Algorithms: MINIMUM SPANNING TREES, SHORTEST PATHS

Alessandro Chiesa, Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 20, 30.04.2025

Recall: Disjoint-set data structures

- Also known as "union find"
- Maintain collection $\mathcal{S} = \{S_1, \ldots, S_k\}$ of disjoint dynamic (changing over time) sets
- Each set is identified by a representative, which is some member of the set

  Doesn't matter which member is the representative, as long as if we ask for the representative twice without modifying the set, we get the same answer both times

Operations

MAKE-SET(x): make a new set $S_i = \{x\}$, and add S_i to $\mathcal{S}$

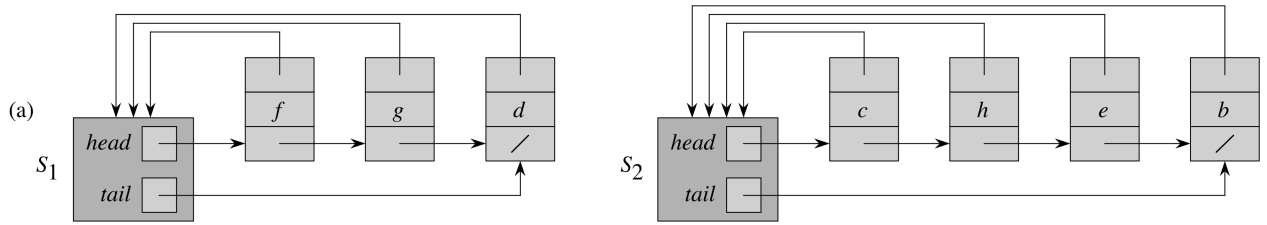
UNION(x, y): if $x \in S_x, y \in S_y$, then $\mathcal{S} = \mathcal{S} - S_x - S_y \cup \{S_x \cup S_y\}$

- Representative of new set is any member in $S_x \cup S_y$, often the representative of one of S_x and S_y
- Destroys S_x and S_y (since sets must be disjoint)

FIND(x): return representative of set containing x

List representation

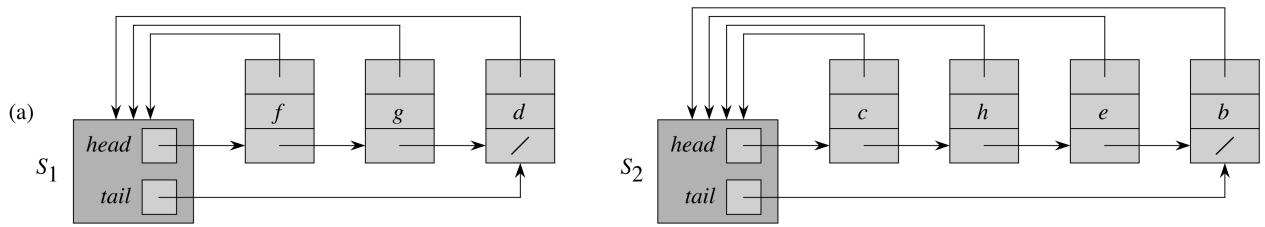
- Each set is a single linked list represented by a set object that has
  - a pointer to the *head* of the list (assumed to be the representative
  - a pointer to the *tail* of the list
- Each object in the list has attributes for the *set member*, *pointer to the set object* and *next*



Make-Set and Find

MAKE-SET(x): Create a single ton list in time $\Theta(1)$

FIND(x): follow the pointer back to the list object, and then follow the *head* pointer to the representative (time $\Theta(1)$)



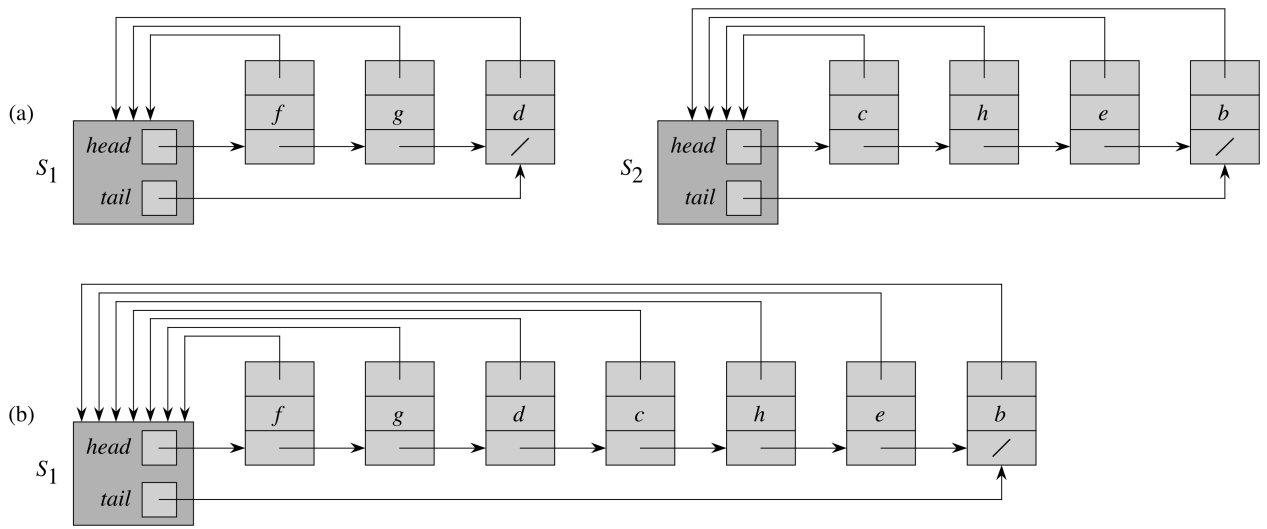
Union

A couple of ways of doing it

Union

A couple of ways of doing it

1. Append y's list onto the end of x's list. Use x's tail pointer to find the end.
   - Need to update the pointer back to the set object for every node on y's list.



Union

A couple of ways of doing it

1. Append y's list onto the end of x's list. Use x's tail pointer to find the end.
   - Need to update the pointer back to the set object for every node on y's list.
   - If appending a large list onto a small list, it can take a while

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$

Union

A couple of ways of doing it

2. **Weighted-union heuristic** Always append the smaller list to the larger list (break ties arbitrarily)

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \log n)$ time.

Weighted-union heuristic

Theorem

With weighted-union heuristic, a sequence of m operations on n elements take $O(m + n \lg n)$ time.

Proof sketch MAKE-SET and FIND still takes constant time. How many times can each objects' representative pointer be updated? It must be in the smaller set each time

times updated	size of resulting set
1	≥ 2
2	≥ 4
3	≥ 8
$\vdots$	$\vdots$
k	$\geq 2^k$
$\vdots$	$\vdots$
$\log n$	$\geq n$

Therefore, each representative is updated $\leq \log n$ times

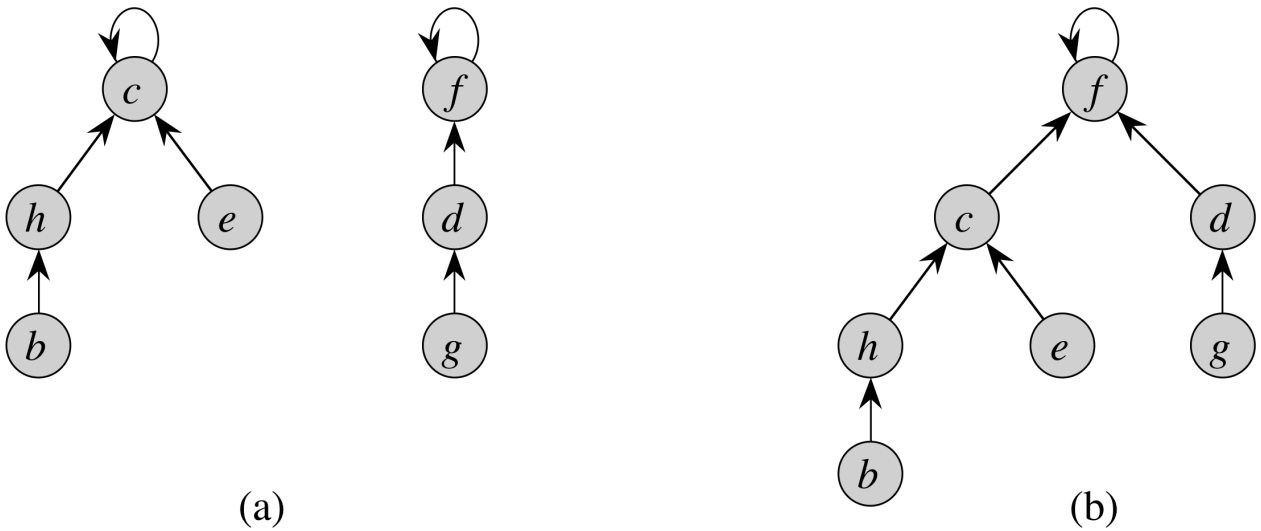
Forest of trees

- One tree per set. Root is representative
- Each node only points to its parent

MAKE-SET(x): Make a single-node tree

FIND(x): follow pointers to the root

UNION(x, y): make one root a child of another

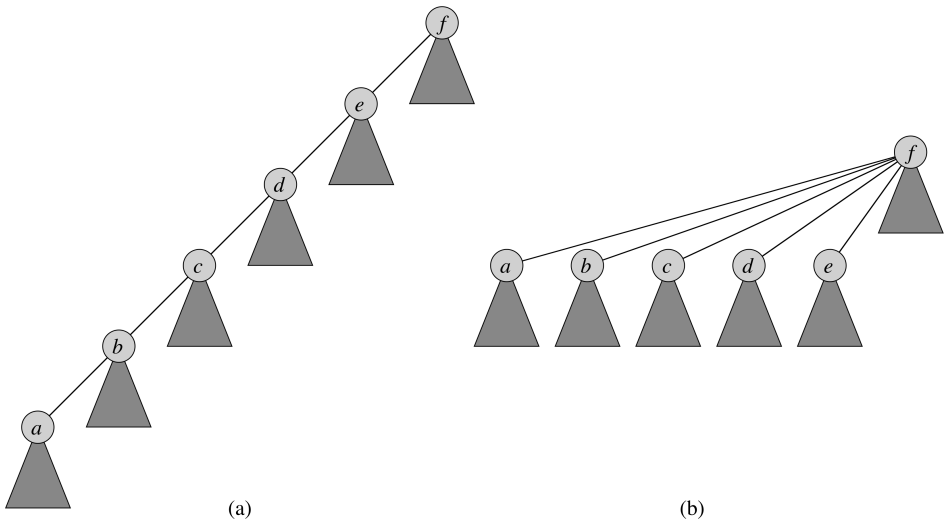


Great heuristics

Union by rank: make the root of the smaller tree a child of the root of the larger tree

- Don't actually use size
- Use rank, which is an upper bound on height of node
- Make the root with the smaller rank a child of the root with the larger rank

Path compression: Find path = nodes visited during FIND on the trip to the root, make all nodes on the find path direct children to root.



Running time

If use both union by rank and path compression,

$$O(m \cdot \alpha(n))$$

where $\alpha(n)$ is an extremely slowly growing function:

n	$\alpha(n)$
$0-2$	0
3	1
$4-7$	2
$8-2047$	3
$2047- \gg 10^8$	4

- $\alpha(n) \leq 5$ for any practical purpose
- The bound $O(m \cdot \alpha(n))$ is tight

MINIMUM SPANNING TREES

Lecture 20, 30.04.2025

Origin of today's lecture

Otakar Boruvka (1926)

- Electrical power company in western Moravia in Brno
- Most economical construction of electrical power network
- Concrete engineering problem led to what is now a cornerstone problem-solving model in combinatorial optimization

A spanning tree of a graph

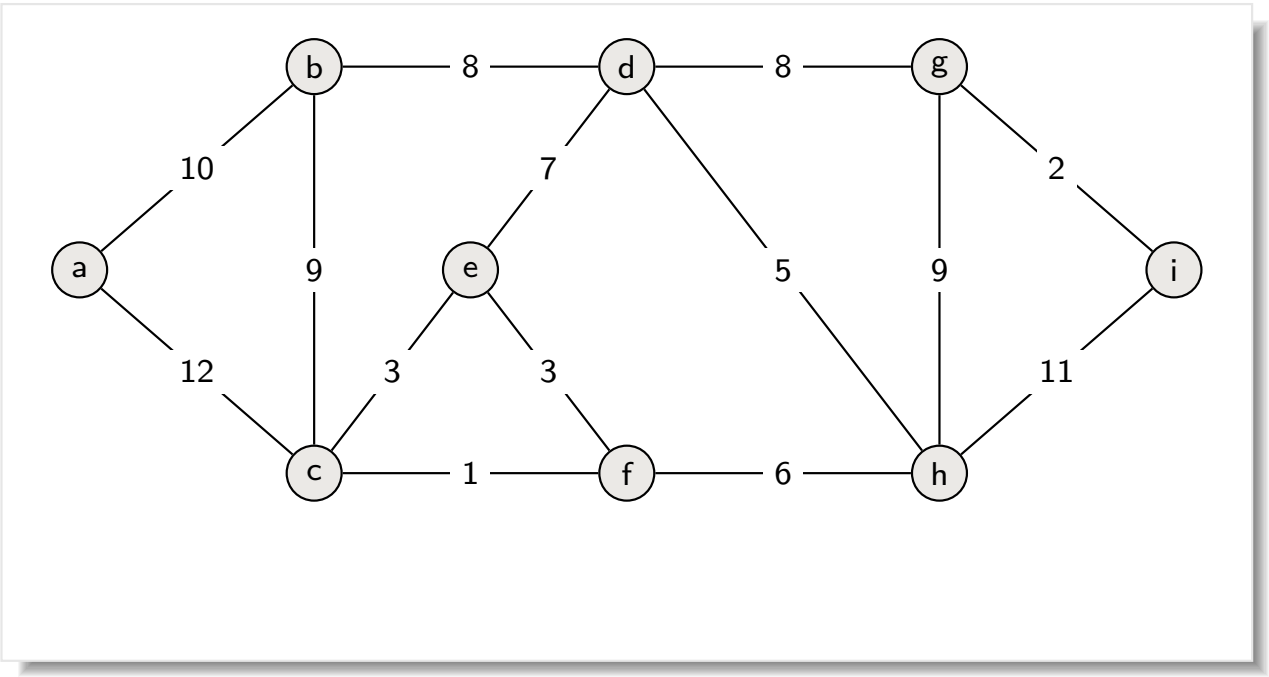
A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

A spanning tree of a graph

A set **T** of edges that is

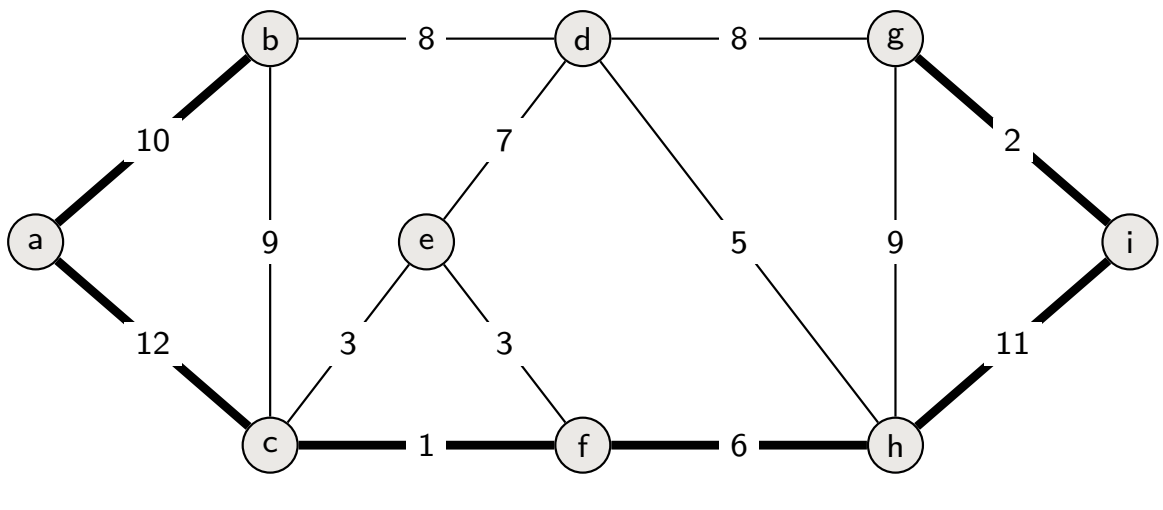
- Acyclic
- Spanning (connects all vertices)



A spanning tree of a graph

A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

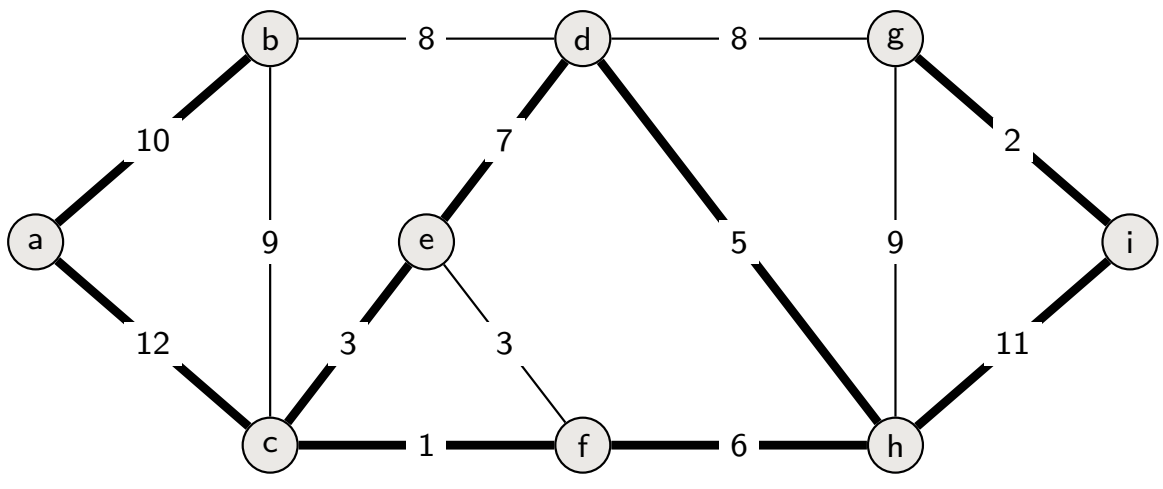


Acyclic, but not connected

A spanning tree of a graph

A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

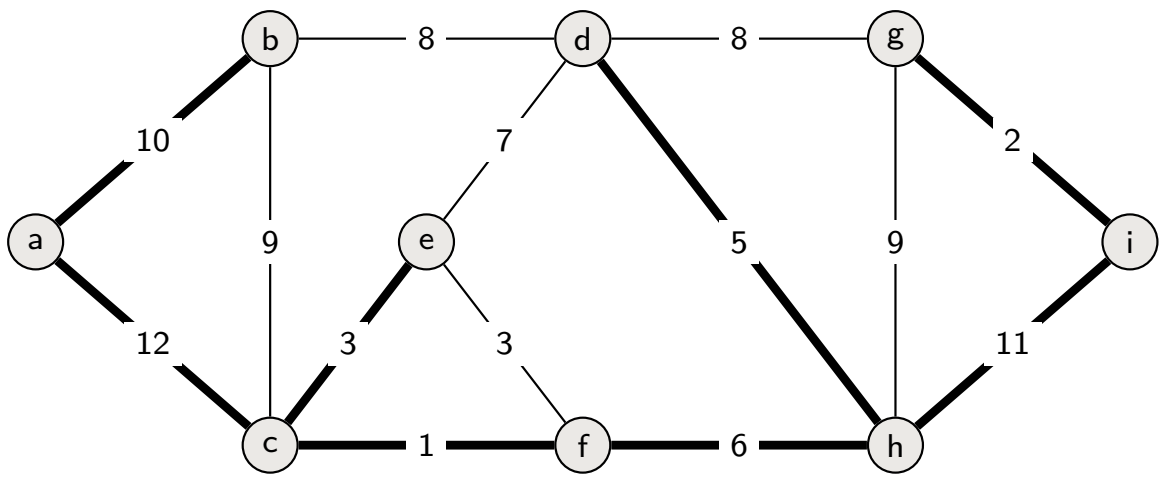


Connected, but not acyclic

A spanning tree of a graph

A set **T** of edges that is

- Acyclic
- Spanning (connects all vertices)

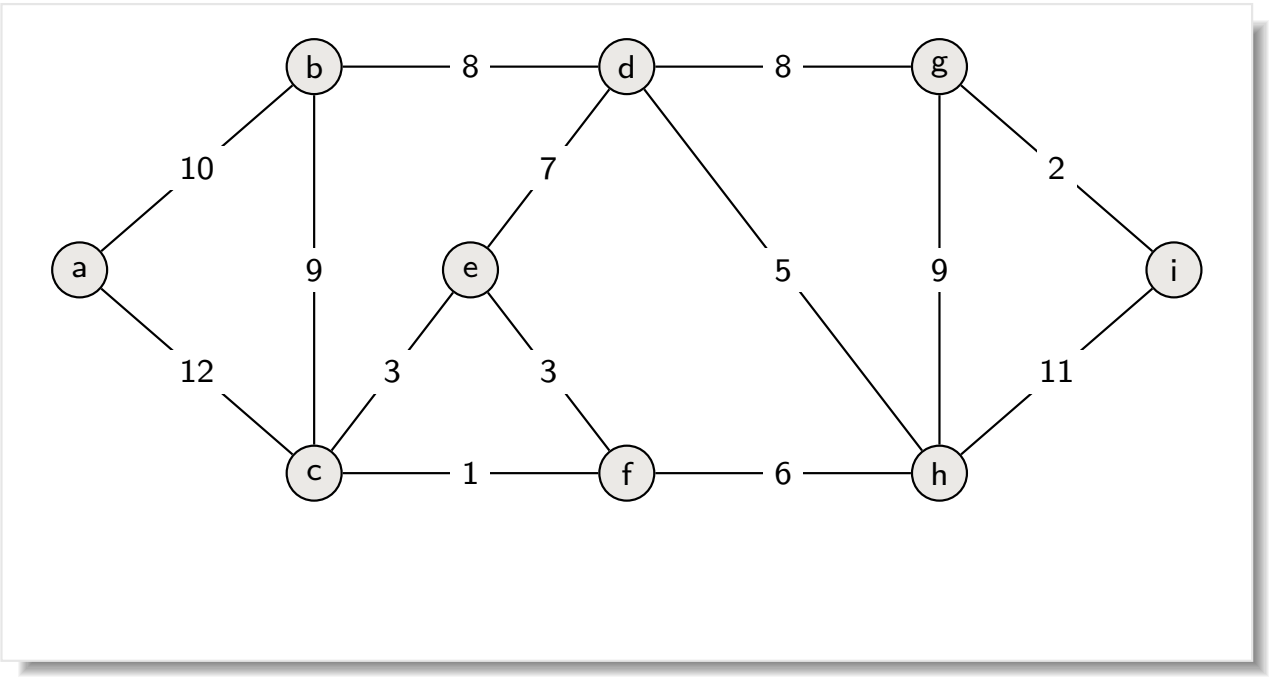


Acyclic and connected = spanning tree

Minimum spanning tree (MST)

INPUT: an undirected graph $G = (V, E)$ with weight $w(u, v)$ for each edge $(u, v) \in E$

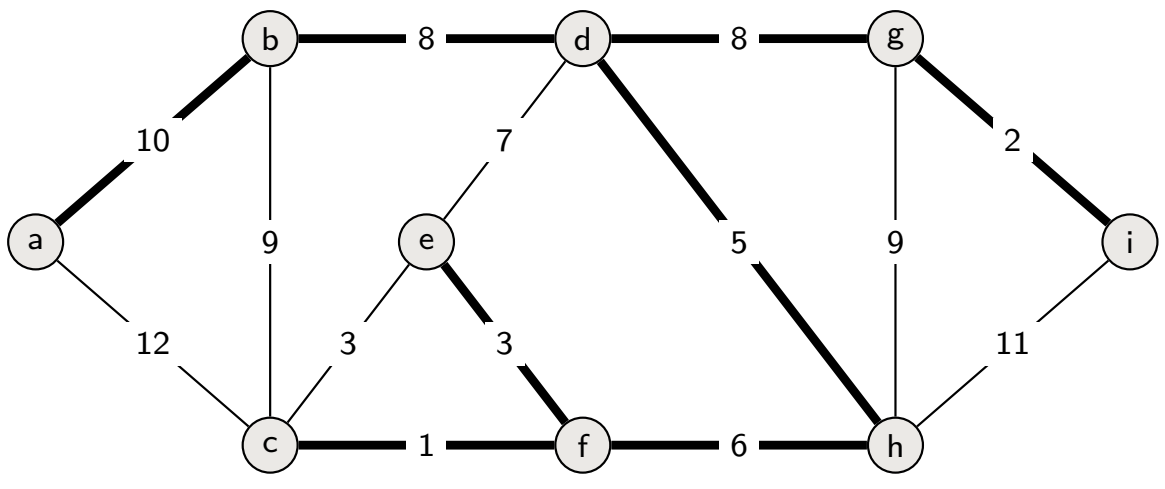
OUTPUT: a spanning tree of minimum total weight



Minimum spanning tree (MST)

INPUT: an undirected graph $G = (V, E)$ with weight $w(u, v)$ for each edge $(u, v) \in E$

OUTPUT: a spanning tree of minimum total weight



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

EXAMPLE APPLICATIONS

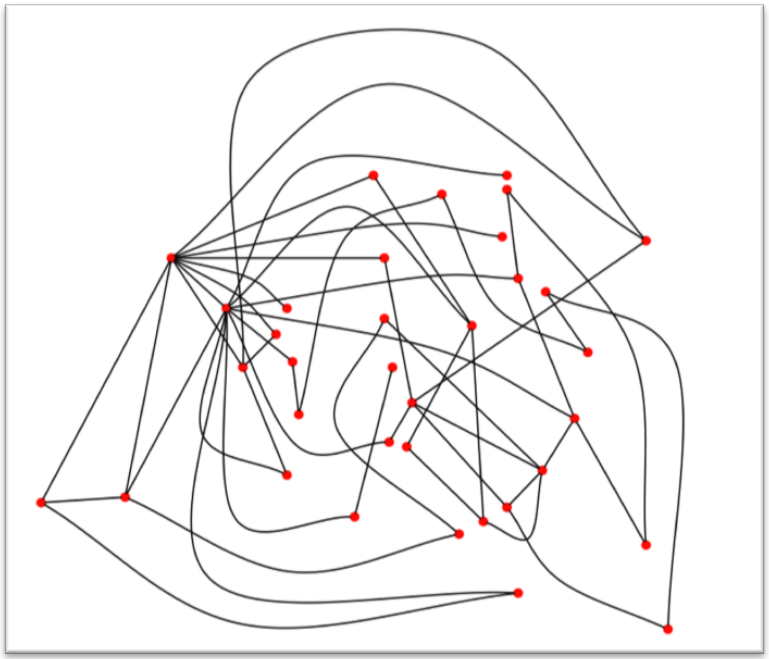
Example 1: Communication networks

A multinational company wants to lease communication lines between its various locations

Example 1: Communication networks

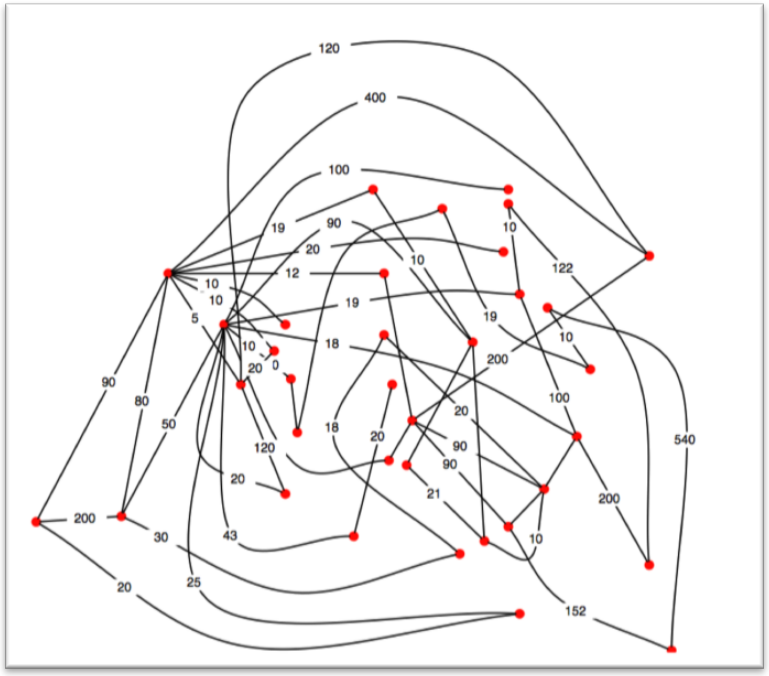
A multinational company wants to lease communication lines between its various locations

Example 1: Communication networks



A multinational company wants to lease communication lines between its various locations

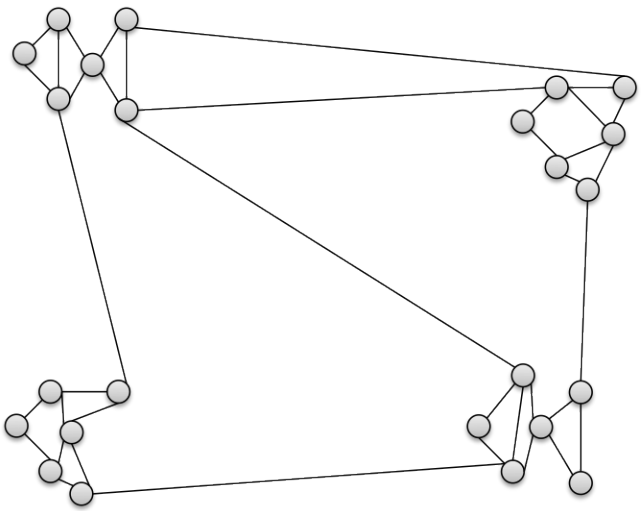
Example 1: Communication networks



A multinational company wants to lease communication lines between its various locations

Solution given by a MST on the graph

Example 2: Clustering

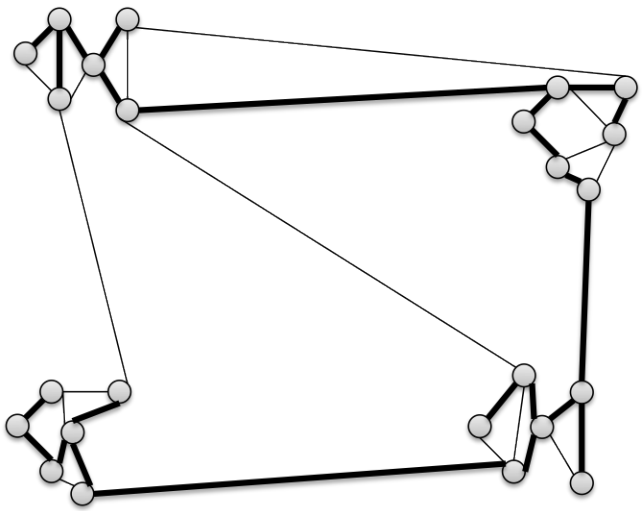


Edge weights equal to distance of nodes

Find: "cluster" of nodes

Possible solution: Find MST. Eliminate "fat" edges

Example 2: Clustering



Edge weights equal to distance of nodes

Find: "cluster" of nodes

Possible solution: Find MST. Eliminate "fat" edges

Example 2: Clustering

Edge weights equal to distance of nodes

Find: "cluster" of nodes

Possible solution: Find MST. Eliminate "fat" edges

Example 2: Clustering

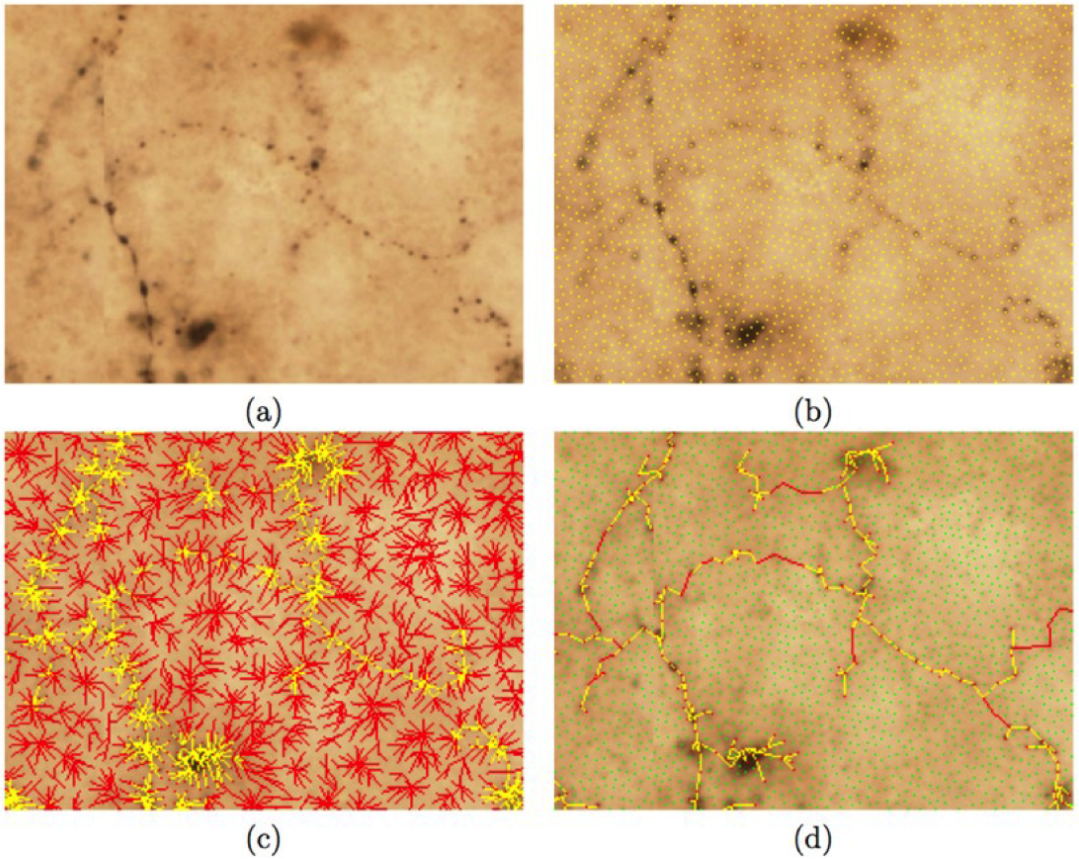
Edge weights equal to distance of nodes

Find: "cluster" of nodes

Possible solution: Find MST. Eliminate "fat" edges

Note: this is a "heuristic" algorithm. Needs analysis

Example 3: Dendritic structures in the brain



Example 4: Phylogenetic trees

Infer evolutionary relationships among various biological species

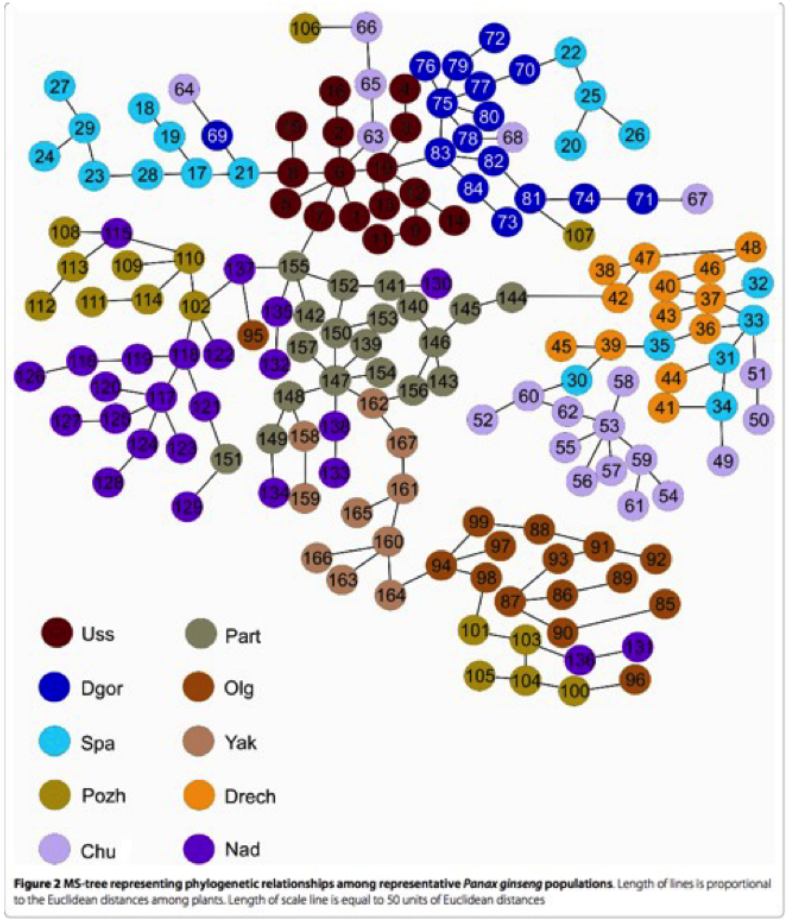
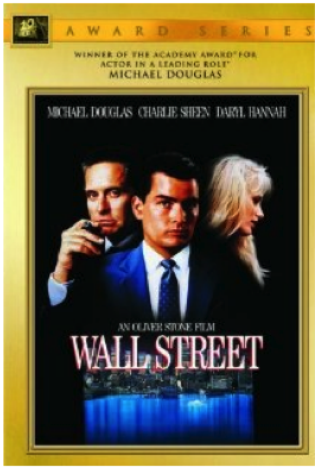


Figure 2 MS-tree representing phylogenetic relationships among representative *Panax ginseng* populations. Length of lines is proportional to the Euclidean distances among plants. Length of scale line is equal to 50 units of Euclidean distances

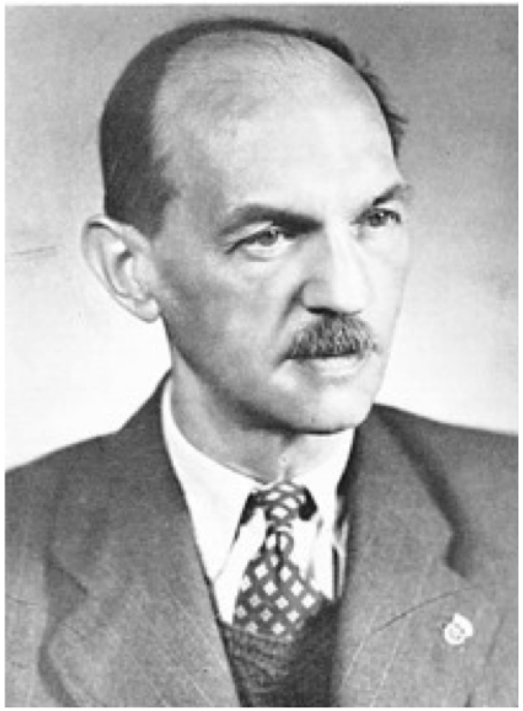


ALGORITHMS FOR MST

"Greed is good. Greed is right. Greed works. Greed clarifies, cuts through and captures the essence of the evolutionary spirit."

- Gordon Gecko

Prim's algorithm



Vojitech Jarnik
1897 - 1970

Robert Prim
1921-

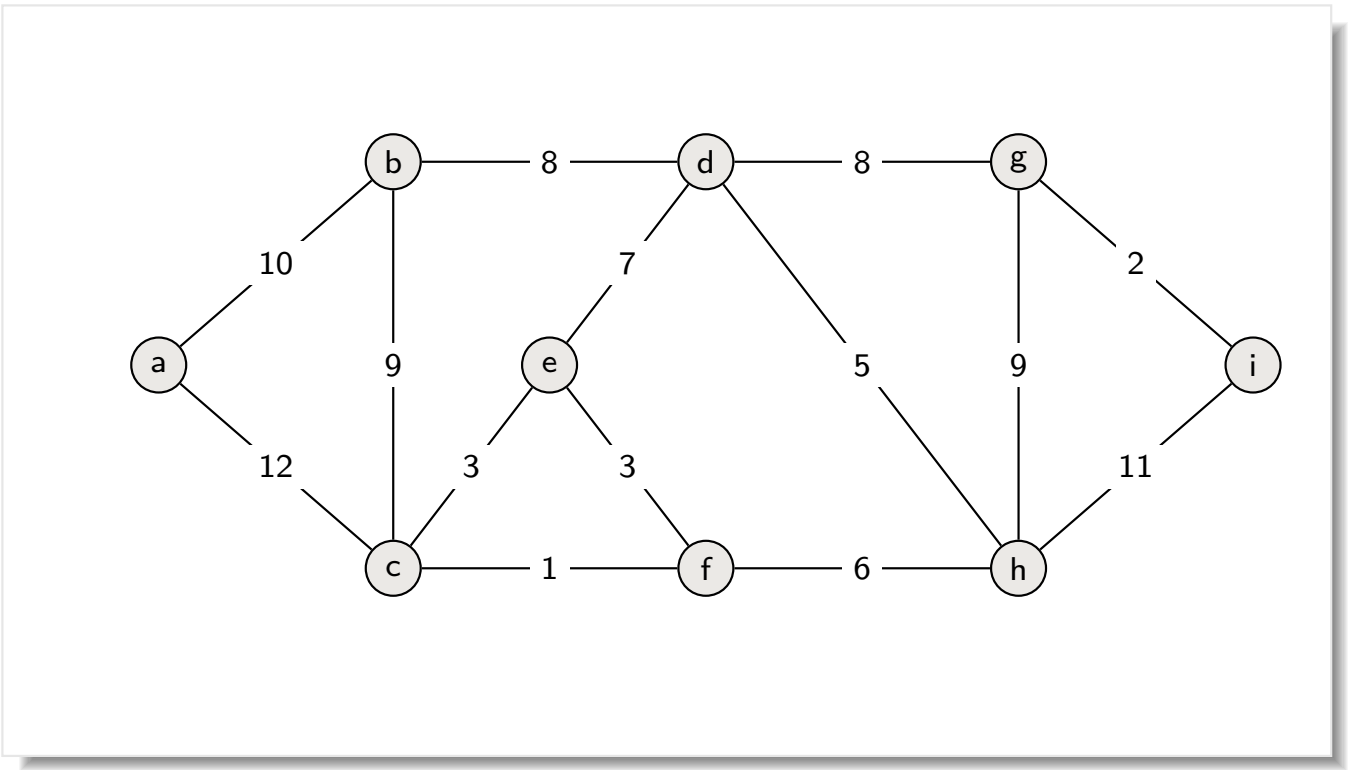
Edsger Dijkstra
1930 - 2002

Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

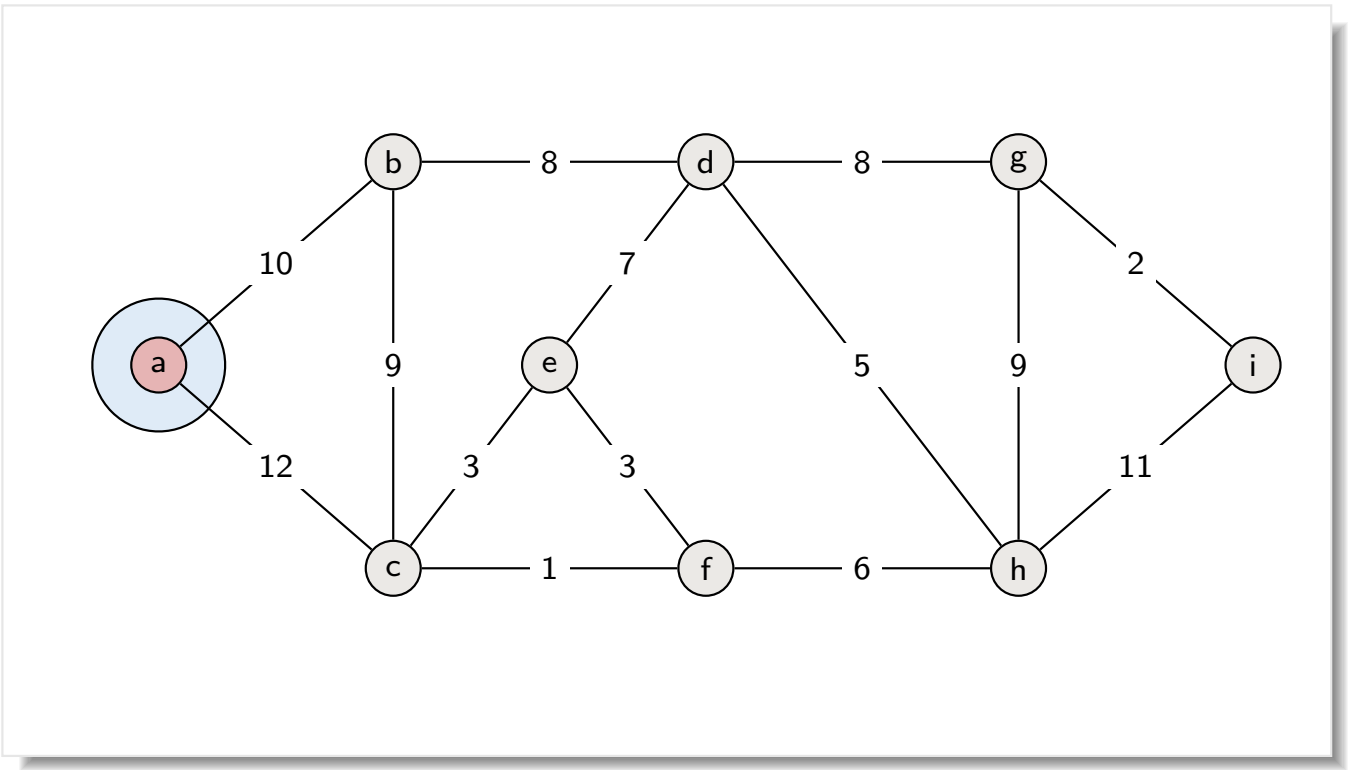


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

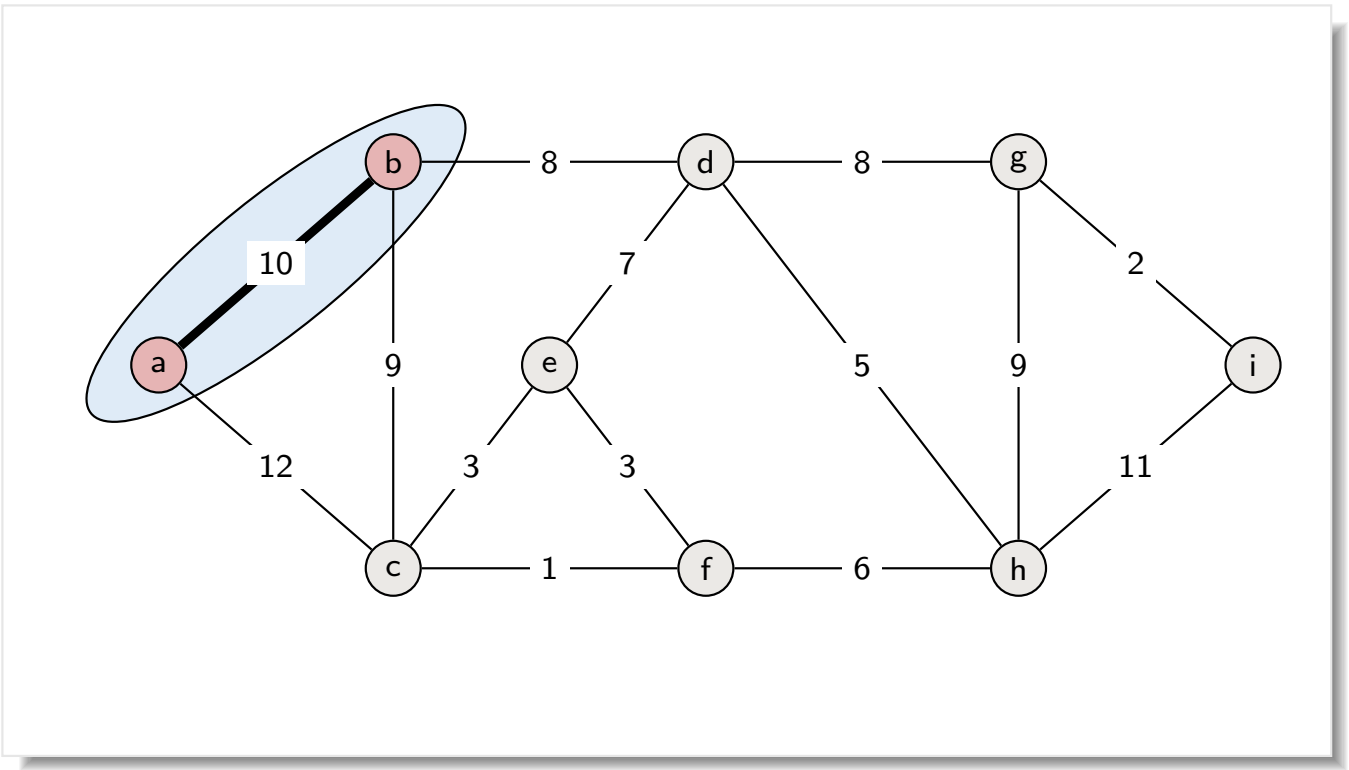


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

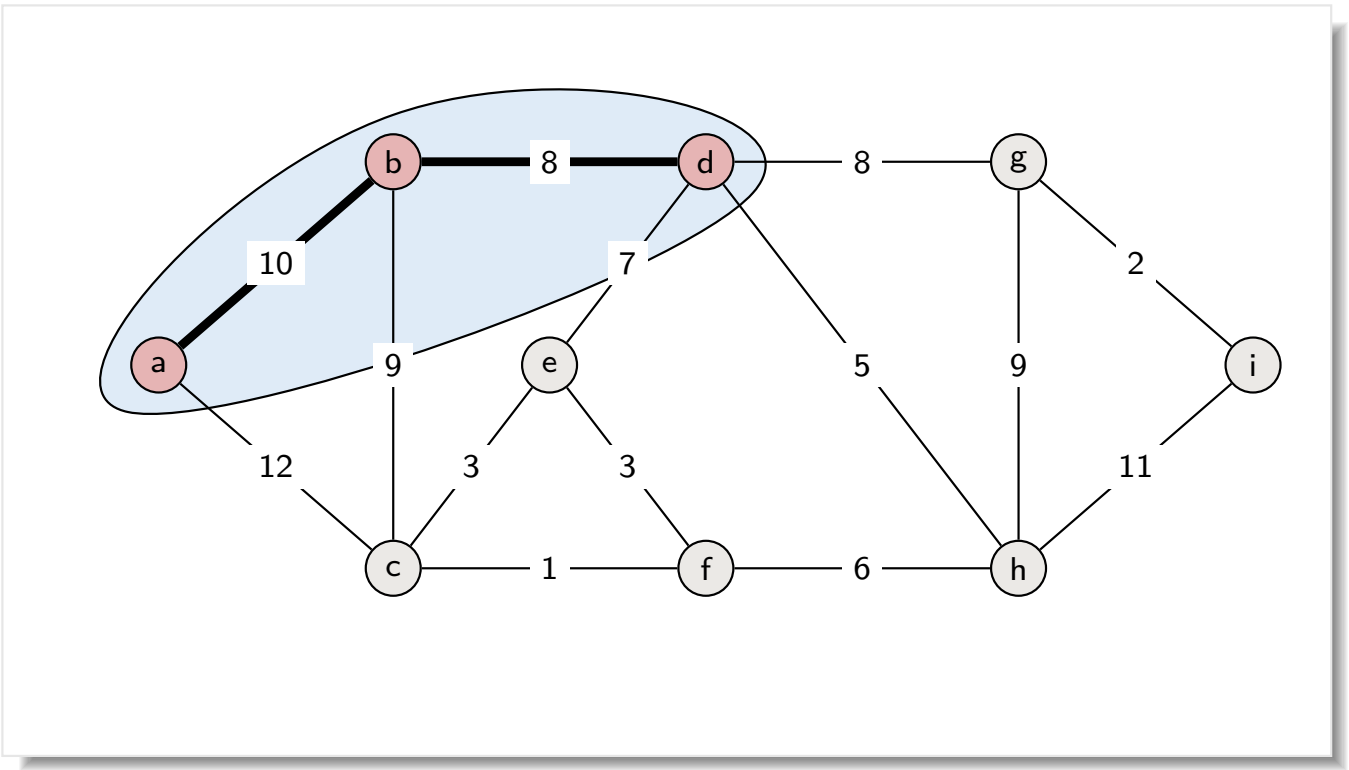


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

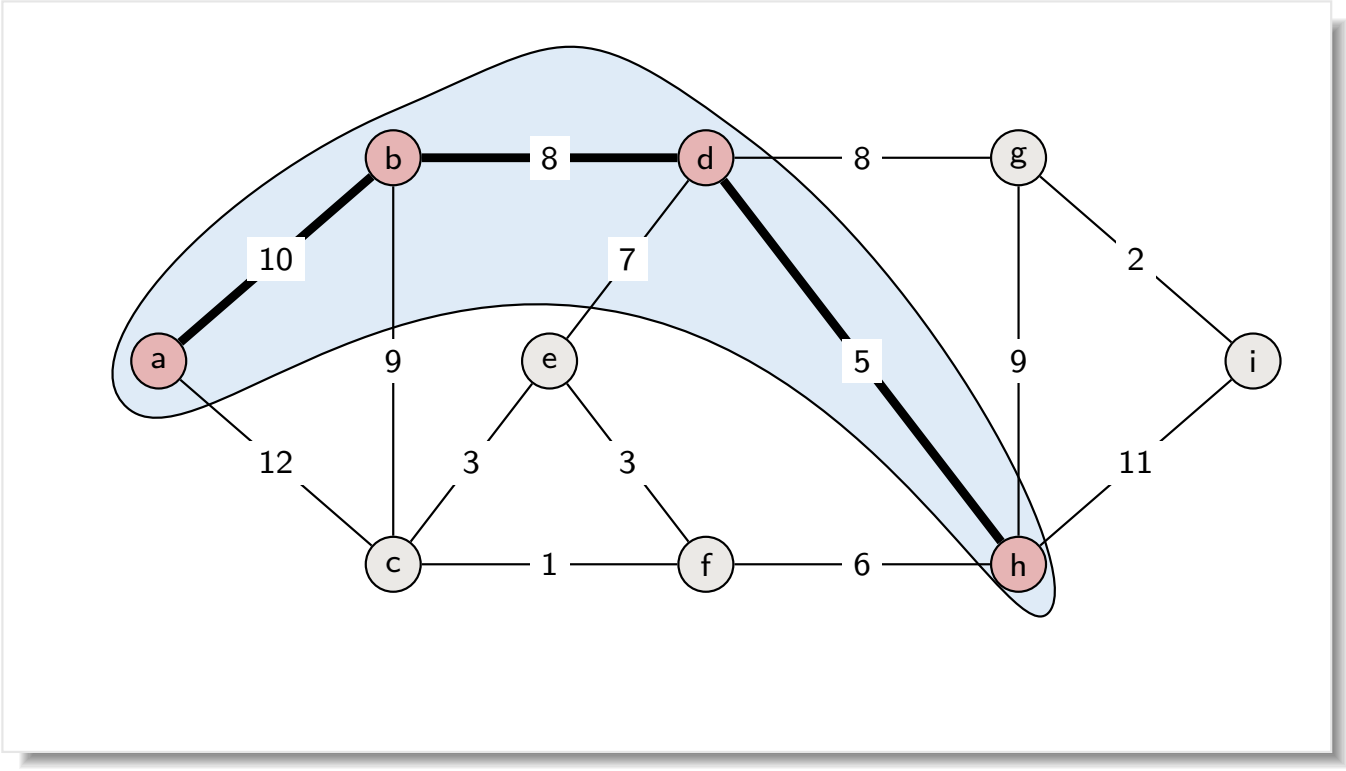


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

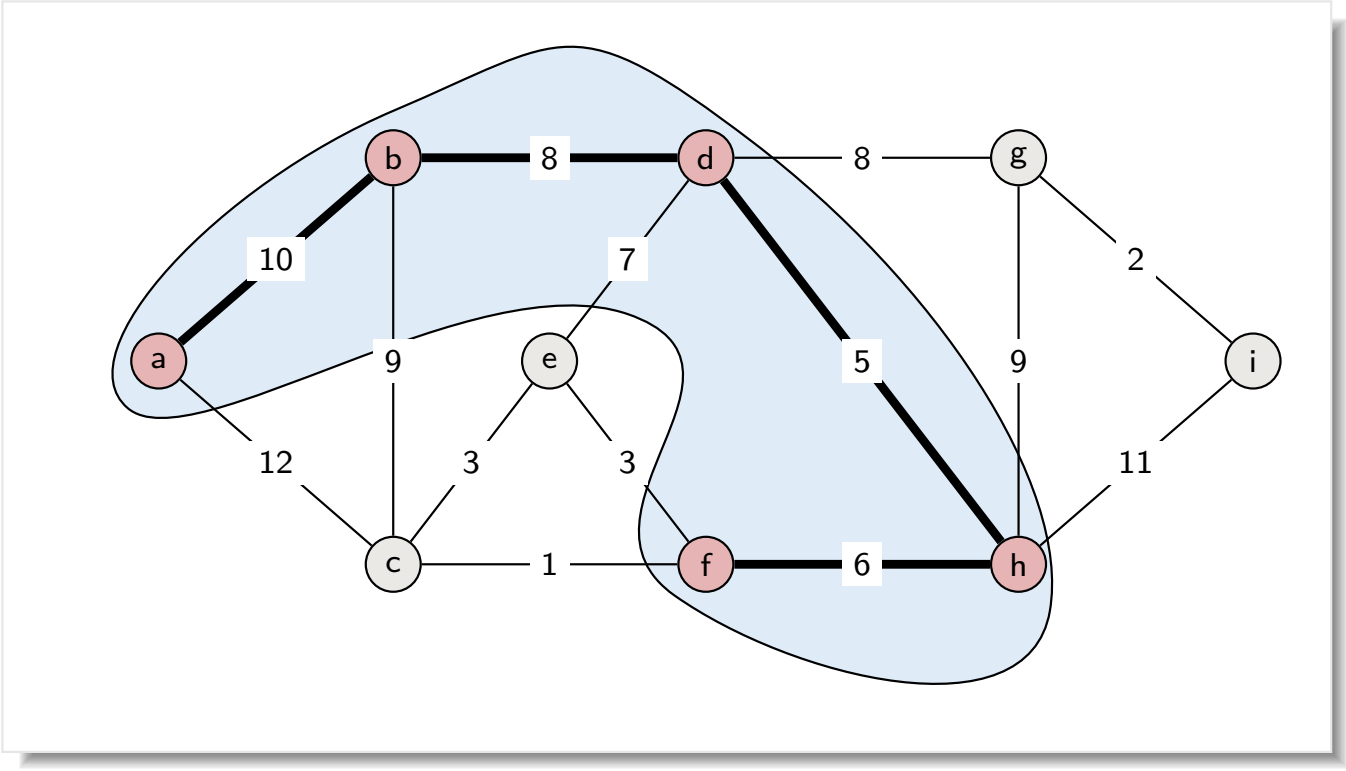


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

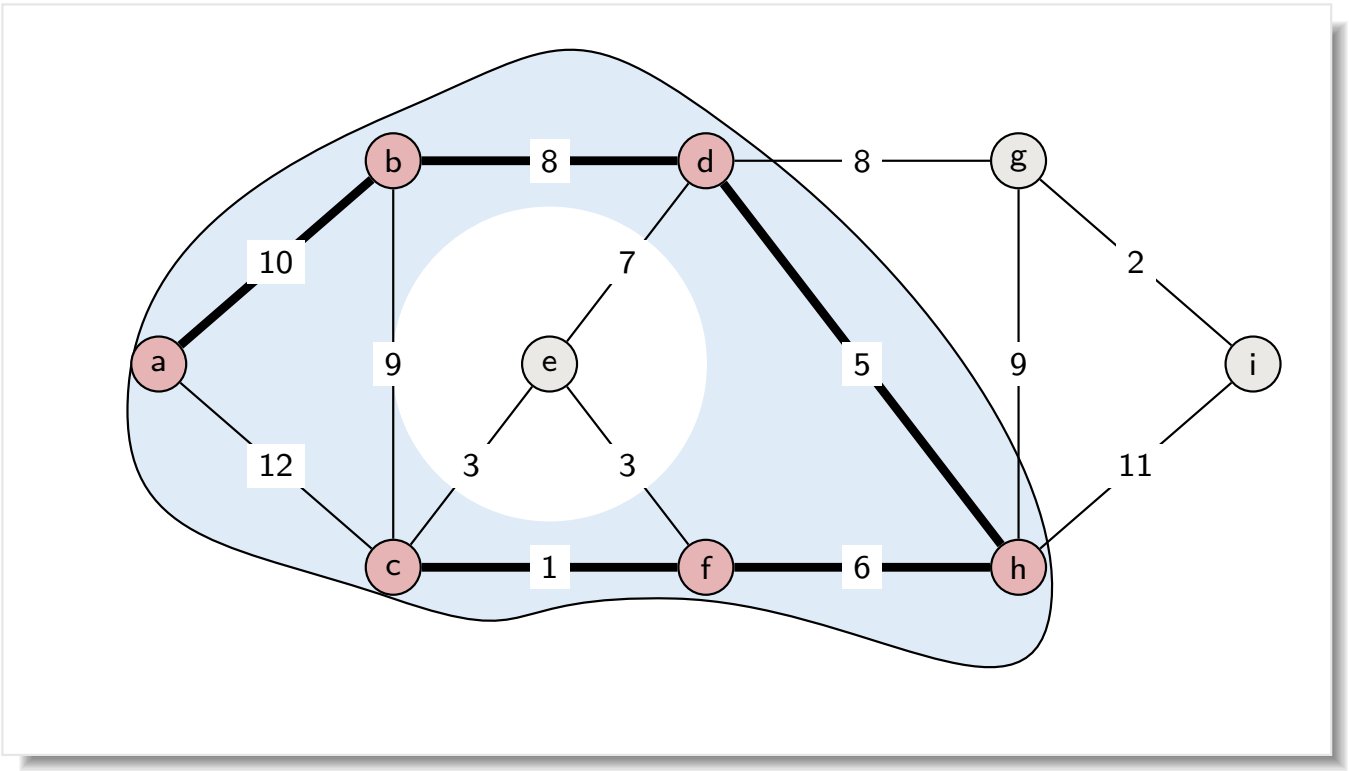


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

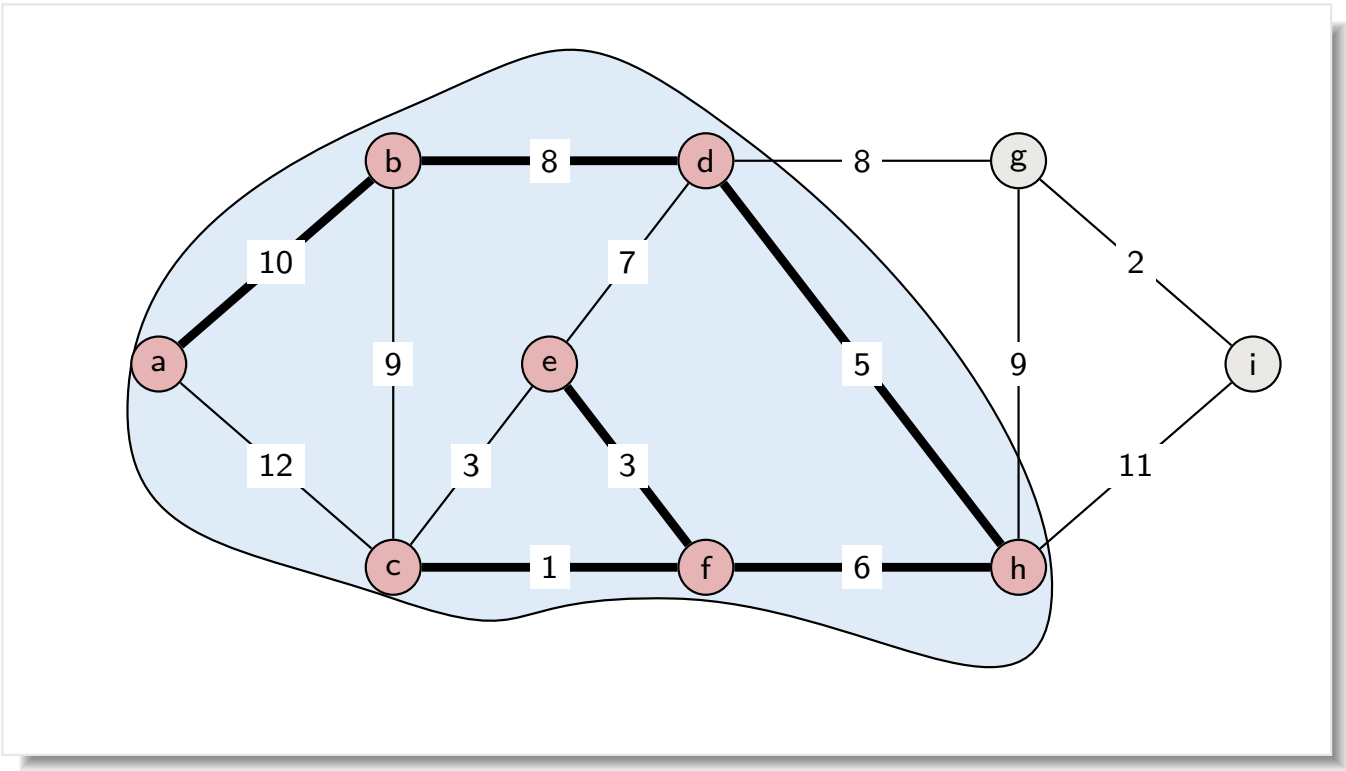


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

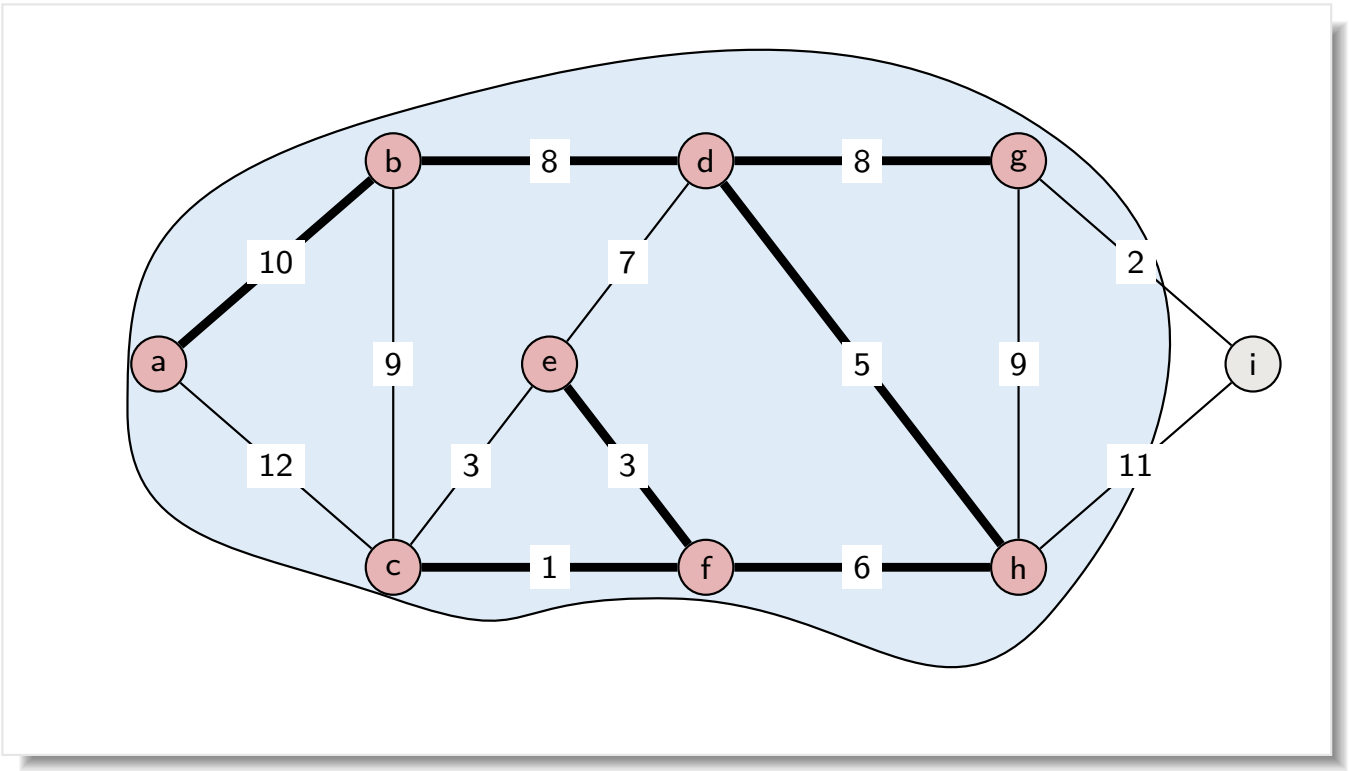


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

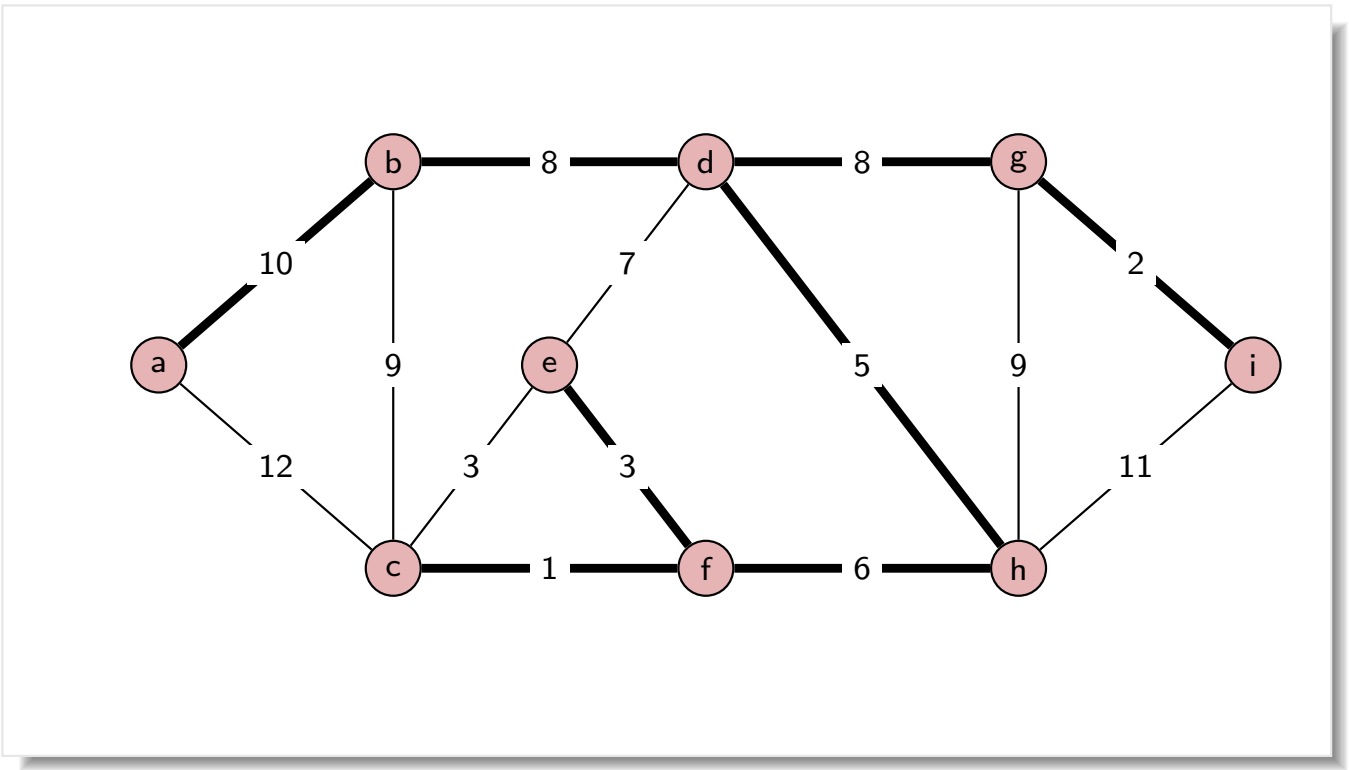


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

at each step add to T a minimum weight crossing edge with respect to the cut induced by T

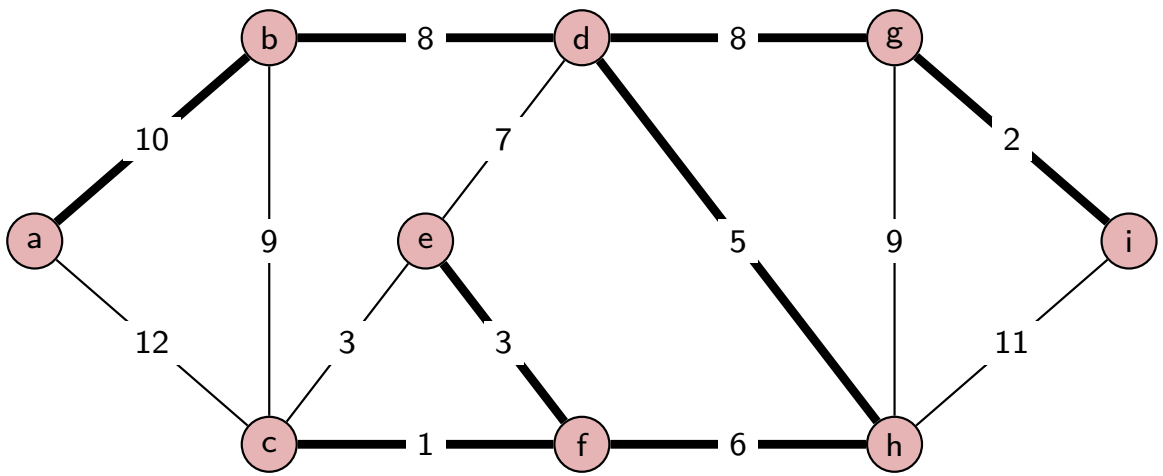


Prim's algorithm

Start with any vertex v, set tree T to singleton v

Greedily grow tree T:

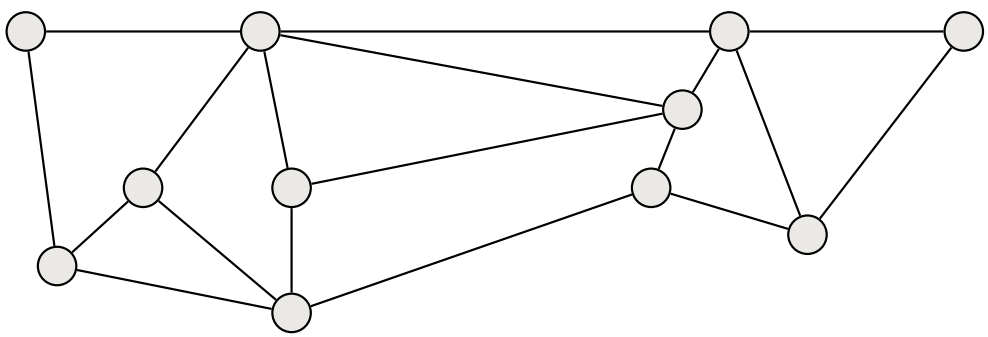
at each step add to T a minimum weight crossing edge with respect to the cut induced by T



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

Why does it work?

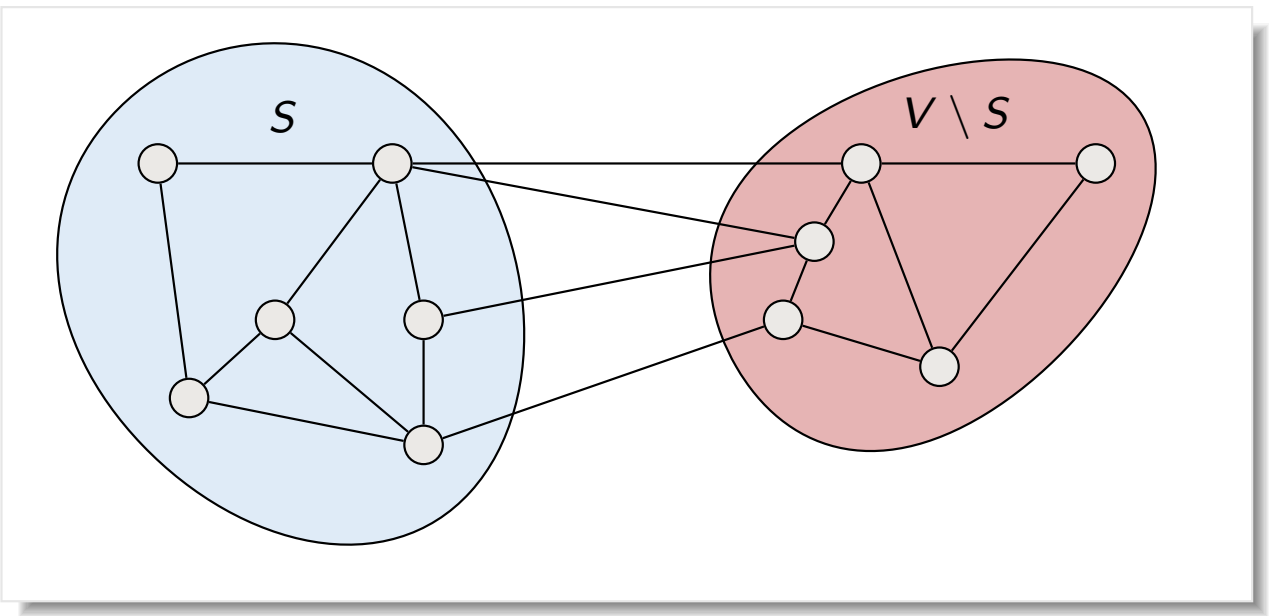
Cuts



Why does it work?

Cuts

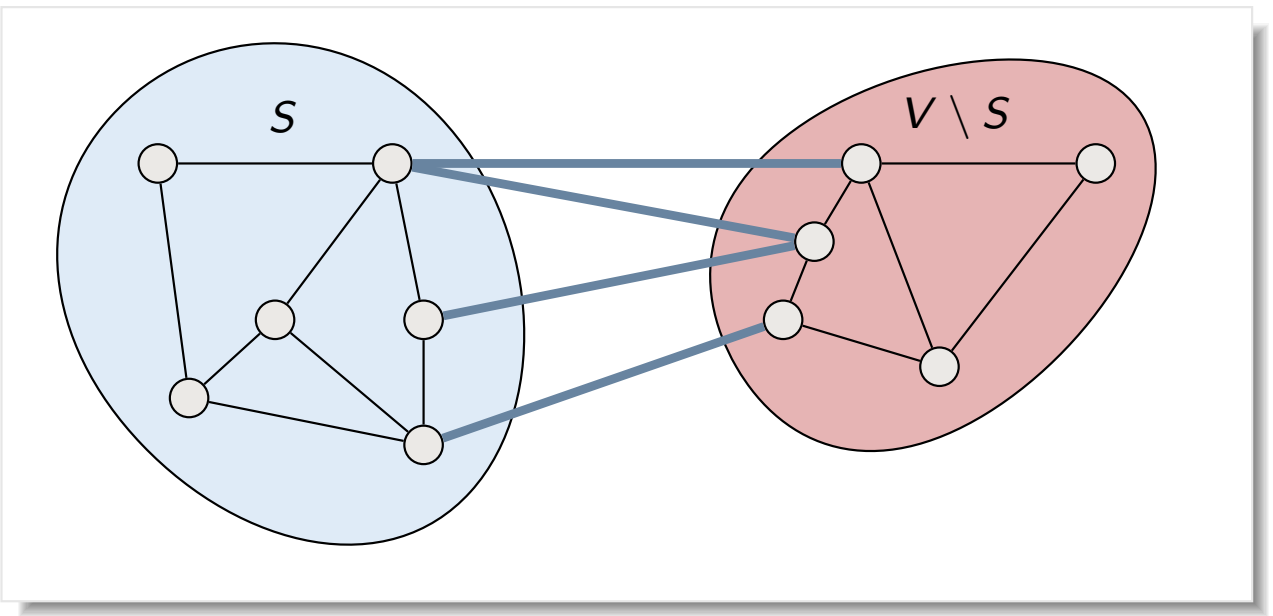
- A **cut** $(S, V \setminus S)$ is a partition of the vertices into two nonempty disjoint sets S and $V \setminus S$



Why does it work?

Cuts

- A **cut** $(S, V \setminus S)$ is a partition of the vertices into two nonempty disjoint sets S and $V \setminus S$
- A **crossing edge** is an edge connecting vertex S to vertex in $V \setminus S$



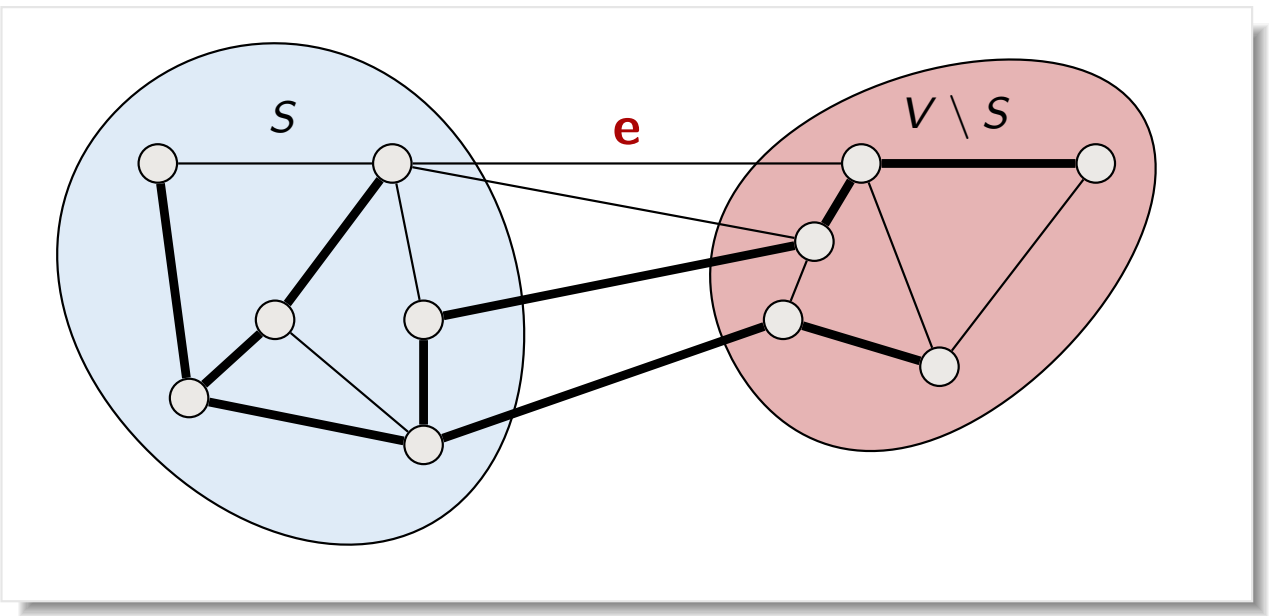
Why does it work?

Cut property

Consider a cut $(S, V \setminus S)$ and let

- T be a tree on S which is part of a MST
- e be a crossing edge of minimum weight

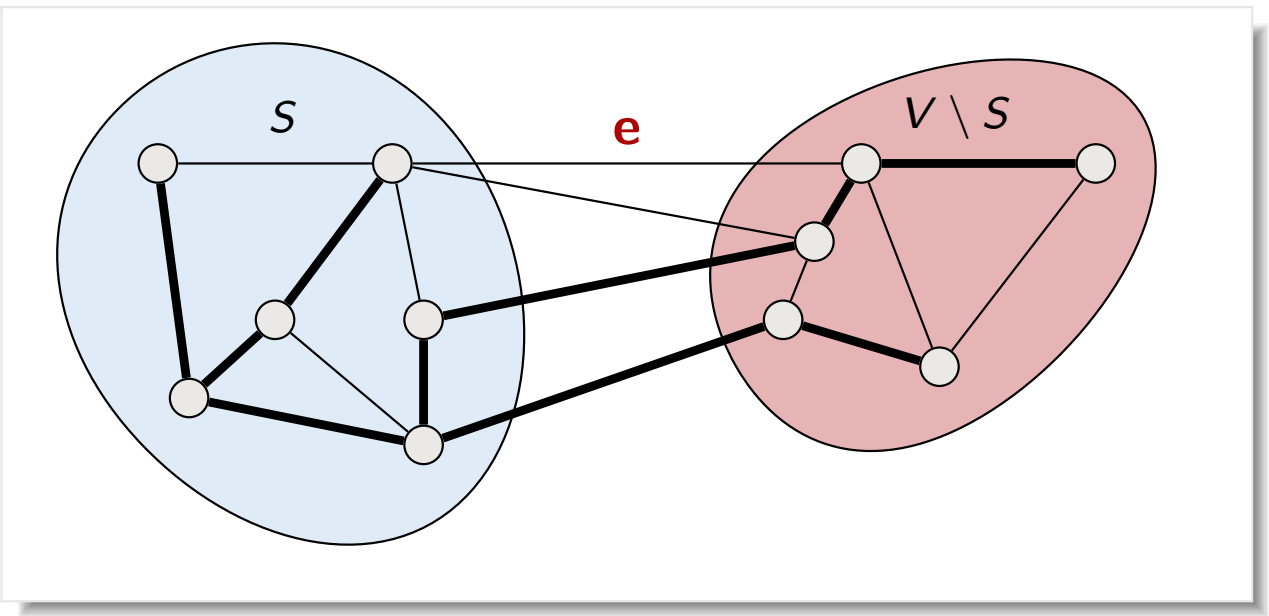
Then there is MST of G containing e and T



Why does it work?

Cut property

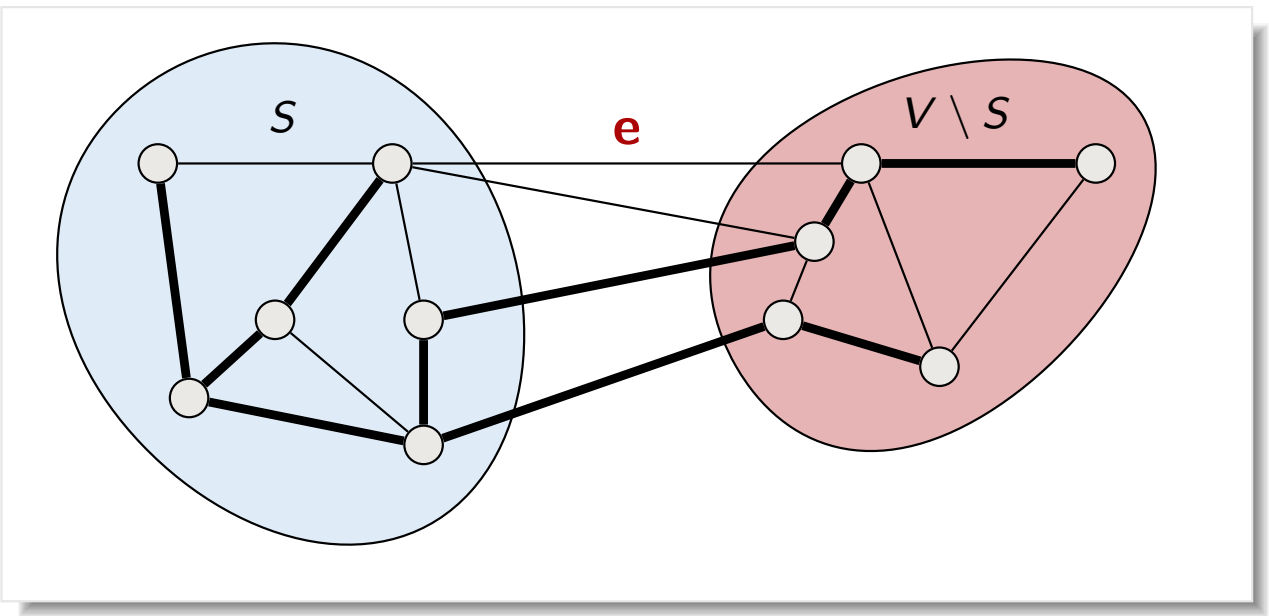
Proof. If e is already in MST we are done.



Why does it work?

Cut property

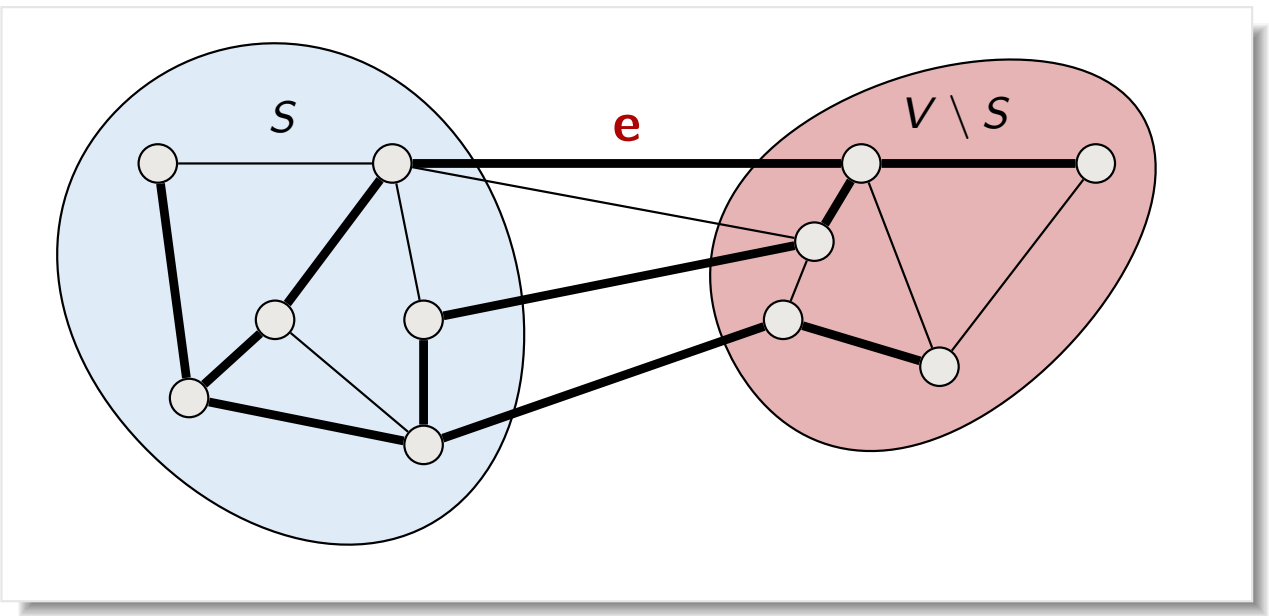
Proof. If e is already in MST we are done.
Otherwise add e to the MST



Why does it work?

Cut property

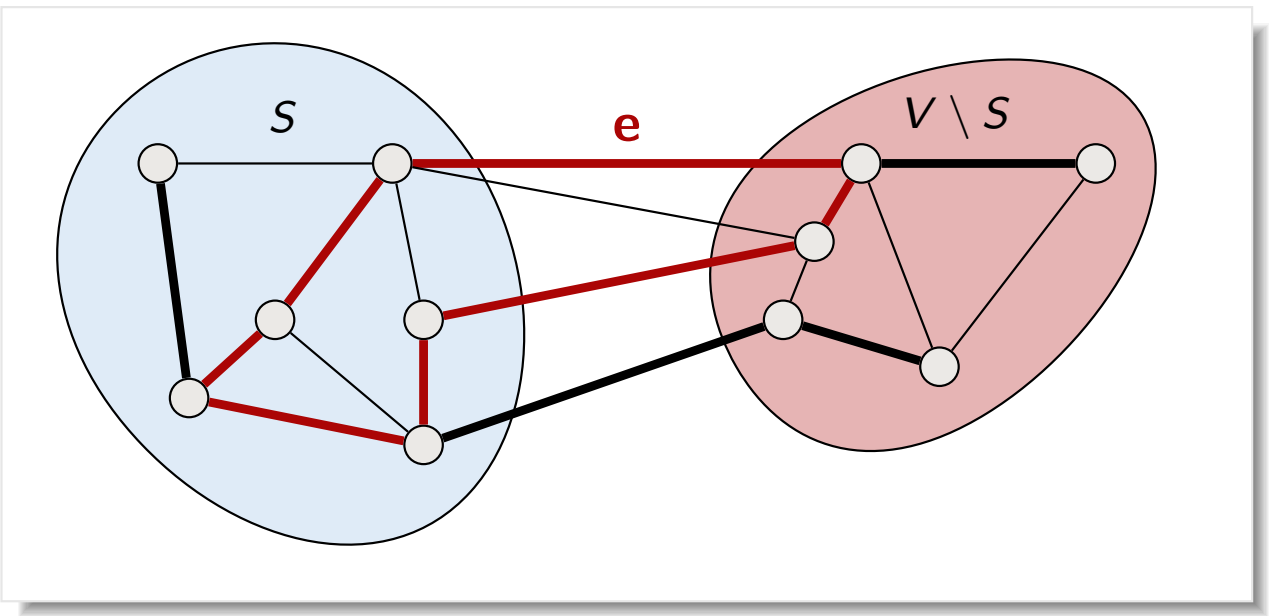
Proof. If e is already in MST we are done.
Otherwise add e to the MST



Why does it work?

Cut property

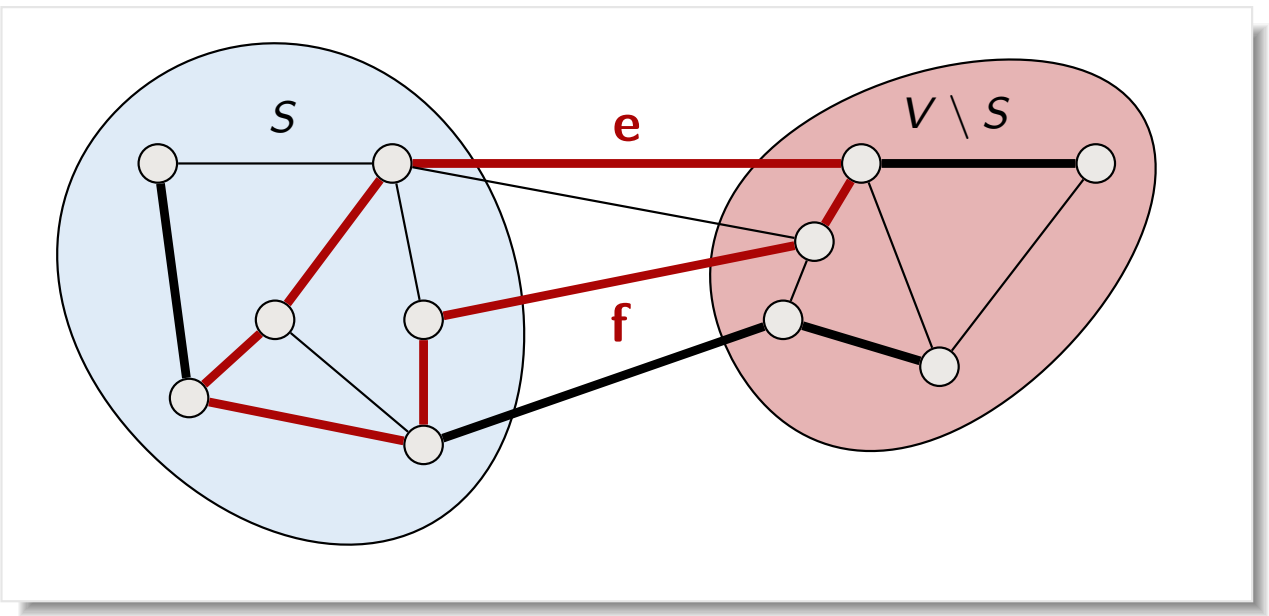
Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle



Why does it work?

Cut property

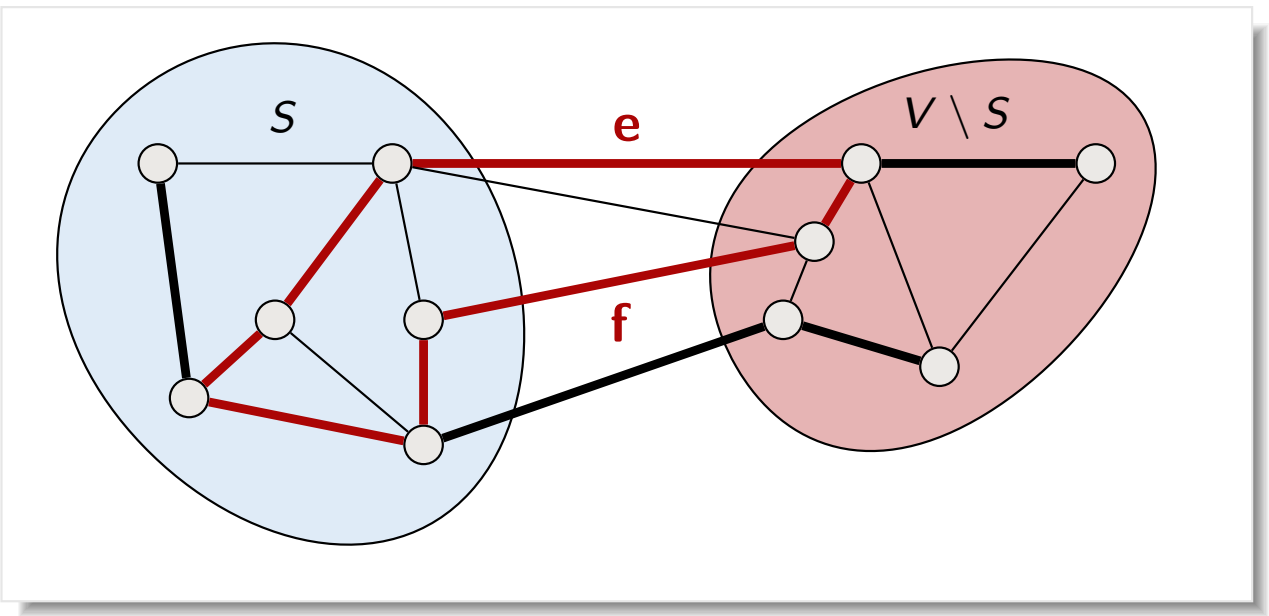
Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle



Why does it work?

Cut property

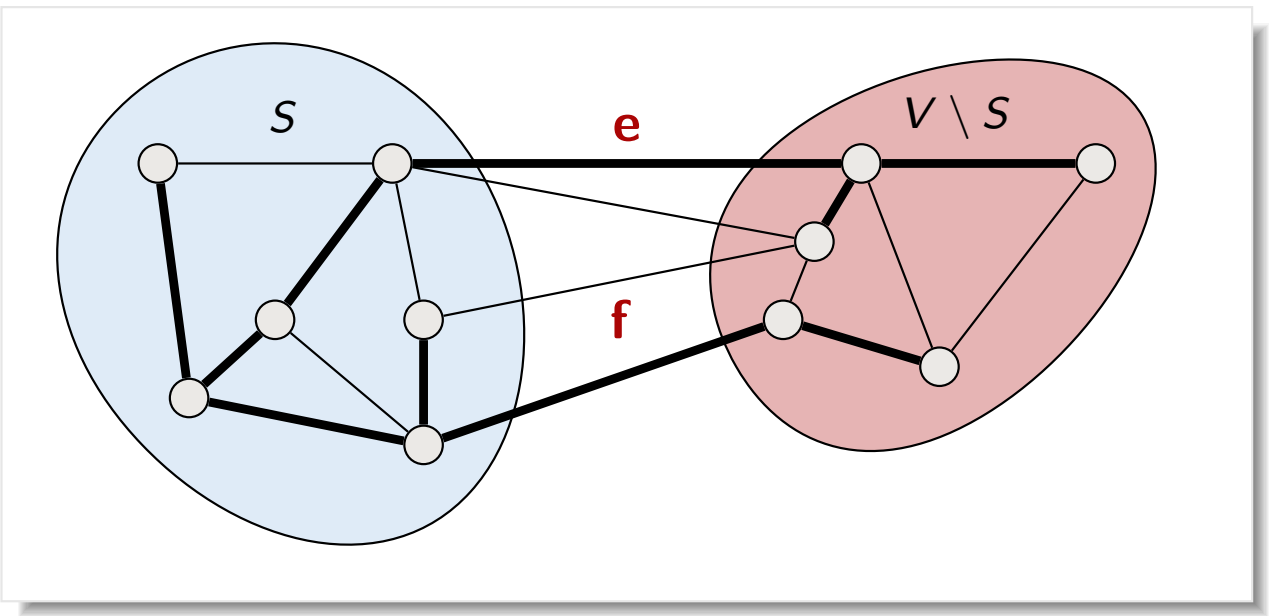
Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle
$w(f) \geq w(e)$ (actually must be equal)



Why does it work?

Cut property

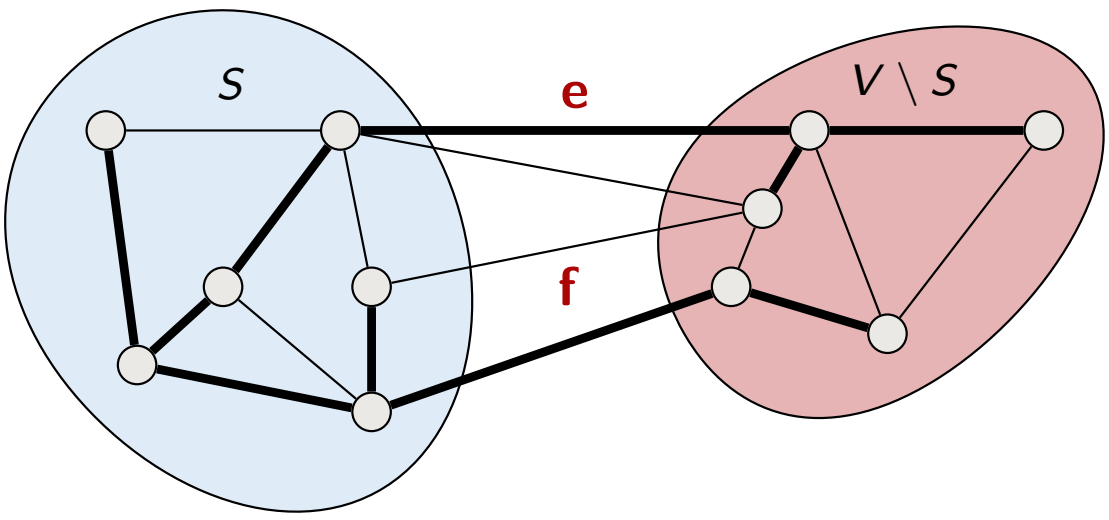
Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle
$w(f) \geq w(e)$ (actually must be equal)
Replace f by e in MST
This gives new MST which contains T and e



Why does it work?

Cut property

Proof. If e is already in MST we are done.
Otherwise add e to the MST
This creates a cycle
At least one other crossing edge f in cycle
$w(f) \geq w(e)$ (actually must be equal)
Replace f by e in MST
This gives new MST which contains T and e



Why does it work?

T is always a subtree of a MST

Proof by induction on number of nodes in T. Final T is MST by this result

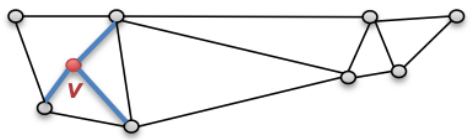
Why does it work?

T is always a subtree of a MST

Proof by induction on number of nodes in T. Final T is MST by this result

Base case: trivial

Singleton v is part of a MST



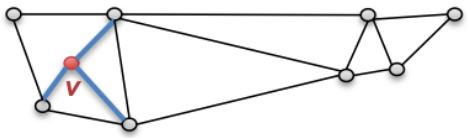
Why does it work?

T is always a subtree of a MST

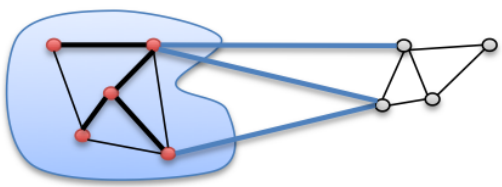
Proof by induction on number of nodes in T. Final T is MST by this result

Base case: trivial

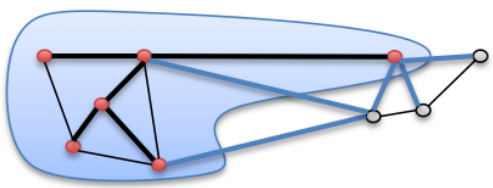
Singleton v is part of a MST



Inductive step: use cut property



In MST by hypothesis



In MST by cut property

Implementation challenge

How do we find minimum crossing edge at every iteration?

Implementation challenge

How do we find minimum crossing edge at every iteration?

Check all outgoing edges:

- O(E) comparisons at every iteration
- O(E V) running time in total

Implementation challenge

How do we find minimum crossing edge at every iteration?

Check all outgoing edges:

- O(E) comparisons at every iteration
- O(E V) running time in total

More clever solution:

- For every node w, keep value $dist(w)$ that measures the "distance" of w from current tree
- When a new node u is added to tree, check whether neighbors of u decreases their distance to tree; if so, decrease distance
- Maintain a min-priority queue for the nodes and their distances

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)          // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Initialize Q and first **for** loop: $O(V \lg V)$

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Initialize Q and first **for** loop: $O(V \lg V)$
- Decrease key of r: $O(\lg V)$

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Initialize Q and first **for** loop: $O(V \lg V)$
- Decrease key of r: $O(\lg V)$
- **while** loop: V EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
  $\leq E$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$

Implementation and Analysis

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

- Initialize Q and first **for** loop: $O(V \lg V)$
- Decrease key of r: $O(\lg V)$
- **while** loop: V EXTRACT-MIN calls $\Rightarrow O(V \lg V)$
  $\leq E$ DECREASE-KEY calls $\Rightarrow O(E \lg V)$
- Total: $O(E \lg V)$ (can be made $O(V \lg V)$ with careful queue implementation)

Kruskal's algorithm

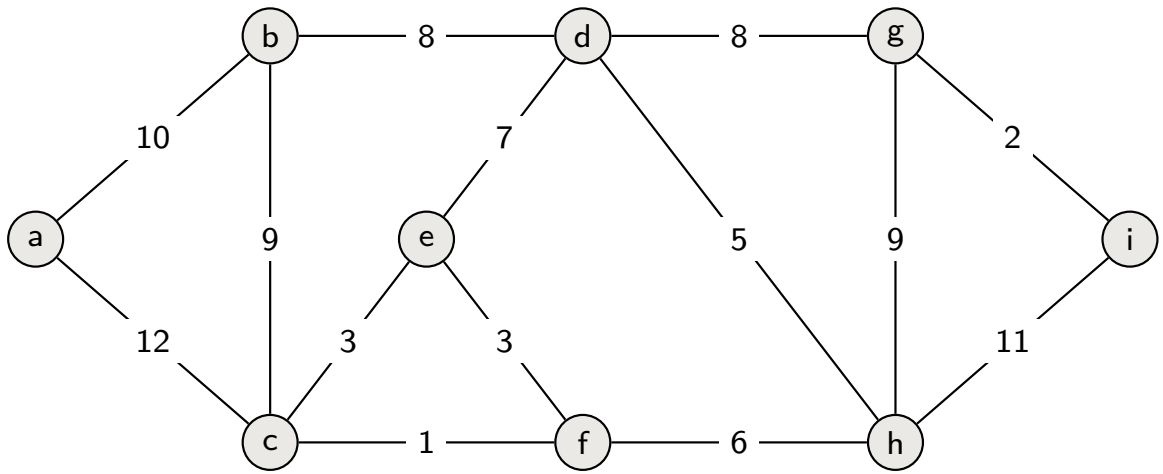
```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

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

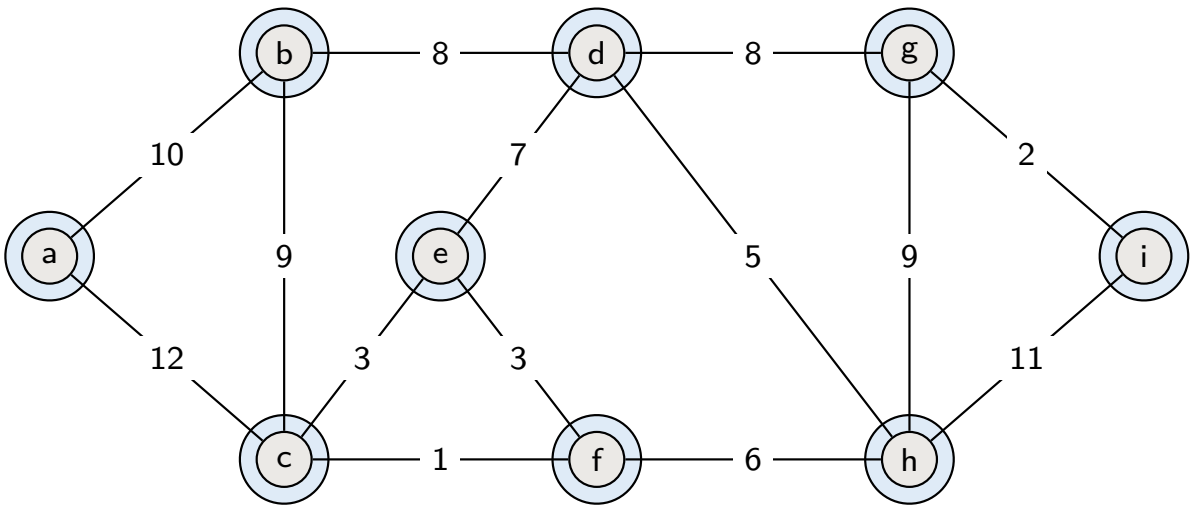


Kruskal's algorithm

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

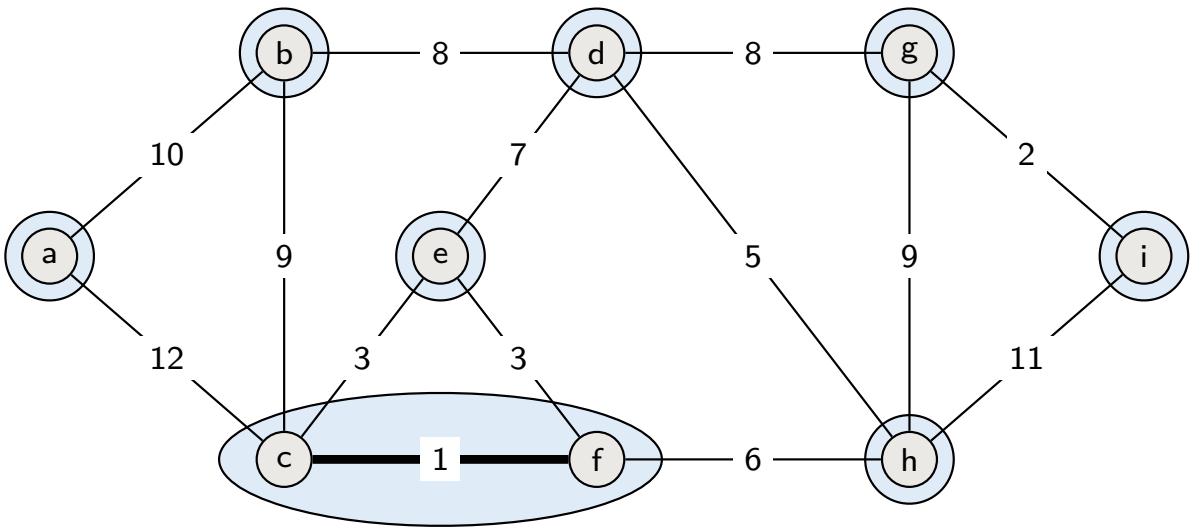


Kruskal's algorithm

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

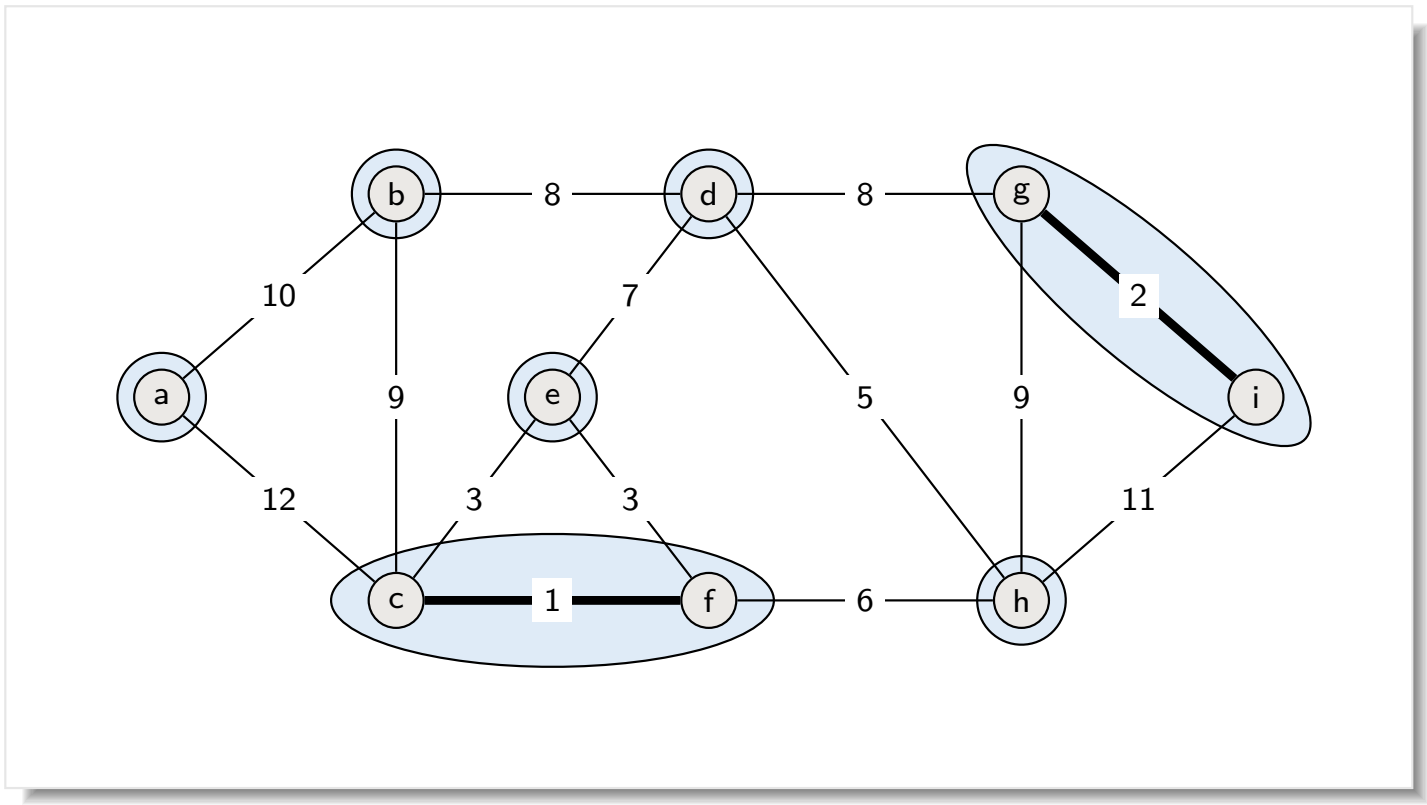


Kruskal's algorithm

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

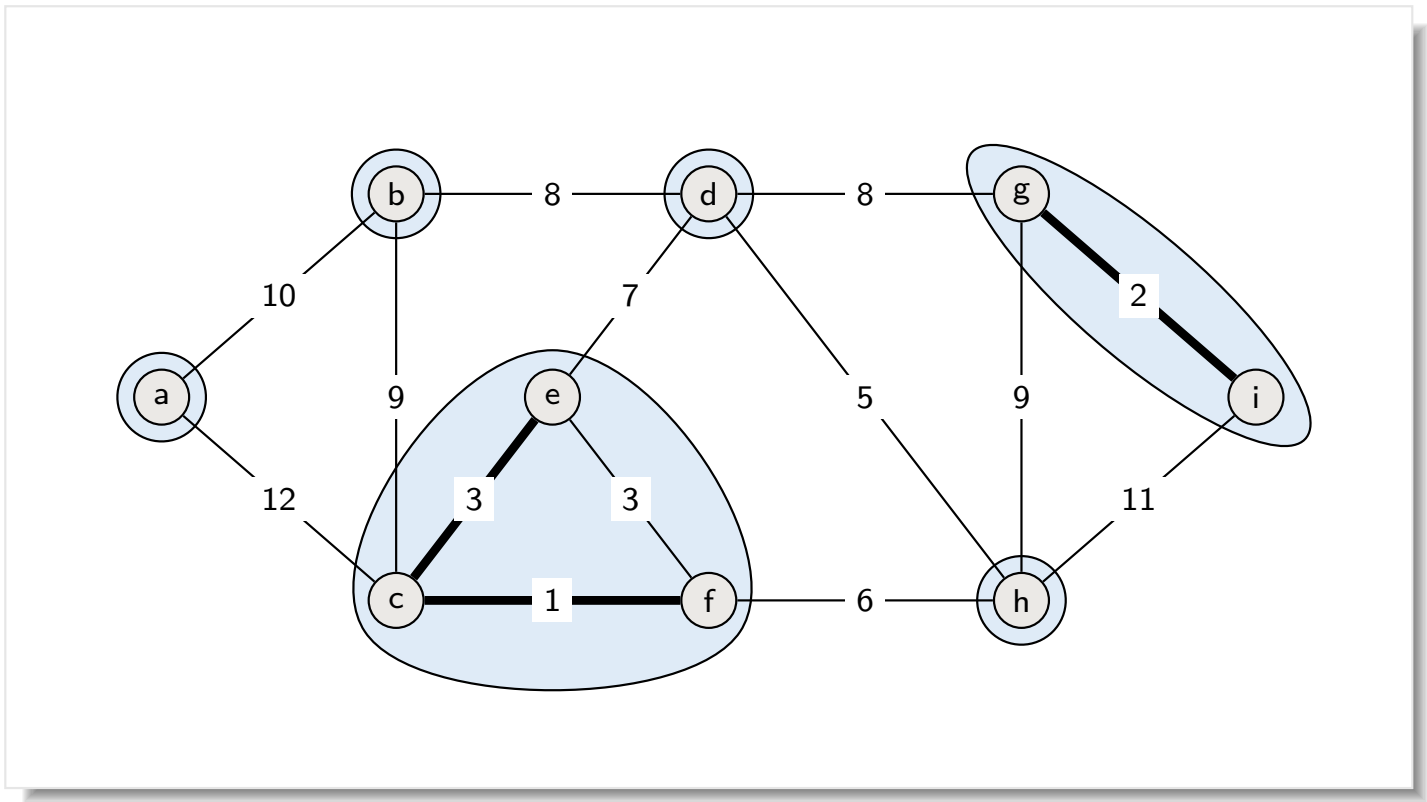


Kruskal's algorithm

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

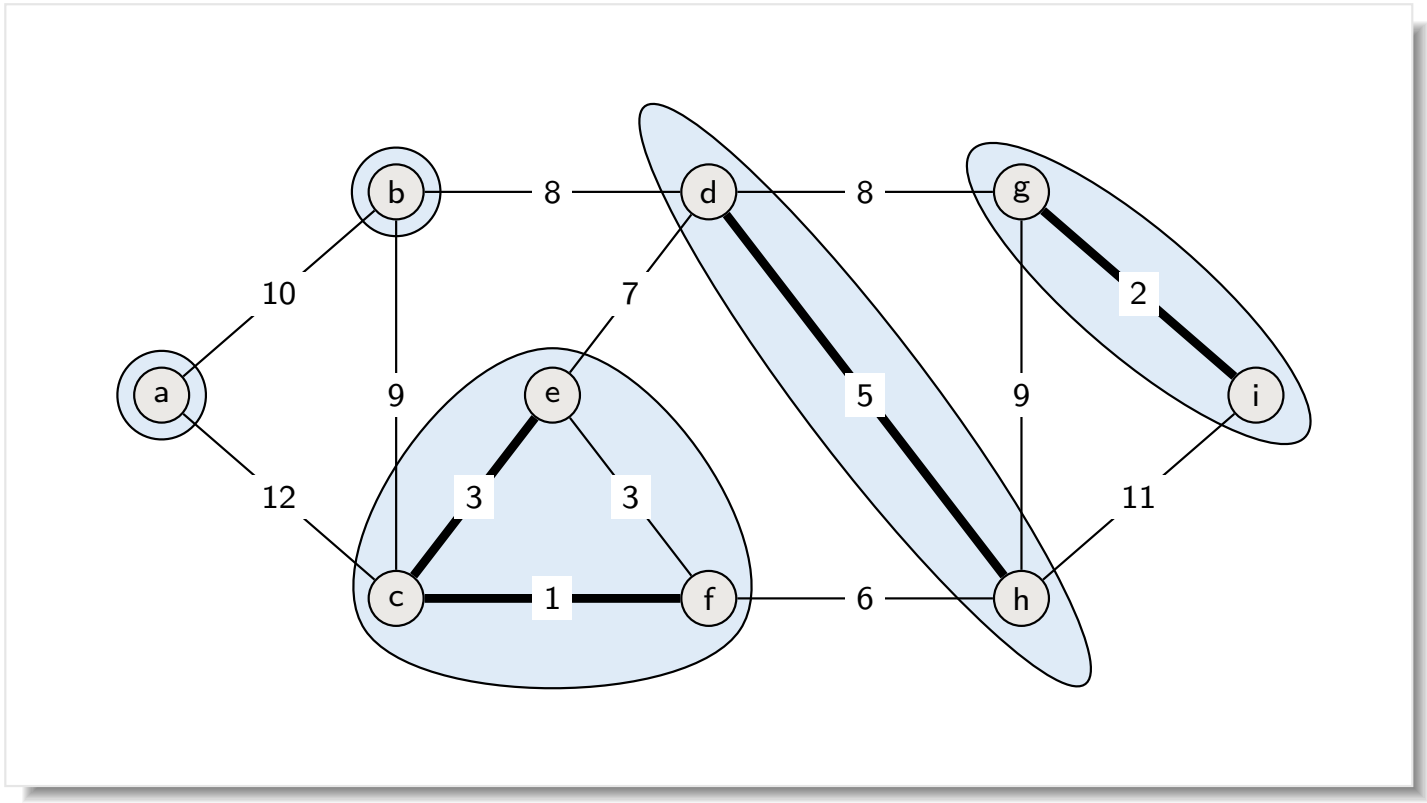


Kruskal's algorithm

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

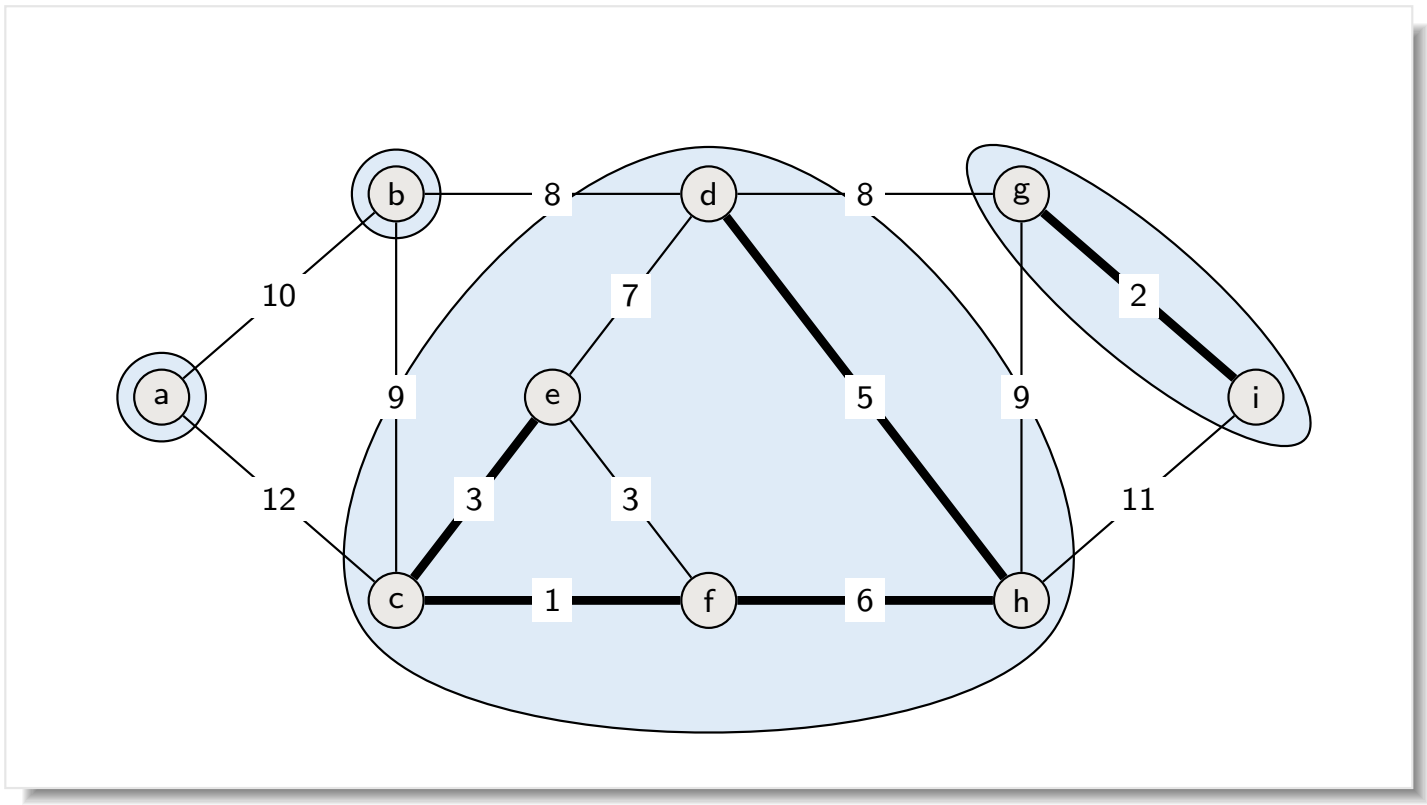


Kruskal's algorithm

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

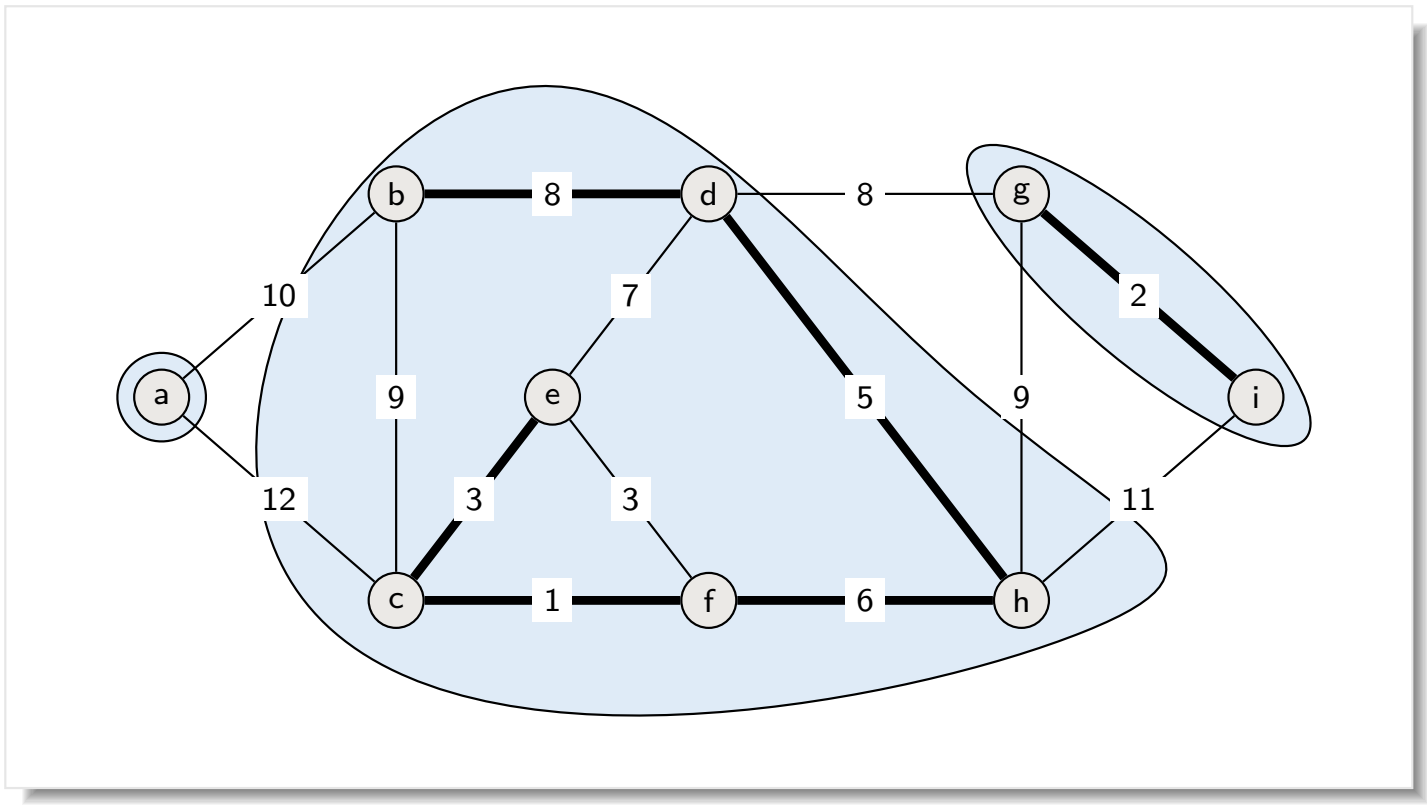


Kruskal's algorithm

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

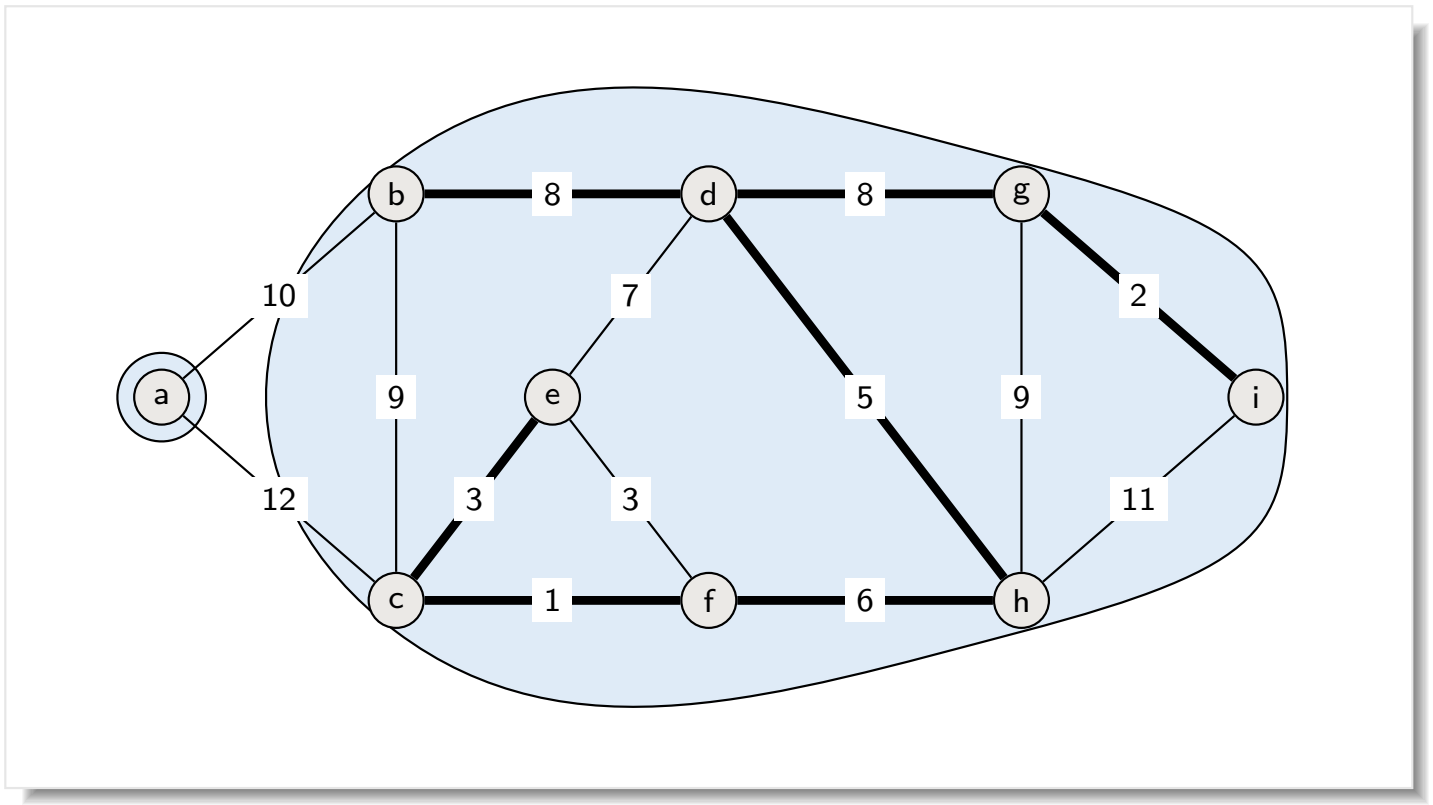


Kruskal's algorithm

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

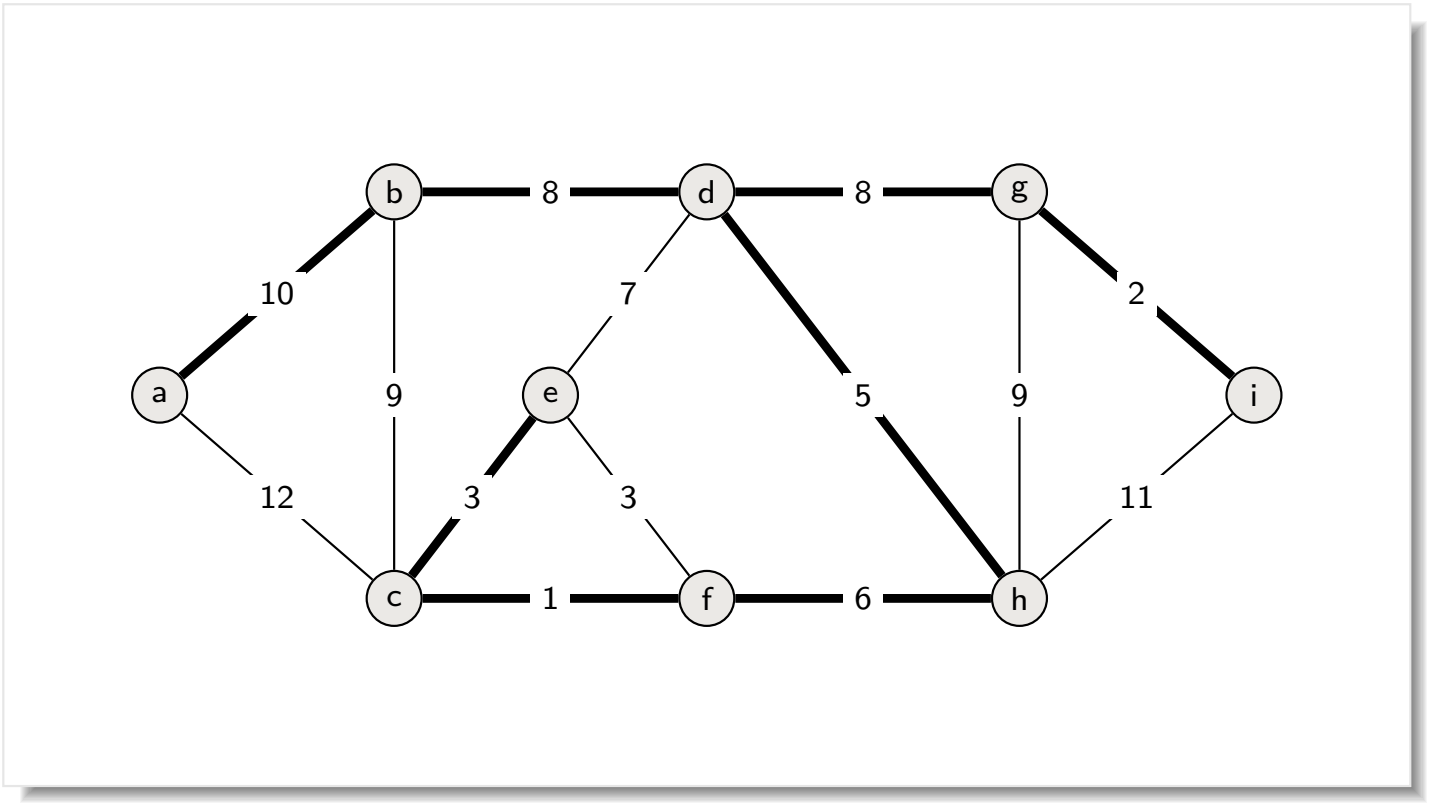


Kruskal's algorithm

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

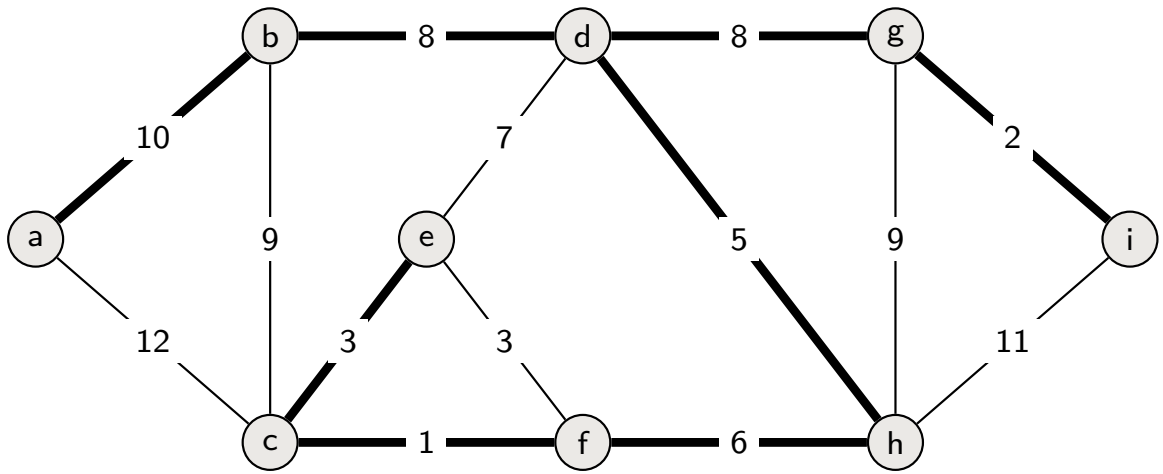


Kruskal's algorithm

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

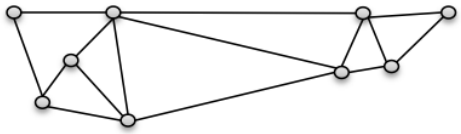
Why does it work?

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



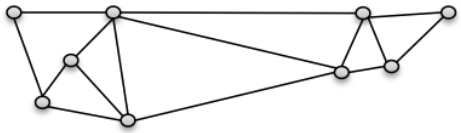
Why does it work?

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

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

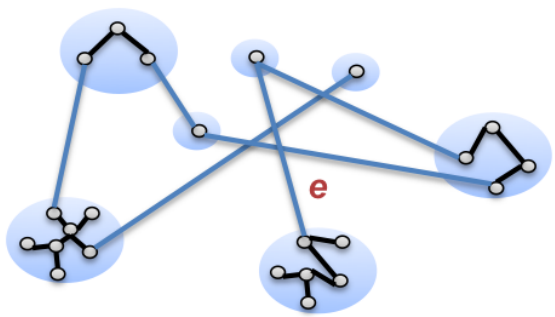
Base case: trivial

T is a union of singleton vertices



Inductive step:

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



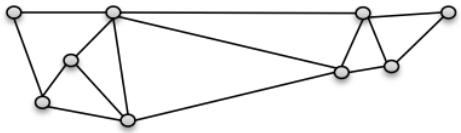
Why does it work?

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

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

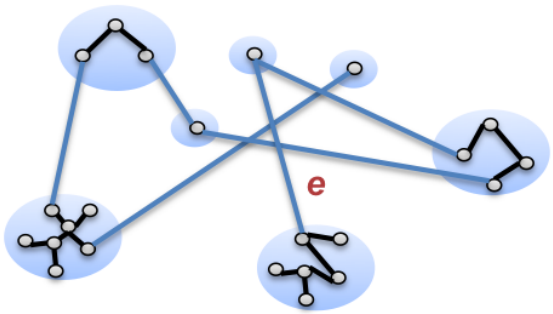
Base case: trivial

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



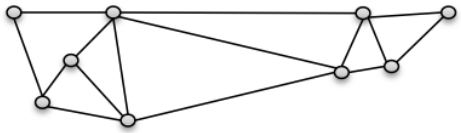
Why does it work?

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

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

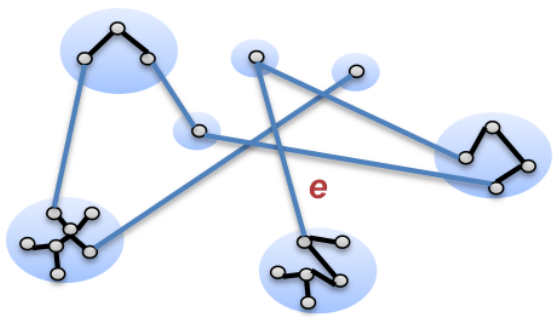
Base case: trivial

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



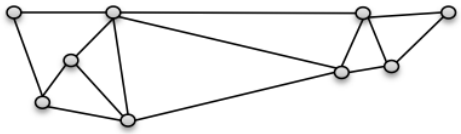
Why does it work?

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

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

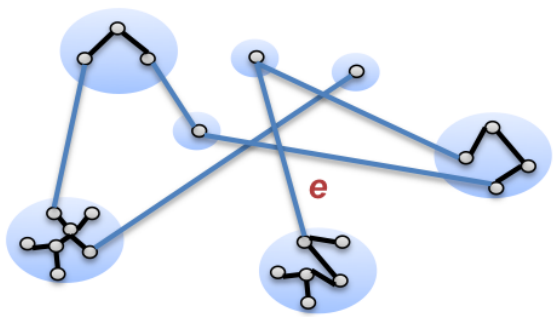
Base case: trivial

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

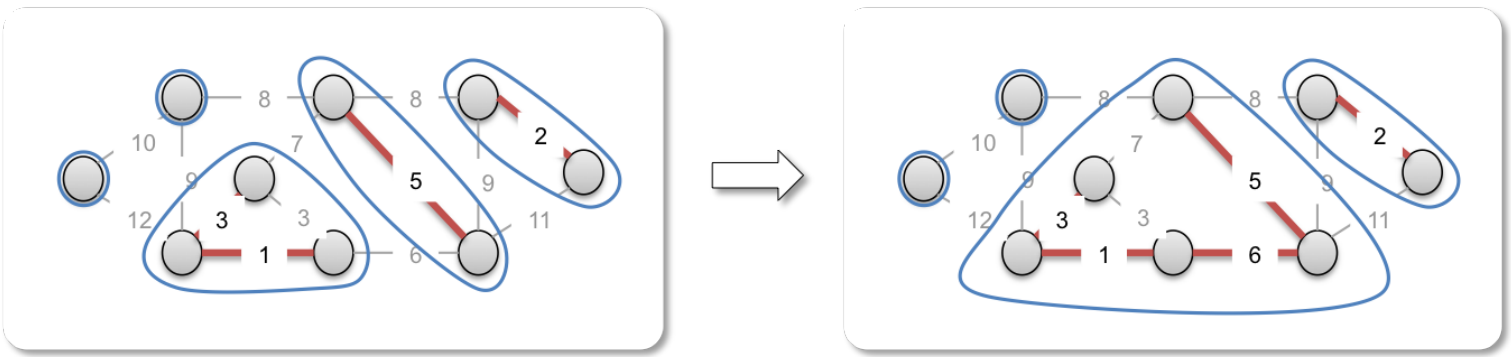
Implementation challenge

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 $\Rightarrow$ **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 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 = ∅
  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: $O((V+E)\alpha(V)) + O(E \lg E) = O(E \lg E) = O(E \lg V)$

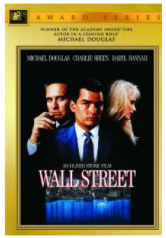
  If edges already sorted time is $O(E\alpha(V))$ which is almost linear

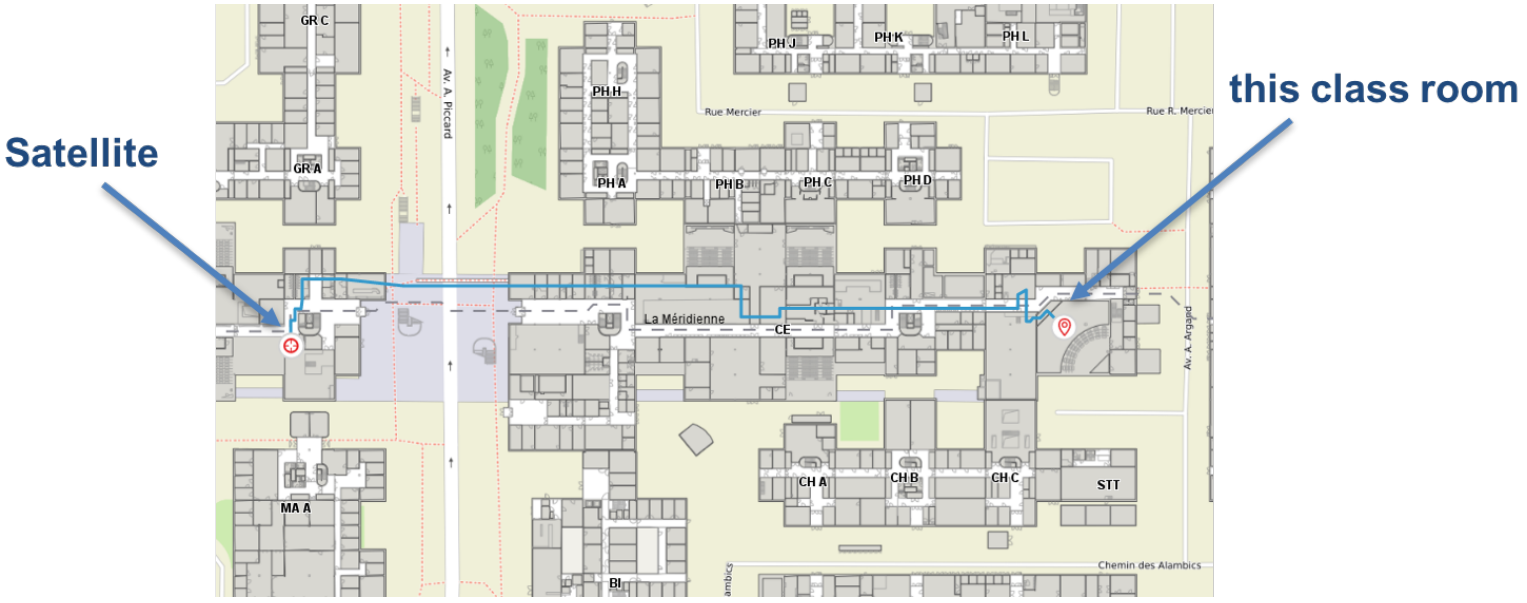
Summary

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

  Min-priority queue for implementation
- Kruskal's algortihm

  Union-Find for implementation
- Many applications

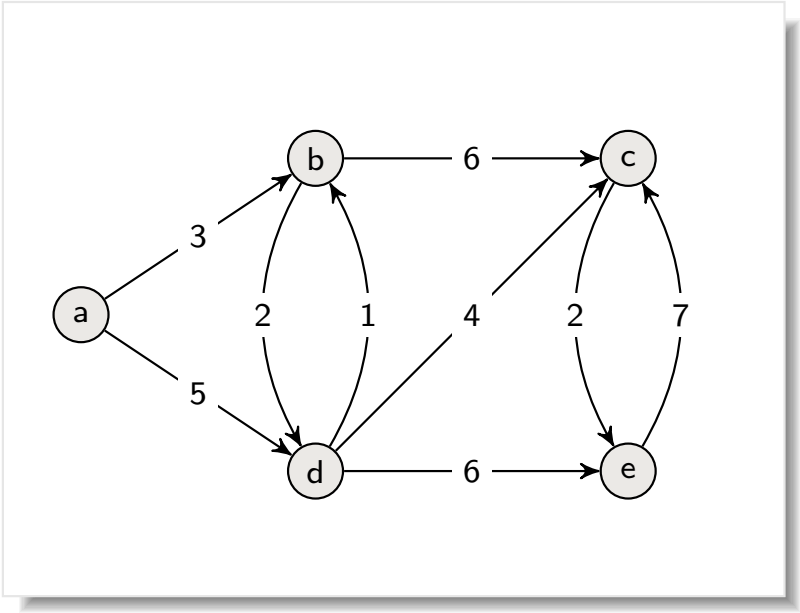




THE SHORTEST PATH PROBLEM

Shortest paths

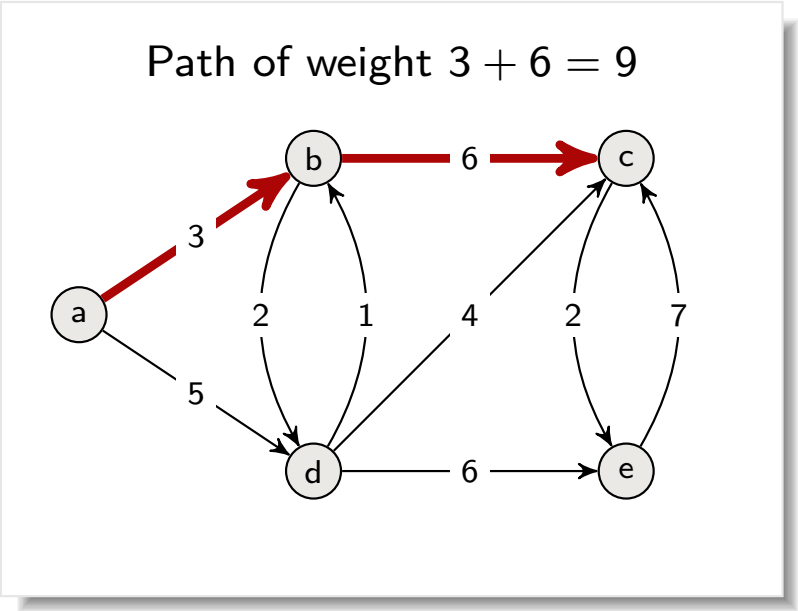
Input: directed graph $G = (V, E)$, edge-weights $w(u, v)$ for $(u, v) \in E$



Shortest paths

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

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

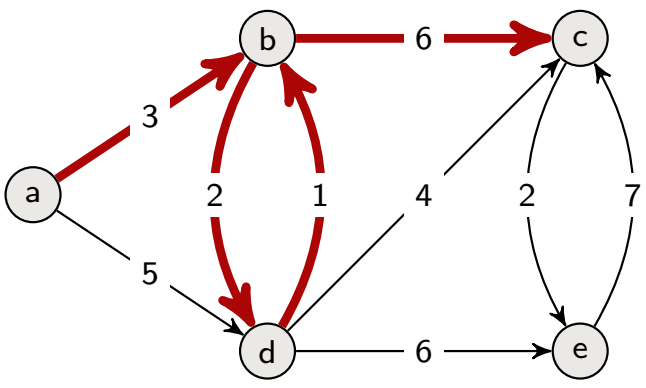


Shortest paths

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

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

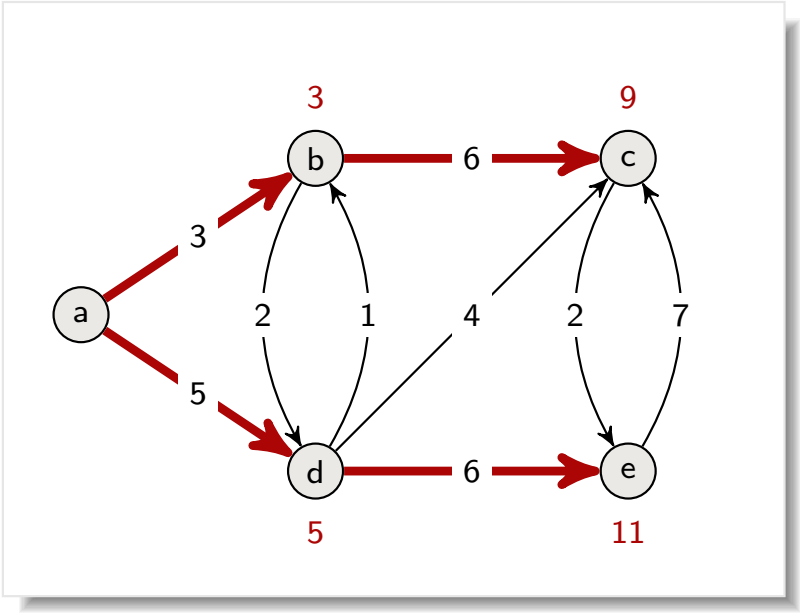
Path of weight $3 + 2 + 1 + 6 = 12$



Shortest paths

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

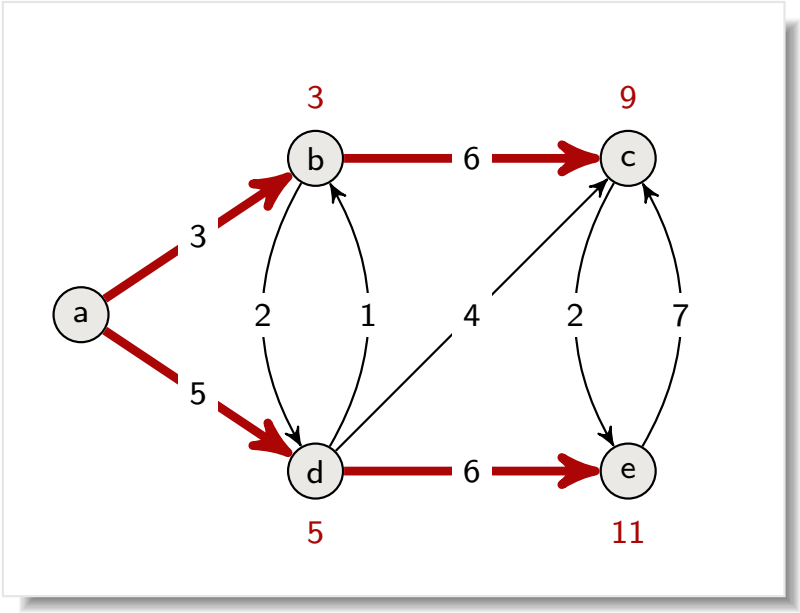
Shortest paths from a: (may have many solutions)

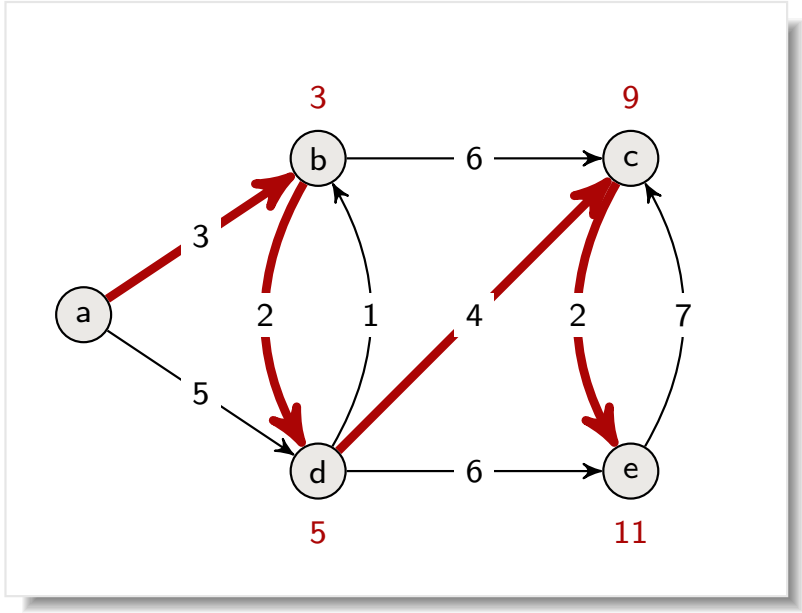


Shortest paths

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

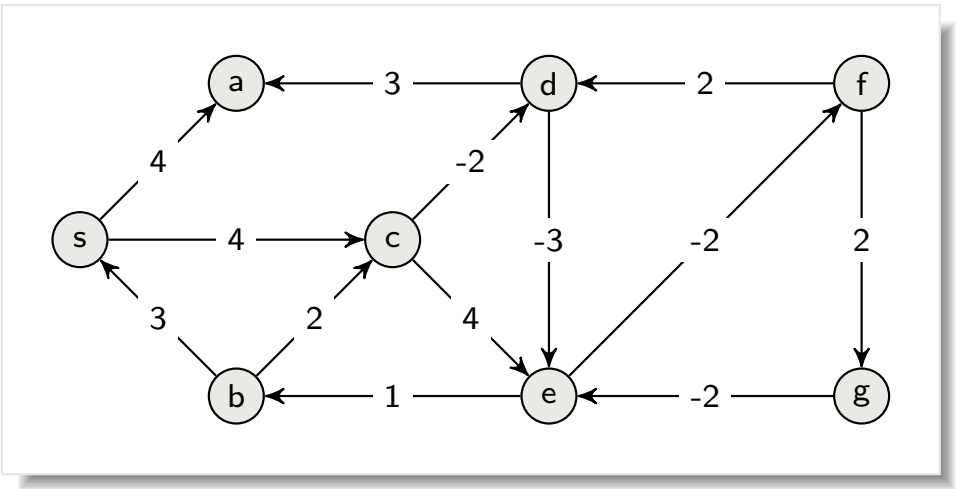
All-pairs: Find shortest path from u to v for all pairs u, v of vertices

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**

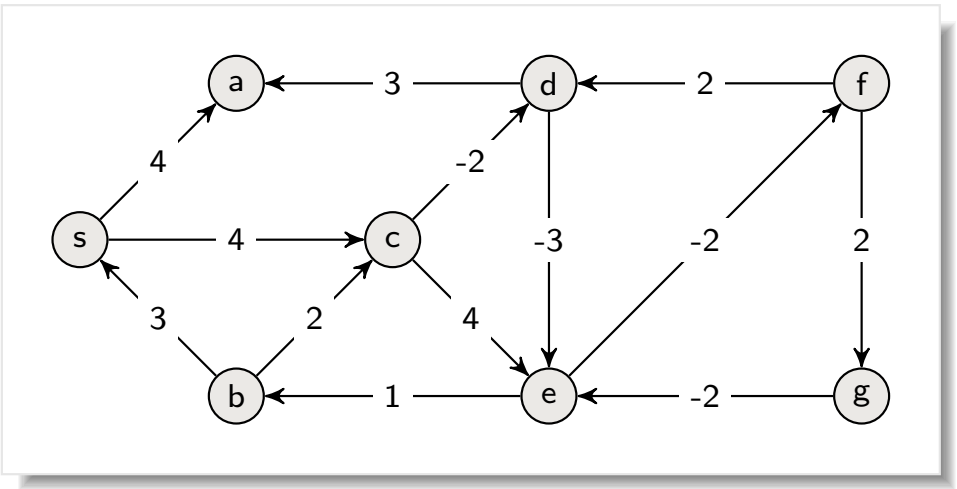


Negative-weight edges

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 $-\infty$



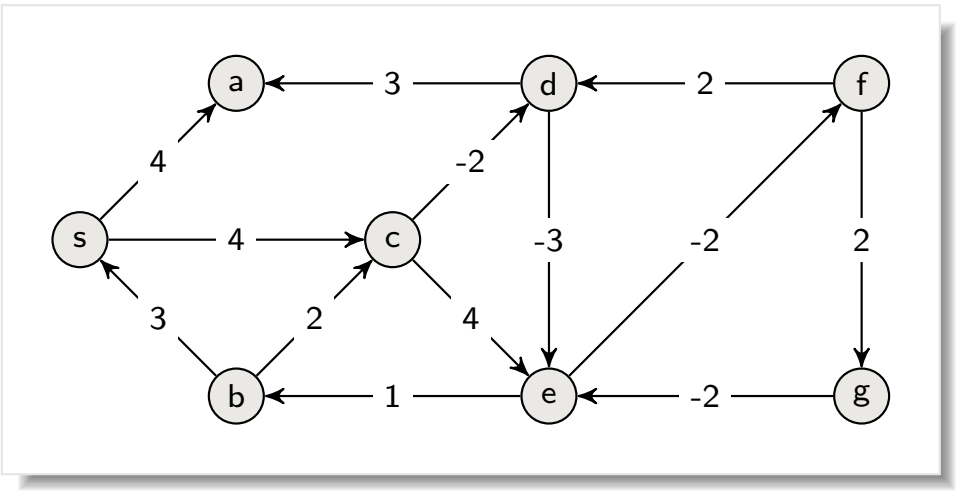
Negative-weight edges

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?

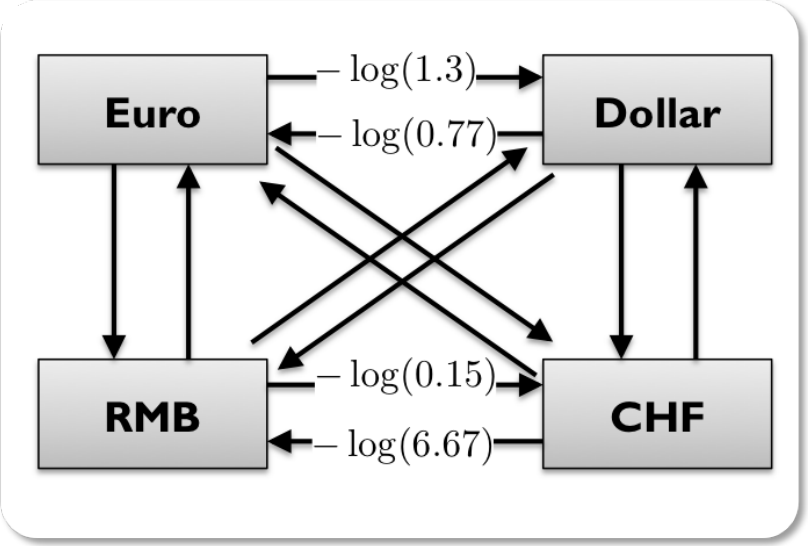
Example: Buying and selling currency

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

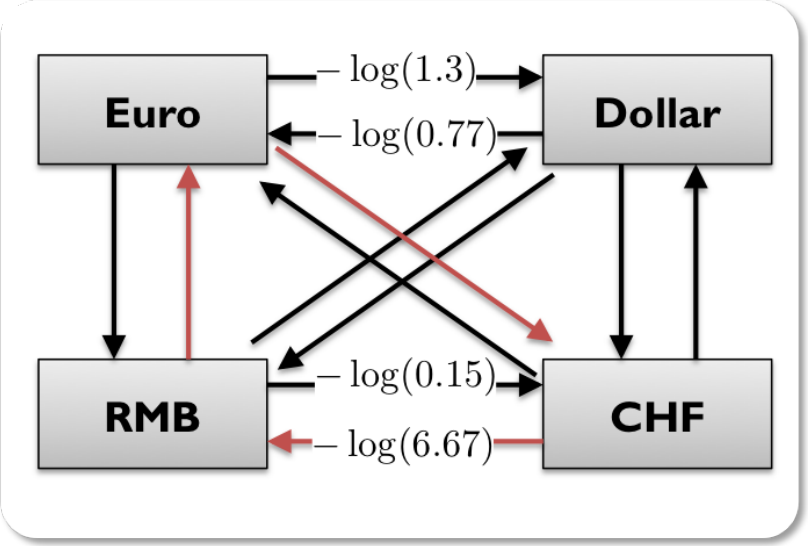
	Euro	Dollar	RMB	CHF
Euro	1.0	1.3	8	1.2
Dollar	0.77	1.0	6.2	0.93
RMB	0.14	0.16	1.0	0.15
CHF	0.83	1.07	6.67	1.0

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

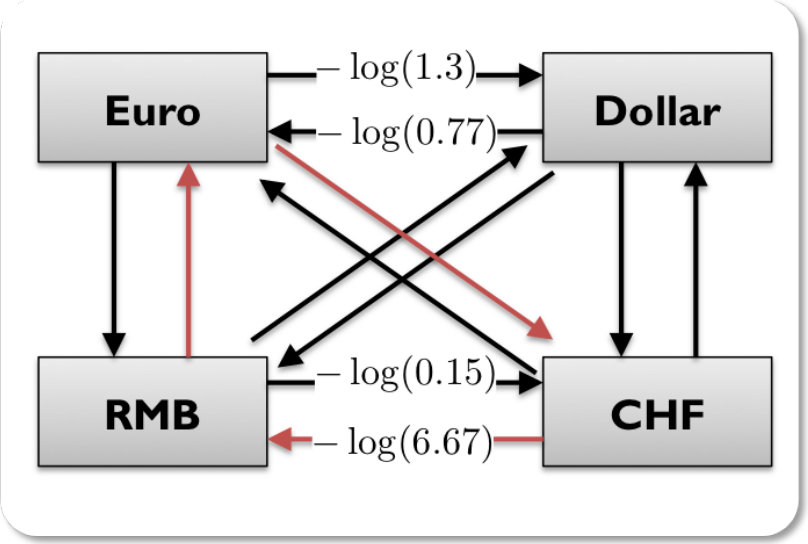
Convert Euro to CHF

Amount of CHF obtained:

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

≈ 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

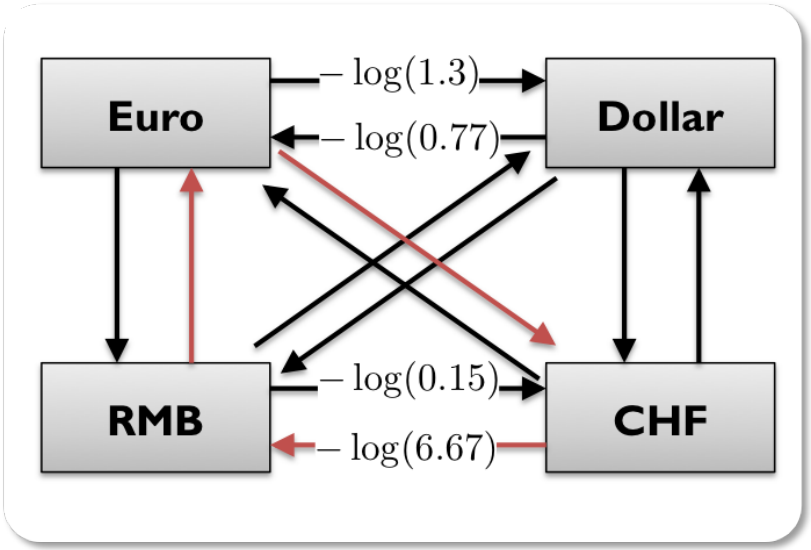
Convert Euro to CHF

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 $v \in G.V$
$v.d = \infty$
$v.\pi = \text{NIL}$
$s.d = 0$

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

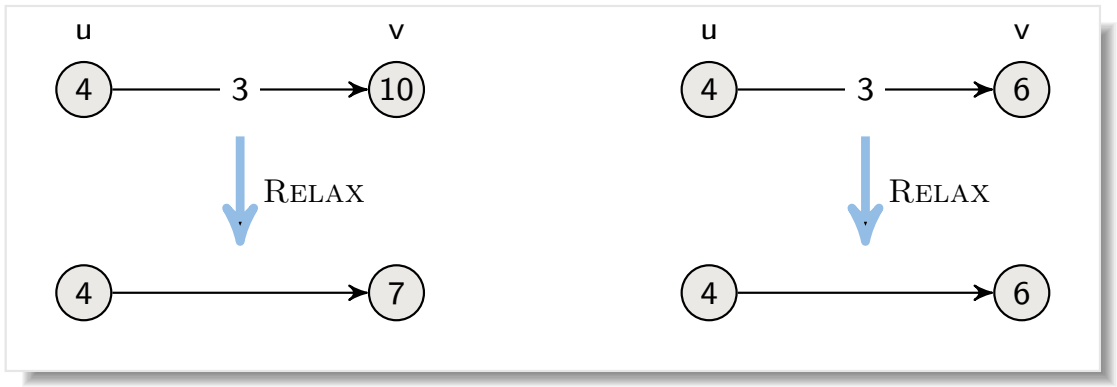
Can we improve the shortest path estimate for v by going through u and taking (u, v)?

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

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

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

$v.\pi = u$



Improving the shortest-path estimate

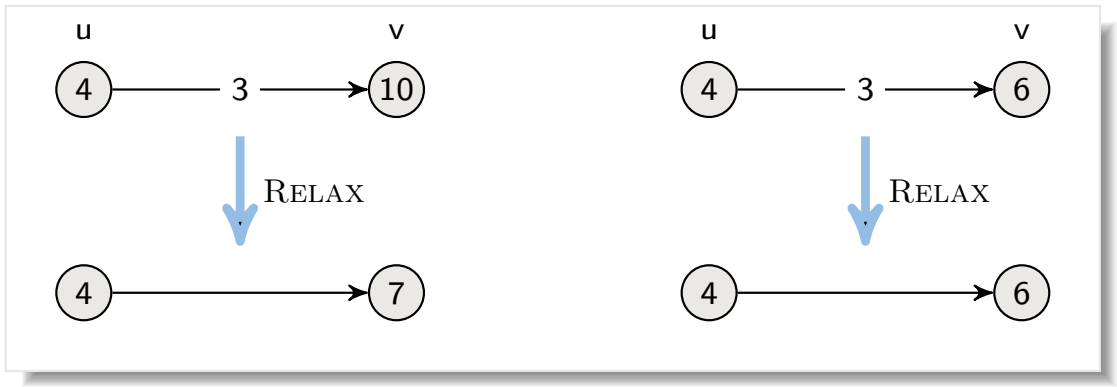
Can we improve the shortest path estimate for v by going through u and taking (u, v)?

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

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

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

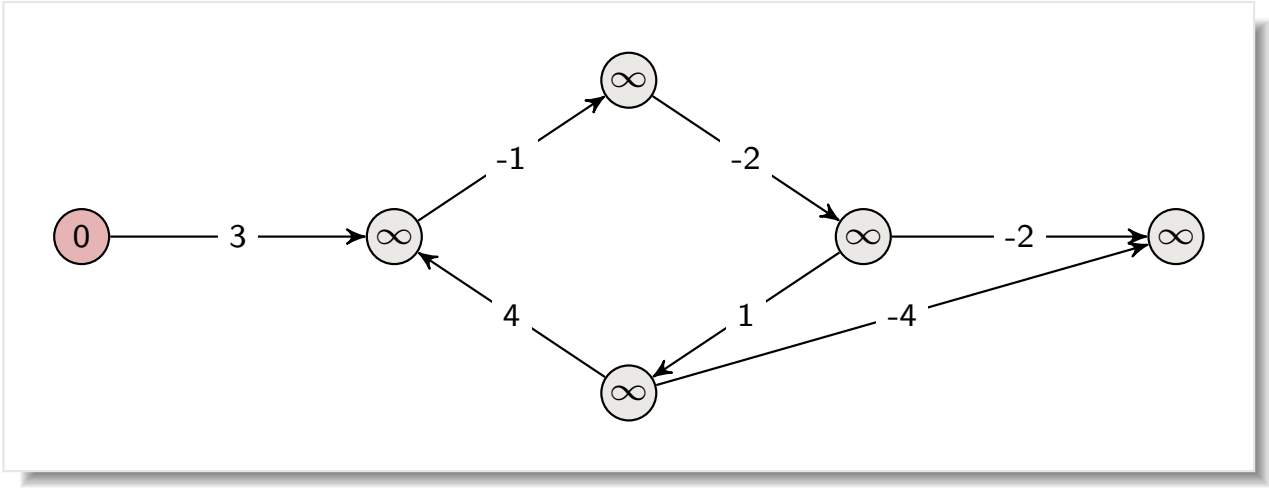
$v.\pi = u$



Bellman-Ford updates shortest-path estimates iteratively by using $\textsc{Relax}$

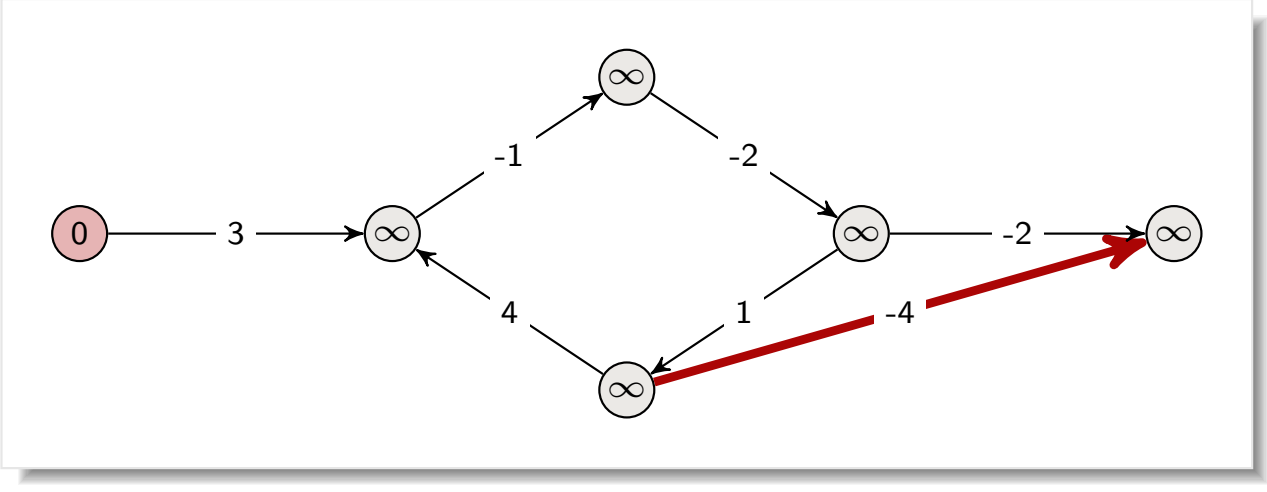
Bellman-Ford Algorithm

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



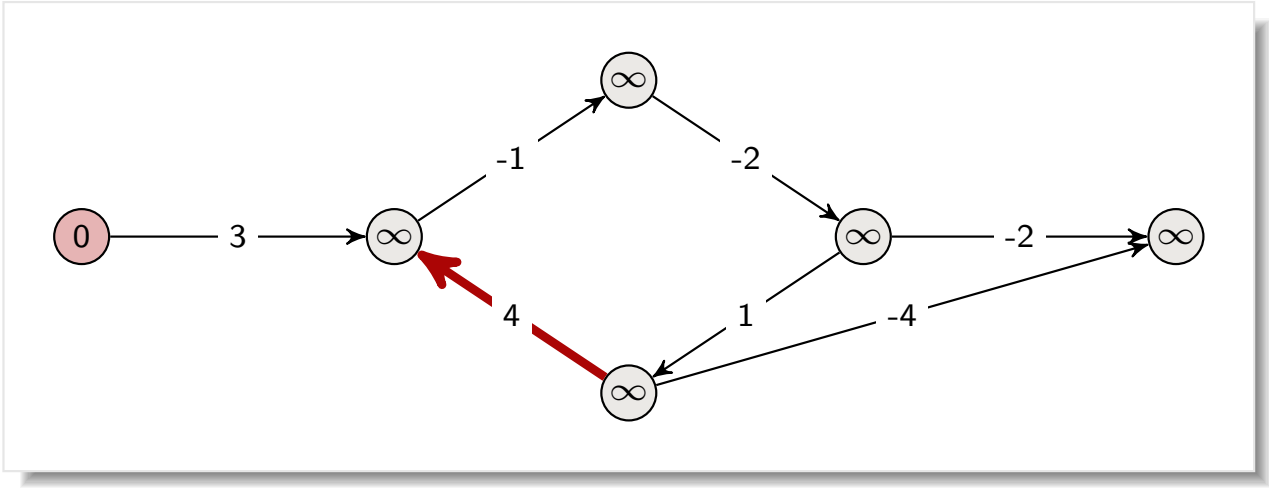
Bellman-Ford Algorithm

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)



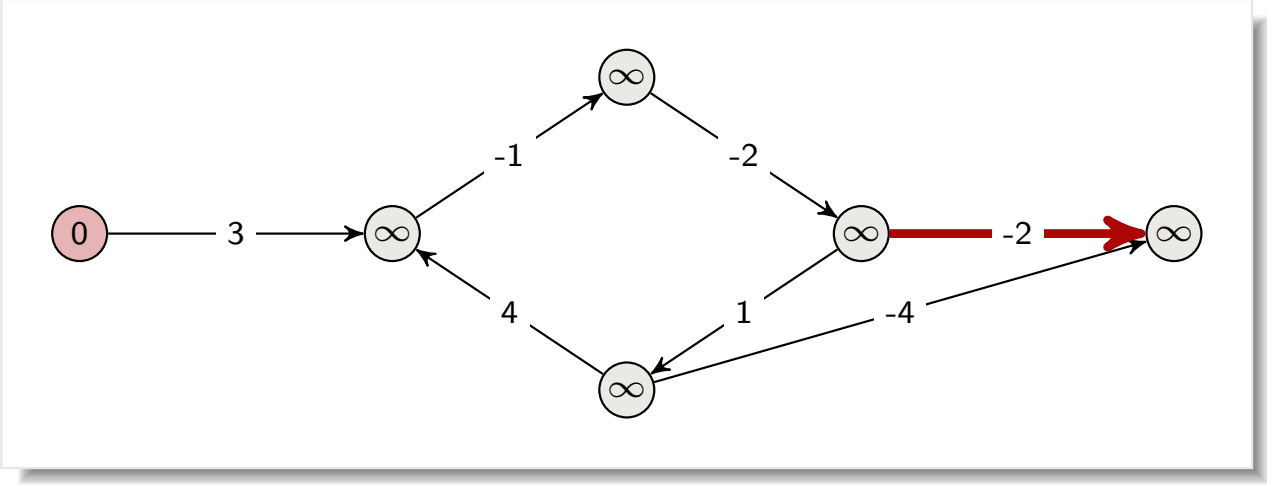
Bellman-Ford Algorithm

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



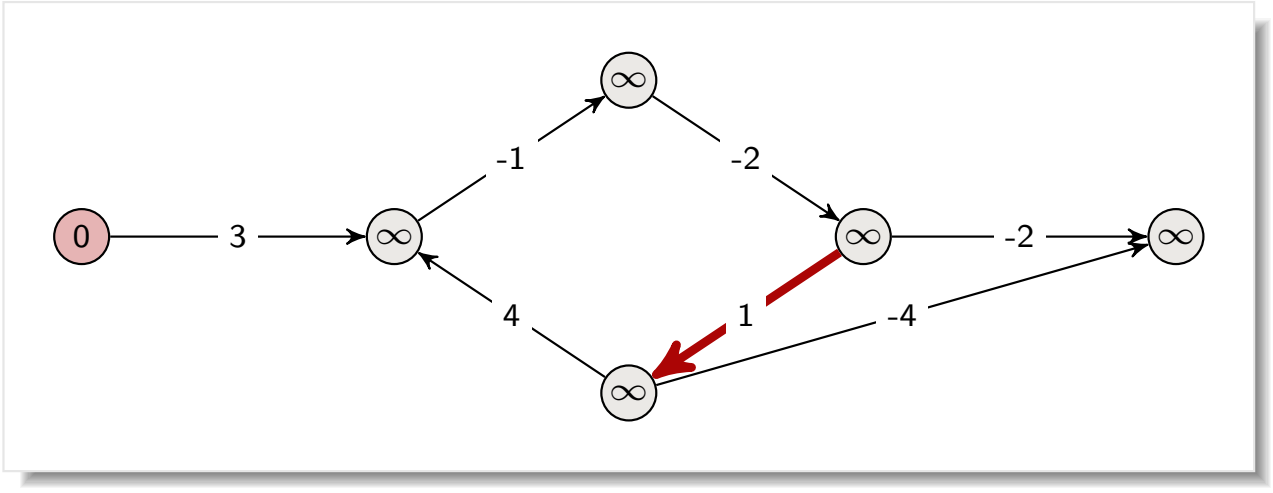
Bellman-Ford Algorithm

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)



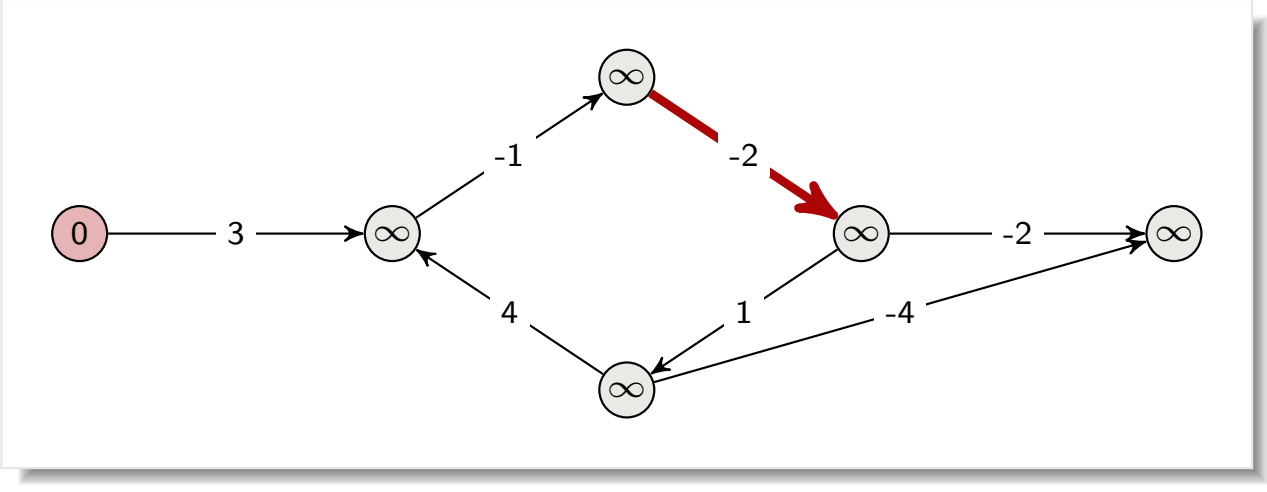
Bellman-Ford Algorithm

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



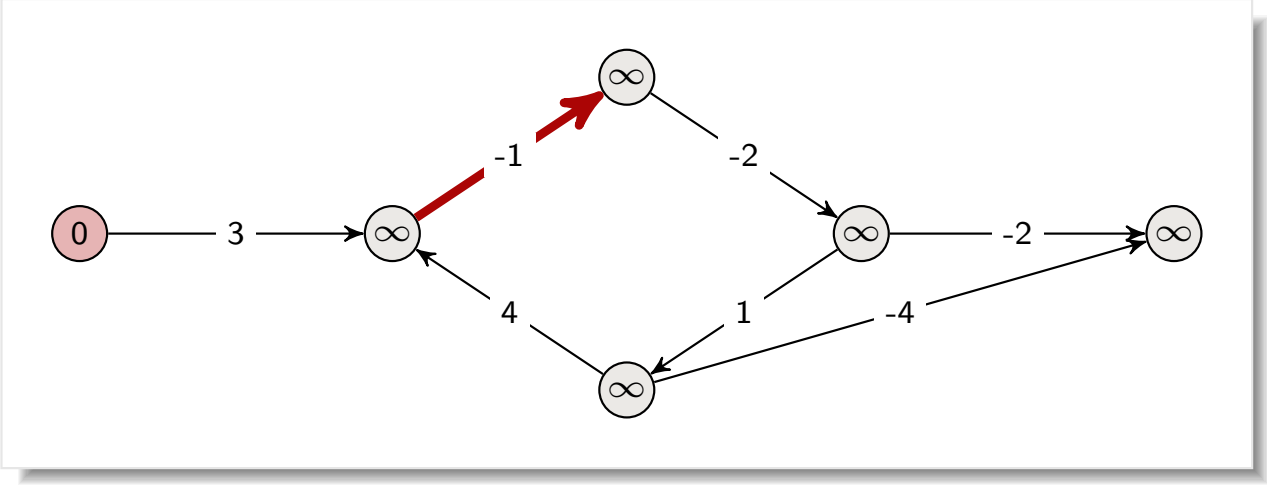
Bellman-Ford Algorithm

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)



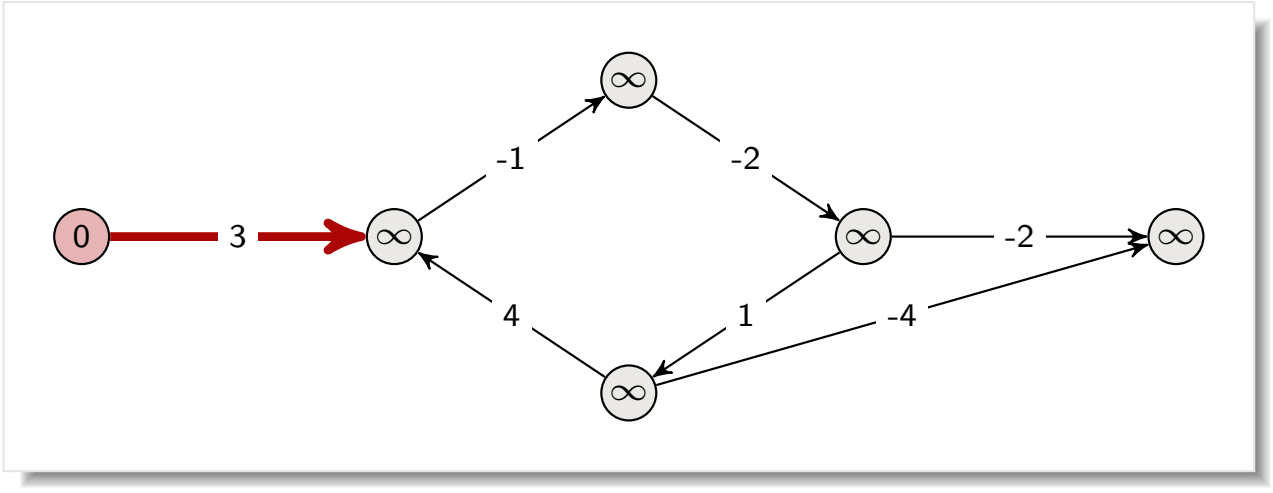
Bellman-Ford Algorithm

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)



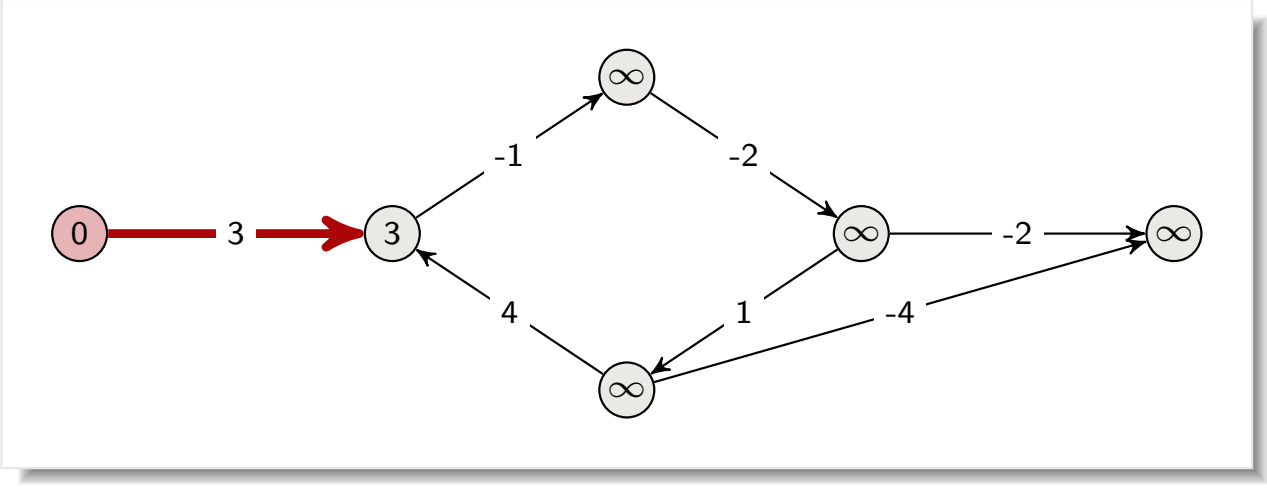
Bellman-Ford Algorithm

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



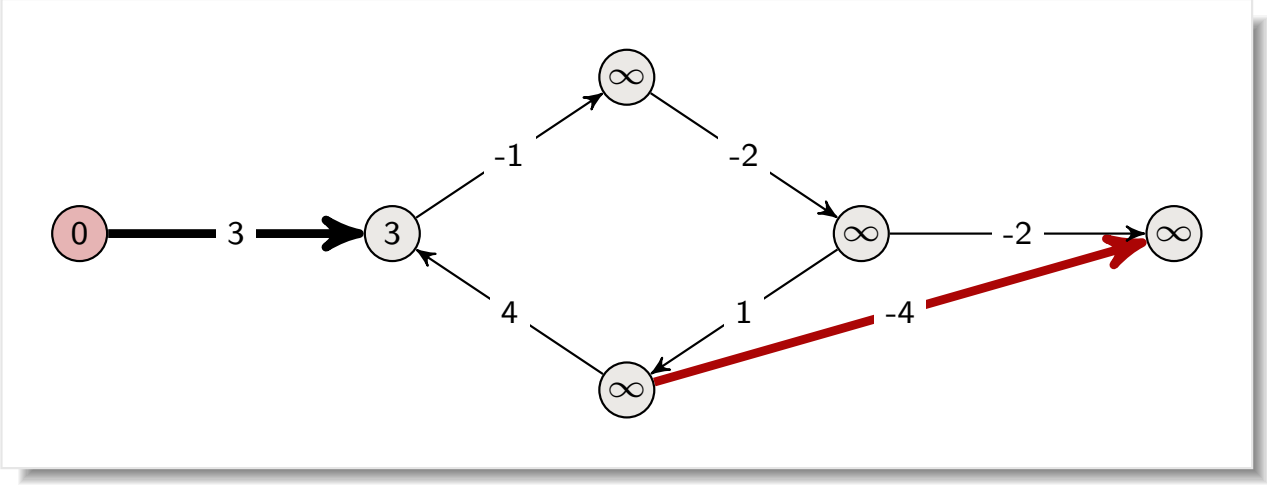
Bellman-Ford Algorithm

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)



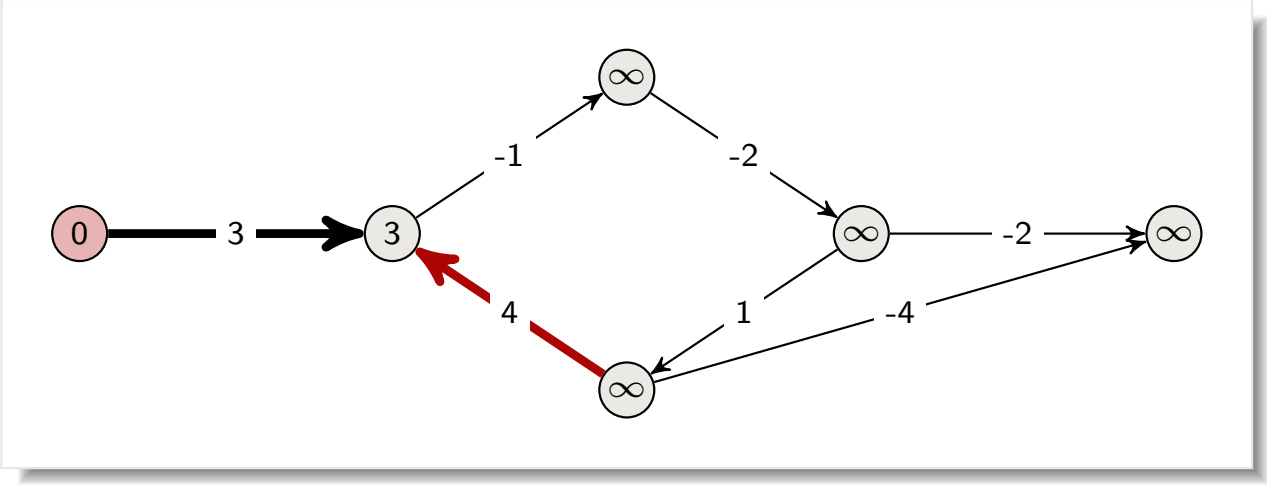
Bellman-Ford Algorithm

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)



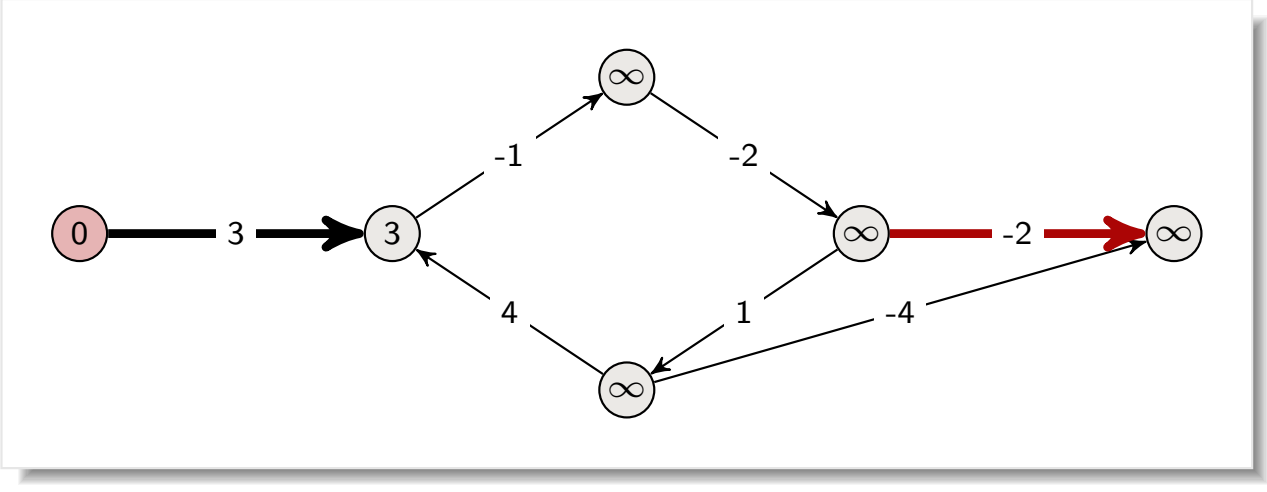
Bellman-Ford Algorithm

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)



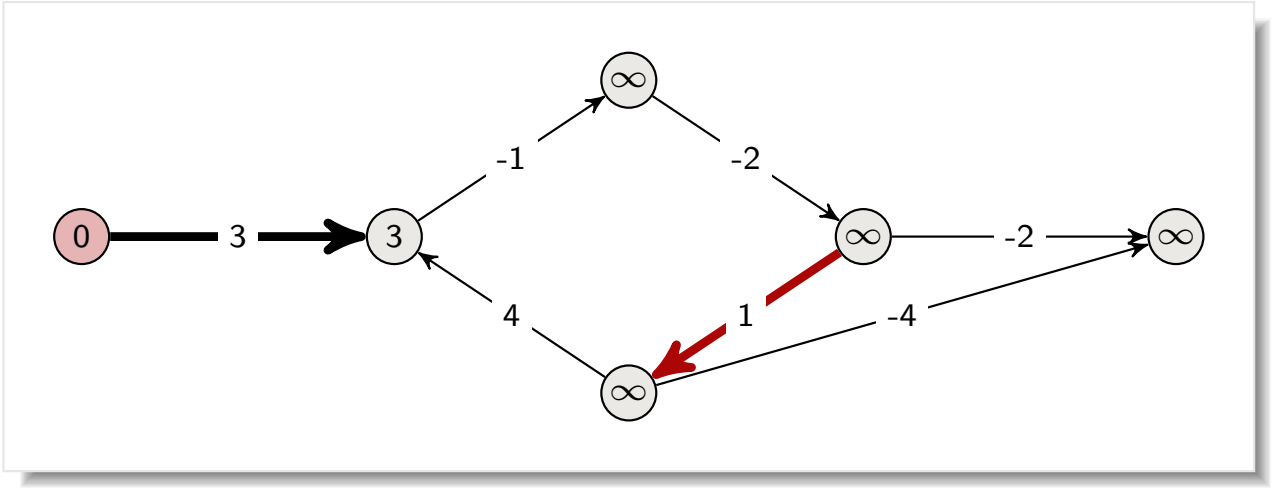
Bellman-Ford Algorithm

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)



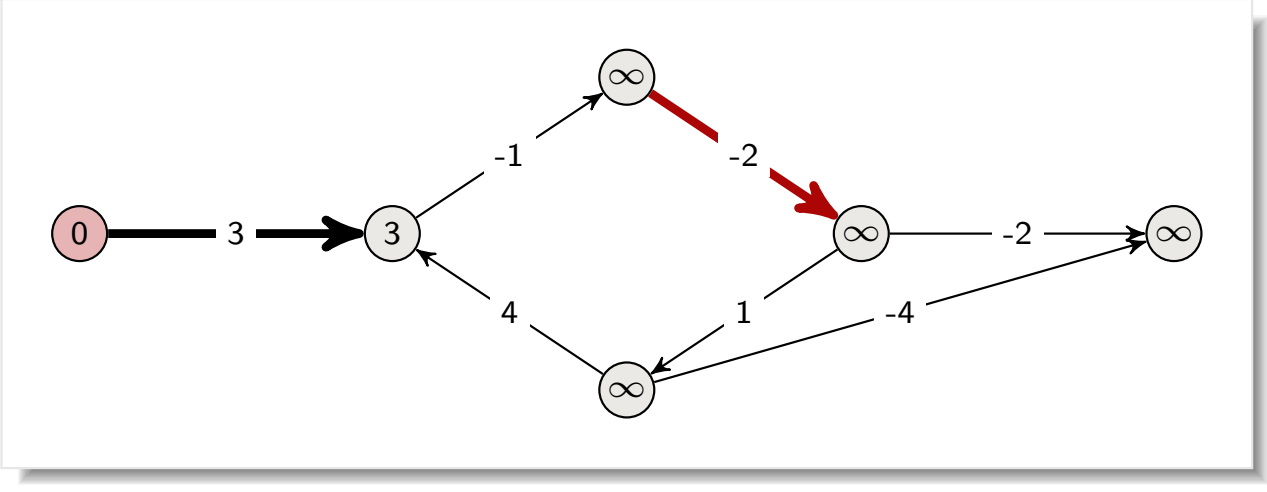
Bellman-Ford Algorithm

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



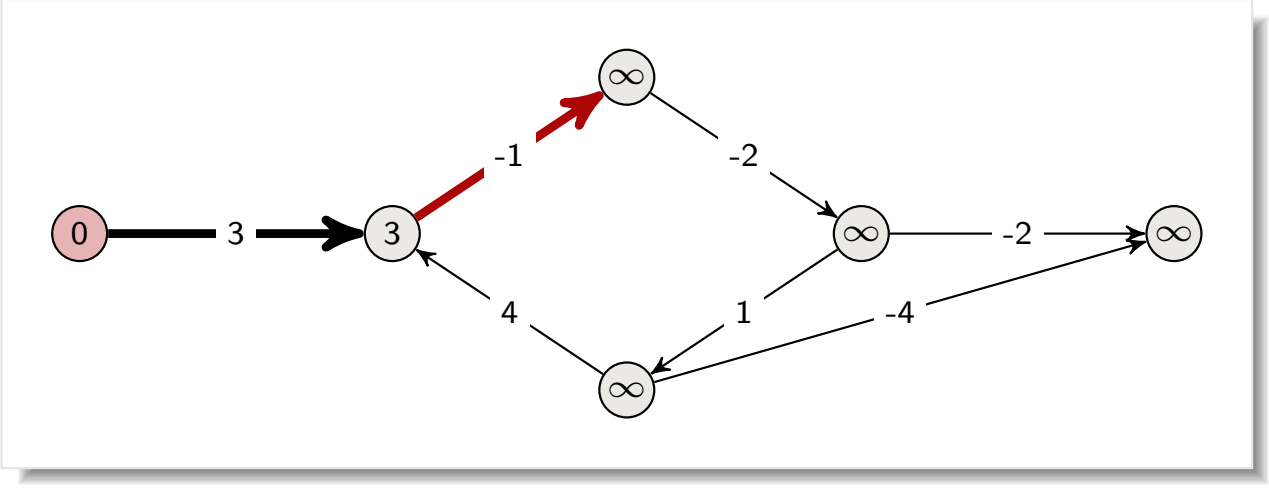
Bellman-Ford Algorithm

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)



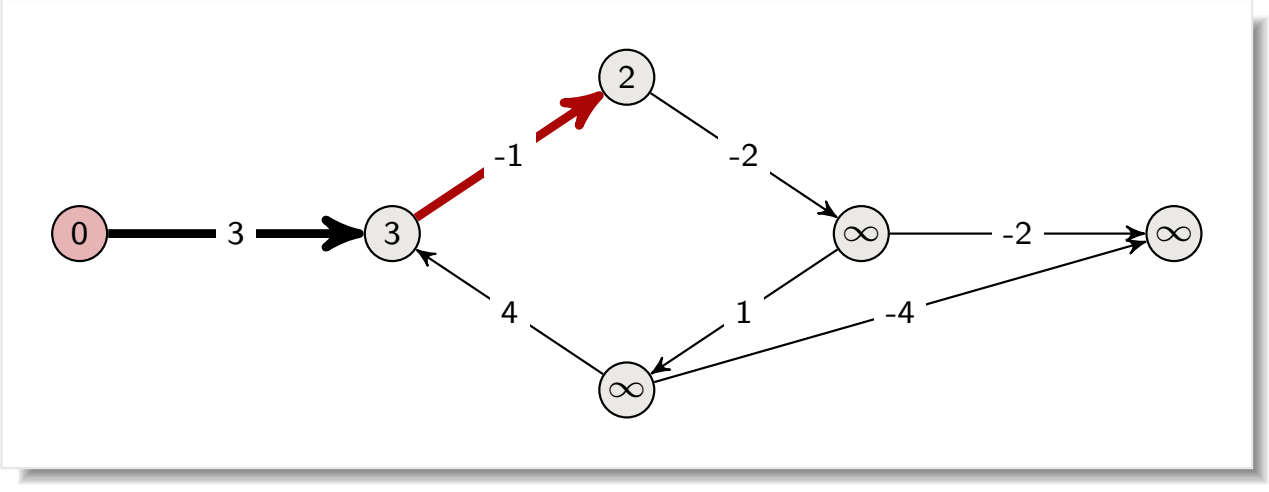
Bellman-Ford Algorithm

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)



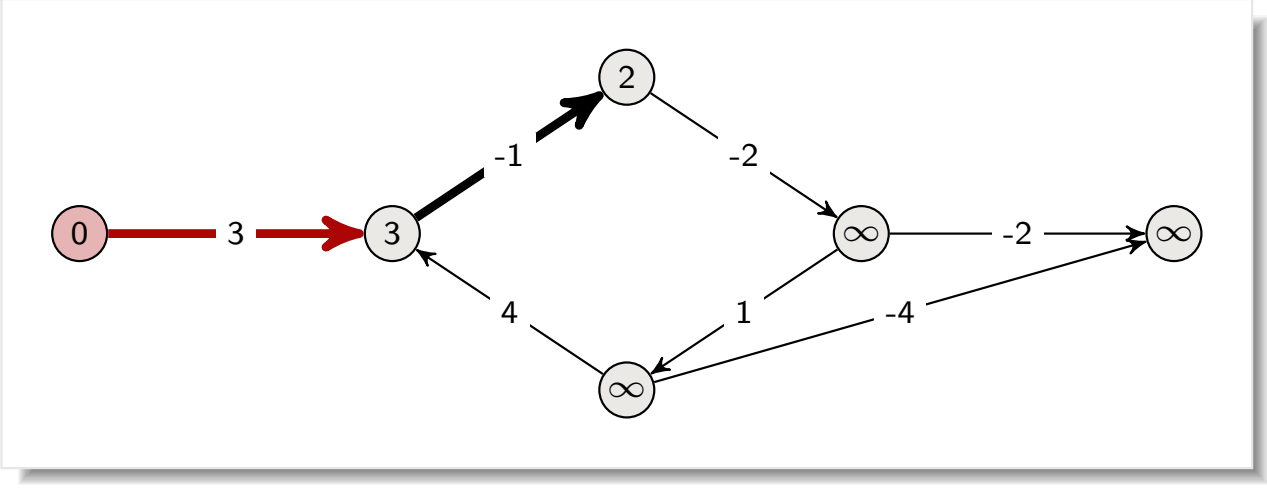
Bellman-Ford Algorithm

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)



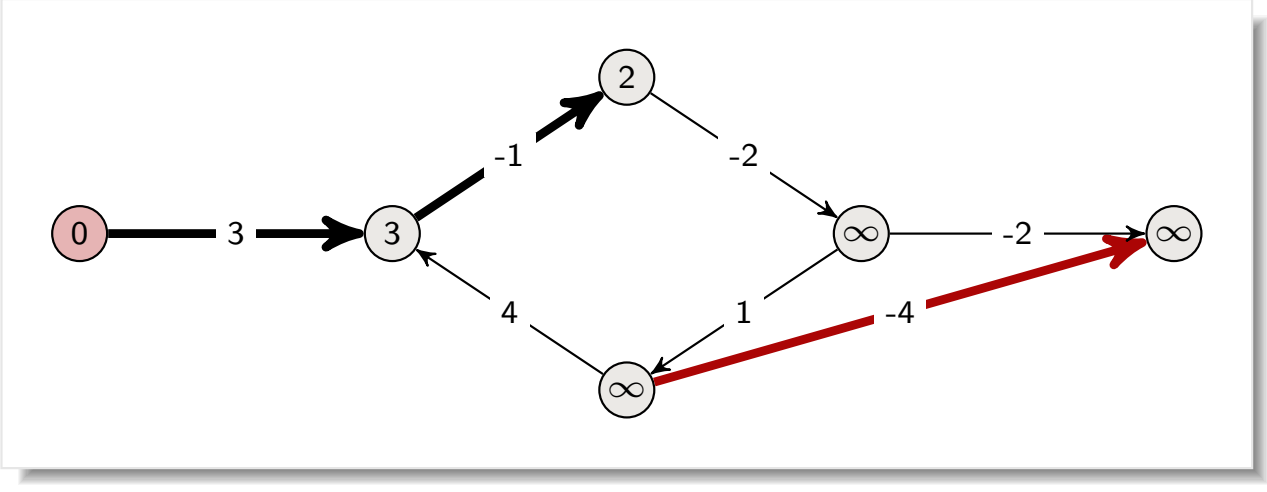
Bellman-Ford Algorithm

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)



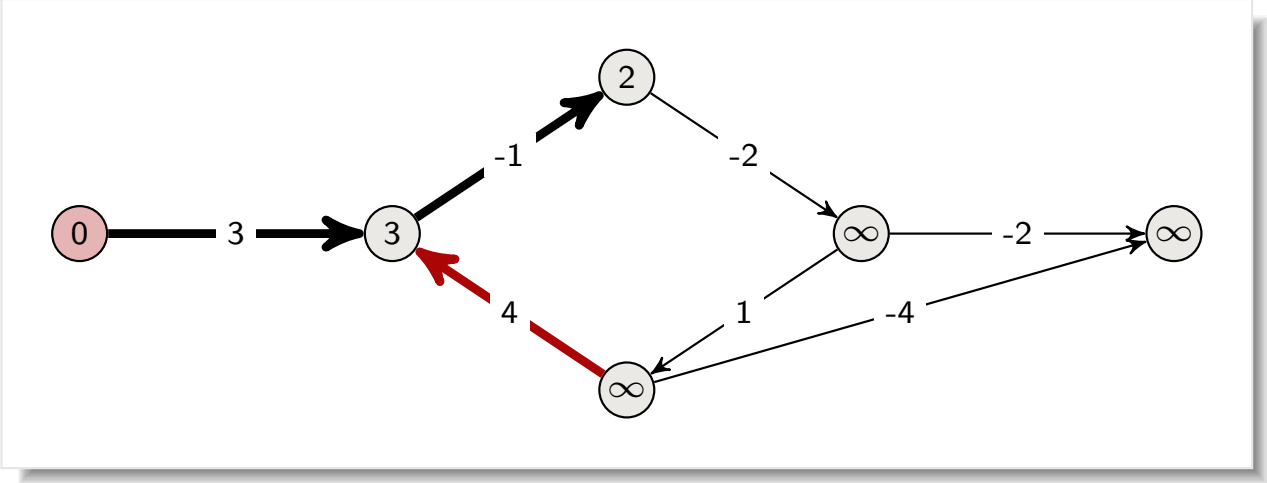
Bellman-Ford Algorithm

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)



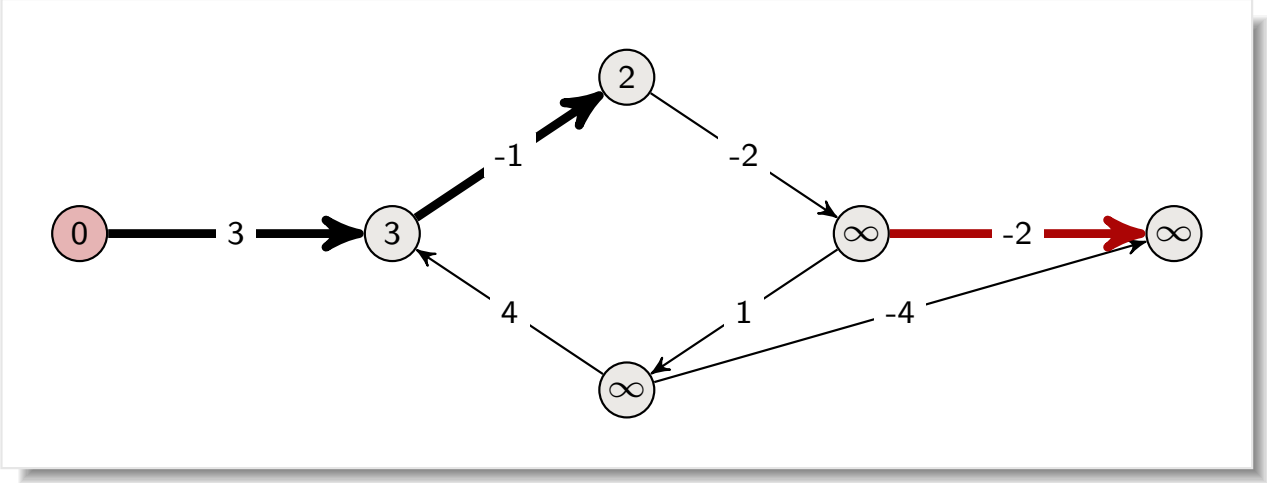
Bellman-Ford Algorithm

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)



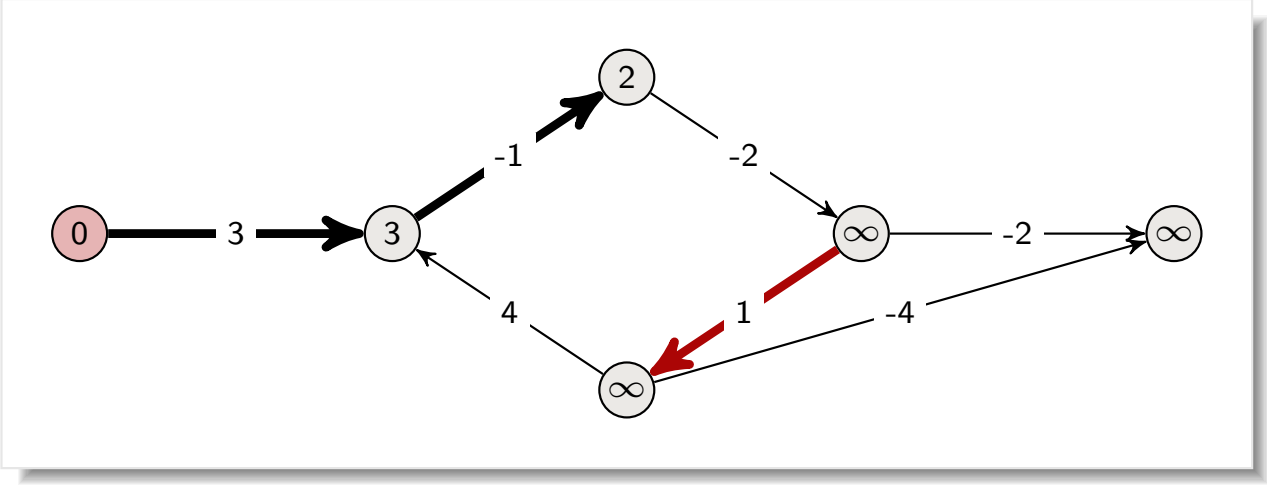
Bellman-Ford Algorithm

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)



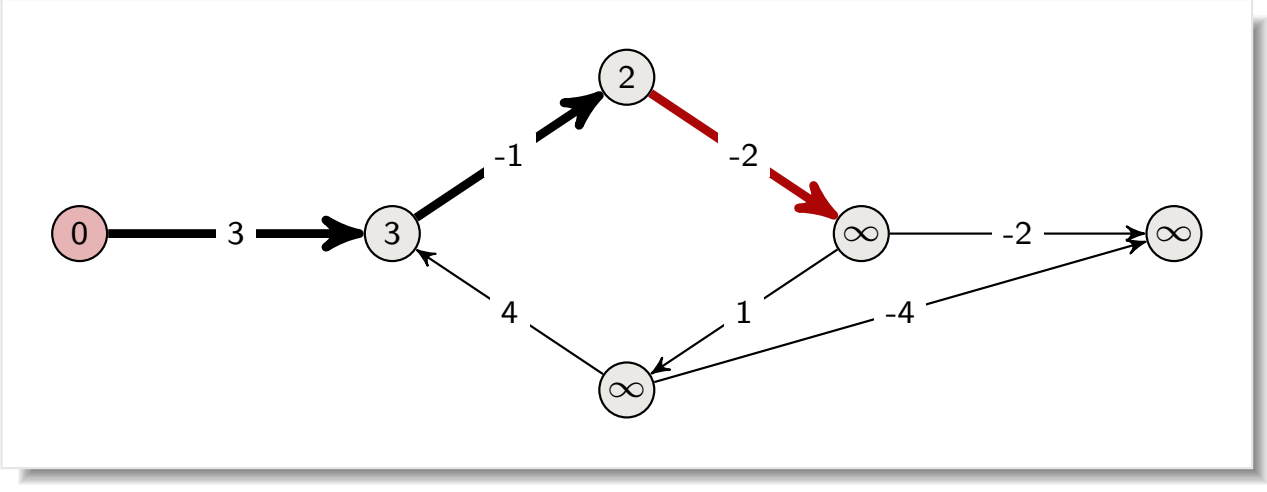
Bellman-Ford Algorithm

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)



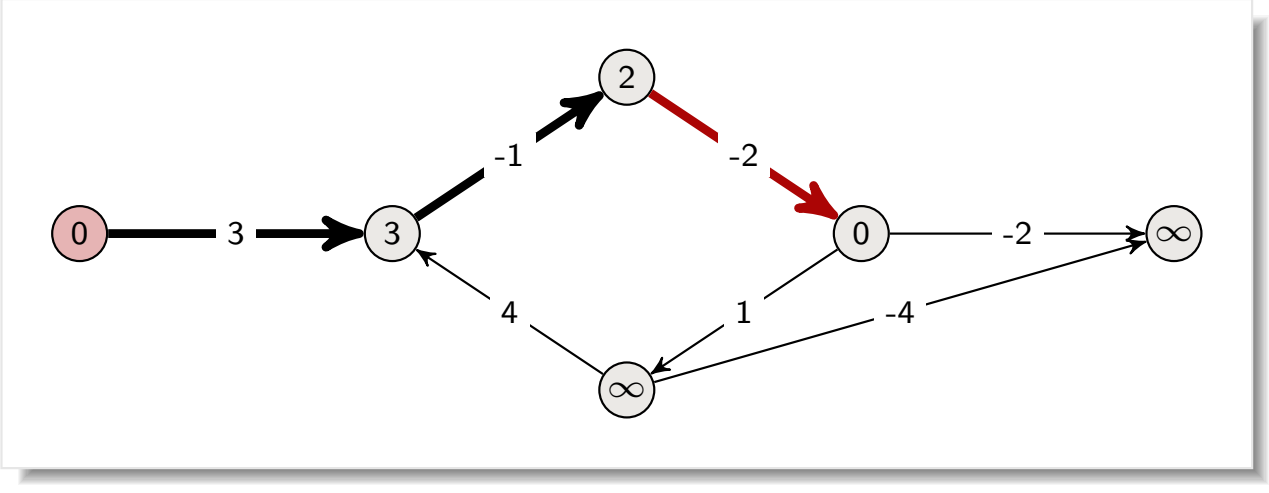
Bellman-Ford Algorithm

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)



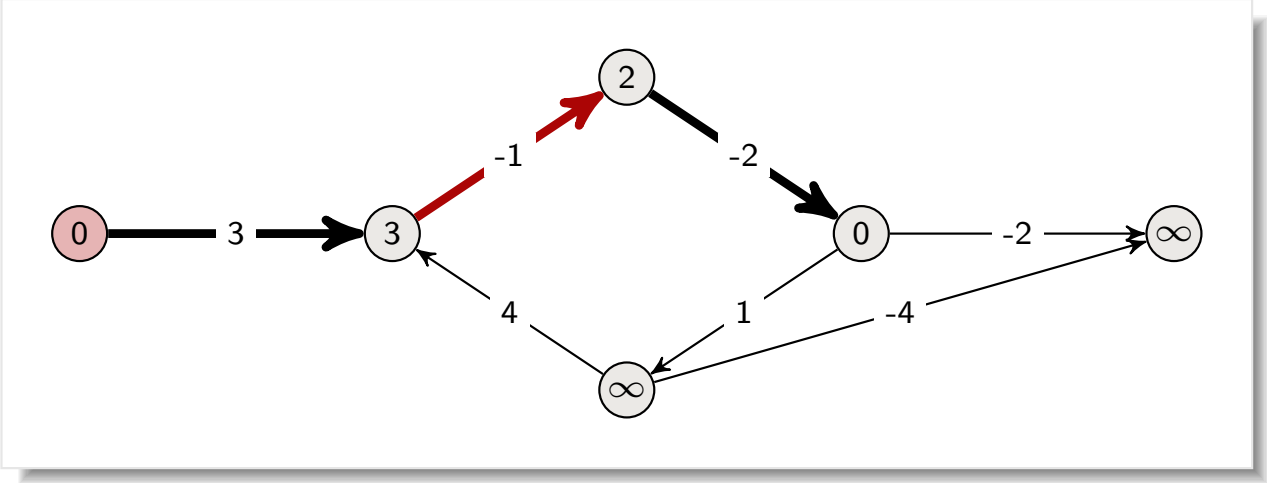
Bellman-Ford Algorithm

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



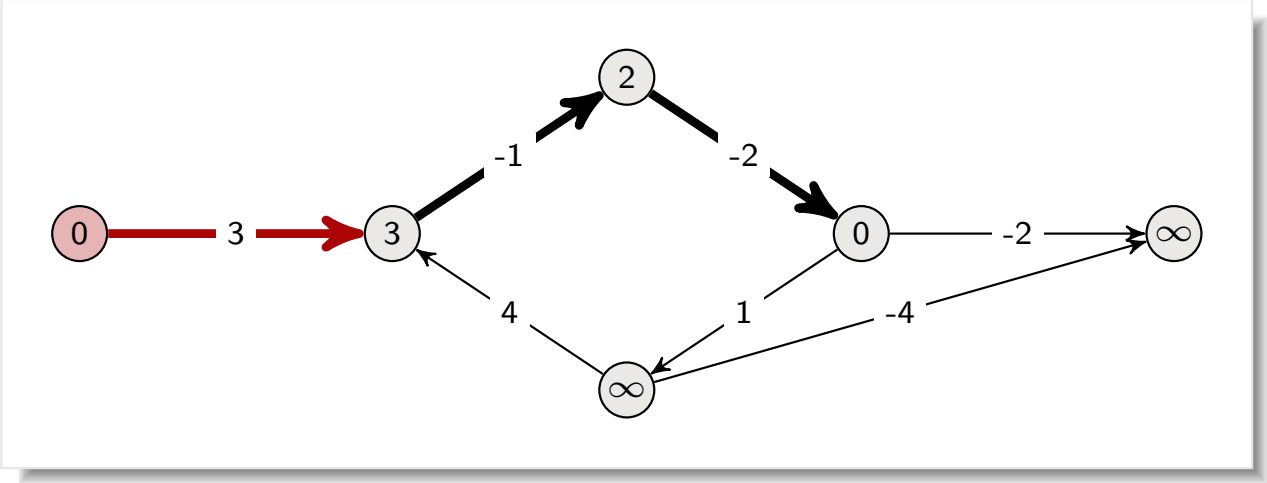
Bellman-Ford Algorithm

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)



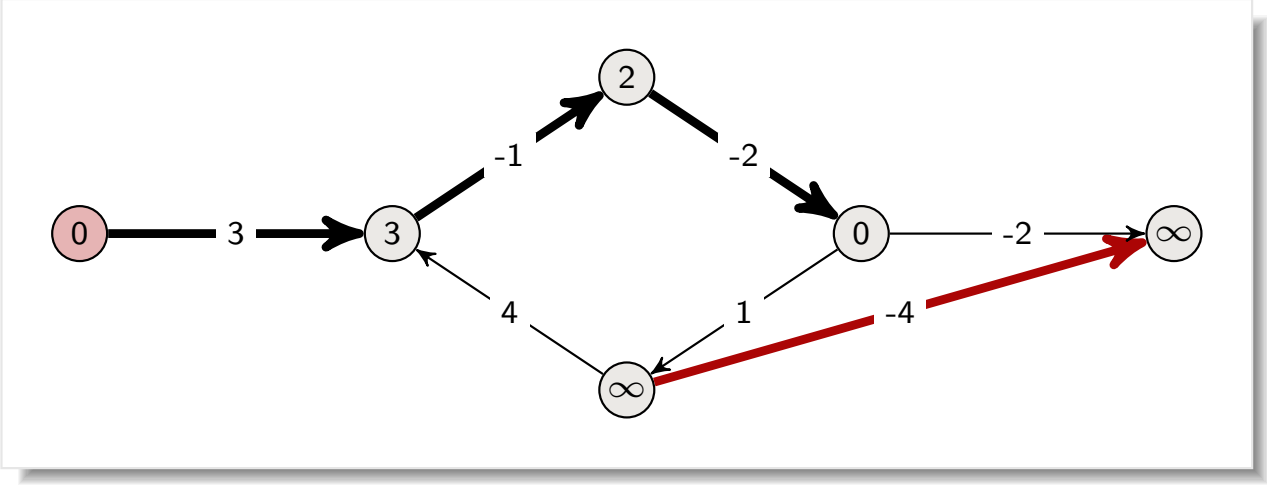
Bellman-Ford Algorithm

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)



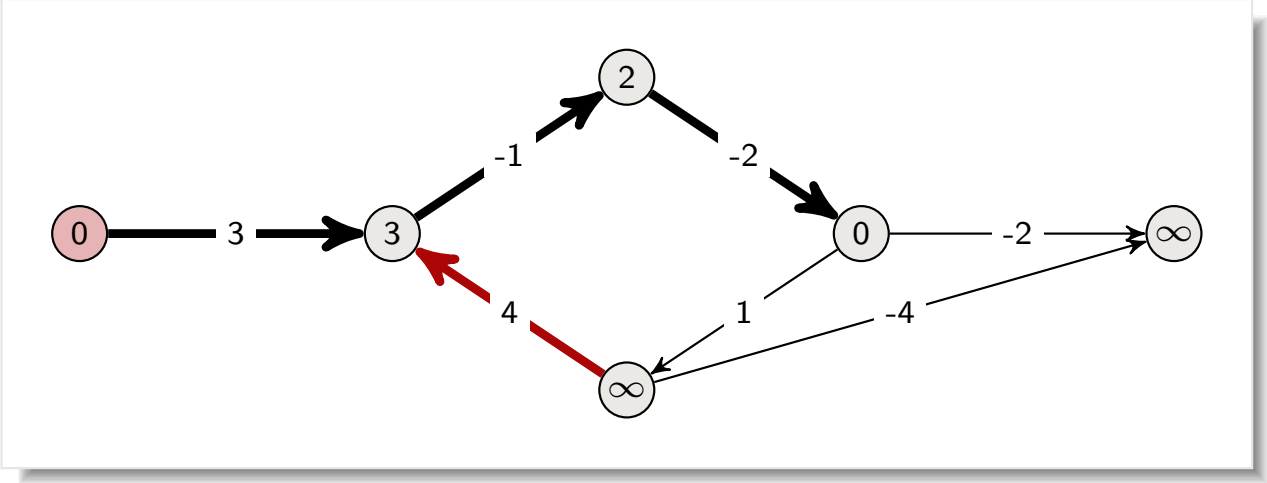
Bellman-Ford Algorithm

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)



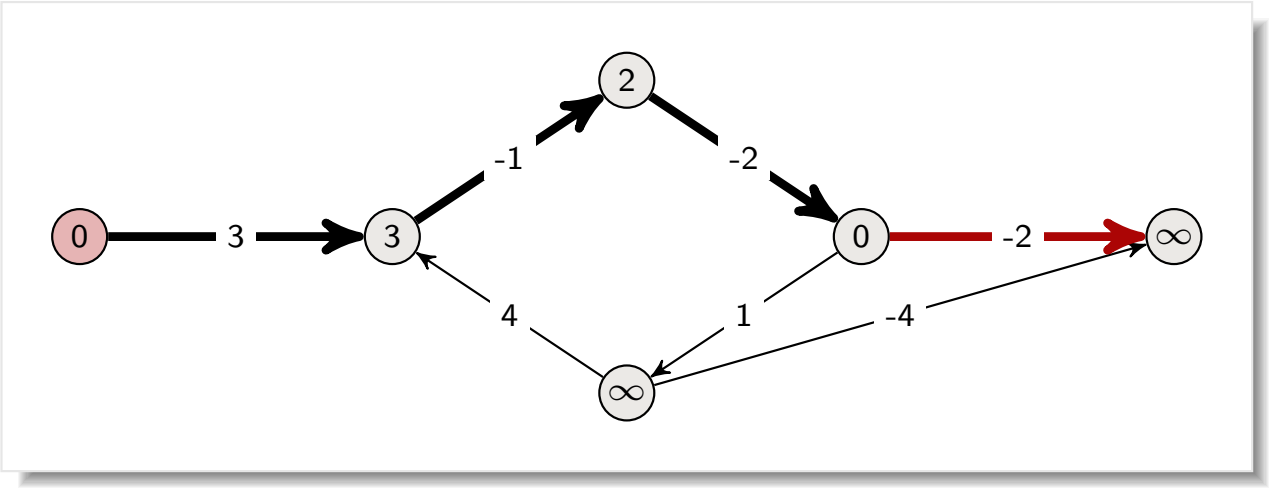
Bellman-Ford Algorithm

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)



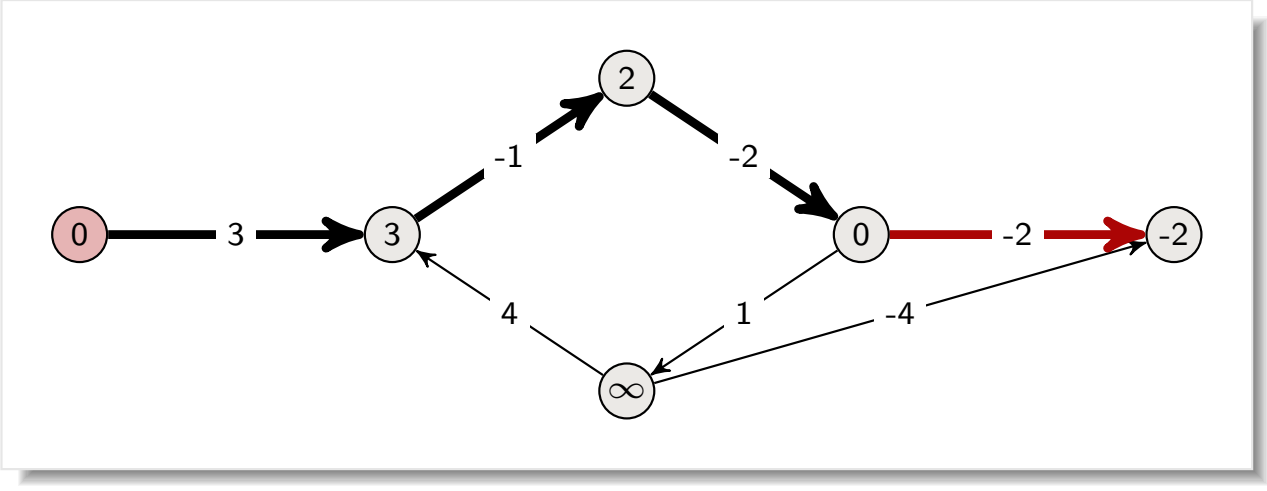
Bellman-Ford Algorithm

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



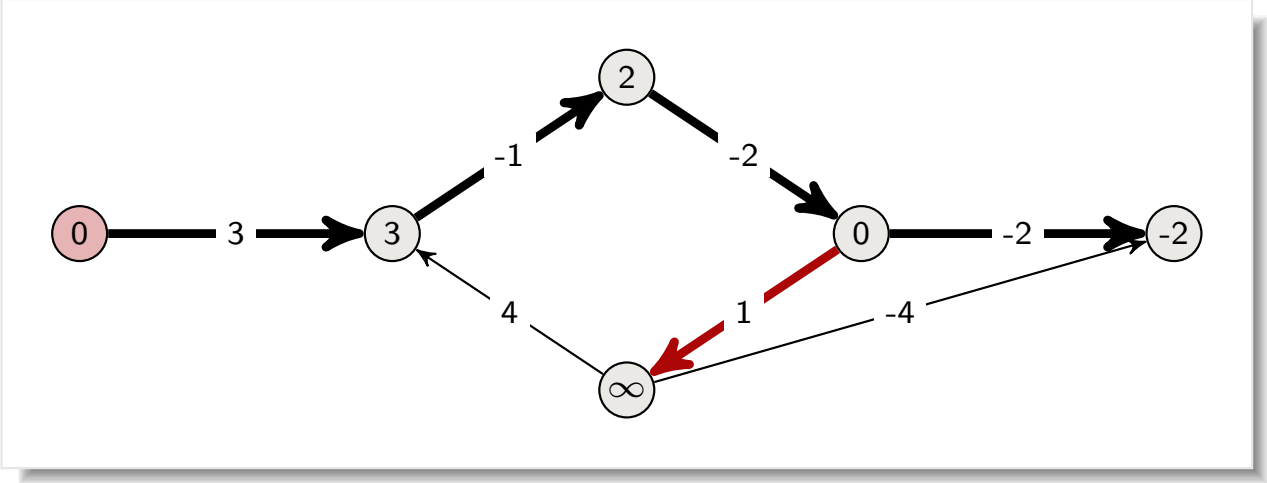
Bellman-Ford Algorithm

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



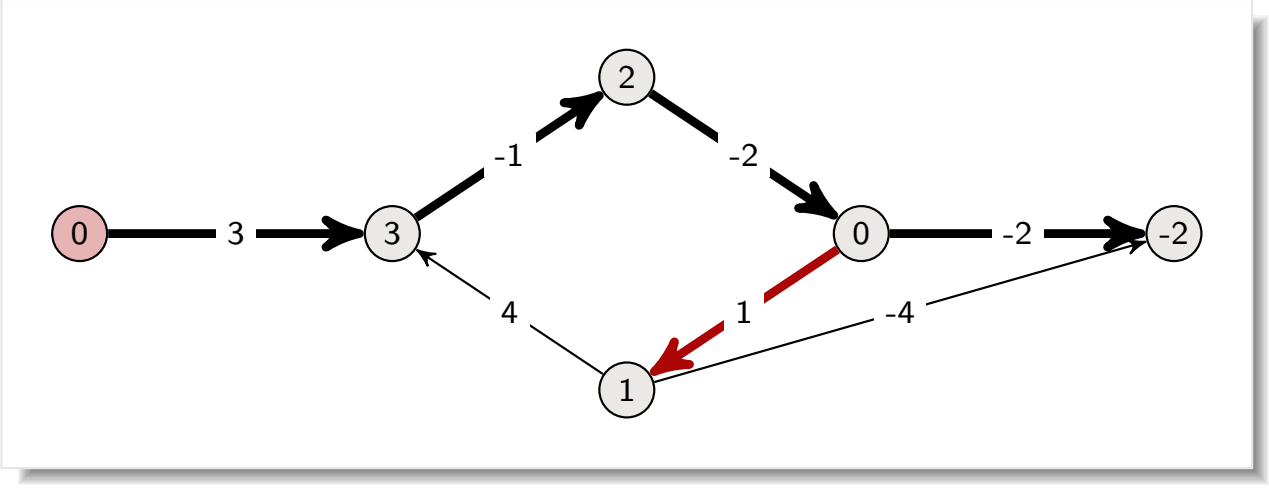
Bellman-Ford Algorithm

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)



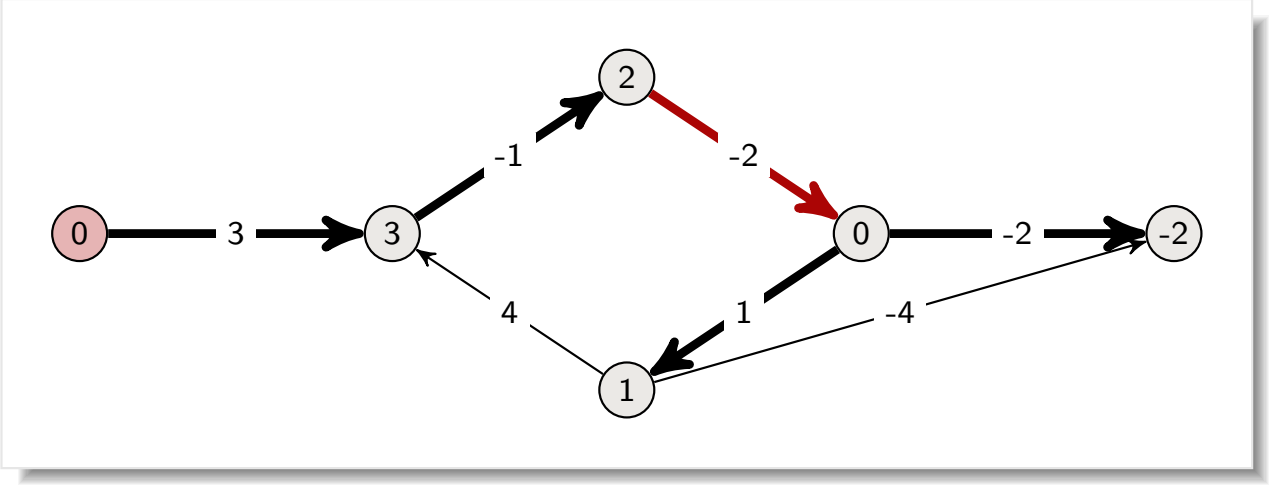
Bellman-Ford Algorithm

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)



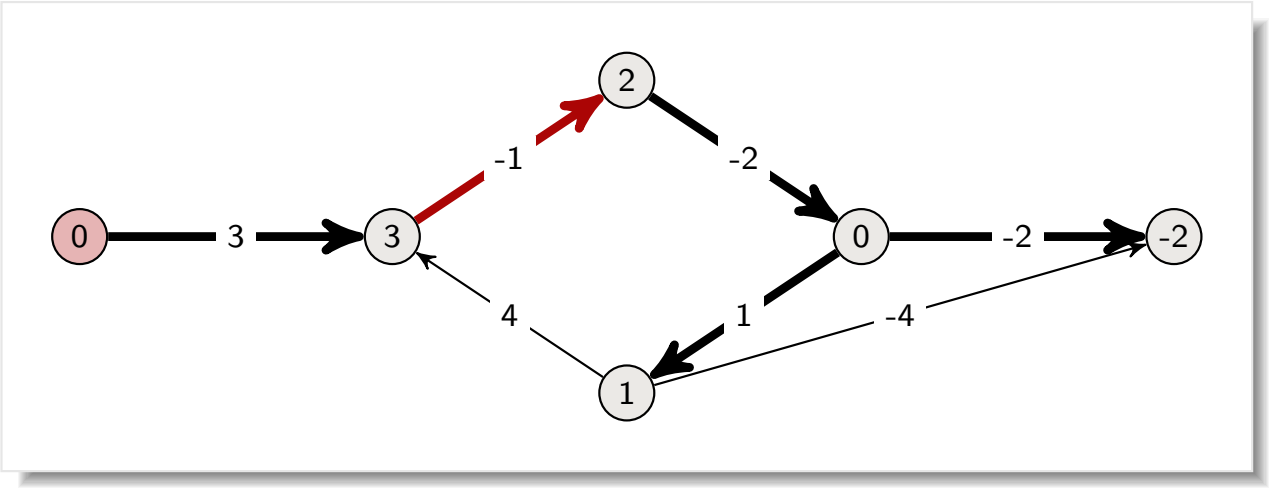
Bellman-Ford Algorithm

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)



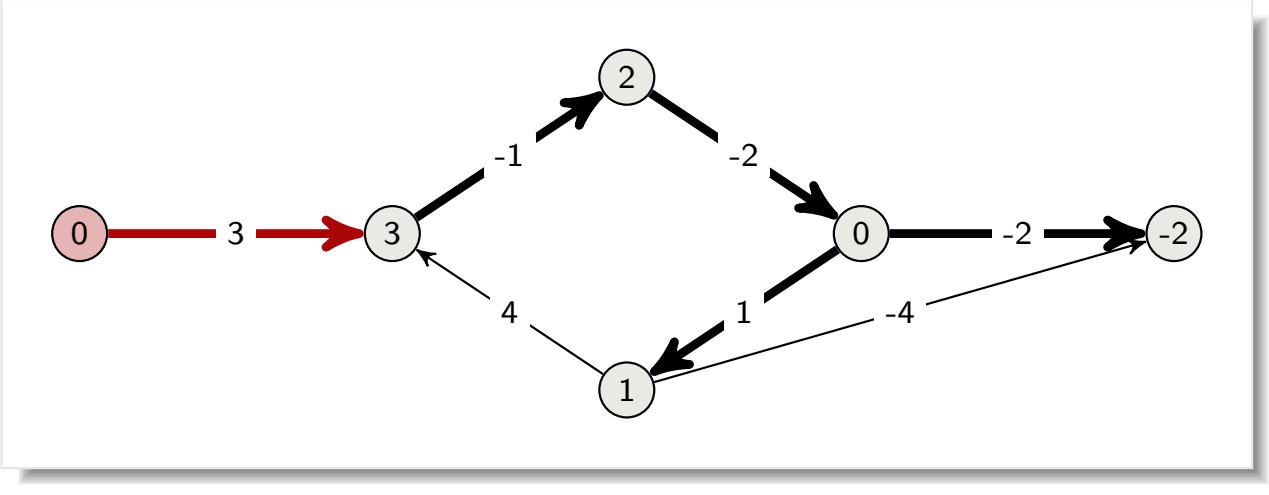
Bellman-Ford Algorithm

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



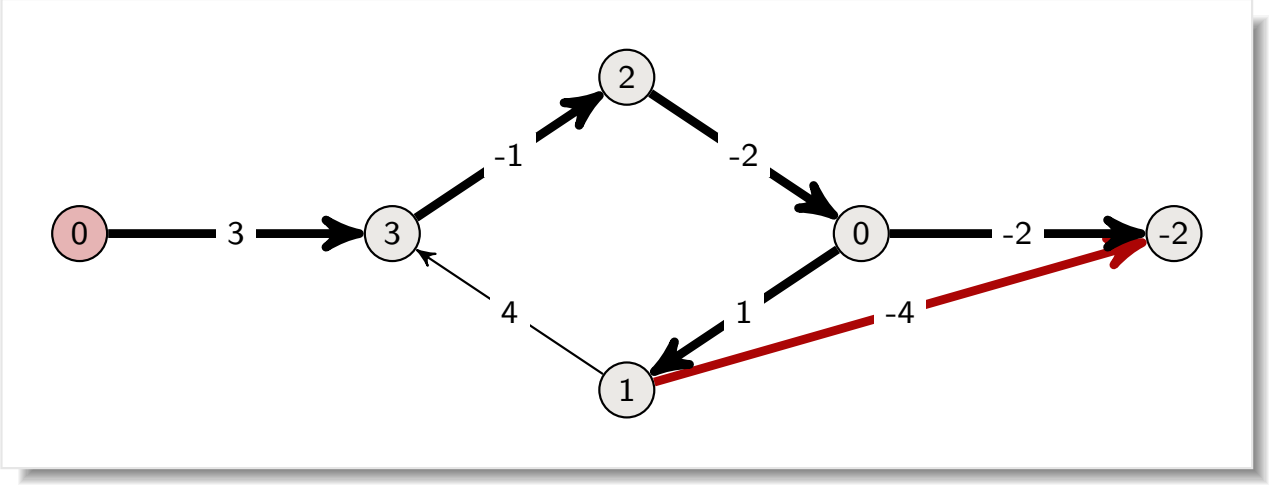
Bellman-Ford Algorithm

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



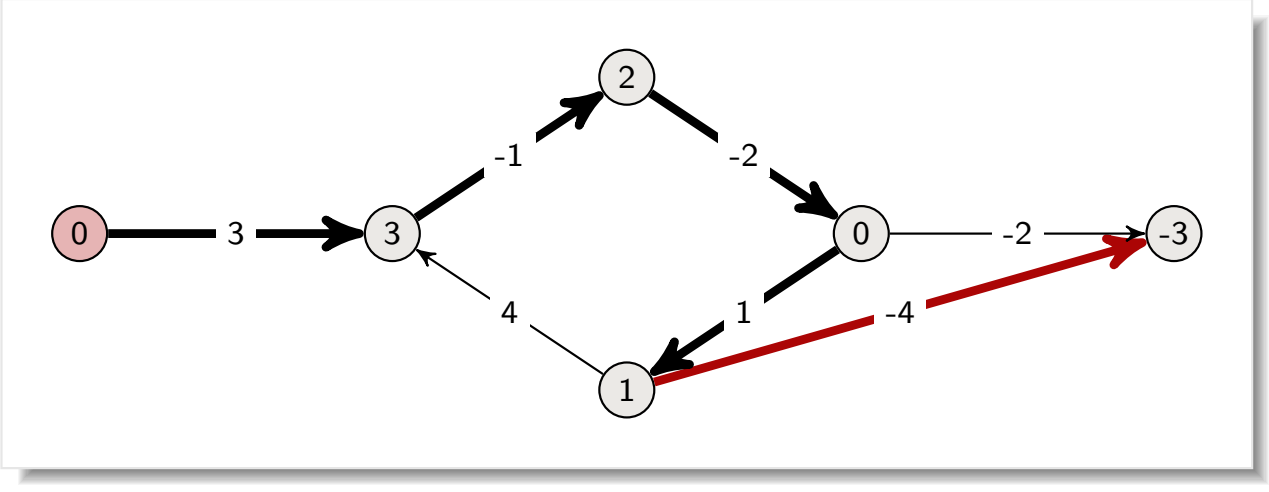
Bellman-Ford Algorithm

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)



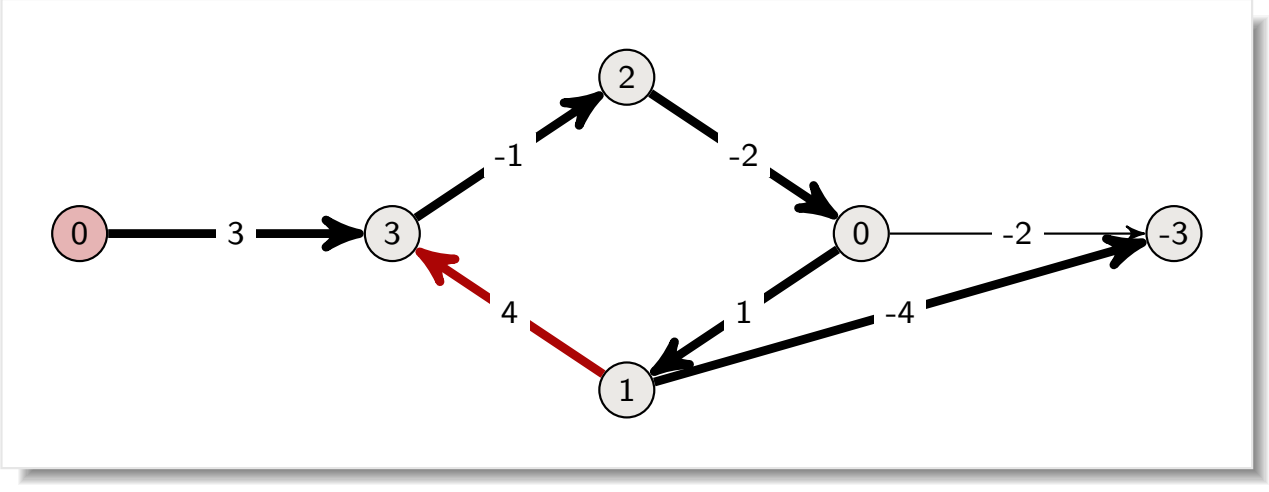
Bellman-Ford Algorithm

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)



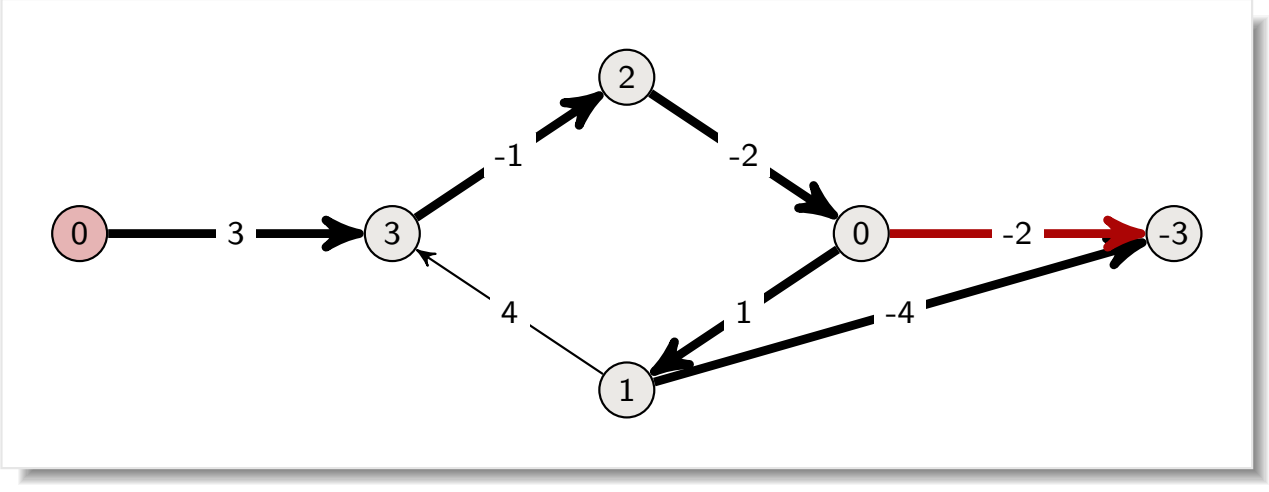
Bellman-Ford Algorithm

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)



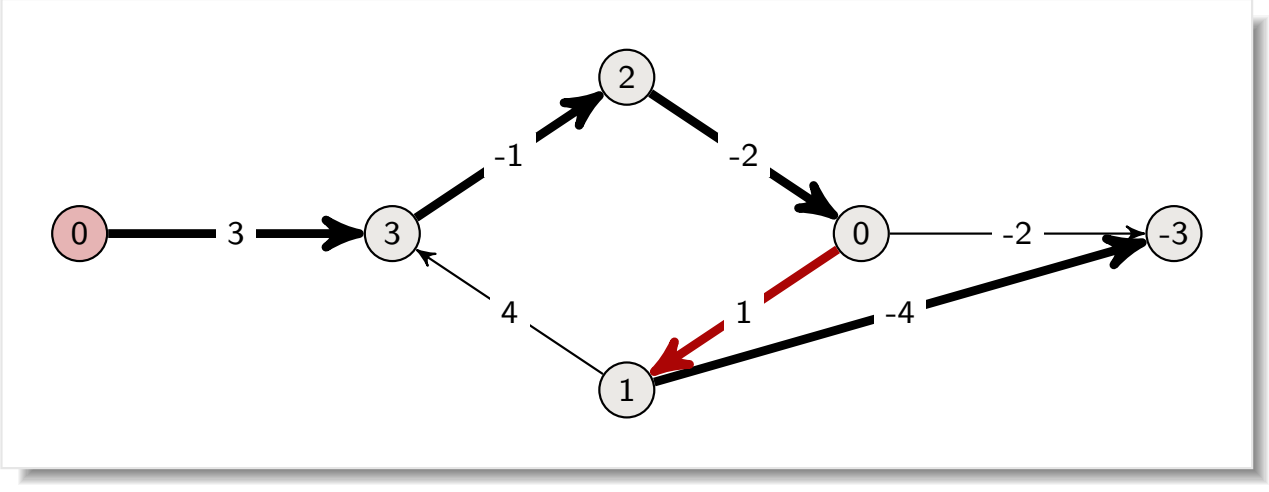
Bellman-Ford Algorithm

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)



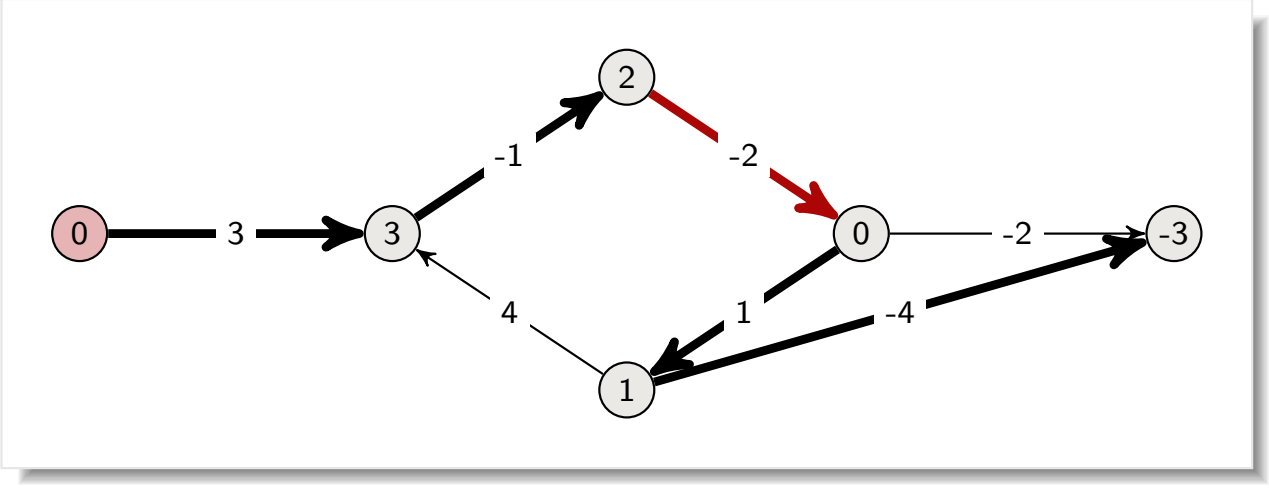
Bellman-Ford Algorithm

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)



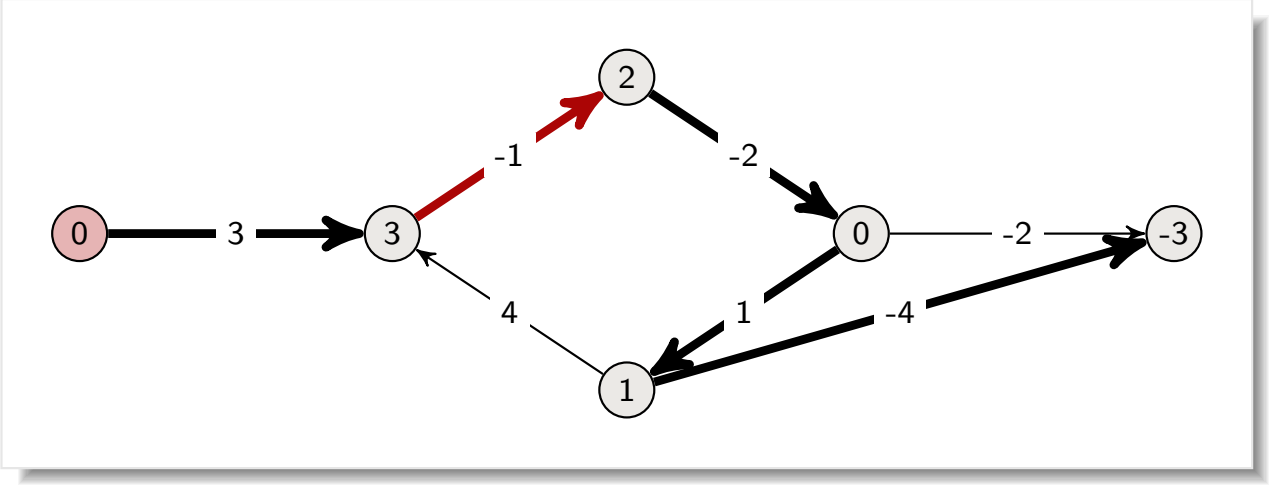
Bellman-Ford Algorithm

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)



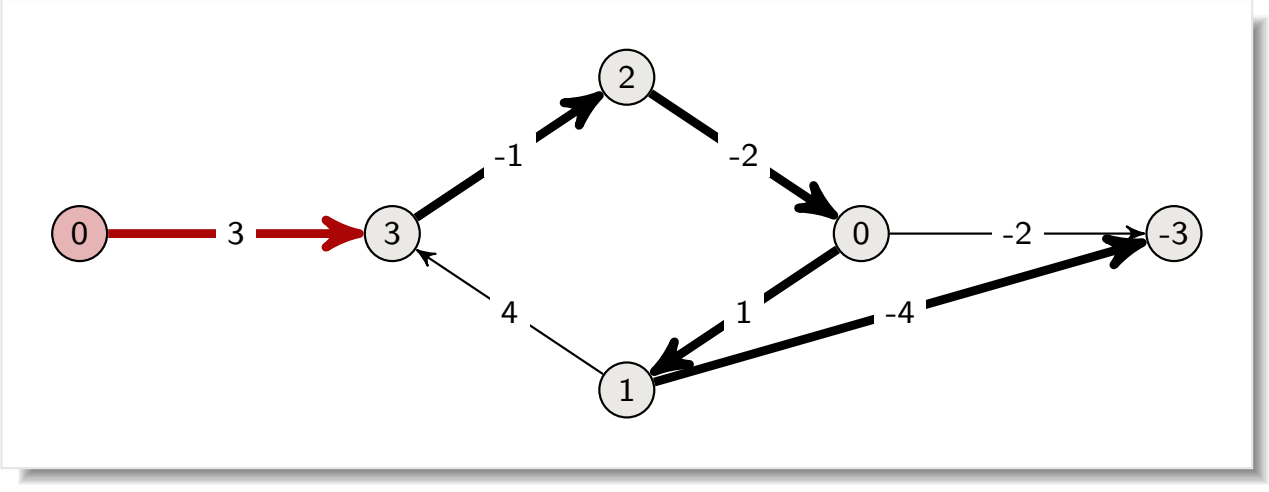
Bellman-Ford Algorithm

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)



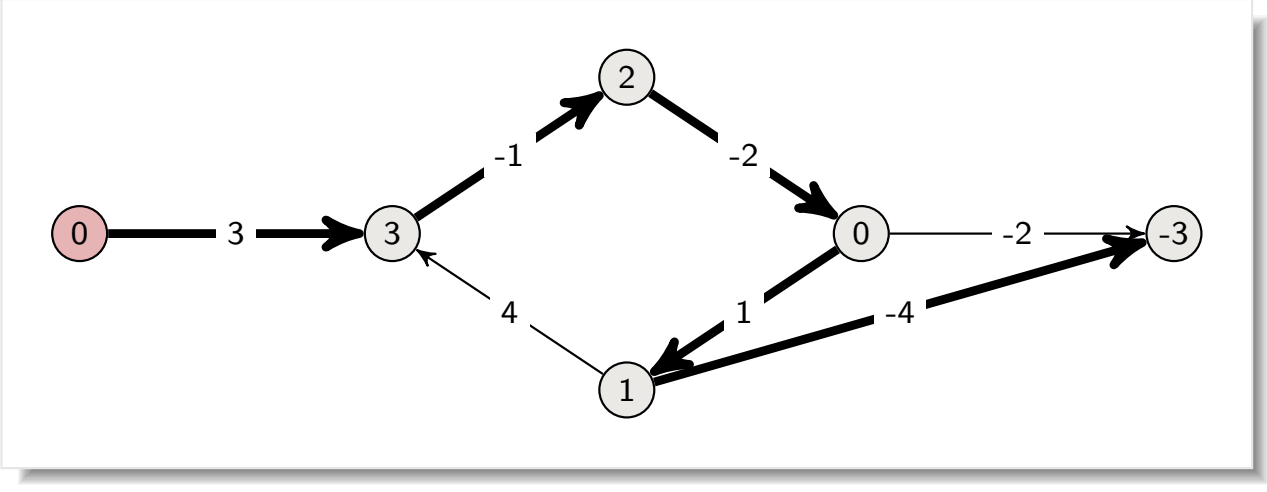
Bellman-Ford Algorithm

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)



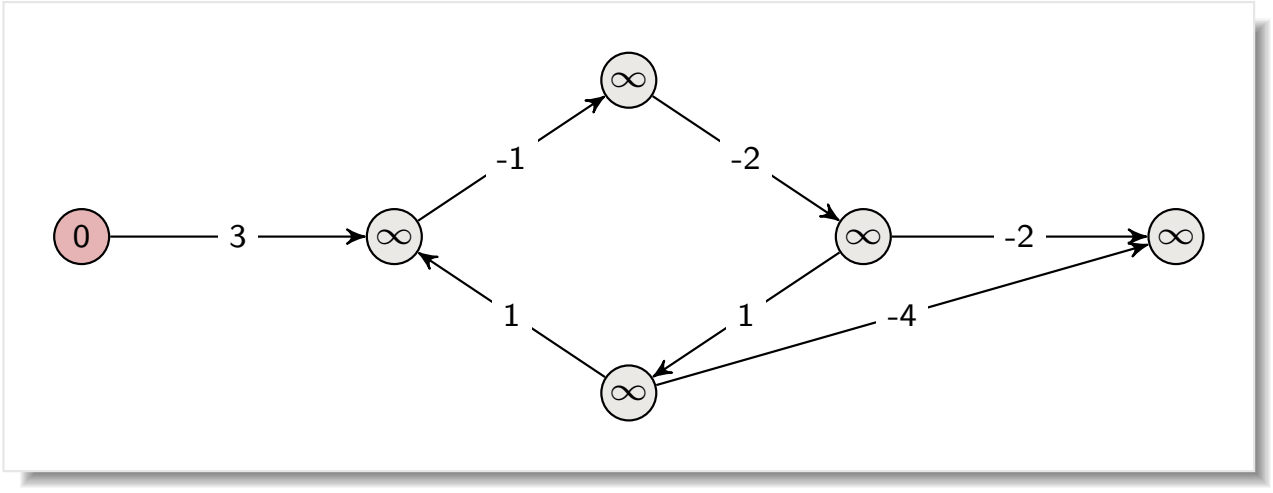
Bellman-Ford Algorithm

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)



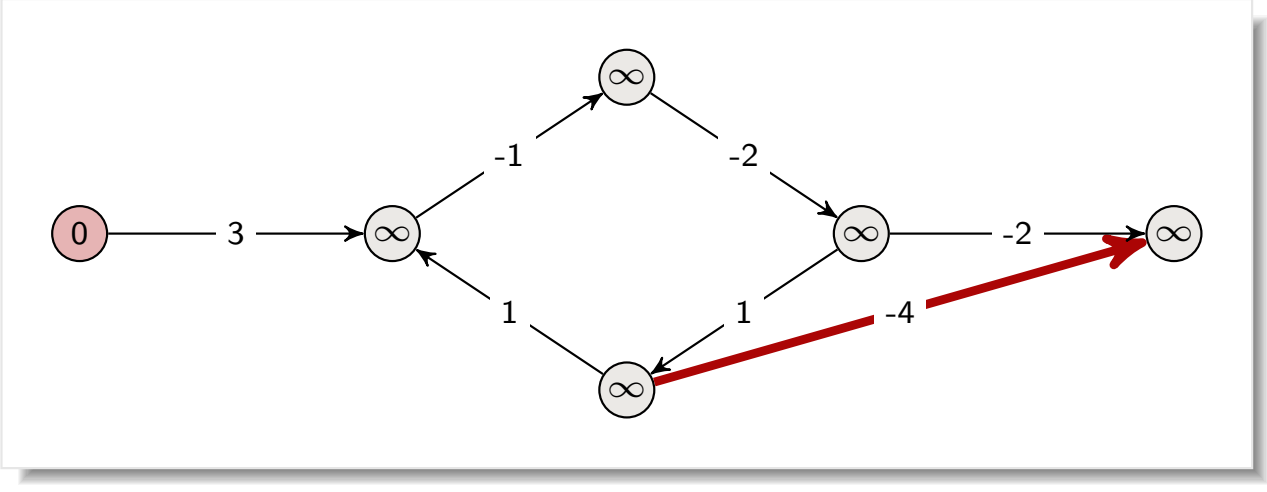
Bellman-Ford Algorithm

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



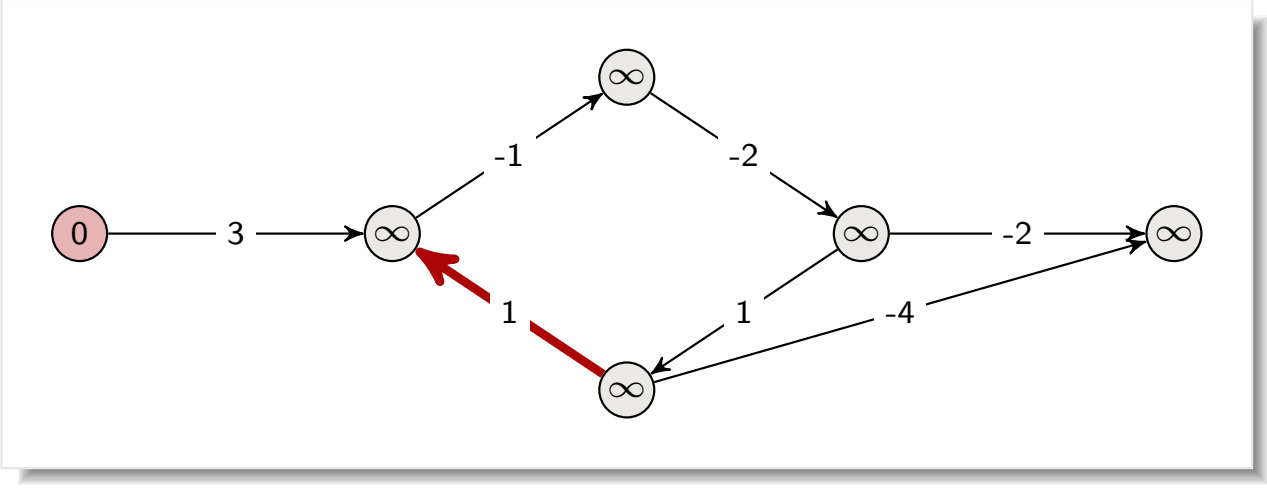
Bellman-Ford Algorithm

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)



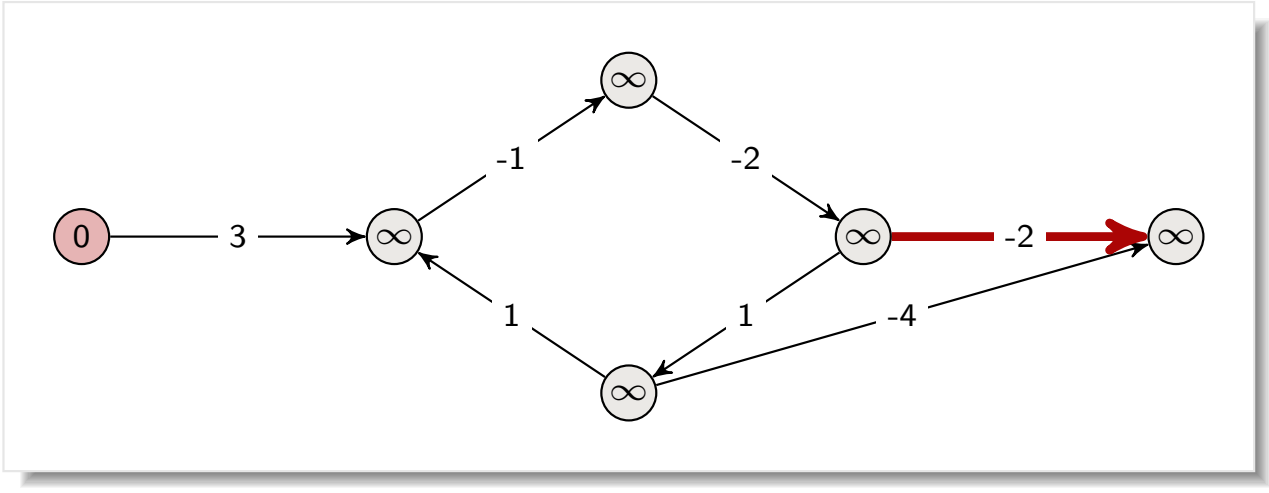
Bellman-Ford Algorithm

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)



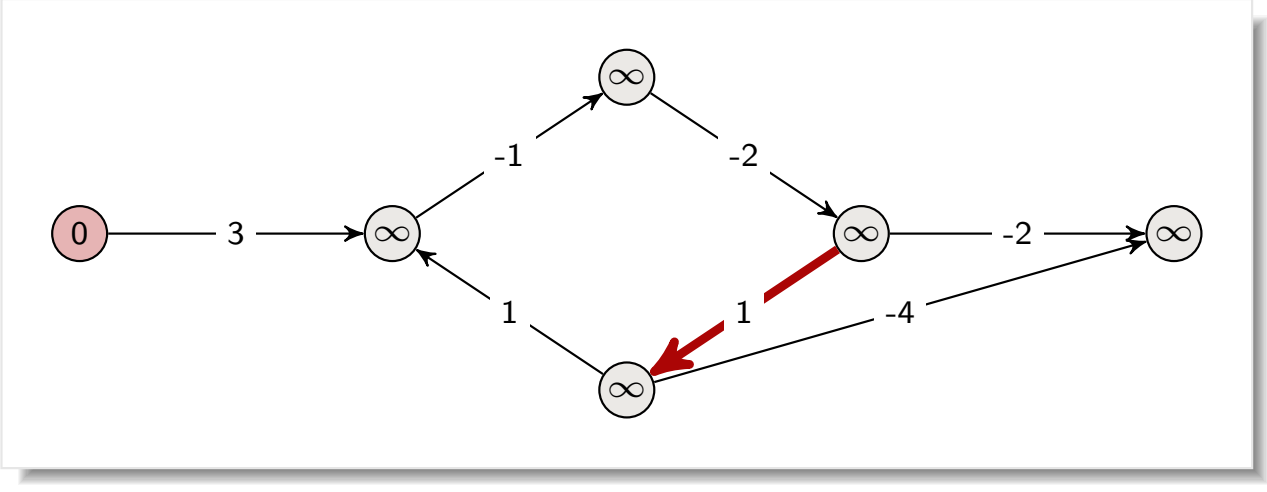
Bellman-Ford Algorithm

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



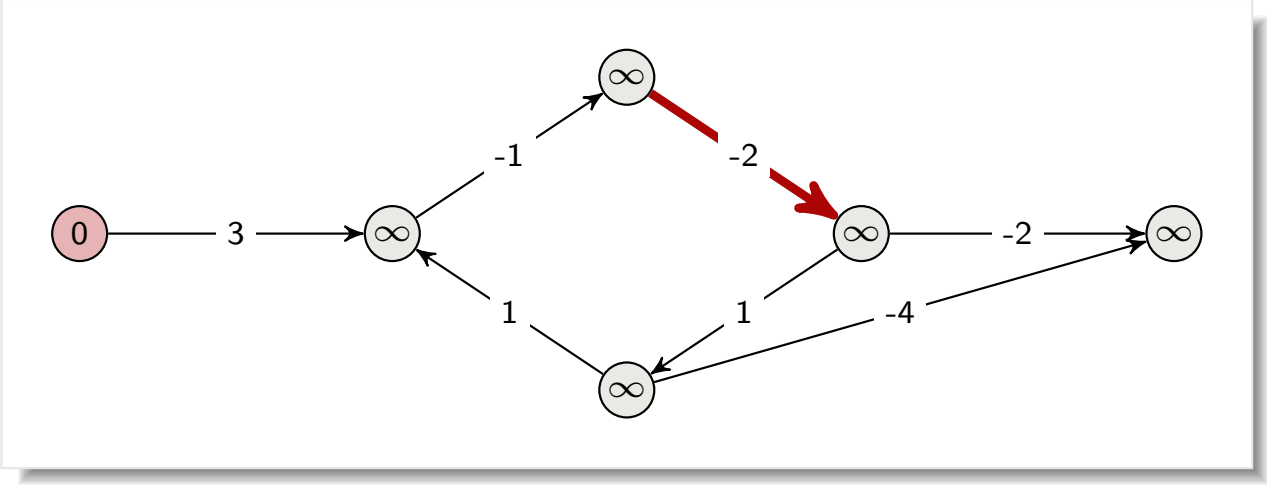
Bellman-Ford Algorithm

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)



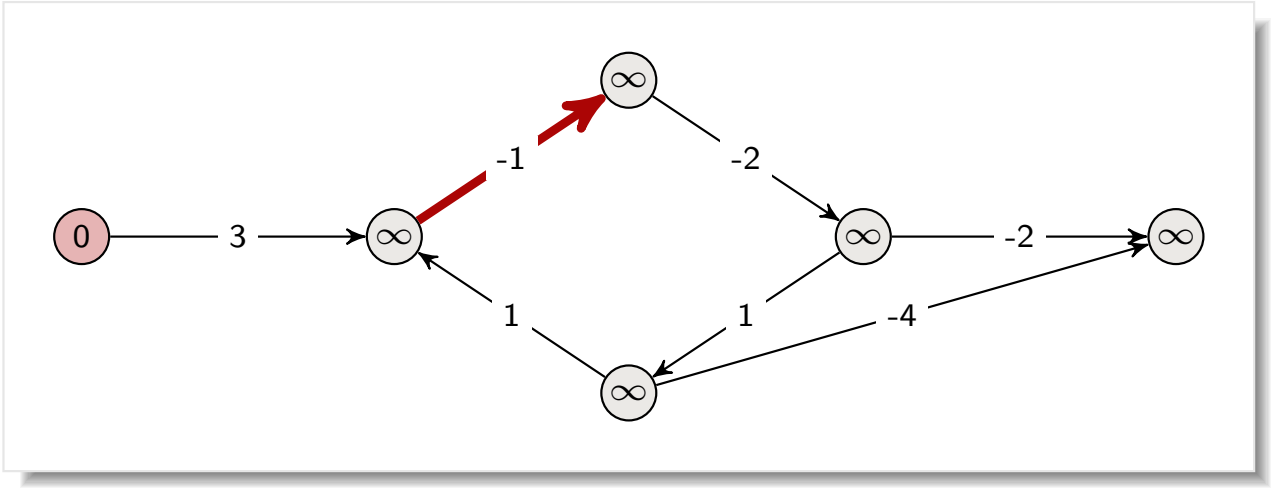
Bellman-Ford Algorithm

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)



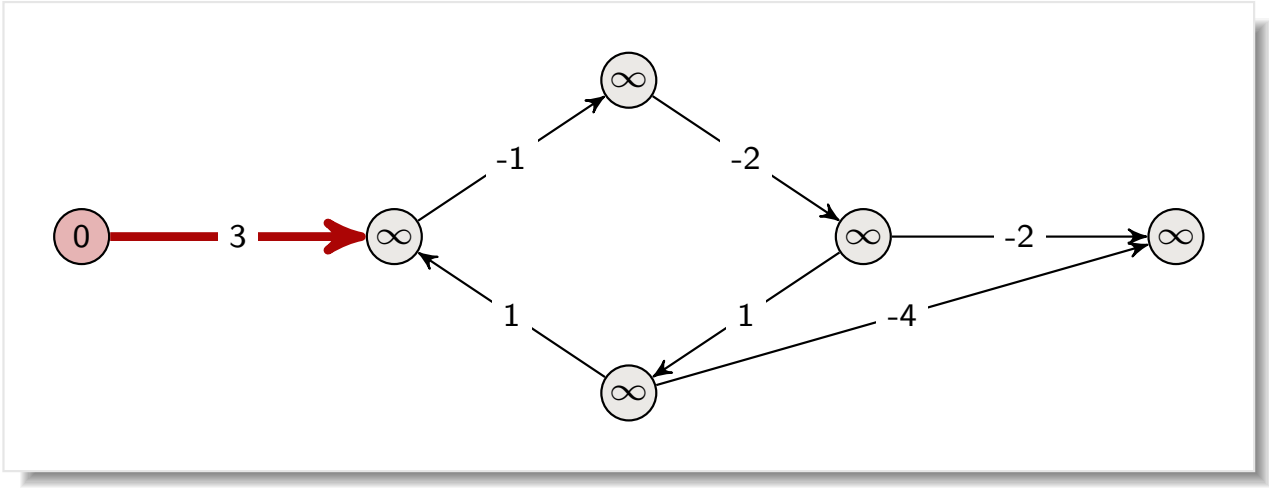
Bellman-Ford Algorithm

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



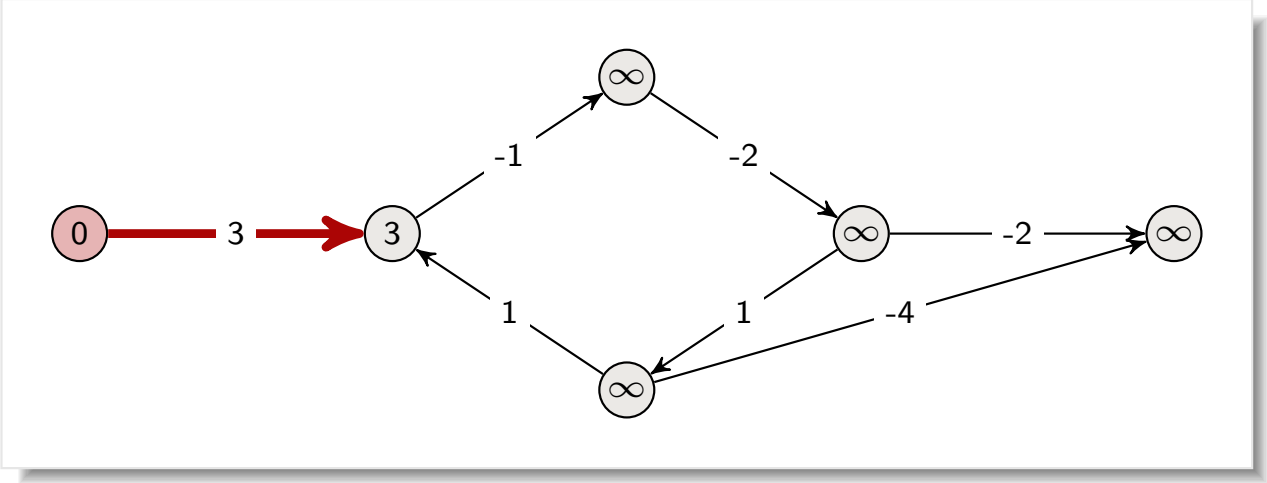
Bellman-Ford Algorithm

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



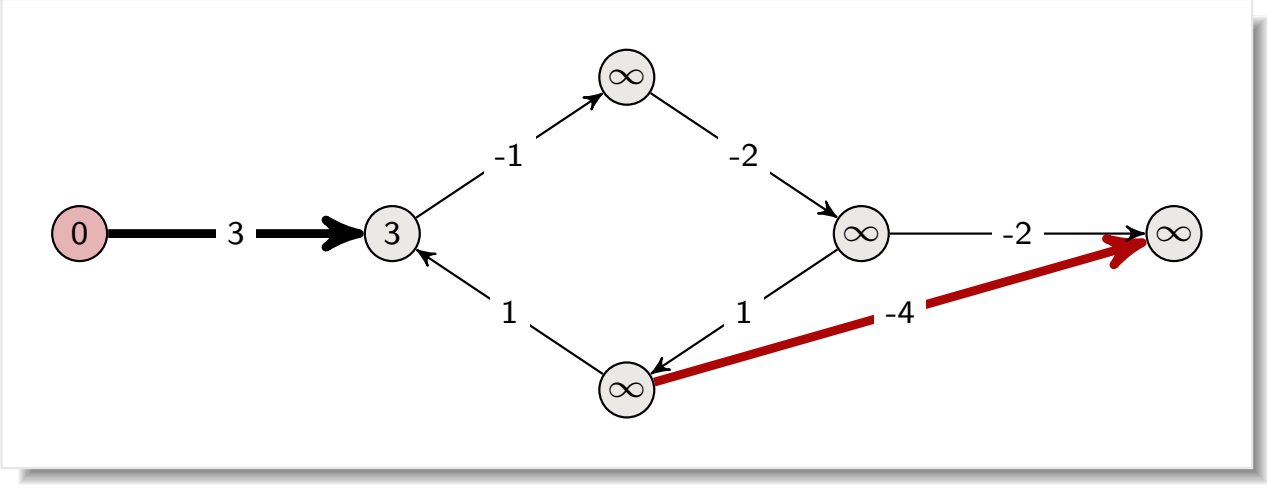
Bellman-Ford Algorithm

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)



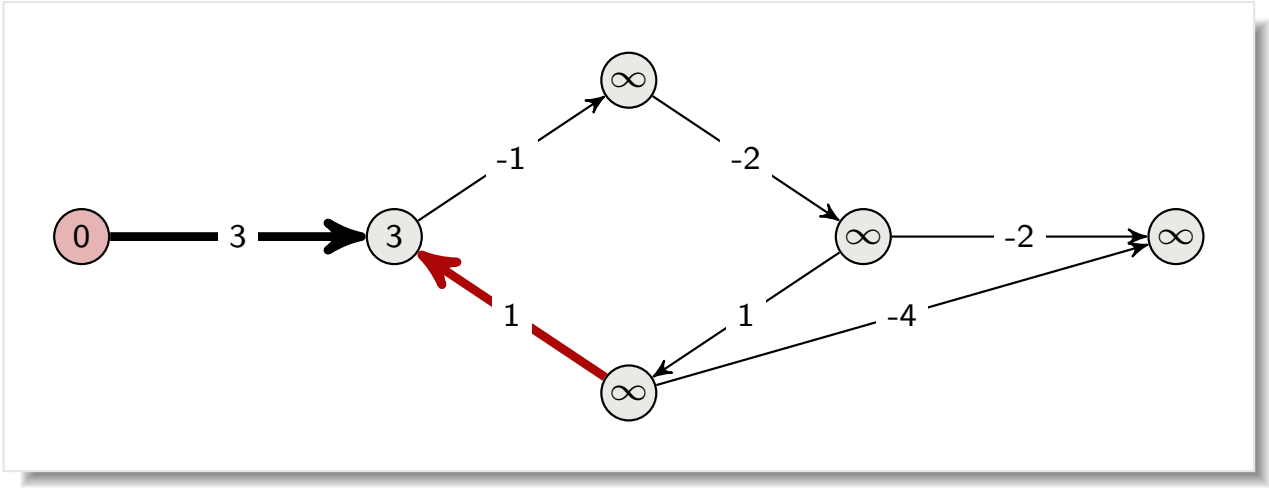
Bellman-Ford Algorithm

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)



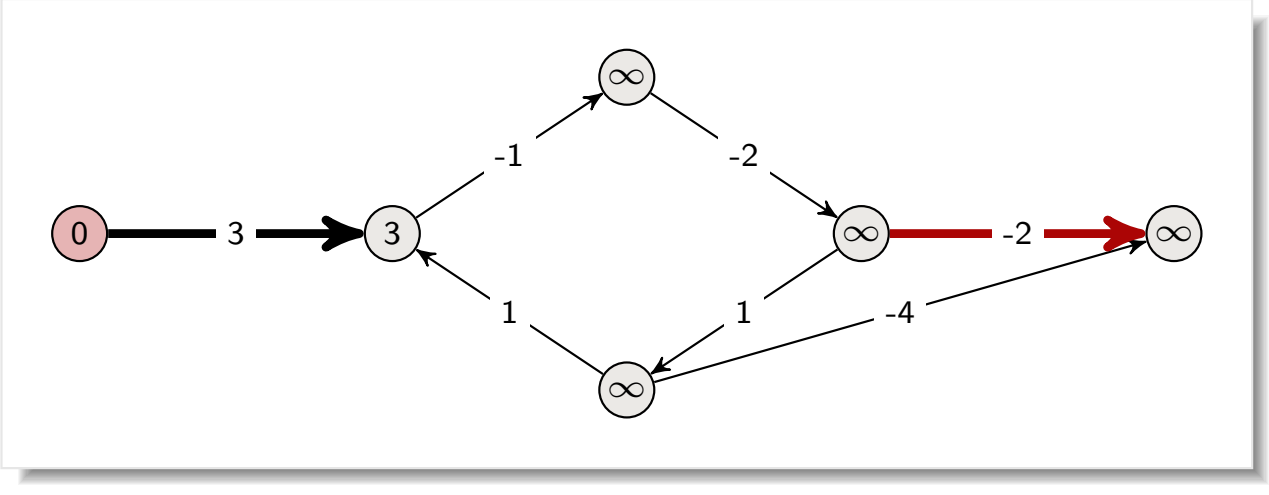
Bellman-Ford Algorithm

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



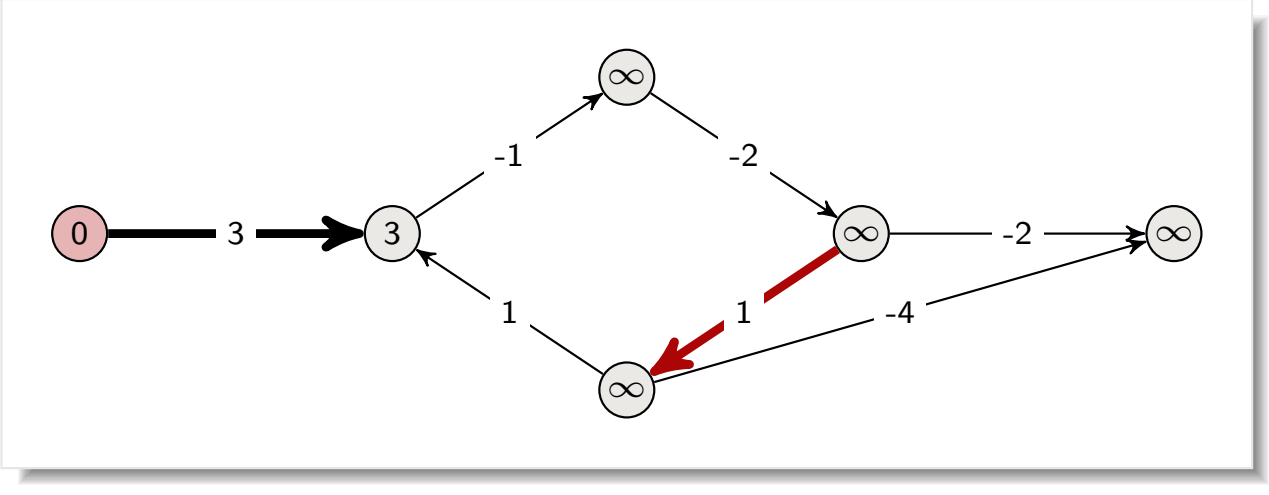
Bellman-Ford Algorithm

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



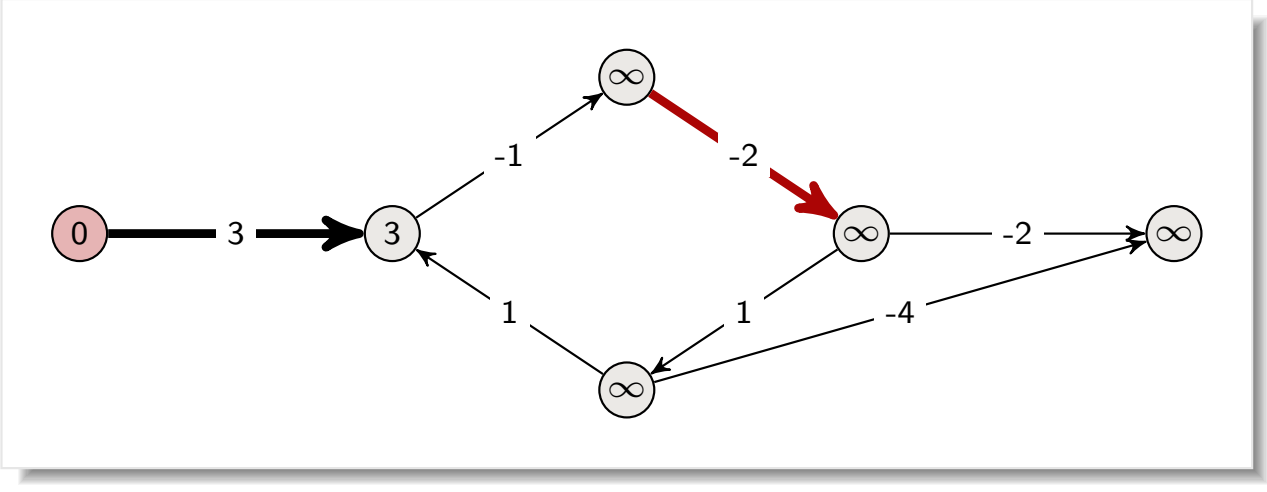
Bellman-Ford Algorithm

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



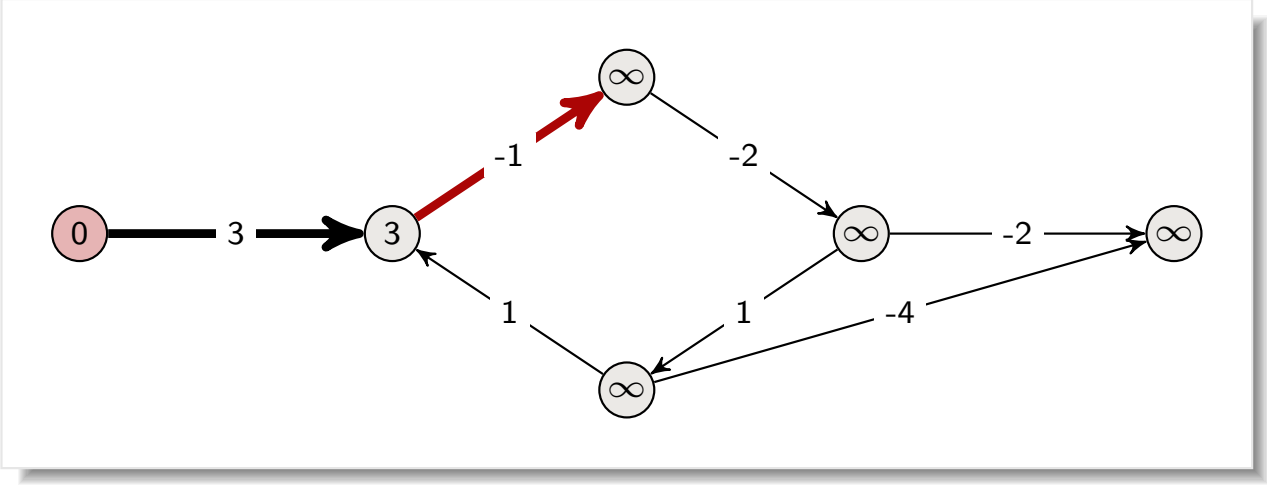
Bellman-Ford Algorithm

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)



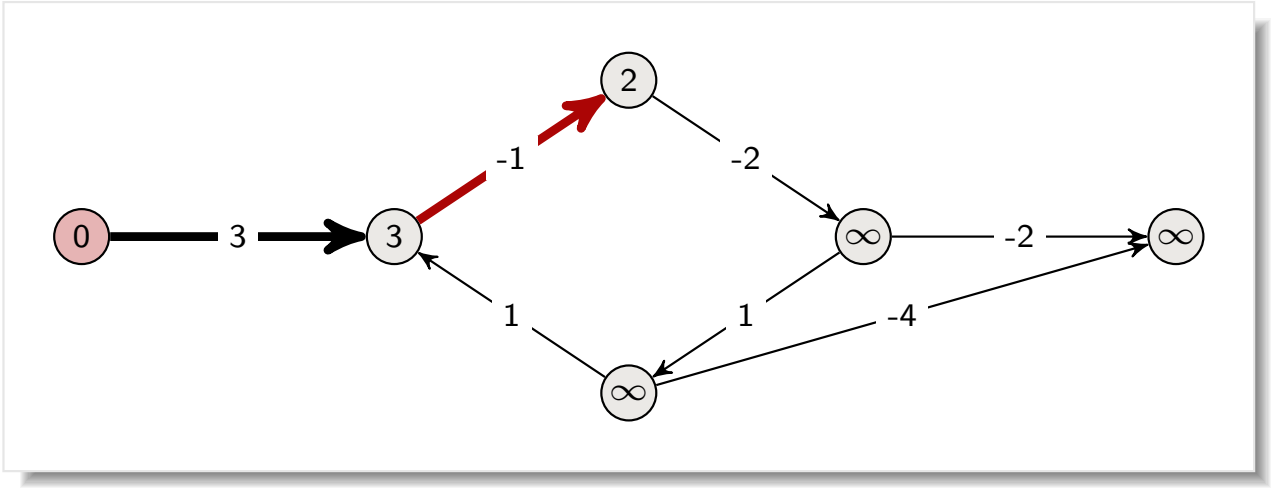
Bellman-Ford Algorithm

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)



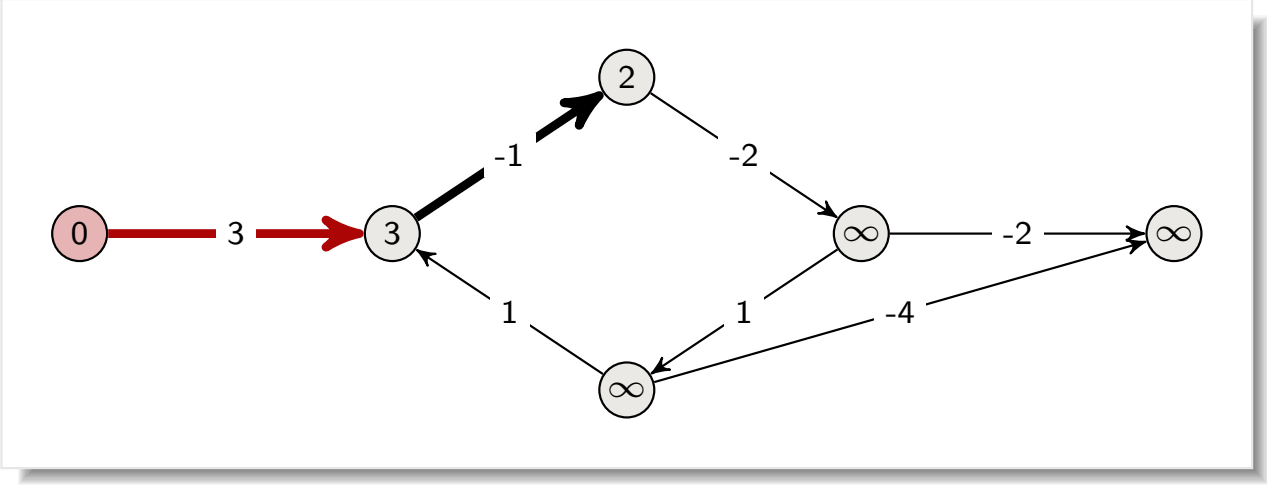
Bellman-Ford Algorithm

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



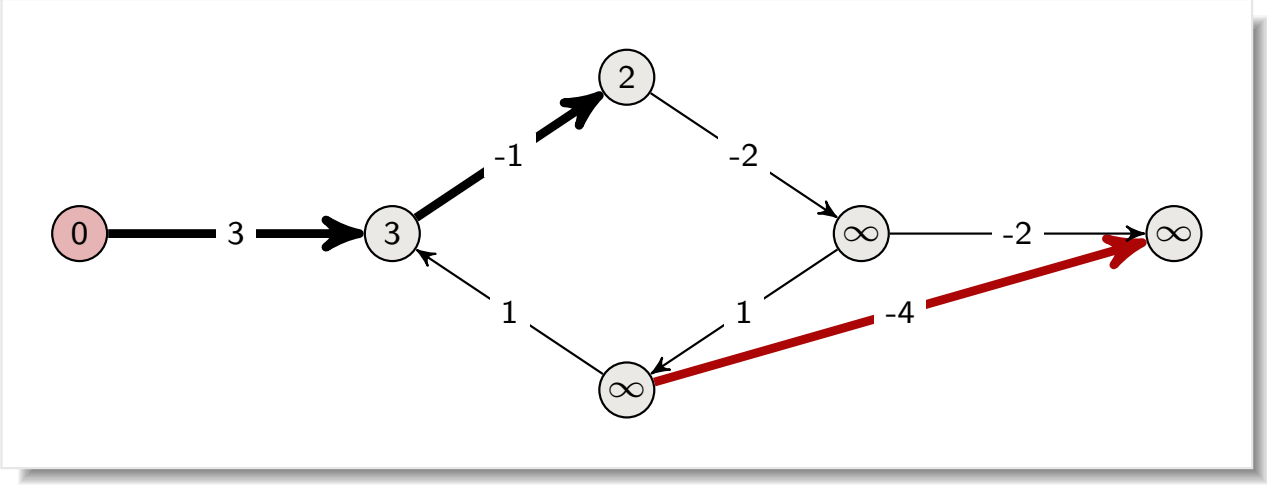
Bellman-Ford Algorithm

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)



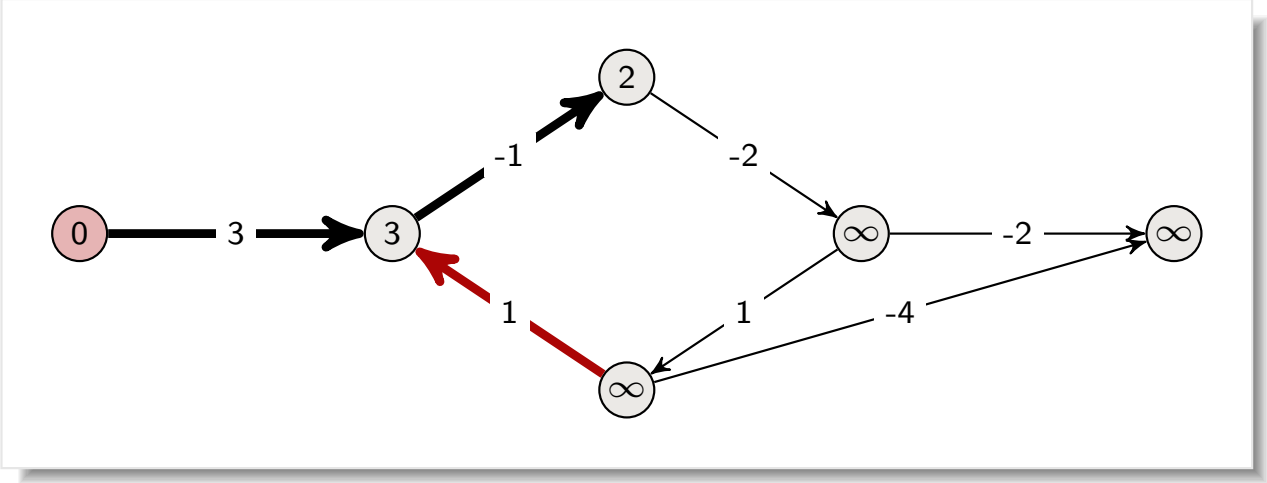
Bellman-Ford Algorithm

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)



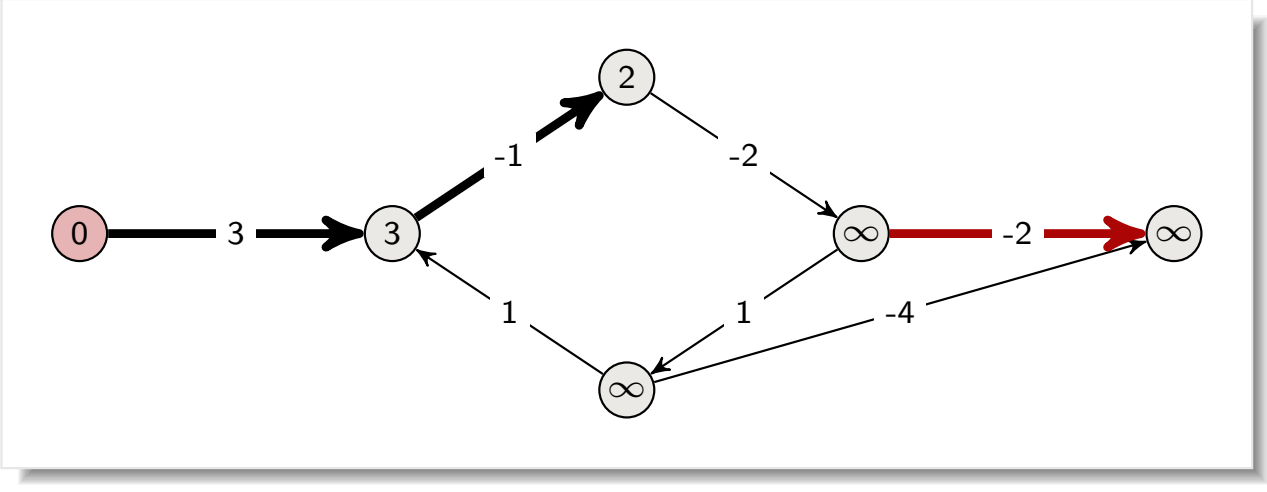
Bellman-Ford Algorithm

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)



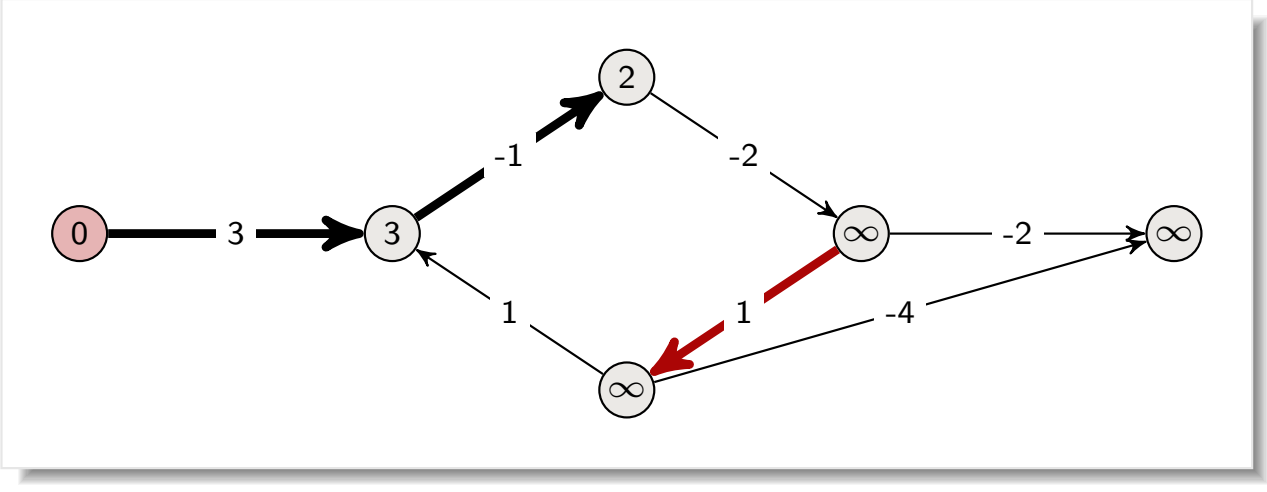
Bellman-Ford Algorithm

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)



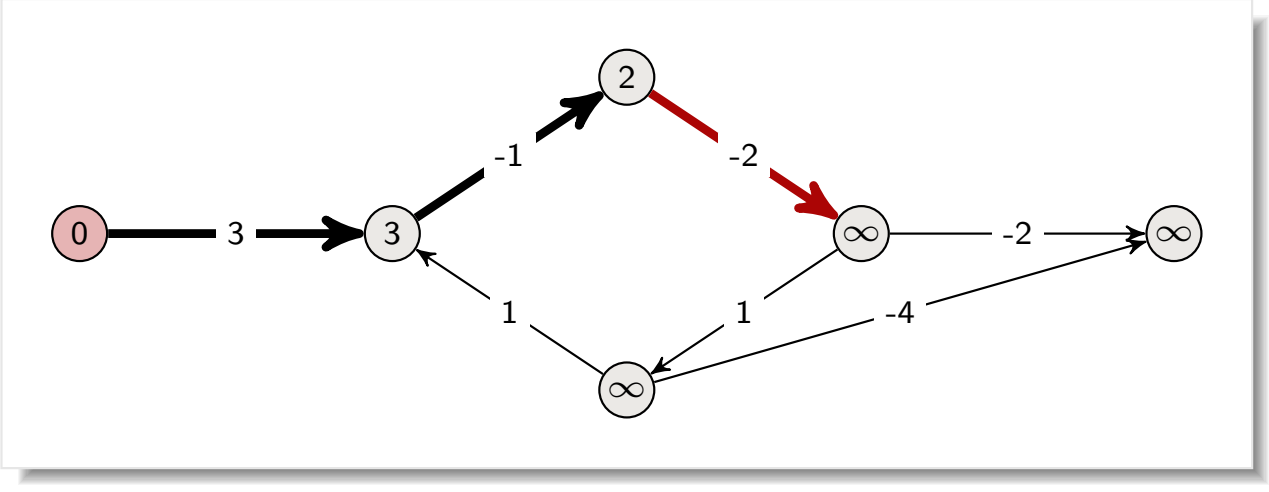
Bellman-Ford Algorithm

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)



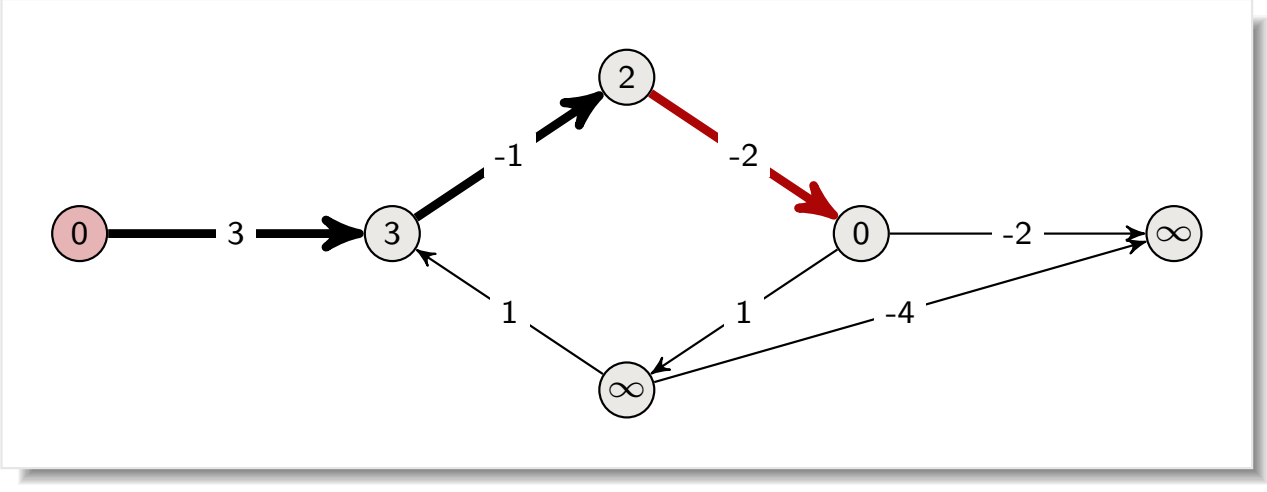
Bellman-Ford Algorithm

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)



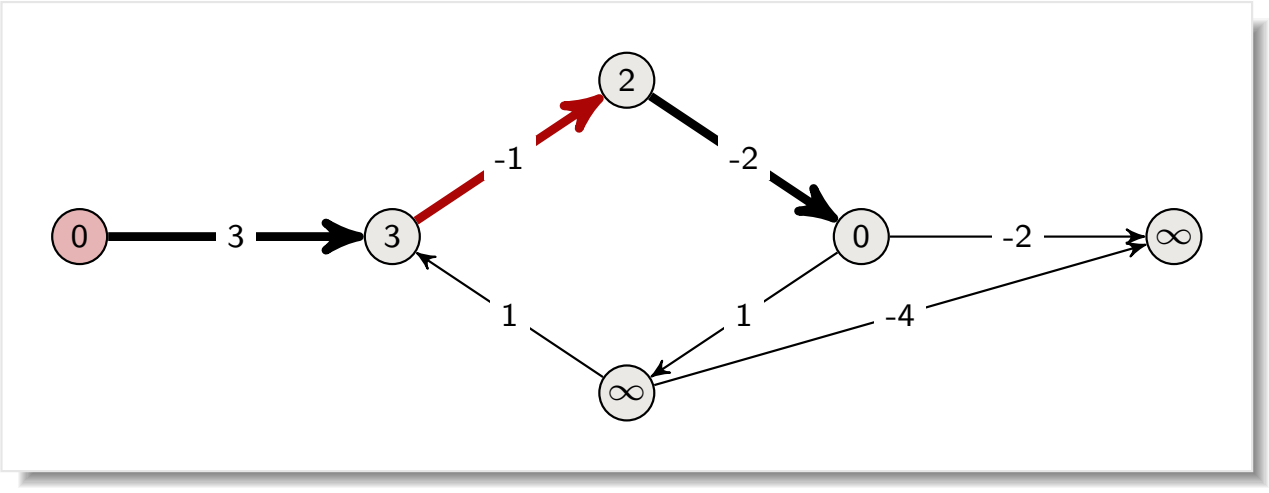
Bellman-Ford Algorithm

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)



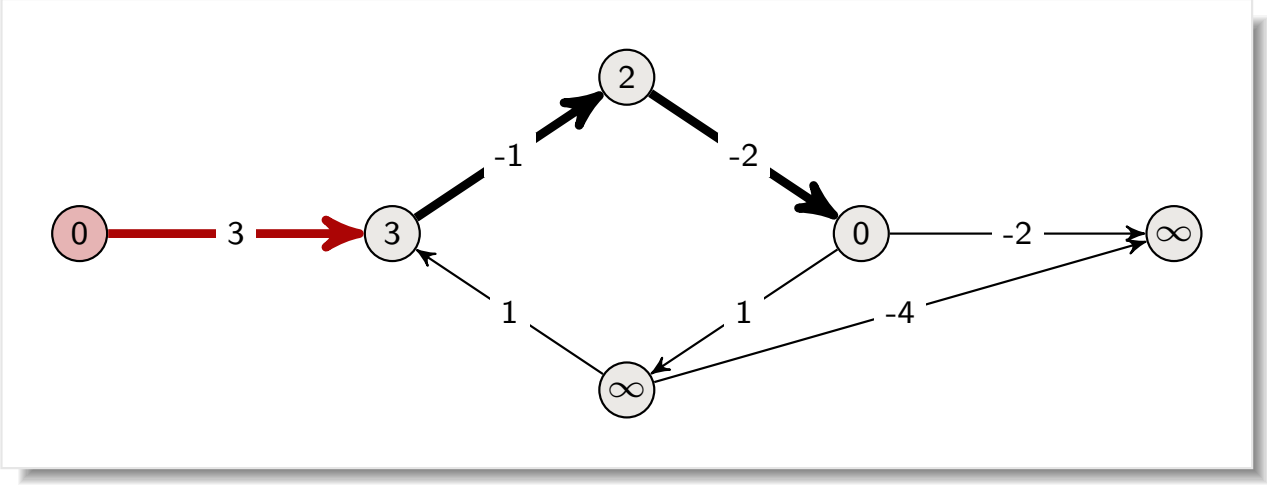
Bellman-Ford Algorithm

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



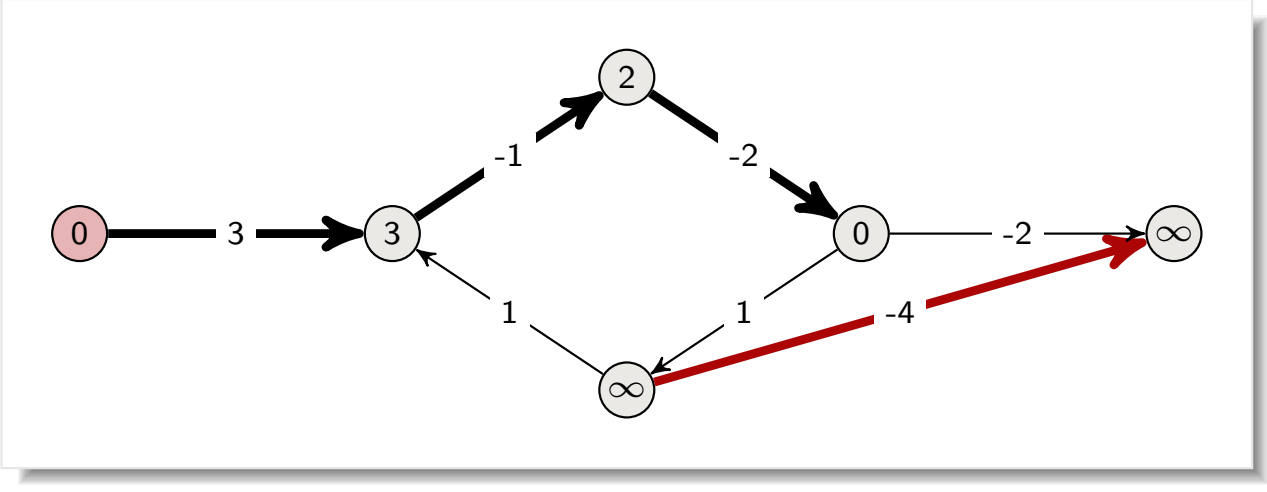
Bellman-Ford Algorithm

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



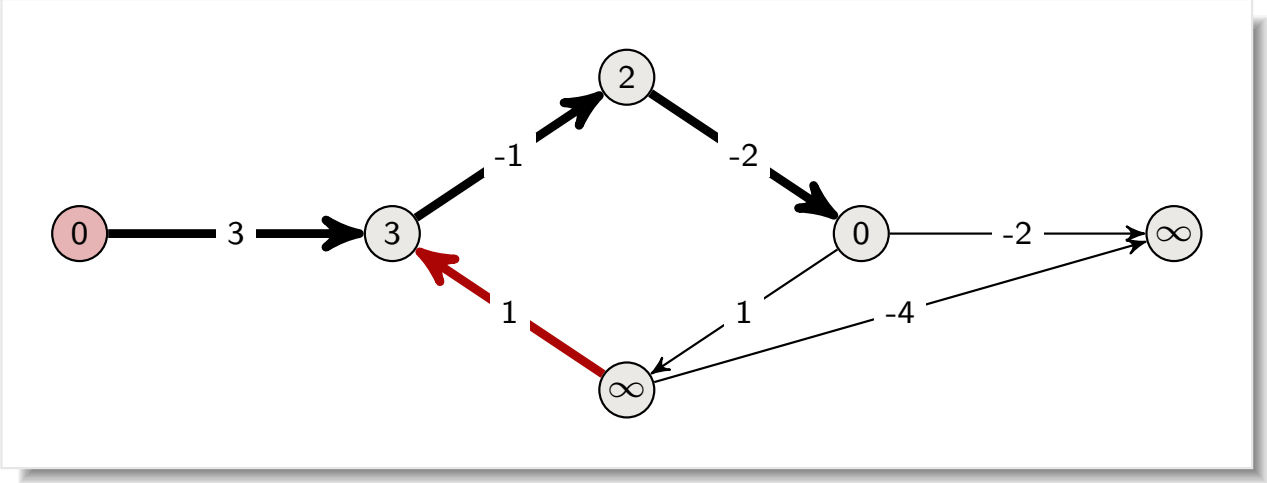
Bellman-Ford Algorithm

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)



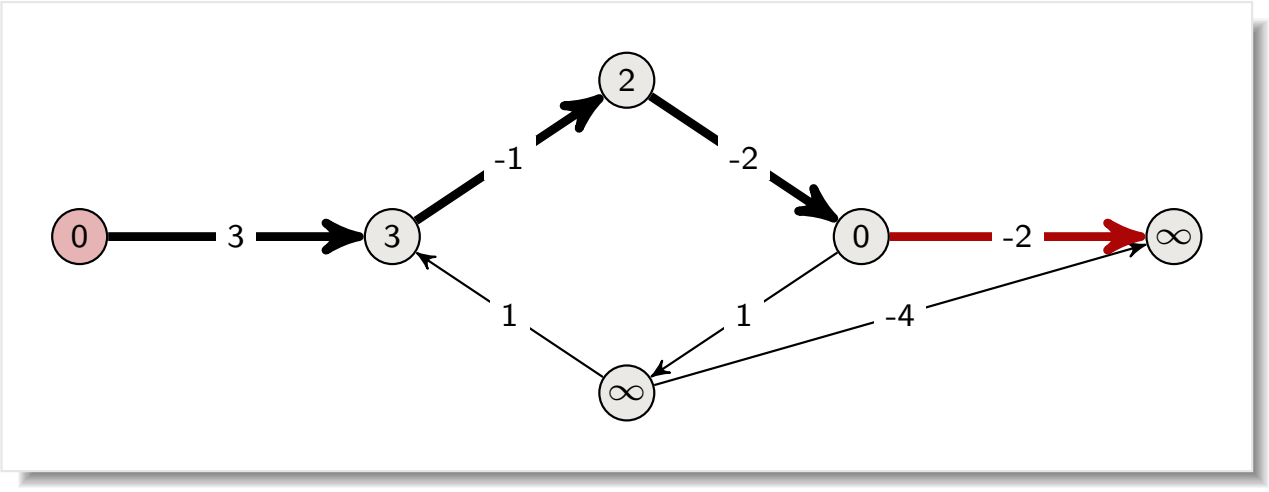
Bellman-Ford Algorithm

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)



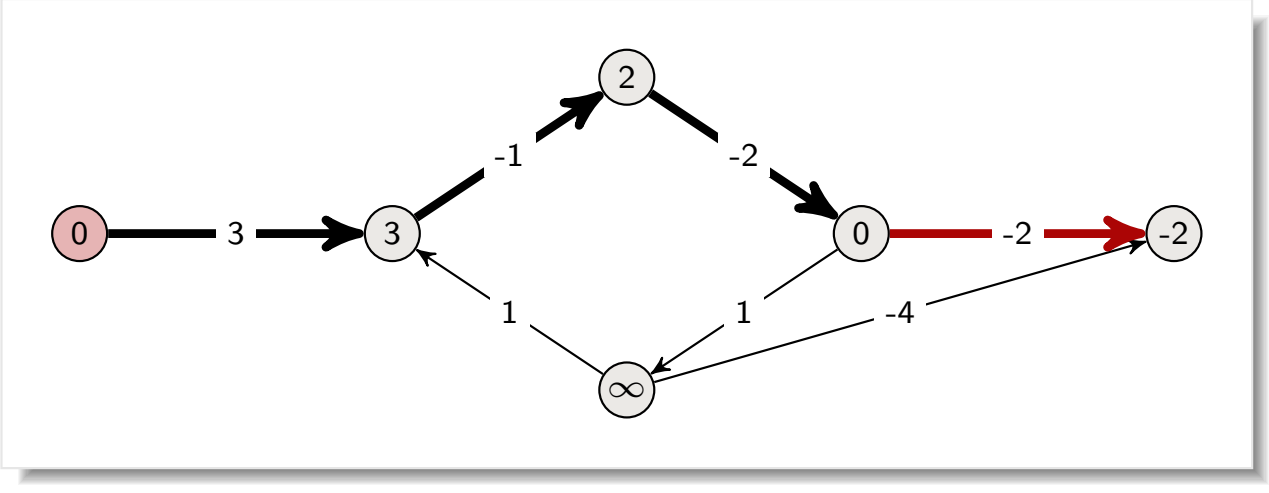
Bellman-Ford Algorithm

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



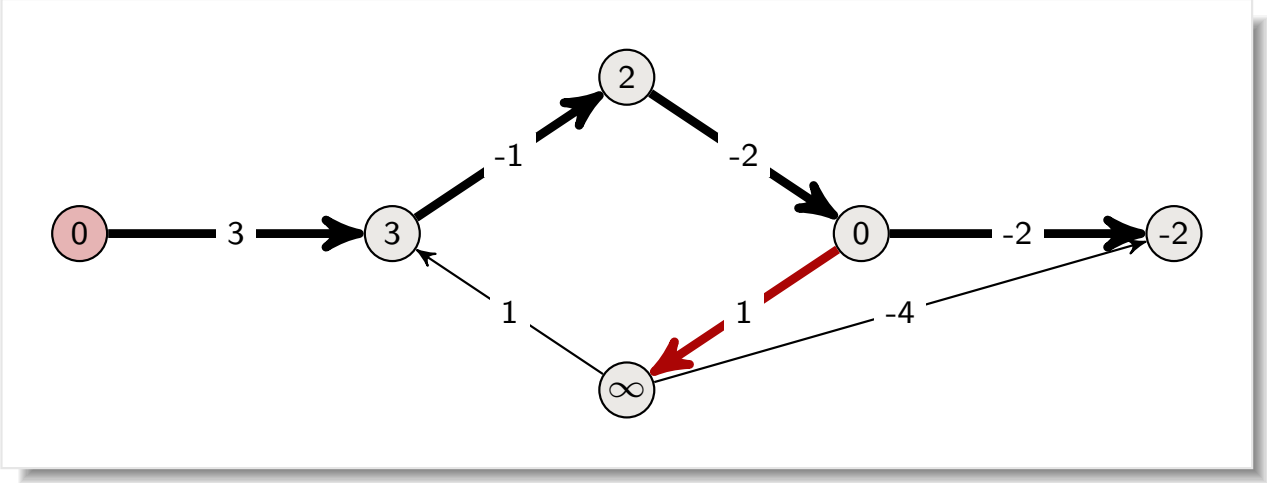
Bellman-Ford Algorithm

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)



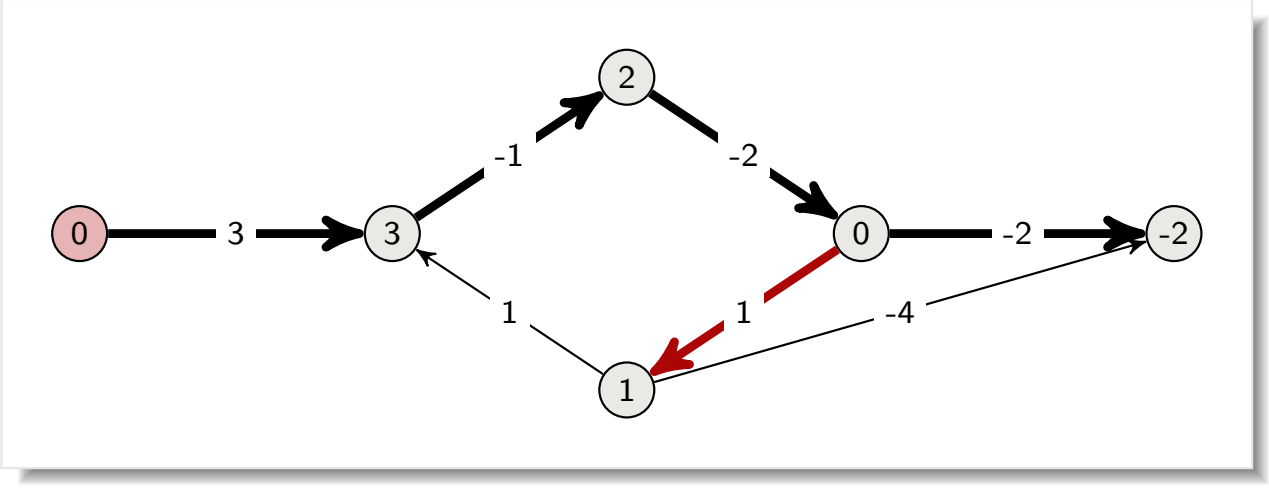
Bellman-Ford Algorithm

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)



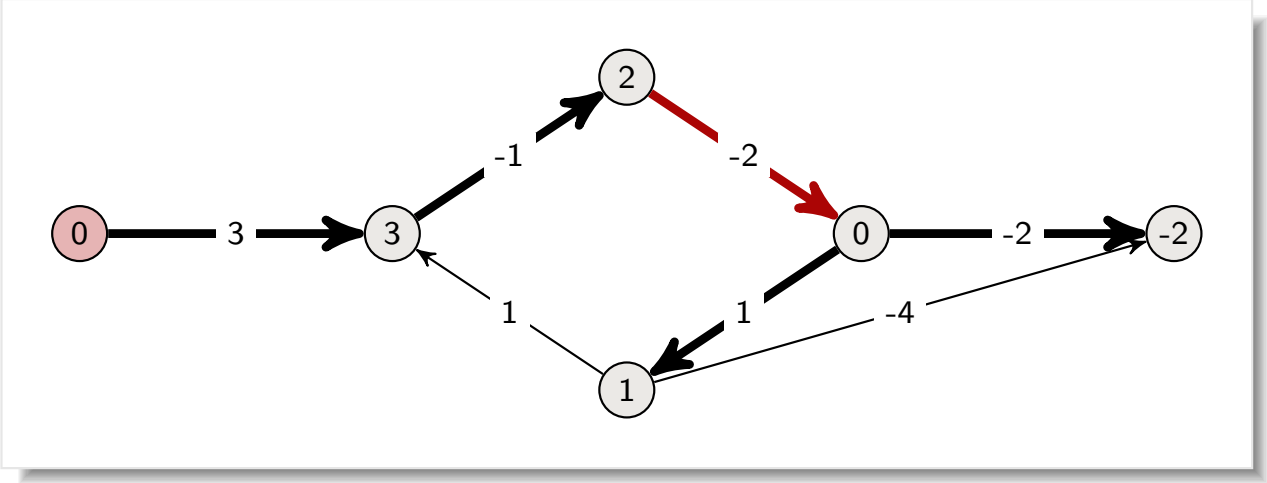
Bellman-Ford Algorithm

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)



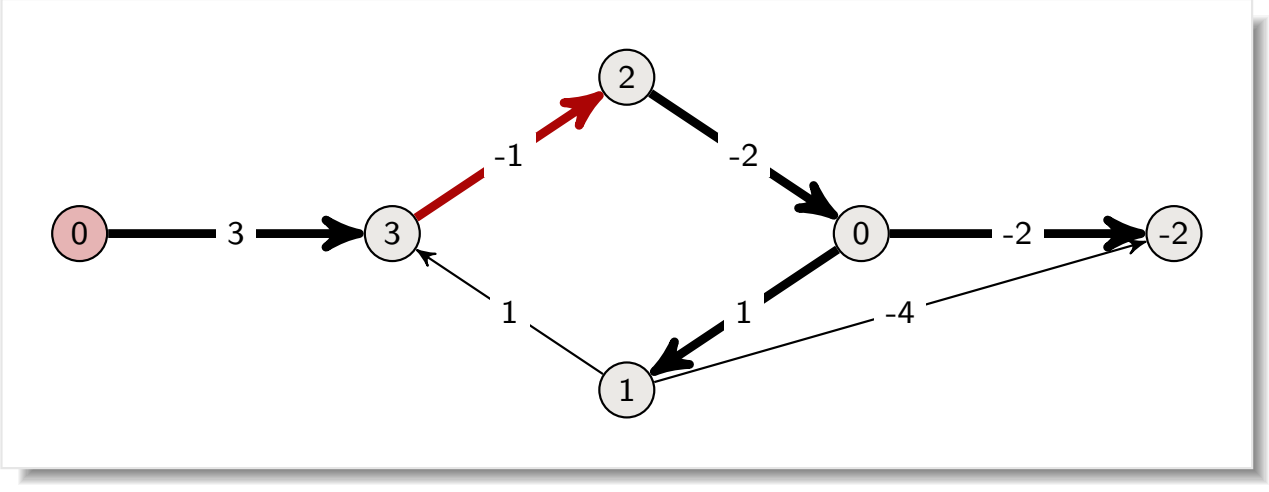
Bellman-Ford Algorithm

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)



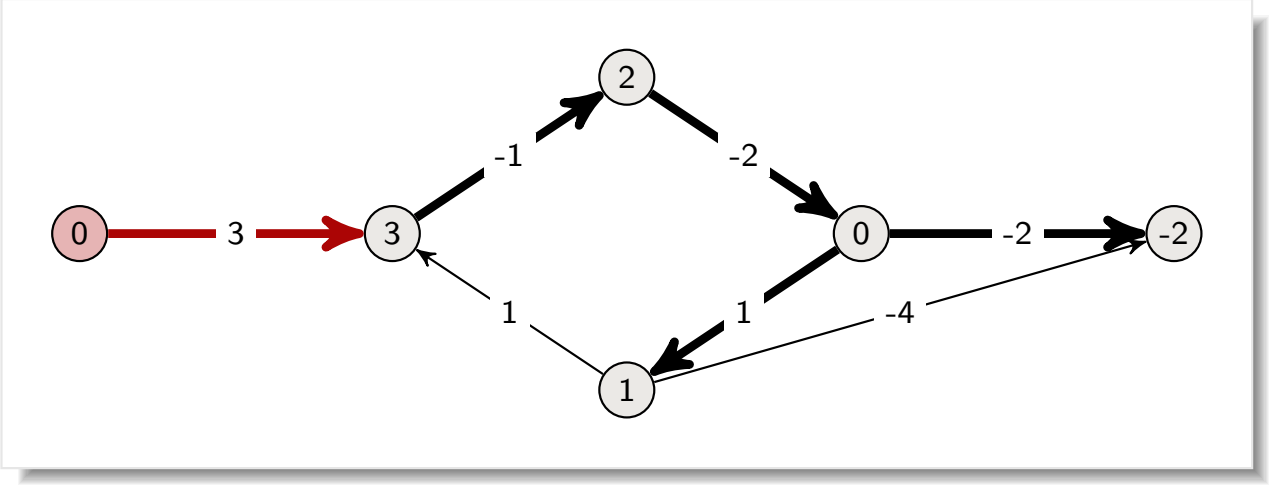
Bellman-Ford Algorithm

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)



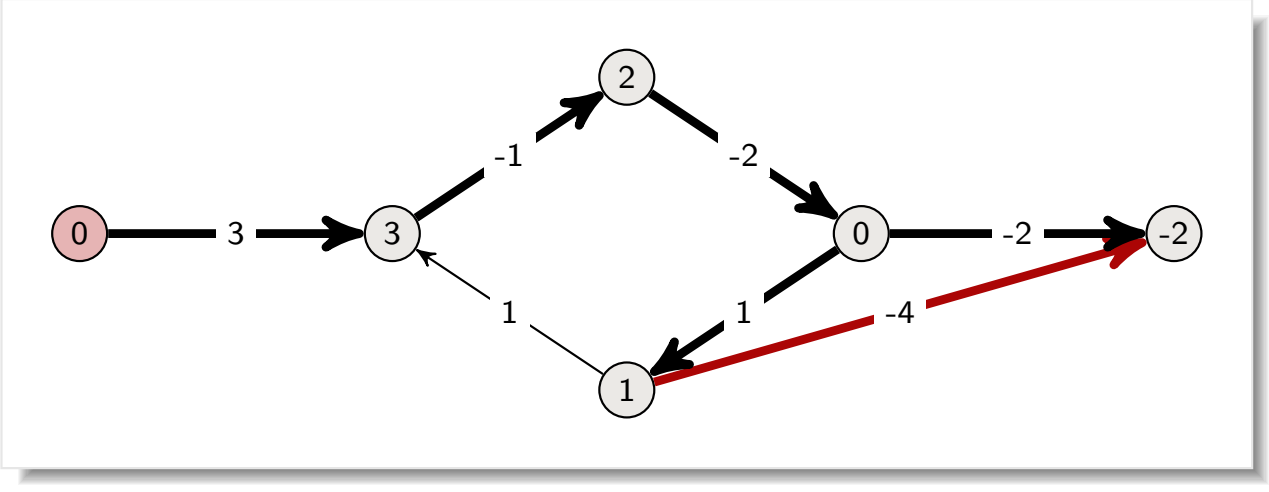
Bellman-Ford Algorithm

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)



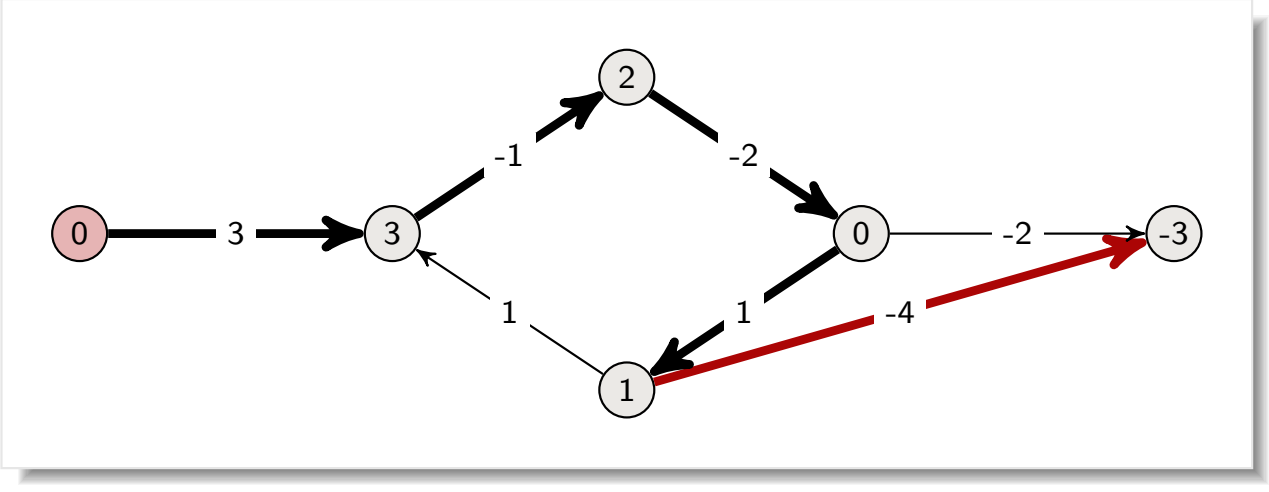
Bellman-Ford Algorithm

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)



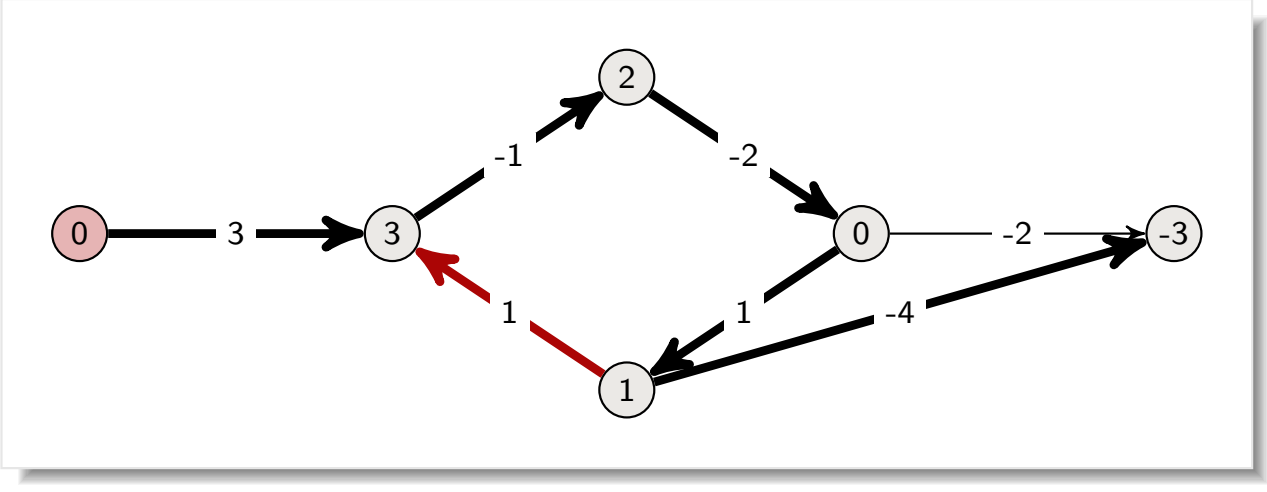
Bellman-Ford Algorithm

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)



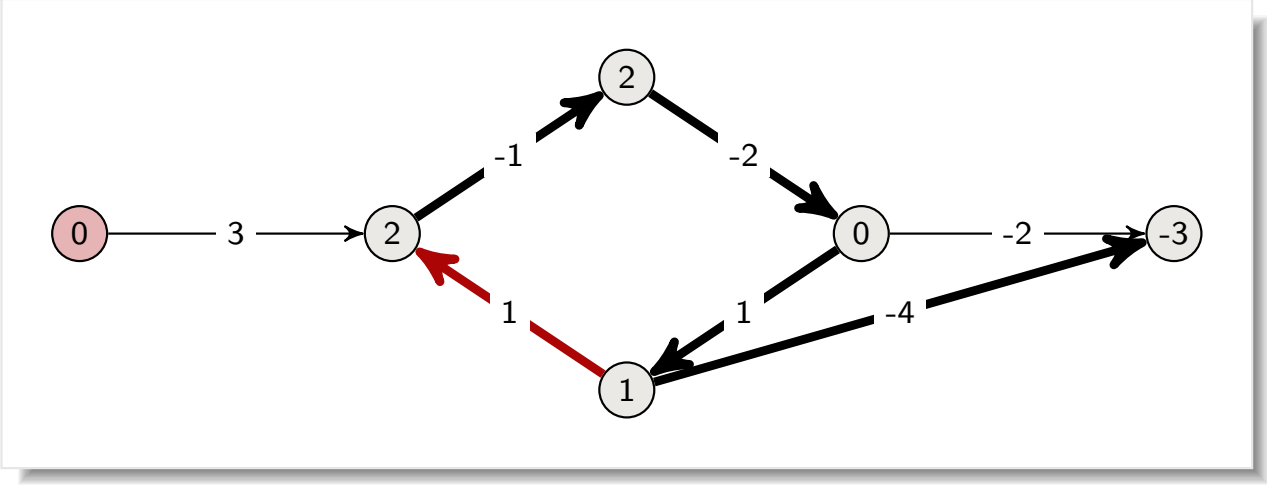
Bellman-Ford Algorithm

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)



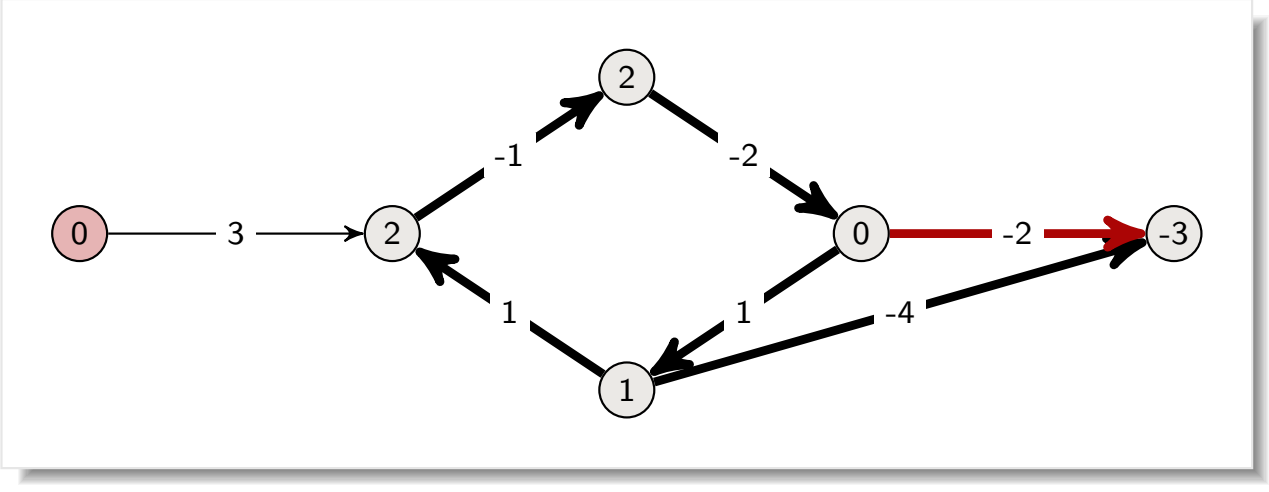
Bellman-Ford Algorithm

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)



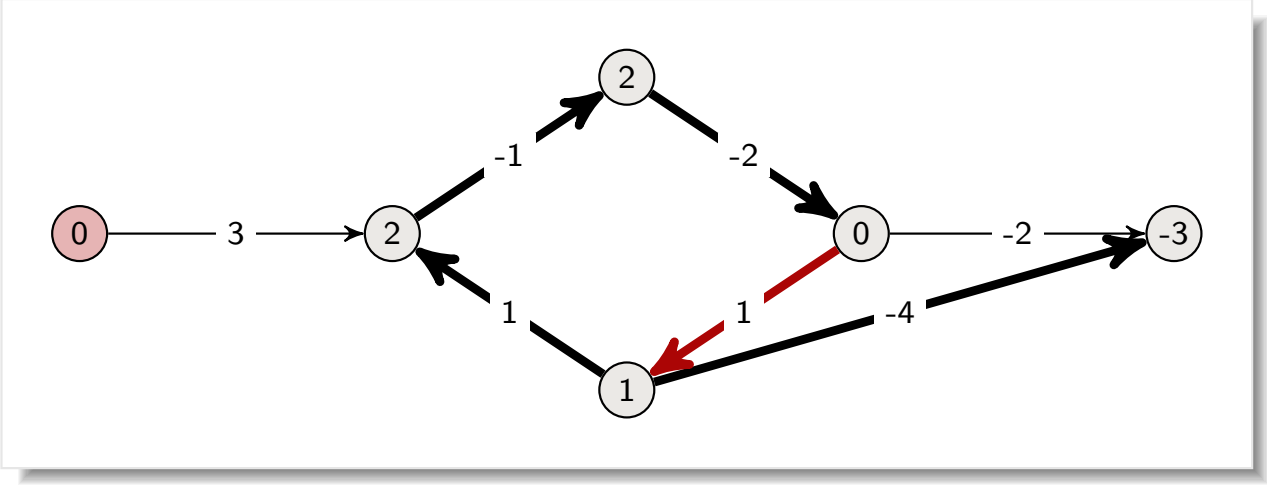
Bellman-Ford Algorithm

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)



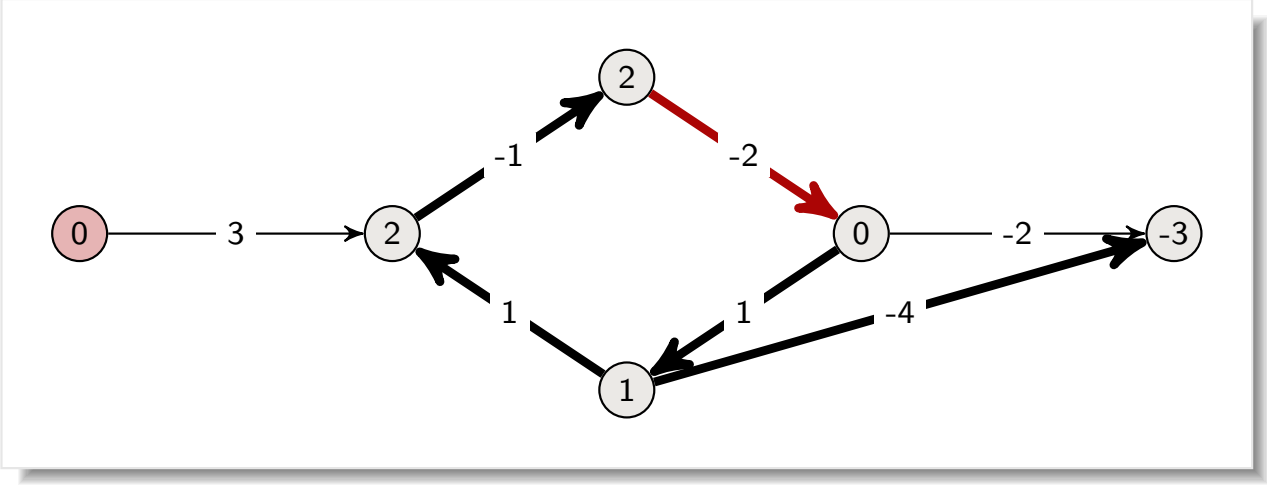
Bellman-Ford Algorithm

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)



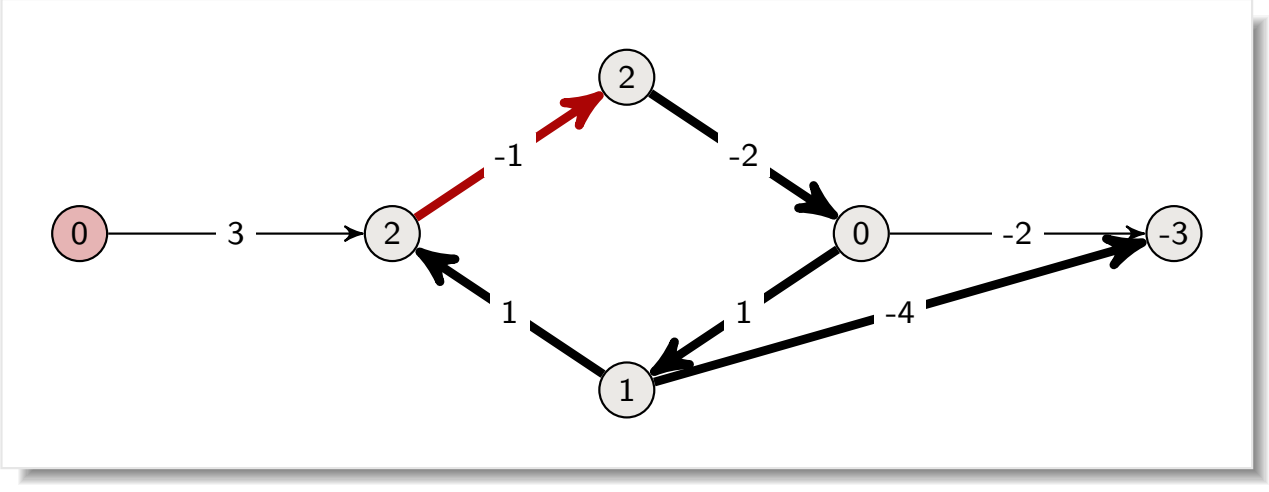
Bellman-Ford Algorithm

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)



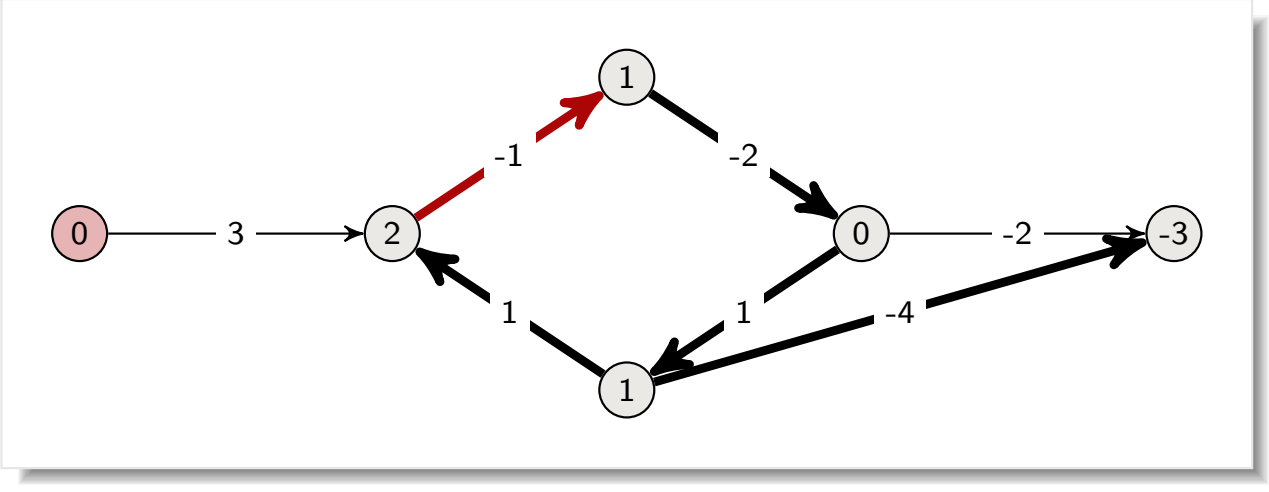
Bellman-Ford Algorithm

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)



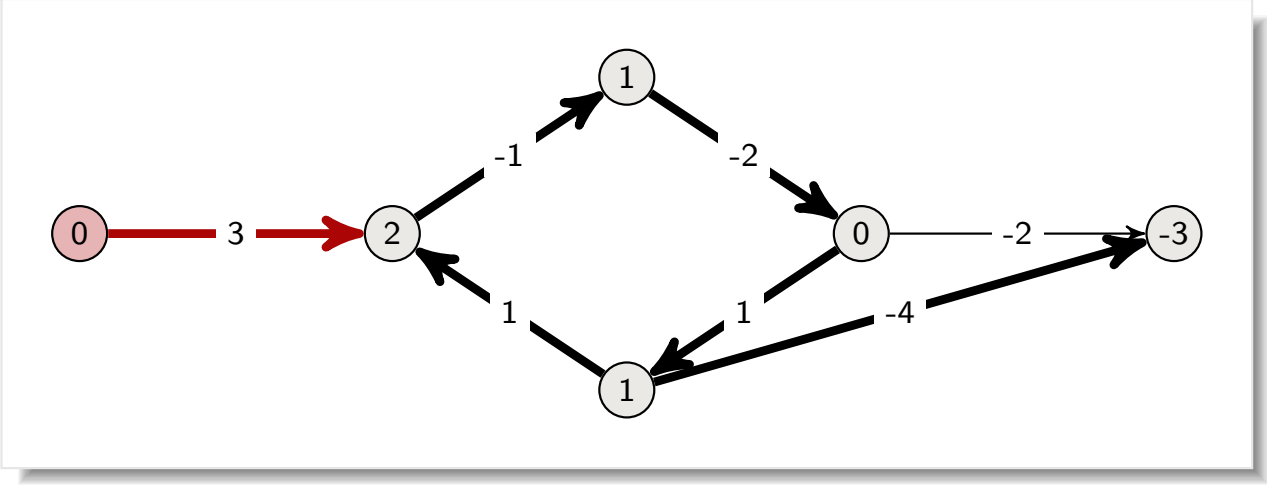
Bellman-Ford Algorithm

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)



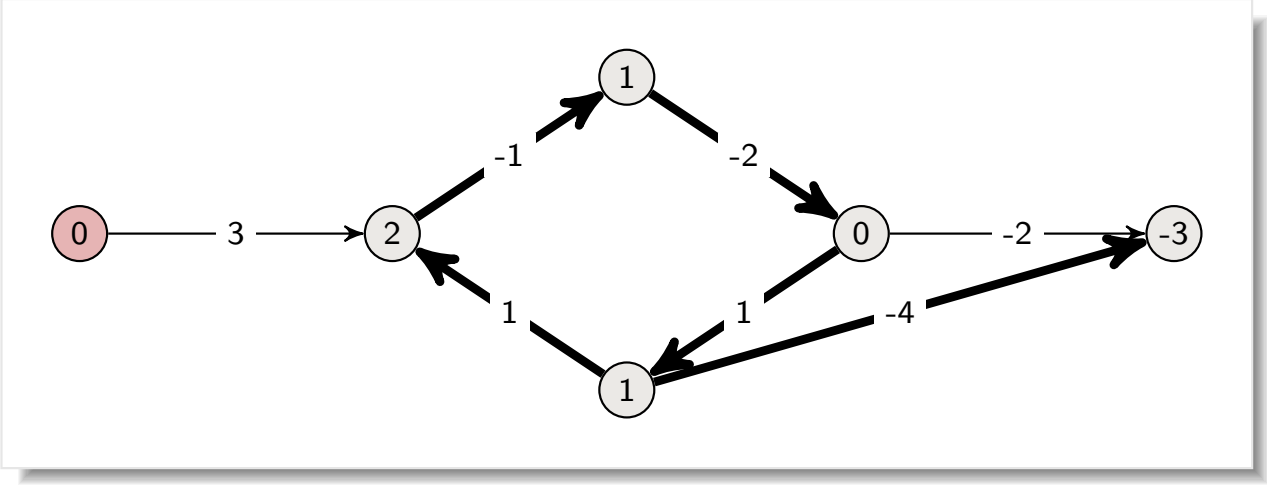
Bellman-Ford Algorithm

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)



Correctness

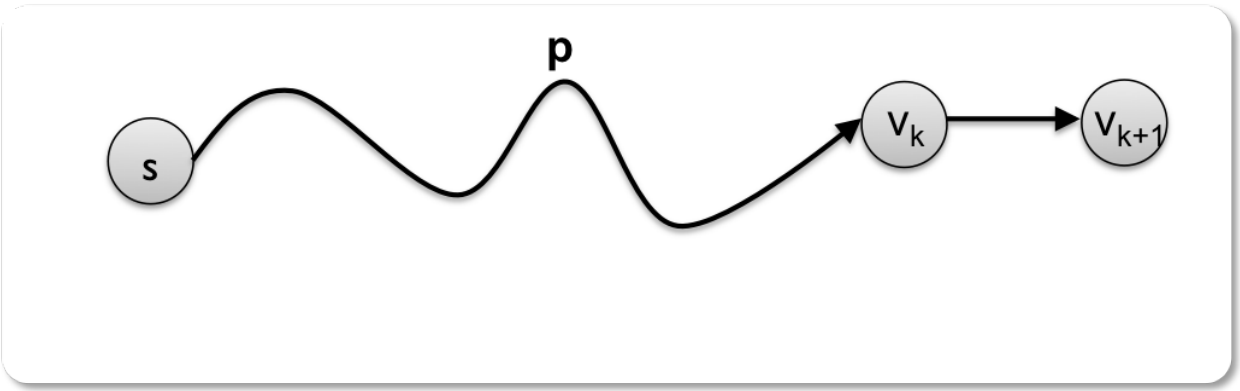
Only guaranteed to work if no negative cycles!

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

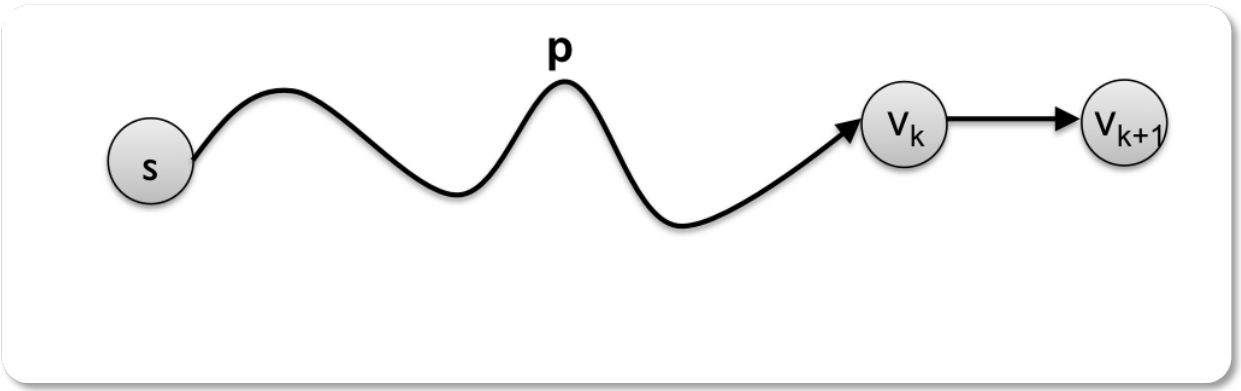


Optimal substructure

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:



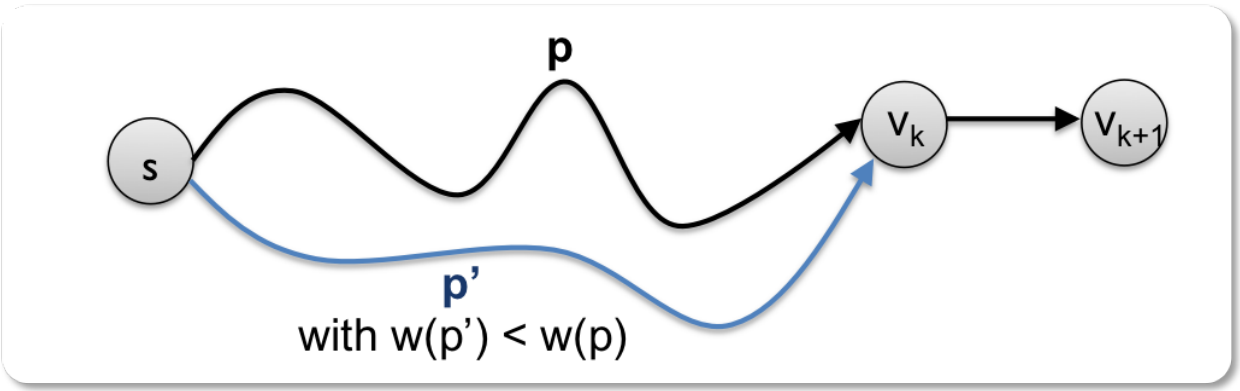
Optimal substructure

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



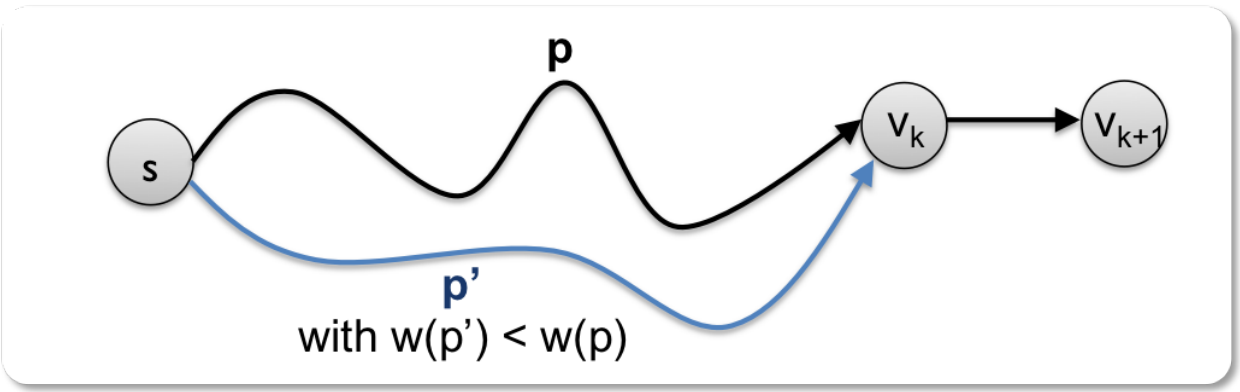
Optimal substructure

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

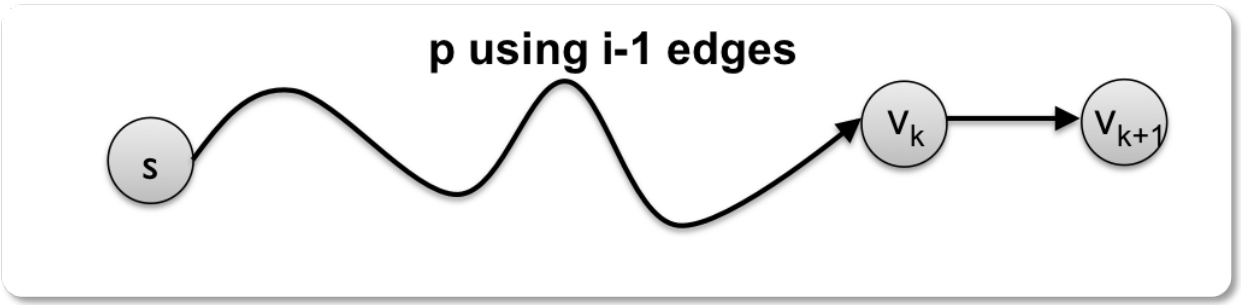
Proof of correctness

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

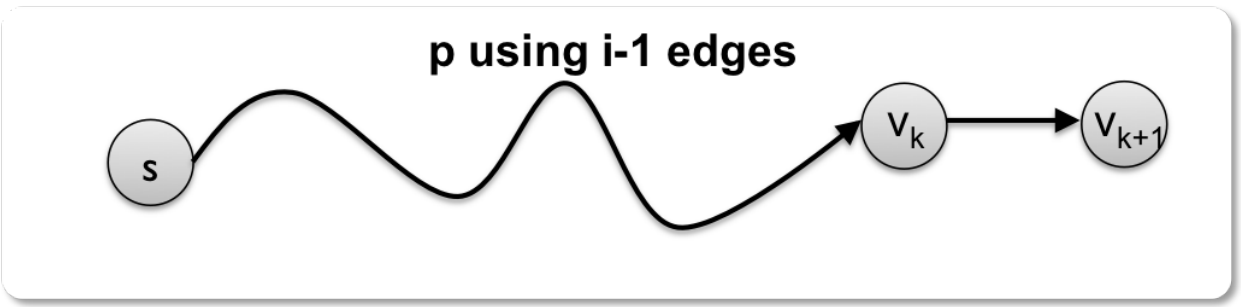
Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

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

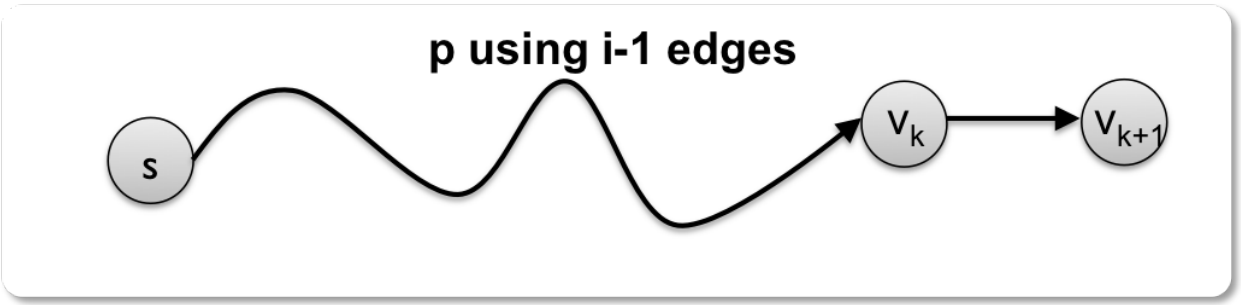
Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

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

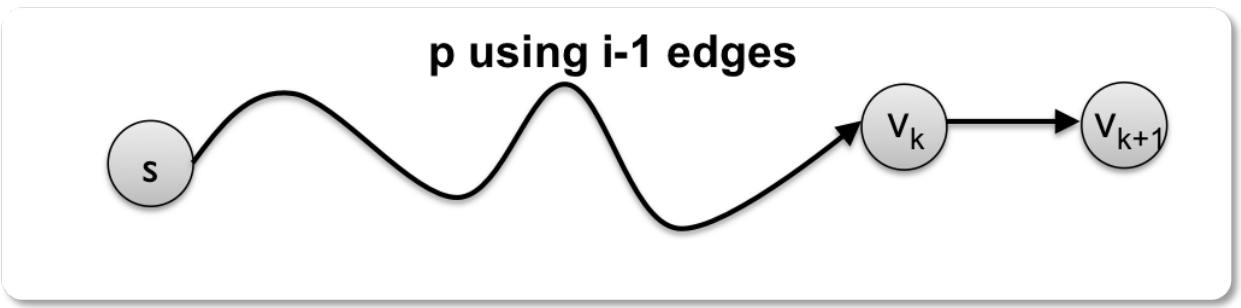
Invariant: $\ell(v)$ is at most the length of the shortest path from s to v using at most i edges after the i'th iteration

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, $\ell(v_{k+1}) \leq \ell(v_k) + w(p) =$ "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

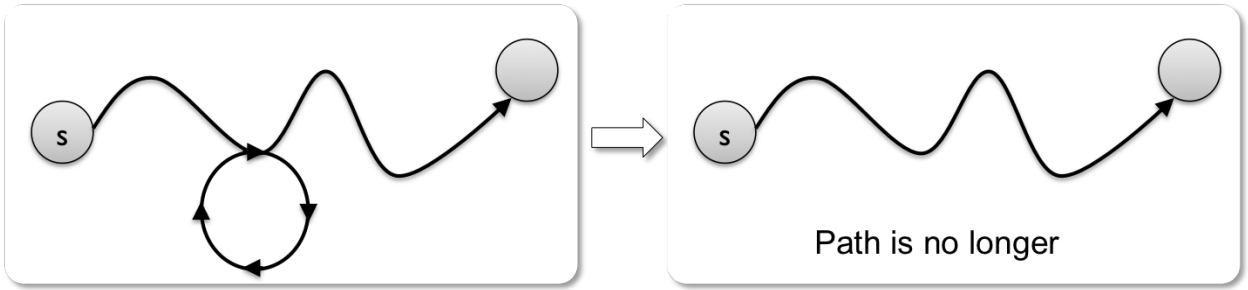
If there are no negative cycles reachable from s, then for any v there is a shortest path from s to v using at most $n-1$ edges

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

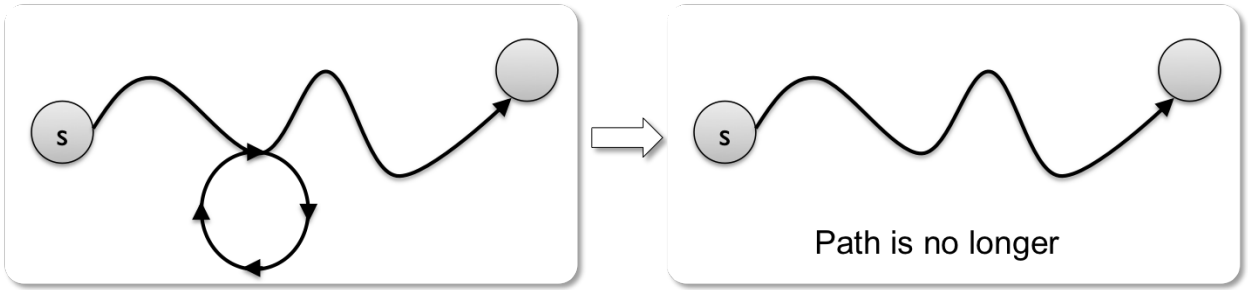


Proof of correctness

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

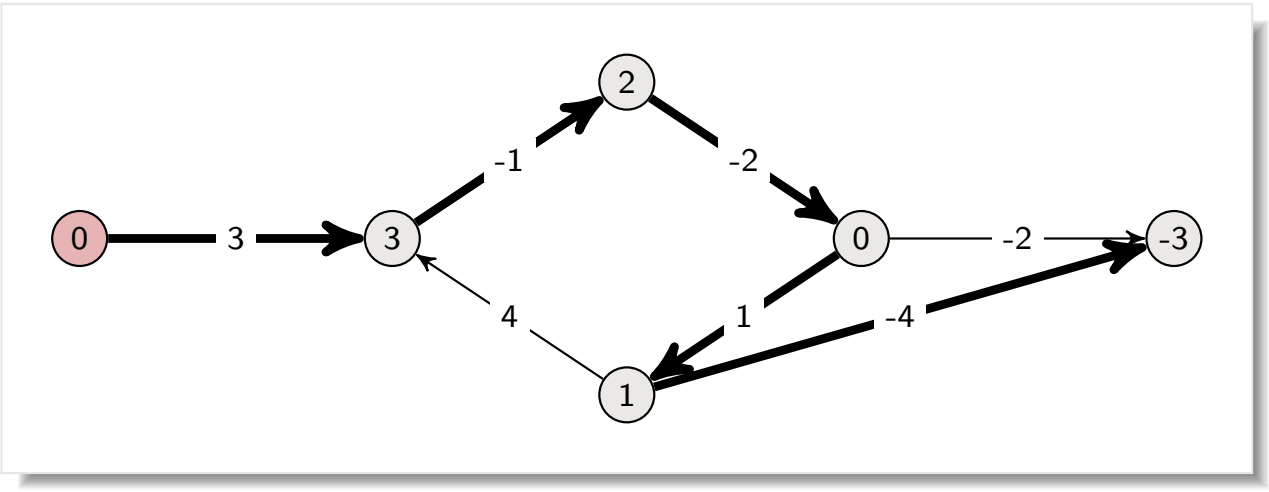
So for a cycle $v_0 - v_1 - \cdots - v_{t-1} - v_t = v_0$,

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

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

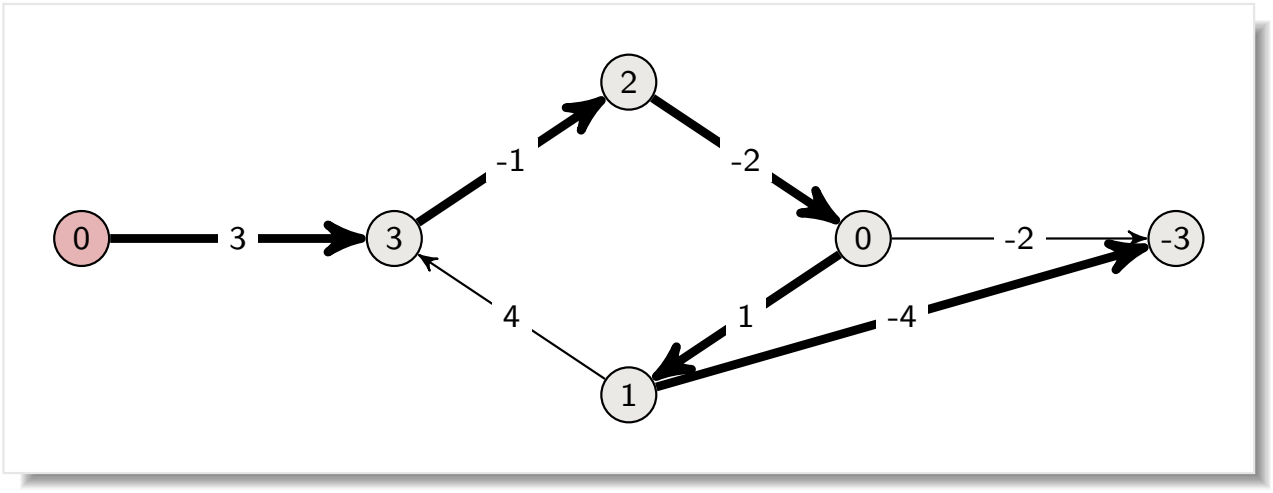
Example 1

```
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| − 1
      for each edge (u, v) ∈ 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 1

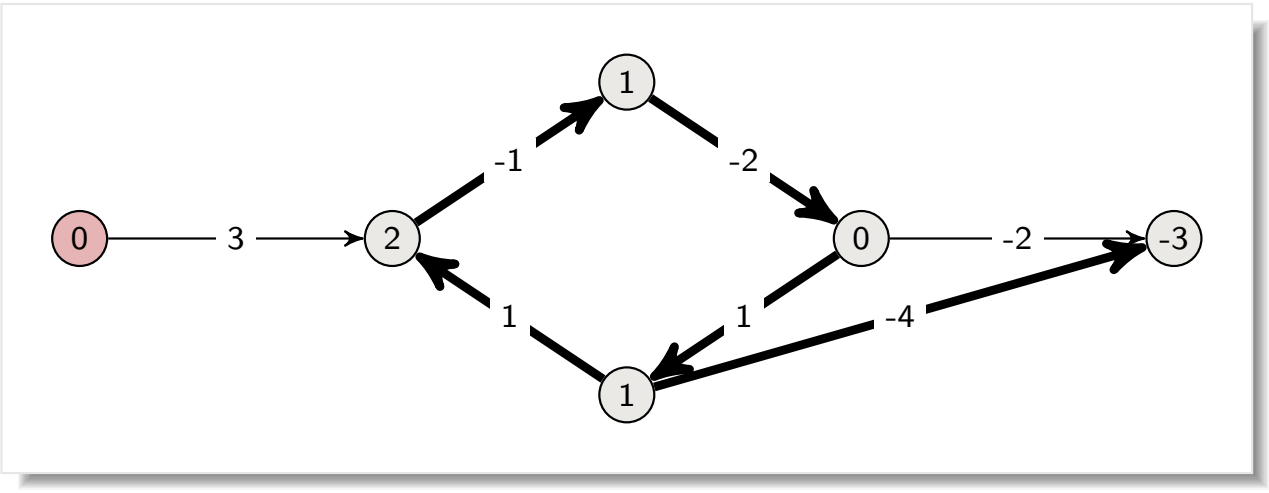
```
BELLMAN-FORD(G, w, s)
  INIT-SINGLE-SOURCE(G, s)
  for i = 1 to |G.V| − 1
      for each edge (u, v) ∈ G.E
          RELAX(u, v, w)
  for each edge (u, v) ∈ G.E
      if v.d > u.d + w(u, v)
          return FALSE
  return TRUE
```



No negative cycles, BF returns TRUE

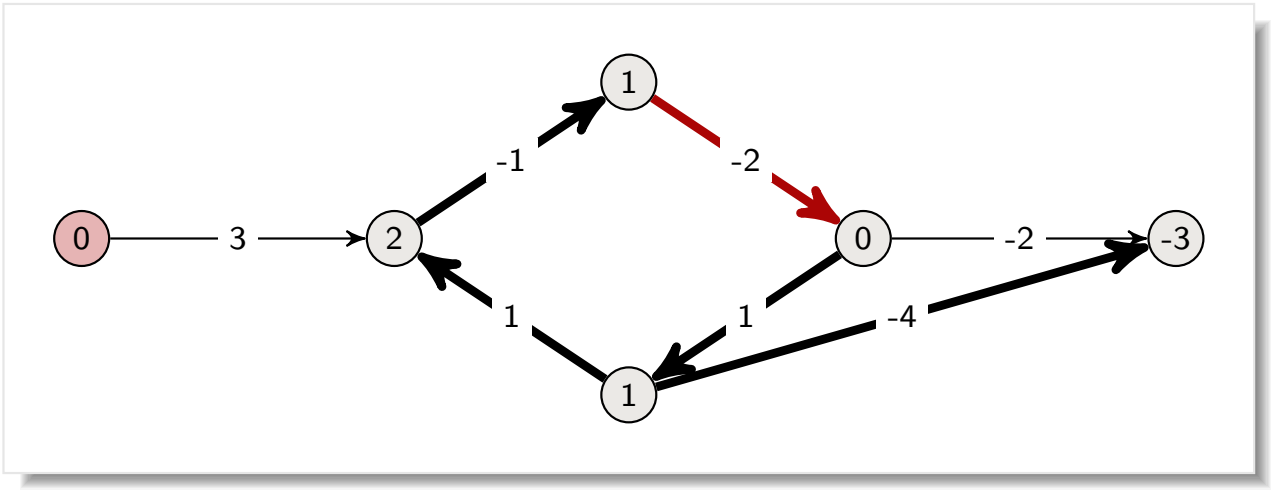
Example 2

```
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) ∈ 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
```



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: $\Theta(E \cdot V)$

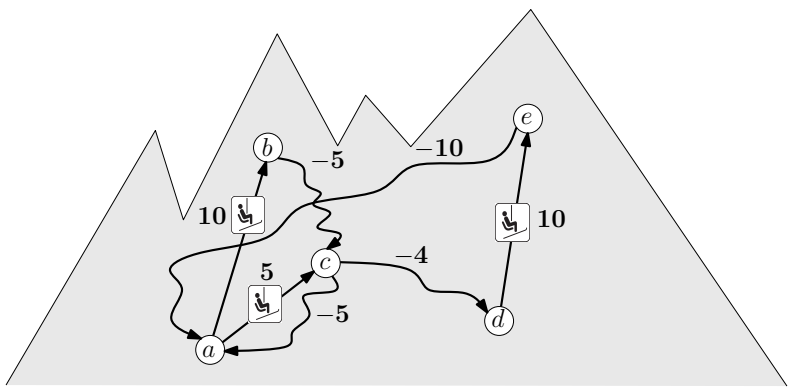
Final comments on Bellman-Ford

- 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, previous exam question)

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)

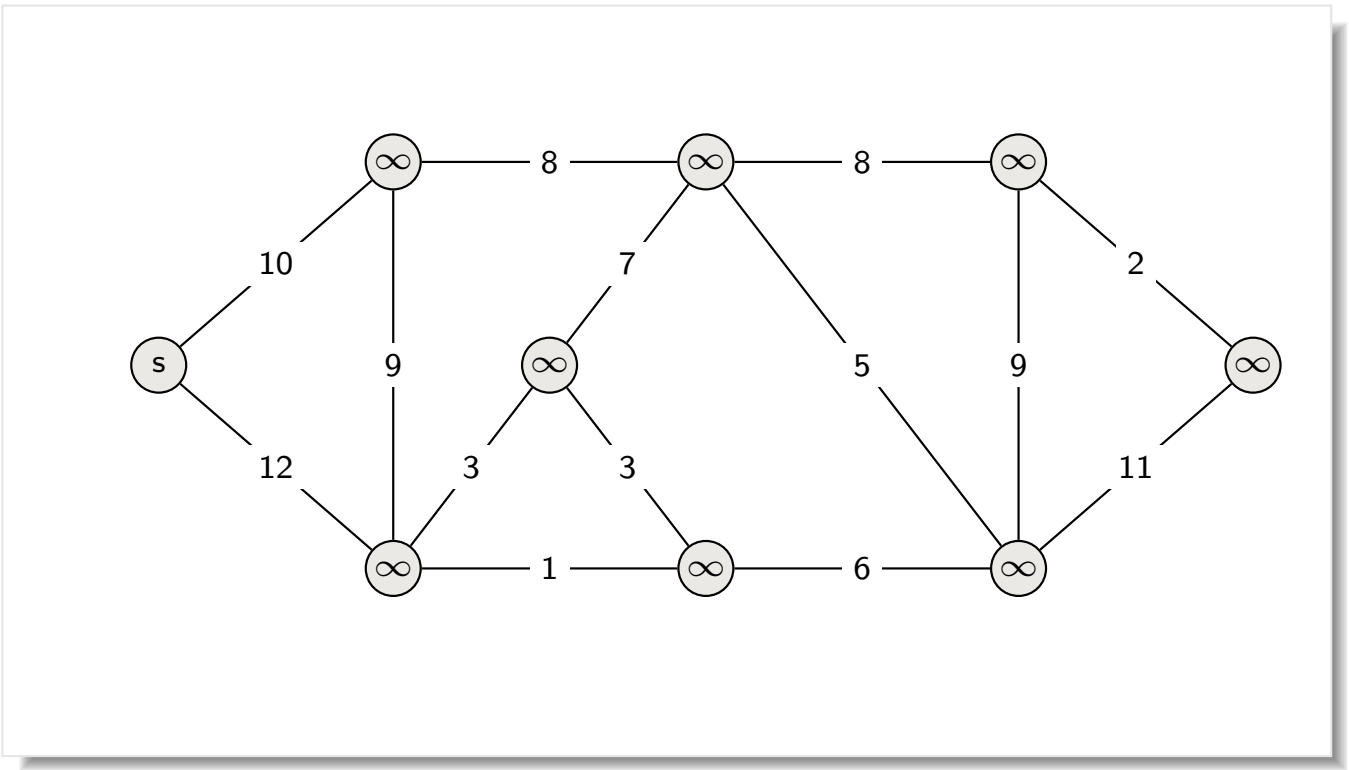
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



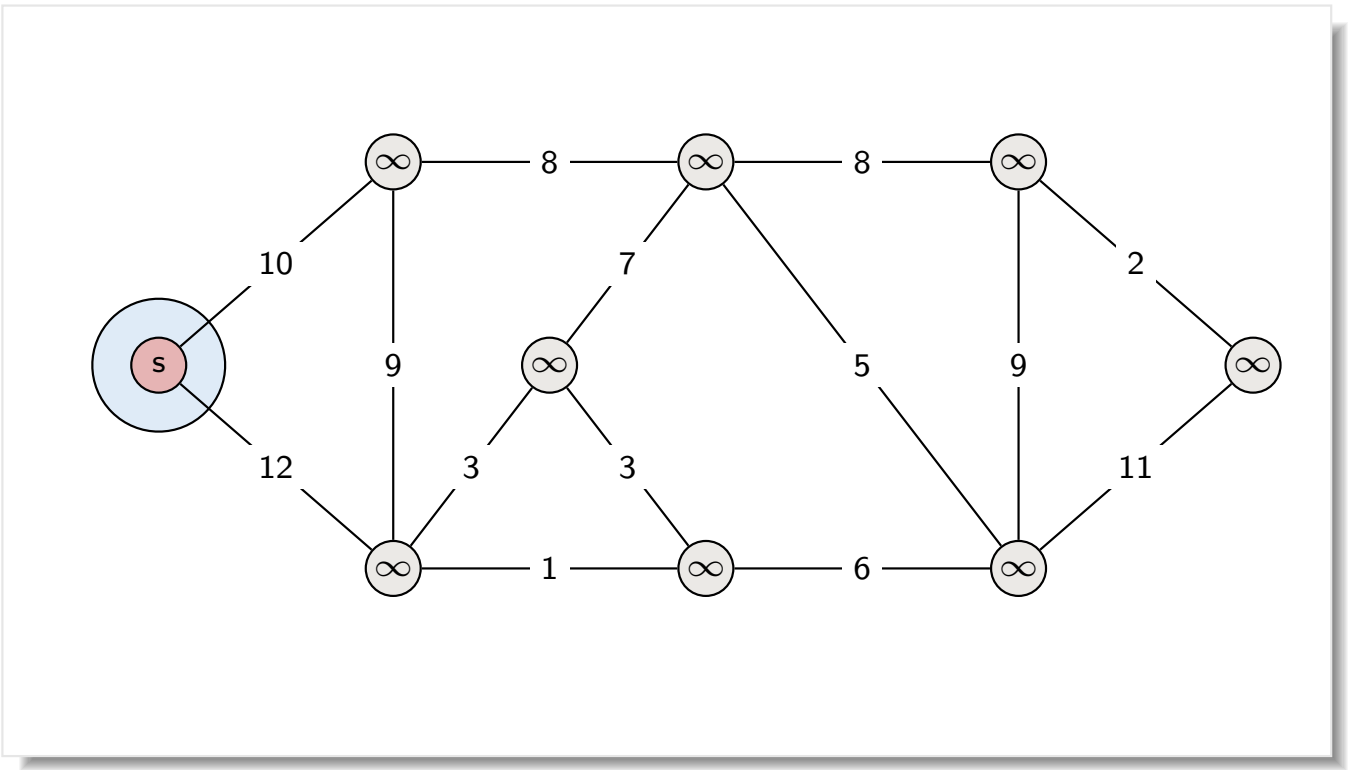
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



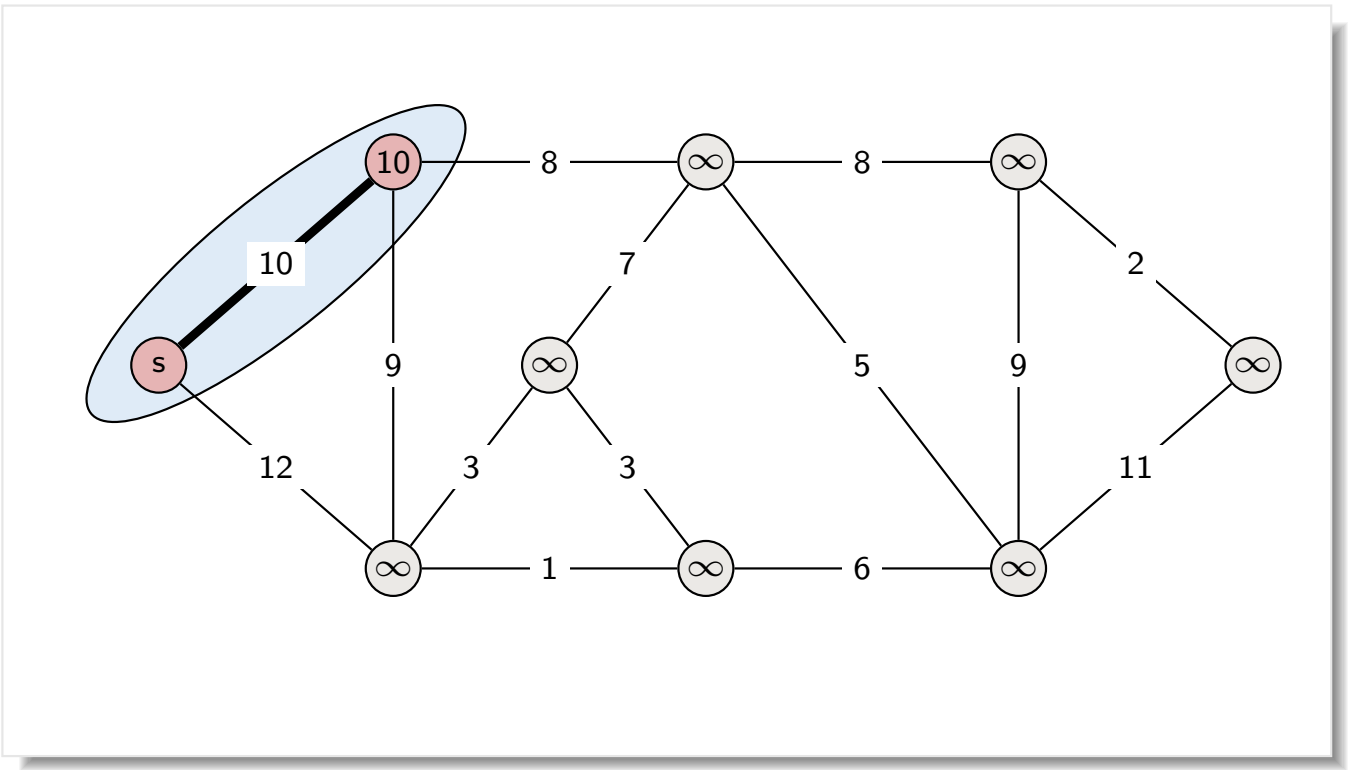
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



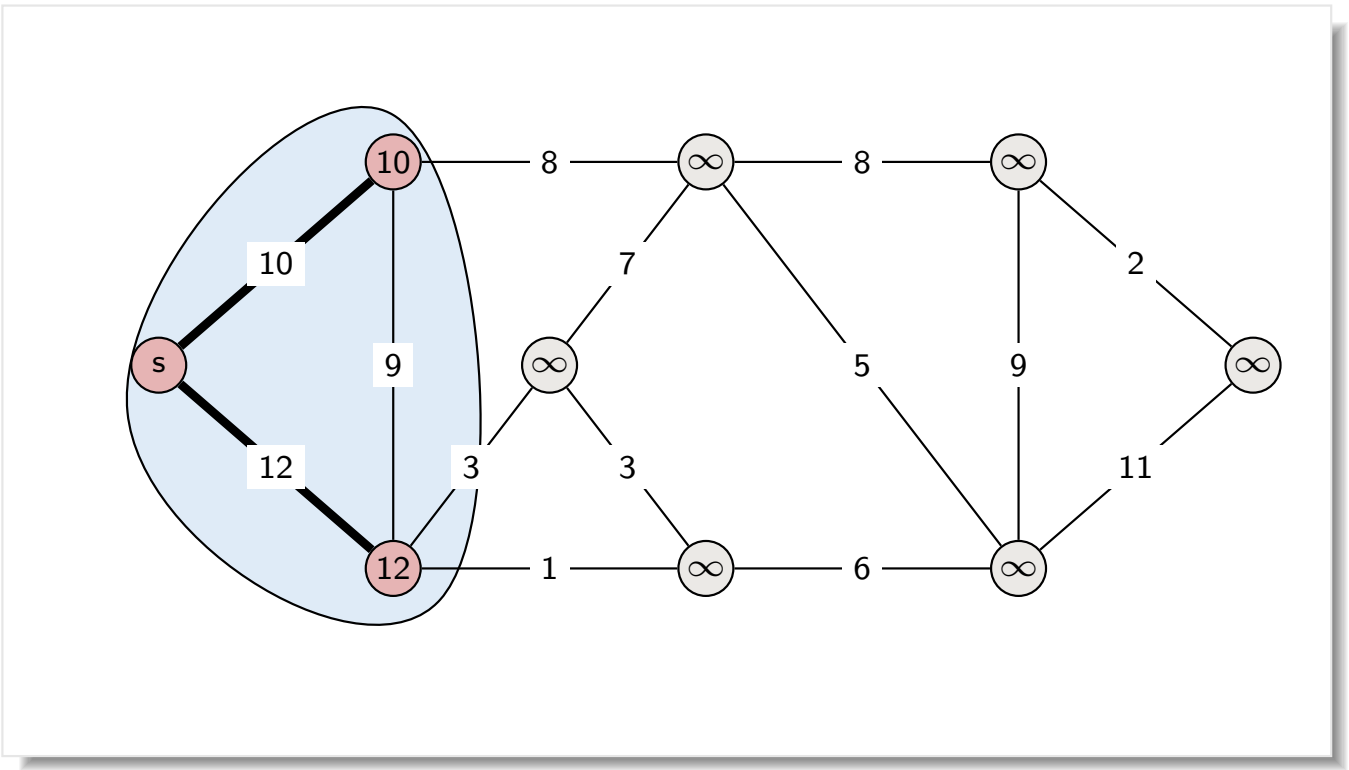
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



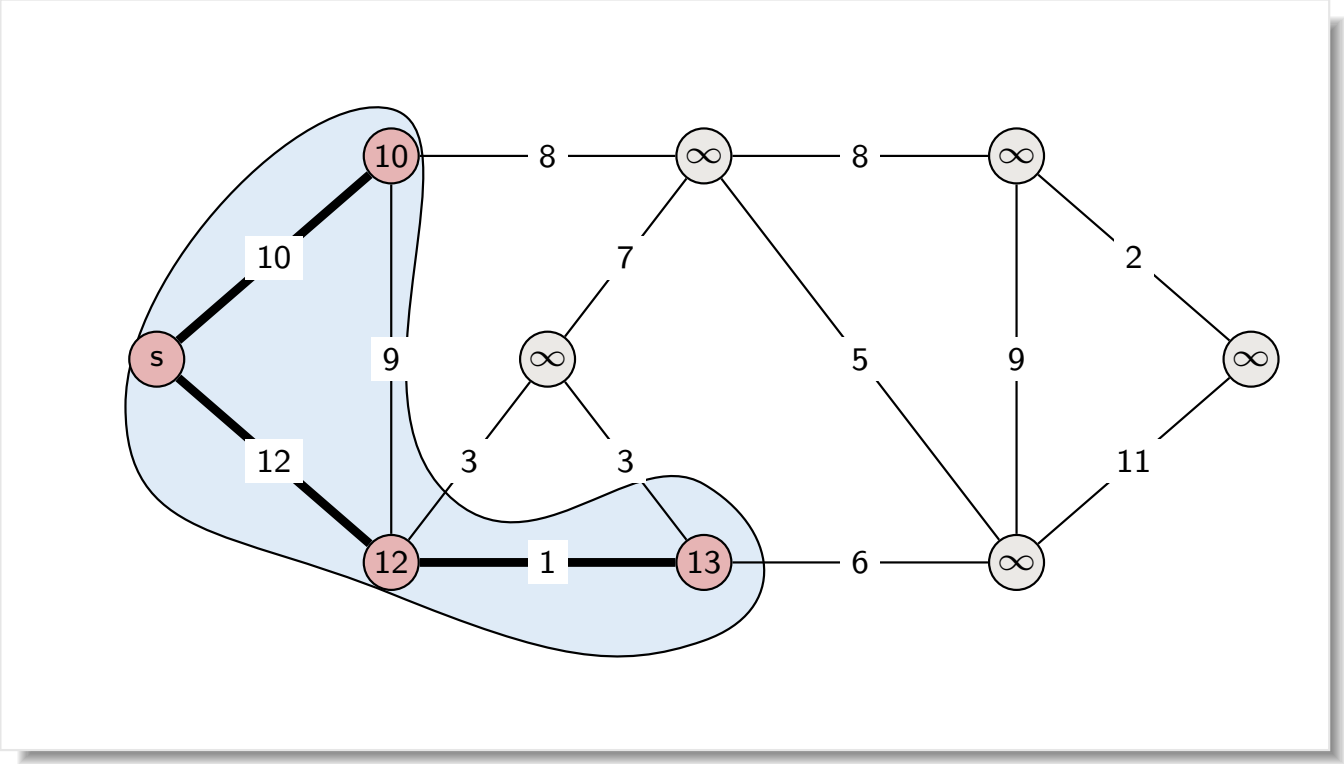
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



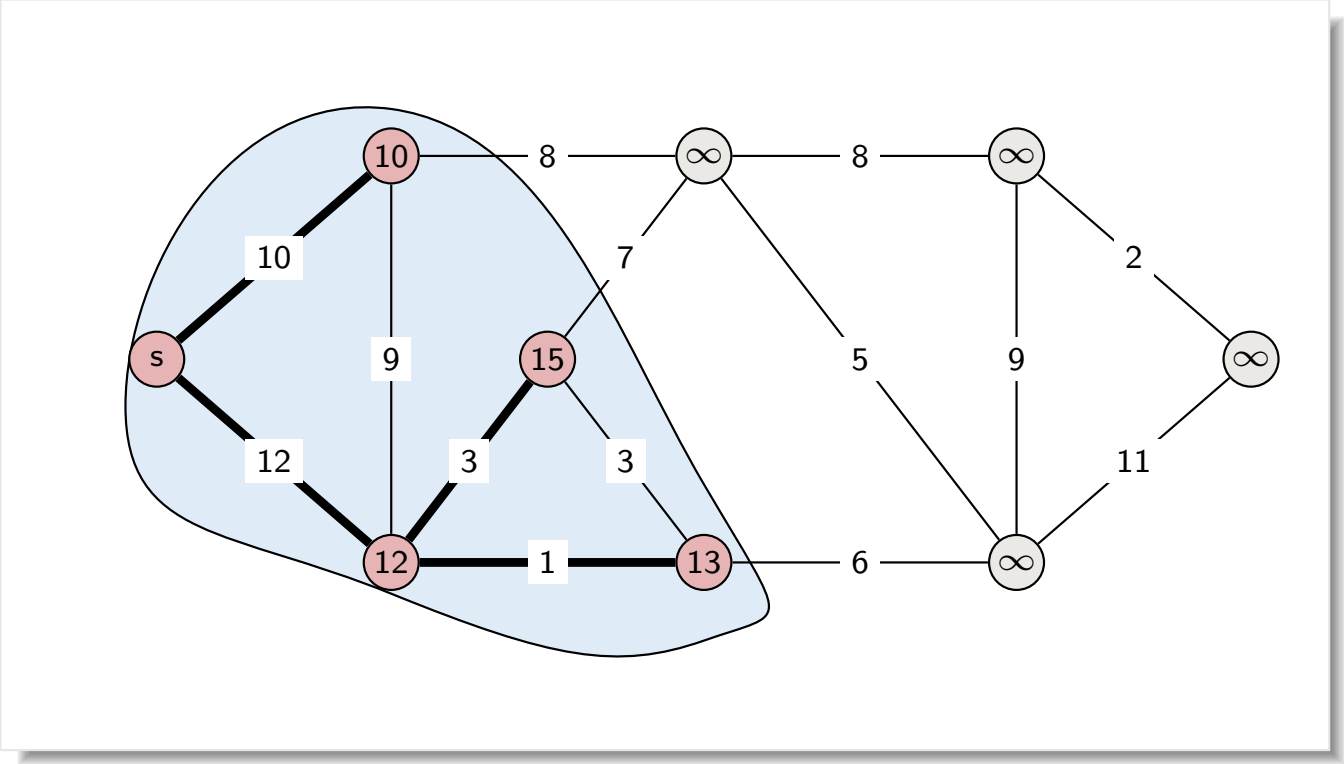
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



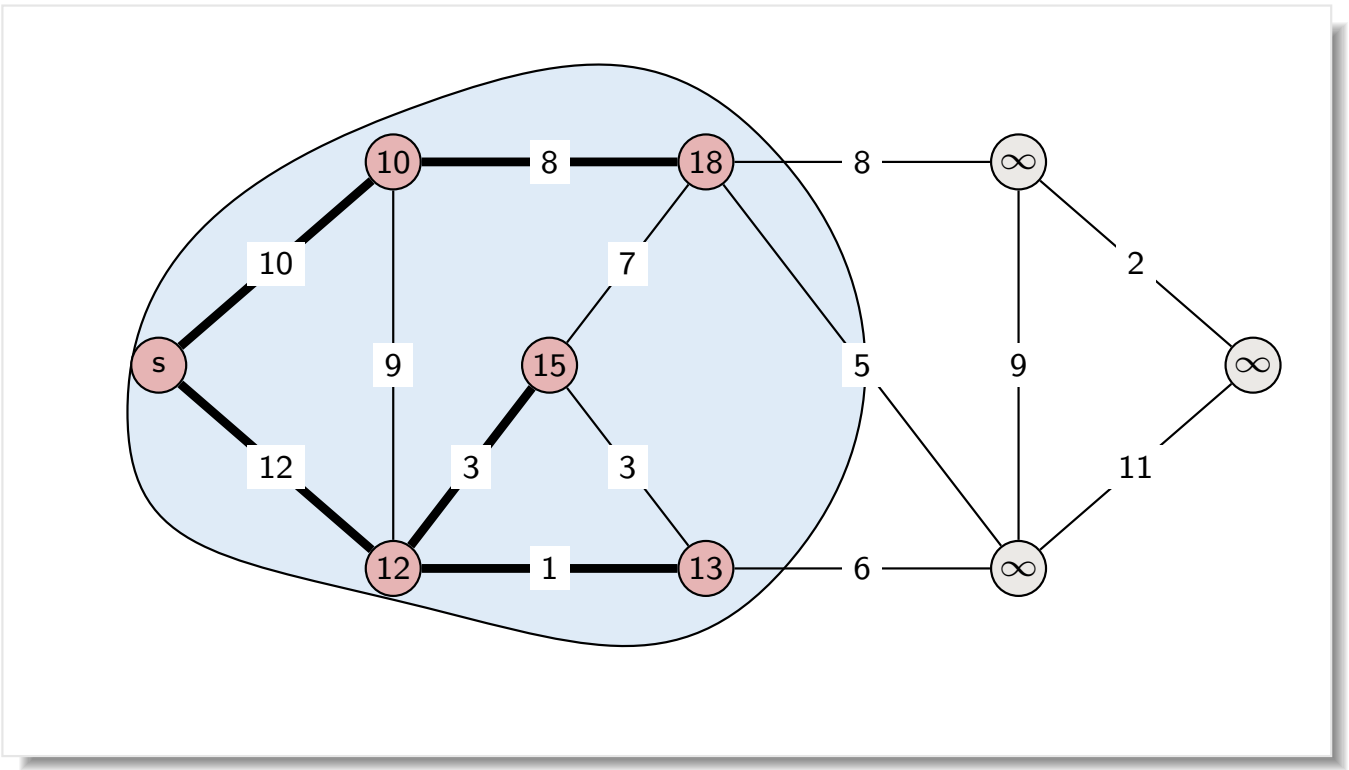
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



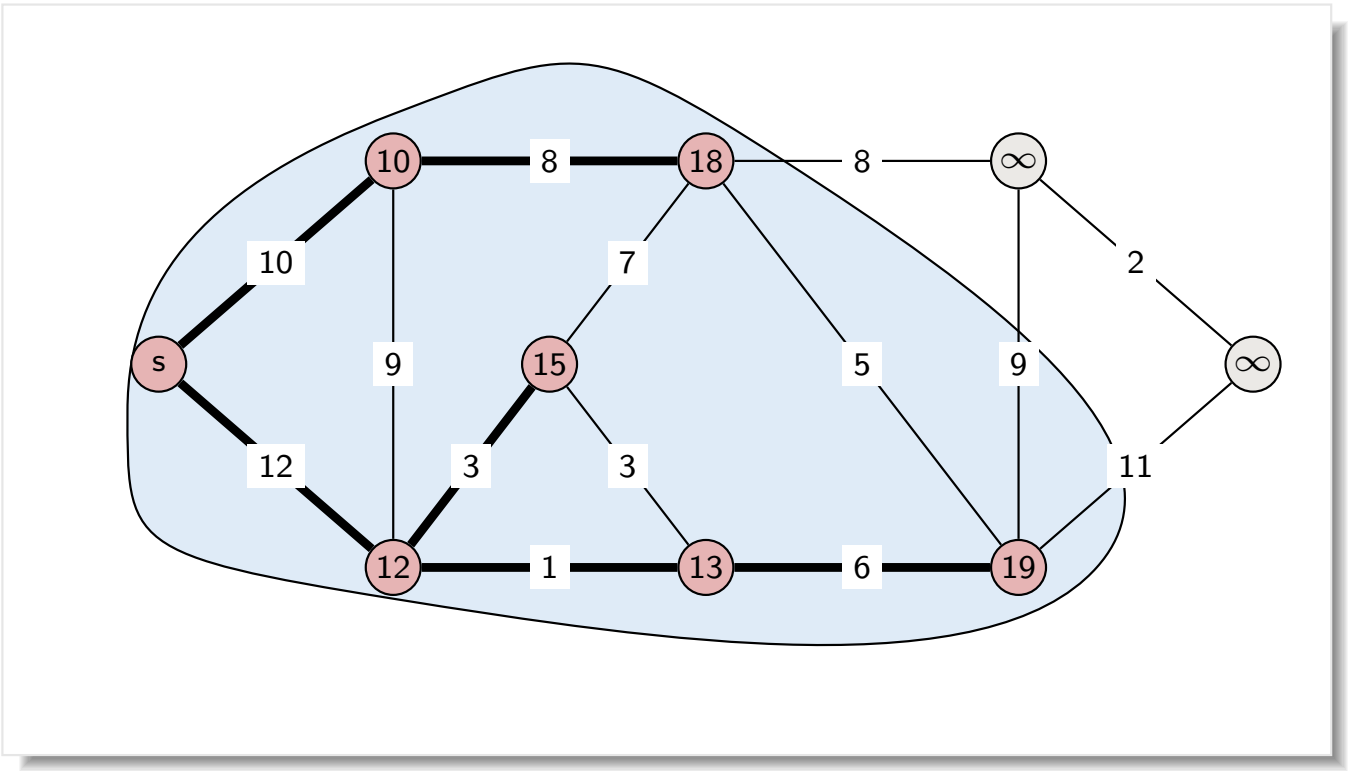
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



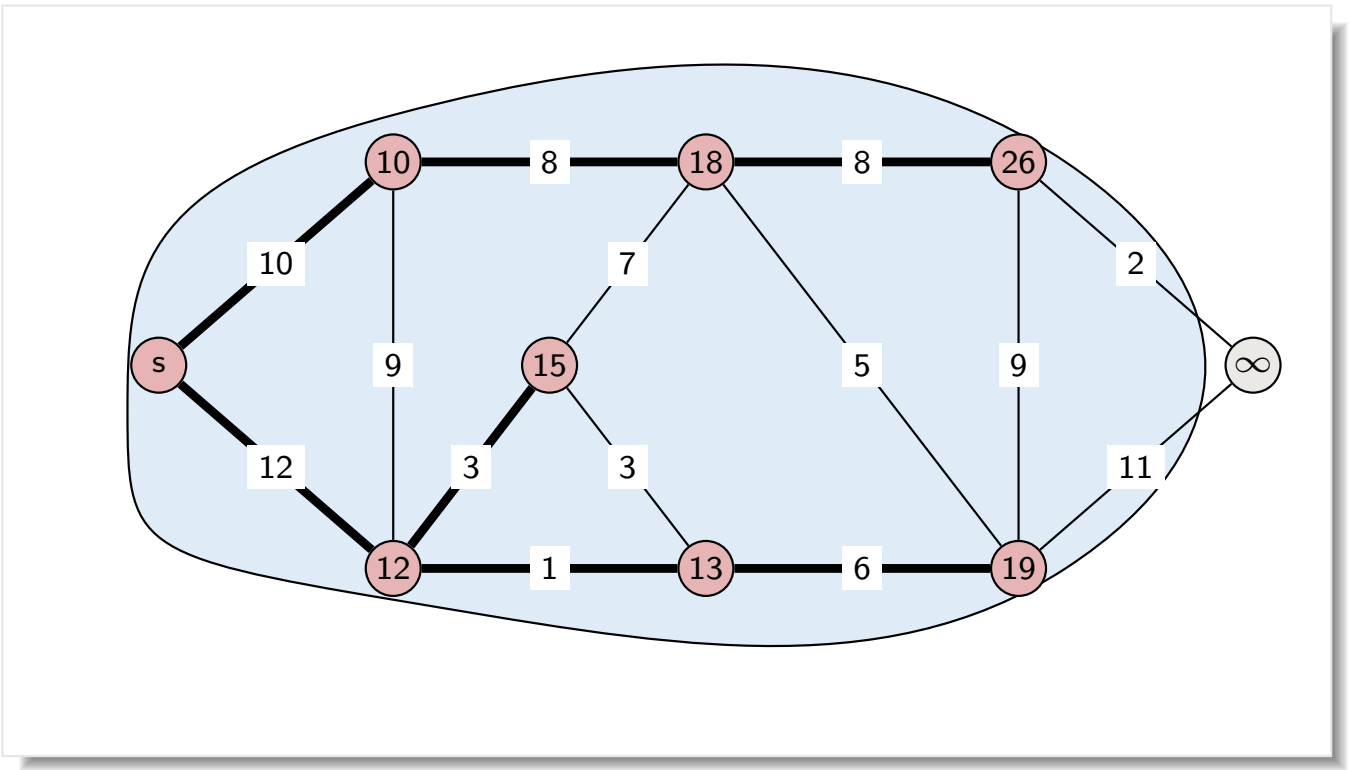
Dijkstra's algorithm

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 $u \in S$, $u.d + w(u, v)$)



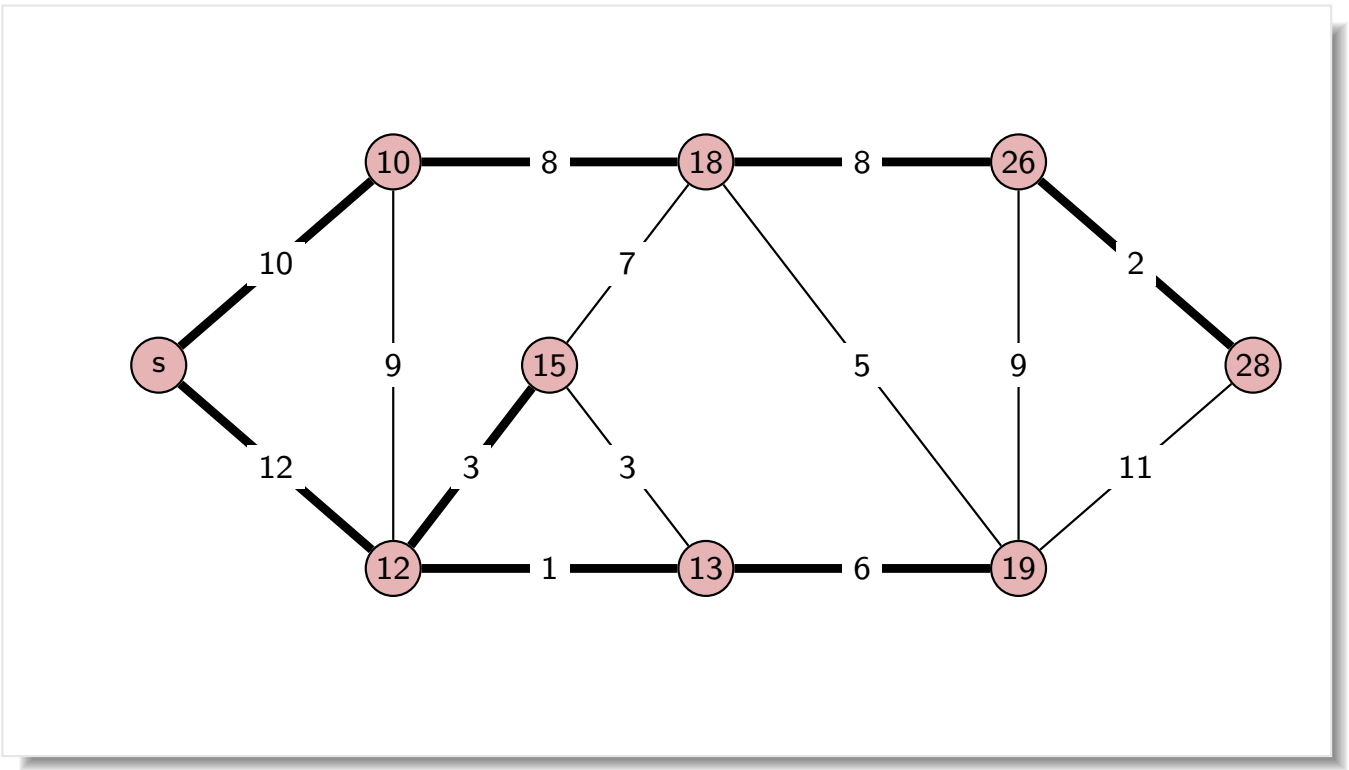
Dijkstra's algorithm

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"