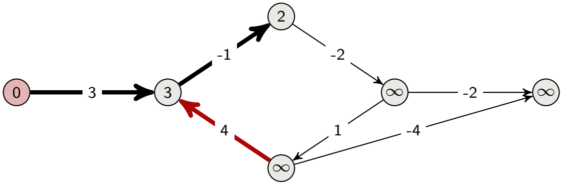
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

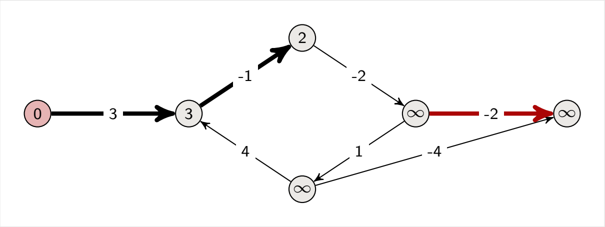
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

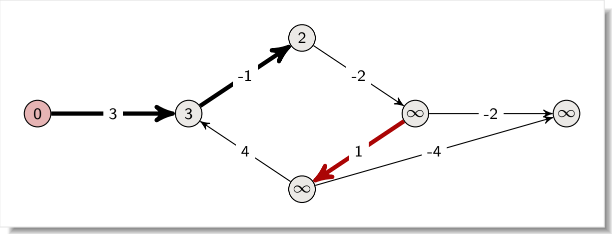
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)

for $i = 1$ to $|G.V| - 1$

  for each edge $(u, v) \in G.E$

    RELAX(u, v, w)



Bellman-Ford Algorithm

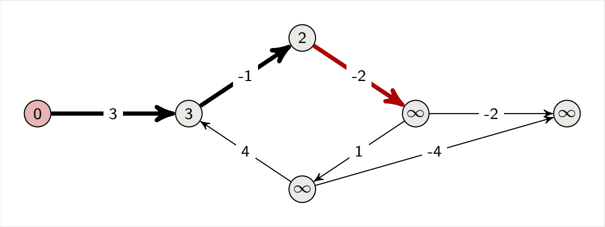
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)

for $i = 1$ to $|G.V| - 1$

for each edge $(u, v) \in G.E$

RELAX(u, v, w)



Bellman-Ford Algorithm

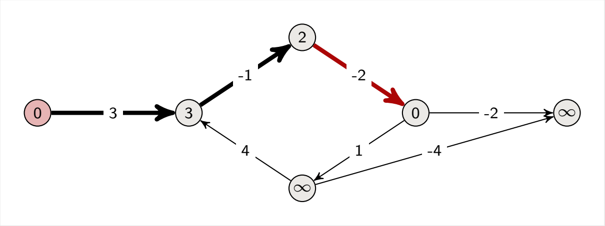
$\textsc{Bellman-Ford}(G, w, s)$

$\quad\textsc{Init-Single-Source}(G, s)$

$\quad\textbf{for } i = 1 \text{ to } |G.V| - 1$

$\qquad\textbf{for} \text{ each edge } (u, v) \in G.E$

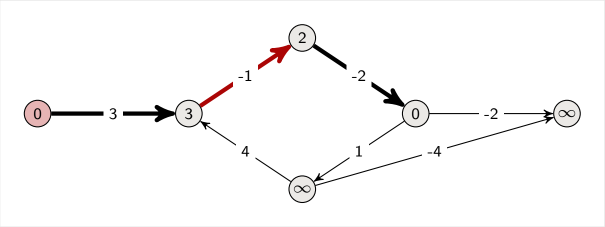
$\qquad\quad\textsc{Relax}(u, v, w)$



Bellman-Ford Algorithm

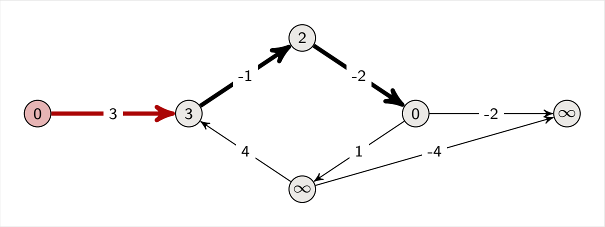
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



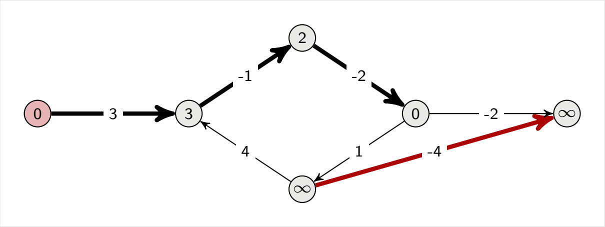
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

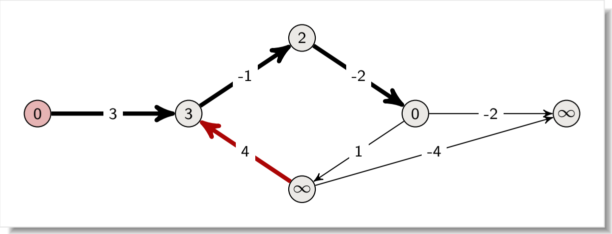
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

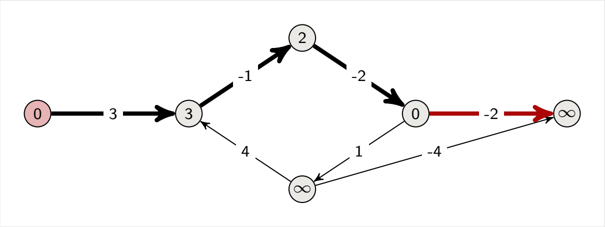
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

$\text{BELLMAN-FORD}(G, w, s)$

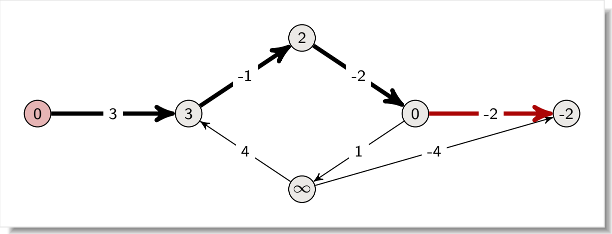
$\quad\text{INIT-SINGLE-SOURCE}(G, s)$
$\quad$**for** $i = 1$ to $|G.V| - 1$
$\qquad$**for** each edge $(u, v) \in G.E$
$\qquad\quad\text{RELAX}(u, v, w)$



Bellman-Ford Algorithm

$\textsc{Bellman-Ford}(G, w, s)$

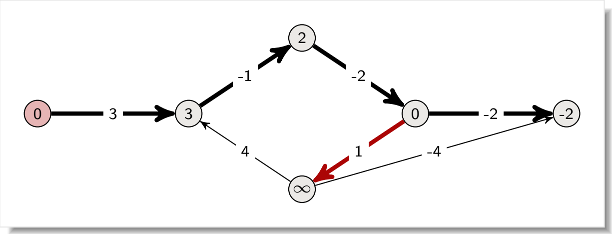
$\quad\textsc{Init-Single-Source}(G, s)$
$\quad\textbf{for } i = 1 \text{ to } |G.V| - 1$
$\qquad\textbf{for} \text{ each edge } (u, v) \in G.E$
$\qquad\quad\textsc{Relax}(u, v, w)$



Bellman-Ford Algorithm

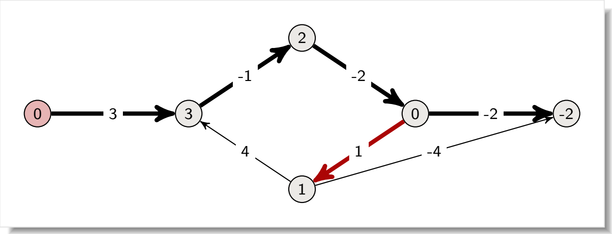
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



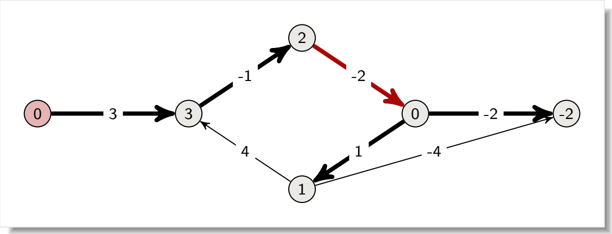
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



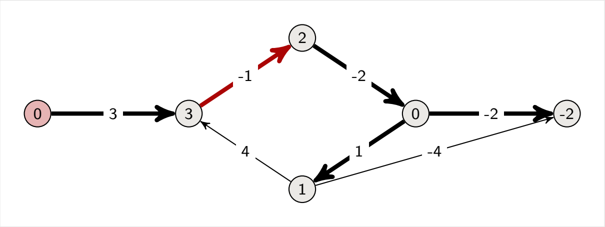
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

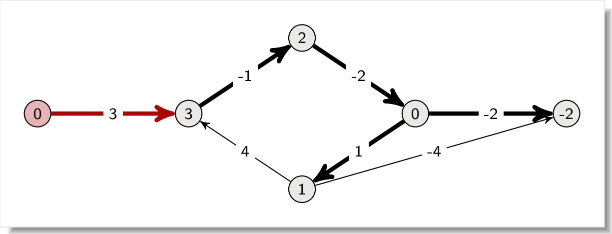
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

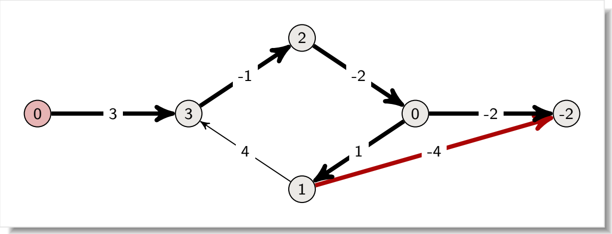
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

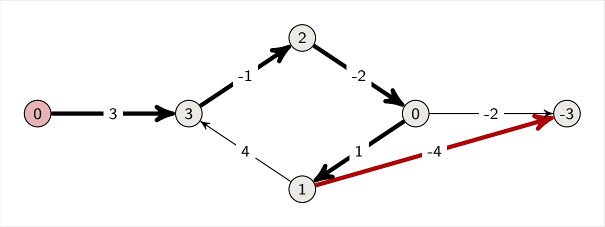
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

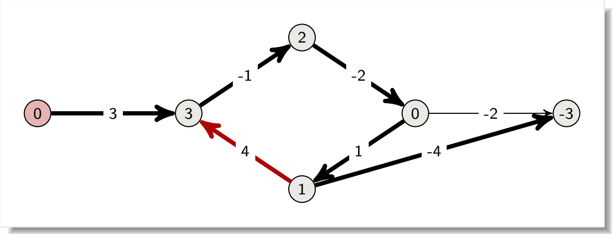
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        RELAX(u, v, w)



Bellman-Ford Algorithm

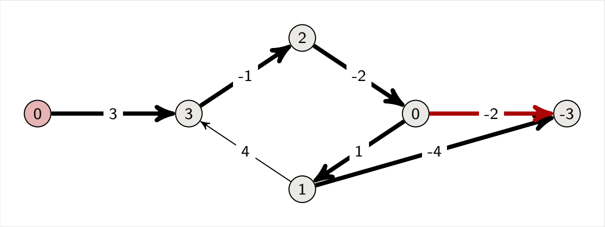
BELLMAN-FORD(G, w, s)
 INIT-SINGLE-SOURCE(G, s)
 for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
   RELAX(u, v, w)



Bellman-Ford Algorithm

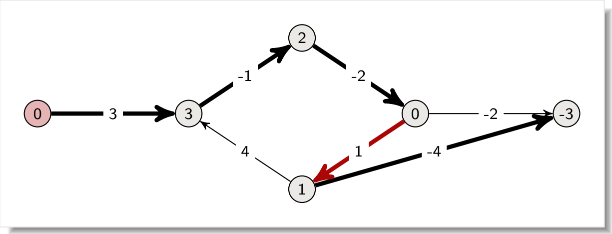
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
      for each edge $(u, v) \in G.E$
          RELAX(u, v, w)



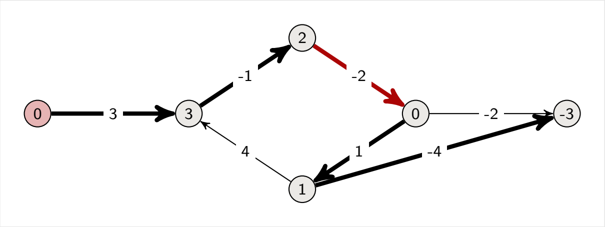
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
for each edge $(u, v) \in G.E$
RELAX(u, v, w)



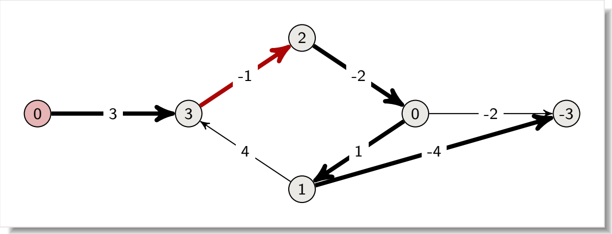
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
      for each edge $(u, v) \in G.E$
          RELAX(u, v, w)



Bellman-Ford Algorithm

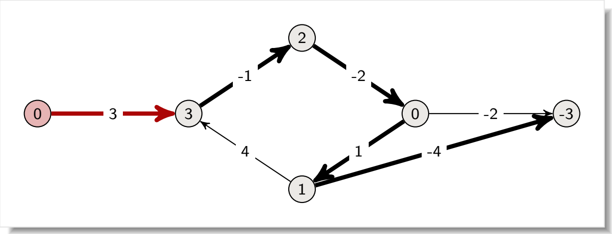
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)

for $i = 1$ to $|G.V| - 1$

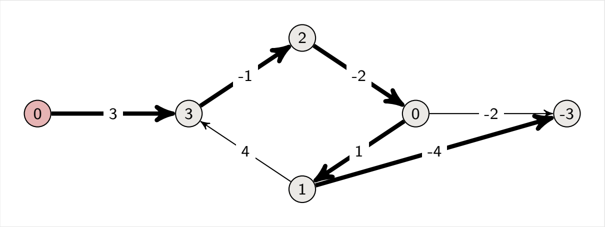
  for each edge $(u, v) \in G.E$

    RELAX(u, v, w)



Bellman-Ford Algorithm

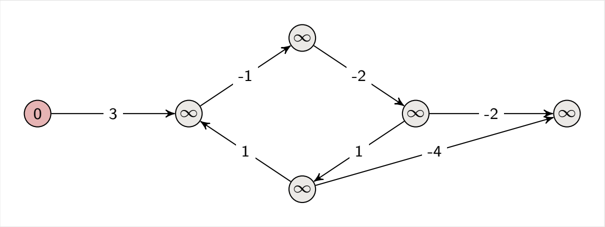
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

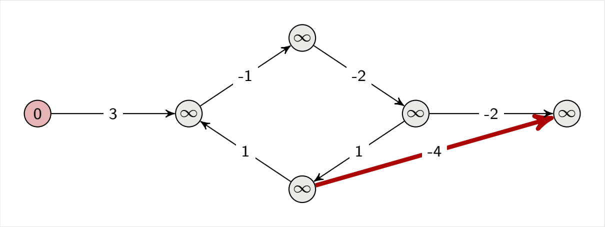
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

Bellman-Ford(G, w, s)

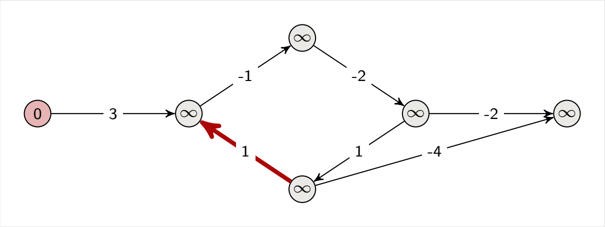
Init-Single-Source(G, s)
for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
        Relax(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

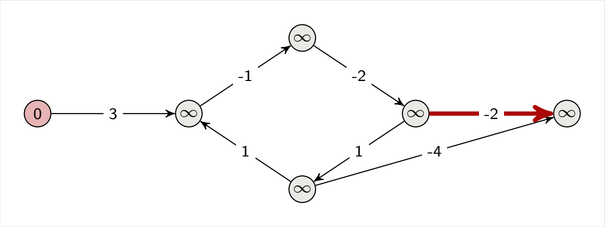
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

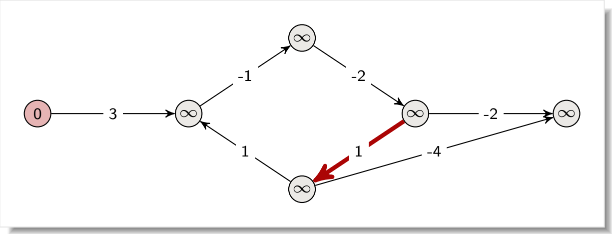
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



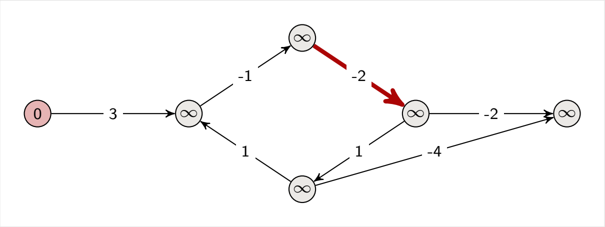
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

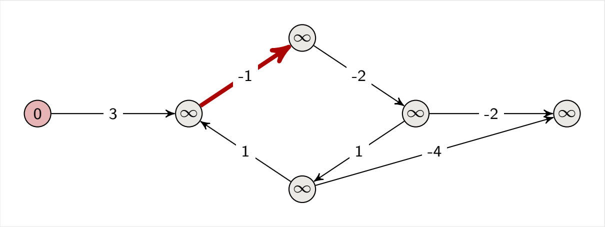
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
      for each edge $(u, v) \in G.E$
          RELAX(u, v, w)



Bellman-Ford Algorithm

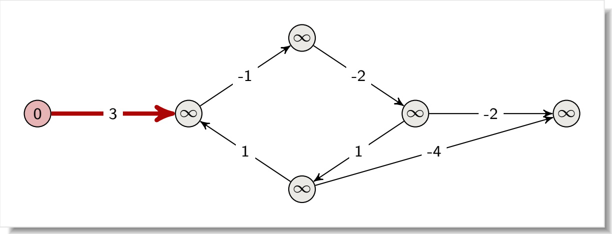
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)

for $i = 1$ to $|G.V| - 1$

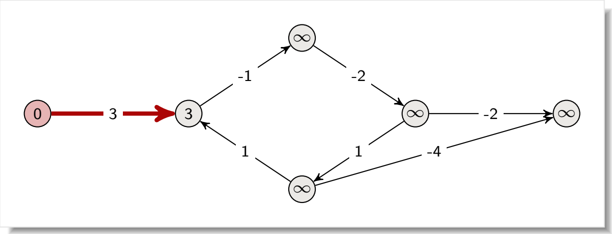
for each edge $(u, v) \in G.E$

RELAX(u, v, w)



Bellman-Ford Algorithm

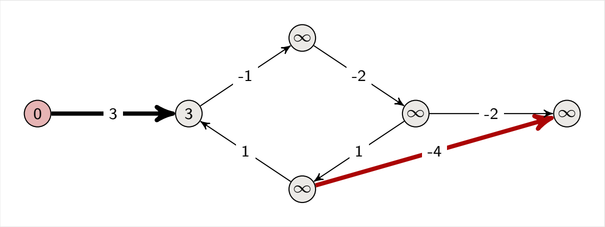
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

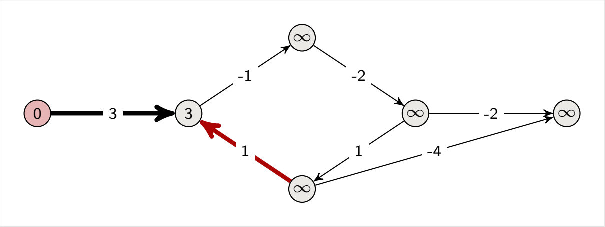
$\textsc{Bellman-Ford}(G, w, s)$

$\quad\textsc{Init-Single-Source}(G, s)$
$\quad$**for** $i = 1$ to $|G.V| - 1$
$\qquad$**for** each edge $(u, v) \in G.E$
$\qquad\quad\textsc{Relax}(u, v, w)$



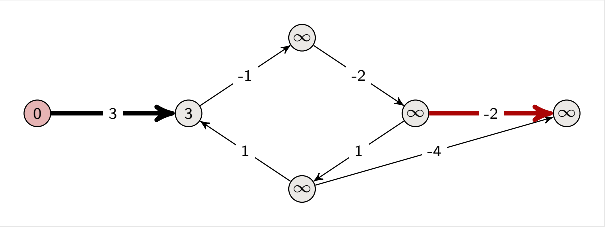
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

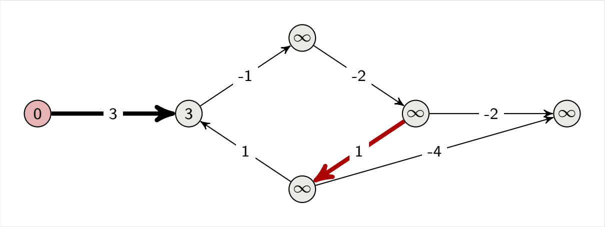
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

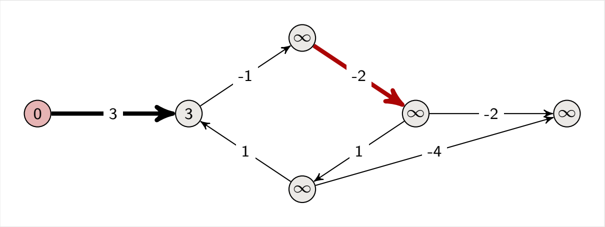
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

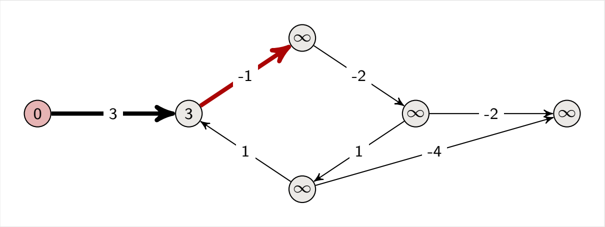
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

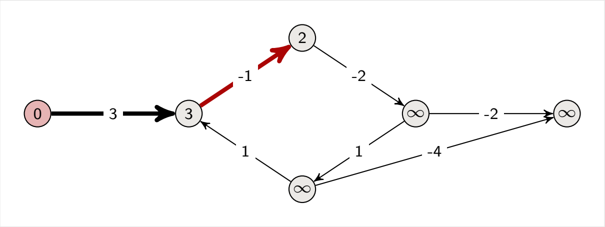
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

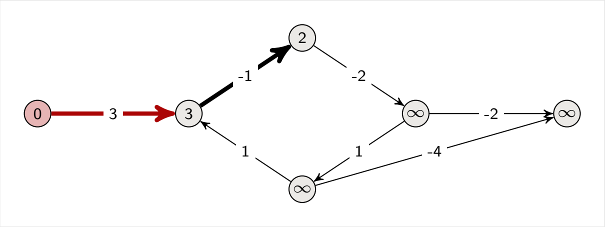
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)

  for $i = 1$ to $|G.V| - 1$

    for each edge $(u, v) \in G.E$

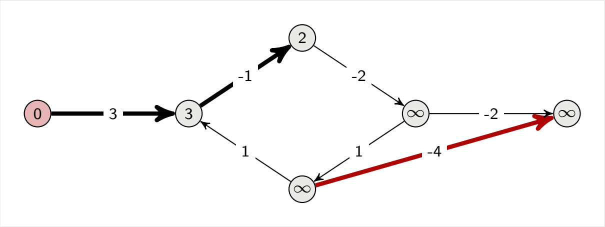
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

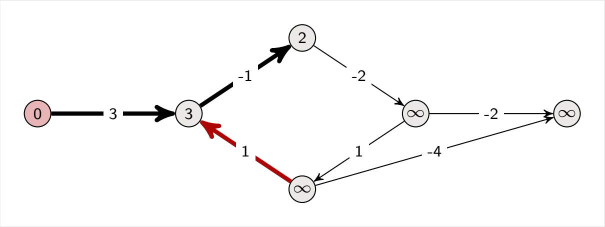
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

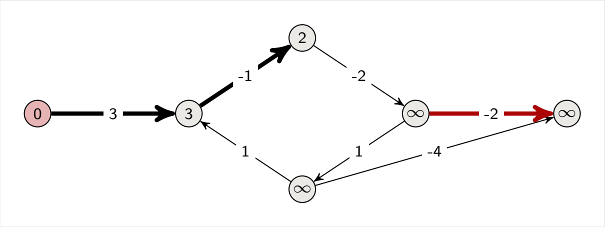
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



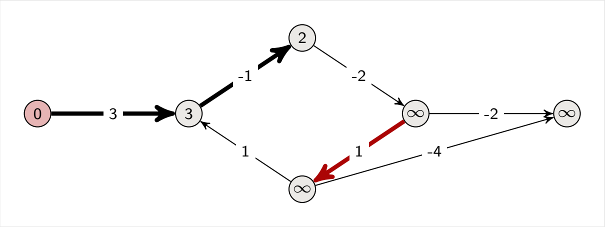
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



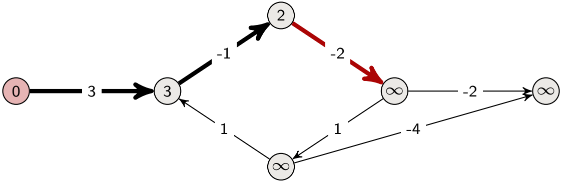
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

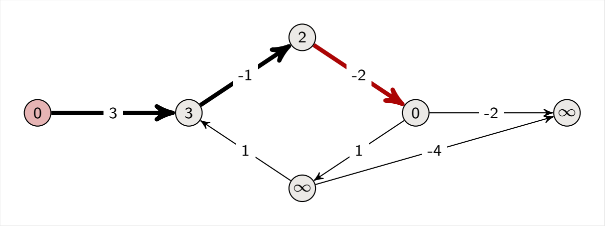
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

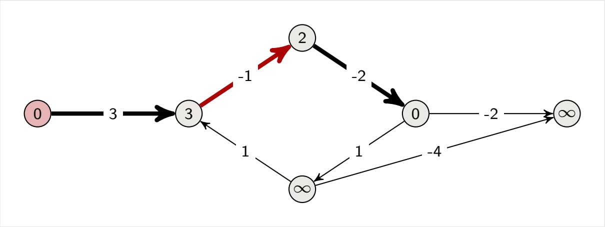
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



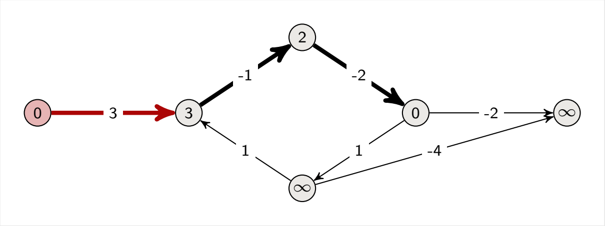
Bellman-Ford Algorithm

$\textsc{Bellman-Ford}(G, w, s)$
$\quad\textsc{Init-Single-Source}(G, s)$
$\quad\textbf{for } i = 1 \text{ to } |G.V| - 1$
$\quad\quad\textbf{for} \text{ each edge } (u, v) \in G.E$
$\quad\quad\quad\textsc{Relax}(u, v, w)$



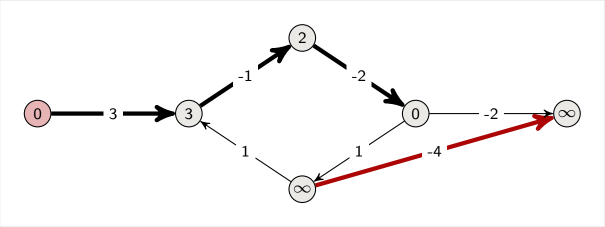
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



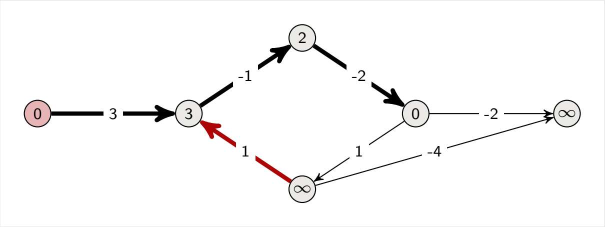
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

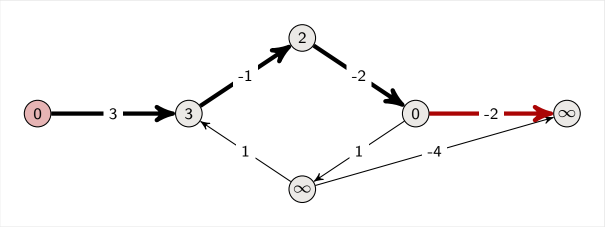
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

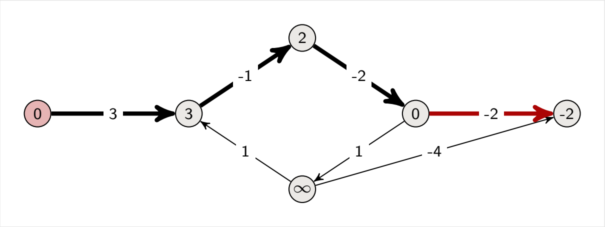
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

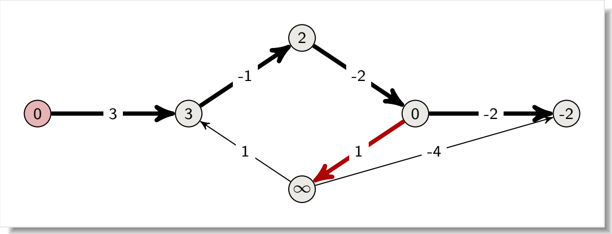
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

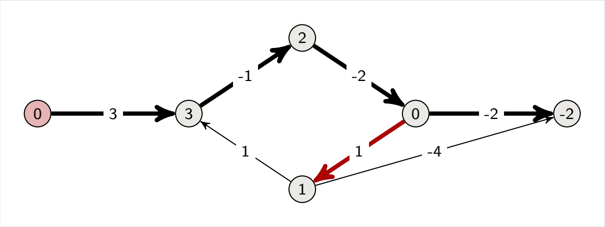
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

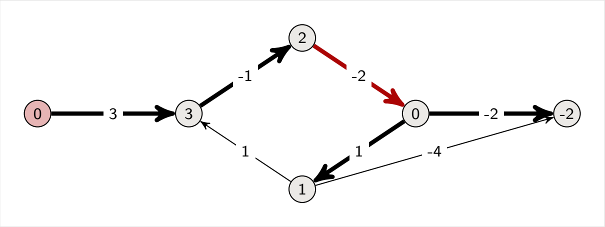
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

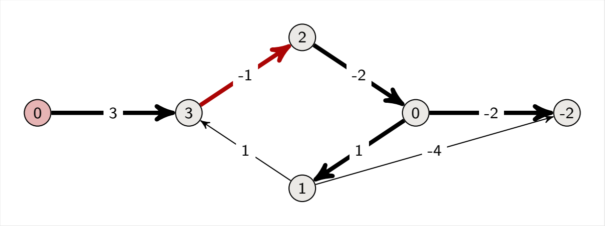
Bellman-Ford(G, w, s)
  Init-Single-Source(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      Relax(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

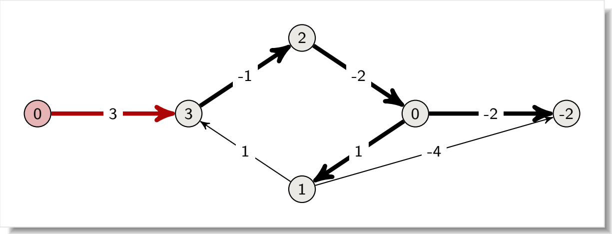
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

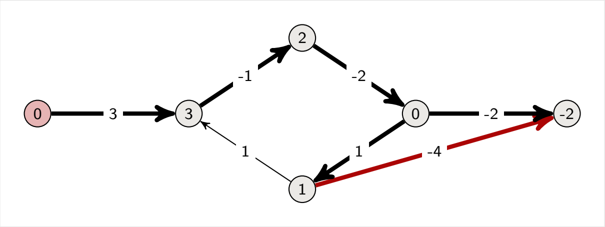
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

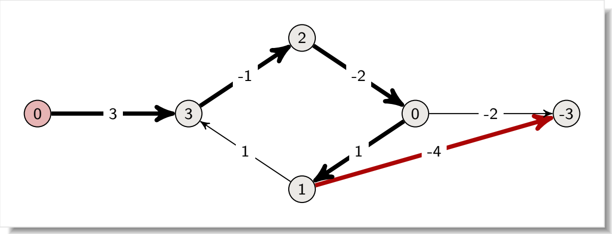
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
      for each edge $(u, v) \in G.E$
          RELAX(u, v, w)



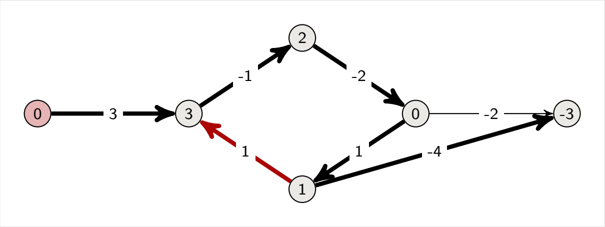
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

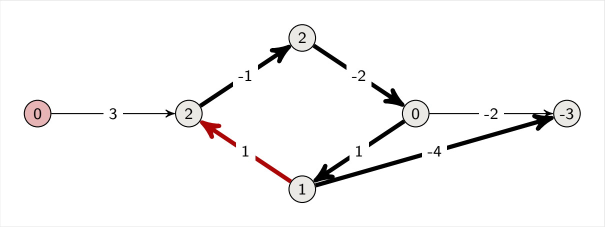
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

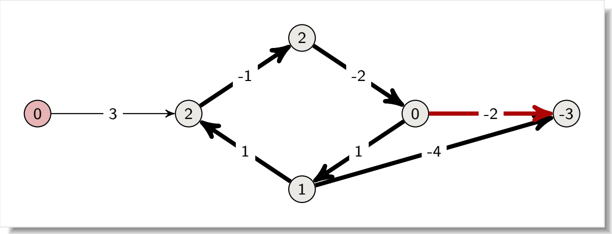
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



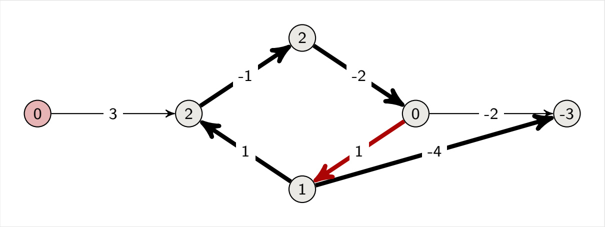
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



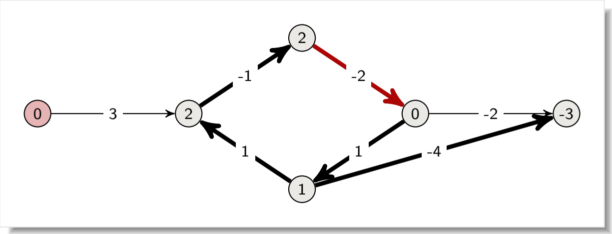
Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Bellman-Ford Algorithm

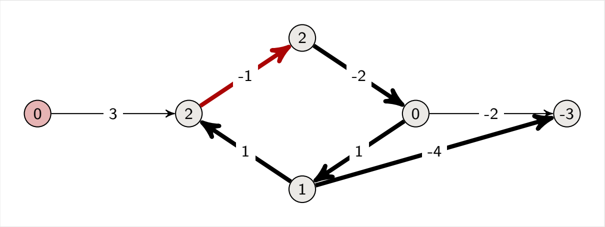
$\textsc{Bellman-Ford}(G, w, s)$
  $\textsc{Init-Single-Source}(G, s)$
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      $\textsc{Relax}(u, v, w)$



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

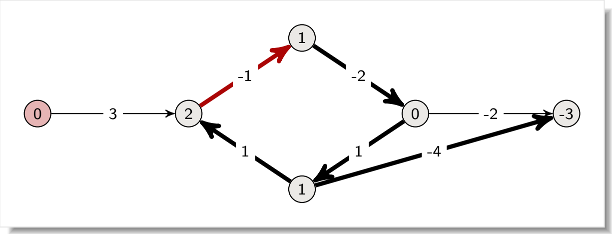
BELLMAN-FORD(G, w, s)

  INIT-SINGLE-SOURCE(G, s)

  for $i = 1$ to $|G.V| - 1$

    for each edge $(u, v) \in G.E$

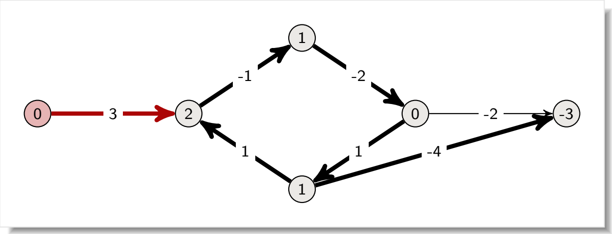
      RELAX(u, v, w)



Bellman-Ford Algorithm

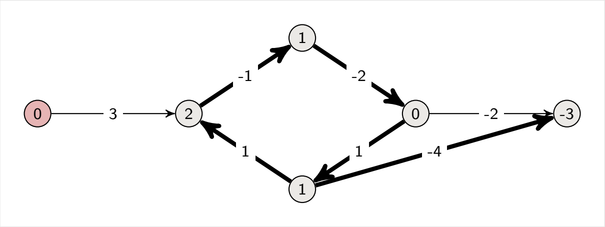
BELLMAN-FORD(G, w, s)

INIT-SINGLE-SOURCE(G, s)
for $i = 1$ to $|G.V| - 1$
  for each edge $(u, v) \in G.E$
    RELAX(u, v, w)



Bellman-Ford Algorithm

BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for $i = 1$ to $|G.V| - 1$
    for each edge $(u, v) \in G.E$
      RELAX(u, v, w)



Correctness

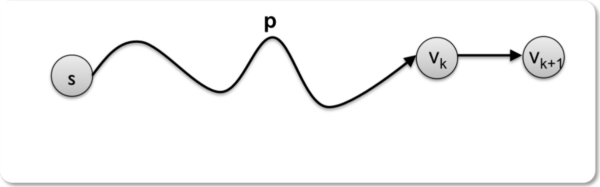
Only guaranteed to work if no negative cycles!

As we shall see later, it can also be used to detect negative cycles

Optimal substructure

If $\langle s, v_1, \ldots, v_k, v_{k+1} \rangle$ is a shortest path from s to v_{k+1}

Then $\langle s, v_1, \ldots, v_k \rangle$ is a shortest path from s to v_k

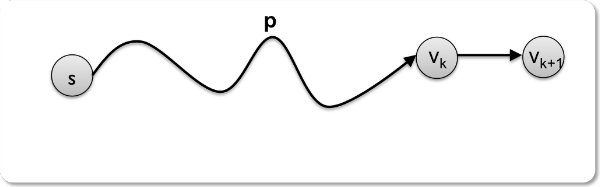


Optimal substructure

If $\langle s, v_1, \ldots, v_k, v_{k+1} \rangle$ is a shortest path from s to v_{k+1}

Then $\langle s, v_1, \ldots, v_k \rangle$ is a shortest path from s to v_k

Proof:



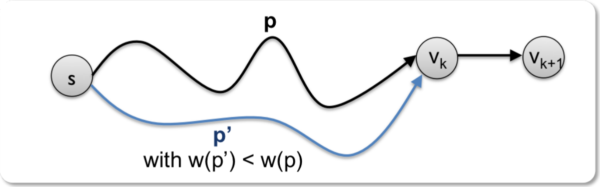
Optimal substructure

If $\langle s, v_1, \ldots, v_k, v_{k+1} \rangle$ is a shortest path from s to v_{k+1}

Then $\langle s, v_1, \ldots, v_k \rangle$ is a shortest path from s to v_k

Proof:

- Suppose toward contradiction: there exists shorter path p' from s to v_k



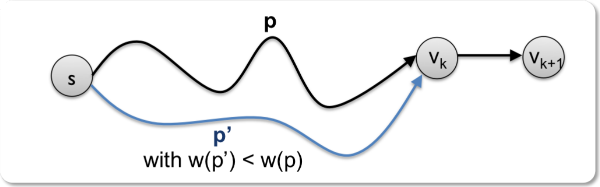
Optimal substructure

If $\langle s, v_1, \ldots, v_k, v_{k+1} \rangle$ is a shortest path from s to v_{k+1}

Then $\langle s, v_1, \ldots, v_k \rangle$ is a shortest path from s to v_k

Proof:

- Suppose toward contradiction: there exists shorter path p' from s to v_k
- Then weight of $p' + (v_k, v_{k+1})$ is smaller than $p + (v_k, v_{k+1})$ which contradicts that $p + (v_k, v_{k+1})$ was a shortest path from s to v_{k+1}



Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof by induction:

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof by induction:

Base case trivial: when 0 iterations $\ell(s) = 0$ and all other equal to infinity

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof by induction:

Base case trivial: when 0 iterations $\ell(s) = 0$ and all other equal to infinity

Inductive step: consider any shortest path from s to v_{k+1} using at most i edges

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof by induction:

Base case trivial: when 0 iterations $\ell(s) = 0$ and all other equal to infinity

Inductive step: consider any shortest path from s to v_{k+1} using at most i edges

- The path p from s to v_{k+1}'s predecessor v_k is the shortest path using at most $i - 1$ edges (by optimal substructure)

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof by induction:

Base case trivial: when 0 iterations $\ell(s) = 0$ and all other equal to infinity

Inductive step: consider any shortest path from s to v_{k+1} using at most i edges

- The path p from s to v_{k+1}'s predecessor v_k is the shortest path using at most $i - 1$ edges (by optimal substructure)
- By induction hypothesis $\ell(v_k) \leq w(p)$ after previous iteration

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof by induction:

Base case trivial: when 0 iterations $\ell(s) = 0$ and all other equal to infinity

Inductive step: consider any shortest path from s to v_{k+1} using at most i edges

- The path p from s to v_{k+1}'s predecessor v_k is the shortest path using at most $i-1$ edges (by optimal substructure)
- By induction hypothesis $\ell(v_k) \leq w(p)$ after previous iteration
- Hence, $\ell(v_{k+1}) \leq \ell(v_k) + w(v_k, v_{k+1}) \leq$ "length of shortest path from s to v_{k+1} using at most i edges" in the i-th iteration

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

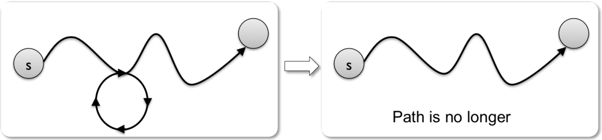
If there are no negative cycles reachable from s, then for any v there is a shortest path from s to v using at most $n - 1$ edges

Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

If there are no negative cycles reachable from s, then for any v there is a shortest path from s to v using at most $n-1$ edges

Proof: If there is a path with n or more edges, then there is a cycle and since its weight is non-negative it can be removed

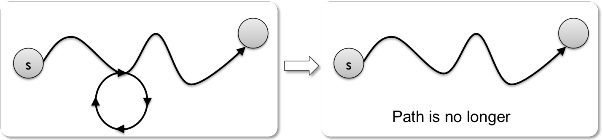


Proof of correctness

Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

If there are no negative cycles reachable from s, then for any v there is a shortest path from s to v using at most $n-1$ edges

Proof: If there is a path with n or more edges, then there is a cycle and since its weight is non-negative it can be removed



Therefore, Bellman-Ford will return correct answer if no negative cycles after $n-1$ iterations

Detecting negative cycles

There is a negative cycle reachable from the source if and only if the ℓ-value of at least one node changes if we run one more (n:th) iteration of Bellman-Ford

Detecting negative cycles

There is a negative cycle reachable from the source if and only if the ℓ-value of at least one node changes if we run one more (n:th) iteration of Bellman-Ford

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

Detecting negative cycles

There is no negative cycle reachable from the source if and only if the ℓ-value of no node changes if we run one more (n:th) iteration of Bellman-Ford

Detecting negative cycles

There is no negative cycle reachable from the source if and only if the ℓ-value of no node changes if we run one more (n:th) iteration of Bellman-Ford

From the correctness proof, we have that if there are no negative cycles reachable from the source, then the ℓ values don't change in n:th iteration.

We need to prove: If the ℓ-value of the vertices don't change in the n:th iteration, then there is no negative cycle that is reachable from the source

Proof.

Detecting negative cycles

There is no negative cycle reachable from the source if and only if the ℓ-value of no node changes if we run one more (n:th) iteration of Bellman-Ford

From the correctness proof, we have that if there are no negative cycles reachable from the source, then the ℓ values don't change in n:th iteration.

We need to prove: If the ℓ-value of the vertices don't change in the n:th iteration, then there is no negative cycle that is reachable from the source

Proof. In this case $\forall(u, v) \in E : \ell(u) + w(u, v) \geq \ell(v)$.

Detecting negative cycles

There is no negative cycle reachable from the source if and only if the ℓ-value of no node changes if we run one more (n:th) iteration of Bellman-Ford

From the correctness proof, we have that if there are no negative cycles reachable from the source, then the ℓ values don't change in n:th iteration.

We need to prove: If the ℓ-value of the vertices don't change in the n:th iteration, then there is no negative cycle that is reachable from the source

Proof. In this case $\forall (u,v) \in E : \ell(u) + w(u,v) \geq \ell(v)$.

So for a cycle $v_0 - v_1 - \cdots - v_{t-1} - v_t = v_0$,

$$\sum_{i=1}^{t} \ell(v_i) \leq \sum_{i=1}^{t} (\ell(v_{i-1}) + w(v_{i-1}, v_i)) = \sum_{i=1}^{t} \ell(v_{i-1}) + \sum_{i=1}^{t} w(v_{i-1}, v_i)$$

Detecting negative cycles

There is no negative cycle reachable from the source if and only if the ℓ-value of no node changes if we run one more (n:th) iteration of Bellman-Ford

From the correctness proof, we have that if there are no negative cycles reachable from the source, then the ℓ values don't change in n:th iteration.

We need to prove: If the ℓ-value of the vertices don't change in the n:th iteration, then there is no negative cycle that is reachable from the source

Proof. In this case $\forall (u, v) \in E : \ell(u) + w(u, v) \geq \ell(v)$.

So for a cycle $v_0 - v_1 - \cdots - v_{t-1} - v_t = v_0$,

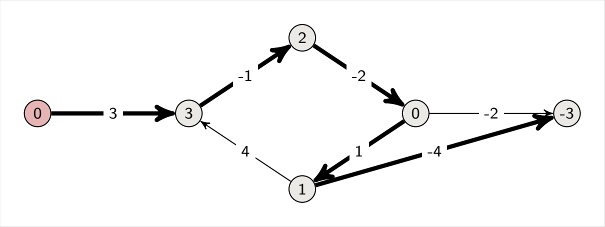
$$\sum_{i=1}^{t} \ell(v_i) \leq \sum_{i=1}^{t} (\ell(v_{i-1}) + w(v_{i-1}, v_i)) = \sum_{i=1}^{t} \ell(v_{i-1}) + \sum_{i=1}^{t} w(v_{i-1}, v_i)$$

Red sums are the same, hence the cycle is non-negative $0 \leq \sum_{i=1}^{t} w(v_{i-1}, v_i)$

Example 1

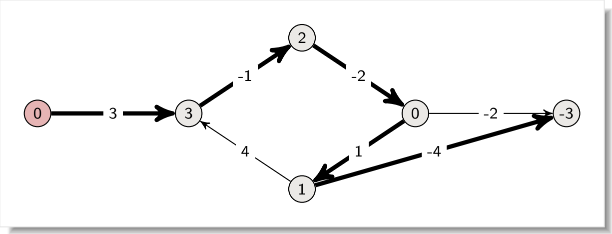
BELLMAN-FORD(G, w, s)

INIT-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



Example 1

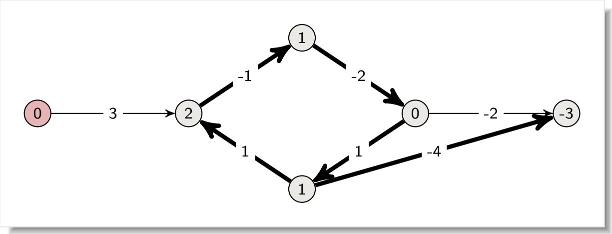
$$
\begin{array}{l}
\textsc{Bellman-Ford}(G, w, s) \\
\quad \textsc{Init-Single-Source}(G, s) \\
\quad \textbf{for } i = 1 \text{ to } |G.V| - 1 \\
\qquad \textbf{for} \text{ each edge } (u, v) \in G.E \\
\qquad\quad \textsc{Relax}(u, v, w) \\
\quad \textbf{for} \text{ each edge } (u, v) \in G.E \\
\qquad \textbf{if } v.d > u.d + w(u, v) \\
\qquad\quad \textbf{return } \text{FALSE} \\
\quad \textbf{return } \text{TRUE}
\end{array}
$$



No negative cycles, BF returns TRUE

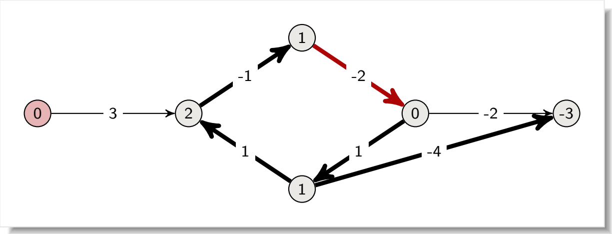
Example 2

```
BELLMAN-FORD(G, w, s)
  INIT-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
```



Example 2

BELLMAN-FORD(G, w, s)
  INIT-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



Negative cycles, BF returns FALSE

Runtime analysis

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

- INIT-SINGLE-SOURCE updates ℓ, π for each vertex in time

Runtime analysis

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

- INIT-SINGLE-SOURCE updates ℓ, π for each vertex in time $\Theta(V)$

Runtime analysis

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

- INIT-SINGLE-SOURCE updates ℓ, π for each vertex in time $\Theta(V)$
- Nested **for** loops runs RELAX $V - 1$ times for each edge. Hence total time for these loops is

Runtime analysis

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

- INIT-SINGLE-SOURCE updates ℓ, π for each vertex in time $\Theta(V)$
- Nested **for** loops runs RELAX $V - 1$ times for each edge. Hence total time for these loops is $\Theta(E \cdot V)$

Runtime analysis

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

- INIT-SINGLE-SOURCE updates ℓ, π for each vertex in time $\Theta(V)$
- Nested **for** loops runs RELAX $V - 1$ times for each edge. Hence total time for these loops is $\Theta(E \cdot V)$
- Final **for** loop runs once for each edge. Time is

Runtime analysis

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

- INIT-SINGLE-SOURCE updates ℓ, π for each vertex in time $\Theta(V)$
- Nested **for** loops runs RELAX $V - 1$ times for each edge. Hence total time for these loops is $\Theta(E \cdot V)$
- Final **for** loop runs once for each edge. Time is $\Theta(E)$

Runtime analysis

```
BELLMAN-FORD(G, w, s)
  INIT-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
```

- INIT-SINGLE-SOURCE updates ℓ, π for each vertex in time $\Theta(V)$
- Nested **for** loops runs RELAX $V - 1$ times for each edge. Hence total time for these loops is $\Theta(E \cdot V)$
- Final **for** loop runs once for each edge. Time is $\Theta(E)$

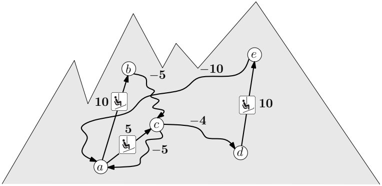
Total time: $\Theta(E \cdot V)$

Final comments on Bellman-Ford

- Can be used to find negative cycles
  - Run for n-iterations and detect cycles in "shortest path tree" these will correspond to negative cycles
- Easy to implement in distributed settings: each vertex repeatedly ask their neighbors for the best path
  - Good for routing and dynamic networks

Problem solving (20 pts * exam question last year)

The famous alpine ski racer Lindsey Vonn has sent her assistant to measure the altitude differences of the ski lifts and slopes of a famous ski resort in Switzerland. The assistant returns after several days of hard work with a map of all ski lifts and all slopes together with the altitude difference between the start station and the end station of each lift and the altitude difference between the start station and end station of each slope. A slope starts from the end station of a lift and ends at the start station of a potentially different lift (see the figure below).
Lindsey wants to verify the map of her assistant by performing the following sanity check: starting from any point A, no matter how we ski (using lifts and slopes), the altitude change when we return to A should be 0 (i.e., neither strictly negative nor strictly positive). Design an algorithm to perform this sanity check and analyze its running time in terms of the number of slopes and ski lifts ($|E|$) and the number of start and end stations ($|V|$). Your algorithm should run in polynomial time (in $|V|$ and $|E|$).



DIJKSTRA'S ALGORITHM

Dijkstra's algorithm

- Only works when all weights are nonnegative
- Greedy and faster than Bellman-Ford
- Similar idea to Prim's algorithm (essentially weighted version of BFS)

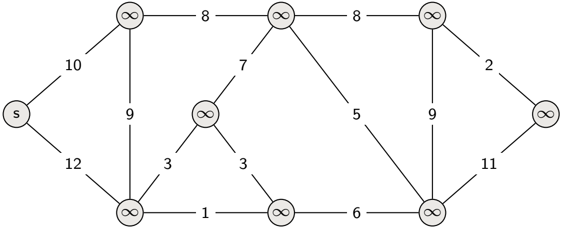
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



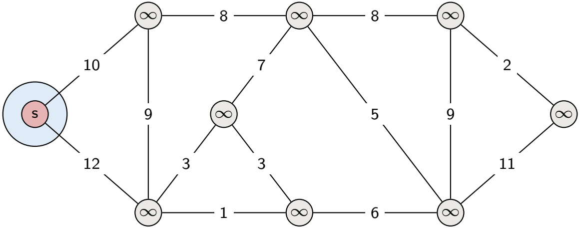
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



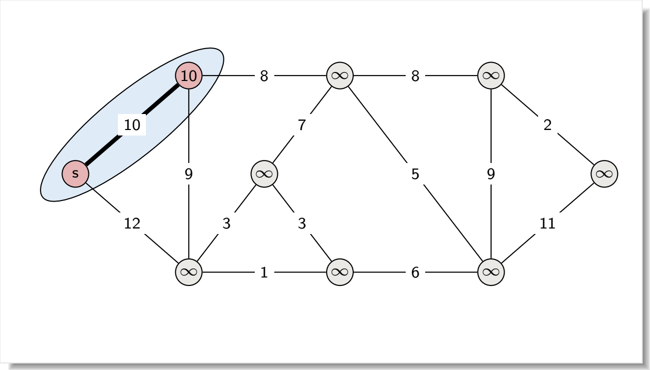
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



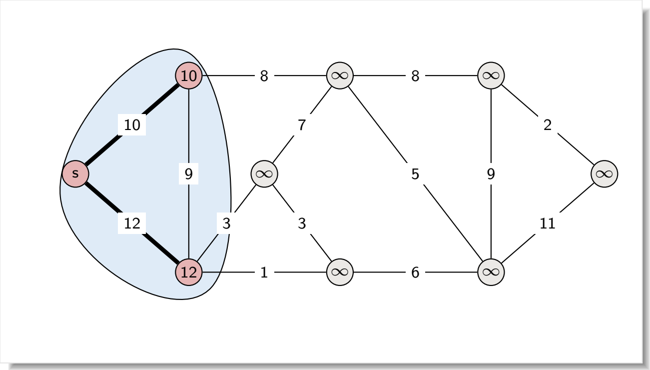
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



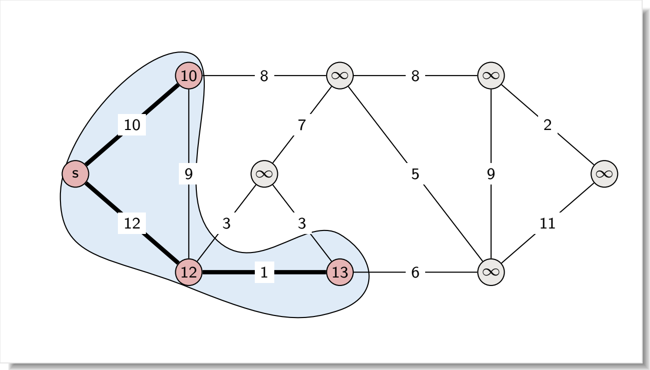
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S$, $u.d + w(u, v)$)



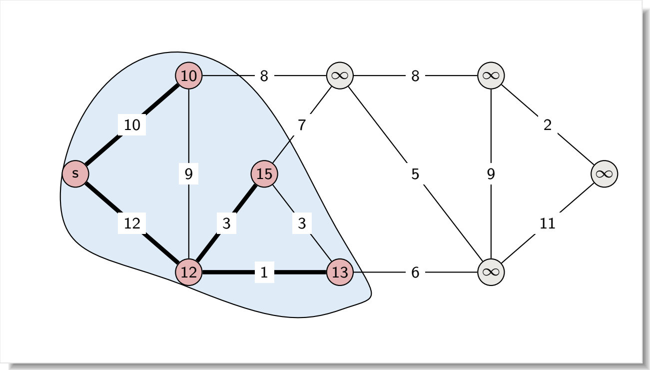
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S$, $u.d + w(u, v)$)



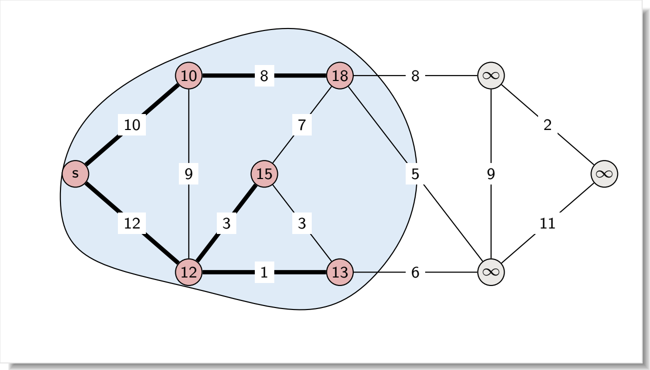
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



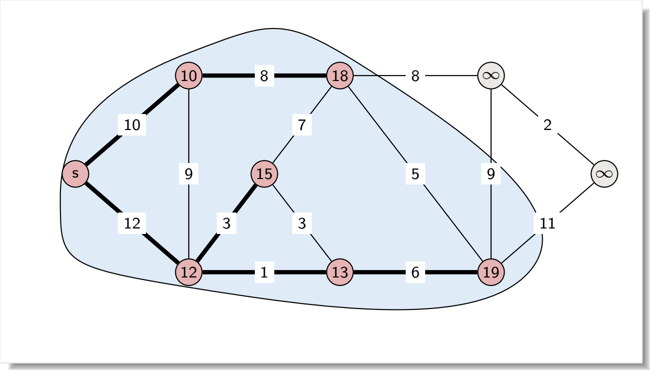
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



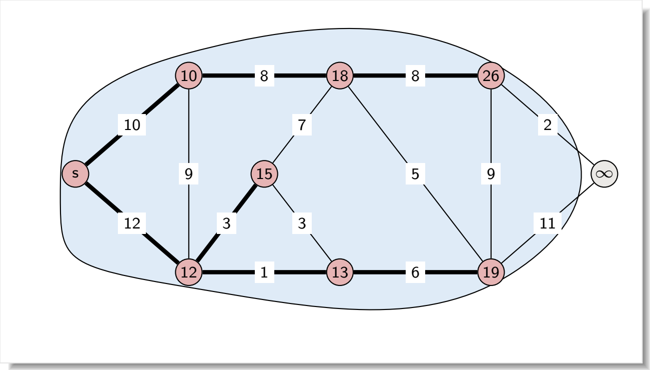
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



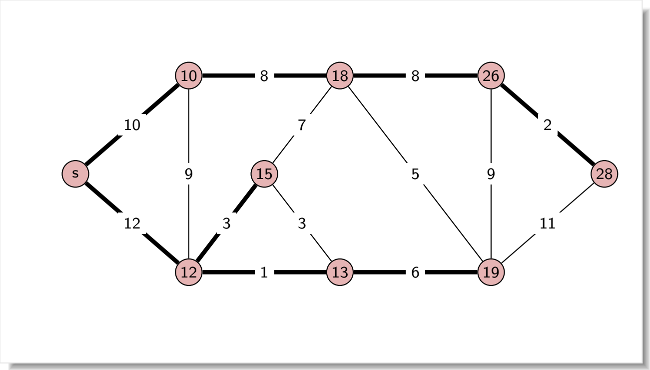
Dijkstra's algorithm

Start with source $S = \{s\}$

Greedily grow S:

at each step add to S the vertex that is closest to s

(minimum over $v \notin S$ of minimum over $u \in S, u.d + w(u, v)$)



Implementation and Running Time

Implementation with priority-queue as Prim's algorithm with shortest path keys:

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Implementation and Running Time

Implementation with priority-queue as Prim's algorithm with shortest path keys:

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Running time Like Prim's dominated by operations on priority queue:

Implementation and Running Time

Implementation with priority-queue as Prim's algorithm with shortest path keys:

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Running time Like Prim's dominated by operations on priority queue:

- If binary heap, each operation takes $O(\lg V)$ time $\Rightarrow O(E \lg V)$

Implementation and Running Time

Implementation with priority-queue as Prim's algorithm with shortest path keys:

```
DIJKSTRA(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  S = ∅
  Q = G.V          // i.e., insert all vertices into Q
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      S = S ∪ {u}
      for each vertex v ∈ G.Adj[u]
          RELAX(u, v, w)
```

Running time Like Prim's dominated by operations on priority queue:

- If binary heap, each operation takes $O(\lg V)$ time $\Rightarrow O(E \lg V)$
- More careful implementation time is $O(V \lg V + E)$

Problem Solving

Show that Dijkstra's Algorithm is correct by proving the following loop invariant:

"At the start of each iteration, we have for all $v \in S$ that the distance $v.d$ from s to v is equal to the shortest path from s to v"