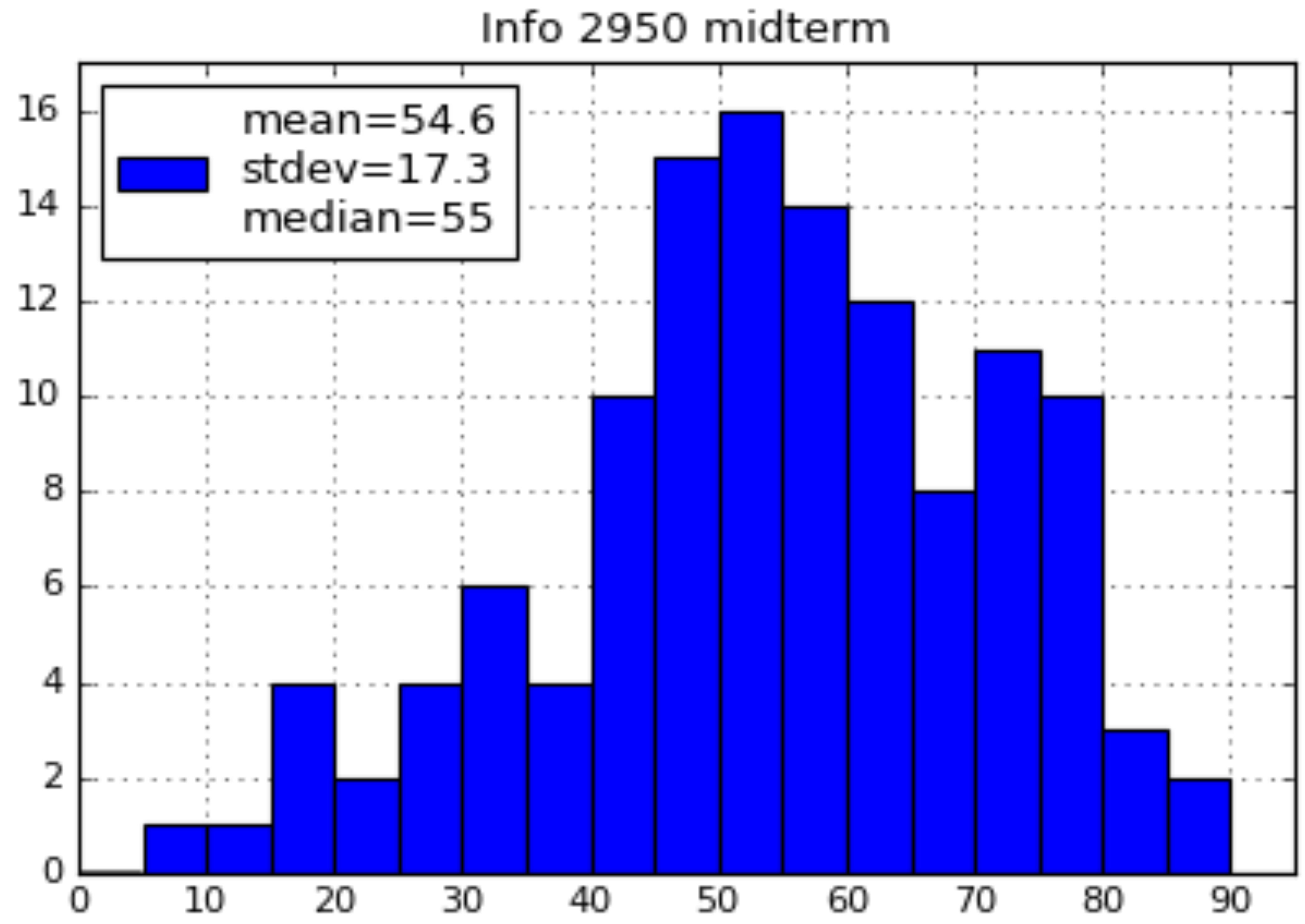
Info 2950, Lecture 16

28 Mar 2017

Prob Set 5: due Fri night 31 Mar



Breadth first search (BFS) and Depth First Search (DFS)

Must have an ordering on the vertices of the graph.

In most examples here, the vertices have been labeled by $\{1, 2, \ldots, n\}$
where n is the number of vertices.
Gives a natural ordering

These algorithms output a rooted spanning tree.

DFS algorithm

Initialize $T = (V, E)$:

$V = v_1$

$E = \{\}$

$v = v_1$

---

if there's an edge (v,w) such that w is not already in V:

    add the edge to E

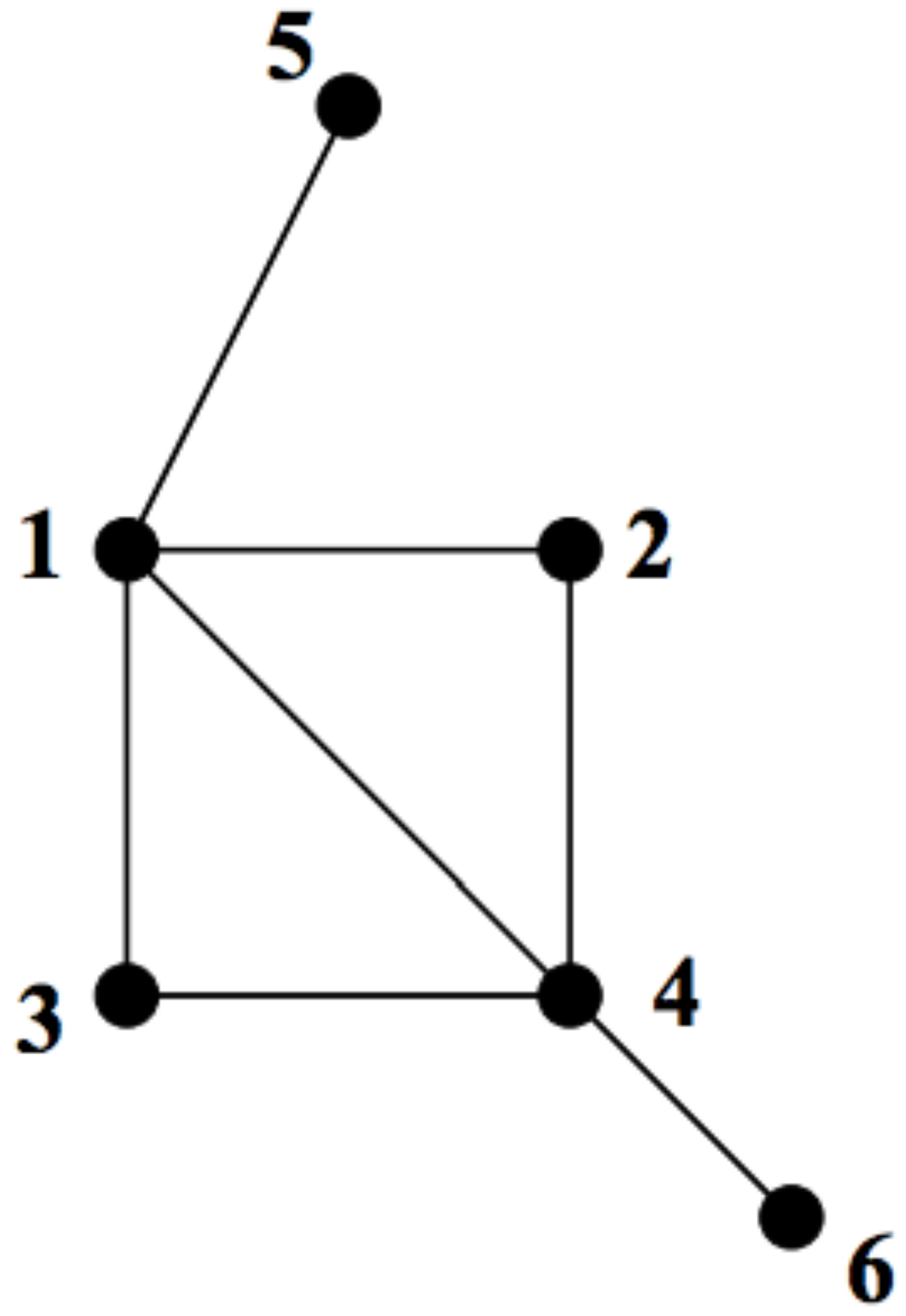
    $v = w$ (i.e., make w the new v), repeat

else:

    if $v == v_1$: done (have made it back up to root)

    else: v = parent of v (make parent of v the new v), and repeat

DFS



if there's an edge (v,w) such that w is not already in V:

    add the edge to E

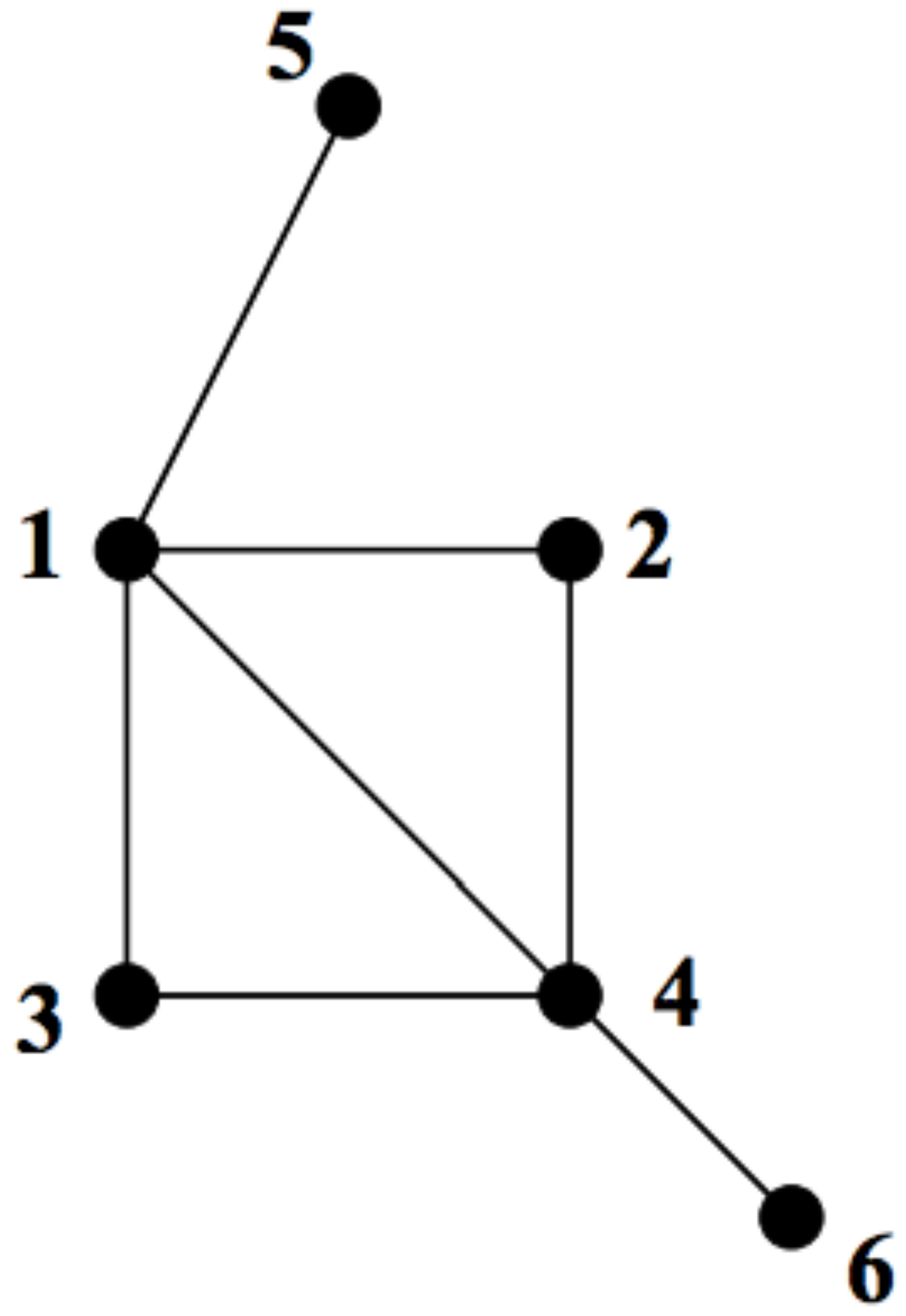
    v = w (i.e., make w the new v), repeat

else:

    if v == v_1 : done (have made it back up to root)

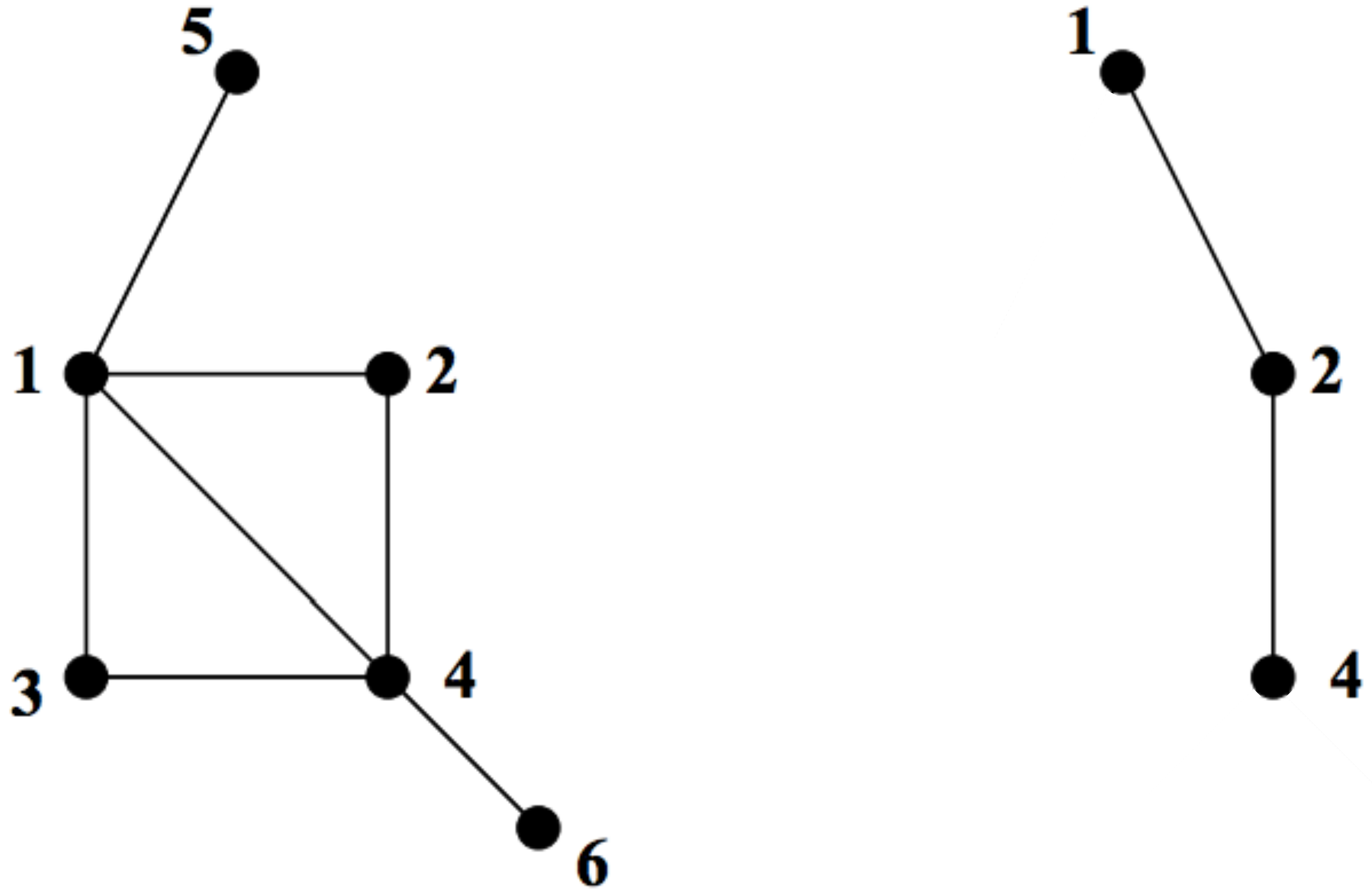
    else: v = parent of v (make parent of v the new v), and repeat

DFS



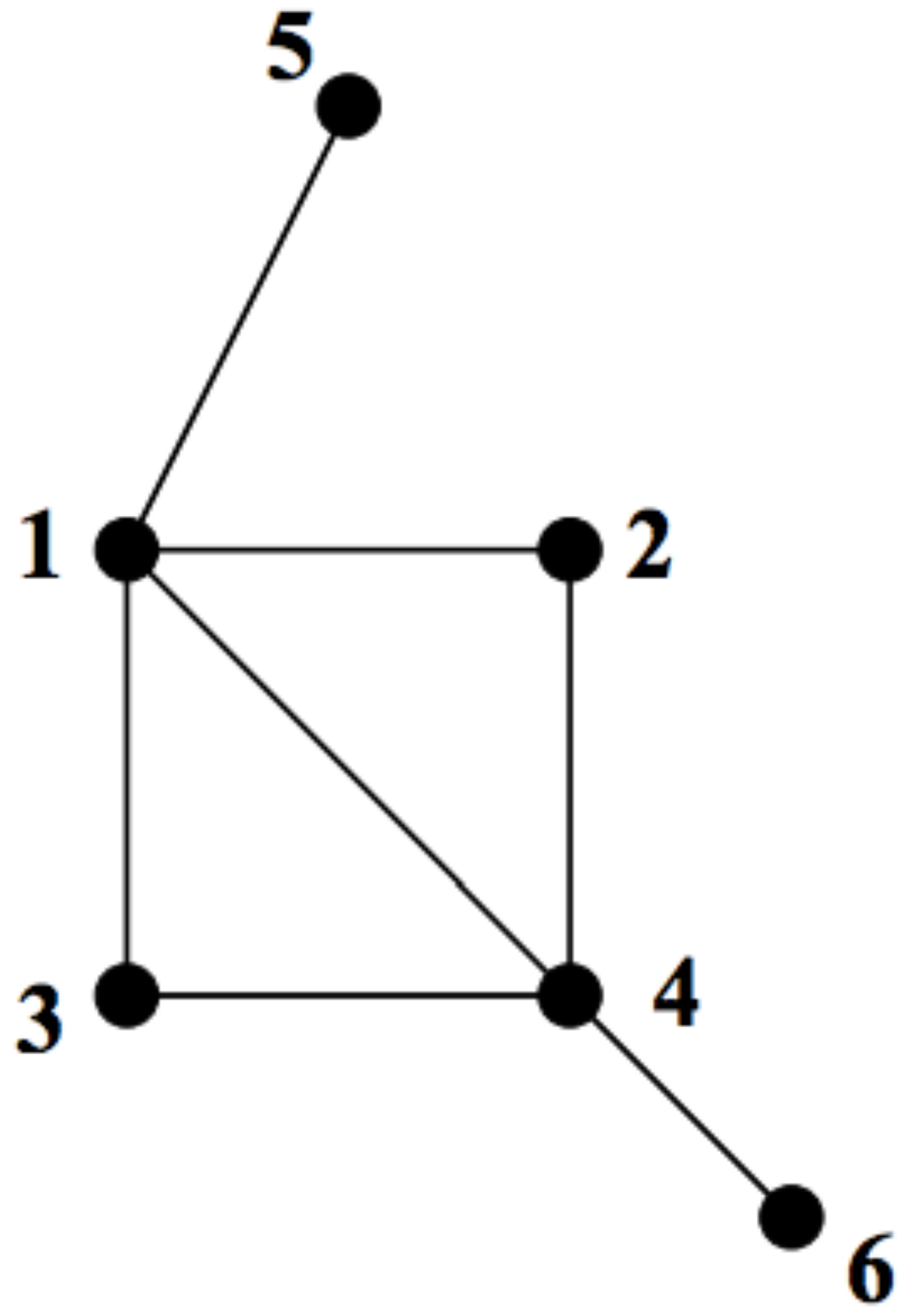
if there's an edge (v,w) such that w is not already in V:
    add the edge to E
    v = w (i.e., make w the new v), repeat
else:
    if v == v_1 : done (have made it back up to root)
    else: v = parent of v (make parent of v the new v), and repeat

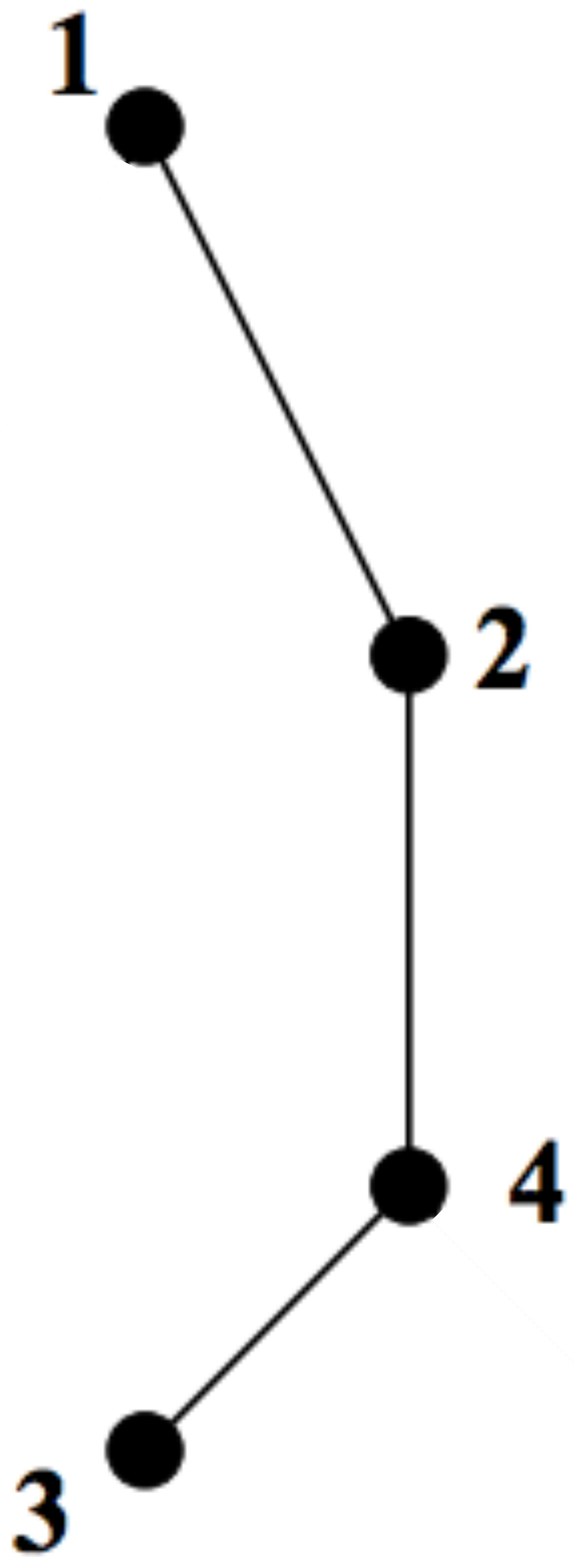
DFS



if there's an edge (v,w) such that w is not already in V:
    add the edge to E
    v = w (i.e., make w the new v), repeat
else:
    if v == v_1 : done (have made it back up to root)
    else: v = parent of v (make parent of v the new v), and repeat

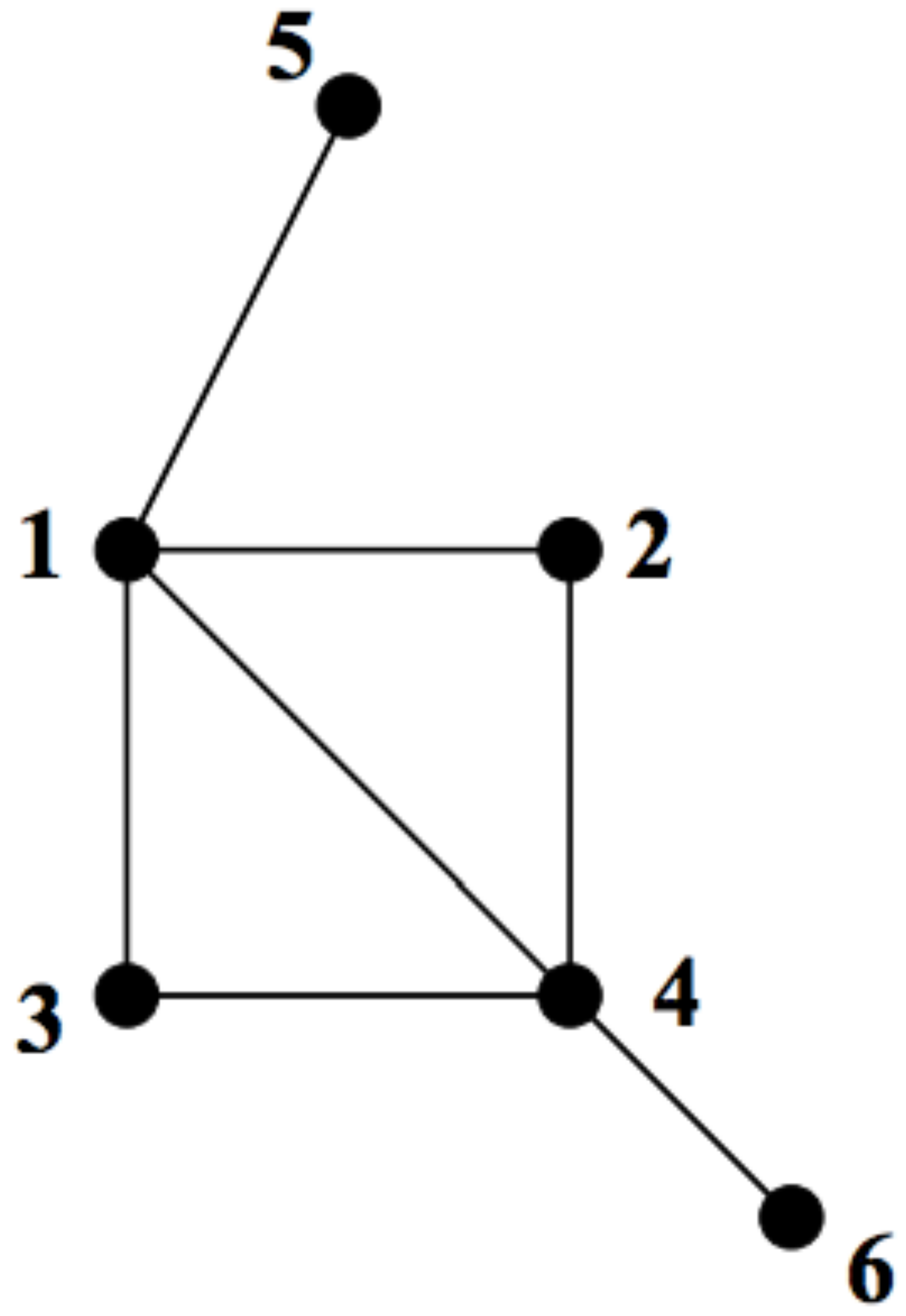
DFS

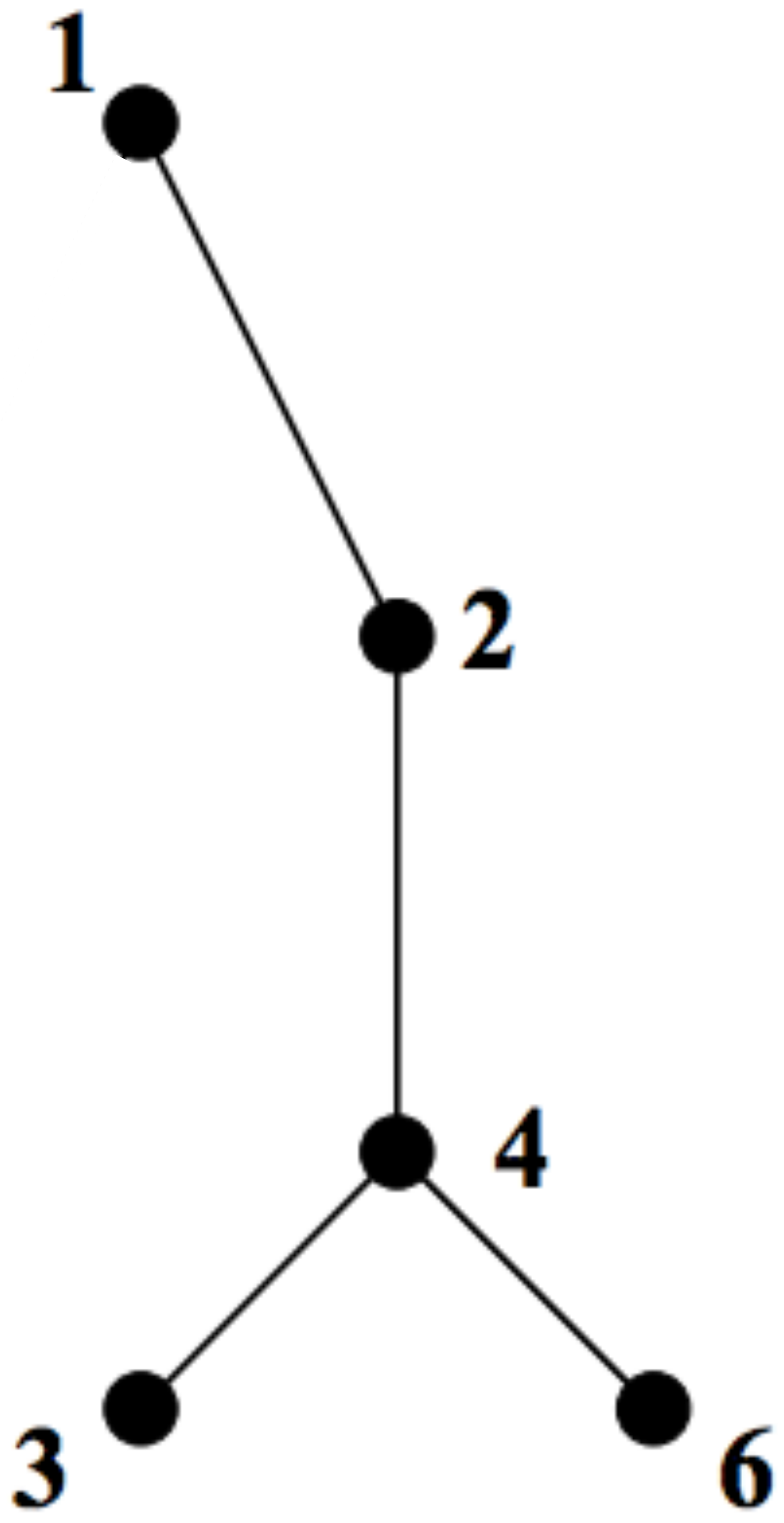




if there's an edge (v,w) such that w is not already in V:
  add the edge to E
  v = w (i.e., make w the new v), repeat
else:
  if v == v_1 : done (have made it back up to root)
  else: v = parent of v (make parent of v the new v), and repeat

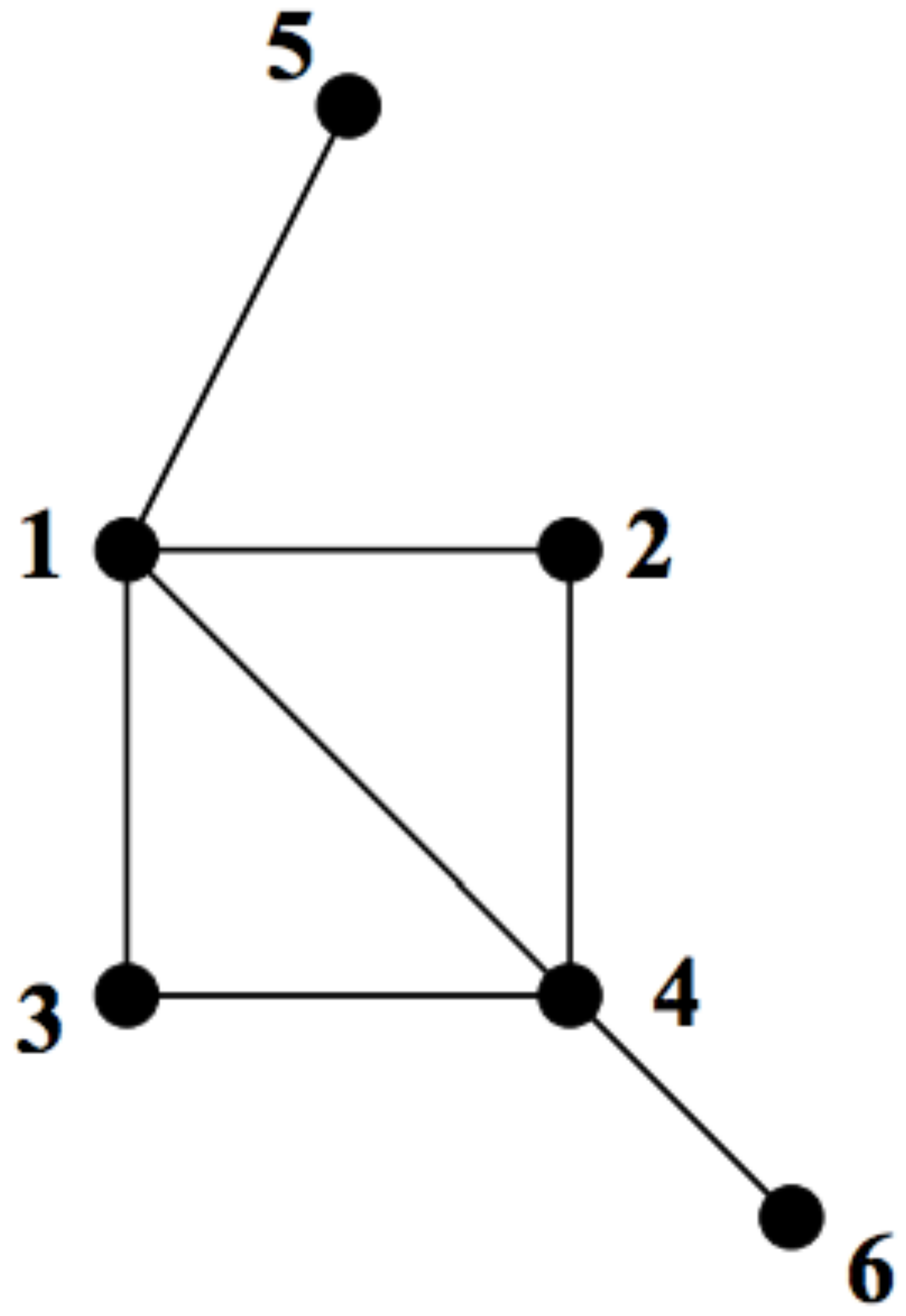
DFS

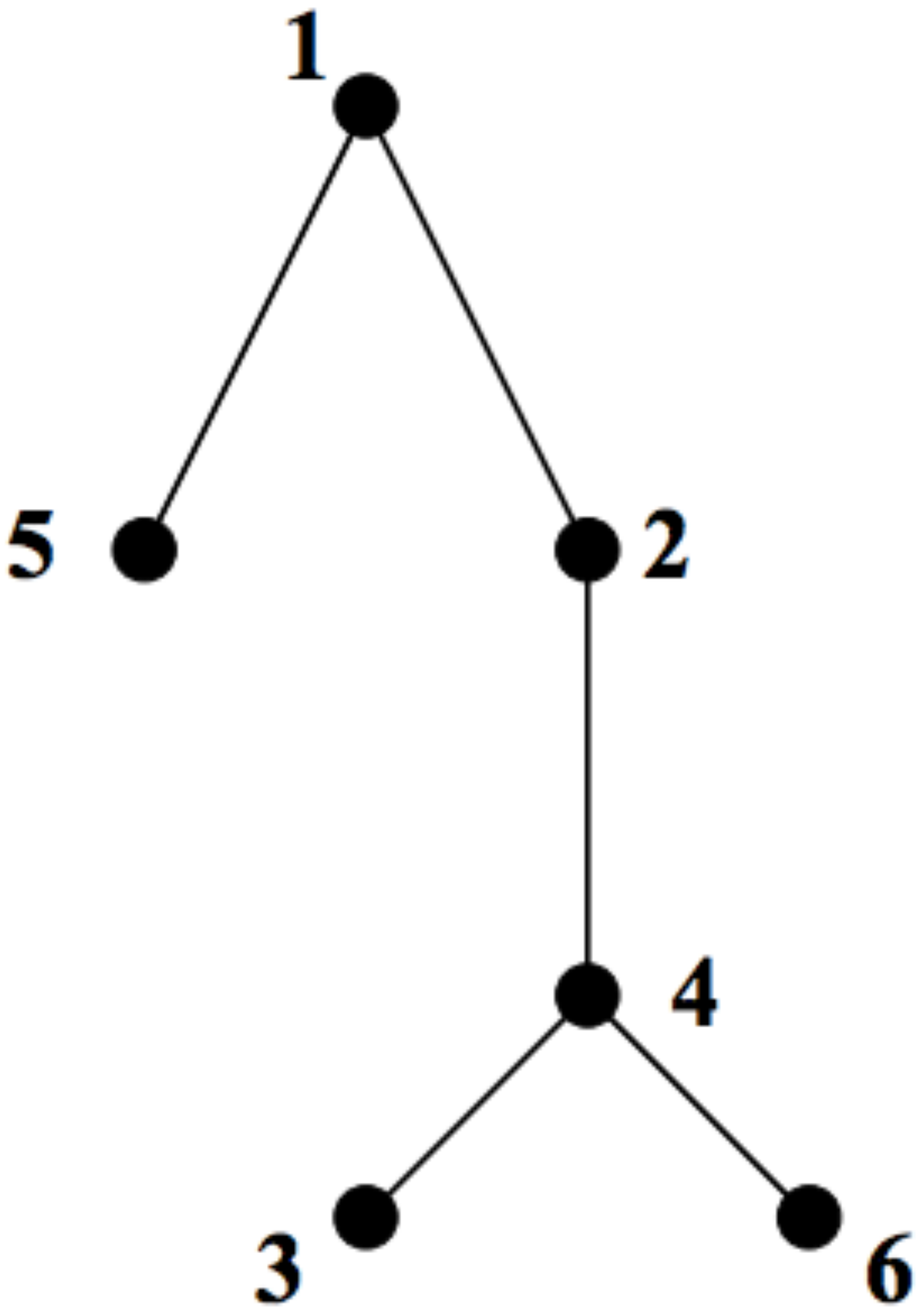




if there's an edge (v,w) such that w is not already in V:
    add the edge to E
    v = w (i.e., make w the new v), repeat
else:
    if v == v_1 : done (have made it back up to root)
    else: v = parent of v (make parent of v the new v), and repeat

DFS





if there's an edge (v,w) such that w is not already in V:

    add the edge to E

    v = w (i.e., make w the new v), repeat

else:

    if v == v_1 : done (have made it back up to root)

    else: v = parent of v (make parent of v the new v), and repeat

BFS algorithm

Suppose original G has n vertices

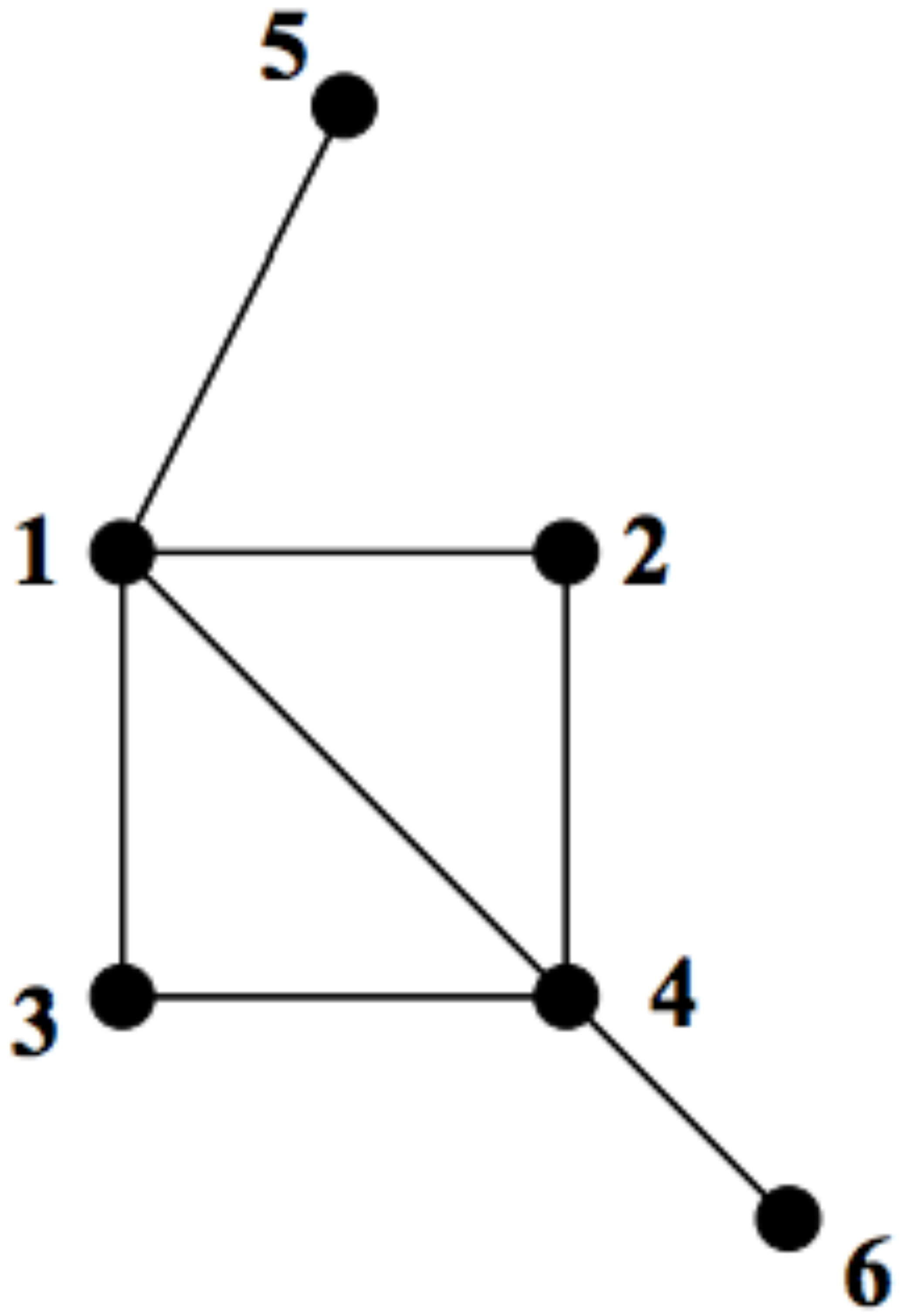
Initialize $T = (V, E)$:
$V = \{v_1\}$
$E = \{\}$

$v = v_1$

---

for all neighbors w of v not in V:
    add w to V
    add edges (v,w) to E
if $|E| == n - 1$: stop
else: v = next element of V, and repeat

BFS



for all neighbors w of v not in V:

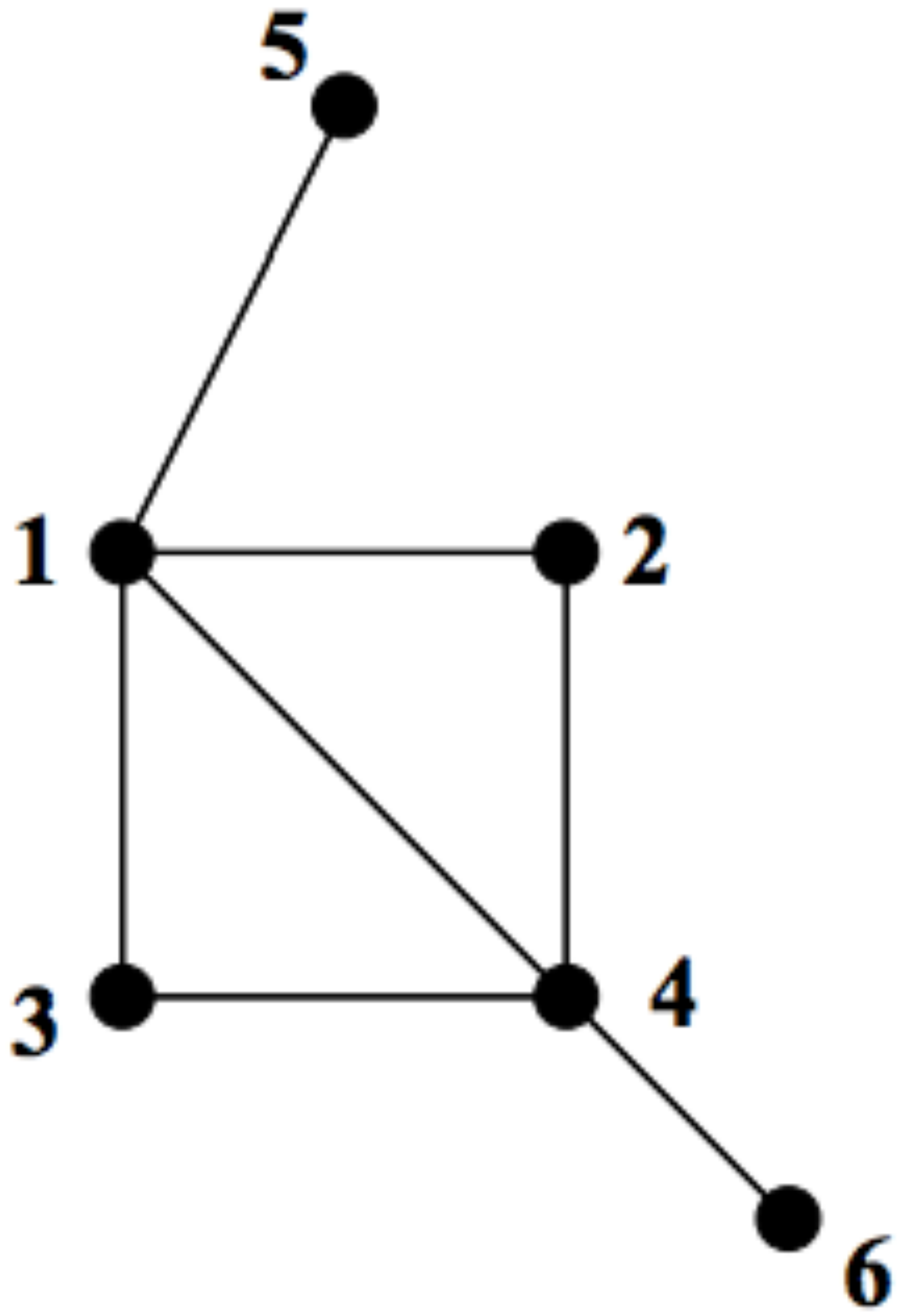
  add w to V

  add edges (v,w) to E

if |E| = n - 1: stop

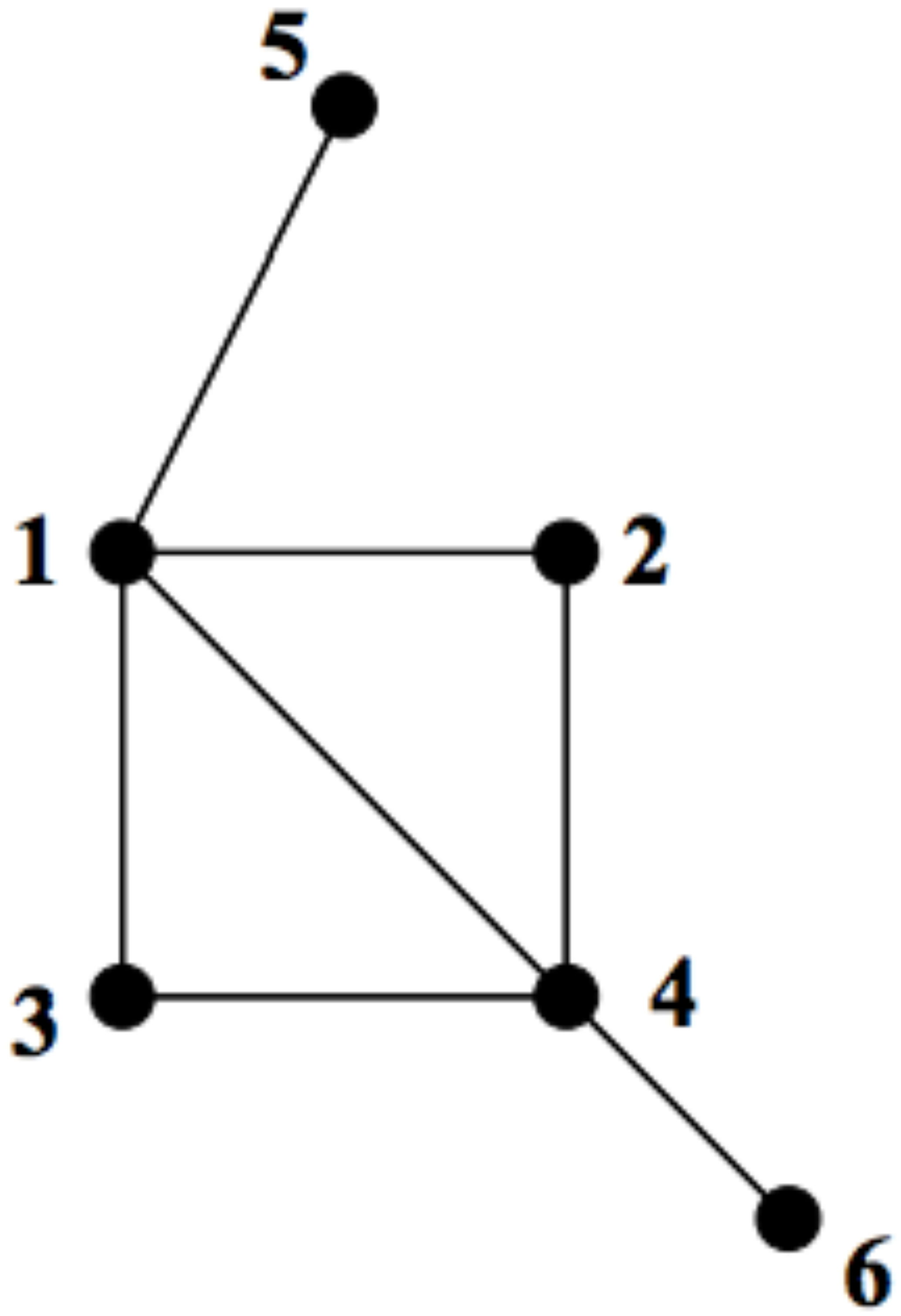
else: v = next element of V, and repeat

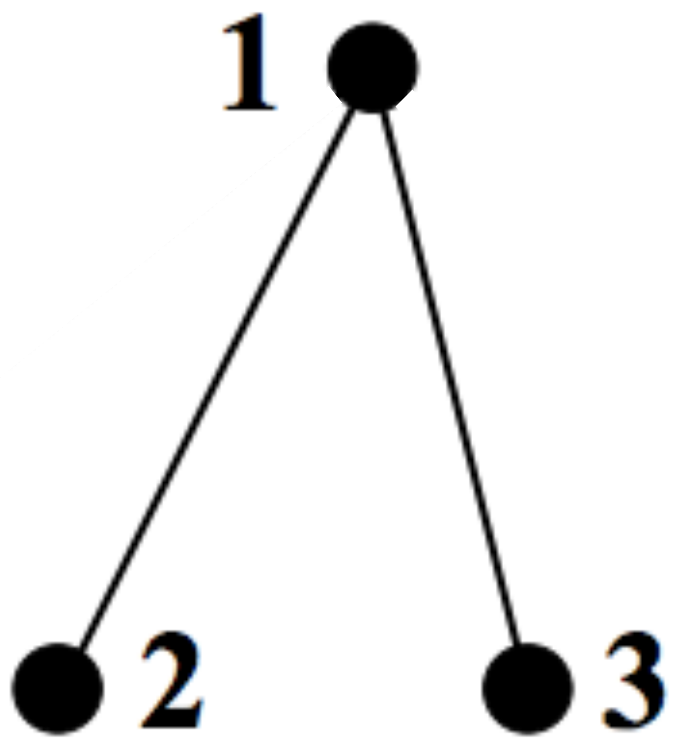
BFS



for all neighbors w of v not in V:
    add w to V
    add edges (v,w) to E
if |E| = n - 1: stop
else: v = next element of V, and repeat

BFS





for all neighbors w of v not in V:

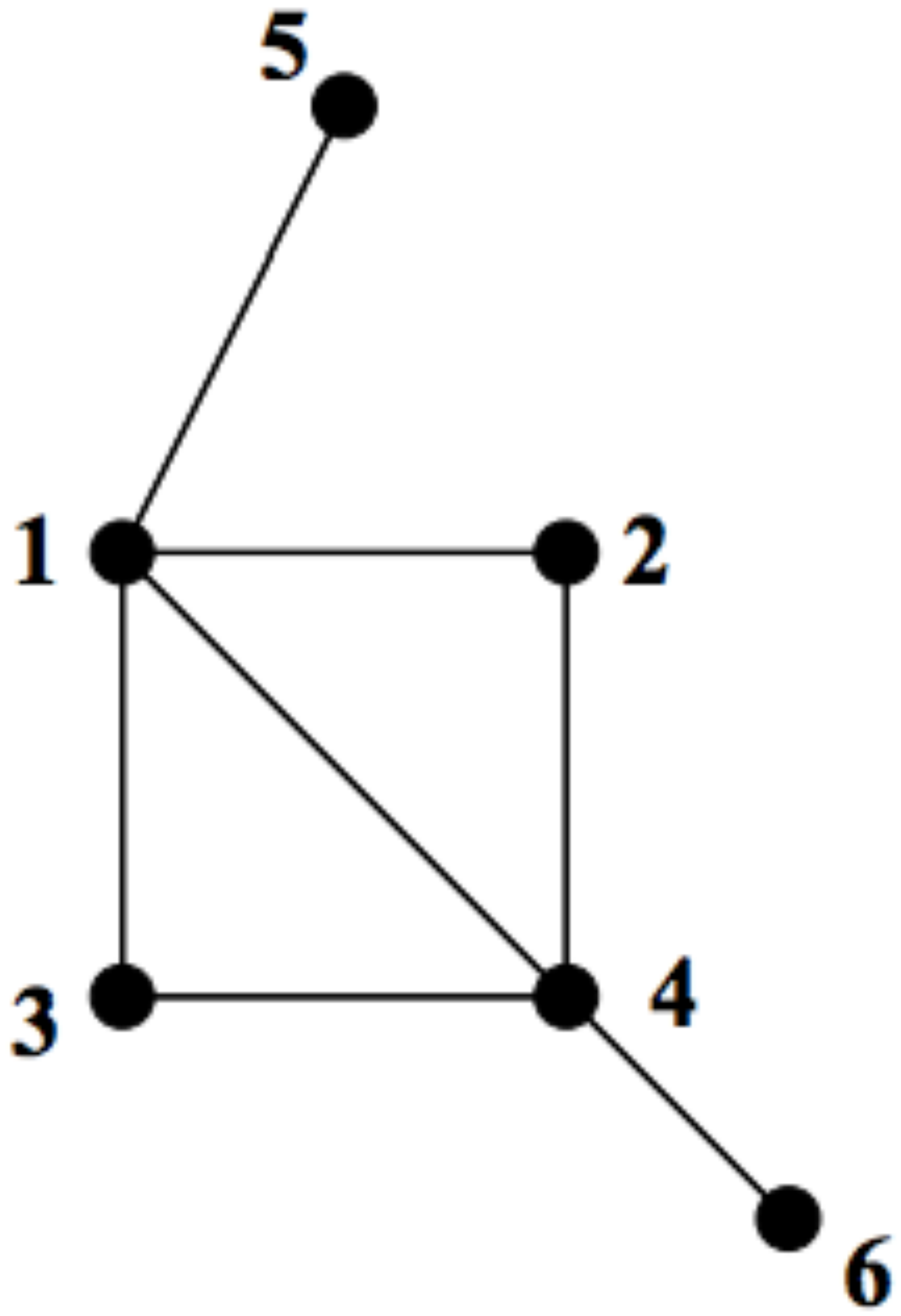
    add w to V

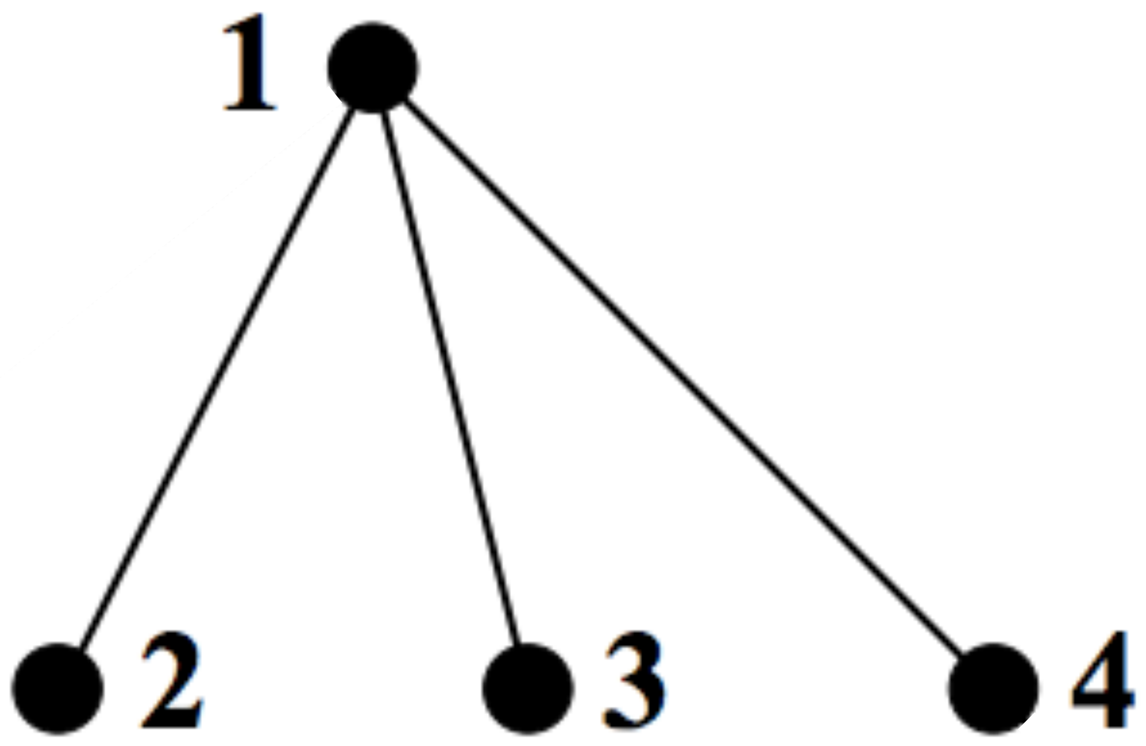
    add edges (v,w) to E

if |E| = n - 1: stop

else: v = next element of V, and repeat

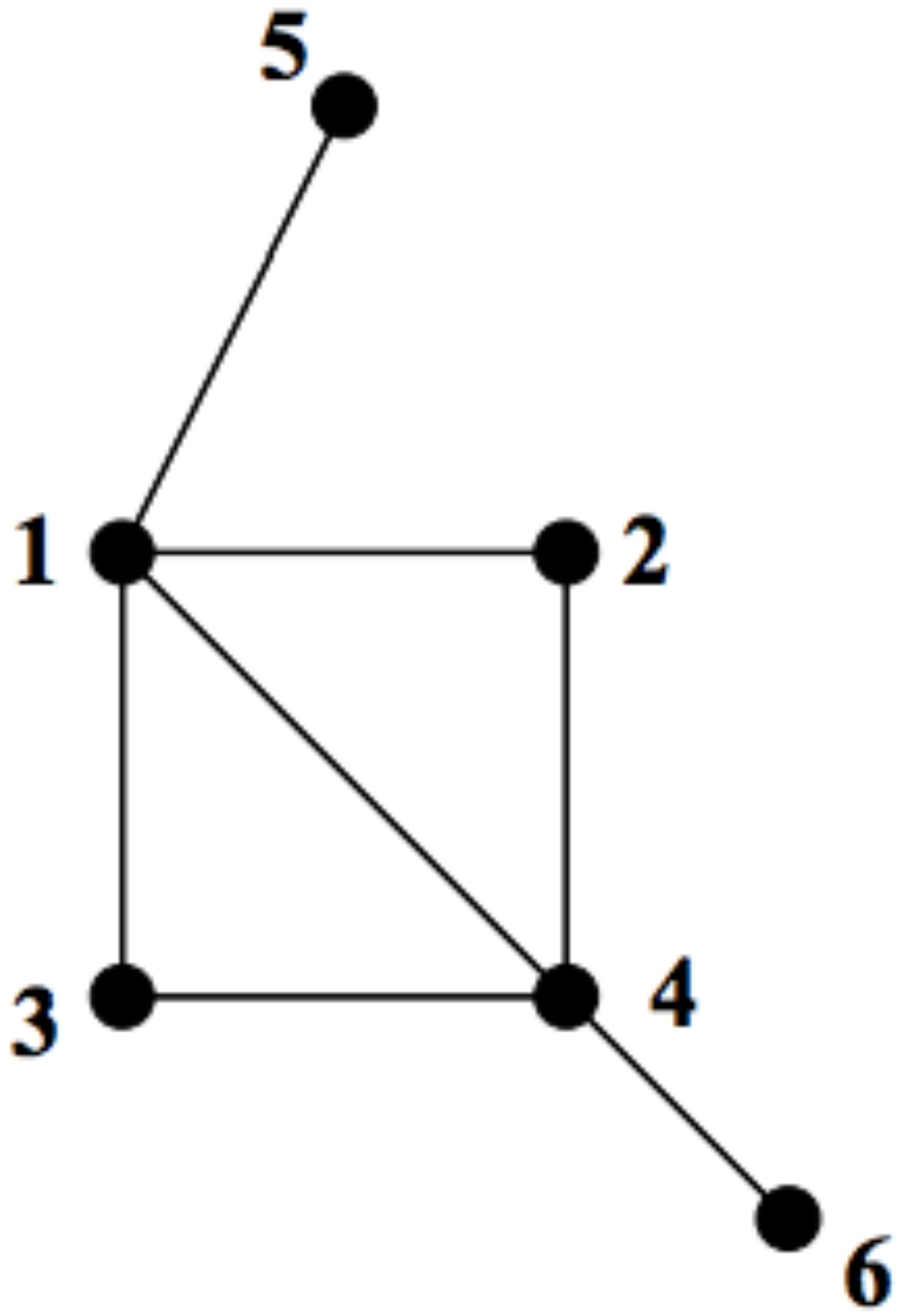
BFS

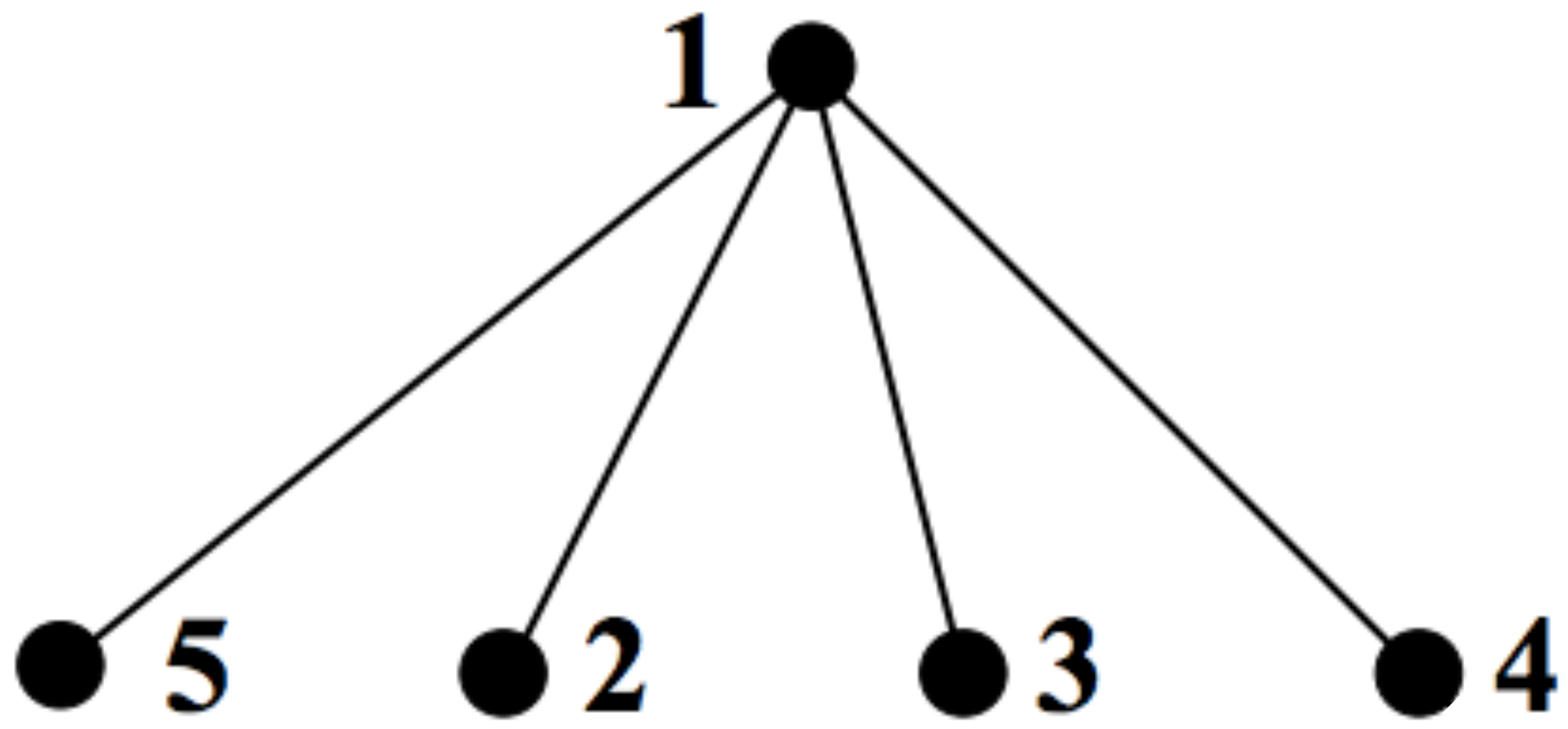




for all neighbors w of v not in V:
    add w to V
    add edges (v,w) to E
if |E| = n - 1: stop
else: v = next element of V, and repeat

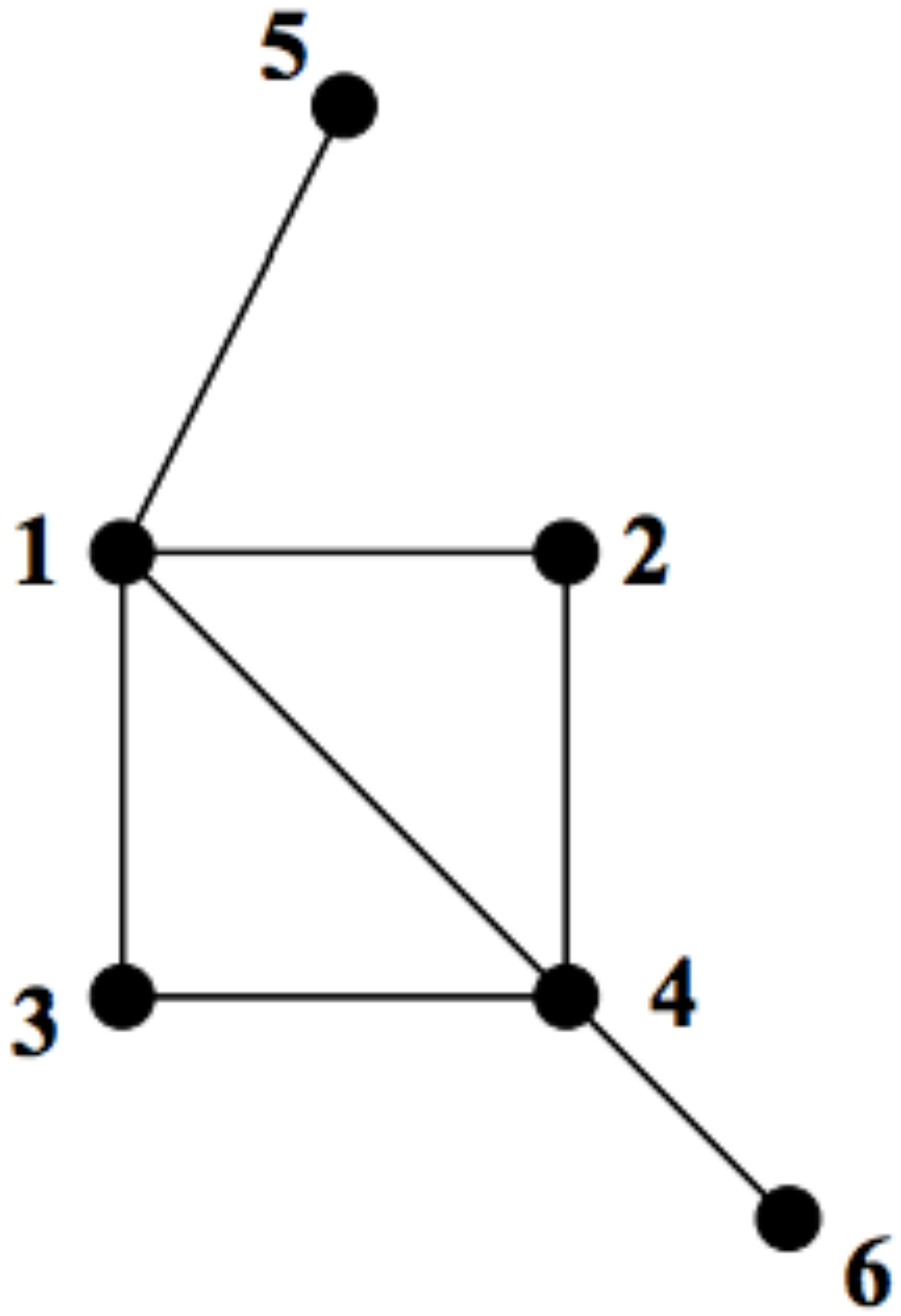
BFS

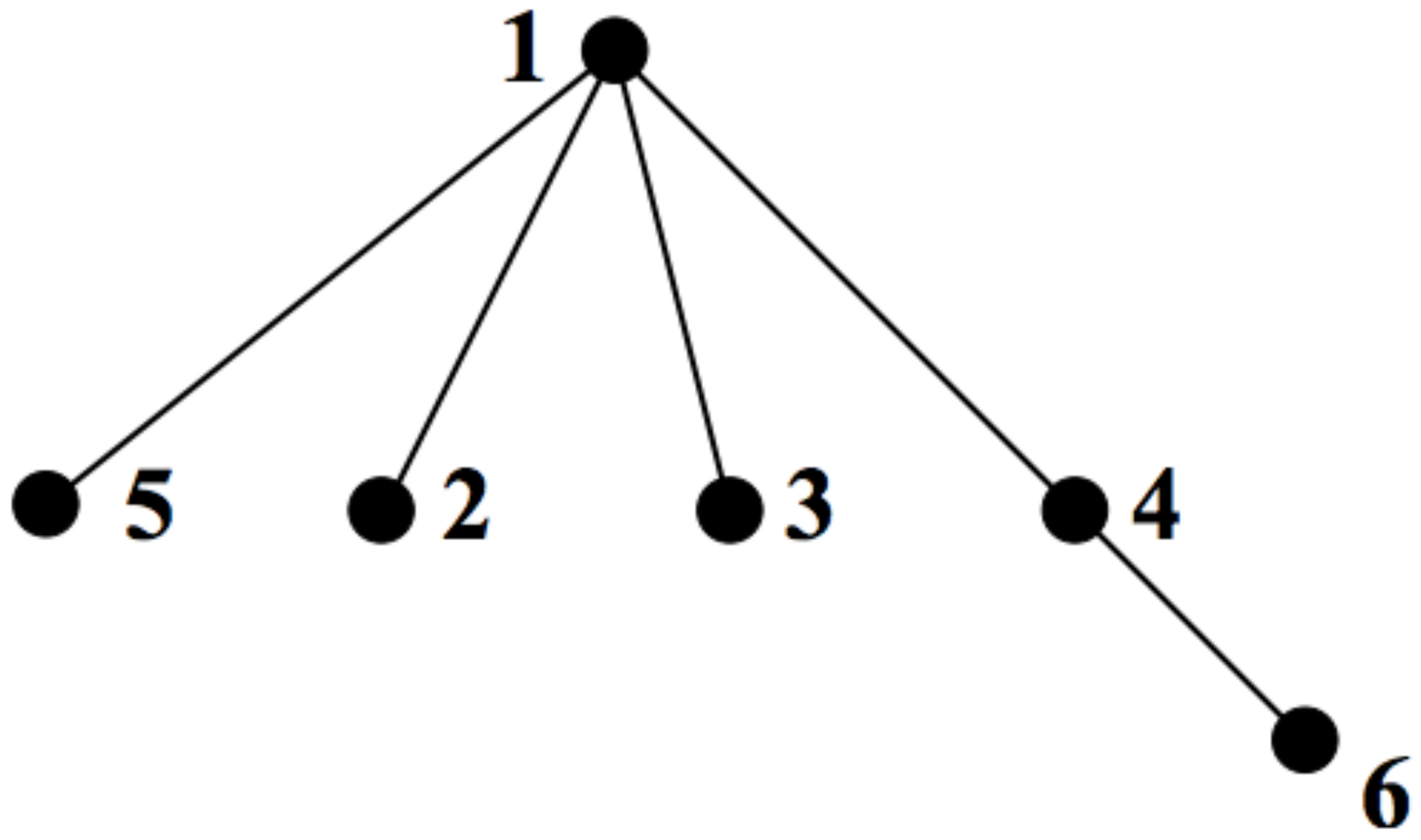




for all neighbors w of v not in V:
    add w to V
    add edges (v,w) to E
if |E| = n - 1: stop
else: v = next element of V, and repeat

BFS





for all neighbors w of v not in V:
    add w to V
    add edges (v,w) to E
if |E| = n - 1: stop
else: v = next element of V, and repeat

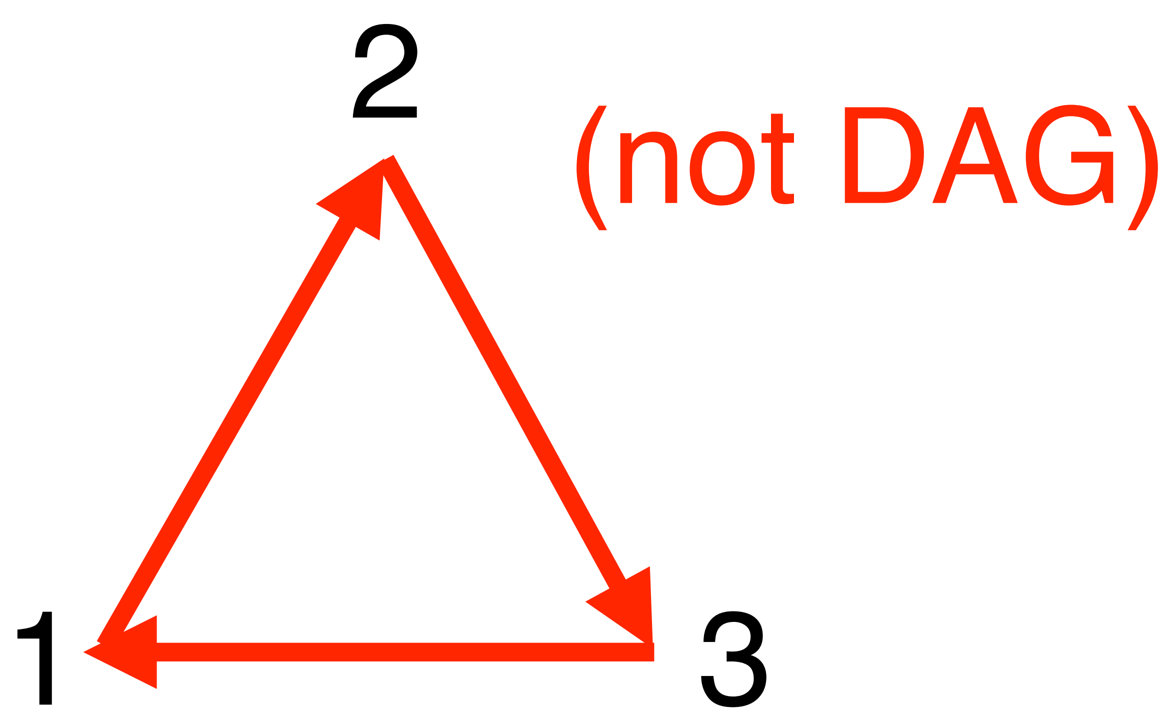
Suppose we have a directed graph whose vertices represent tasks,
and edges represent dependence:

An edge (i, j) means that task j cannot be accomplished until task i is complete

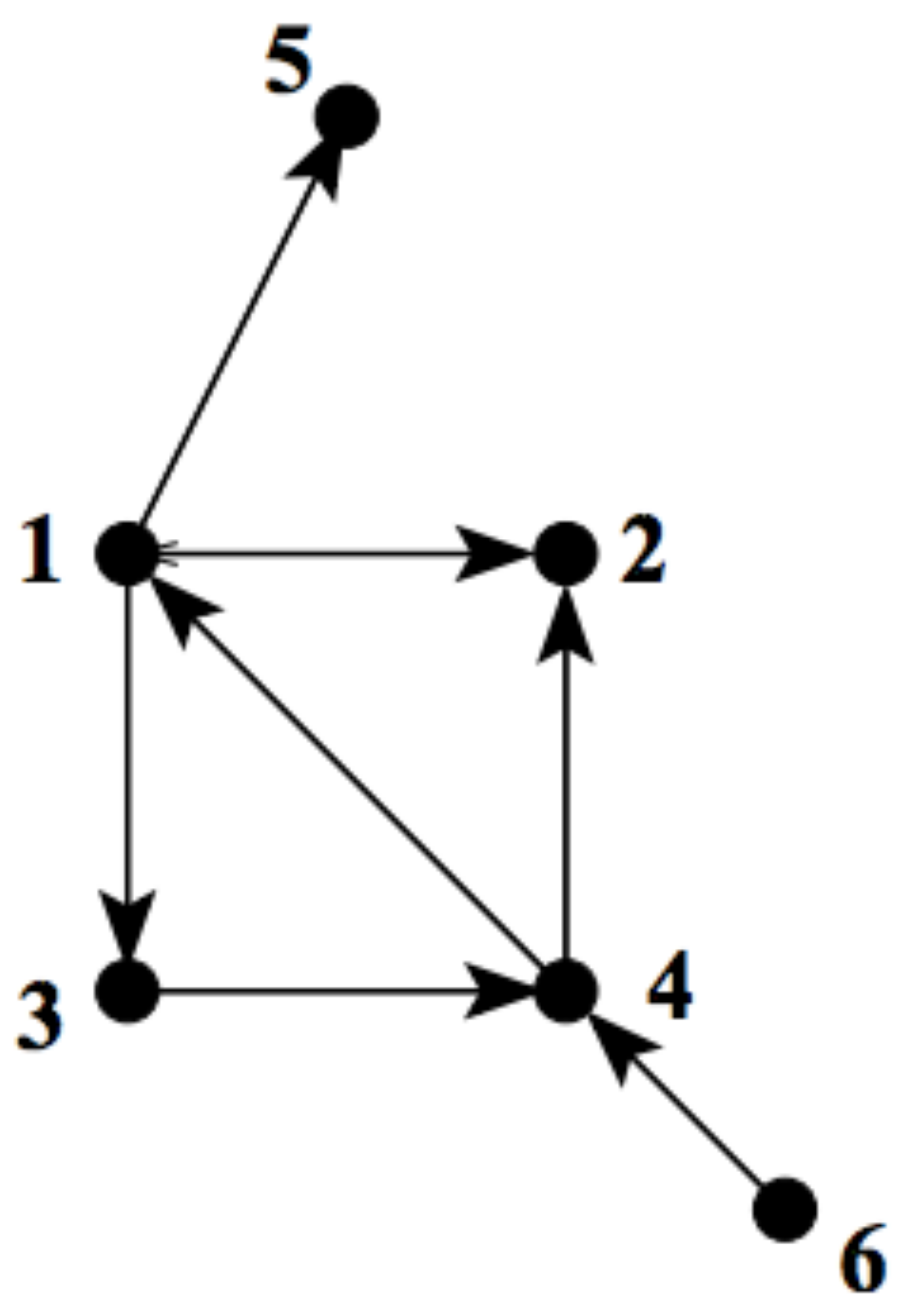
Given such a graph, determine an order to complete all tasks:
called a total order for a directed graph.

Is such an order is possible?

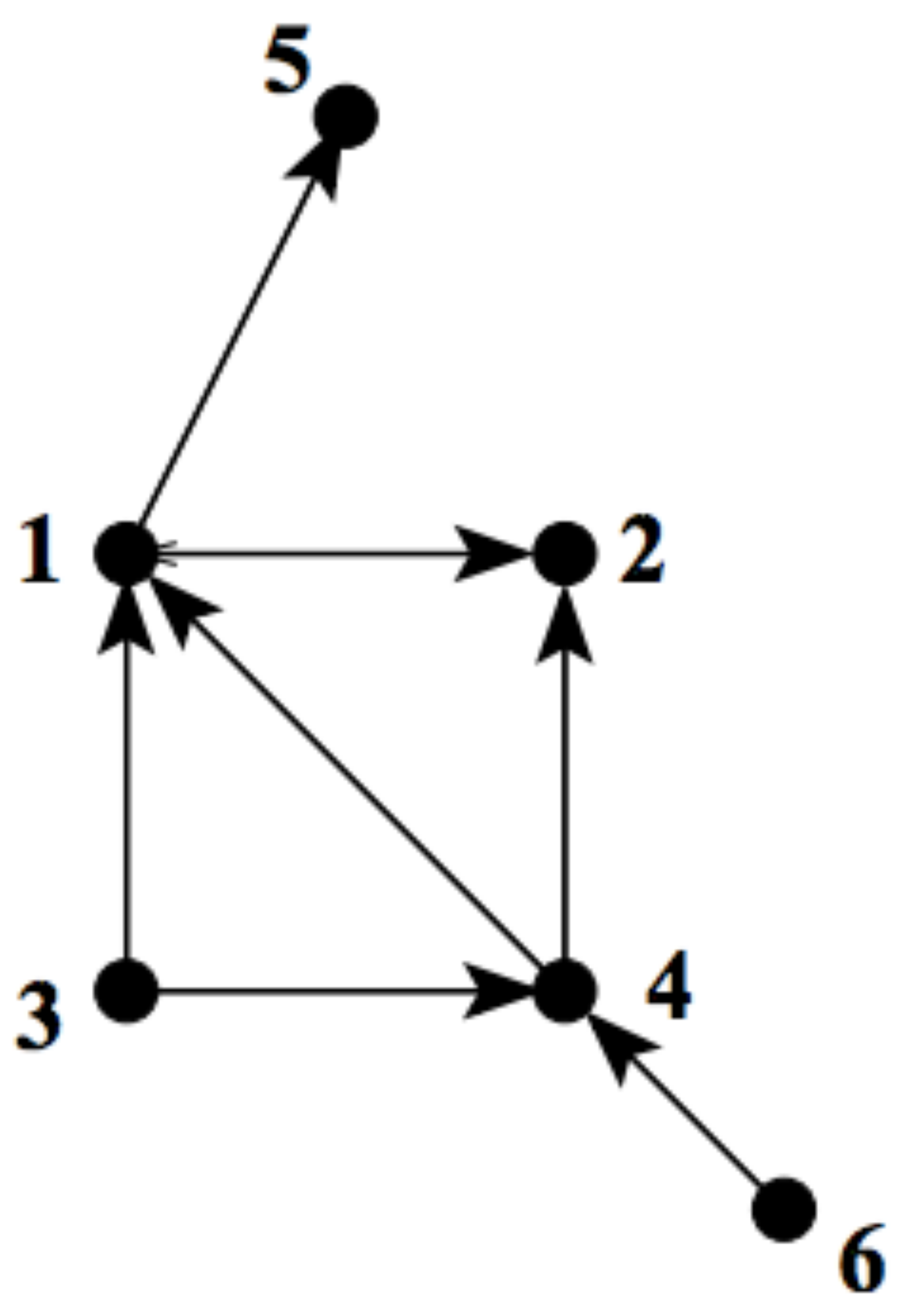
If graph contains a directed cycle, no such order:



A directed graph with no directed cycle is called an acyclic graph, or DAG



(not DAG)



DAG

A directed graph has a total order if and only if it is acyclic.

Suppose we have an acyclic graph.

Algorithm for finding a total order called Topological Sort:

Let $i = 1$ and G be an acyclic graph on n vertices.

Find a vertex v_i such that $\text{outdeg}(v_i) = 0$.

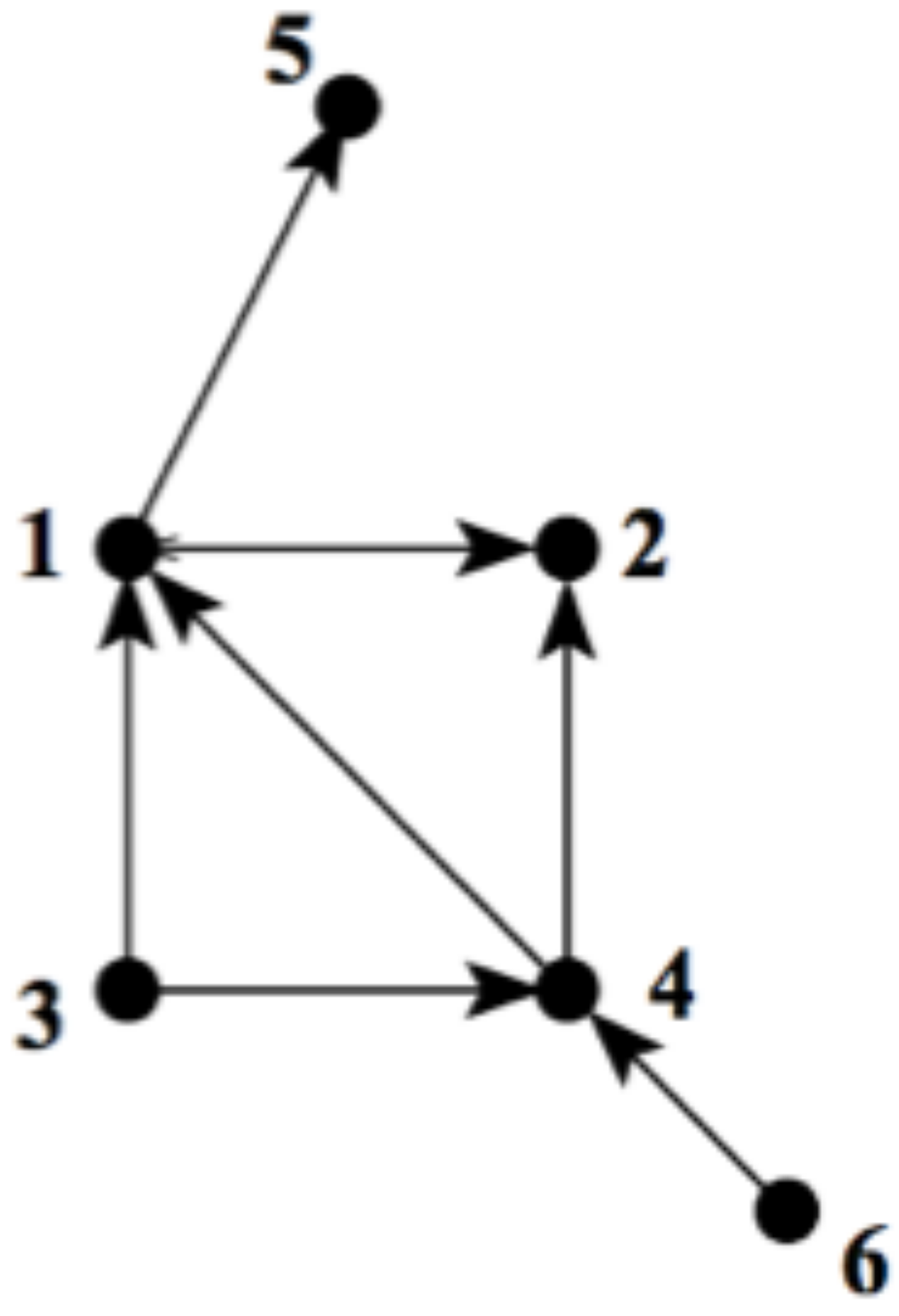
(Nothing depends on it … do it last.)

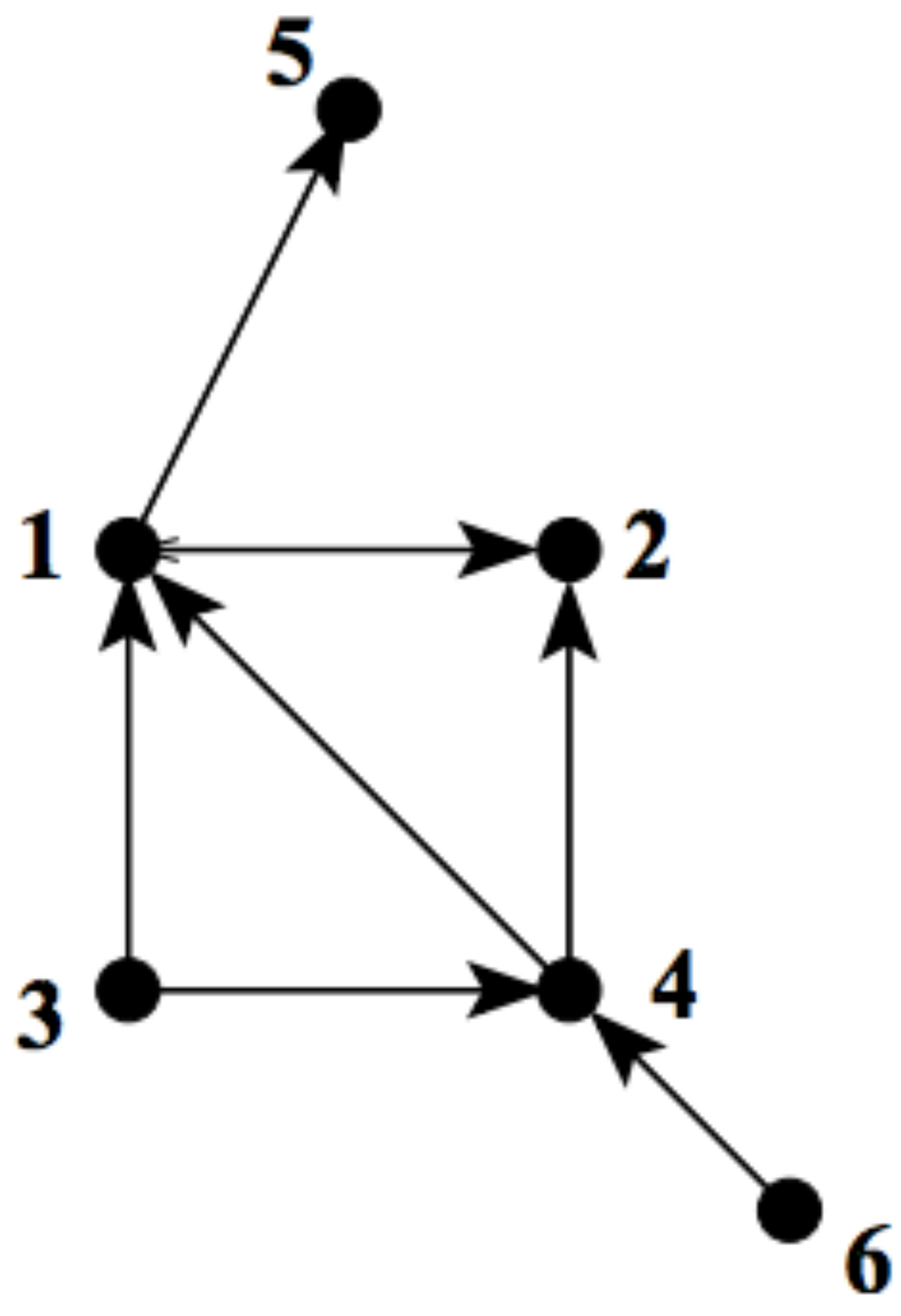
If $i = n$ (last vertex): then stop

$v_n < v_{n-1} < . . . v_2 < v_1$ is a total order.

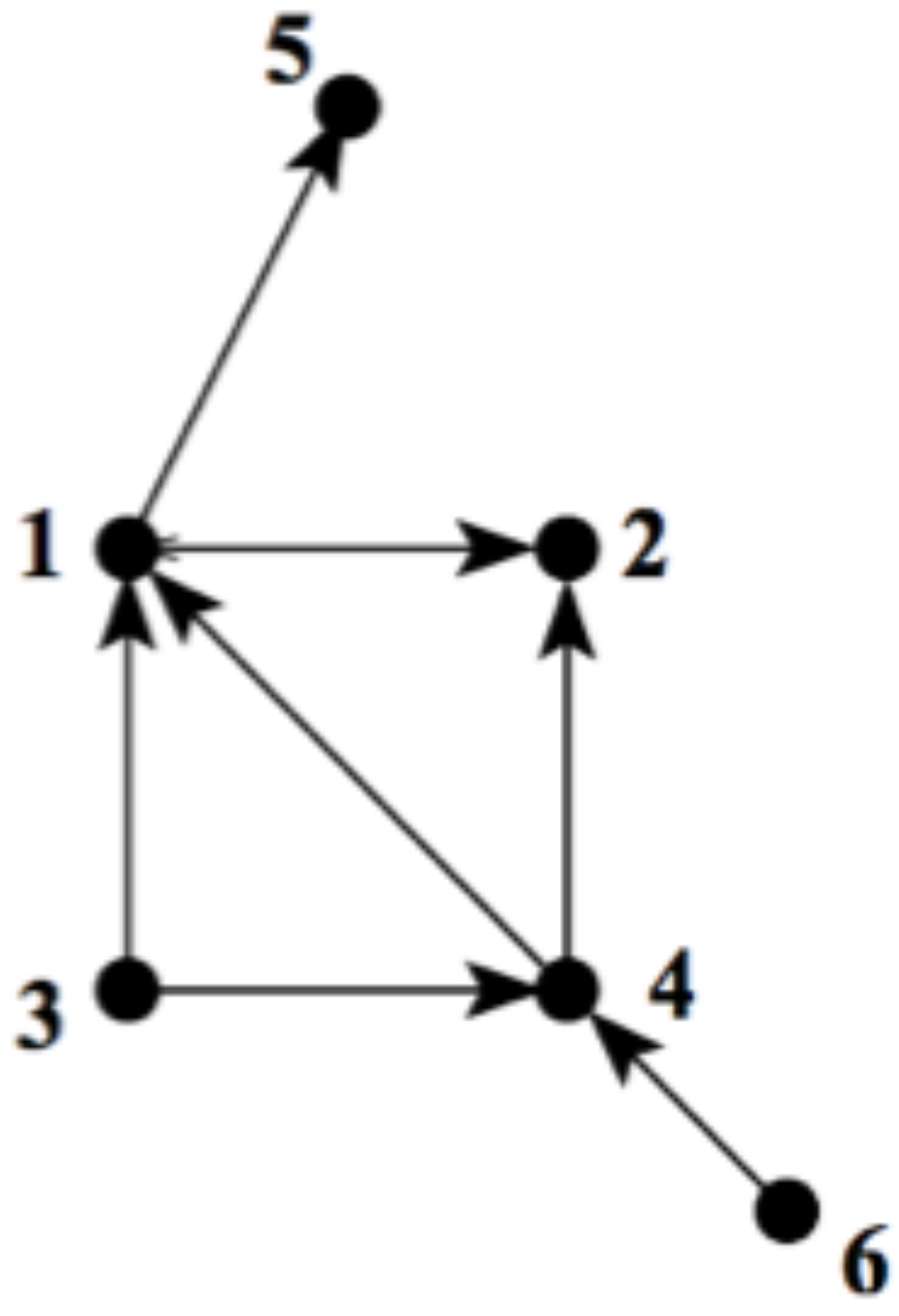
else: remove v_i from G

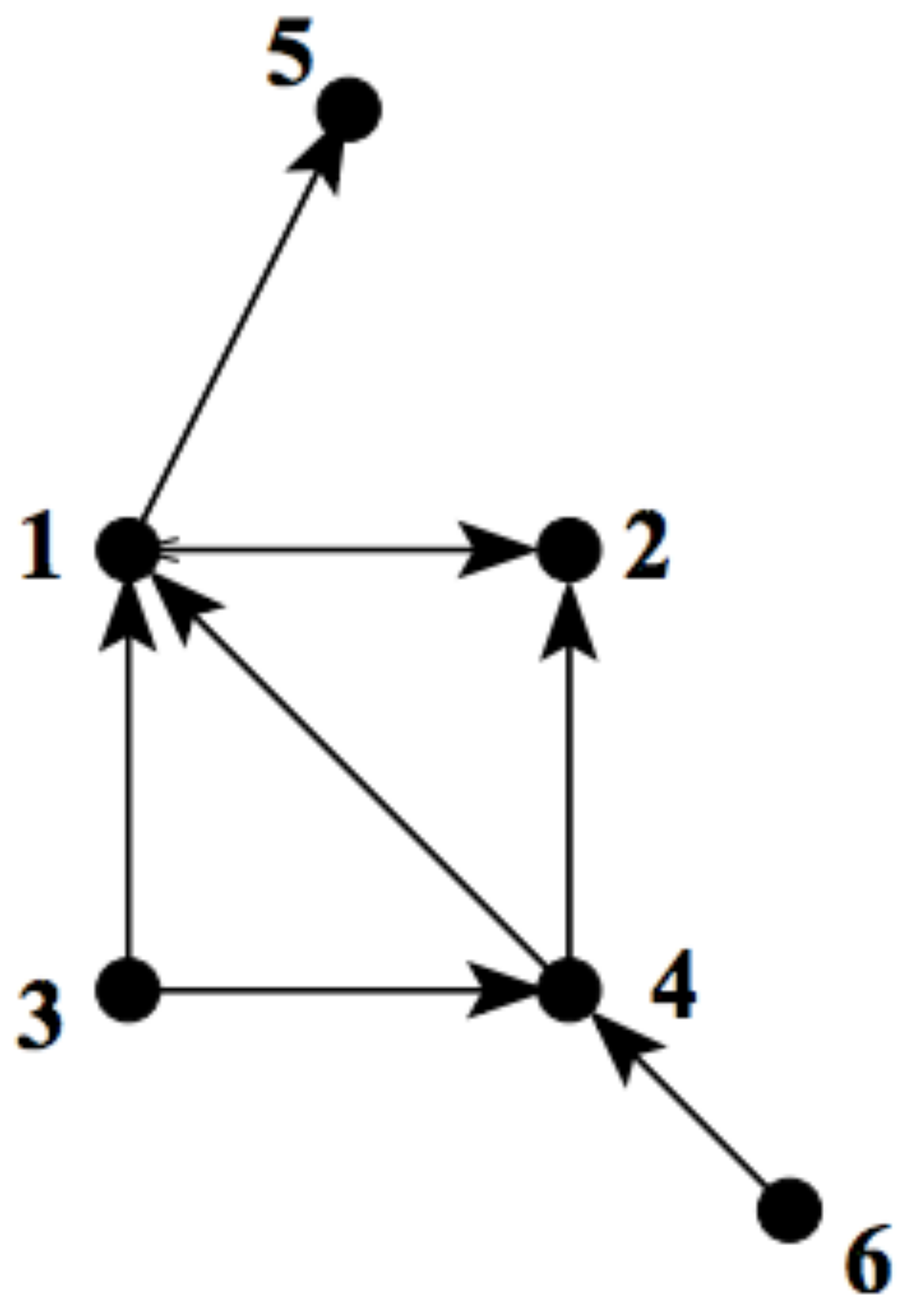
$i = i + 1$, repeat





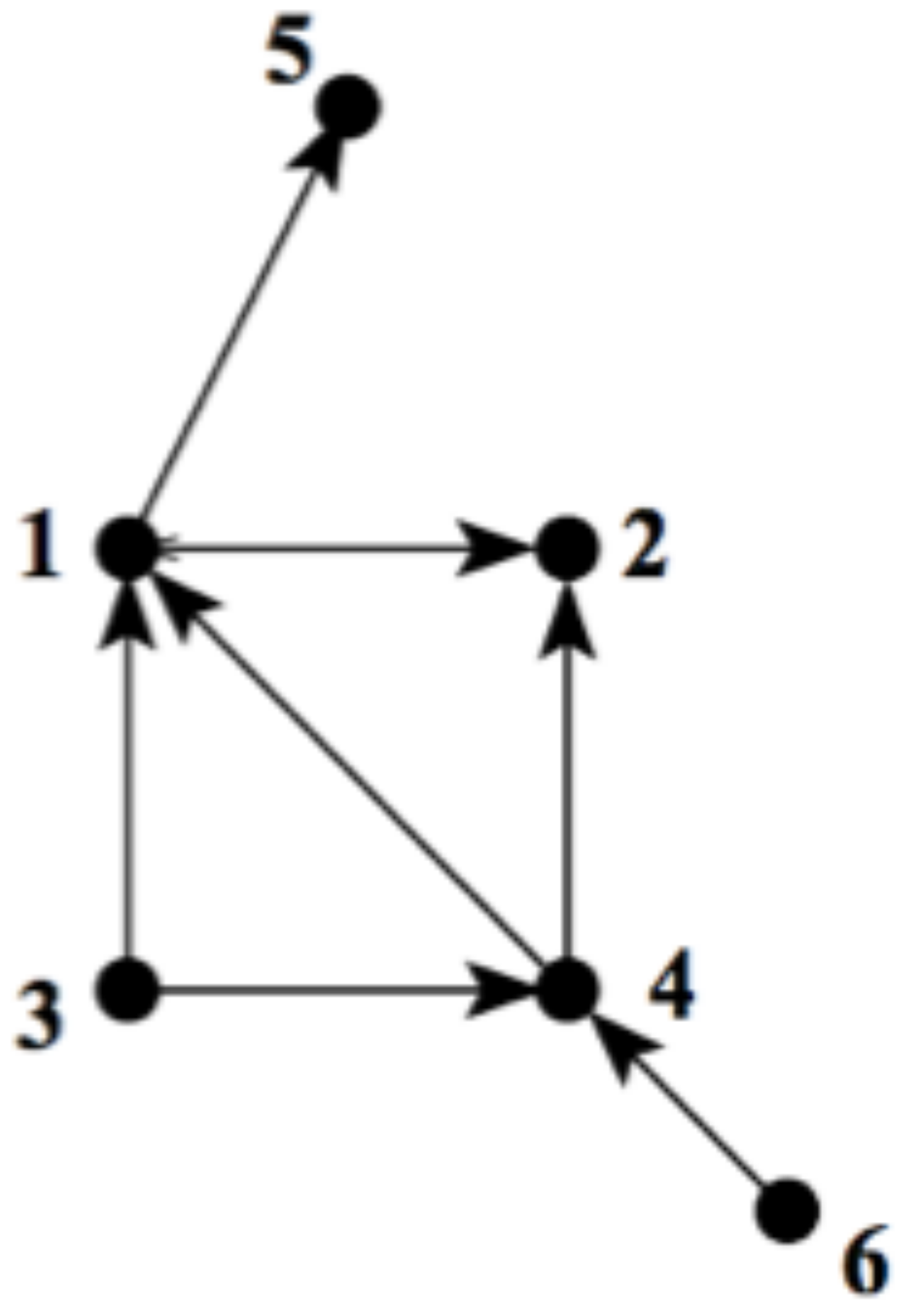
Example: one total ordering found by topological search is:

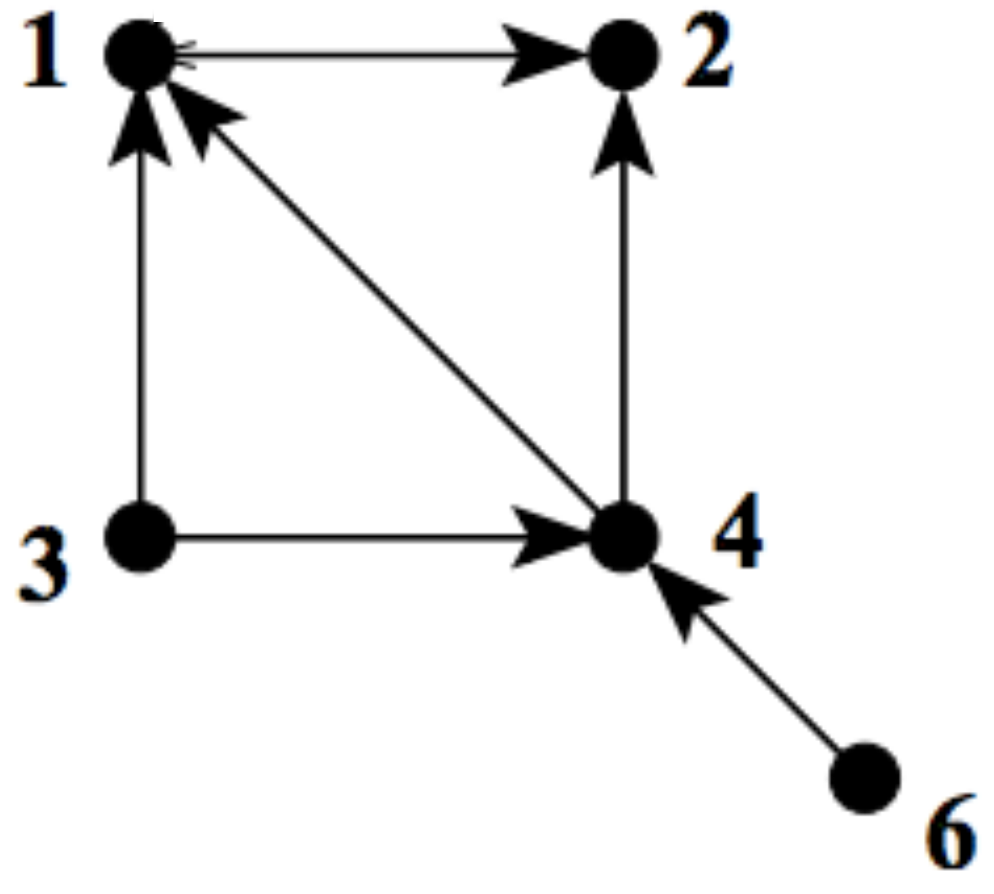




Example: one total ordering found by topological search is:

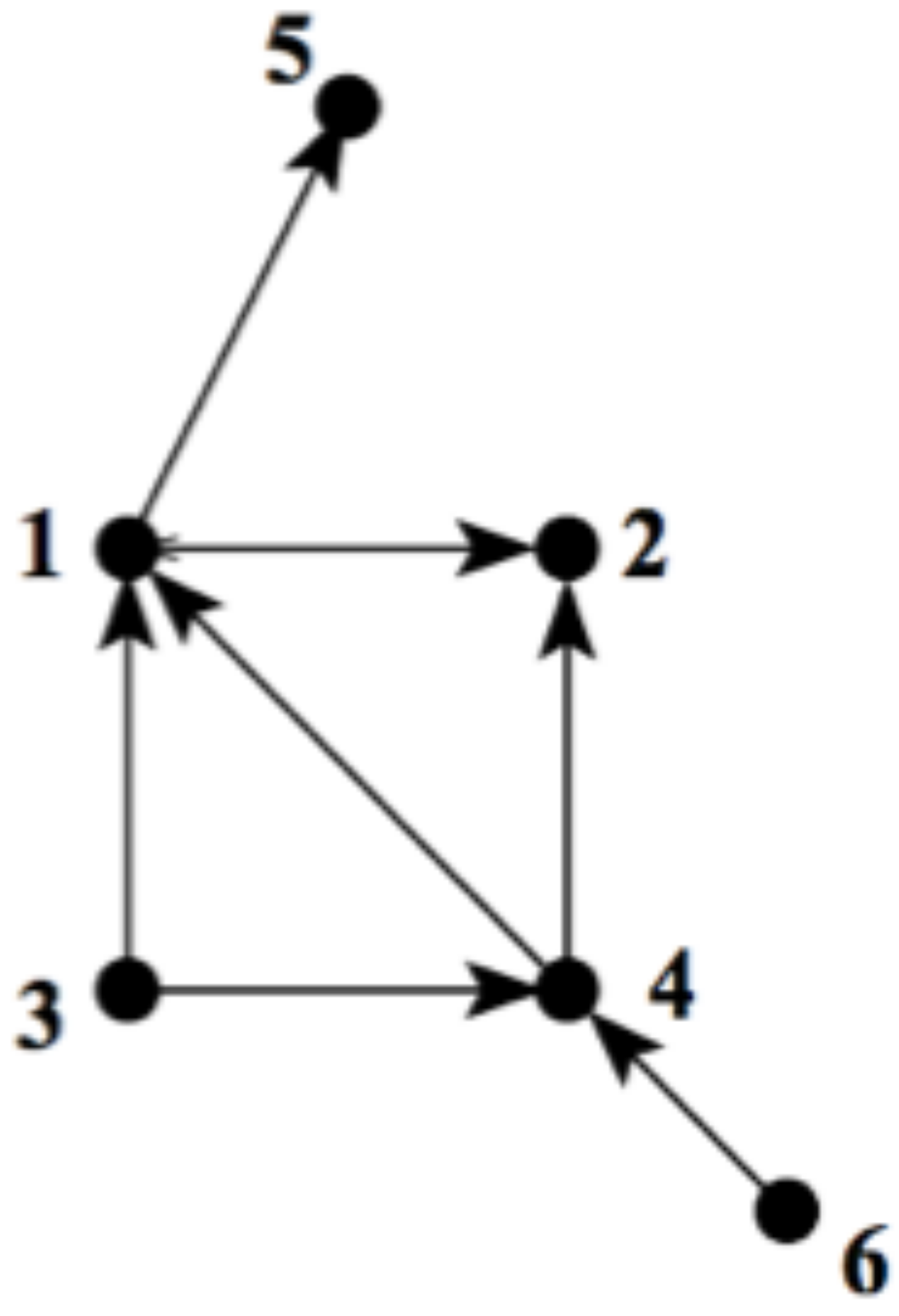
5

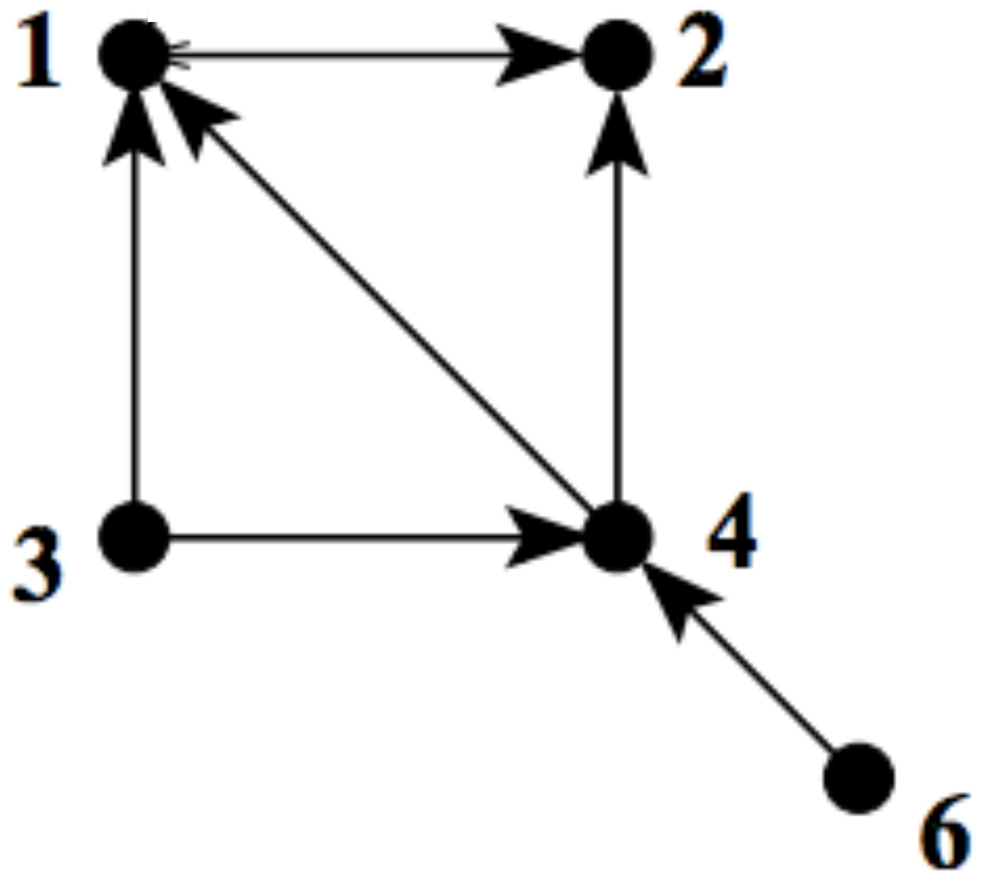




Example: one total ordering found by topological search is:

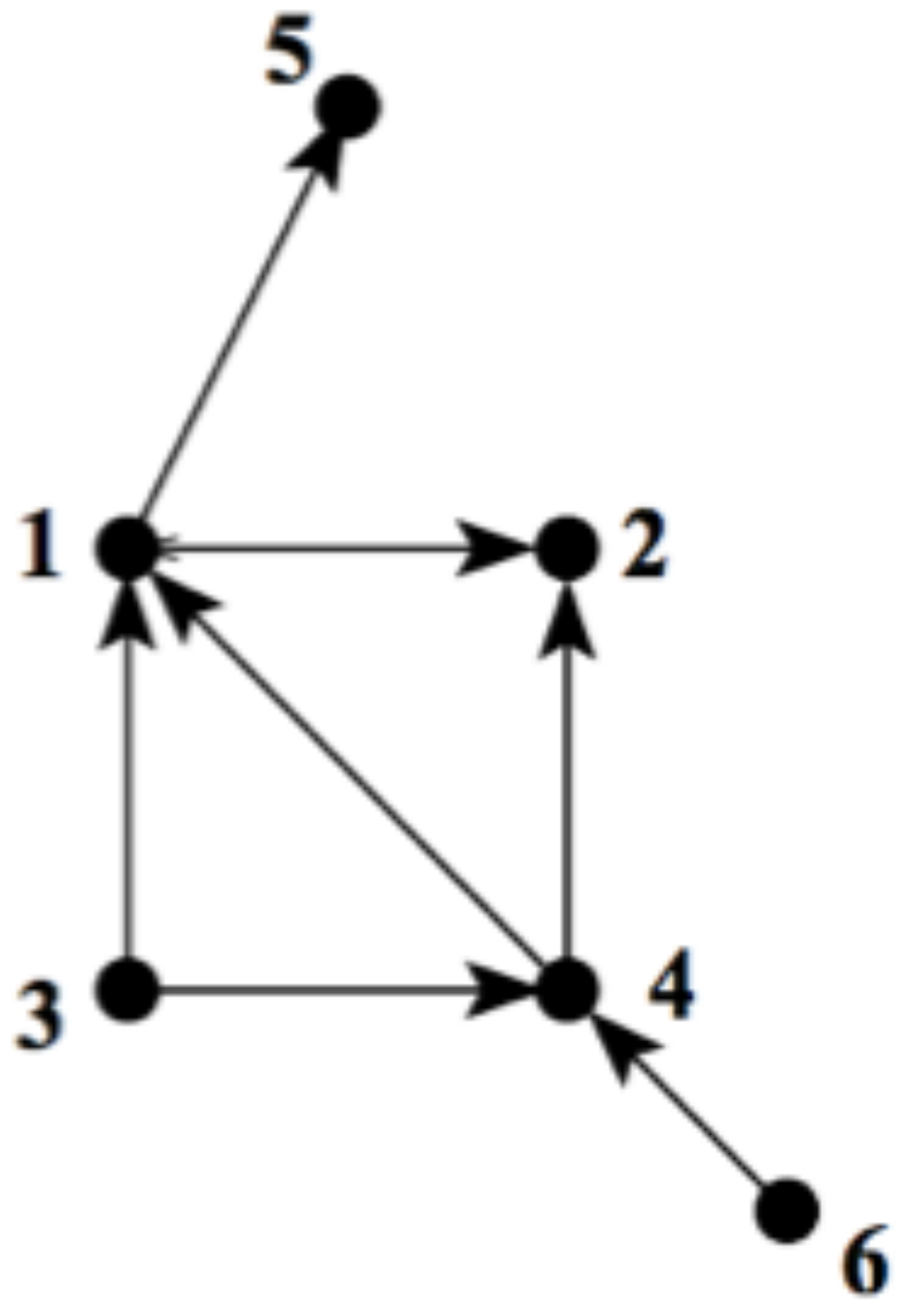
5

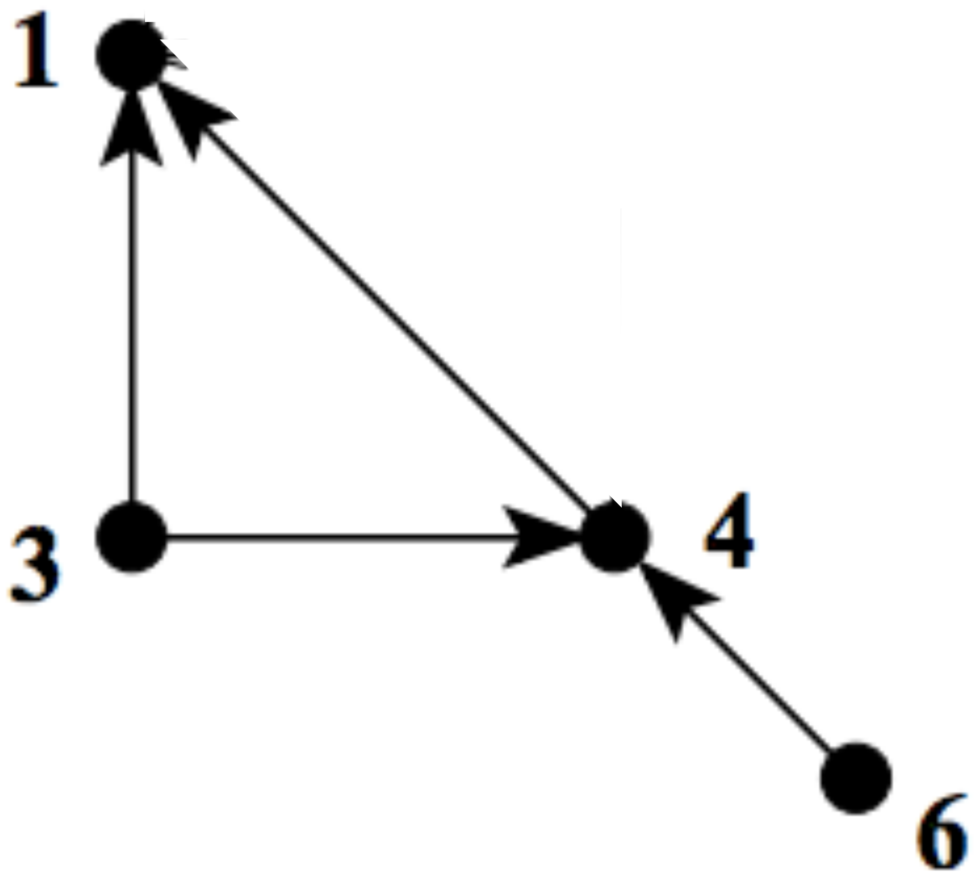




Example: one total ordering found by topological search is:

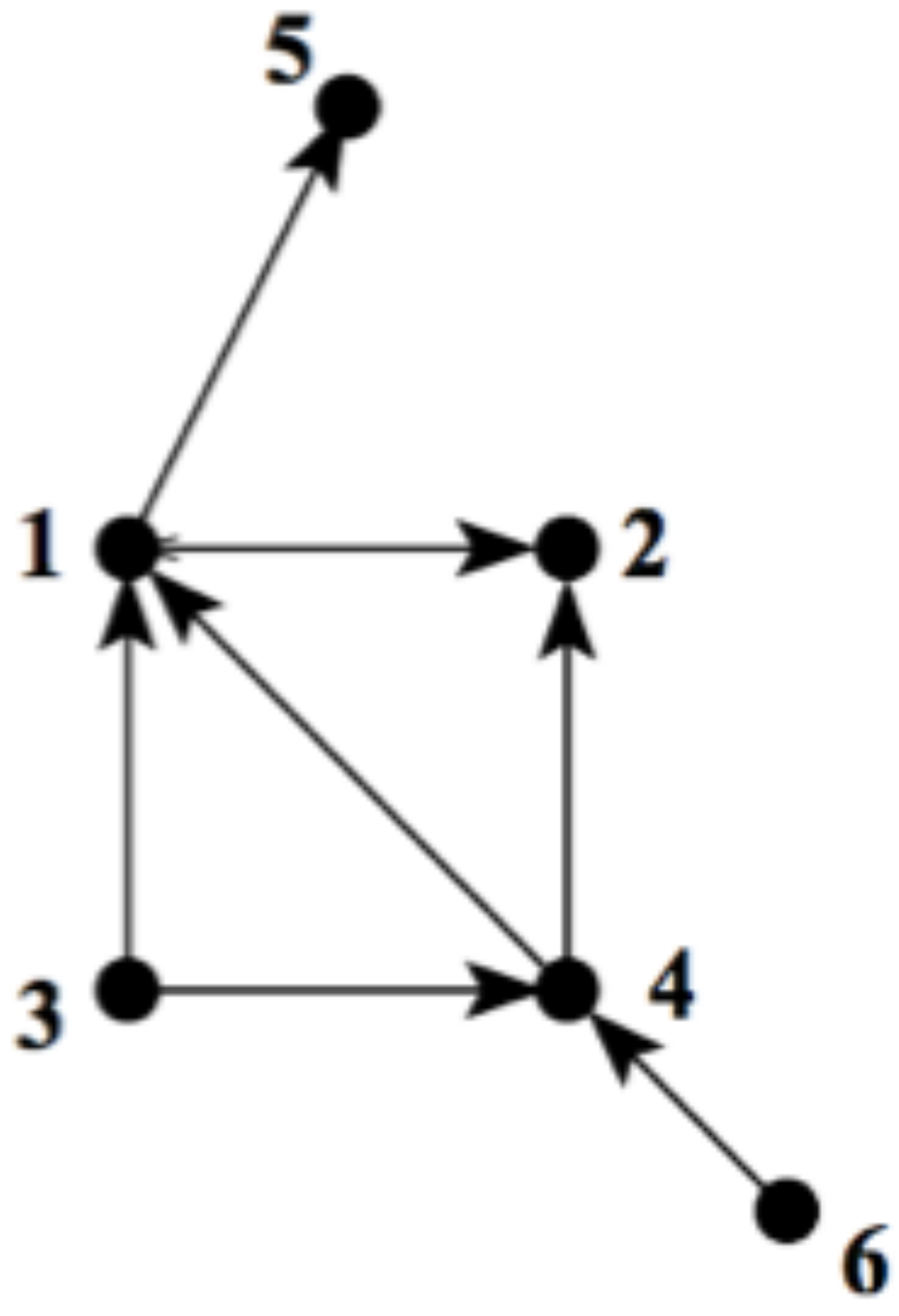
2 < 5

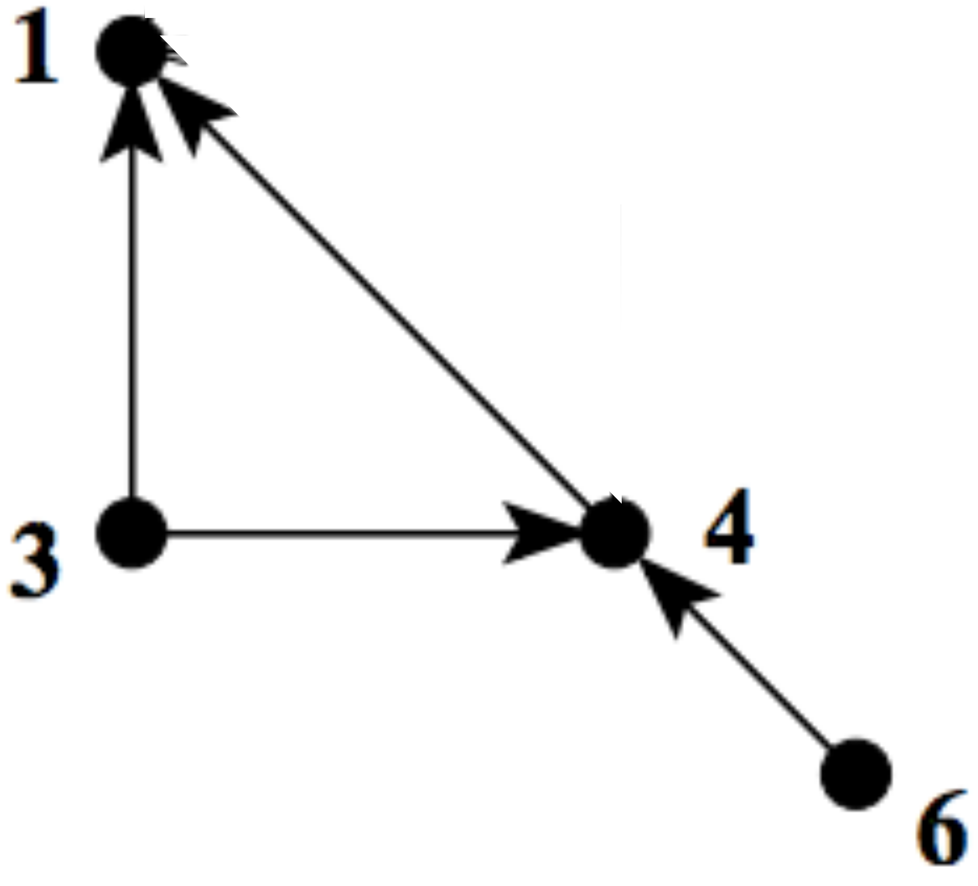




Example: one total ordering found by topological search is:

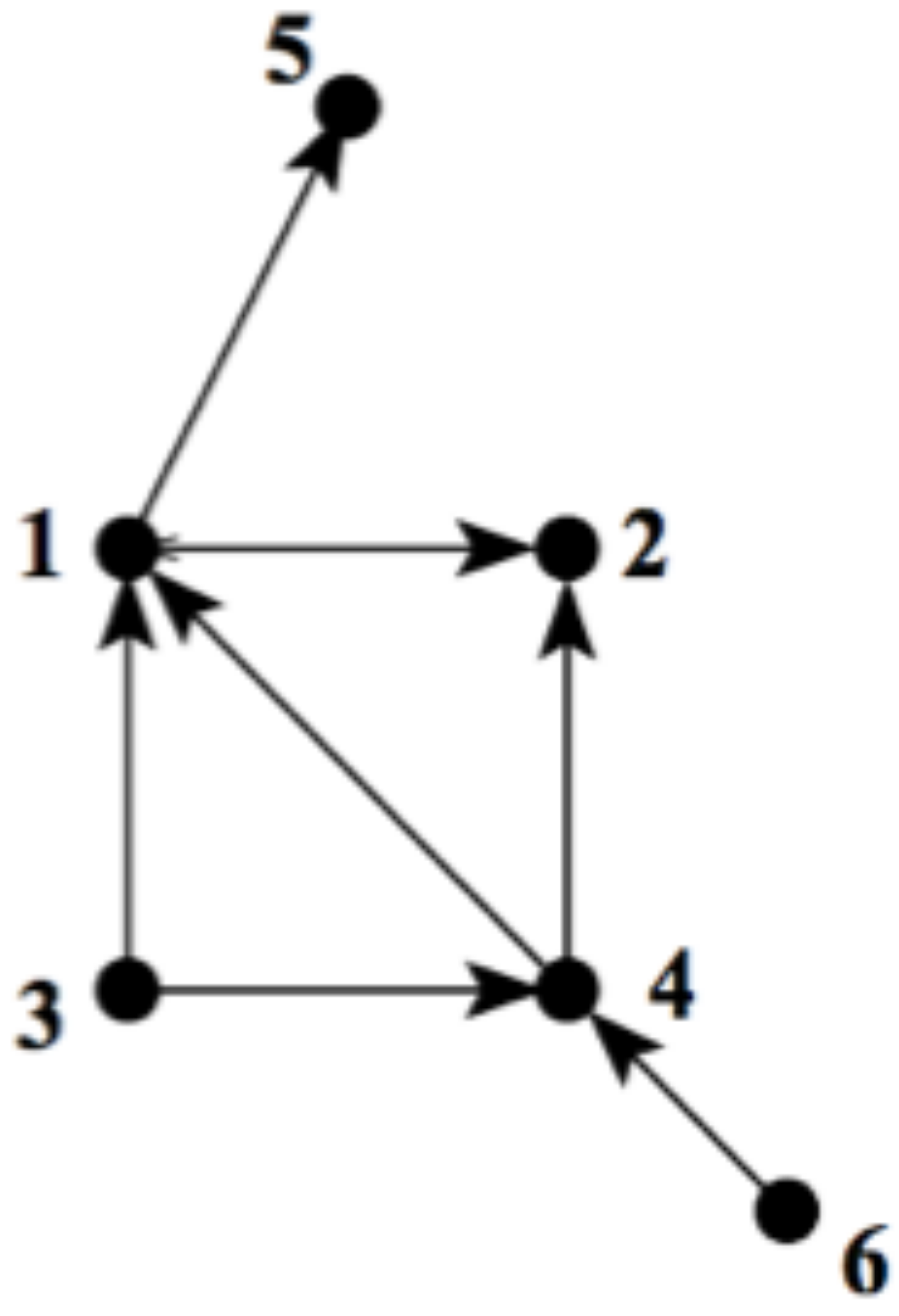
2 < 5

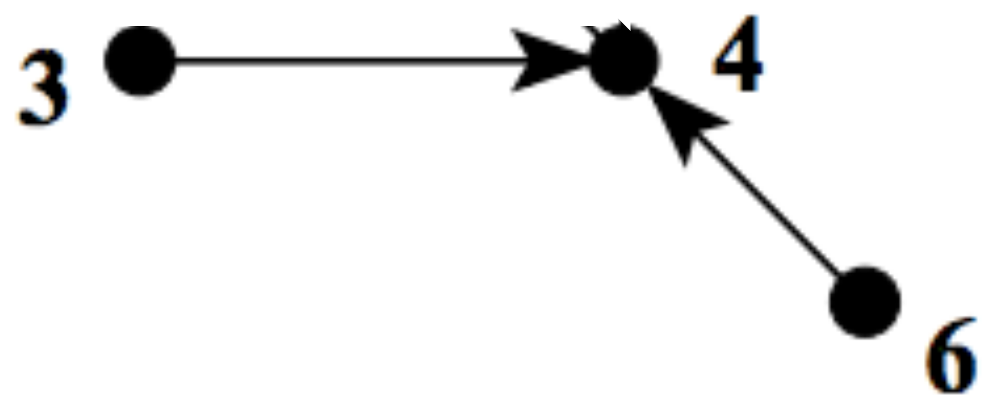




Example: one total ordering found by topological search is:

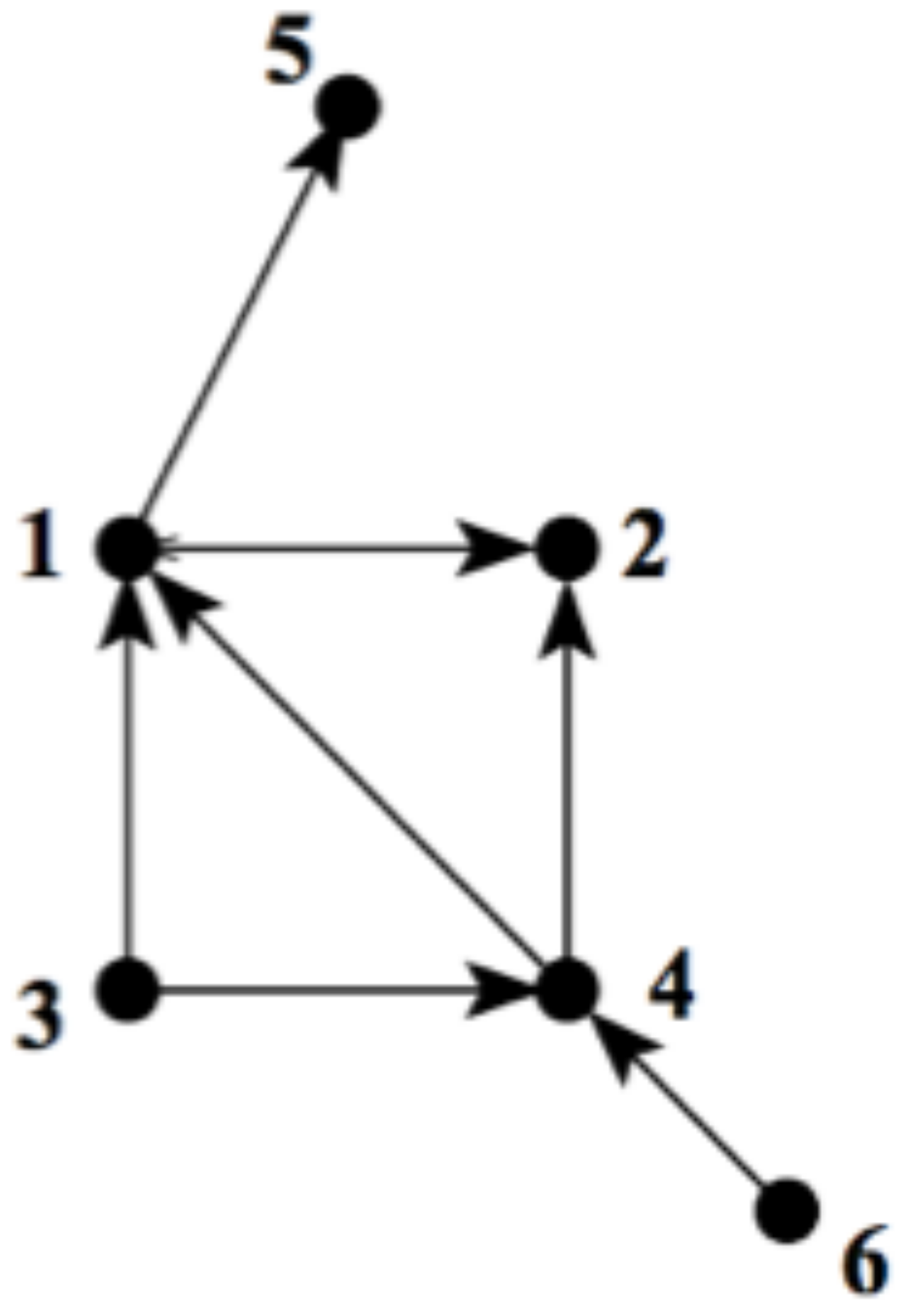
$1 < 2 < 5$

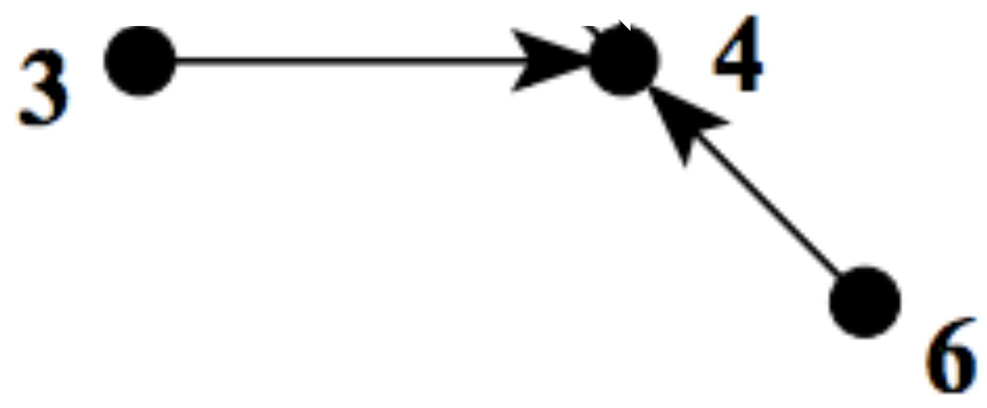




Example: one total ordering found by topological search is:

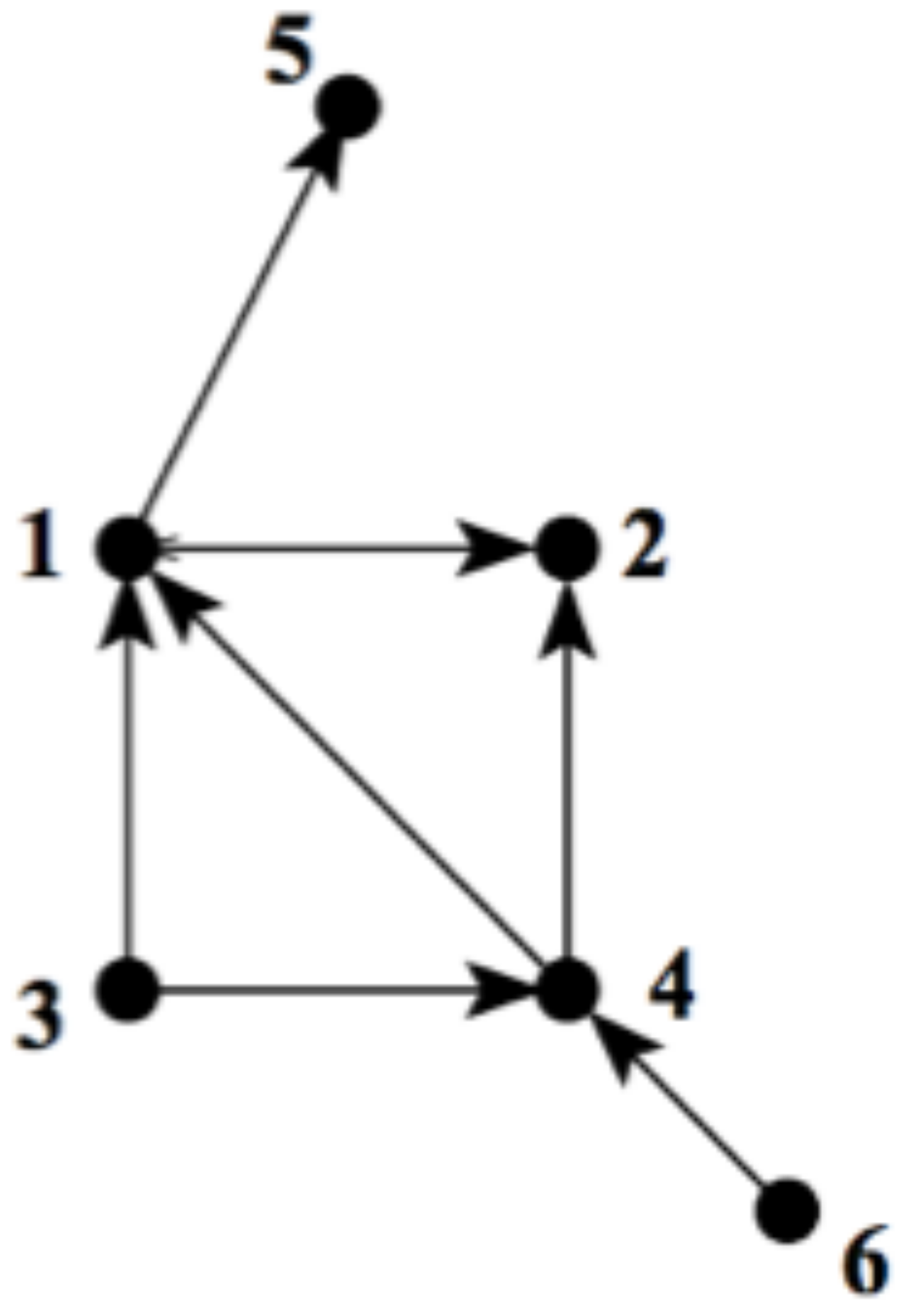
$1 < 2 < 5$





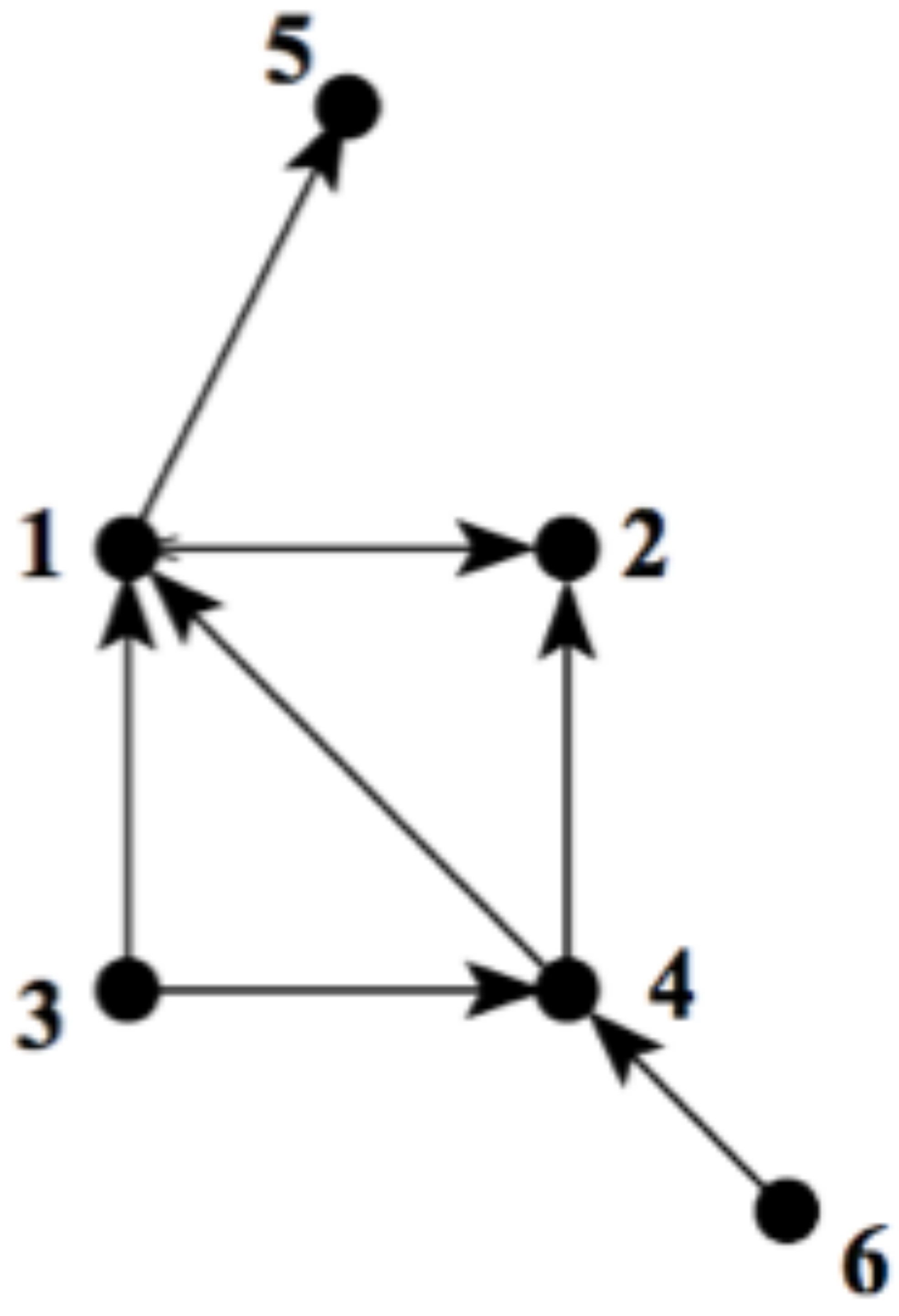
Example: one total ordering found by topological search is:

4 < 1 < 2 < 5



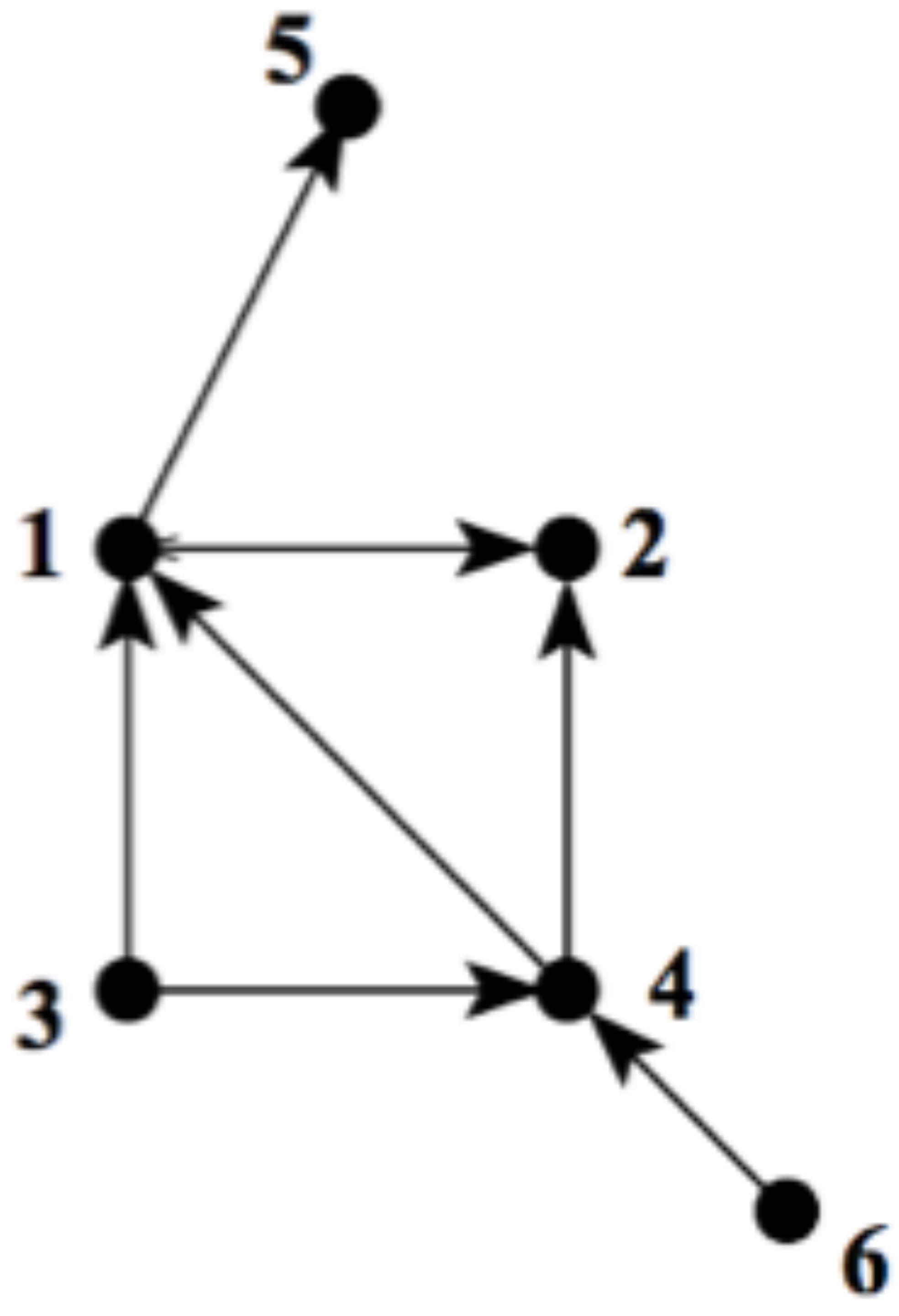
Example: one total ordering found by topological search is:

4 < 1 < 2 < 5



Example: one total ordering found by topological search is:

3 < 6 < 4 < 1 < 2 < 5



Example: one total ordering found by topological search is:

$3 < 6 < 4 < 1 < 2 < 5$

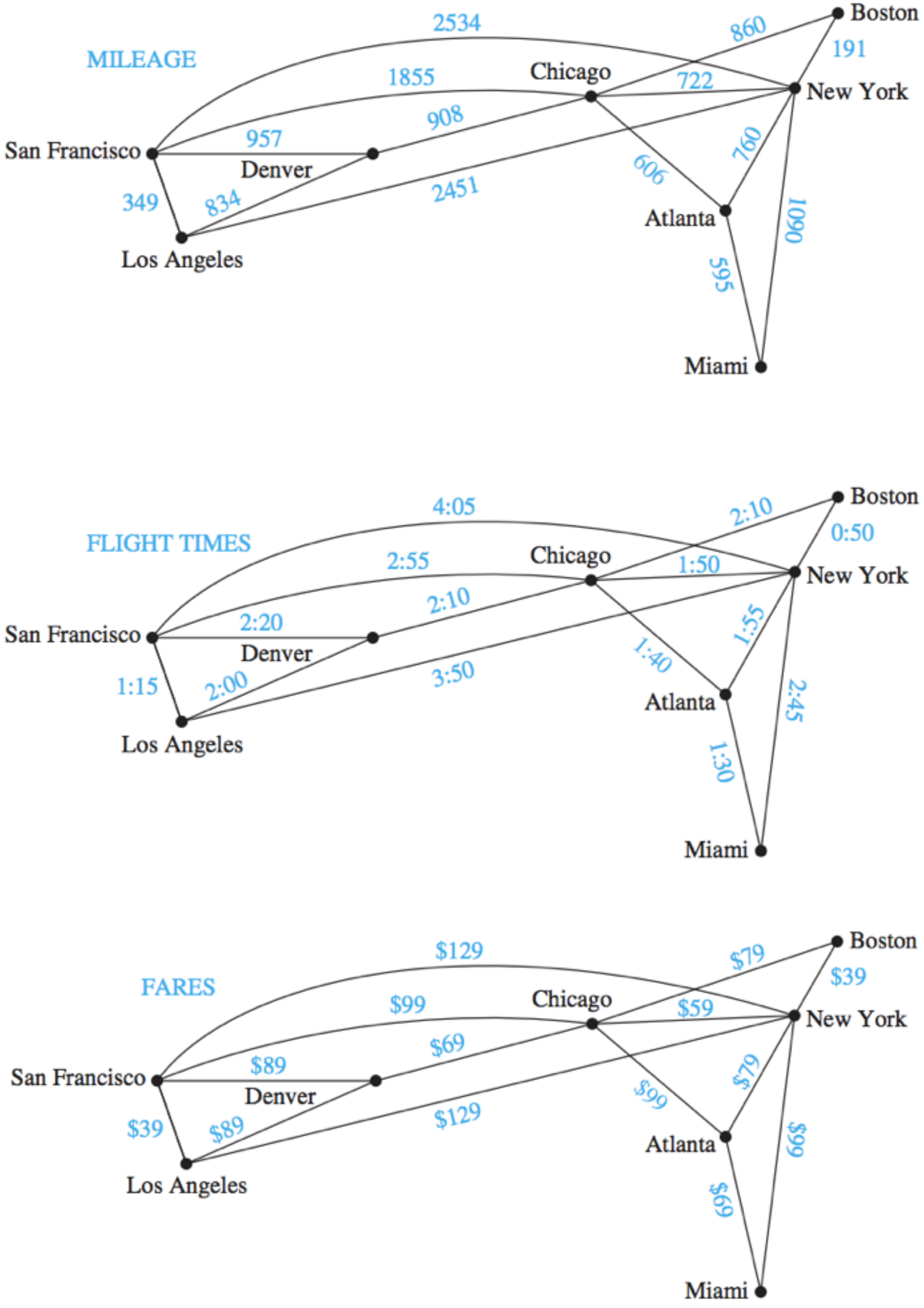


FIGURE 1 Weighted Graphs Modeling an Airline System.

(from Rosen, 10.6)

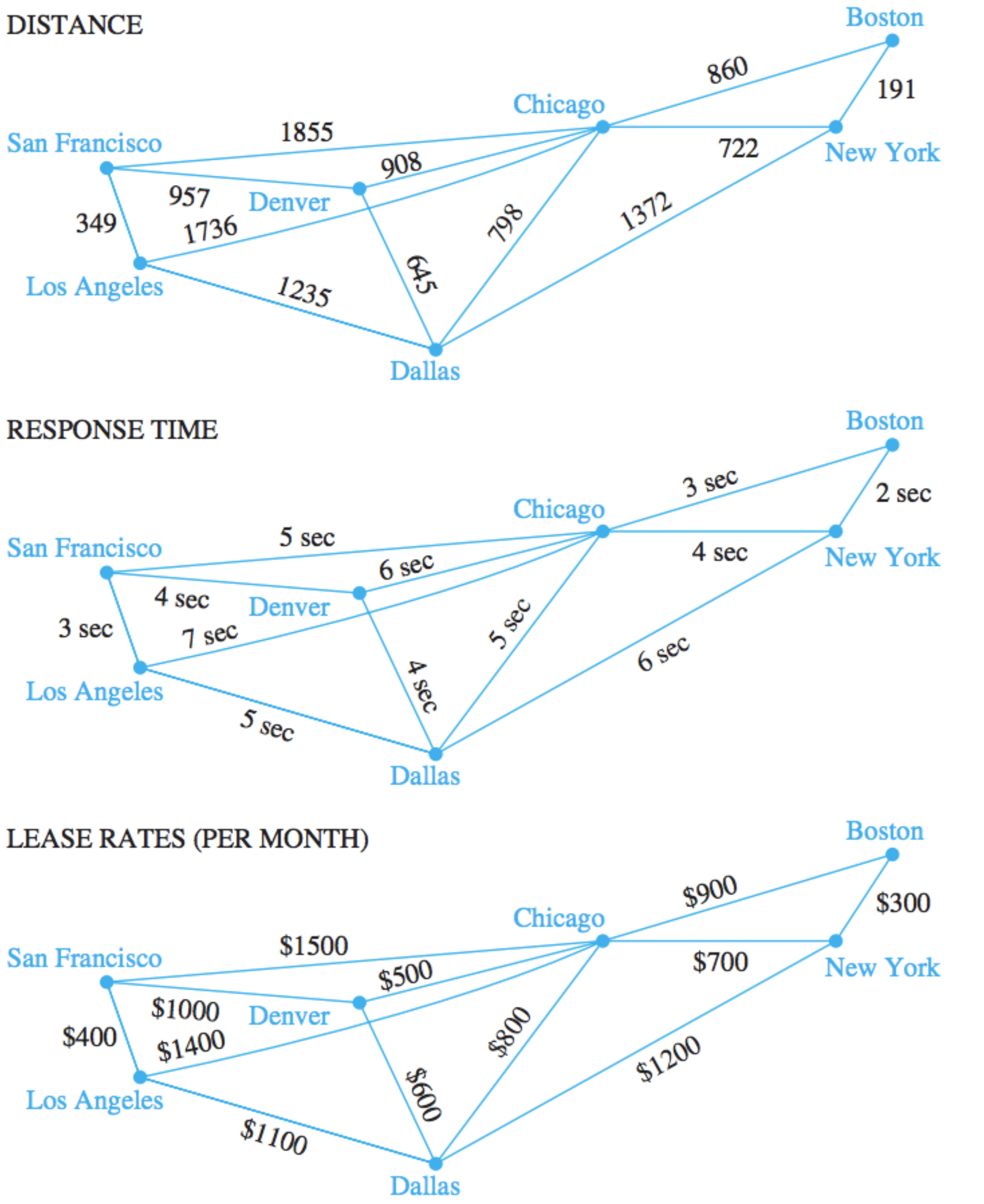


FIGURE 2 Weighted Graphs Modeling a Computer Network.

A weighted graph is a graph such that each edge e
has an associated real number $w(e)$ called the weight of the edge.

Given a graph with weights on each of its edges,
we want to determine a spanning tree with the smallest total weight

This is called a minimal spanning tree.

The total weight of a tree (or any graph) is the sum of the weights of its edges.

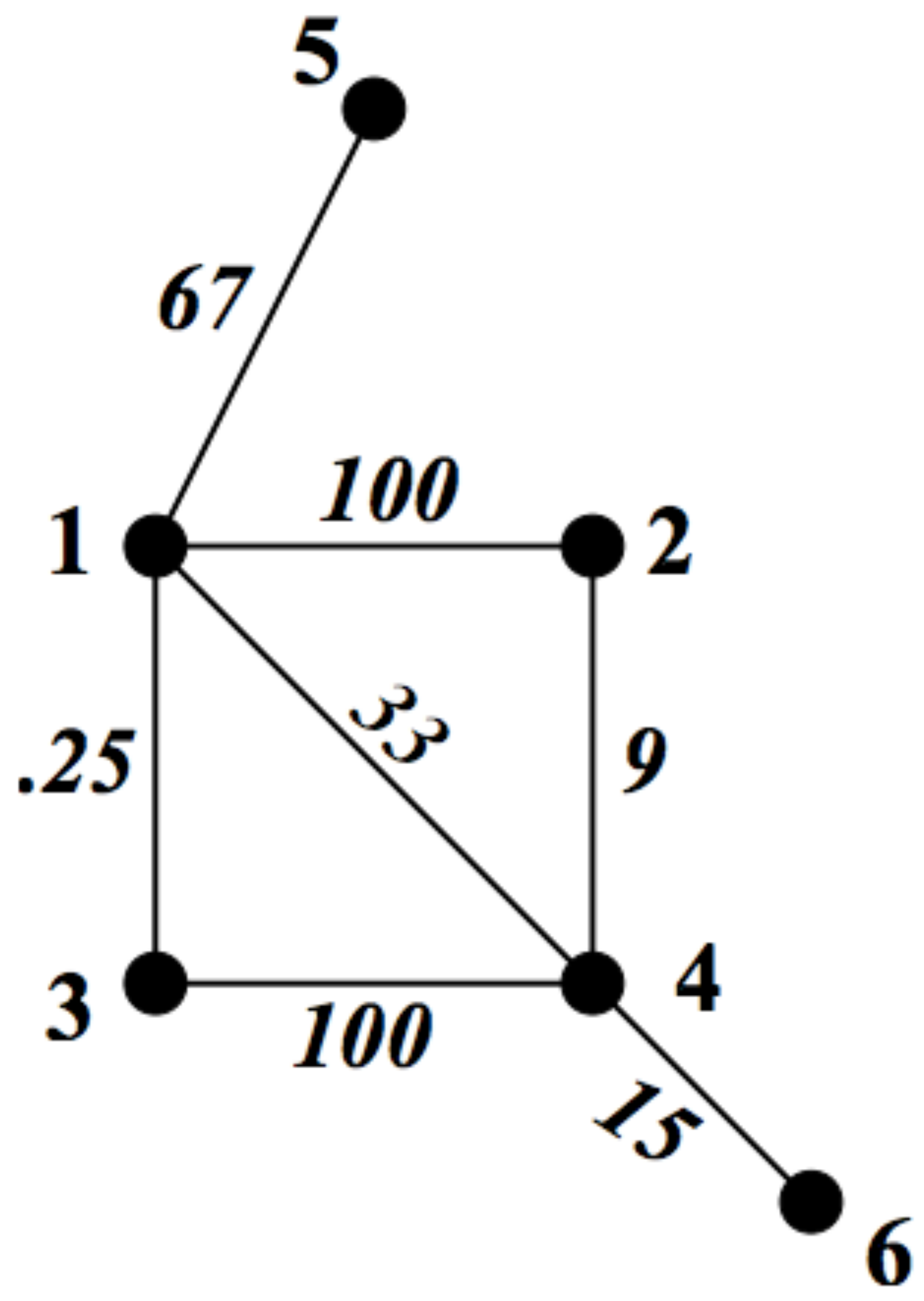
Here we consider a greedy algorithm, Kruskal's Algorithm,
to find the minimal spanning tree.

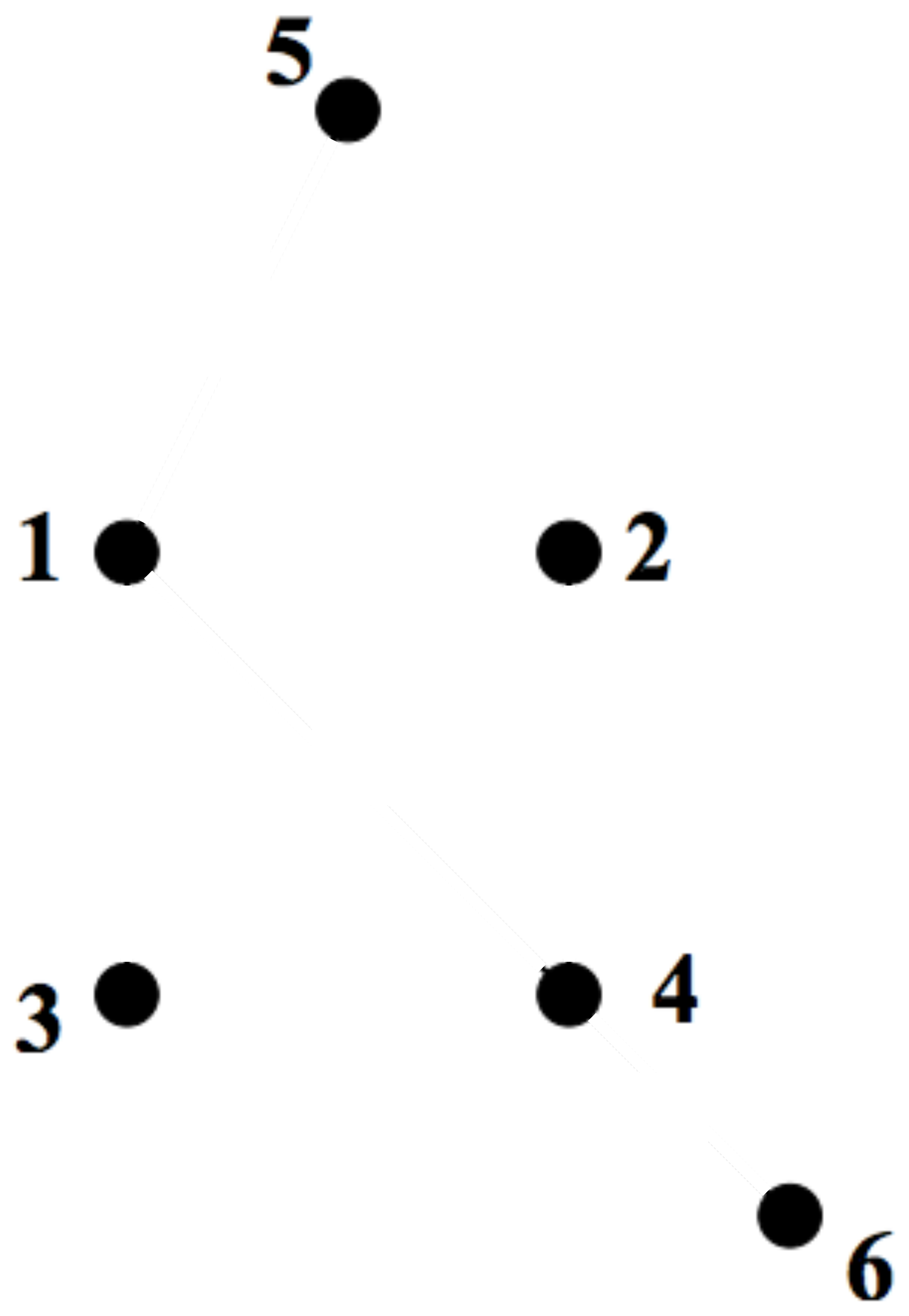
Kruskal's Algorithm

Find minimal spanning tree

Initialize T = (V, E):
V = {vertices of G}
E = {}

1. Take an edge $e \in G$
   such that w(e) is minimal
   If $E \cup e$ is a tree, add e to E
2. Remove e from G.
3. If $|T| == n - 1$ stop
   else: go to step 1



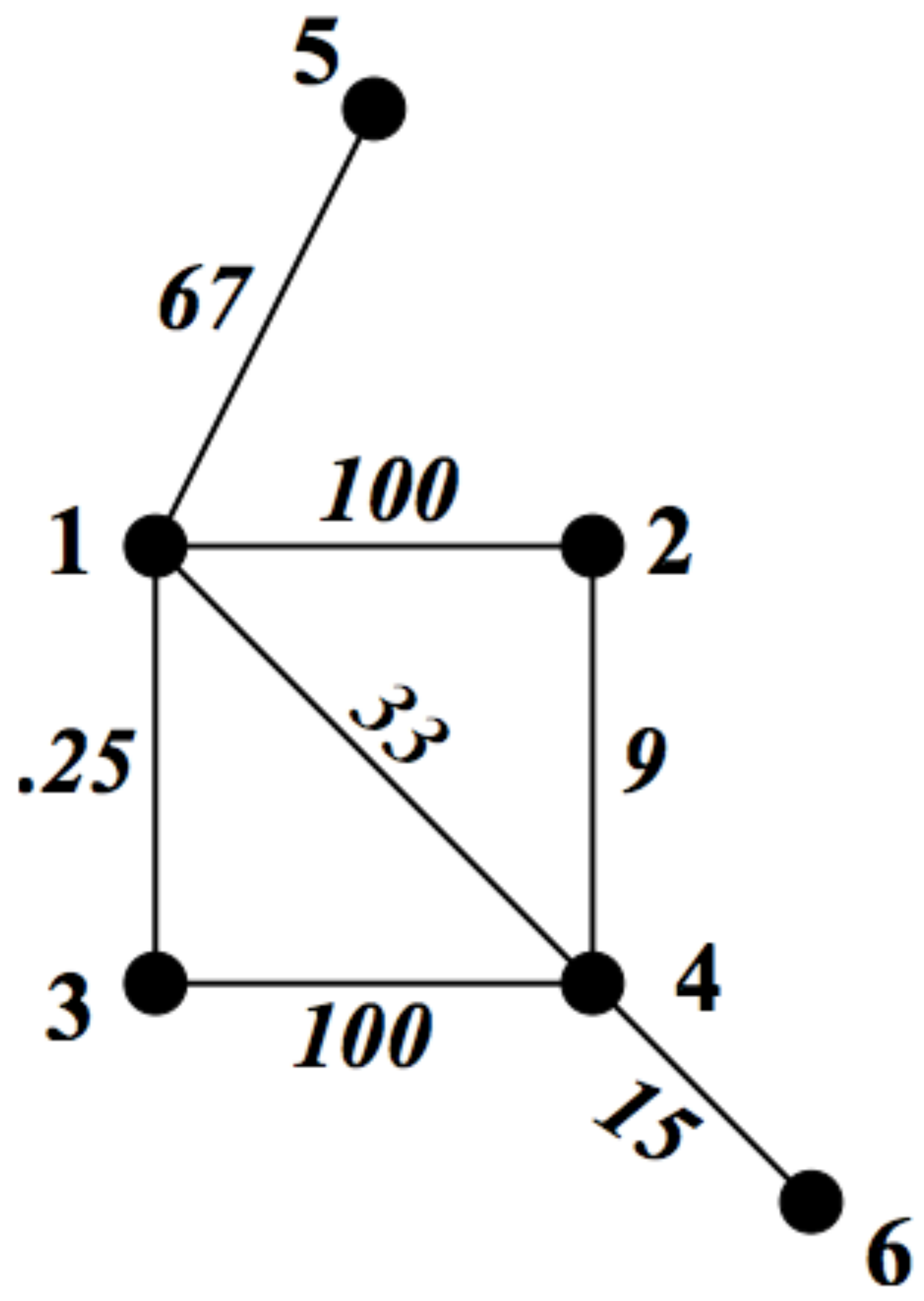


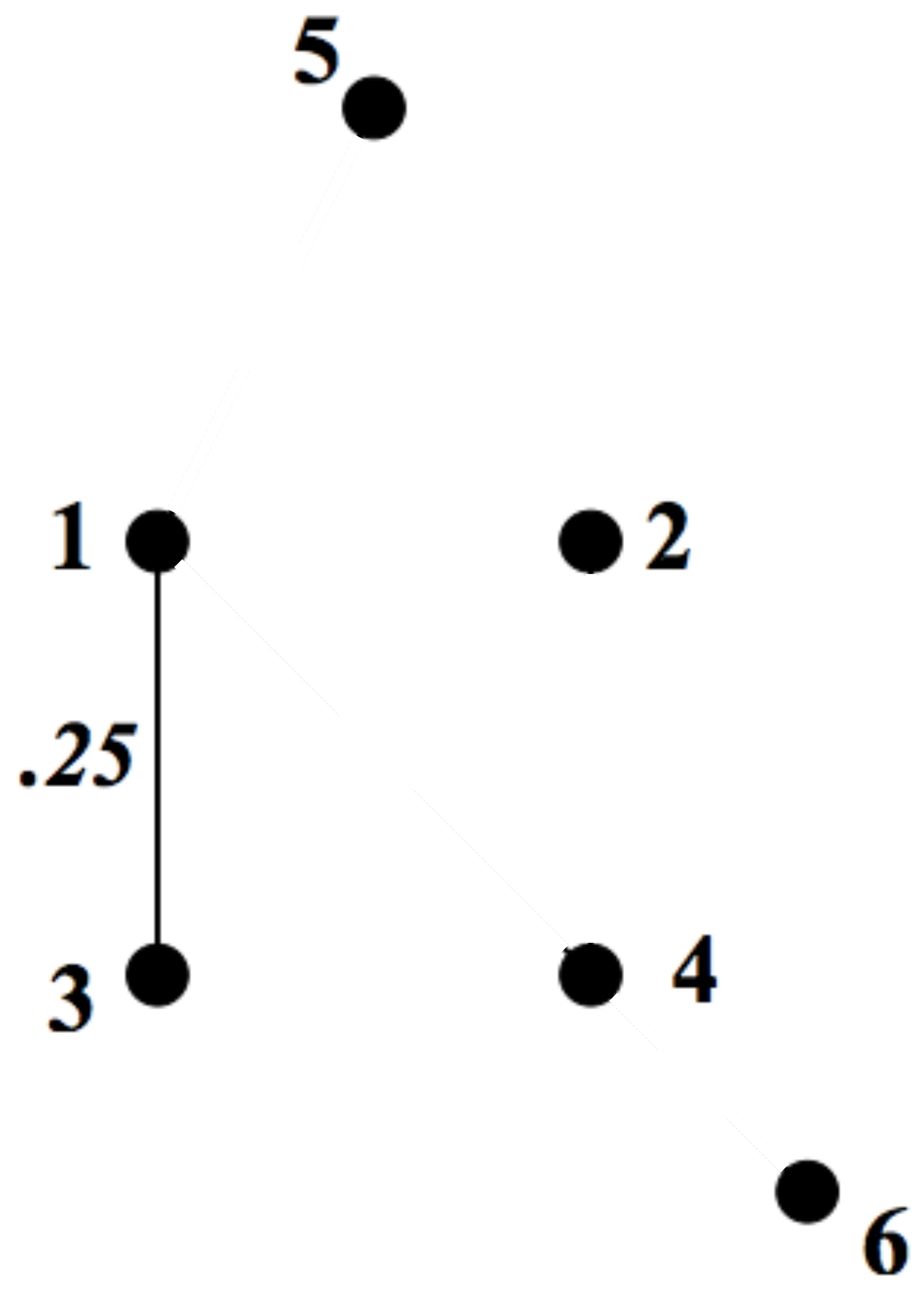
Kruskal's Algorithm

Find minimal spanning tree

Initialize T = (V, E):
V = {vertices of G}
E = {}

1. Take an edge e ∈ G
   such that w(e) is minimal
   If E ∪ e is a tree, add e to E
2. Remove e from G.
3. If |T| == n – 1 stop
   else: go to step 1



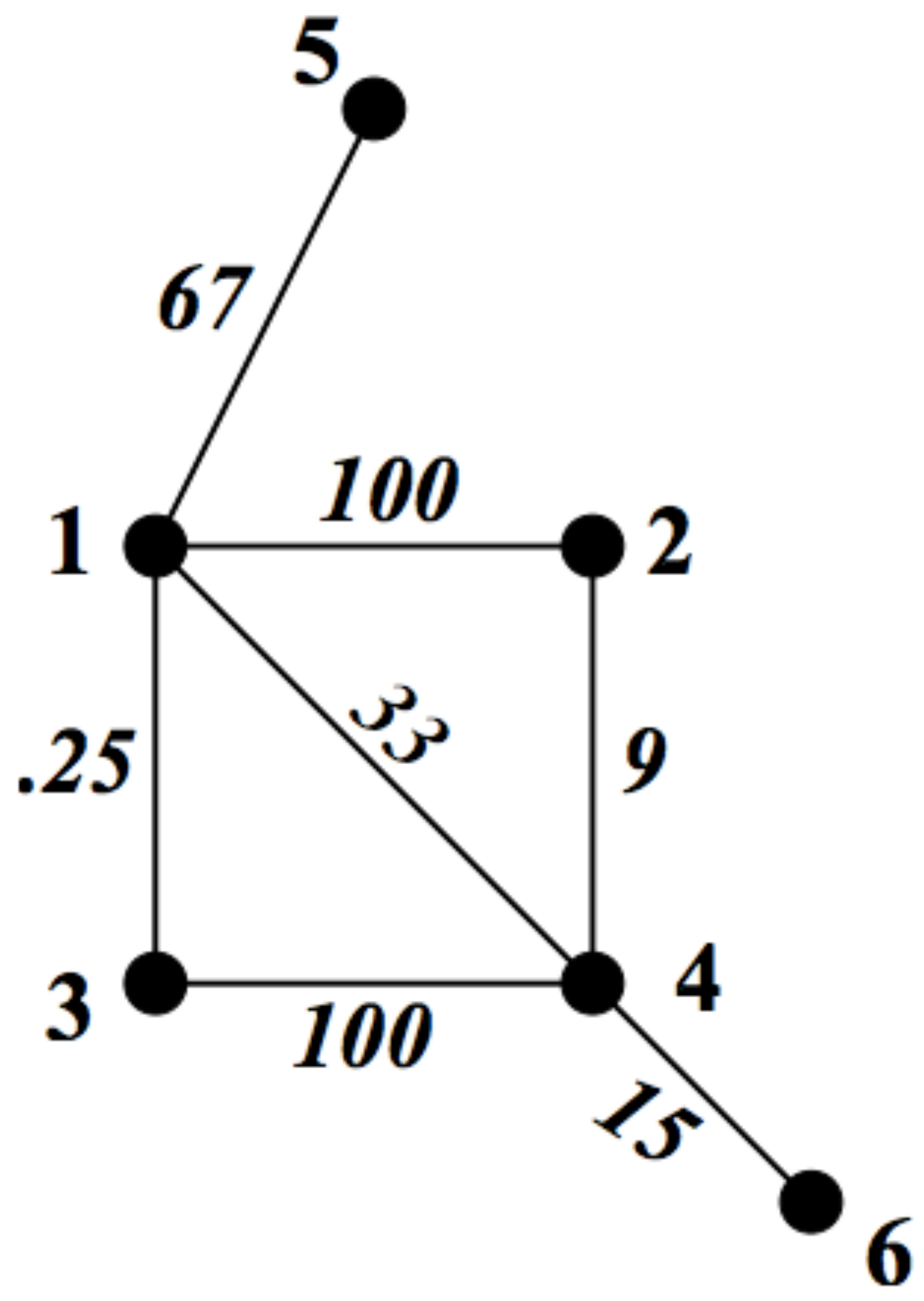


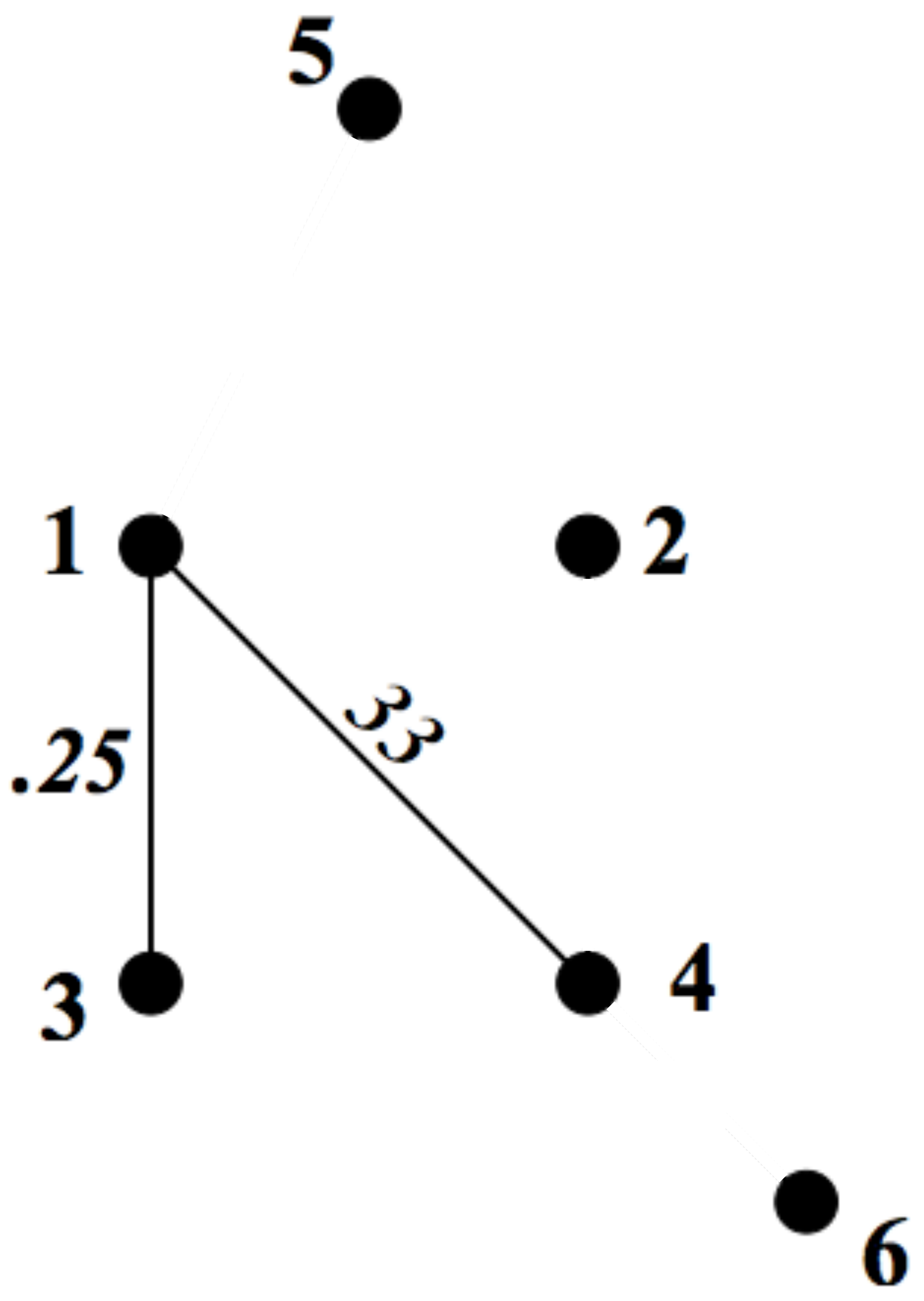
Kruskal's Algorithm

Find minimal spanning tree

Initialize T = (V, E):
V = {vertices of G}
E = {}

1. Take an edge $e \in G$
   such that w(e) is minimal
   If $E \cup e$ is a tree, add e to E
2. Remove e from G.
3. If $|T| == n - 1$ stop
   else: go to step 1



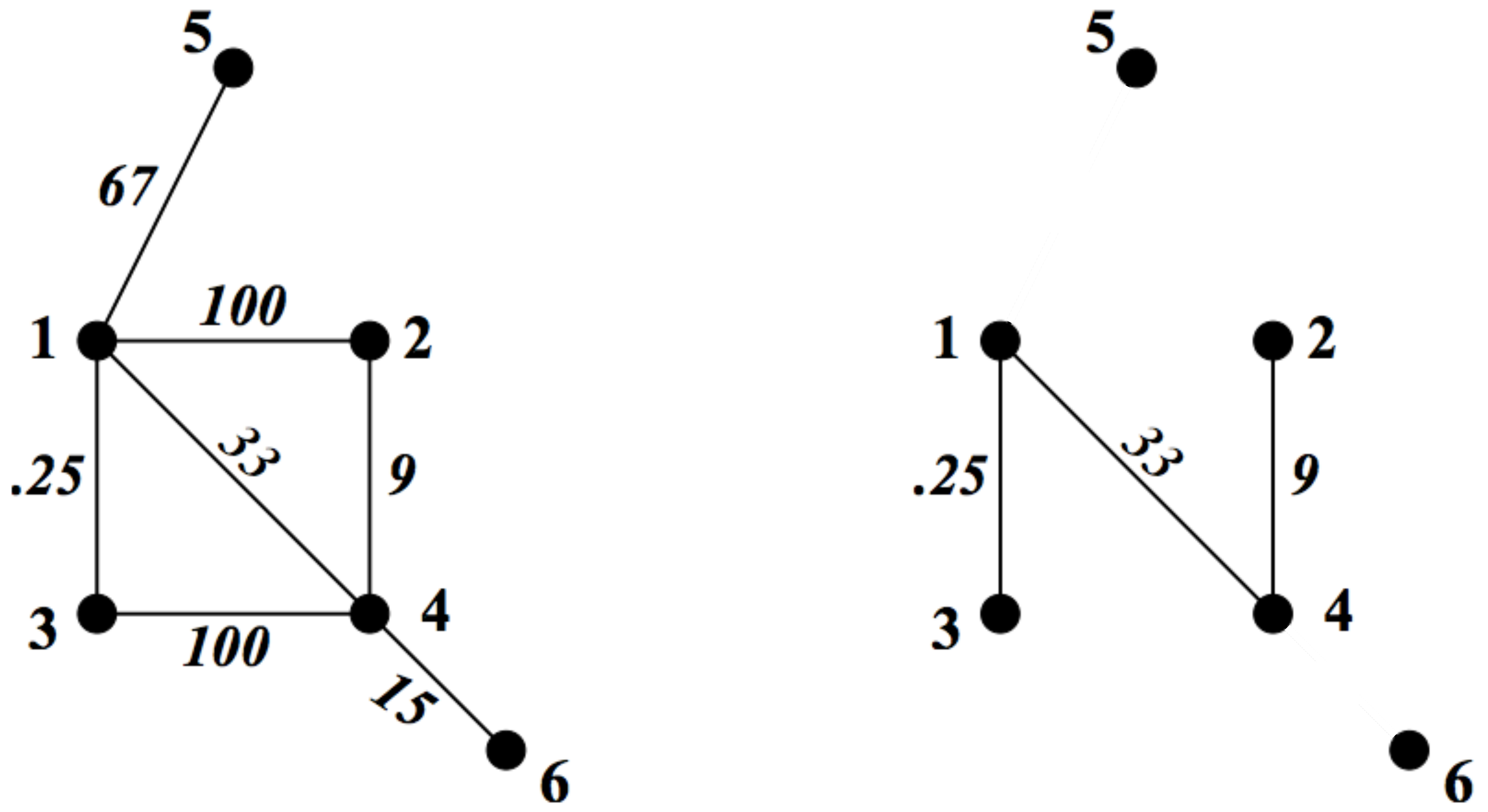


Kruskal's Algorithm

Find minimal spanning tree

Initialize T = (V, E):
V = {vertices of G}
E = {}

1. Take an edge e ∈ G
   such that w(e) is minimal
   If E ∪ e is a tree, add e to E
2. Remove e from G.
3. If |T| == n – 1 stop
   else: go to step 1

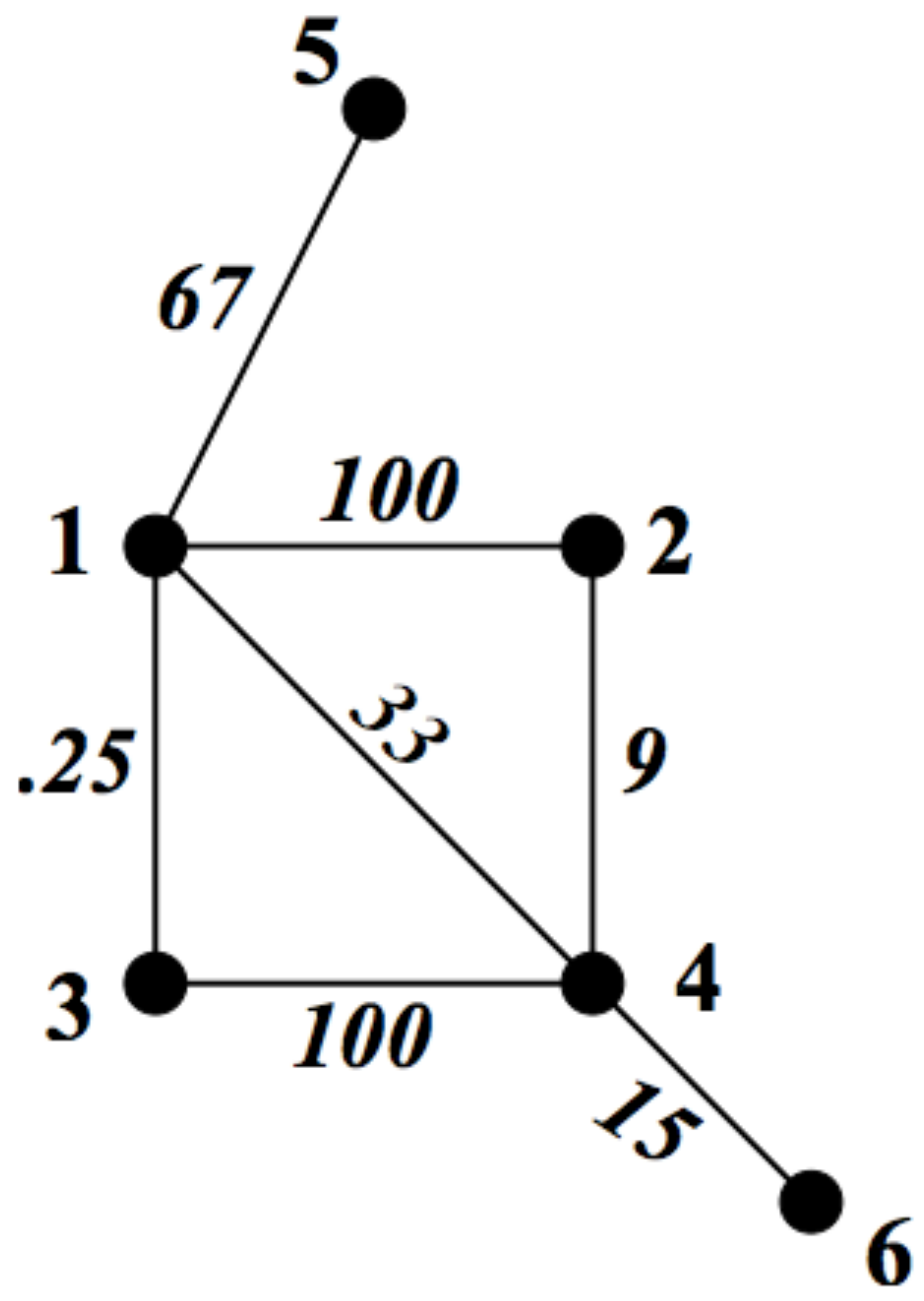


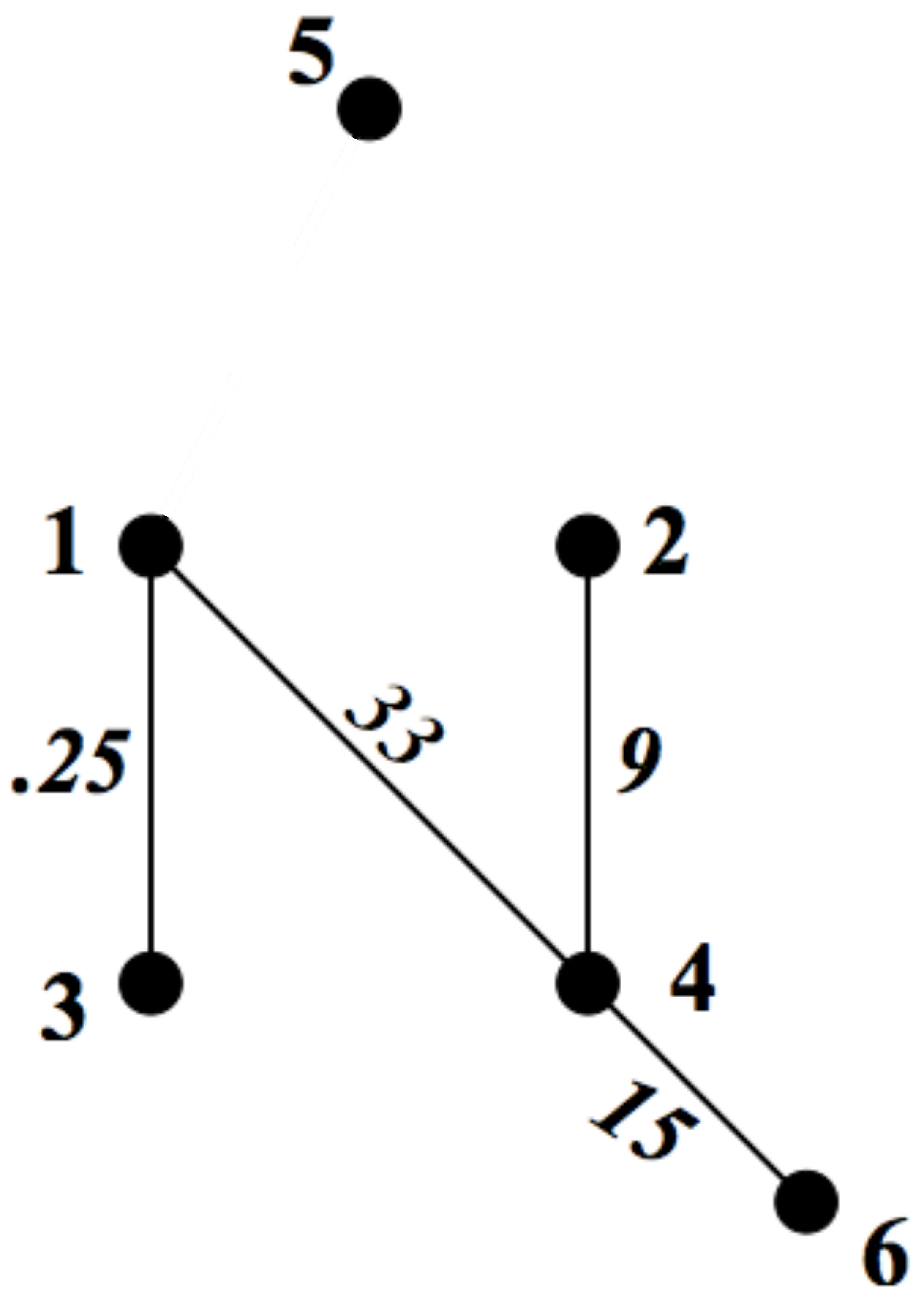
Kruskal’s Algorithm

Find minimal spanning tree

Initialize T = (V, E):
V = {vertices of G}
E = {}

1. Take an edge e $\in$ G
   such that w(e) is minimal
   If E $\cup$ e is a tree, add e to E
2. Remove e from G.
3. If |T| == n – 1 stop
   else: go to step 1





Kruskal's Algorithm

Find minimal spanning tree

Initialize T = (V, E):
V = {vertices of G}
E = {}

1. Take an edge $e \in G$
   such that w(e) is minimal
   If $E \cup e$ is a tree, add e to E
2. Remove e from G.
3. If $|T| == n - 1$ stop
   else: go to step 1