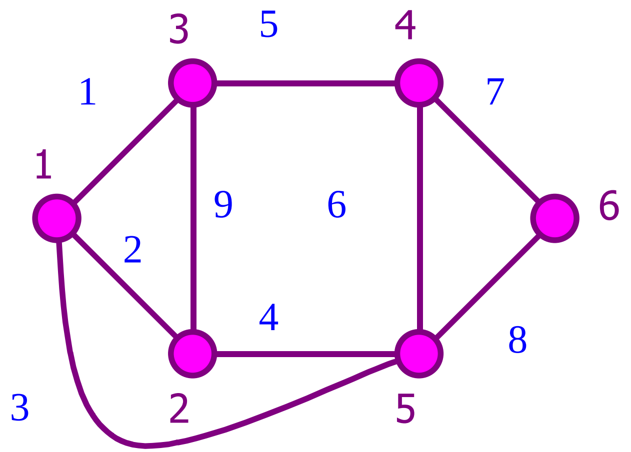
Basic Algorithms on Graphs

VLSI CAD

COMPILED BY OLEG VENGER

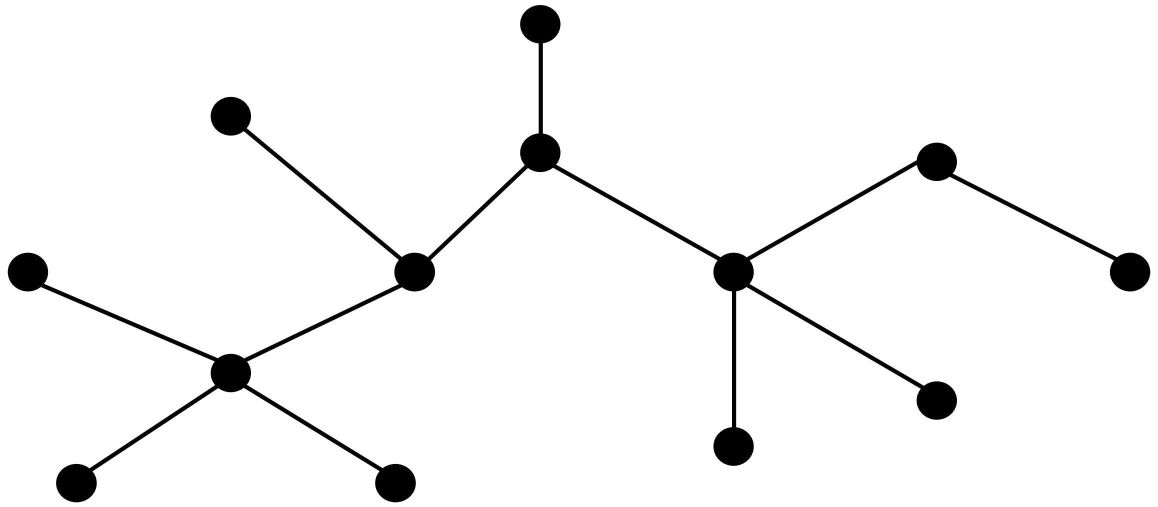
Graphs

- An abstract way of representing connectivity
- Graph $G = (V, E)$ consists of finite set of nodes (also called vertices) $V = \{v_1, \ldots v_n\}$ and finite set of edges $E = \{e_1, \ldots, e_m\}$ such that $E \subseteq VxV$
- Edges can be directed on undirected
- Nodes and edges can have associated auxiliary information
- Graphs are usually represented by adjacency matrix or adjacency list
- Want to support operations such as:
  - Retrieving all edges incident to a particular node
  - Testing if two given nodes are directly connected



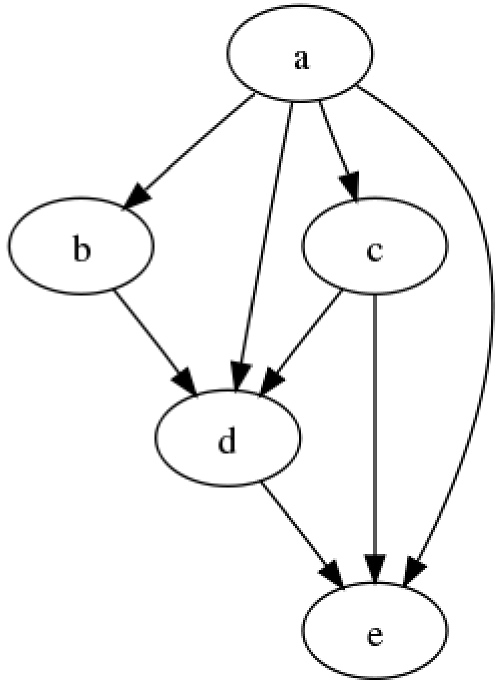
Tree

- An undirected connected acyclic graph
- For n nodes has $n - 1$ edges
- There is exactly one path between every pair of nodes
- Adding any edge results in a cycle
- Removing any edge disconnects graph
- Graph that is a union of trees is called "forest"



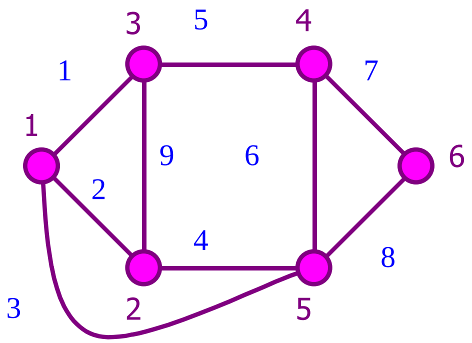
DAG

- Directed Acyclic Graph
- No directed cycles
- May contain “parallel” paths



Adjacency matrix

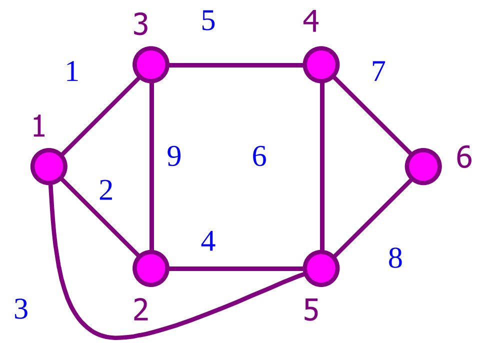
- For graph with V nodes: $V\ x\ V$ matrix A where:
  - $a_{ij} = 1$ if there is an edge from i to j.
  - $a_{ij} = 0$ otherwise
- $O(V^2)$ memory. Inefficient for large and sparse graphs
- $O(1)$ for testing if 2 nodes are connected
- $O(V)$ to find all adjacent nodes



	1	2	3	4	5	6
1	0	1	1	0	1	0
2	1	0	1	0	1	0
3	1	1	0	1	0	0
4	0	0	1	0	1	1
5	1	1	0	1	0	1
6	0	0	0	1	1	0

Incidence matrix

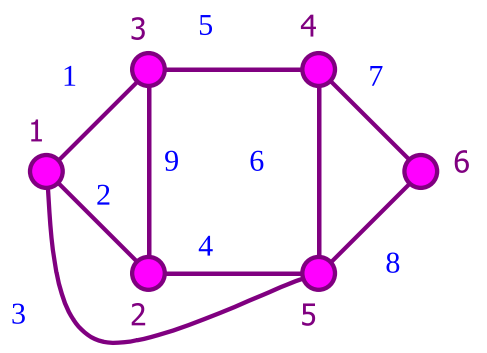
- $V\ x\ E$ matrix A where:
  - $a_{ij} = 1$ if vertex i connected to edge j.
  - $a_{ij} = 0$ otherwise
- $O(VxE)$ space complexity
- $O(E)$ for testing if 2 nodes are connected
- $O(VxE)$ to find all adjacent nodes

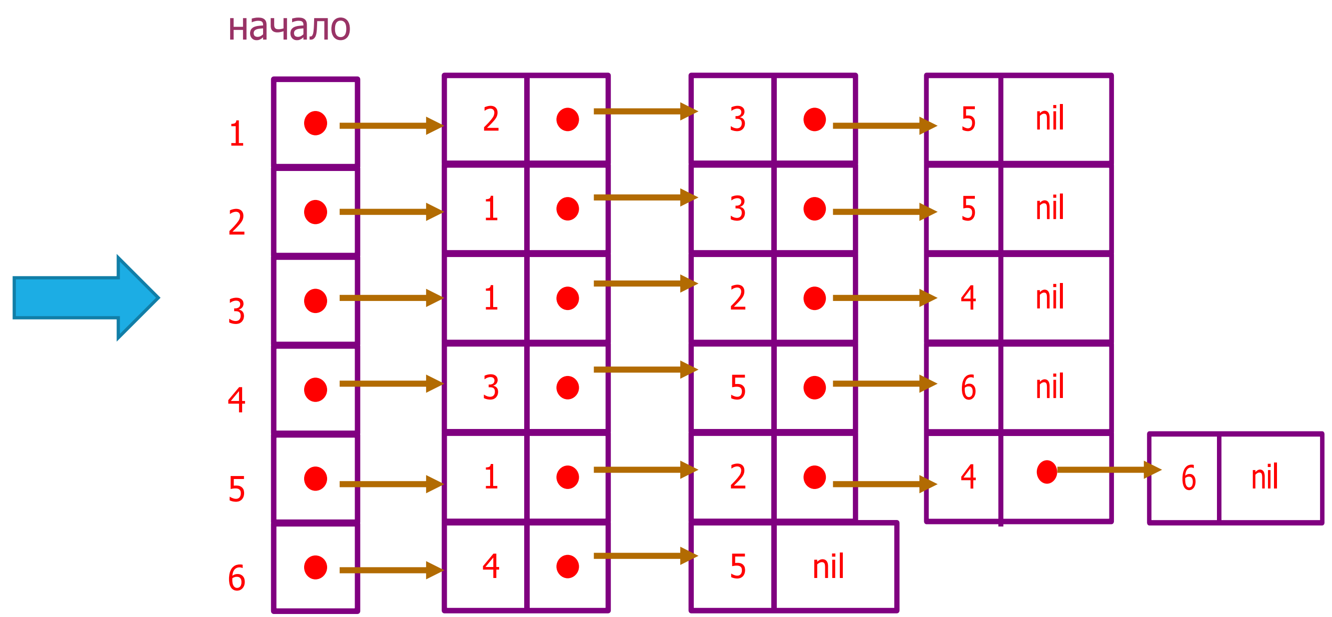


	1	2	3	4	5	6	7	8	9
1	1	1	1	0	0	0	0	0	0
2	0	1	0	1	0	0	0	0	1
3	1	0	0	0	1	0	0	0	1
4	0	0	0	0	1	1	1	0	0
5	0	0	1	1	0	1	0	1	0
6	0	0	0	0	0	0	1	1	0

Adjacency list

- Each node has a list of its adjacent nodes
  - Easy to iterate over edges incident to a particular node
  - Uses $O(V+E)$ memory



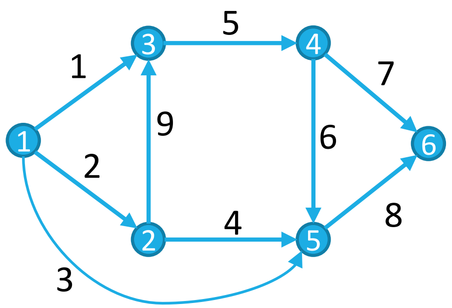


Adjacency list: implementation using array

- Two arrays: E of size m and V of size n
  - E contains edges
  - V contains the starting pointers of the edge lists
- Initialize $V[i] = -1$ for all i
- Inserting a new edge $u \to v$ with ID k :
  - $E[k].to = v$
  - $E[k].nextID = V[u]$
  - $V[u] = k$
- Iterating over all edges starting at u :
  - $for(ID = V[u]; ID! = -1; ID = E[ID].nextID)$
- Once built, it is hard to modify edges, but adding more edges is easy

V :

From	1	2	3	4	5	6
Last Edge ID	3	9	5	7	8	-



E:

ID	To	Next Edge ID
1	3	-
2	2	1
3	5	2
4	5	-
5	4	-
6	5	-
7	6	6
8	6	-
9	3	4

Depth-First Search

- One of two the most basic graph algorithms that visits nodes of a graph in certain order
  - Used as a subroutine in many other algorithms
- $DFS(v)$: visits all nodes reachable from v in depth-first order
  - Mark v as visited
  - **For** each edge $v \to u$:
    - **If** u is not visited, then call $DFS(u)$
- Use non-recursive version if recursion depth is too big: replace recursive calls with a stack

Breadth-First Search

- $BFS(v)$: visit all the nodes reachable from v in breadth-first order:

  - Mark v as visited and push it to queue Q
  - **While** Q is not empty:
    - Take the front element of Q and call it w
    - **For** each edge $w \rightarrow u$
      - **If** u is not visited, mark it as visited and push it to Q

Topological sort: DFS based

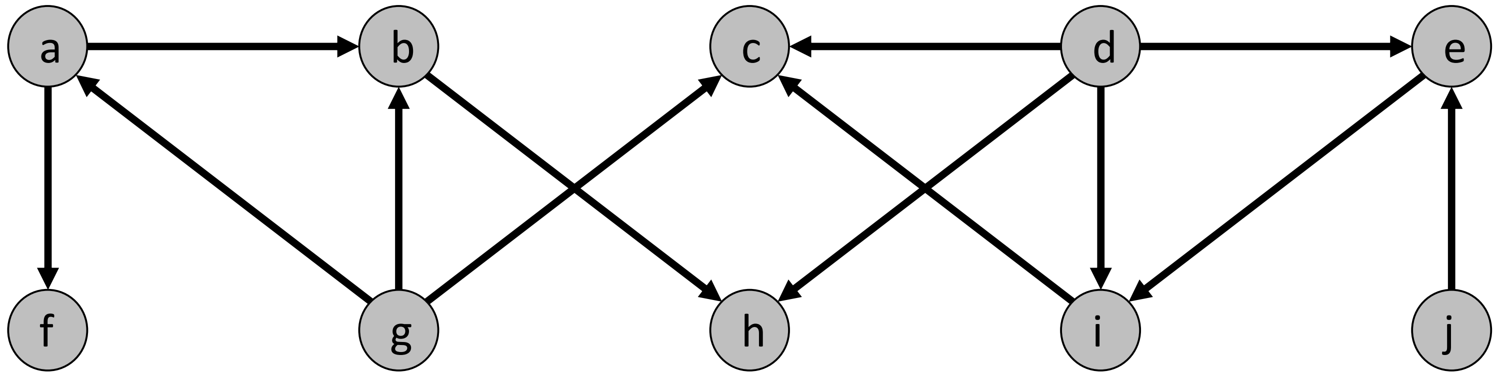
- **Input:** a Directed Acyclic Graph (DAG) $G = (V, E)$
- **Output:** an ordering of nodes such that for each edge $u \rightarrow v, u$ comes before v
  - There can be many answers
- **Applications**:
  - CAD: timing analysis, technology mapping, …
  - Non-CAD: Build systems, task scheduling, …

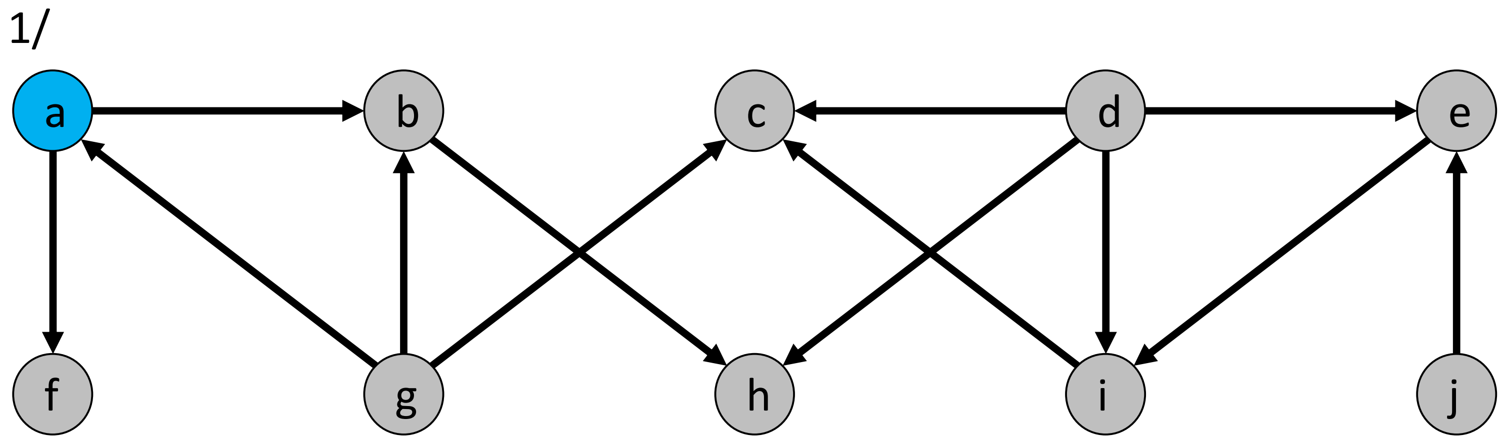
- **Algorithm:**

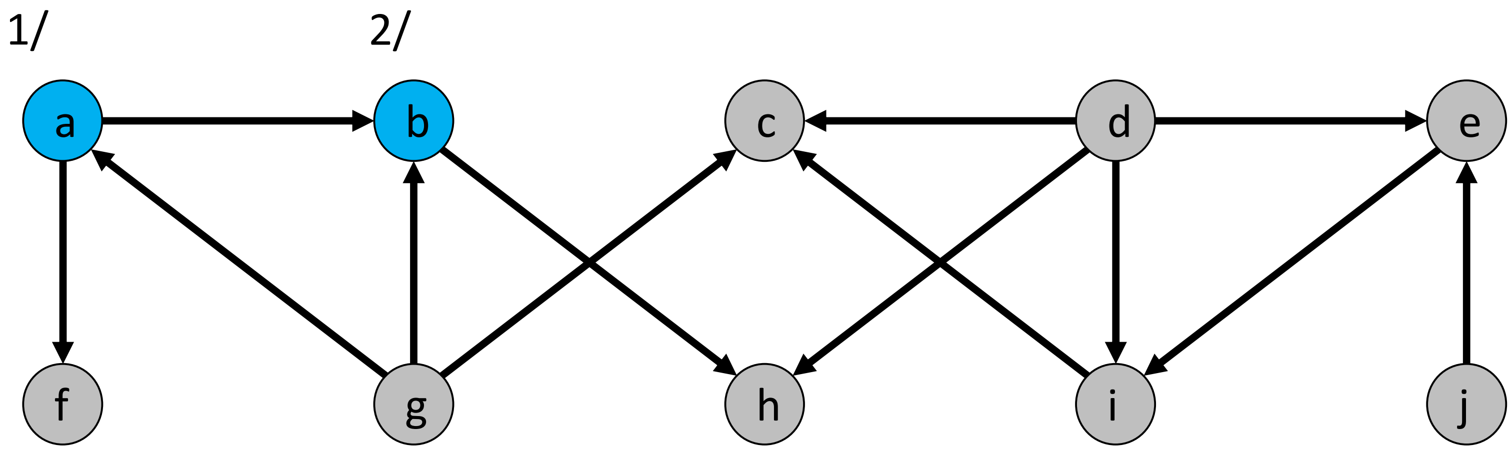
  - Do DFS with start/end timestamps on nodes
  - Order by decreasing value of end timestamps

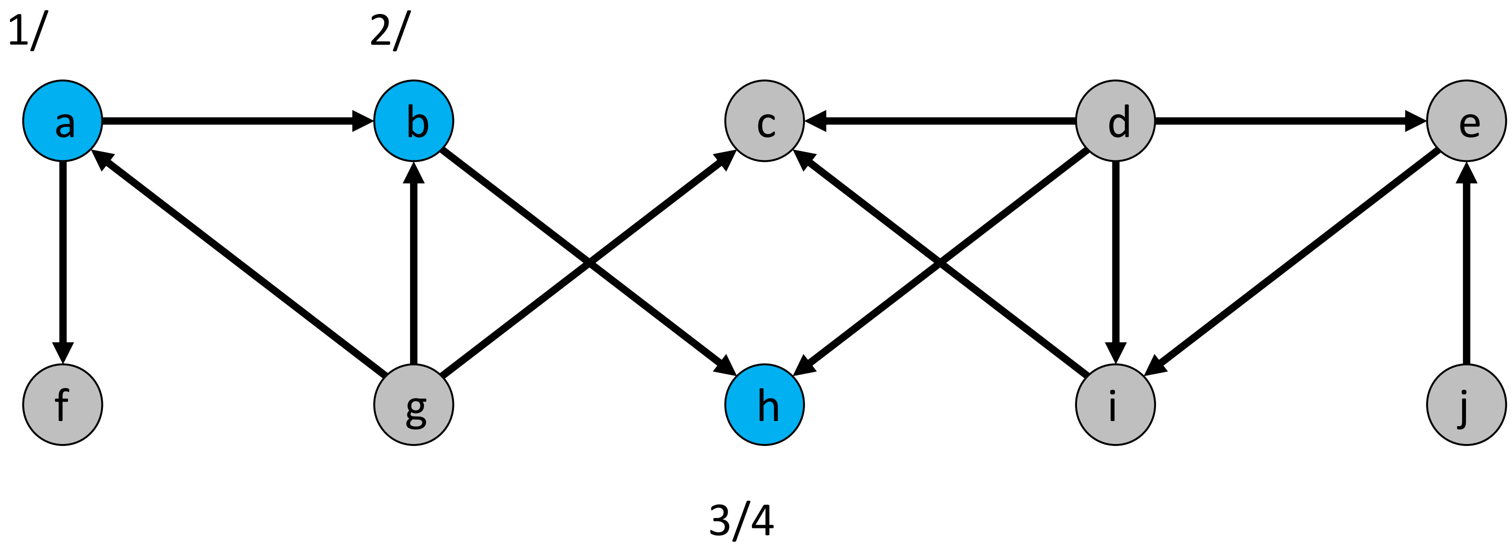
- Time complexity: $O(V + E)$

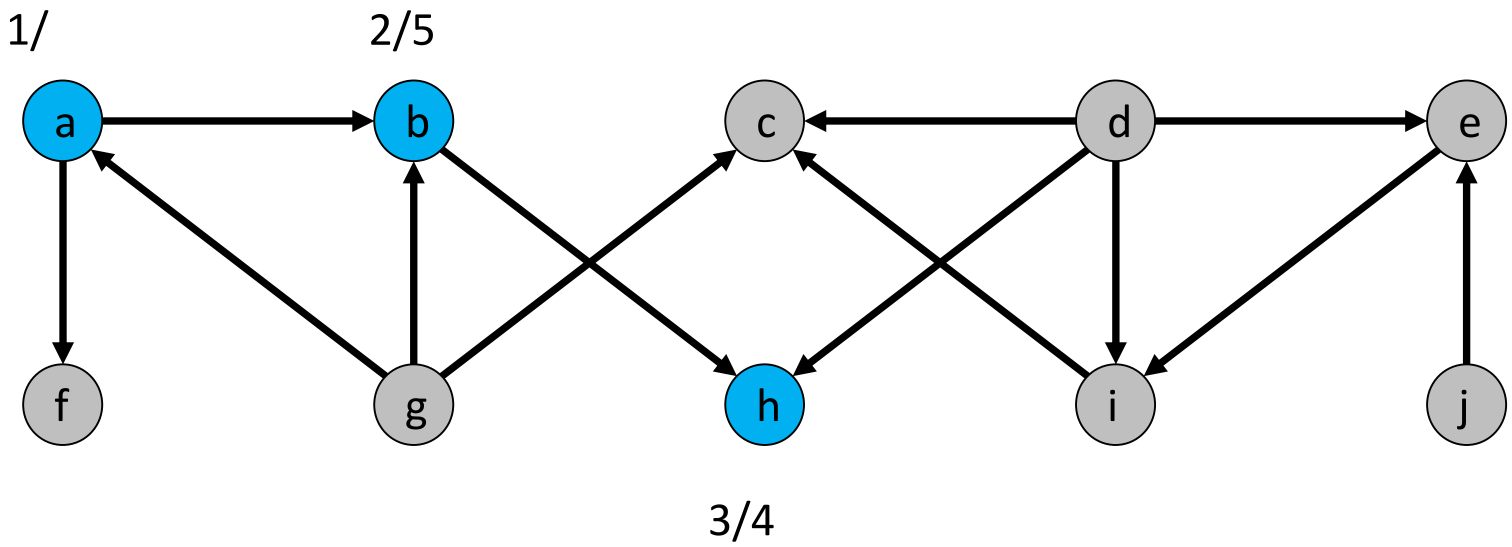
Topological sort example

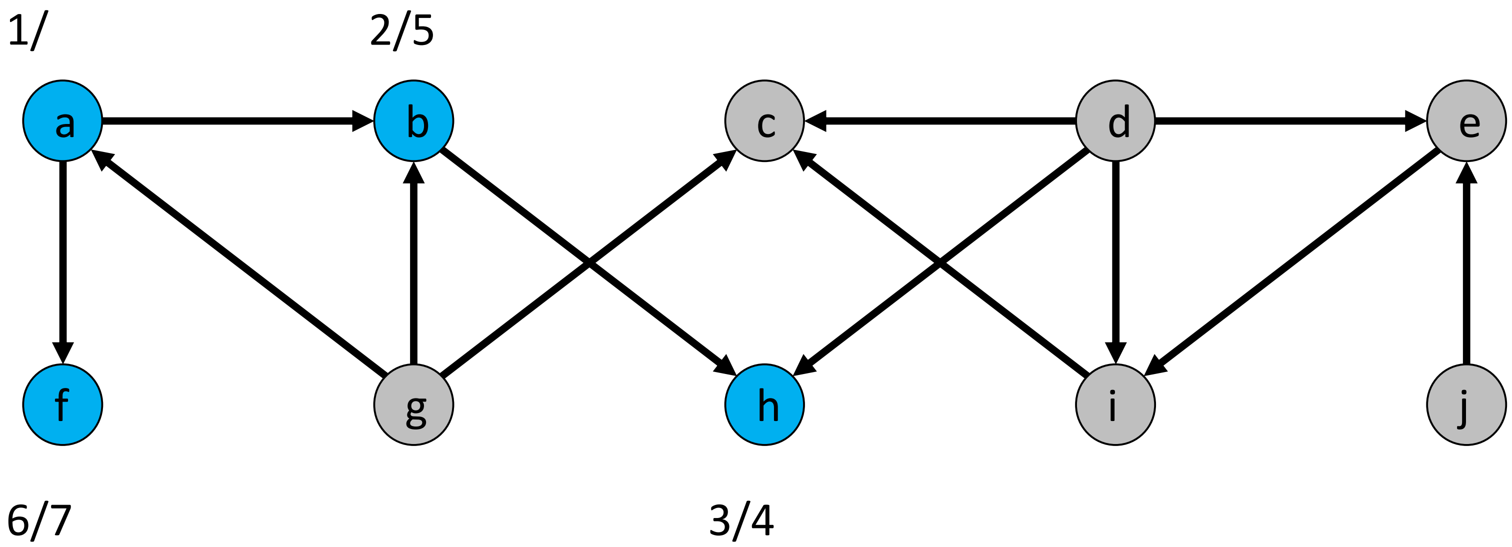


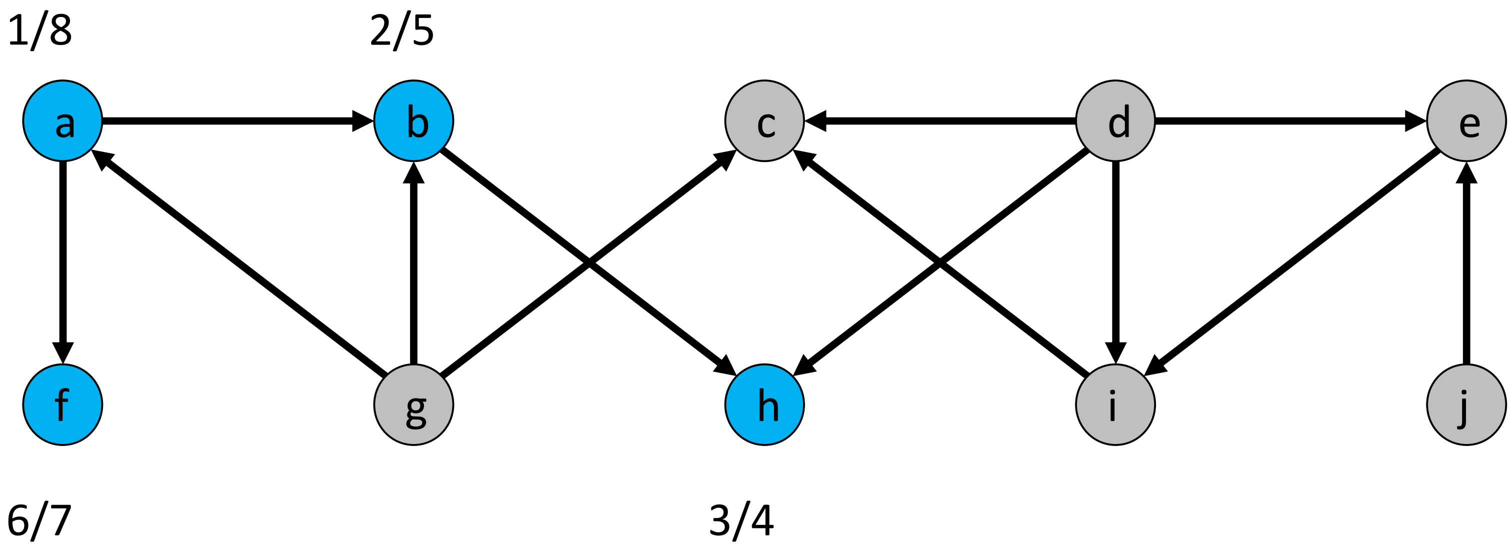

1/
a
b
c
d
e
f
g
h
i
j

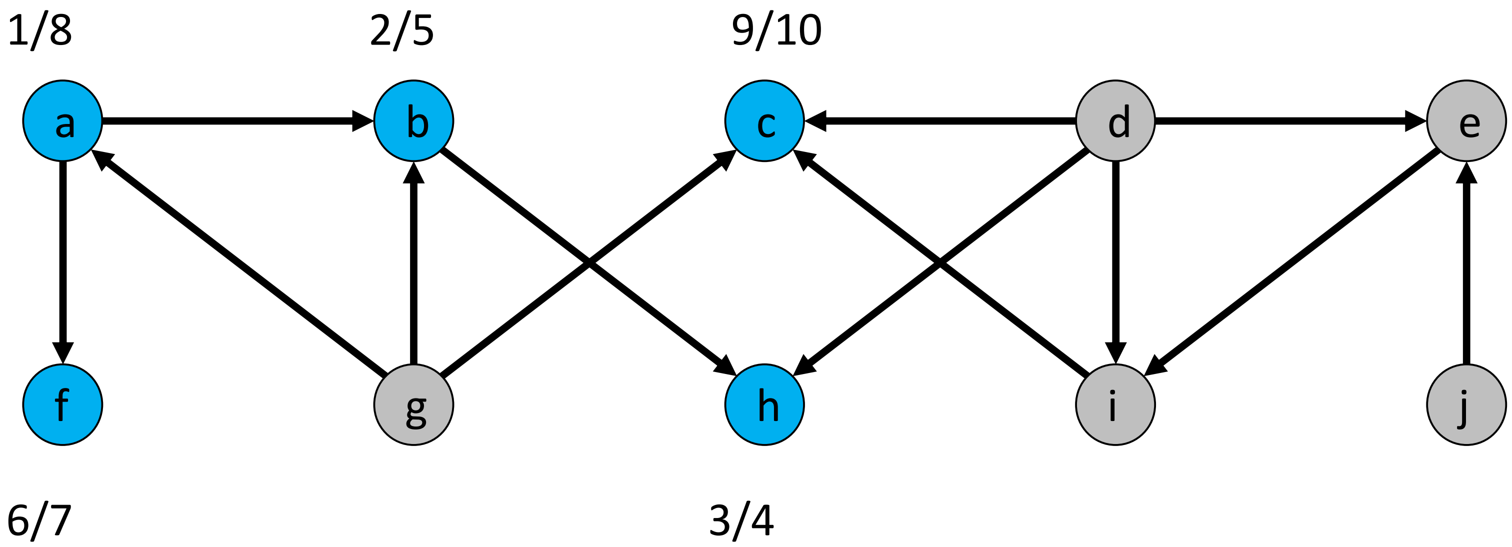

1/
2/
a
b
c
d
e
f
g
h
i
j

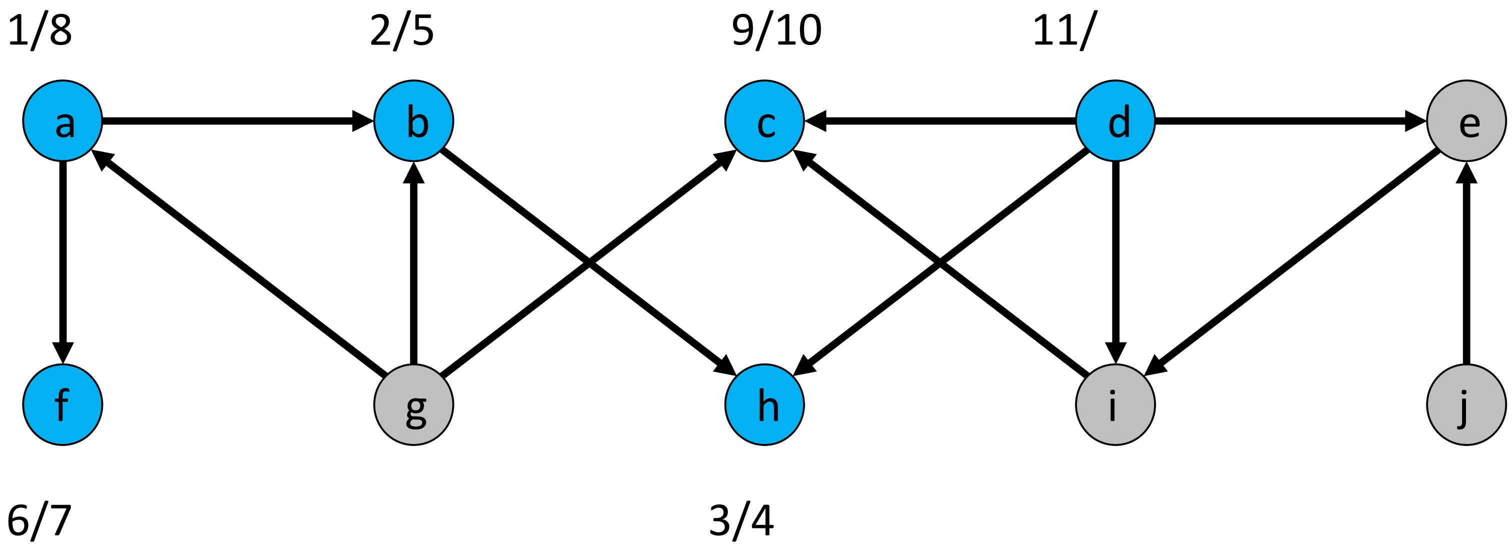

1/
2/
a
b
c
d
e
f
g
h
i
j
3/4

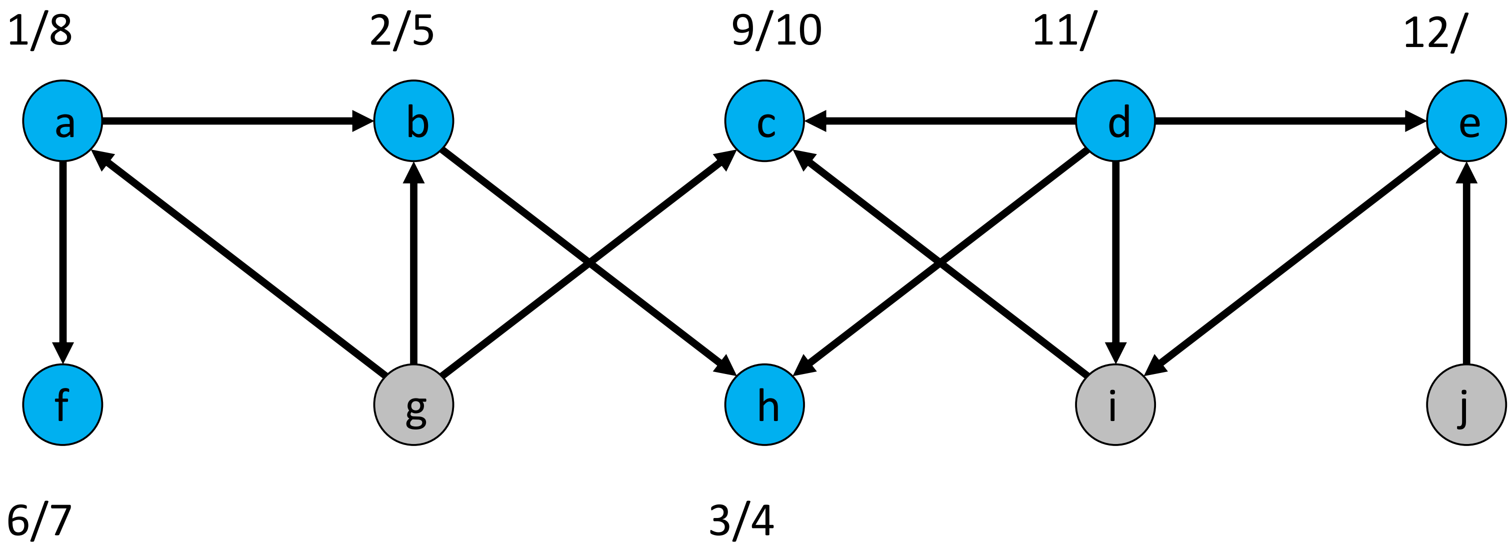

1/
2/5
a
b
c
d
e
f
g
h
i
j
3/4

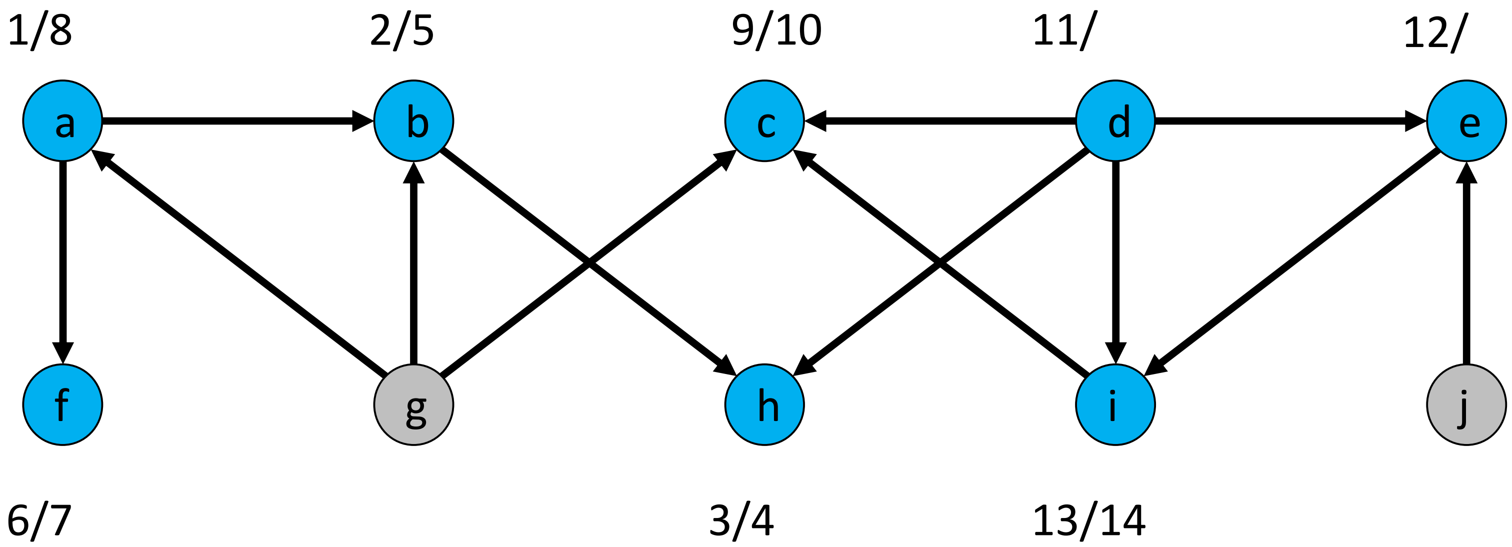

1/
2/5
a
b
c
d
e
f
g
h
i
j
6/7
3/4

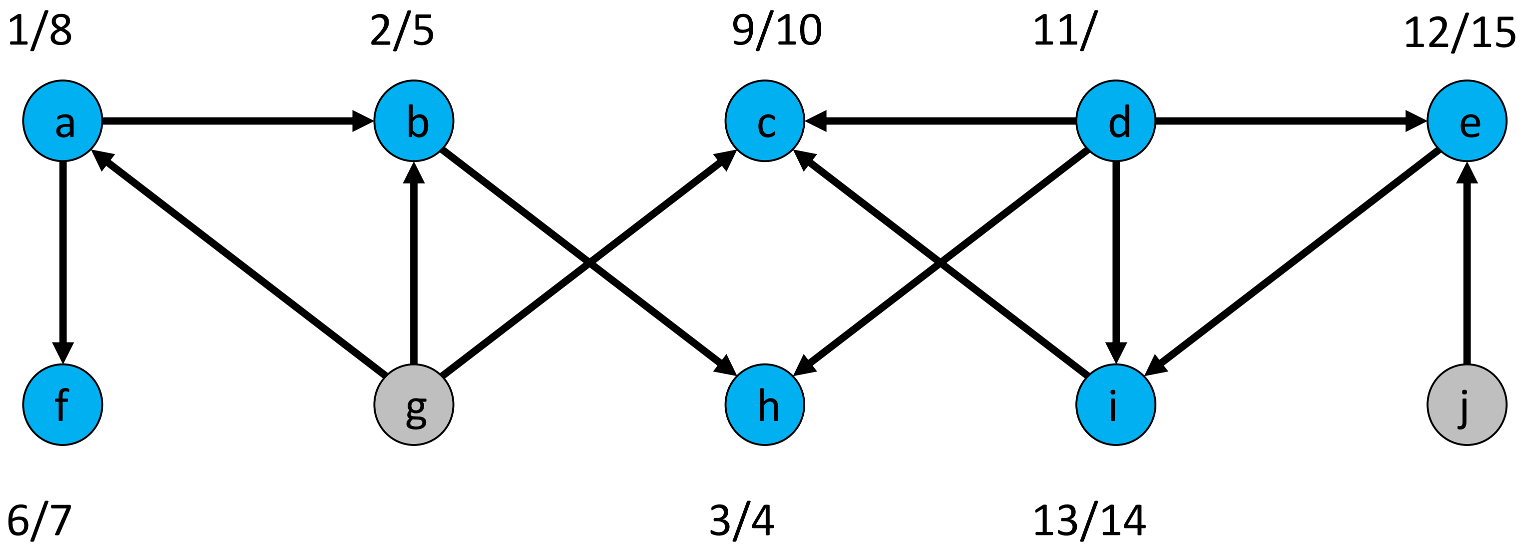

1/8
2/5
a
b
c
d
e
f
g
h
i
j
6/7
3/4

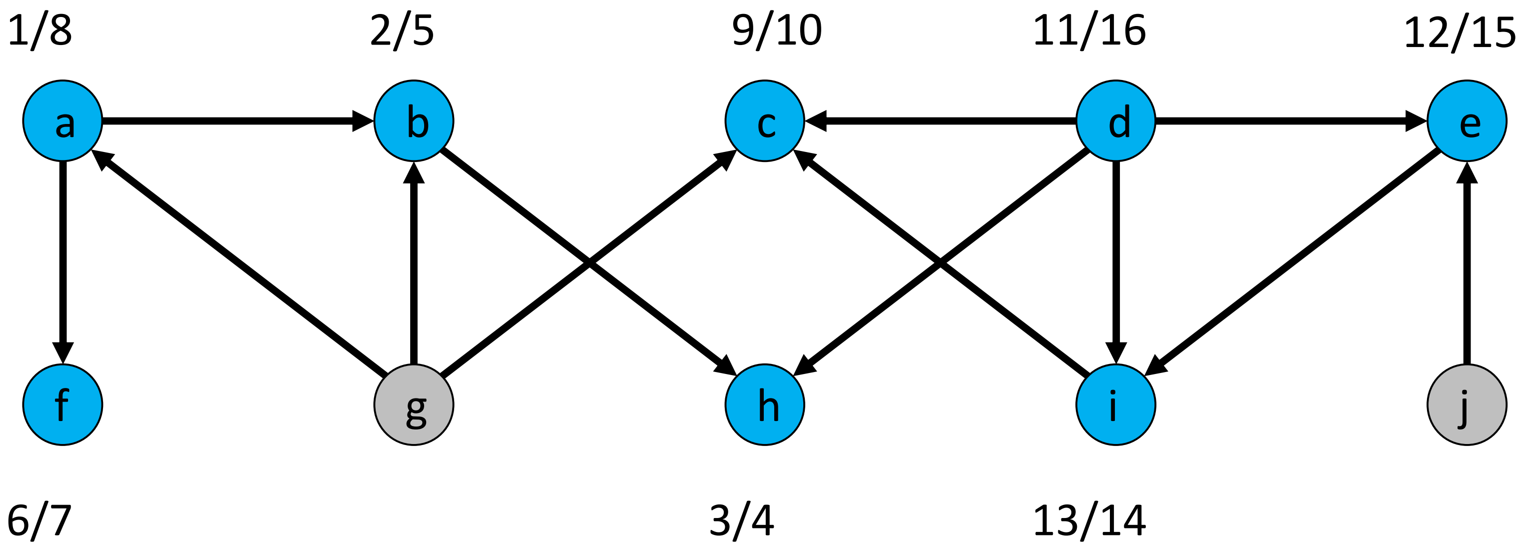

1/8
2/5
9/10
a
b
c
d
e
f
g
h
i
j
6/7
3/4

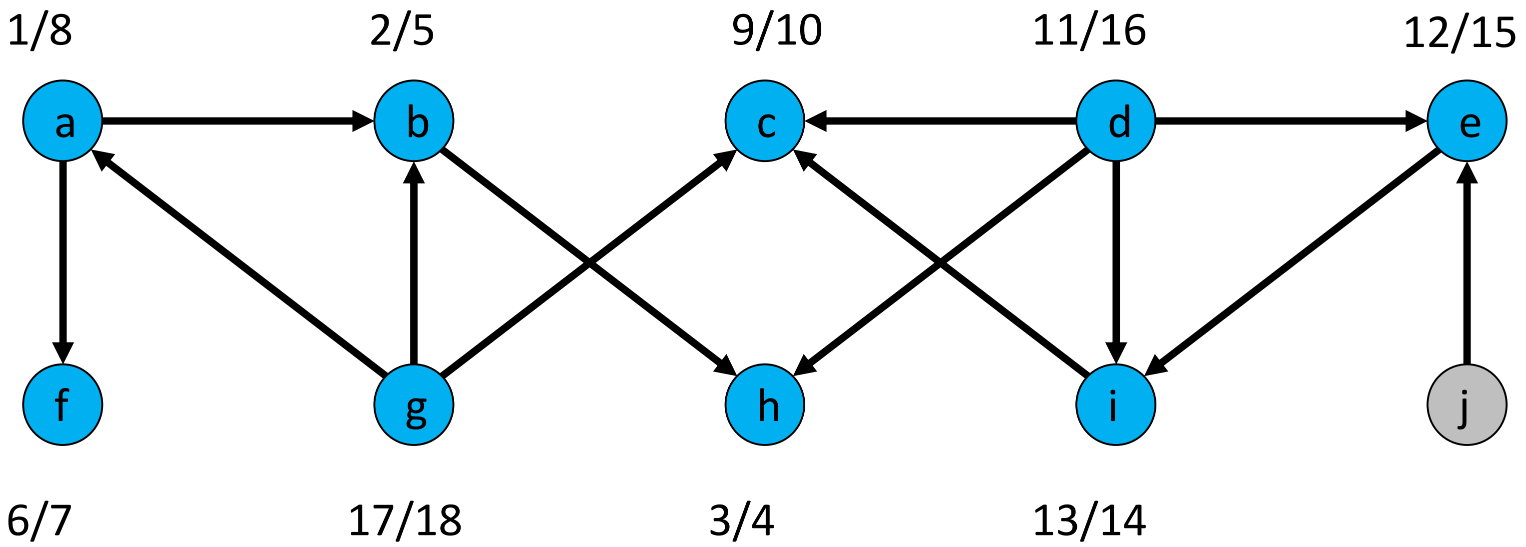

1/8
2/5
9/10
11/
a
b
c
d
e
f
g
h
i
j
6/7
3/4

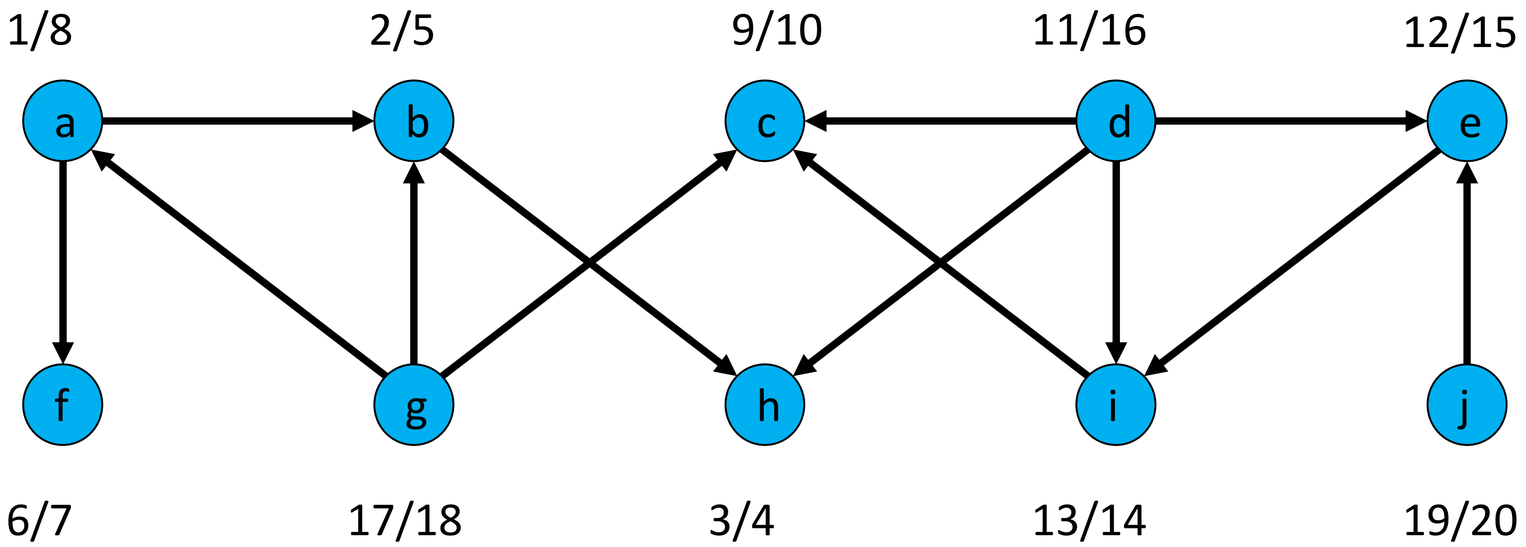

1/8
2/5
9/10
11/
12/
a
b
c
d
e
f
g
h
i
j
6/7
3/4


1/8
2/5
9/10
11/
12/
a
b
c
d
e
f
g
h
i
j
6/7
3/4
13/14


1/8
2/5
9/10
11/
12/15
a
b
c
d
e
f
g
h
i
j
6/7
3/4
13/14


1/8
2/5
9/10
11/16
12/15
a
b
c
d
e
f
g
h
i
j
6/7
3/4
13/14


1/8
2/5
9/10
11/16
12/15
a
b
c
d
e
f
g
h
i
j
6/7
17/18
3/4
13/14



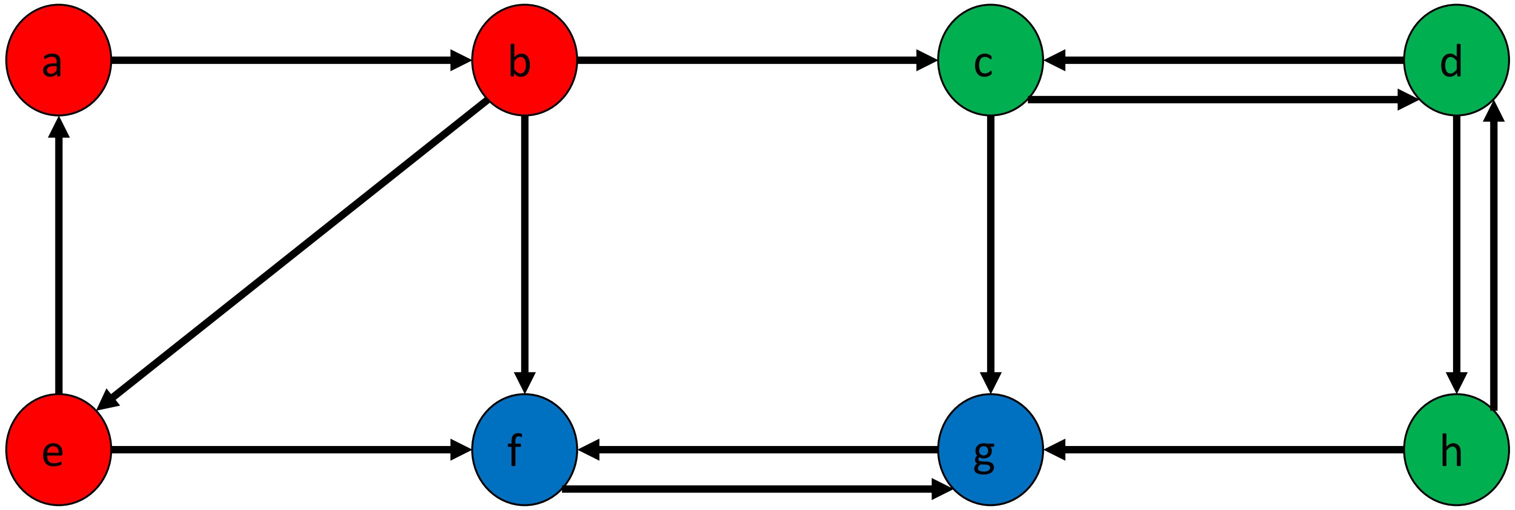
Ordering: j, g, d, e, i, c, a, f, b, h

Topological sort: Kahn's algorithm

- Precompute the number of incoming edges $deg(v)$ for each node v
- Put all nodes v with $deg(v) = 0$ into a queue Q
- **While** Q is not empty:
  - Take v from Q
  - **For** each edge $v \to u$:
    - Decrement $deg(u)$
    - If $deg(u) = 0$, push u to Q
- Time complexity: $\Theta(V + E)$

Strongly connected components

- Given a directed graph $G = (V, E)$
- A graph is strongly connected if all nodes are reachable from every single node in V
- Strongly connected components of G are maximal strongly connected subgraphs of G

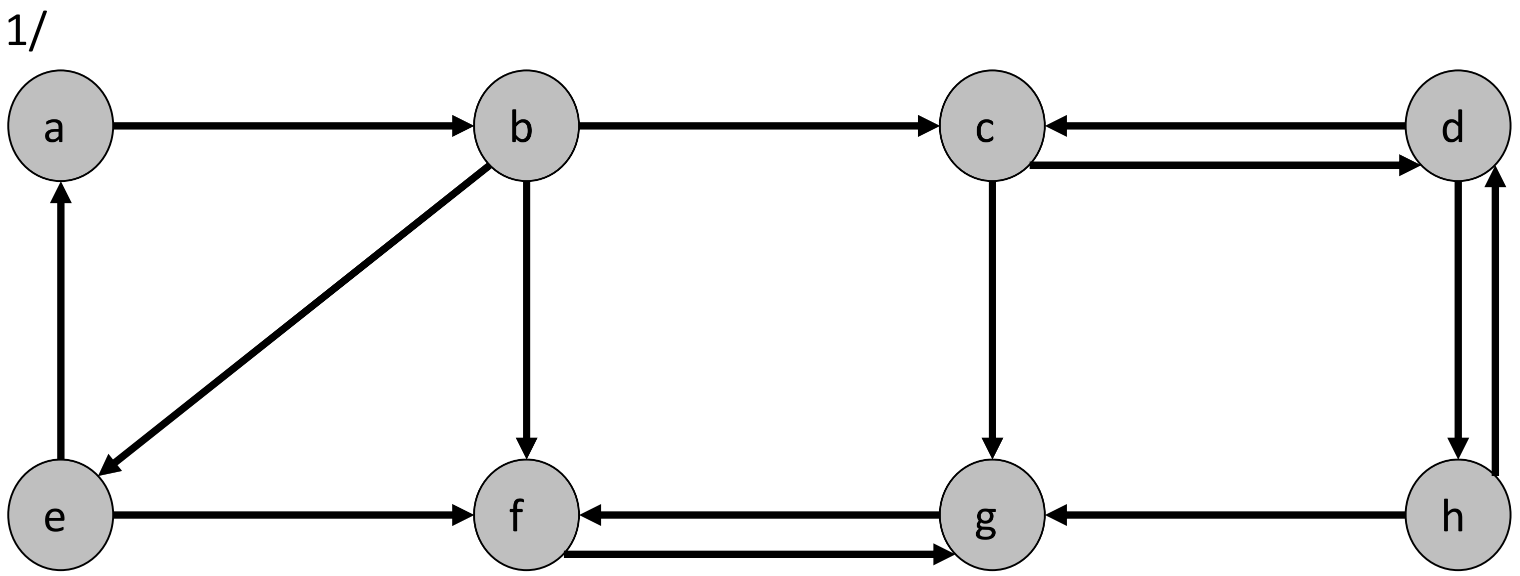


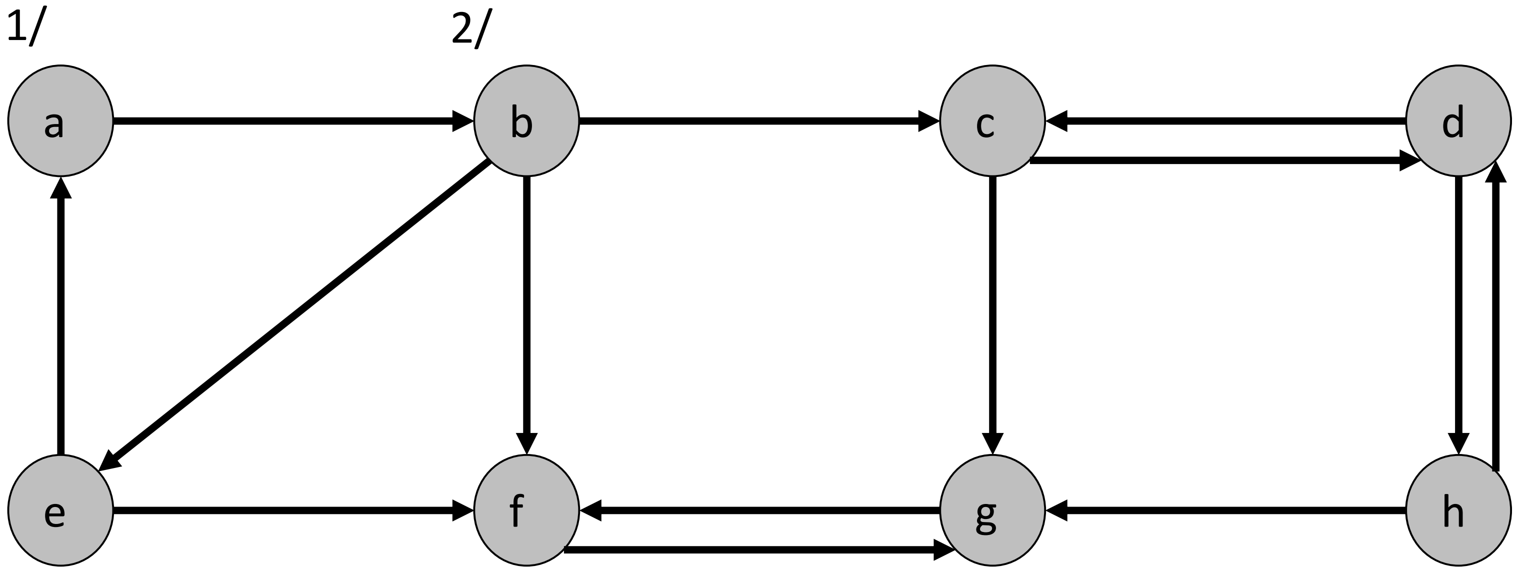
Kosaraju's algorithm

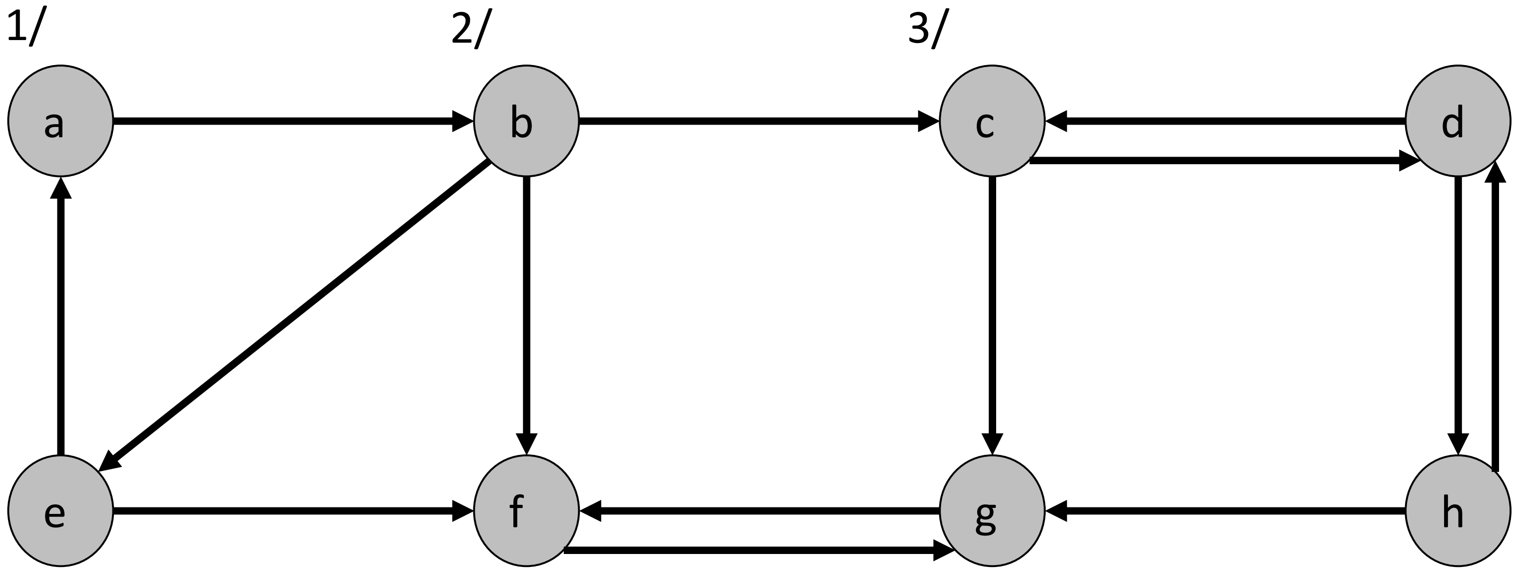
- Do DFS(G) and find finish time for each vertex u
- Build G^T (reverse direction on all edges)
- For node v with label $n, n-1, \dots, 1$:
  - Find all reachable nodes from v and group them as SCC

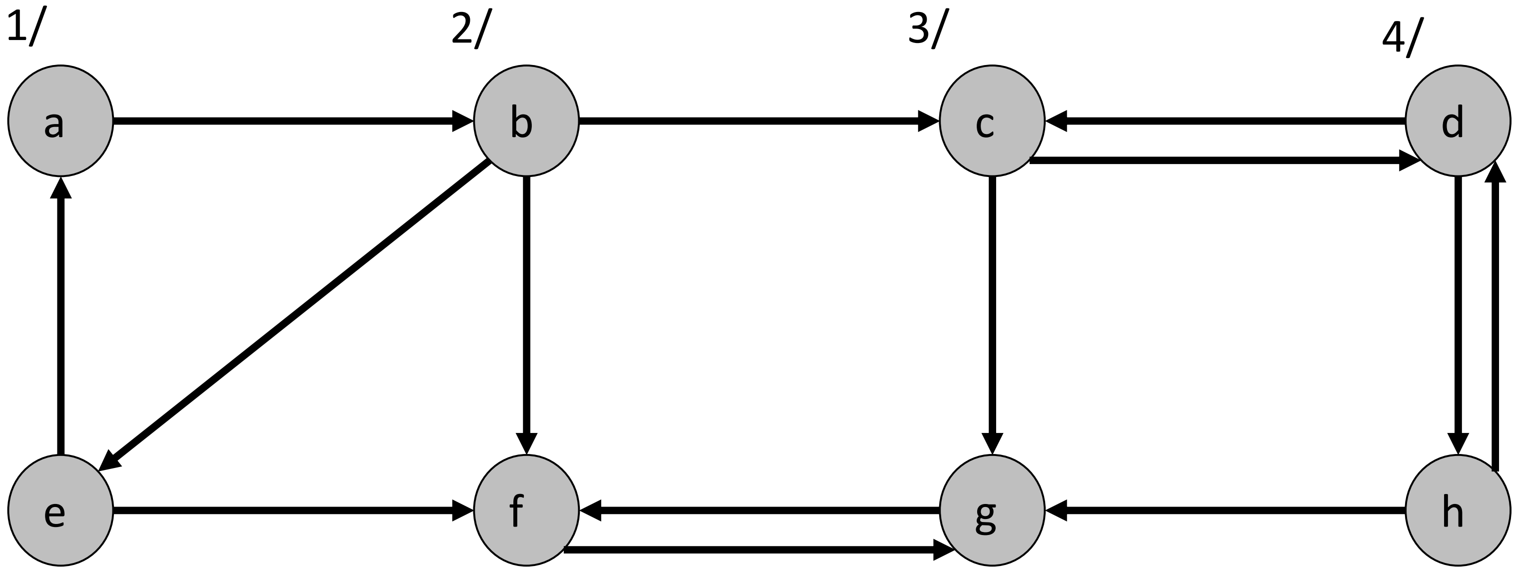
- Two graph traversals are performed
- Timing complexity: $\Theta(V+E)$

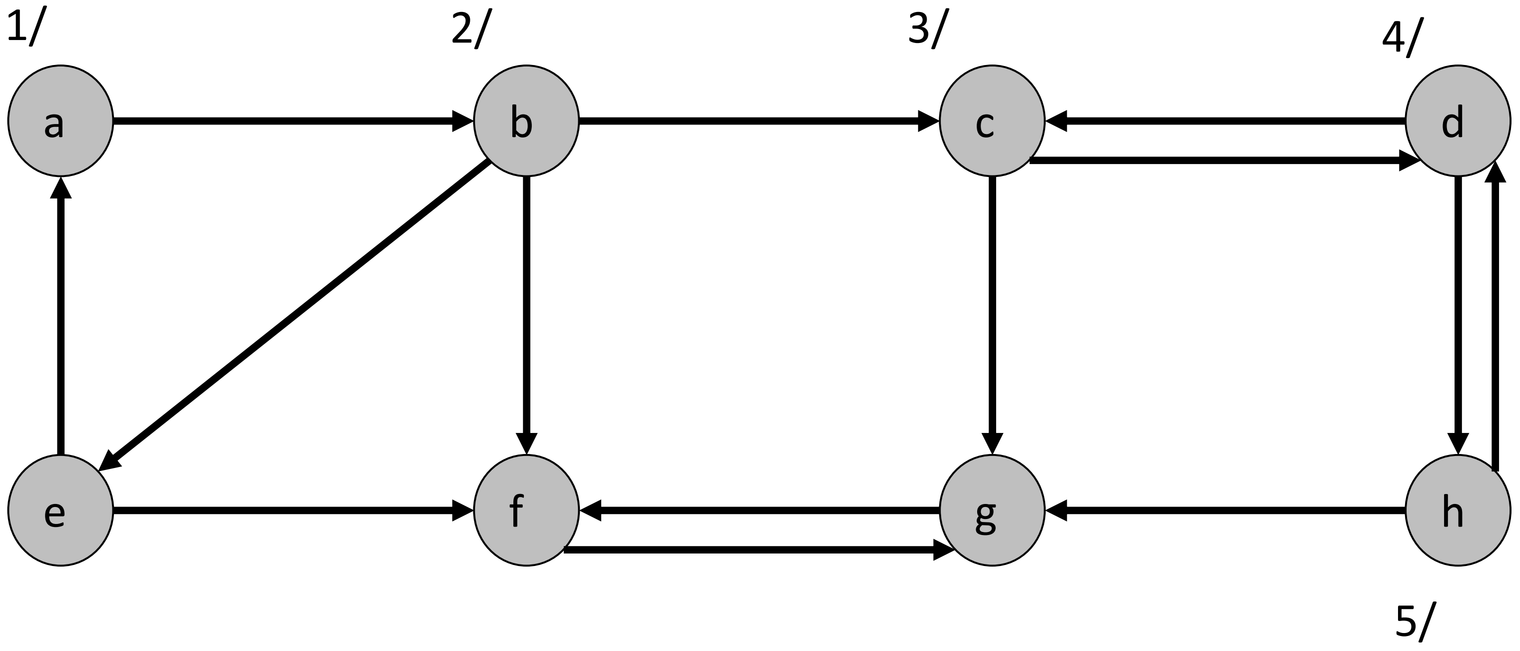
Example

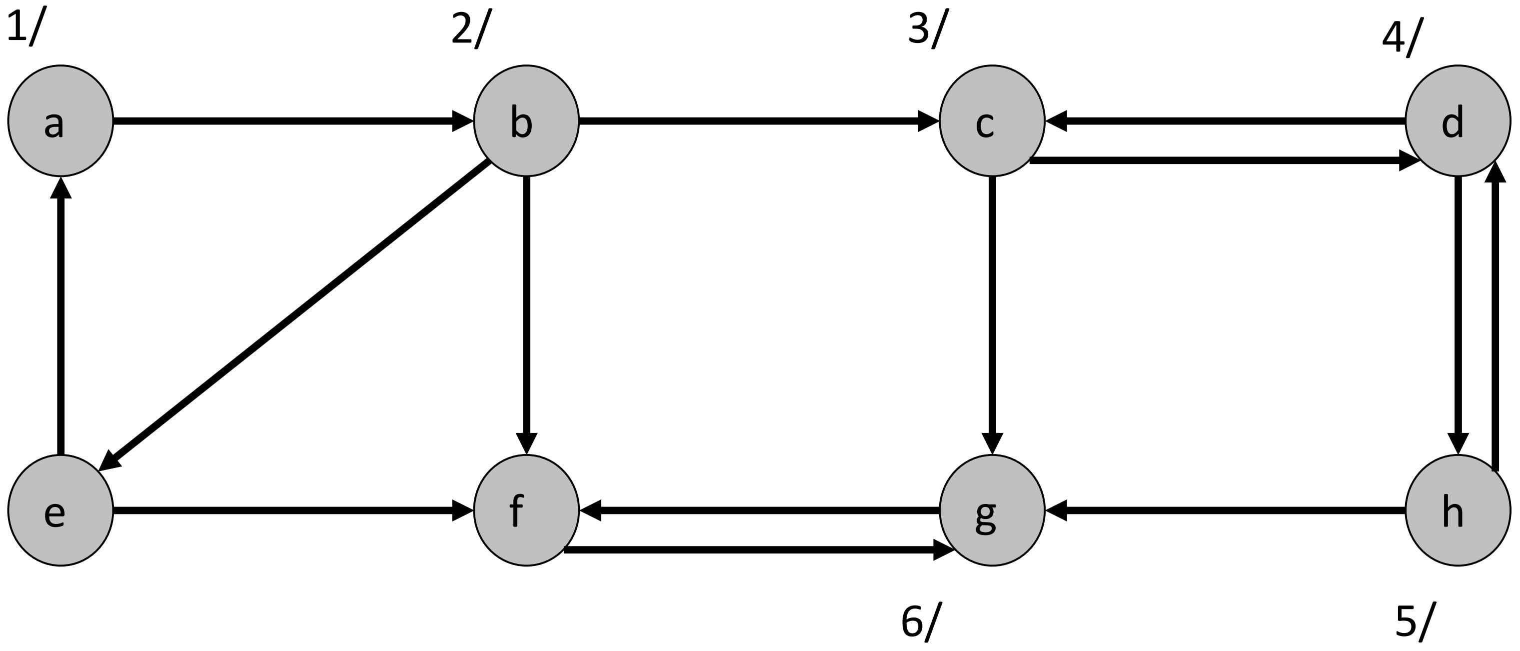


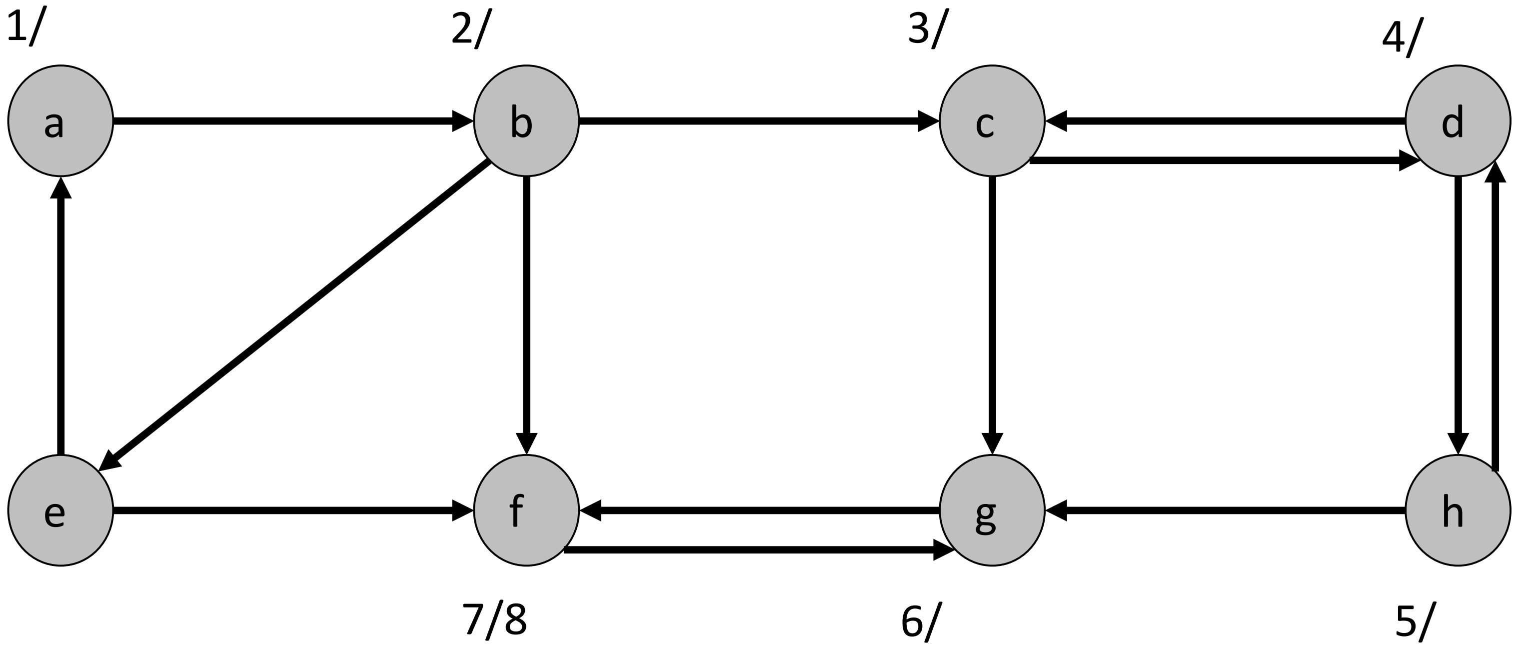

1/
2/
a
b
c
d
e
f
g
h

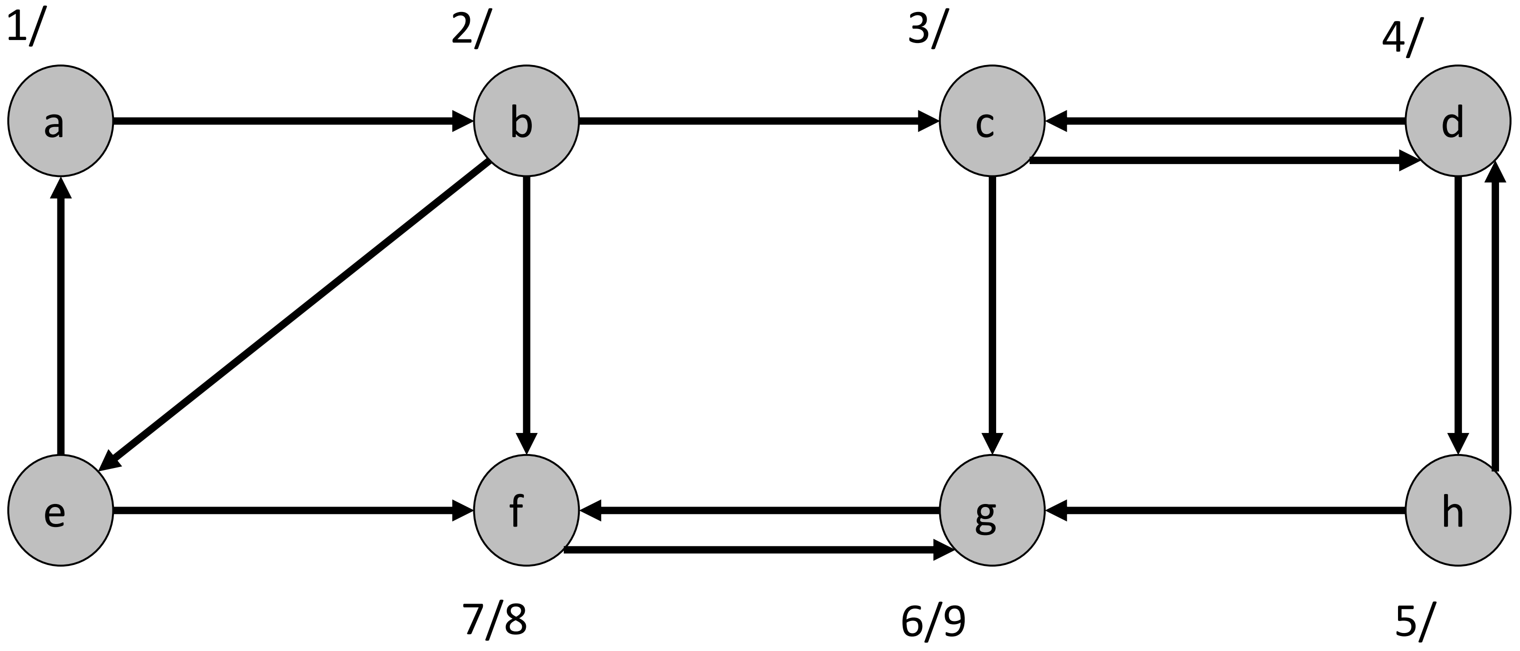

1/
2/
3/
a
b
c
d
e
f
g
h

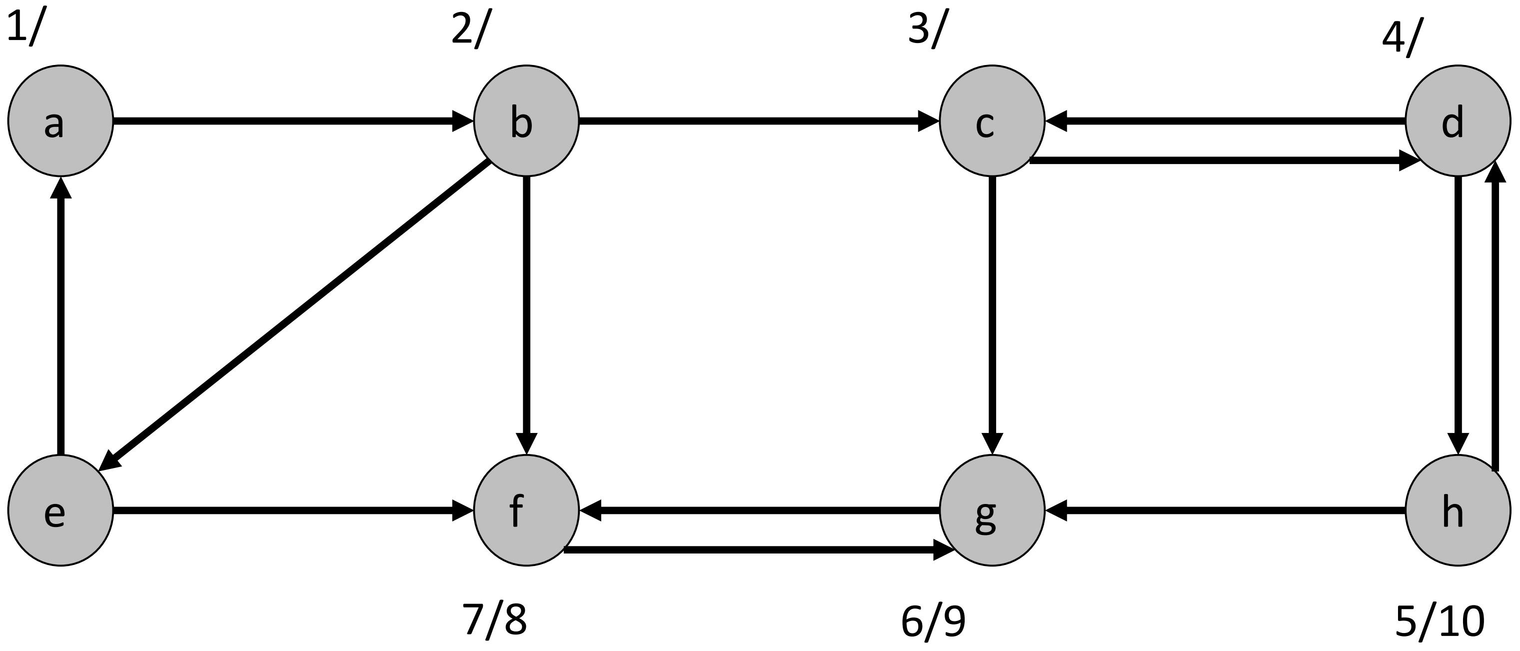

1/
2/
3/
4/
a
b
c
d
e
f
g
h

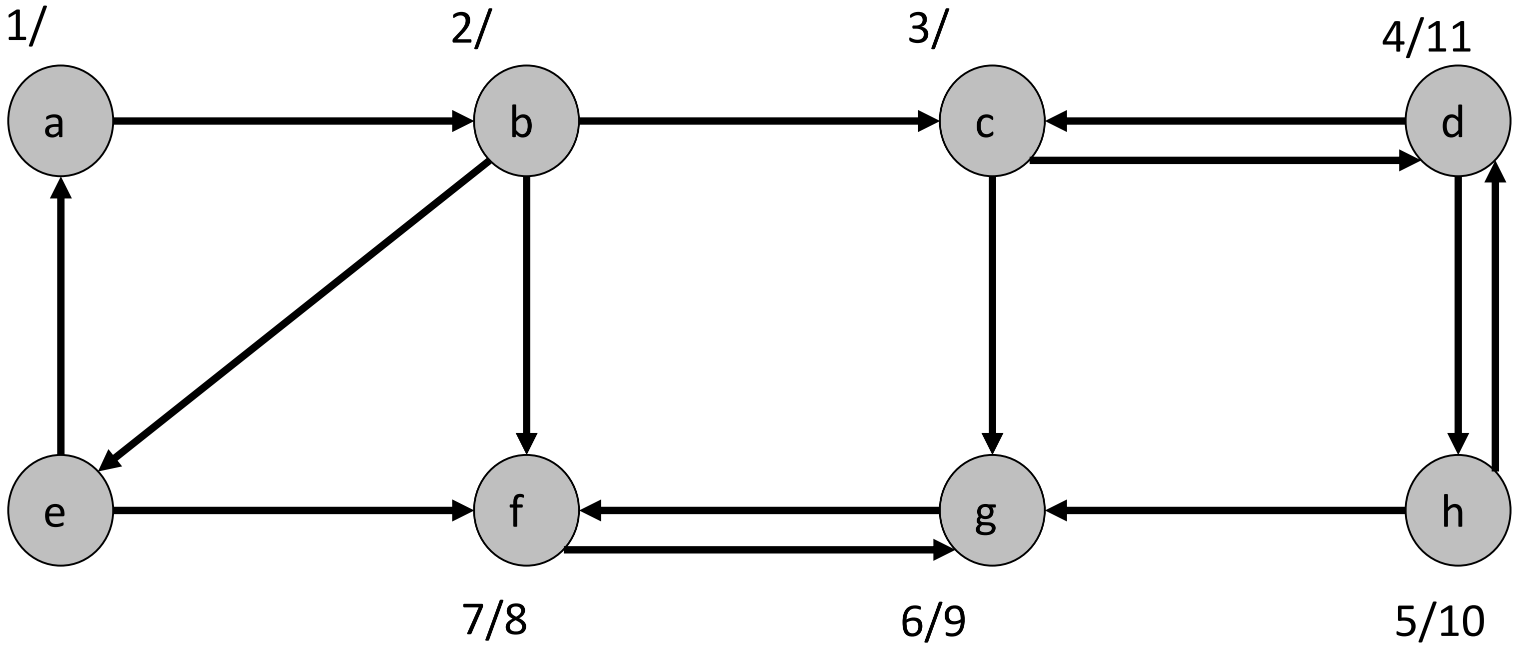

1/
2/
3/
4/
a
b
c
d
e
f
g
h
5/

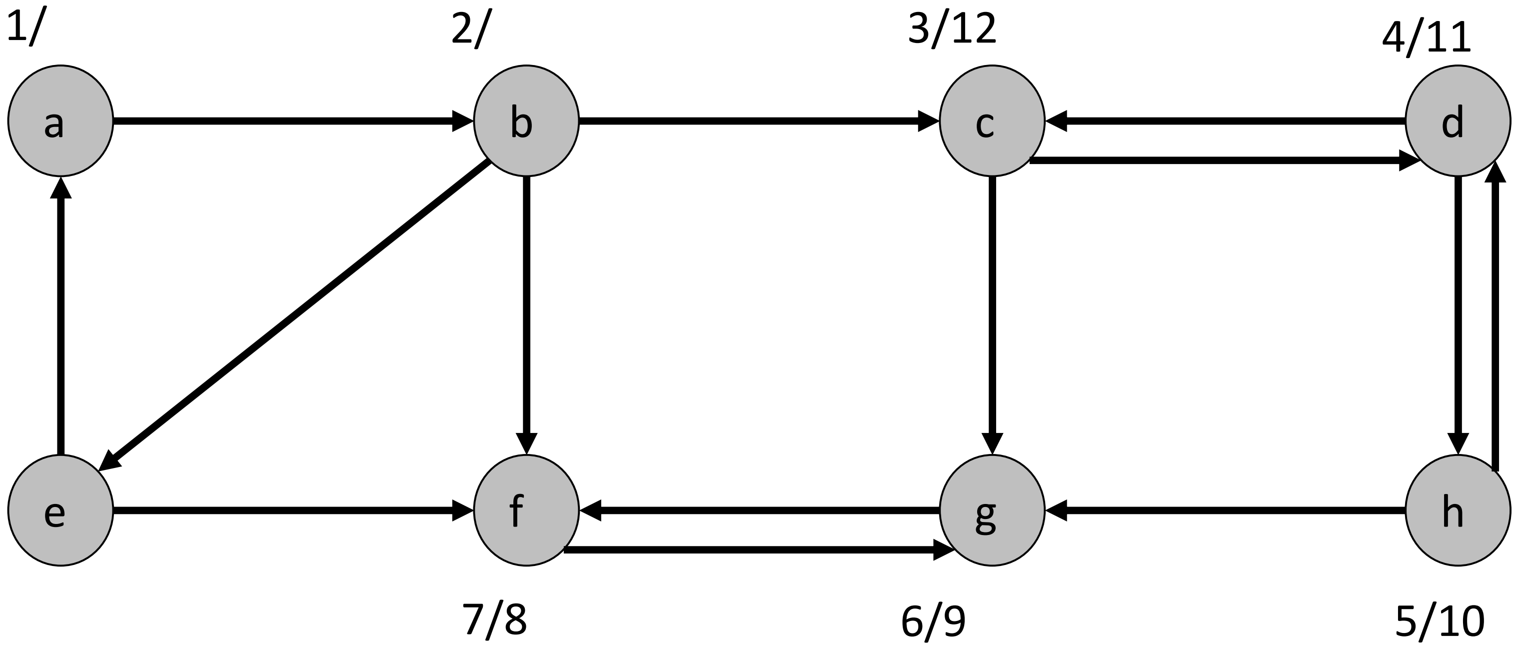

1/
2/
3/
4/
a
b
c
d
e
f
g
h
6/
5/

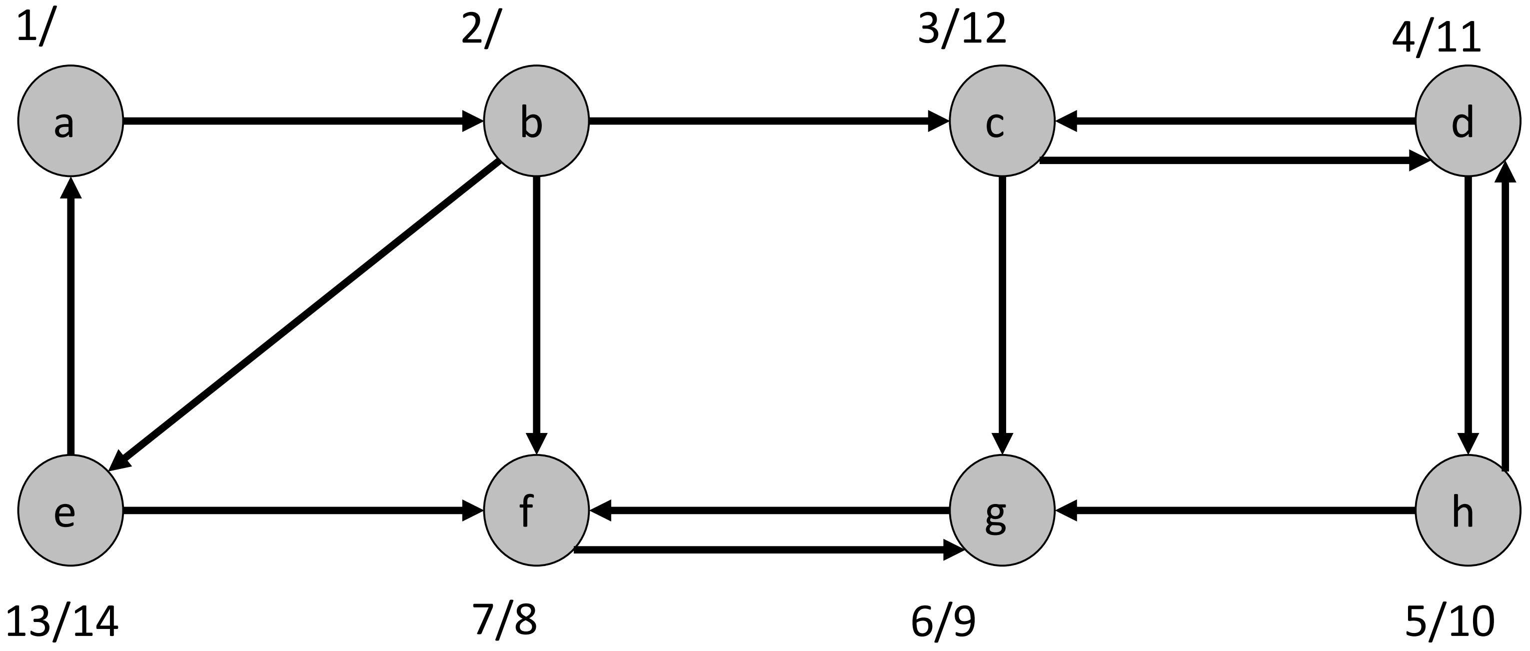

1/
2/
3/
4/
a
b
c
d
e
f
g
h
7/8
6/
5/

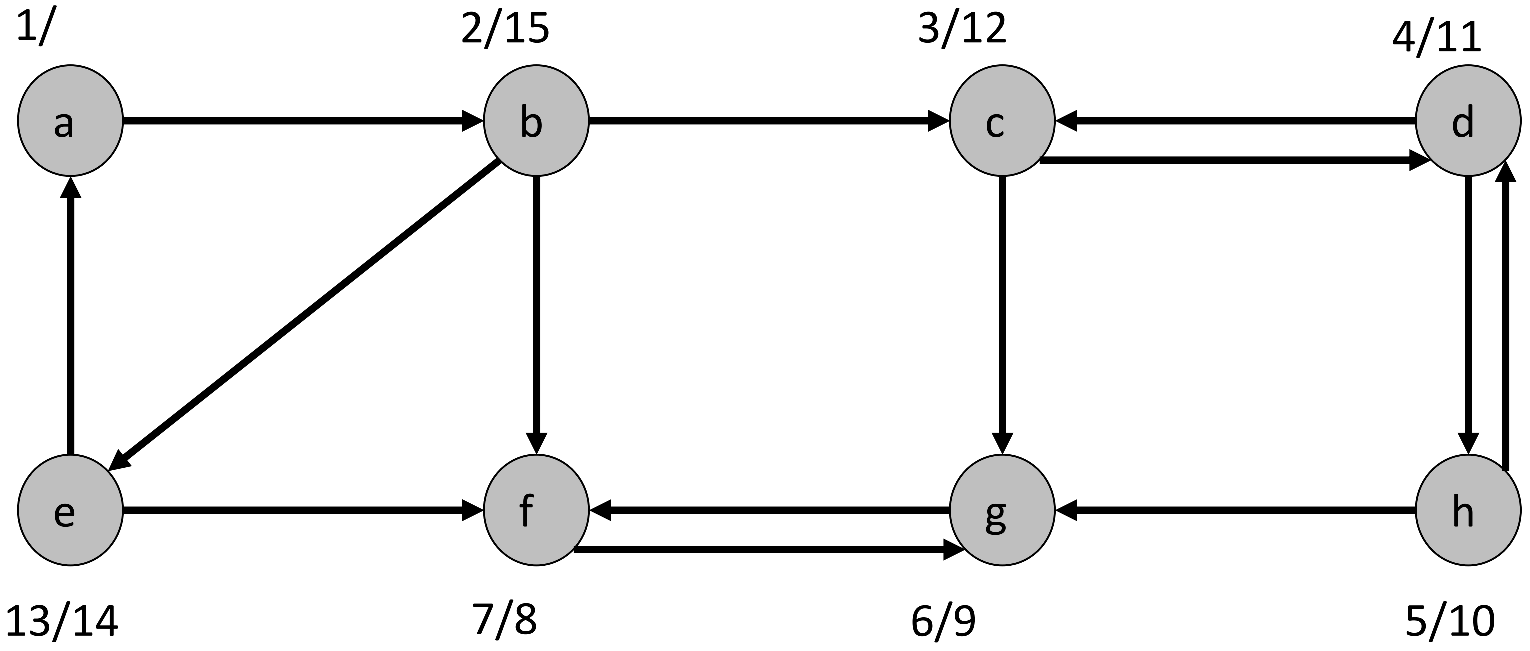

1/
2/
3/
4/
a
b
c
d
e
f
g
h
7/8
6/9
5/

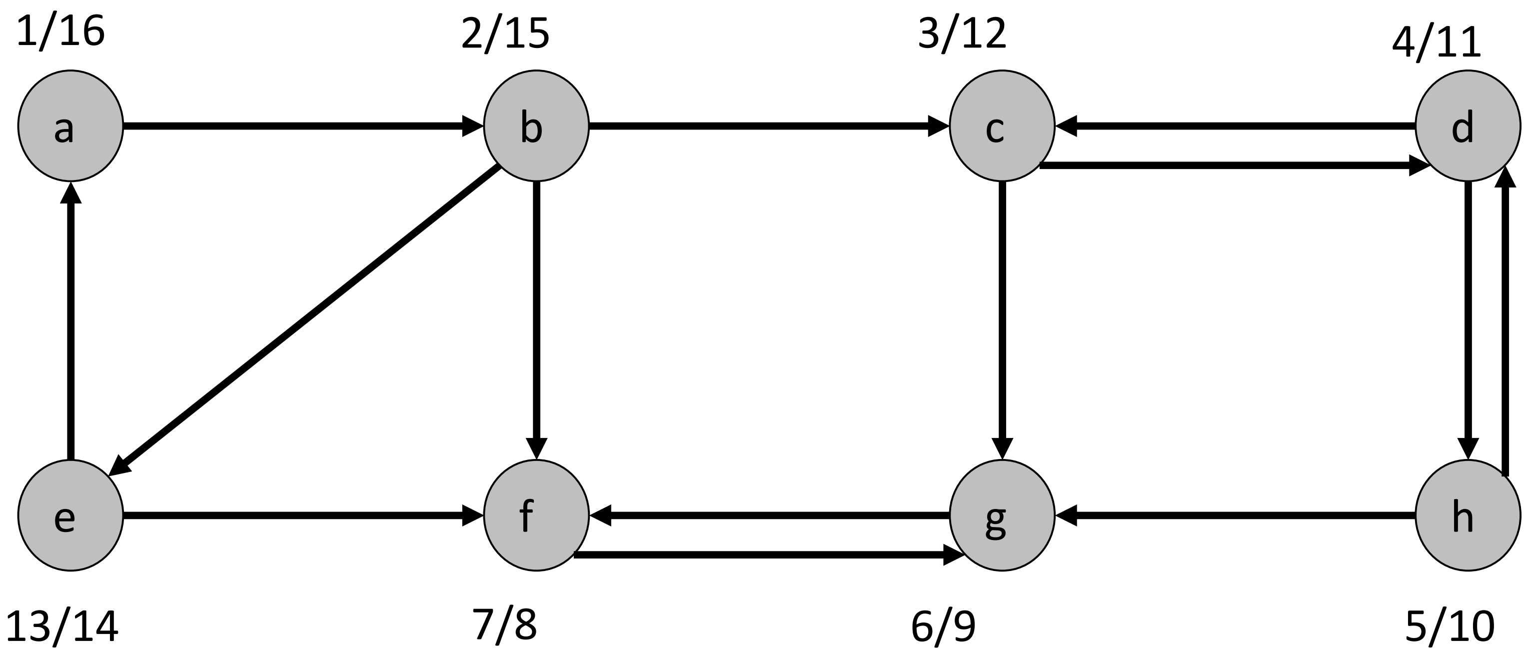

1/
2/
3/
4/
a
b
c
d
e
f
g
h
7/8
6/9
5/10

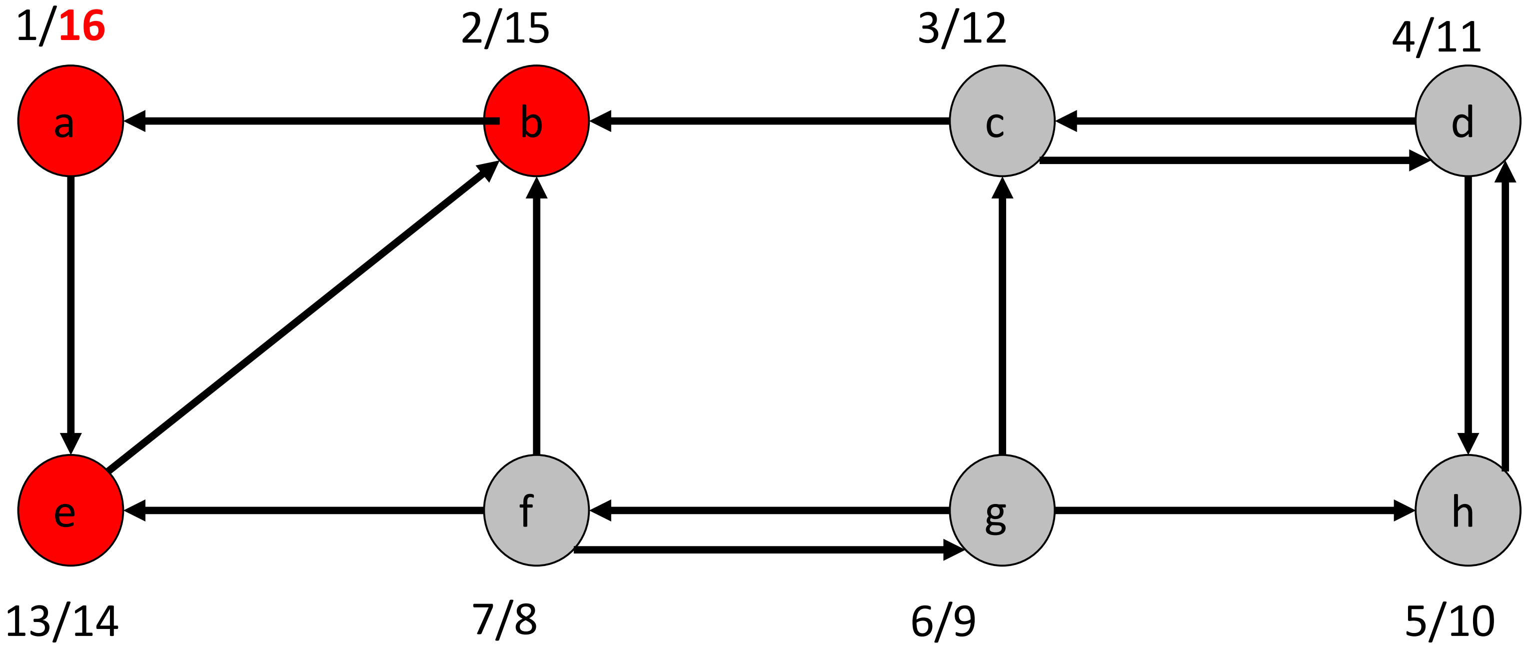

1/
2/
3/
4/11
a
b
c
d
e
f
g
h
7/8
6/9
5/10


1/
2/
3/12
4/11
a
b
c
d
e
f
g
h
7/8
6/9
5/10

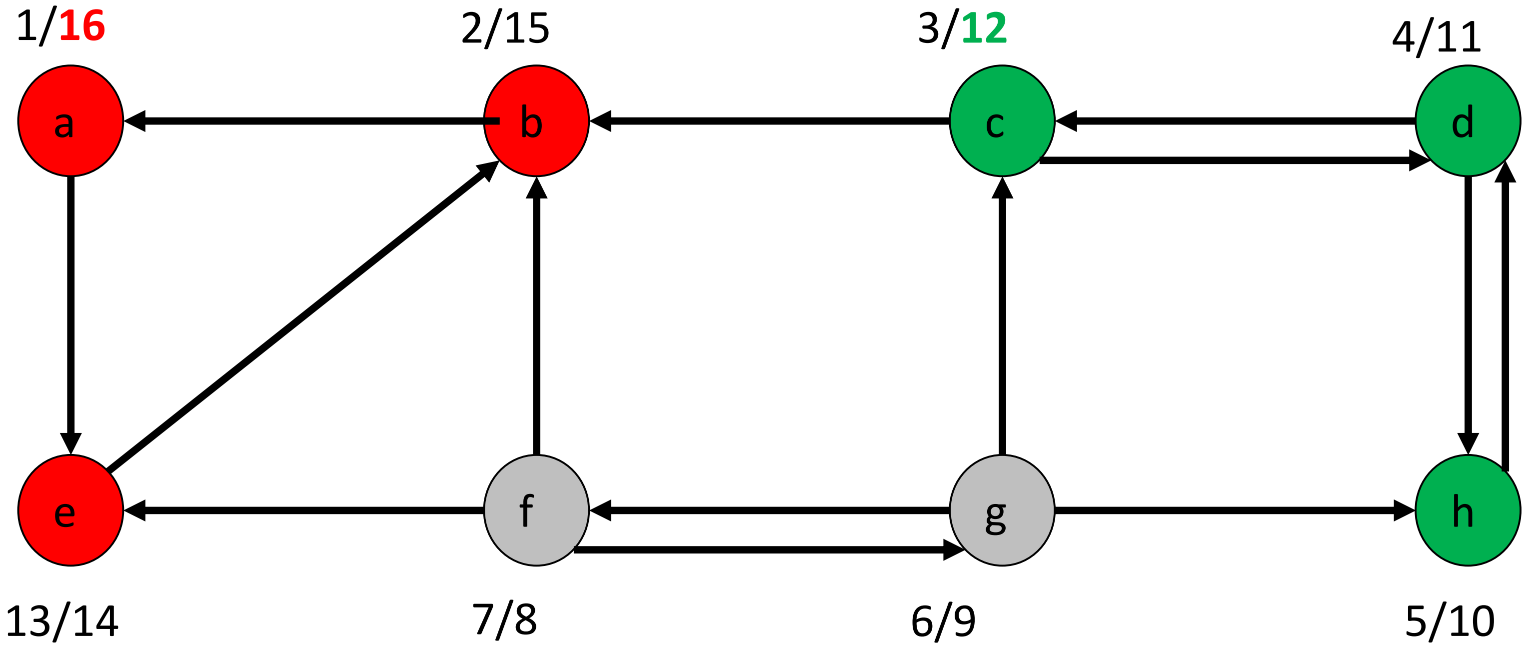

1/
2/
3/12
4/11
a
b
c
d
e
f
g
h
13/14
7/8
6/9
5/10


1/
2/15
3/12
4/11
a
b
c
d
e
f
g
h
13/14
7/8
6/9
5/10

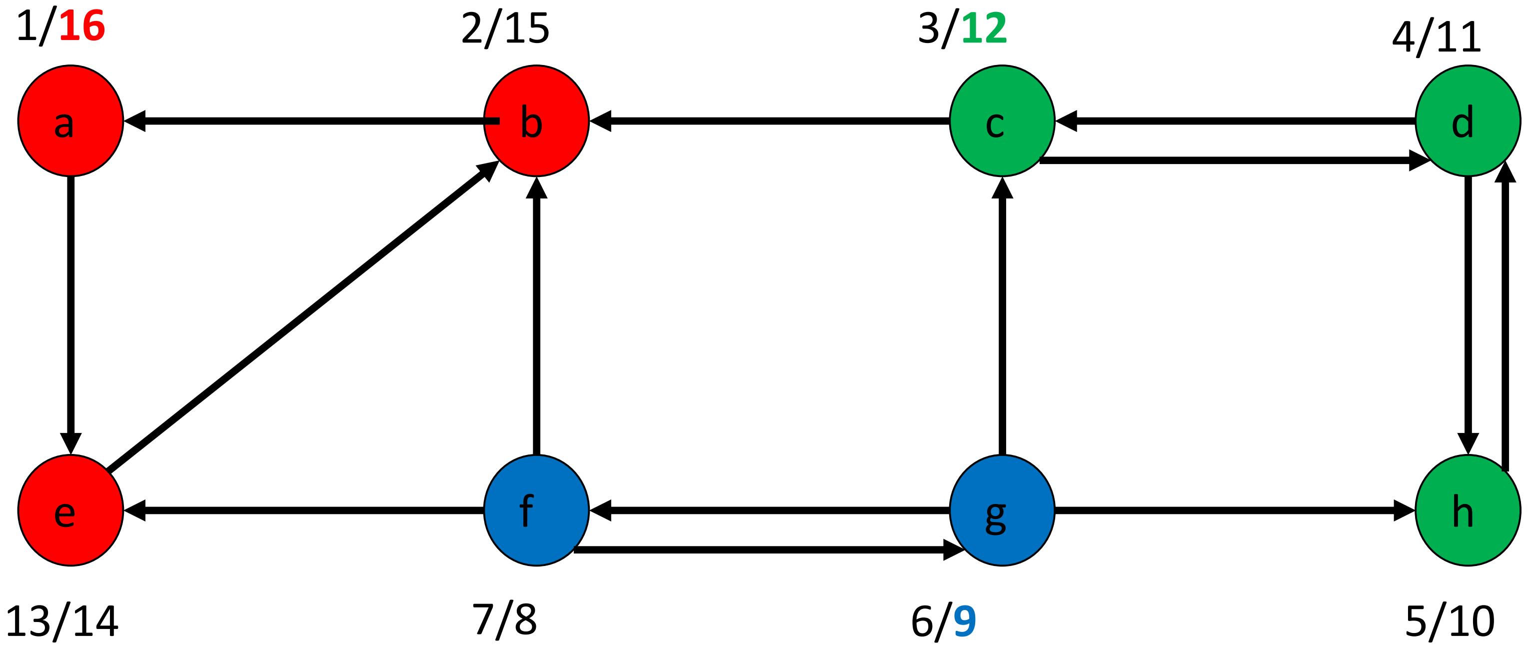

1/16
2/15
3/12
4/11
a
b
c
d
e
f
g
h
13/14
7/8
6/9
5/10



Note edge direction change



1/16
2/15
3/12
4/11
a
b
c
d
e
f
g
h
13/14
7/8
6/9
5/10


1/16
2/15
3/12
4/11
a
b
c
d
e
f
g
h
13/14
7/8
6/9
5/10

Tarjan’s algorithm for SCC

- Mark the id of each node as unvisited
- Start DFS. Upon visiting a node assign it as id and a low-link value*. Mark current nodes as visited and add them to a seen stack
- On DFS callback, if the previous node is on the stack then min the current node’s low-link value with the last node’s low-link value
- After visiting all neighbors, if the current node started a connected component then pop nodes off stack until current node is reached

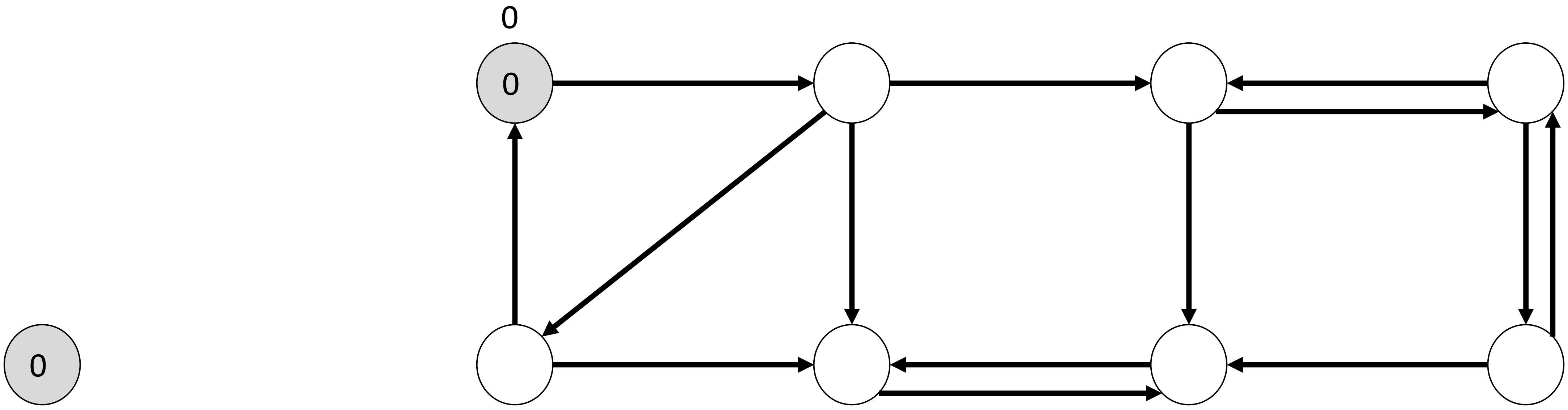
*The low-link value of a node is the smallest node id reachable from that node when doing DFS, including itself

Example

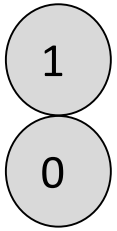
Nodes: Unvisited — Visiting neighbors — Visited all neighbors

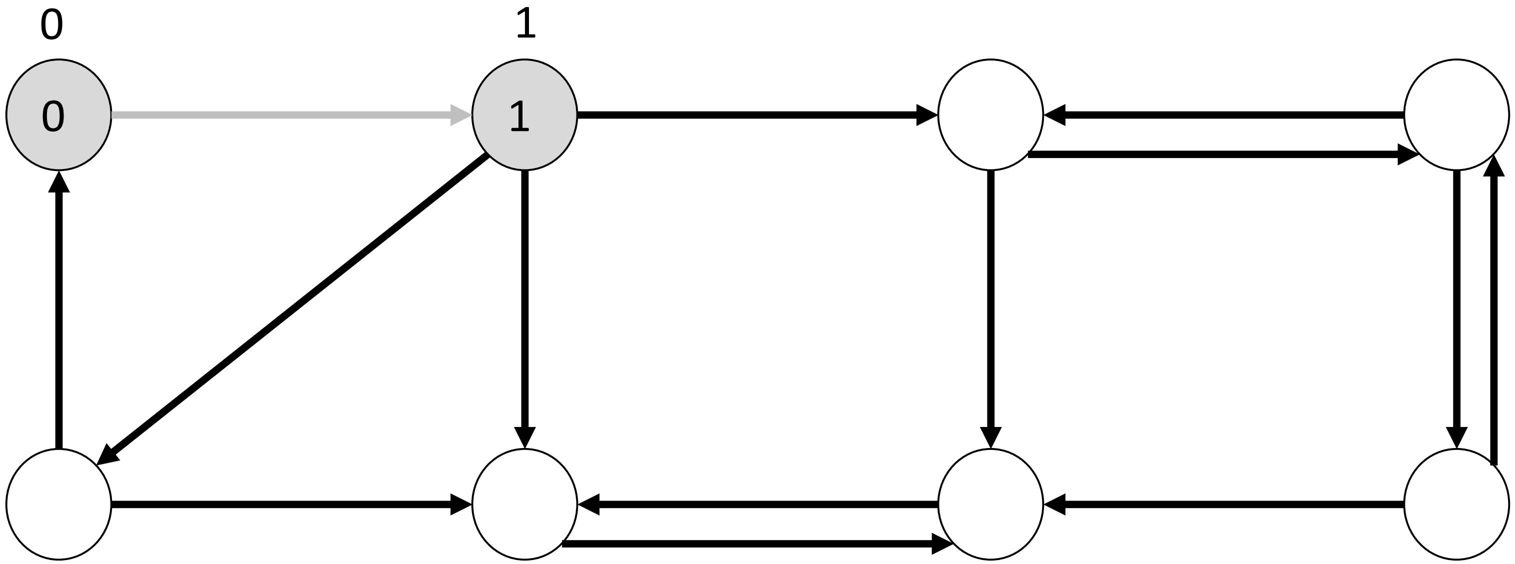
Edges: Unvisited — Visited

Stack

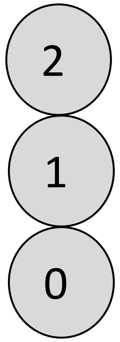


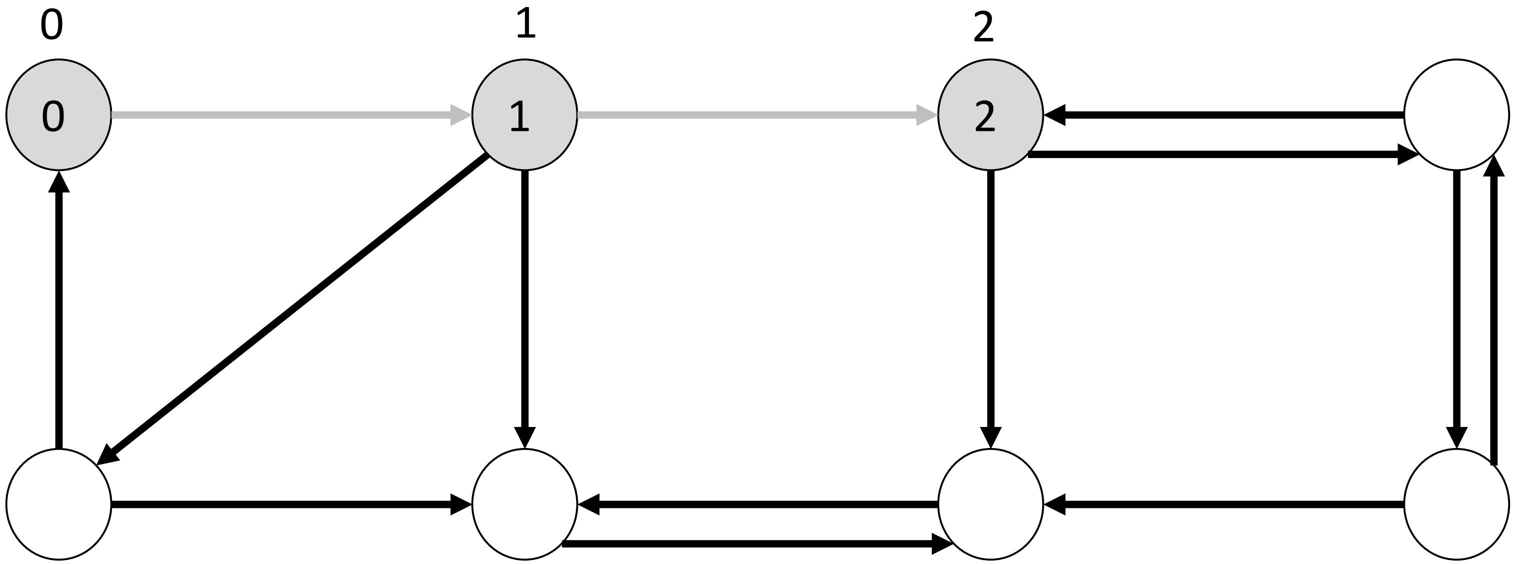
Stack



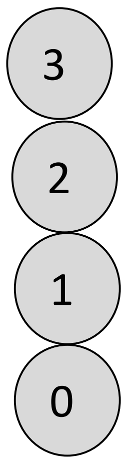


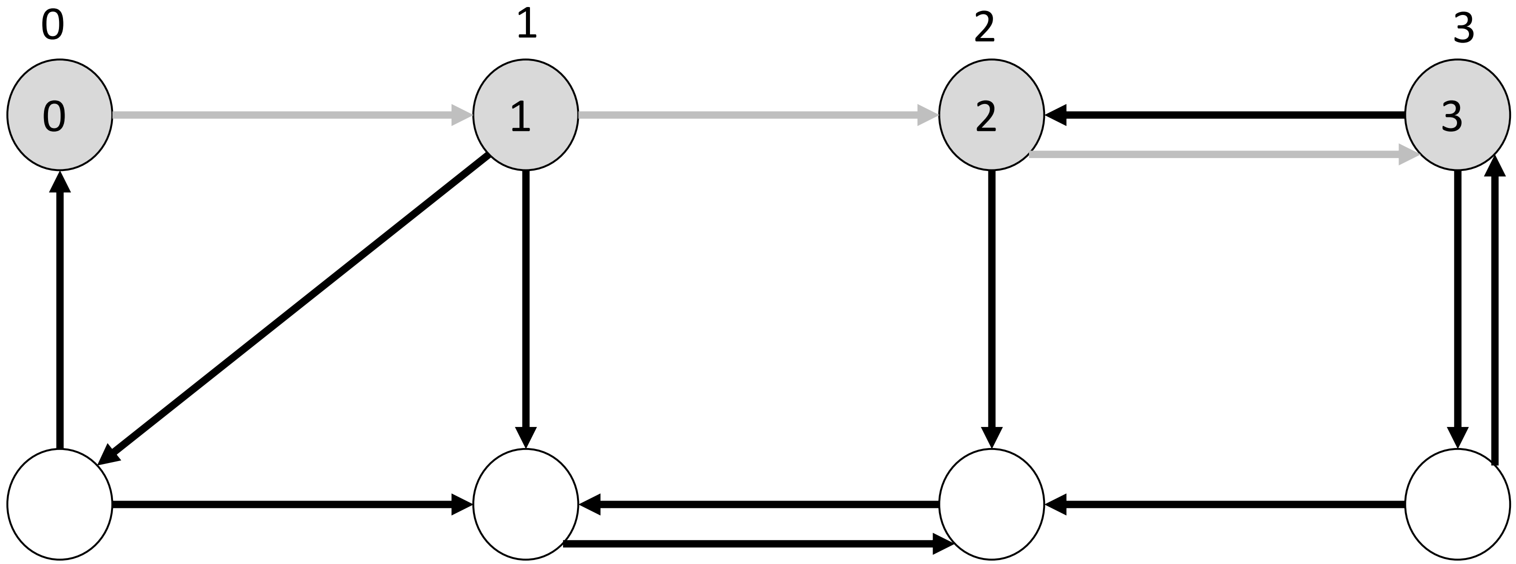
Stack



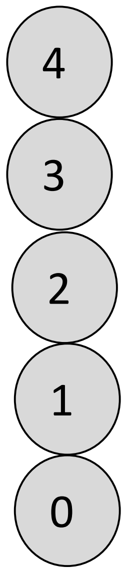


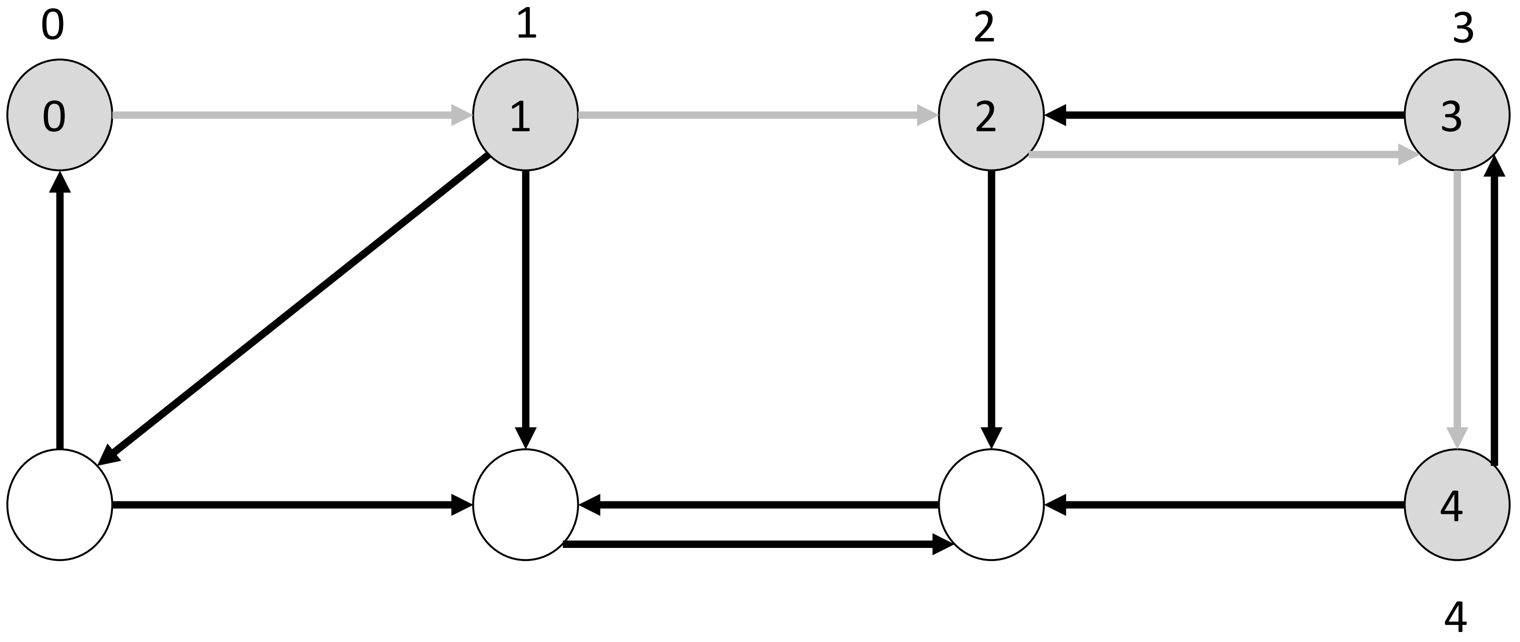
Stack



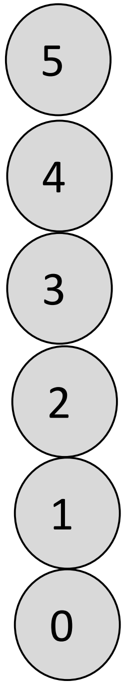


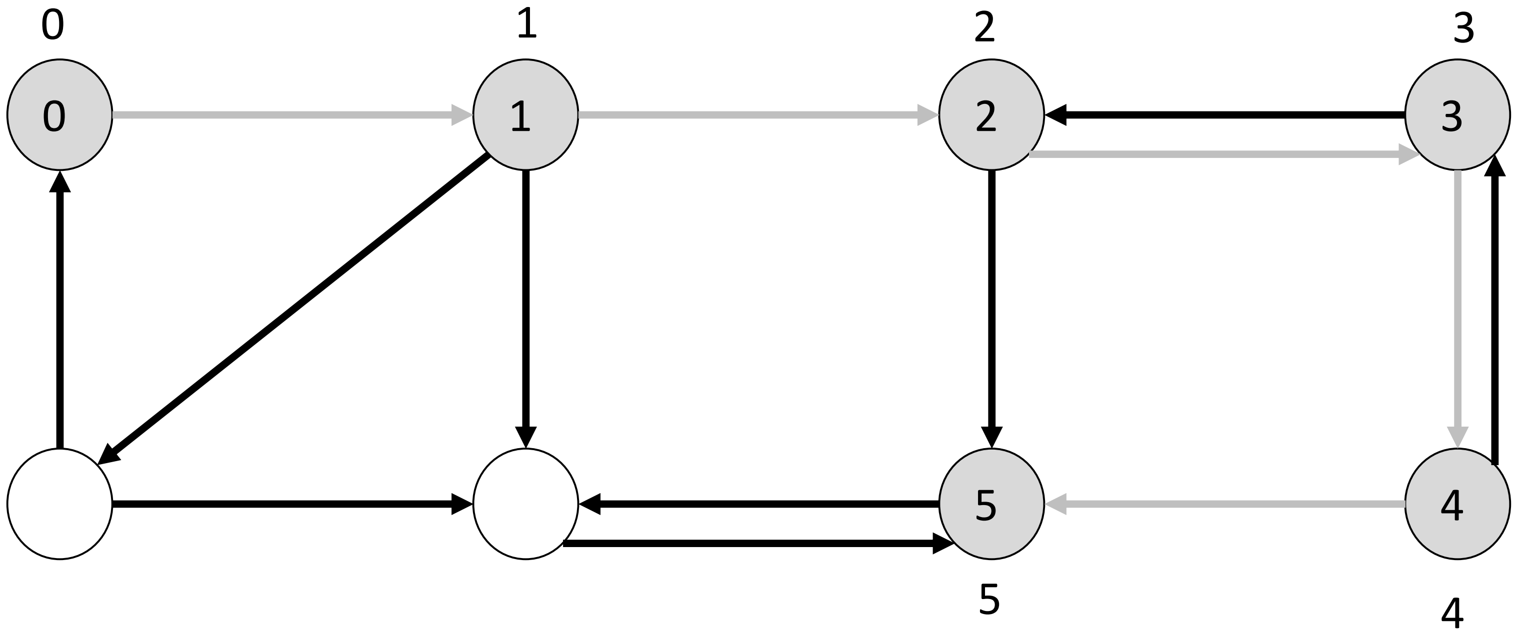
Stack



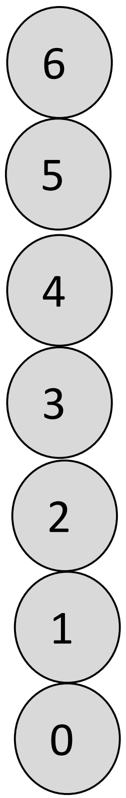


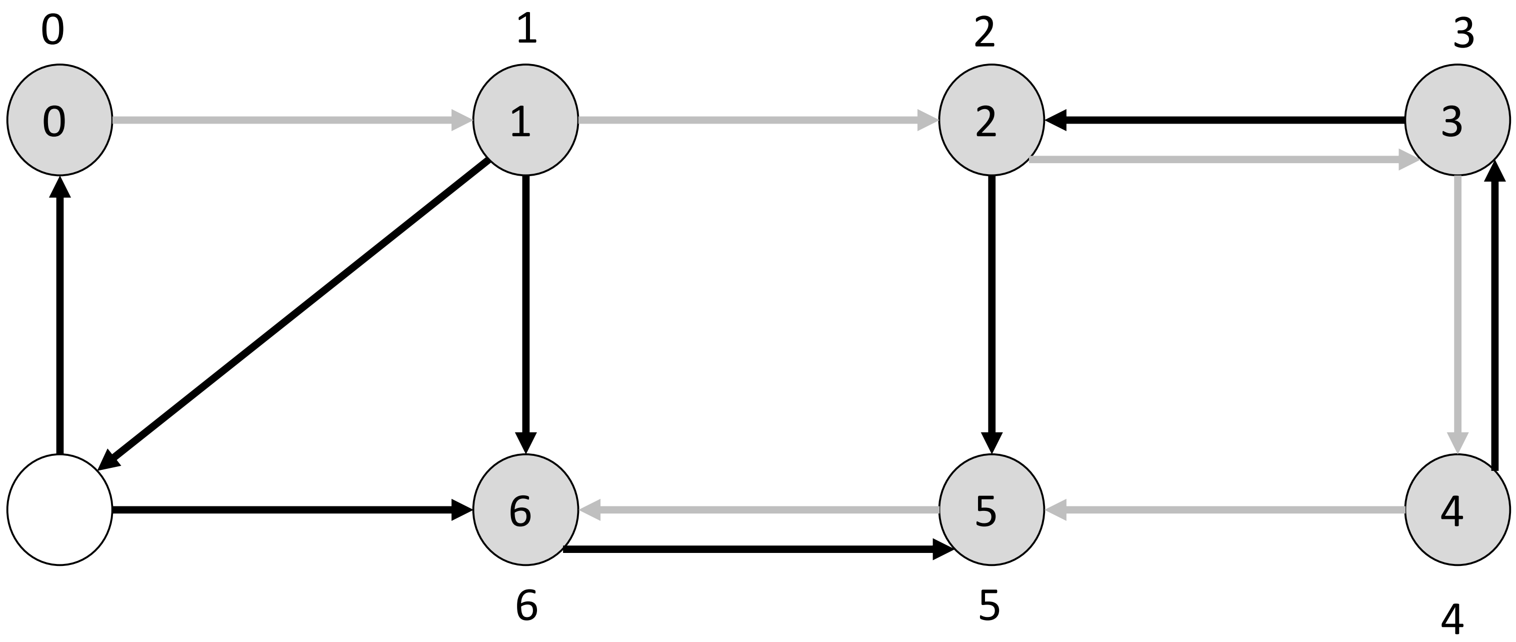
Stack

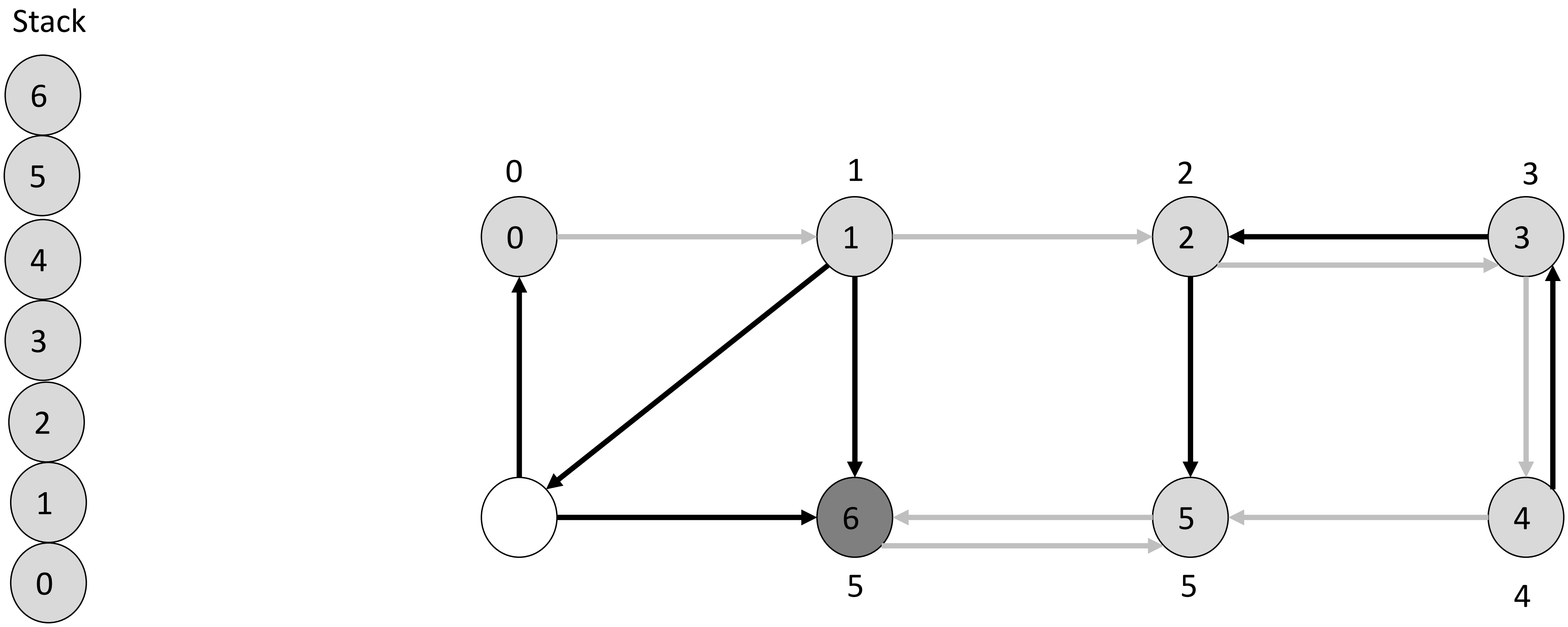




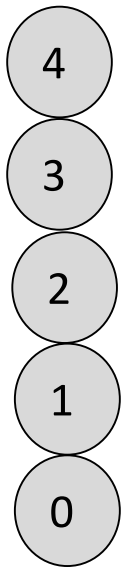
Stack

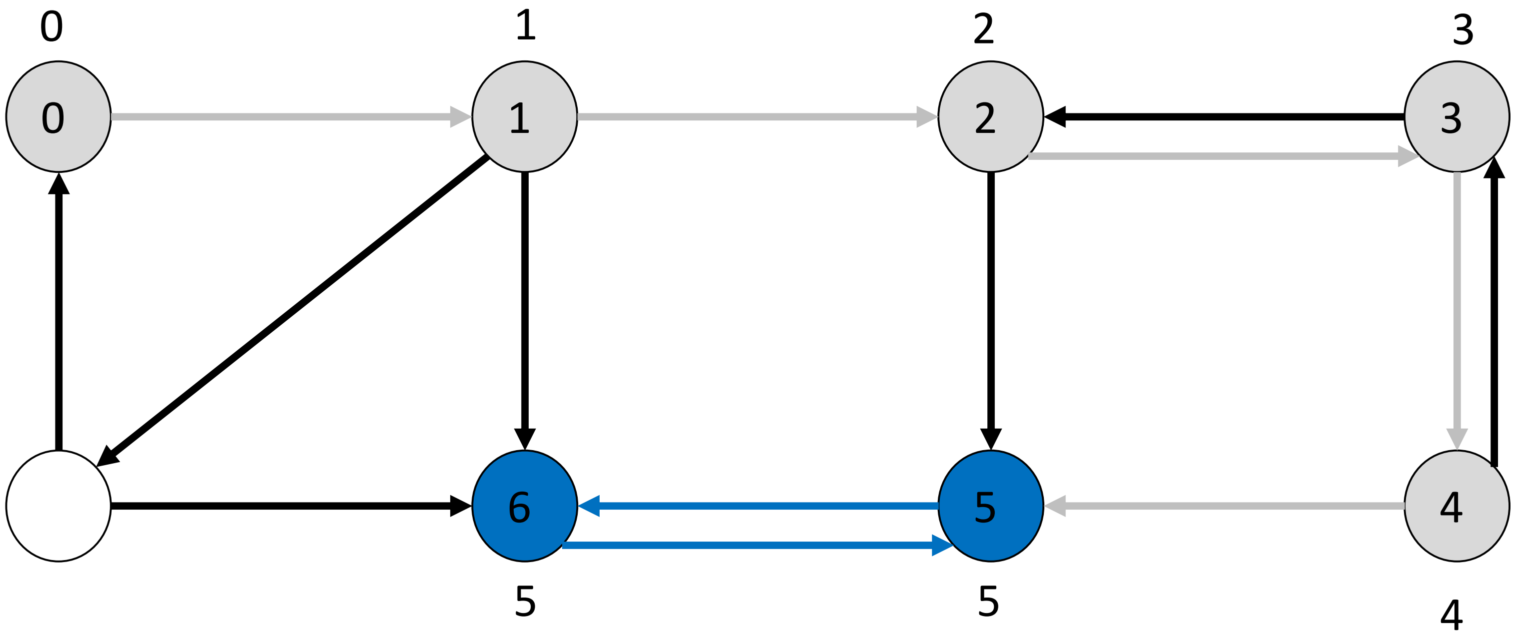





Stack
6
5
4
3
2
1
0
0
1
2
3
0
1
2
3
6
5
4
5
5
4

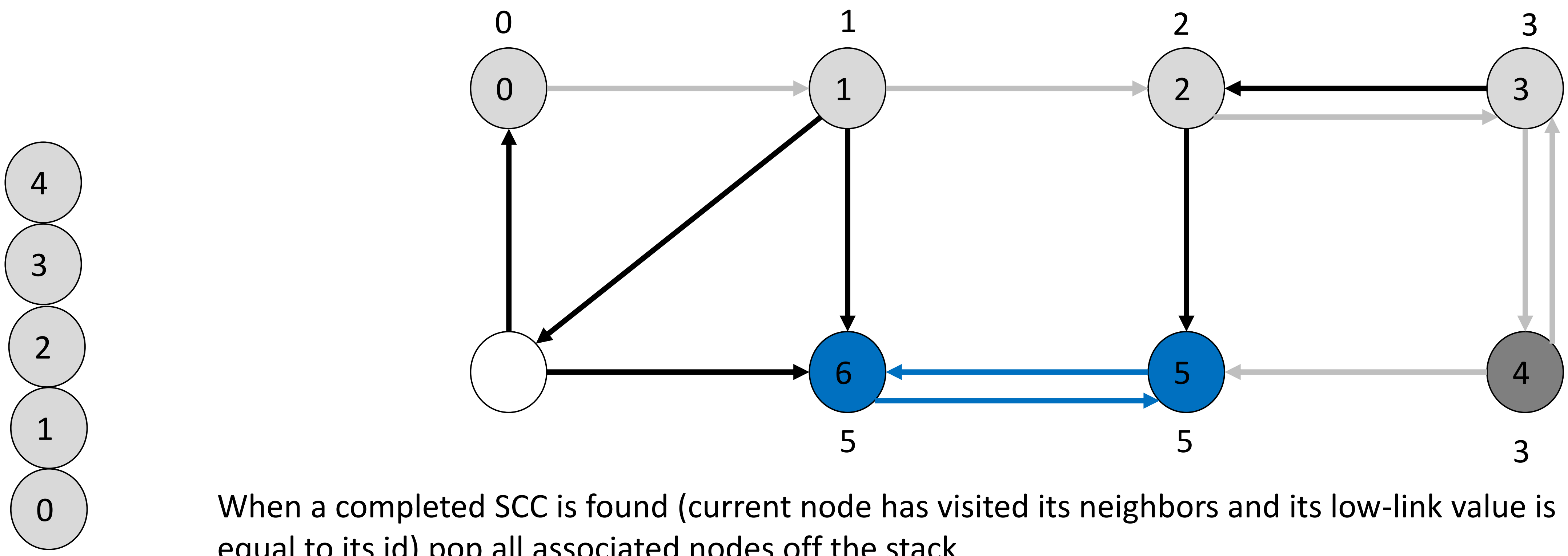
Stack





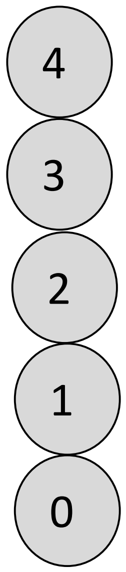
When a completed SCC is found (current node has visited its neighbors and its low-link value is equal to its id) pop all associated nodes off the stack

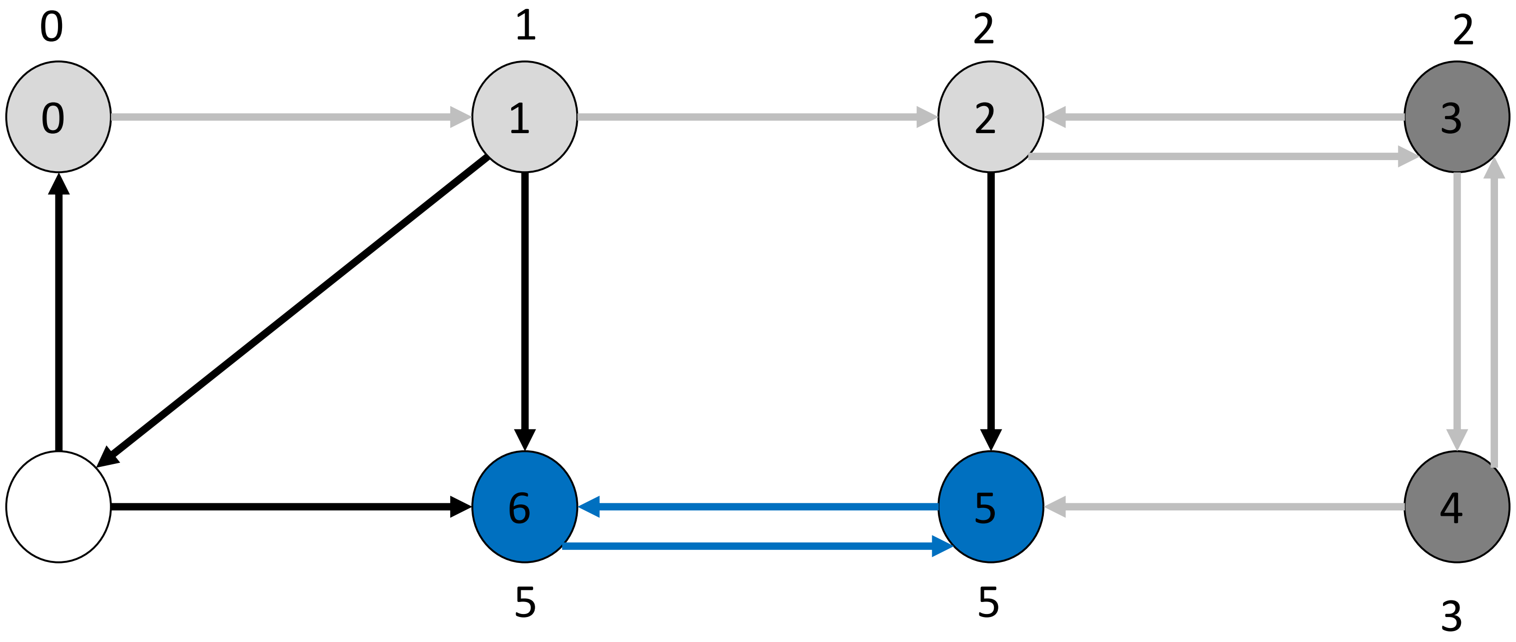
Stack



When a completed SCC is found (current node has visited its neighbors and its low-link value is equal to its id) pop all associated nodes off the stack

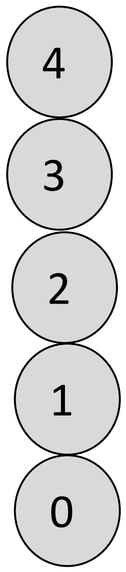
Stack

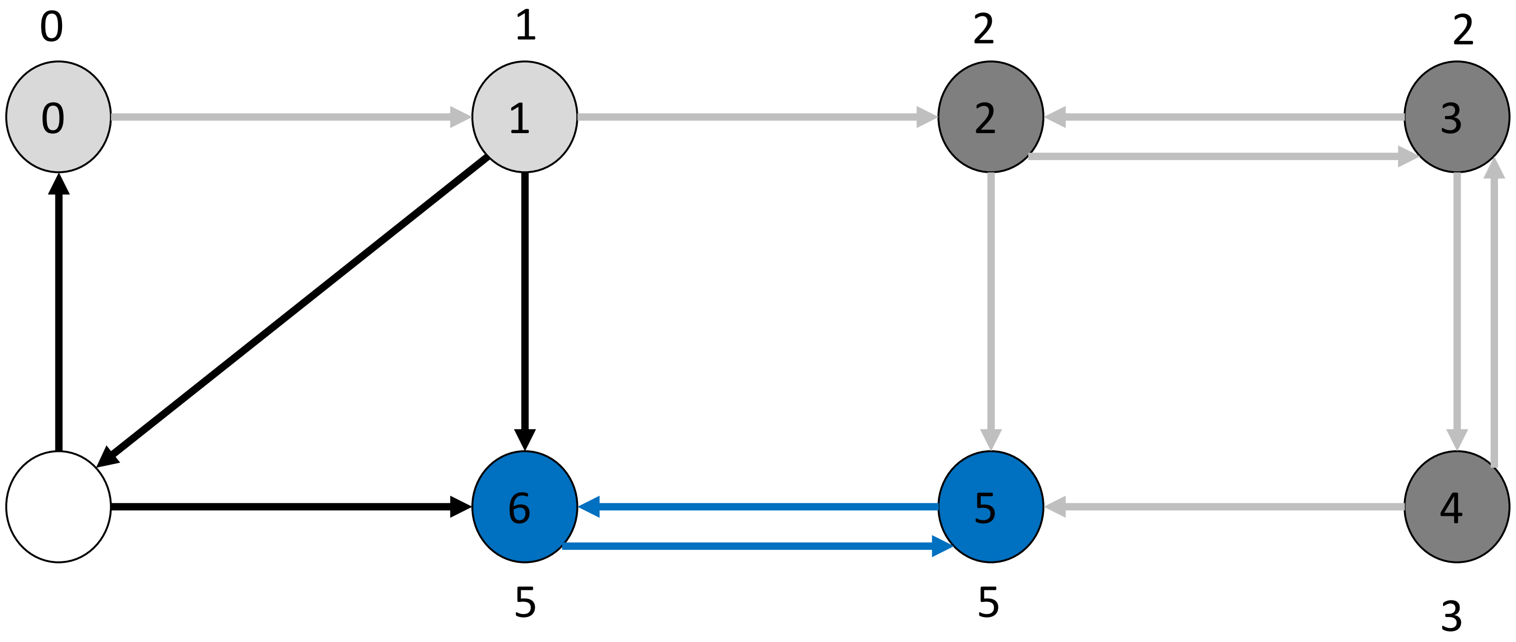




When a completed SCC is found (current node has visited its neighbors and its low-link value is equal to its id) pop all associated nodes off the stack

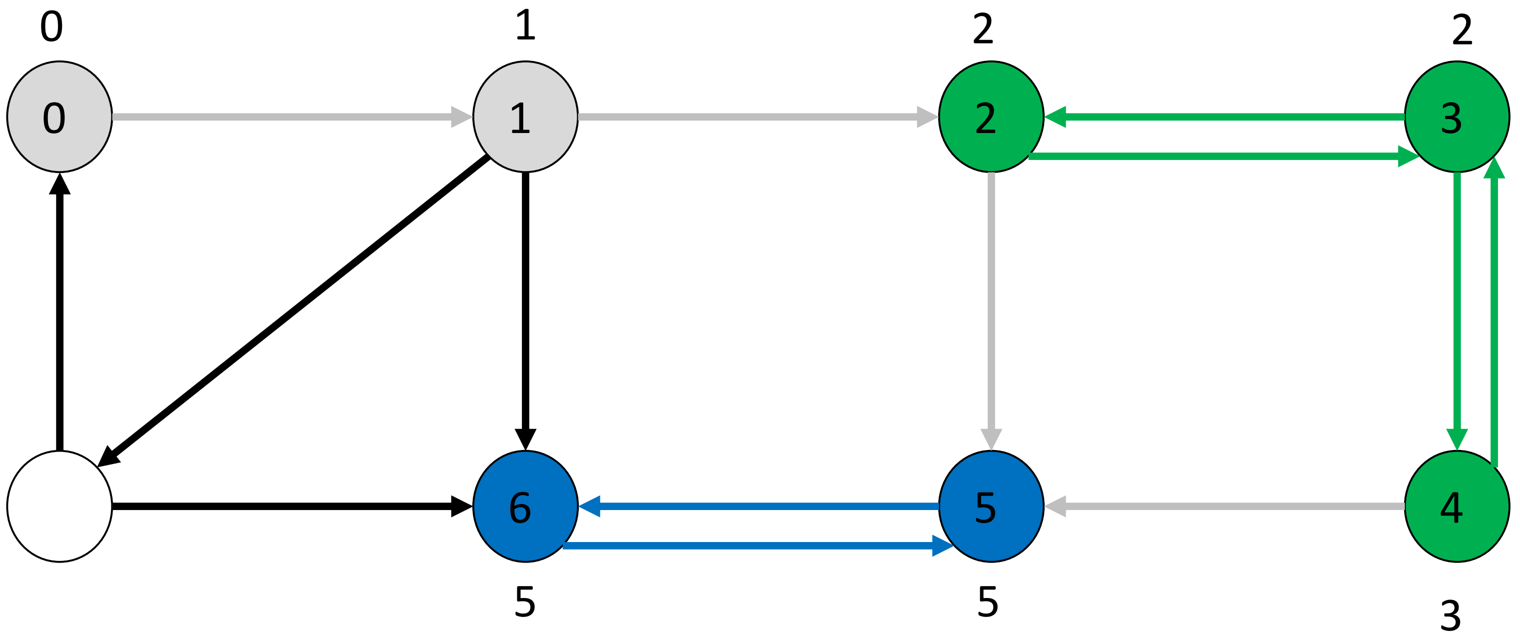
Stack

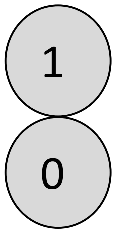




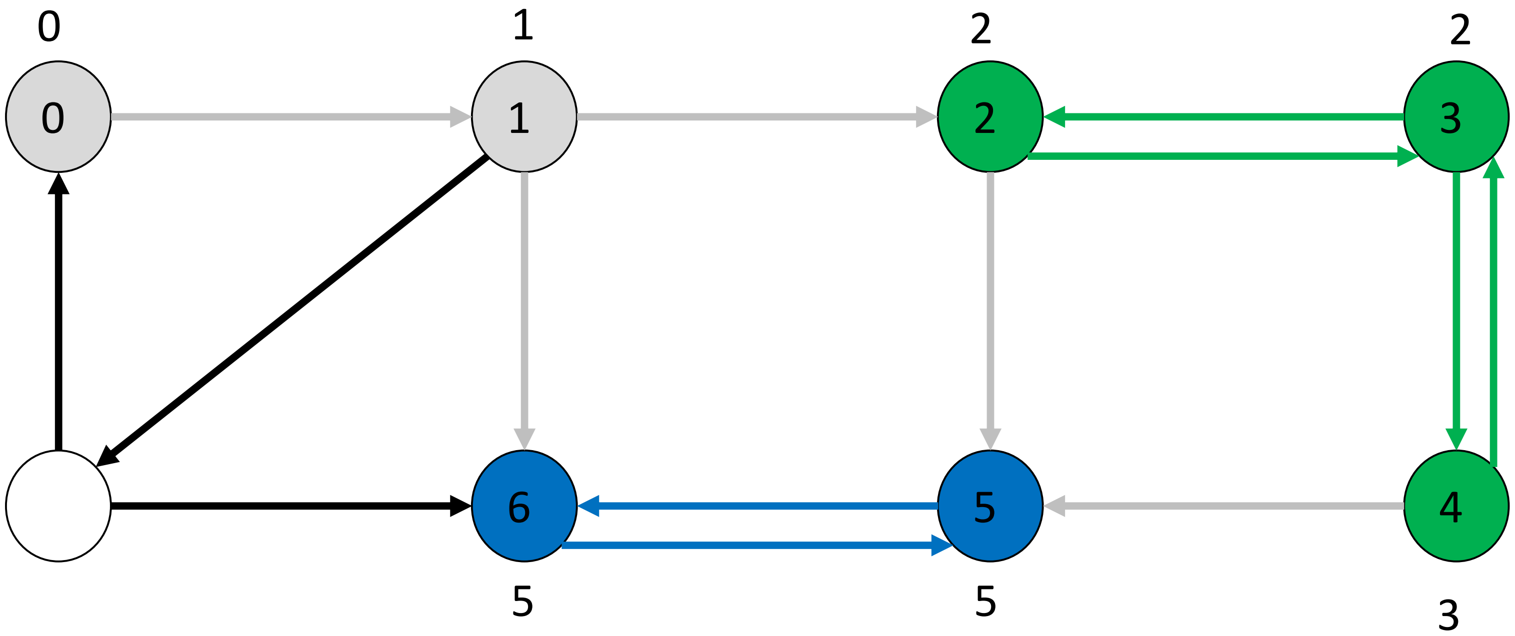
When a completed SCC is found (current node has visited its neighbors and its low-link value is equal to its id) pop all associated nodes off the stack

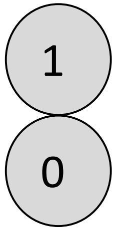
Stack



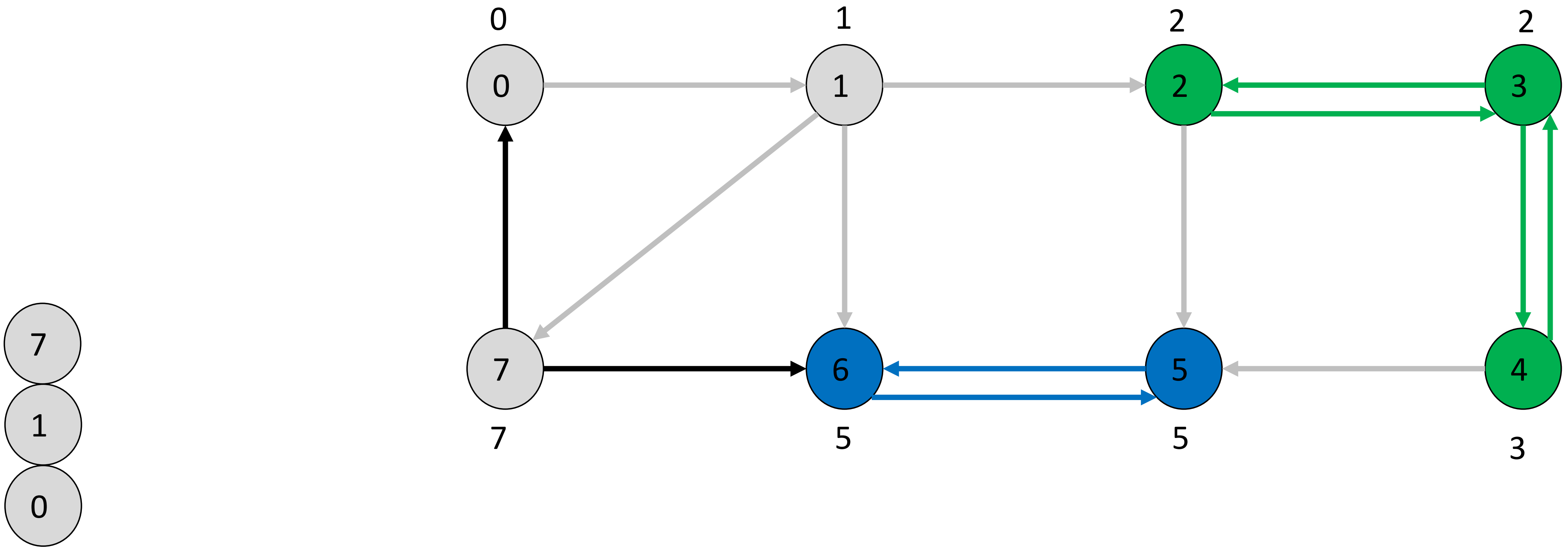


Stack

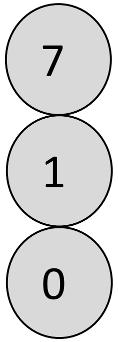


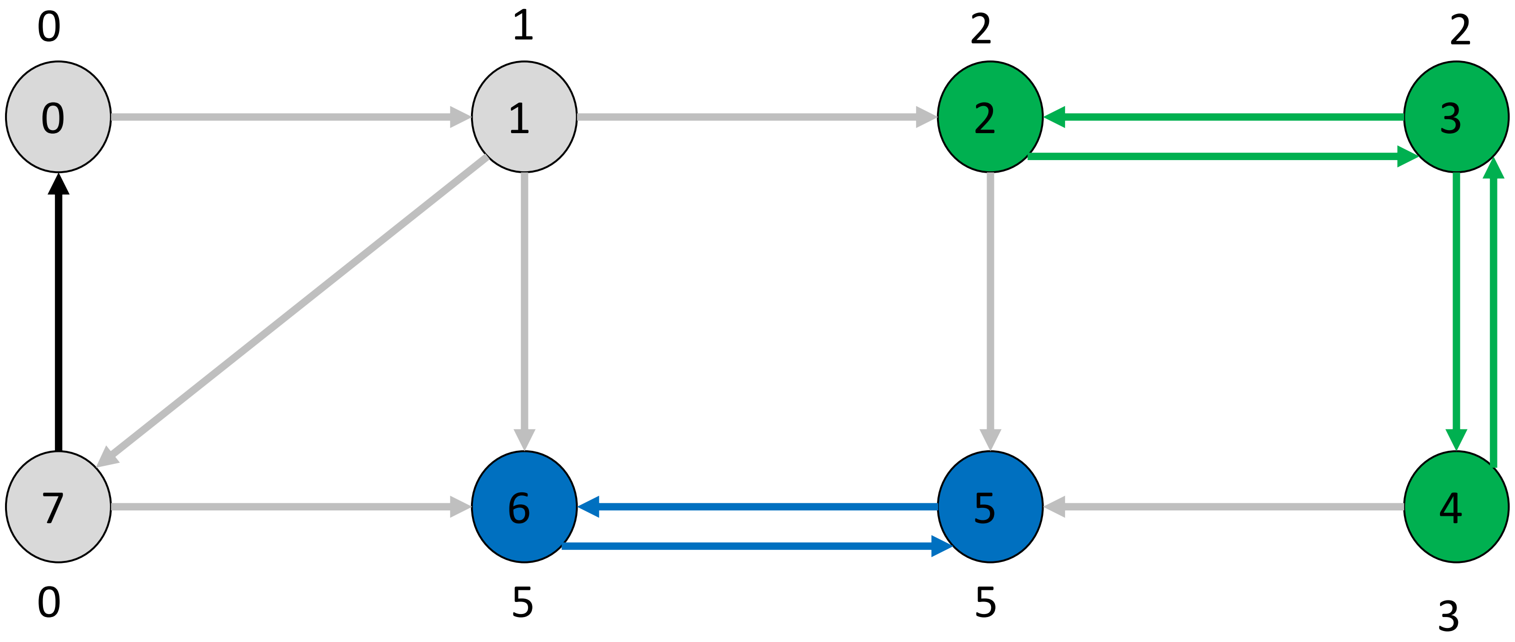


Stack

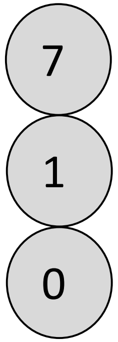


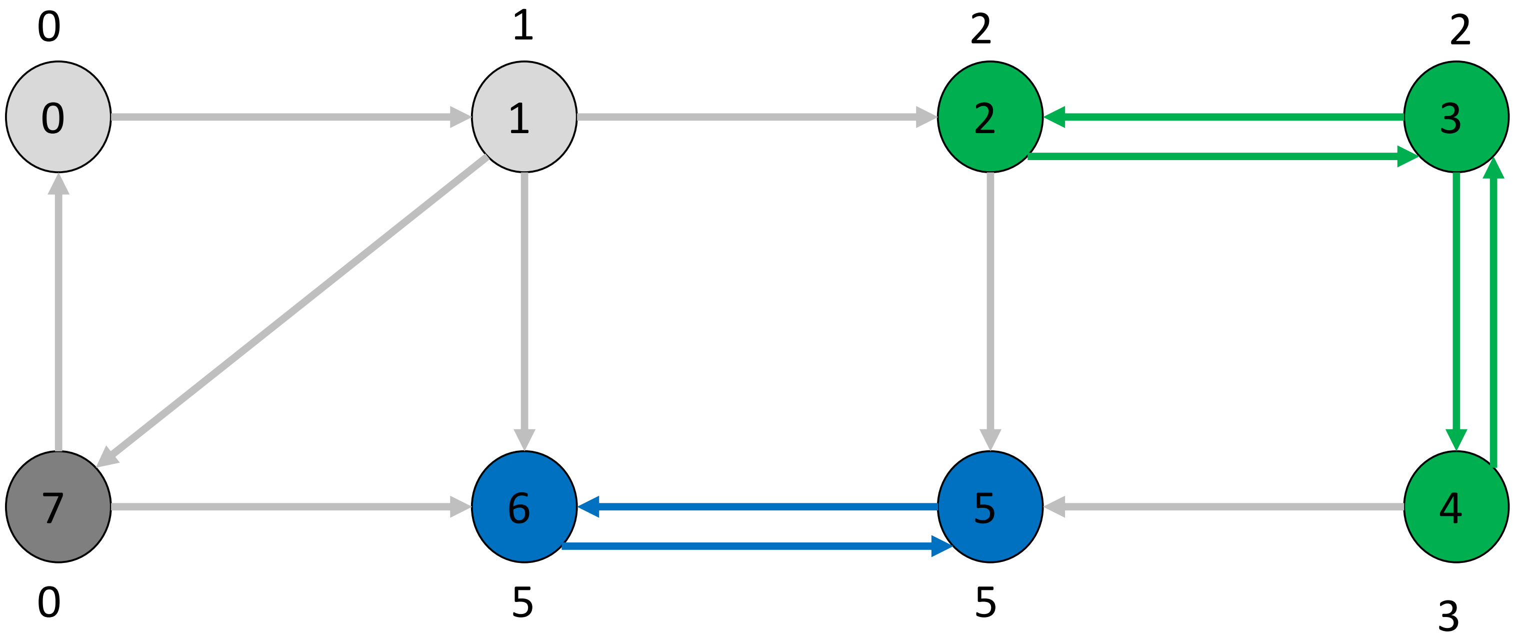
Stack



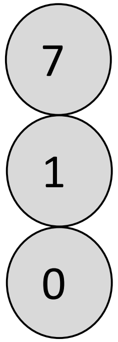


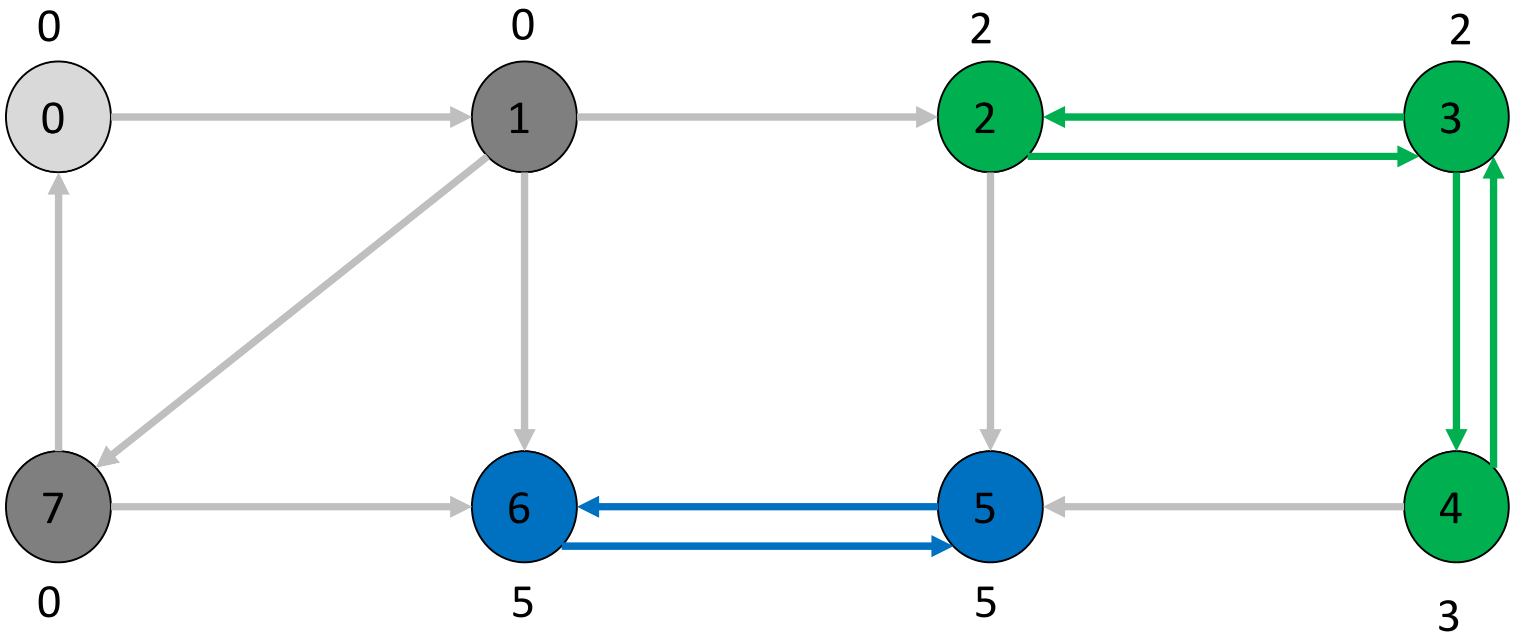
Stack



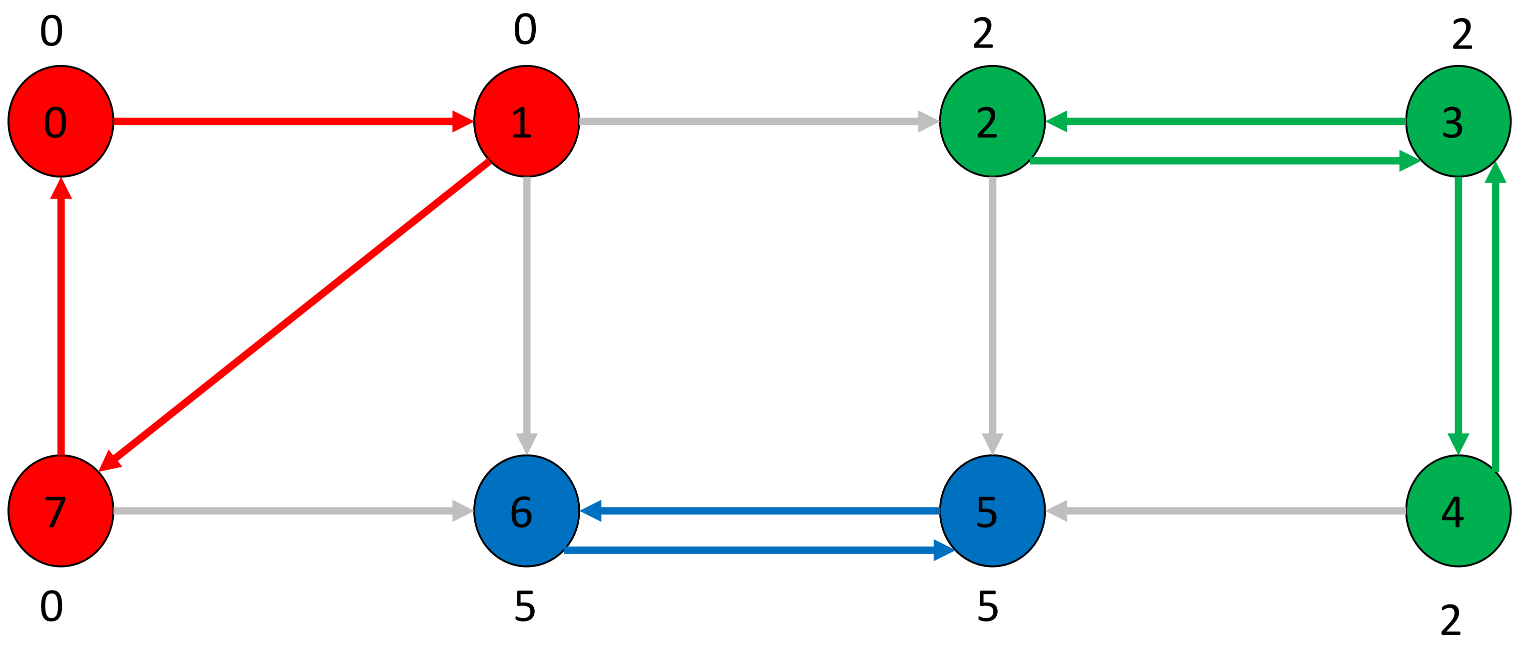


Stack





Stack



Minimum Spanning Tree (MST) problem

- Given an undirected weighted graph $G = (V, E)$:
  - Find a subset of E with the minimum total weight that connects all the nodes into a tree

- Solved using greedy approach:
  - Construct the solution by adding one “safe” edge to partial solution at each step

How to find safe edge

- Let:
  - A is set of edges which is a subset of some MST
  - $(S, V\setminus S)$ – is a cut such that no edge from A crosses the cut
  - (u, v) is the minimal weight edge from the cut-crossing edge set
- Then:
  - (u, v) is safe edge for A.

Kruskal’s algorithm

- Main idea: the edge with the smallest weight has to be in the MST
- Another main idea: after an edge is chosen, the two nodes at the ends can be merged and considered as single node (supernode)

$A = \emptyset$

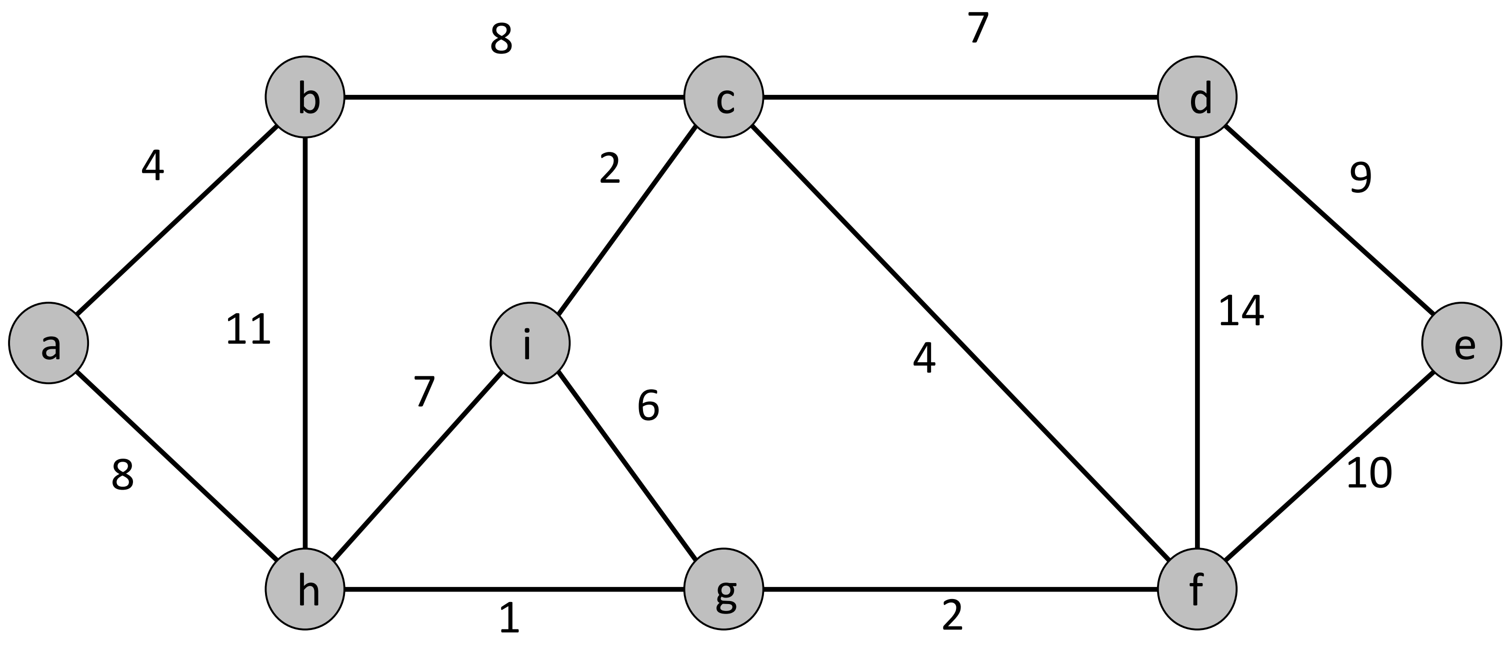
For each $v \in V$:
- $MAKE\_SET(v)$

Sort E

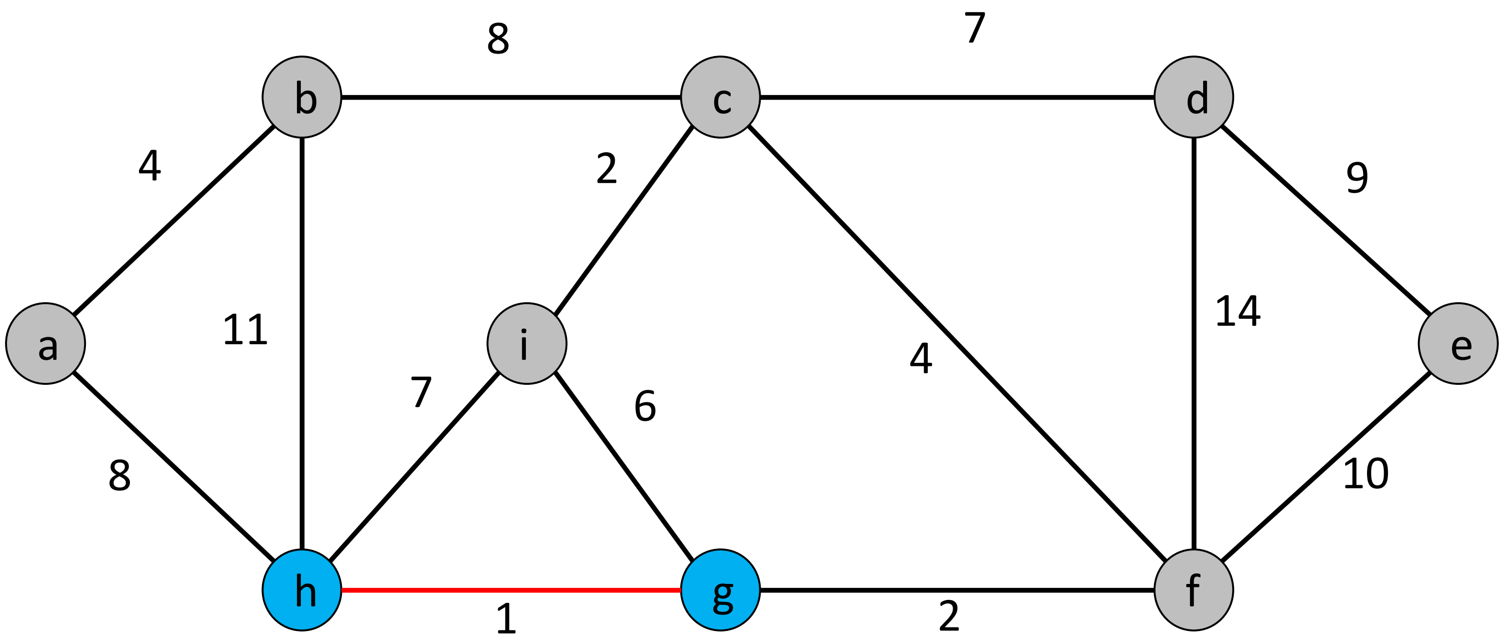
For $(u, v) \in E$ (in weight increasing order):
- **If** $FIND\_SET(u)$!= $FIND\_SET(v)$:
  - $A = A \cup (u, v)$
  - $UNION(u, v)$

return A

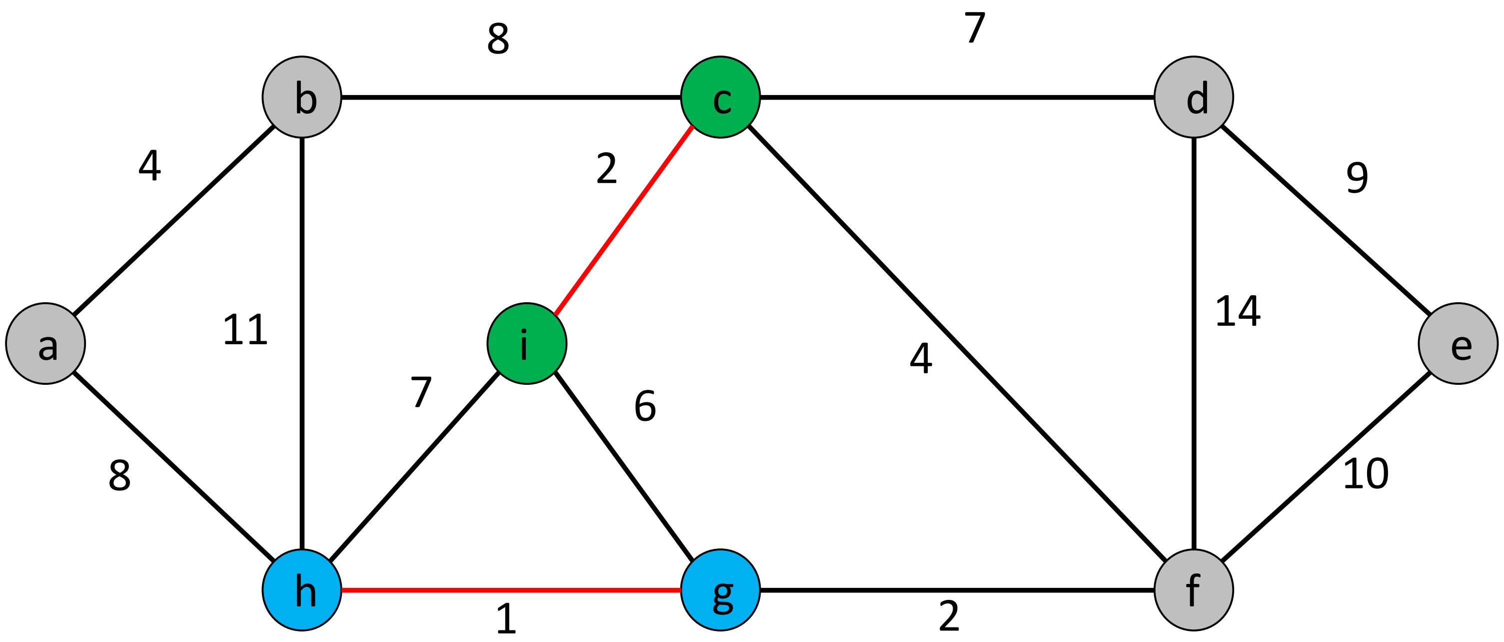
Example



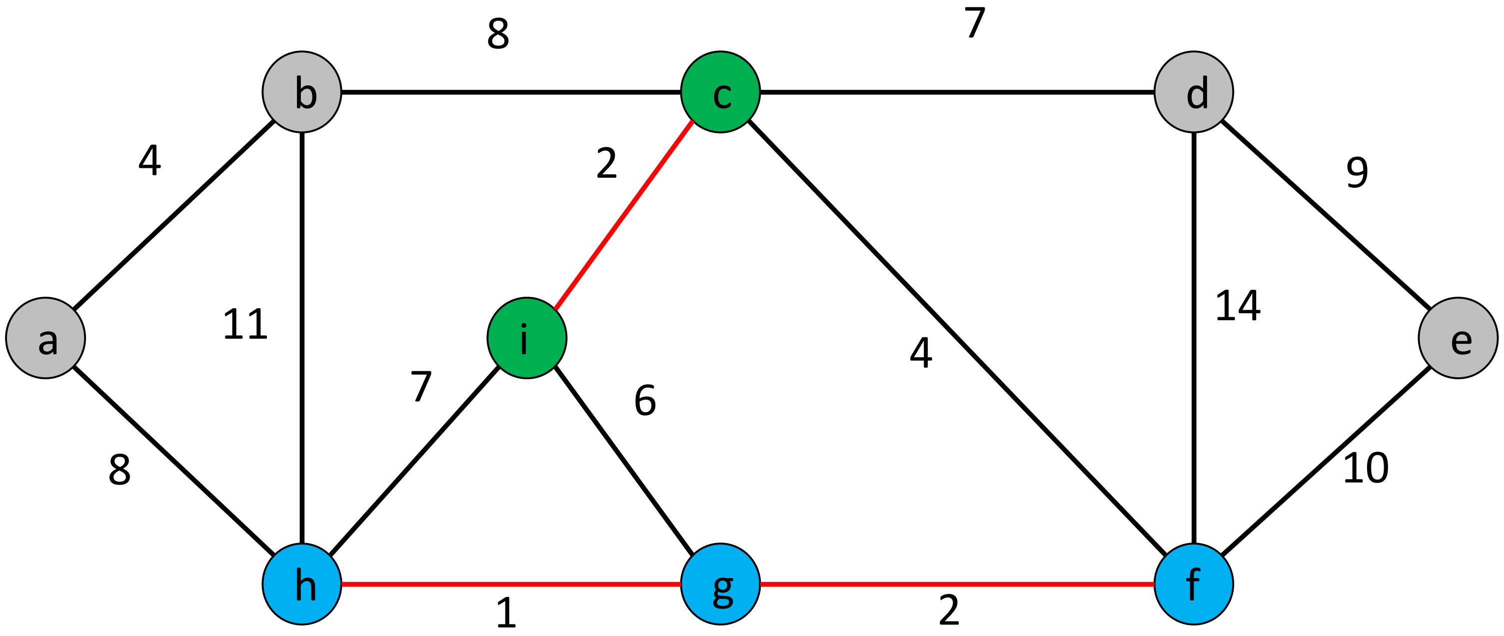
Example



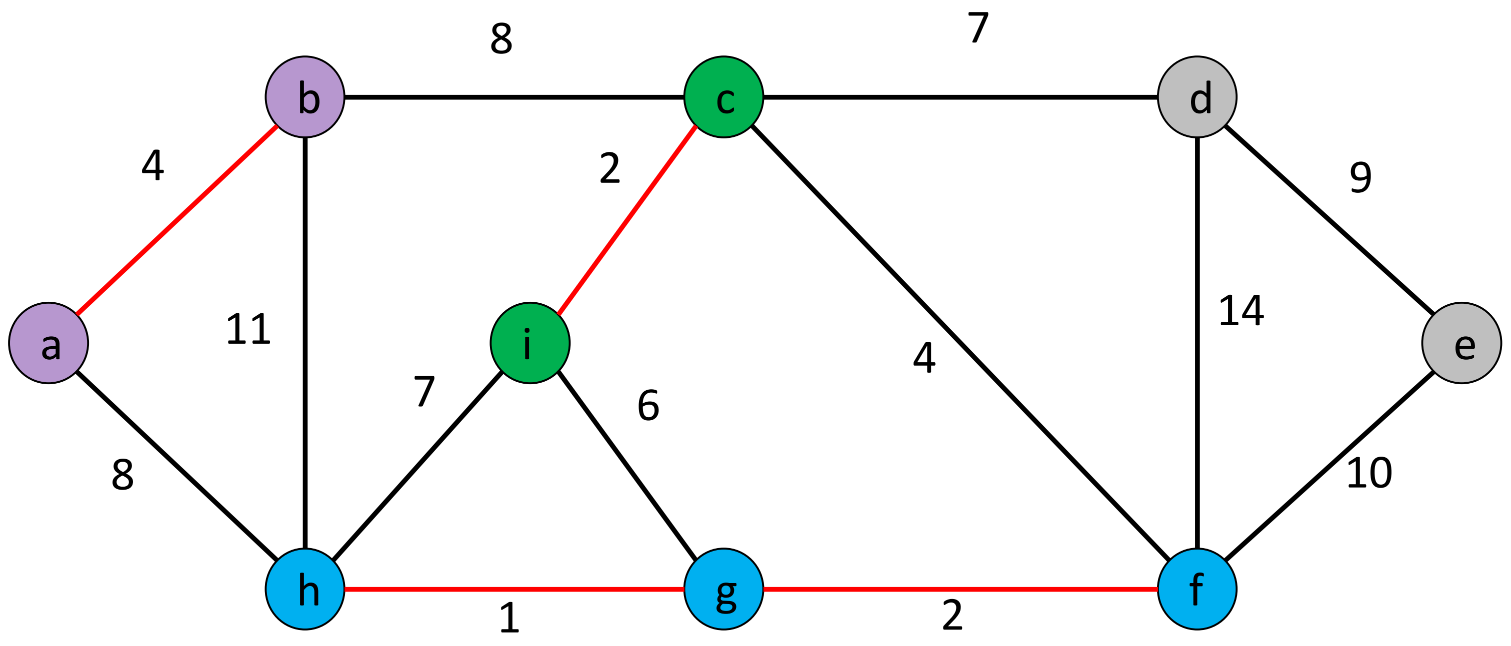
Example



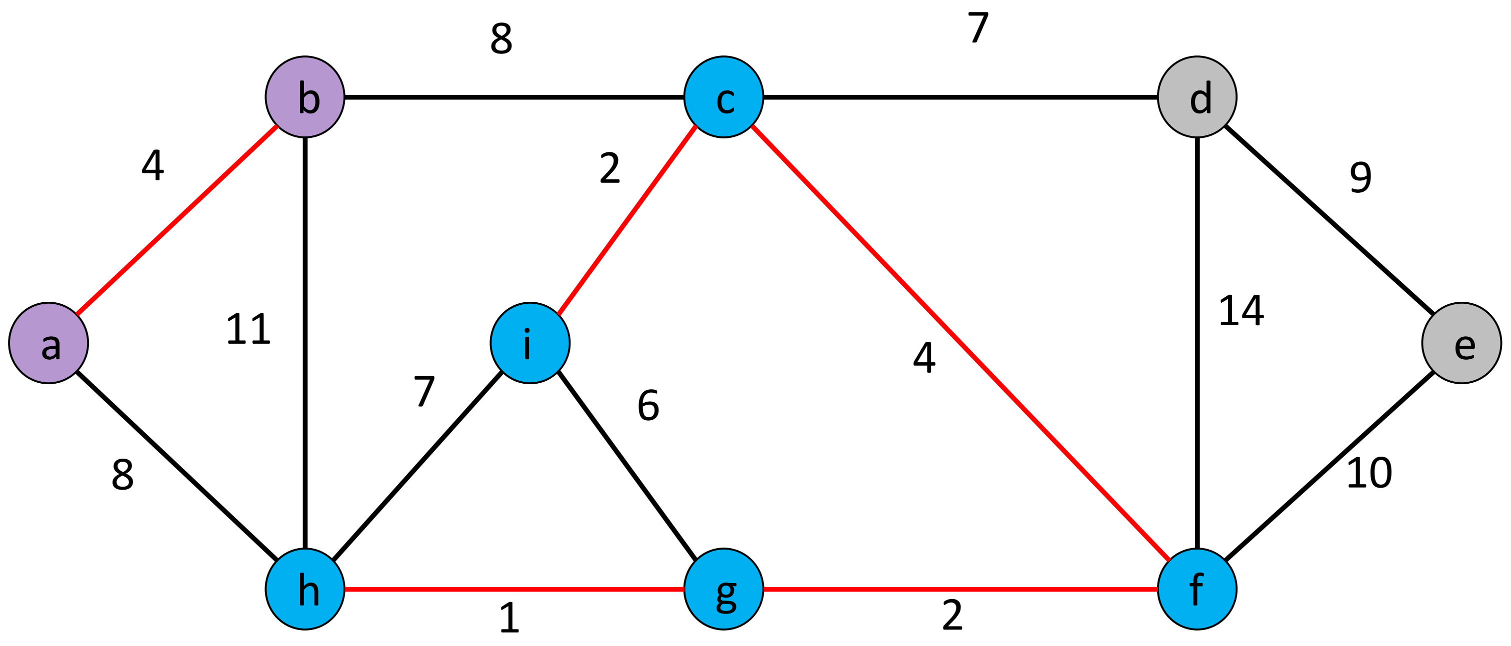
Example



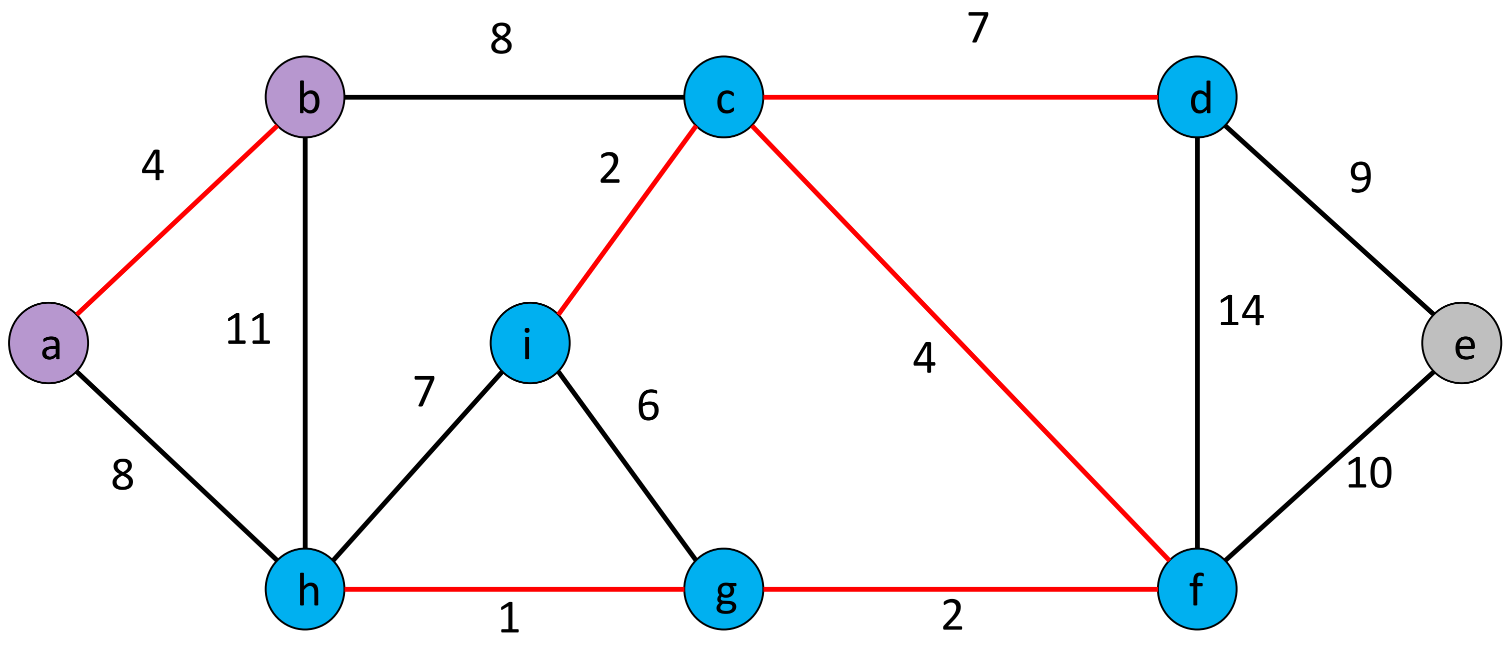
Example



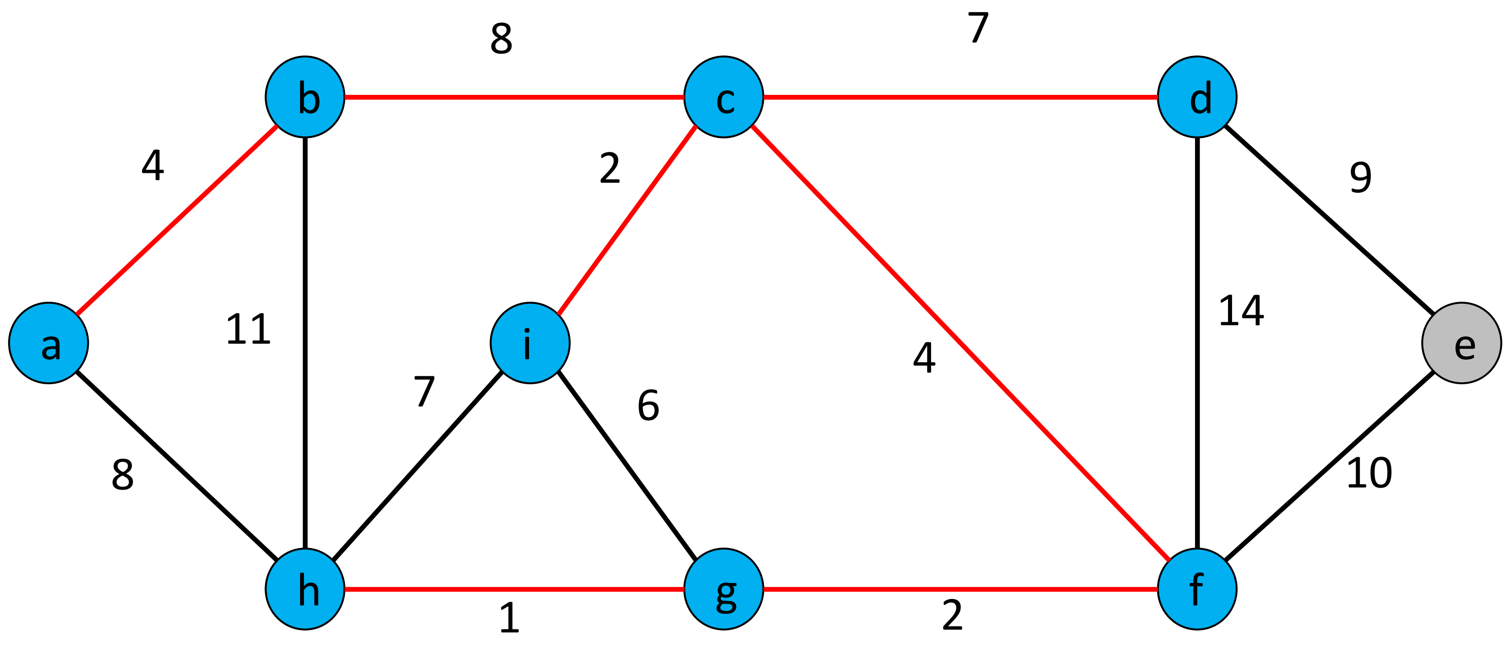
Example



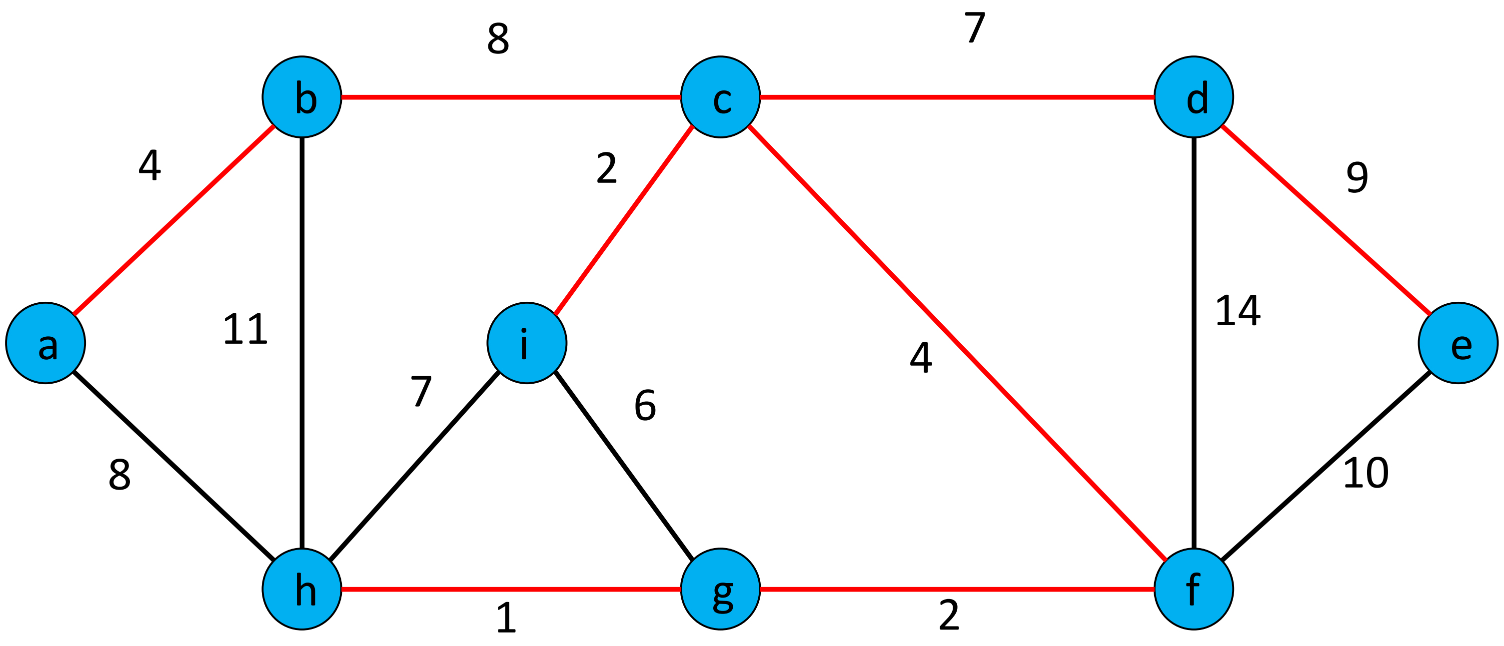
Example



Example



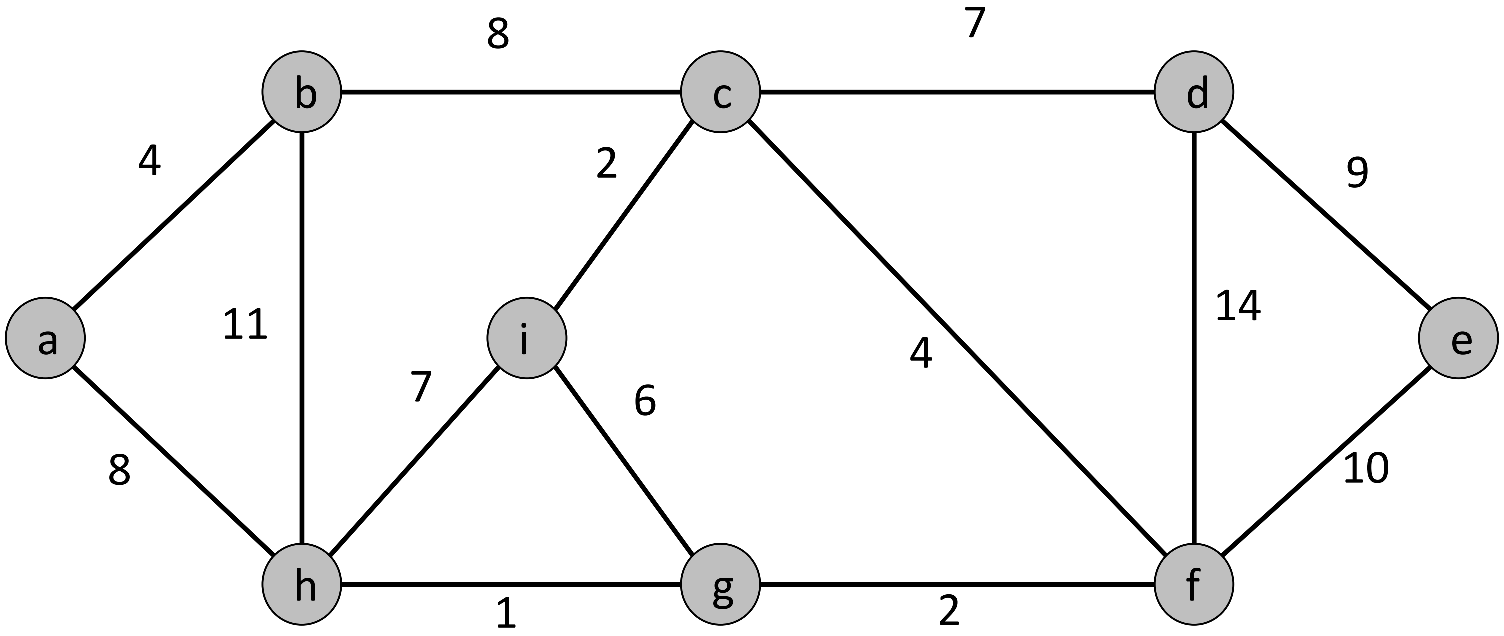
Example



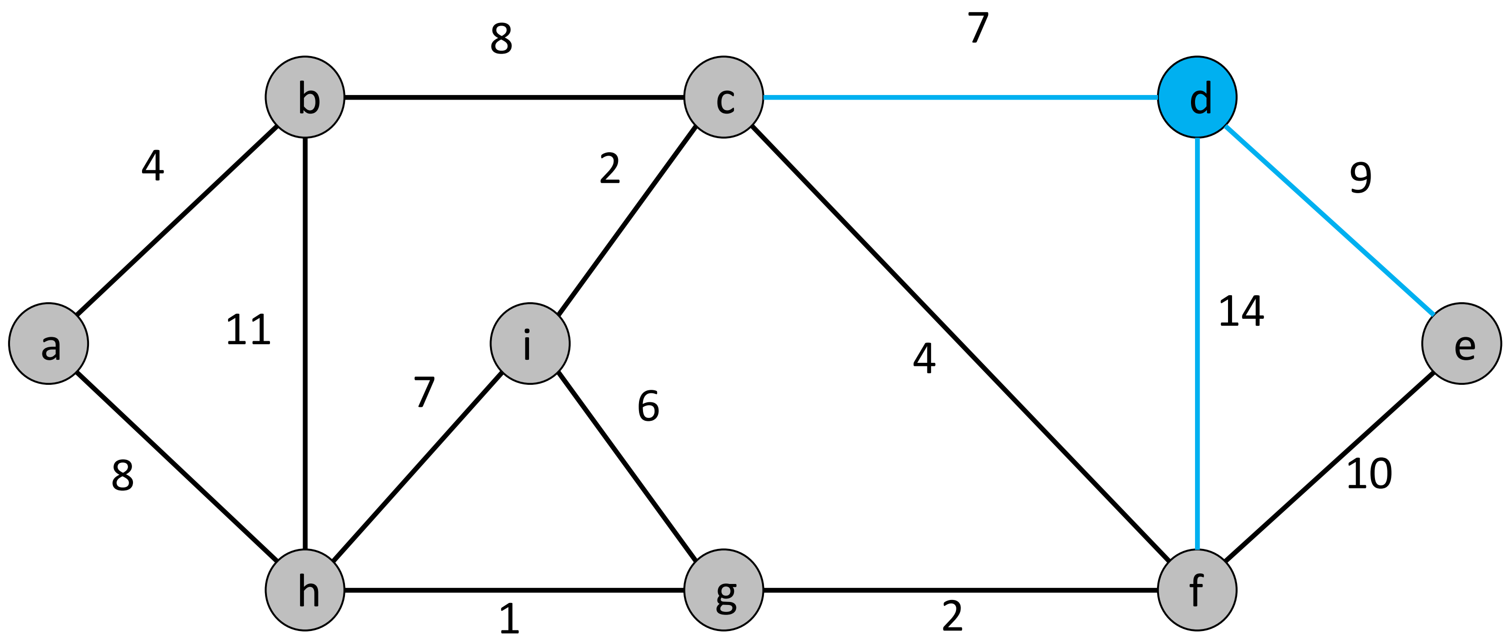
Prim's Algorithm

- Maintain a set S that starts out with a single node s
- For a node v:
  - maintain $k(v)$ – min weight of edge $\mathrm{e} = (u, v)$ connecting v and S or ∞ if no such edges
  - Maintain $\pi(v)$ – pointer to node u or parent in MST
- Find the smallest weighted edge $e = (u, v)$ that connects $u \in S$ and $v \notin S$
- Add v to S
- For $w \in Adj(v)$:
  - If $w \notin S$ and $weight(v, w) < k(w)$:
    - $\pi(w) = \mathrm{v}$
    - $k(w) = \mathrm{weight}(\mathrm{v}, \mathrm{w})$
- Repeat until $S = V$

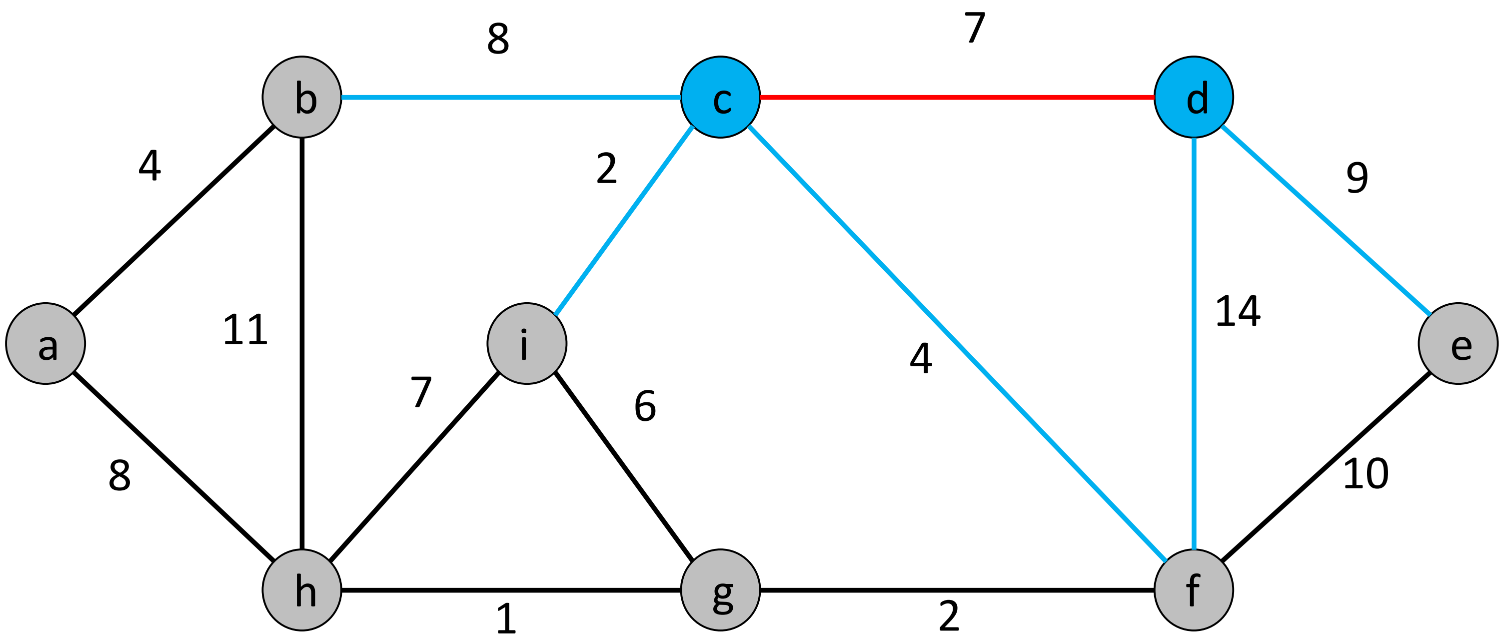
Example



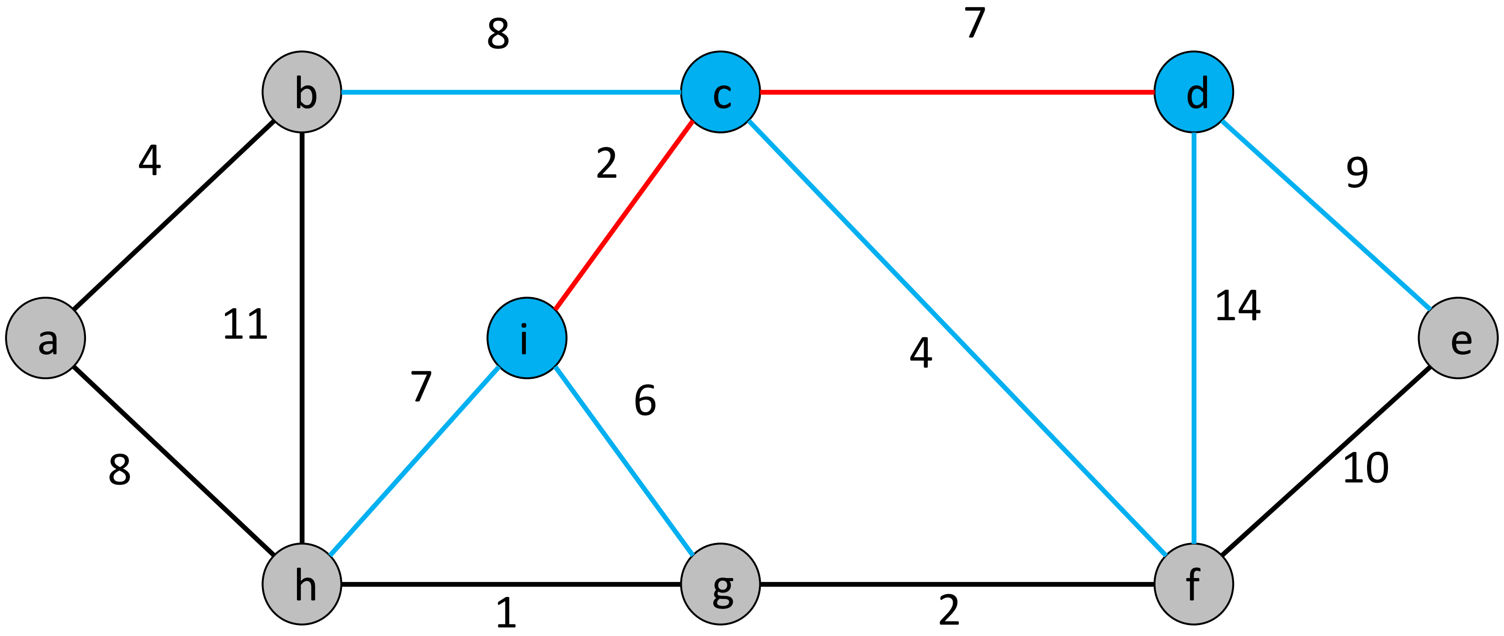
Example



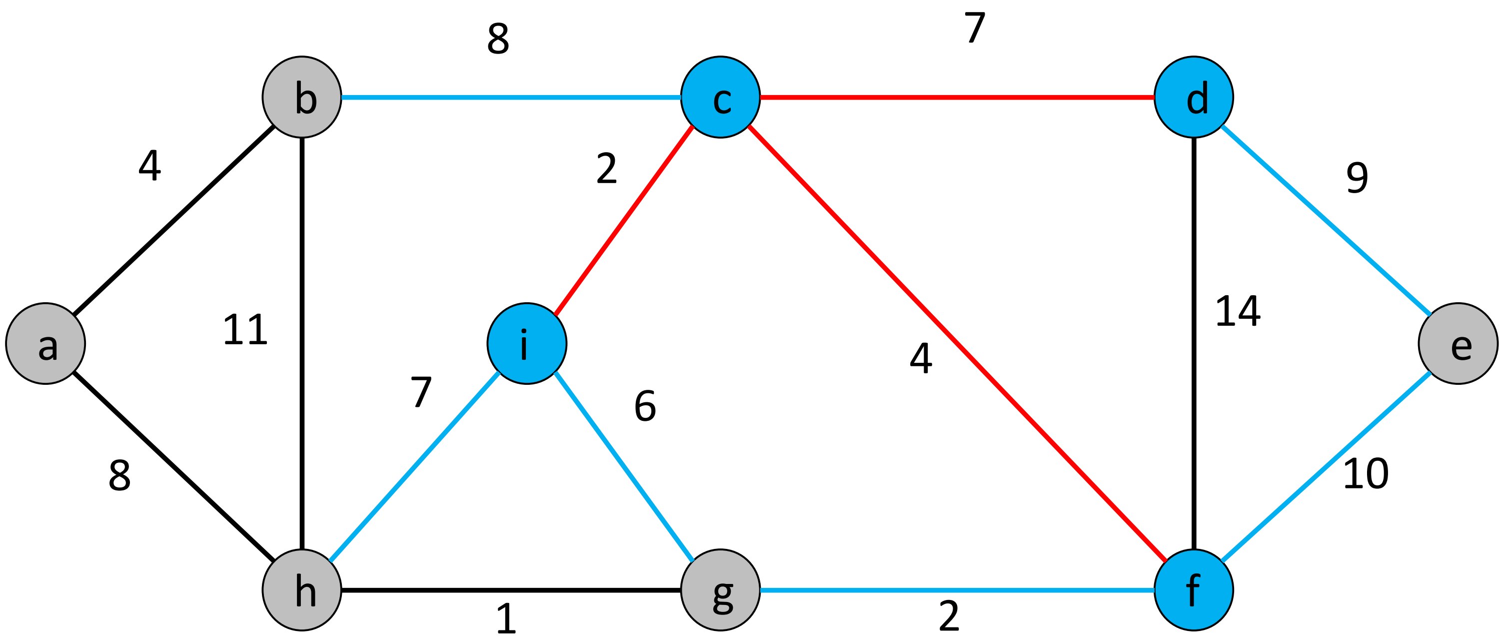
Example



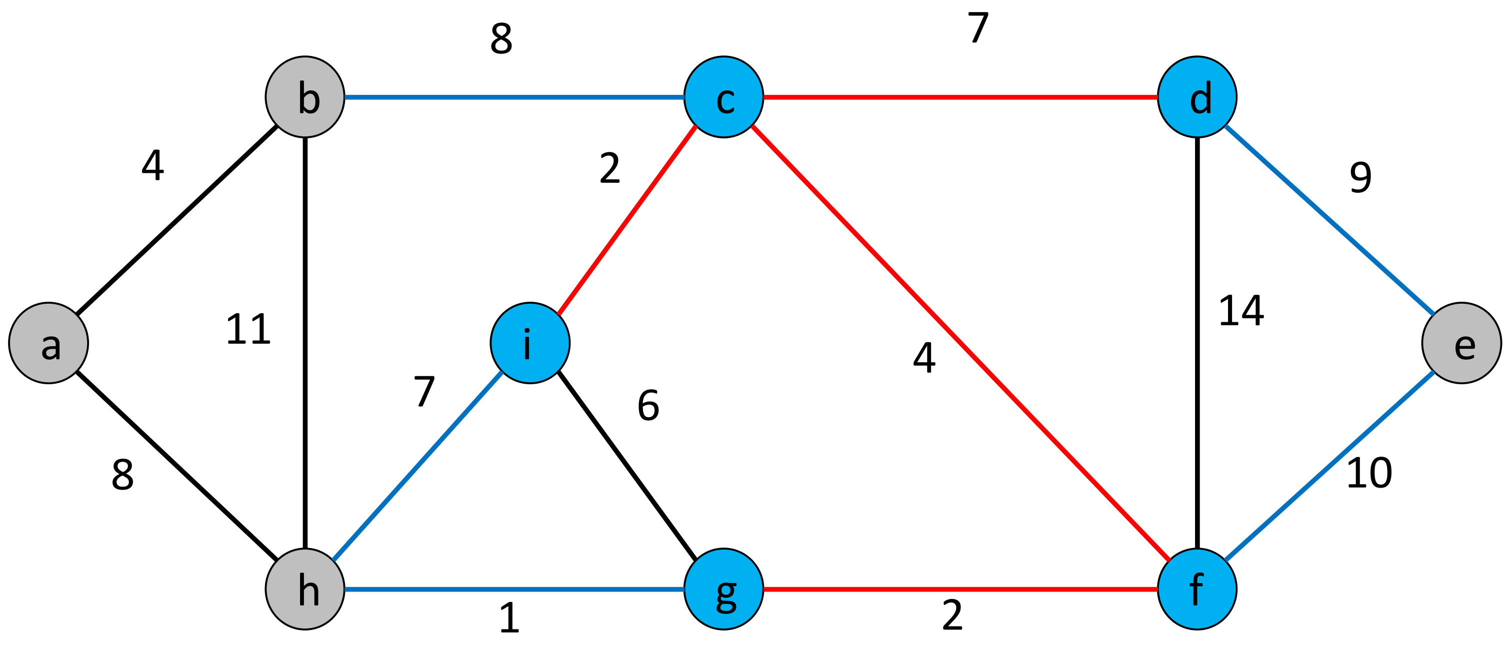
Example



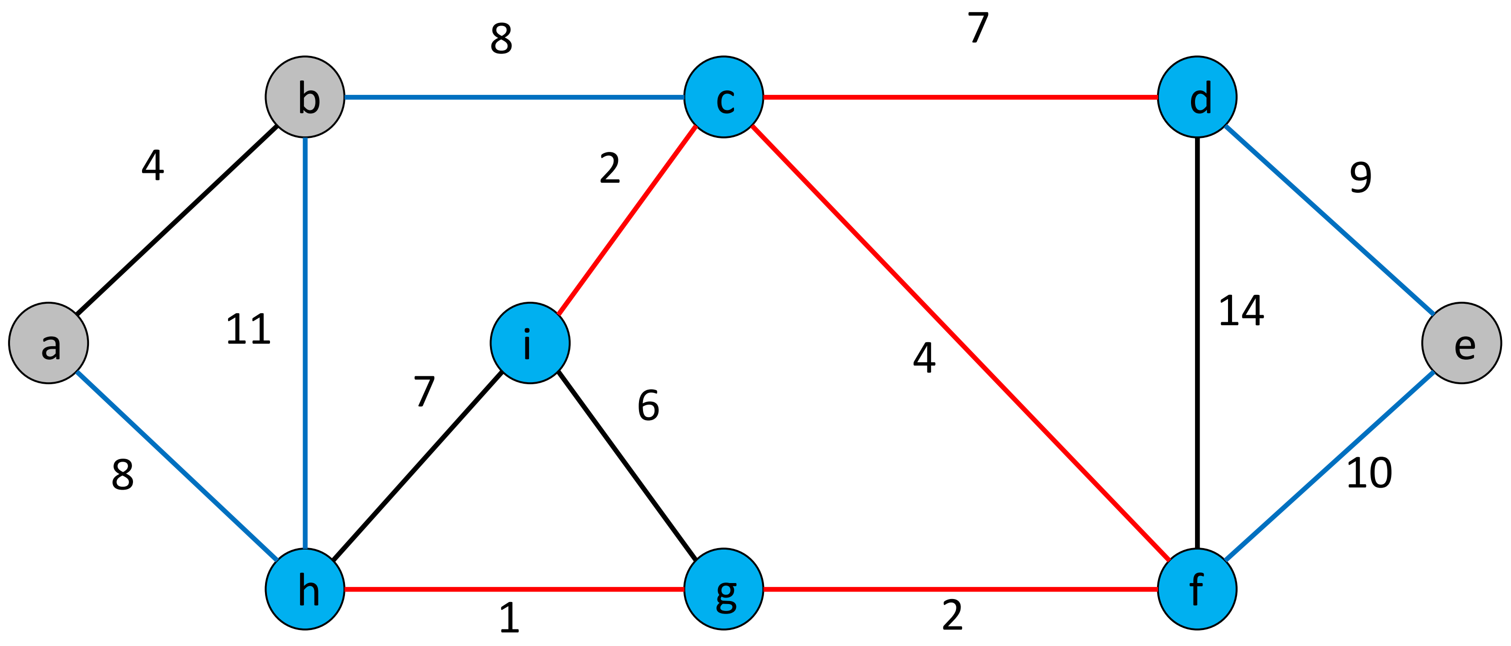
Example



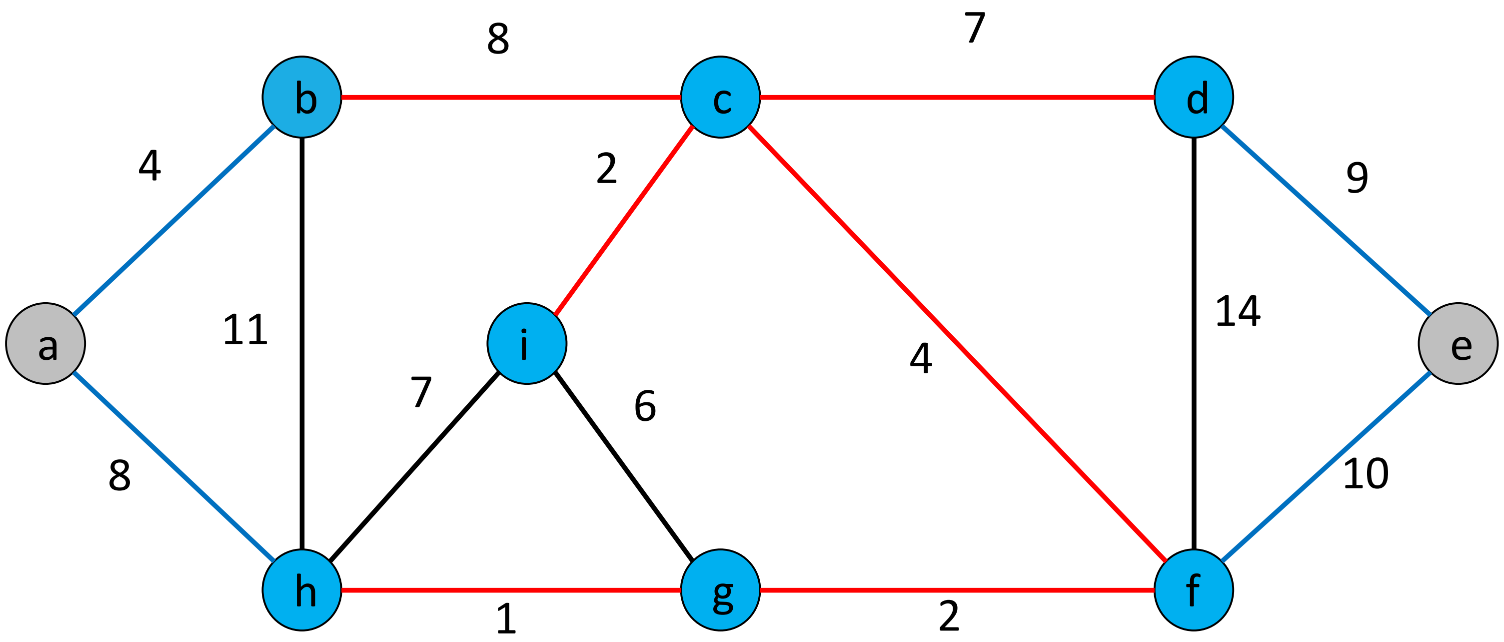
Example



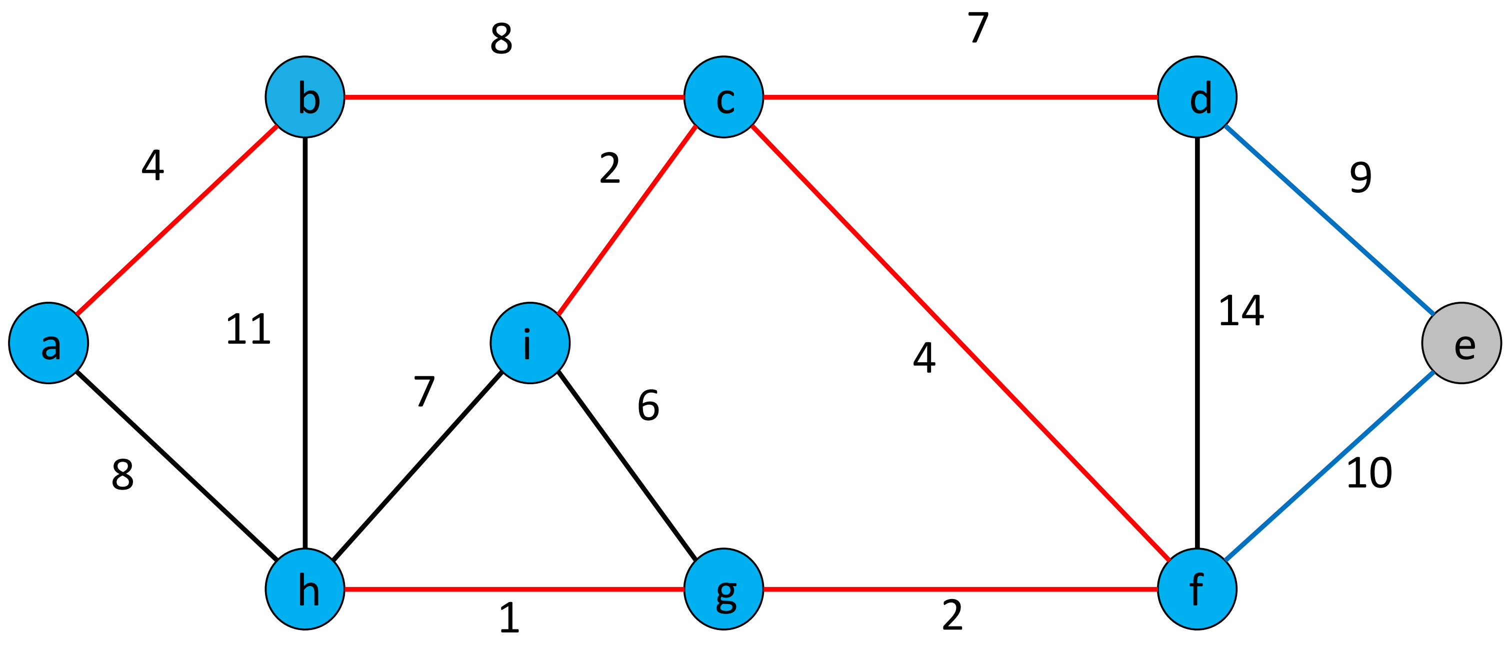
Example



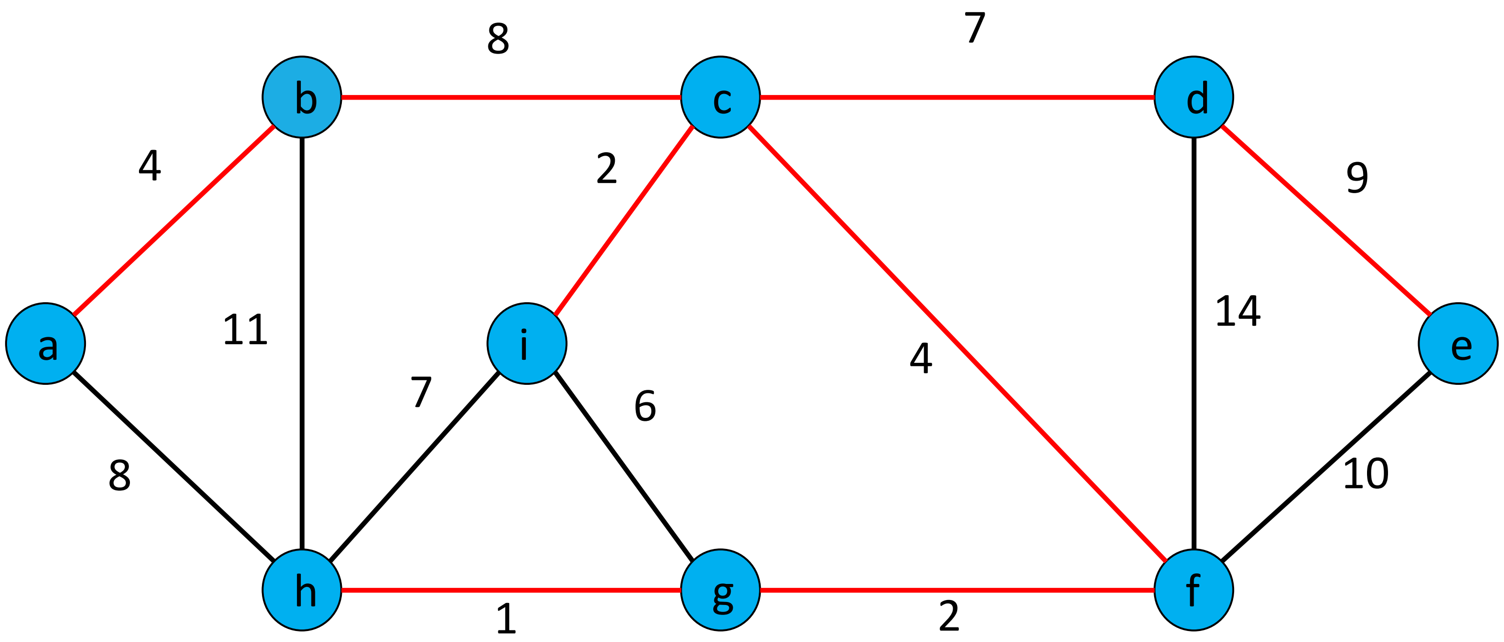
Example



Example



Example



Kruskal vs. Prim

- Kruskal's algorithm:
  - Takes $O(E \log E)$ time
  - Easy to code
  - Generally faster on sparse graphs
- Prim's algorithm:
  - Time complexity depends on the implementation
    - Can be $\mathrm{O}(V^2 + E)$, $O(E \log(V)), or\ O\big(E + V\ log(V)\big)$
  - Generally faster than Kruskal's on dense graphs

Shortest path

- **Input**: a weighted graph $G = (V, E)$
  - The edges can be directed or not
  - Note: use BFS for unweighted graphs
- **Output**: the path between two given nodes u and v that minimizes the total weight
  - Variation: compute shortest paths from u to all other nodes
  - Variation: compute all-pair shortest paths

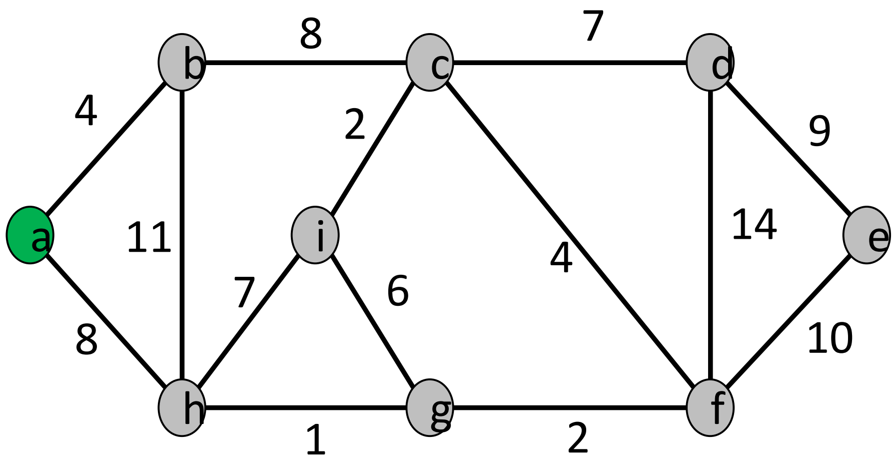
Dijkstra’s algorithm

- Given a weighted graph $G = (V, E)$ with non-negative weights, and source s:
- Output a vector d where d_i is the shortest path from s to node i
- Time complexity depends on the implementation
  - Can be $O(V^2 + E), O(E \log(V)), or\ O\big(E + V\ log(V)\big)$
  - **Idea:** find the closest node to s, then the second closest one, then the third, etc.

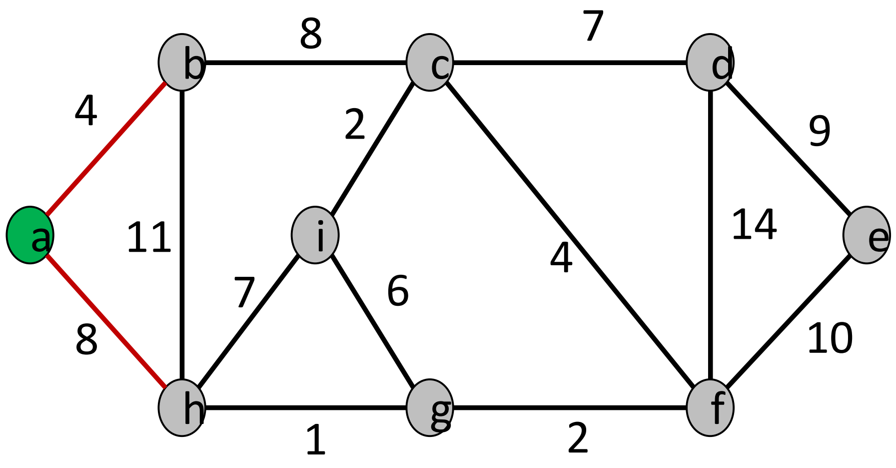
Dijkstra's algorithm

- Maintain a set of nodes S, the shortest distances to which are decided
- Maintain a vector d, the shortest distance estimate from s
- Initially: $S = \{s\}$, and $d_v = cost(s, v)$
- Repeat until $S = V$:
  - Find $v \notin S$ with the smallest d_v and add it to S
  - For each edge $v \to u$ with cost c: $d_u = \min(d_u, d_v + c)$

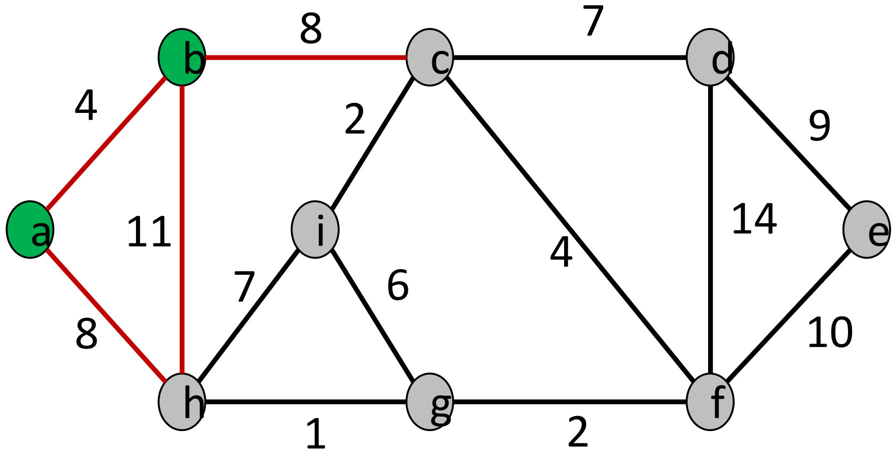
Dijkstra's algorithm example



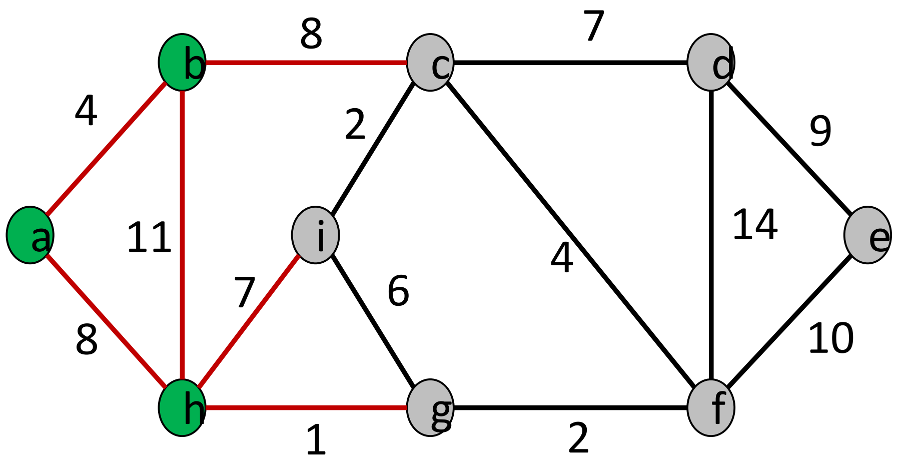
S	a	b	c	d	e	f	g	h	i
a	0(a)	∞	∞	∞	∞	∞	∞	∞	∞



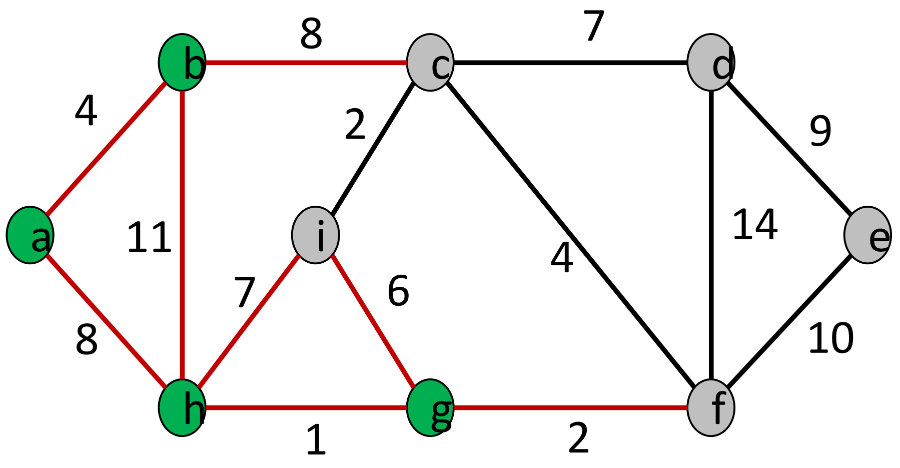
S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞



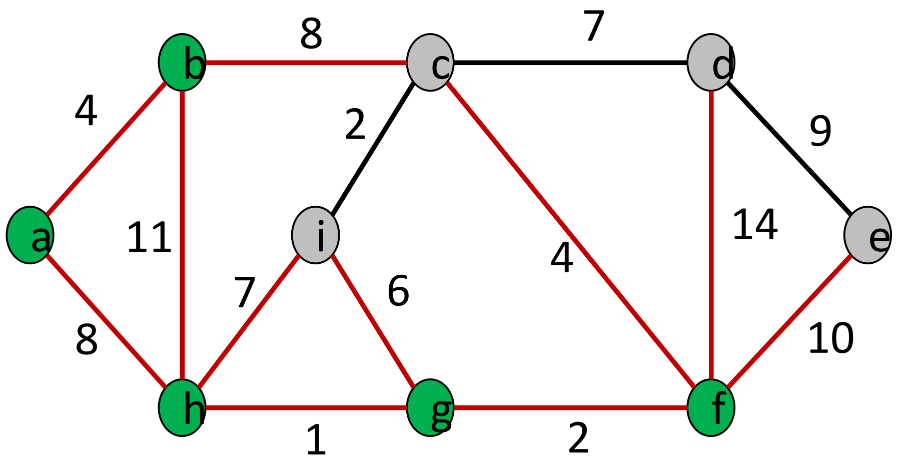
S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞



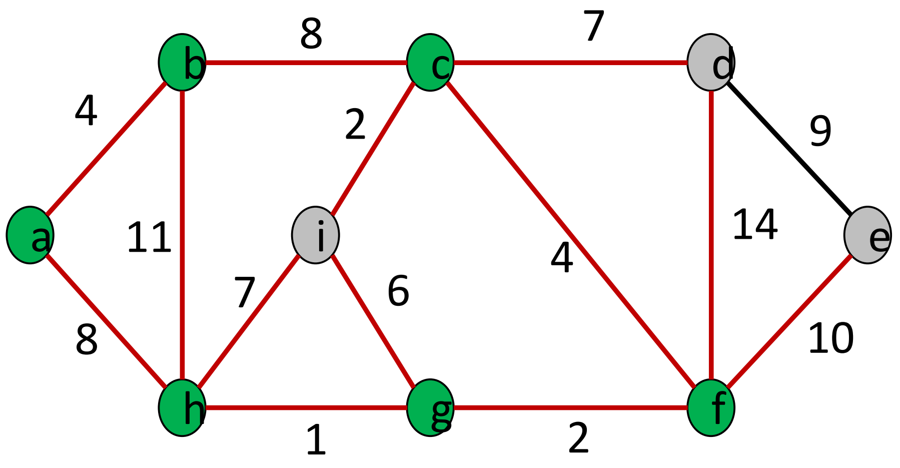
S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞
h	0(a)	4(a)	12 (b)	∞	∞	∞	**9(h)**	8(a)	15(h)



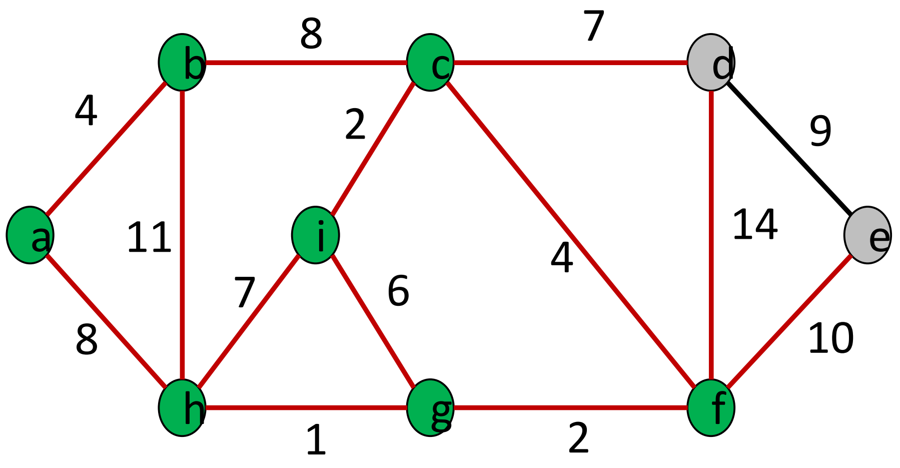
S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞
h	0(a)	4(a)	12 (b)	∞	∞	∞	**9(h)**	8(a)	15(h)
g	0(a)	4(a)	12(b)	∞	∞	**11(g)**	9(h)	8(a)	15(h)



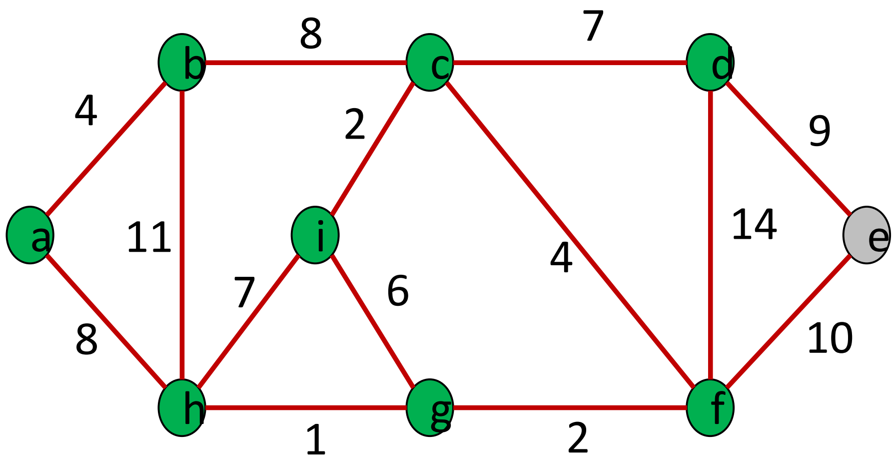
S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞
h	0(a)	4(a)	12 (b)	∞	∞	∞	**9(h)**	8(a)	15(h)
g	0(a)	4(a)	12(b)	∞	∞	**11(g)**	9(h)	8(a)	15(h)
f	0(a)	4(a)	**12(b)**	25(f)	21(f)	11(g)	9(h)	8(a)	15(h)



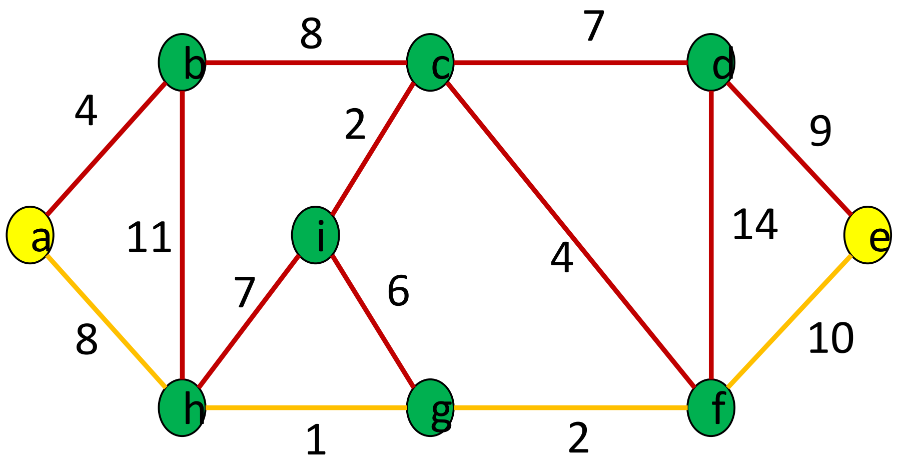
S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞
h	0(a)	4(a)	12 (b)	∞	∞	∞	**9(h)**	8(a)	15(h)
g	0(a)	4(a)	12(b)	∞	∞	**11(g)**	9(h)	8(a)	15(h)
f	0(a)	4(a)	**12(b)**	25(f)	21(f)	11(g)	9(h)	8(a)	15(h)
c	0(a)	4(a)	12(b)	19(c)	21(f)	11(g)	9(h)	8(a)	**14(c)**



S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞
h	0(a)	4(a)	12 (b)	∞	∞	∞	**9(h)**	8(a)	15(h)
g	0(a)	4(a)	12(b)	∞	∞	**11(g)**	9(h)	8(a)	15(h)
f	0(a)	4(a)	**12(b)**	25(f)	21(f)	11(g)	9(h)	8(a)	15(h)
c	0(a)	4(a)	12(b)	19(c)	21(f)	11(g)	9(h)	8(a)	**14(c)**
i	0(a)	4(a)	12(b)	**19(c)**	21(f)	11(g)	9(h)	8(a)	14(c)



S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞
h	0(a)	4(a)	12 (b)	∞	∞	∞	**9(h)**	8(a)	15(h)
g	0(a)	4(a)	12(b)	∞	∞	**11(g)**	9(h)	8(a)	15(h)
f	0(a)	4(a)	**12(b)**	25(f)	21(f)	11(g)	9(h)	8(a)	15(h)
c	0(a)	4(a)	12(b)	19(c)	21(f)	11(g)	9(h)	8(a)	**14(c)**
i	0(a)	4(a)	12(b)	**19(c)**	21(f)	11(g)	9(h)	8(a)	14(c)
d	0(a)	4(a)	12(b)	19(c)	**21(f)**	11(g)	9(h)	8(a)	14(c)



S	a	b	c	d	e	f	g	h	i
a	0(a)	**4(a)**	∞	∞	∞	∞	∞	8(a)	∞
b	0(a)	4(a)	12(b)	∞	∞	∞	∞	**8(a)**	∞
h	0(a)	4(a)	12 (b)	∞	∞	∞	**9(h)**	8(a)	15(h)
g	0(a)	4(a)	12(b)	∞	∞	**11(g)**	9(h)	8(a)	15(h)
f	0(a)	4(a)	**12(b)**	25(f)	21(f)	11(g)	9(h)	8(a)	15(h)
c	0(a)	4(a)	12(b)	19(c)	21(f)	11(g)	9(h)	8(a)	**14(c)**
i	0(a)	4(a)	12(b)	**19(c)**	21(f)	11(g)	9(h)	8(a)	14(c)
d	0(a)	4(a)	12(b)	19(c)	**21(f)**	11(g)	9(h)	8(a)	14(c)

Bellman-Ford algorithm

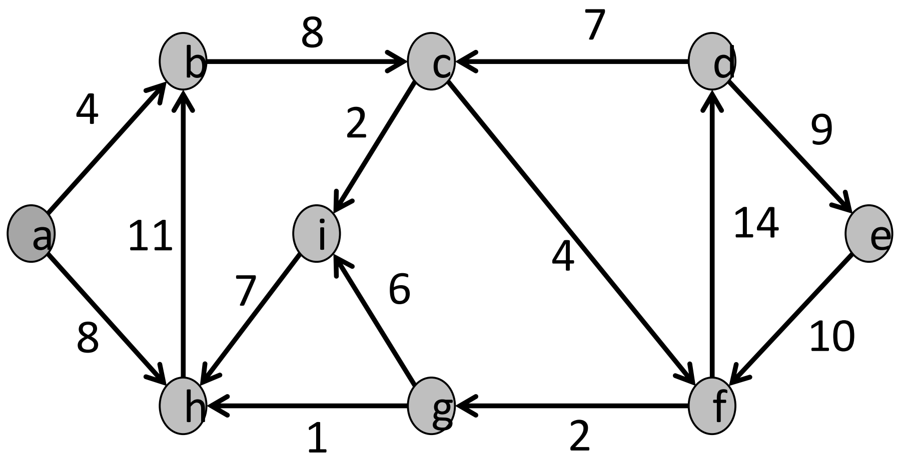
- Given a directed weighted graph $G = (V, E)$ and source s:
- Output a vector d where d_i is the shortest path from s to node i
- Can detect a negative-weight cycle
- Time complexity: $\Theta(nm)$
- Easy to code

Bellman-Ford algorithm

- Initialize $d_s = 0$ and $d_v = \infty$ for all $v \neq s$
- **For** $\mathrm{i} = 1$ to $n - 1$:
  - For each edge $u \rightarrow v$ with cost c: $d_v = \min(d_v, d_u + c)$
- **For** each edge $u \rightarrow v$ with cost c:
  - **If** $d_v > d_u + c$: **then** the graph contains a negative-weight cycle

Example

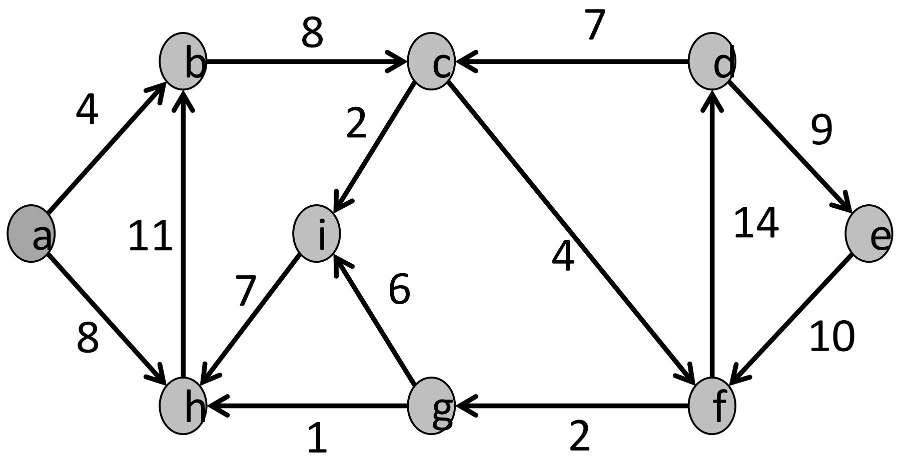
Edge order:
(i,h)
(h,b)
(g,i)
(g,h)
(f,g):
(f,d):
(e,f)
(d,e)
(d,c)
(c,i):
(c,f):
(b,c):
(a,h): $d_h = 8$
(a,b): $d_b = 4$



a	b	c	d	e	f	g	h	i
0	∞	∞	∞	∞	∞	∞	∞	∞
0	4	∞	∞	∞	∞	∞	8	∞

Example

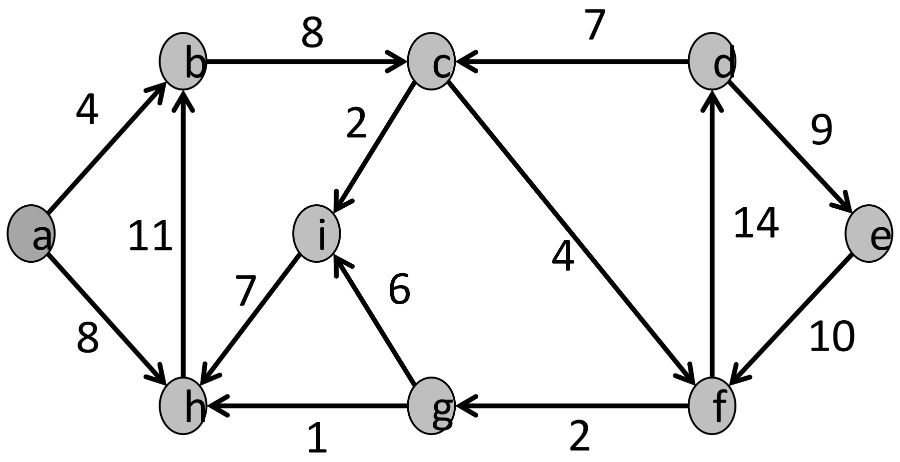
Edge order:
(i,h)
(h,b)
(g,i)
(g,h)
(f,g):
(f,d):
(e,f)
(d,e)
(d,c)
(c,i):
(c,f):
(b,c): $d_c = 12$
(a,h):
(a,b):



a	b	c	d	e	f	g	h	i
0	∞	∞	∞	∞	∞	∞	∞	∞
0	4	∞	∞	∞	∞	∞	8	∞
0	4	12	∞	∞	∞	∞	8	∞

Example

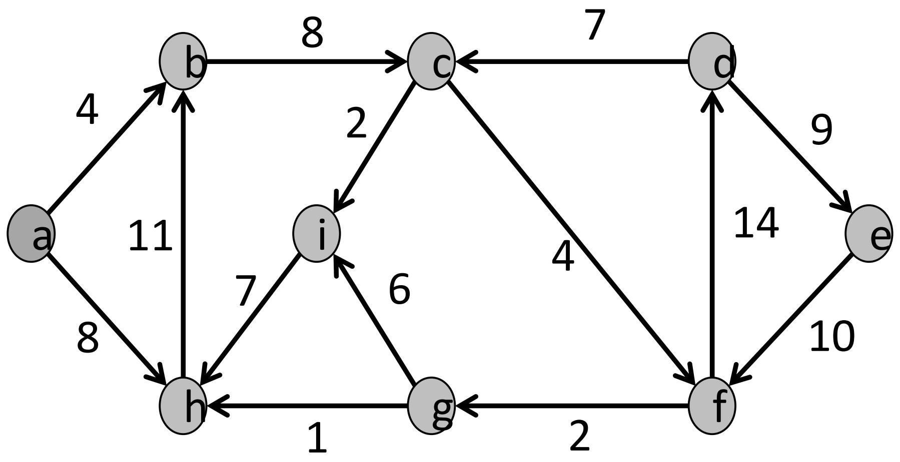
Edge order:
(i,h)
(h,b)
(g,i)
(g,h)
(f,g):
(f,d):
(e,f)
(d,e)
(d,c)
(c,i): $d_i = 14$
(c,f): $d_f = 16$
(b,c):
(a,h):
(a,b):



a	b	c	d	e	f	g	h	i
0	∞	∞	∞	∞	∞	∞	∞	∞
0	4	∞	∞	∞	∞	∞	8	∞
0	4	12	∞	∞	∞	∞	8	∞
0	4	12	∞	∞	16	∞	8	14

Example

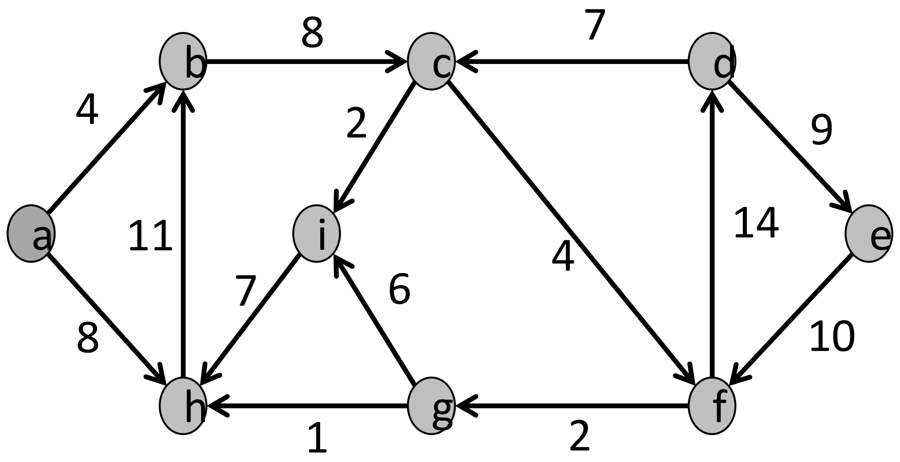
Edge order:
(i,h):
(h,b)
(g,i)
(g,h)
(f,g): $d_g = 18$
(f,d): $d_d = 30$
(e,f)
(d,e): $d_e = 39$
(d,c)
(c,i):
(c,f):
(b,c):
(a,h):
(a,b):



a	b	c	d	e	f	g	h	i
0	∞	∞	∞	∞	∞	∞	∞	∞
0	4	∞	∞	∞	∞	∞	8	∞
0	4	12	∞	∞	∞	∞	8	∞
0	4	12	∞	∞	16	∞	8	14
0	4	12	30	39	16	18	8	14

Example

Edge order:
(i,h):
(h,b)
(g,i)
(g,h)
(f,g):
(f,d):
(e,f)
(d,e):
(d,c)
(c,i):
(c,f):
(b,c):
(a,h):
(a,b):



a	b	c	d	e	f	g	h	i
0	∞	∞	∞	∞	∞	∞	∞	∞
0	4	∞	∞	∞	∞	∞	8	∞
0	4	12	∞	∞	∞	∞	8	∞
0	4	12	∞	∞	16	∞	8	14
0	4	12	30	39	16	18	8	14
0	4	12	30	39	16	18	8	14

No change. Can stop.

Why it works

- A shortest path can have at most $n-1$ edges
- At the k^{th} iteration, all shortest paths using k or less edges are computed
- After $n-1$ iterations, all distances must be final; for every edge $u \rightarrow v$ with cost c, $d_v \leq d_u + c$ holds
  - Unless there is a negative-weight cycle
  - This is how the negative-weight cycle detection works

References

- Jaehyun Park, Basic Graph Algorithms, CS97SI, Stanford University, 2015
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, Introduction to Algorithms, 2001