Graph Algorithms 1

COMP 6651 – Algorithm Design Techniques

Denis Pankratov

Basic graph terminology

Vertices/nodes:

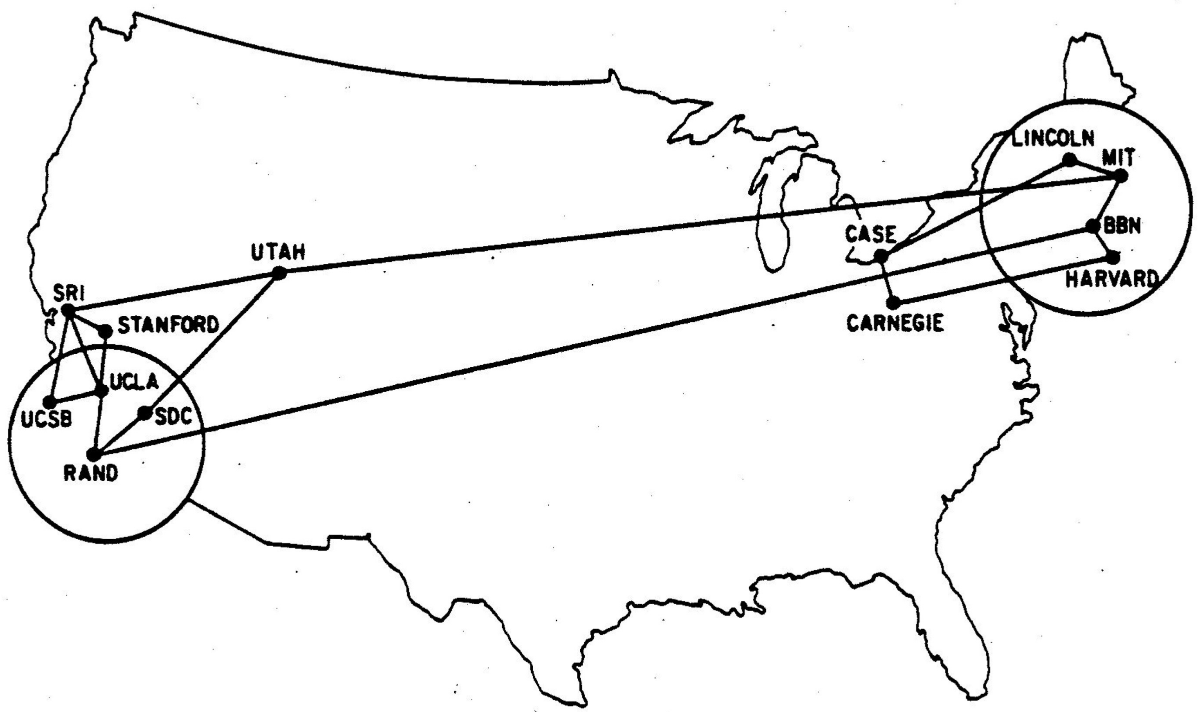
entities (people, countries, organizations, etc.)

Edges/links:

relationship between entities (friendship, classmates, same political party, membership in the same club, etc.)

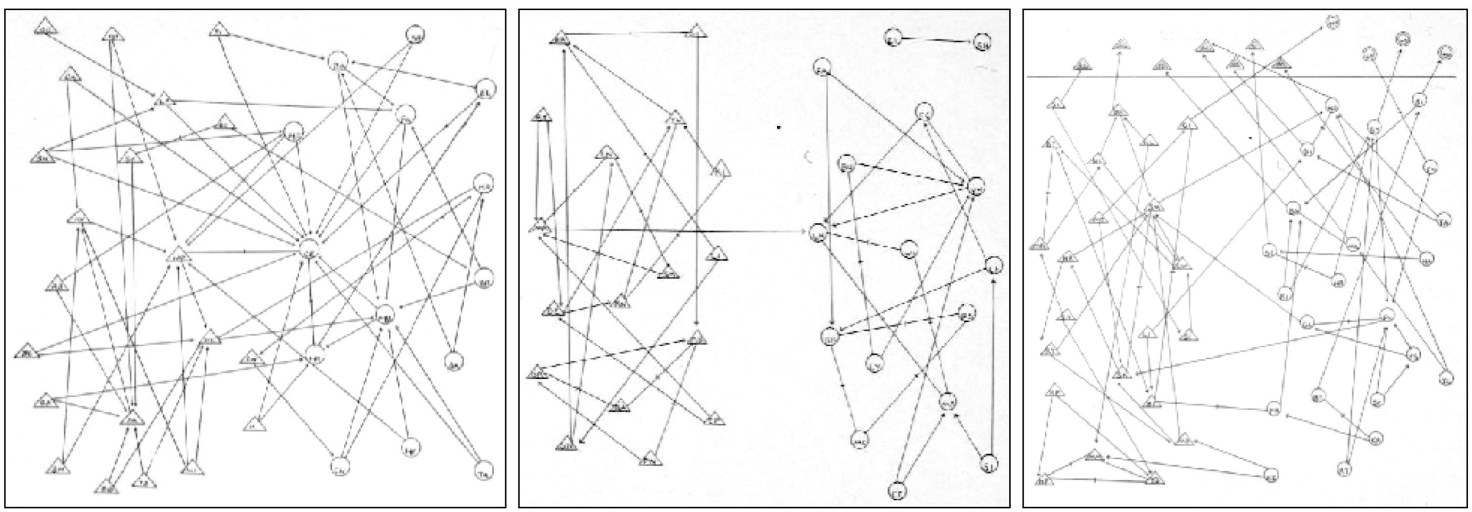
Examples: communication networks, social networks, organization of roads in a country, electrical grid, etc.

Internet as of December 1970 as a graph (Heart et al 1978)



First social network analysis (Moreno 1934)

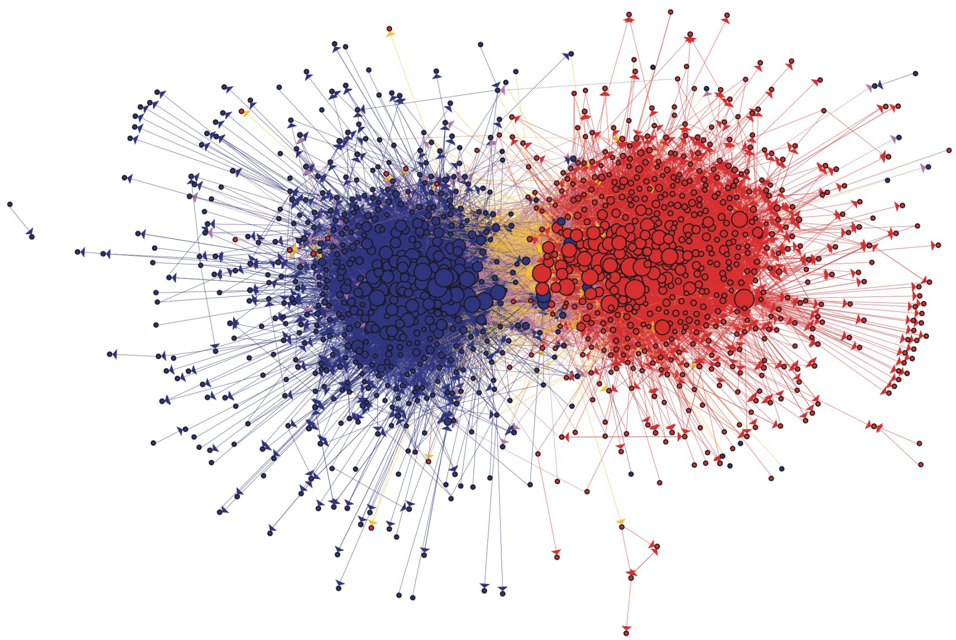
- Sociogram: each child chooses two children to sit next to
- Boys are depicted by triangles
- Girls are depicted by circles



1st grade 4th grade 8th grade

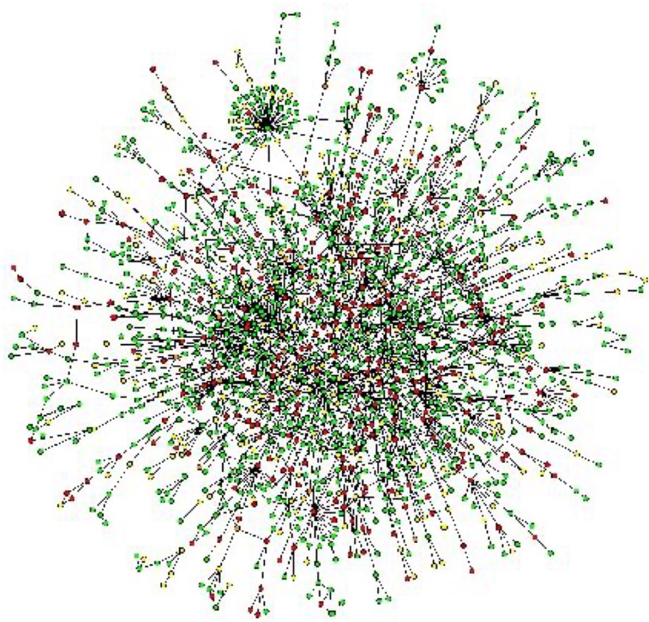
2004 blogosphere

- Community structure of political blogs
- Red – conservative, blue – liberal, edge – existence of a hyperlink



Protein-protein interaction networks

- Nodes – proteins
- Edges – physical interactions



Why study graphs?

- One of the most useful mathematical abstractions
- Many problems can be expressed precisely and clearly in language of graphs
- We use graphs as a **model** of real systems
- A model typically simplifies things, but makes precise analysis possible

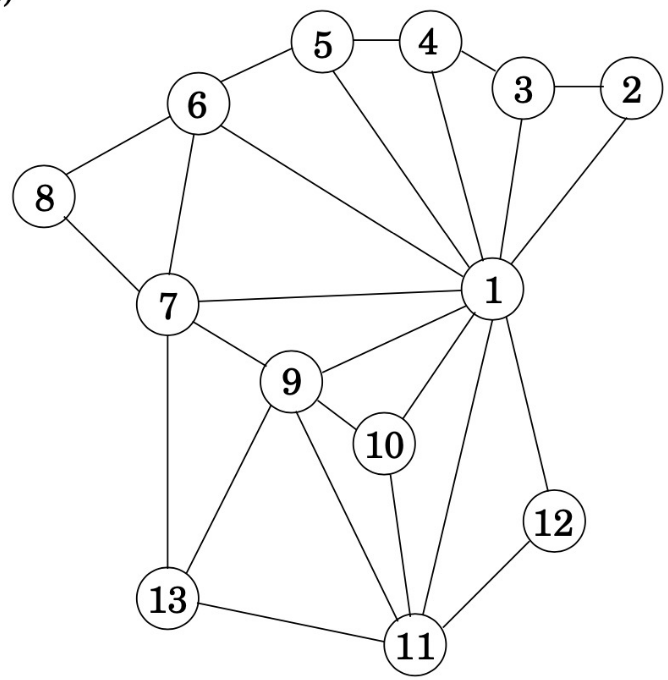
Example

- Task of coloring a political map
- Neighboring countries should receive different colors
- What is the minimum number of colors needed?
- Rephrase as a graph problem:
  - countries = vertices
  - neighborhood relationship = edges

(a)


Guyana
Venezuela
Suriname
Colombia
Guiana Highlands
French Guiana
(France)
uador
Amazon R.
Amazon
Basin
Andes
Brazil
Brazilian
Highlands
Peru
Bolivia
Paraguay
Gran
Chaco
Andes
Uruguay
Chile
Pampas
Argentina
s Juan
ández
ATLANT
Patagonia
Falkland Is.
(U.K.)

(b)


1
2
3
4
5
6
7
8
9
10
11
12
13

Graphs, formally

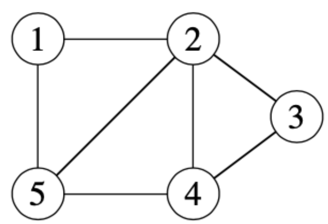
A graph G is a ***pair*** of sets

$$G = (V, E)$$

V – set of vertices

E – set of edges

Simple graphs: self-loops are not allowed, multiple edges between same pair of vertices are not allowed



Undirected:

edges do not have orientation

each edge is a ***subset*** of V of size 2

Example: $\{u, v\}$ - an undirected edge between u and v, $u, v \in V$

Maximum number of edges is $\binom{|V|}{2}$ in simple undirected graphs

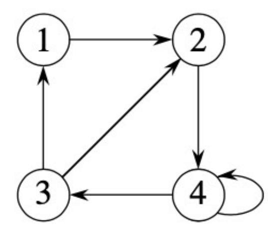
Graphs, formally

A graph G is a ***pair*** of sets

$$G = (V, E)$$

V – set of vertices

E – set of edges



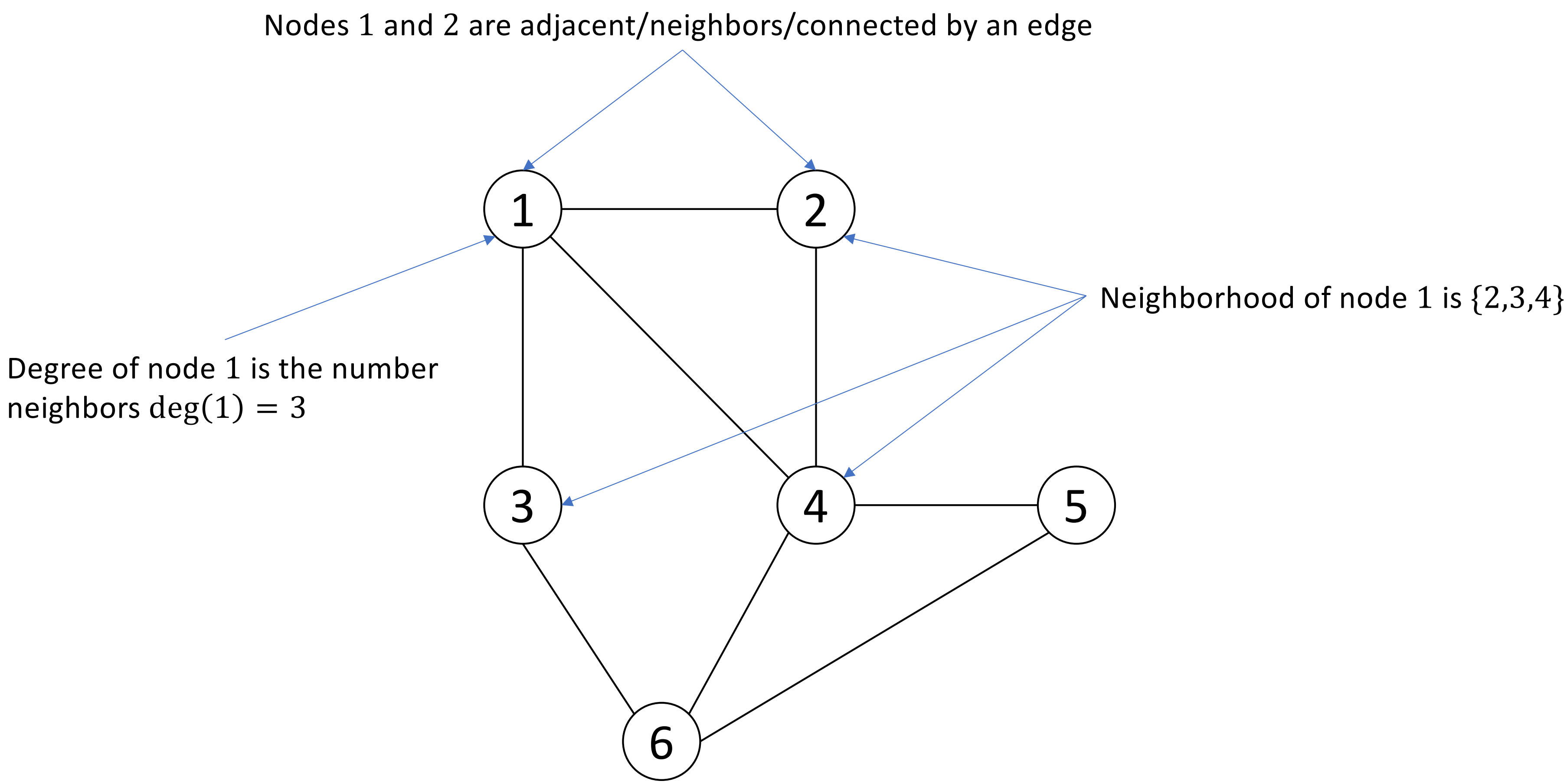
Directed (aka digraphs):

edges have orientation

each edge is a pair of elements of V

Example: (u, v) - a directed edge from u to v, $u, v \in V$

Maximum number of edges is $|V|^2$ allowing self-loops but no multiple edges



Nodes 1 and 2 are adjacent/neighbors/connected by an edge
Degree of node 1 is the number neighbors $\deg(1) = 3$
Neighborhood of node 1 is $\{2,3,4\}$
1
2
3
4
5
6

Weighted graphs

Vertices and/or edges can have weights

Weights on edges help to encode strength or importance of connections

Formally given by a function $w : E \to \mathbb{R}$

Weights on vertices help to encode importance of entities

Formally given by a function $w : V \to \mathbb{R}$

Representations of graphs (CLRS 22.1)

Two most common representations:

Adjacency matrix

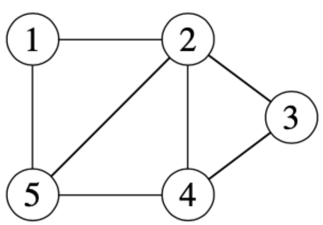
Adjacency lists

Adjacency matrix

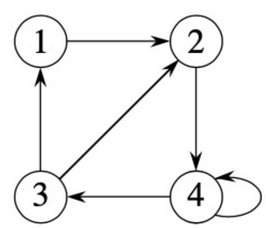
$|V|\times|V|$ matrix $A = (a_{ij})$ such that

$$a_{ij} = \begin{cases} 1, & (i,j) \in E \\ 0, & (i,j) \notin E \end{cases}$$

Examples:



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



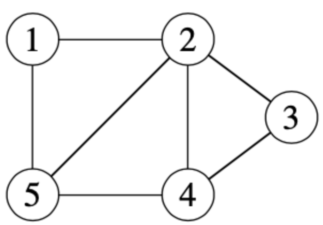
	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

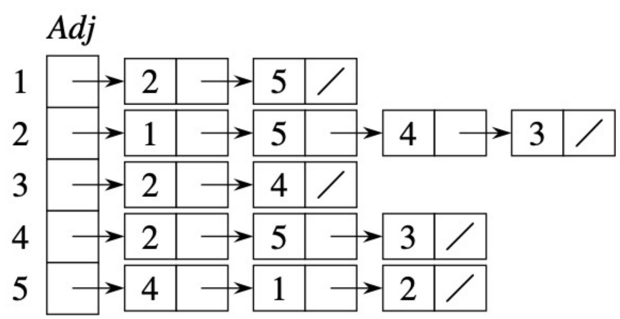
Adjacency lists

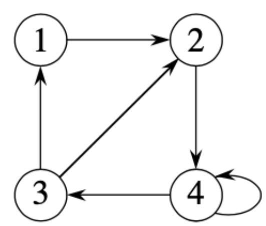
Array Adj of $|V|$ lists, one per vertex

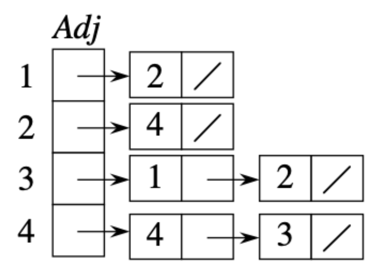
$Adj[u]$ is a list of all vertices v such that $(u, v) \in E$

Examples:









Comparison of representations

Adjacency matrix

Works for both directed and undirected graphs

For weighted graphs: can store weight of an edge in the matrix

Space: $\Theta(|V|^2)$

Time:

to list all neighbors of u: $\Theta(|V|)$

to determine $(u, v) \in E$: $\Theta(1)$

Adjacency lists

Works for both directed and undirected graphs

For weighted graphs: can store weight of an edge in a corr. list elt.

Space: $\Theta(|V| + |E|)$

Time:

to list all neighbors of u: $\Theta(\deg(u))$

to determine $(u, v) \in E$: $O(\deg(u))$

Breadth-First Search BFS (CLRS, 22.2)

Input: Graph $G = (V, E)$, either directed or undirected

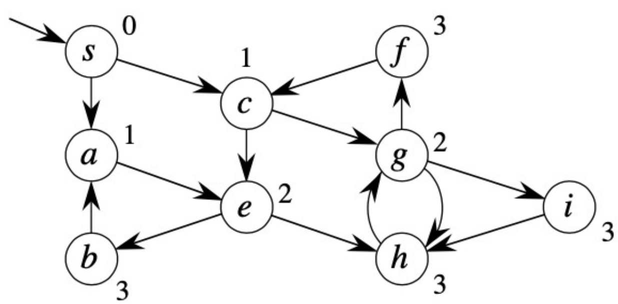
$s \in V$ – the source vertex

Output: $v.d$ = distance (smallest # of edges) from s to v, for all $v \in V$

Also known as unweighted shortest path.

Can be used to solve reachability problem.

Example:



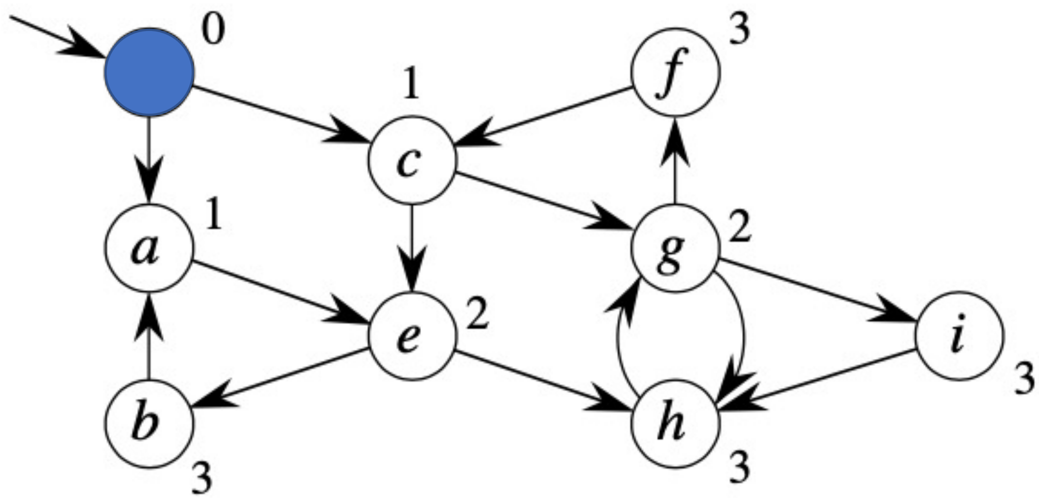
Idea

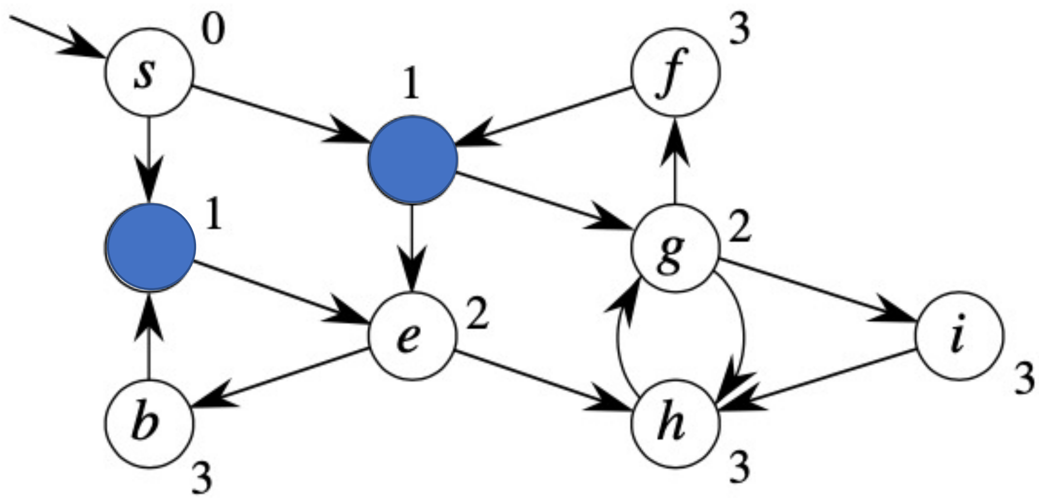
Send a wave out of s

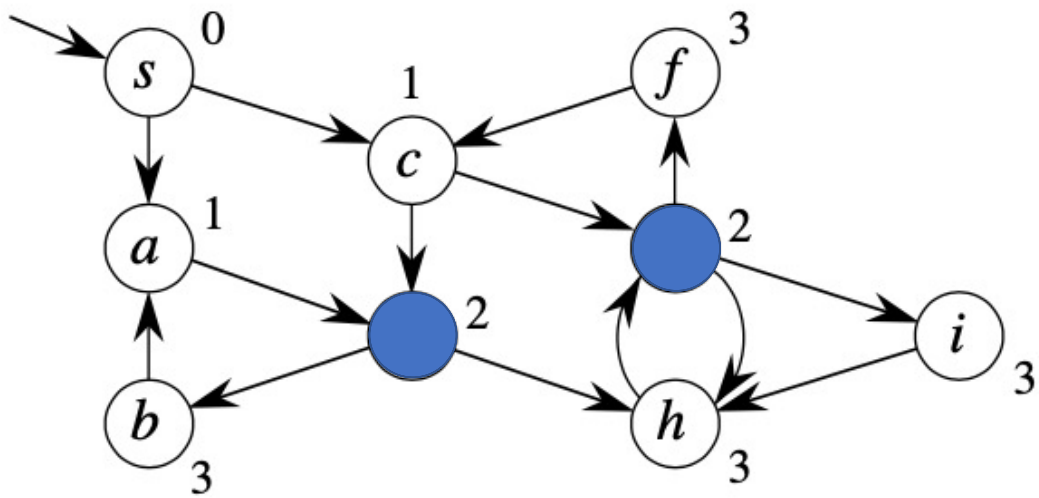
- First hits all vertices 1 edge from s
- Then hits all vertices 2 edges from s
- So on…

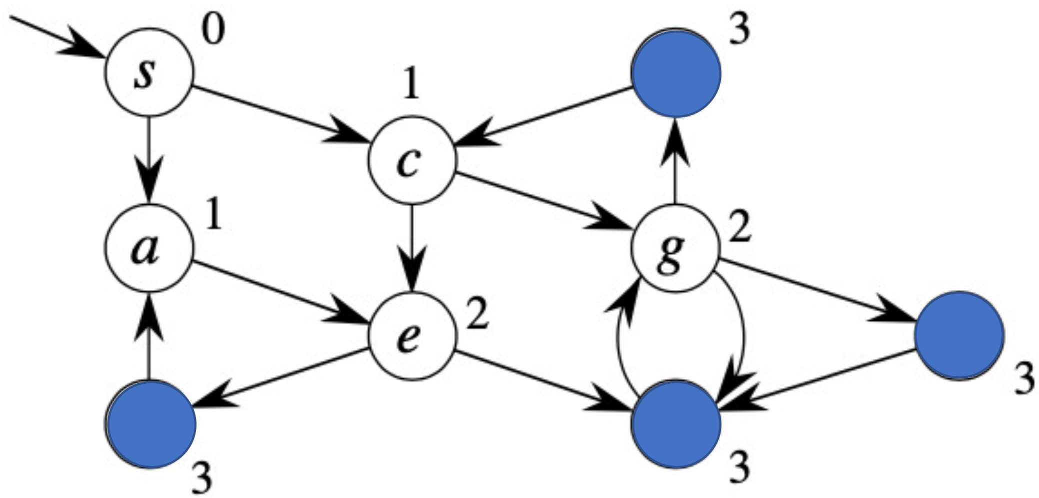
Use FIFO queue Q to maintain wavefront

- $v \in Q$ if and only if wave has hit v but hasn't come out of v yet

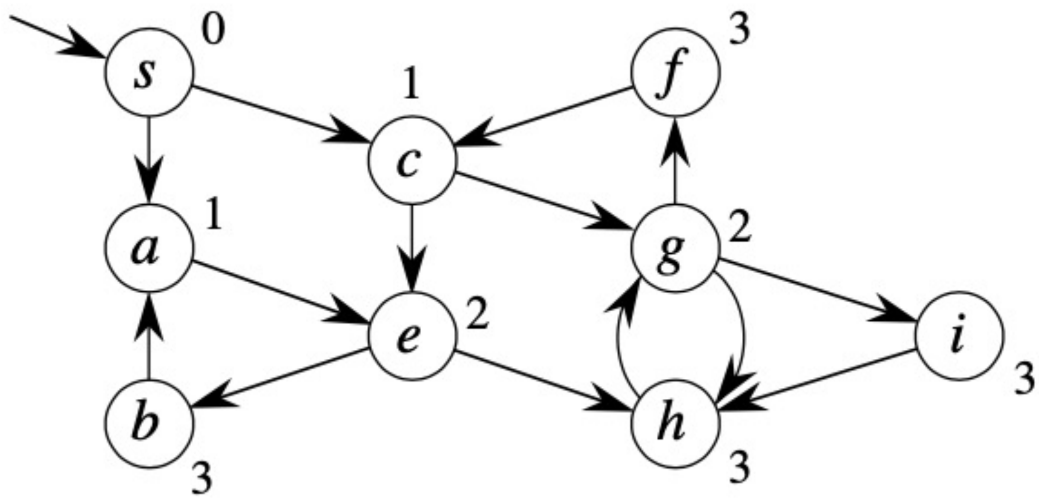

0
1
1
2
2
3
3
3
3
3
a
b
c
e
f
g
h
i


s
0
1
f
3
1
g
2
e
2
i
3
b
3
h
3


s
0
c
1
f
3
a
1
2
2
i
3
b
3
h
3



s
0
c
1
a
1
e
2
g
2
3
3
3
3



s
0
c
1
f
3
a
1
g
2
e
2
i
3
b
3
h
3

$BFS(G = (V, E), s)$

$\quad$ $\boldsymbol{for}\ u \in V - \{s\}$

$\quad\quad$ $u.d \leftarrow \infty$

$\quad$ $s.d \leftarrow 0$

$\quad$ initialize queue Q

$\quad$ $Q.enqueue(s)$

$\quad$ $\boldsymbol{while}\ Q.size(\,) > 0$

$\quad\quad$ $u \leftarrow Q.dequeue()$

$\quad\quad$ $\boldsymbol{for}\ v \in G.Adj[u]$

$\quad\quad\quad$ $\boldsymbol{if}\ v.d = \infty$

$\quad\quad\quad\quad$ $v.d \leftarrow u.d + 1$

$\quad\quad\quad\quad$ $Q.enqueue(v)$

BFS may not reach all vertices

Time = $O(|V| + |E|)$

$O(|V|)$ because every vertex is enqueued at most once

$O(|E|)$ because every vertex is dequeued at most once and we examine (u, v) only when u is dequeued.

Outstanding issues

What if we want to construct actual path from s to v realizing $v.d$?

Keep another attribute $v.\pi$ – predecessor of v, namely, $v.\pi$ is the vertex u responsible for enqueueing v

Set of edges $\{(v.\pi, v) : v \neq s\}$ forms a tree

See CLRS for more details and a formal proof of correctness

Depth-First Search DFS (CLRS, 22.3)

Input: $G = (V, E)$, directed or undirected

Output: 2 timestamps on each vertex

- $v.d$ = discovery time
- $v.f$ = finishing time

Can be used to solve reachability, but **NOT** unweighted shortest paths

Goal is to methodically explore every edge

Start over from different vertices as necessary

As soon as we discover a vertex, explore from it

- Unlike BFS, which puts a vertex on a queue to explore from it later

Discovery and finishing times:

- Unique integers from 1 to $2|V|$
- For all $v \in V$ we have $v.d < v.f$

As DFS progresses, every vertex has a color (for analysis and discussion purposes):

- WHITE = undiscovered
- GRAY = discovered, but not finished (not done exploring from it)
- BLACK = finished (have found everything reachable from it)

$DFS(G)$

$\quad for\ u \in V$

$\quad\quad u.color \leftarrow WHITE$

$\quad time \leftarrow 0$ // global variable

$\quad for\ u \in V$

$\quad\quad if\ u.color = WHITE$

$\quad\quad\quad DFS - Visit(G, u)$

$DFS - Visit(G, u)$

$\quad time \leftarrow time + 1$

$\quad u.d \leftarrow time$

$\quad u.color \leftarrow GRAY$ // discover u

$\quad for\ v \in Adj[u]$ // explore (u, v)

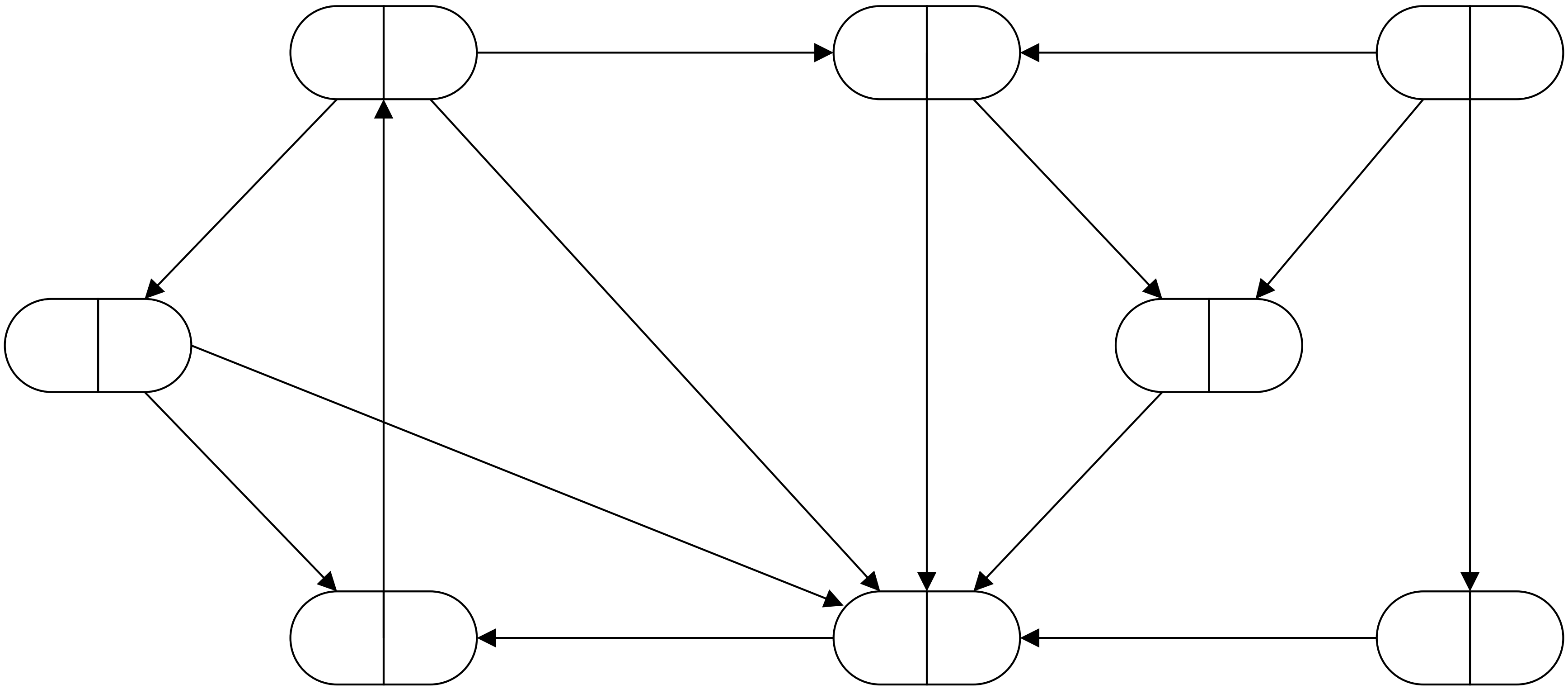
$\quad\quad if\ v.color = WHITE$

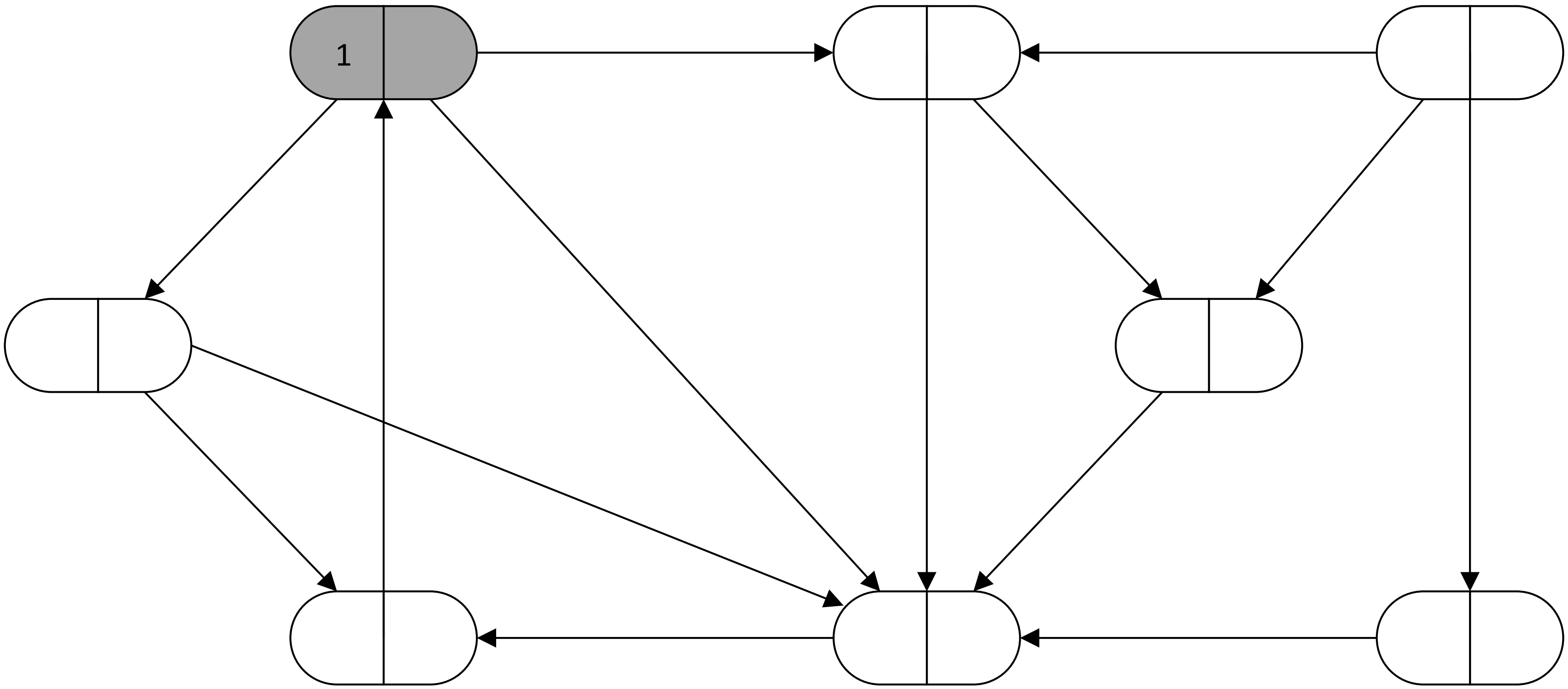
$\quad\quad\quad DFS - Visit(G, v)$

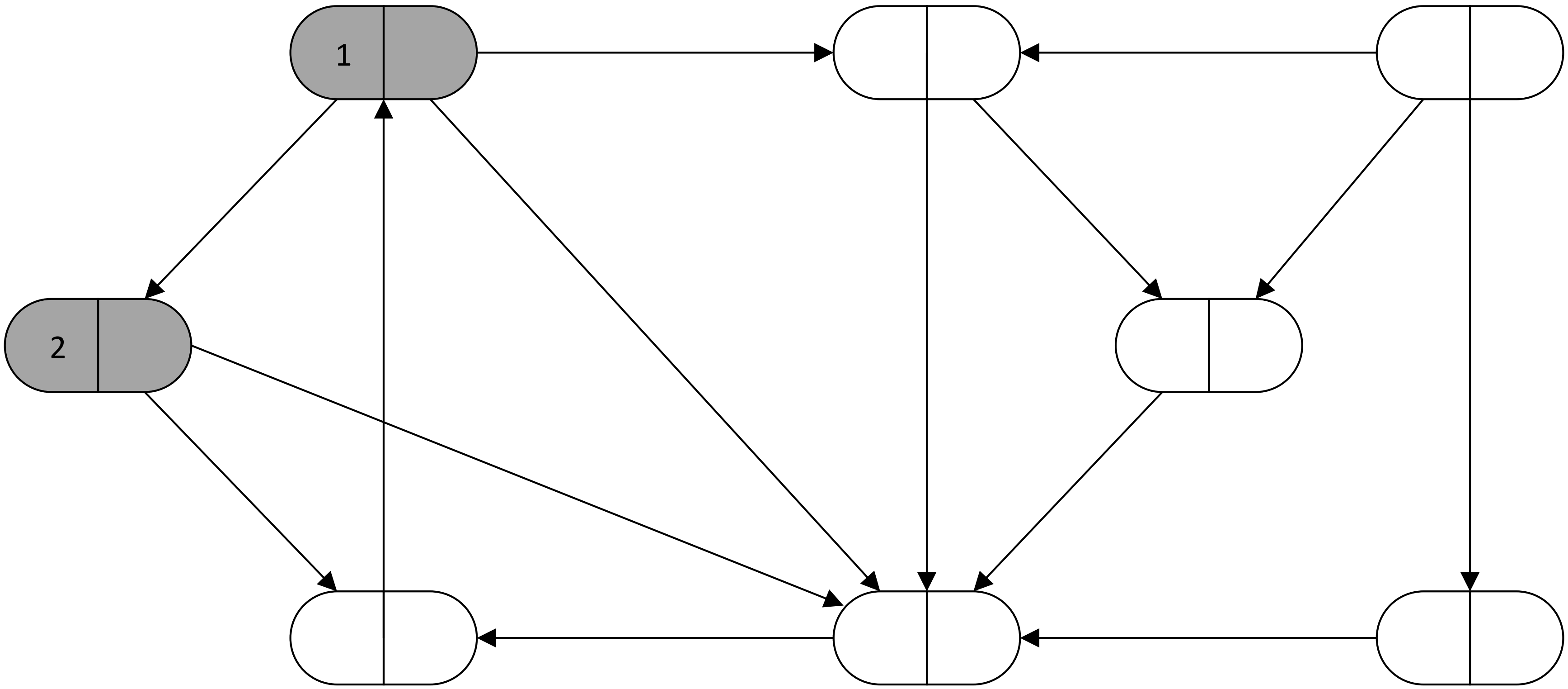
$\quad u.color \leftarrow BLACK$

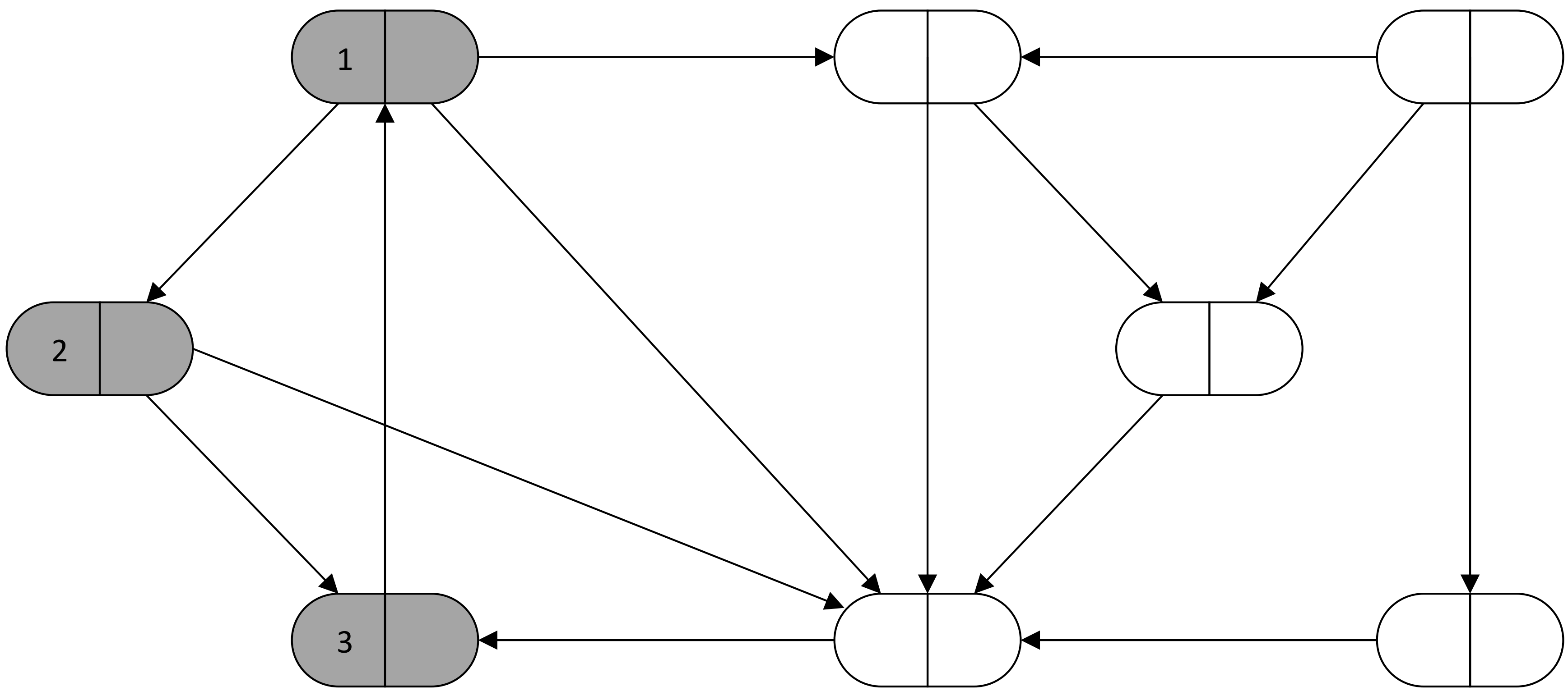
$\quad time \leftarrow time + 1$

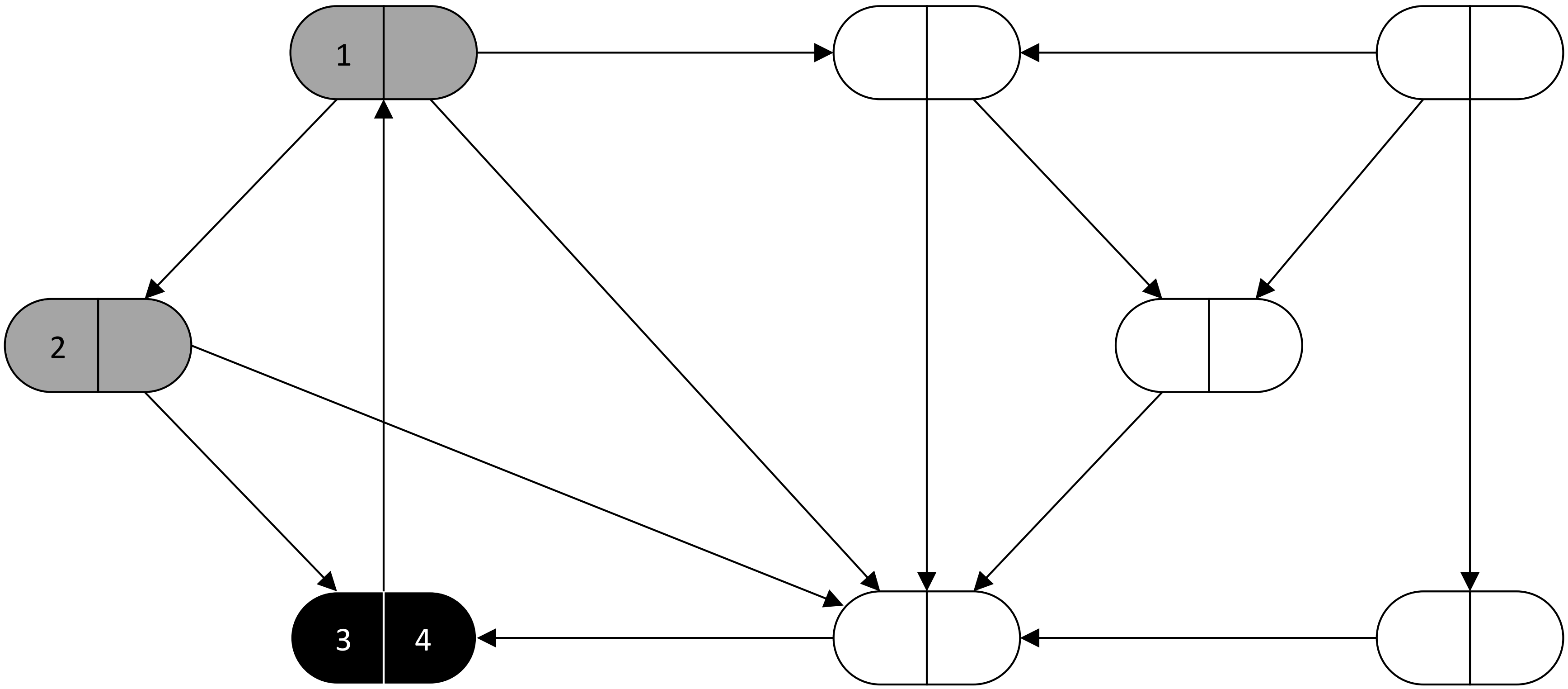
$\quad u.f \leftarrow time$ // finish u

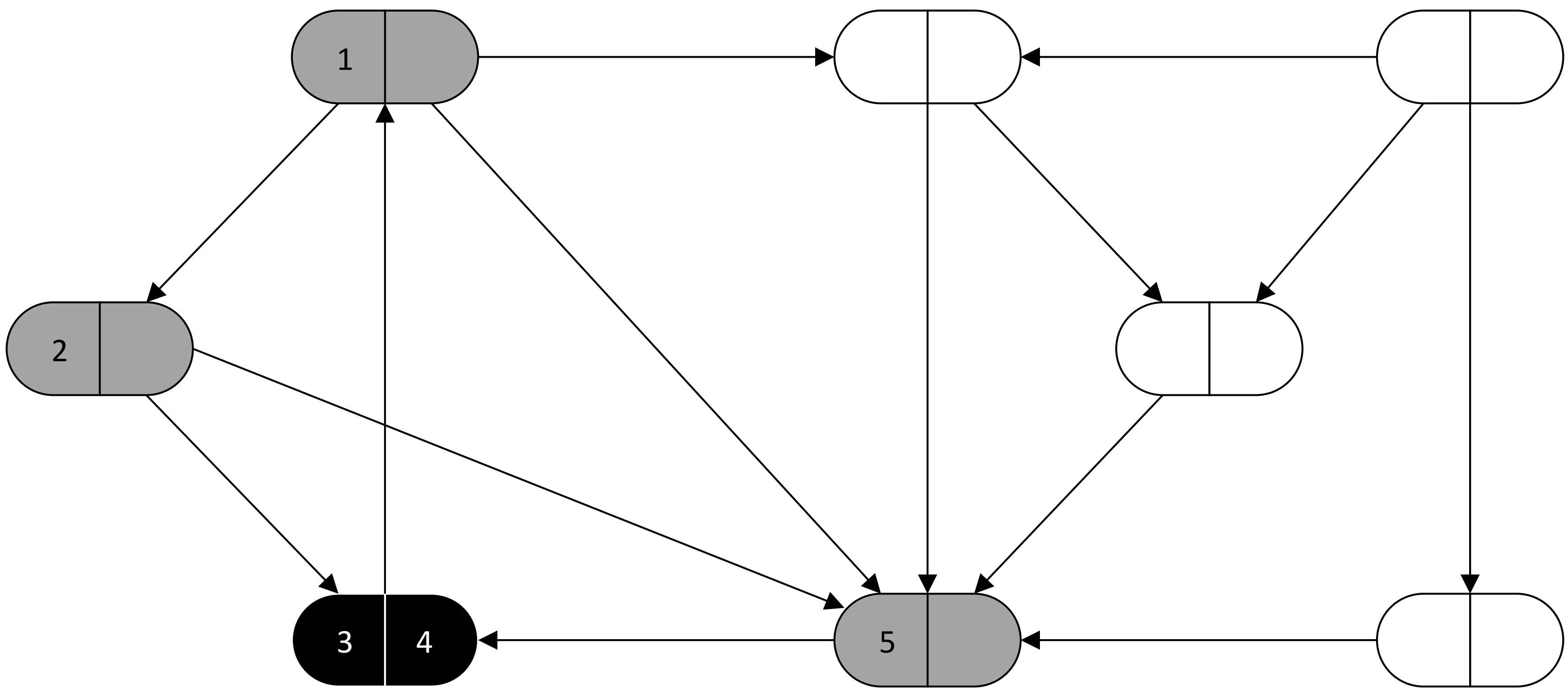


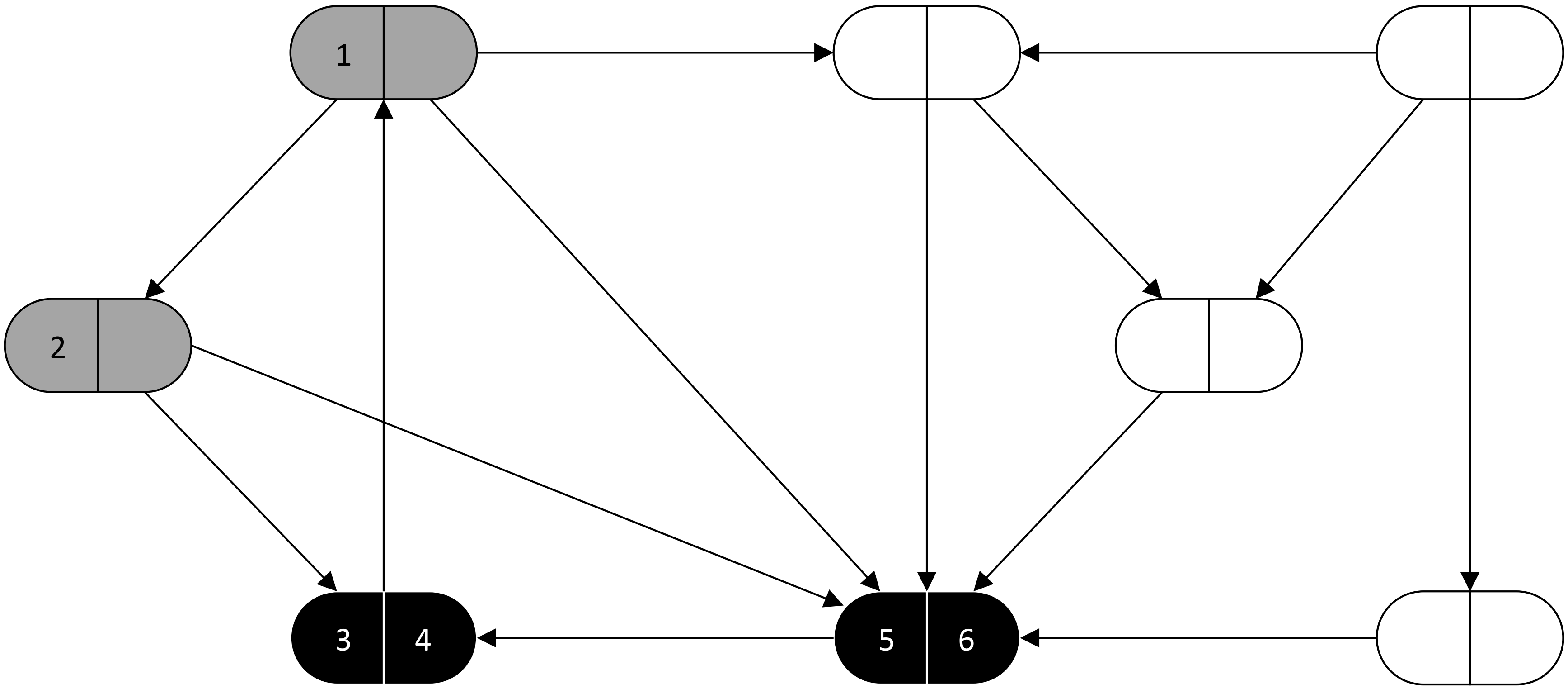

1

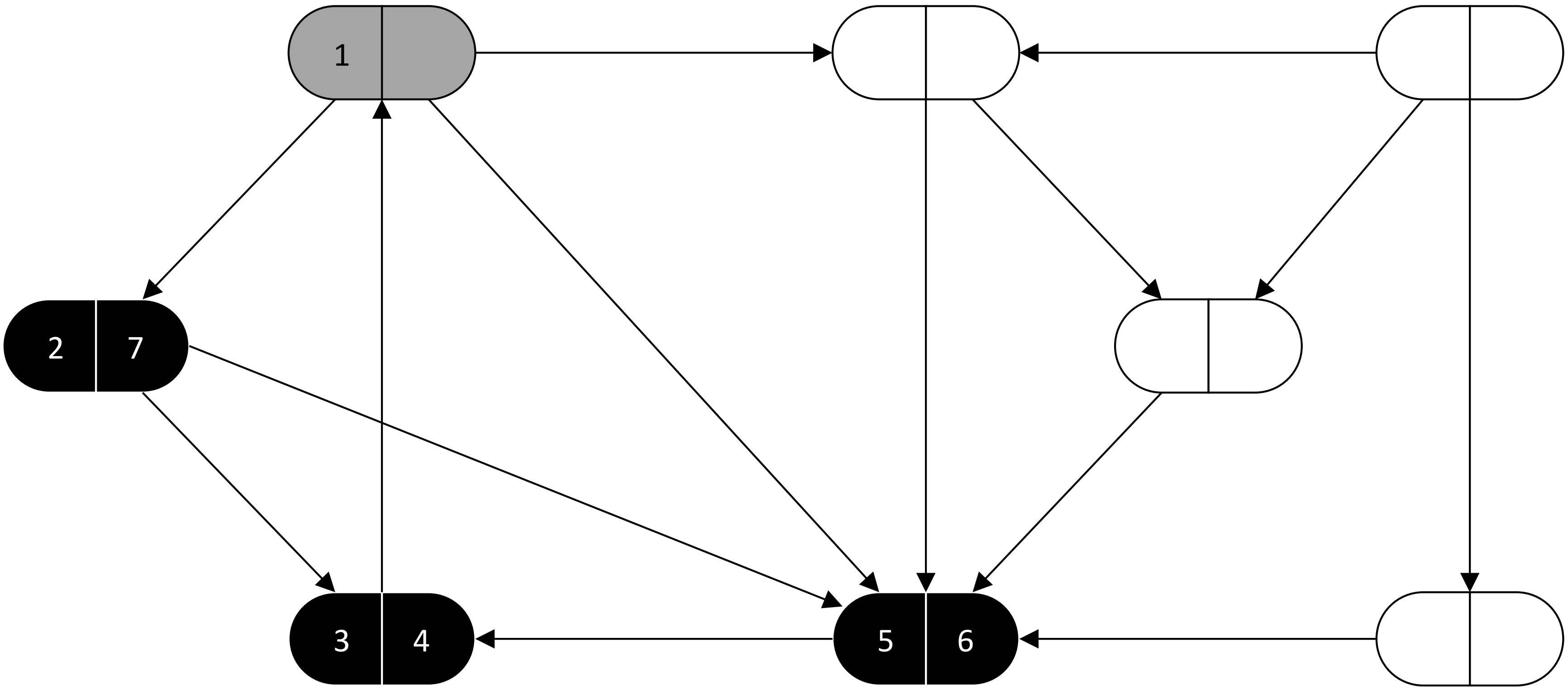

1
2

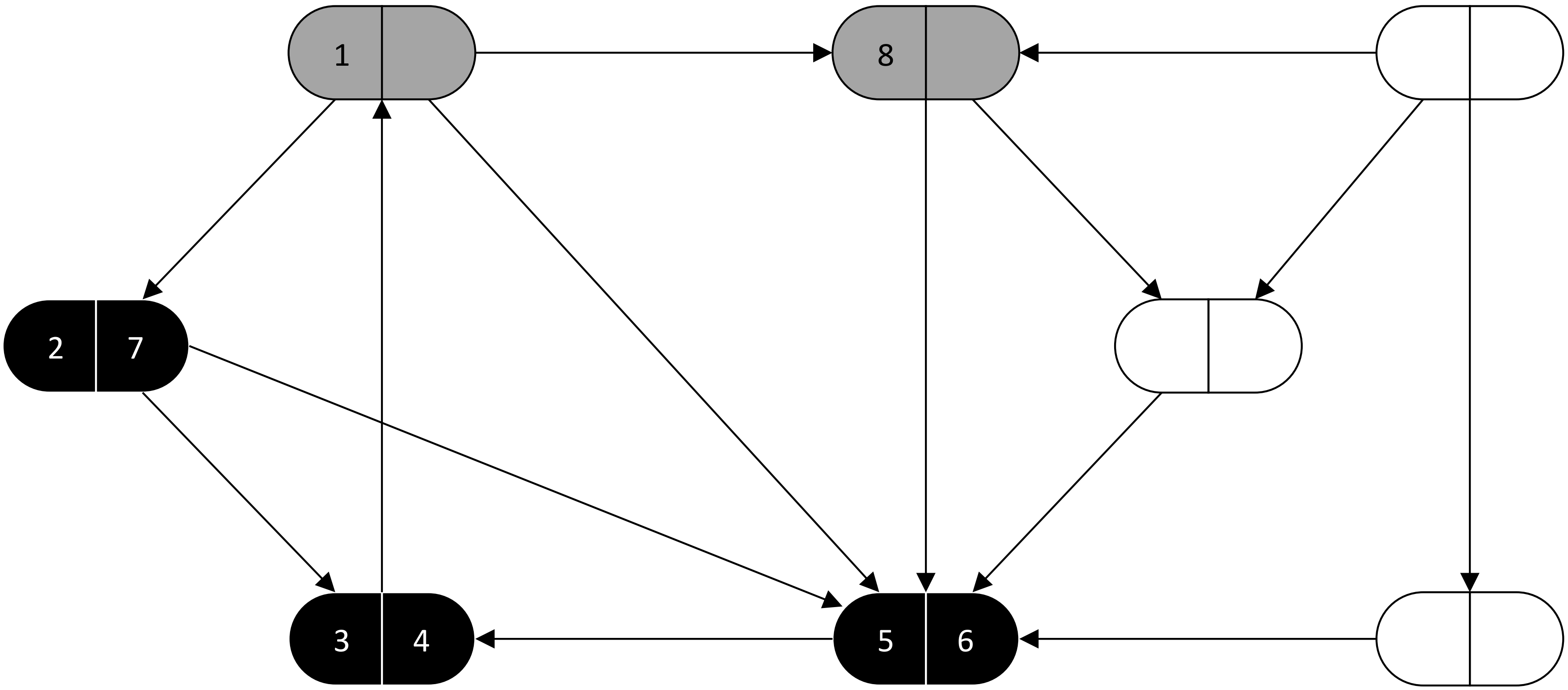

1
2
3

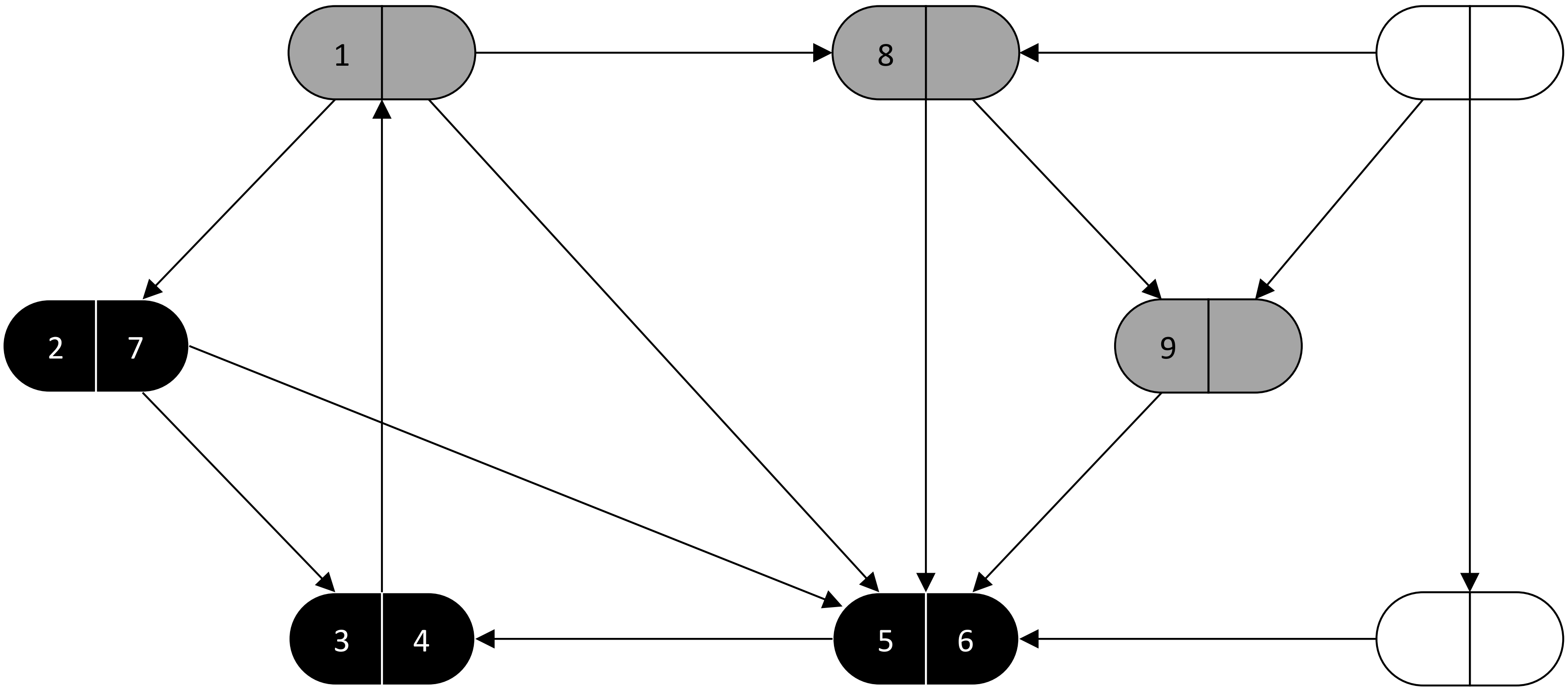

1
2
3
4

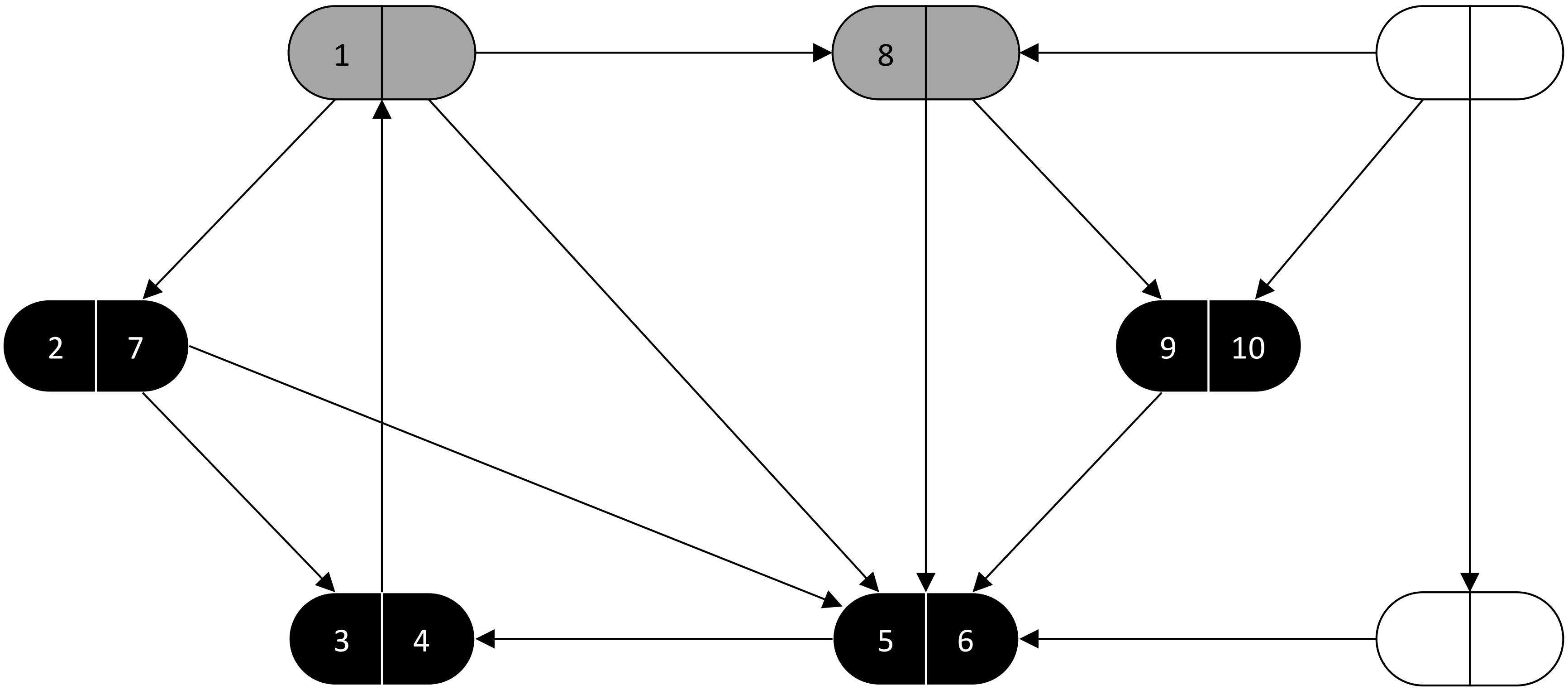

1
2
3
4
5

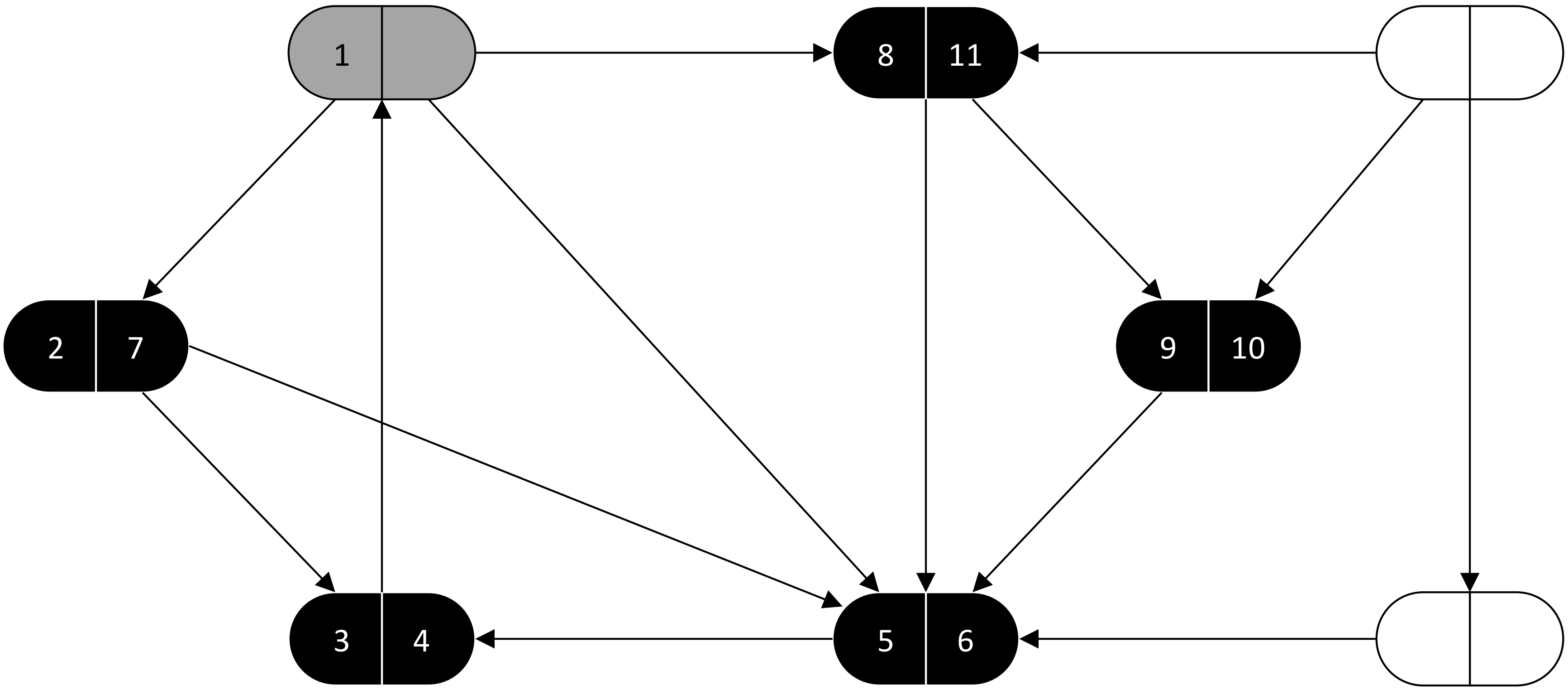

1
2
3
4
5
6

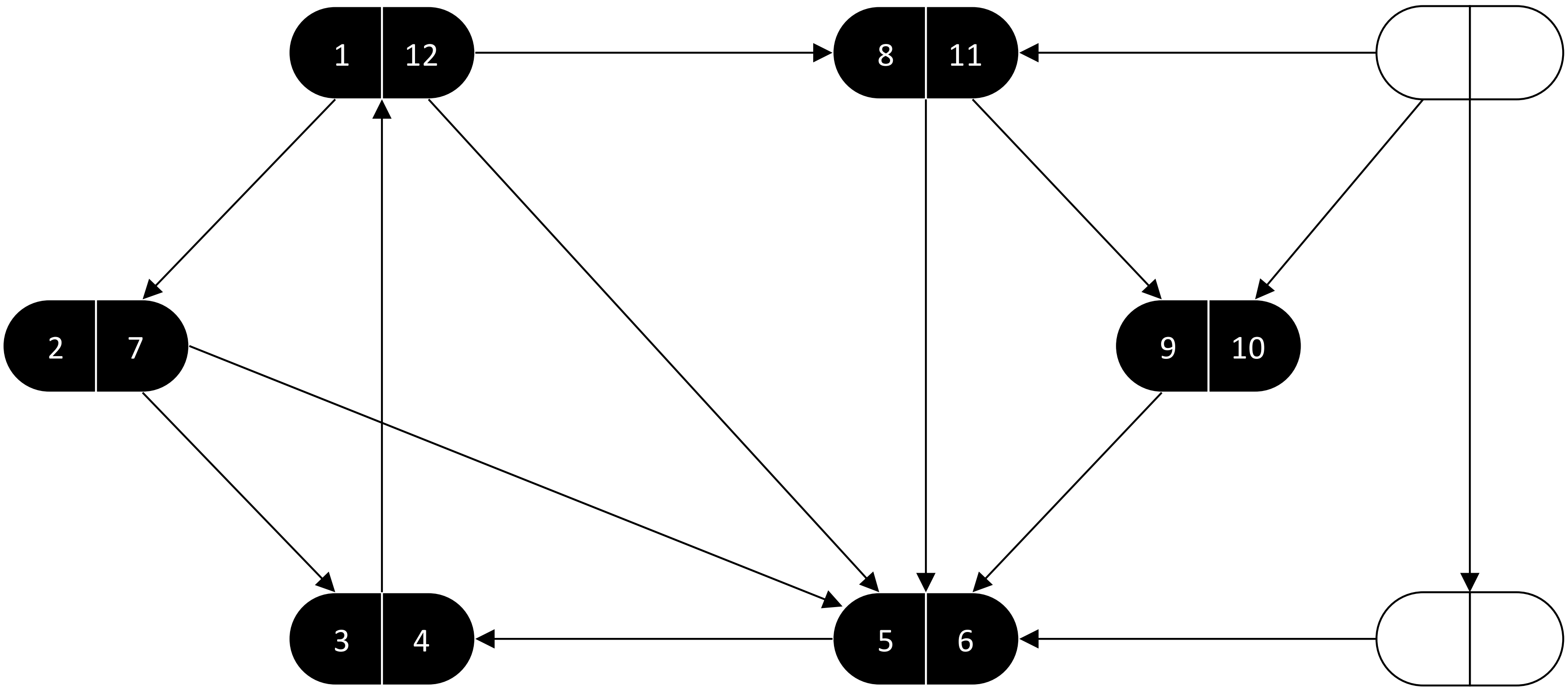

1
2
7
3
4
5
6

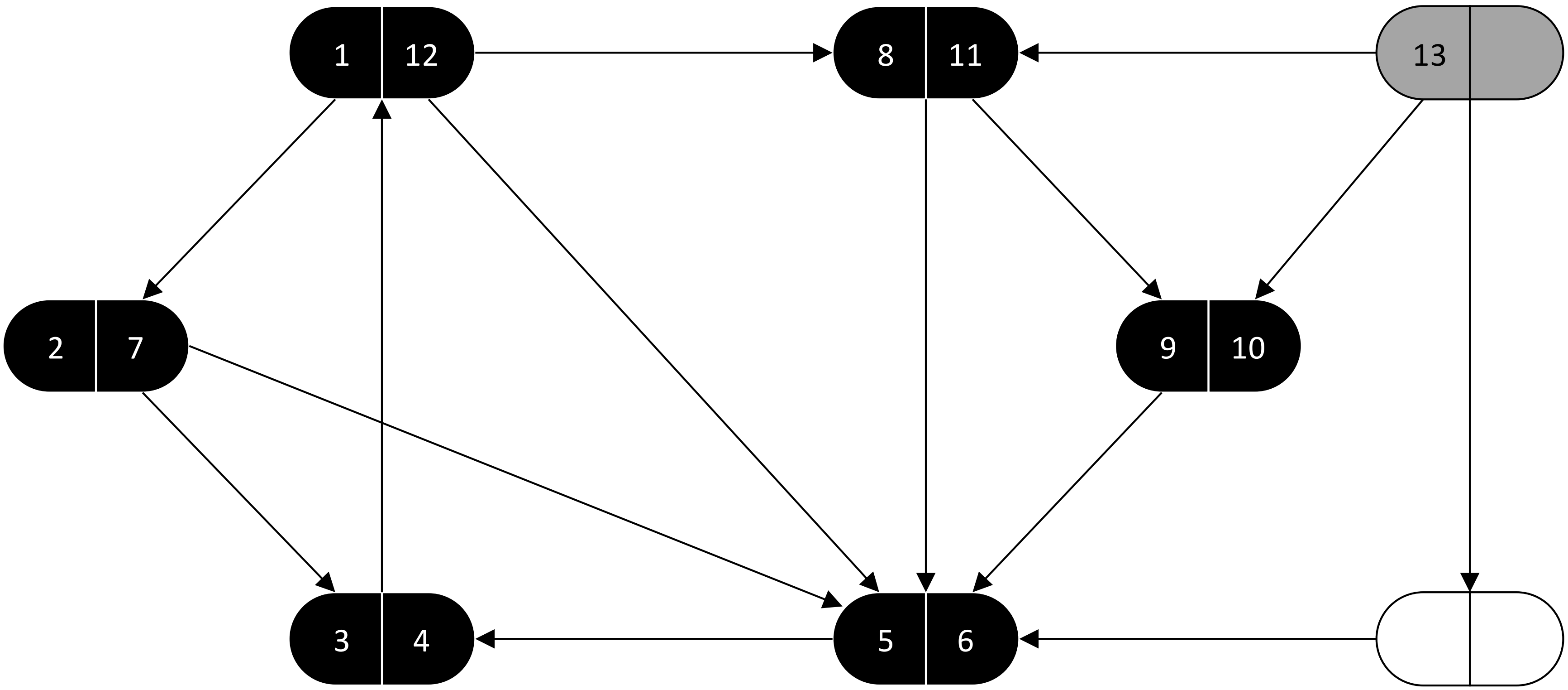

1
8
2
7
3
4
5
6

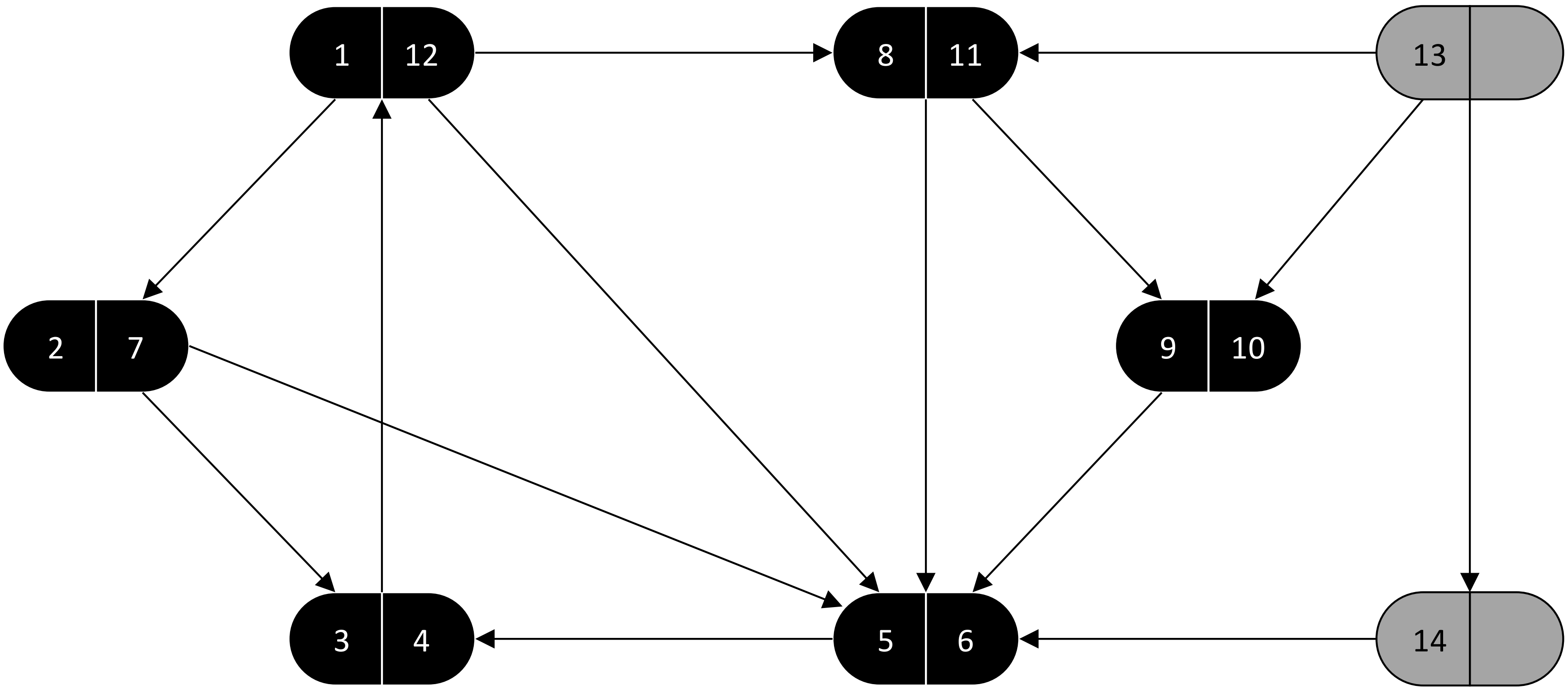

1
8
2
7
9
3
4
5
6


1
8
2
7
9
10
3
4
5
6

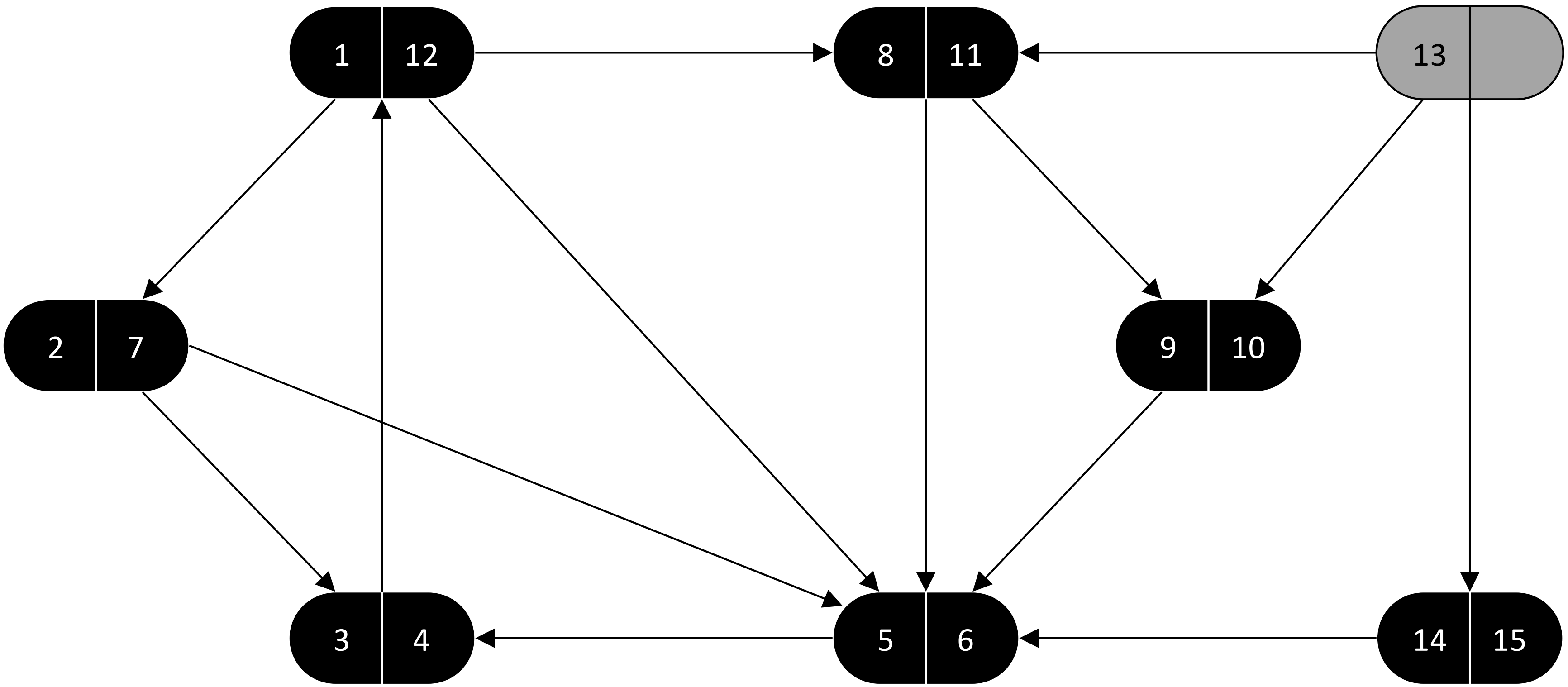

1
8
11
2
7
9
10
3
4
5
6

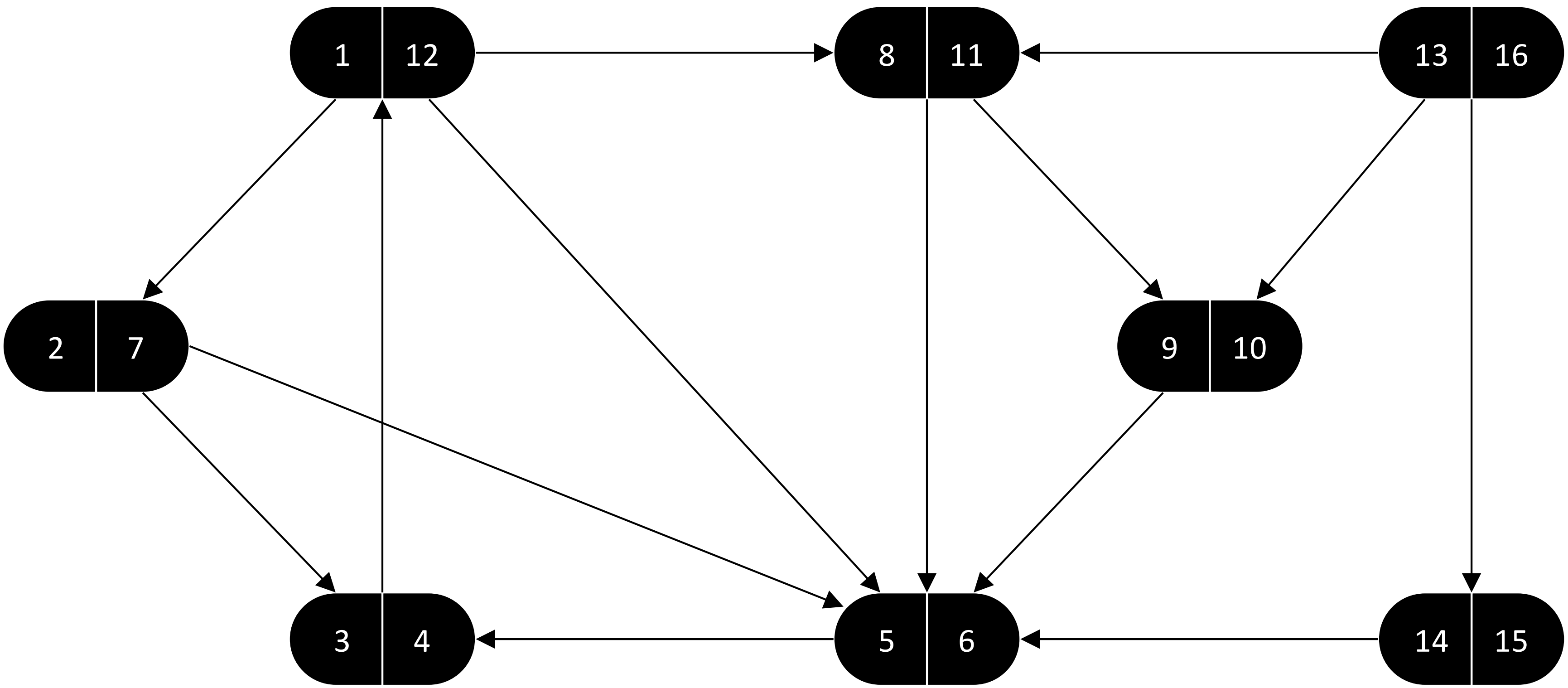

1
12
8
11
2
7
9
10
3
4
5
6


1
12
8
11
13
2
7
9
10
3
4
5
6



1
12
8
11
13
2
7
9
10
3
4
5
6
14


1
12
8
11
13
2
7
9
10
3
4
5
6
14
15


1
12
8
11
13
16
2
7
9
10
3
4
5
6
14
15

Time = $\Theta(|V| + |E|)$

- Similar to BFS analysis
- $\Theta(|V| + |E|)$ instead of $O(|V| + |E|)$, since we are guaranteed to examine each edge

DFS forms a depth-first forest consisting of at least one depth-first tree

Each tree edge is (u, v) such that $u.color = GRAY$ and $v.color = WHITE$ when (u, v) is explored

Parenthesis theorem

For all u, v exactly one of the following holds:

1. Time intervals $[u.d, u.f]$ and $[v.d, v.f]$ are disjoint (u and v belong to different depth-first trees or different branches of same tree)
2. Time interval $[v.d, v.f]$ is a subinterval of $[u.d, u.f]$ (v is a descendant of u in depth-first tree)
3. Time interval $[u.d, u.f]$ is a subinterval of $[v.d, v.f]$ (u is a descendant of v in depth-first tree)

"Time intervals of vertices behave as parenthesis"

- ( )[ ], ([ ]), [( )] are OK
- ([ )], [( ]) are NOT OK

White-path theorem

v is a descendant of u if and only if at time $u.d$ there is a path $u \to v$ consisting only of WHITE vertices (except for u which is colored GRAY)

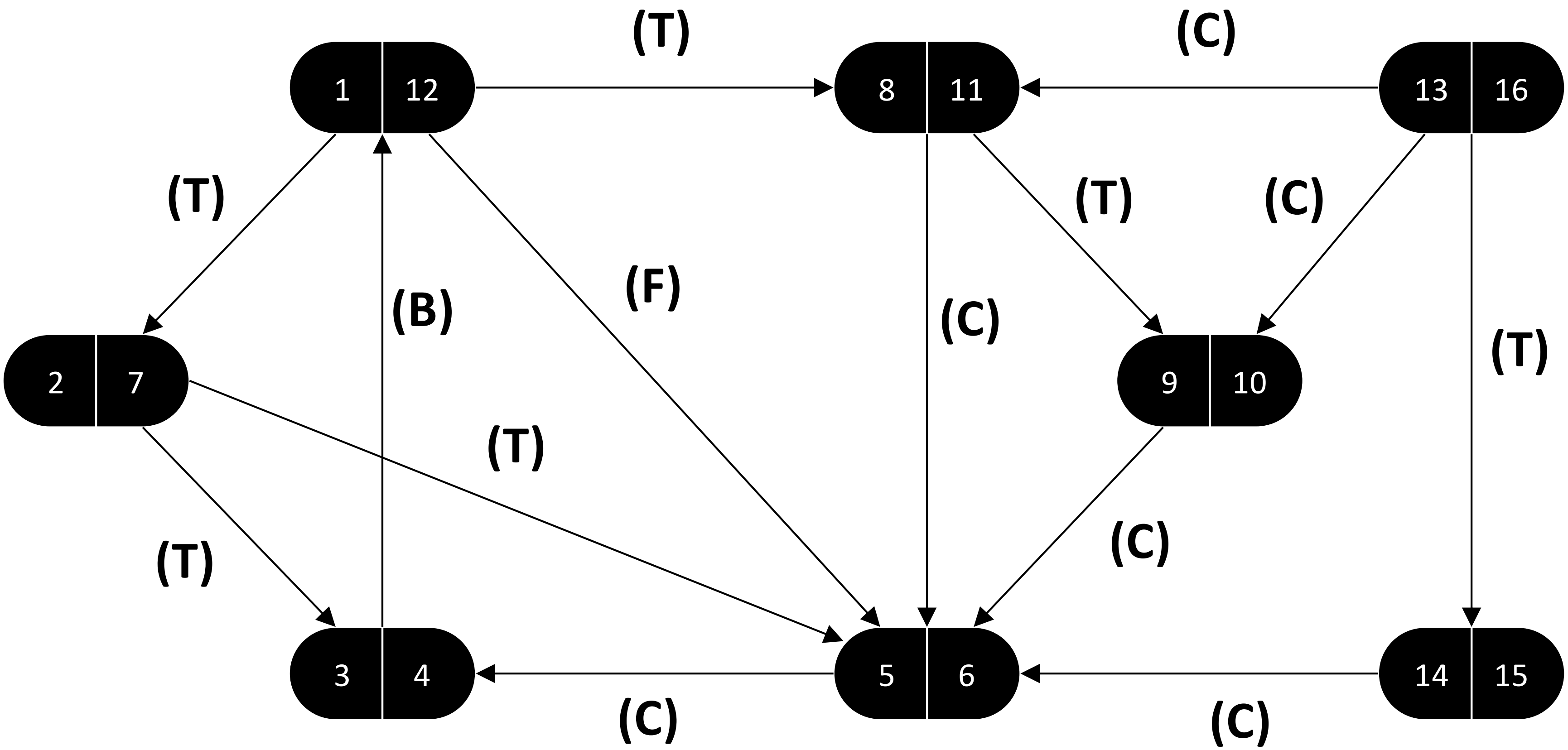
Classification of edges

Tree edge (T): appears in the depth-first forest.

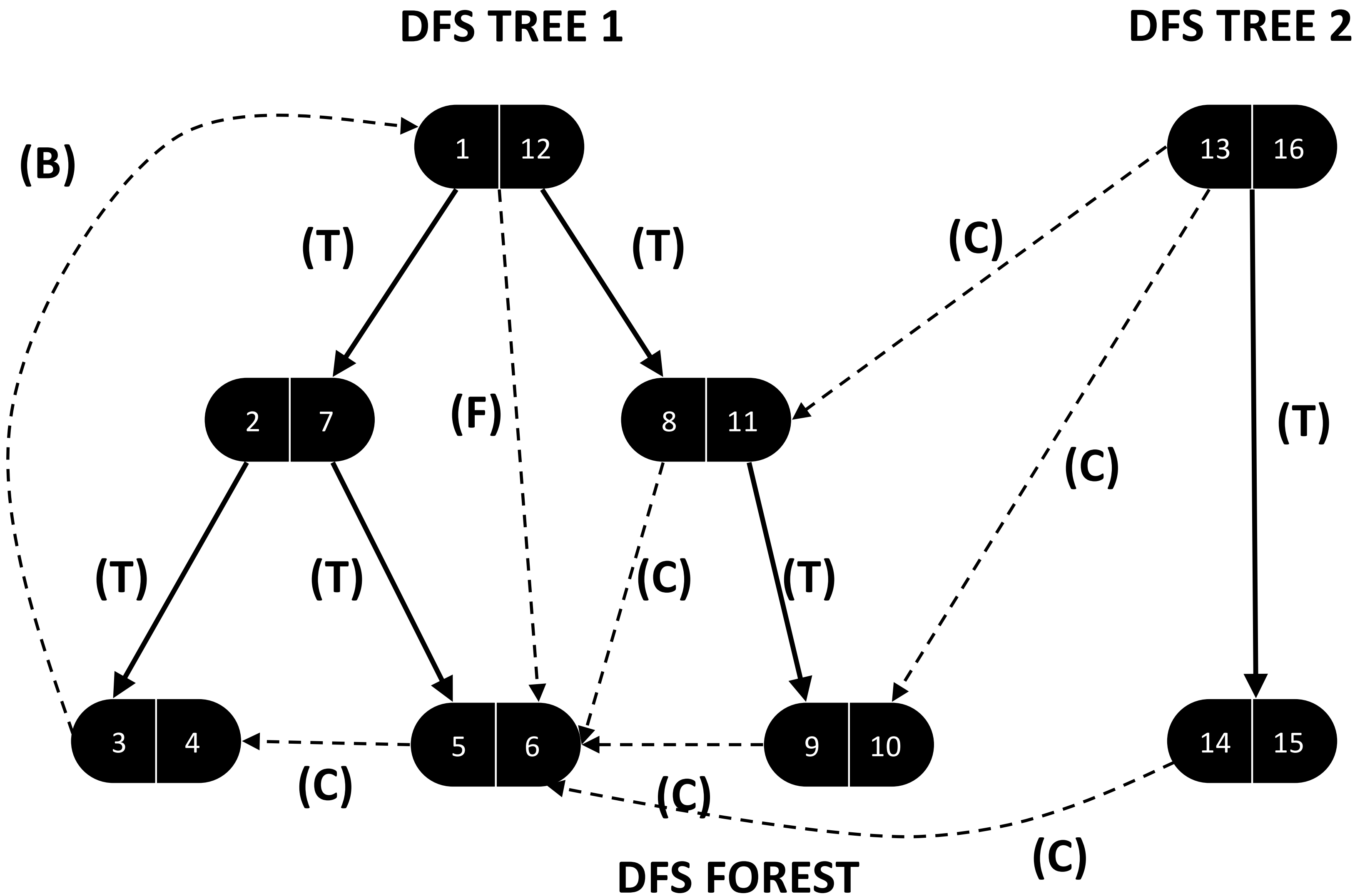
Back edge (B): (u, v), where u is a descendant of v

Forward edge (F): (u, v), where v is a descendant of u, but not a tree edge

Cross edge (C): any other edge



(T)
(C)
1
12
8
11
13
16
(T)
(T)
(C)
(B)
(F)
(C)
(T)
2
7
9
10
(T)
(T)
(C)
3
4
5
6
14
15
(C)
(C)


DFS TREE 1
DFS TREE 2
1
12
2
7
8
11
3
4
5
6
9
10
13
16
14
15
(B)
(T)
(T)
(F)
(T)
(T)
(C)
(T)
(C)
(C)
(C)
(T)
(C)
(C)
DFS FOREST

Extra properties:

1. A directed graph contains a cycle if and only if DFS reveals a back edge.

2. DFS on undirected graphs reveals only tree and back edges, no forward or cross edges

Topological sort (CLRS 22.4)

DAG = directed acyclic graph

A directed graph with no cycles (DFS reveals no back edges)

Good for modelling partial order:

1. $a > b$ and $b > c$ implies that $a > c$
2. But may have a and b that are incomparable

Can always complete it to a total order

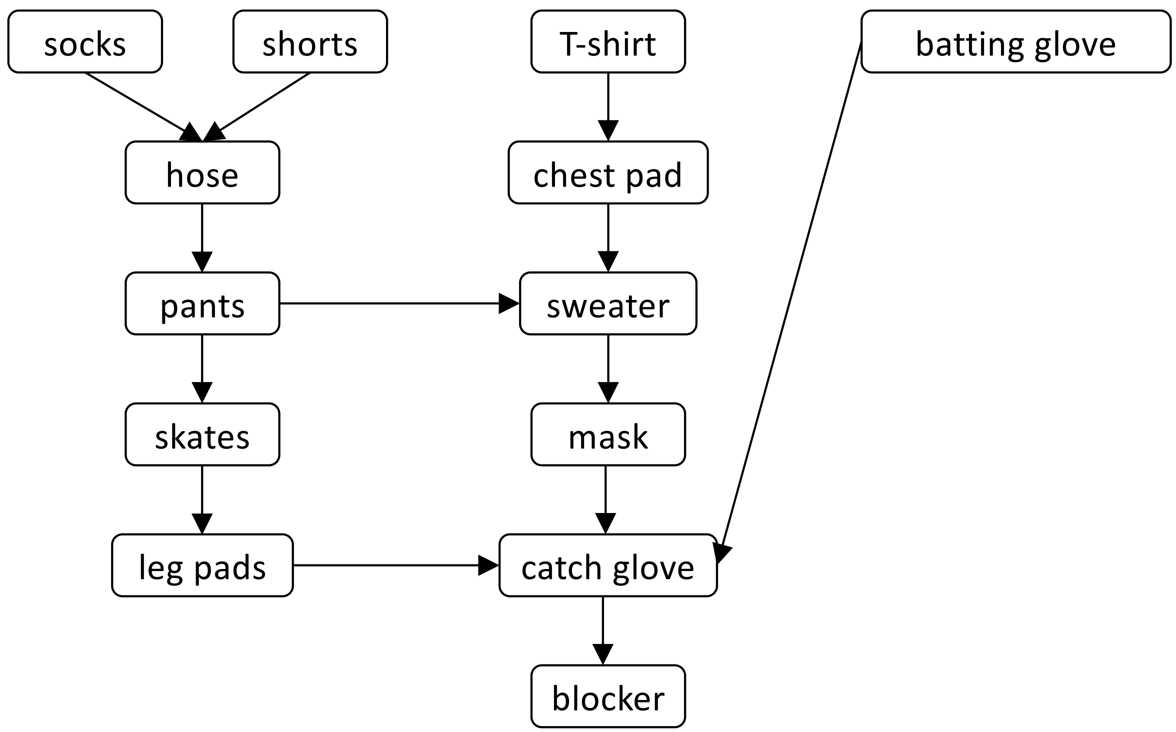
This is what topological sort does

Topological sort formally

Input: $G = (V, E)$ – a dag

Output: a linear ordering of V such that if $(u, v) \in E$ then u appears before v in the ordering

Dag of dependencies for putting on goalie equipment



To perform topological sort:

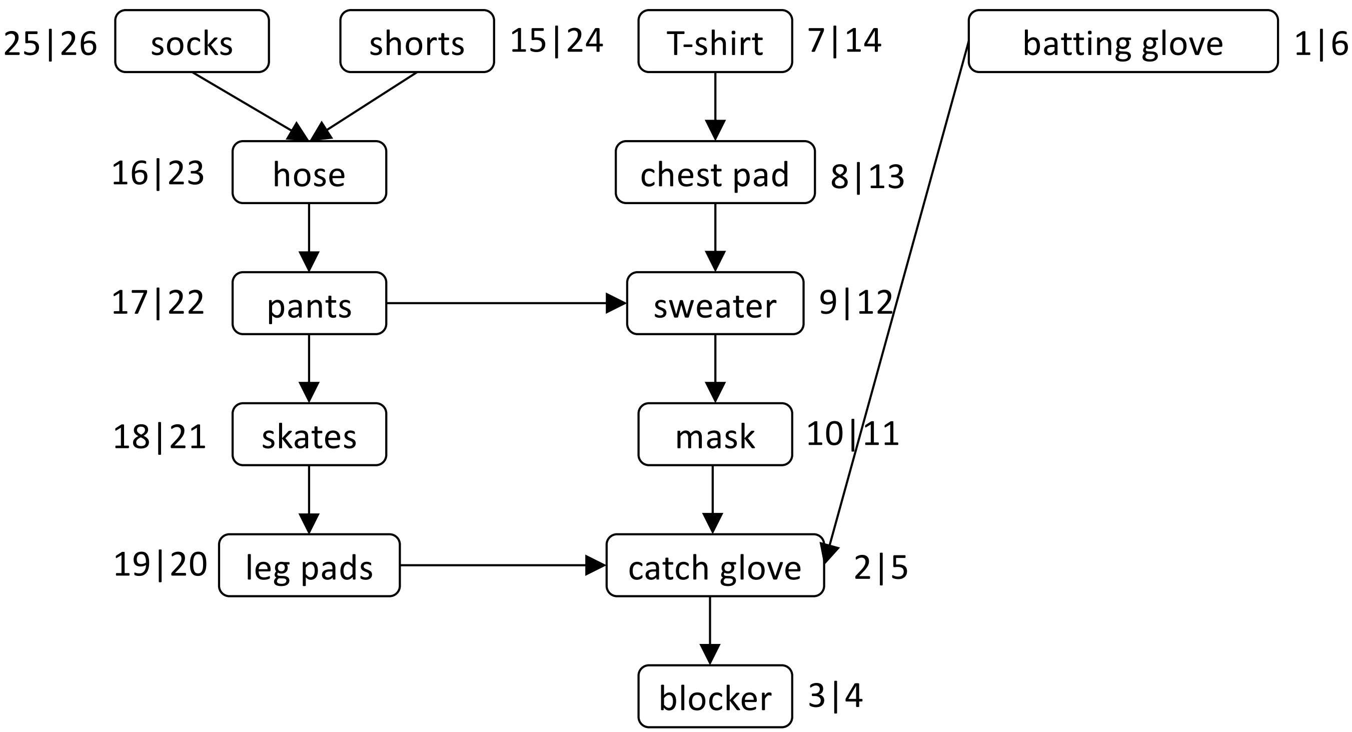
- call DFS to compute finishing times $v.f$ for all $v \in V$
- output vertices in order of decreasing finishing times

Do not explicitly sort vertices after DFS (this would blow up running time)

- As a vertex is finished being explored, place it in the front of the output list
- When done, the list contains vertices in topological order

Time complexity: $\Theta(|V| + |E|)$

Exercise: write down pseudocode from scratch



Topological sort:

26	socks
24	shorts
23	hose
22	pants
21	skates
20	leg pads
14	T-shirt
13	chest pad
12	sweater
11	mask
6	batting glove
5	catch glove
4	blocker

Proof of correctness

Only need to show that $(u, v) \in E$ implies $v.f < u.f$

When we explore (u, v) what are colors of u and v?

- u is GRAY
- Case v is WHITE:
  - v becomes descendant of u (by white-path theorem)
  - $[v.d, v.f]$ is a subinterval of $[u.d, u.f]$ (by parenthesis theorem) therefore $u.d < v.d < v.f < u.f$, as desired
- Case v is BLACK:
  - v is already finished, while we are still exploring u
  - Therefore $v.f < u.f$

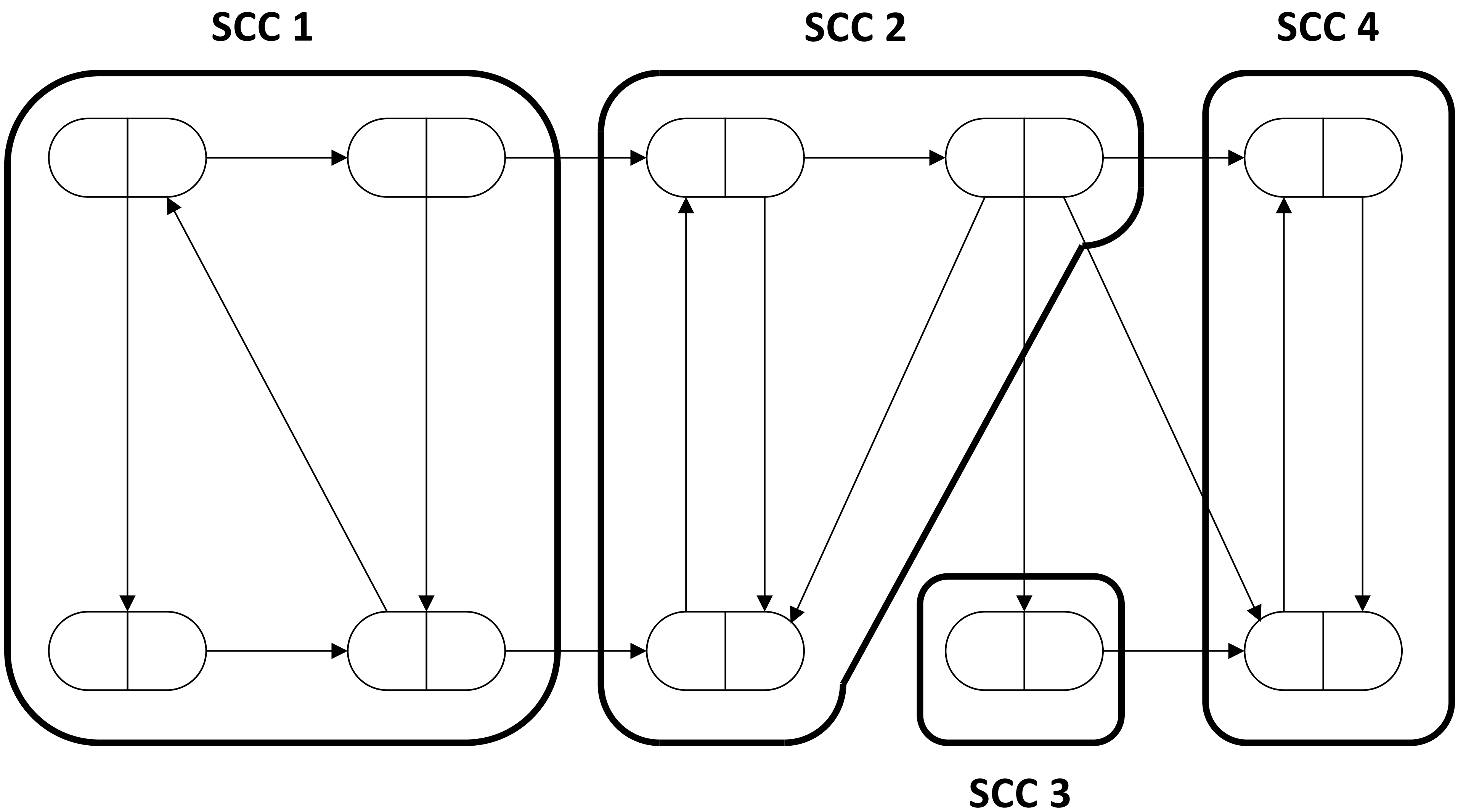
Case v is GRAY:
- (u, v) becomes a back edge
- contradicts property of dags
- impossible

Strongly connected components (CLRS, 22.5)

A strongly connected component (SCC) of G is a maximal set of vertices $C \subseteq V$ such that for all $u, v \in C$ there is a path from u to v and there is a path from v to u

"vertices that are mutually reachable from each other"

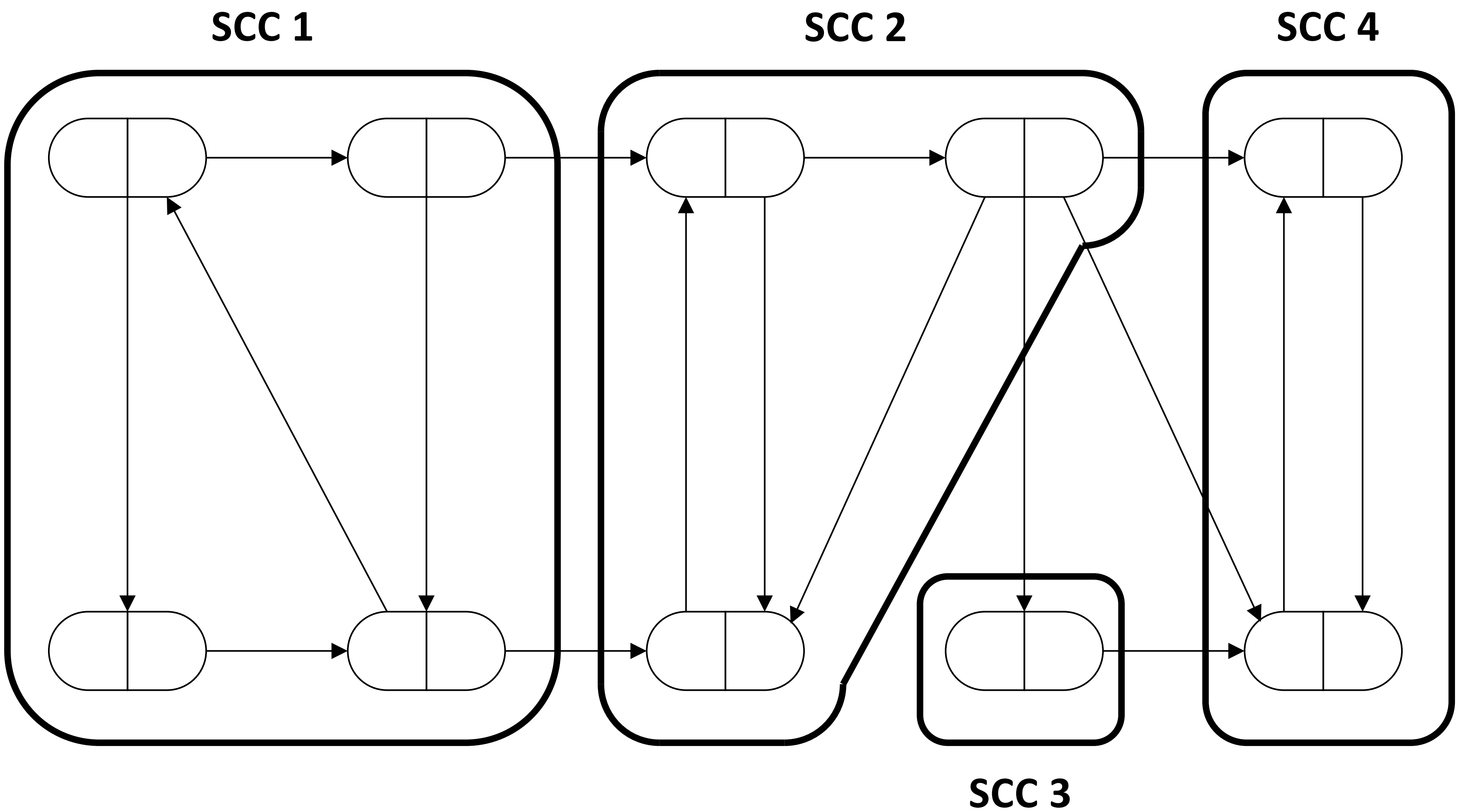
Vertices of a dag are partitioned into disjoint SCCs

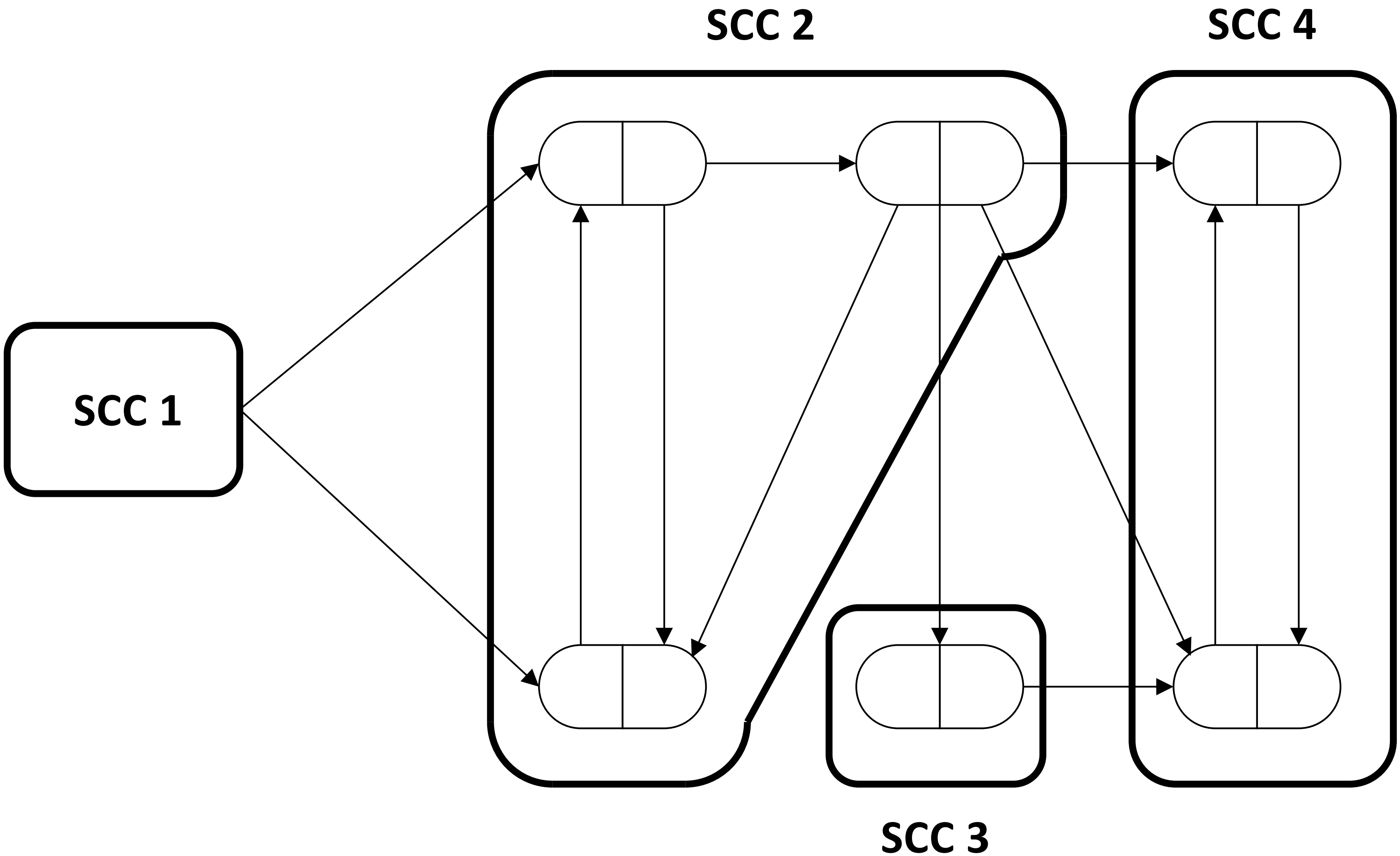

SCC 1
SCC 2
SCC 4
SCC 3

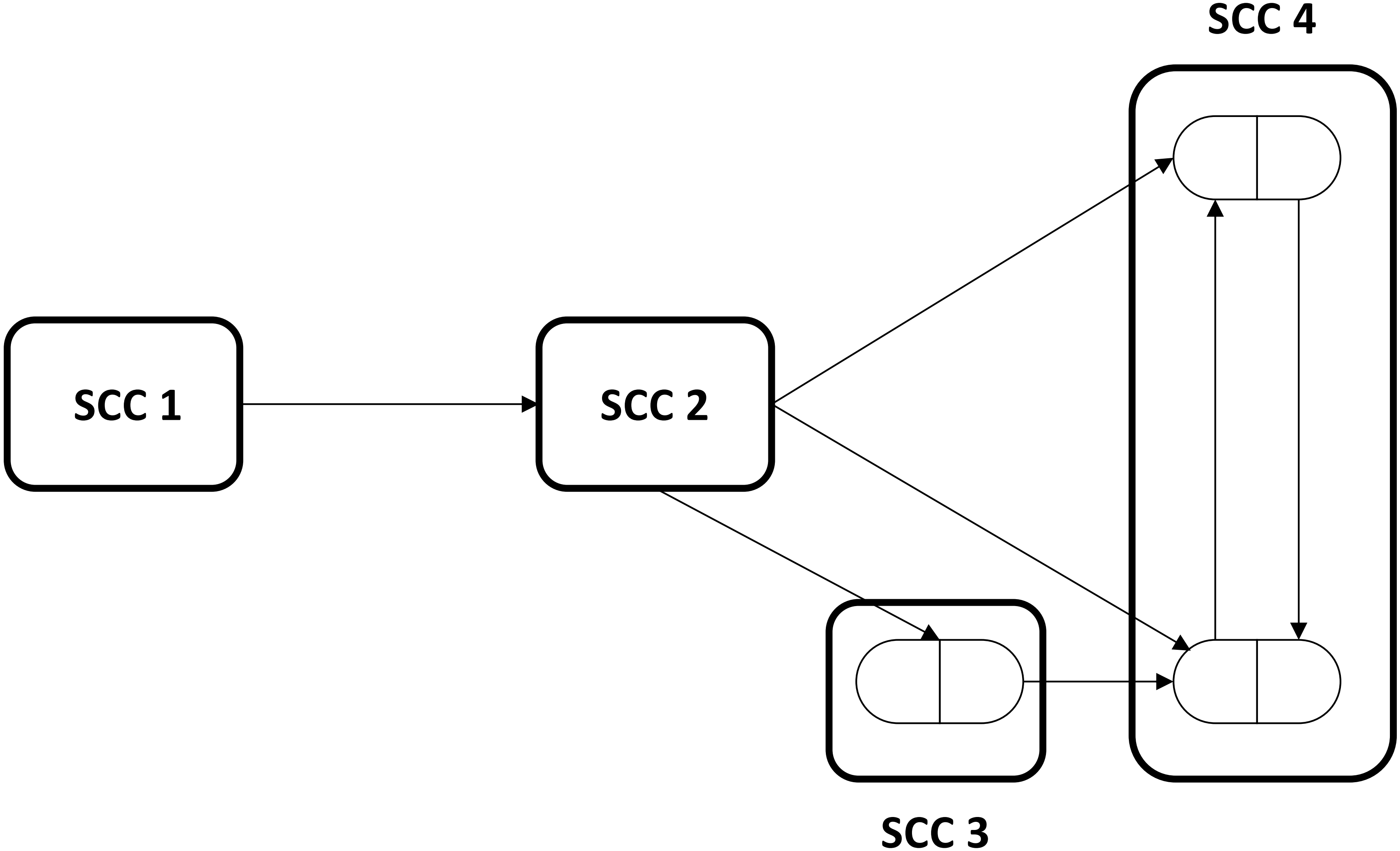
Component graph

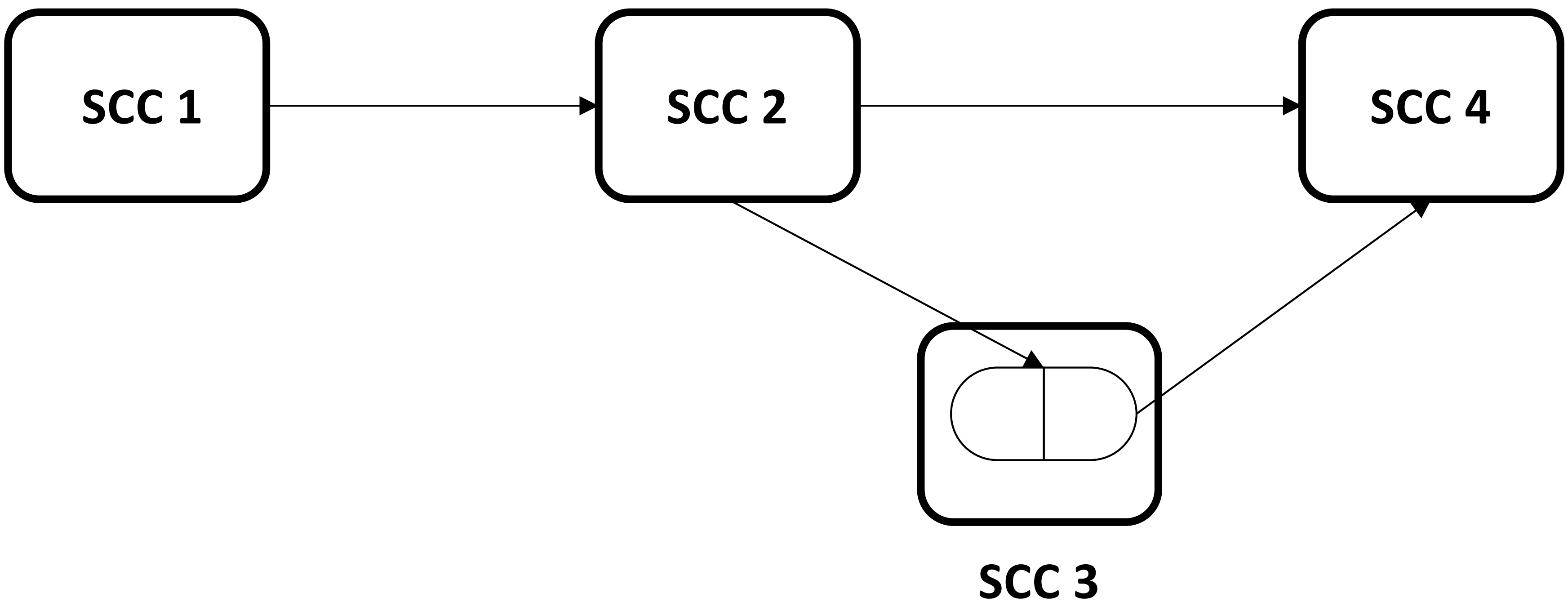
“Shrink each SCC into a single vertex, remove duplicate edges”

- $G^{scc} = (V^{scc}, E^{scc})$
- V^{scc} has one vertex for each SCC in G
- E^{scc} has an edge if there is an edge between the corresponding SCC’s in G

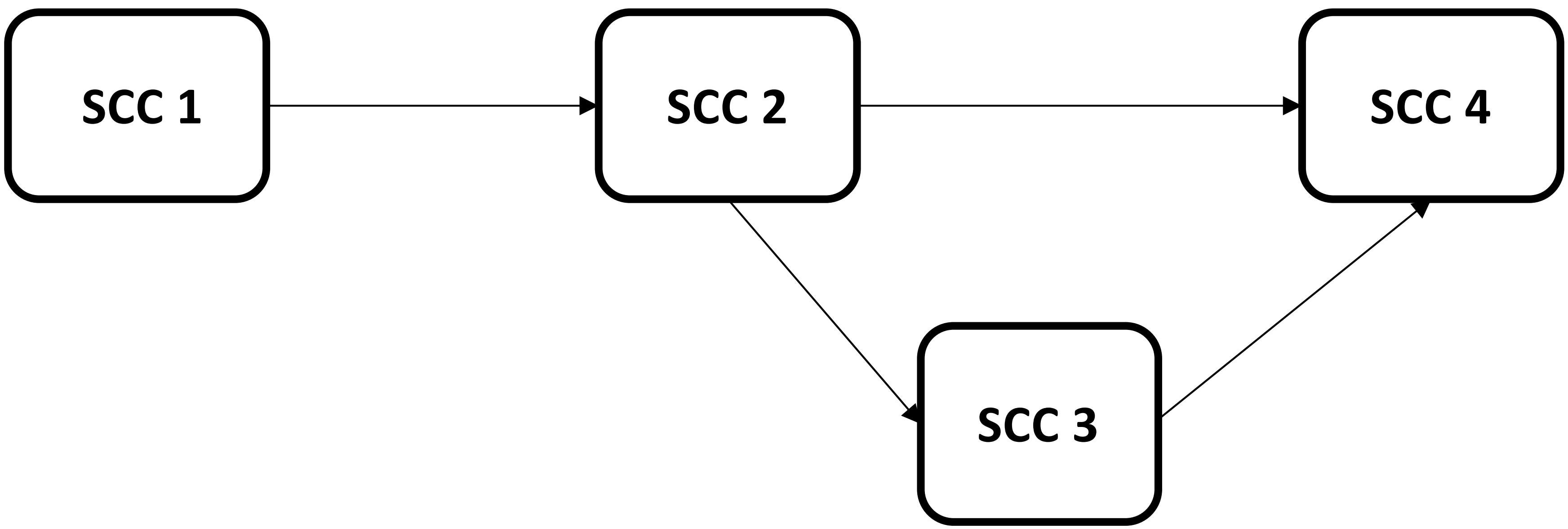

SCC 1
SCC 2
SCC 4
SCC 3


SCC 2
SCC 4
SCC 1
SCC 3


SCC 4
SCC 1
SCC 2
SCC 3


SCC 1
SCC 2
SCC 4
SCC 3

G^{scc}



Transpose of a graph

Algorithm for computing SCCs uses the notion of **transpose** of a graph

G^T is the transpose of $G = (V, E)$ defined as

- $G^T = (V, E^T)$
- $E^T = \{(v, u) : (u, v) \in E\}$
- G^T is G with all edges reversed

Can be created in $\Theta(|V| + |E|)$ running time using $Adj[\,]$

$SCC(G)$

call $DFS(G)$ to compute finishing times $u.f$ for all $u \in V$

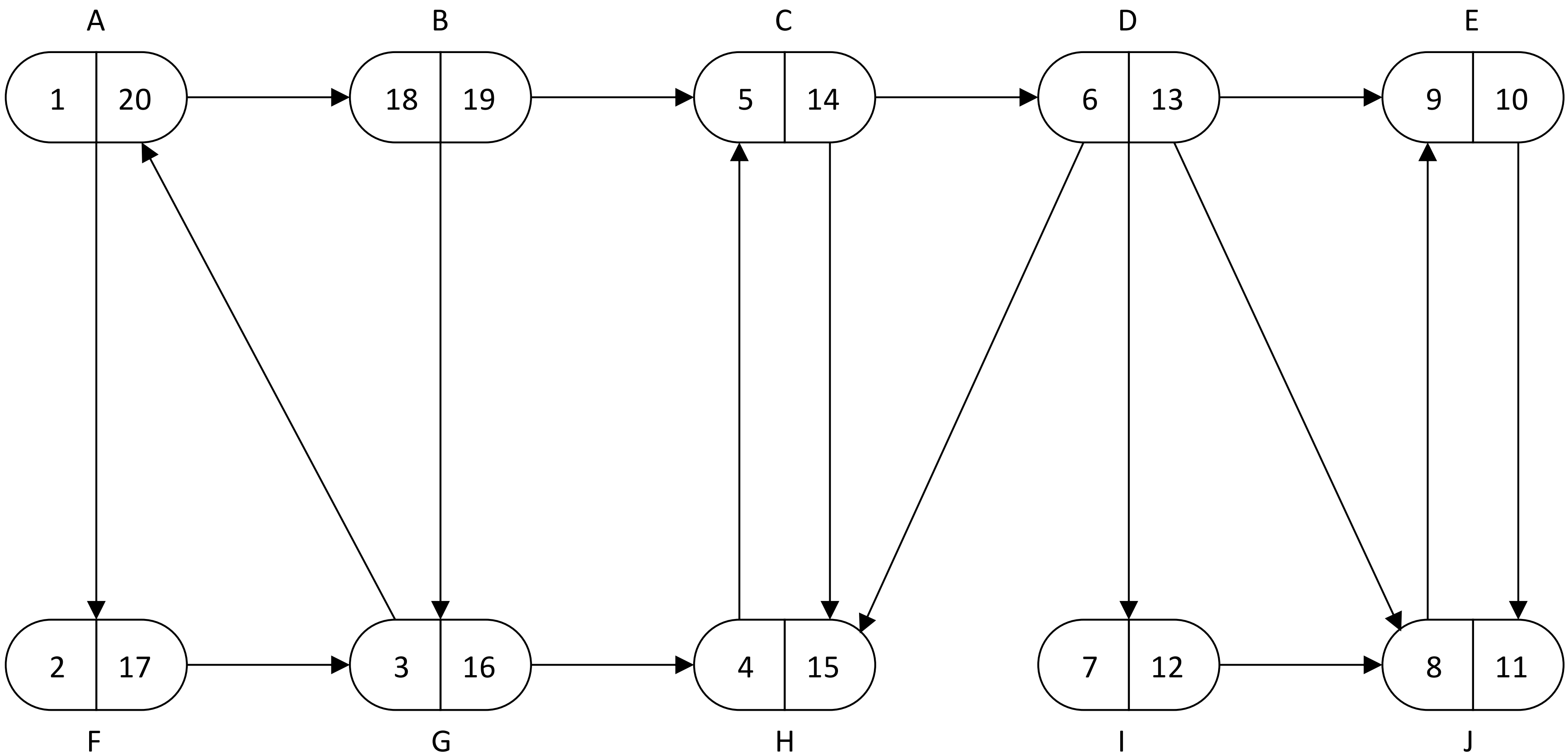
compute G^T

call $DFS(G^T)$, but in the main loop consider vertices in order of decreasing $u.f$ (from first DFS)

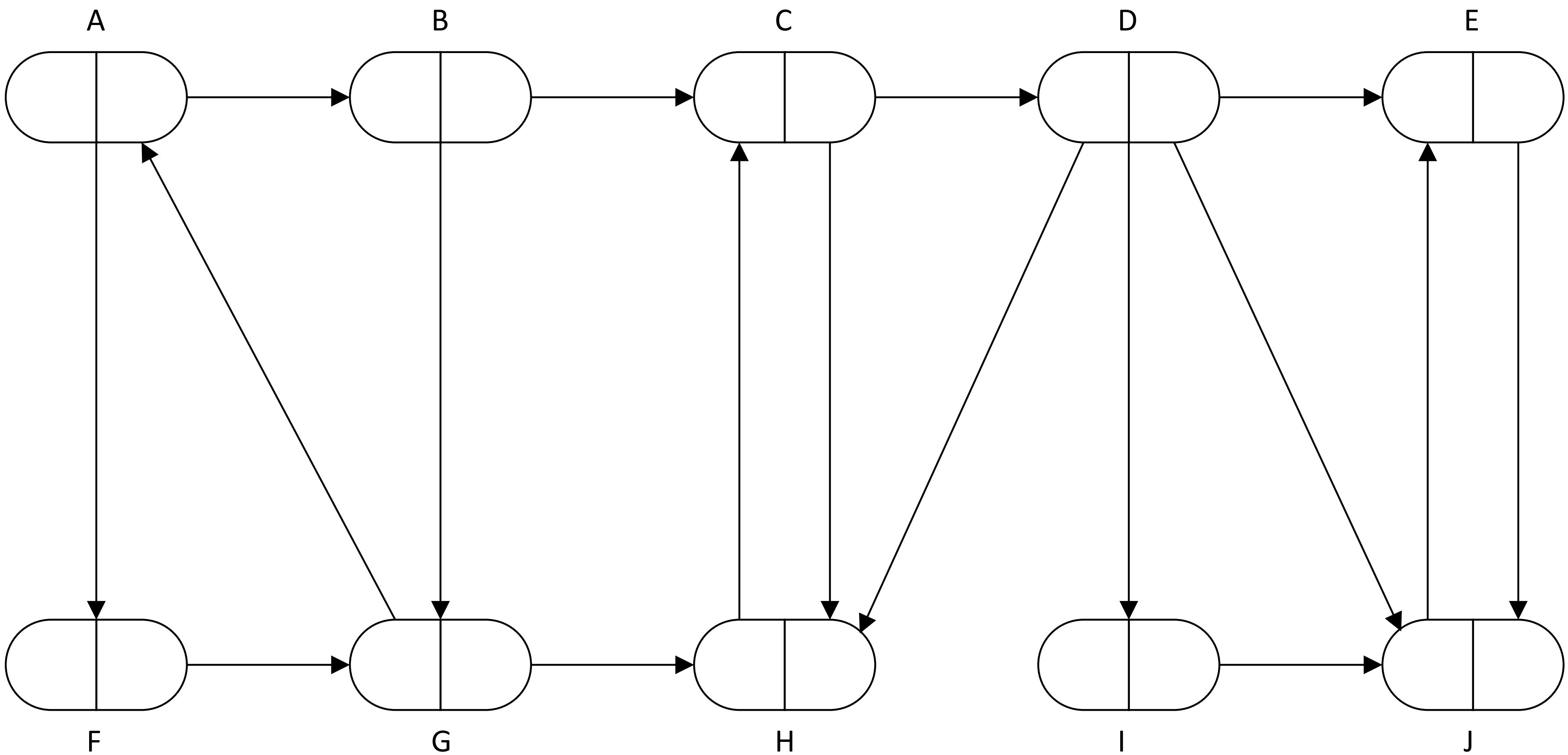
output vertices in each tree of the DFS-forest formed in second DFS as a separate SCC

Time $\Theta(|V| + |E|)$

First DFS

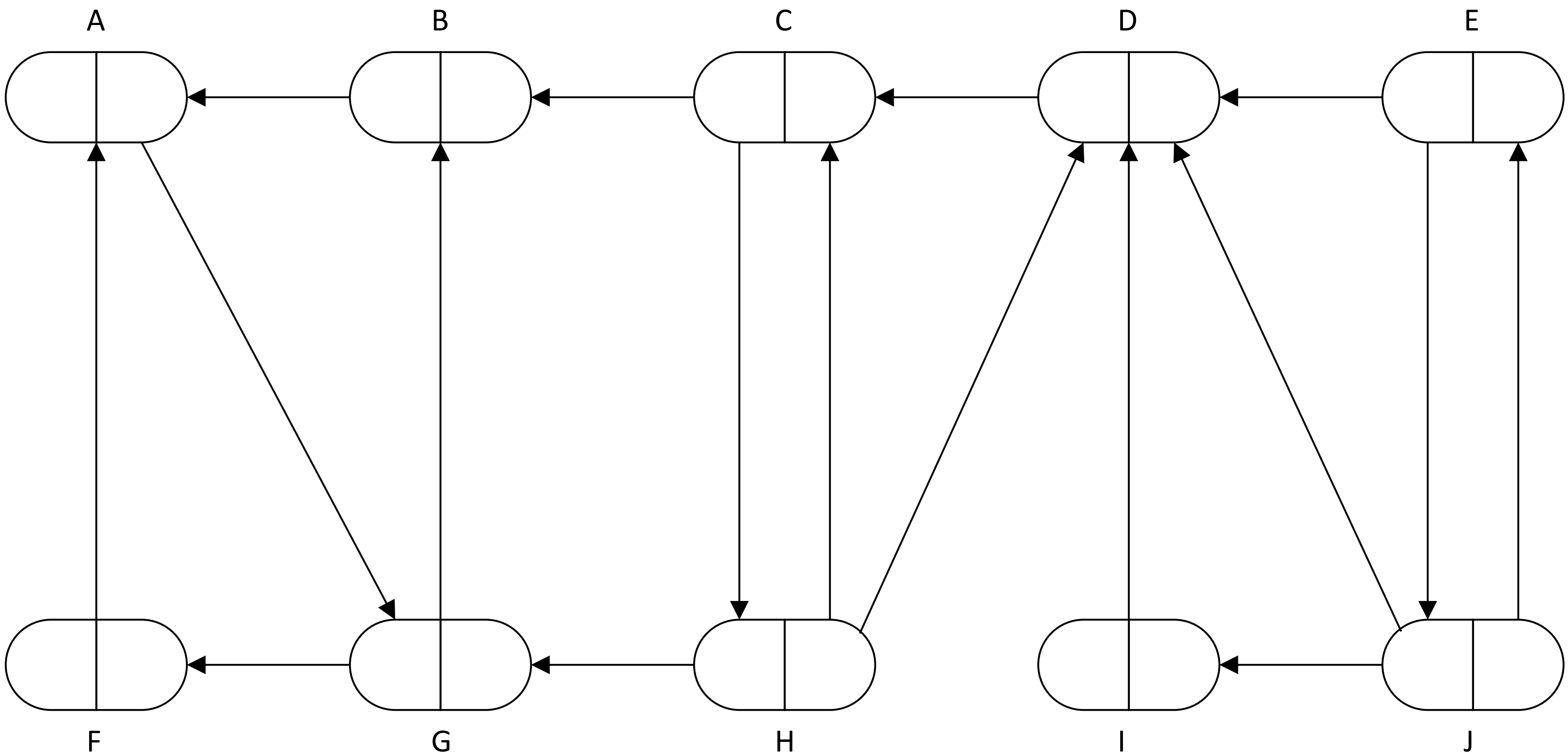


Order of vertices for 2nd DFS: A,B,F,G,H,C,D,I,J,E



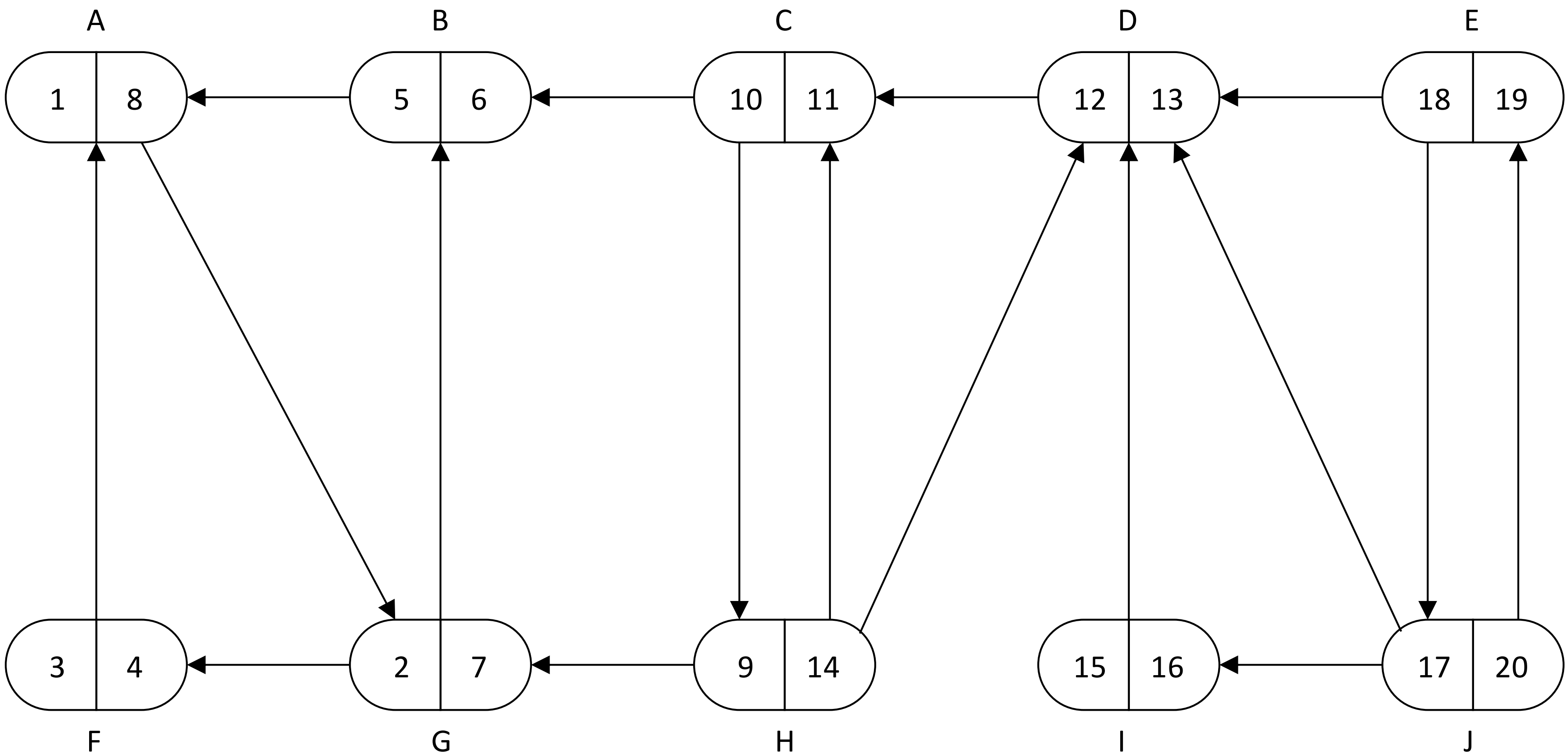
Order of vertices for 2nd DFS: A,B,F,G,H,C,D,I,J,E

Compute transpose



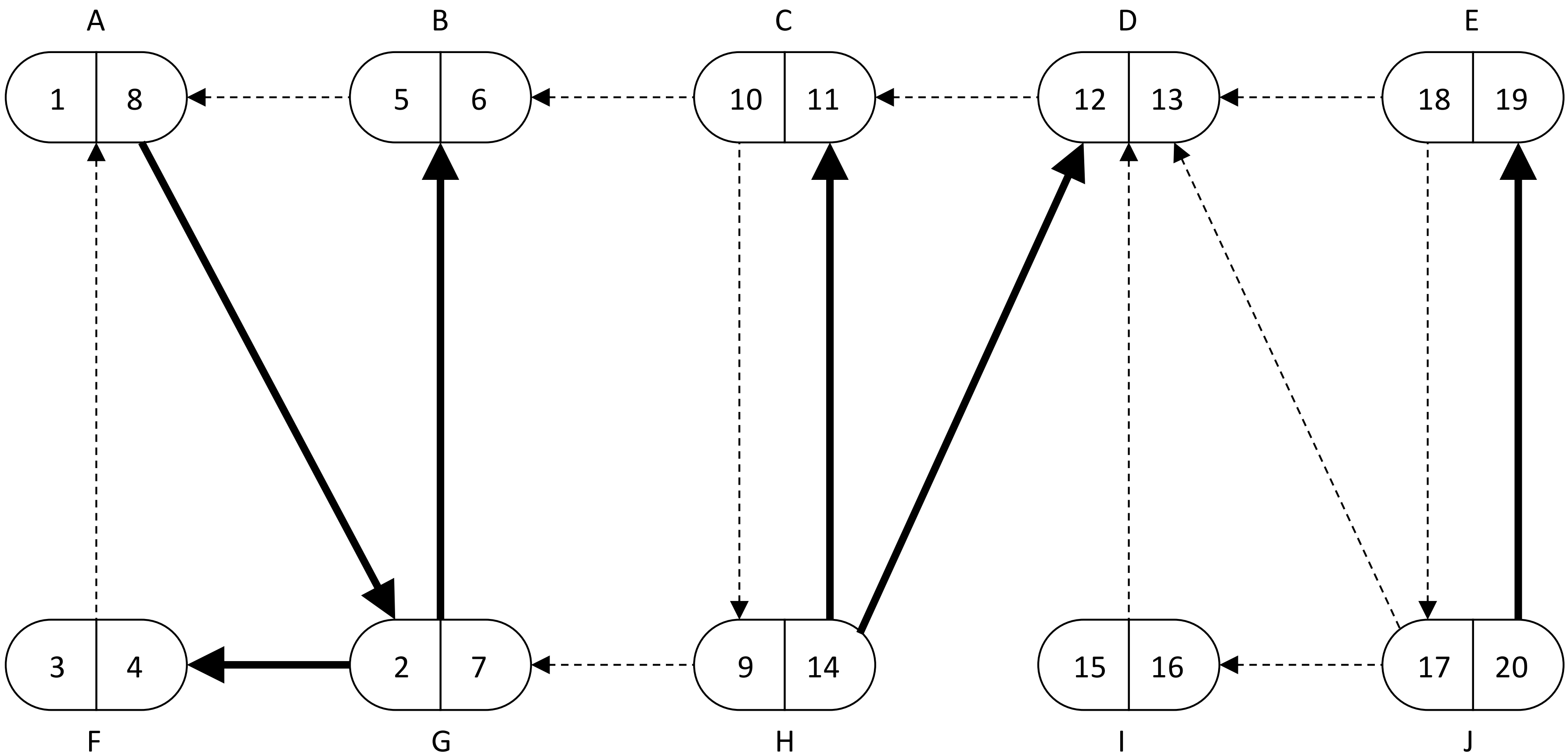
Order of vertices for 2nd DFS: A,B,F,G,H,C,D,I,J,E

2^{nd} DFS (on transpose)

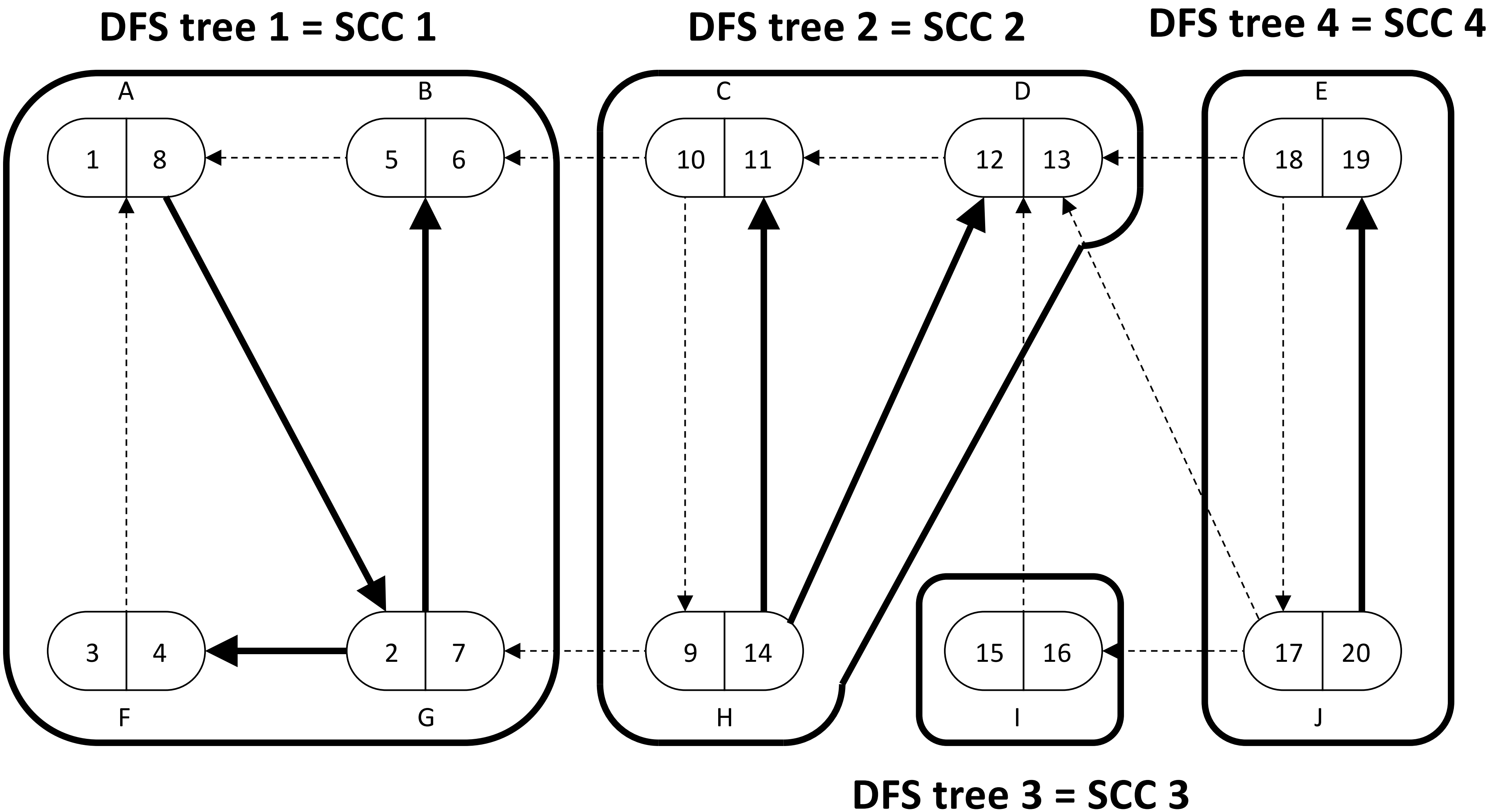


Order of vertices for 2^{nd} DFS: A,B,F,G,H,C,D,I,J,E

Tree edges



Order of vertices for 2nd DFS: A,B,F,G,H,C,D,I,J,E

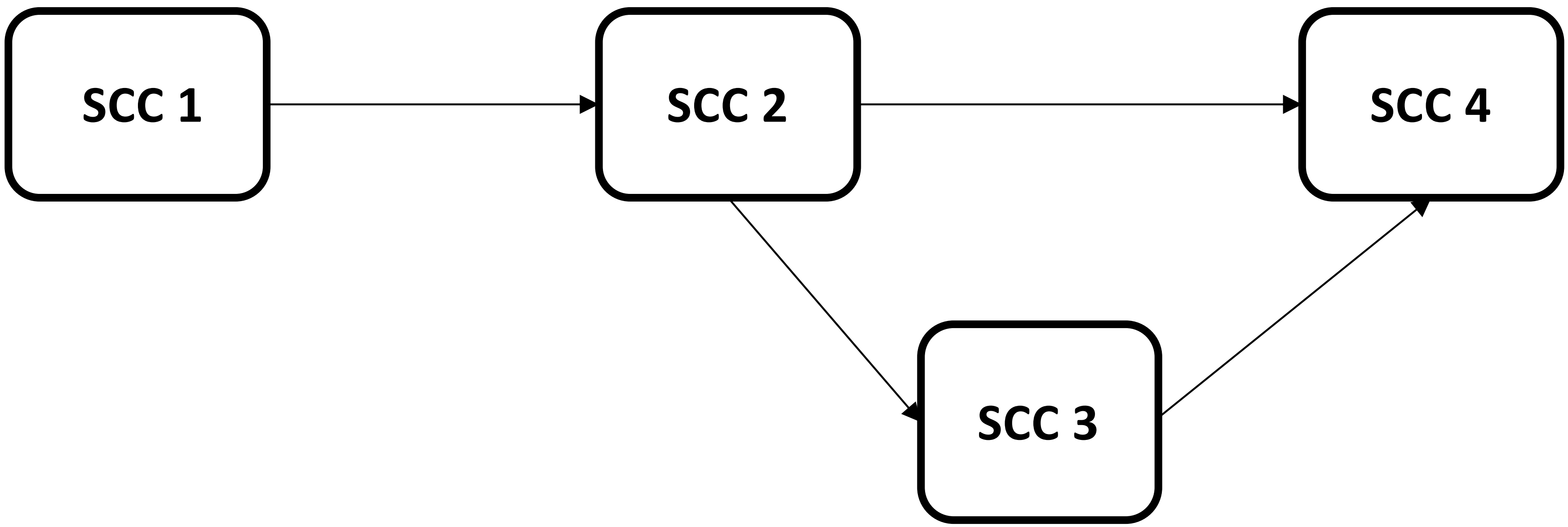


DFS tree 1 = SCC 1
DFS tree 2 = SCC 2
DFS tree 4 = SCC 4
A
B
C
D
E
1
8
5
6
10
11
12
13
18
19
3
4
2
7
9
14
15
16
17
20
F
G
H
I
J
DFS tree 3 = SCC 3

Why does this work?

- G and G^T have the same SCCs
- Component graph G^{scc} is a dag
- Considering vertices in second DFS in decreasing order of finishing times from first DFS, we are visiting vertices of component graph in topological sort order
- But in the second DFS the edges have been reversed!
- Therefore the second DFS explores vertices in a single component first, then it has to start a new DFS tree to process the next component, and so on
- See CLRS for a formal proof

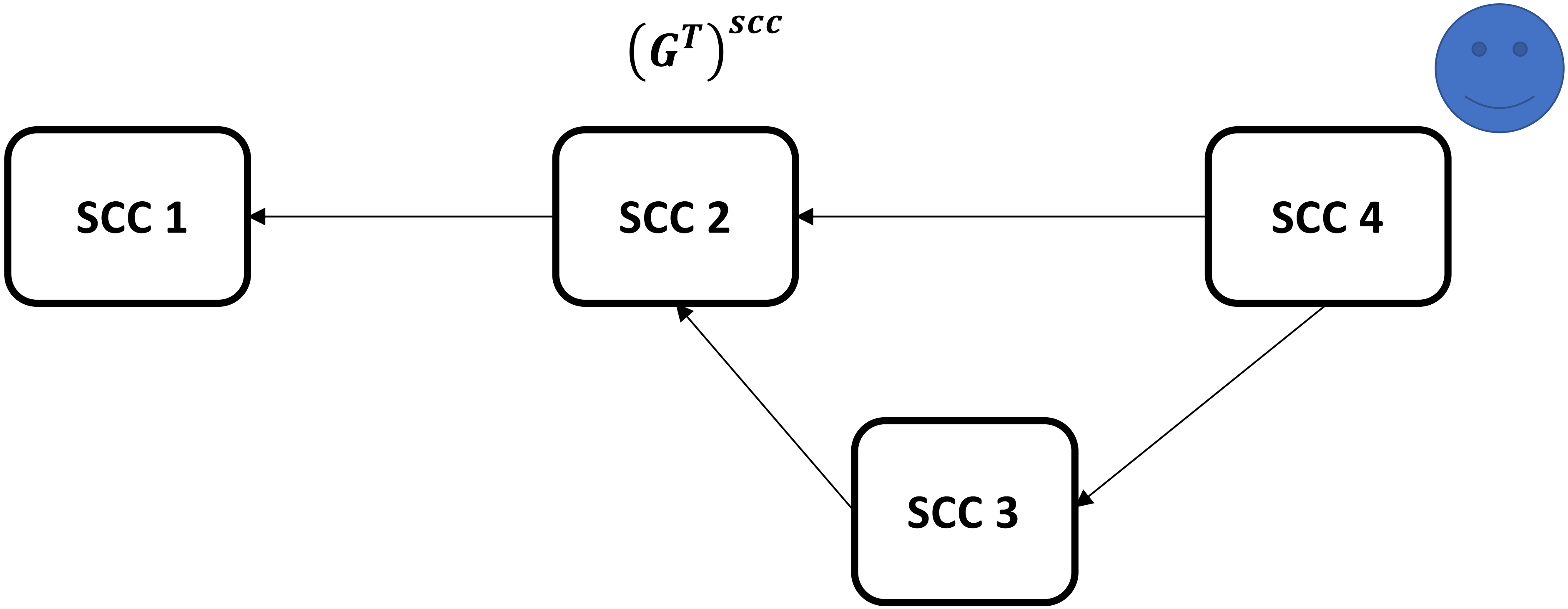
G^{scc}



According to the order of vertices in after the first DFS:

- Vertices in SCC 1 appear first in that ordering
- Vertices in SCC 2 appear after that
- Vertices in SCC 3 appear after that
- Vertices in SCC 4 appear after that

$$(G^T)^{scc}$$



Second DFS starts by exploring SCC 1 vertices
Note that it cannot reach SCC 2 vertices from SCC 1
Therefore, DFS is forced to start a new tree
This is repeated for SCC 3 vertices and SCC 4 vertices

Minimum spanning trees MSTs (CLRS 23)

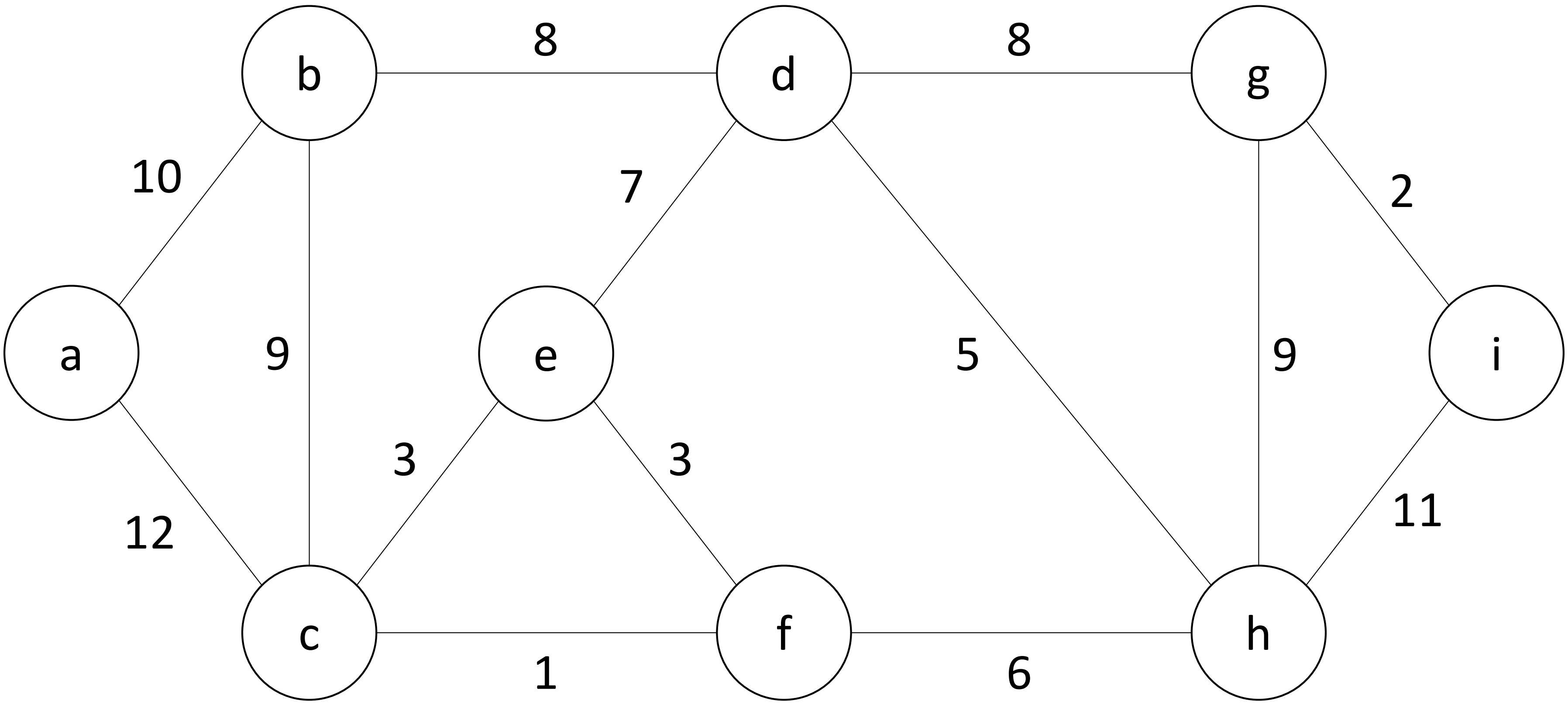
Input: $G = (V, E)$ undirected graph

$w : E \to \mathbb{R}$ - edge weights

Output: Find $T \subseteq E$ such that

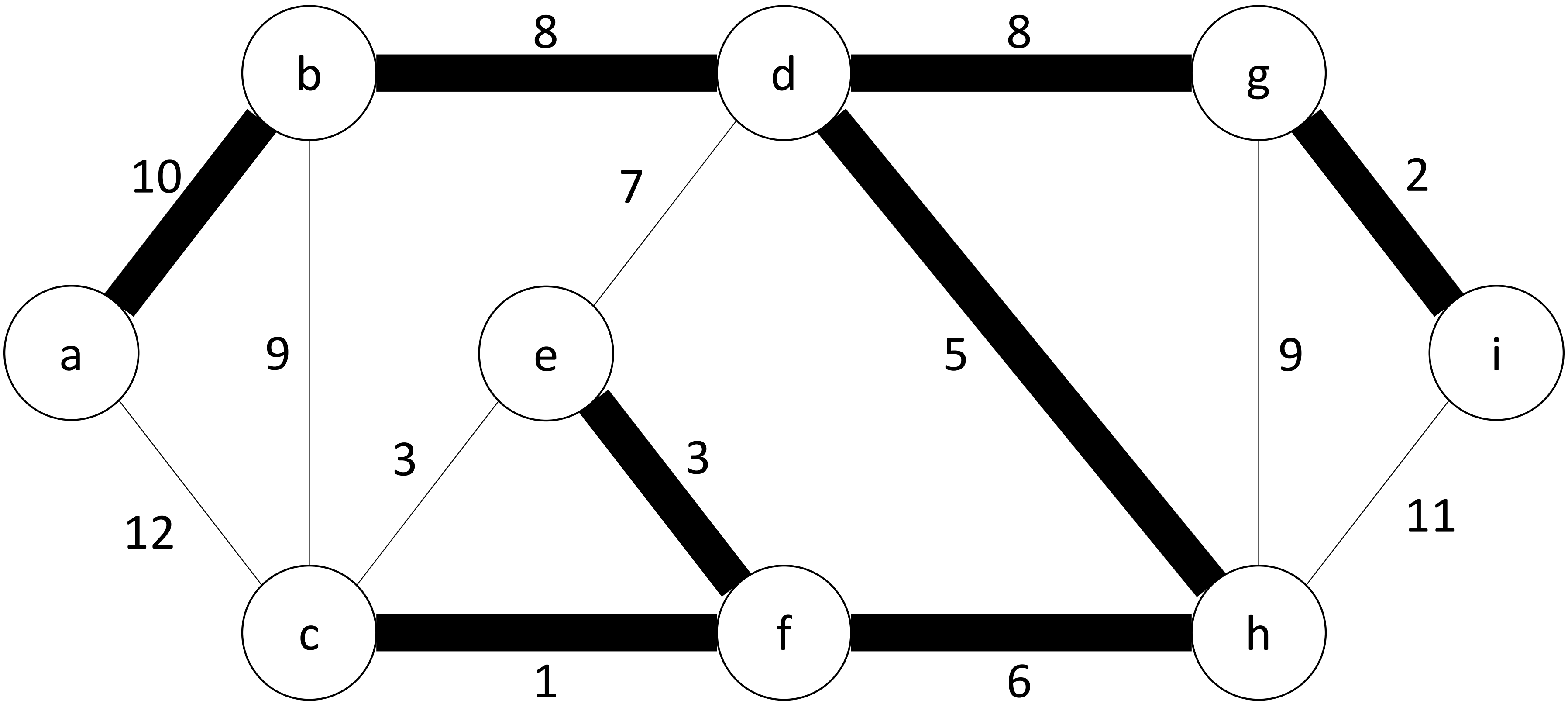
- T connects all vertices (T is a spanning tree), and
- $w(T) = \sum_{\{u,v\} \in T} w(\{u, v\})$ is minimized

A spanning tree whose weight is minimum over all spanning trees is called a **minimum spanning tree** (**MST**)

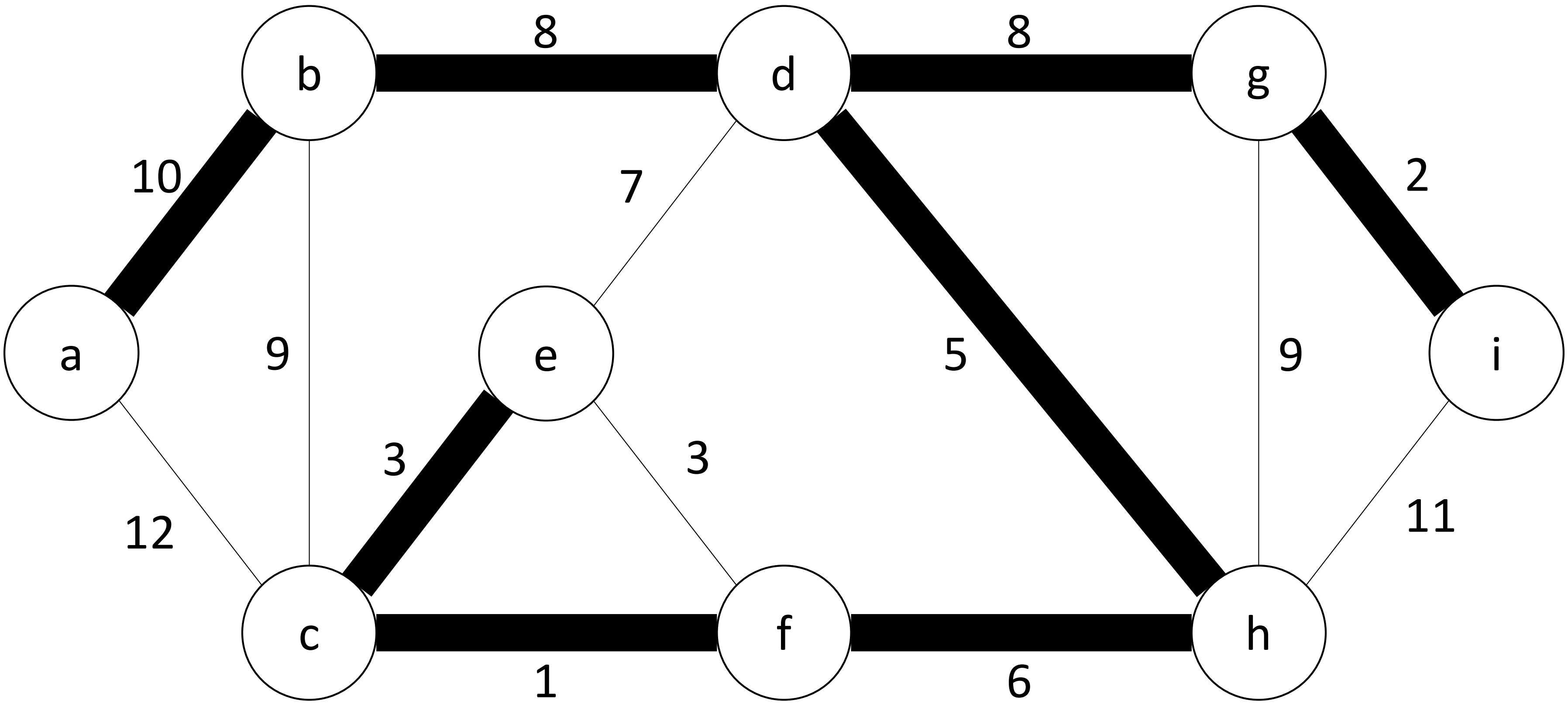


b
d
g
a
e
i
c
f
h
8
8
10
7
2
9
5
9
3
3
12
11
1
6

MST



Another MST



- MST has $|V| - 1$ edges
- MST is a tree – connected acyclic graph
- MST might not be unique

Building a solution:

- Build a set A of edges
- Initially, A is empty
- Add edges to A to maintain the **invariant**:

A is a subset of some MST

- Add only **safe** edges to maintain the invariant:

$\{u, v\}$ is safe if $A \cup \{\{u, v\}\}$ is a subset of some MST

Generic MST algorithm

$Generic - MST(G = (V, E), w)$

$A \leftarrow \emptyset$

$\boldsymbol{while}$ A is not a spanning tree

find an edge $\{u, v\}$ that is safe for A

$A \leftarrow A \cup \{\{u, v\}\}$

$\boldsymbol{return}$ A

How to find safe edges?

Let $S \subseteq V$ and $A \subseteq E$

A **cut** $(S, V - S)$ is a partition of vertices into disjoint sets S and $V - S$

Edge $\{u, v\} \in E$ **crosses** cut $(S, V - S)$ if one endpoint is in S and another endpoint is in $V - S$

A cut **respects** A if no edge in A crosses the cut

An edge is a **light edge** crossing a cut if and only if its weight is minimum over all edges crossing the cut

Main theorem

Suppose

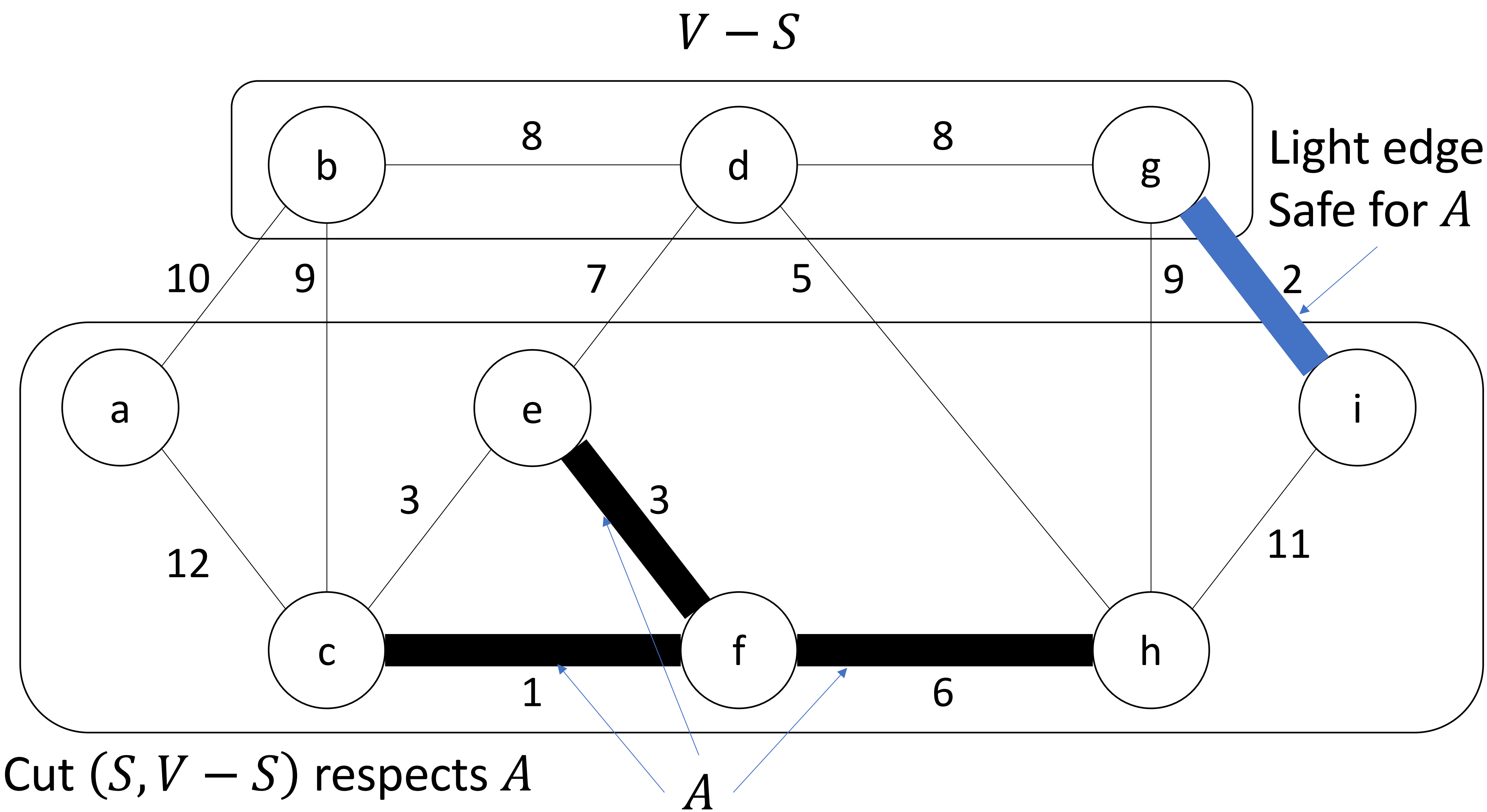
A is a subset of some MST

$(S, V - S)$ is a cut that respects A

$\{u, v\}$ is a light edge crossing $(S, V - S)$

Then

$$\{u, v\} \text{ is safe for } A$$


$V - S$
b
d
g
8
8
Light edge
Safe for A
10
9
7
5
9
2
a
e
i
3
3
12
11
c
f
h
1
6
Cut $(S, V - S)$ respects A
A

In $Generic - MST$

- A is a forest containing connected components. Initially each component is a single vertex.
- Any safe edge merges two of these components into one. Each component is a tree.
- Since an MST has exactly $|V| - 1$ edges, the $\boldsymbol{for}$ loop iterates $|V| - 1$ times.

Kruskal's algorithm

- Start with each vertex being in its own component
- Repeatedly merge two components by choosing a light edge between them
- Scan the set of edges in monotonically non-decreasing order by weight
- Use disjoint-set data structure to determine whether an edge connects vertices in different components (see CLRS 21)

$Kruskal(G = (V, E), w)$

$A \leftarrow \emptyset$

$\boldsymbol{for}\ v \in V$

$MakeSet(v)$

sort E in non-decreasing order by weight

$\boldsymbol{for}\ \{u, v\}$ taken from the sorted list

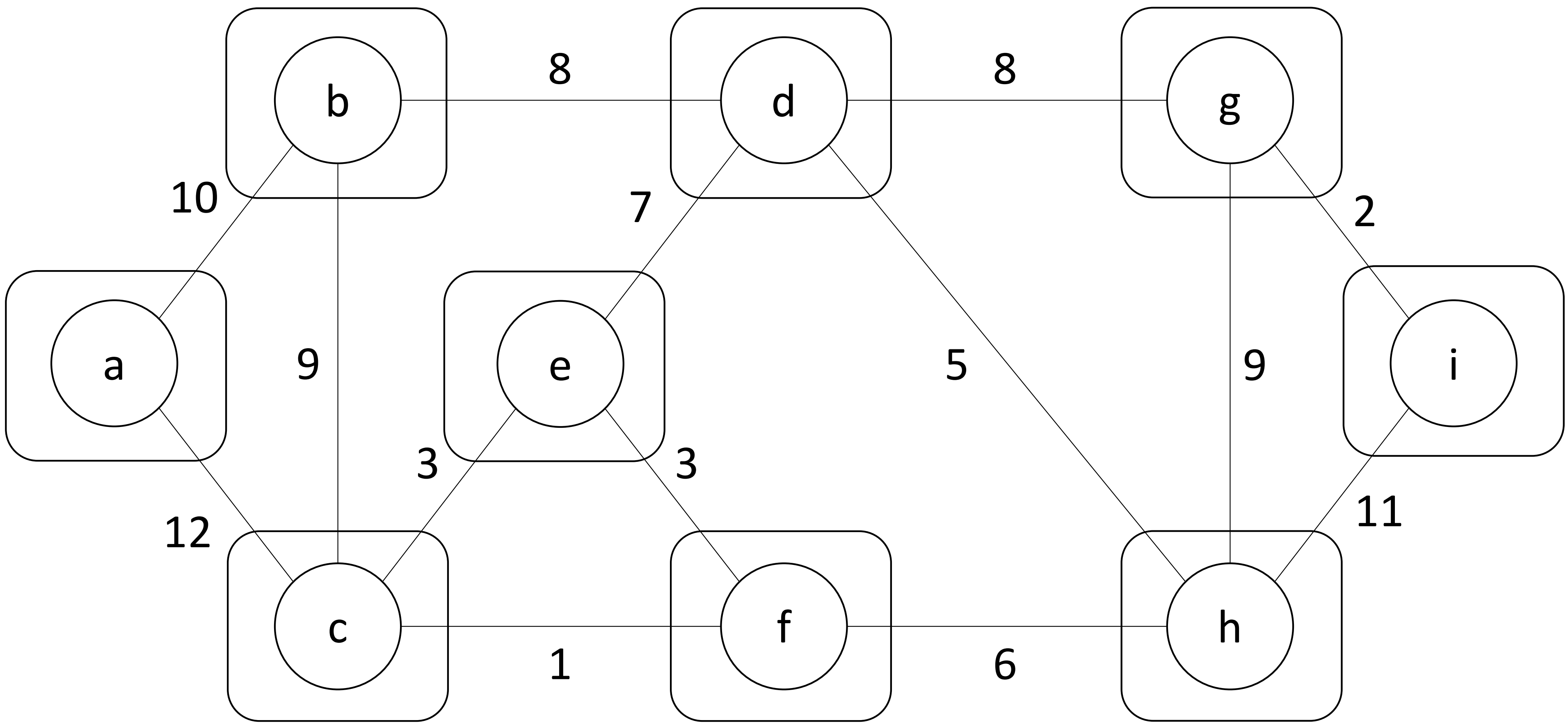
$\boldsymbol{if}\ FindSet(u) \neq FindSet(v)$

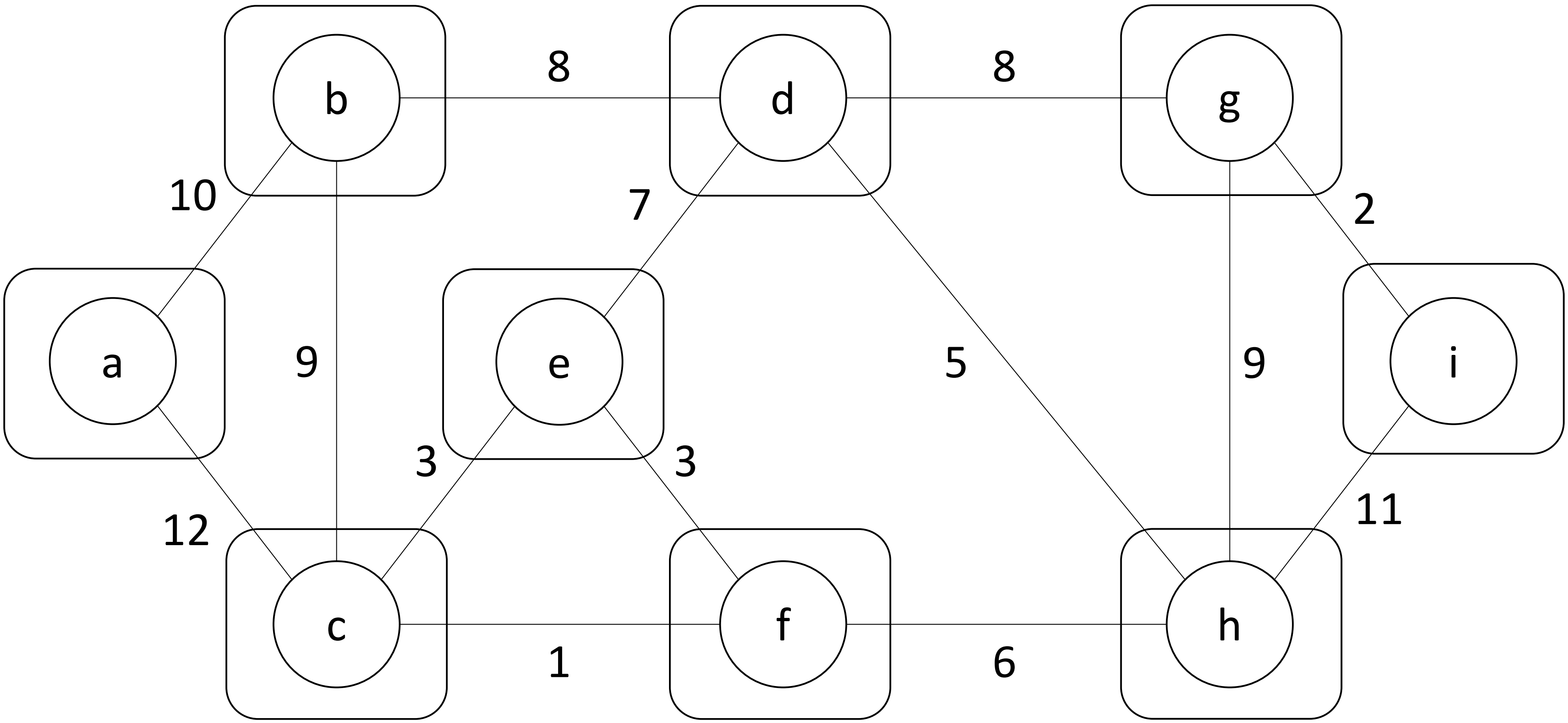
$A \leftarrow A \cup \big\{\{u, v\}\big\}$

$Union(u, v)$

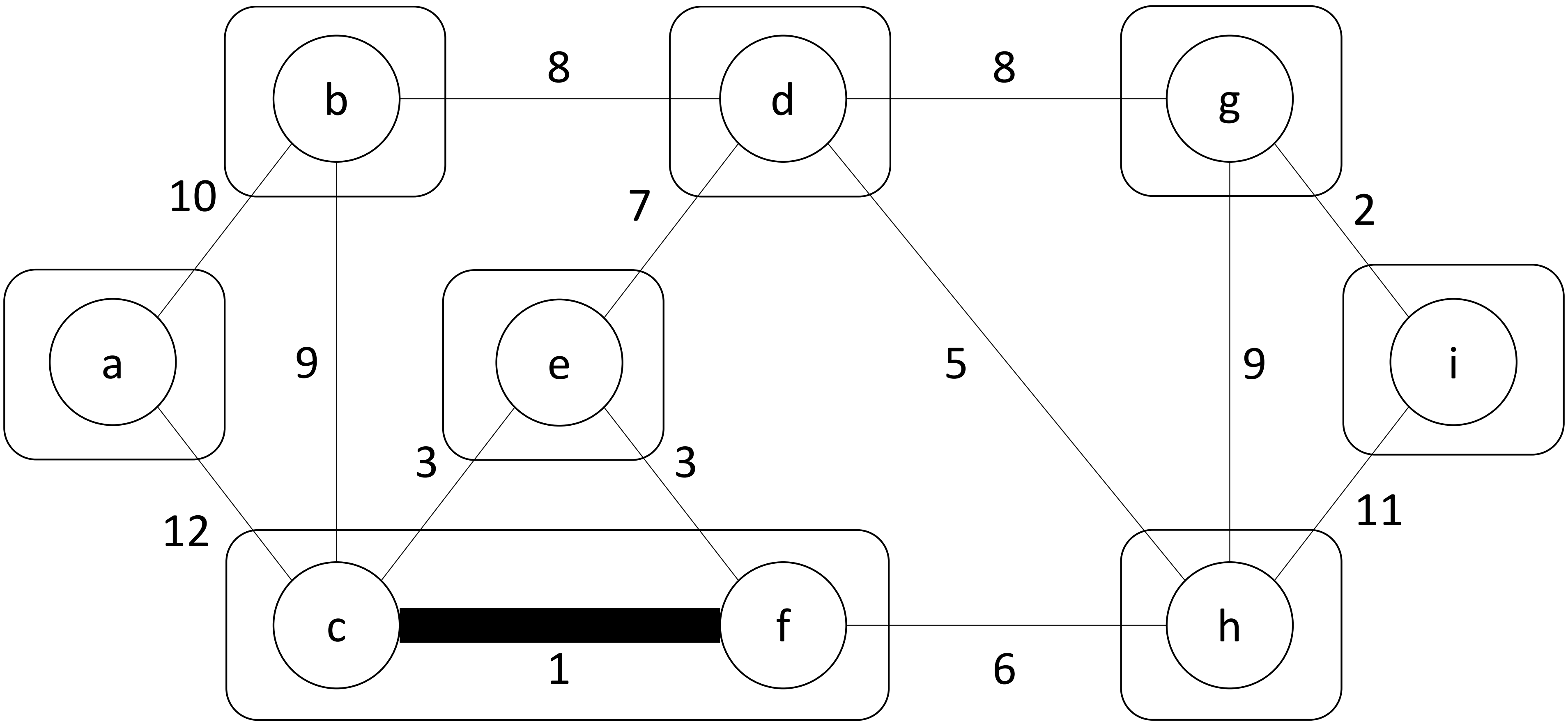
$\boldsymbol{return}\ A$

$MakeSet(v)$ for each $v \in V$



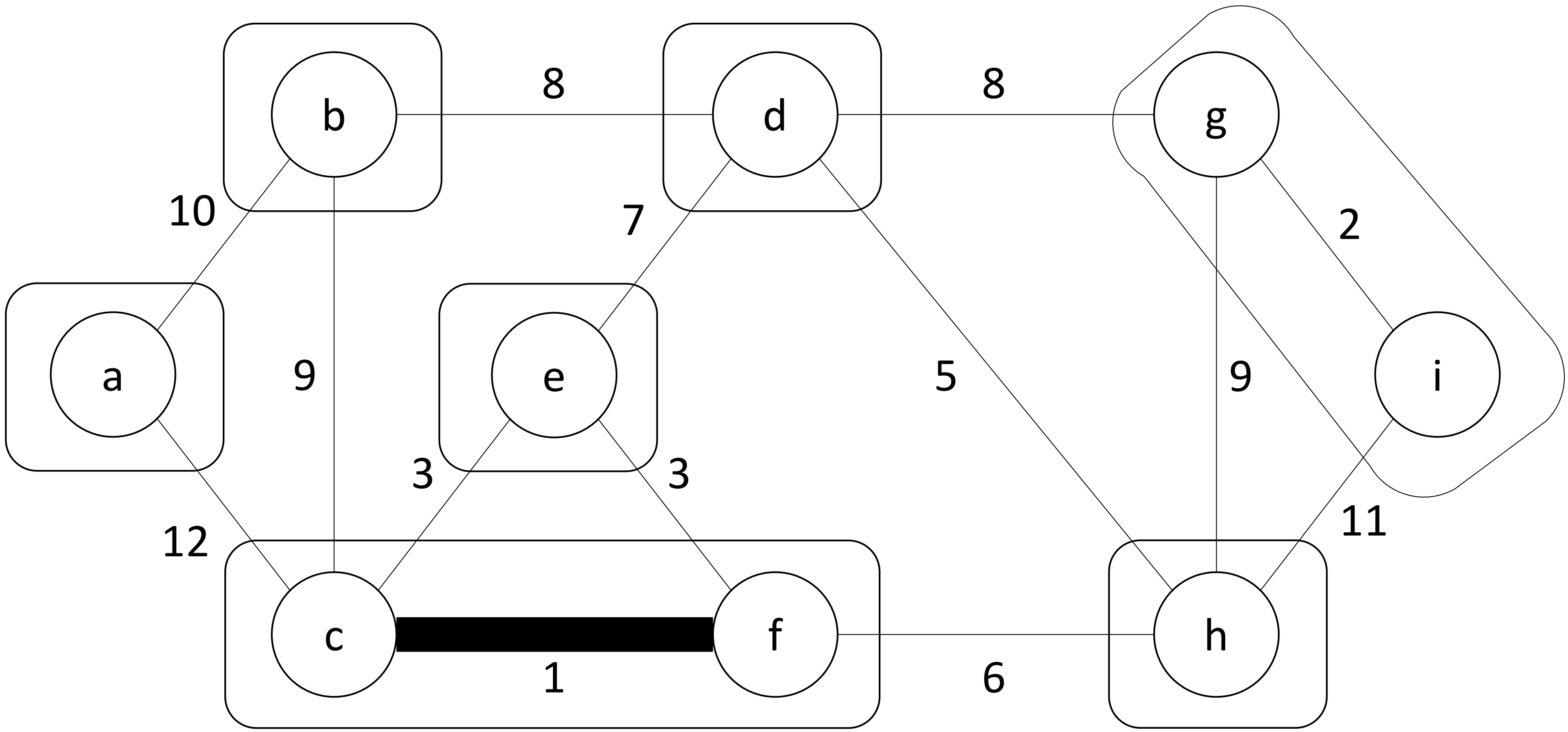


$$FindSet(c) \neq FindSet(f)$$

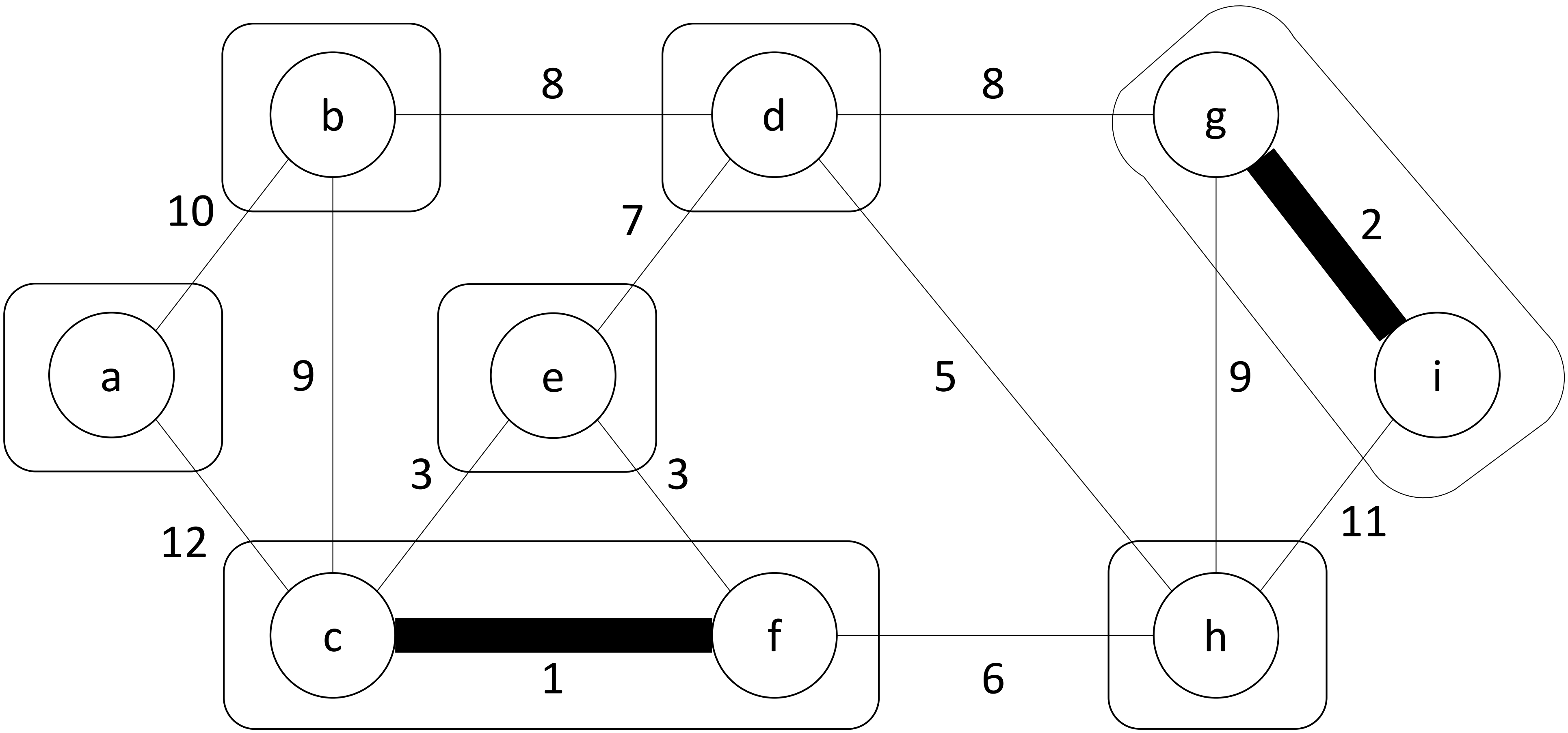


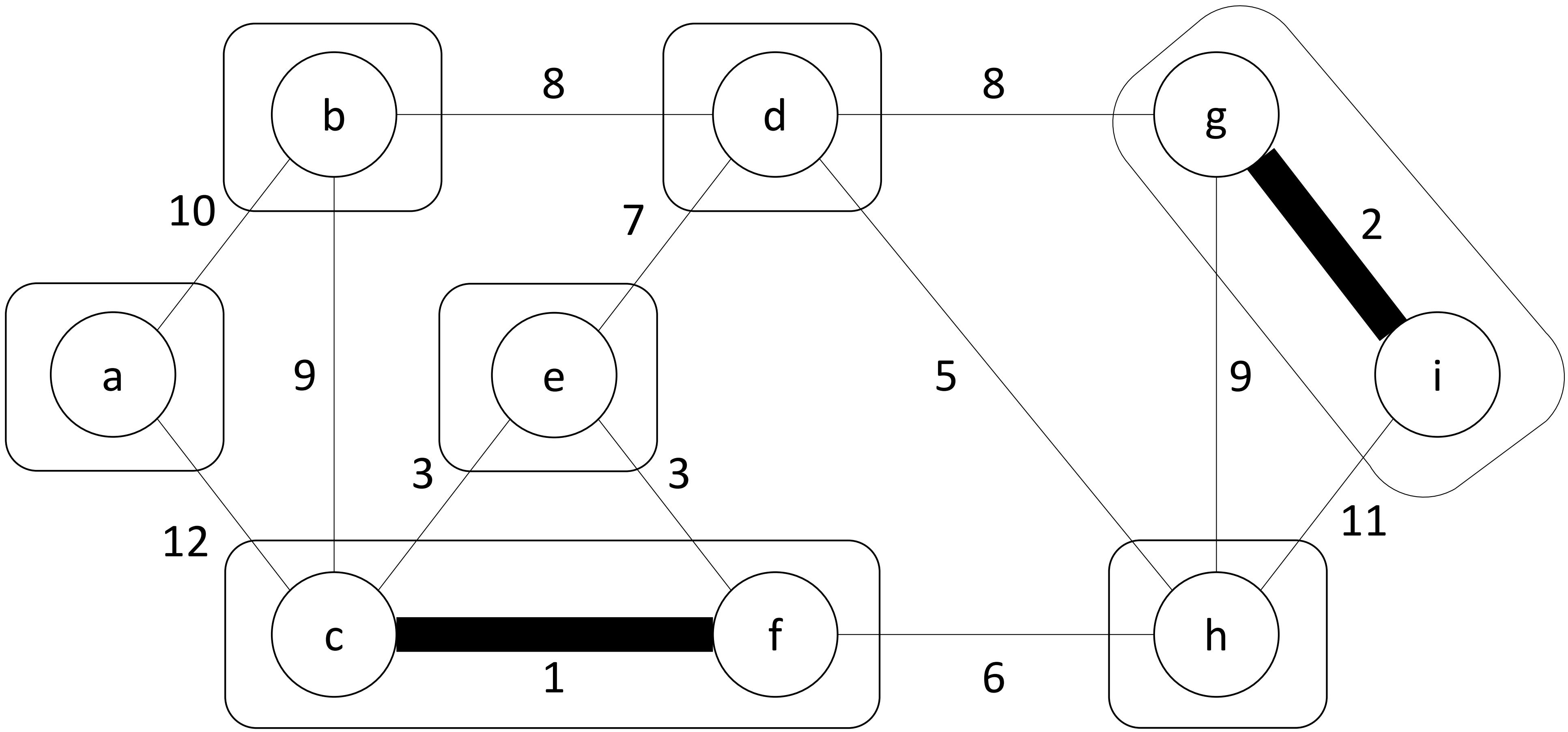
$Union(c, f)$

$$FindSet(g) \neq FindSet(i)$$

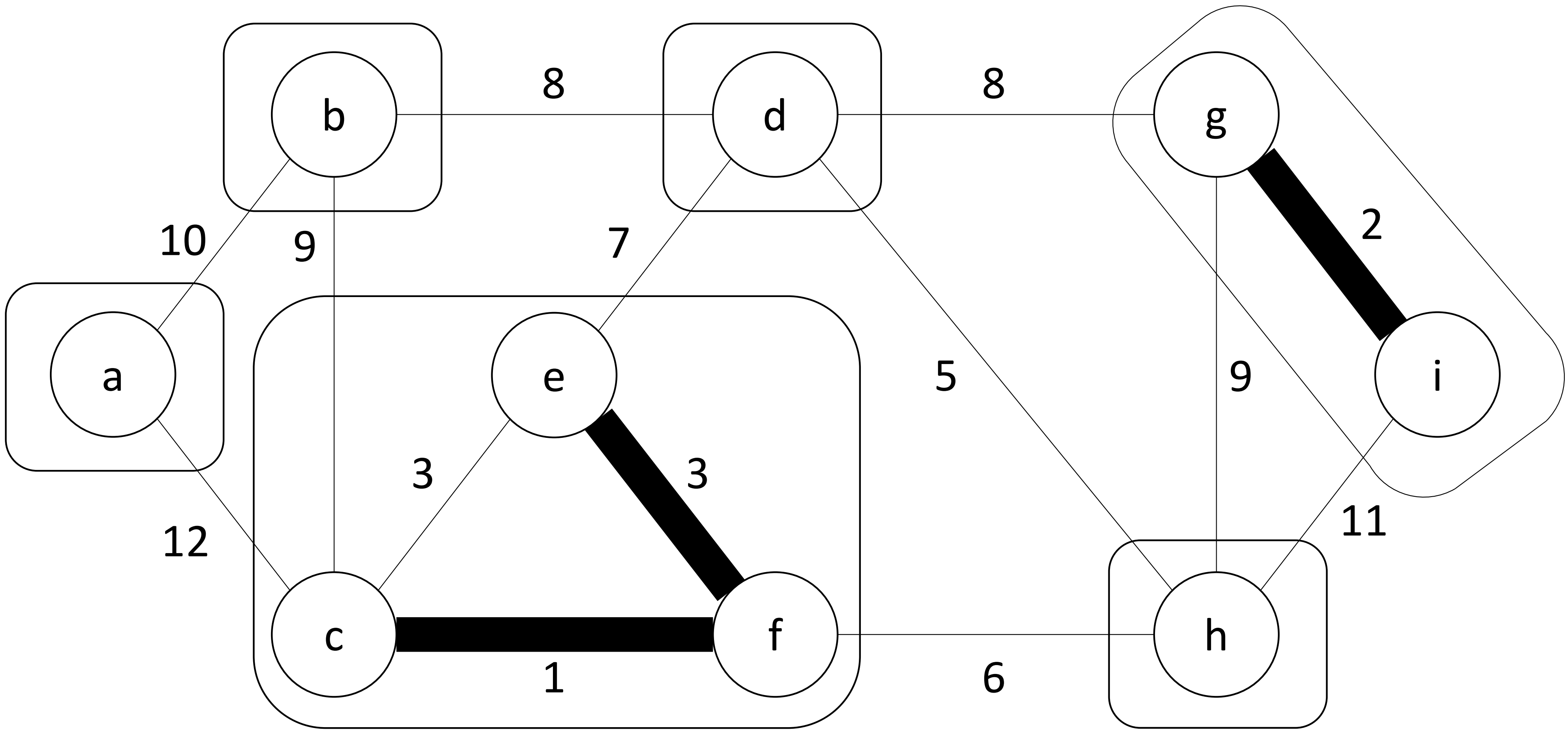


$Union(g, i)$

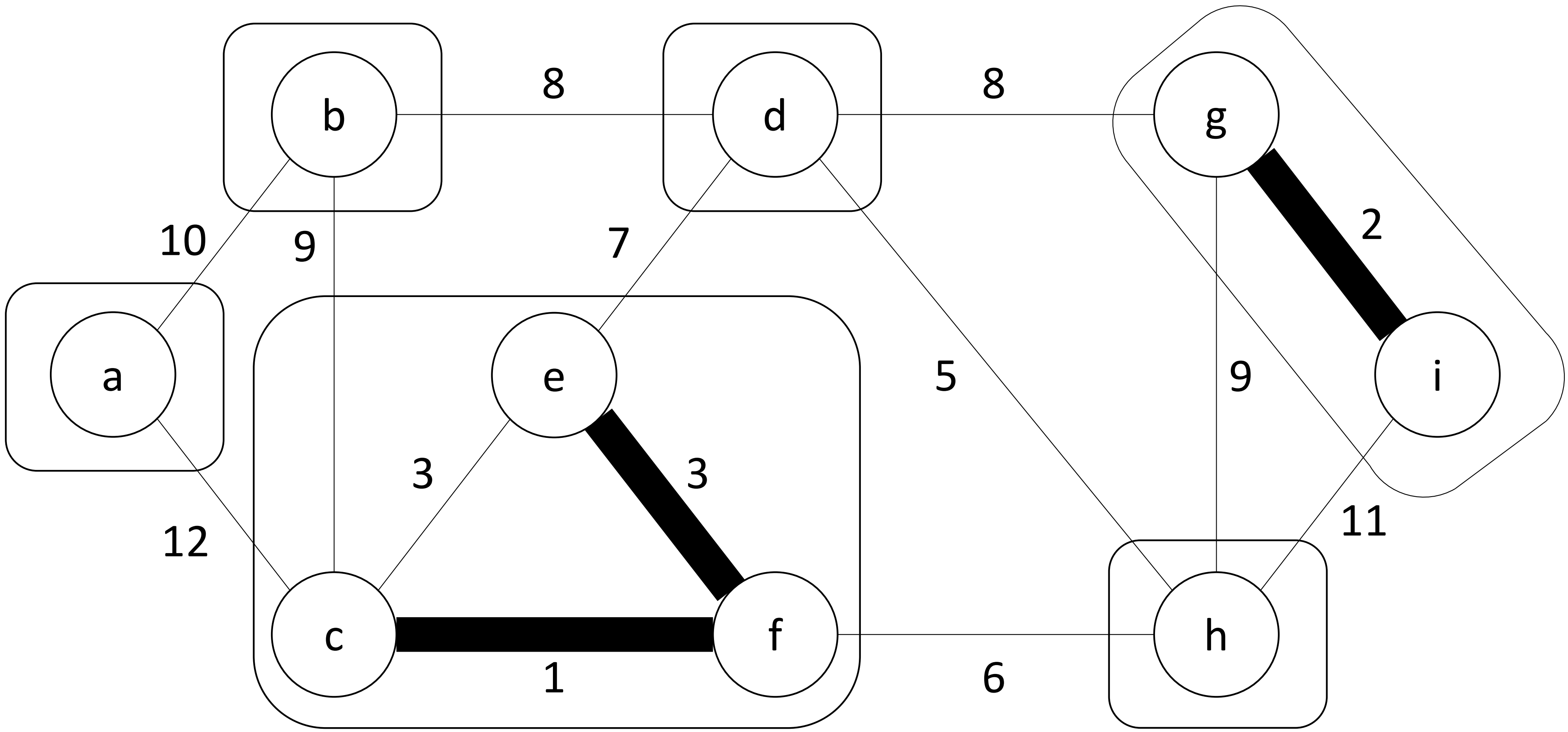




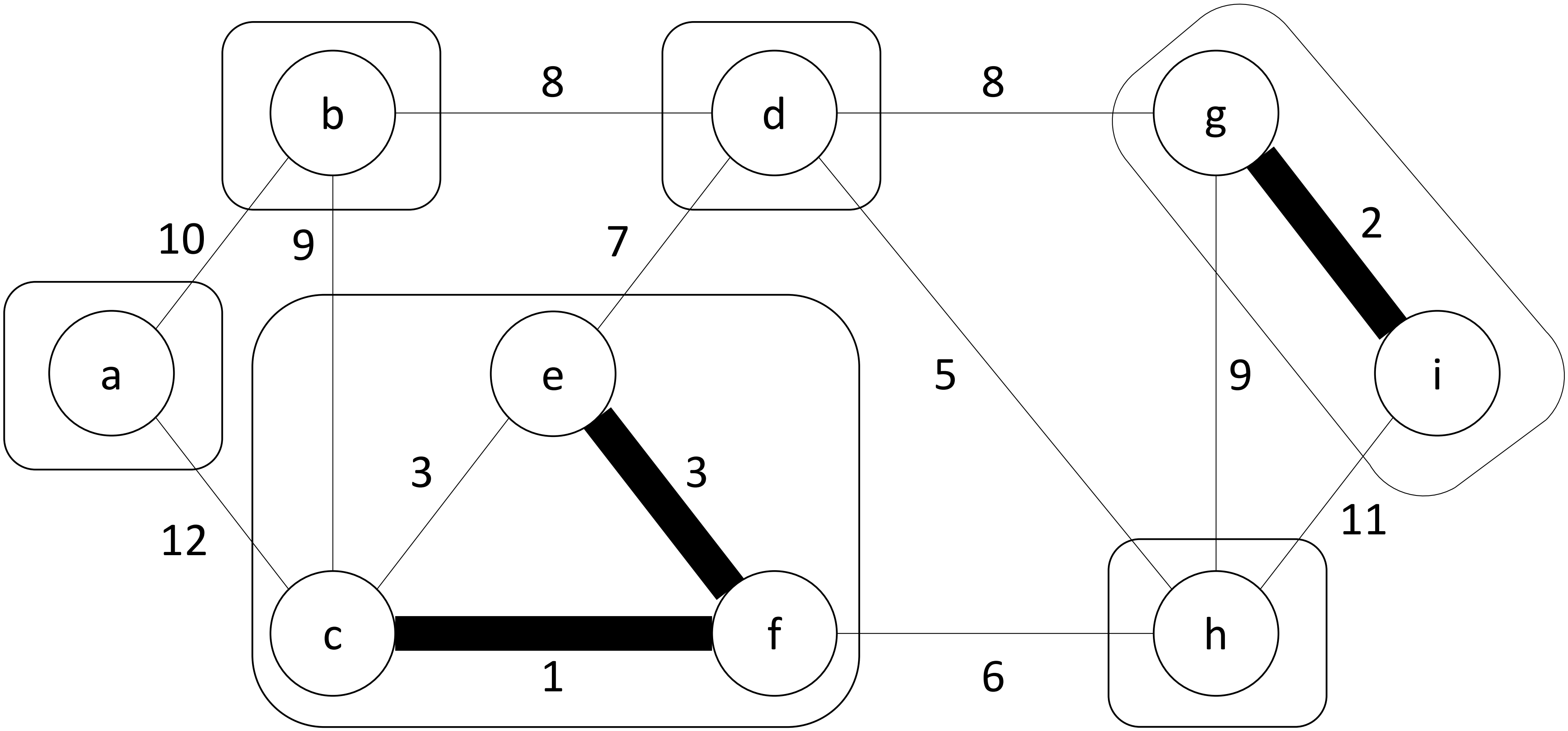
$$FindSet(e) \neq FindSet(f)$$



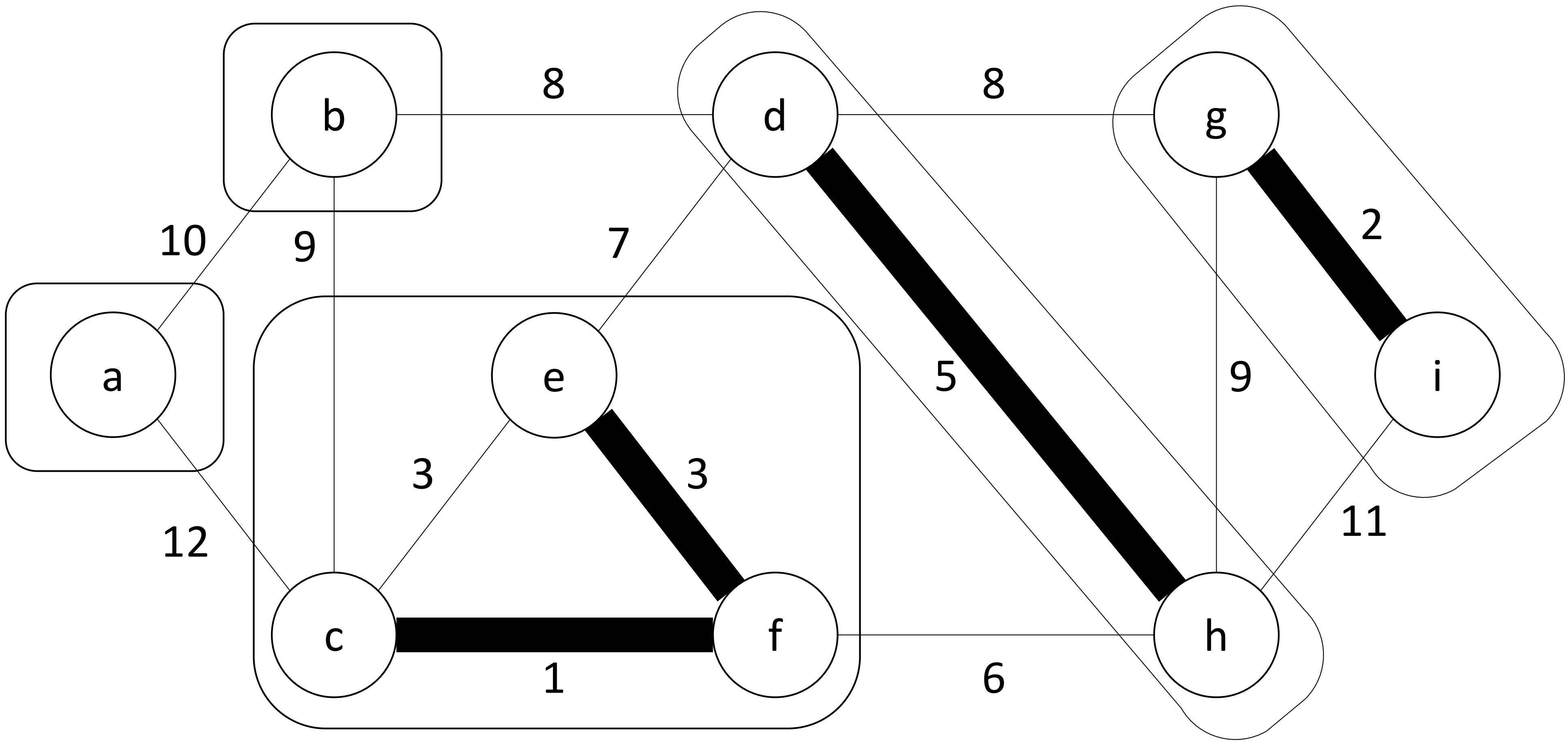
$Union(f, e)$



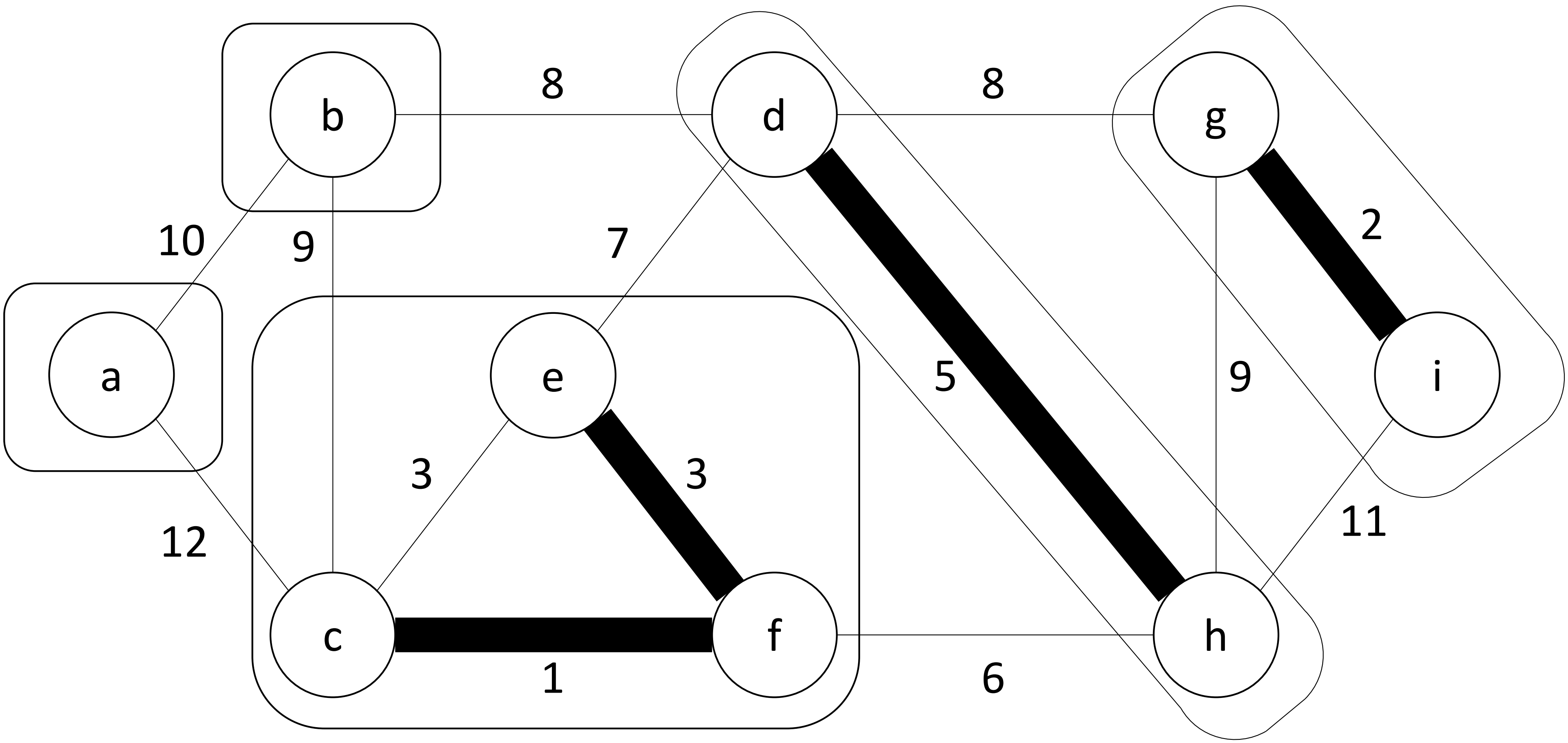
$$FindSet(e) = FindSet(c)$$



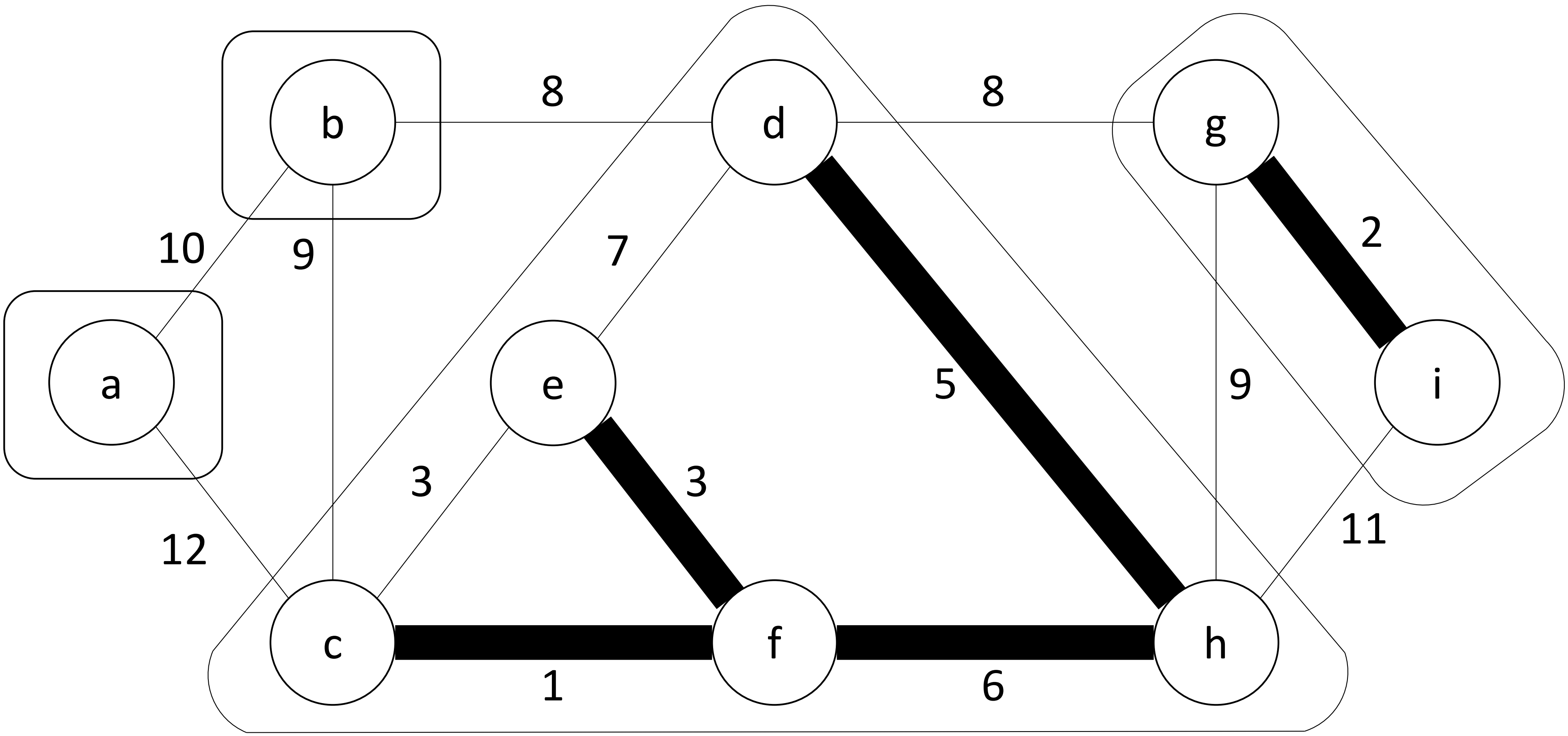
$$FindSet(d) \neq FindSet(h)$$



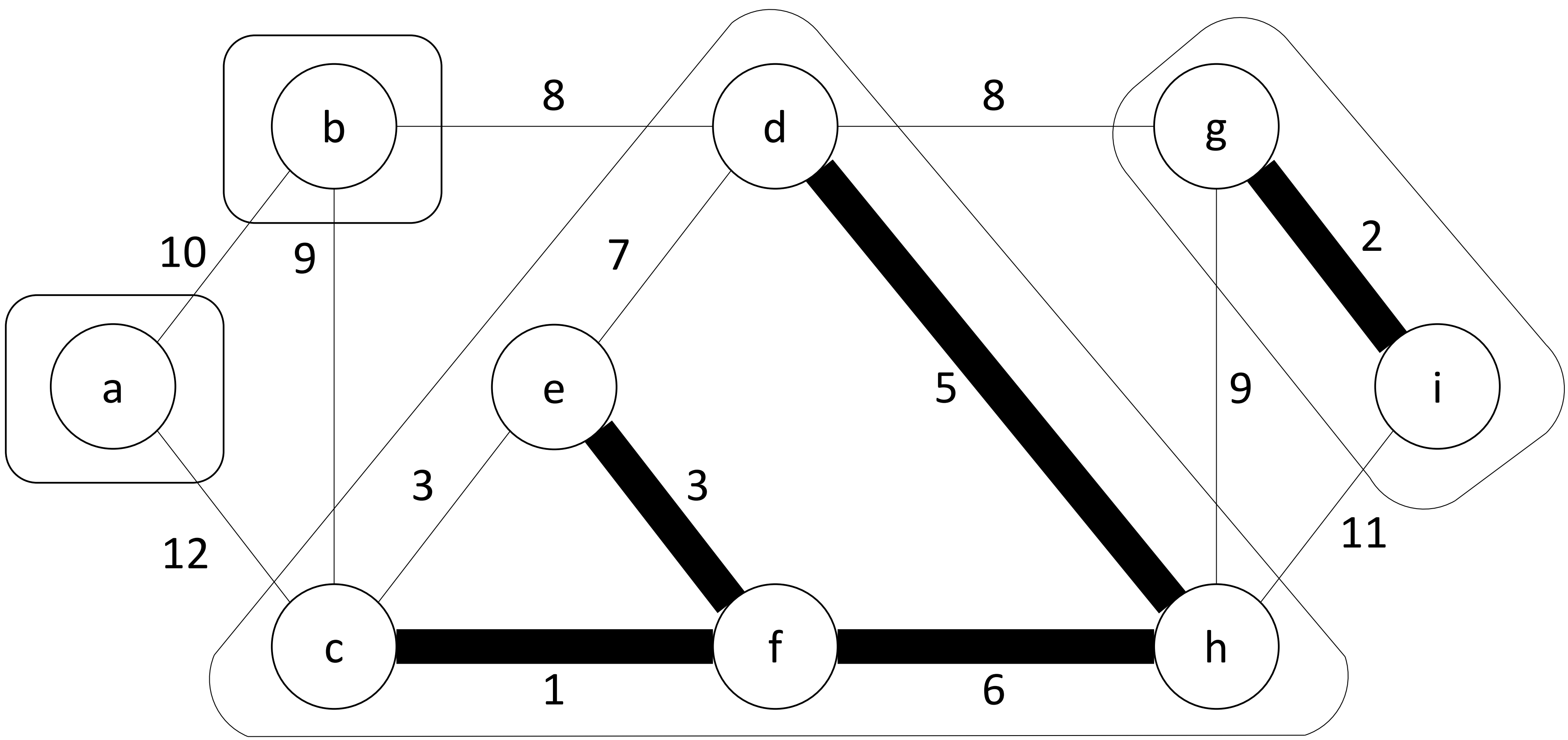
$Union(d, h)$



$$FindSet(f) \neq FindSet(h)$$

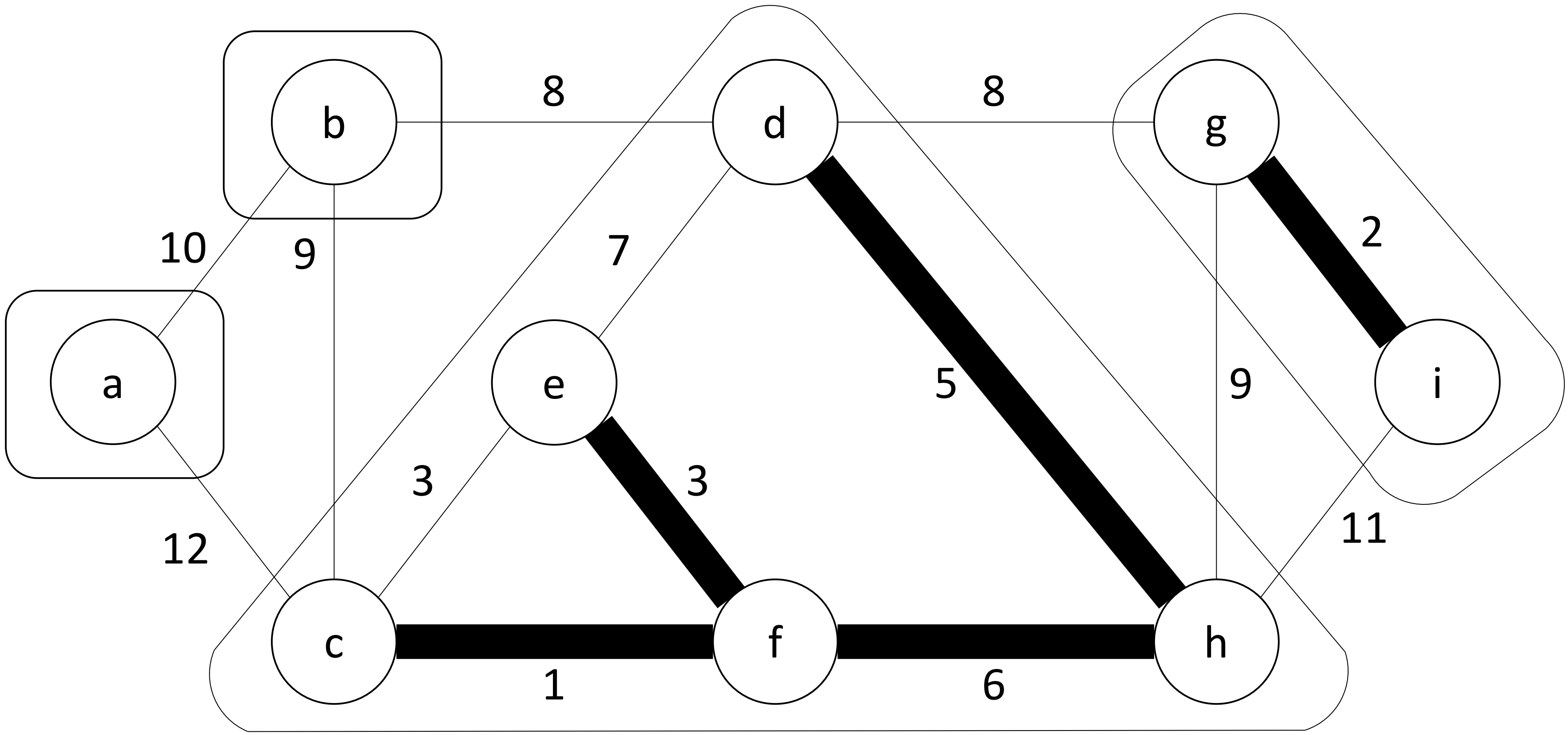


$Union(f, h)$

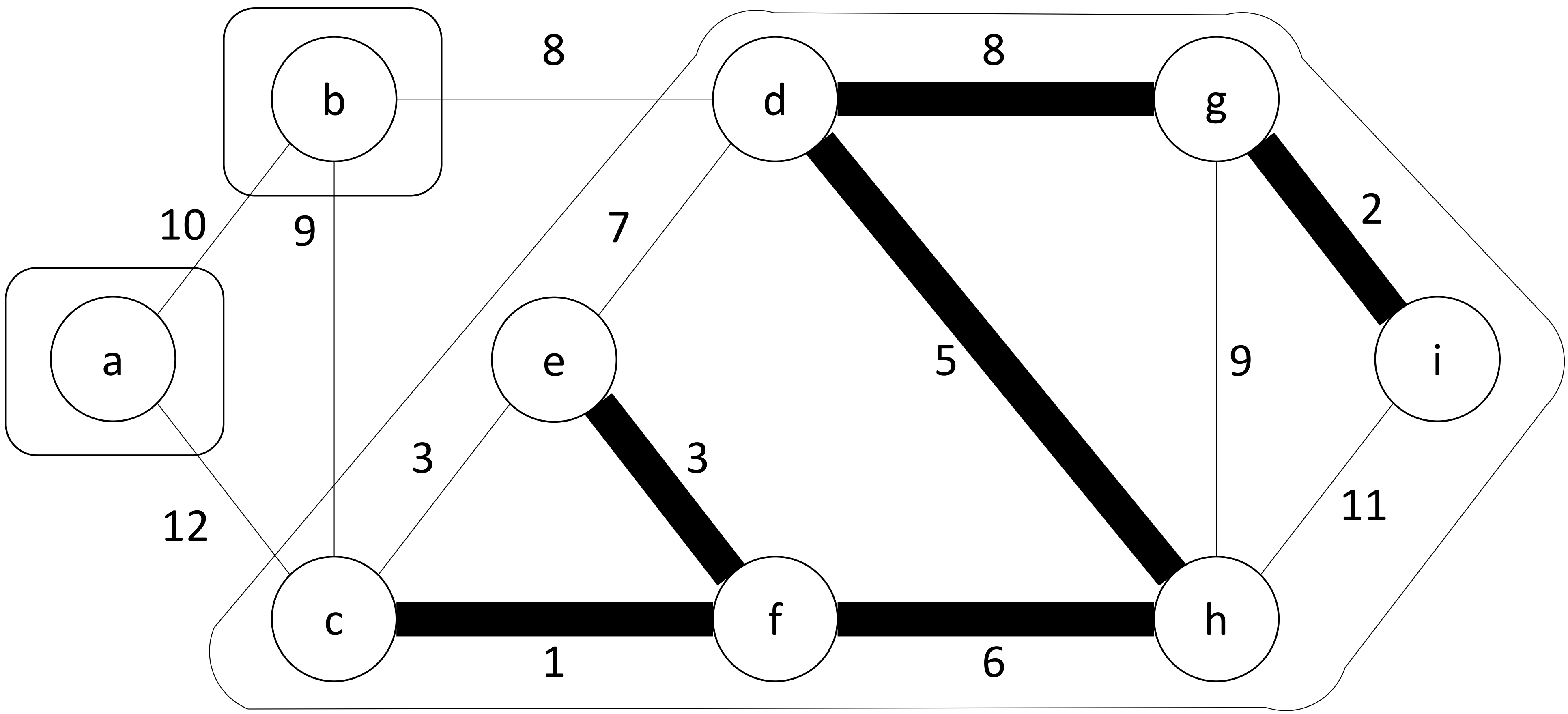


$$FindSet(e) = FindSet(d)$$

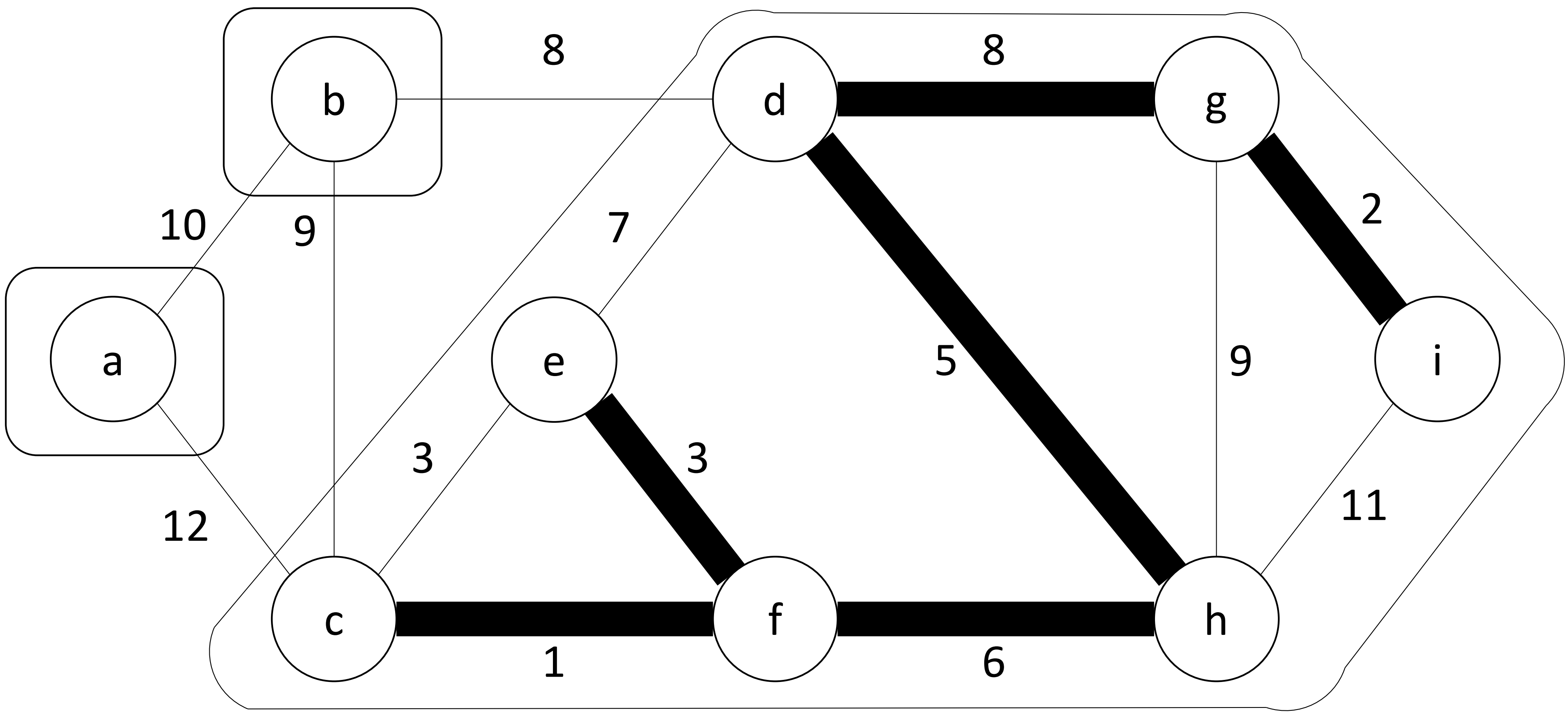
$FindSet(d) \neq FindSet(g)$



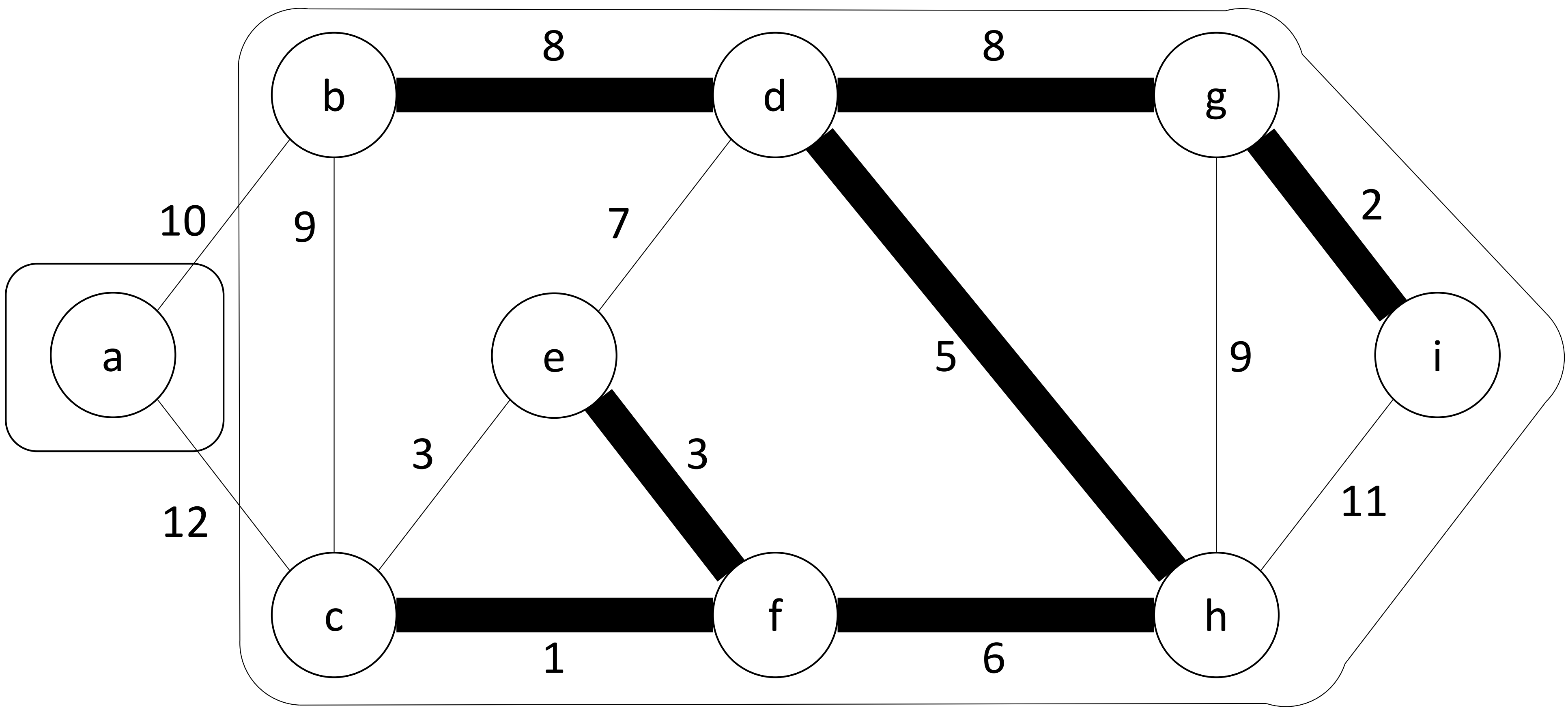
$Union(d, g)$



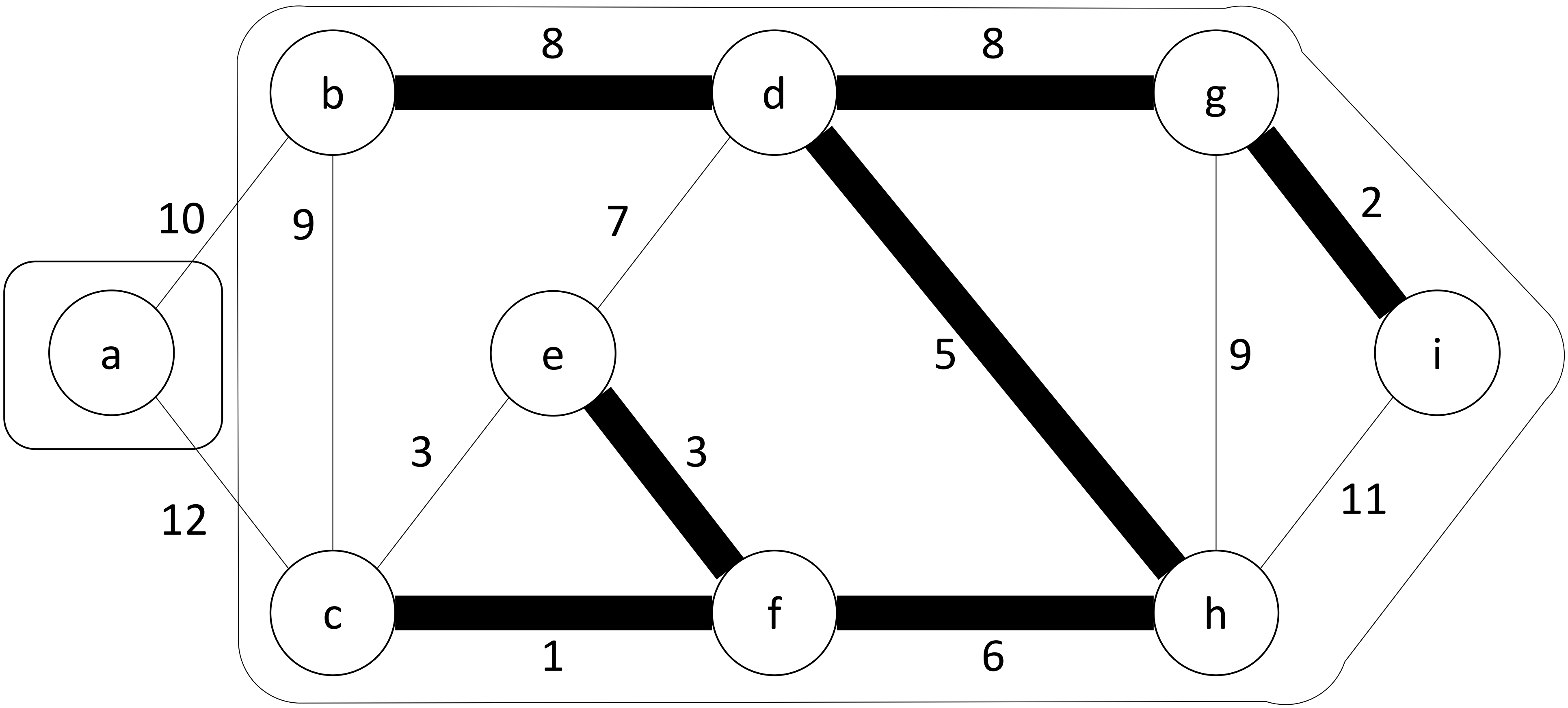
$$FindSet(b) \neq FindSet(d)$$



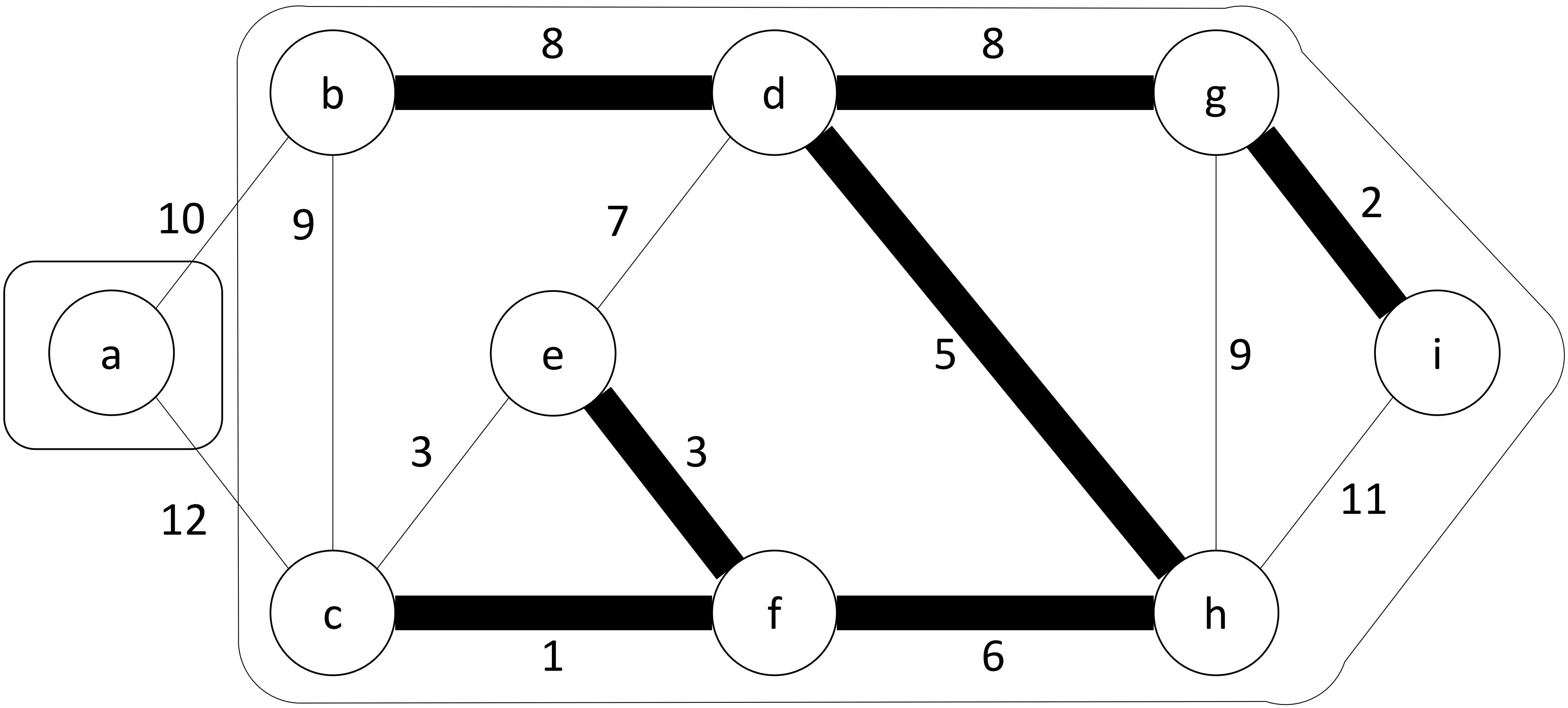
$Union(b, d)$



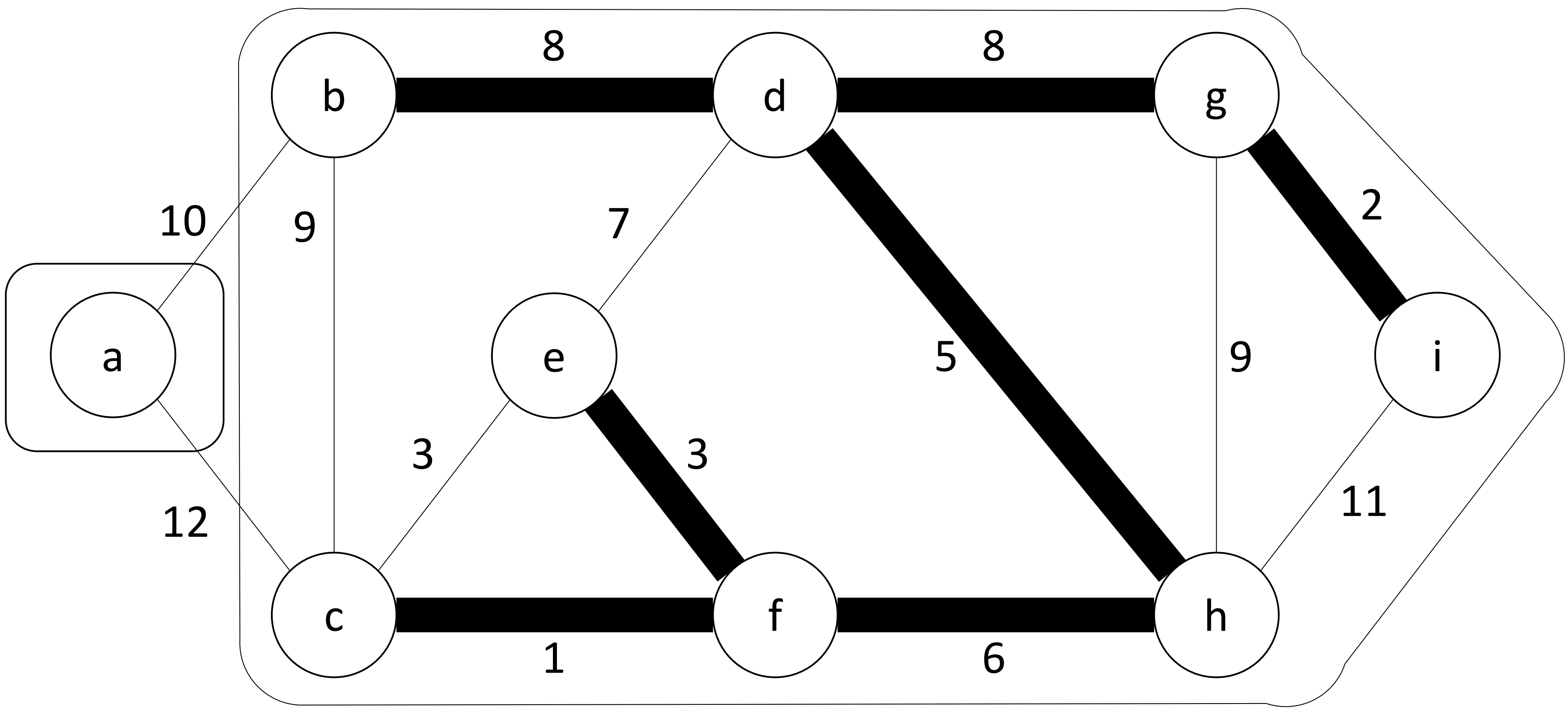
$$FindSet(b) = FindSet(c)$$



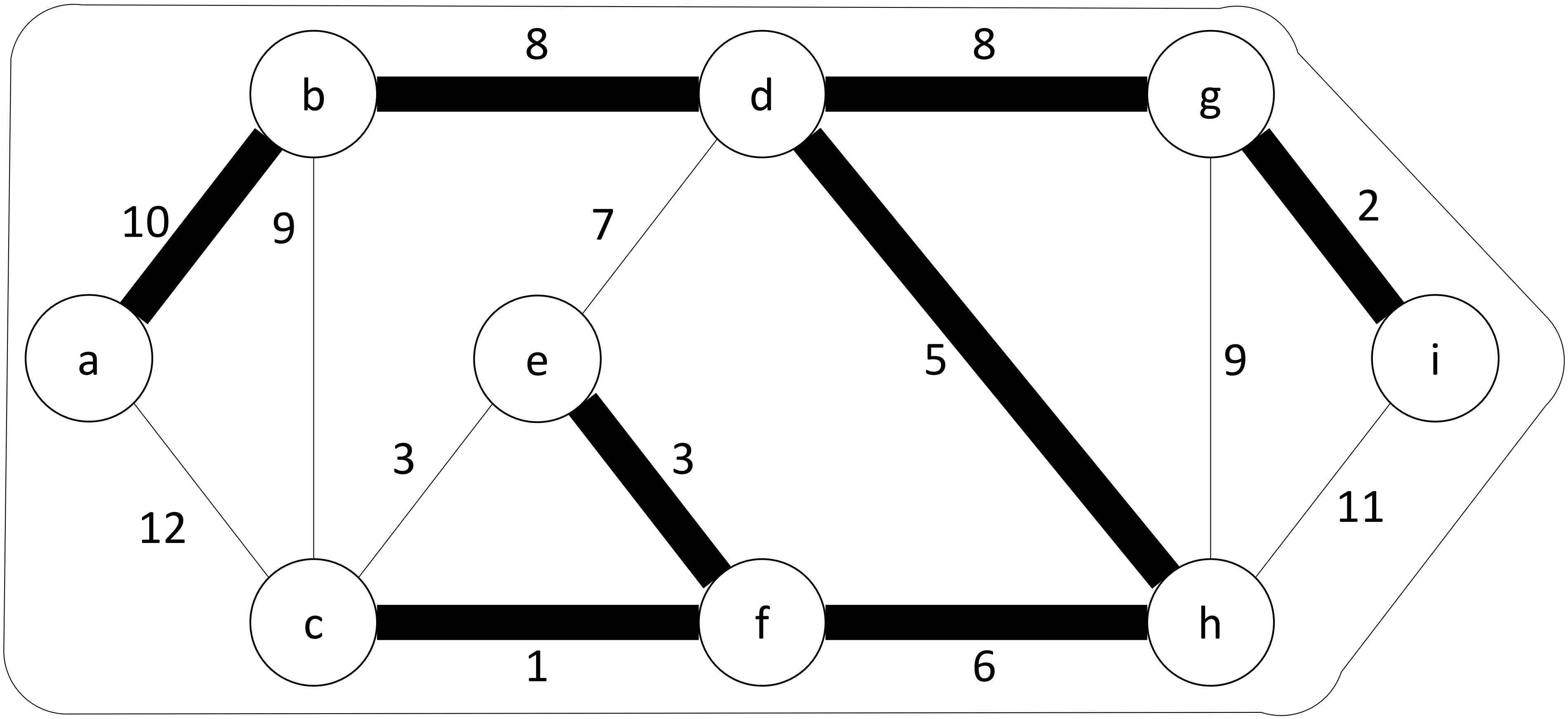
$$FindSet(g) = FindSet(h)$$



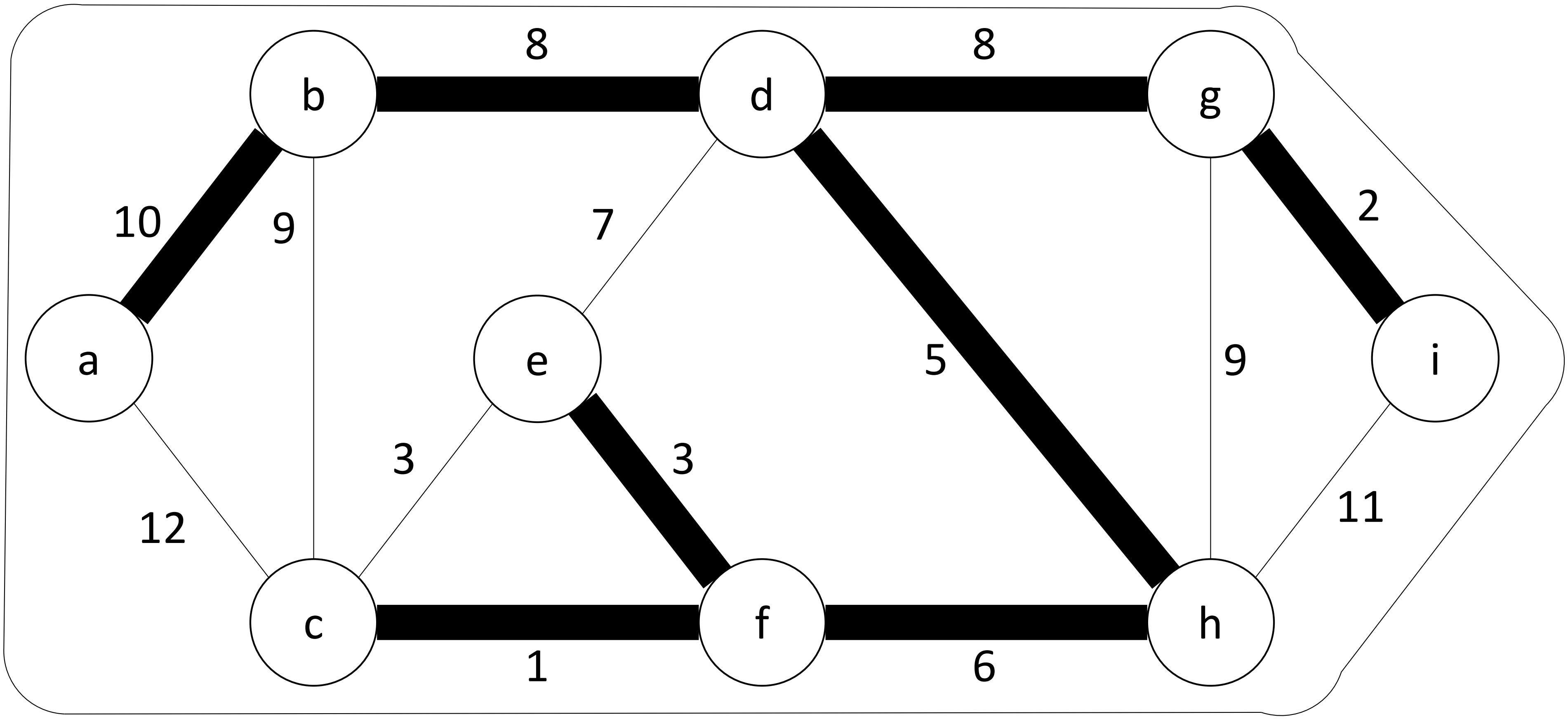
$$FindSet(a) \neq FindSet(b)$$



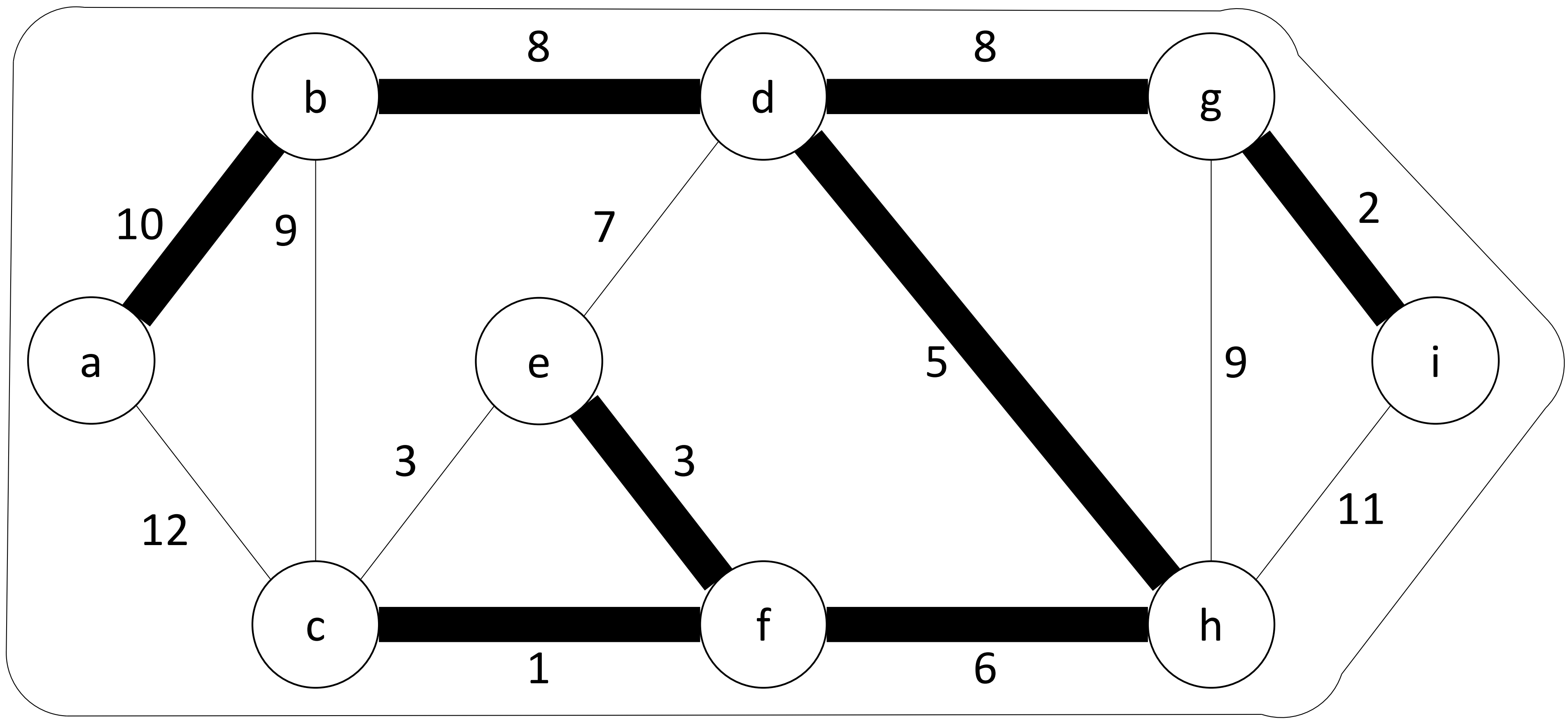
$Union(a, b)$



$$FindSet(h) = FindSet(i)$$



$FindSet(a) = FindSet(c)$



$Kruskal(G = (V, E), w)$

$A \leftarrow \emptyset$

$\boldsymbol{for}\ v \in V$

$MakeSet(v)$

sort E in non-decreasing order by weight

$\boldsymbol{for}\ \{u, v\}$ taken from the sorted list

$\boldsymbol{if}\ FindSet(u) \neq FindSet(v)$

$A \leftarrow A \cup \big\{\{u, v\}\big\}$

$Union(u, v)$

$\boldsymbol{return}\ A$

Running time analysis

Initialize A:	$O(1)$
First $\boldsymbol{for}$ loop:	$\lvert V\rvert$ MakeSets
Sort $\lvert E\rvert$:	$O(\lvert E\rvert \log\lvert E\rvert)$
Second $\boldsymbol{for}$ loop:	$O(\lvert E\rvert)$ FindSets and Unions

Using disjoint-sets datastructure:
$$O\big((|V| + |E|)\log|V|\big) + O(|E|\log|E|)$$

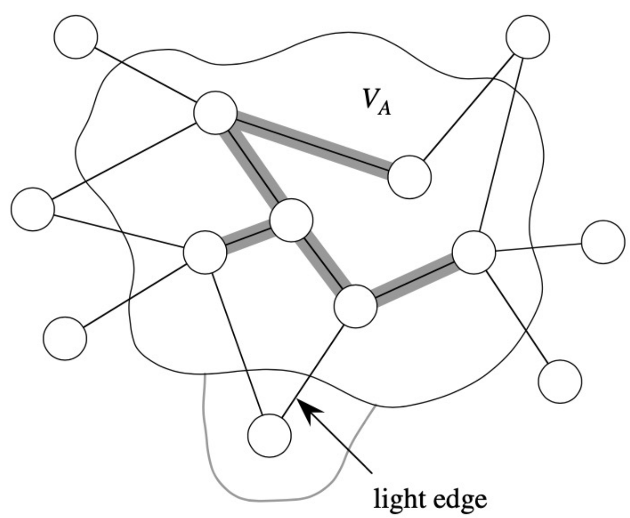
Since G is connected $|E| \geq |V| - 1$

Since $|E| \leq |V|^2$ we have
$$\log|E| = O(\log|V|)$$

Therefore, overall running time is
$$O(|E|\log|V|)$$

Prim's algorithm

- Build one tree, so A is always a tree
- Starts from an arbitrary "root" r
- At each step, find a light edge crossing $(V_A, V - V_A)$, where V_A = vertices that A is incident on. Add this edge to A.



To find a light edge quickly

- Use priority queue Q
- Each element of Q is a vertex in $V - V_A$ with key of v being minimum weight of an edge (u, v) where $u \in V_A$
- Key is ∞ if v is not adjacent to any vertex in V_A
- ExtractMin returns v such that there exists $u \in V_A$ and (u, v) is a light edge
- Edges of A form a rooted tree with root r
- Each vertex knows its parent stored in attribute $v.\pi$

EXERCISE: run this algorithm on the previous example

$Prim(G = (V, E), w, r)$

$\quad Q \leftarrow \emptyset$

$\quad \boldsymbol{for}\ u \in V$

$\qquad u.key \leftarrow \infty$

$\qquad u.\pi \leftarrow NIL$

$\qquad Q.insert(u)$

$\quad Q.decreaseKey(r, 0)\ //r.key \leftarrow 0$

$\quad \boldsymbol{while}\ Q.size(\,) > 0$

$\qquad u \leftarrow Q.extractMin(\,)$

$\qquad \boldsymbol{for}\ v \in Adj[u]$

$\qquad\quad \boldsymbol{if}\ v \in Q\ \boldsymbol{and}\ w(u, v) < v.key$

$\qquad\qquad v.\pi \leftarrow u$

$\qquad\qquad Q.decreaseKey\big(v, w(u, v)\big)$

Depends on priority queue implementation

Using binary heap:

Initialize Q and first $\boldsymbol{for}$ loop

$O(|V| \log|V|)$

Decrease key of r

$O(\log|V|)$

$\boldsymbol{while}$ loop

$|V|$ extractMin calls

$O(|V| \log|V|)$

$\leq |E|$ decreaseKey calls

$O(|E| \log|V|)$

Overall $O(|E| \log|V|)$

Possible to improve to

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

Shortest paths

- Edge-weighted graph $G = (V, E), w : E \to \mathbb{R}$
- **Weight of path** $p = \langle v_0, v_1, \dots, v_k \rangle$ is

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) = \text{sum of edge weights on } p$$

- **Shortest-path weight** u to v:

$$\delta(u, v) = \begin{cases} \min\left(w(p) : u \overset{p}{\to} v\right) & \text{if there is a path from } u \text{ to } v \\ \infty & \text{otherwise} \end{cases}$$

- Can think of weights as representing any measure that accumulates linearly along a path and we wish to minimize it

Variants of shortest paths problems

- **Single-source**
  - Find shortest paths from a given source vertex $s \in V$ to every vertex $v \in V$
- **Single-destination**
  - Find shortest paths to a given destination vertex
- **Single-pair**
  - Find shortest path from u to v. Not known how to do it faster than single-source.
- **All-pairs**
  - Find shortest path from u to v for all $u, v \in V$.

Negative-weight edges

Some algorithms will not work when negative-weight edges are present

Other algorithms will work with negative-weight edges so long as there are no negative-weight cycles reachable from the source

If we have a negative-weight cycle, we can just keep going around it, and get $\delta(s, v) = -\infty$ for all v on the cycle

Some algorithms allow one to detect presence of negative-weight cycles

Some properties of shortest paths

- **Optimal substructure property**

  Any subpath of a shortest path is a shortest path itself

- **No cycles property**

  Shortest paths do not contain cycles without loss of generality

- **Triangle inequality**

  For all $(u, v) \in E$ we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$

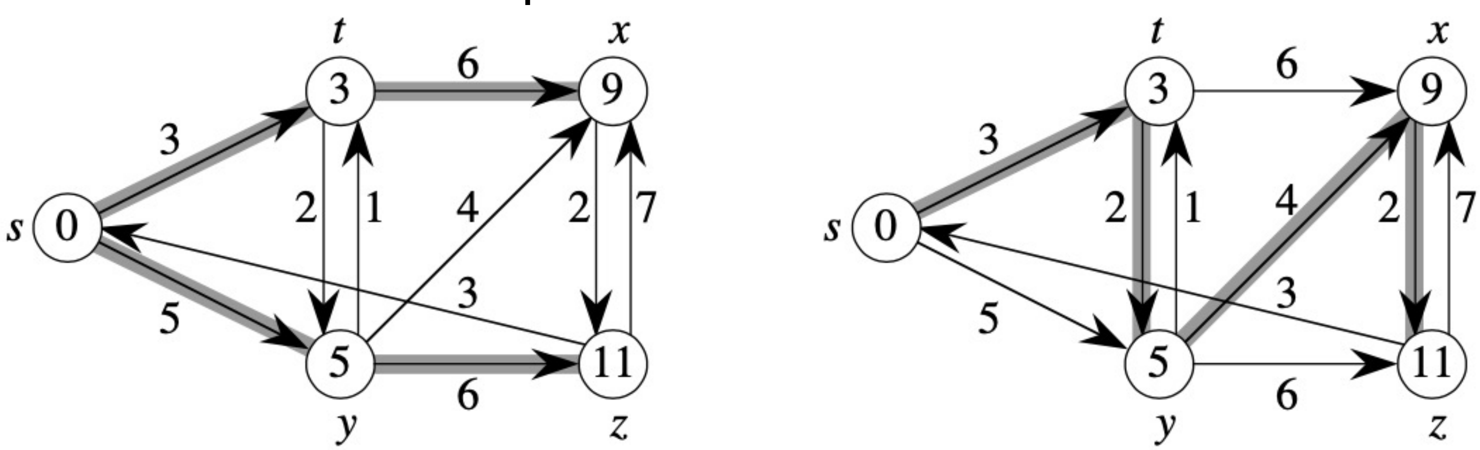
Single-source shortest paths (CLRS 24)

Input: $G = (V, E), w : E \to \mathbb{R}$

source vertex $s \in V$

Output: for each vertex v populate attribute $v.d = \delta(s, v)$

for each vertex v populate attribute $v.\pi$ = predecessor of v on shortest path from s



Generic algorithm

- Initially set $v.d \leftarrow \infty$
- As an algorithm progresses, $v.d$ reduces but satisfies $v.d \geq \delta(s, v)$
- Call $v.d$ a **shortest path estimate**
- Initially set $v.\pi \leftarrow NIL$
- The predecessor graph $\{(v.\pi, v)\}$ forms a tree called **shortest-path tree**
- Shortest path estimate is improved by **relaxing an edge**

Generic algorithm

$InitSingleSource(G = (V, E), s)$

  $\boldsymbol{for}\ v \in V$

    $v.d \leftarrow \infty$

    $v.\pi \leftarrow NIL$

  $s.d \leftarrow 0$

$Relax(u, v, w)$

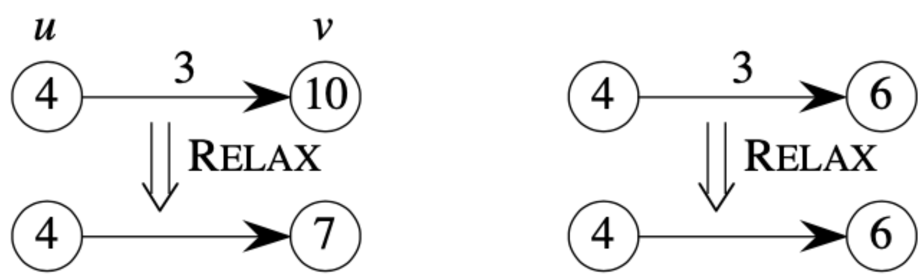
  // (u, v) is an edge

  // w is the weight function

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

    $v.d \leftarrow u.d + w(u, v)$

    $v.\pi \leftarrow u$



- All single-source shortest paths algorithms we consider
  - Start by calling $InitSingleSource$
  - Then relax edges
- Algorithms differ in the order and number of times edges are relaxed

- Upper bound property
  - Always have $v.d \geq \delta(s, v)$ for all $v \in V$
- Path relaxation property
  - If $p = \langle v_0, v_1, \ldots, v_k \rangle$ is a shortest path from $v_0 = s$ to $v = v_k$. If we relax edges in order $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$ even intermixed with other relaxations then we get $v.d = \delta(s, v)$

Dijkstra's algorithm

- Solves single-source shortest-paths problem
- Assume input graph **has no negative-weight edges**
- Essentially a weighted version of BFS
  - Instead of regular queue, use a priority queue
  - Keys are shortest-path weights $v.d$
- Have two sets of vertices
  - $S =$ vertices whose final shortest-path weights have been determined
  - $Q =$ priority queue $= V - S$

$Dijkstra(G = (V, E), w, s)$

$\quad InitSingleSource(G, s)$

$\quad S \leftarrow \emptyset$

$\quad \boldsymbol{for}\ u \in V$

$\quad\quad Q.insert(u)$

$\quad \boldsymbol{while}\ Q.size(\,) > 0$

$\quad\quad u \leftarrow Q.extractMin(\,)$

$\quad\quad S \leftarrow S \cup \{u\}$

$\quad\quad \boldsymbol{for}\ v \in Adj[u]$

$\quad\quad\quad Relax(u, v, w)$

$\quad\quad\quad \boldsymbol{if}\ v.d$ changed

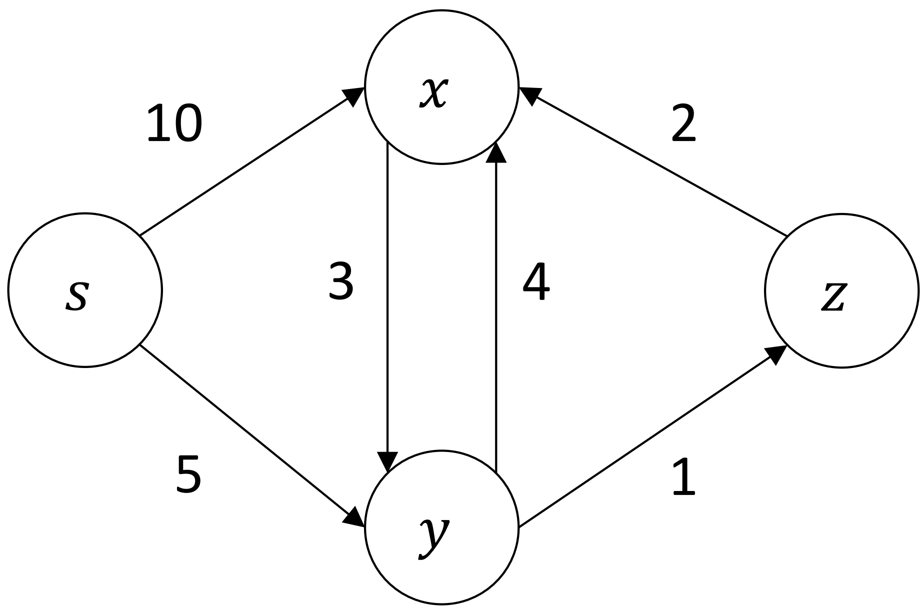
$\quad\quad\quad\quad Q.decreaseKey(v, v.d)$

$S = \emptyset$
$Q = \langle s, x, y, z \rangle$

$InitSingleSource$

$x.\pi = NIL$
$x.d = \infty$

$s.\pi = NIL$
$s.d = 0$

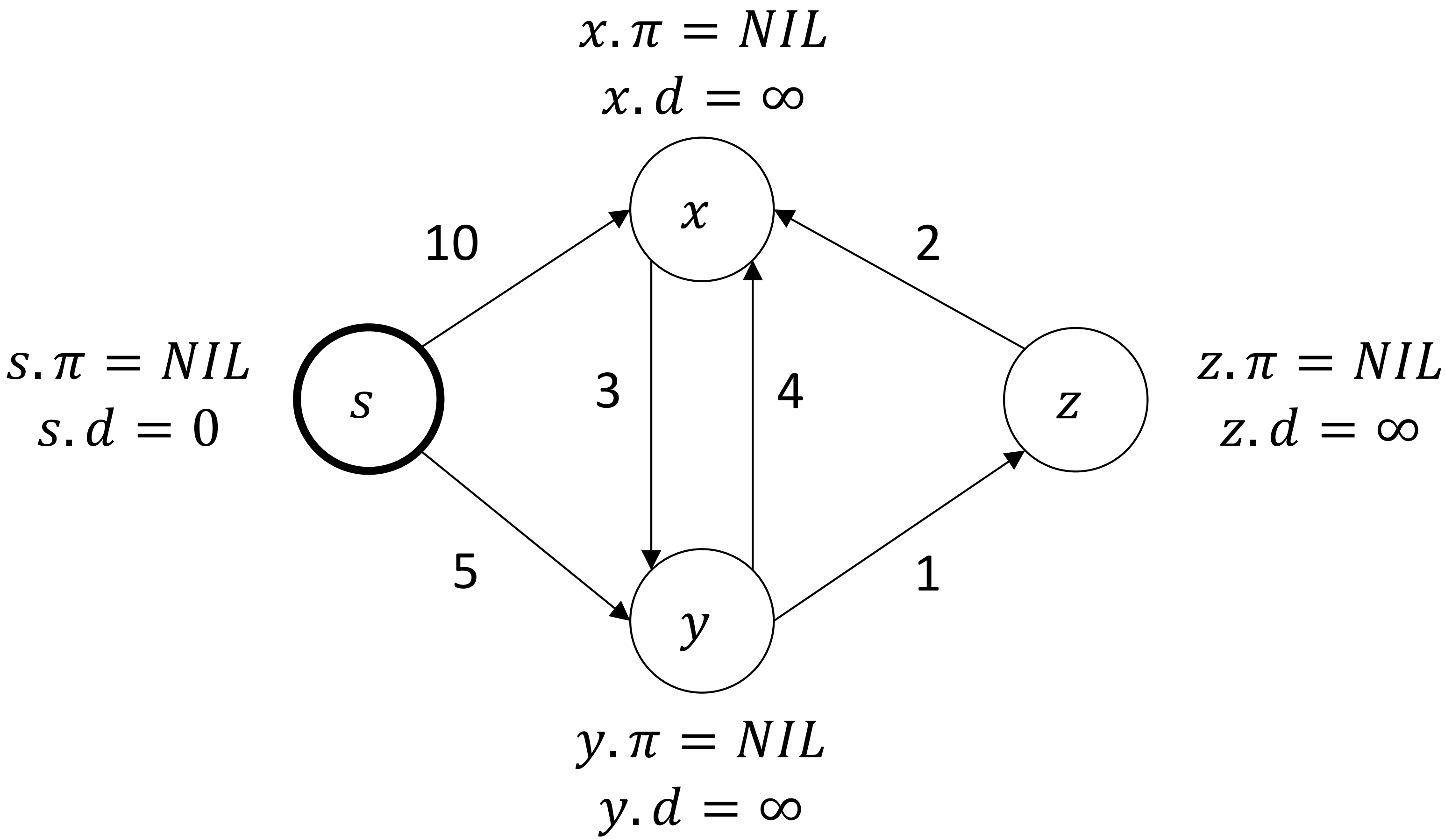


$z.\pi = NIL$
$z.d = \infty$

$y.\pi = NIL$
$y.d = \infty$

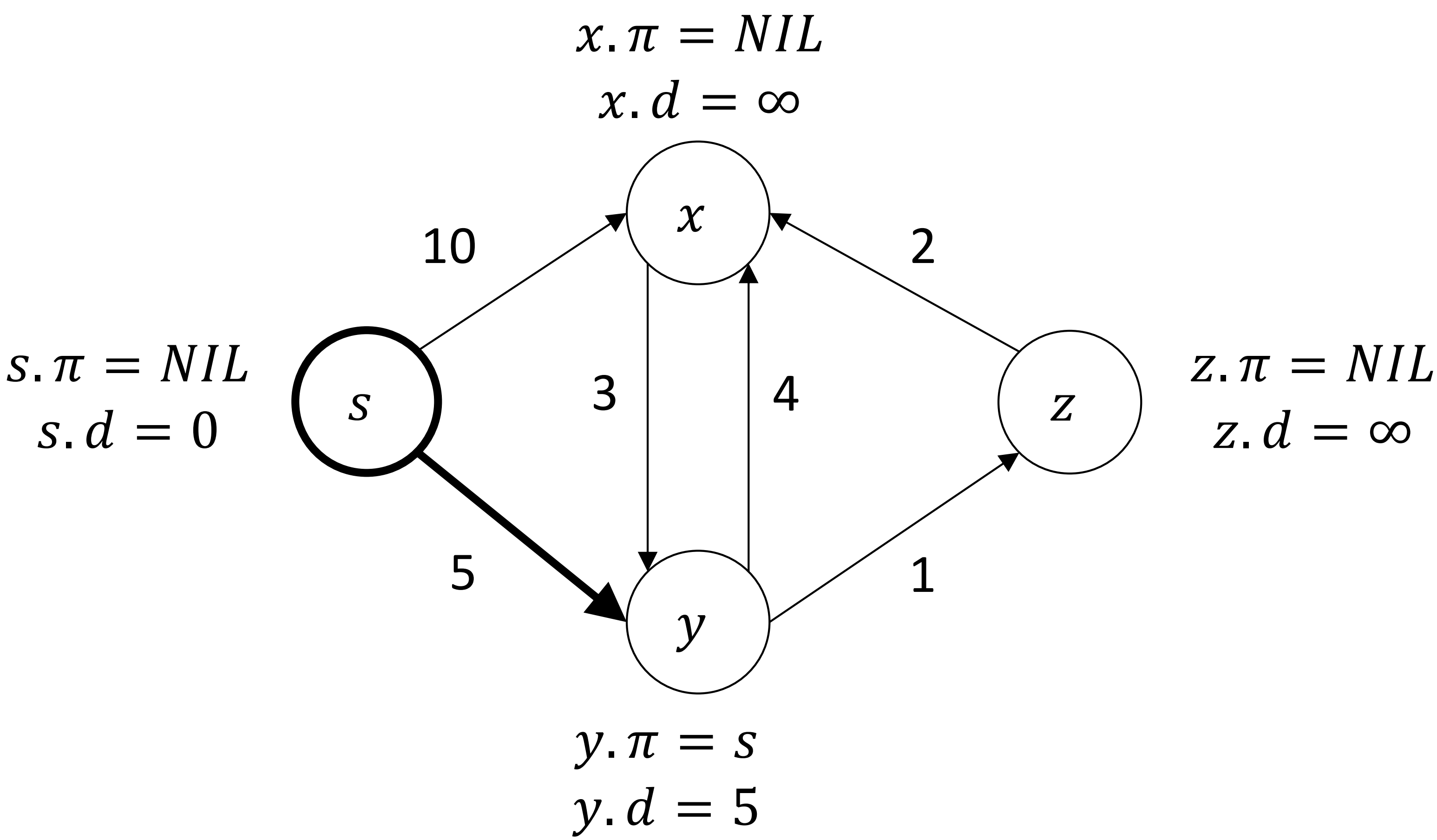
$S = \{s\}$
$Q = \langle x, y, z \rangle$

Process s



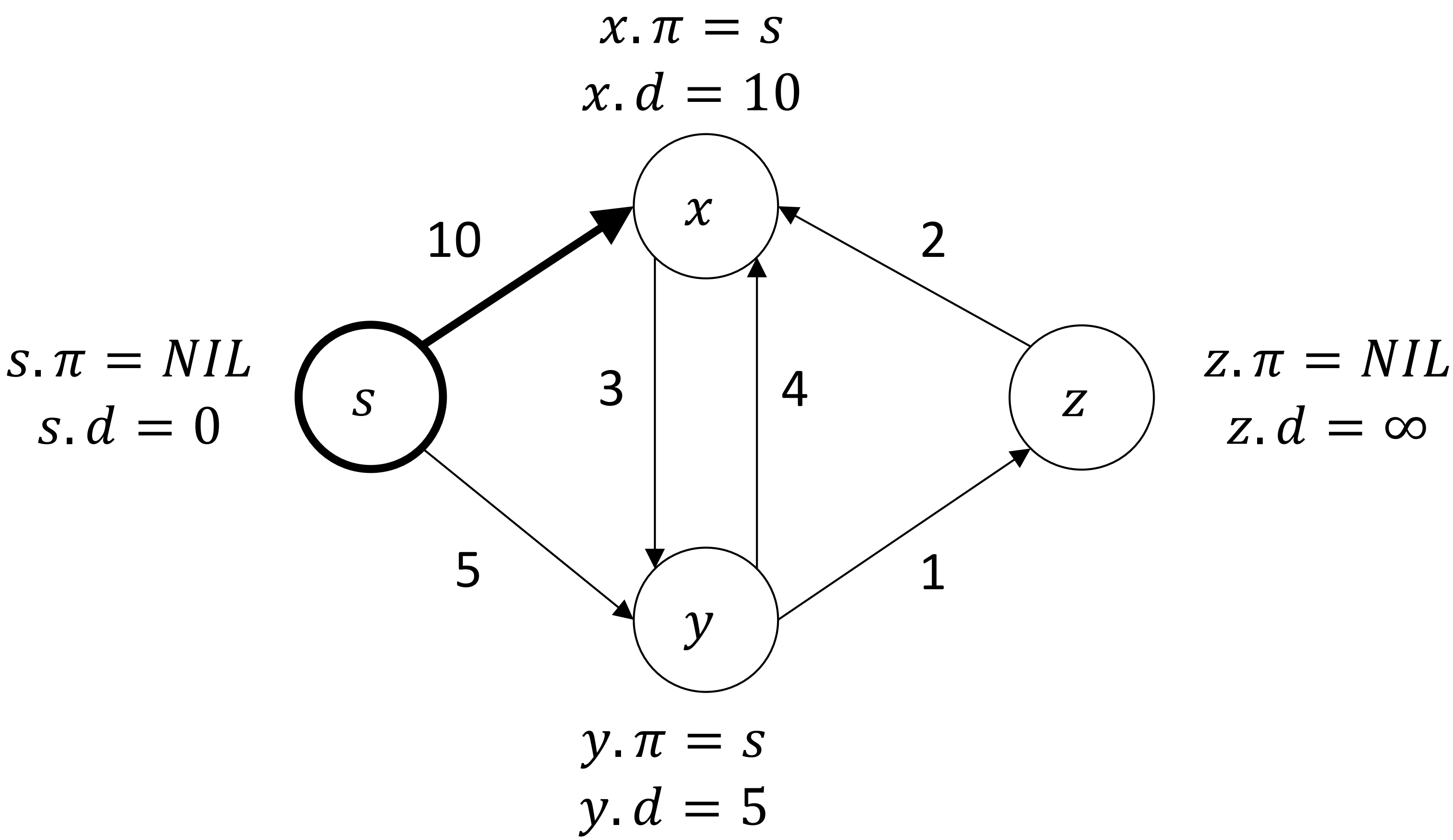
$S = \{s\}$
$Q = \langle x, y, z \rangle$

Process s $\quad Relax(s, y, w)$



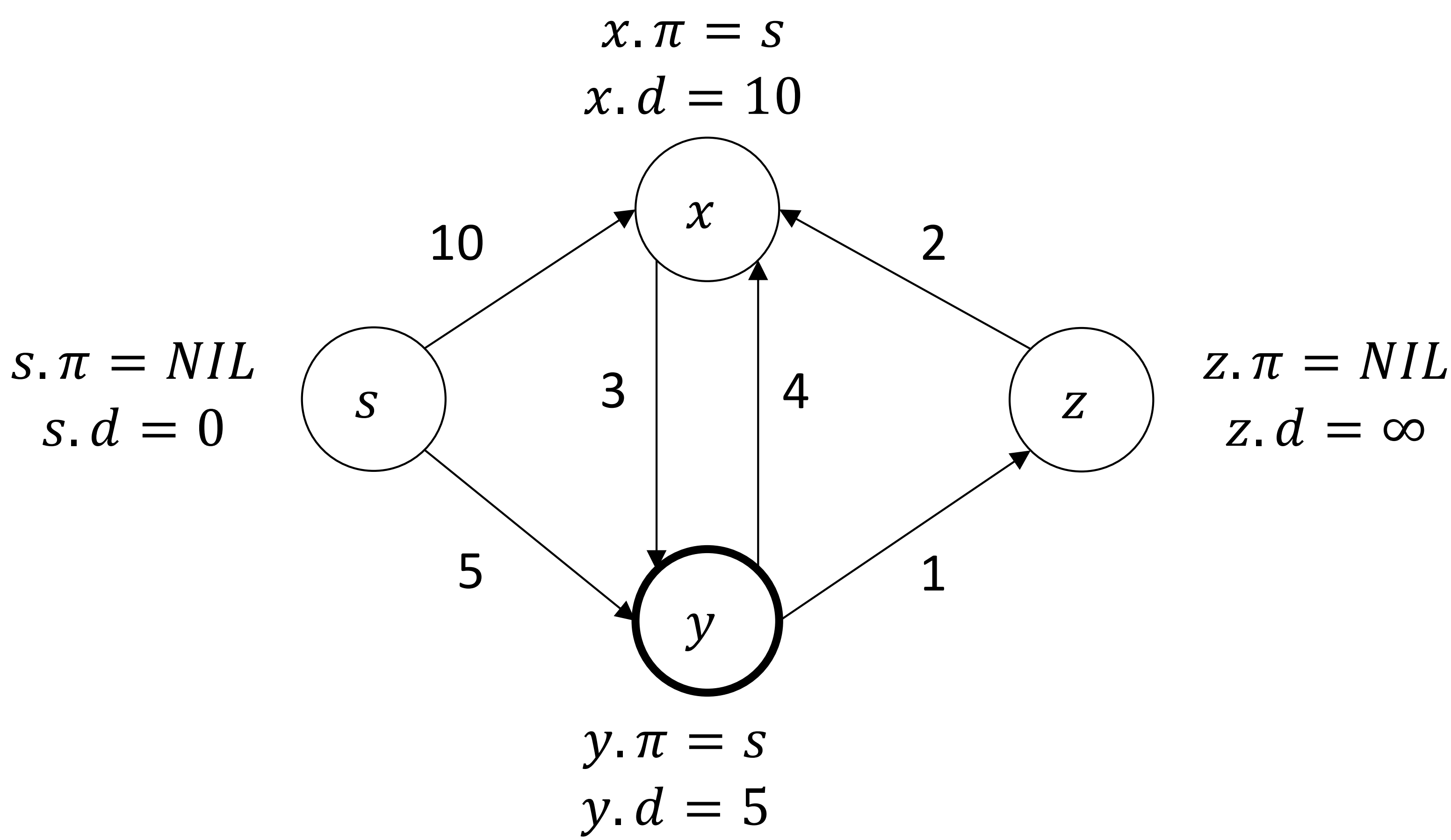
$S = \{s\}$
$Q = \langle y, x, z \rangle$

Process s $\quad$ $Relax(s, x, w)$



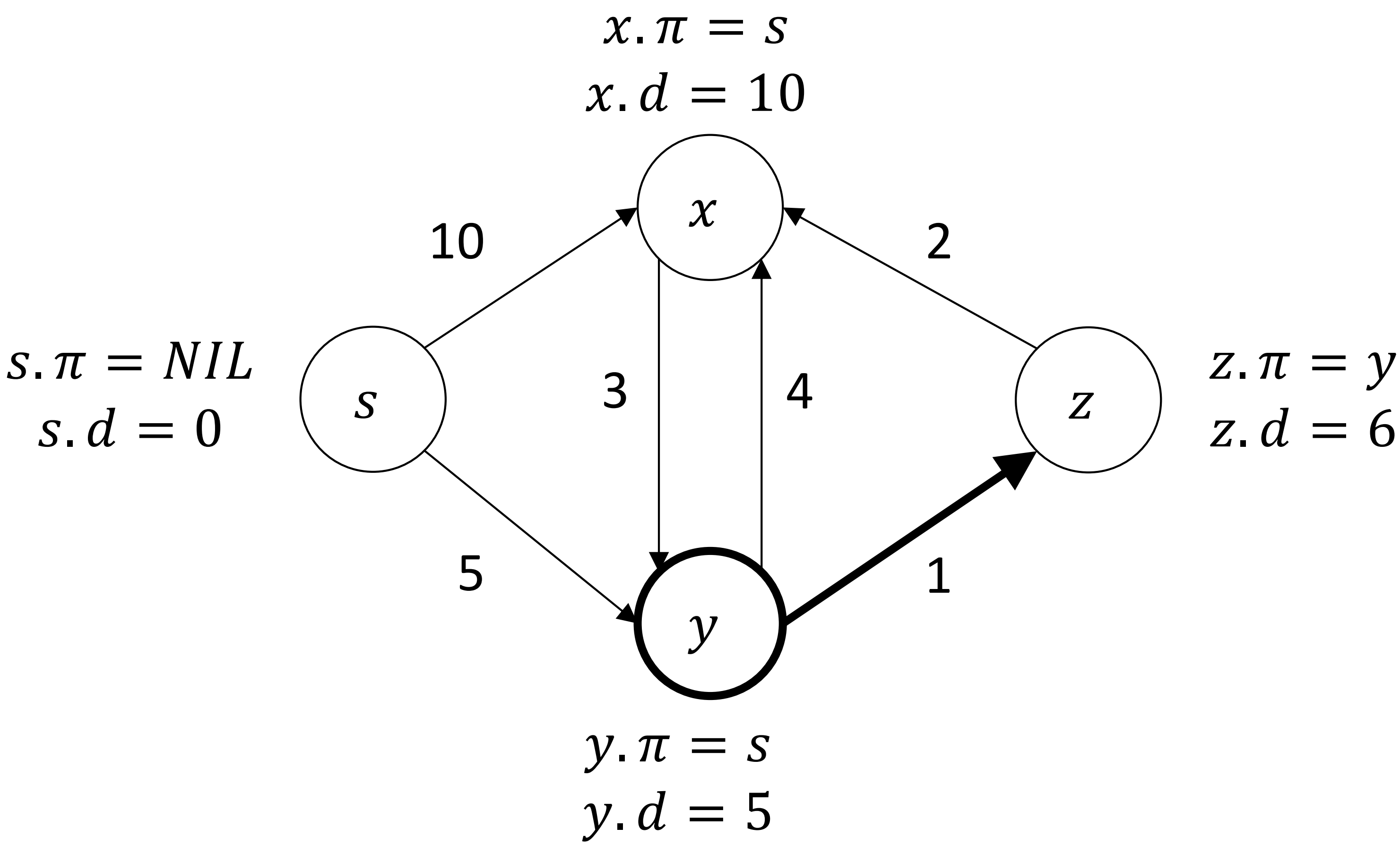
$S = \{s, y\}$
$Q = \langle x, z \rangle$

Process y



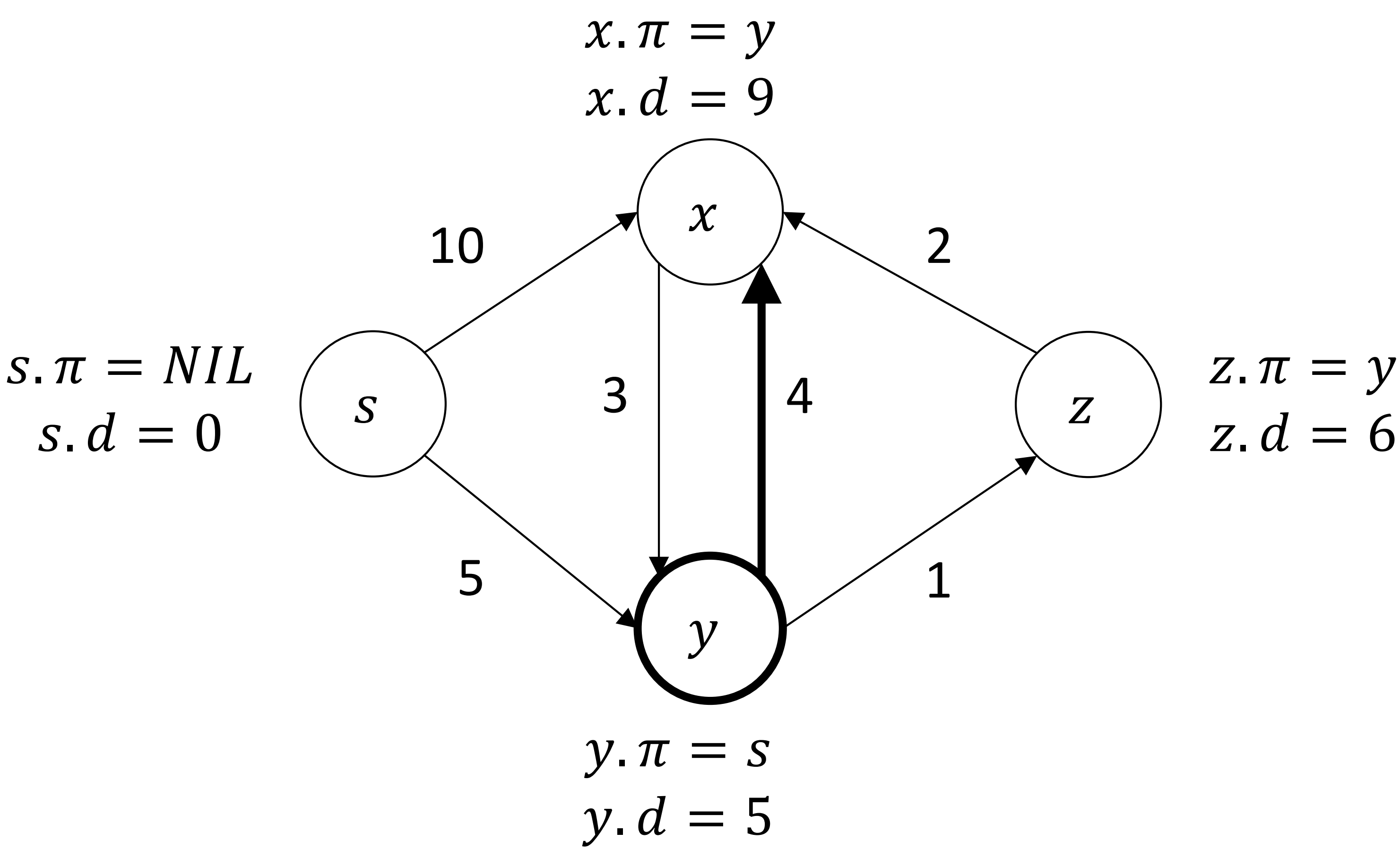
$S = \{s, y\}$
$Q = \langle z, x \rangle$

Process y $\quad$ $Relax(y, z, w)$



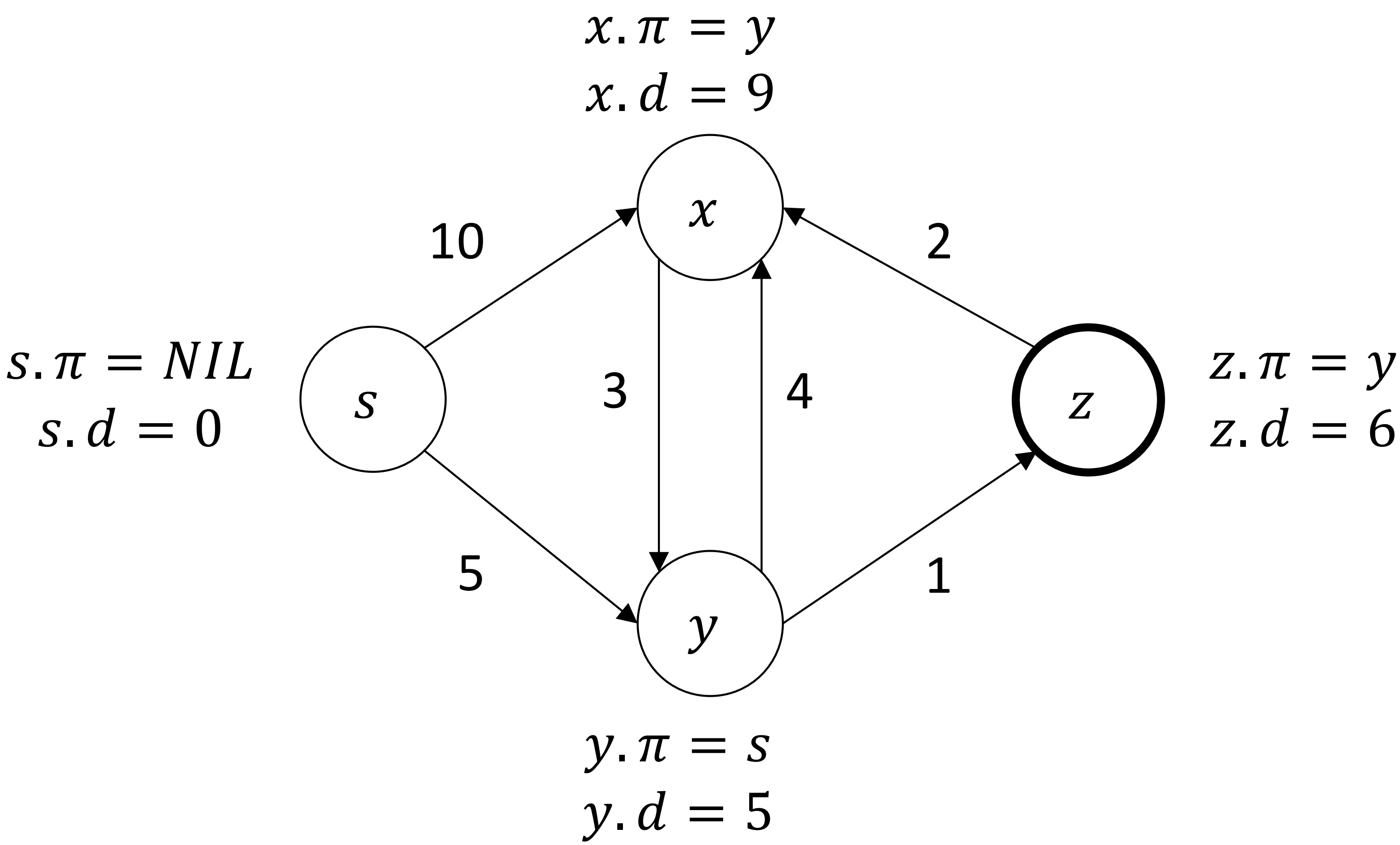
$S = \{s, y\}$
$Q = \langle z, x \rangle$

Process y $\quad$ $Relax(y, x, w)$



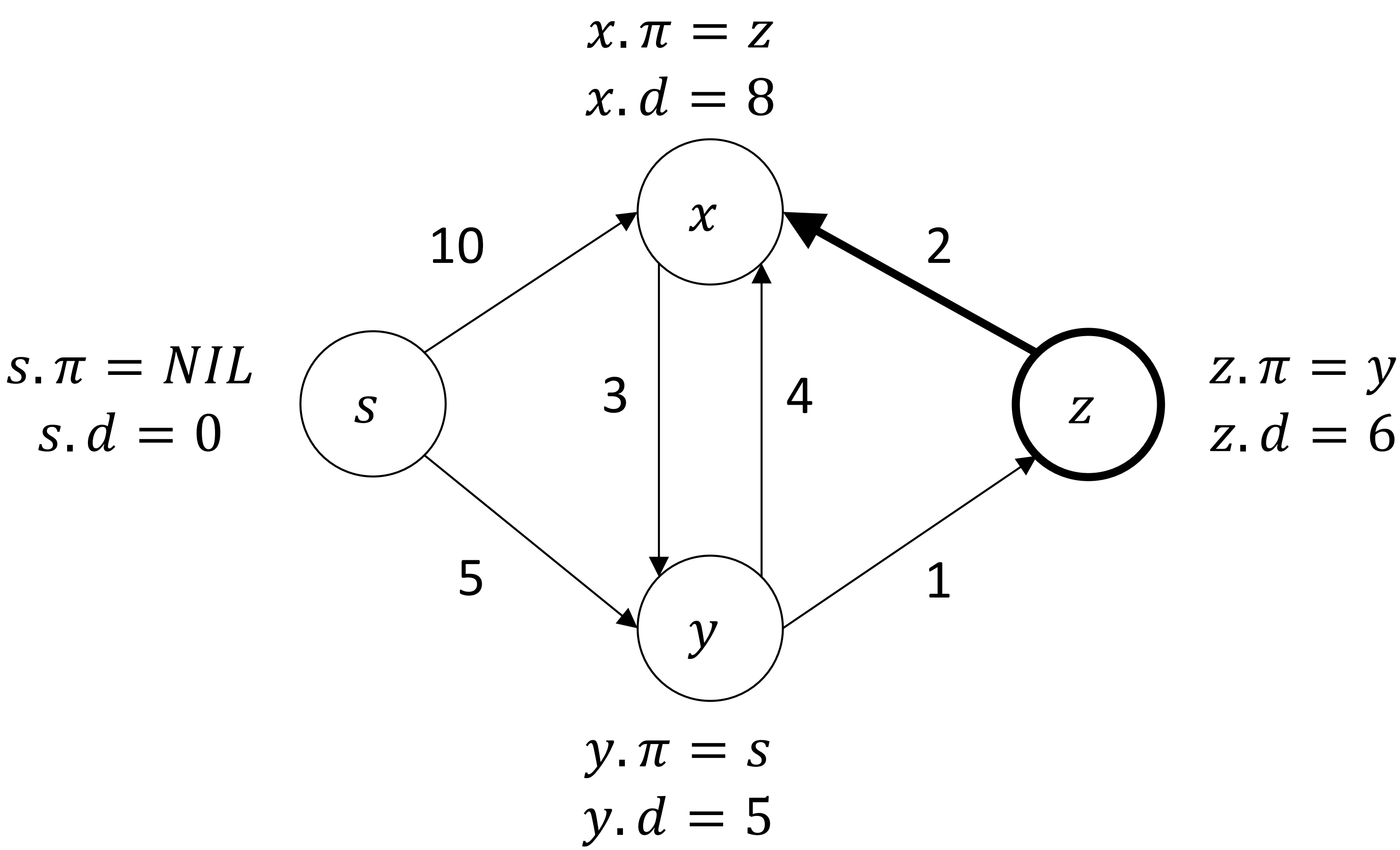
$S = \{s, y, z\}$
$Q = \langle x \rangle$

Process z



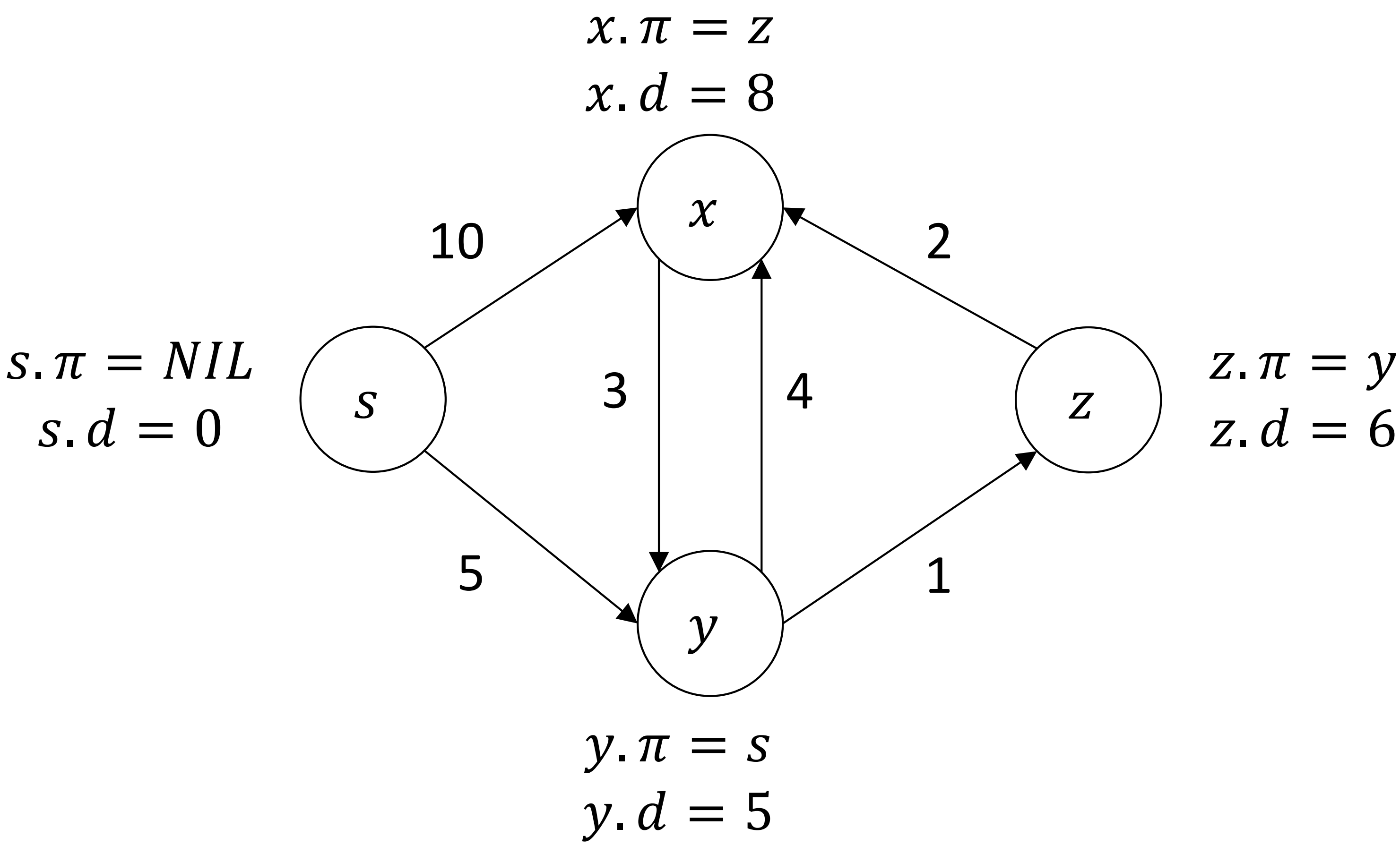
$S = \{s, y, z\}$
$Q = \langle x \rangle$

Process z $\quad$ $Relax(z, x, w)$



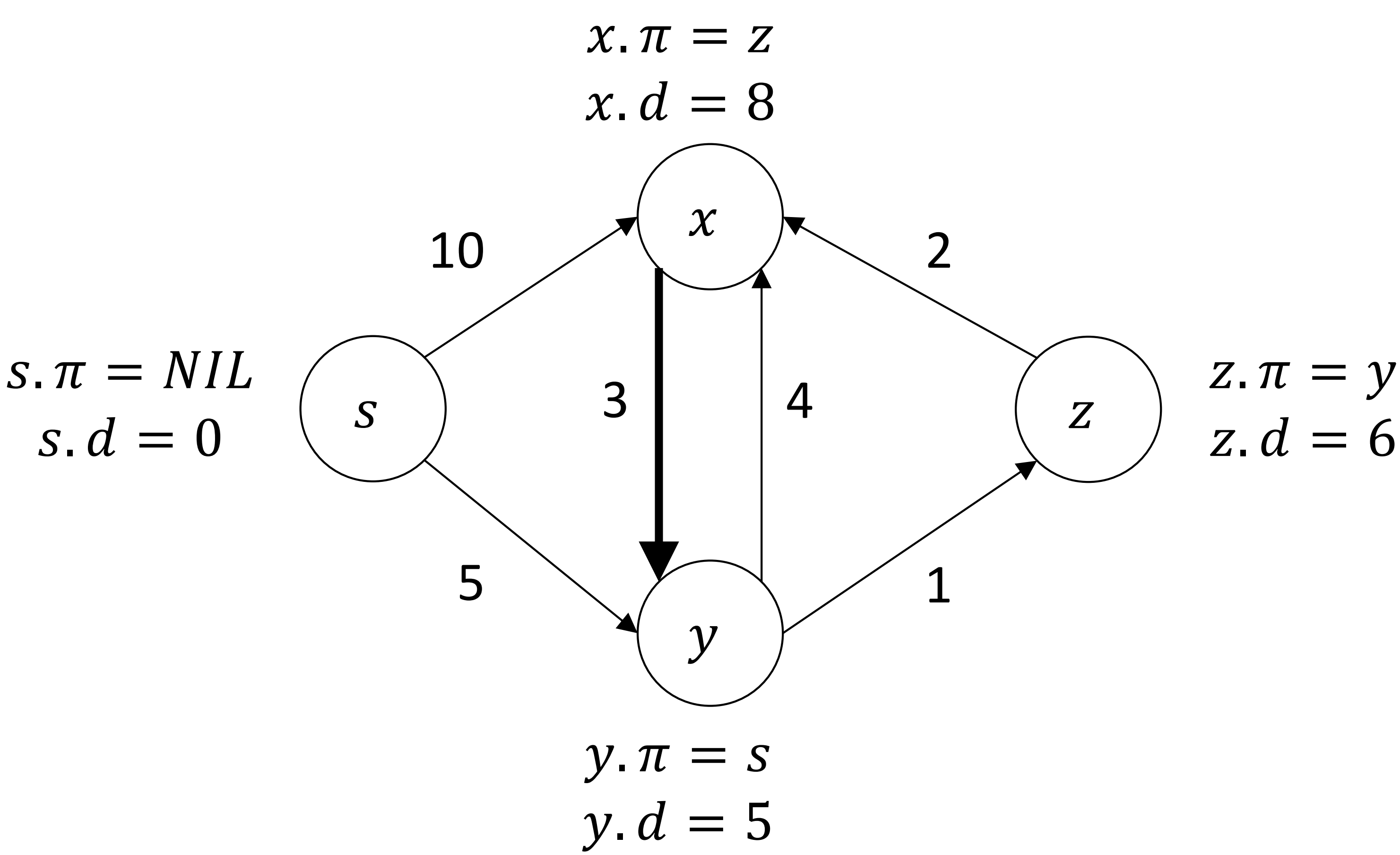
$S = \{s, y, z, x\}$
$Q = \emptyset$

Process x



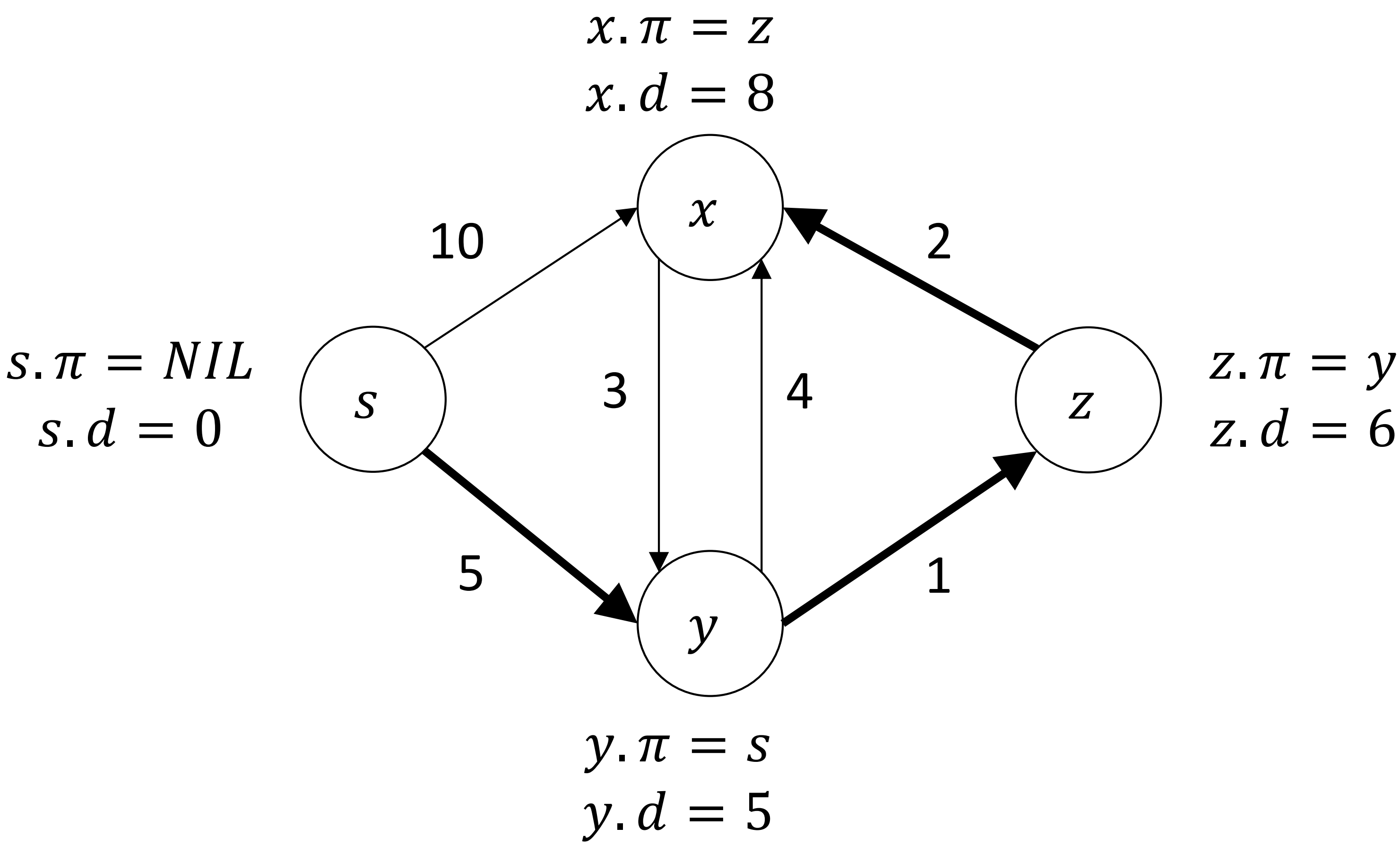
$S = \{s, y, z, x\}$
$Q = \emptyset$

Process x $\quad$ $Relax(x, y, w)$



$S = \{s, y, z, x\}$
$Q = \emptyset$

Final result with shortest-path tree



Now you should be able to…

- Use the basic graph terminology effectively
- Represent graphs using adjacency matrix and adjacency lists
- Describe BFS/DFS in plain English, pseudocode, explain their properties and running time
- Use BFS/DFS as a subroutine to solve various graph problems
- Take transpose of a graph
- Compute topological sort of a dag
- Compute SCCs of a digraph
- Solve single-source shortest paths problem in weighted directed graphs without negative-weight edges

Review questions

- Write down pseudocode for BFS, DFS, topological sort, SCCs, and Dijkstra without using any external resources
- Analyze correctness and running time of each of the above algorithms
- For each of the above algorithms, decide what algorithmic paradigm it belongs to? Greedy? Divide and conquer? Dynamic programming? Why?