1 Weighted Graphs

Given weighted, directed graph $G = (V, E)$, $w : E \to \mathbb{R}$.

- SSSP (single source shortest paths):
  find shortest path from a source node s to all other vertices

- APSP (all pairs shortest paths):
  find shortest paths between all vertex pairs

Reminder:
A path in a graph between x and y is a sequence of vertices $v_1, \ldots, v_k$ (not necessarily distinct), s.t.

- $x = v_1$, $v_k = y$
- $(v_i, v_{i+1}) \in E$ for all $i \in 1, \ldots, k-1$

(note that alternative definitions exists)

1 Weighted Graphs

Given weighted, directed graph $G = (V, E)$, $w : E \to \mathbb{R}$.

- SSSP (single source shortest paths):
  find shortest path from a source node s to all other vertices

- APSP (all pairs shortest paths):
  find shortest paths between all vertex pairs

Reminder:
A path in a graph between x and y is a sequence of vertices $v_1, \ldots, v_k$ (not necessarily distinct), s.t.

- $x = v_1$, $v_k = y$
- $(v_i, v_{i+1}) \in E$ for all $i \in 1, \ldots, k-1$

(note that alternative definitions exists)

1 Weighted Graphs

The distance $d(x,y)$ between two vertices/nodes x and y is the length of a shortest path.

Formally:

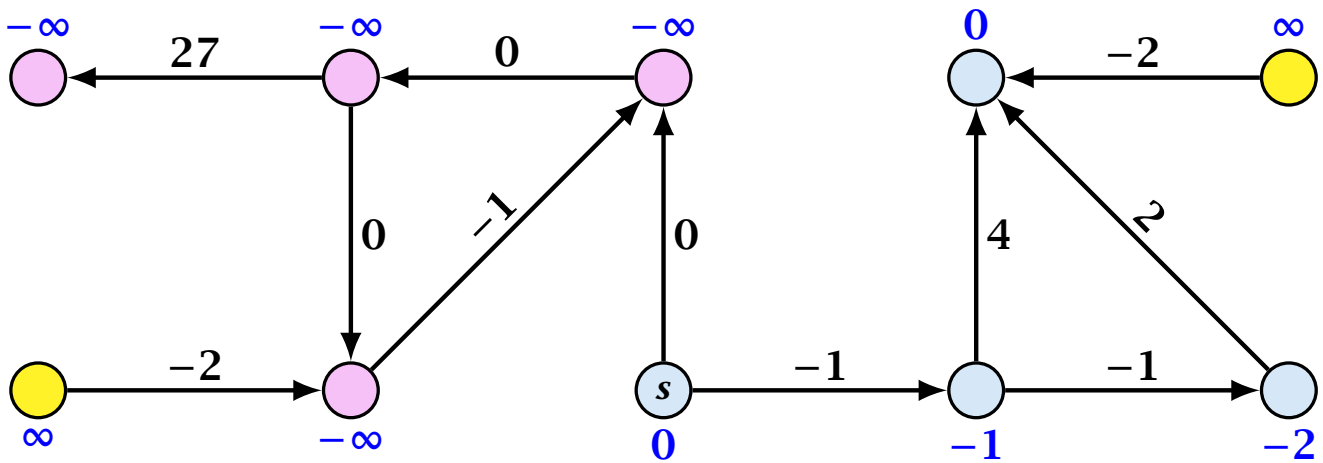
$$d(s,v) = \begin{cases} +\infty & \text{no path from } s \text{ to } v \\ -\infty & \text{no shortest path from } s \text{ to } v \\ \min_{p \text{ path from } s \text{ to } v} w(p) & \text{otherwise} \end{cases}$$

1 Weighted Graphs

The distance $d(x,y)$ between two vertices/nodes x and y is the length of a shortest path.

Formally:

$$d(s,v) = \begin{cases} +\infty & \text{no path from } s \text{ to } v \\ -\infty & \text{no shortest path from } s \text{ to } v \\ \min_{p \text{ path from } s \text{ to } v} w(p) & \text{otherwise} \end{cases}$$



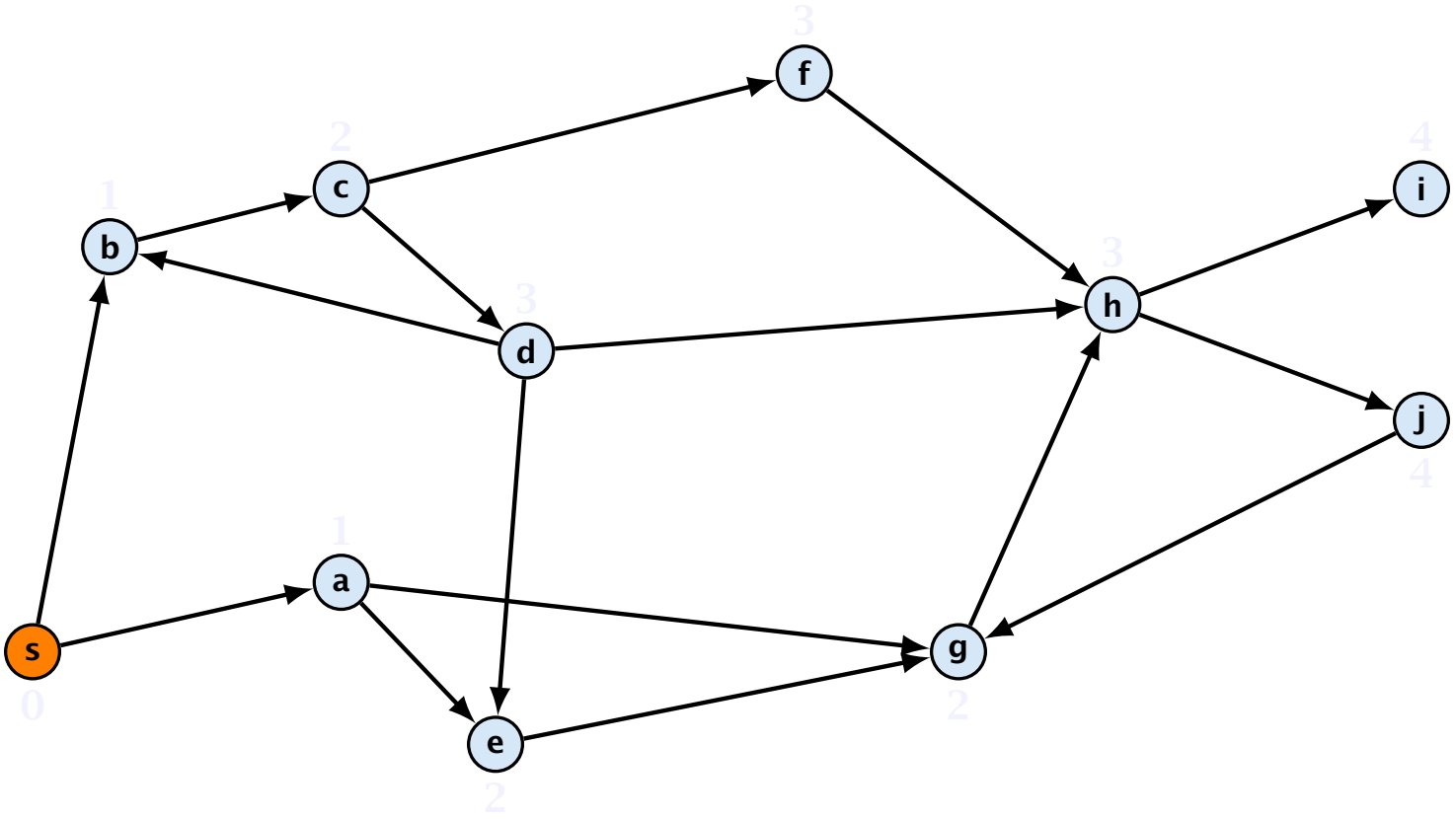
1 Weighted Graphs

Variants of the problem:

- uniform weights (all weights 1)
  BFS, running time $\mathcal{O}(m+n)$
- arbitrary weights in a DAG (**D**irected **A**cyclic **G**raph)
  topological sort + BFS, running time $\mathcal{O}(m+n)$
- general graph with positive weights
  Dijkstras Algorithm,
  running time $\mathcal{O}((m+n)\log n)$ or $\mathcal{O}(m+n\log n)$
- general graph with arbitrary weights
  a) without negative cycles
  b) with negative cycles

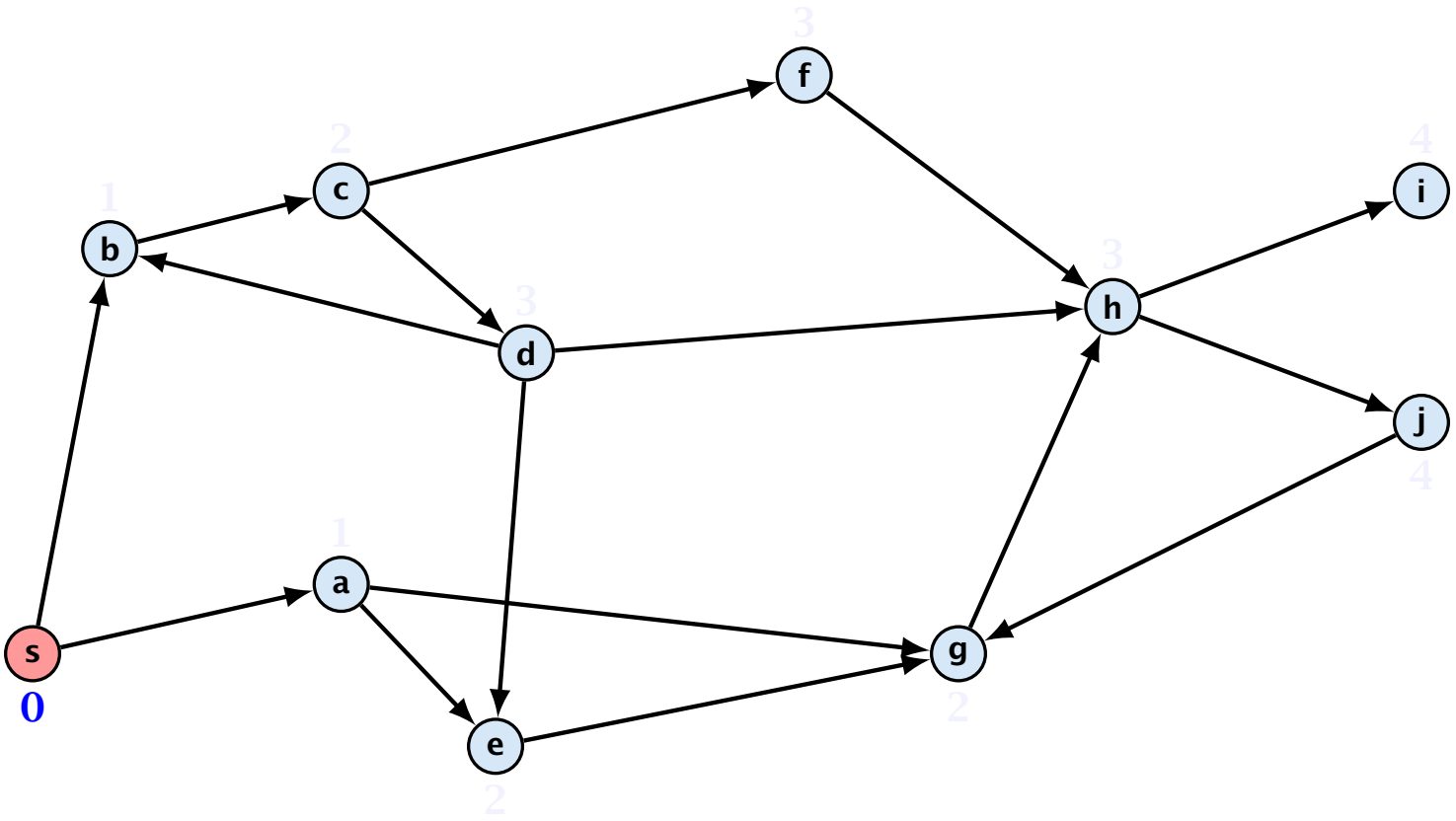
Uniform Weights

For uniform weights a simple BFS works.



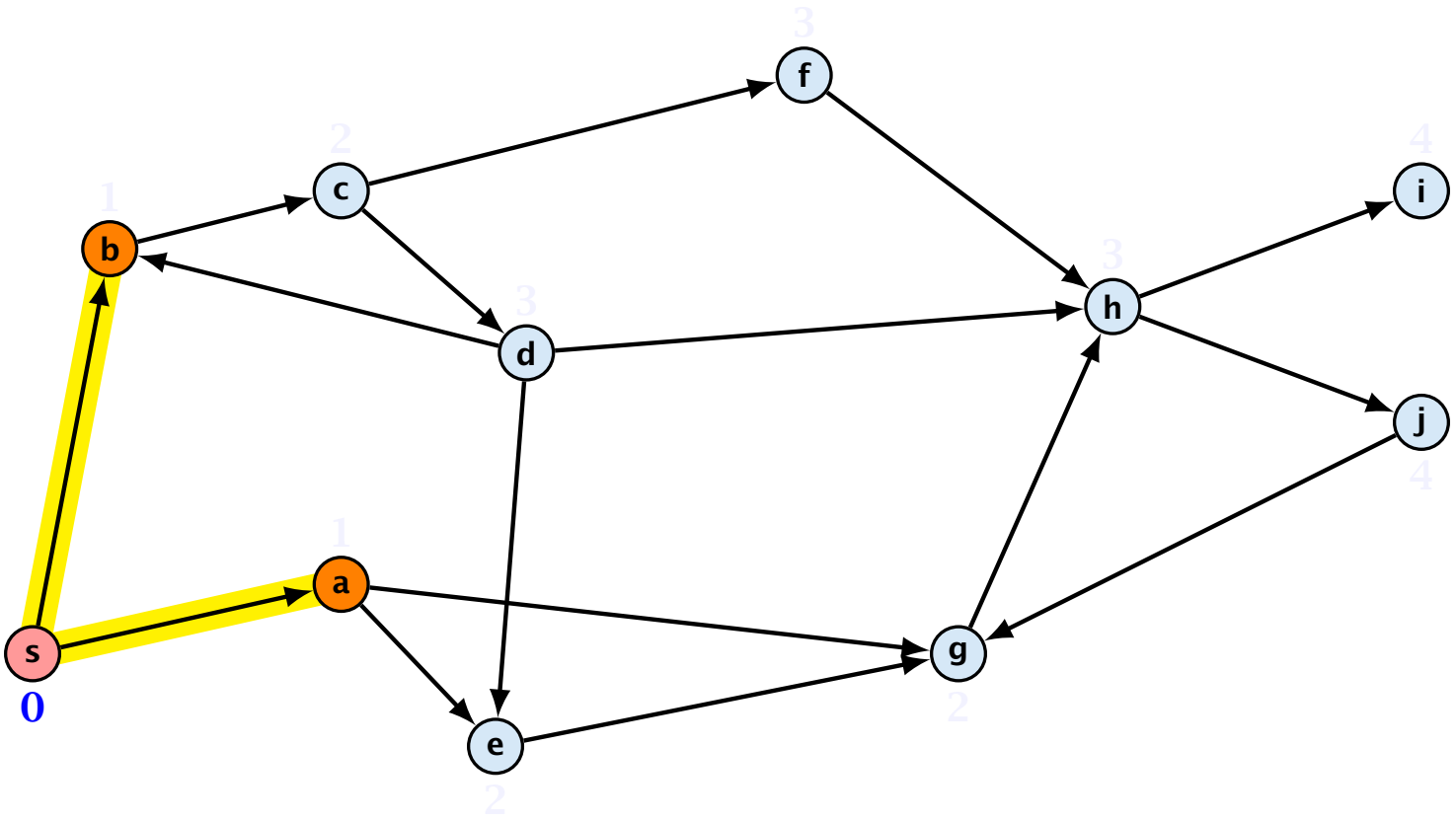
Uniform Weights

For uniform weights a simple BFS works.



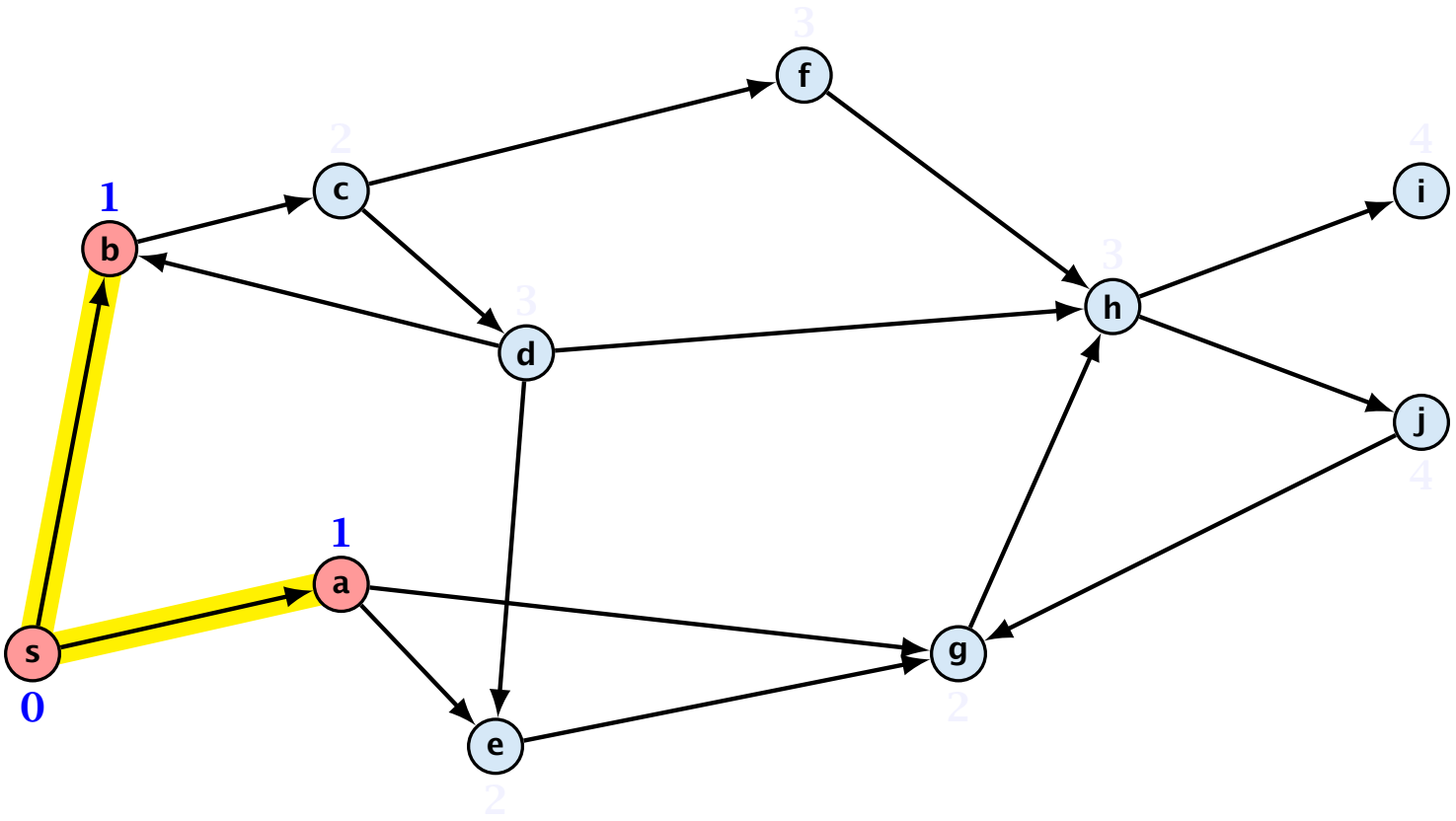
Uniform Weights

For uniform weights a simple BFS works.



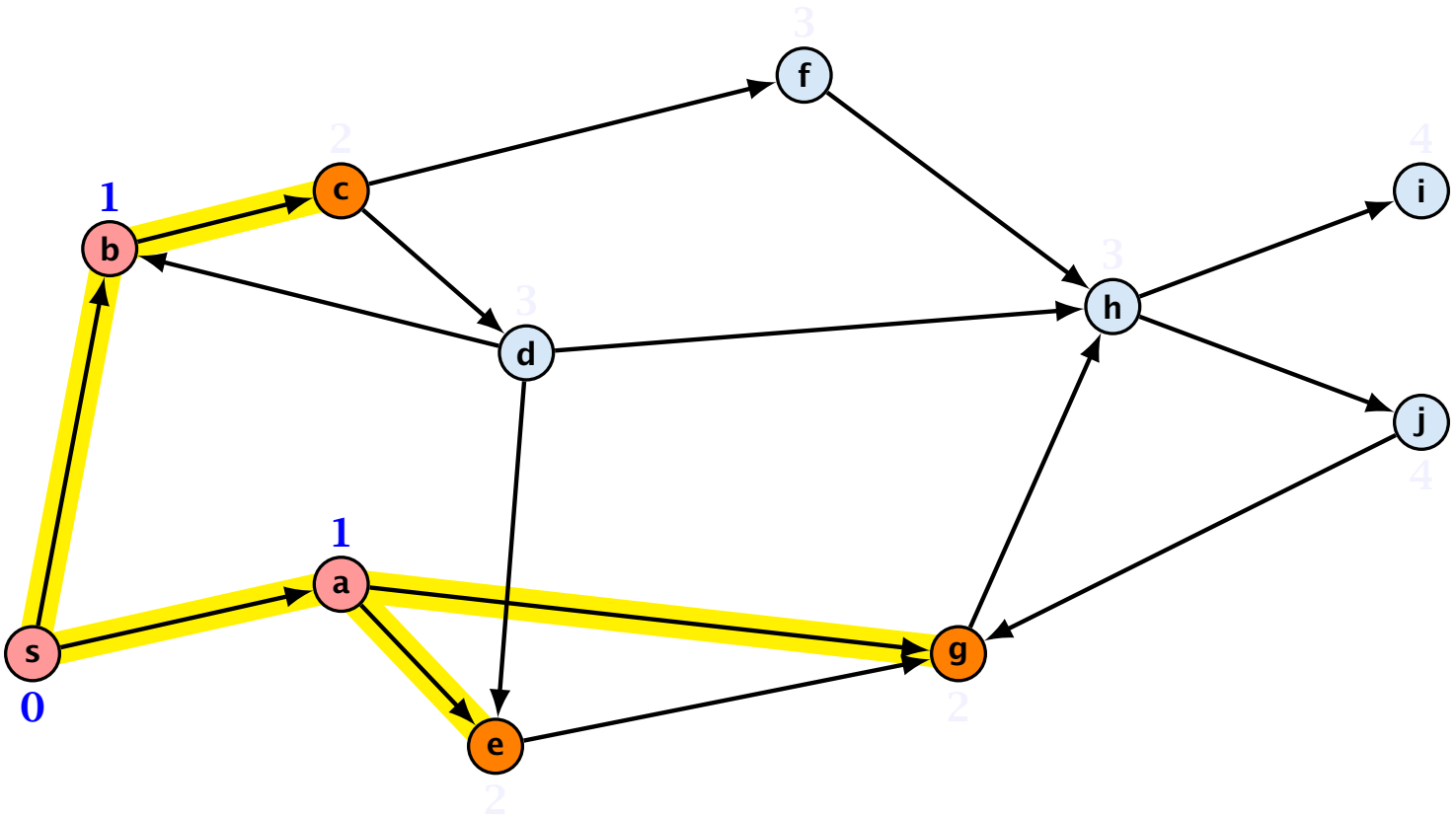
Uniform Weights

For uniform weights a simple BFS works.



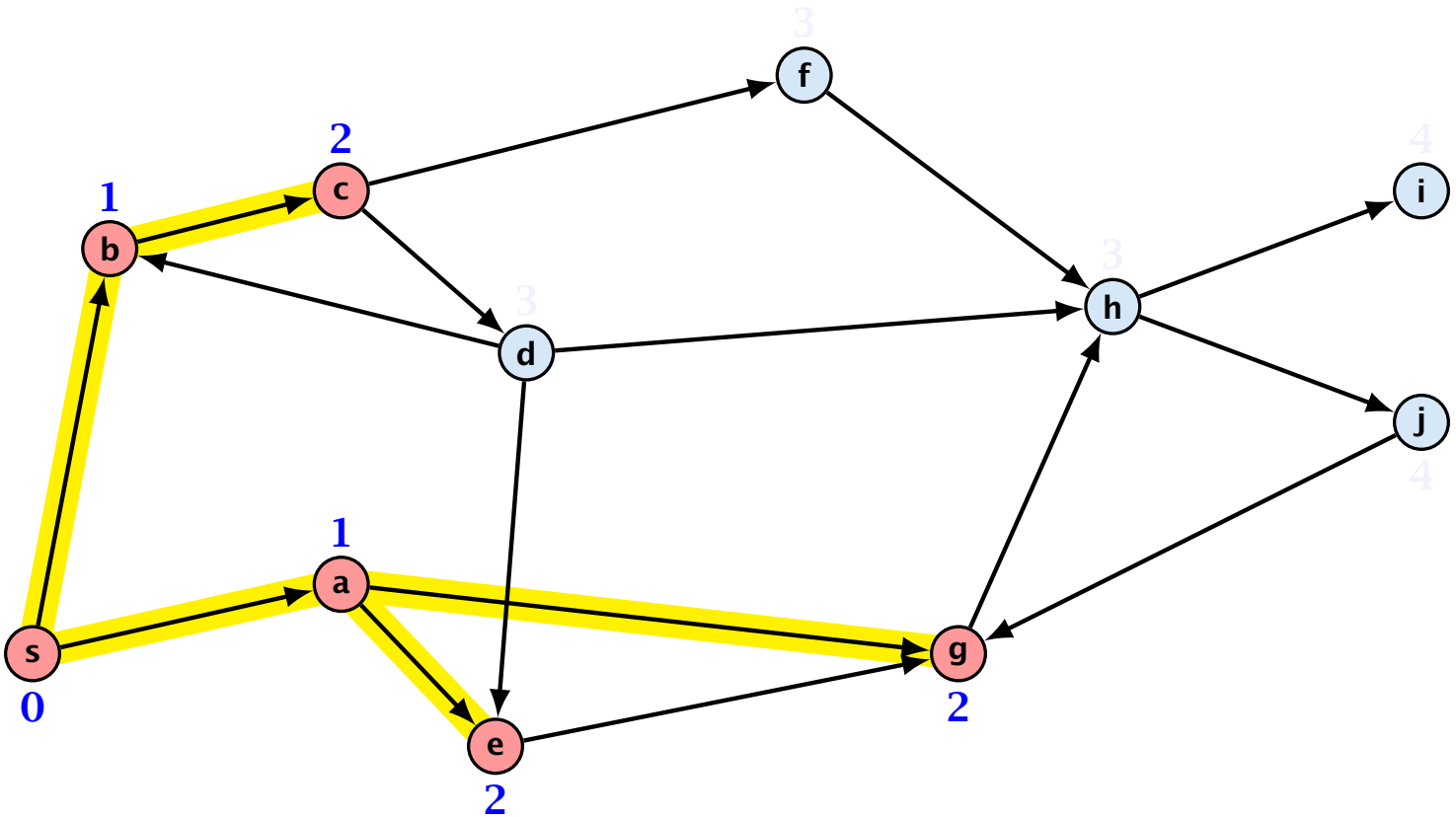
Uniform Weights

For uniform weights a simple BFS works.



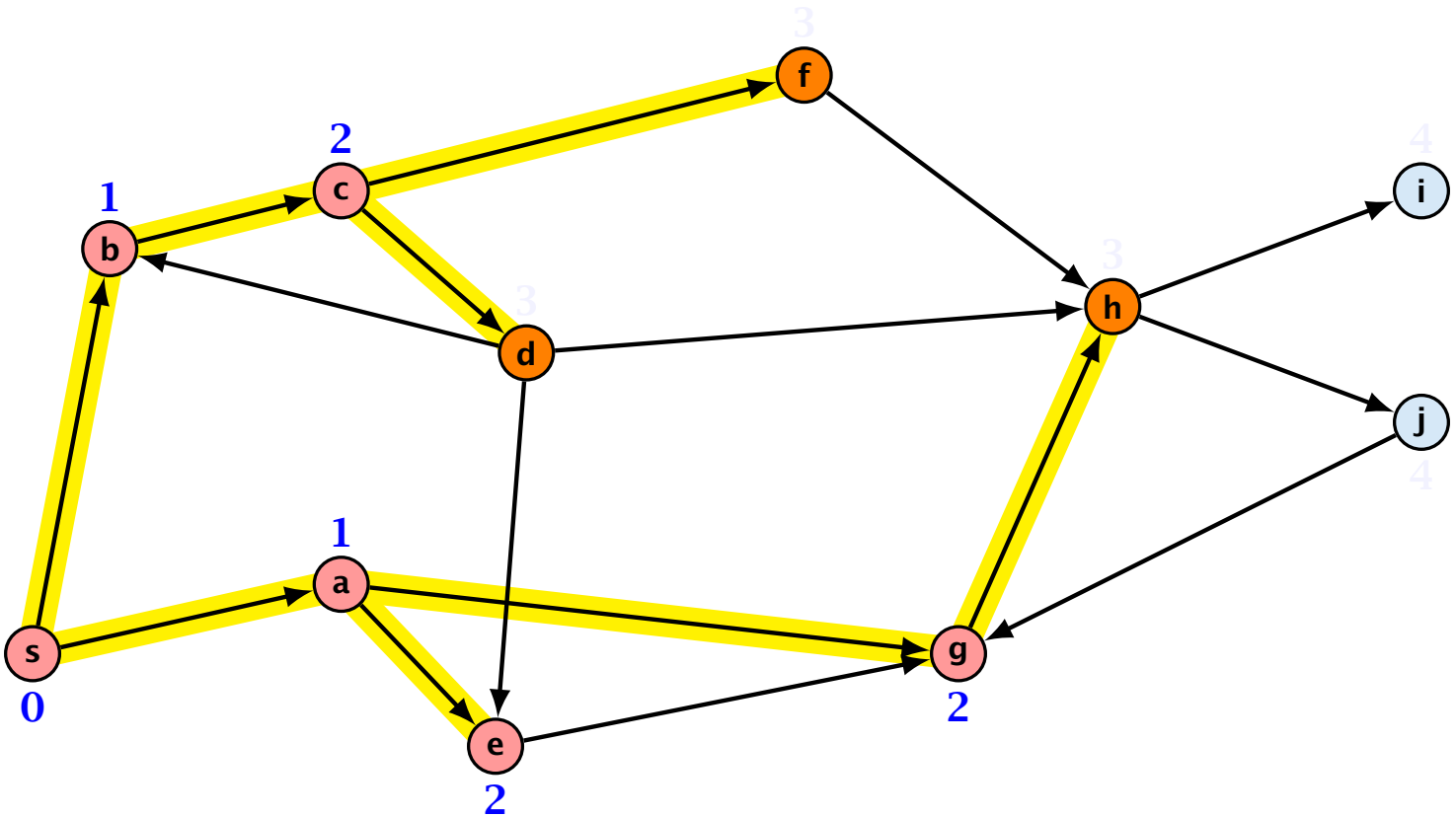
Uniform Weights

For uniform weights a simple BFS works.



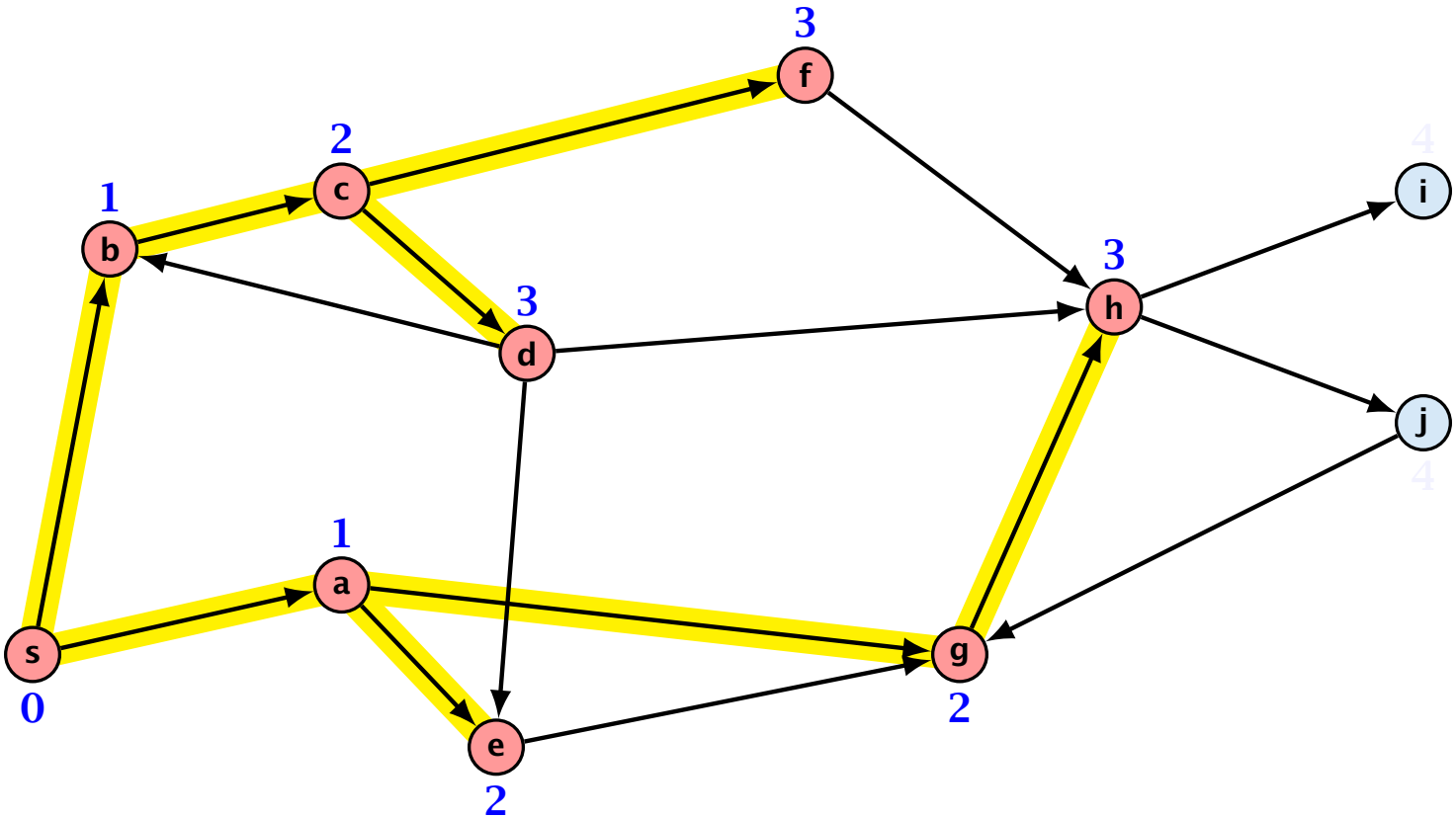
Uniform Weights

For uniform weights a simple BFS works.



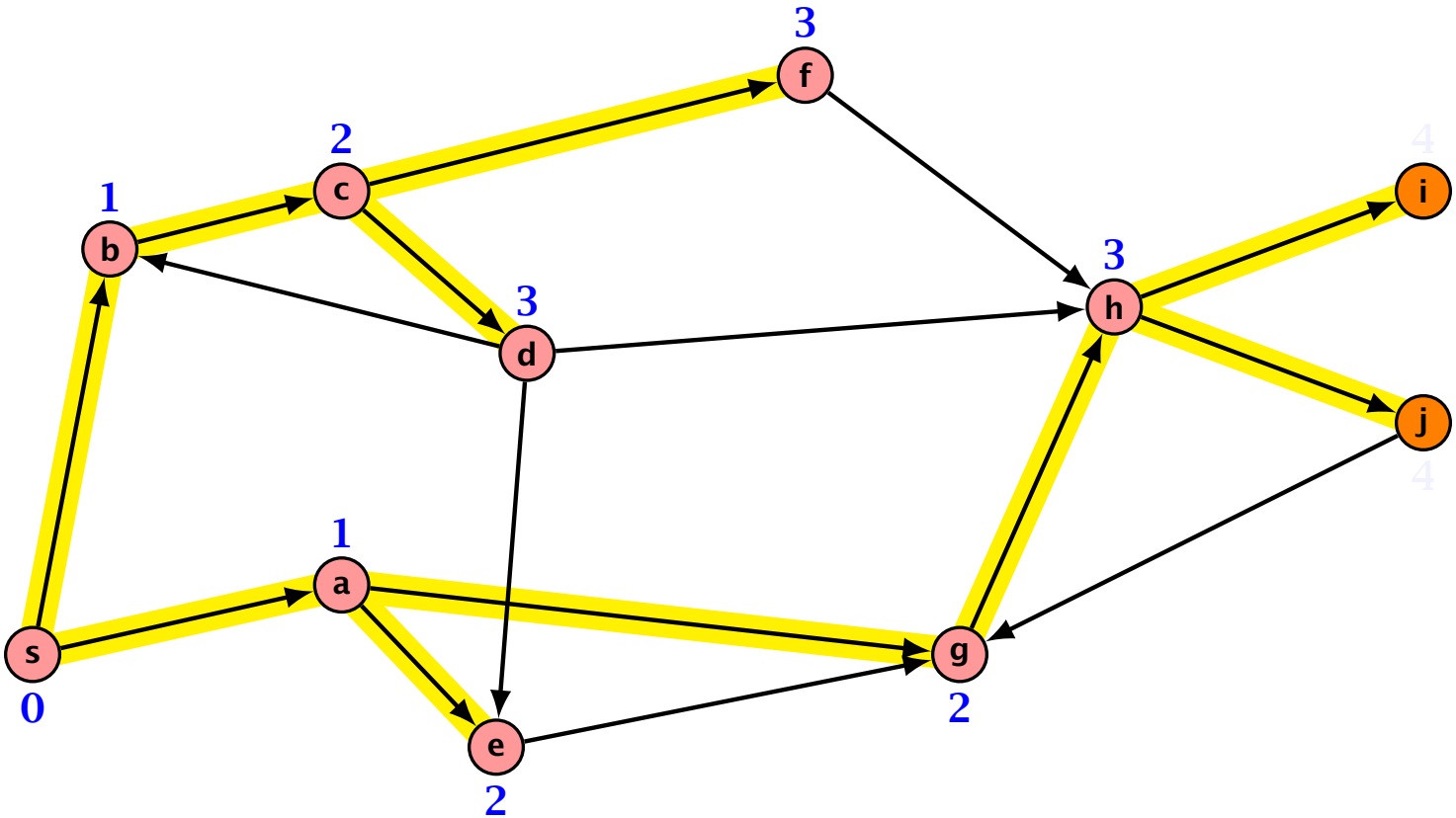
Uniform Weights

For uniform weights a simple BFS works.



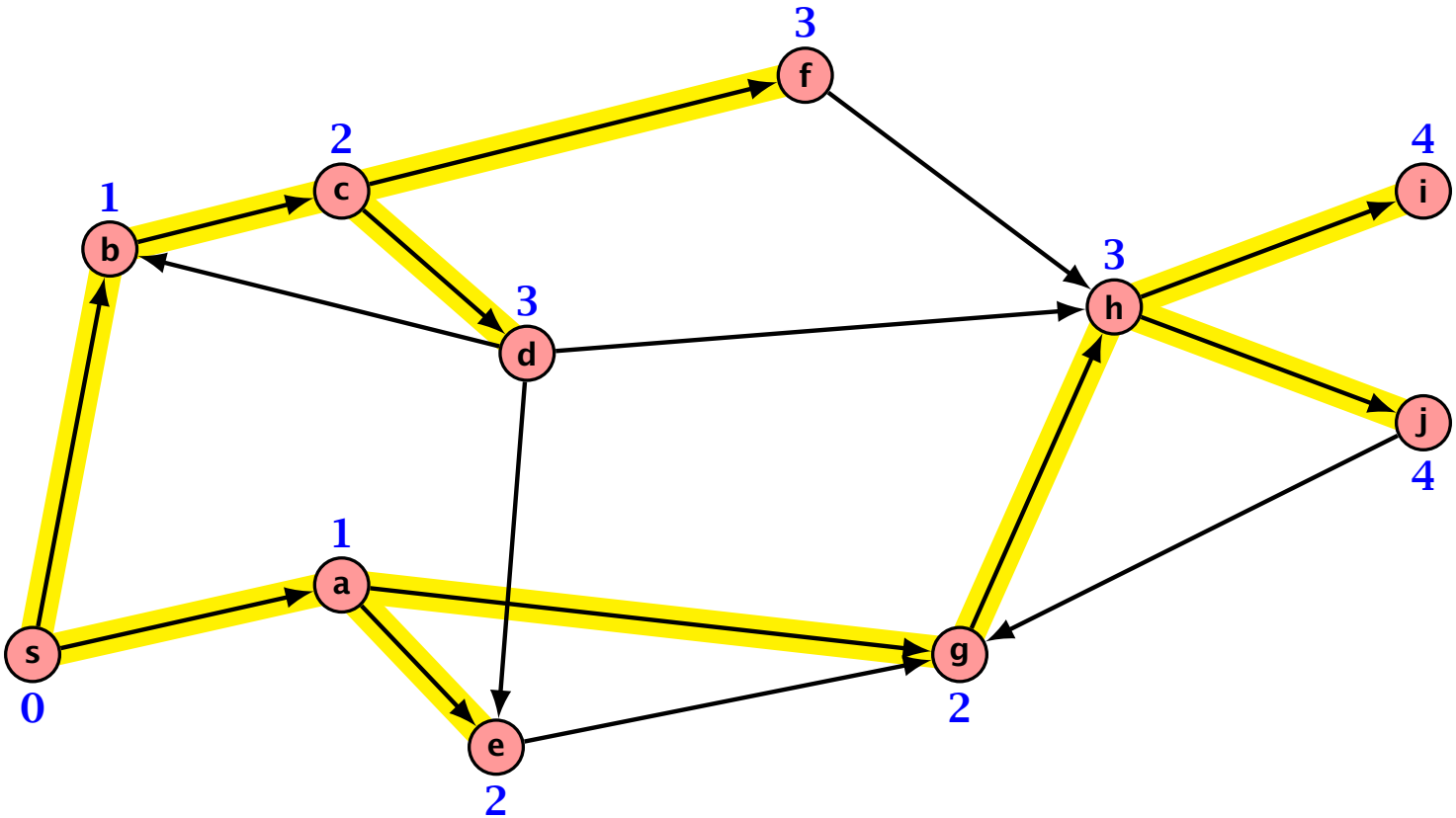
Uniform Weights

For uniform weights a simple BFS works.



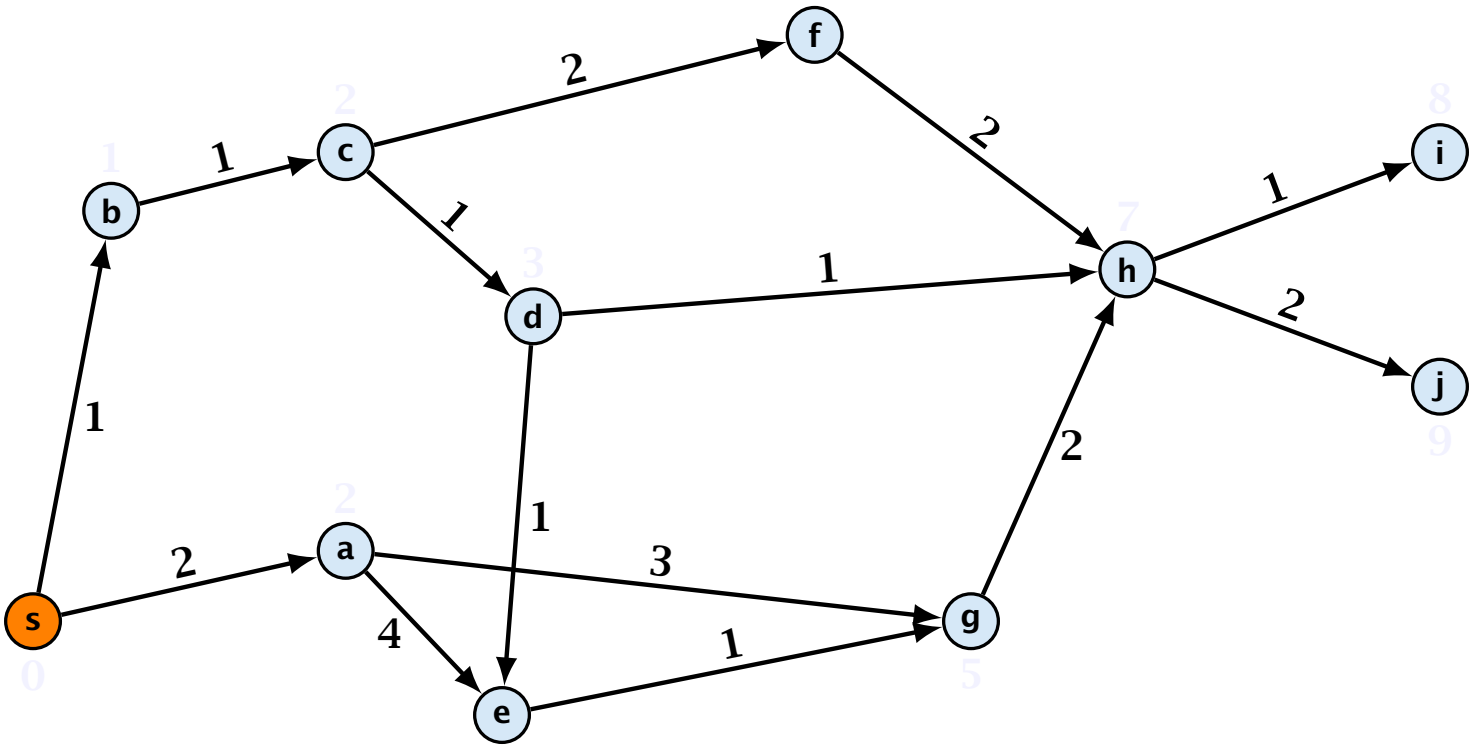
Uniform Weights

For uniform weights a simple BFS works.



Shortest Paths in DAGs

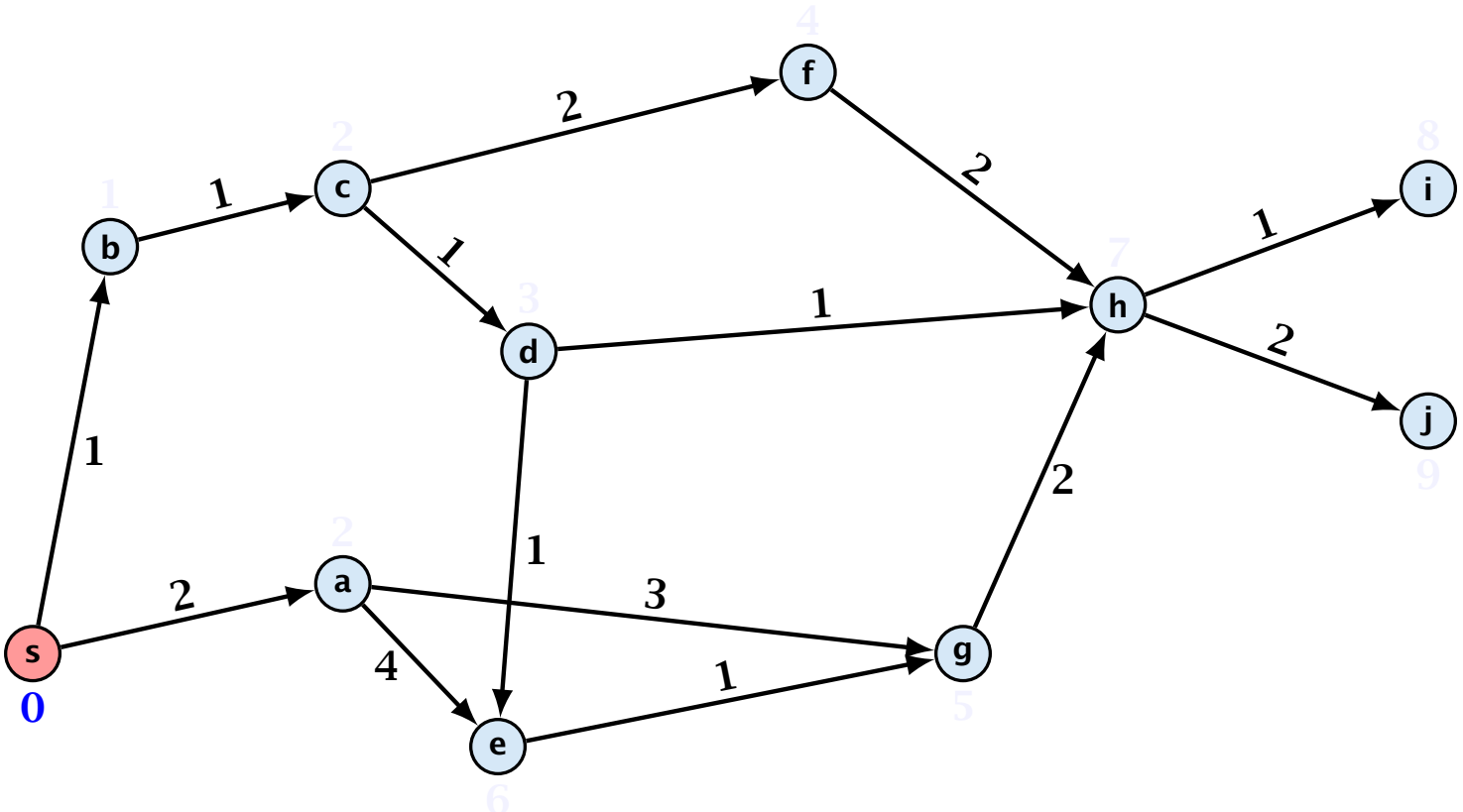
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

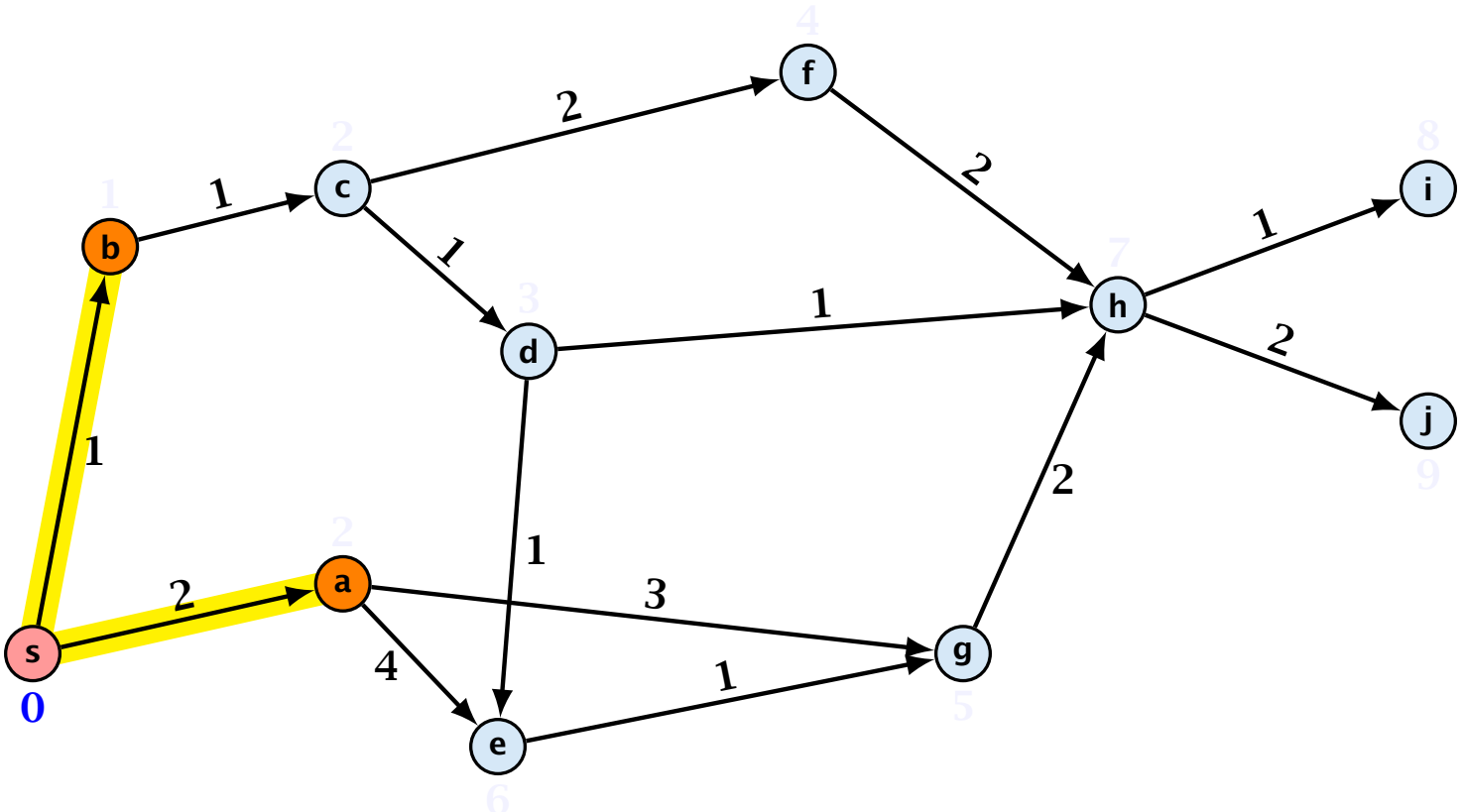
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

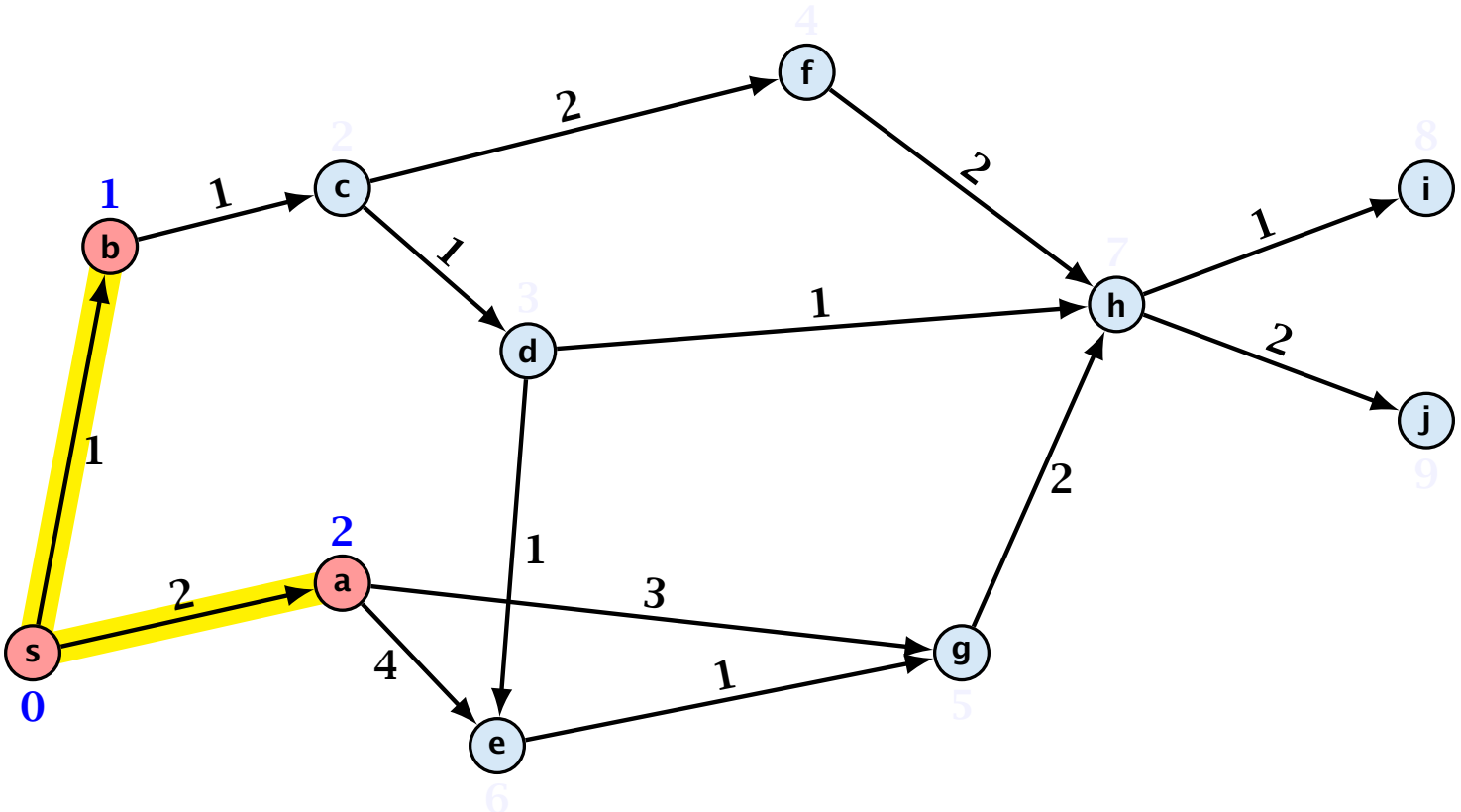
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

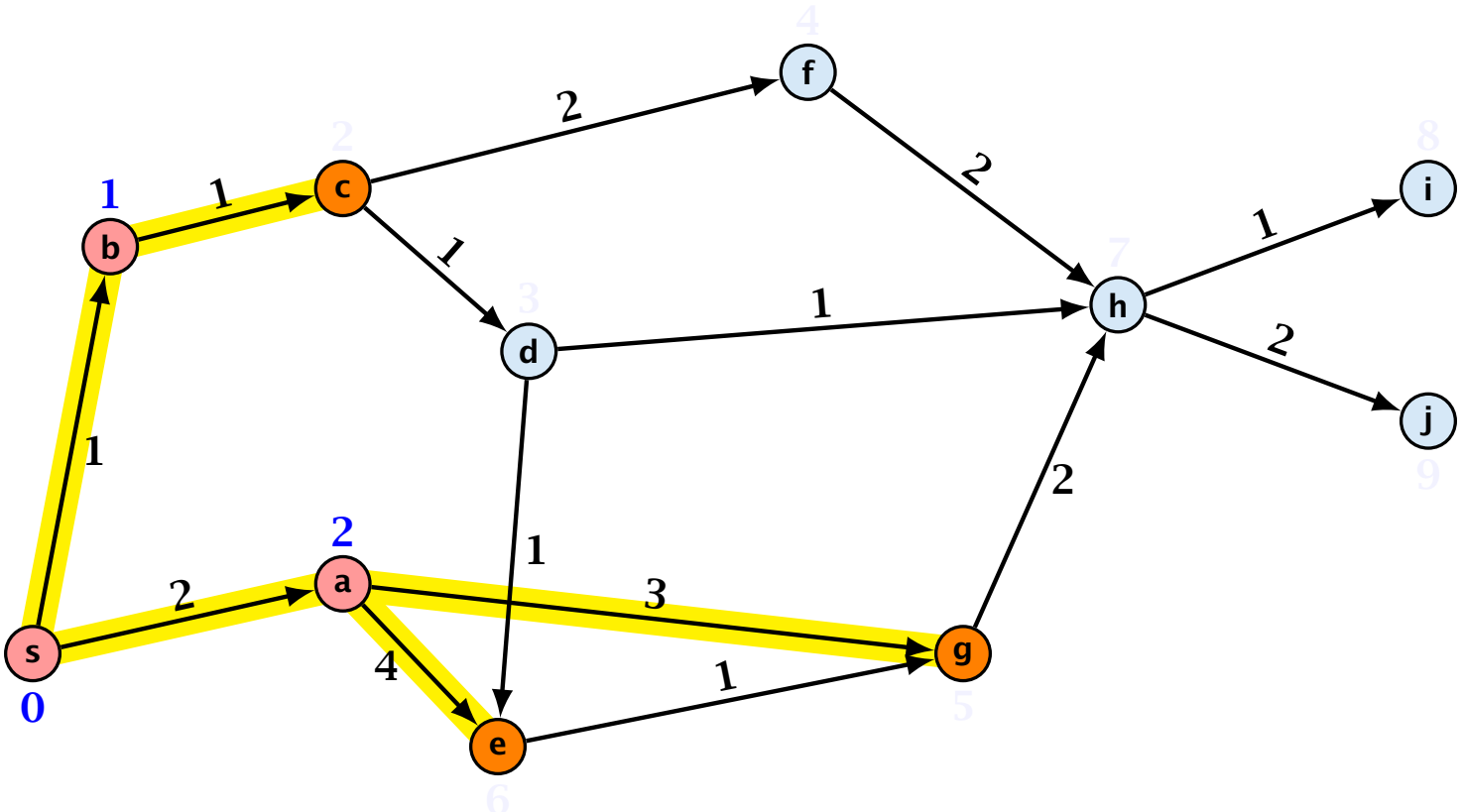
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

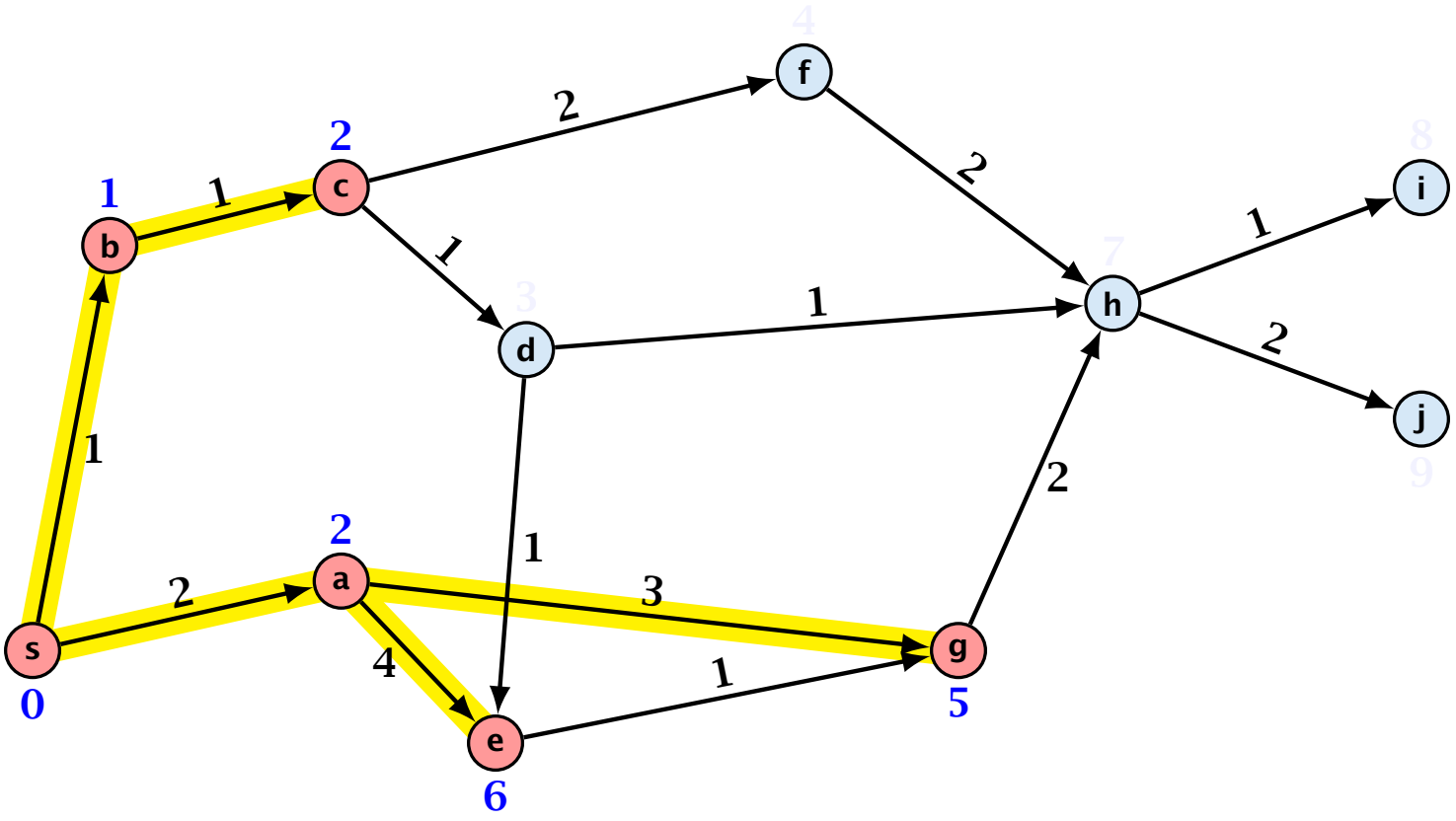
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

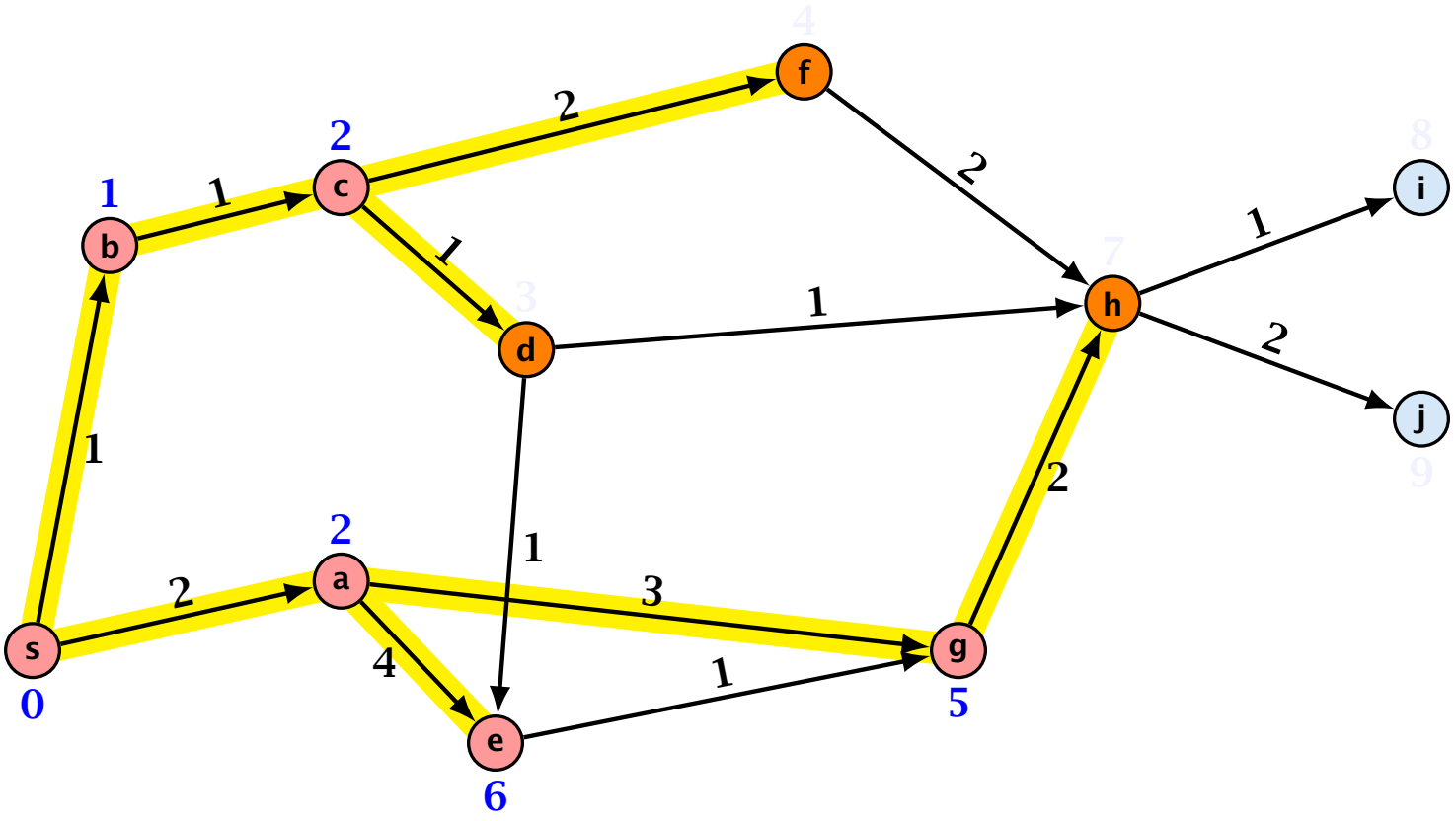
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

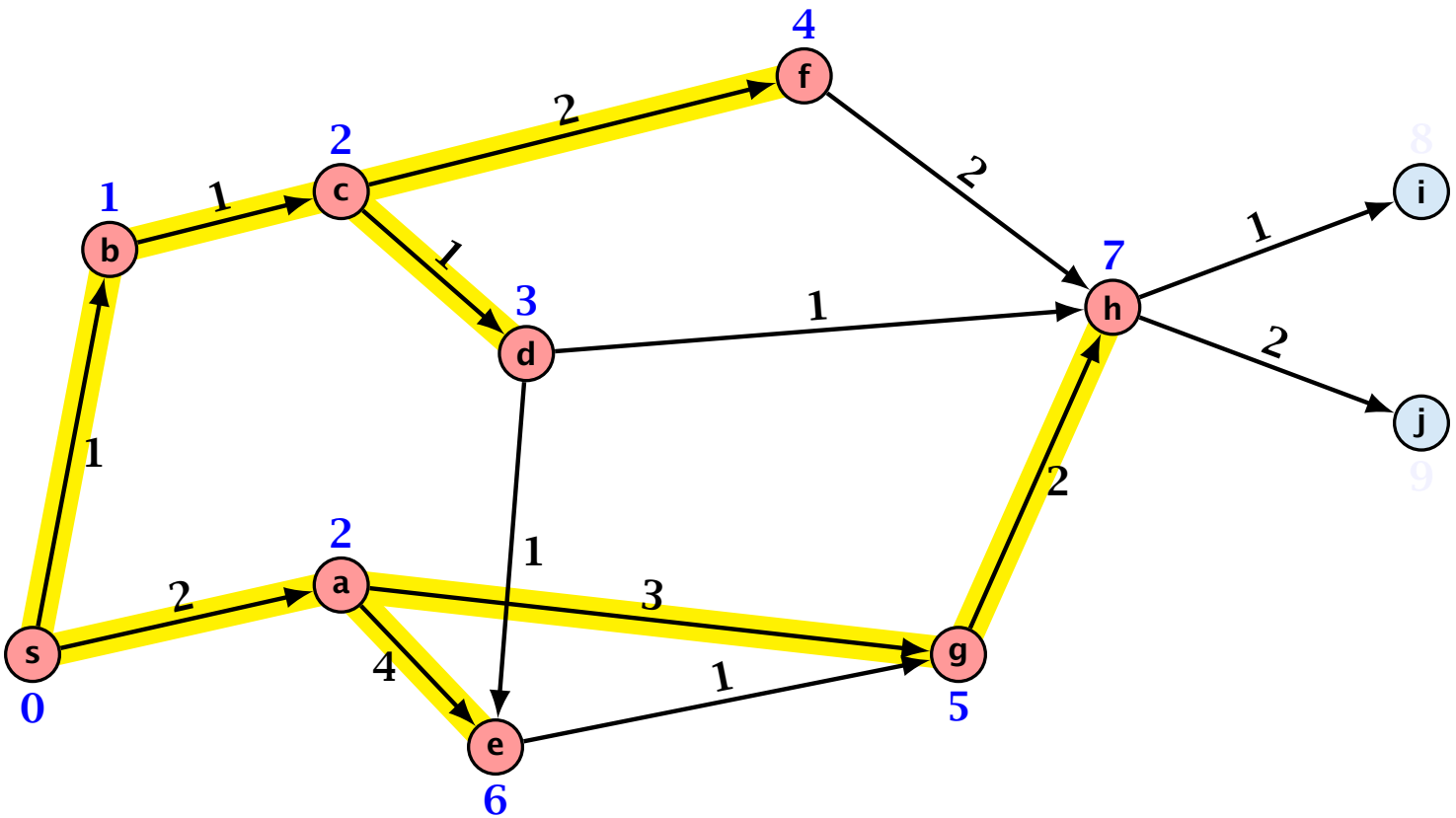
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

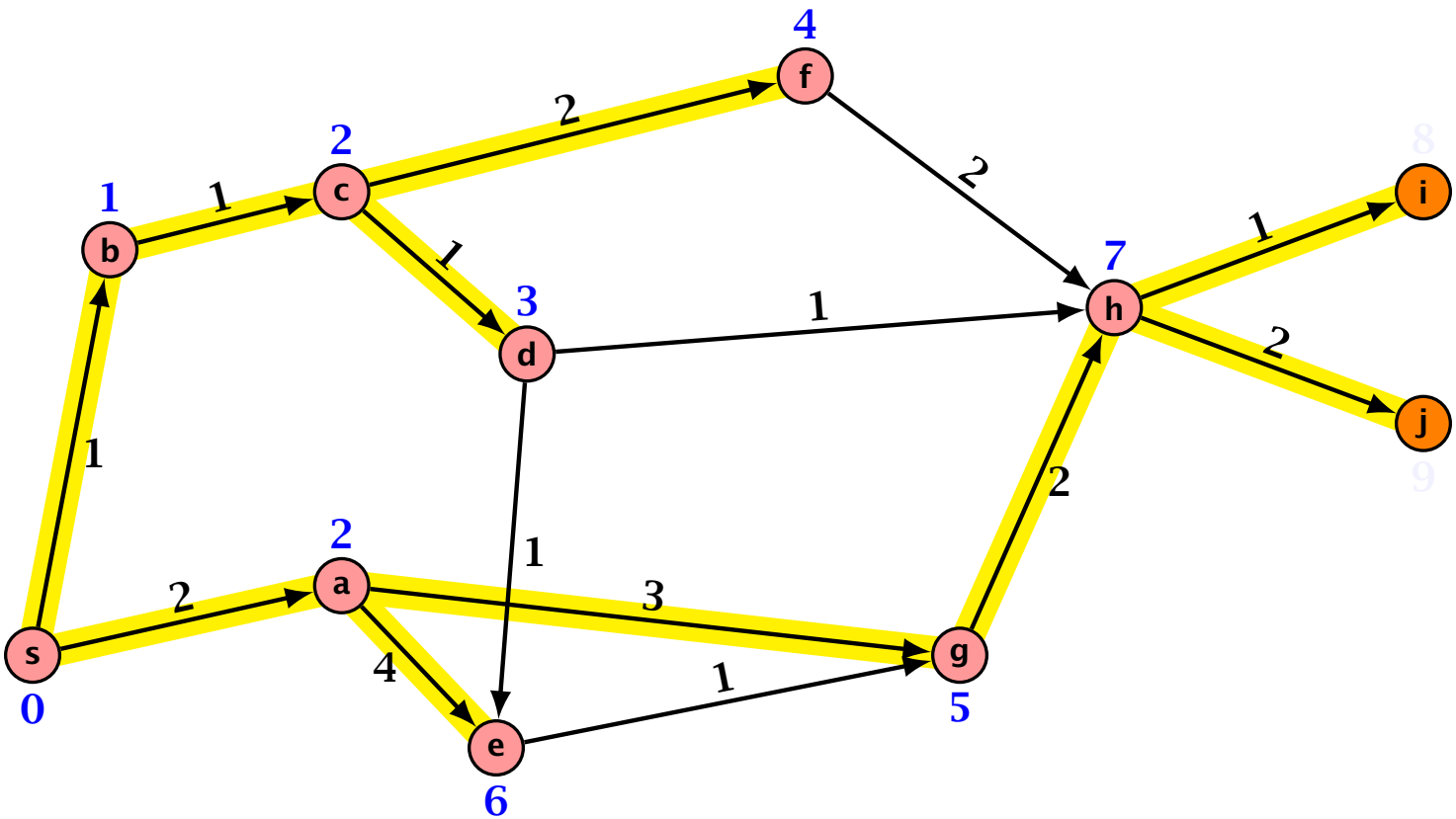
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

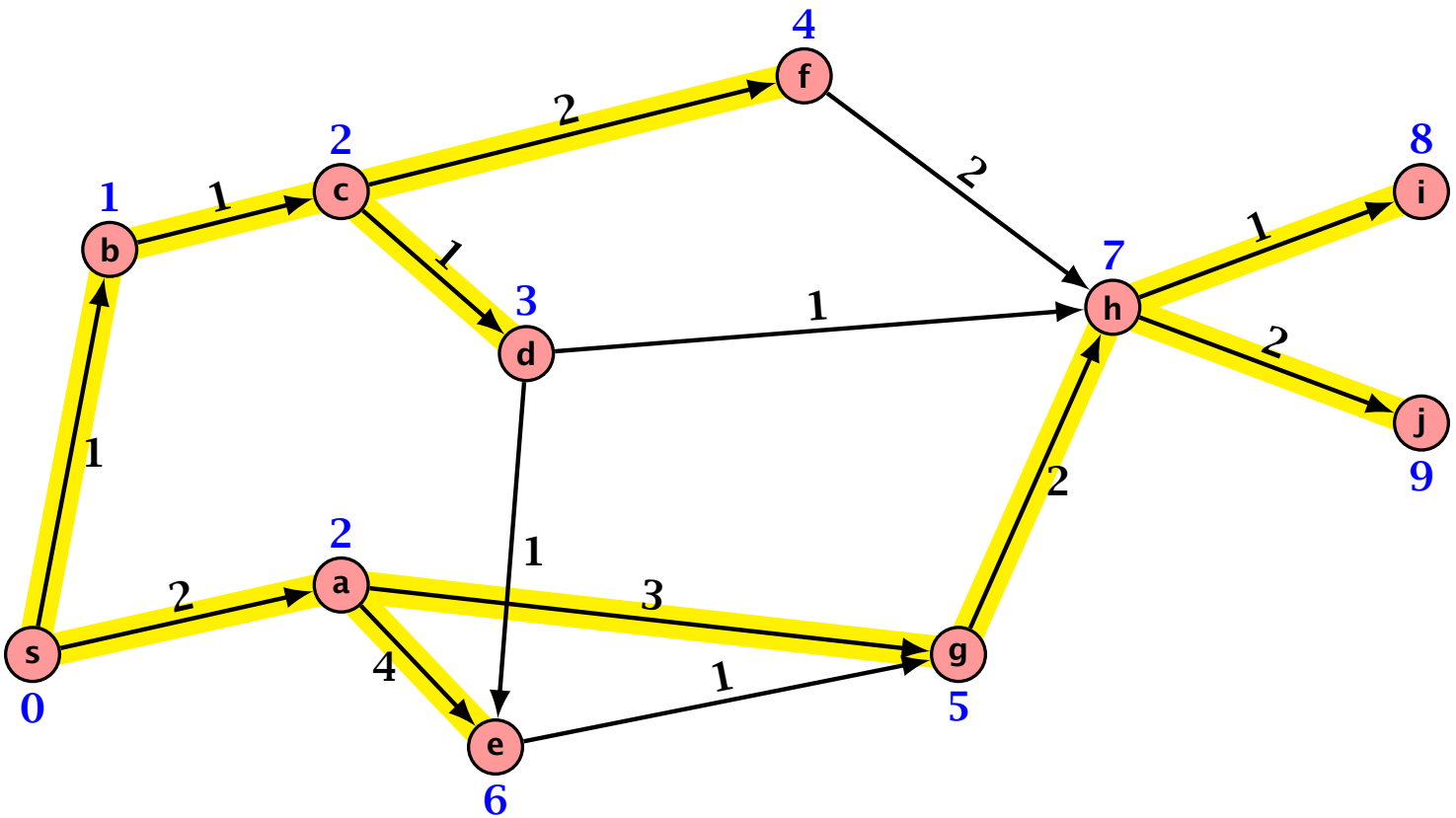
For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths in DAGs

For DAGs with edge-weights a BFS does not work.



Problem: there may be paths with short lengths but many hops.

Shortest Paths on DAGs

On DAGs there exists a topological sort (order of vertices):

- $\text{top} : V \to \{1, \ldots, n\}$, bijective
- for all $(x, y) \in E$: $\text{top}(x) < \text{top}(y)$

For example $\text{top}(v) = n - \text{DFSnum}(v) + 1$.

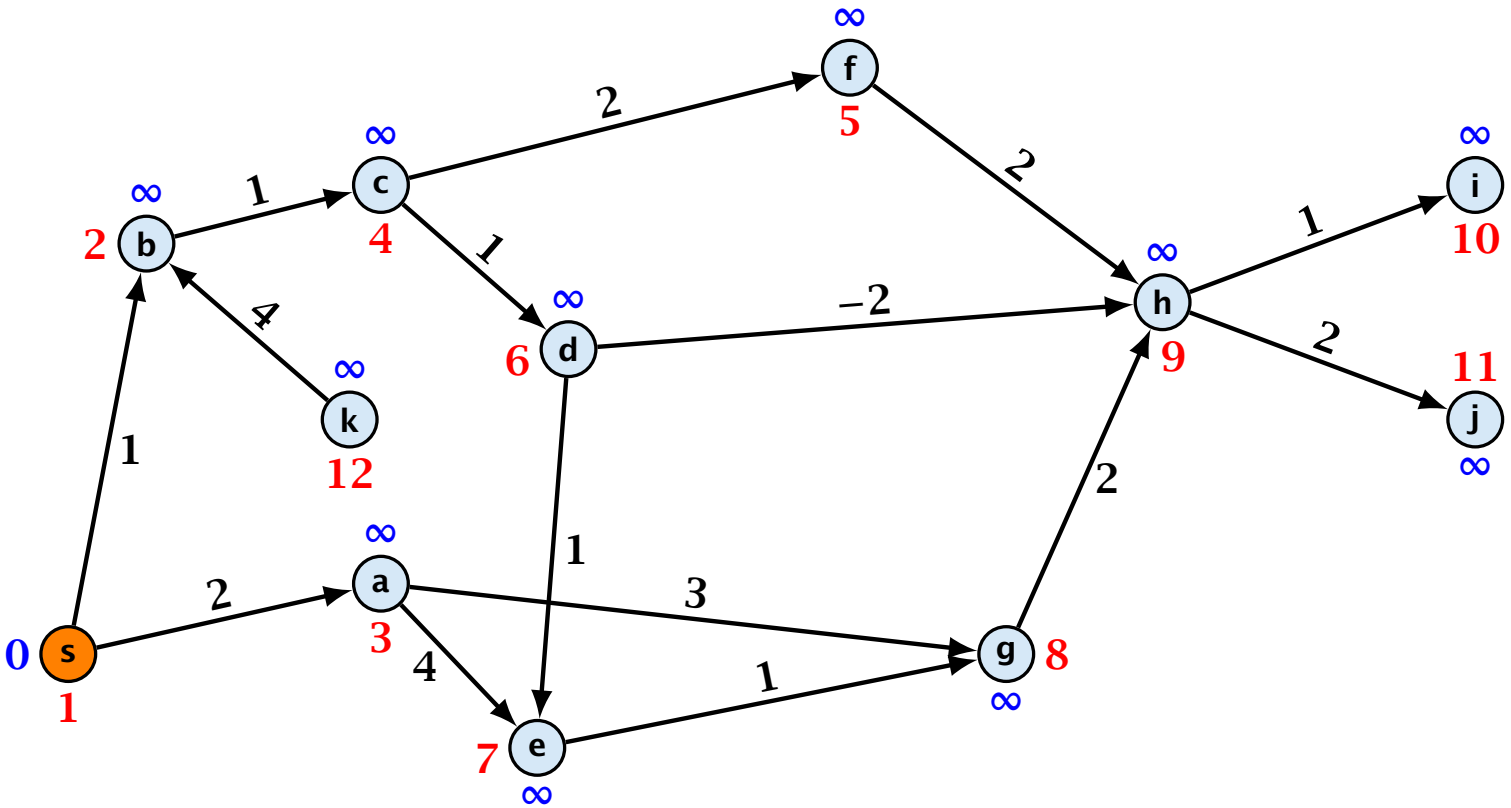
Algorithm 1 ShortestPathDag$(G = (V, E, w), top, s)$

Input: weighted DAG $G = (V, E, w)$; start vertex $s \in V$;
map $top : \{1, \ldots, n\} \to V$ (topological ordering of G)
Output: key-field of every node contains distance from s;
for all $v \in V$ **do** $v.\text{key} \leftarrow \infty$;
$s.\text{key} \leftarrow 0$;
for $i = 1$ **to** n **do**
    $v \leftarrow top[i]$
    for all $x \in V$ s.t. $(v, x) \in E$ **do**
        $x.\text{key} \leftarrow \min\{v.\text{key} + w(v, x), x.\text{key}\}$;

Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

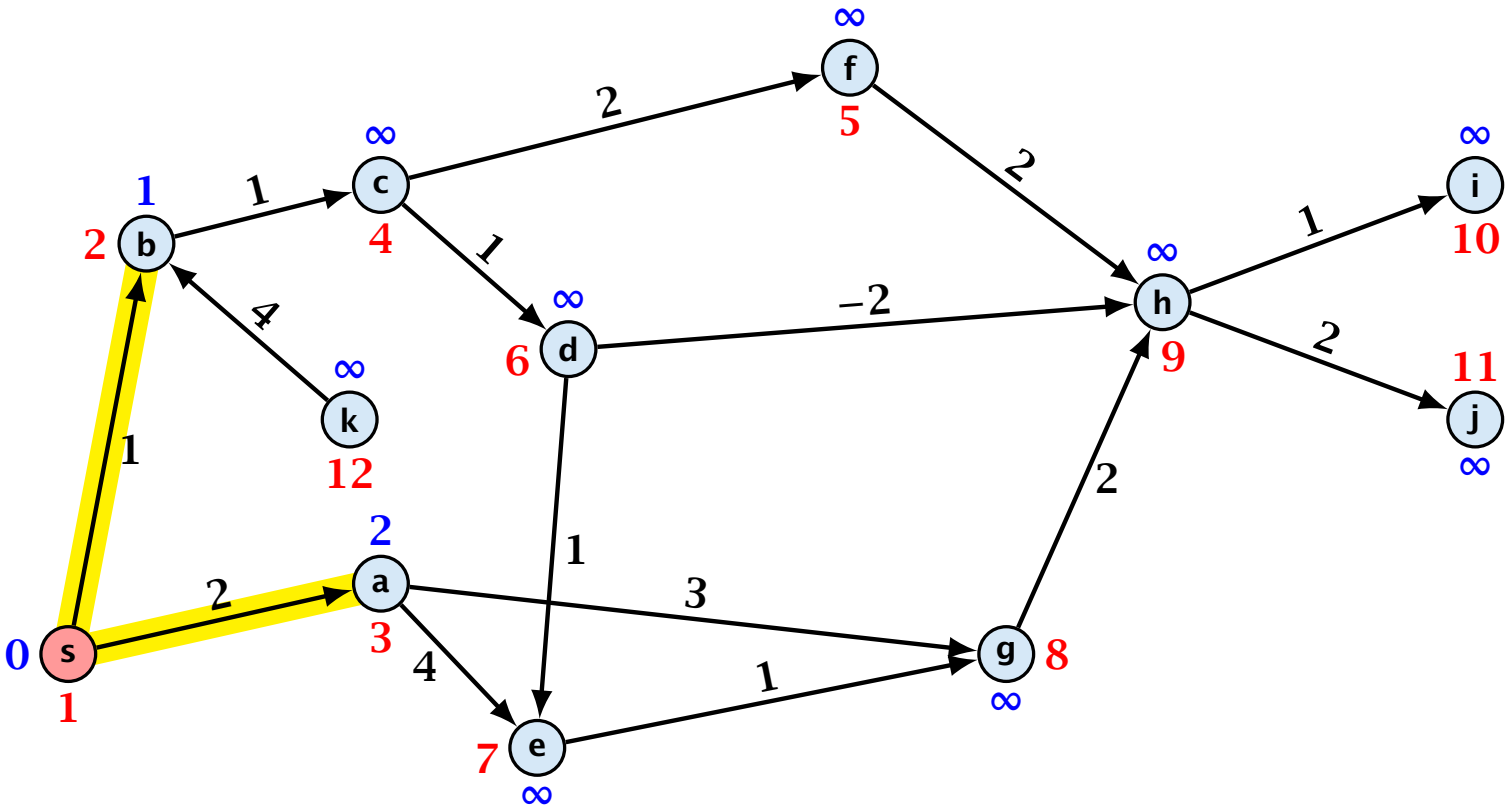
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

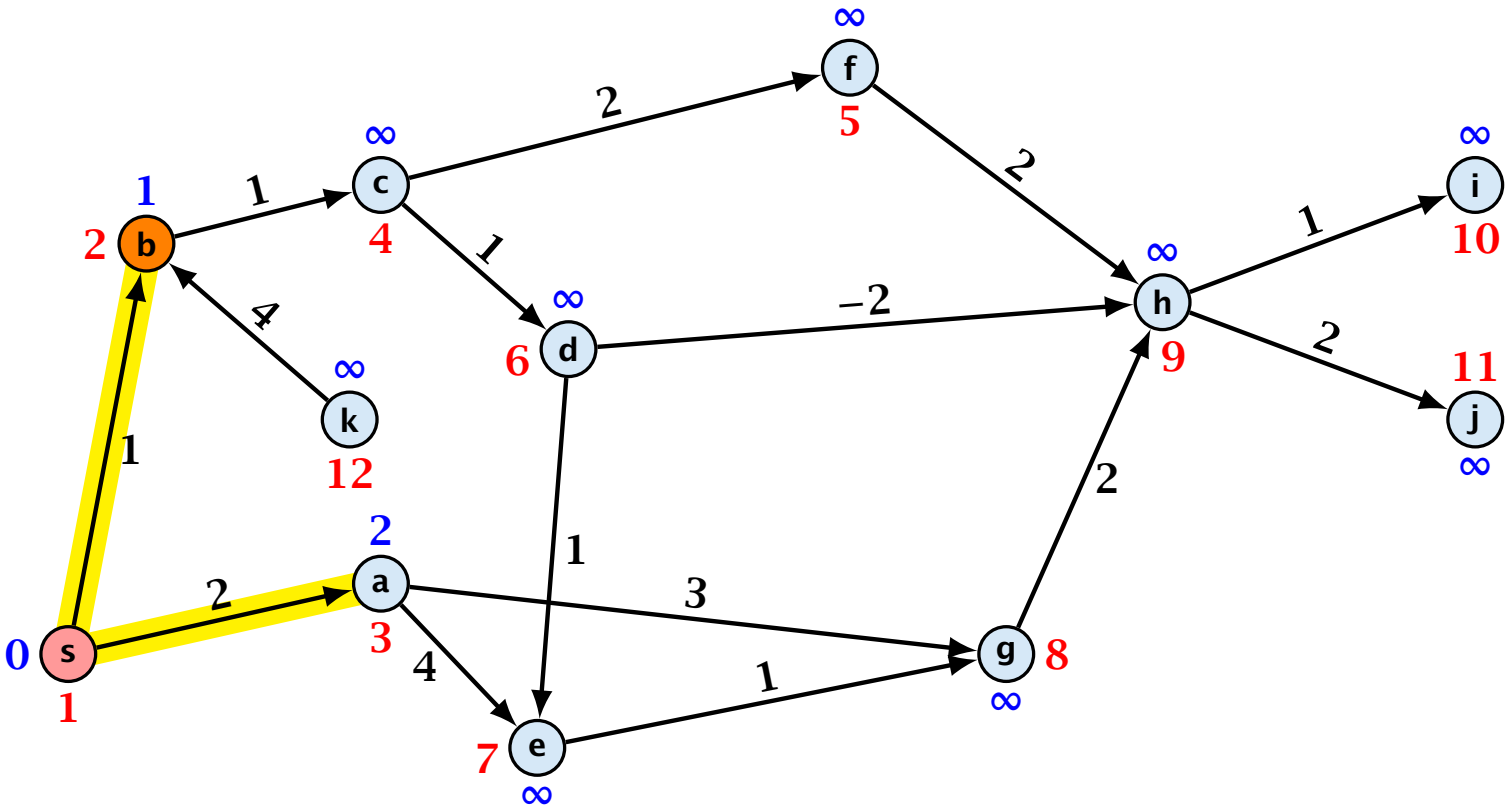
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

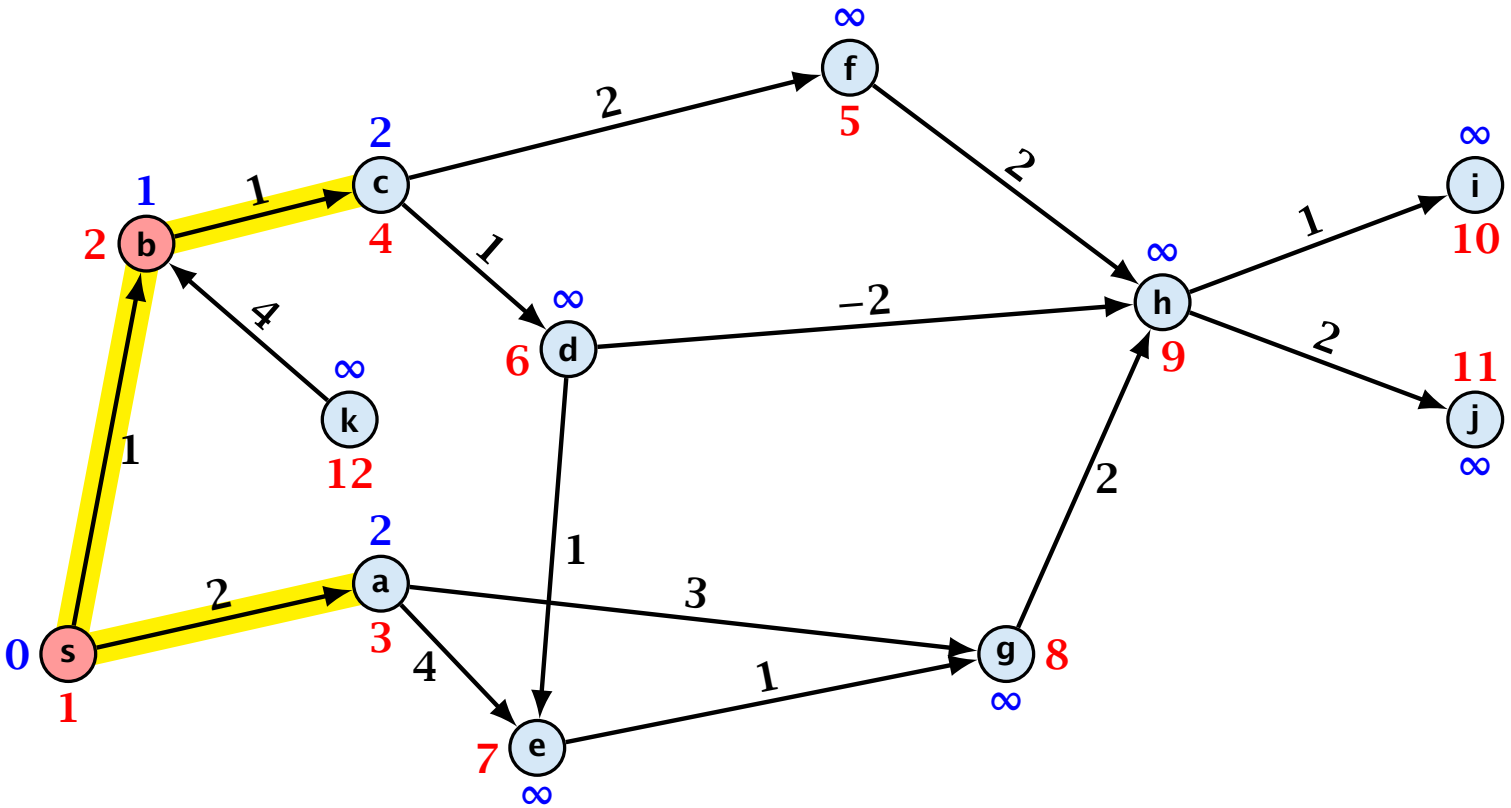
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

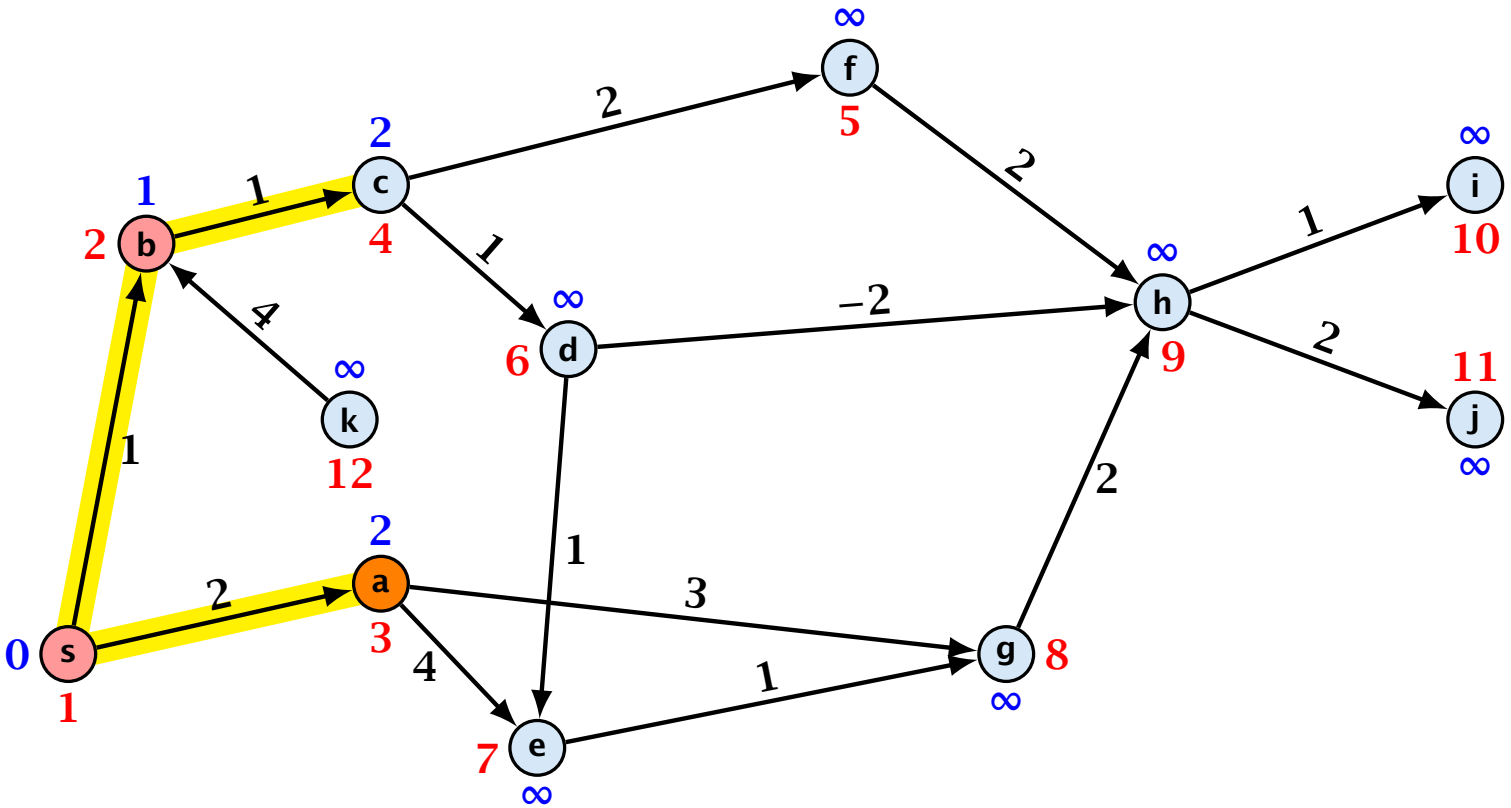
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

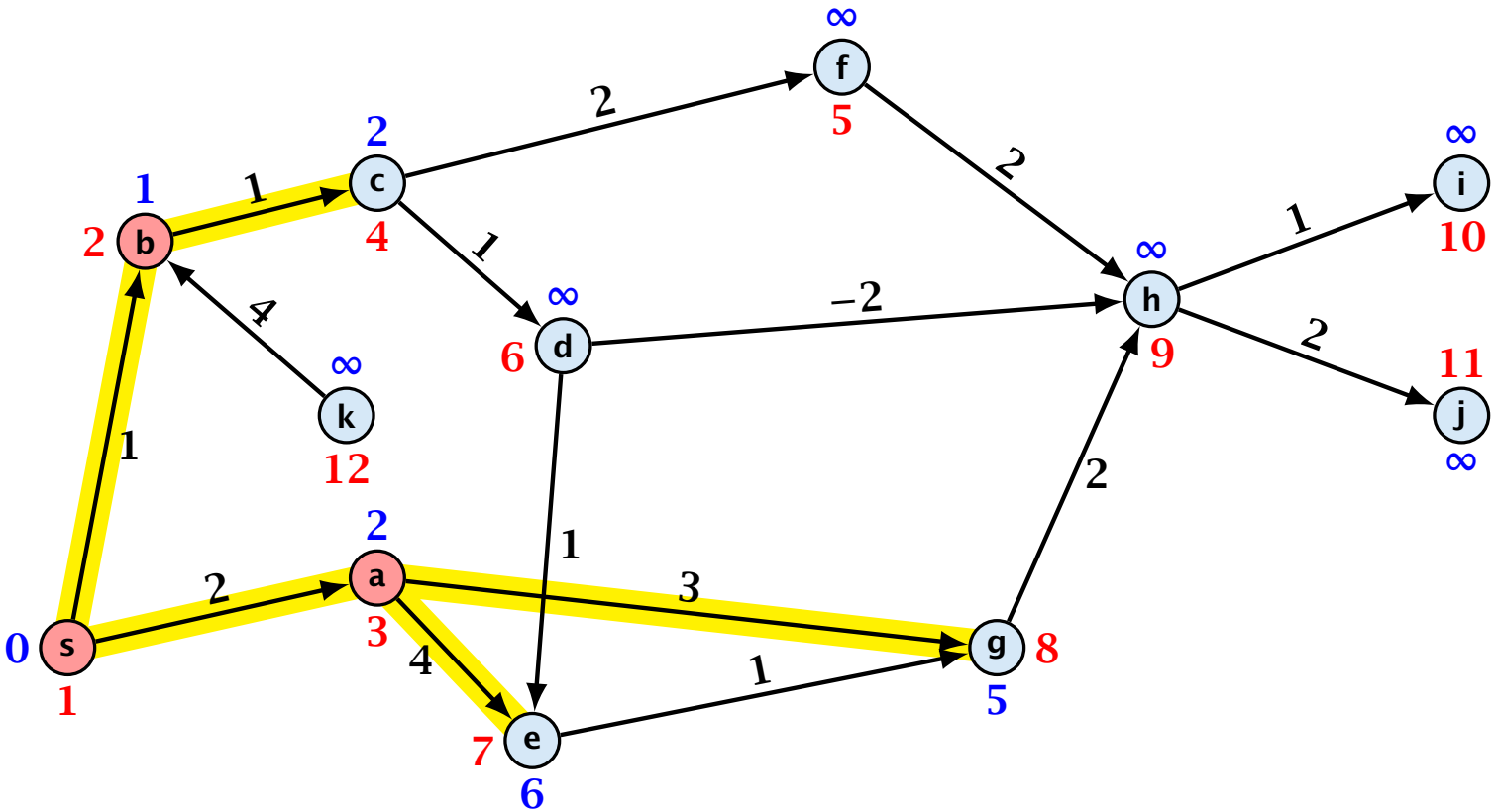
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

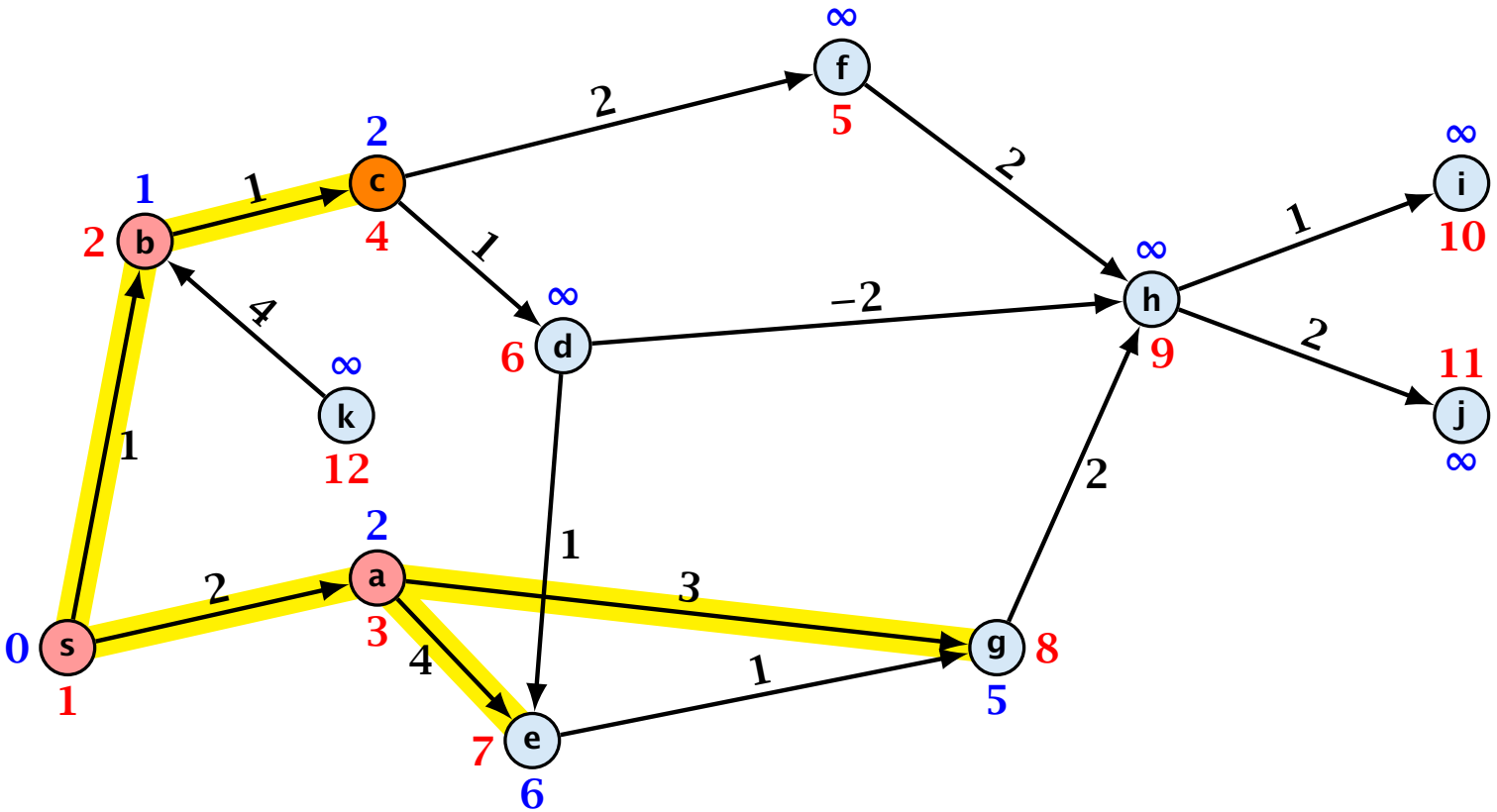
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

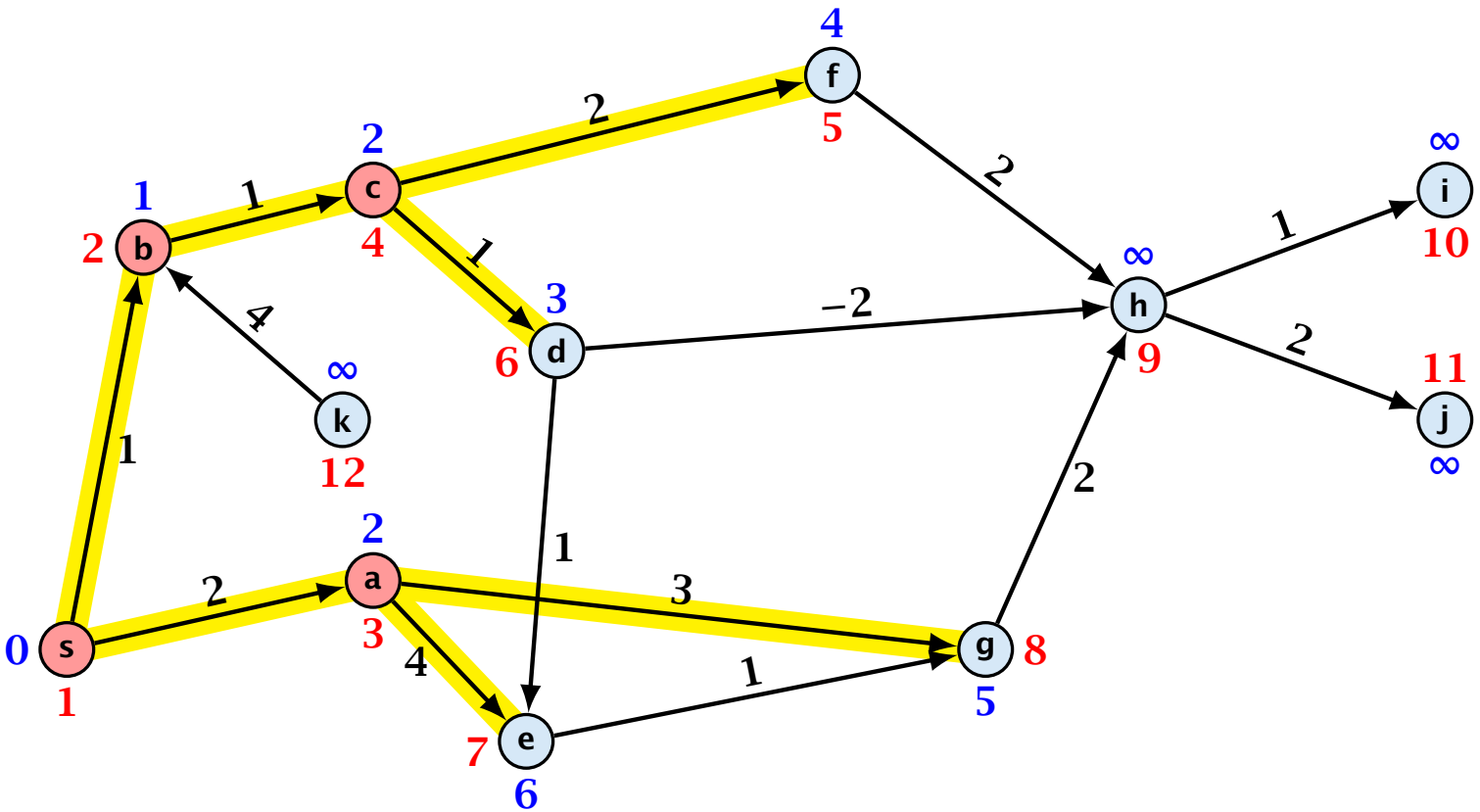
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

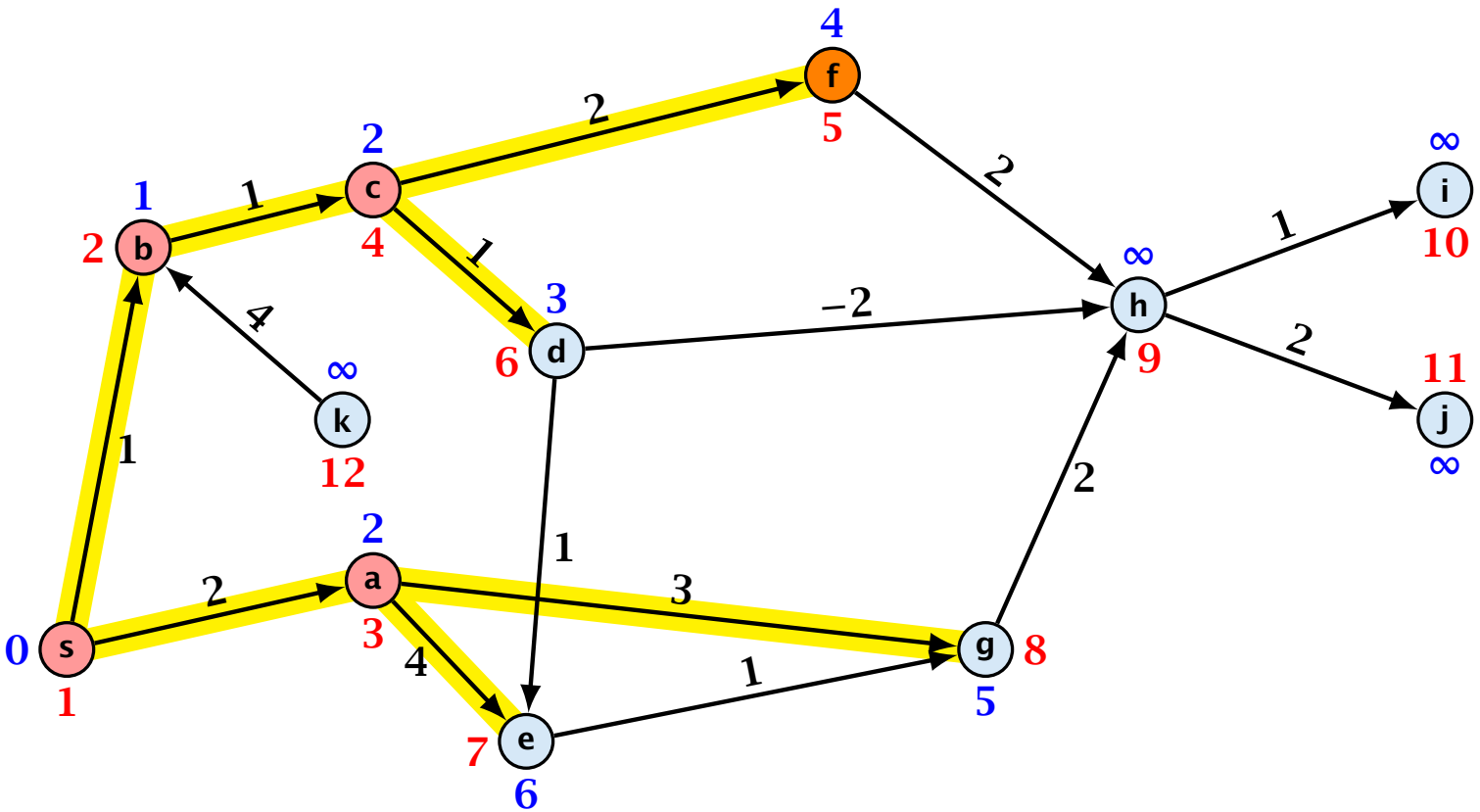
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

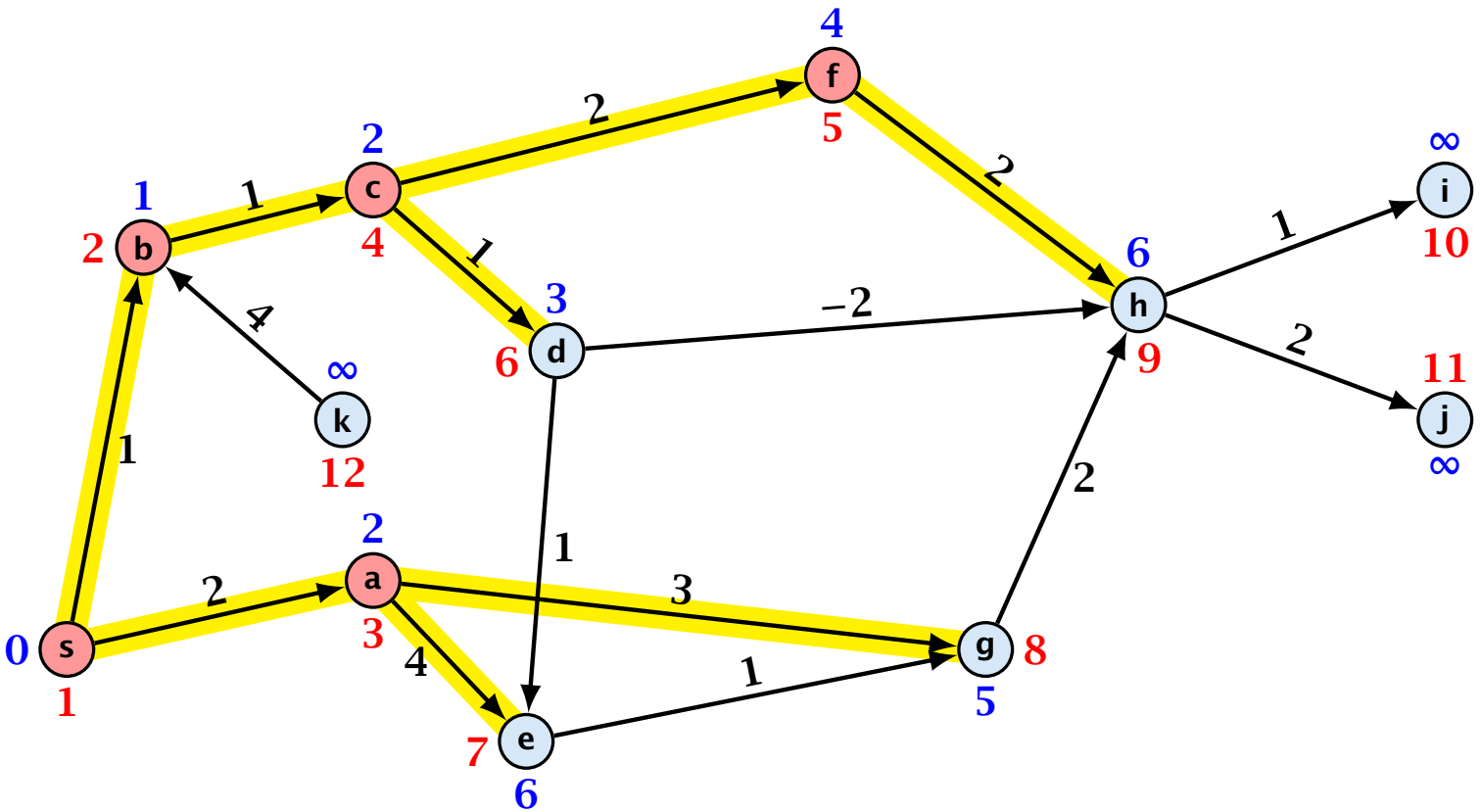
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

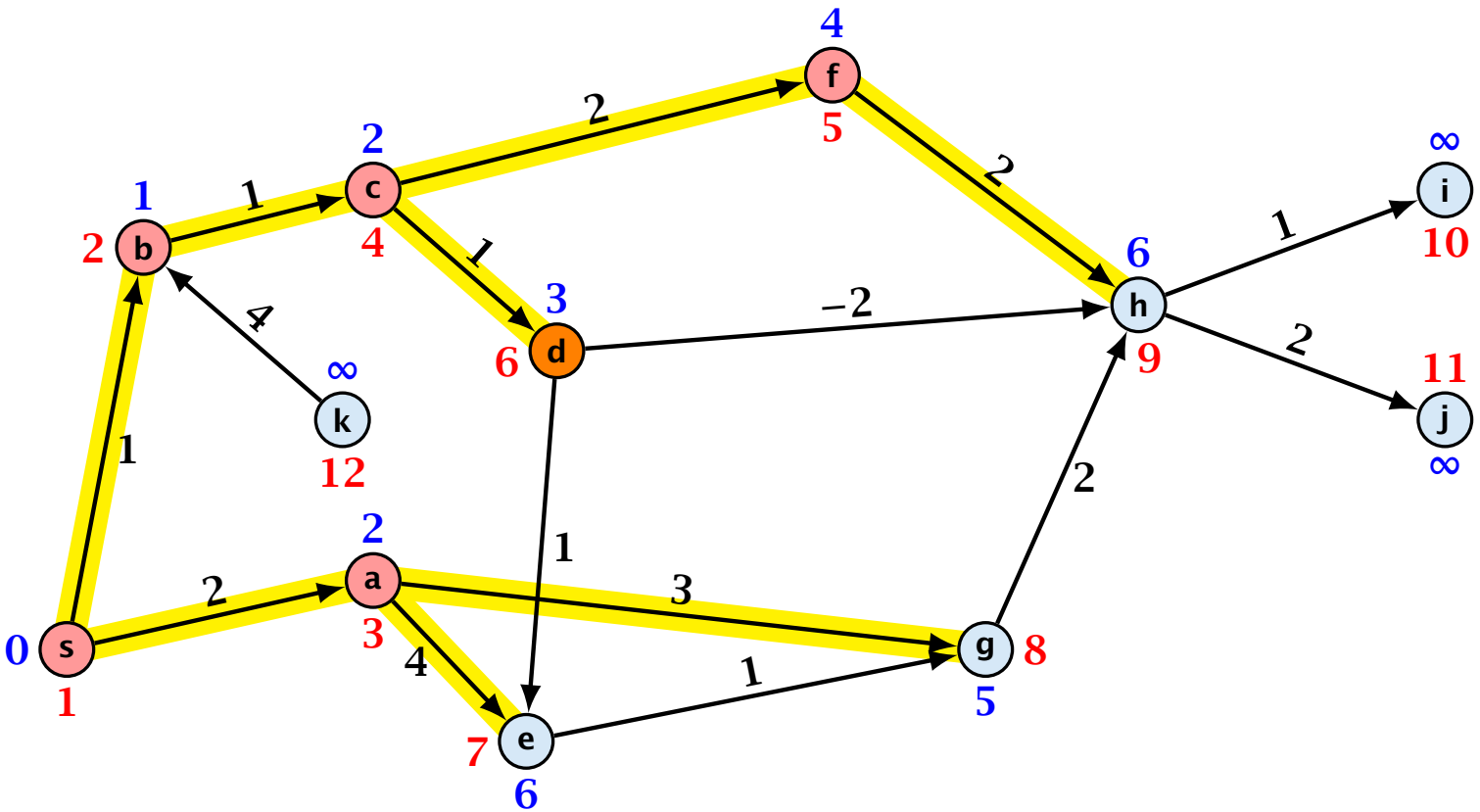
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

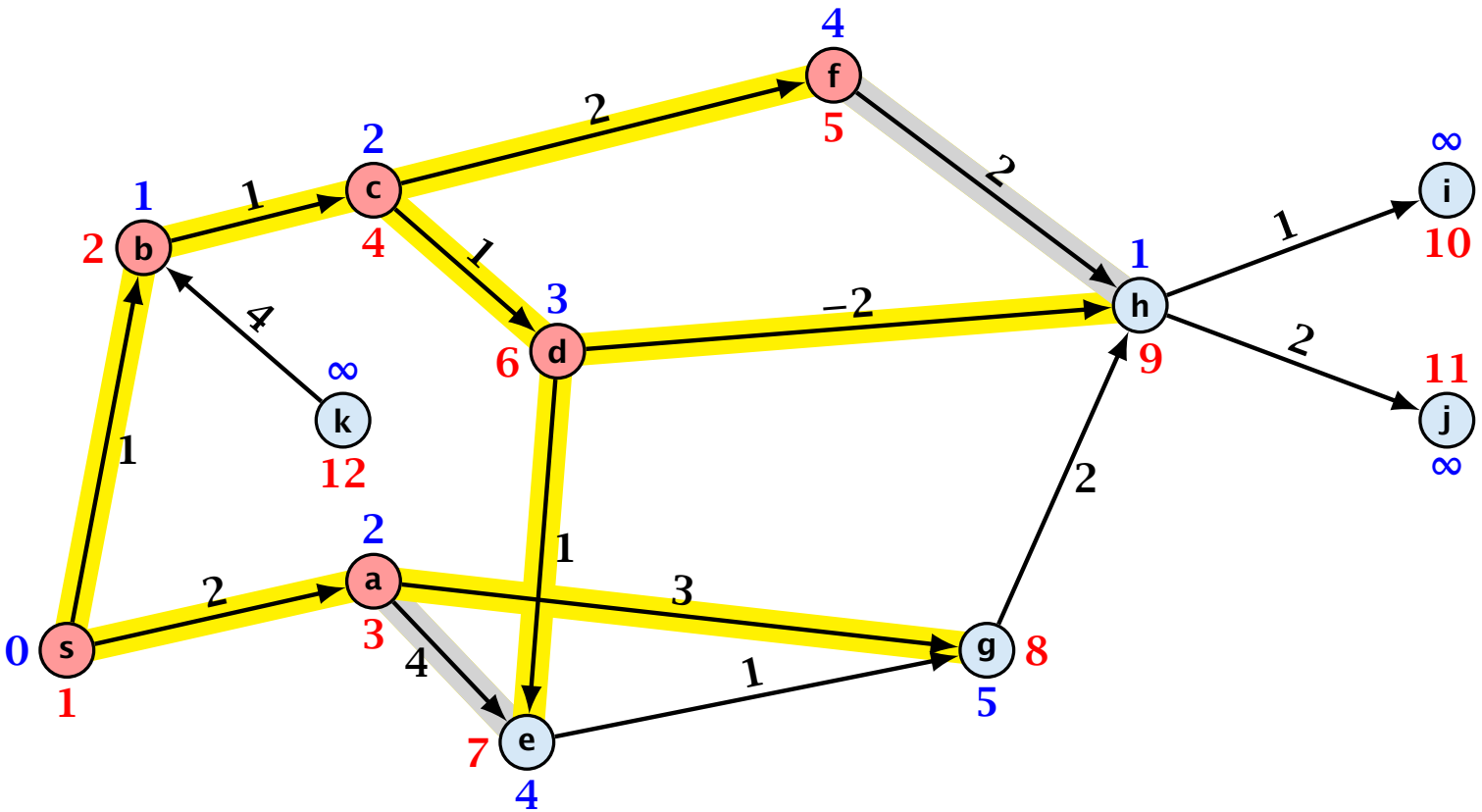
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

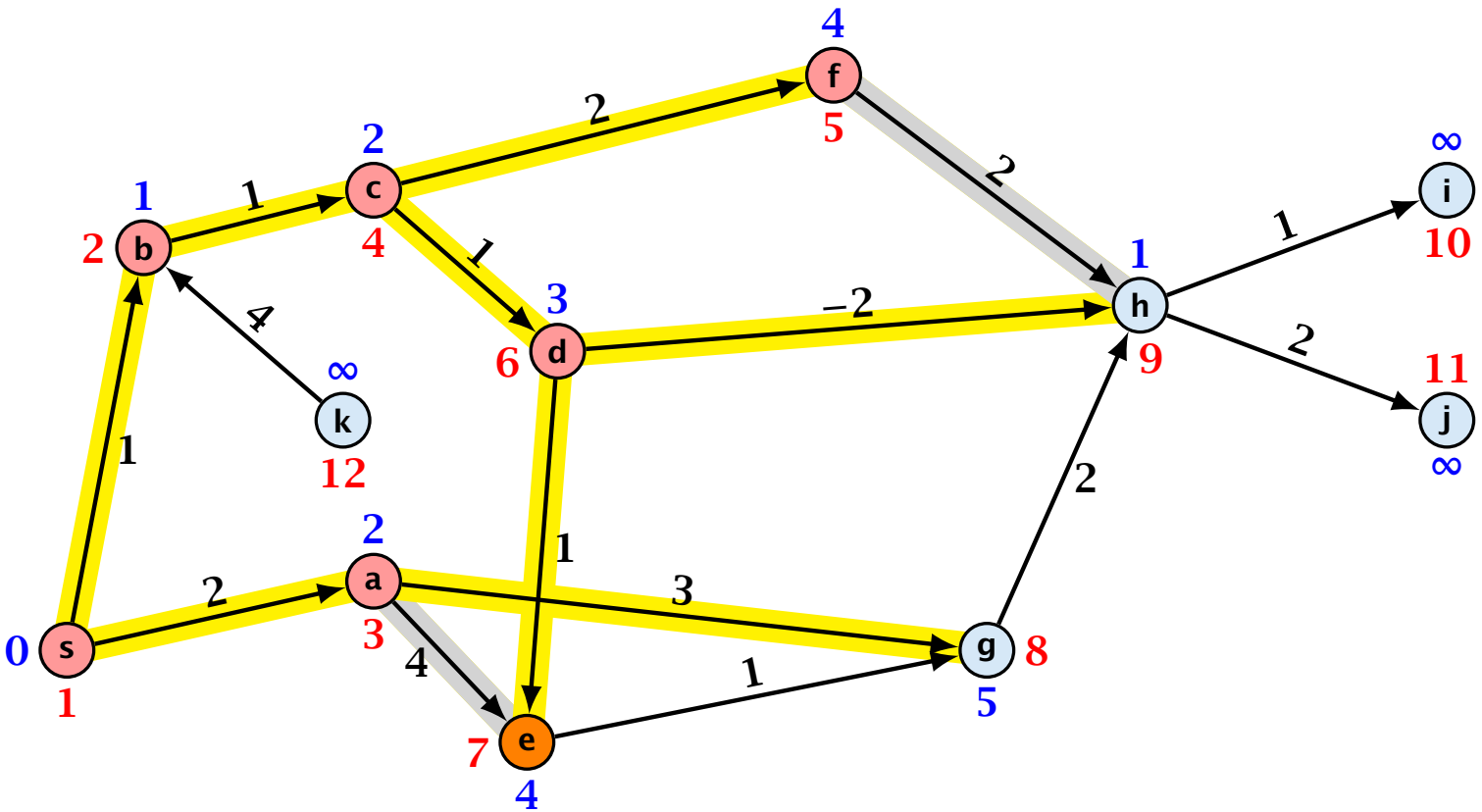
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

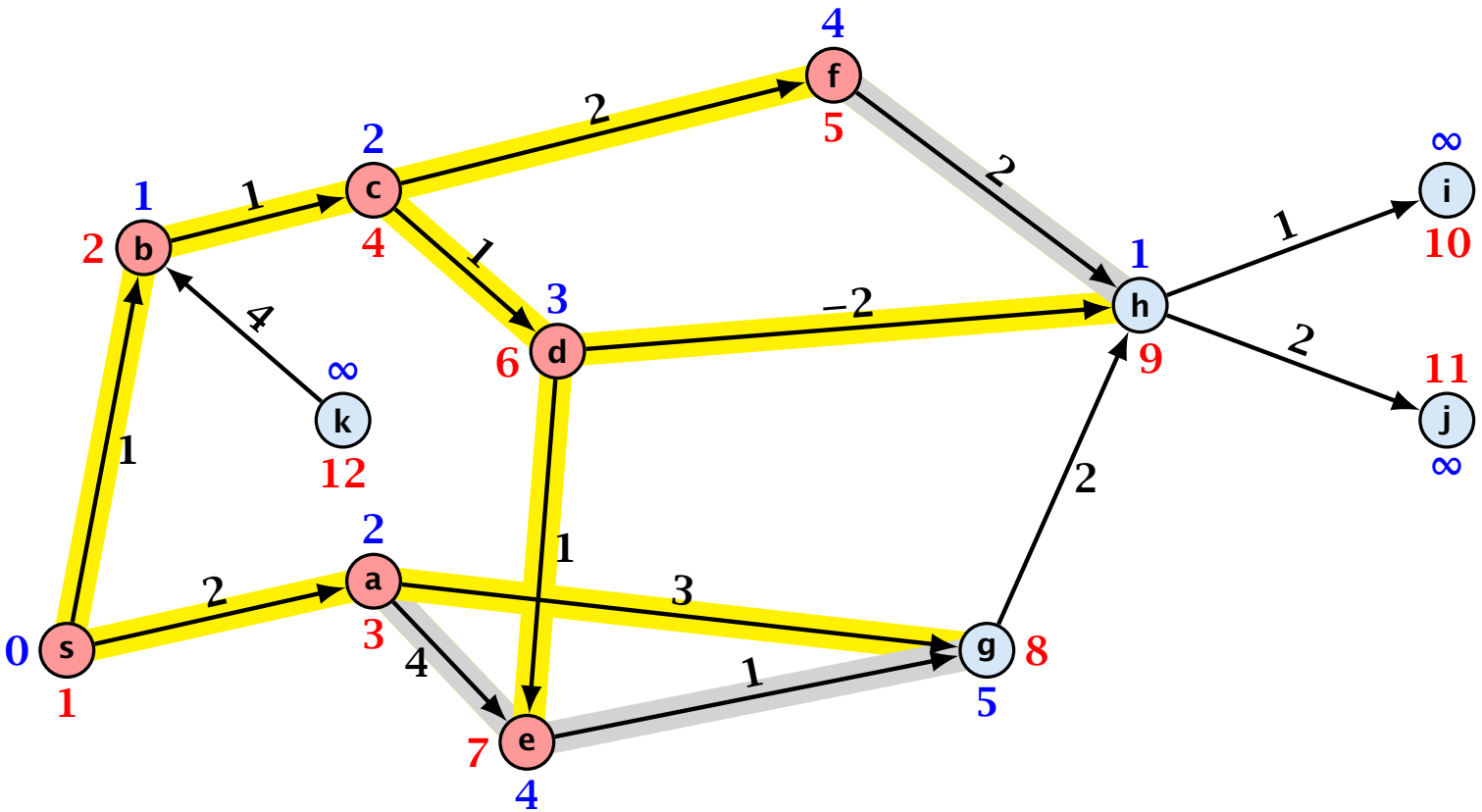
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

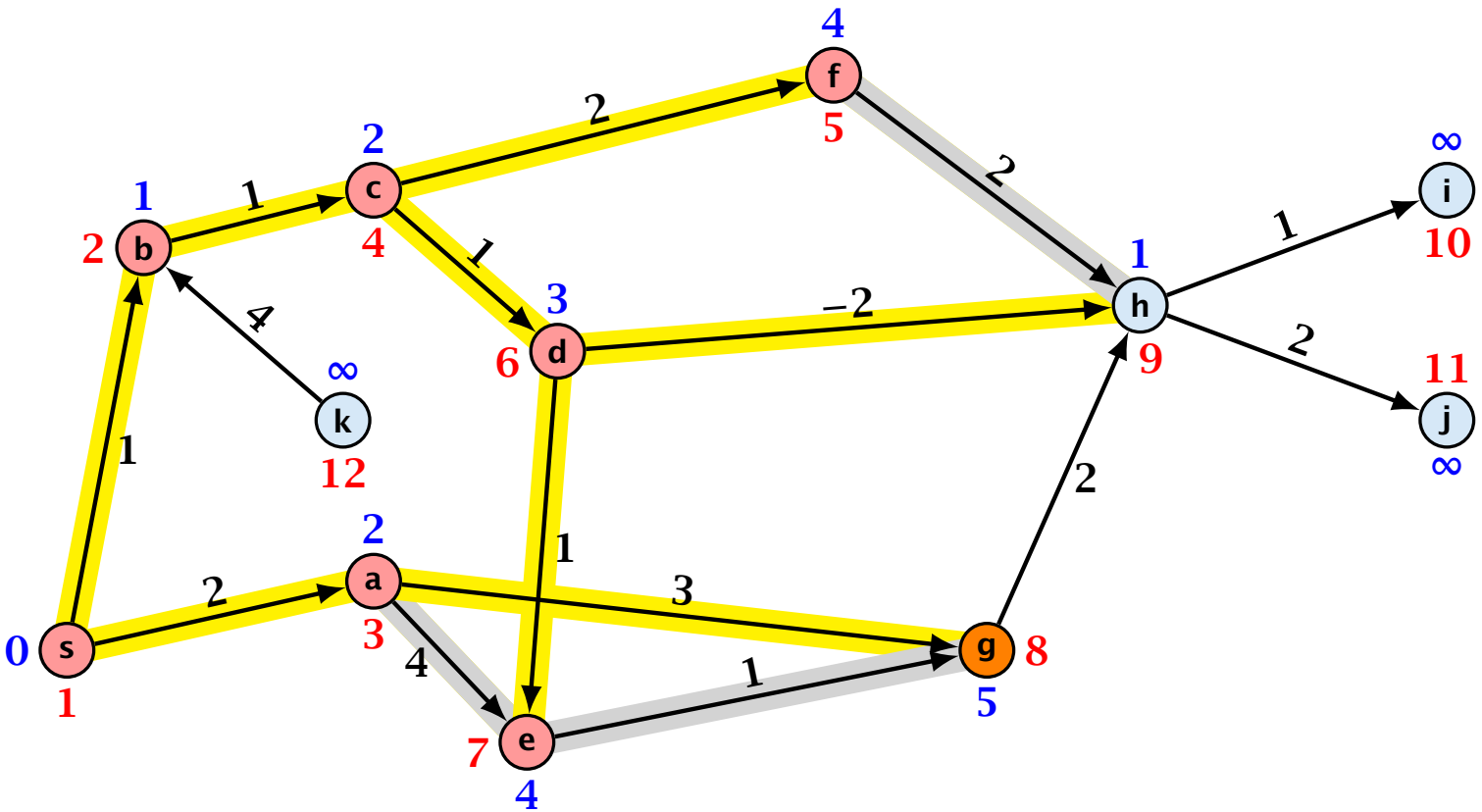
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

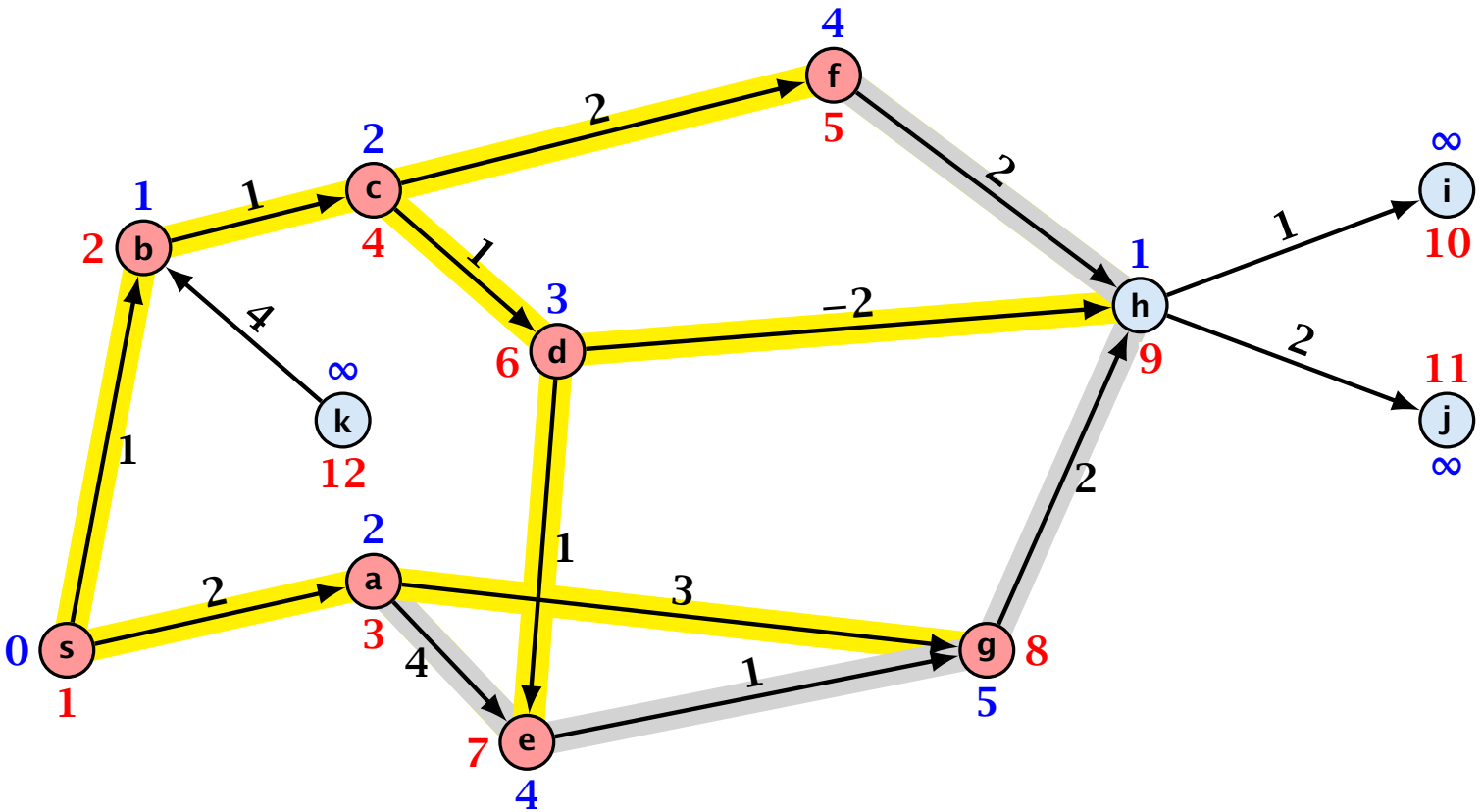
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

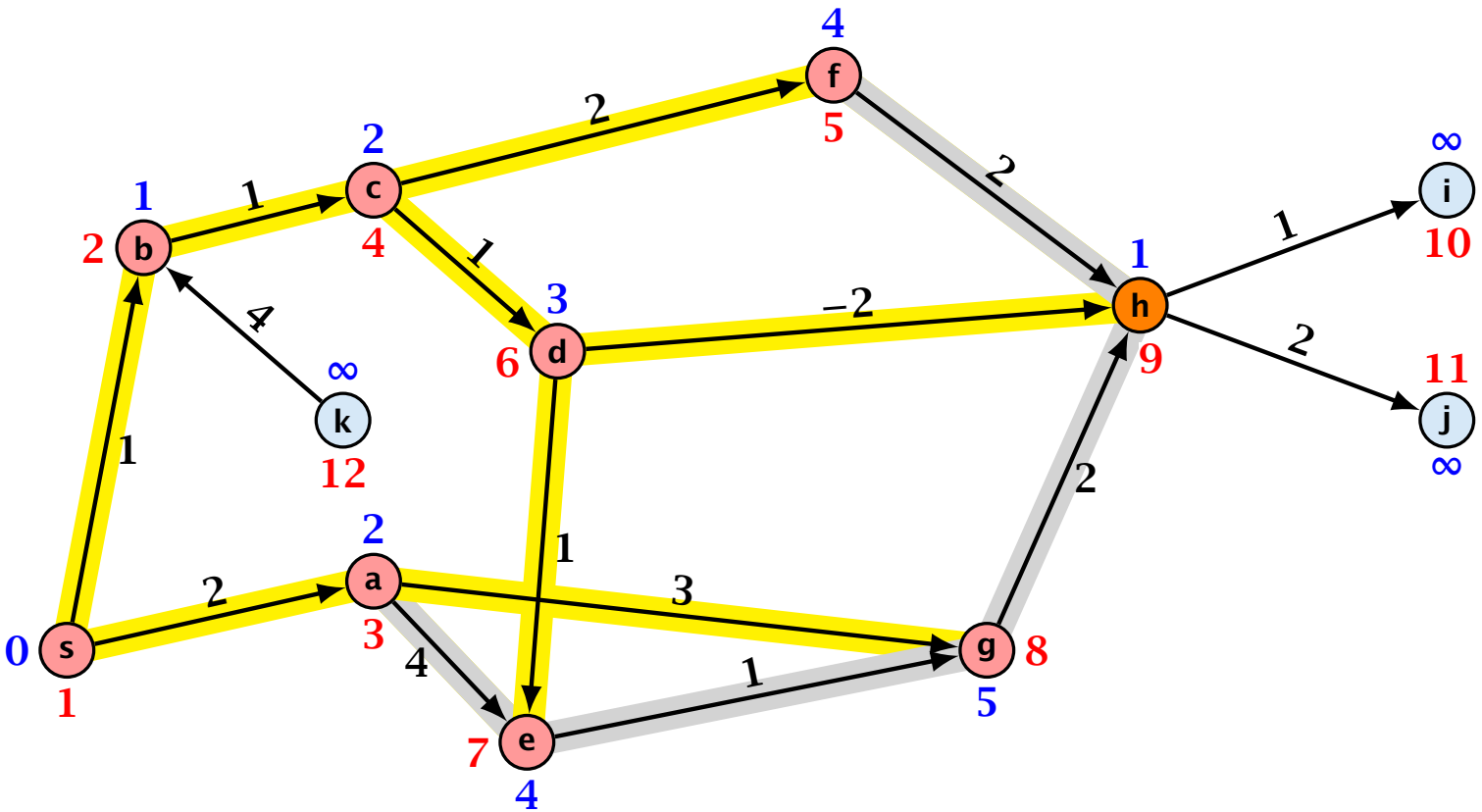
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

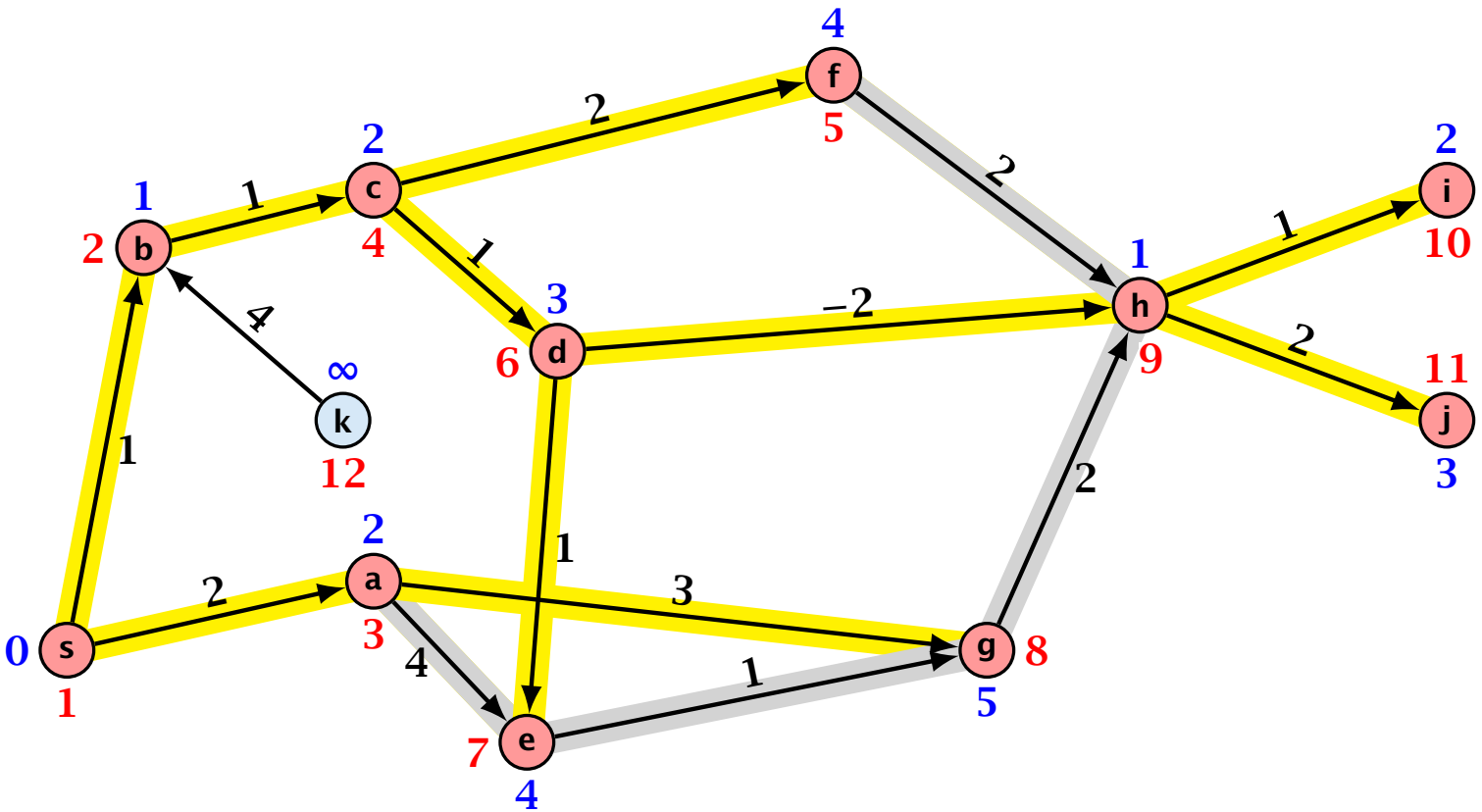
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

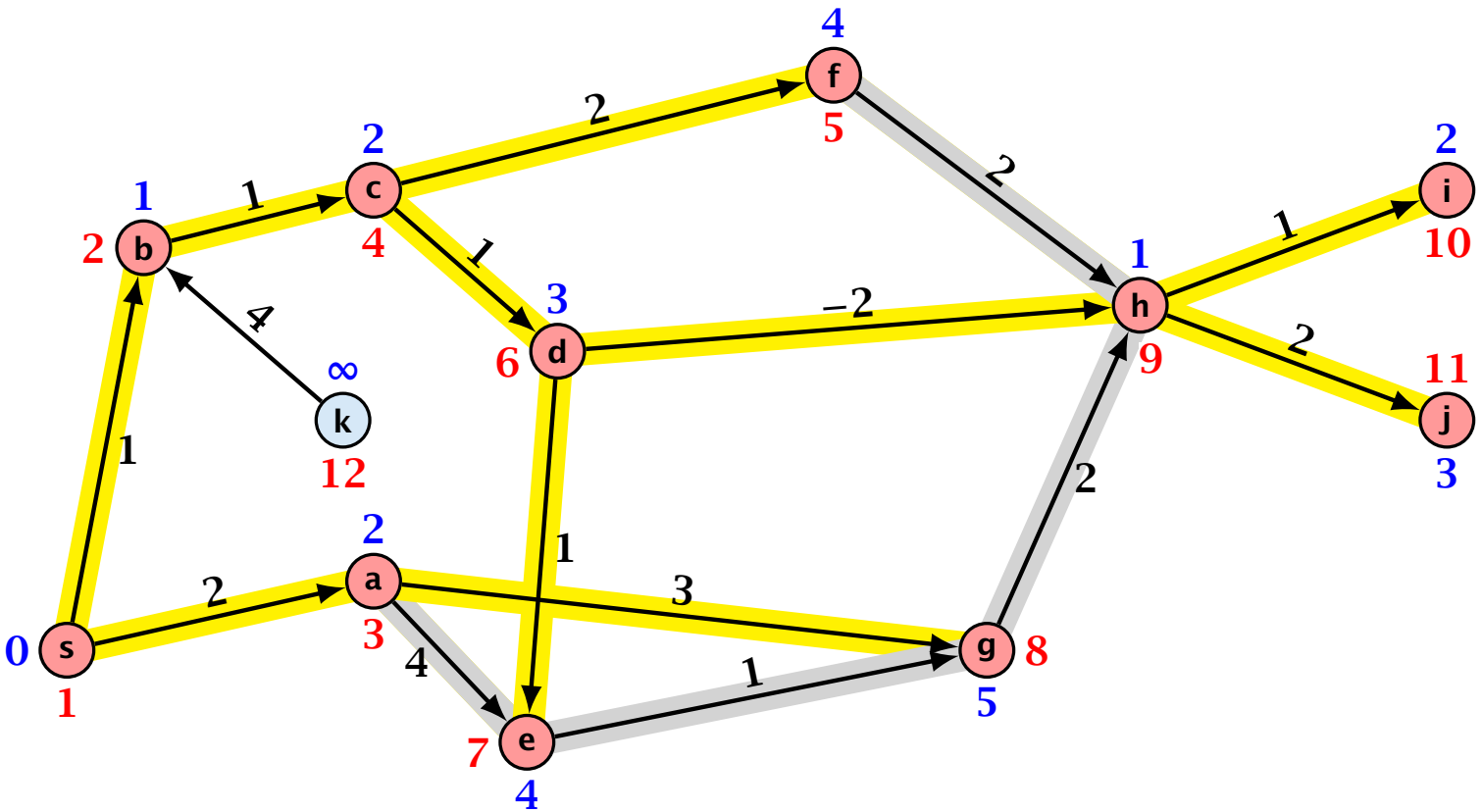
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

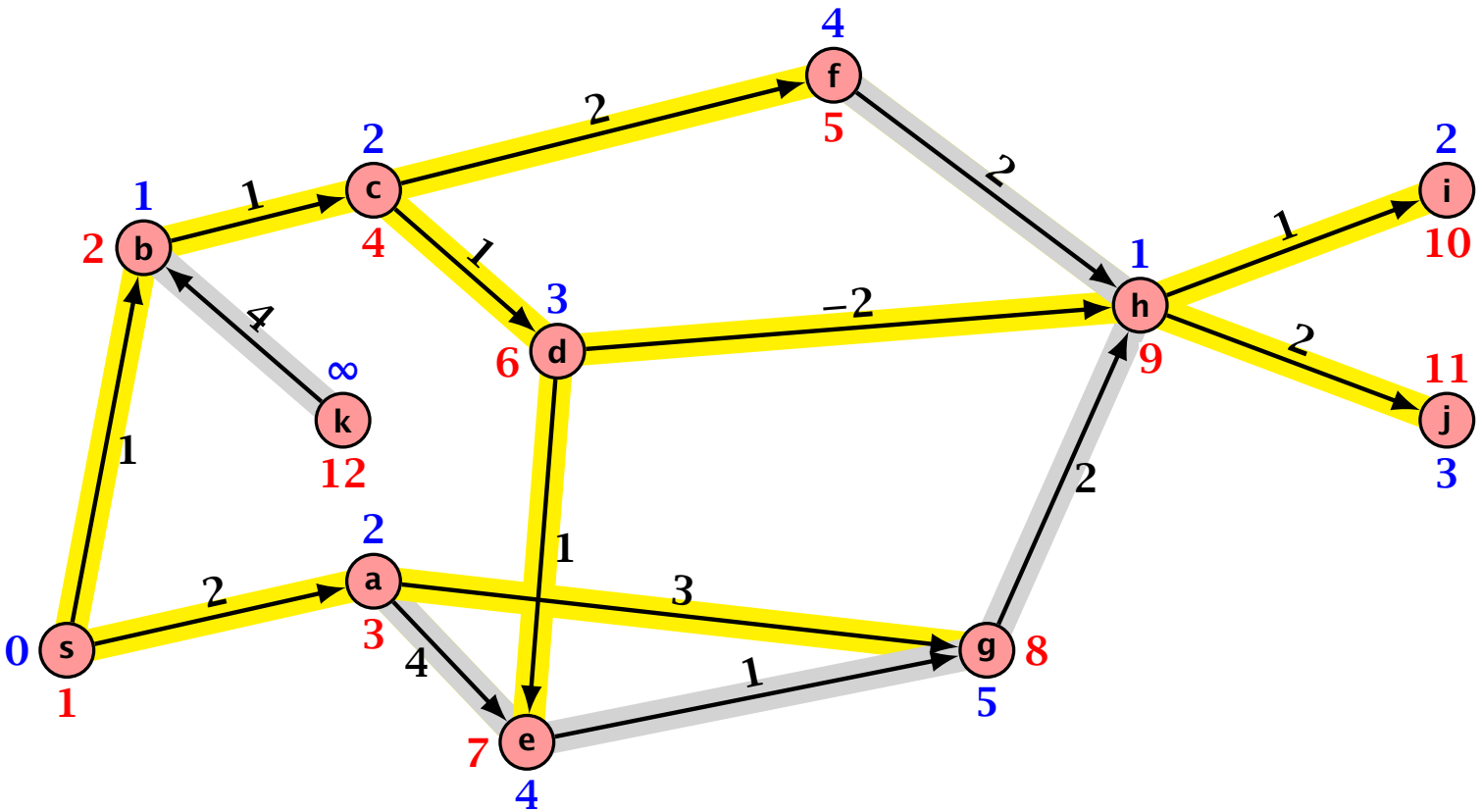
$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Visit nodes in order of topological sort. When visiting x perform relaxation of outgoing edges (x, y):

$$\text{dist}(y) := \min\{\text{dist}(x) + w(x, y), \text{dist}(y)\}$$



Shortest Paths in DAGs

Why does this work?

A. the distance-labels are upper bounds on real distances

- initially, $\mathrm{dist}(s) = 0$ and $\mathrm{dist}(v) = \infty$ for $v \neq s$; hence invariant holds
- a relaxation ($\mathrm{dist}(y) = \mathrm{dist}(x) + w(x,y)$) cannot violate invariant

Shortest Paths in DAGs

Why does this work?

A. the distance-labels are upper bounds on real distances

- initially, $\mathrm{dist}(s) = 0$ and $\mathrm{dist}(v) = \infty$ for $v \neq s$; hence invariant holds
- a relaxation ($\mathrm{dist}(y) = \mathrm{dist}(x) + w(x,y)$) cannot violate invariant

B. in the end $\mathrm{dist}(y) \leq w(p)$ for any s-y path p

- relaxations along p occur in order of the path;
- by induction over the number of relaxations after the $i-1$-st relaxation (along p) the i-th node on the path has a distance label $\mathrm{dist}(v_i) \leq w(p_i)$, where p_i is sub-path of p containing just the first i nodes

Shortest Paths in DAGs

Running Time:

We perform a DFS for topological sorting:

- time $\mathcal{O}(n+m)$

Afterwards the algorithm visits each edge exactly once. In addition each vertex is scanned:

- time $\mathcal{O}(n+m)$

Hence, total time is $\mathcal{O}(n+m)$.

DAG with Self-Loops

suppose that we allow self-loops at the vertices of an otherwise **D**irected **A**cyclic **G**raph

only self-loops of negative length can influence shortest path distances

we only need to know for every vertex if it has a negative-length self-loop (actual length is not important)

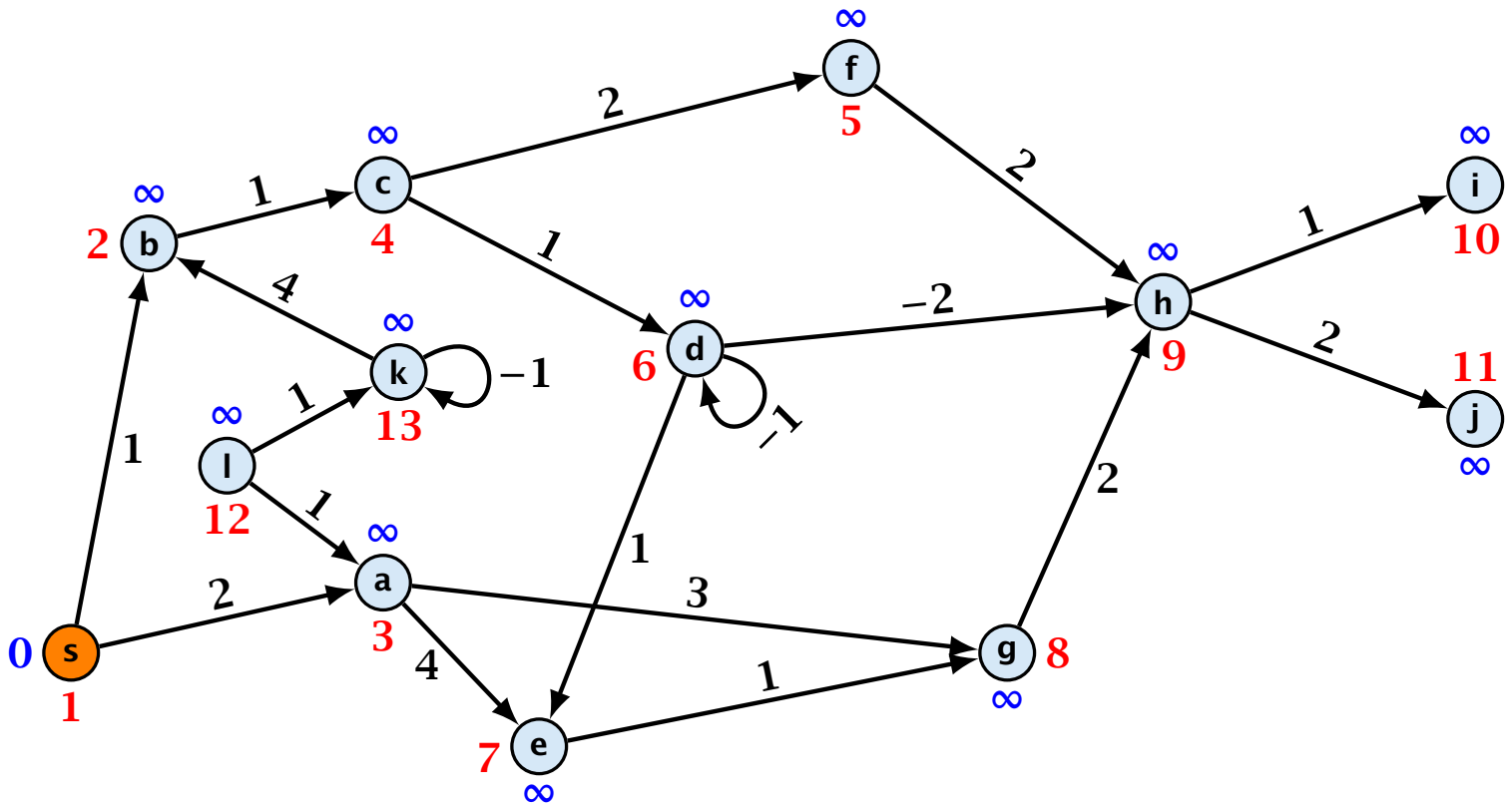
DAG with Self-Loops

Algorithm 2 SP-Dag-Selfloops$(G = (V, E, w), top, s)$

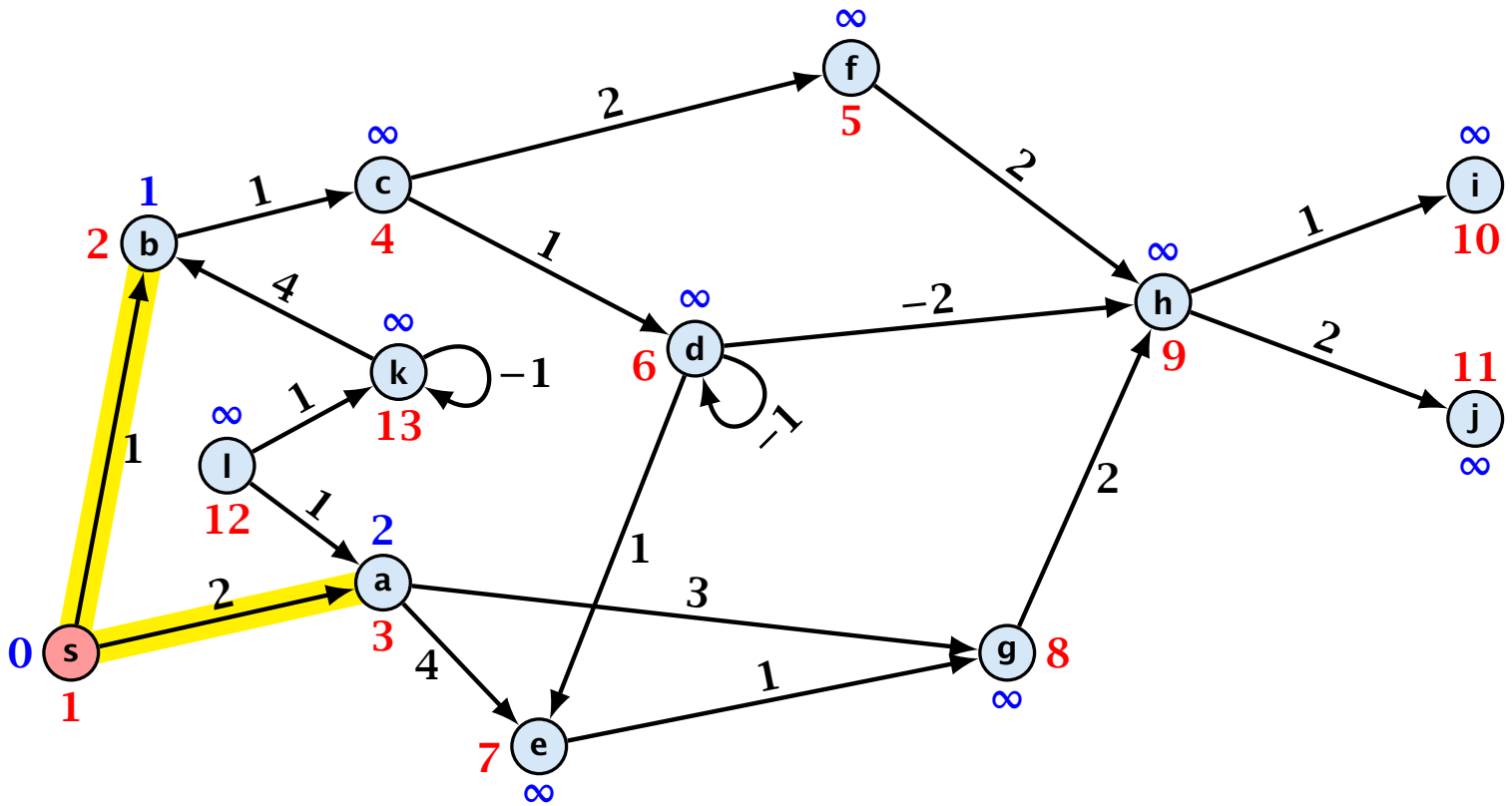
```
 1: Input: weighted DAG G = (V, E, w); start vertex s ∈ V;
       map top : {1, ..., n} → V (topological ordering of G)
       boolean x.self indicates negative self-loop at x
 2: Output: key-field of every node contains distance from s;
 3: for all v ∈ V do v.key ← ∞;
 4: if s.self then s.key = −∞ else s.key ← 0;
 5: for i = 1 to n do
 6:       v ← top[i]
 7:       if v.key = +∞ then next;
 8:       for all x ∈ V s.t. (v, x) ∈ E do
 9:             if x.self then x.key = −∞
10:             x.key ← min{v.key + w(v, x), x.key};
```

Only relax edges whose tail has label $\neq \infty$. If you reach a vertex with self-loop set its label to $-\infty$.

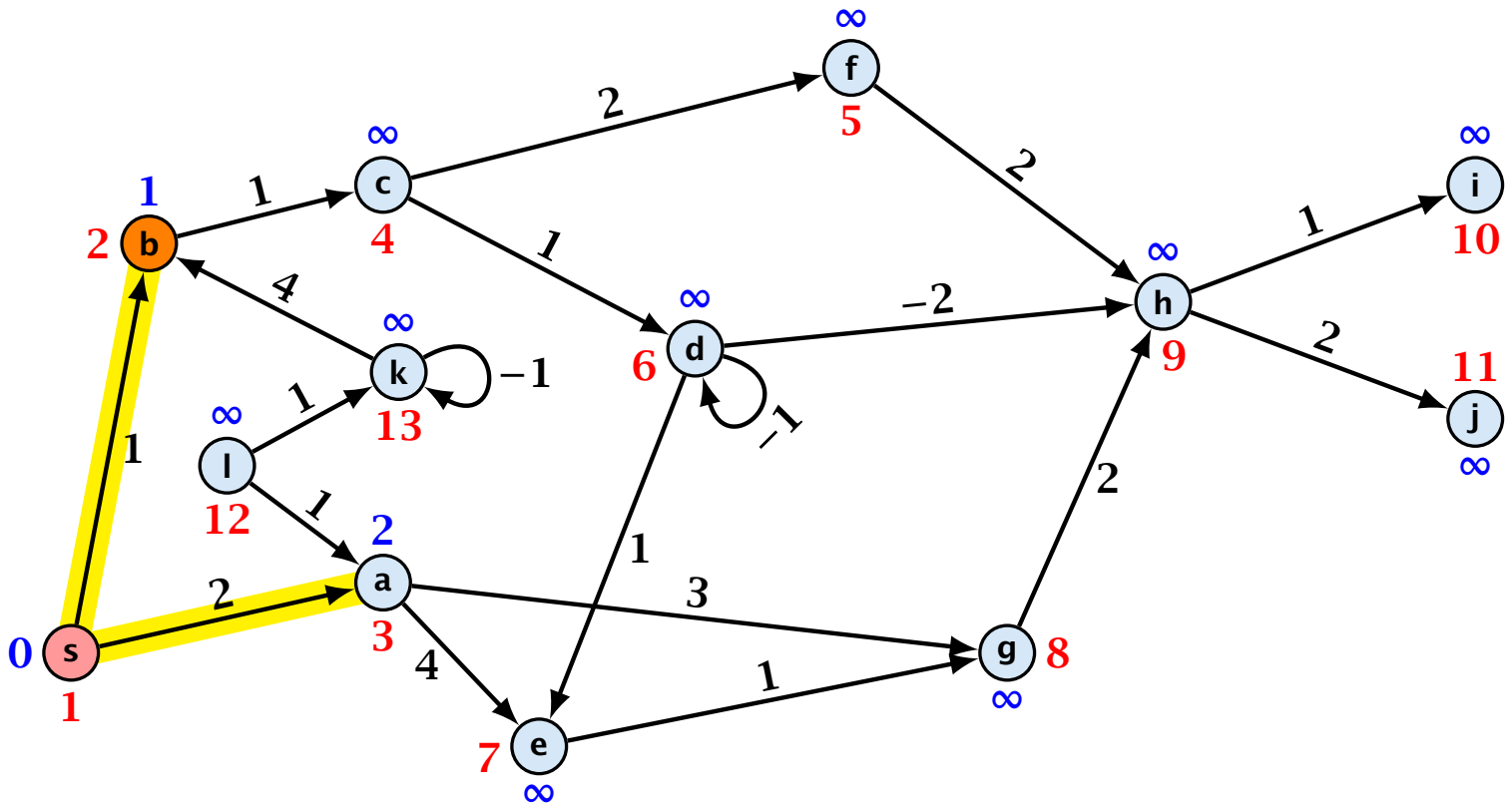
DAG with Self-Loops



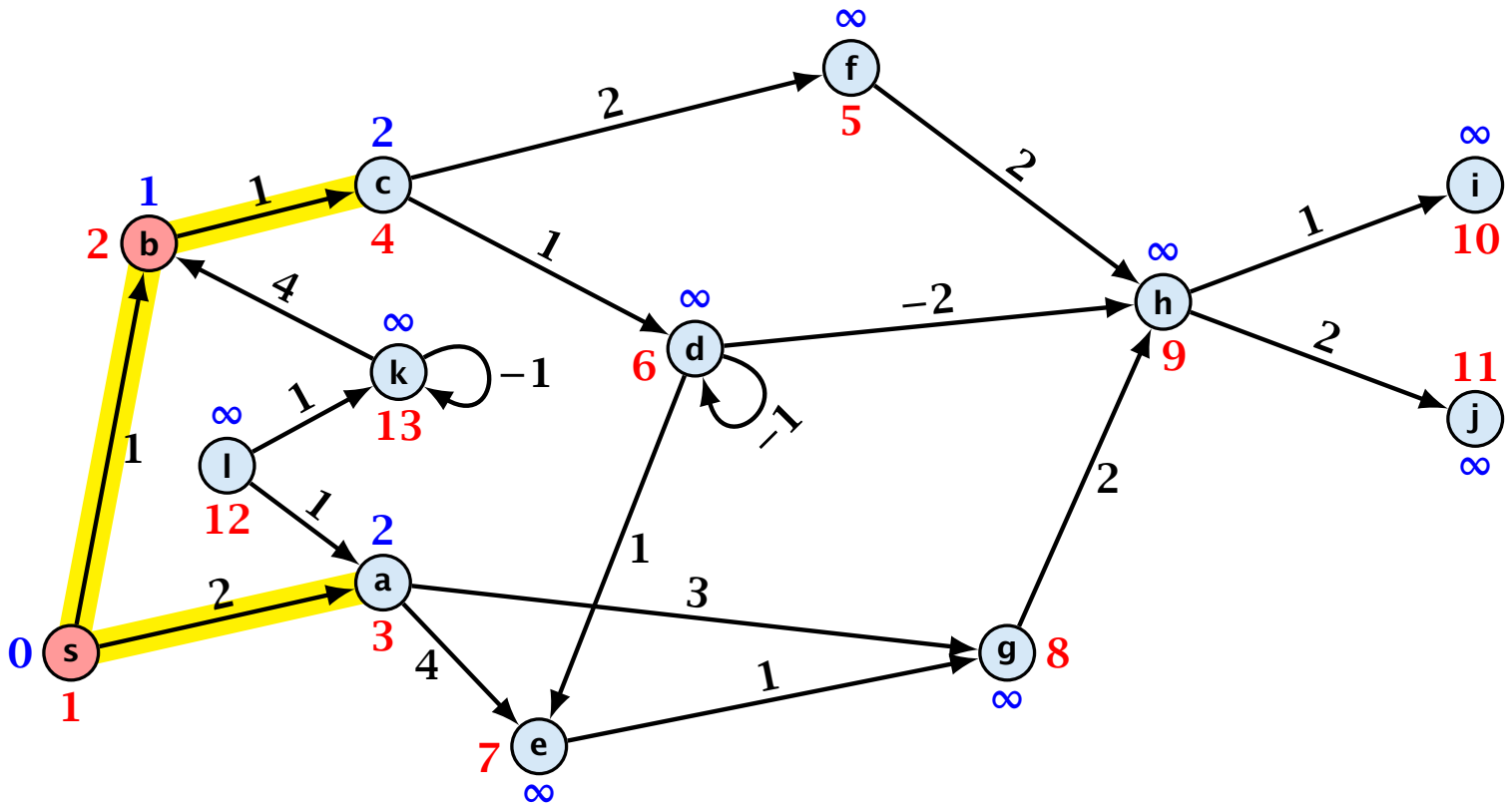
DAG with Self-Loops



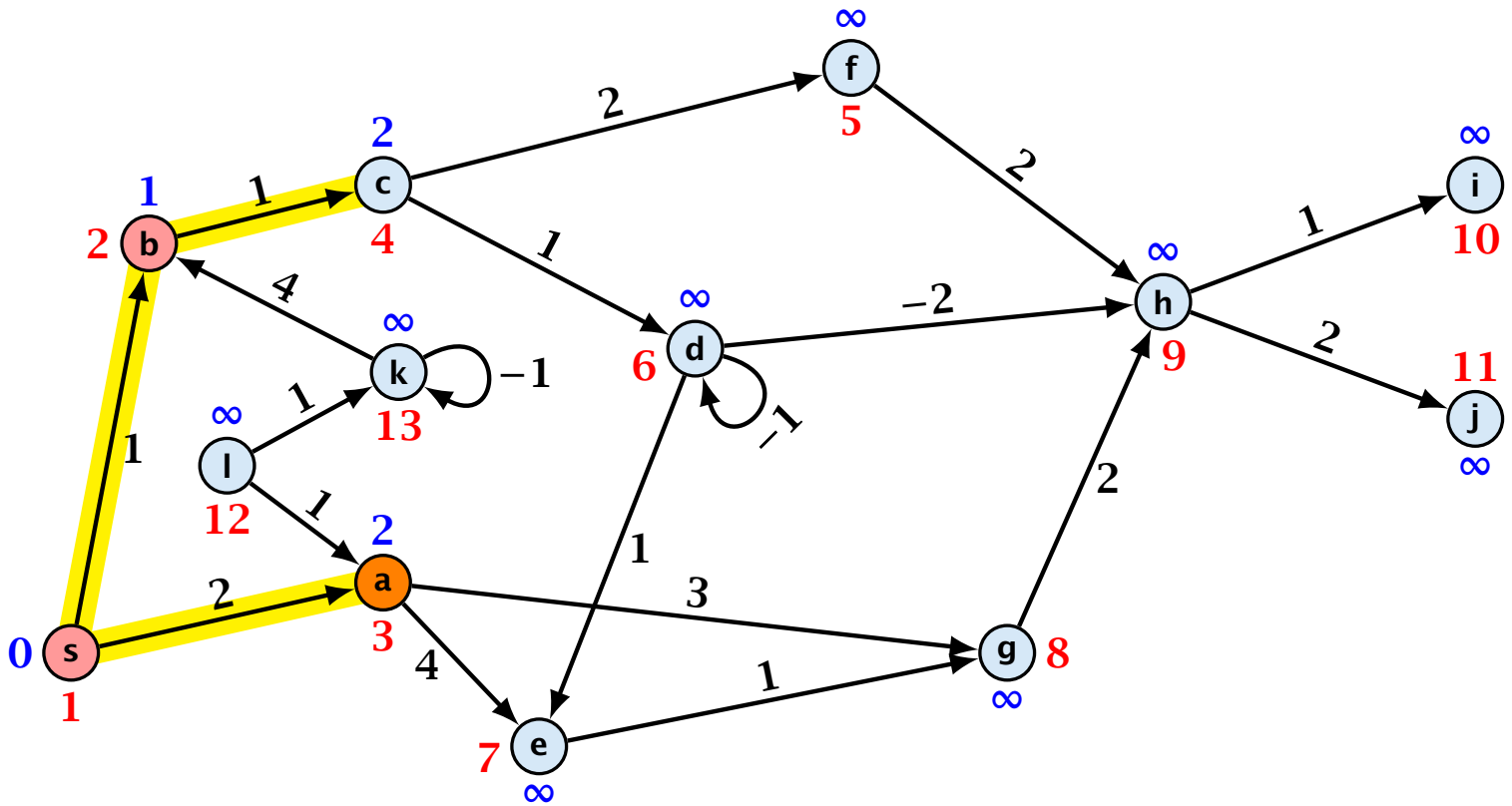
DAG with Self-Loops



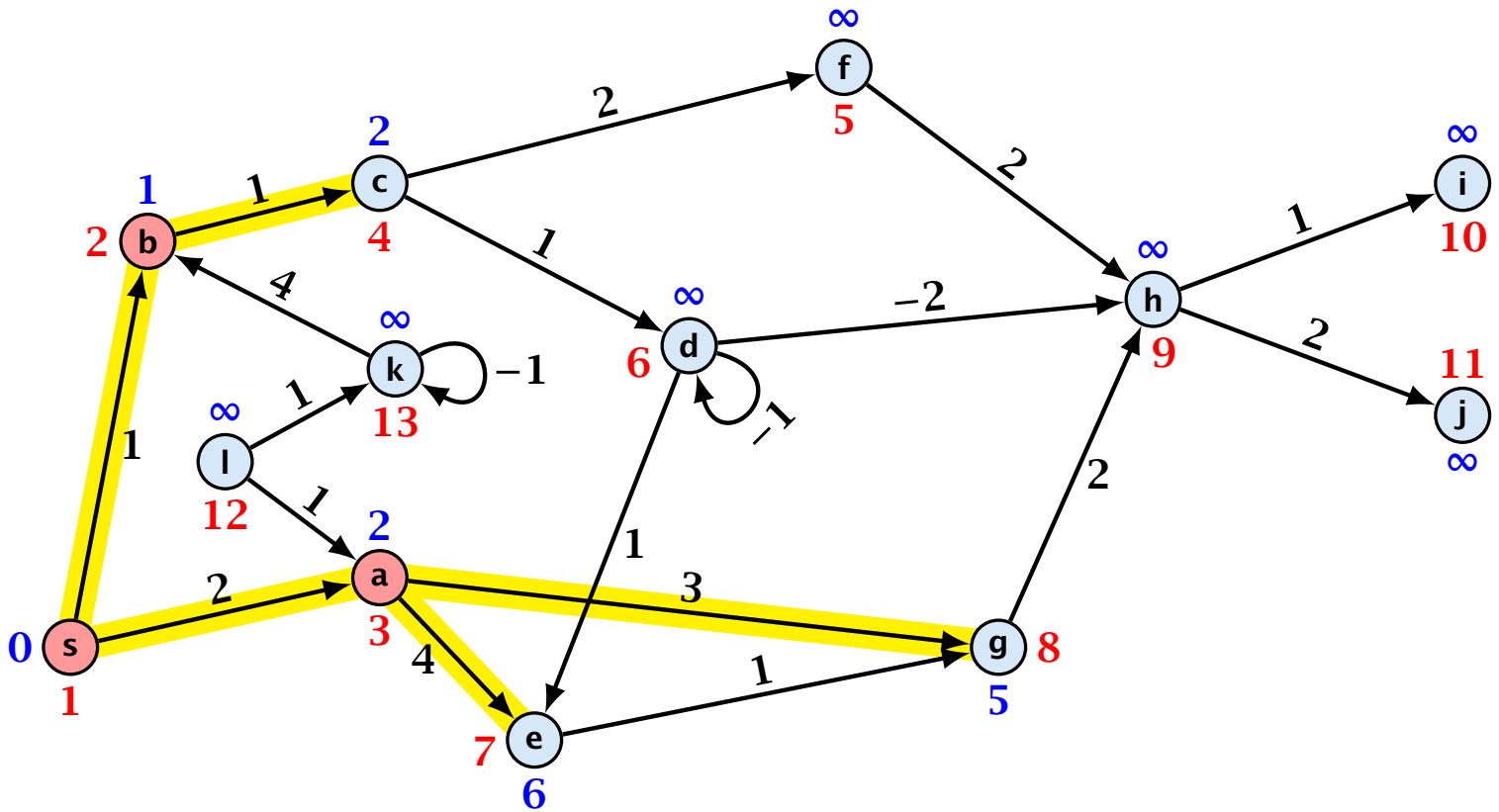
DAG with Self-Loops



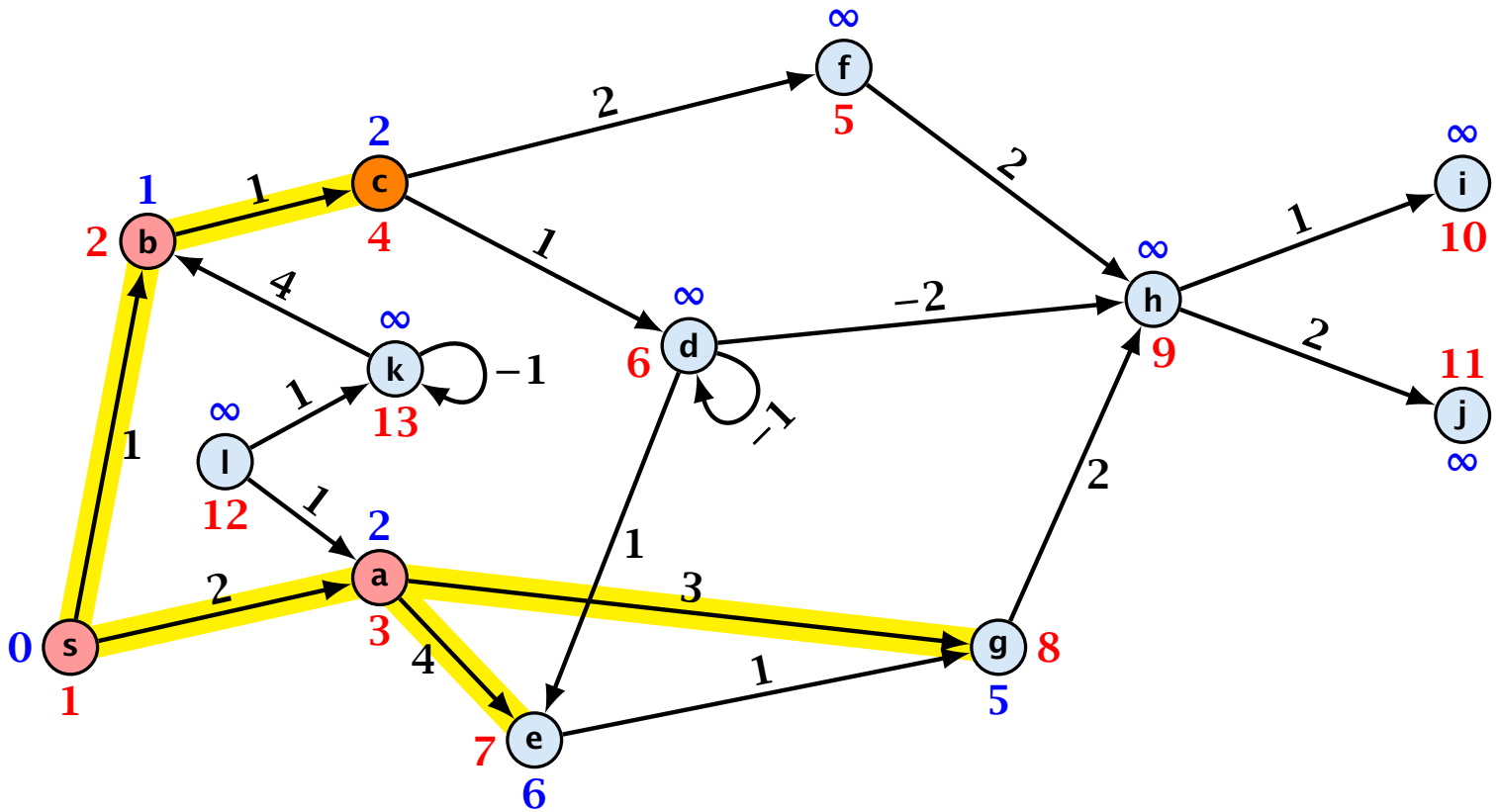
DAG with Self-Loops



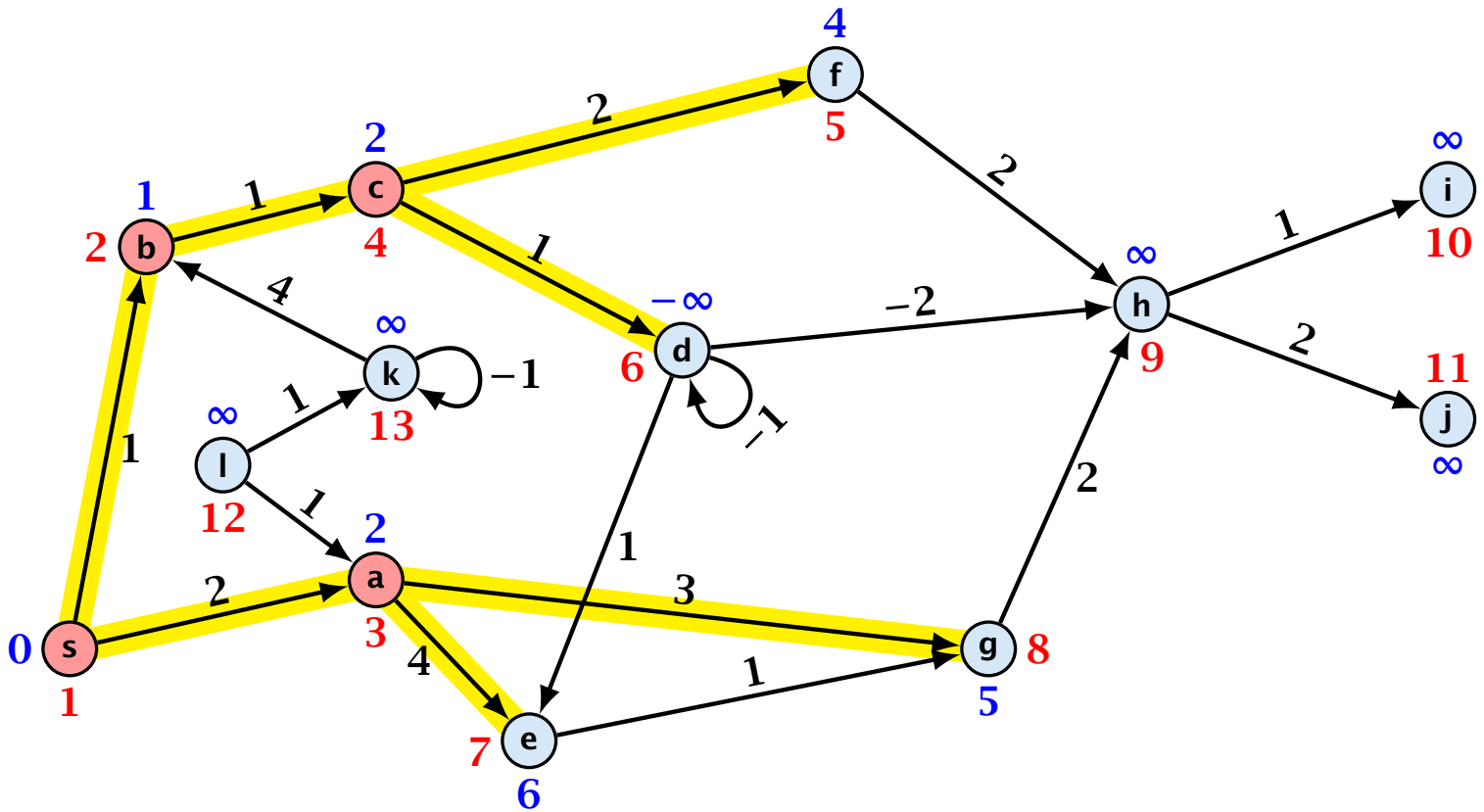
DAG with Self-Loops



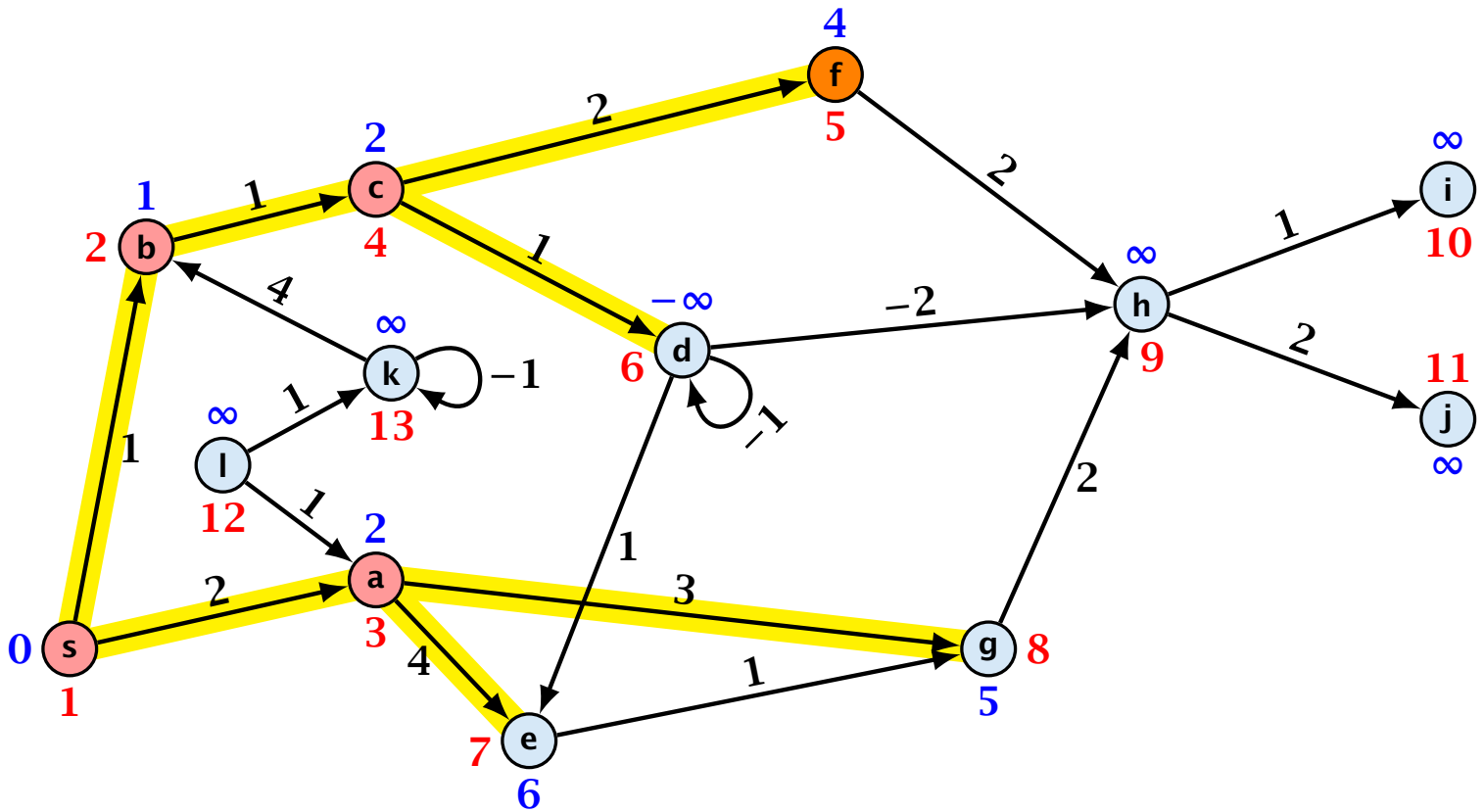
DAG with Self-Loops



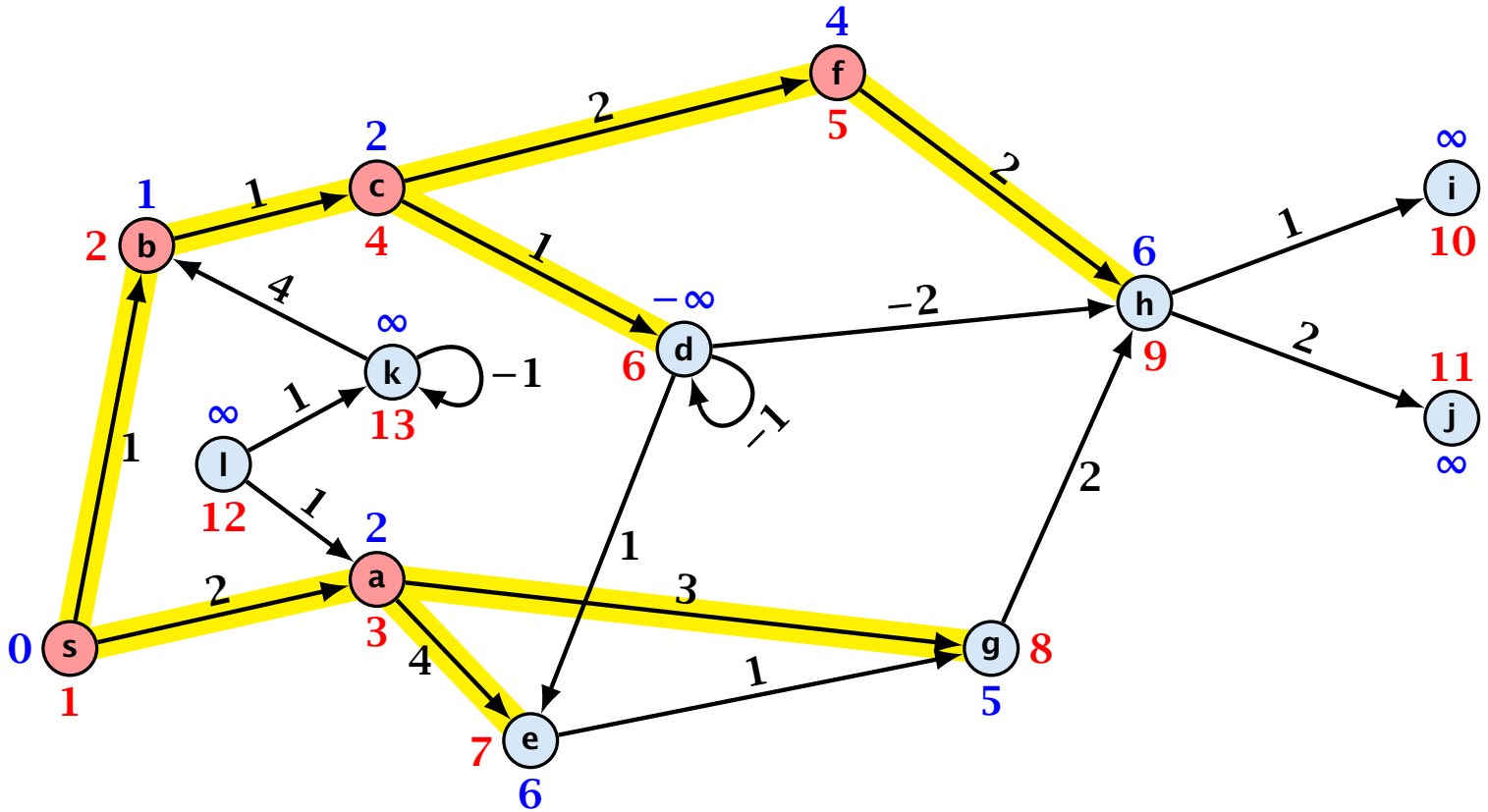
DAG with Self-Loops



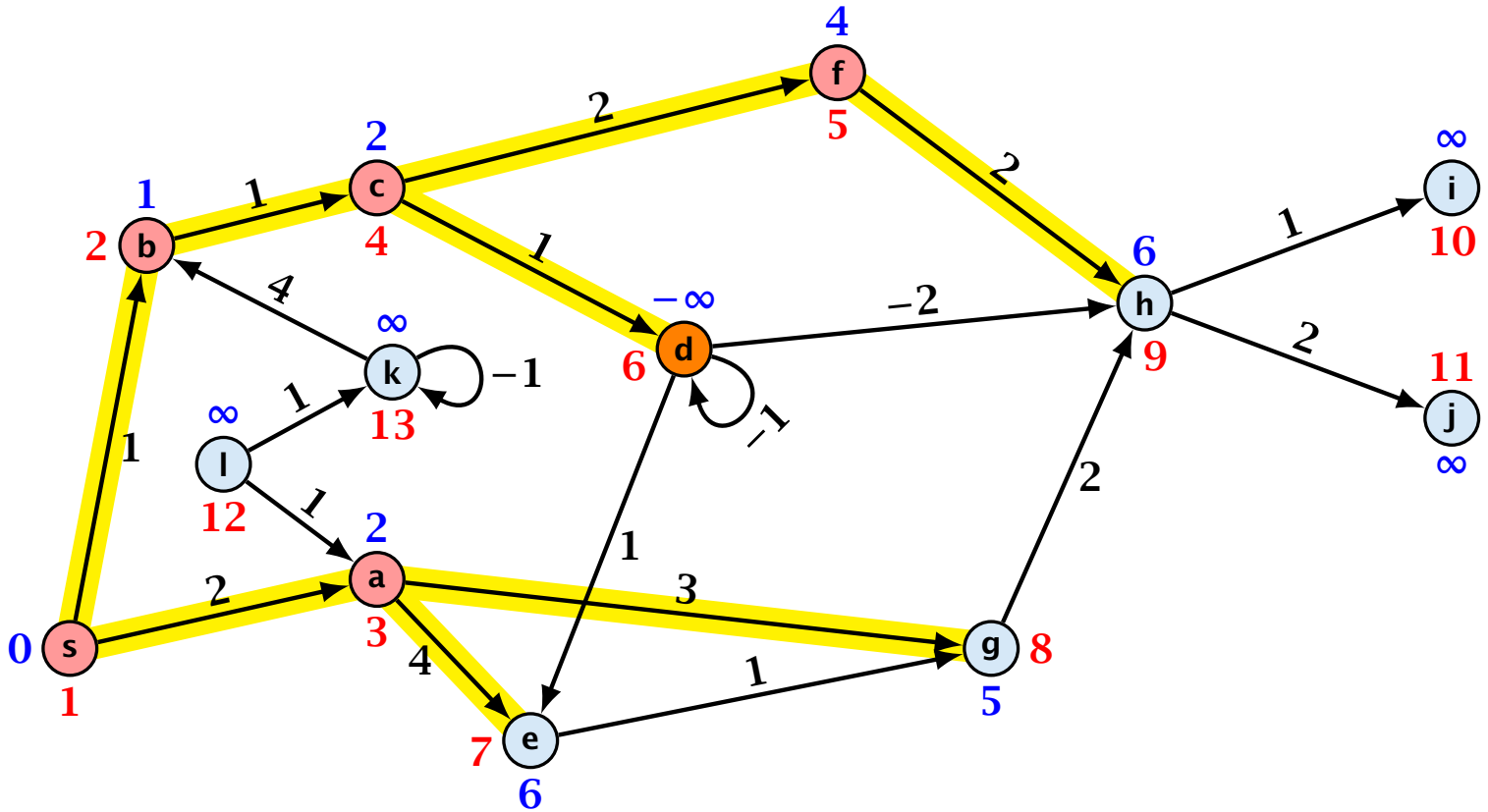
DAG with Self-Loops



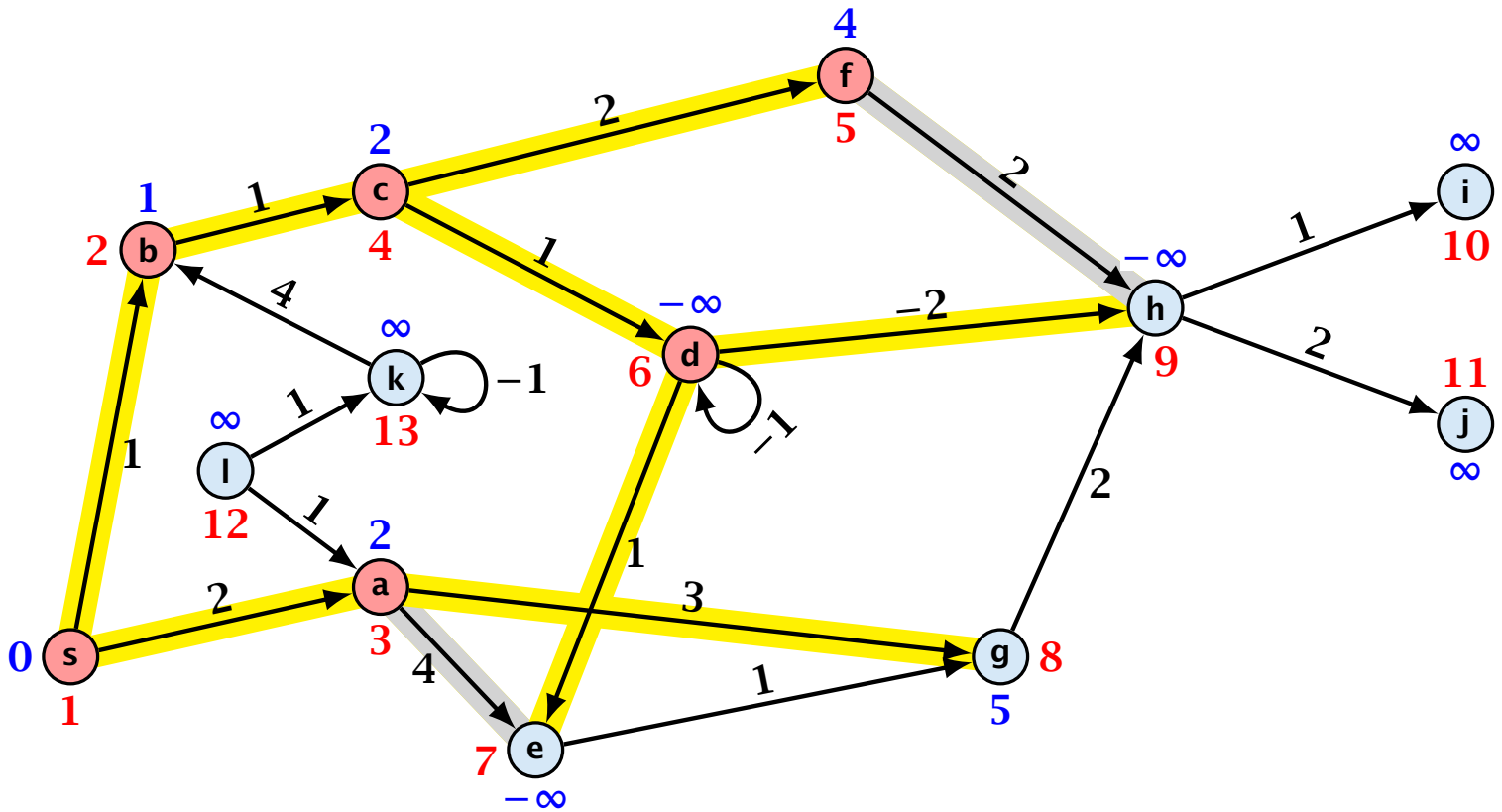
DAG with Self-Loops



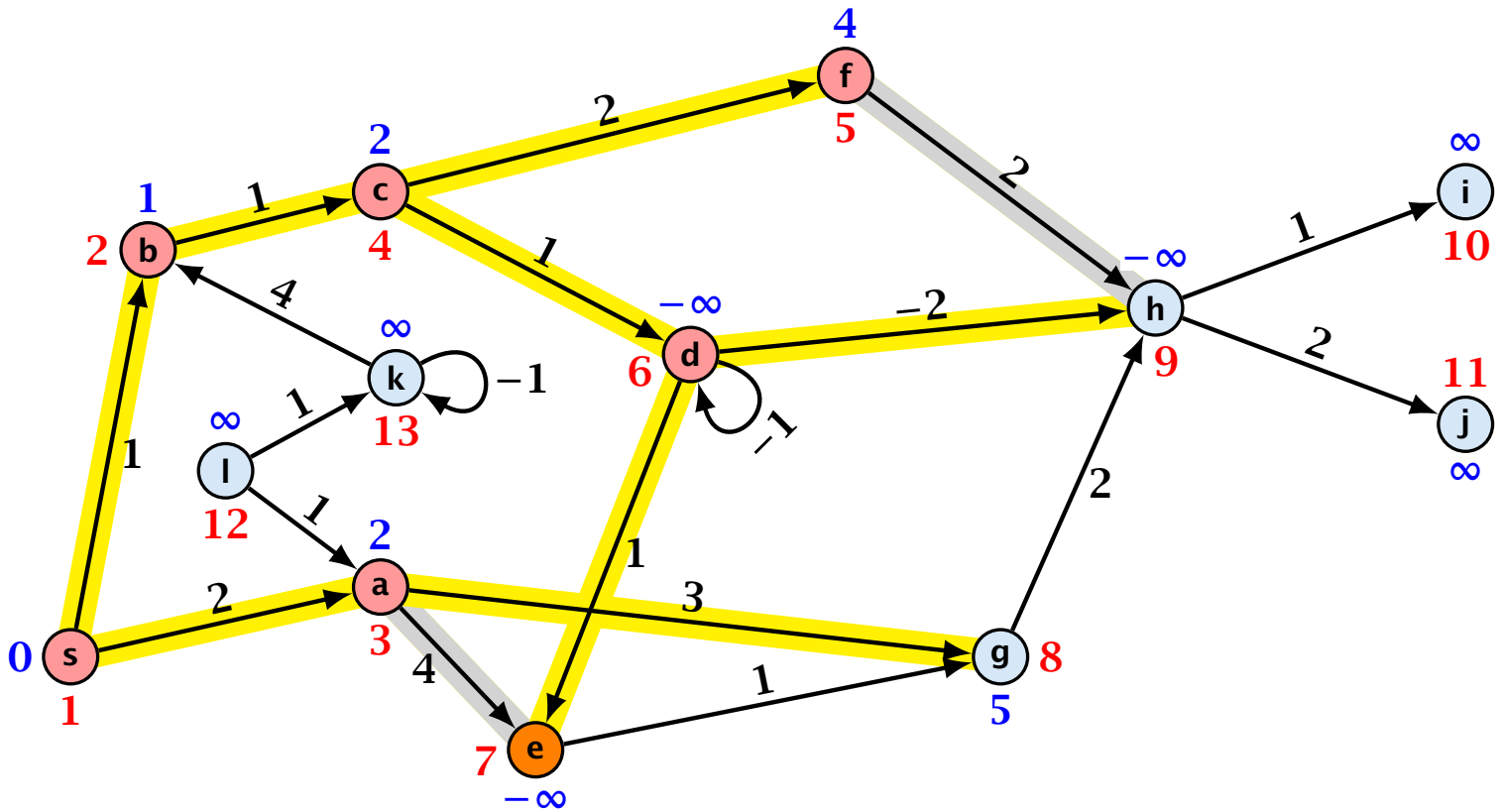
DAG with Self-Loops



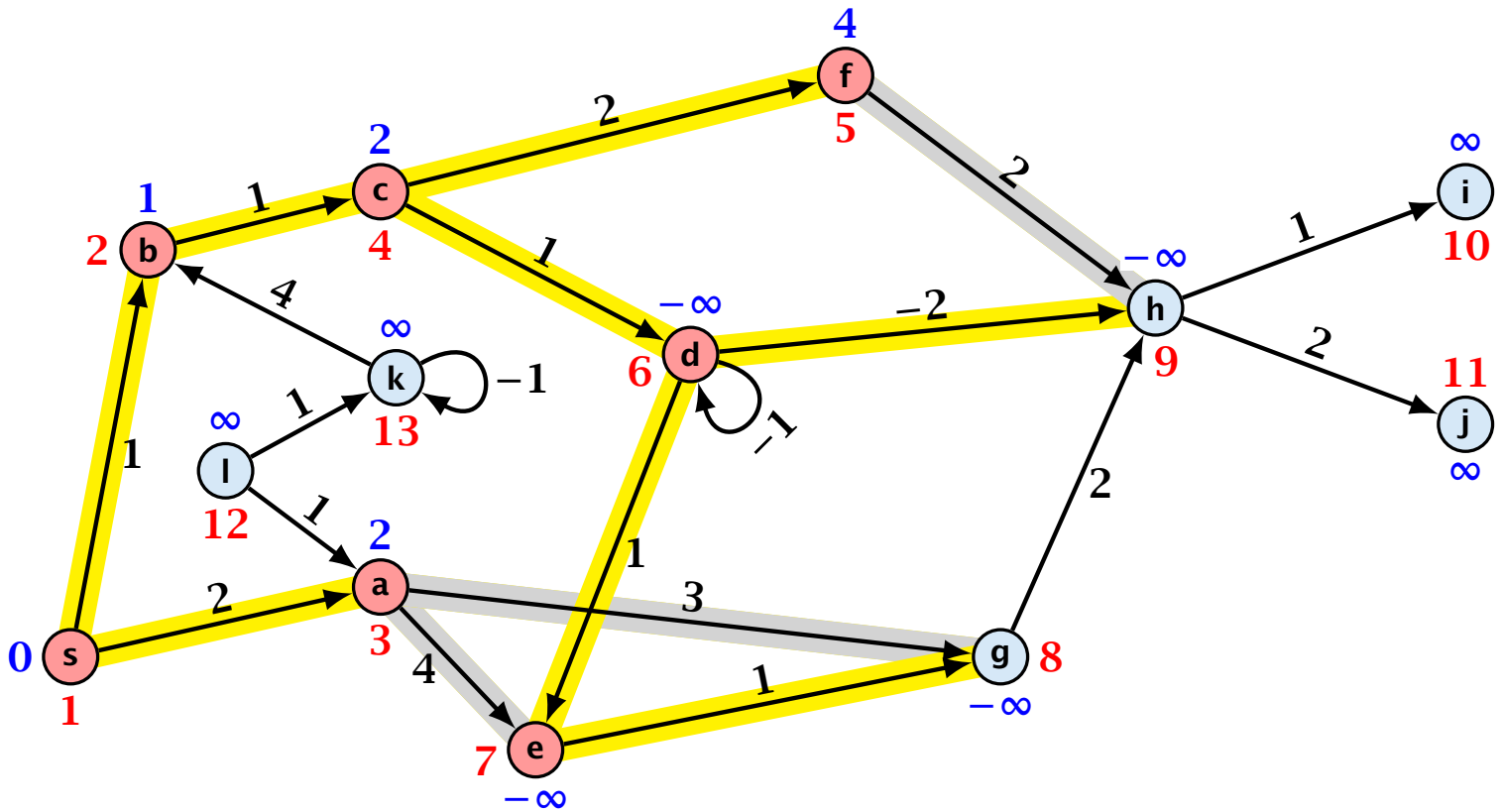
DAG with Self-Loops



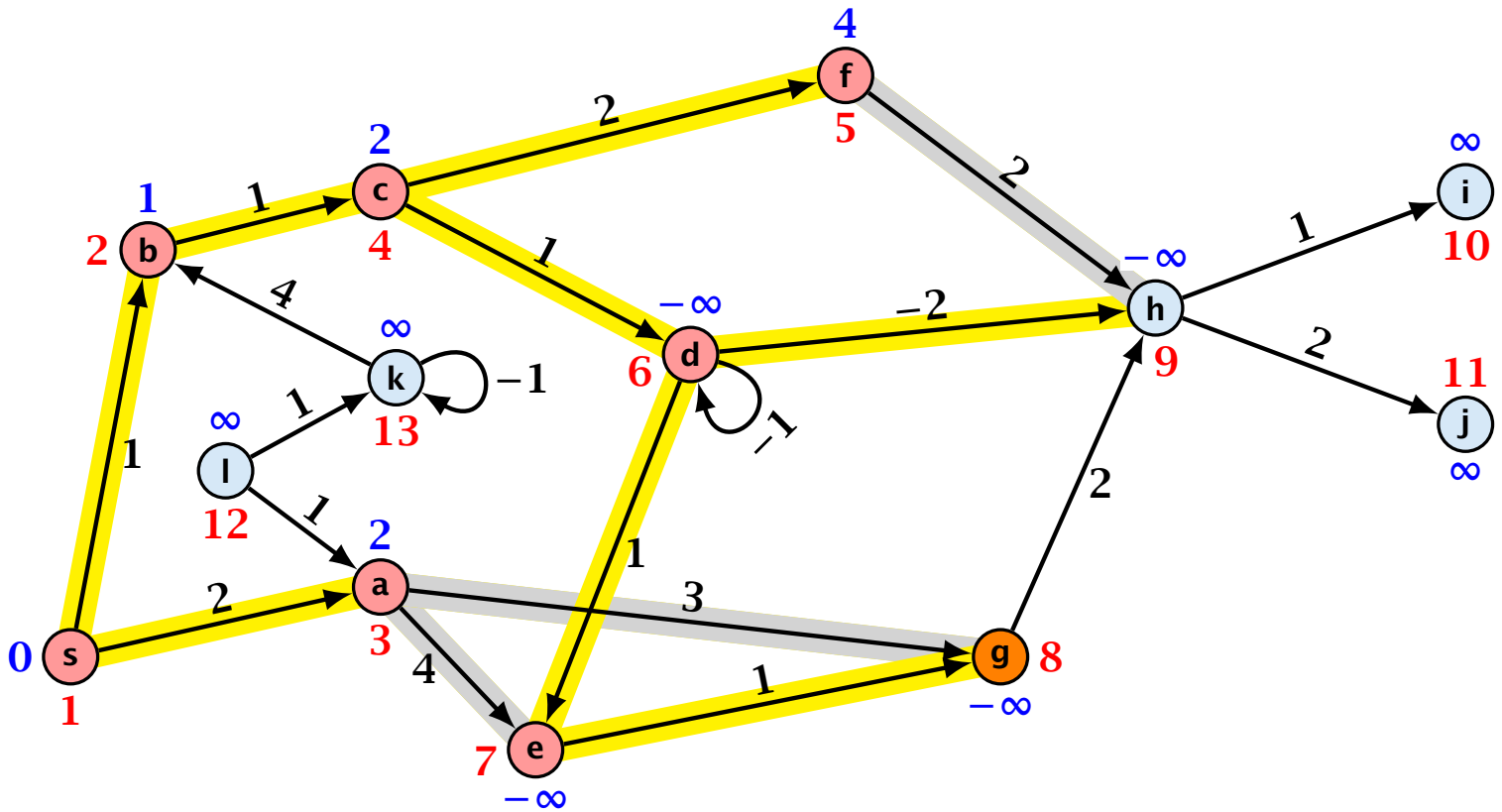
DAG with Self-Loops



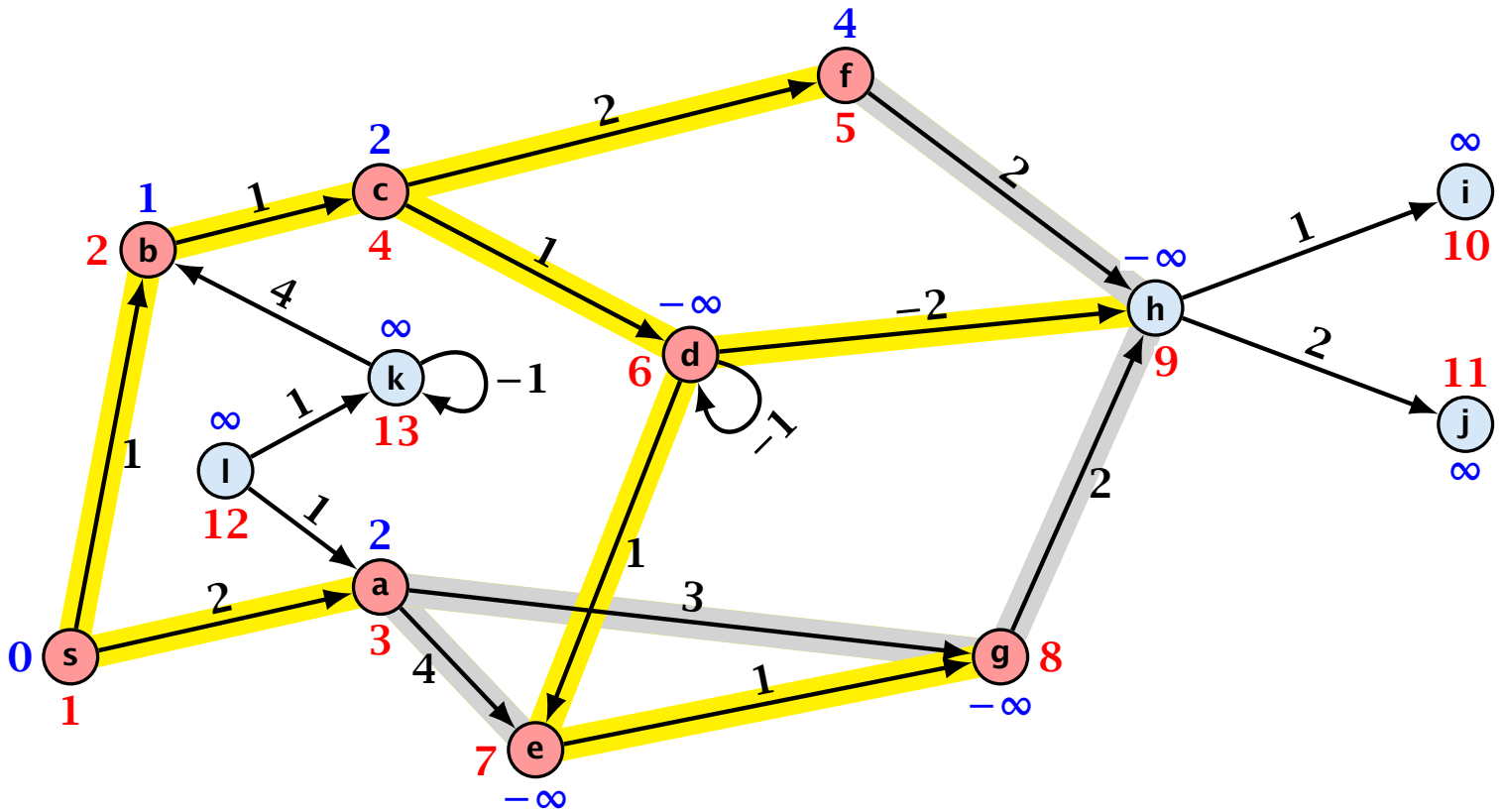
DAG with Self-Loops



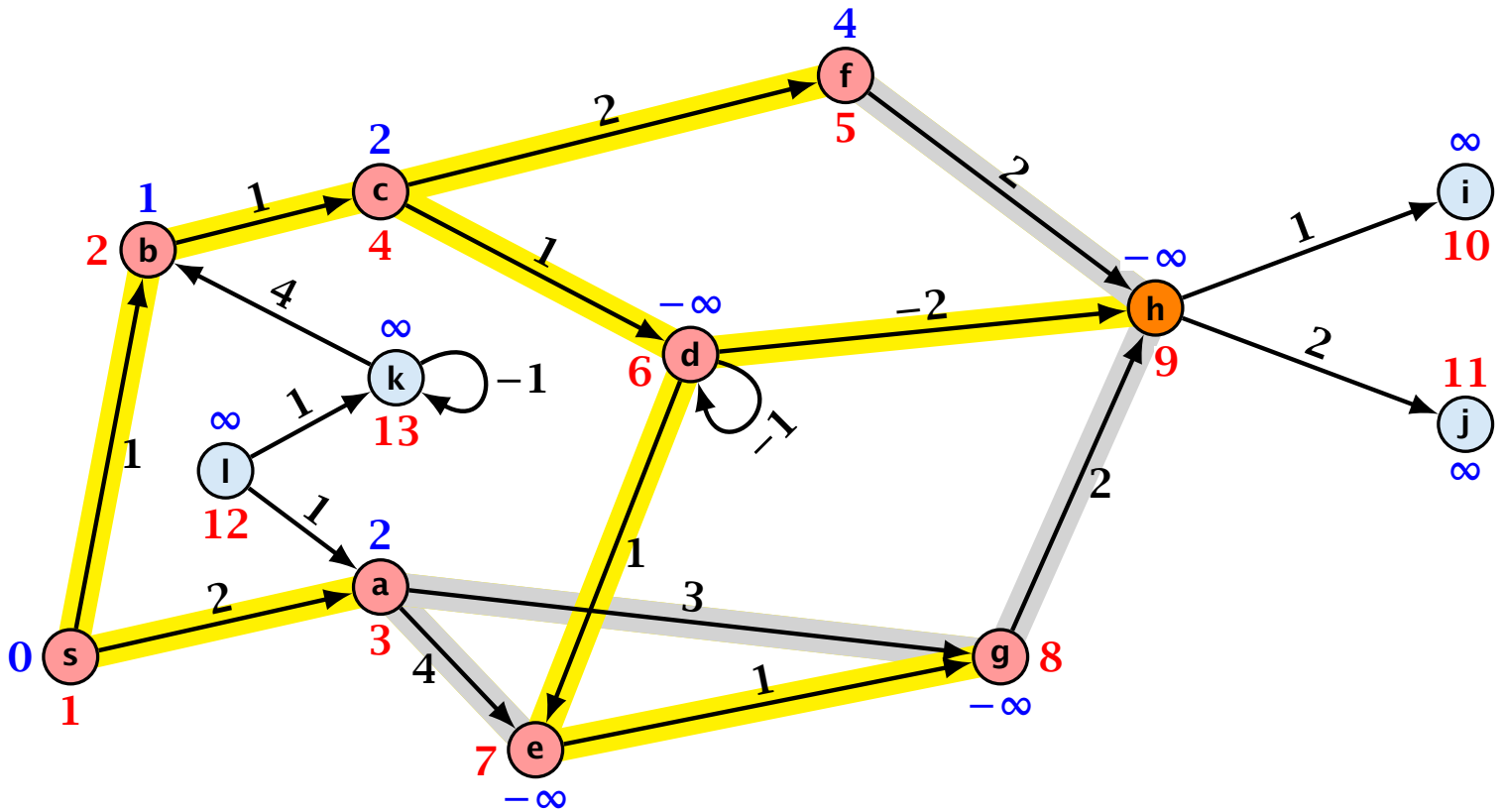
DAG with Self-Loops



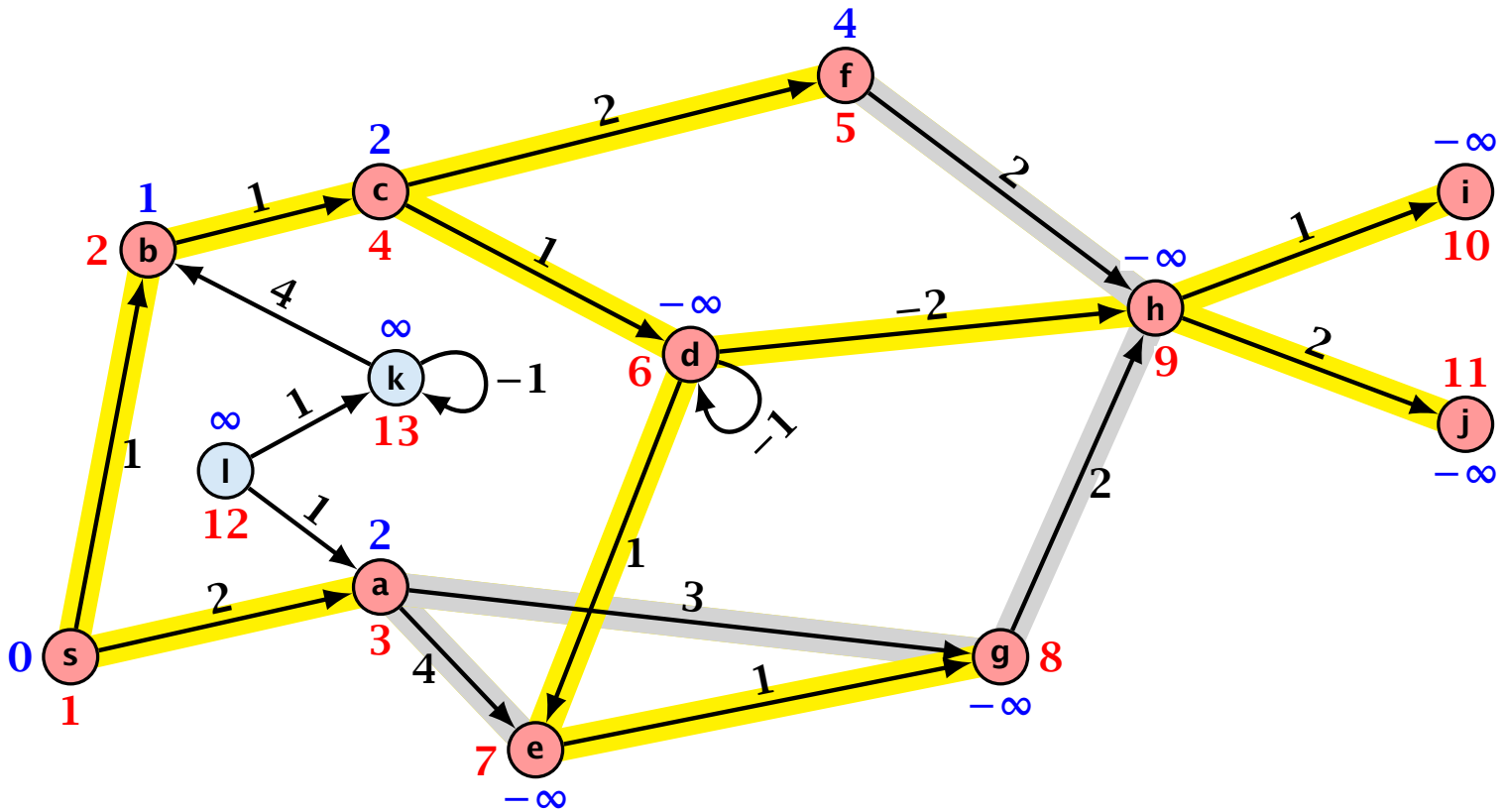
DAG with Self-Loops



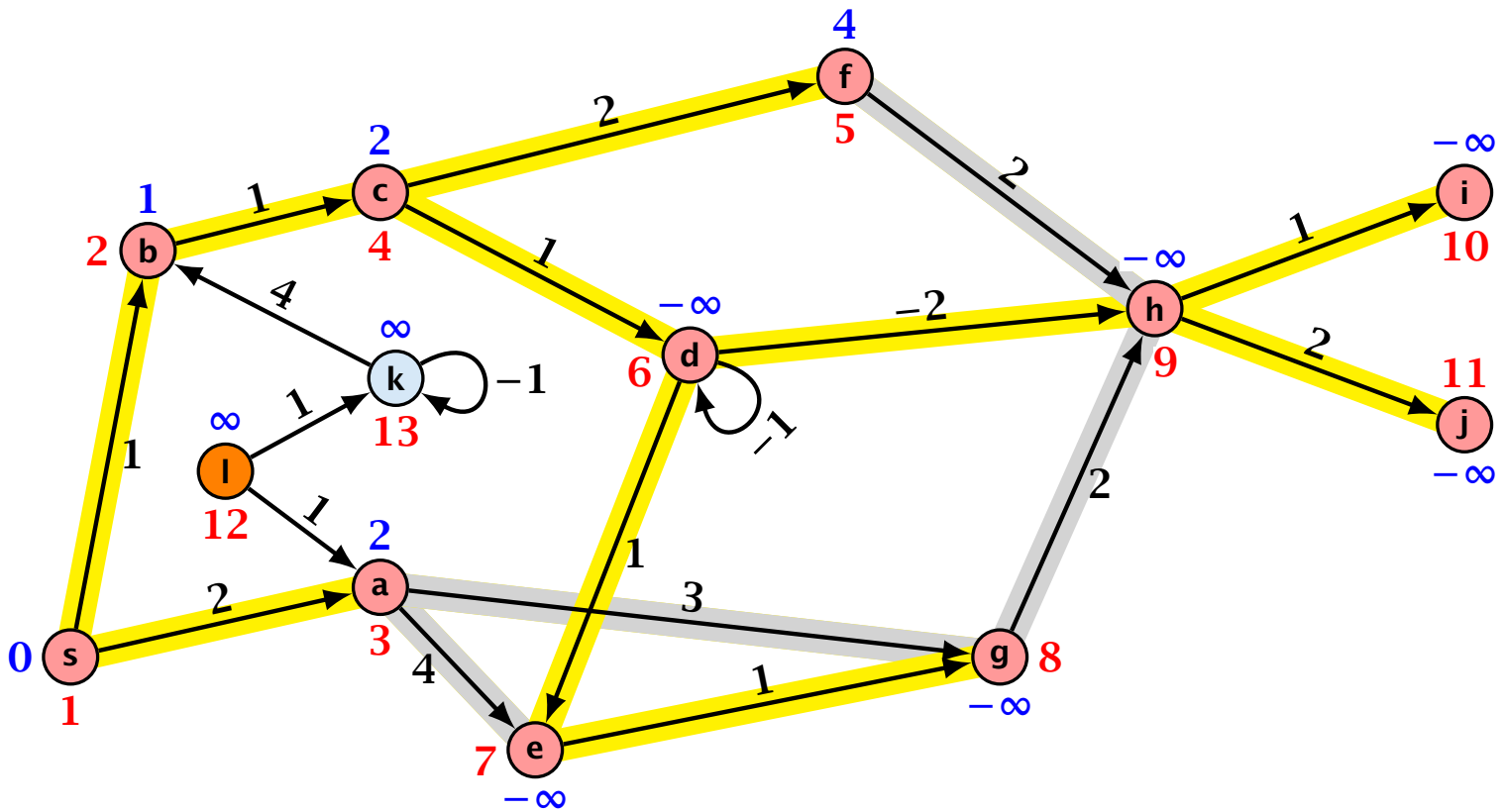
DAG with Self-Loops



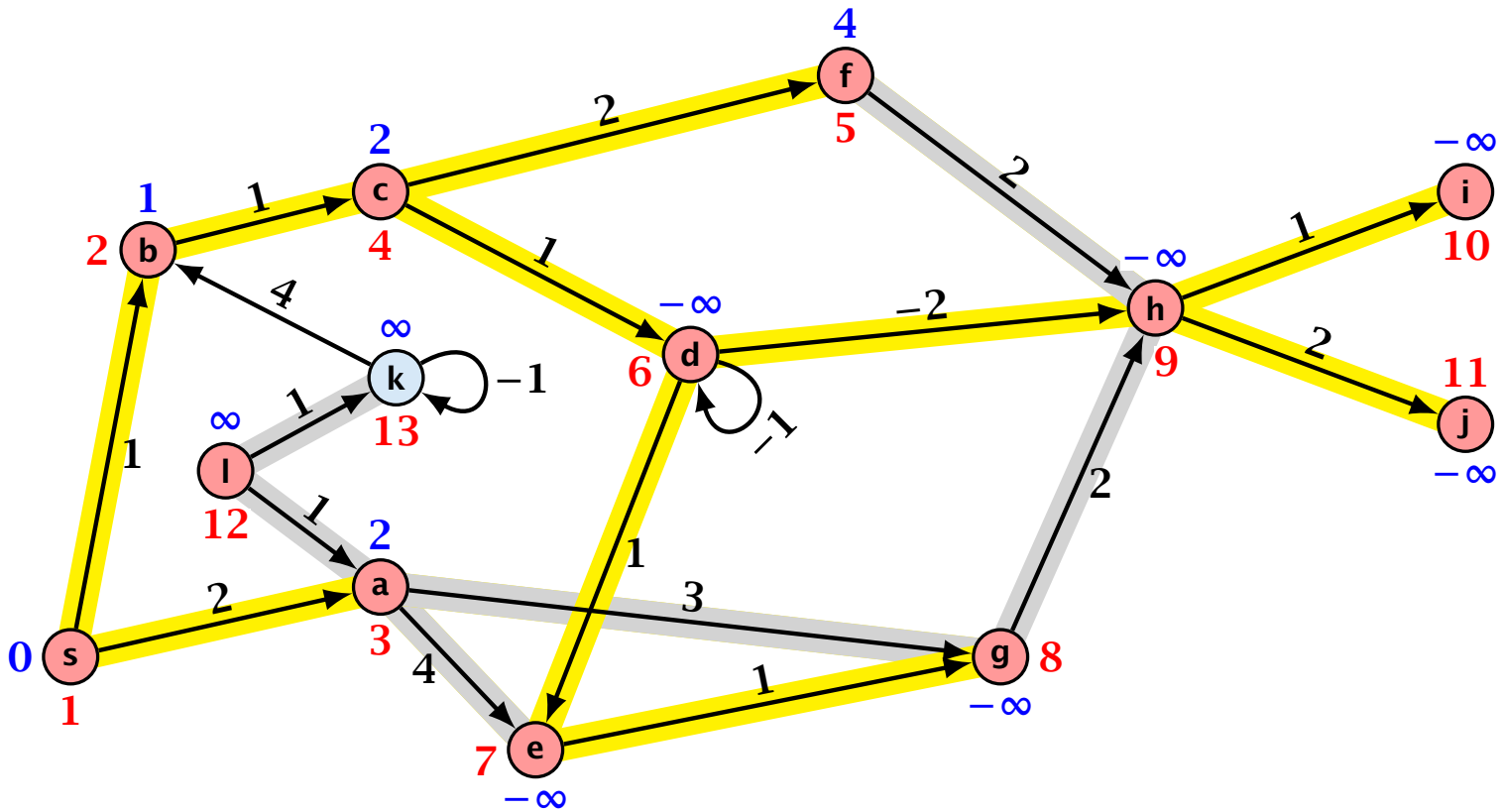
DAG with Self-Loops



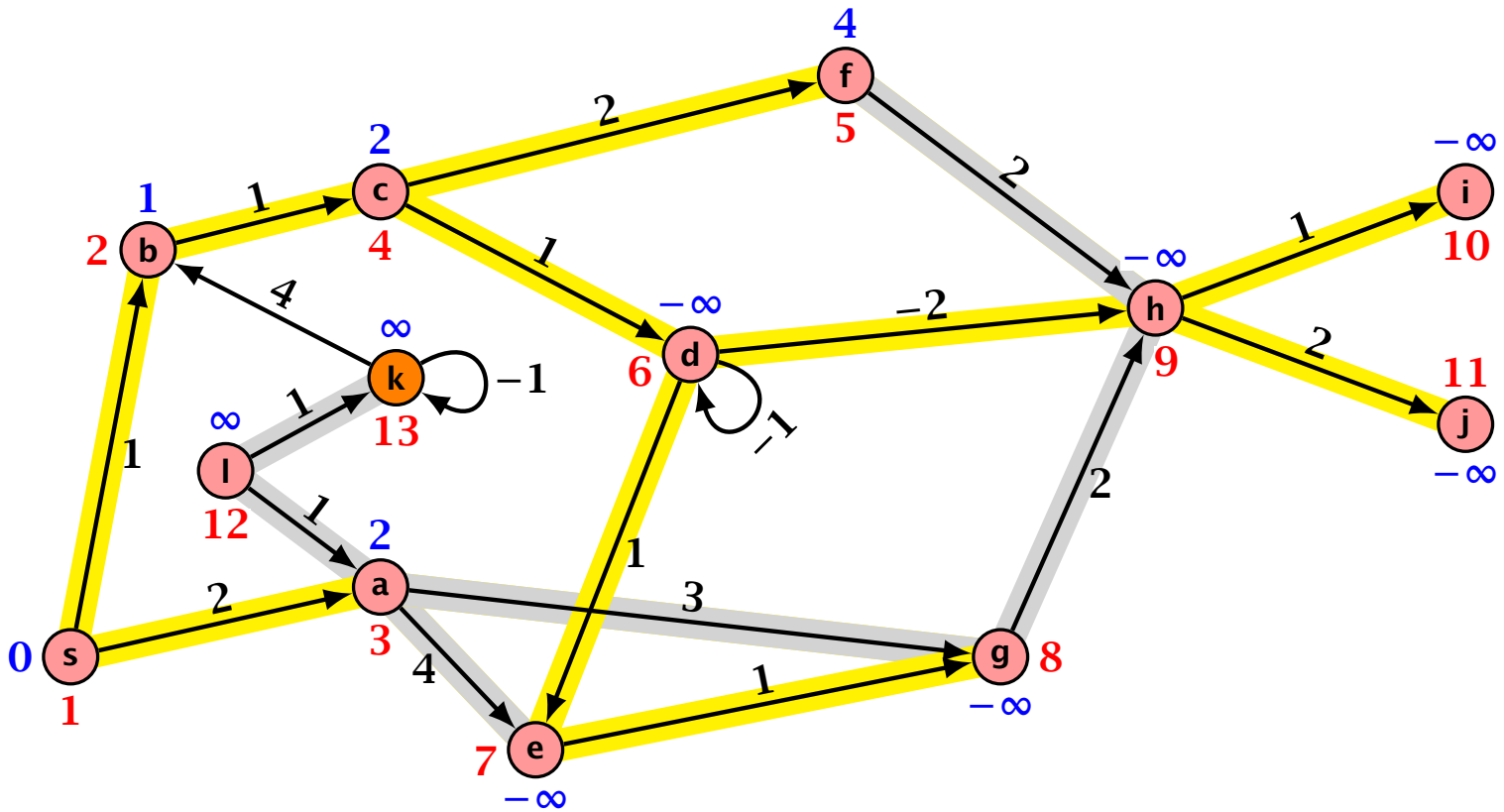
DAG with Self-Loops



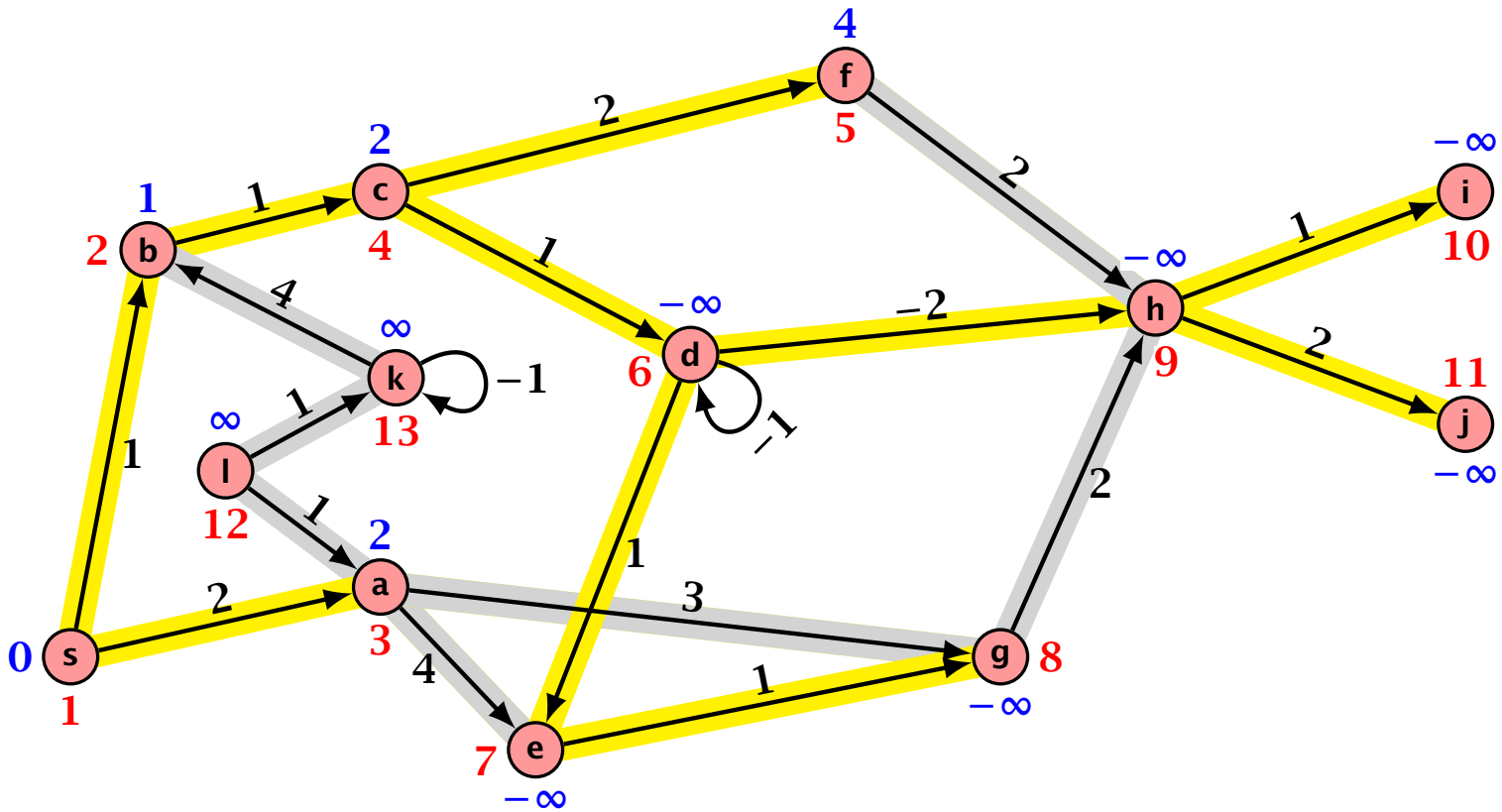
DAG with Self-Loops



DAG with Self-Loops



DAG with Self-Loops



General Graphs — Nonnegative Weights

Input:

- arbitrary graph (directed or undirected)
- non-negative edge-weights

General Graphs — Nonnegative Weights

Input:

- arbitrary graph (directed or undirected)
- non-negative edge-weights

Idea:

- do distance-relaxations along a shortest s-x path for every $x \in V$, $x \neq s$

General Graphs — Nonnegative Weights

Input:

- arbitrary graph (directed or undirected)
- non-negative edge-weights

Idea:

- do distance-relaxations along a shortest s-x path for every $x \in V$, $x \neq s$

Problem:

- perform relaxations in the right order

General Graphs — Nonnegative Weights

Input:

- arbitrary graph (directed or undirected)
- non-negative edge-weights

Idea:

- do distance-relaxations along a shortest s-x path for every $x \in V$, $x \neq s$

Problem:

- perform relaxations in the right order

Solution(?):

- perform relaxations in order of shortest path distance from source s

Dijkstra's Shortest Path Algorithm

Algorithm 3 Shortest-Path$(G = (V, E, w), s \in V)$

```
 1: Input: weighted graph G = (V,E,w); start vertex s;
 2: Output: key-field of every node contains distance from s;
 3: S.init(); // build empty priority queue
 4: for all v ∈ V do
 5:         v.key ← ∞;
 6:         v.parent ← ∅;
 7:         h_v ← S.insert(v);
 8: s.key ← 0; S.decrease-key(h_s,0);
 9: while S.is-empty() = false do
10:         v ← S.delete-min();
11:         for all x ∈ V s.t. (v,x) ∈ E do
12:             if x.key > v.key + w(v,x) then
13:                 S.decrease-key(h_x,v.key +w(v,x));
14:                 x.key ← v.key +w(v,x);
15:                 x.parent ← v;
```

General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**.init()**: Creates an empty set.
- **handle** S**.insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**.is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**.delete(**h**)**: Deletes element specified through handle h.
- S**.decrease-key(**h**,** k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**. init()**: Creates an empty set.
- **handle** S**. insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**. delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**. is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**. delete(**h**)**: Deletes element specified through handle h.
- S**. decrease-key(**h**,** k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**. init()**: Creates an empty set.
- **handle** S**. insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**. delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**. is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**. delete(**h**)**: Deletes element specified through handle h.
- S**. decrease-key(**h**,** k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**.init()**: Creates an empty set.
- **handle** S**.insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**.is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**.delete(**h**)**: Deletes element specified through handle h.
- S**.decrease-key(**h**,** k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**. init()**: Creates an empty set.
- **handle** S**. insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**. delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**. is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**. delete(**h**)**: Deletes element specified through handle h.
- S**. decrease-key(**h**,** k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**.init()**: Creates an empty set.
- **handle** S**.insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**.is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**.delete(**h**)**: Deletes element specified through handle h.
- S**.decrease-key(**h, k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

- S**.init()**: Creates an empty set.
- **handle** S**.insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**.is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**.delete(**h**)**: Deletes element specified through handle h.
- S**.decrease-key(**h**,** k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

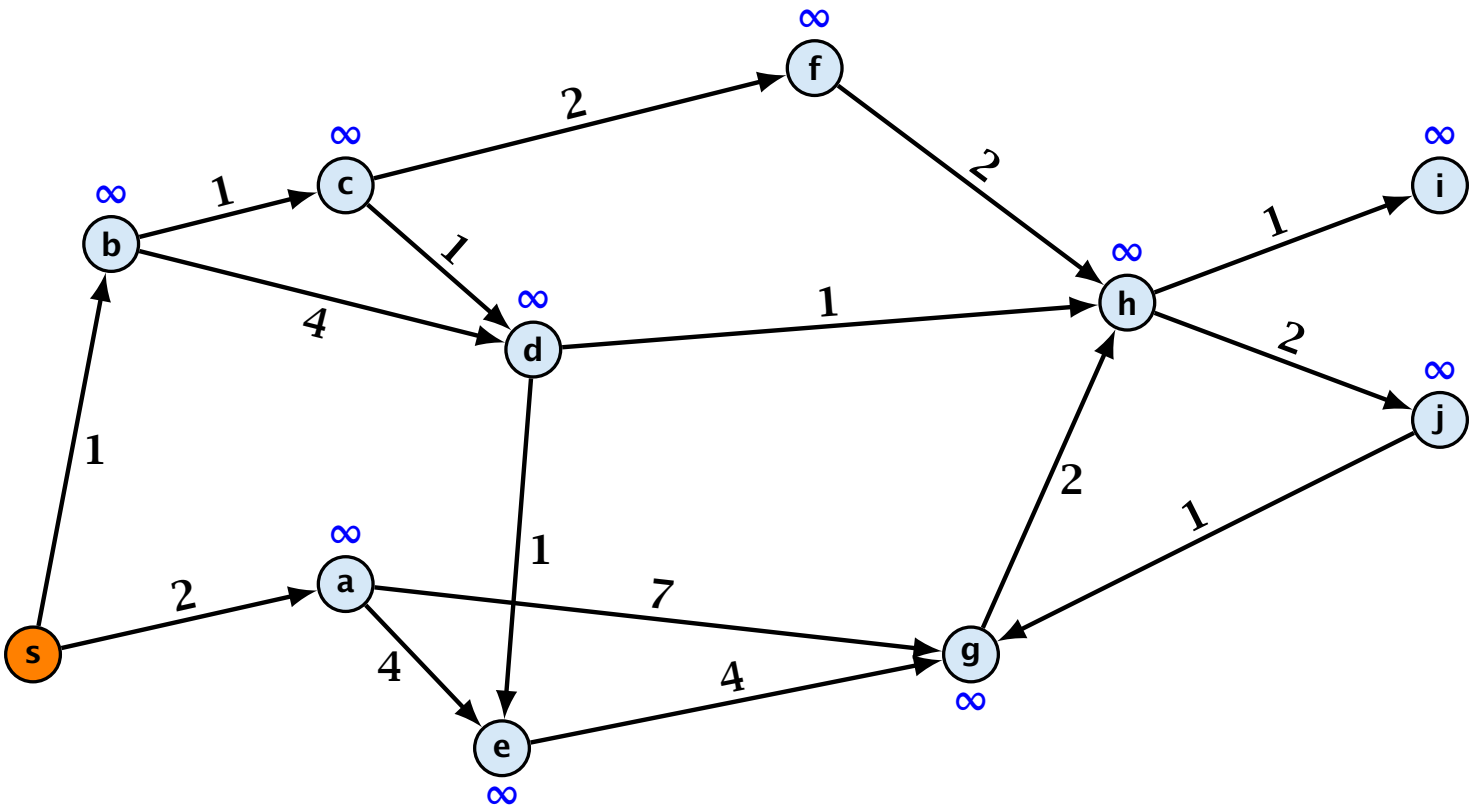
General Graphs — Nonnegative Weights

A Priority Queue S is a dynamic set data structure that supports the following operations:

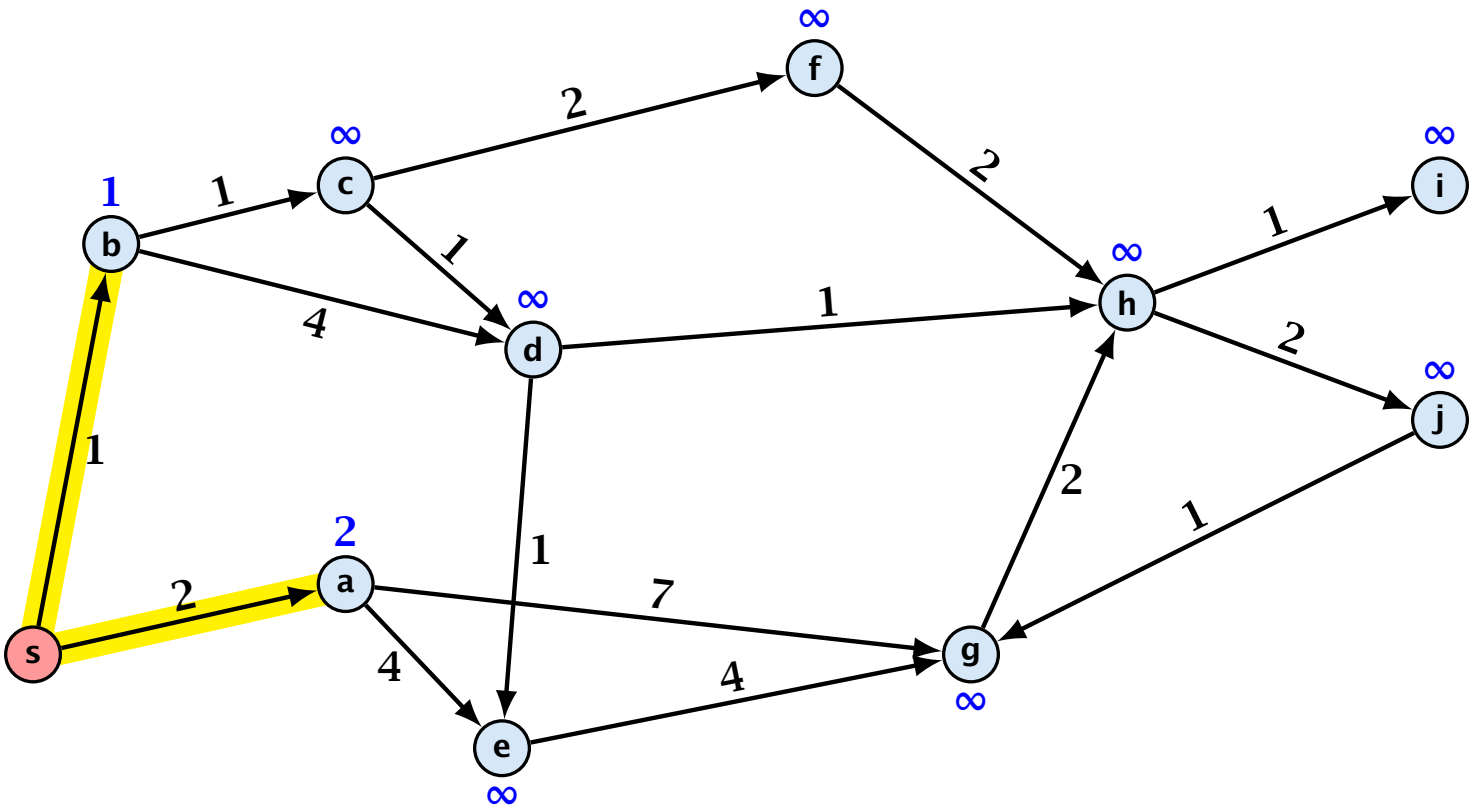
- S**.init()**: Creates an empty set.
- **handle** S**.insert(**x**)**: Adds element x to the data-structure, and returns a handle to the object for future reference.
- **element** S**.delete-min()**: Deletes the element with minimum key-value from S and returns it.
- **boolean** S**.is-empty()**: Returns true if the data-structure is empty and false otherwise.
- S**.delete(**h**)**: Deletes element specified through handle h.
- S**.decrease-key(**h**,** k**)**: Decreases the key of the element specified by handle h to k. Assumes that the key is at least k before the operation.

Using AVL-trees we can get time $\mathcal{O}(\log n)$ for all operations

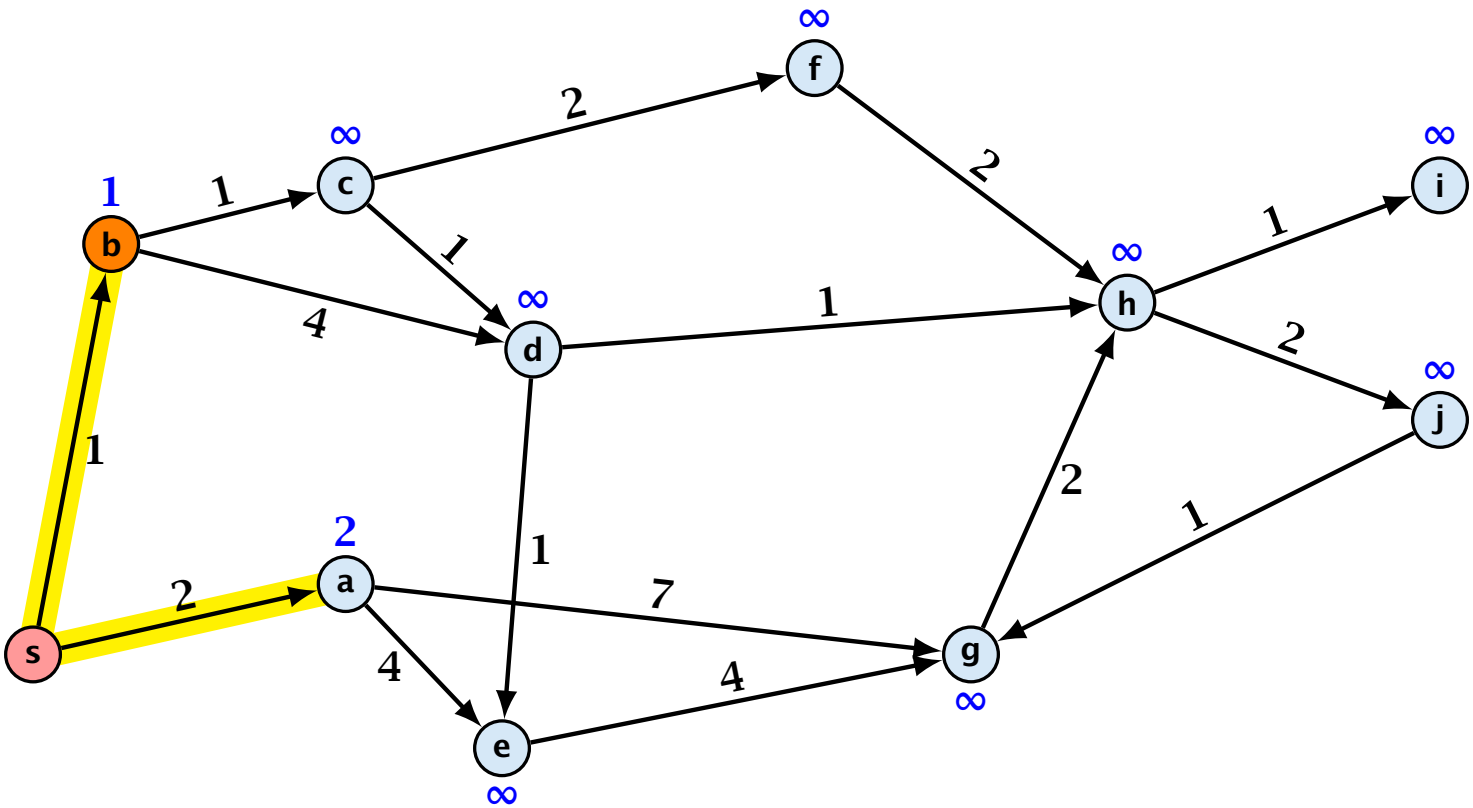
General Graphs — Nonnegative Weights



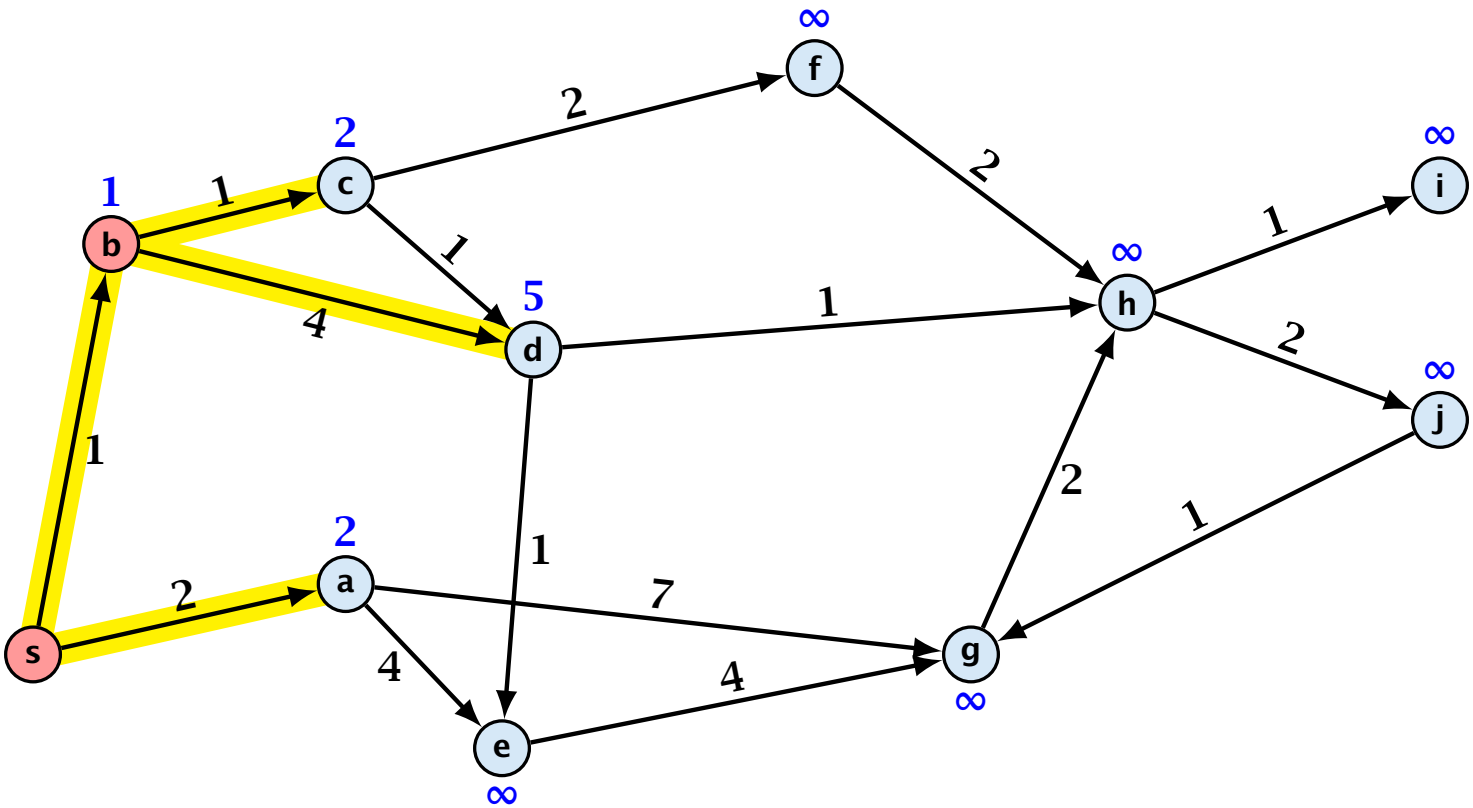
General Graphs — Nonnegative Weights



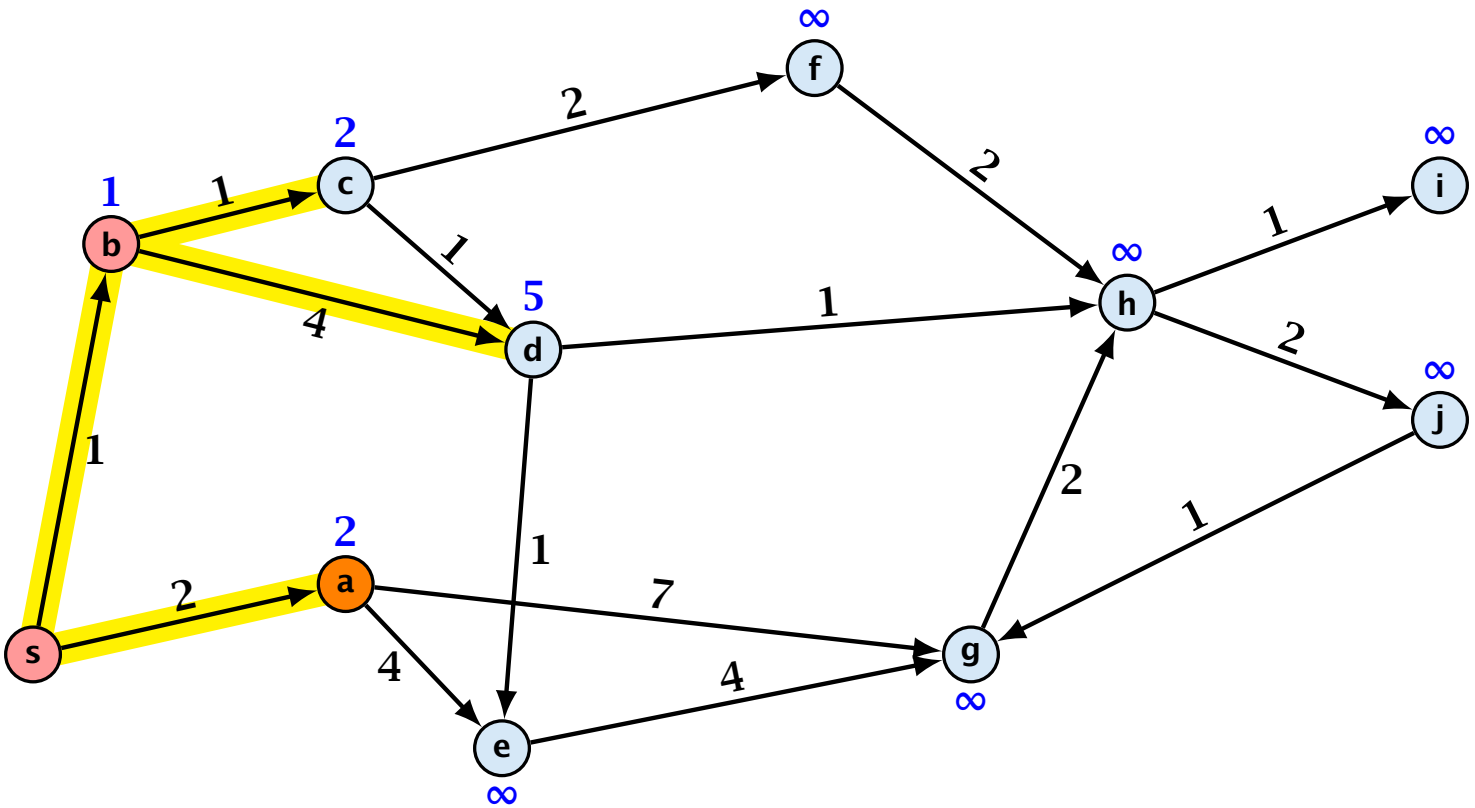
General Graphs — Nonnegative Weights



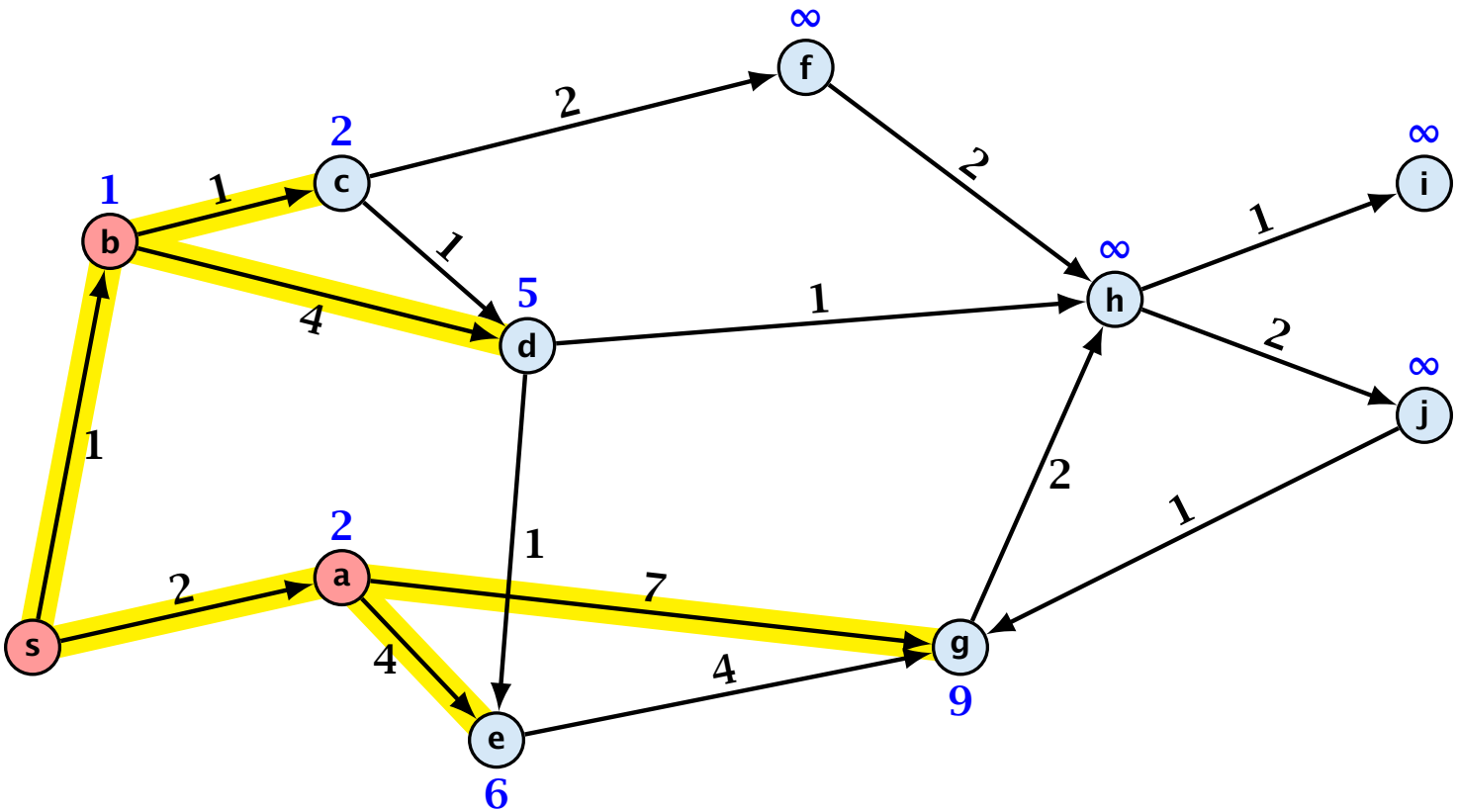
General Graphs — Nonnegative Weights



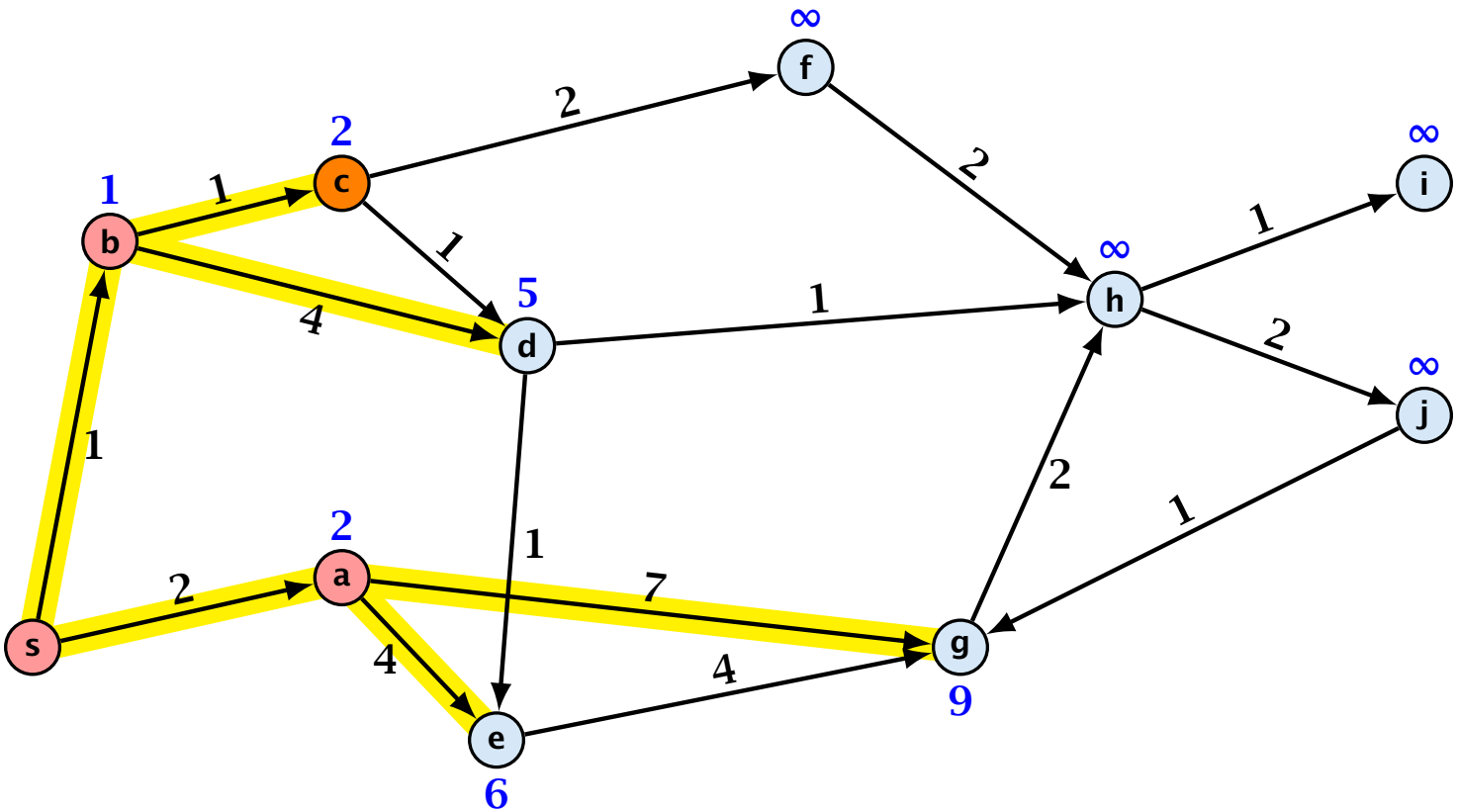
General Graphs — Nonnegative Weights



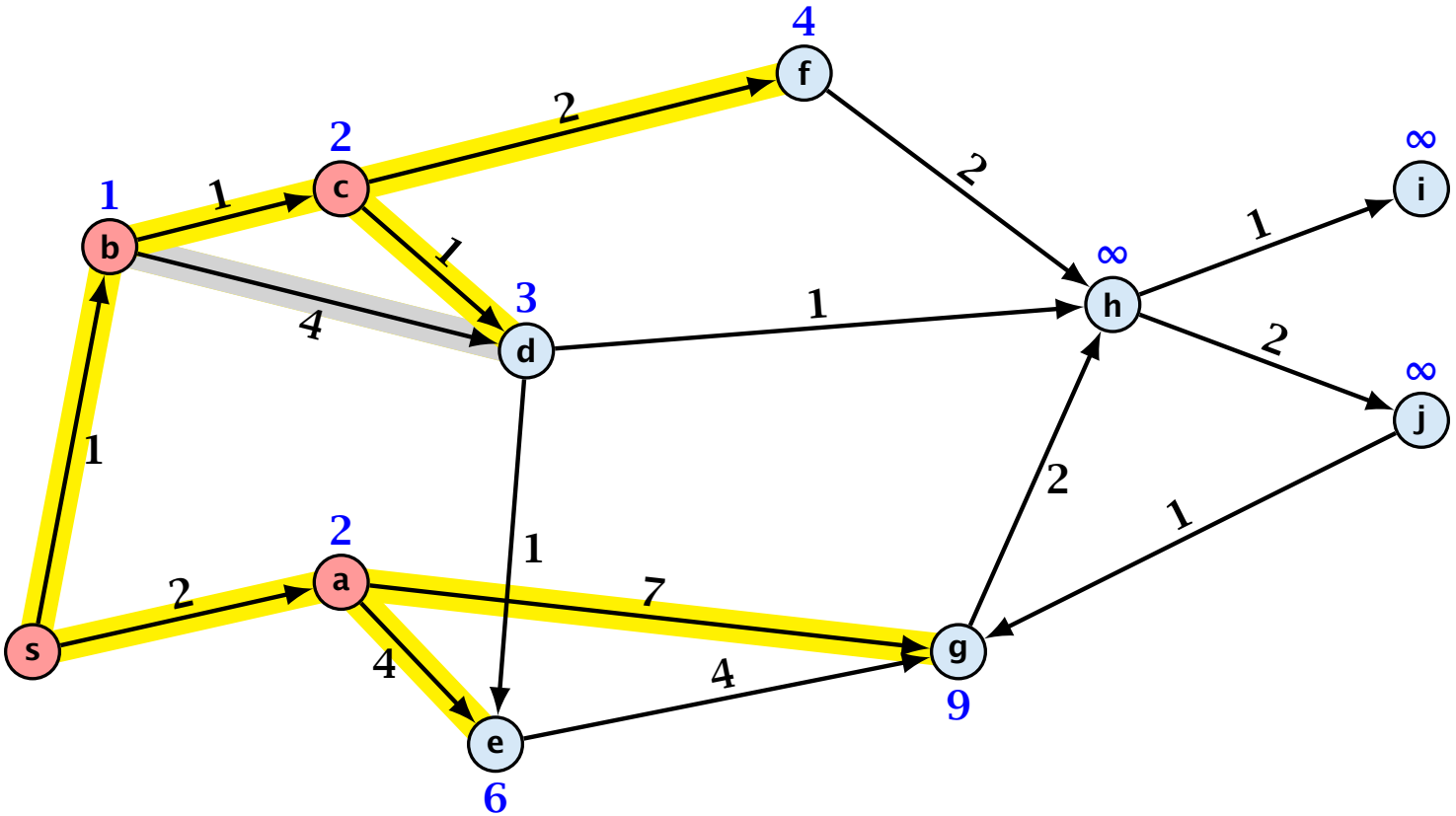
General Graphs — Nonnegative Weights



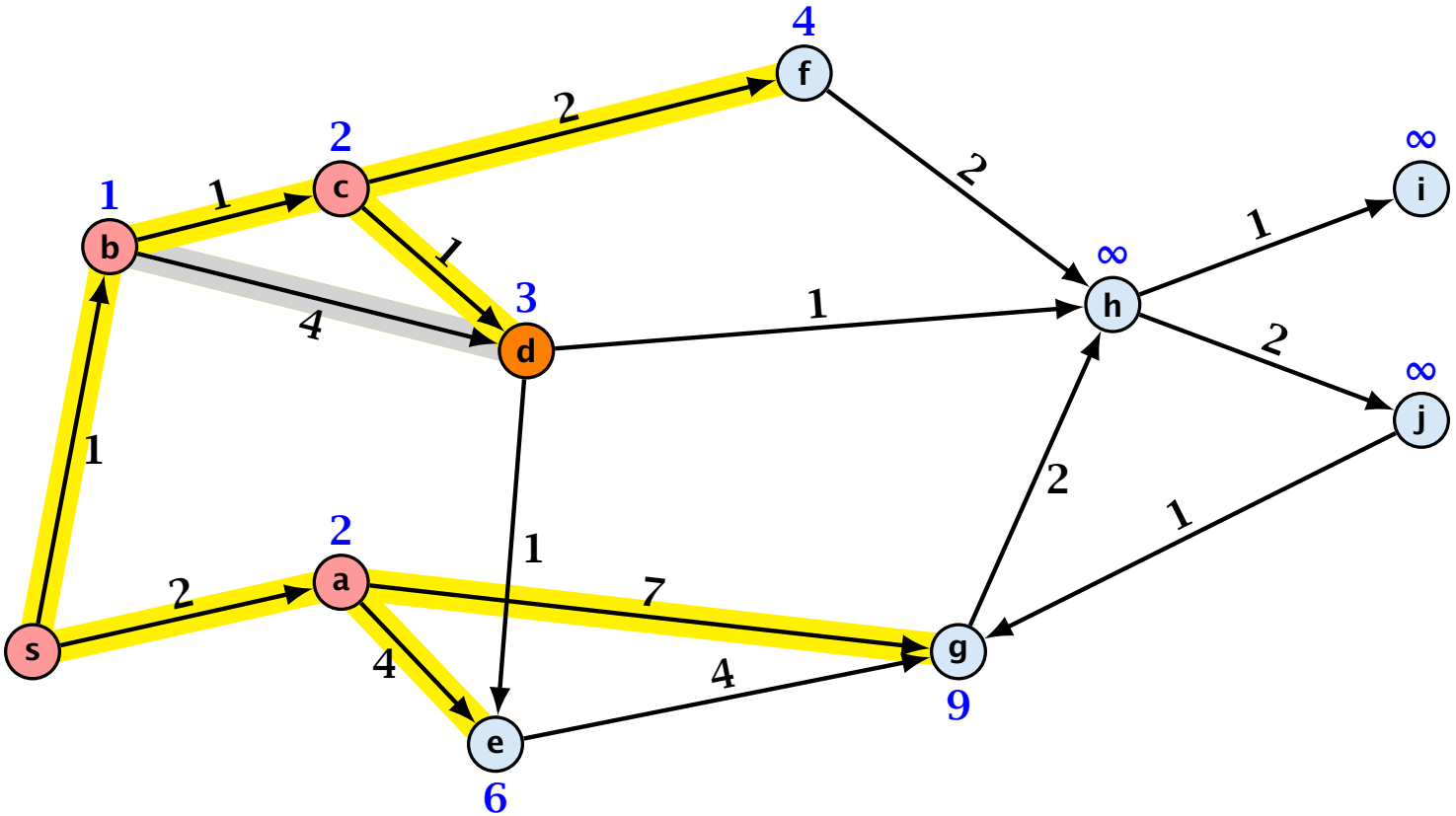
General Graphs — Nonnegative Weights



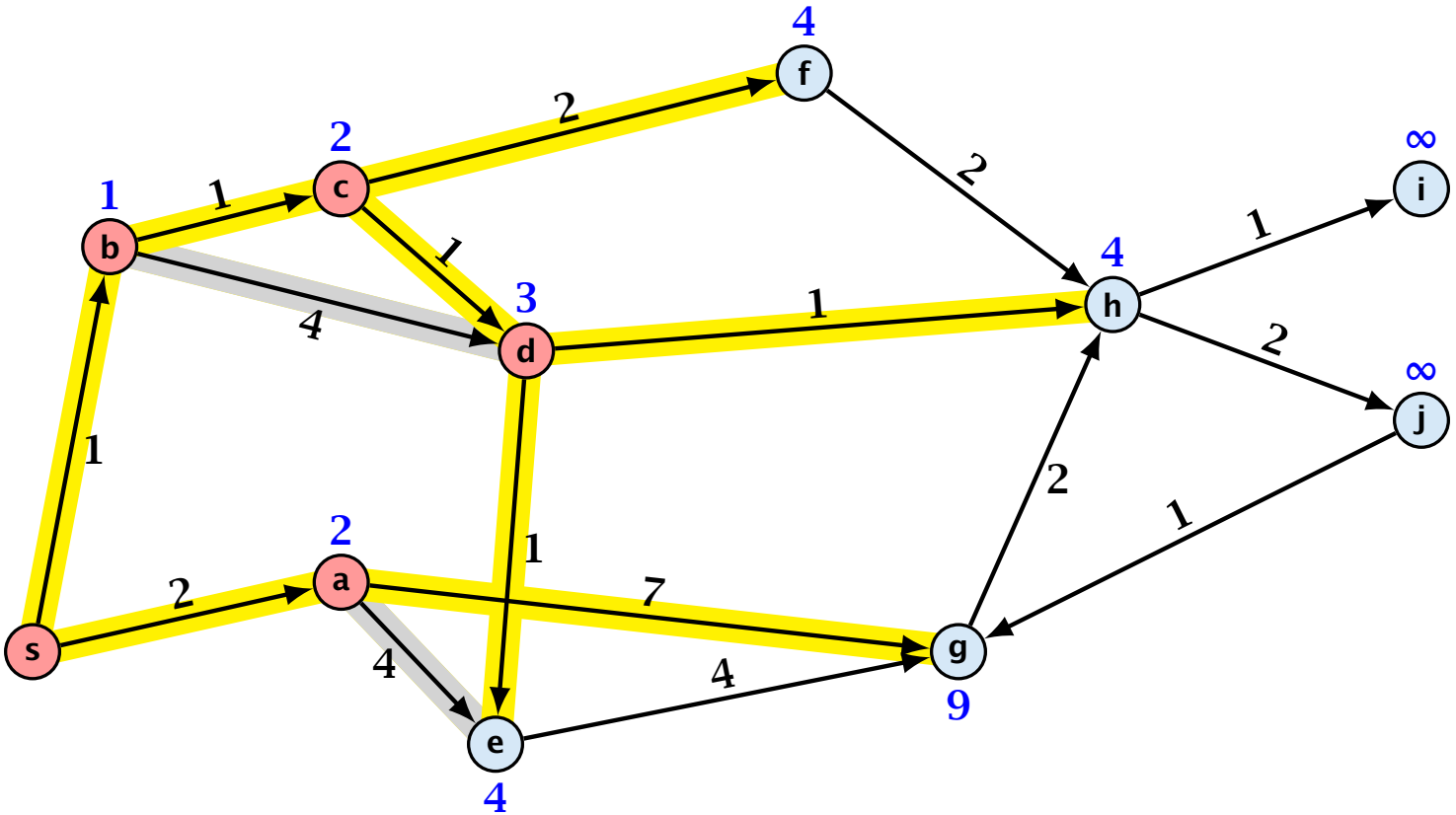
General Graphs — Nonnegative Weights



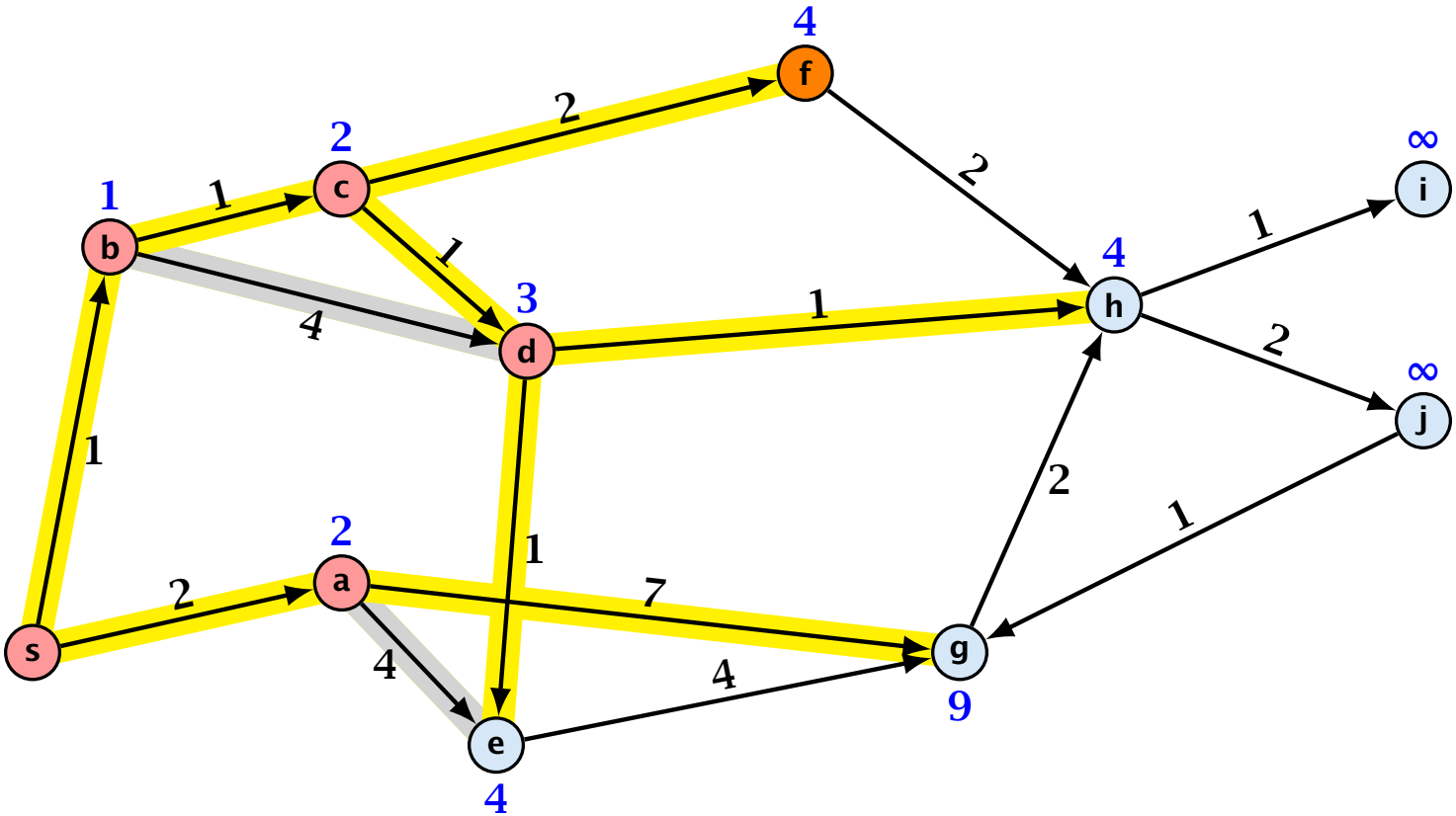
General Graphs — Nonnegative Weights



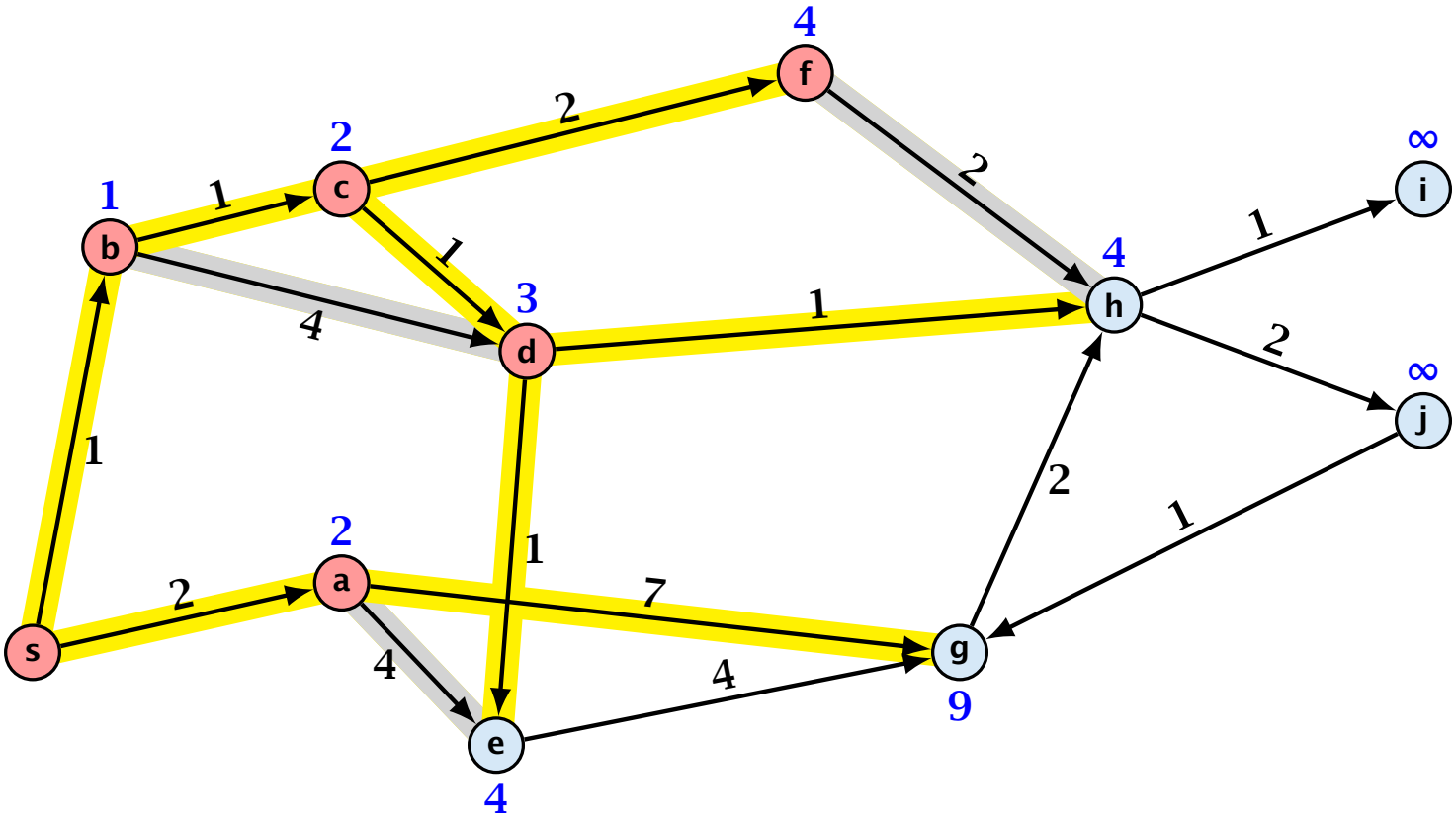
General Graphs — Nonnegative Weights



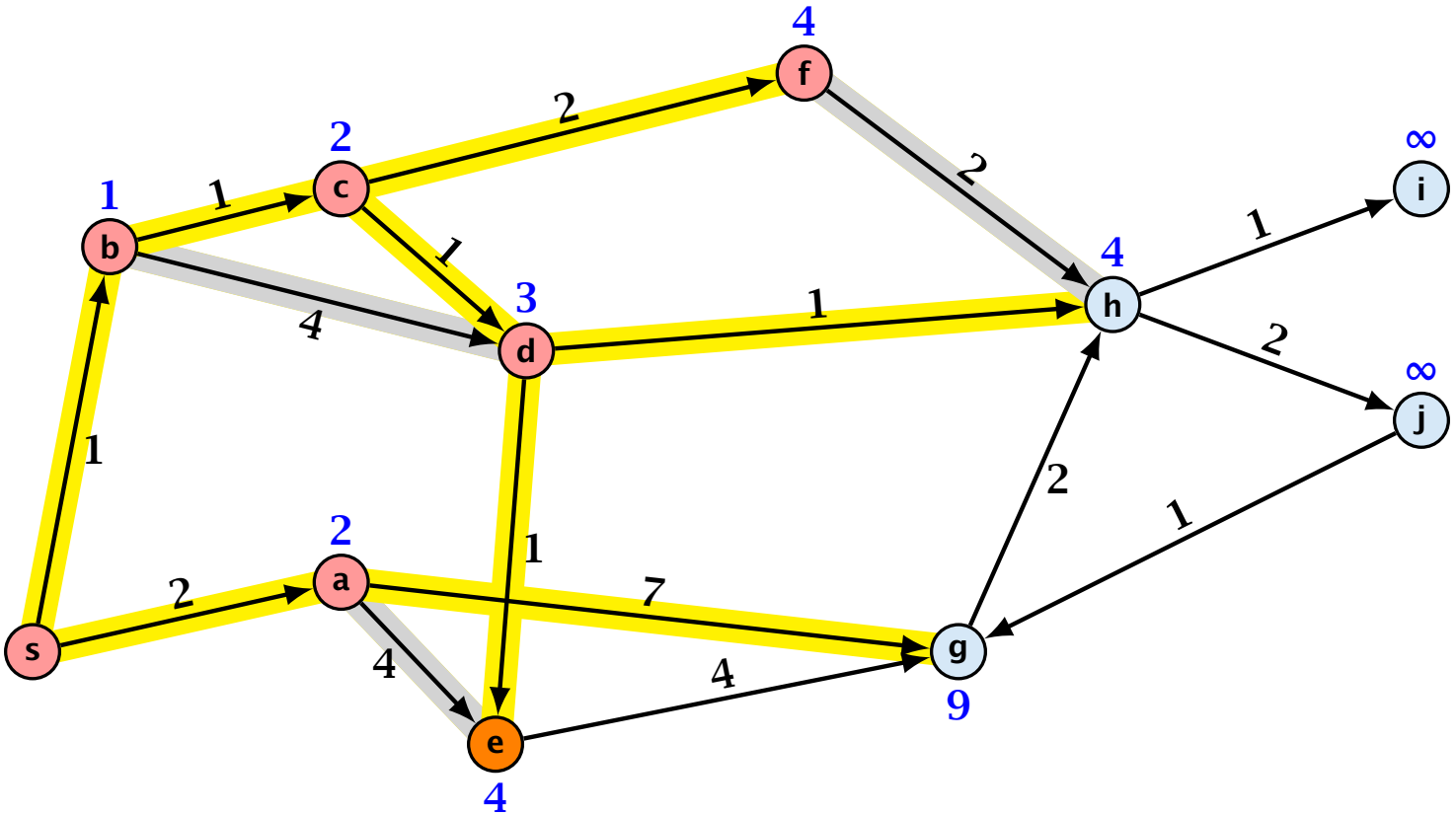
General Graphs — Nonnegative Weights



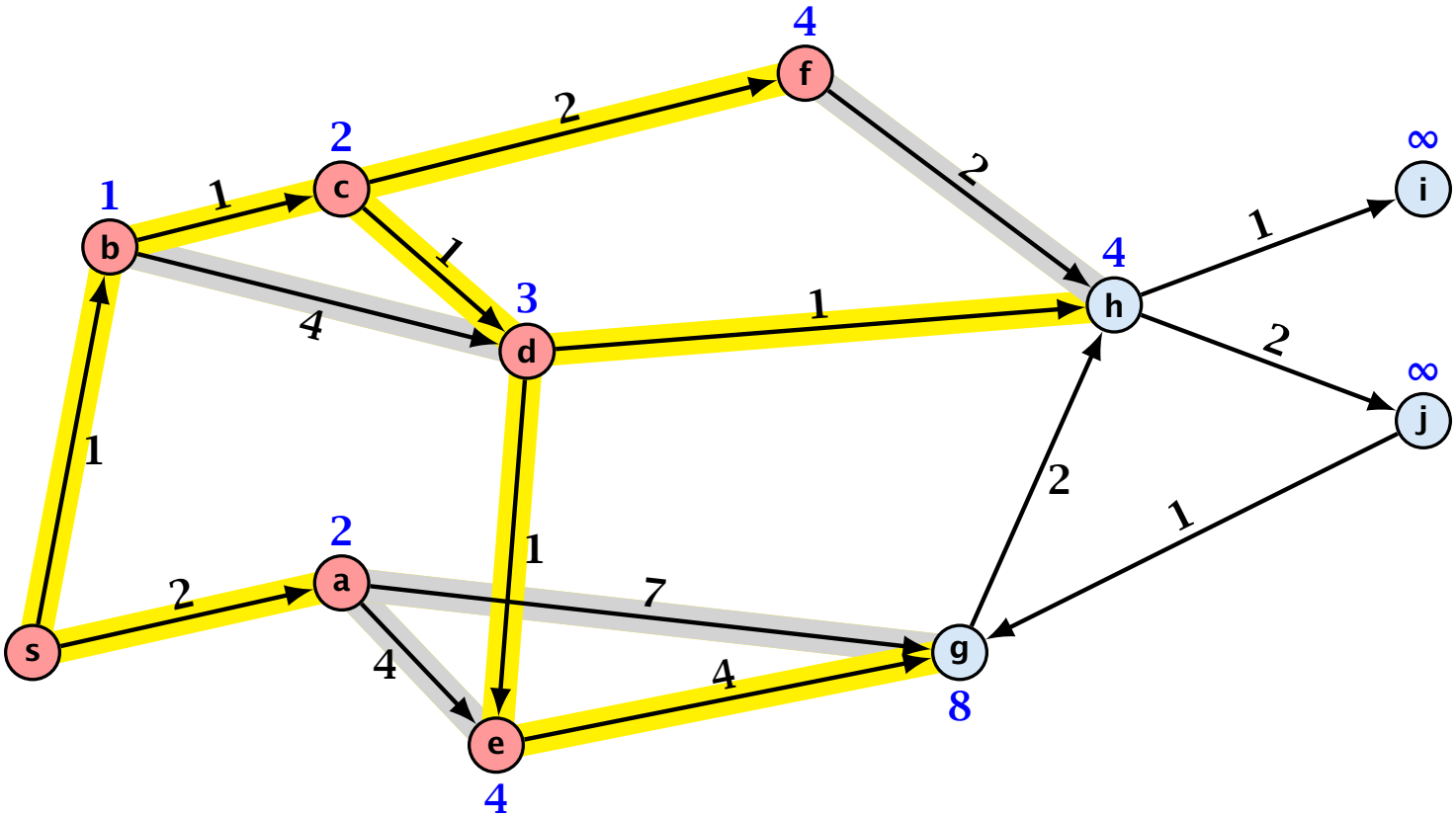
General Graphs — Nonnegative Weights



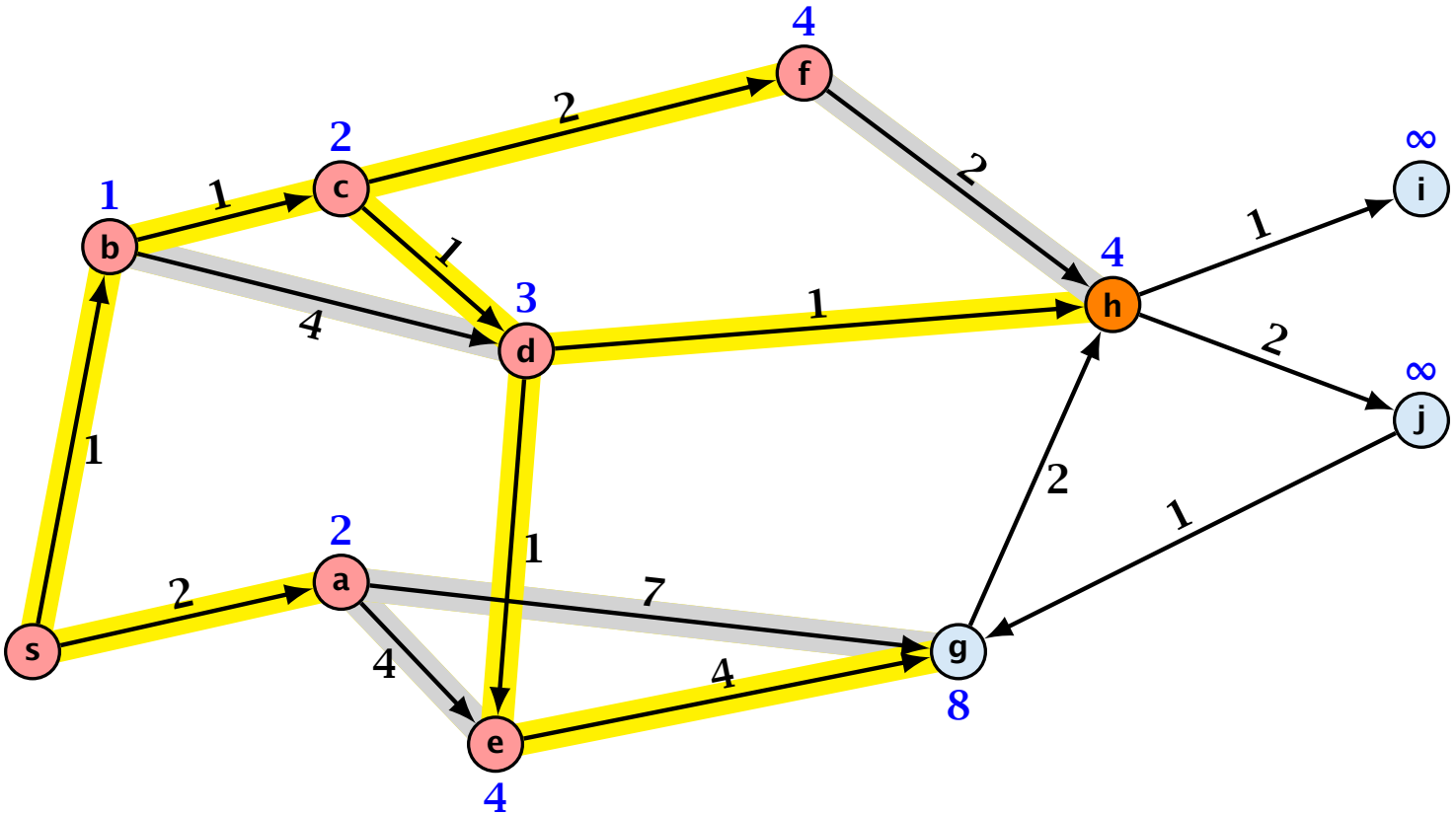
General Graphs — Nonnegative Weights



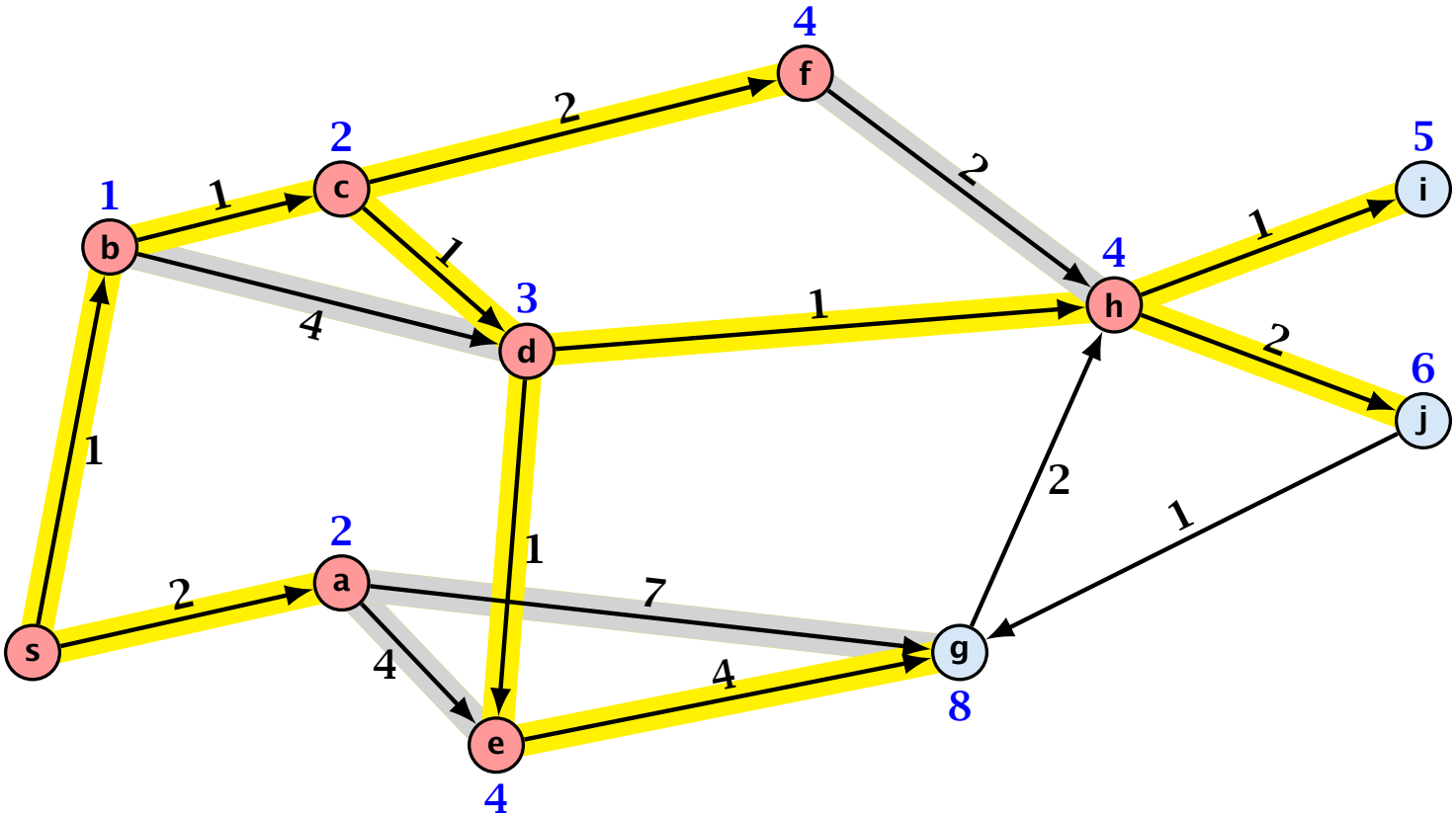
General Graphs — Nonnegative Weights



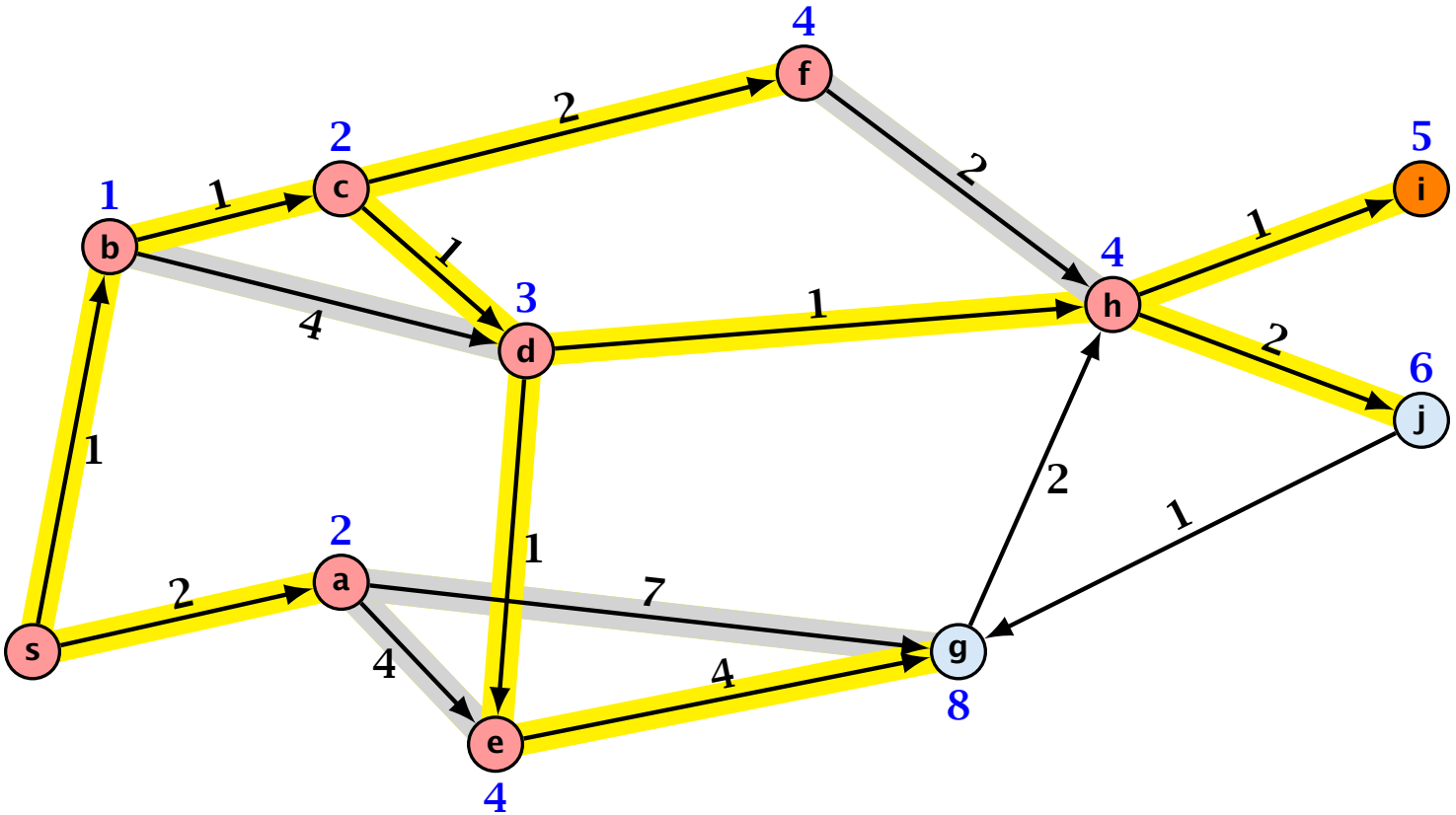
General Graphs — Nonnegative Weights



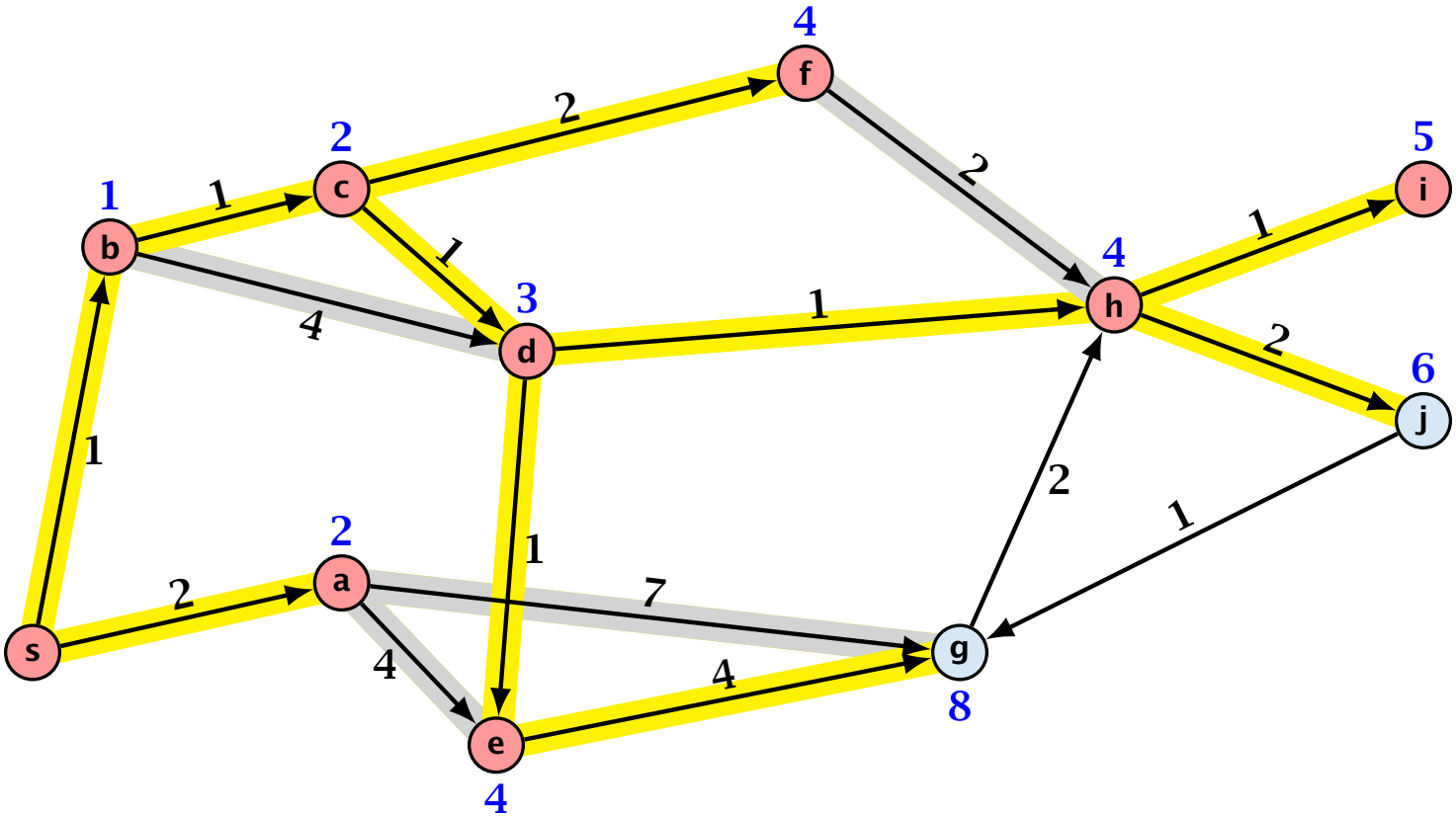
General Graphs — Nonnegative Weights



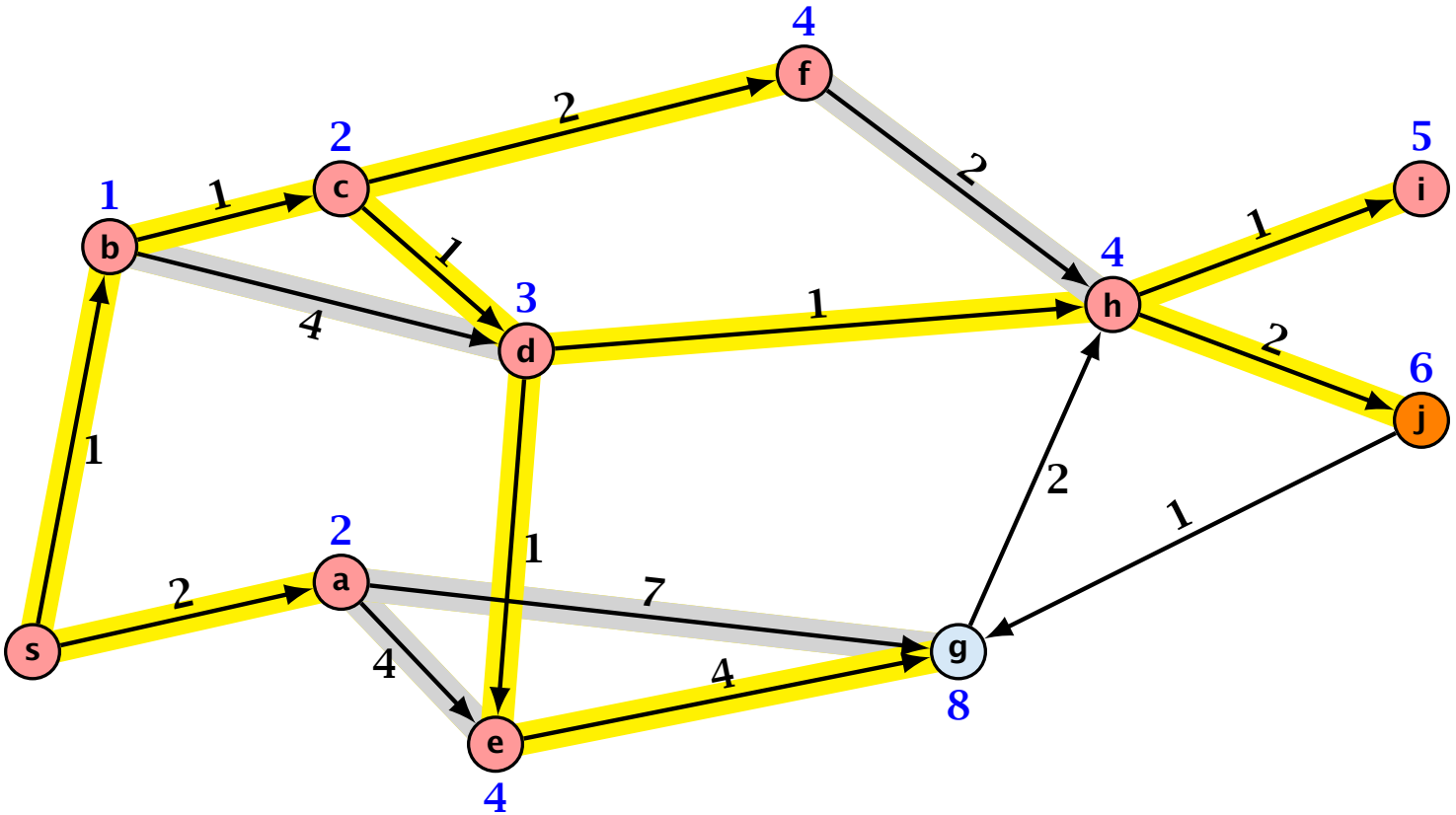
General Graphs — Nonnegative Weights



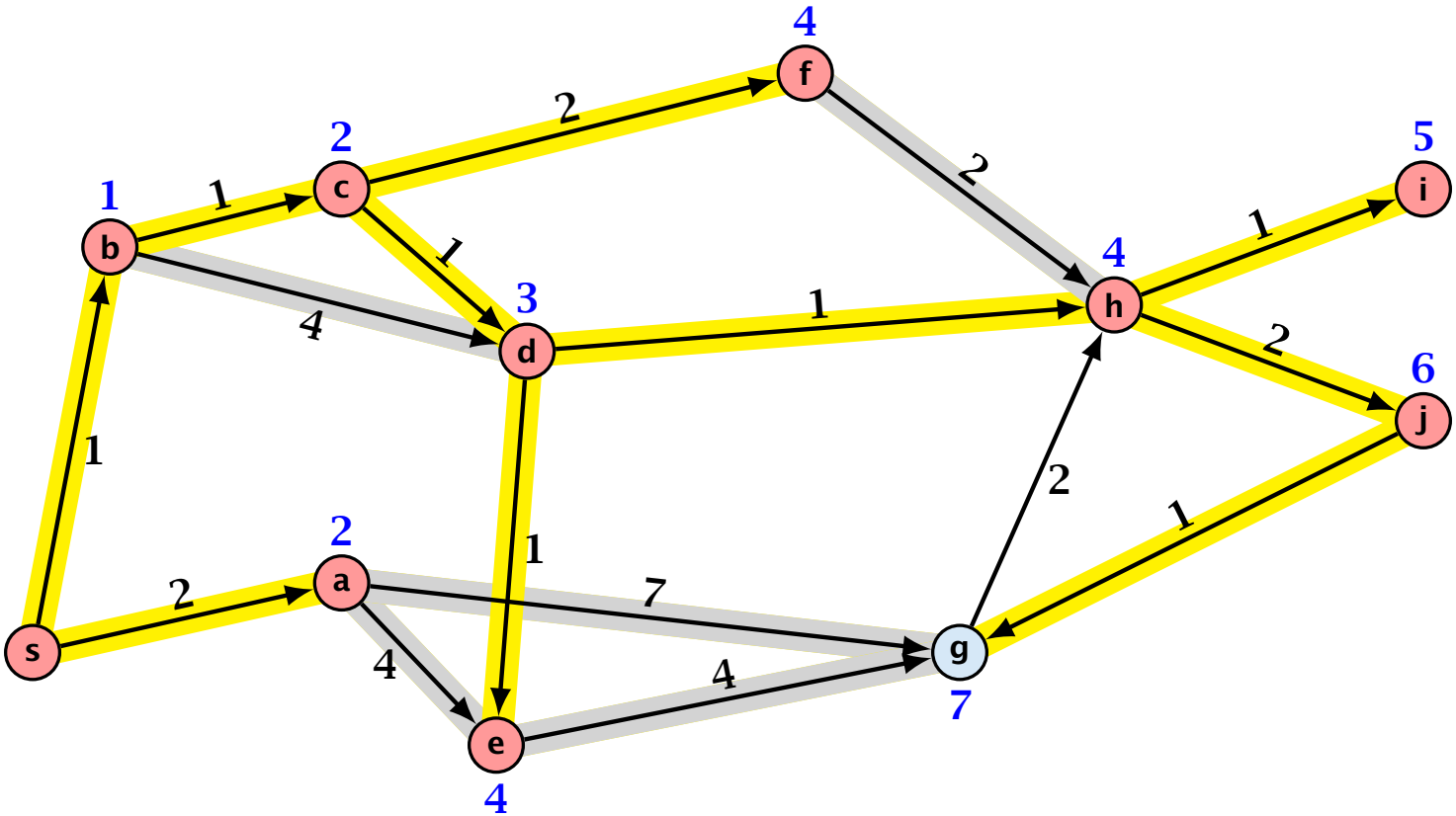
General Graphs — Nonnegative Weights



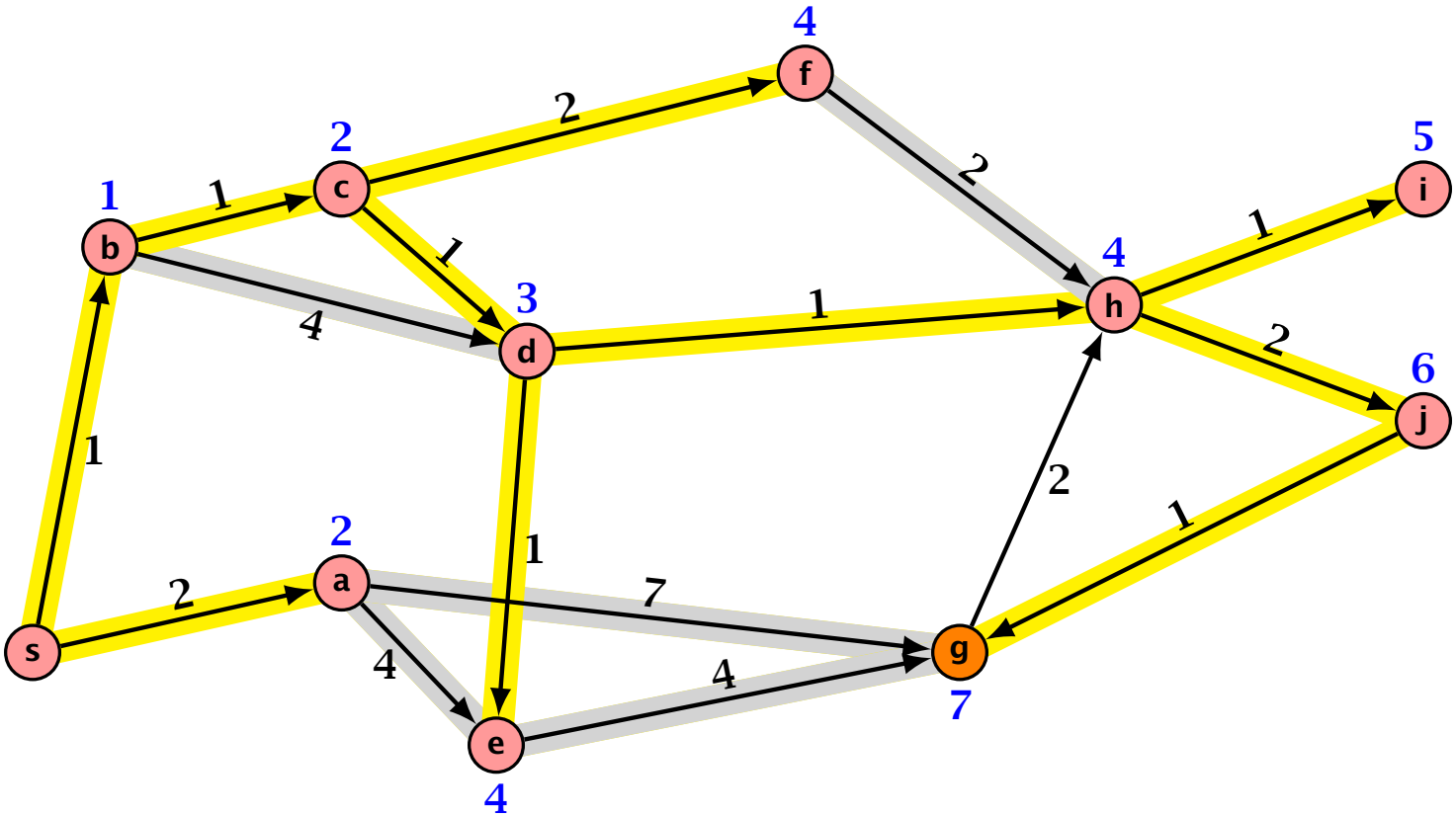
General Graphs — Nonnegative Weights



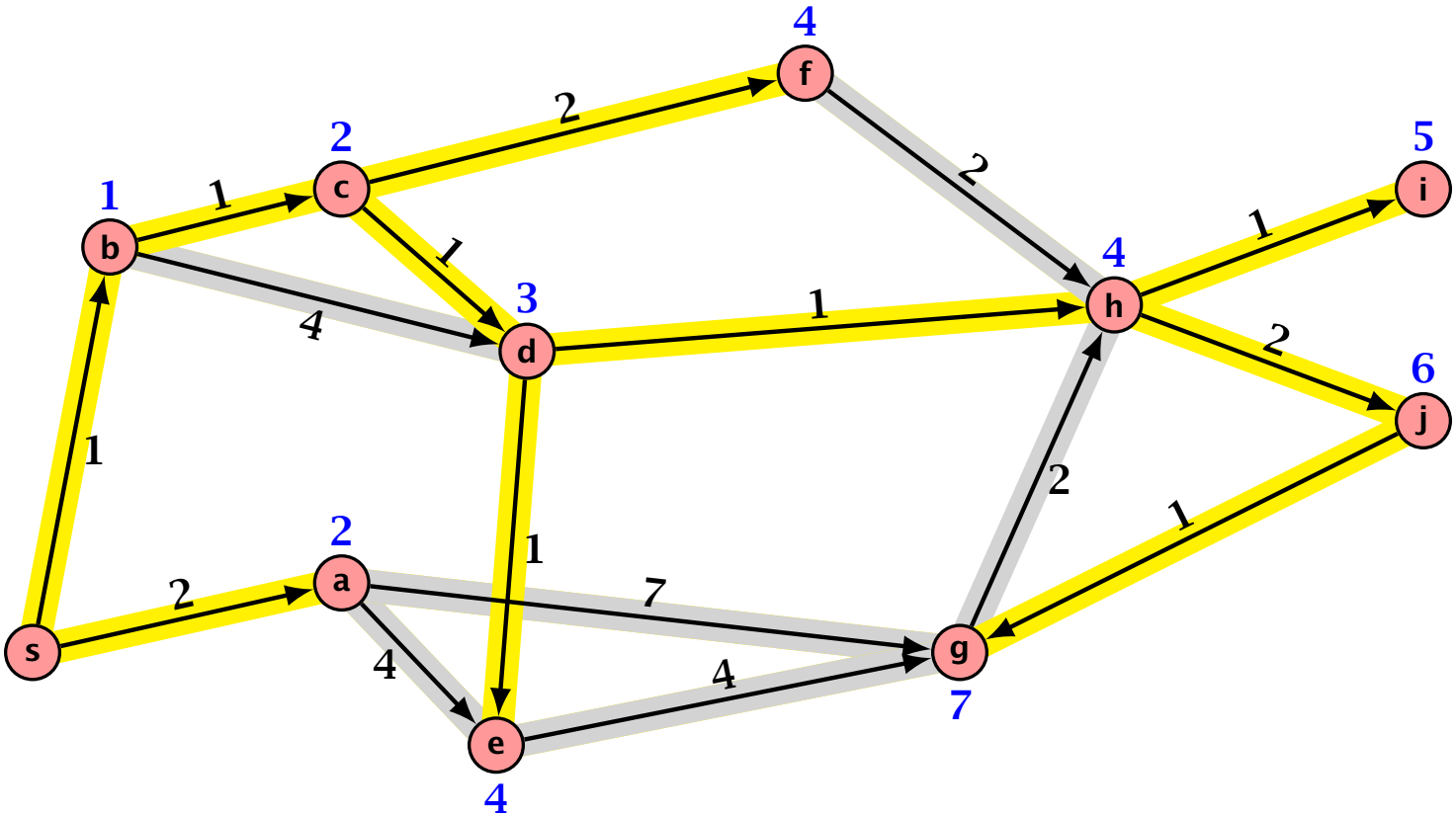
General Graphs — Nonnegative Weights



General Graphs — Nonnegative Weights



General Graphs — Nonnegative Weights



Why does this work?

Invariant:
Let A denote set of vertices already removed from the priority queue.

- **A** The distance label $x.\text{key}$ fulfills $x.\text{key} \leq w(P)$ for all s-x path P that **either** end in edge (a,x), $a \in A$ **or** are empty (consist only of vertex s).
- **B** Vertices in A have their correct distance.

In addition, we use the fact that $x.\text{key}$ is always an upper bound on the length of a shortest s-x path (this follows since we are only performing edge relaxations).

Initialization:

A Initially, A is empty; hence s is the only path allowed. Therefore, giving s a distance of 0 and all other vertices infinity is correct.

B Nothing to prove as A is empty.

Initialization:

A Initially, A is empty; hence s is the only path allowed. Therefore, giving s a distance of 0 and all other vertices infinity is correct.

B Nothing to prove as A is empty.

Maintenance:

B We take the vertex v that has smallest key-value. A path P from s to v must have length at least $v.\text{key}$.

This holds because already by the time P **first** leaves A (to vertex x) the length is at least $x.\text{key} \geq v.\text{key}$ (by Invariant A for $x.\text{key}$). It cannot become shorter due to nonnegative weights.

Hence, the distance-label for v is correct (it is an upper bound on the length of a shortest s-v path and no path is shorter).

This gives that Invariant B holds for $A' = A \cup \{v\}$.

A Since, v has correct distance, after relaxing all edges of the form (v,x) a distance label $x.\text{key} \leq w(P)$ for all paths that end in edge (v,x).

As by invariant we have $x.\text{key} \leq w(P)$ for paths that end in (a,x), $a \in A$ or are empty we get Invariant A for $A' = A \cup \{v\}$.

Running time: m decrease-key operations, n insert-operations, n delete-min operations

Running time: $\mathcal{O}((m+n)\log n)$.

A running time of $\mathcal{O}(m + n\log n)$ can be obtained by not using AVL-trees as priority queue but by using Fibonacci Heaps.

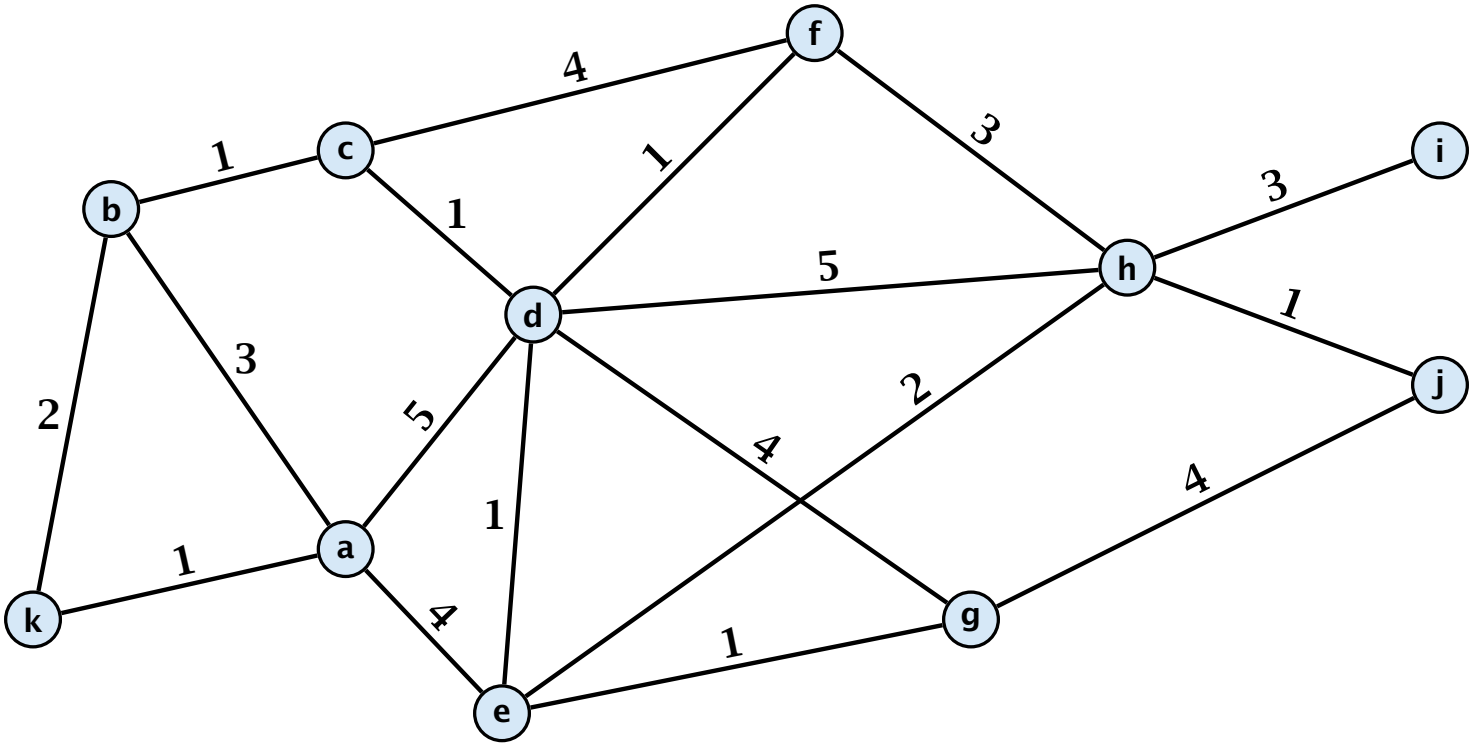
Running time: m decrease-key operations, n insert-operations, n delete-min operations

Running time: $\mathcal{O}((m+n)\log n)$.

A running time of $\mathcal{O}(m + n\log n)$ can be obtained by not using AVL-trees as priority queue but by using Fibonacci Heaps.

Minimum Spanning Tree

Which edges should we choose to connect all vertices?



Minimize cost!!!

Minimum Spanning Tree

Input:

- undirected connected graph $G = (V, E)$
- positive edge weights (costs) $w : E \to \mathbb{R}^+$

Output:

- subset $T \subseteq E$ s.t. $G = (V, T)$ connected and $w(T) = \sum_{e \in T} w(e)$ minimal

Observation:

- since edge-weights are positive T always forms a tree

Minimum Spanning Tree

Lemma 1

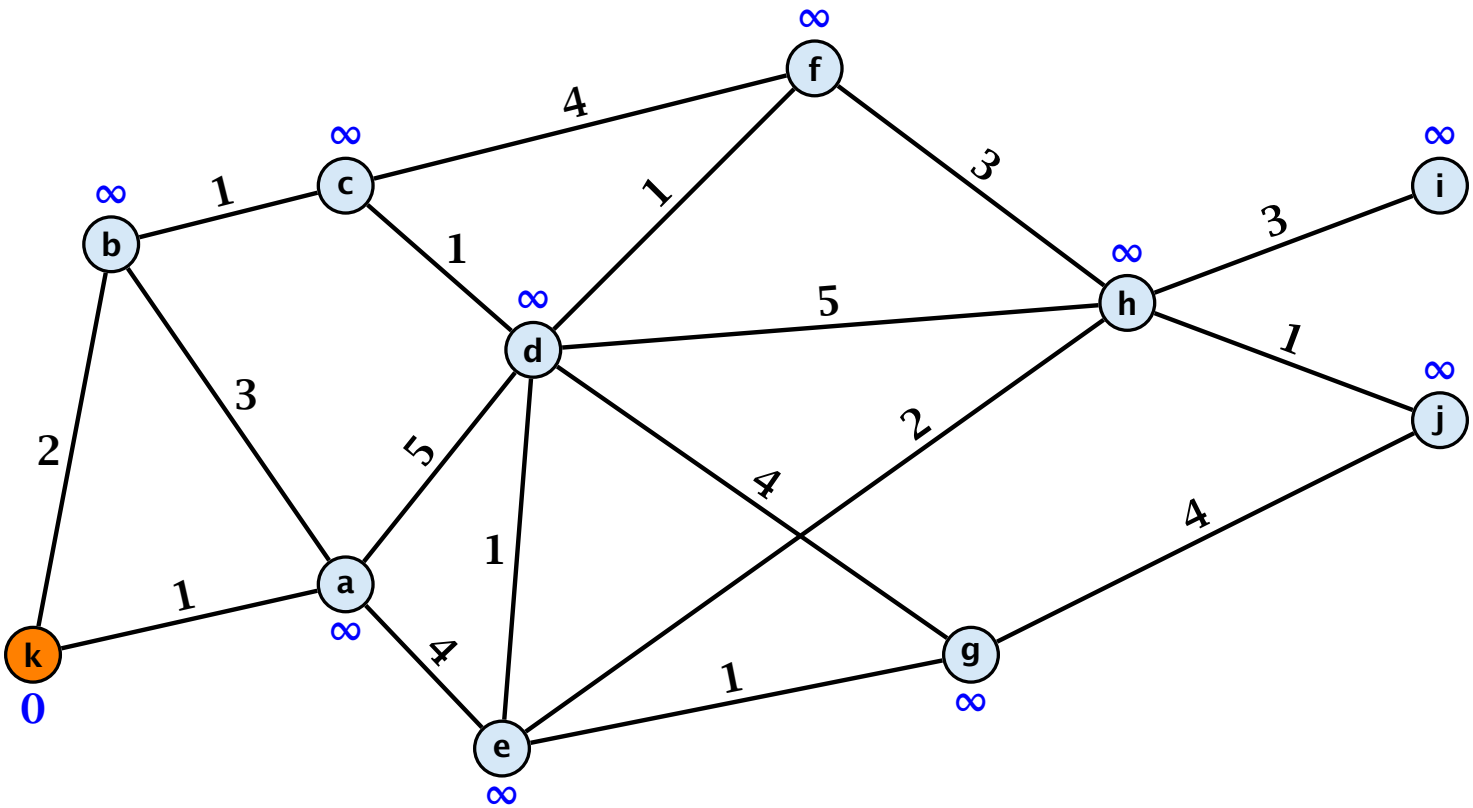
Let X, Y be a partition of V (i.e., $X \cup Y = V$ and $X \cap V = \emptyset$), and let $e = (x, y)$ be a minimum cost edge connecting a vertex from X to a vertex from Y. Then there exists an MST that contains e.

Prims MST Algorithm

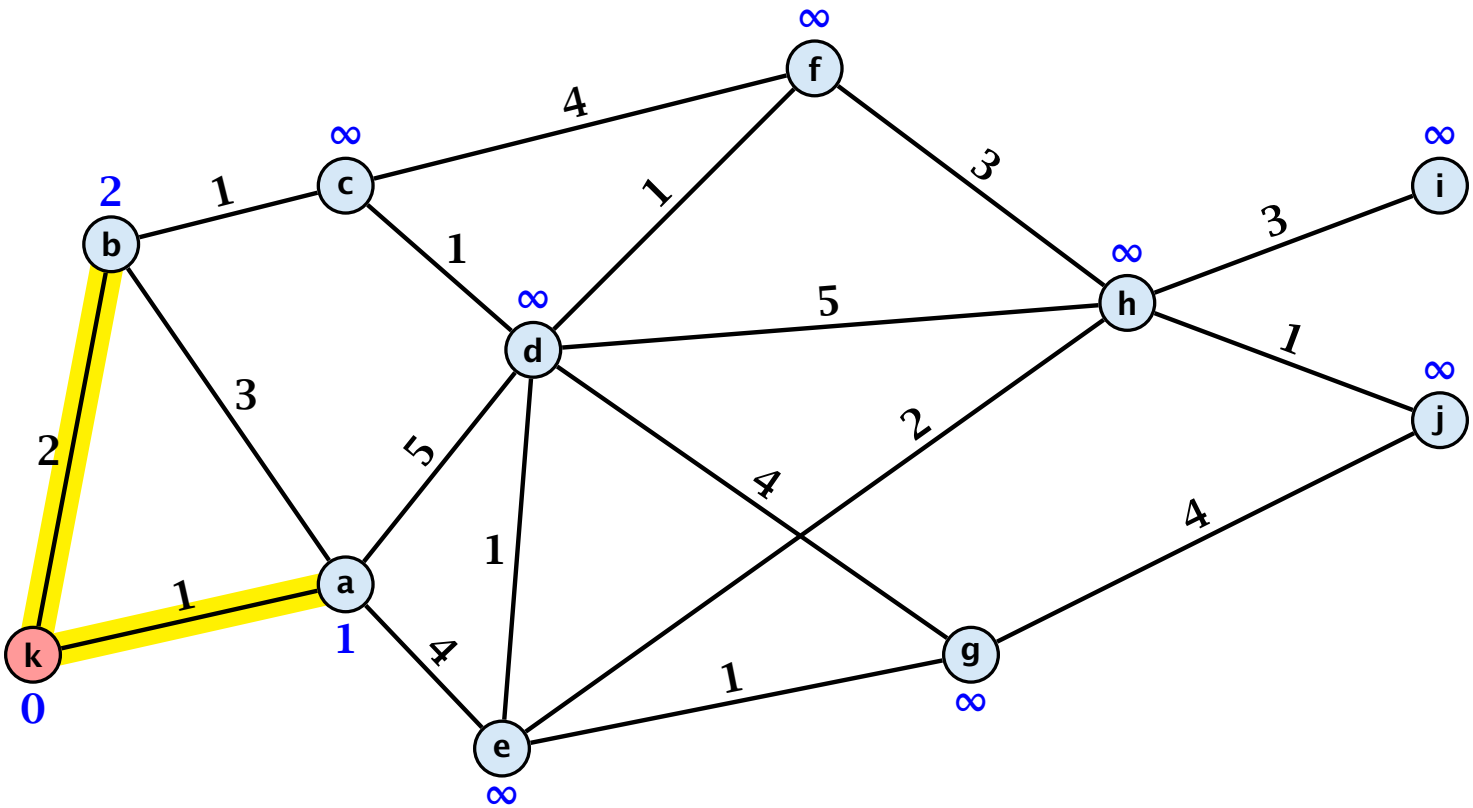
Algorithm 4 MST$(G = (V, E, w))$

```
 1: Input: weighted graph G = (V, E, w);
 2: Output: parent-field of nodes encode MST
 3: S.init(); // build empty priority queue
 4: for all v ∈ V do
 5:         v.key ← ∞;
 6:         v.parent ← ∅;
 7:         h_v ← S.insert(v);
 8: s.key ← 0; S.decrease-key(h_s,0); // s is arbitrary node
 9: while S.is-empty() = false do
10:         v ← S.delete-min();
11:         for all x ∈ V s.t. (v, x) ∈ E do
12:             if x.key > v.key + w(v, x) then
13:                 S.decrease-key(h_x, v.key + w(v, x));
14:                 x.key ← v.key + w(v, x);
```

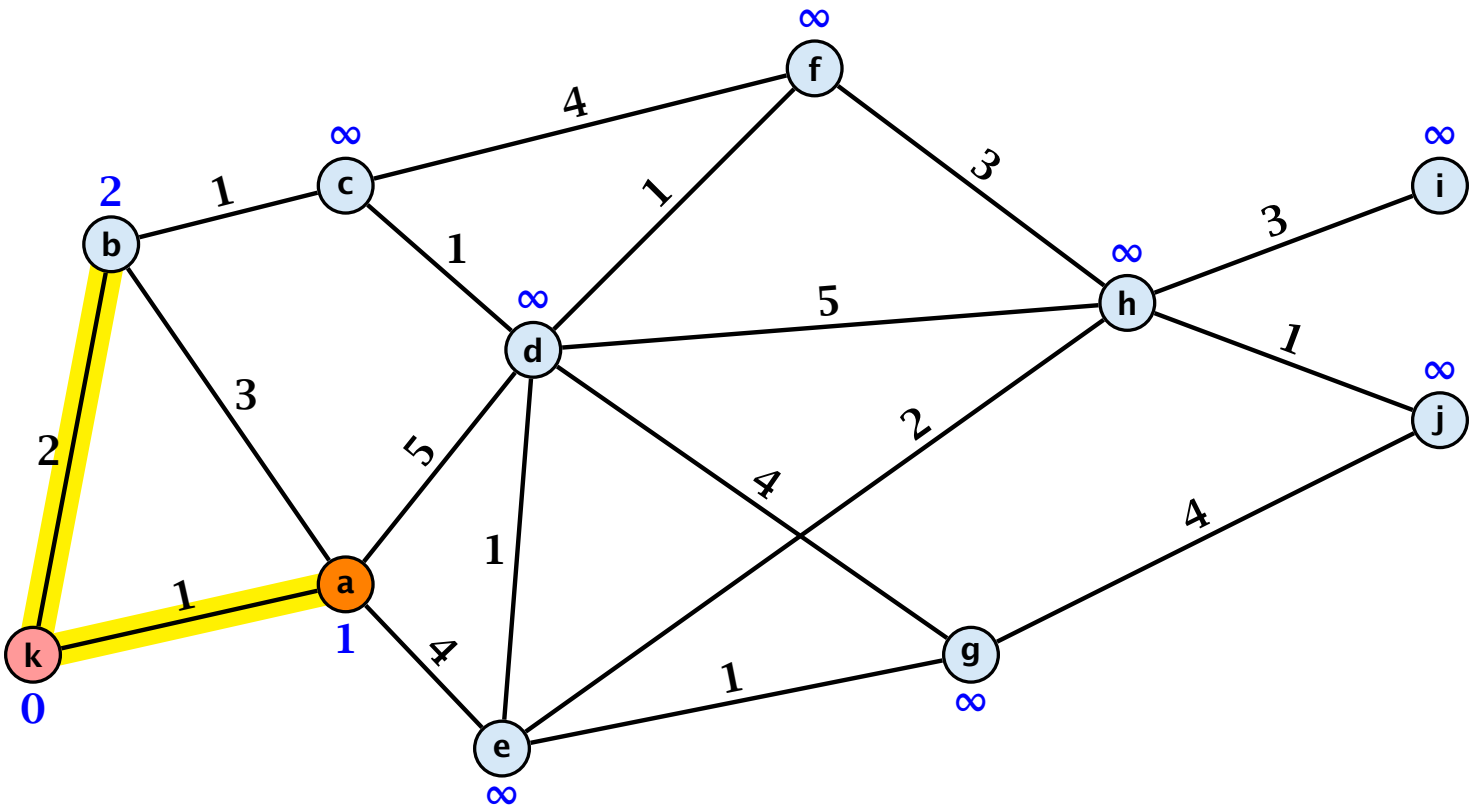
Minimum Spanning Tree



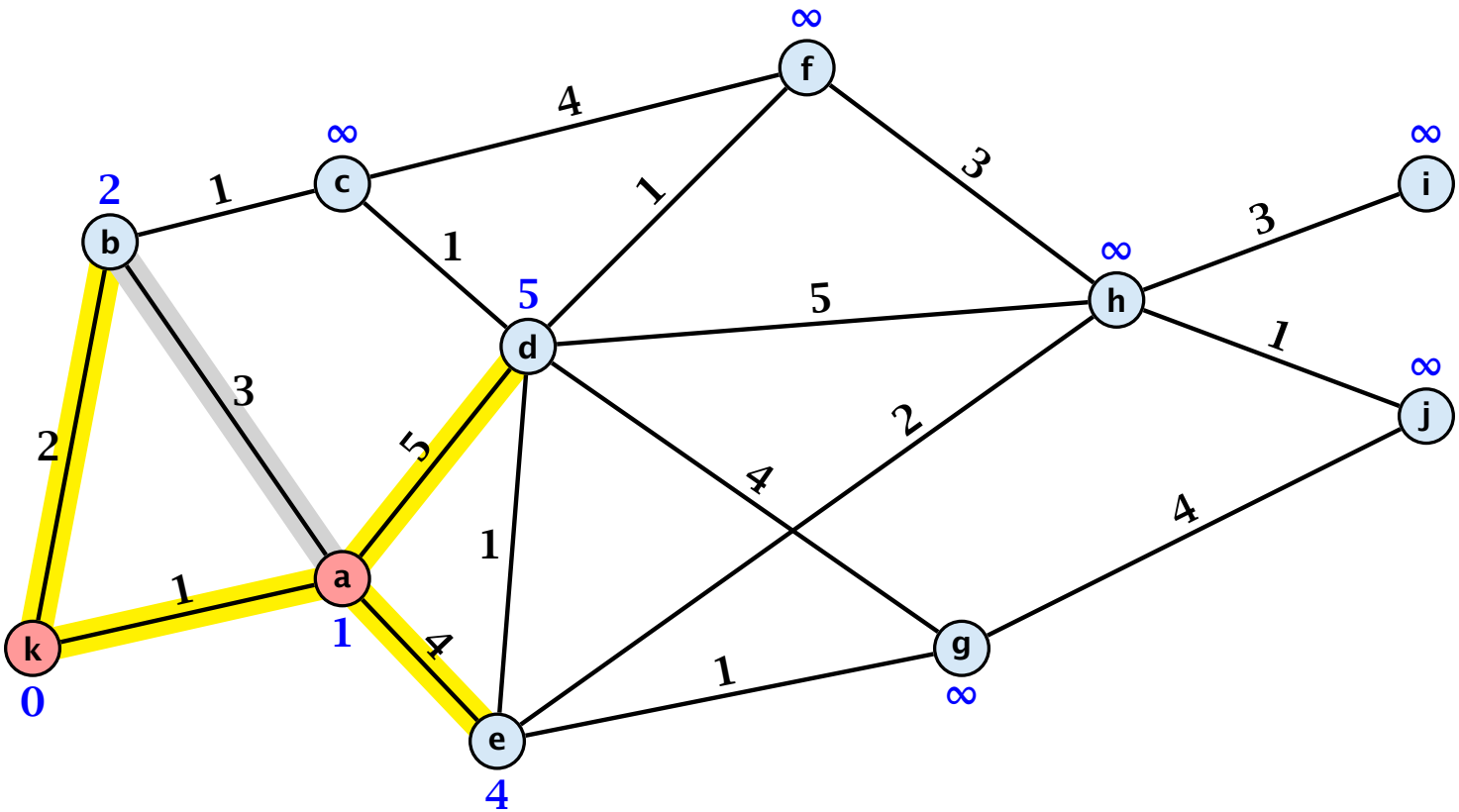
Minimum Spanning Tree



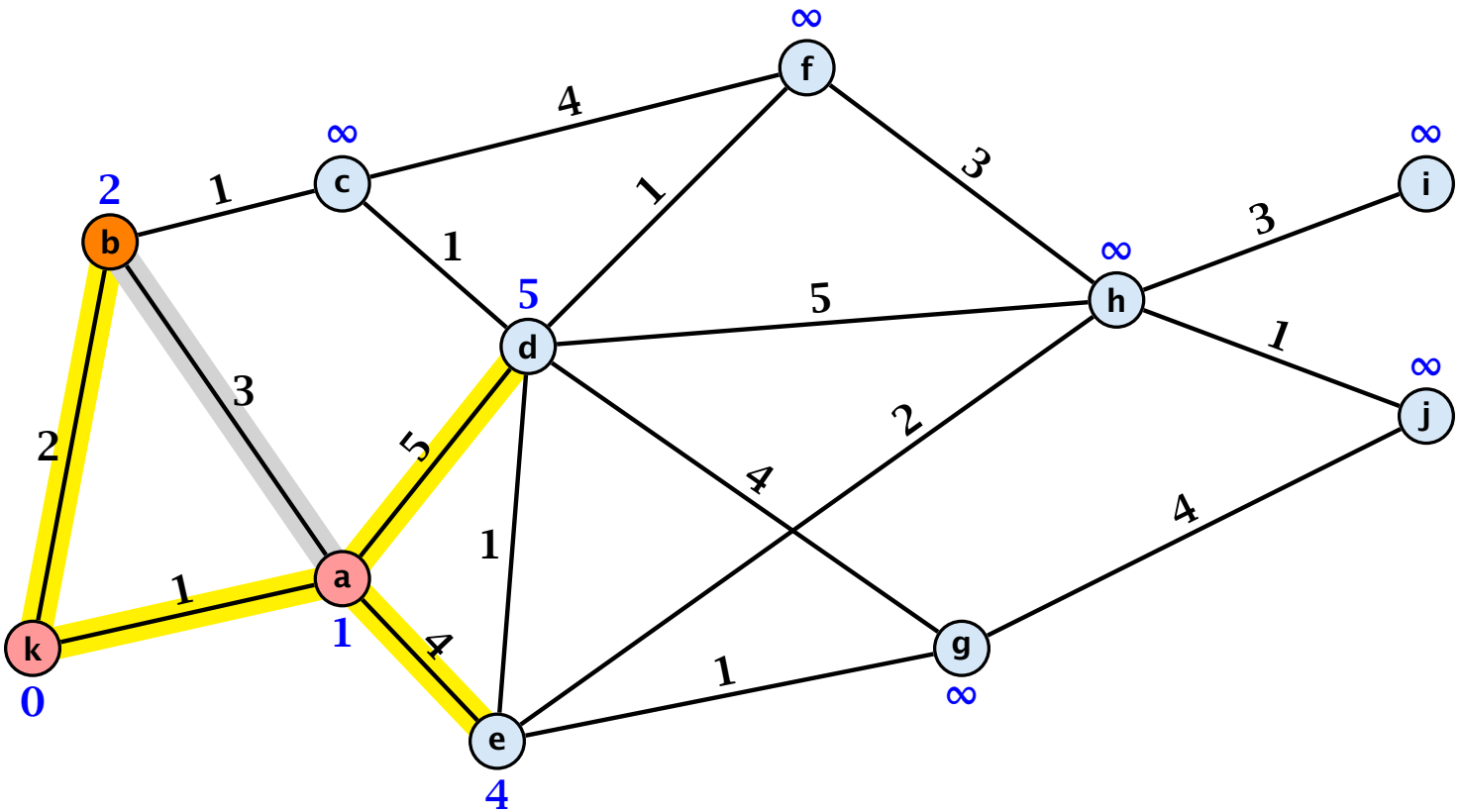
Minimum Spanning Tree



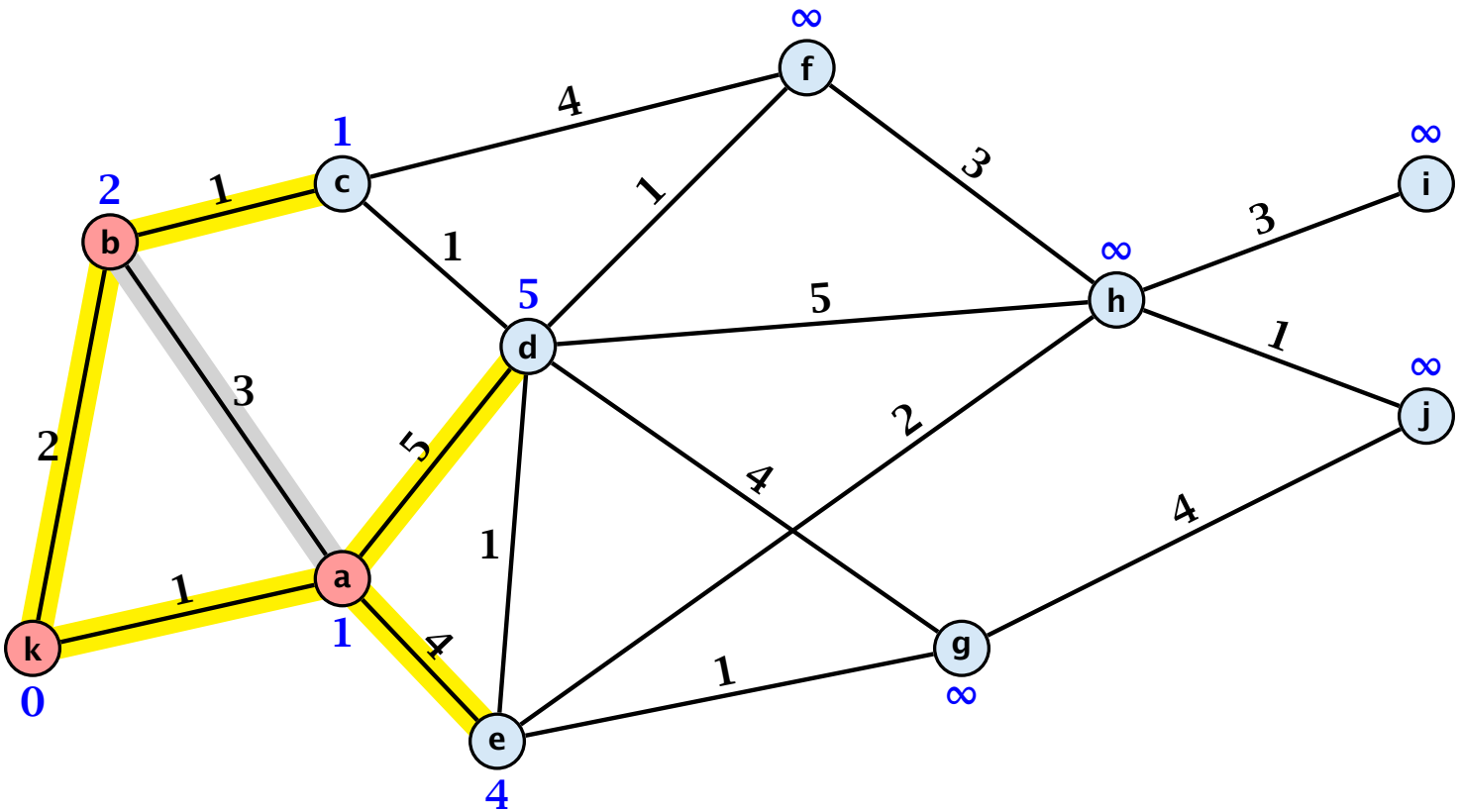
Minimum Spanning Tree



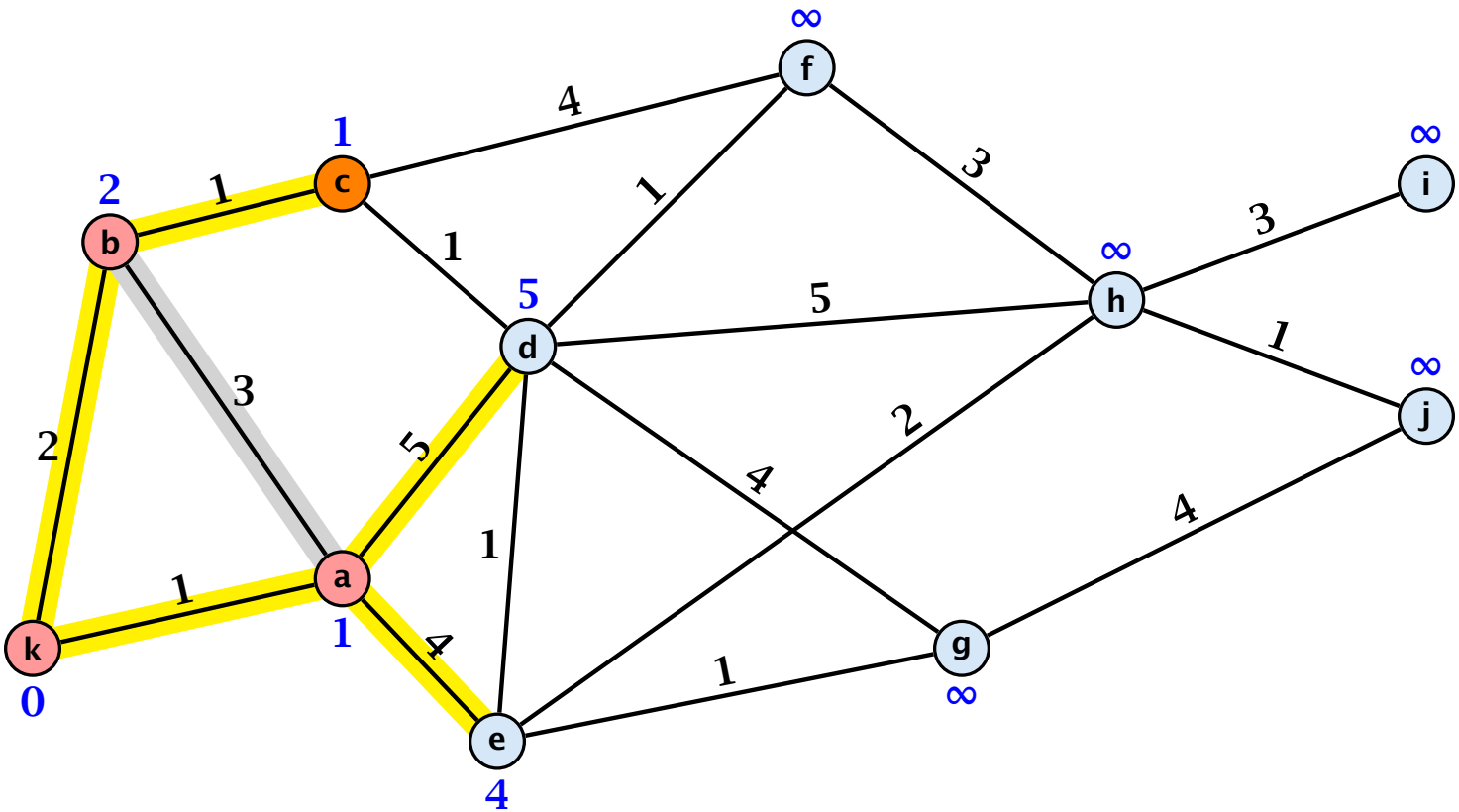
Minimum Spanning Tree



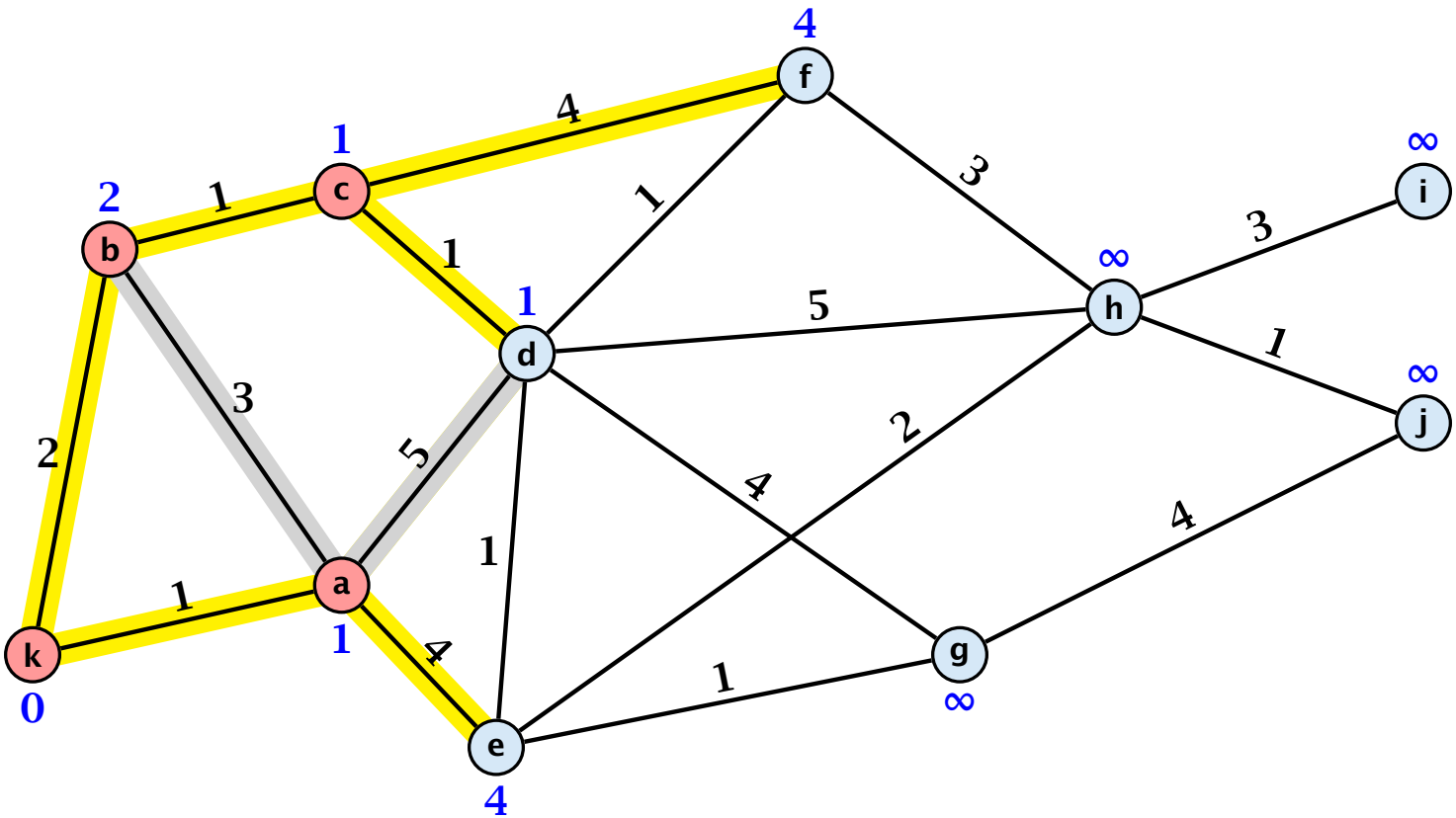
Minimum Spanning Tree



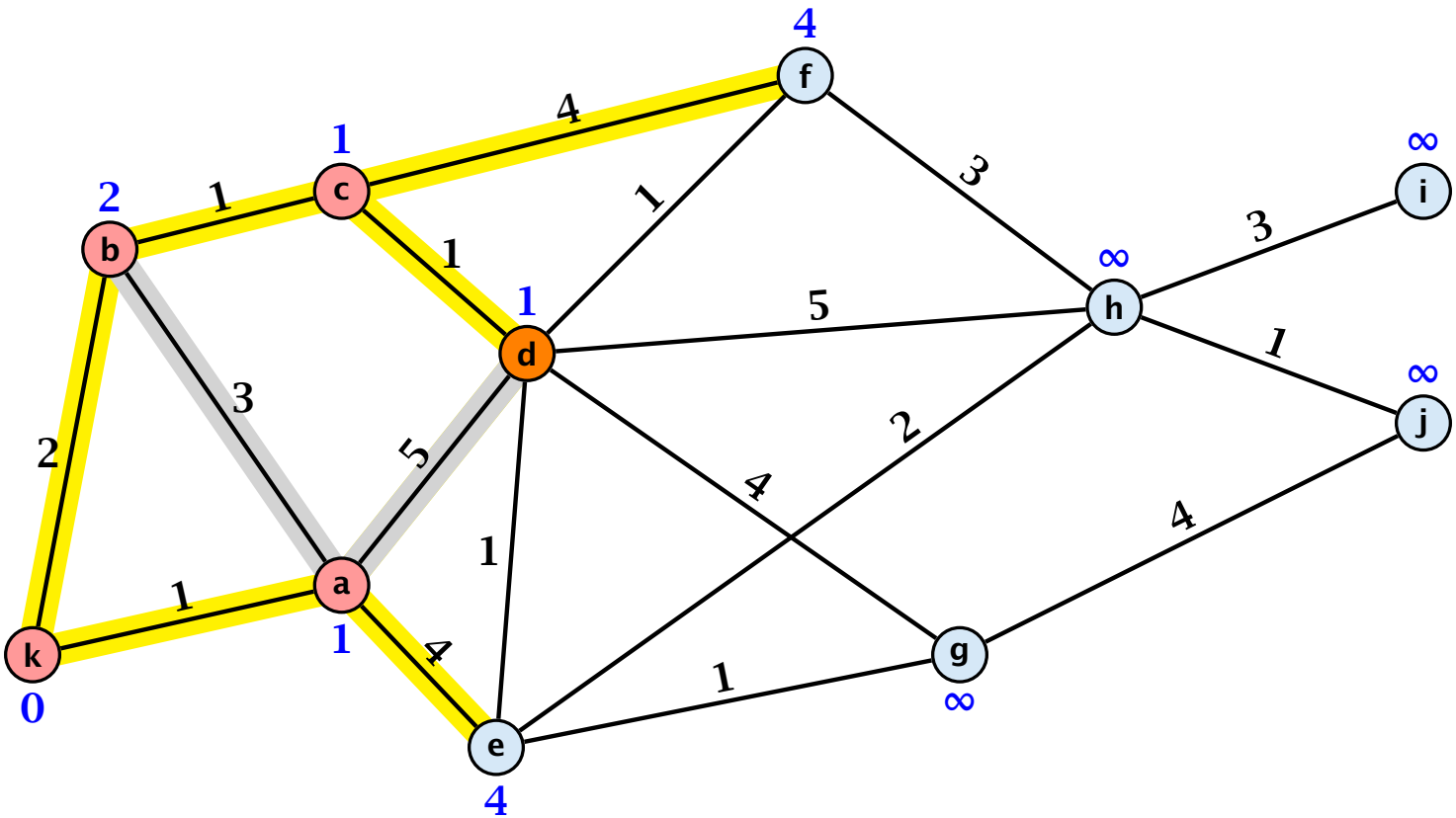
Minimum Spanning Tree



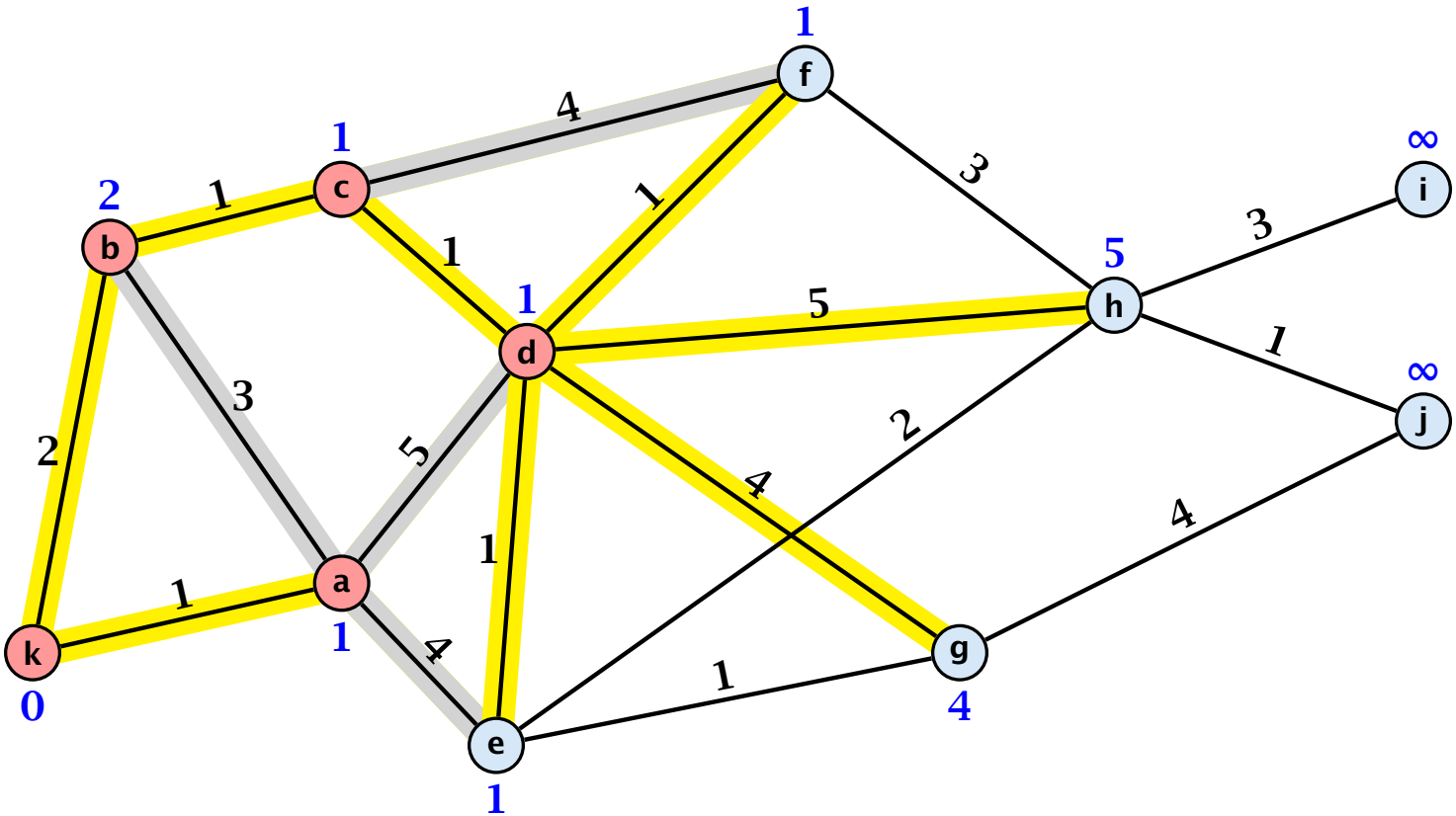
Minimum Spanning Tree



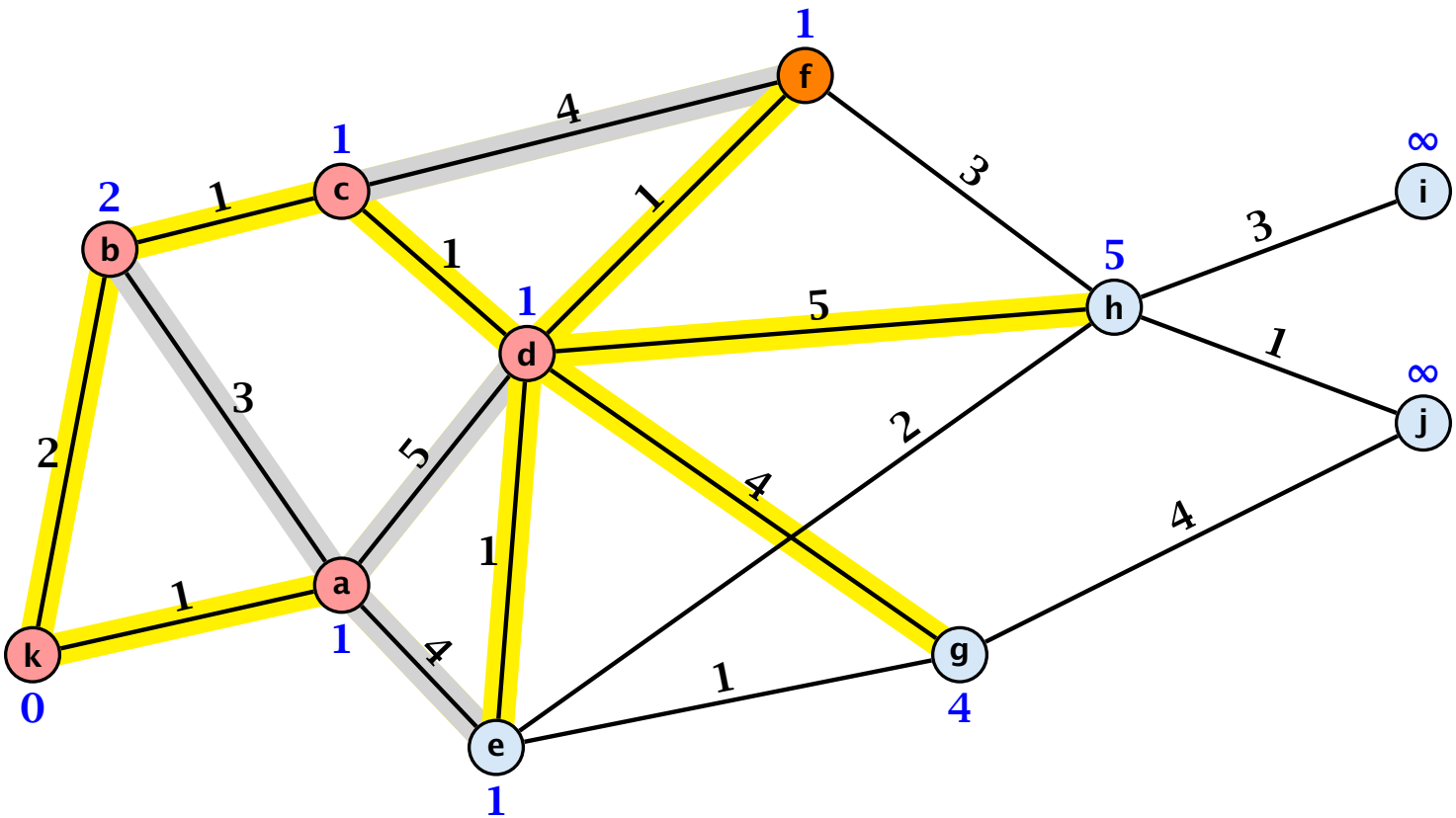
Minimum Spanning Tree



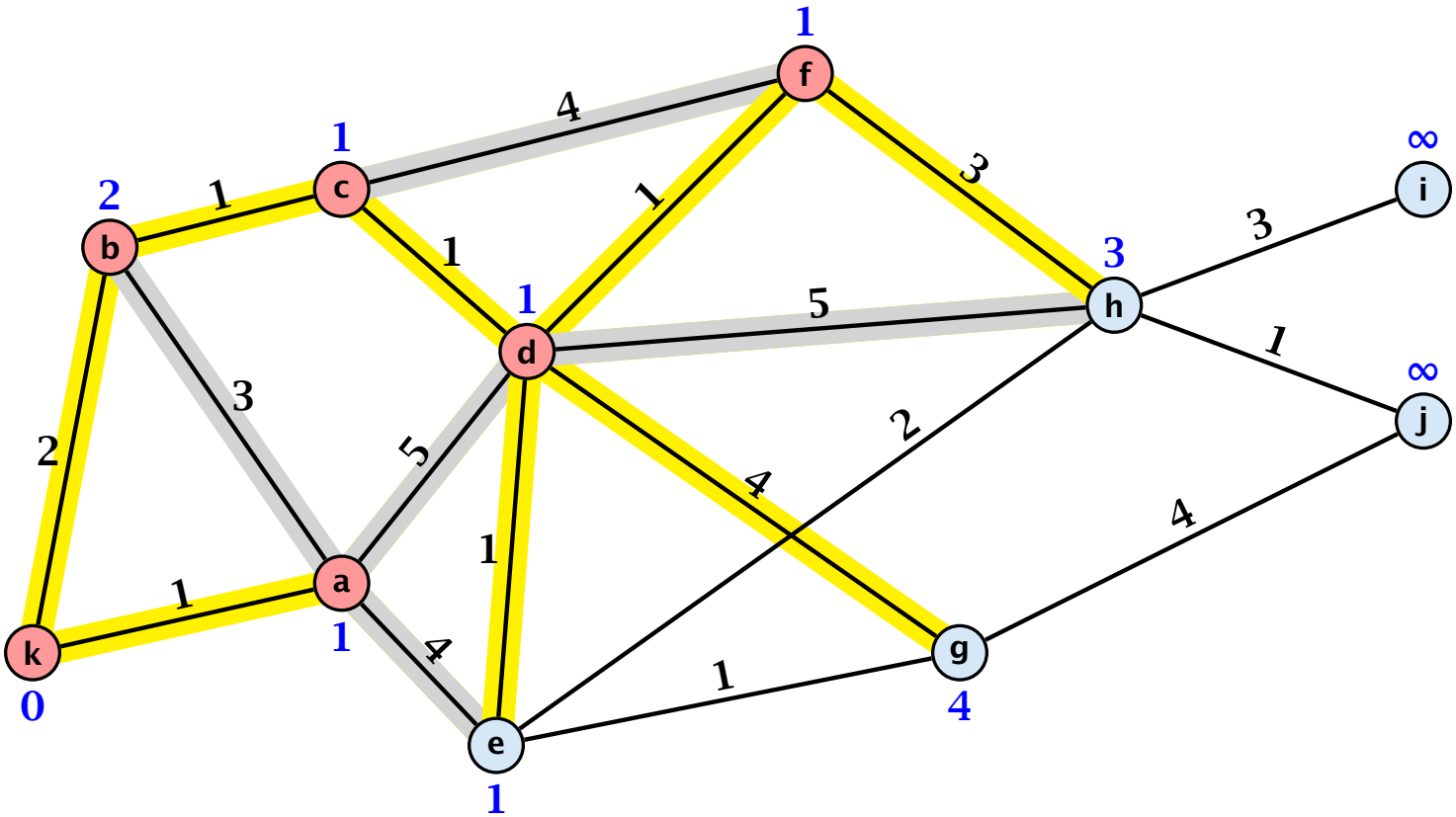
Minimum Spanning Tree



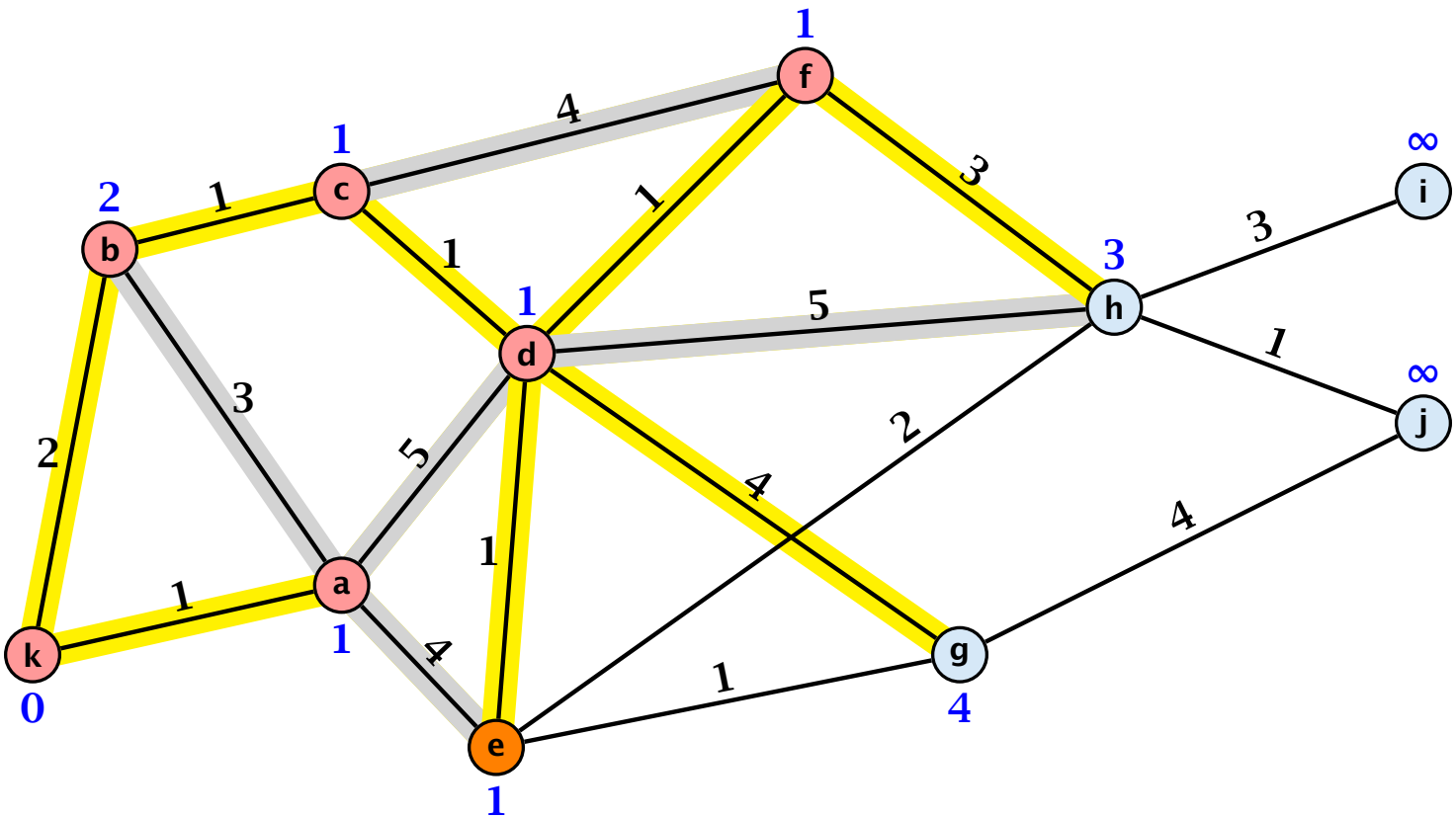
Minimum Spanning Tree



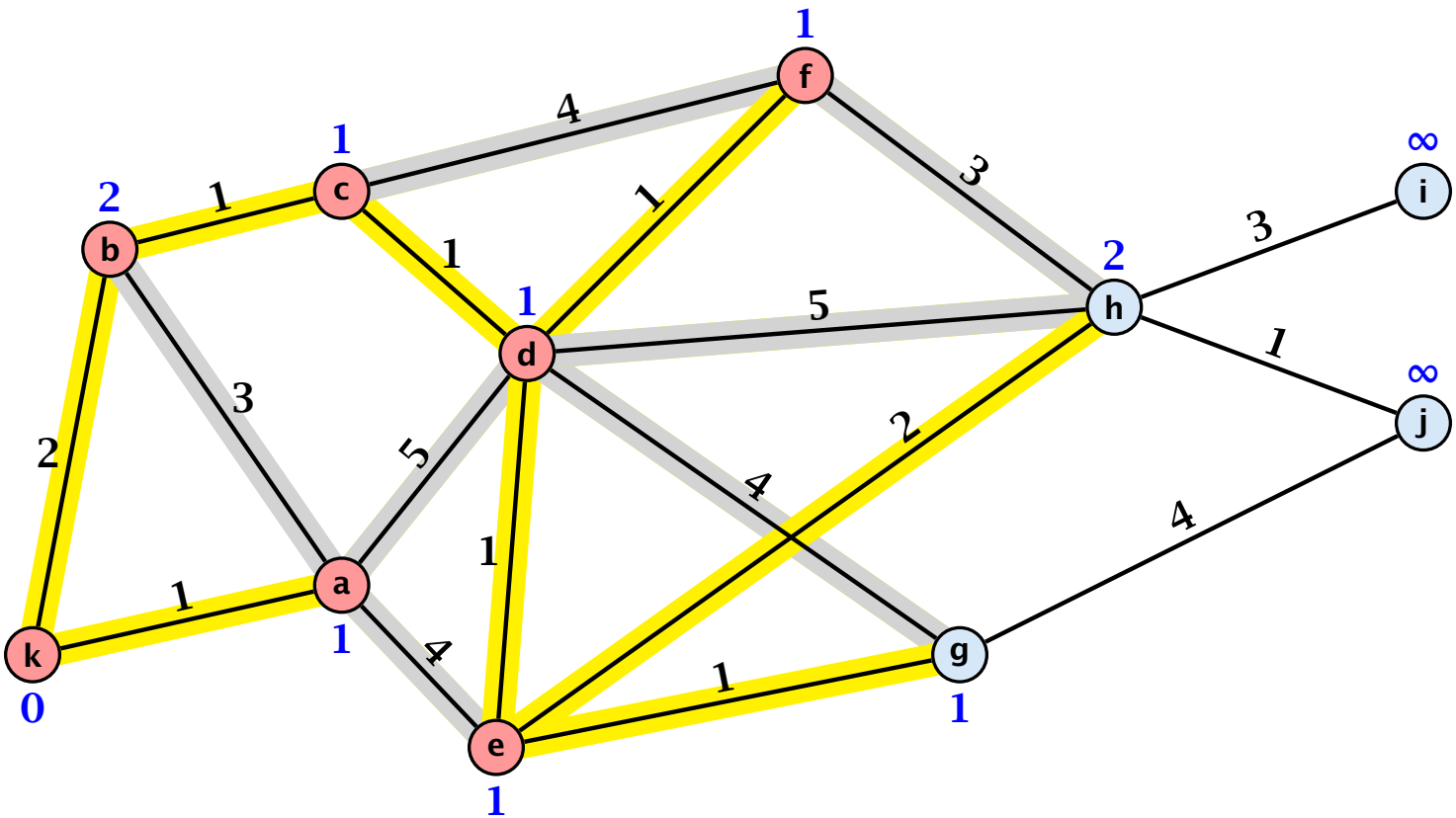
Minimum Spanning Tree



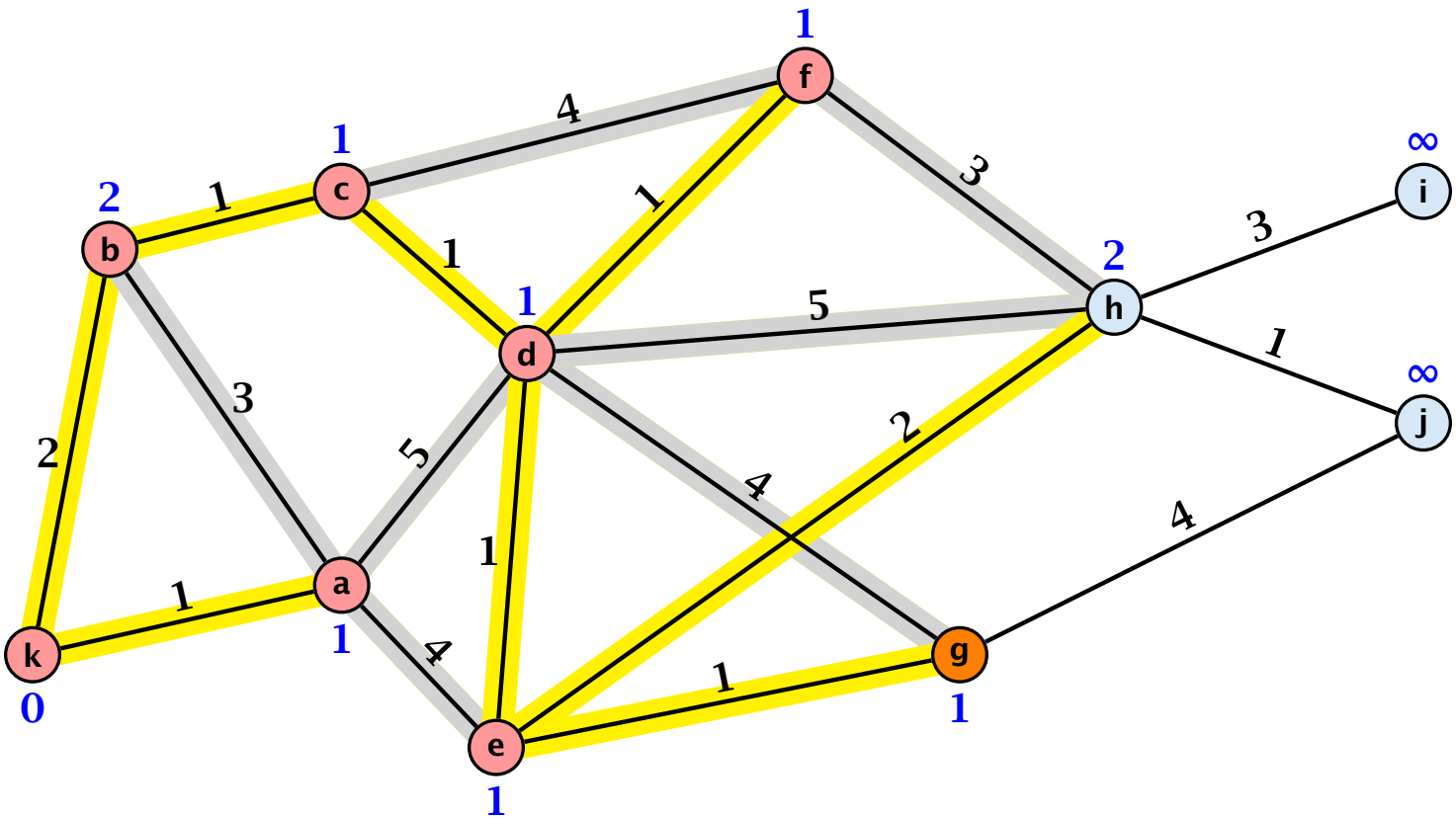
Minimum Spanning Tree



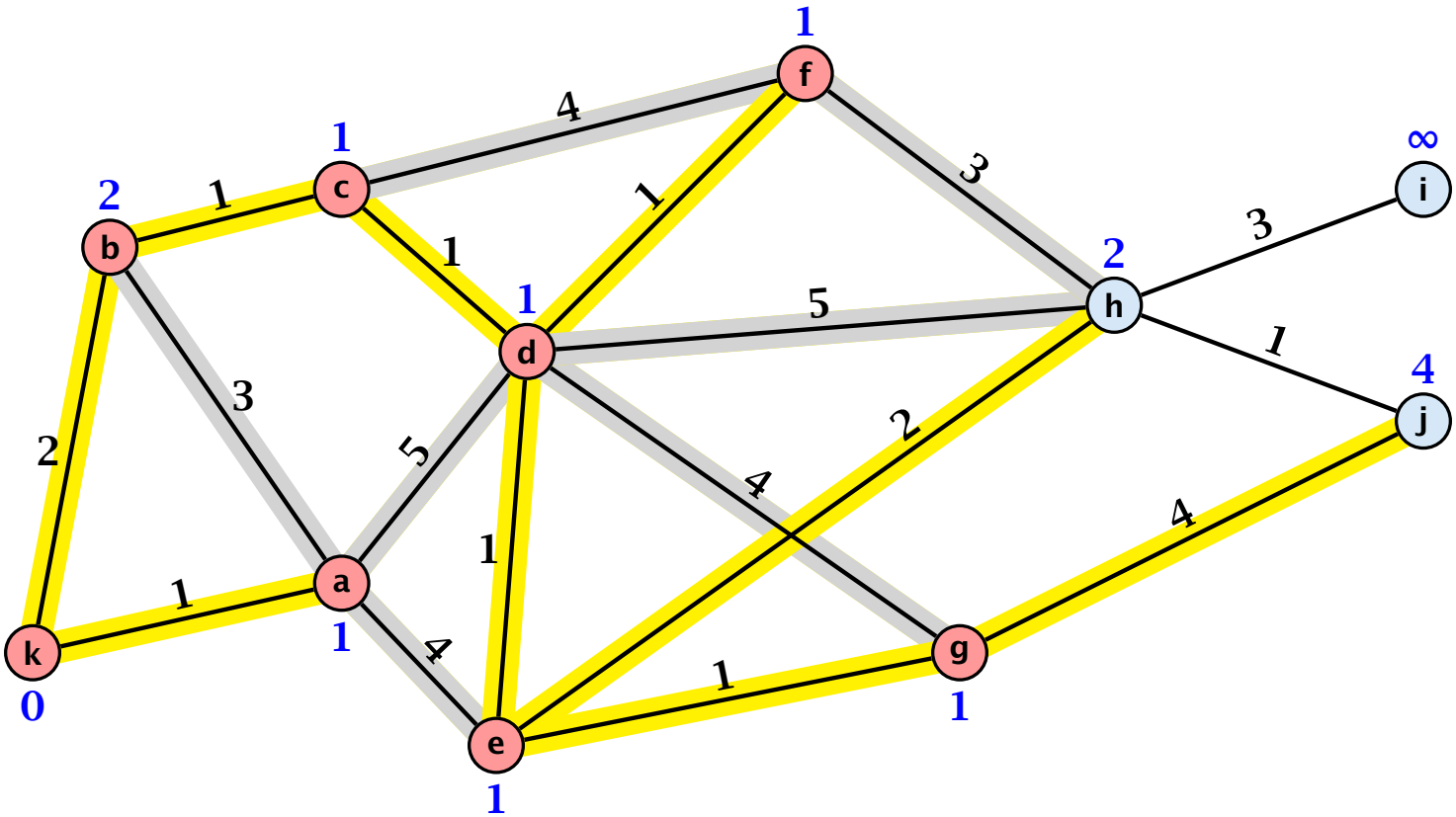
Minimum Spanning Tree



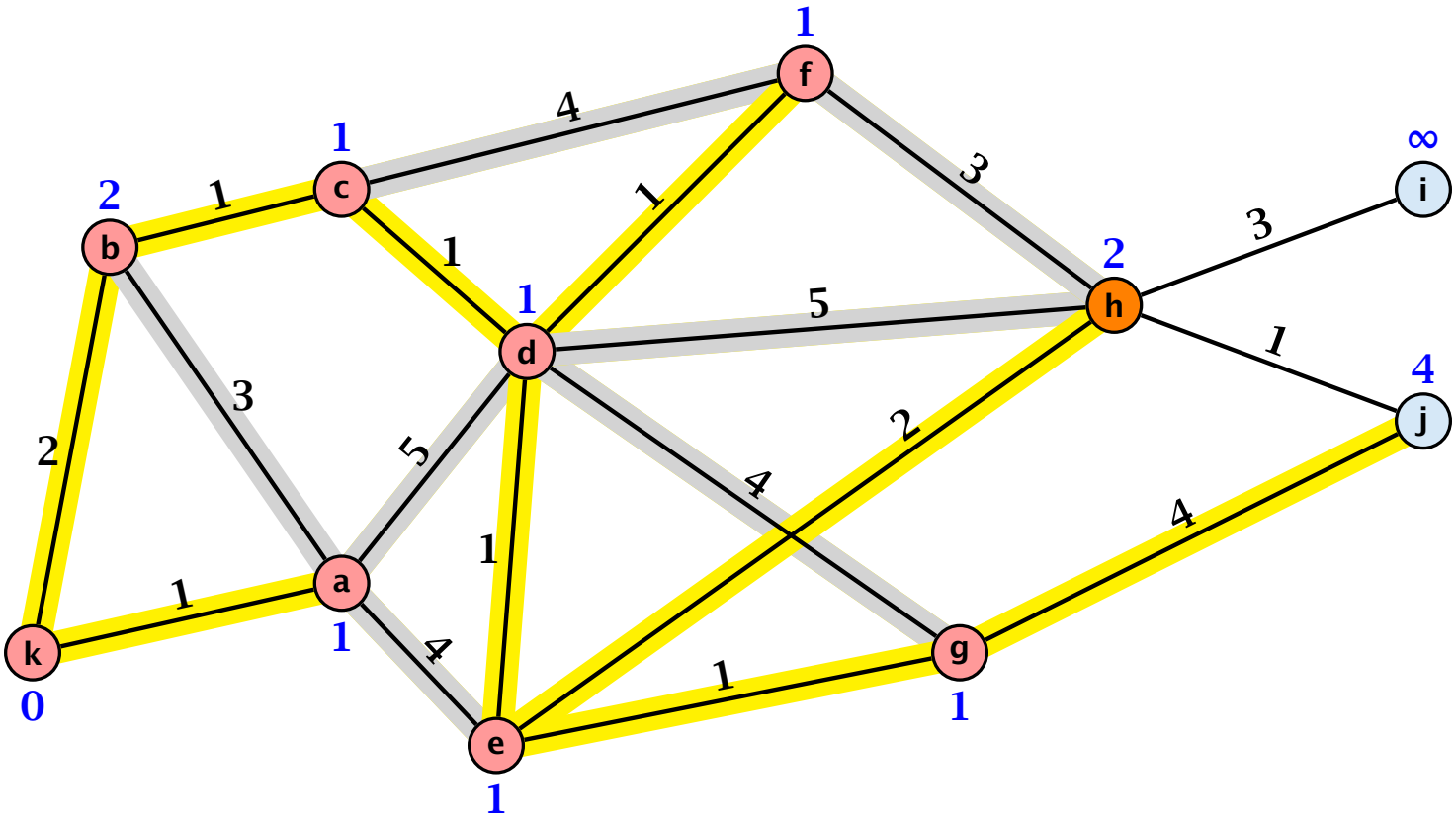
Minimum Spanning Tree



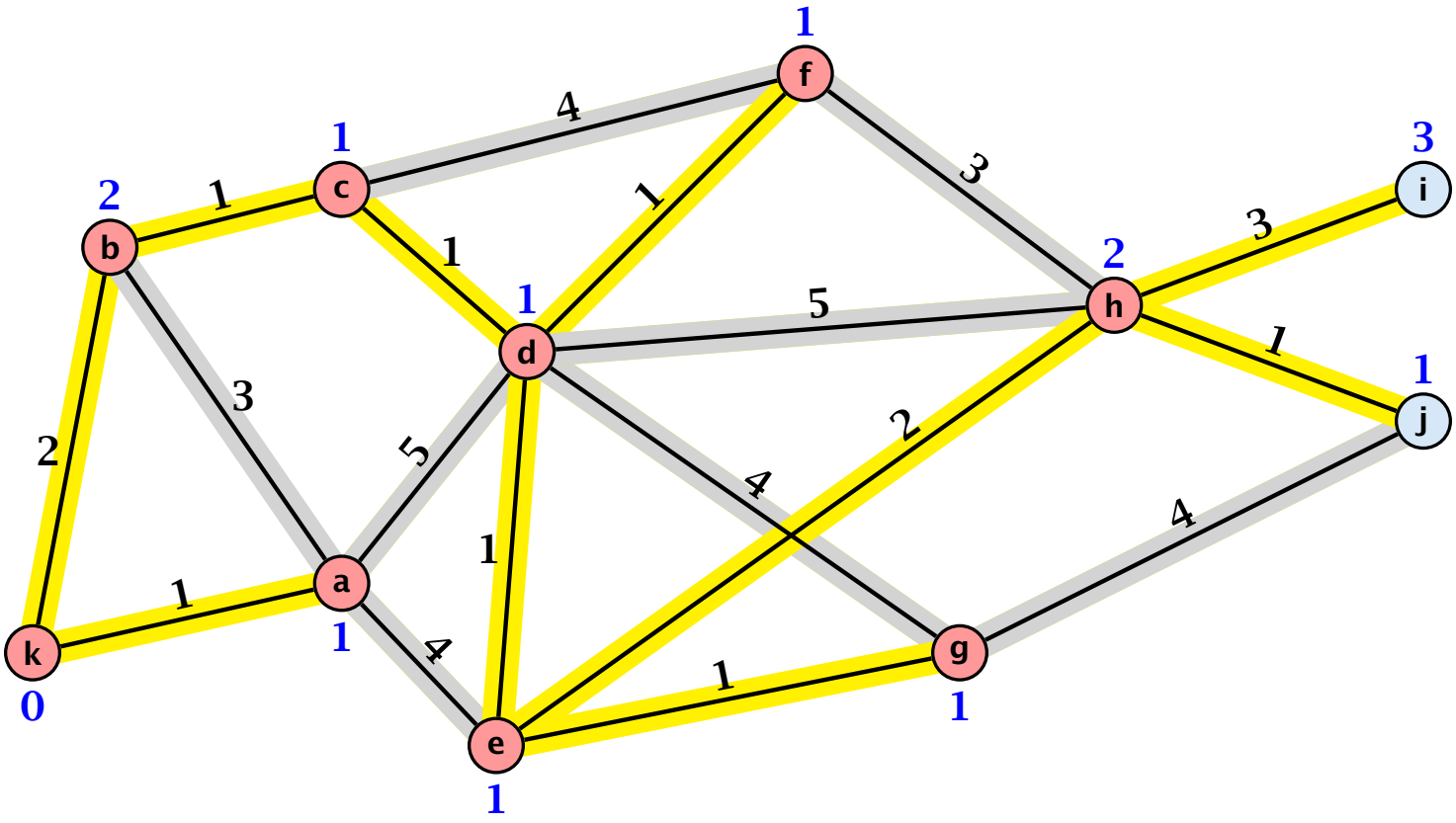
Minimum Spanning Tree



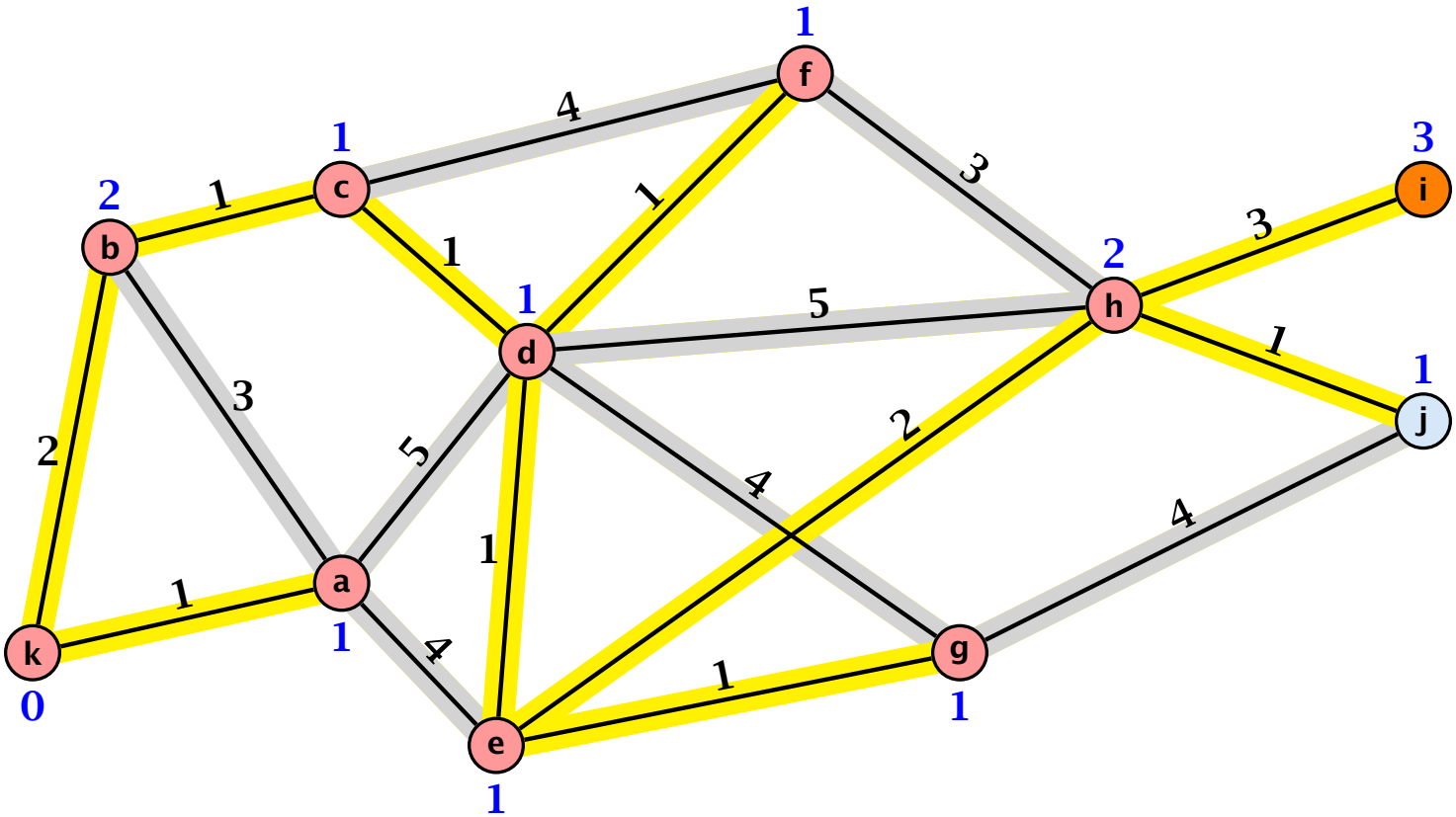
Minimum Spanning Tree



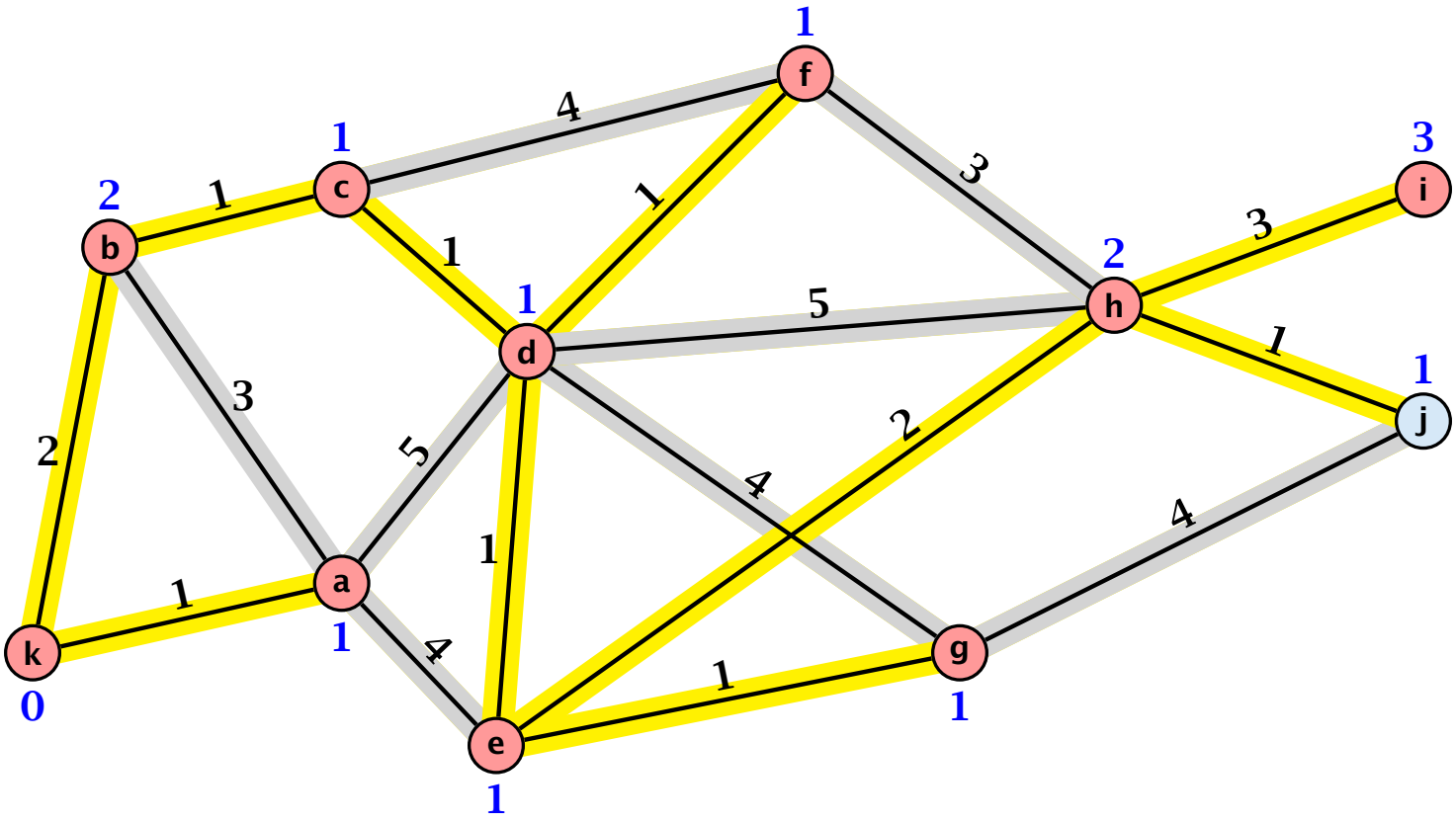
Minimum Spanning Tree



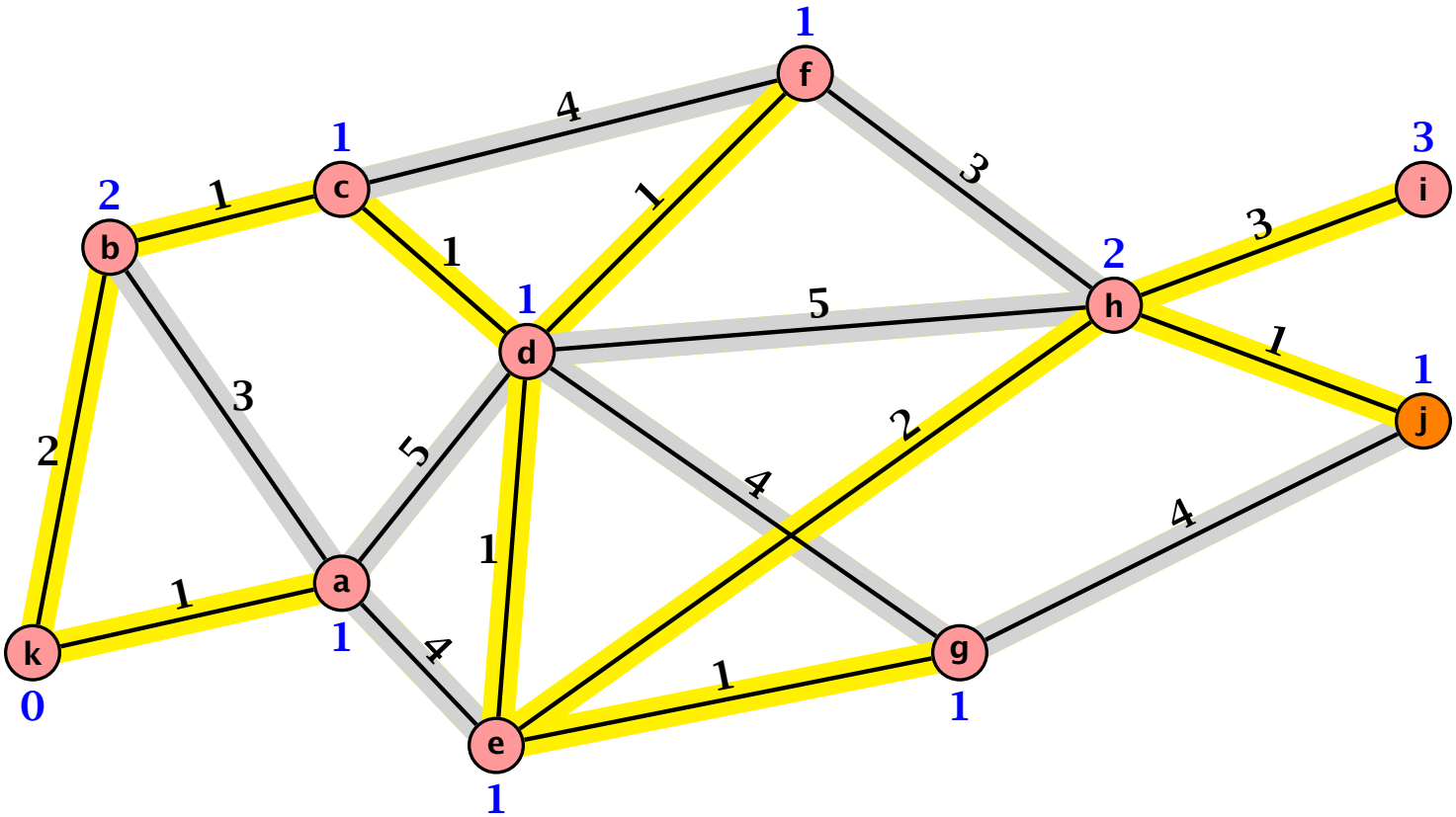
Minimum Spanning Tree



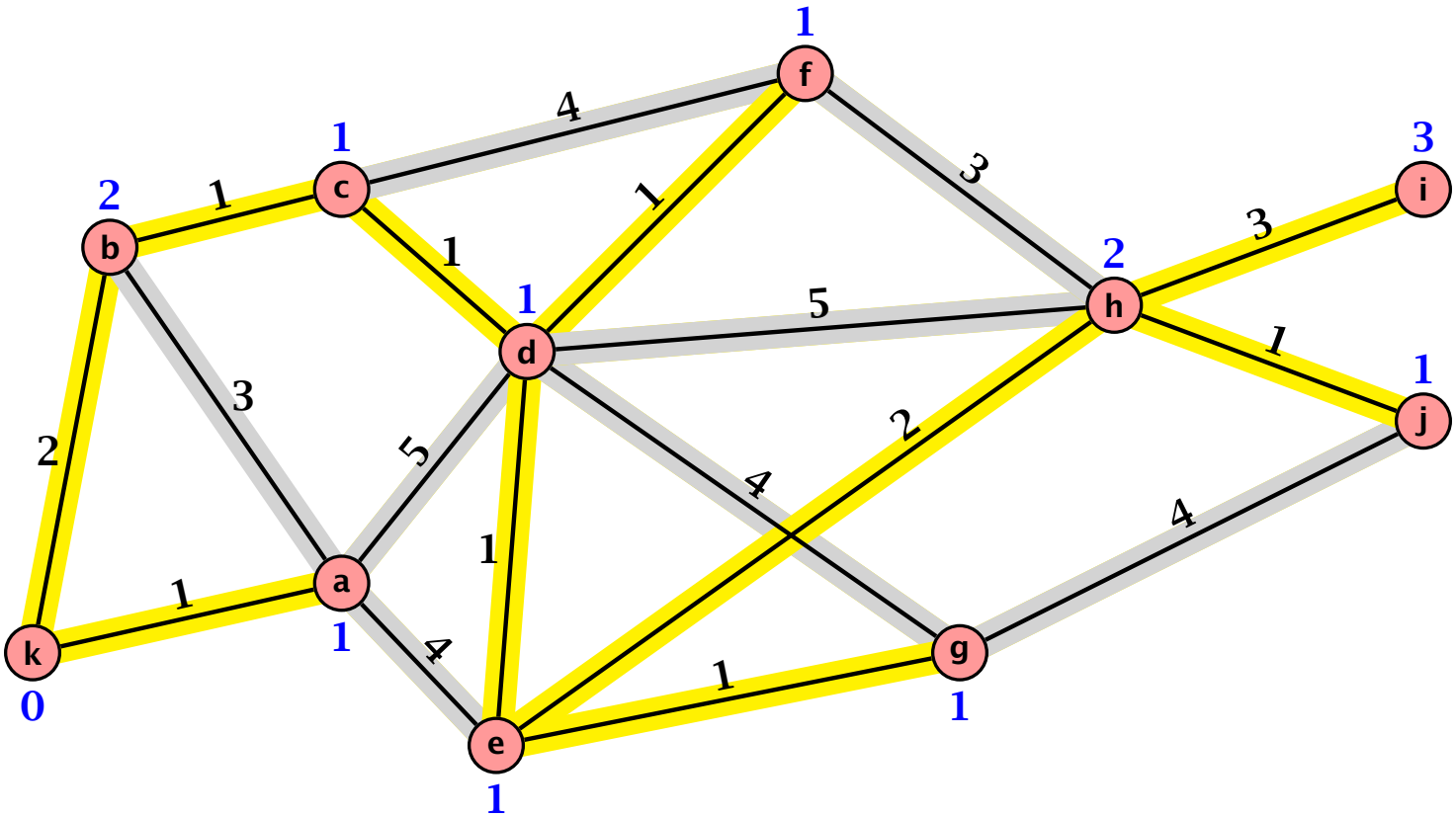
Minimum Spanning Tree



Minimum Spanning Tree

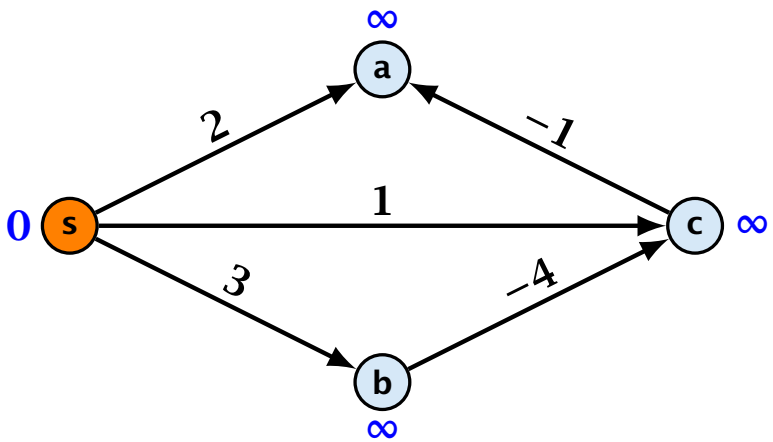


Minimum Spanning Tree



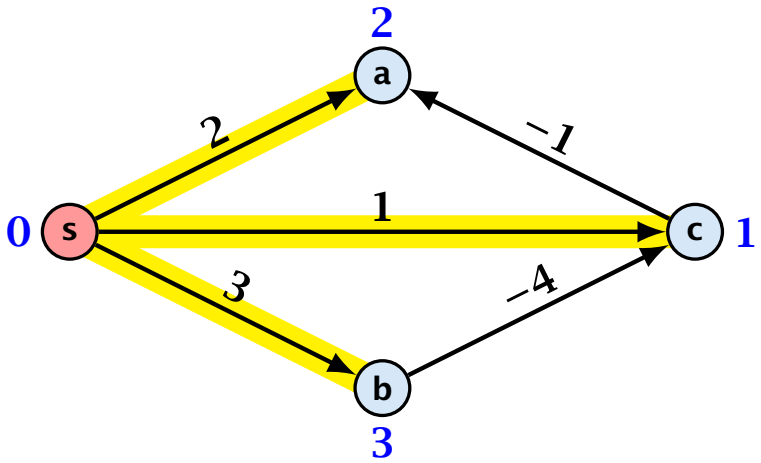
Negative Weights

Dijkstra clearly doesn't work:



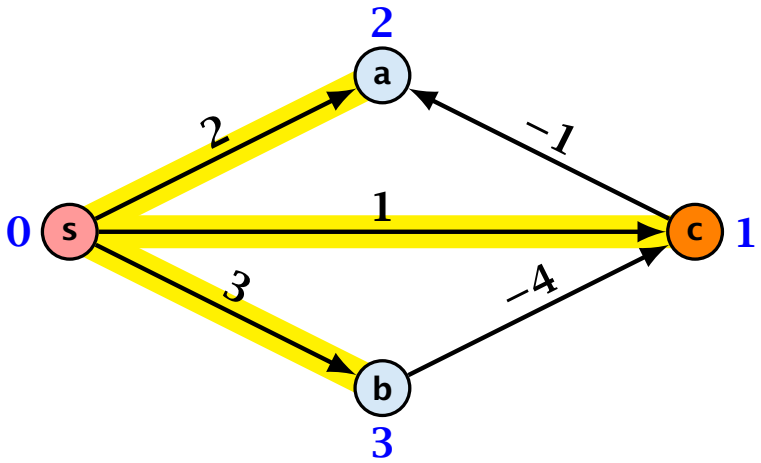
Negative Weights

Dijkstra clearly doesn't work:



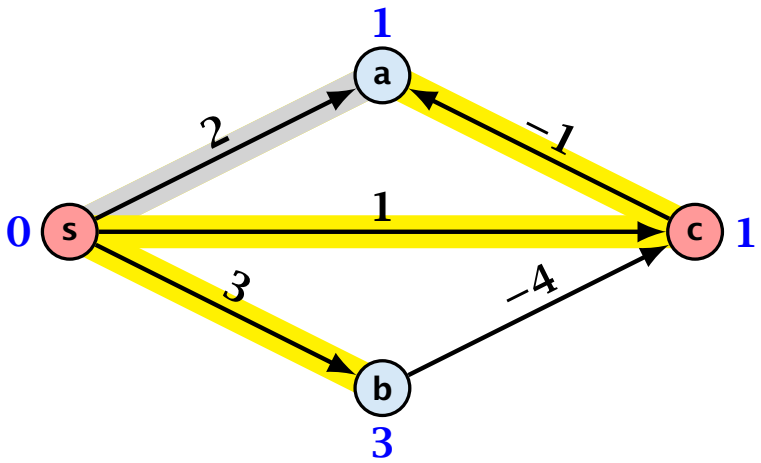
Negative Weights

Dijkstra clearly doesn't work:



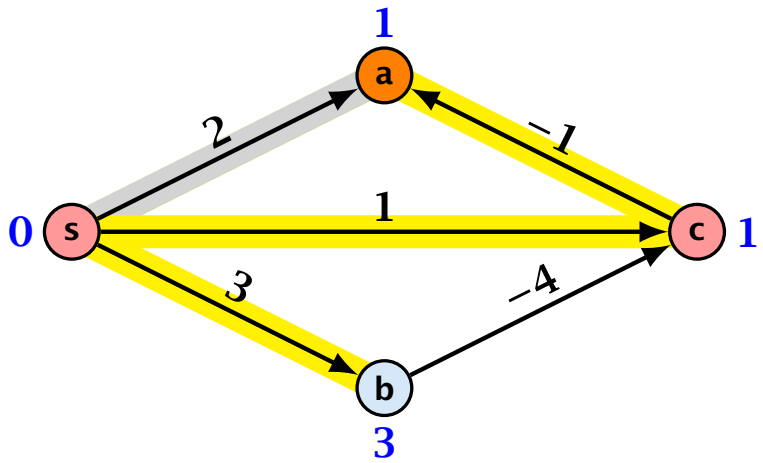
Negative Weights

Dijkstra clearly doesn't work:



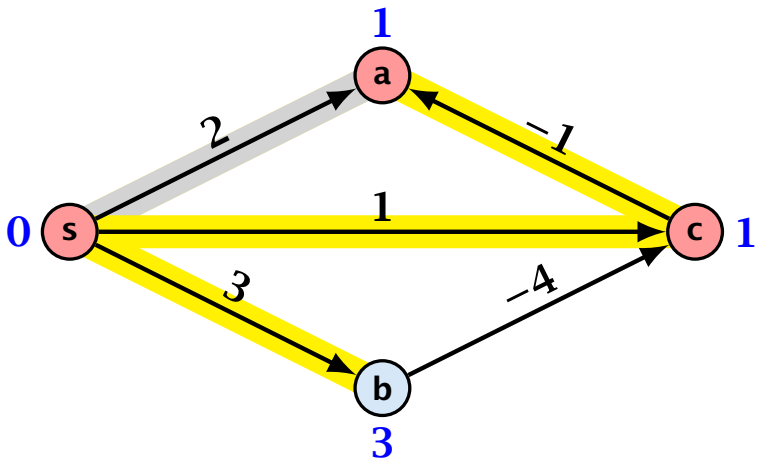
Negative Weights

Dijkstra clearly doesn't work:



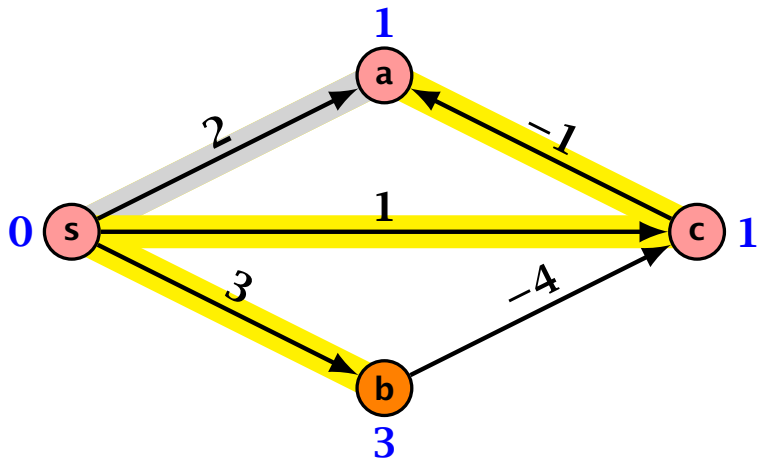
Negative Weights

Dijkstra clearly doesn't work:



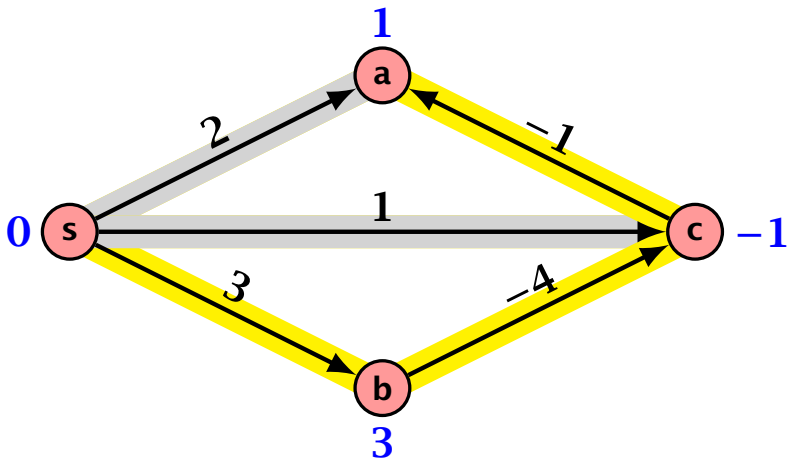
Negative Weights

Dijkstra clearly doesn't work:



Negative Weights

Dijkstra clearly doesn't work:



Negative Weights

Lemma 2

For every vertex x with $d(s,x) \neq \pm\infty$, there exists a shortest path with at most $n-1$ edges between s and v.

Proof:

- let x be vertex with $d(s,x) \neq \pm\infty$, hence, shortest s-x path exists; let P be shortest s-x path with minimum hop number
- assume, for contradiction, that P has at least n edges ($\geq n+1$ vertices)
- hence, P contains a directed cycle
- if the cycle is of negative weight the distance to x is $-\infty$ (↯)
- otherwise removing the cycle from P gives an s-x path with smaller number of hops and at most length $w(P)$ (↯)

The Bellman-Ford Algorithm

For $n-1$ phases go through all edges and relax every edge.

Algorithm 5 Naive-Bellman-Ford$(G=(V,E,w), s \in V)$

1: **Input:** weighted graph $G=(V,E,w)$; start vertex s;
2: **Output:** key-field of every node contains distance from s;
3: $s.\text{key} \leftarrow 0$;
4: **for** $v \in V \setminus \{s\}$ **do** $v.\text{key} \leftarrow \infty$;
5: **for** $\ell \leftarrow 1$ **to** $n-1$ **do**
6: **for** $(x,y) \in E$ **do** $y.\text{key} \leftarrow \min\{y.\text{key}, x.\text{key} + w(x,y)\}$

Invariant:
After phase ℓ the distance label of a vertex x is at most the length of a shortest s-x path with at most ℓ hops.

The invariant can easily be proven by induction for $\ell \in \{0,\ldots,n-1\}$.
Here, phase $\ell = 0$ corresponds to the initialization before Line 5.

The Bellman-Ford Algorithm

Plugging $\ell = n - 1$ into the invariant gives that at the end of the algorithm every node x that has distance $d(s,x) \neq -\infty$ has the correct distance label.

How, do we detect vertices that can be reached from s via a path that touches a vertex on a negative cycle? (these are **exactly** the vertices that have distance $-\infty$)

The Bellman-Ford Algorithm

Add one more phase n. Suppose, an edge-relaxation for edge (x, y) in phase n changes the distance label for y.

Then, y has distance $-\infty$, and every vertex reachable from y also has distance $-\infty$.

Lemma 3

Every negative cycle reachable from s contains a vertex that changes its distance-label in phase n.

The Bellman-Ford Algorithm

Proof:

- Let C be a negative cycle reachable from s. After phase $n-1$ all vertices in this cycle have $x.\text{key} \neq \infty$.
- Assume for contradiction that none of the vertices in the cycle changes its label. This means

$$v_{i+1}.\text{key} \leq v_i.\text{key} + w(v_i, v_{i+1})$$

  for all $i \in \{1, \ldots, s-1\}$, where $v_1, \ldots, v_{s-1}, v_s$ with $v_1 = v_s$ are the vertices of C.
- Applying this equation repeatedly gives

$$v_s \leq v_1 + w(C)$$

  which is a contradiction since $v_s = v_1$ and $w(C) < 0$.

The Bellman-Ford Algorithm

This means we just need to mark all vertices that change distance label in phase n, and mark all vertices reachable from these vertices.

This is done by the following sub-routine.

Algorithm 6 mark(x)

1: **Input:** vertex $x \in V$;
2: **Output:** sets $v.\text{key}$ to $-\infty$ for every vertex reachable from x
3: **if** $x.\text{key} \neq -\infty$ **then**
4: $\quad x.\text{key} \leftarrow -\infty$ **do**
5: $\quad$ **for all** $(x,y) \in E$ **do** mark(y)

If we call $\text{mark}(x)$ for all $x \in S \subseteq V$ the total running time is only $\mathcal{O}(|S| + m)$, since an edge is checked at most once.

The Bellman-Ford Algorithm

Algorithm 7 Bellman-Ford$(G = (V, E, w), s \in V)$

```
 1: Input: weighted graph G = (V, E, w); start vertex s;
 2: Output: key-field of every node contains distance from s;
 3: for all v ∈ V do
 4:         v.key ← ∞;  v.parent ← ∅;
 5: s.key ← 0;
 6: for ℓ ← 1 to n − 1 do
 7:         for all (x, y) ∈ E do
 8:                 if x.key > v.key + w(v, x) then
 9:                         x.key ← v.key + w(v, x);
10:                         x.parent ← y;
11: for all (x, y) ∈ E do
12:         if x.key > v.key + w(v, x) then
13:                 x.parent ← y;
14:                 mark(x);
```

Running time is $\mathcal{O}(mn)$.

The Bellman-Ford Algorithm

Invariant:
After the ℓ-th phase the parent pointer of x points to the predecessor of x on a shortest s-x path with at most ℓ hops.

$x.\text{parent} = \emptyset$ means that **either** there is no s-x path with at most ℓ hops **or** the shortest among these paths has zero hops ($s = x$).

After termination:
Following the parent pointer for nodes v with $d(s,v) \neq \pm\infty$ leads to s (backwards along a shortest s-v path).

Following the parent pointer for nodes v with $d(s,v) = -\infty$ leads to a negative cycle (hence, we can use this algorithm to find a negative cycle).

The Bellman-Ford Algorithm

Improvements

- Maintain a list of edges (x, y) that have to be relaxed because their endpoint x has changed its distance label in the last phase (initially the list just contains s).
- A phase just relaxes all edges in the list (and generates the list for the next phase).
- If the list is empty the algorithm can stop before reaching phase $n - 1$.

All Pairs Shortest Path

Input:

- directed graph $G = (V, E)$ with edge weights $w : E \to \mathbb{R}$

Output:

a distance matrix, giving the shortest path distance between all pairs

b n shortest path trees (shortest path tree with root x encodes the shortest paths from x to all other vertices)

output size is $\Theta(n^2)$ in both variants

All Pairs Shortest Path

For positive edge-weights we can run Dijkstra n-times which gives a running time of $\mathcal{O}(mn + n^2 \log n)$ when using Fibonacci heaps (nothing a lot better is known).

The Johnson-Dijkstra algorithm allows to obtain the same running time for the case of arbitrary weights without negative cycles.

Adding some preprocessing allows to obtain the same asymptotic running time also for the general case of arbitrary weights with possible negative cycles.