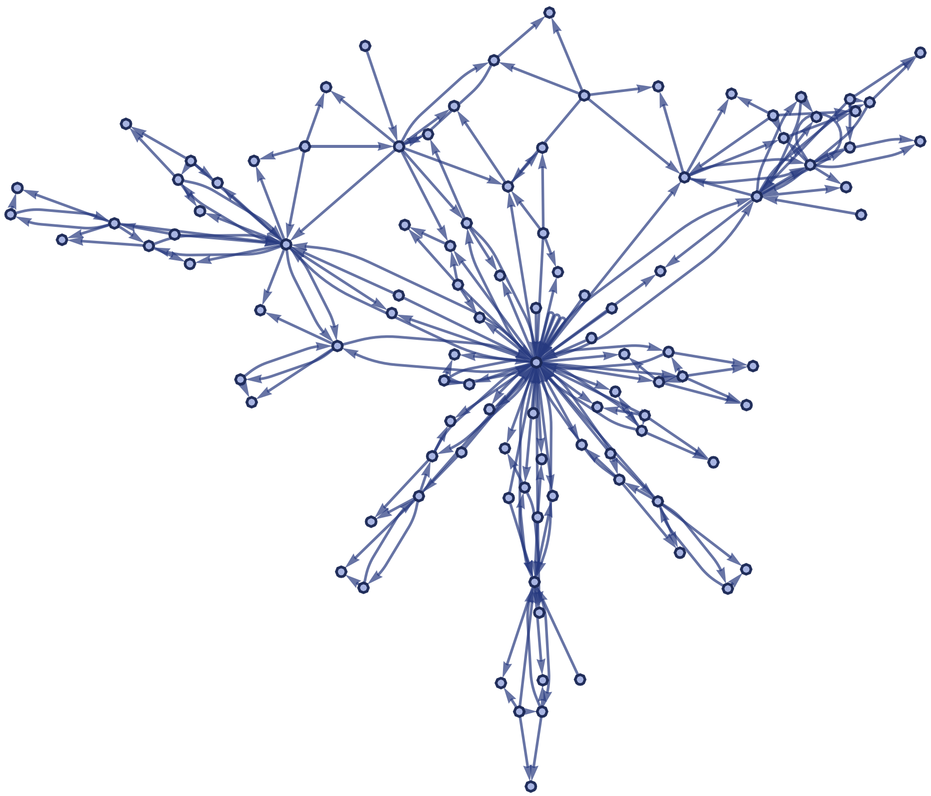
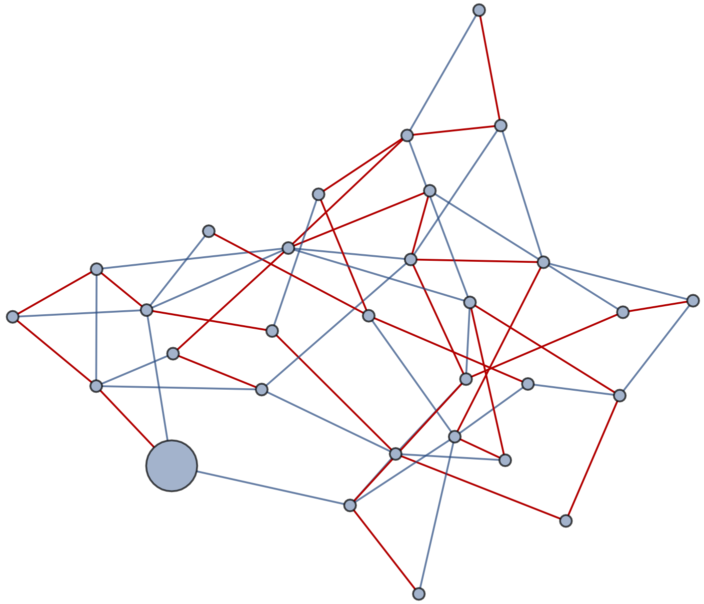


Graph Search Algorithms

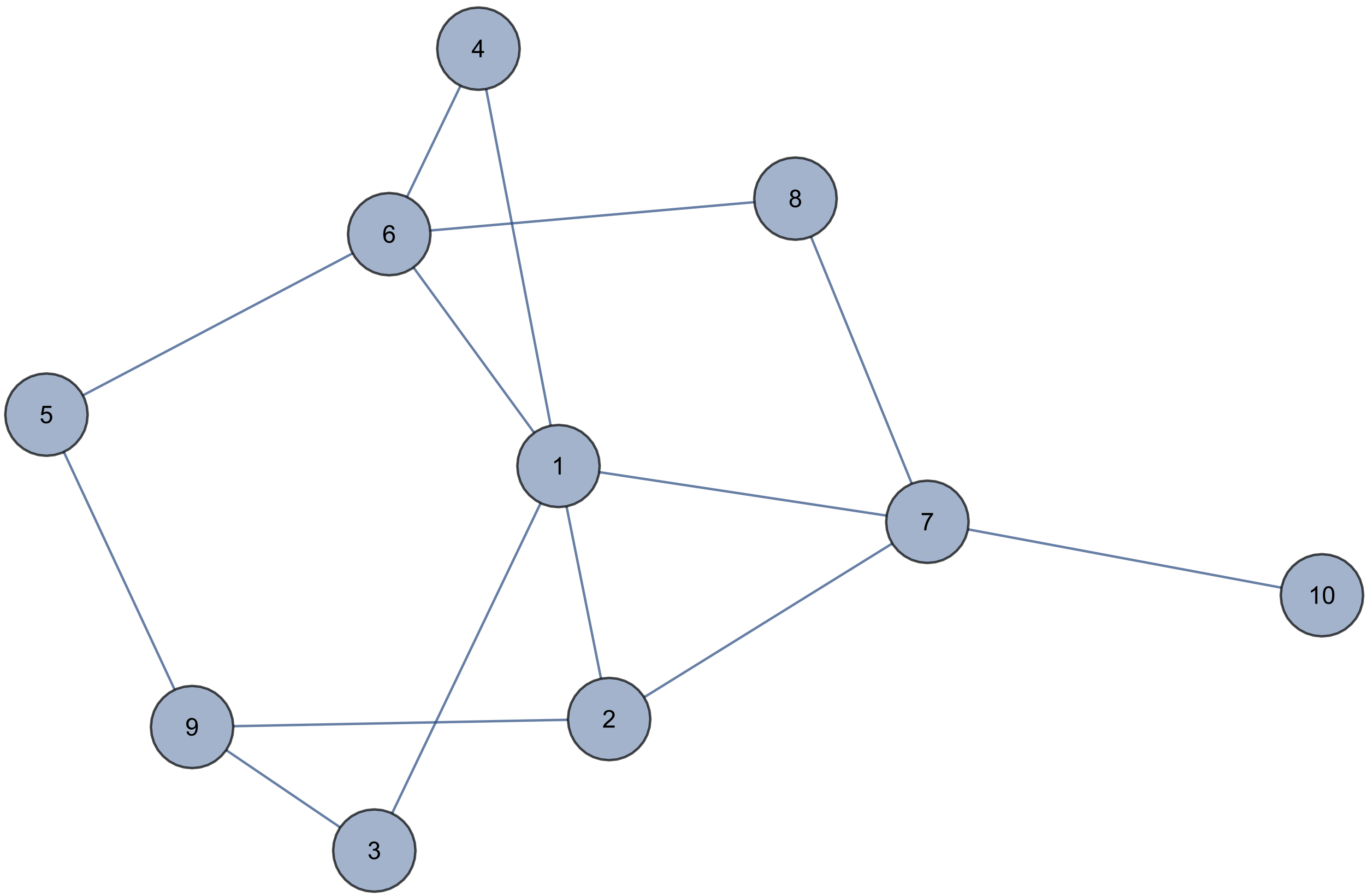
Tang Anke

Depth First Search

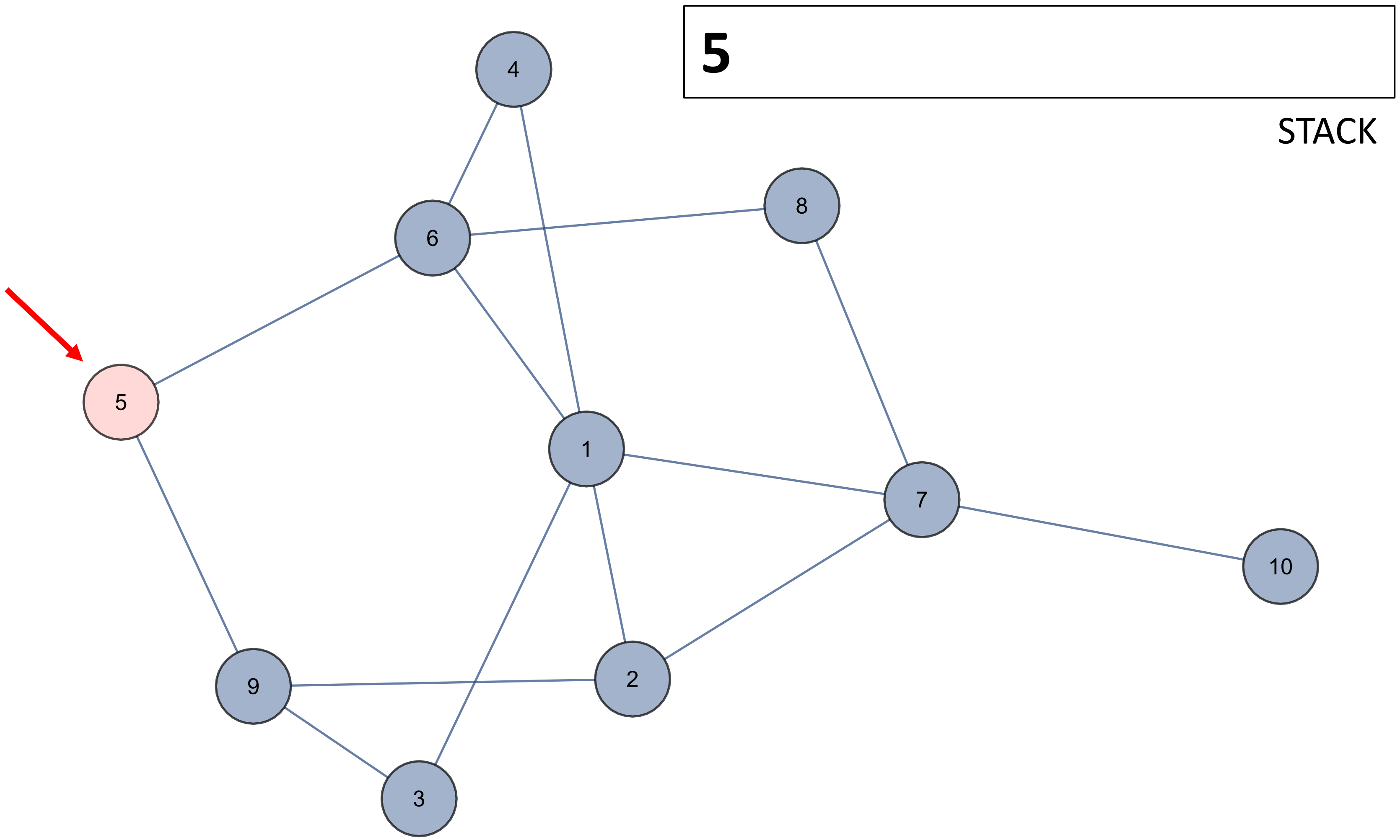
Depth-first search (**DFS**) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking.
A version of depth-first search was investigated in the 19th century by French mathematician Charles Pierre Trémaux as a strategy for solving mazes.



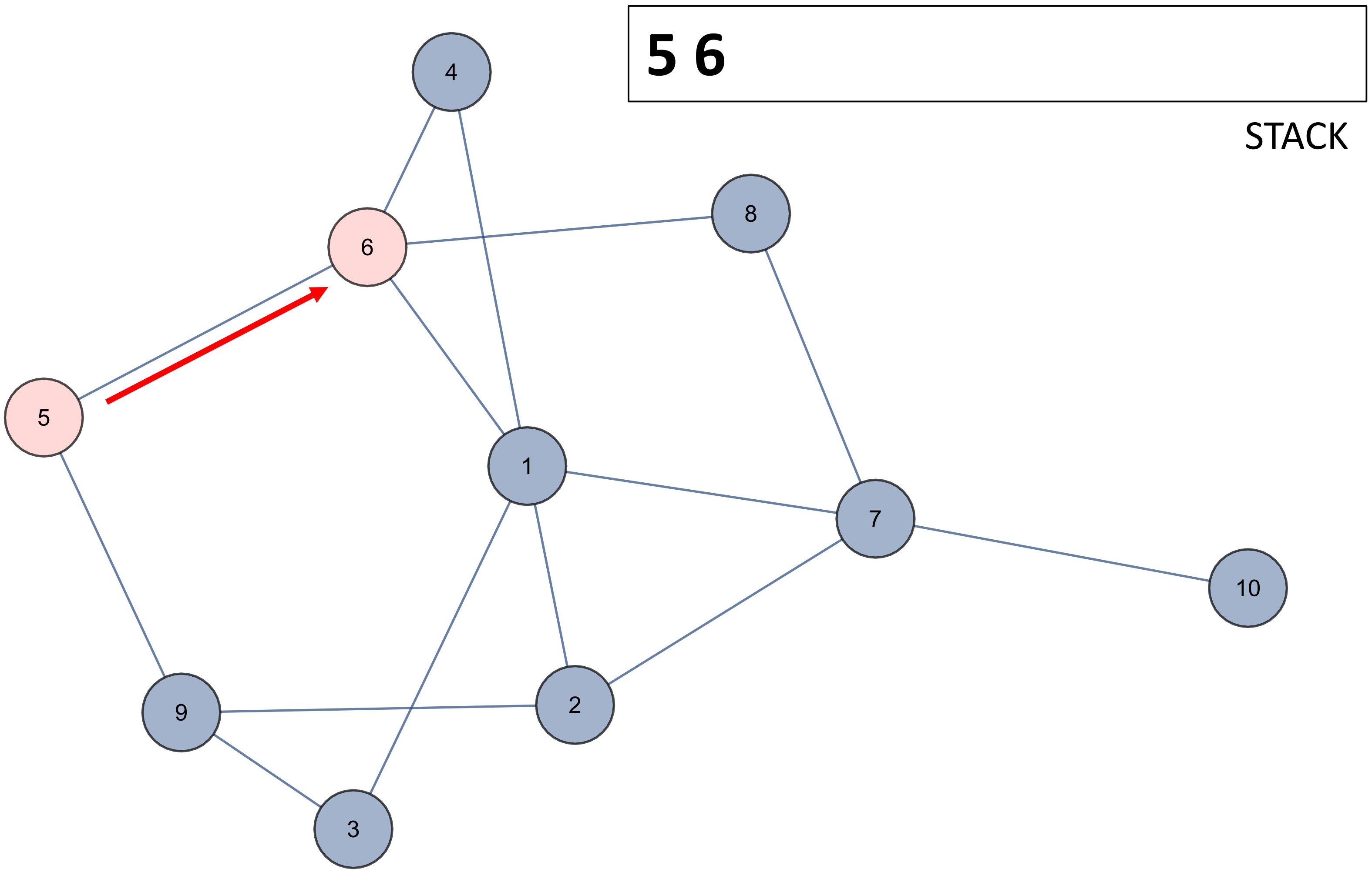
Depth First Search



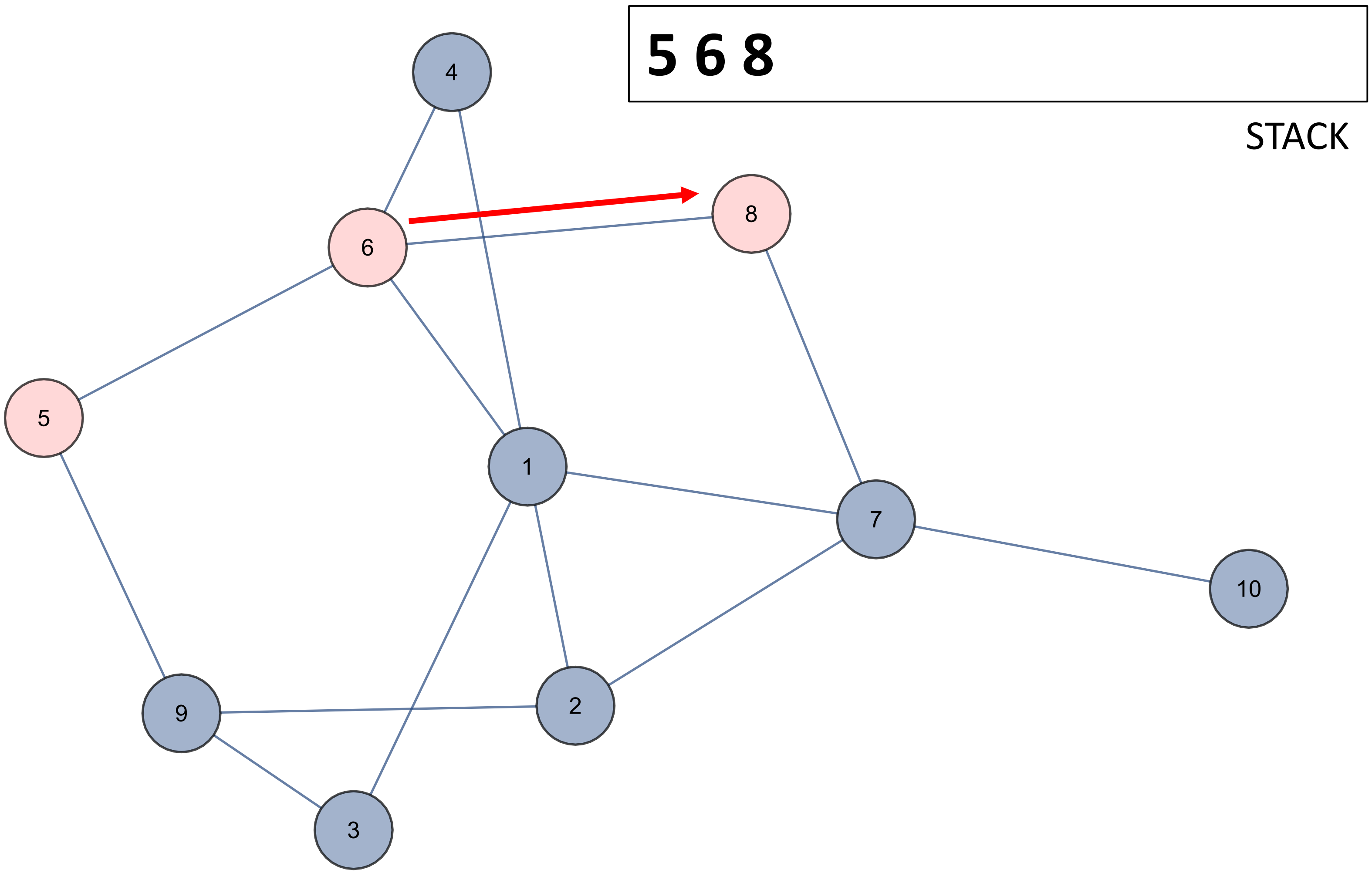
Depth First Search



Depth First Search



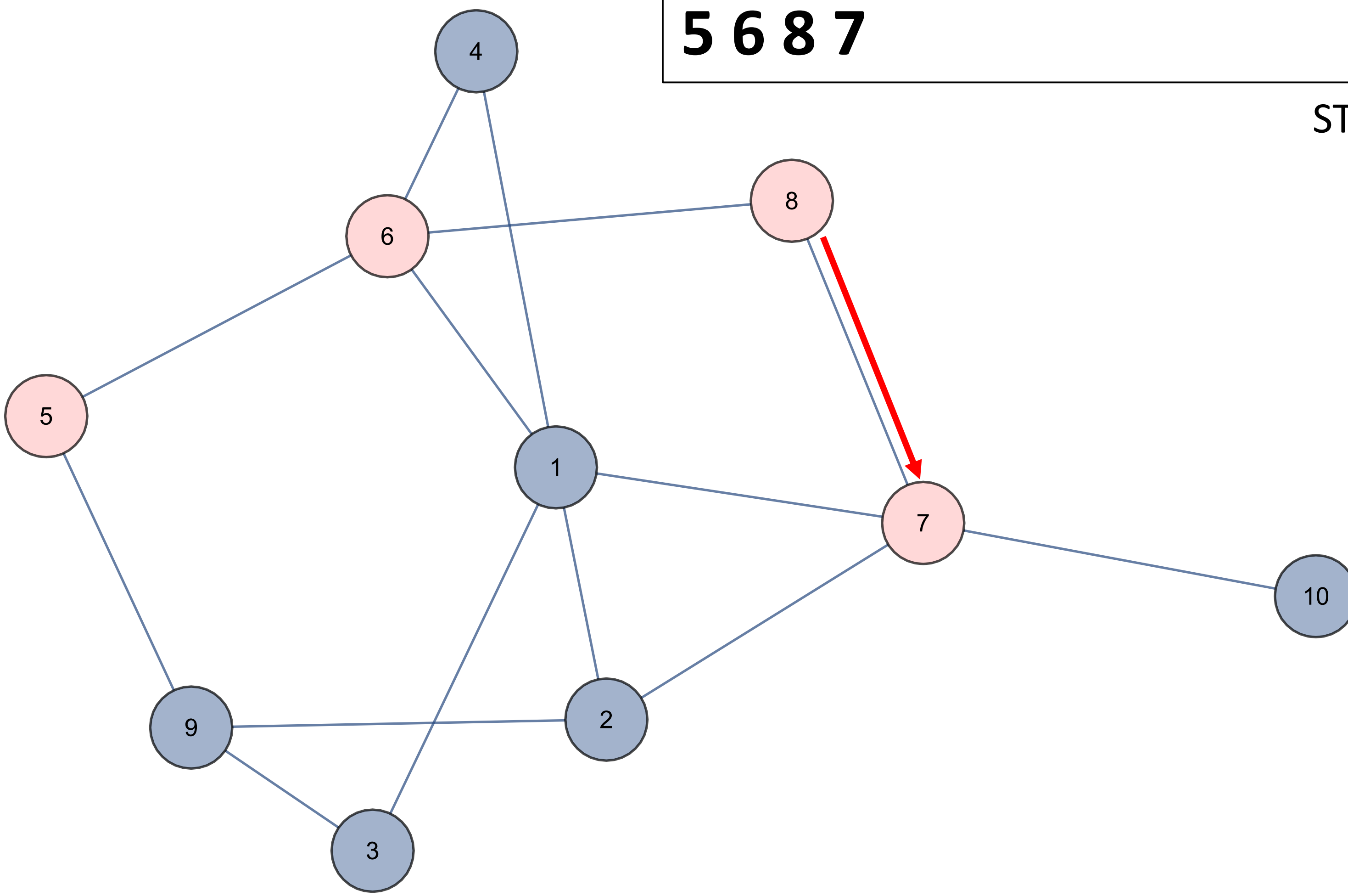
Depth First Search



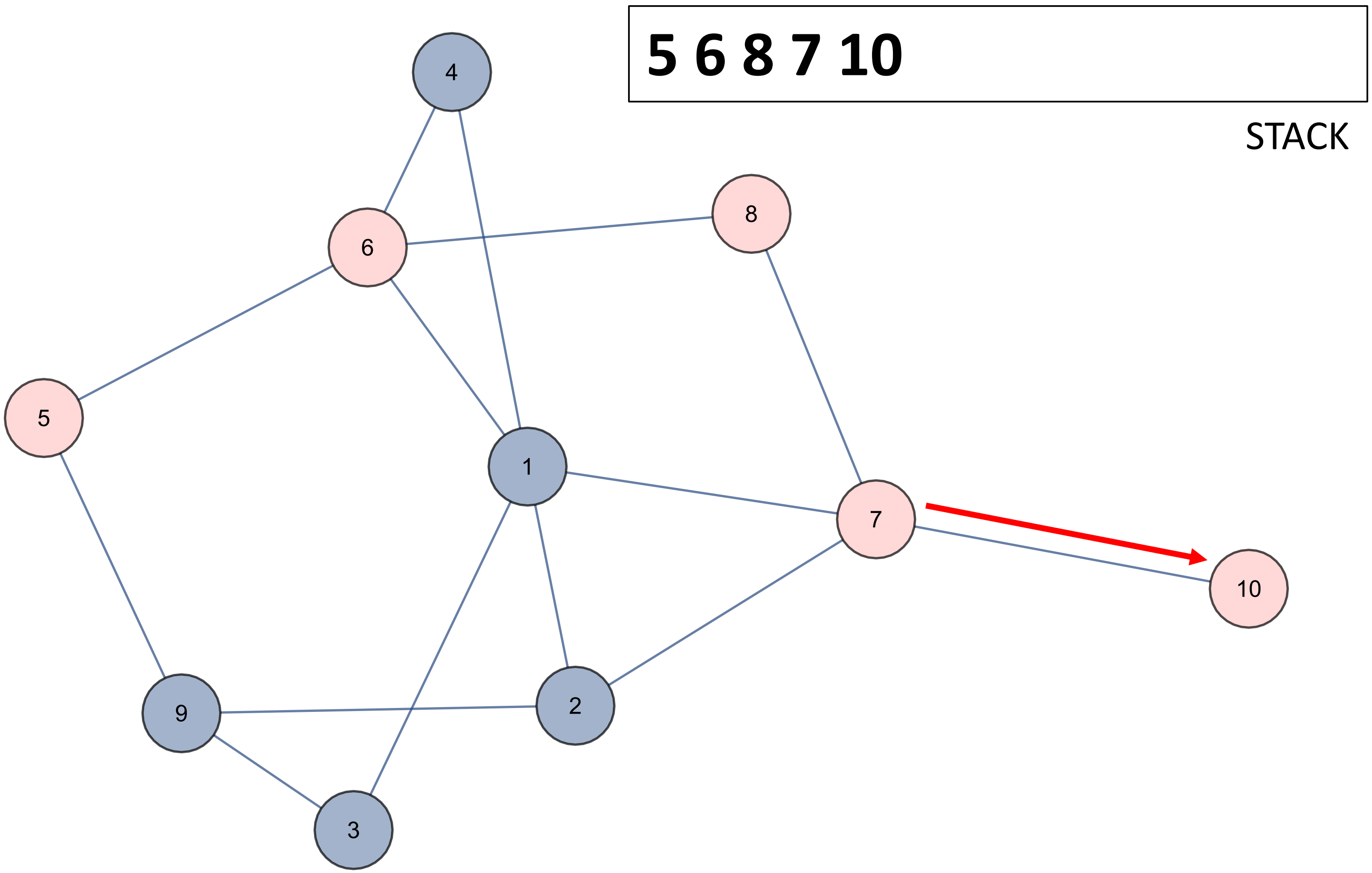
Depth First Search

5 6 8 7

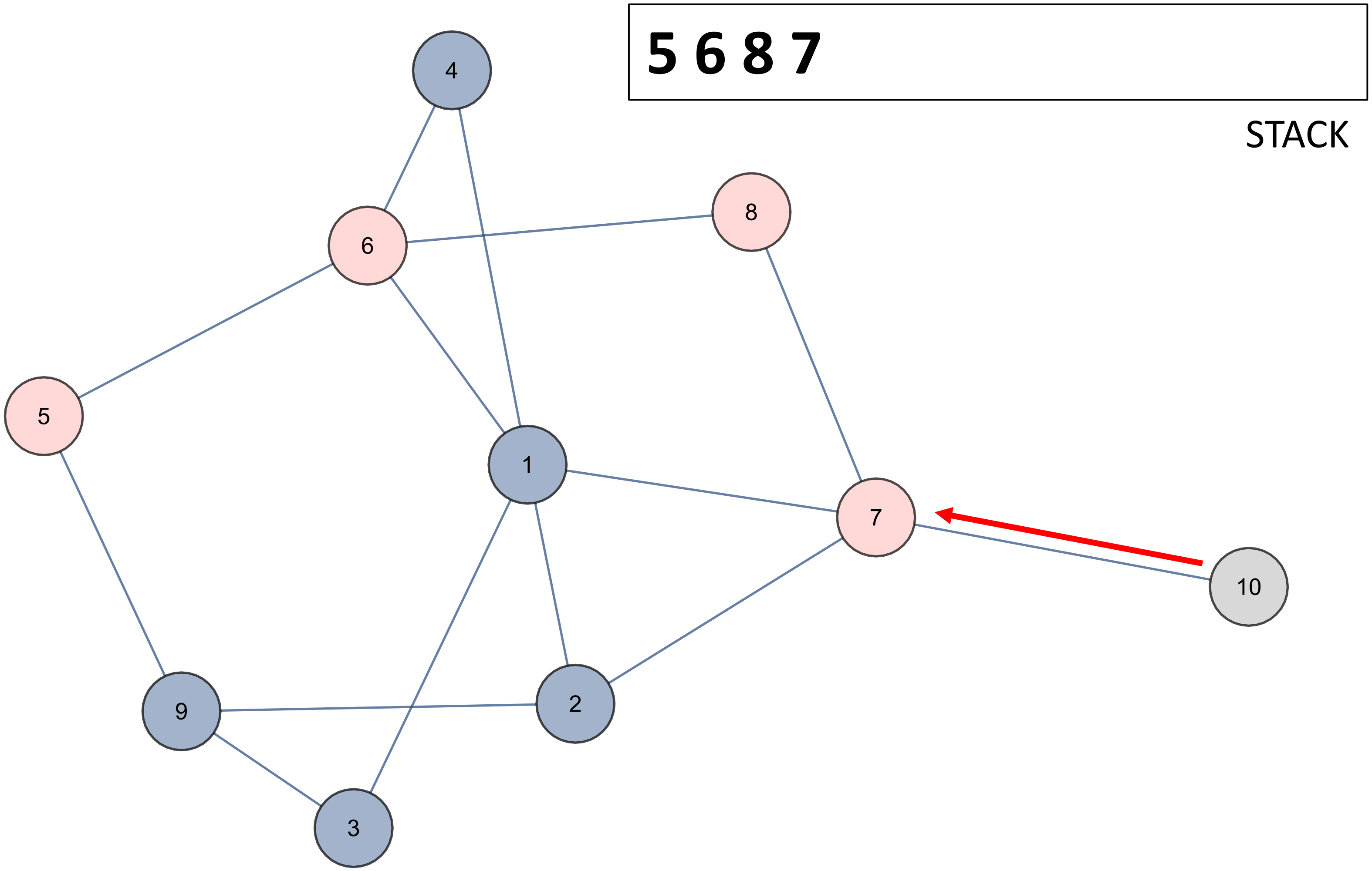
STACK



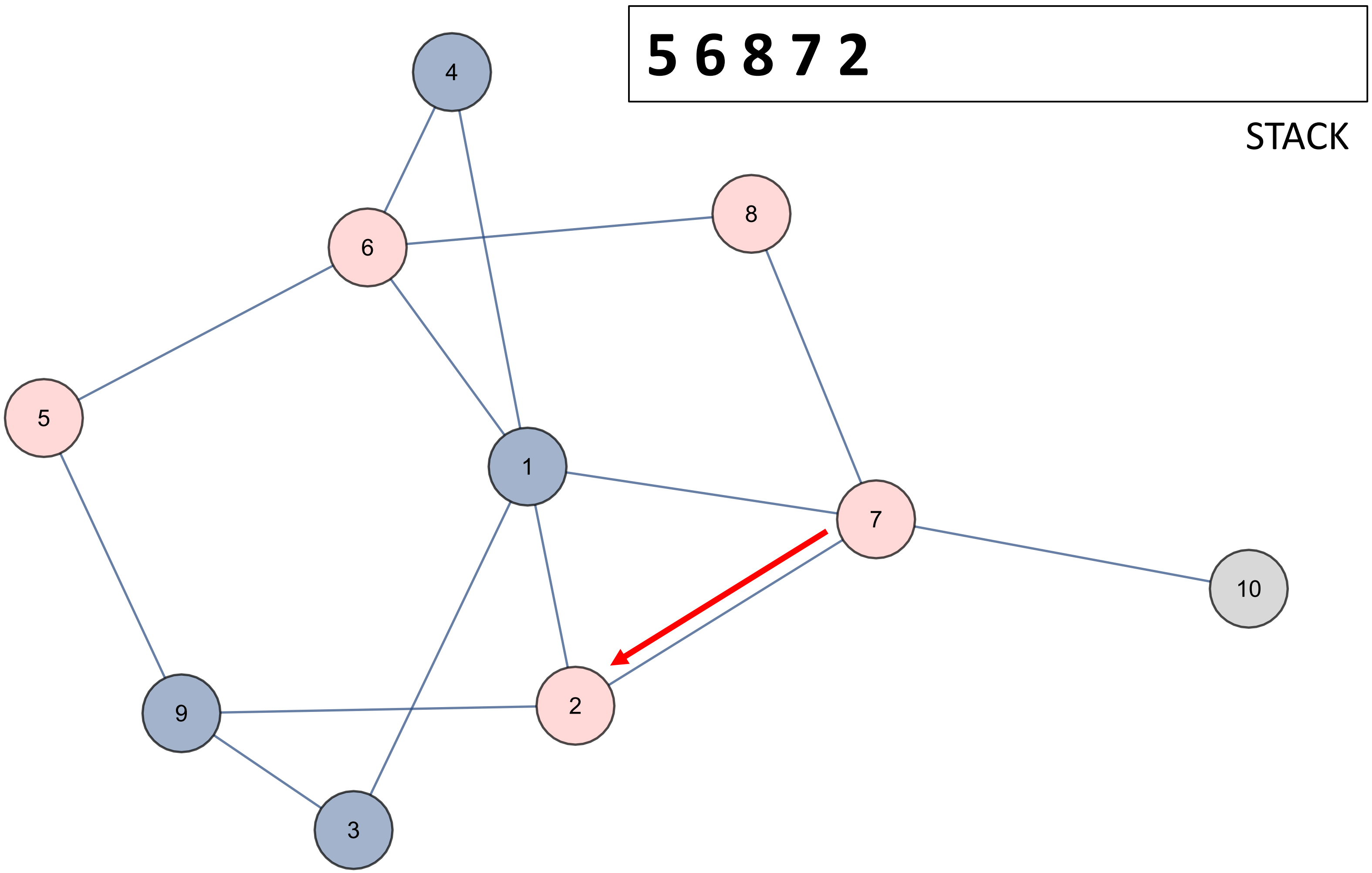
Depth First Search



Depth First Search



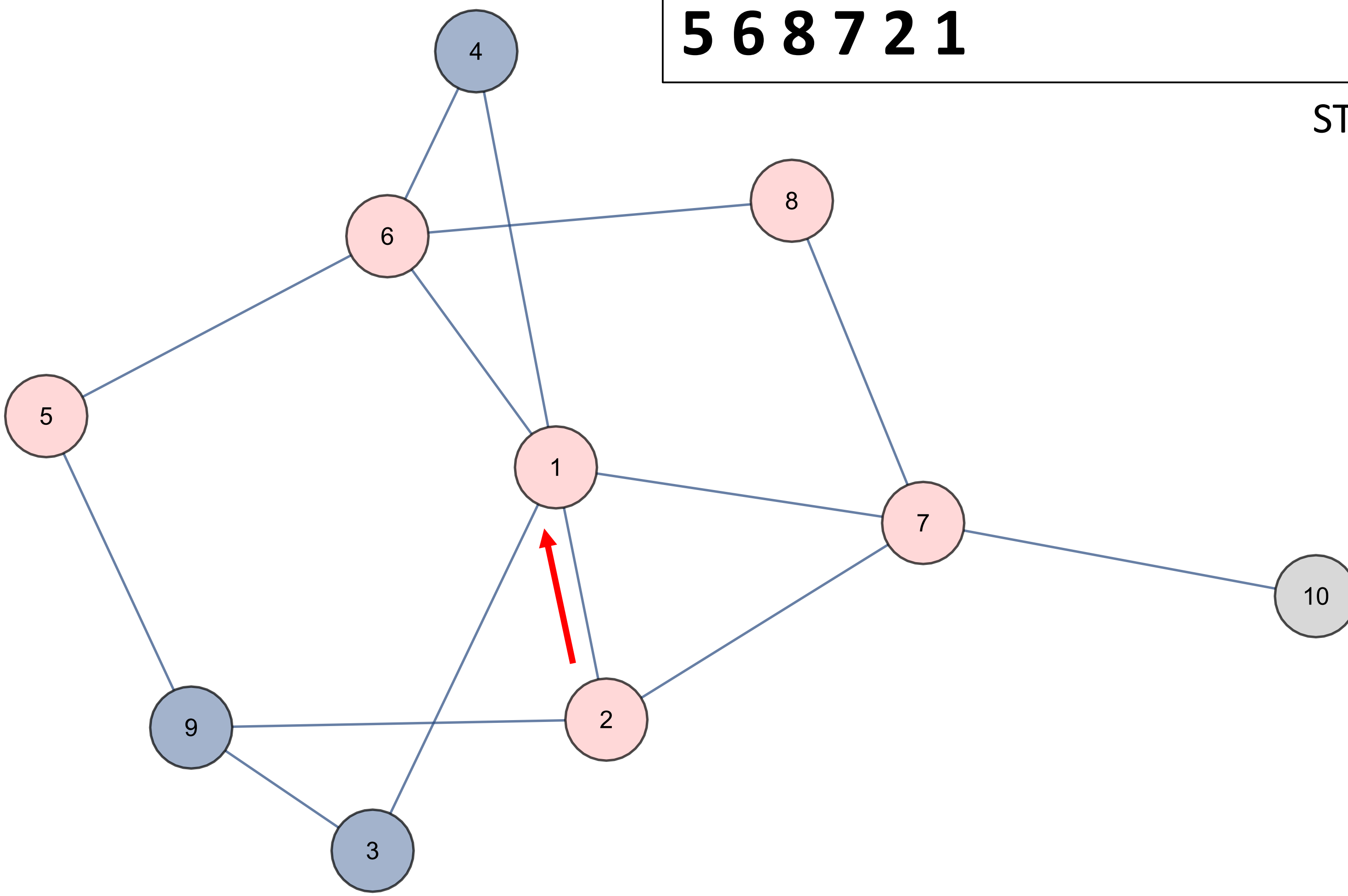
Depth First Search



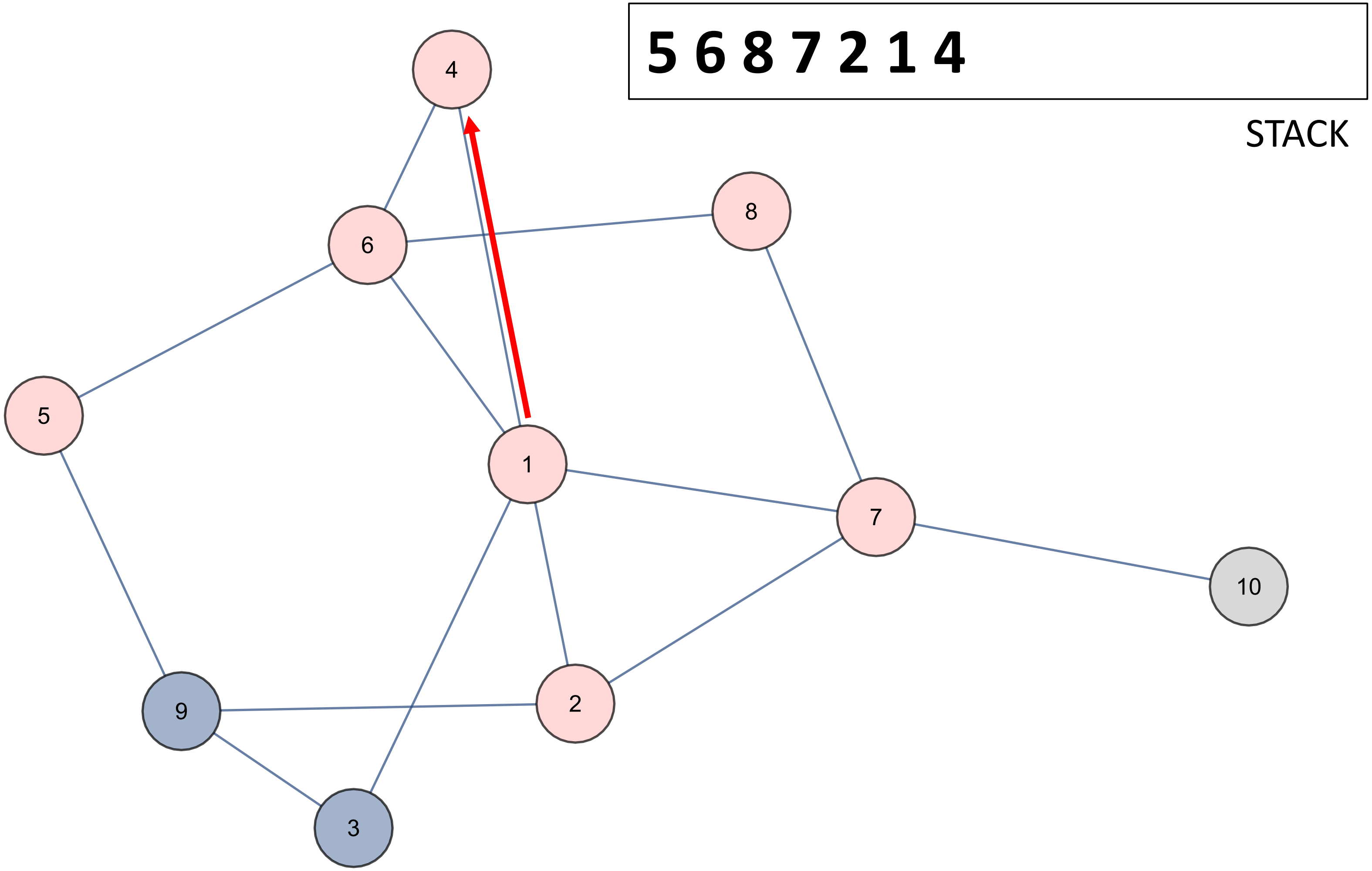
Depth First Search

5 6 8 7 2 1

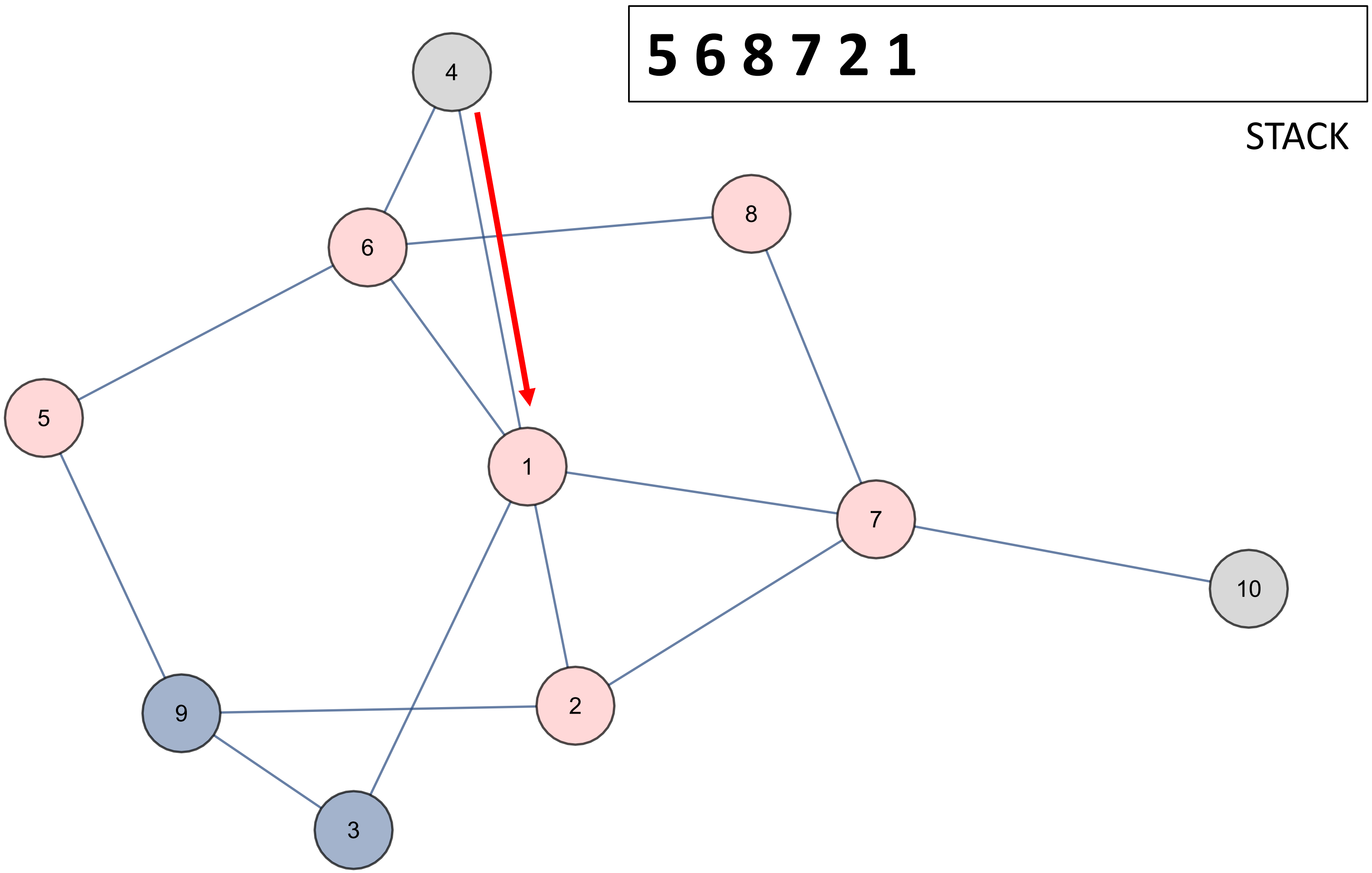
STACK



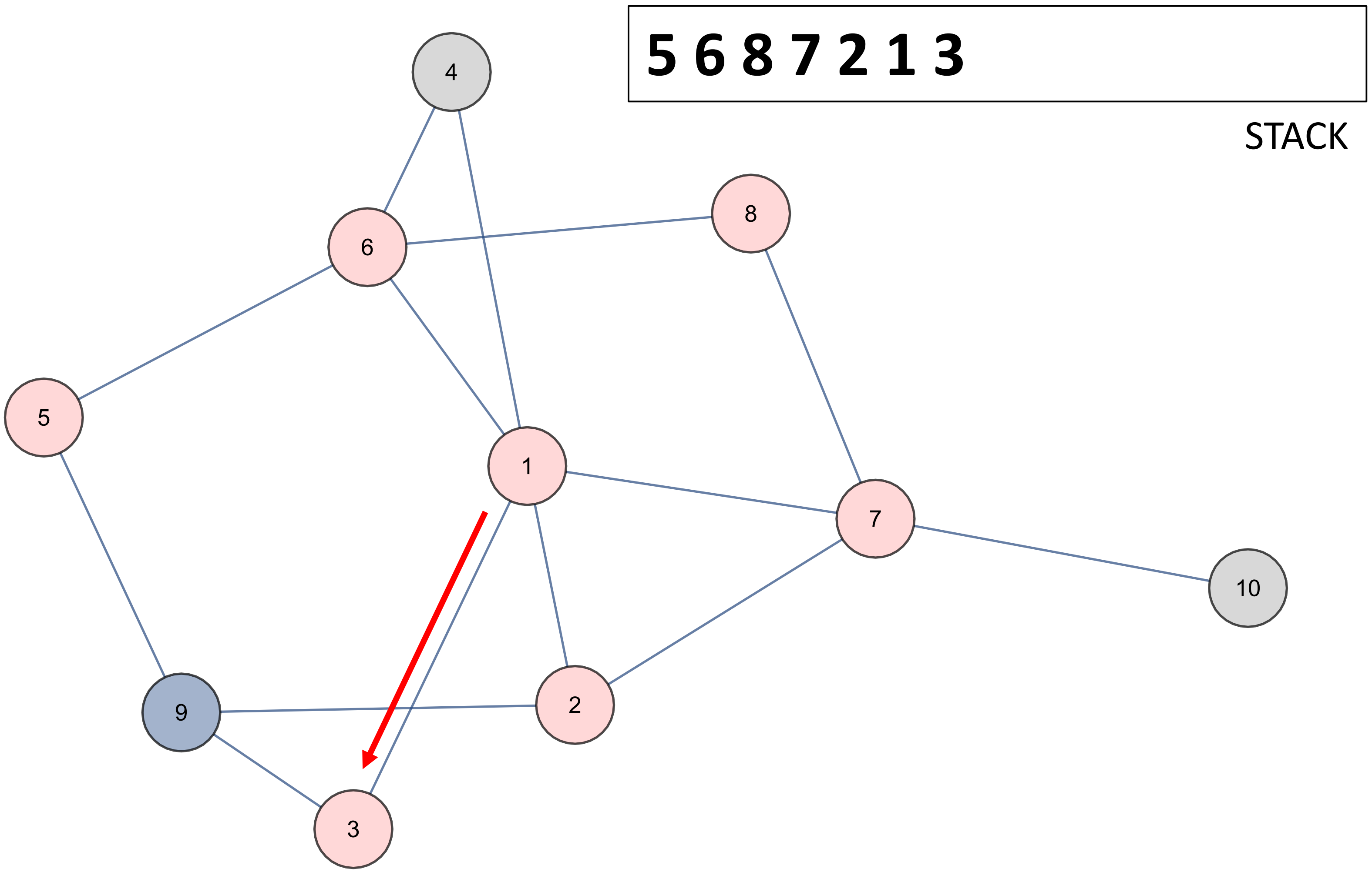
Depth First Search



Depth First Search



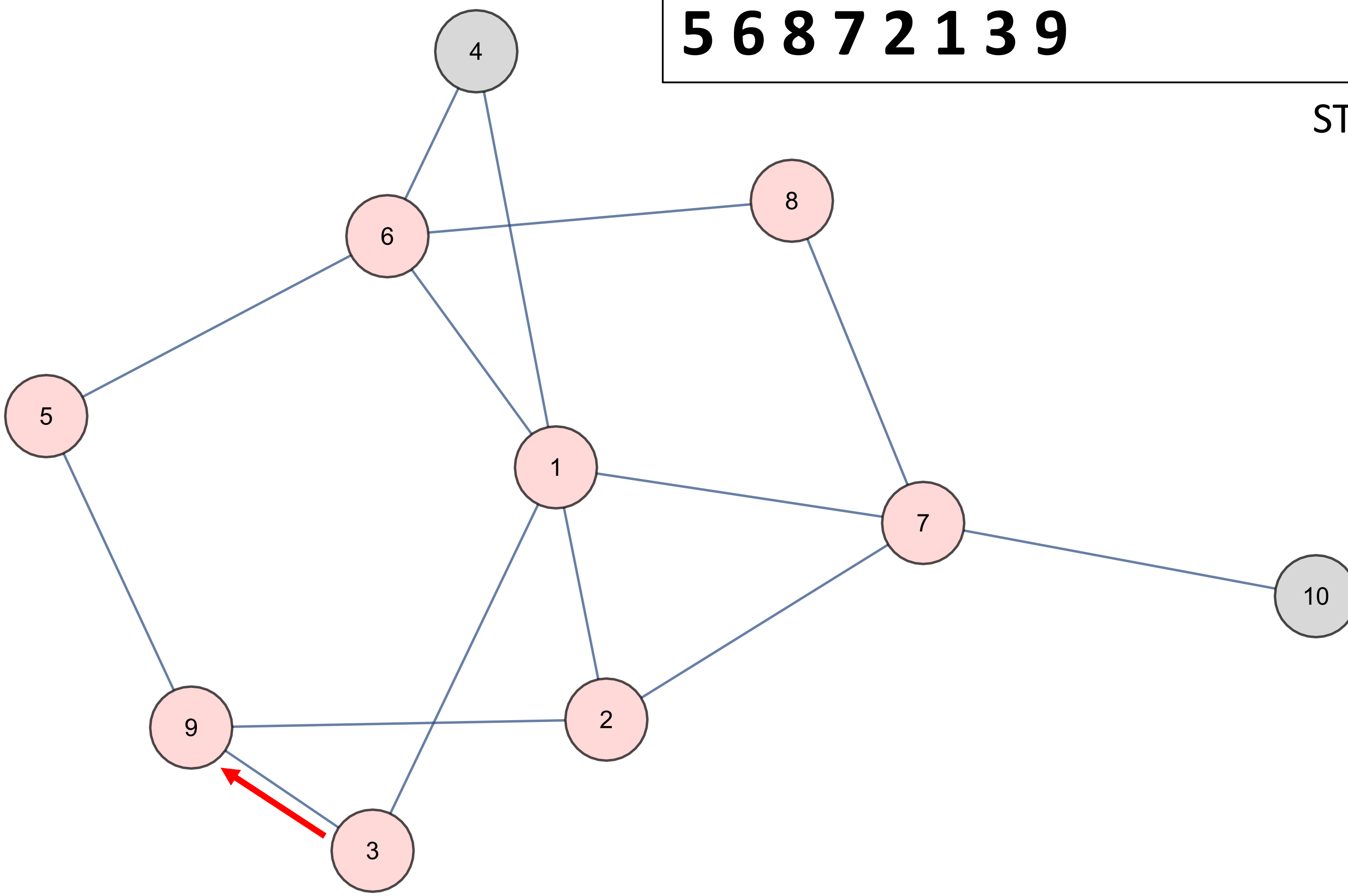
Depth First Search



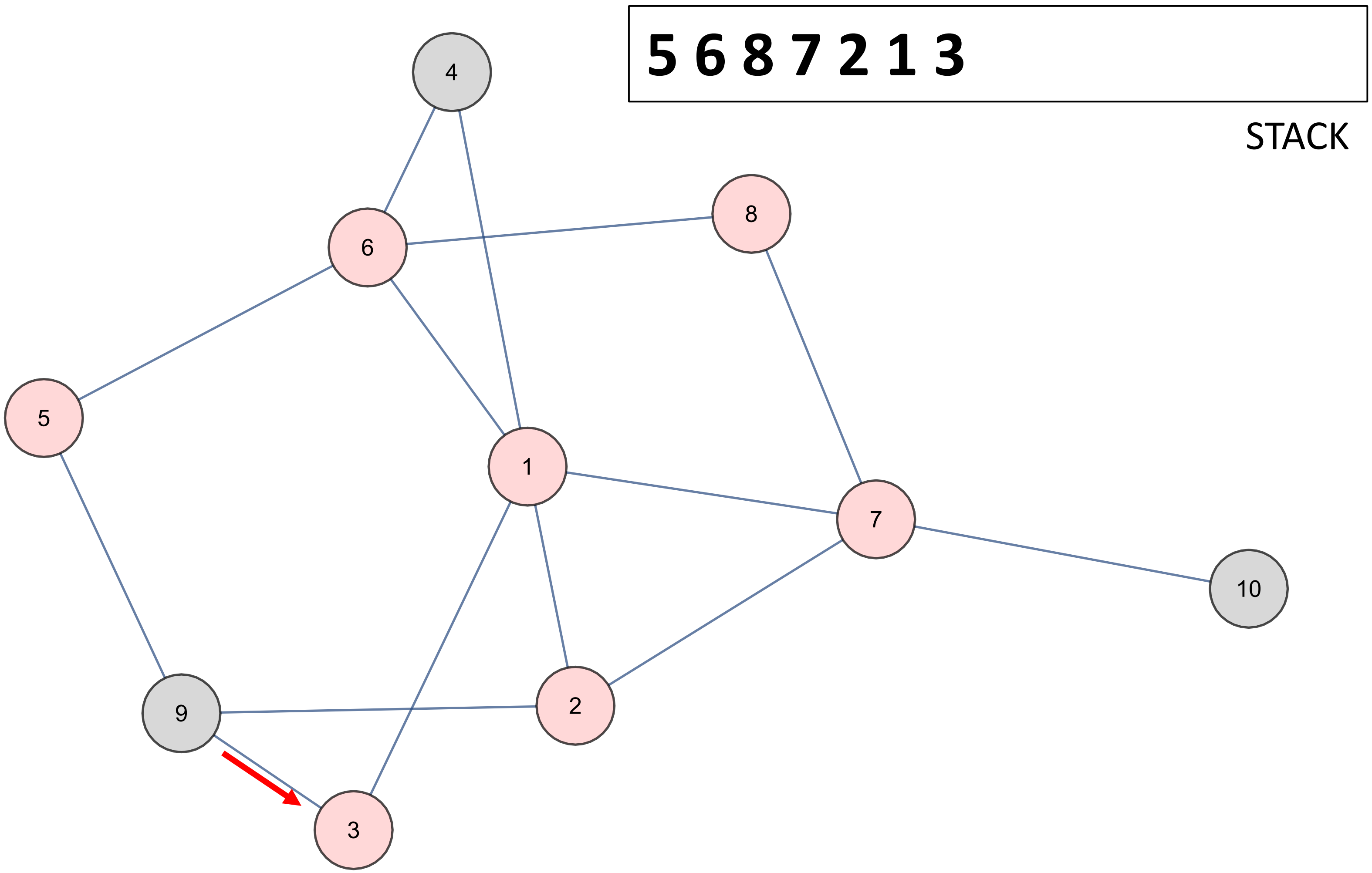
Depth First Search

5 6 8 7 2 1 3 9

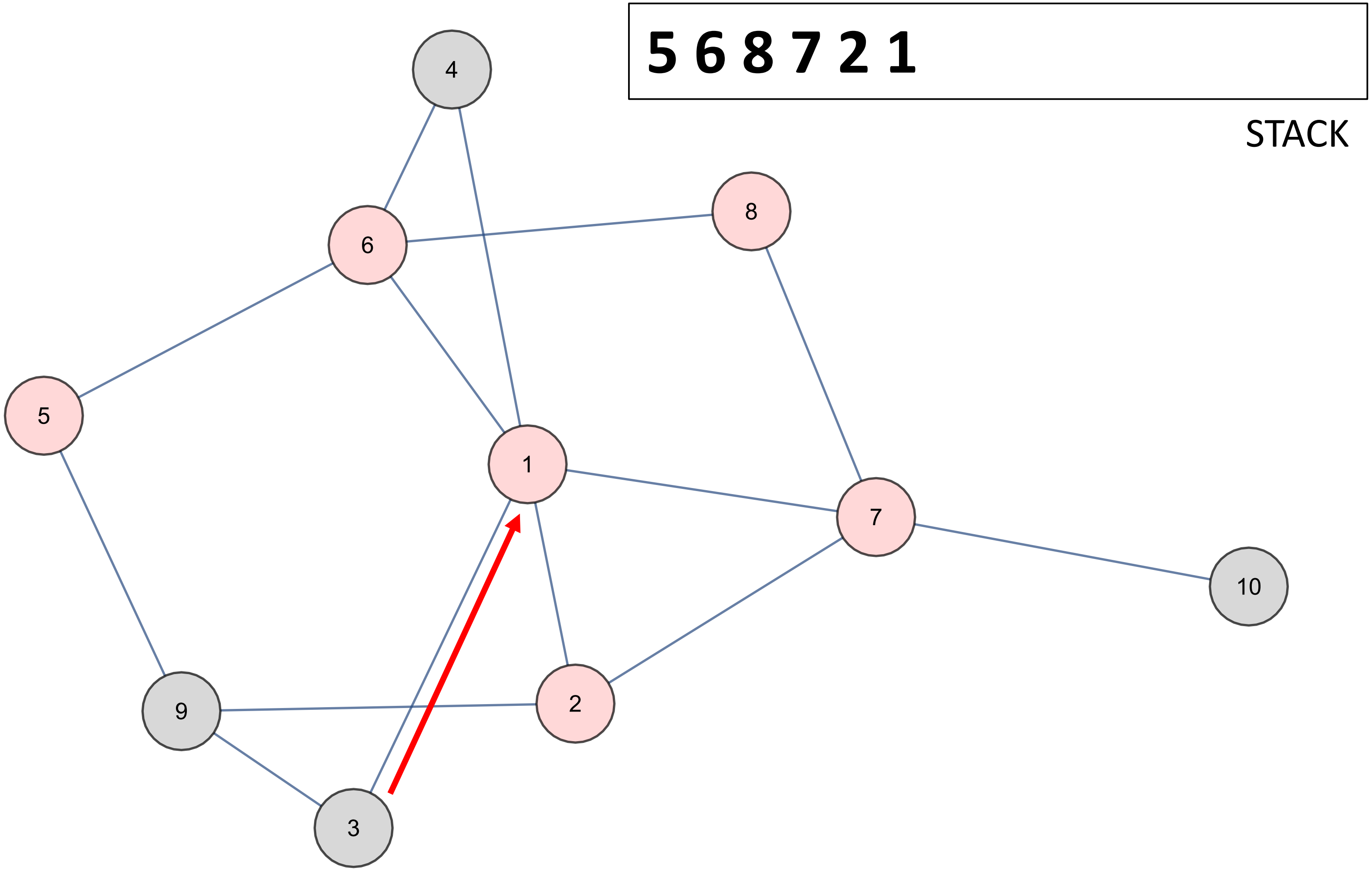
STACK



Depth First Search



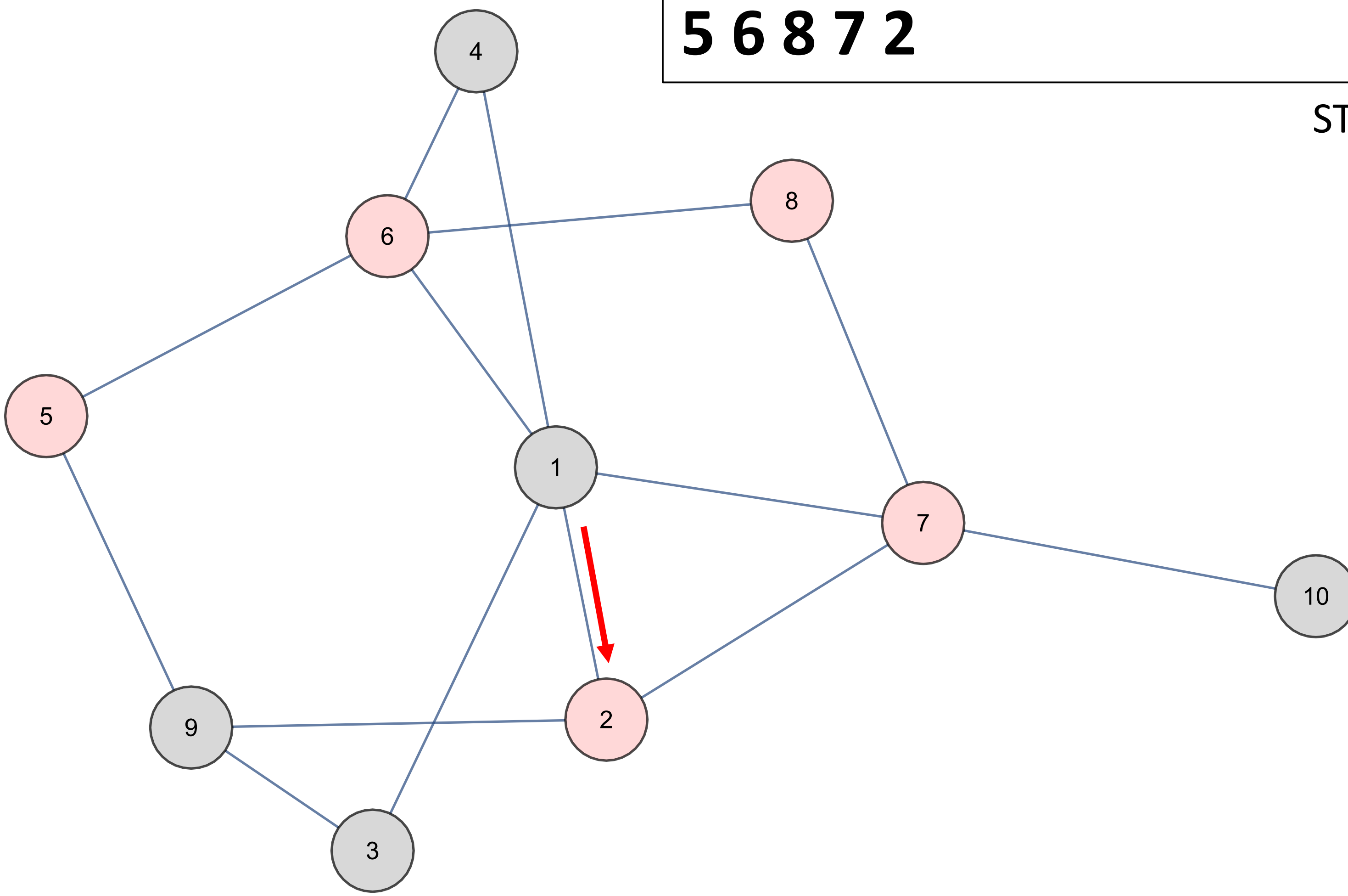
Depth First Search



Depth First Search

5 6 8 7 2

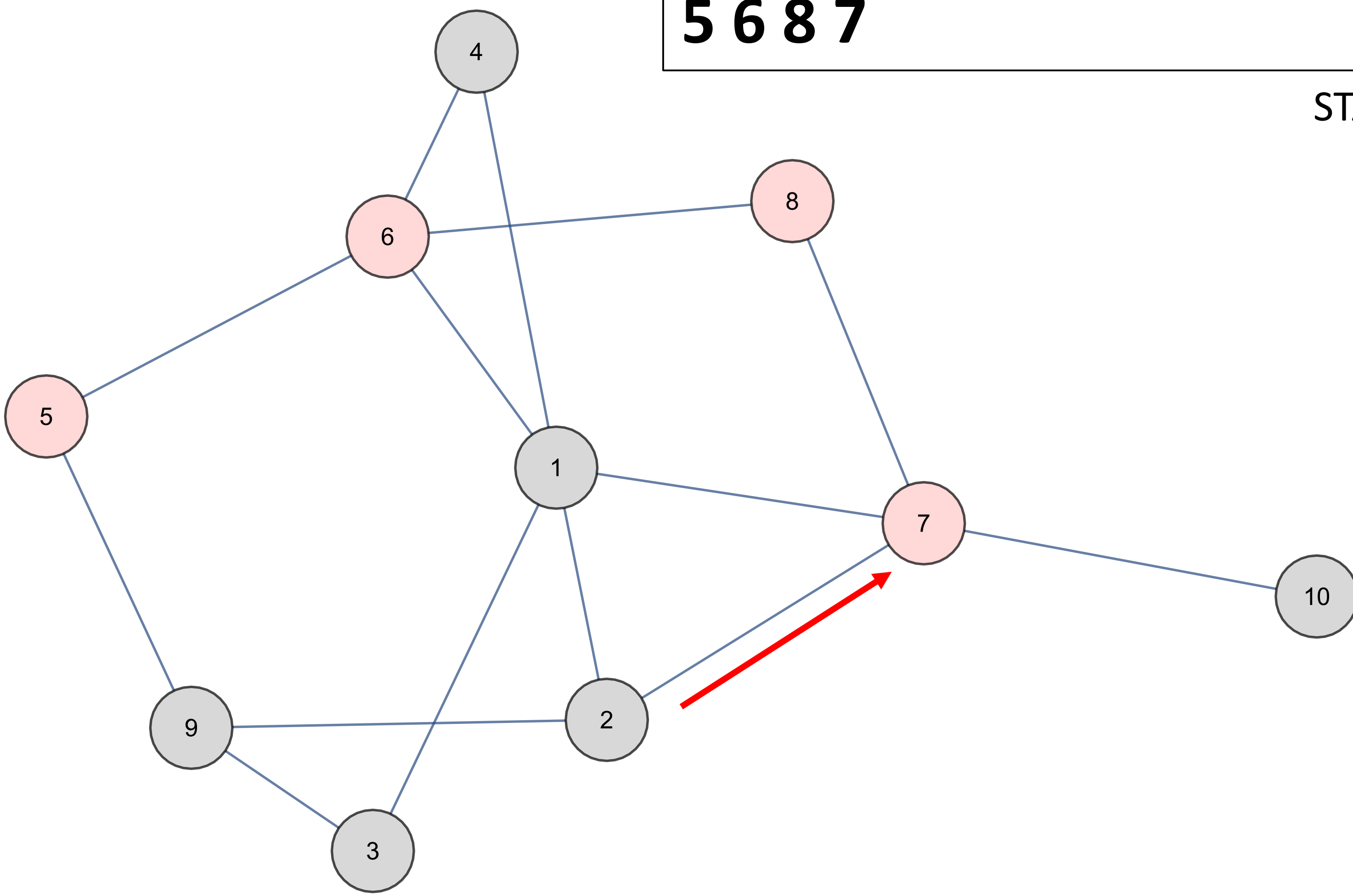
STACK



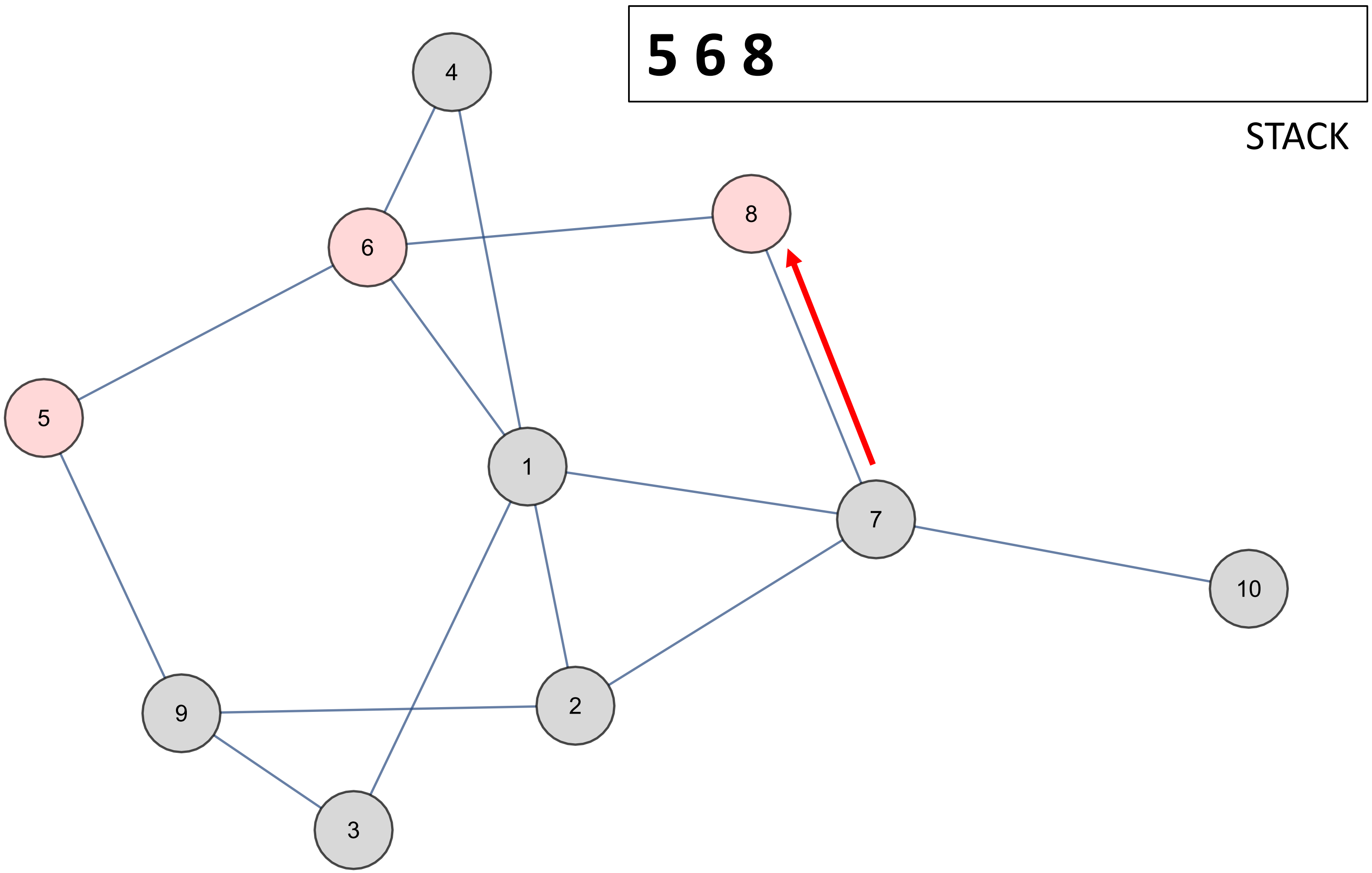
Depth First Search

5 6 8 7

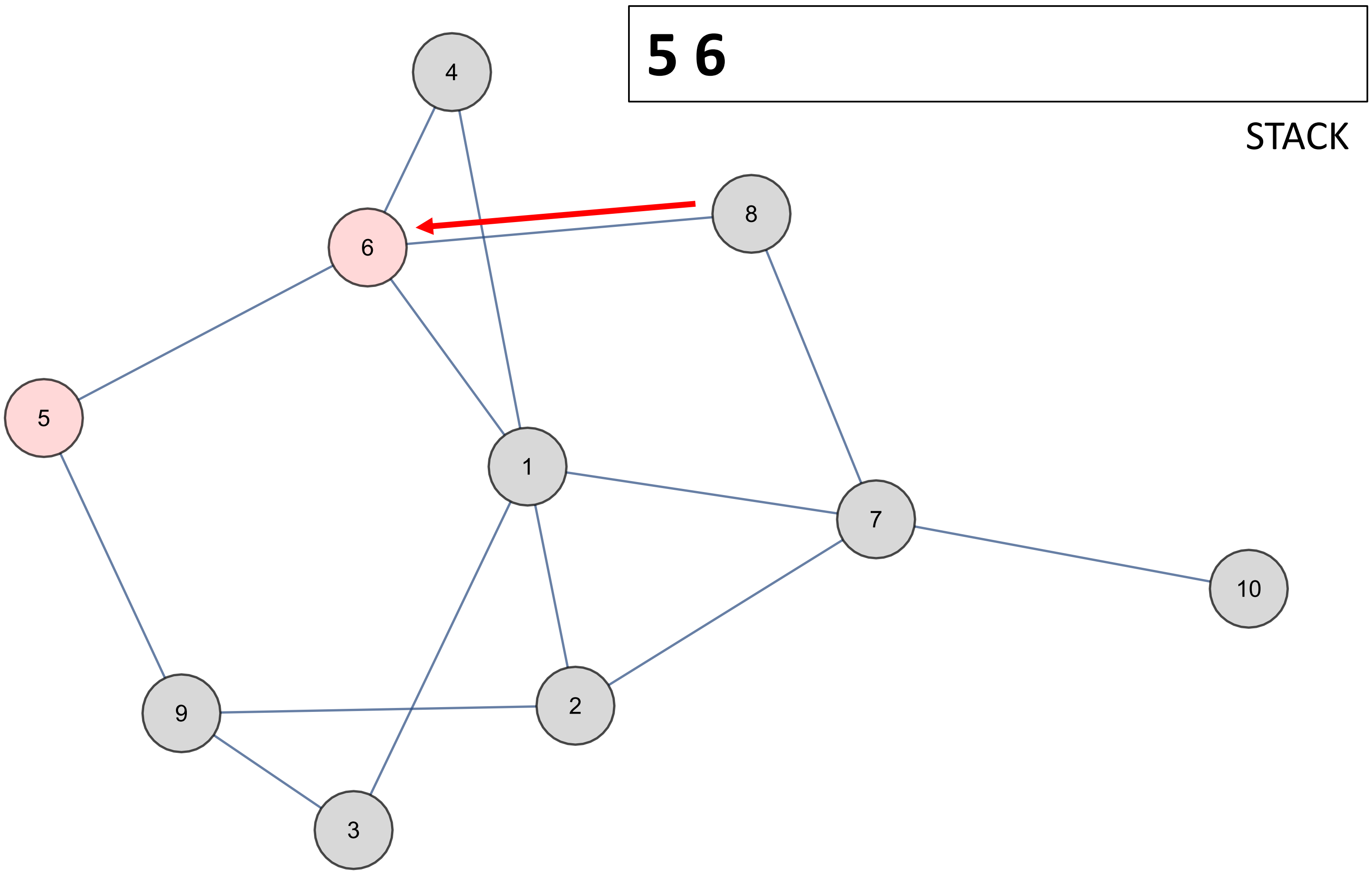
STACK



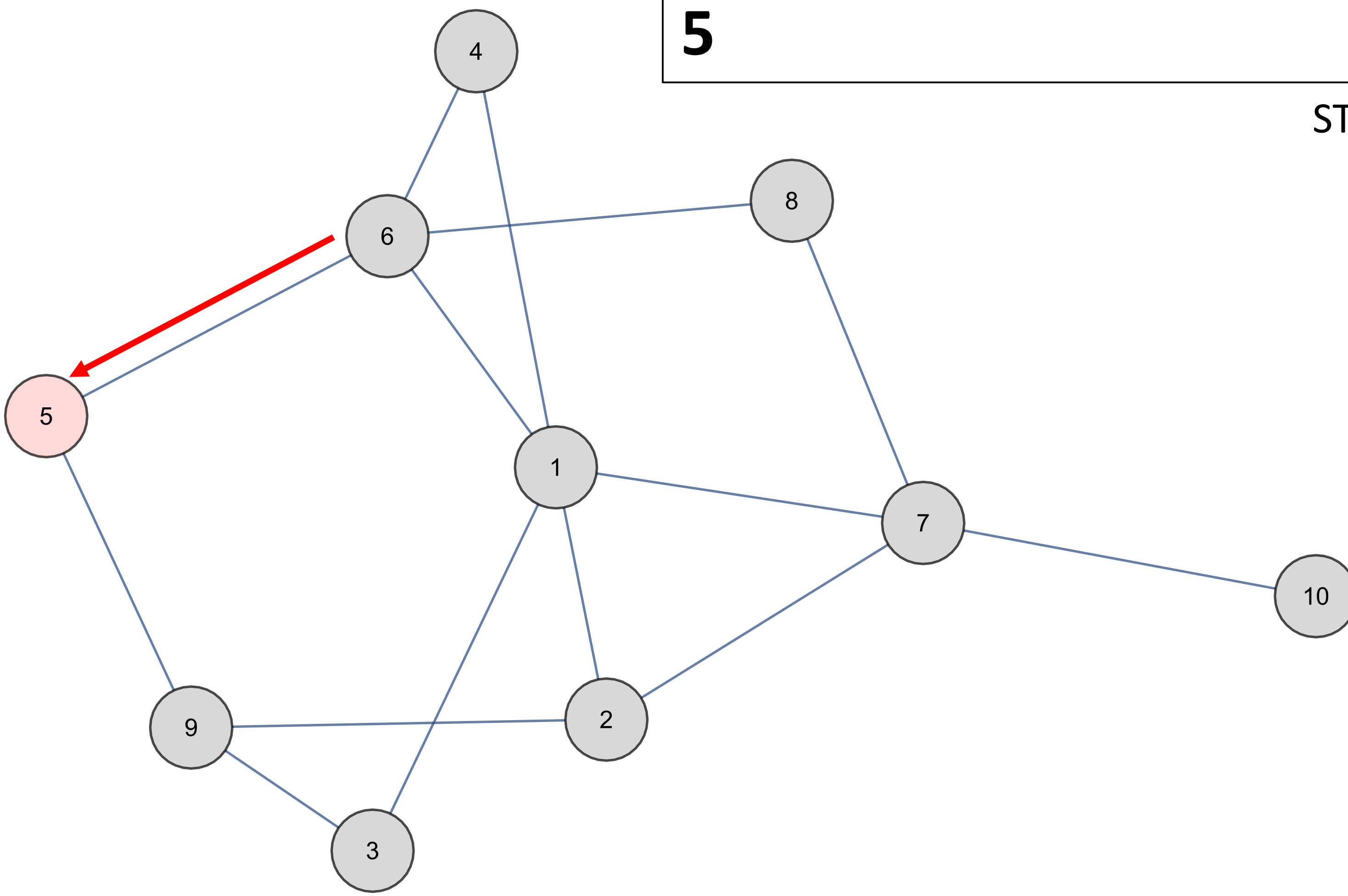
Depth First Search



Depth First Search



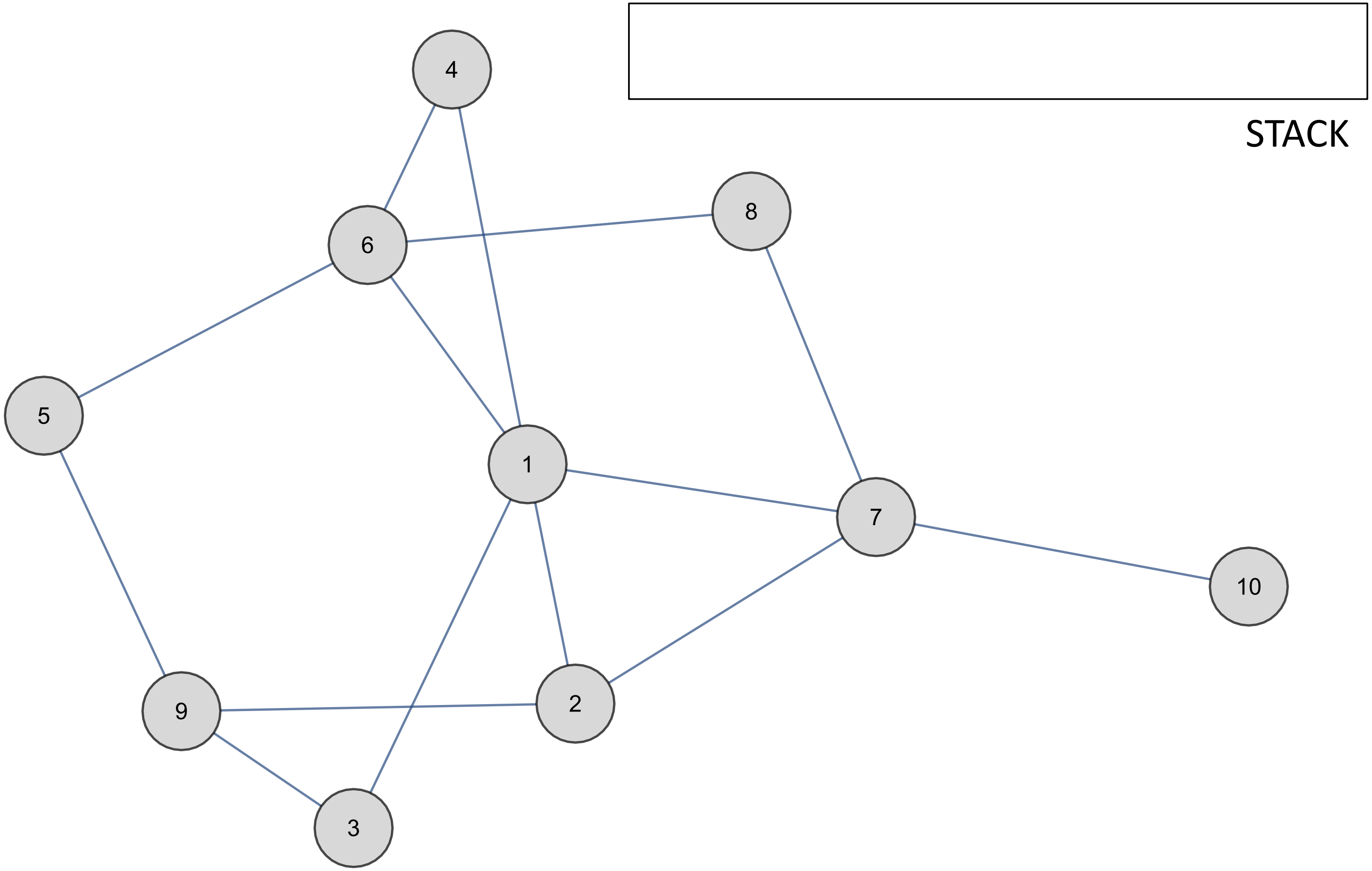
Depth First Search



5

STACK

Depth First Search

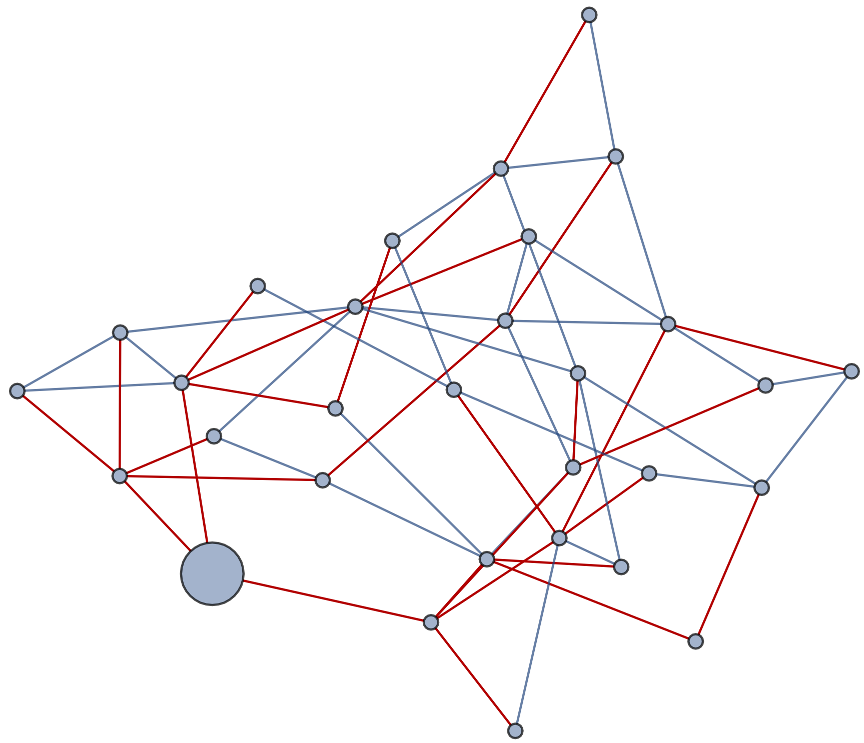


Depth First Search

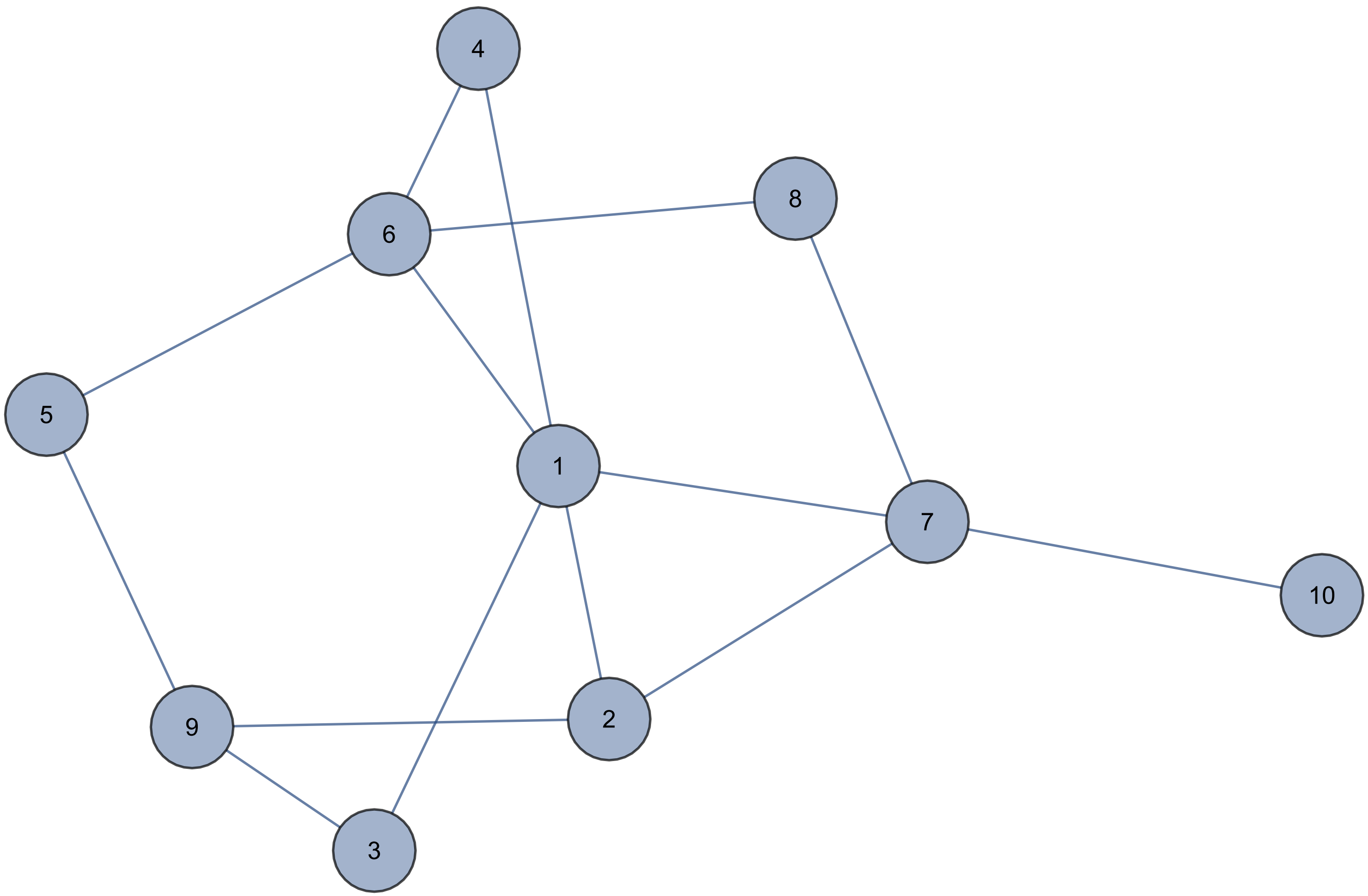
```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

function dfs(at):
  if visited[at]: return
  visited[at] = true

  neighbours = graph[at]
  for next in neighbours:
    dfs(next)

# Start DFS at node zero
start_node = 0
dfs(start_node)
```

Breadth First Search

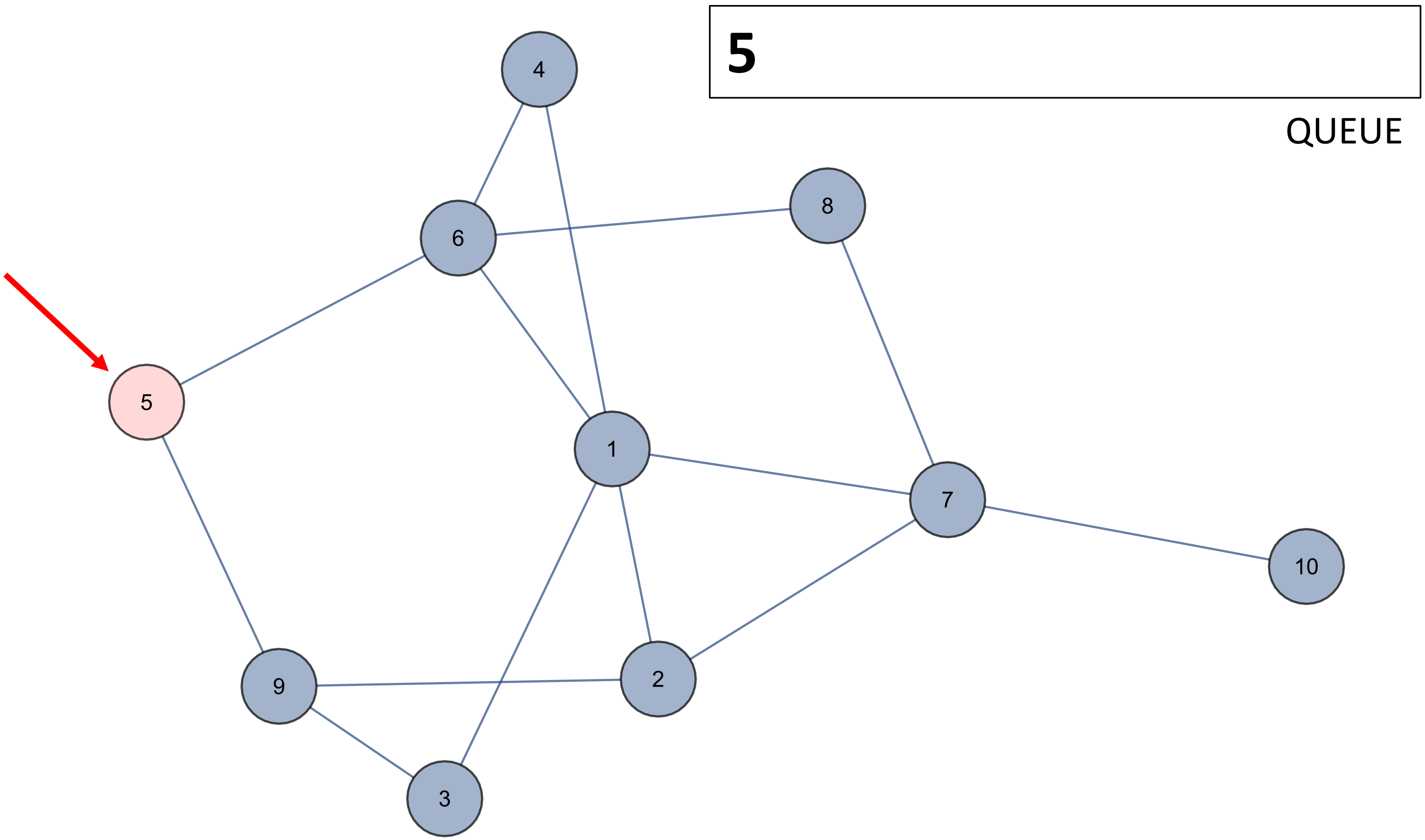
Breadth-first search (**BFS**) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level.



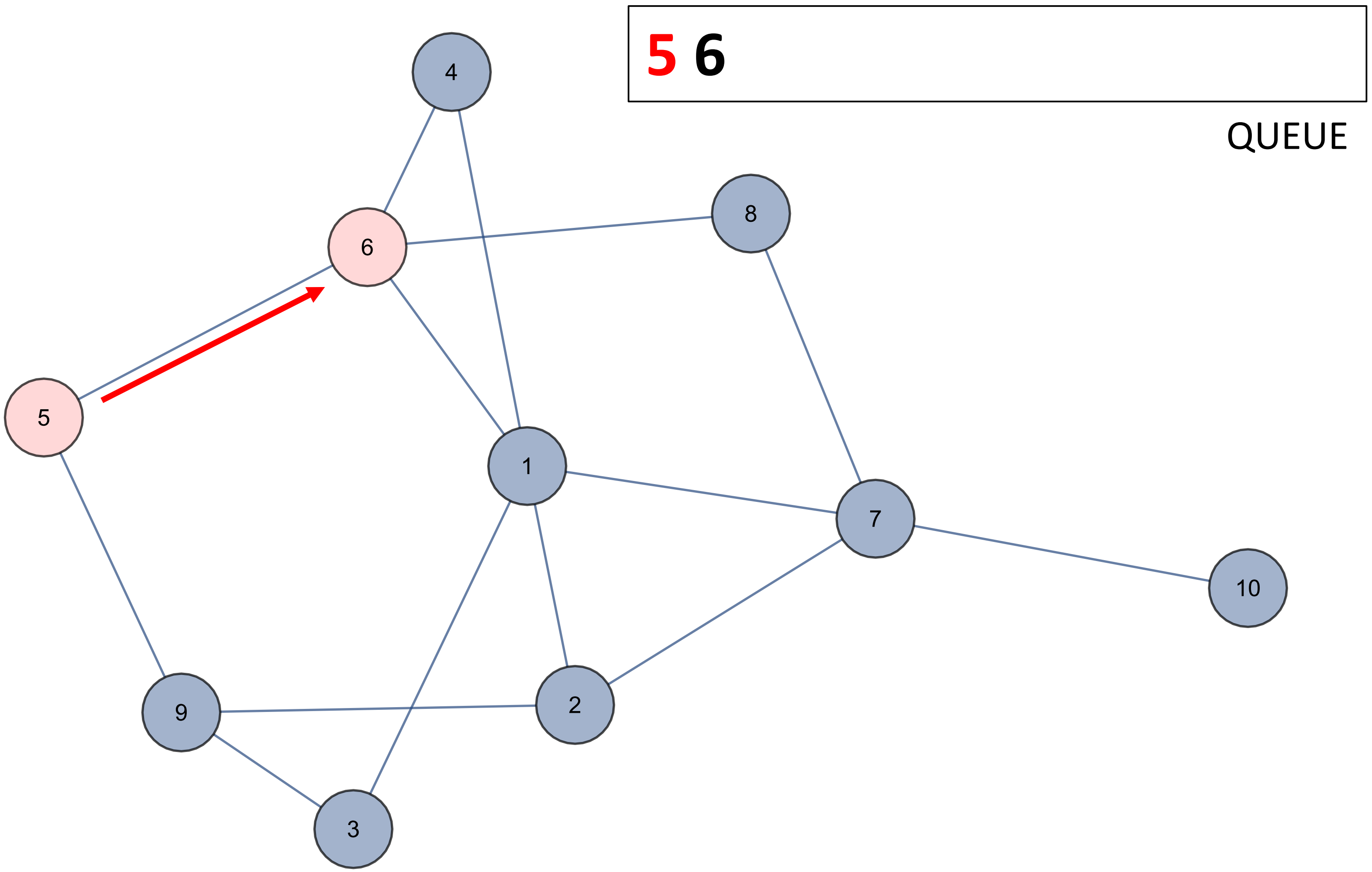
Breadth First Search



Breadth First Search



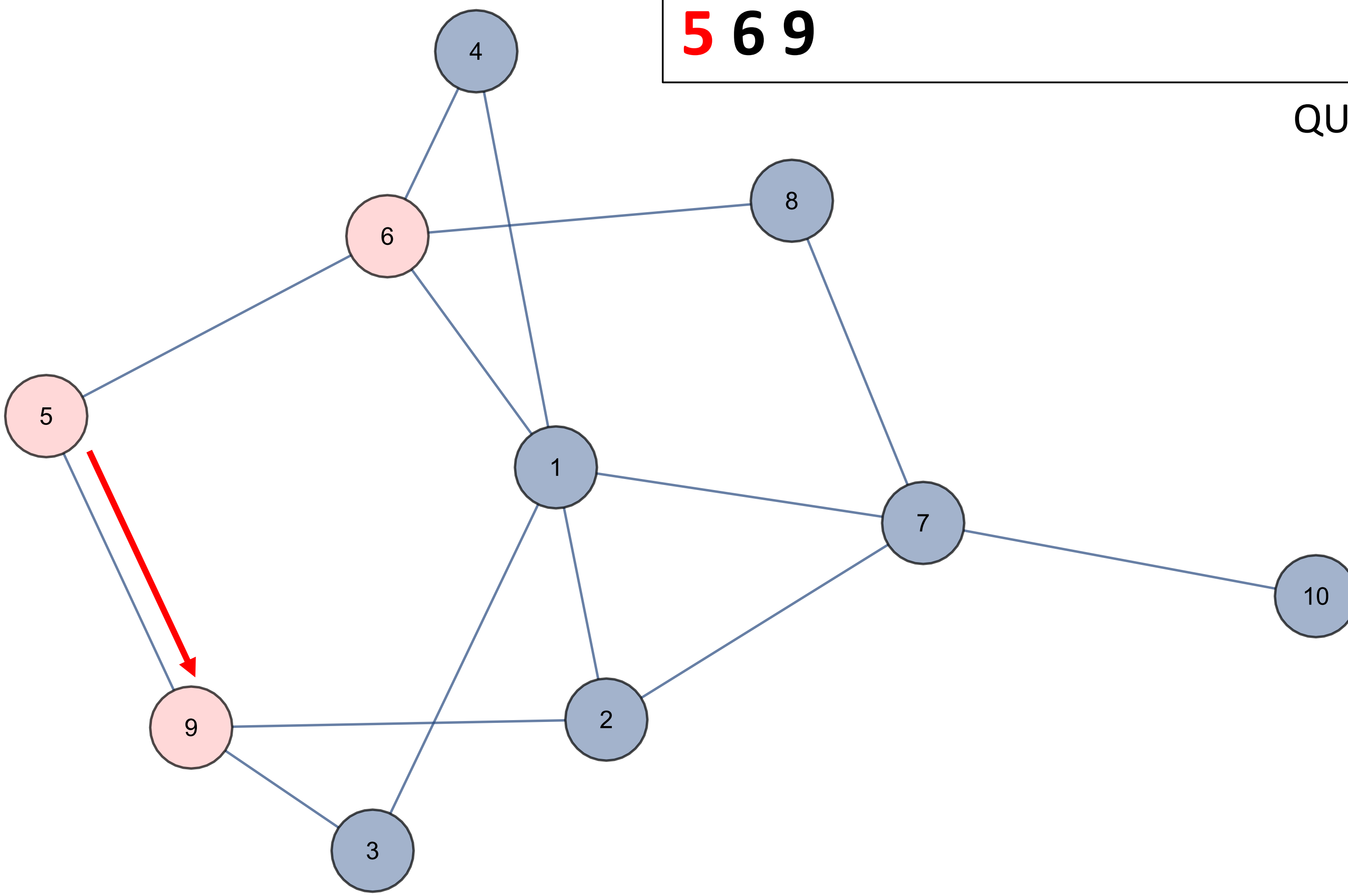
Breadth First Search



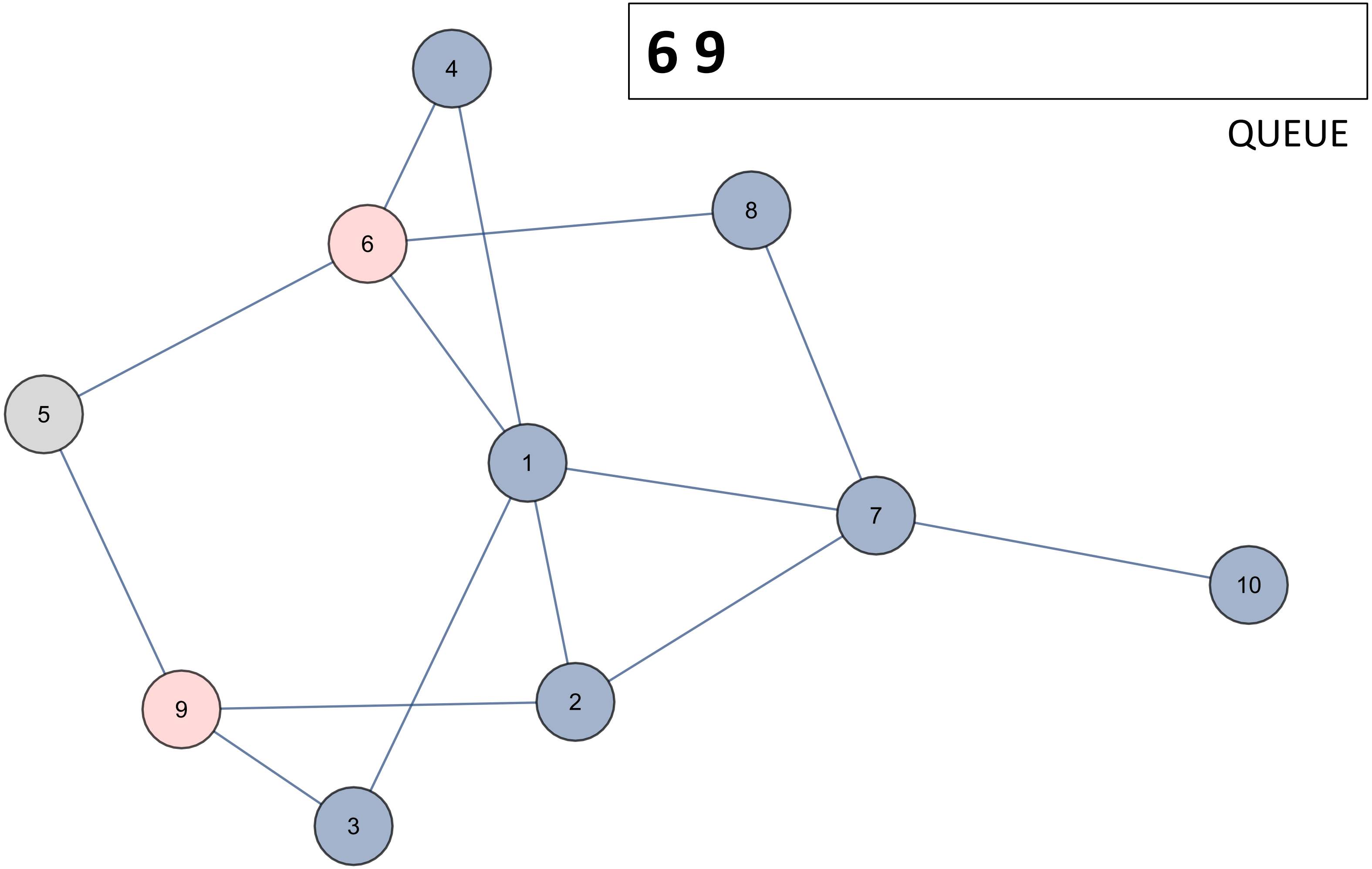
Breadth First Search

5 6 9

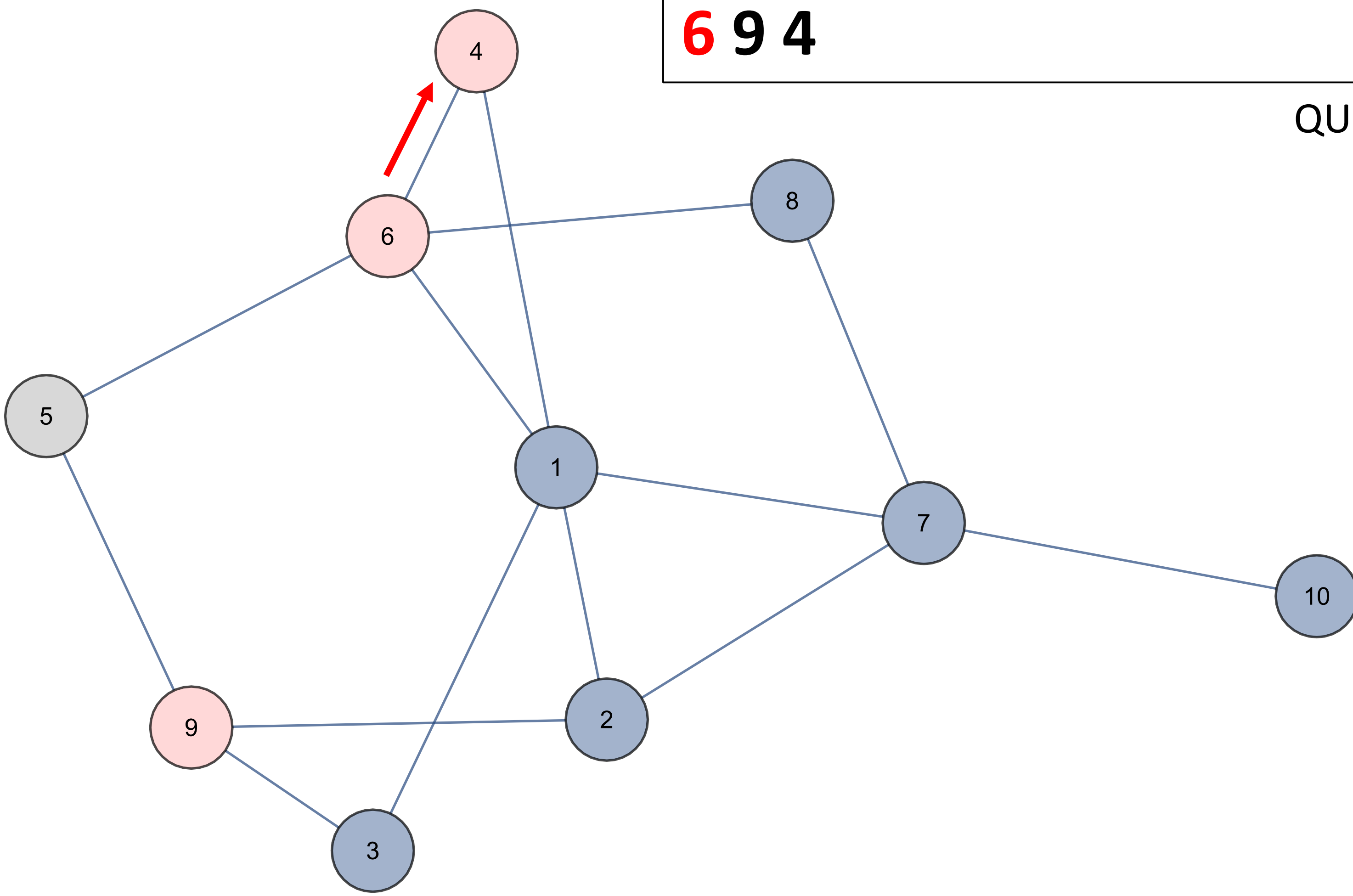
QUEUE



Breadth First Search



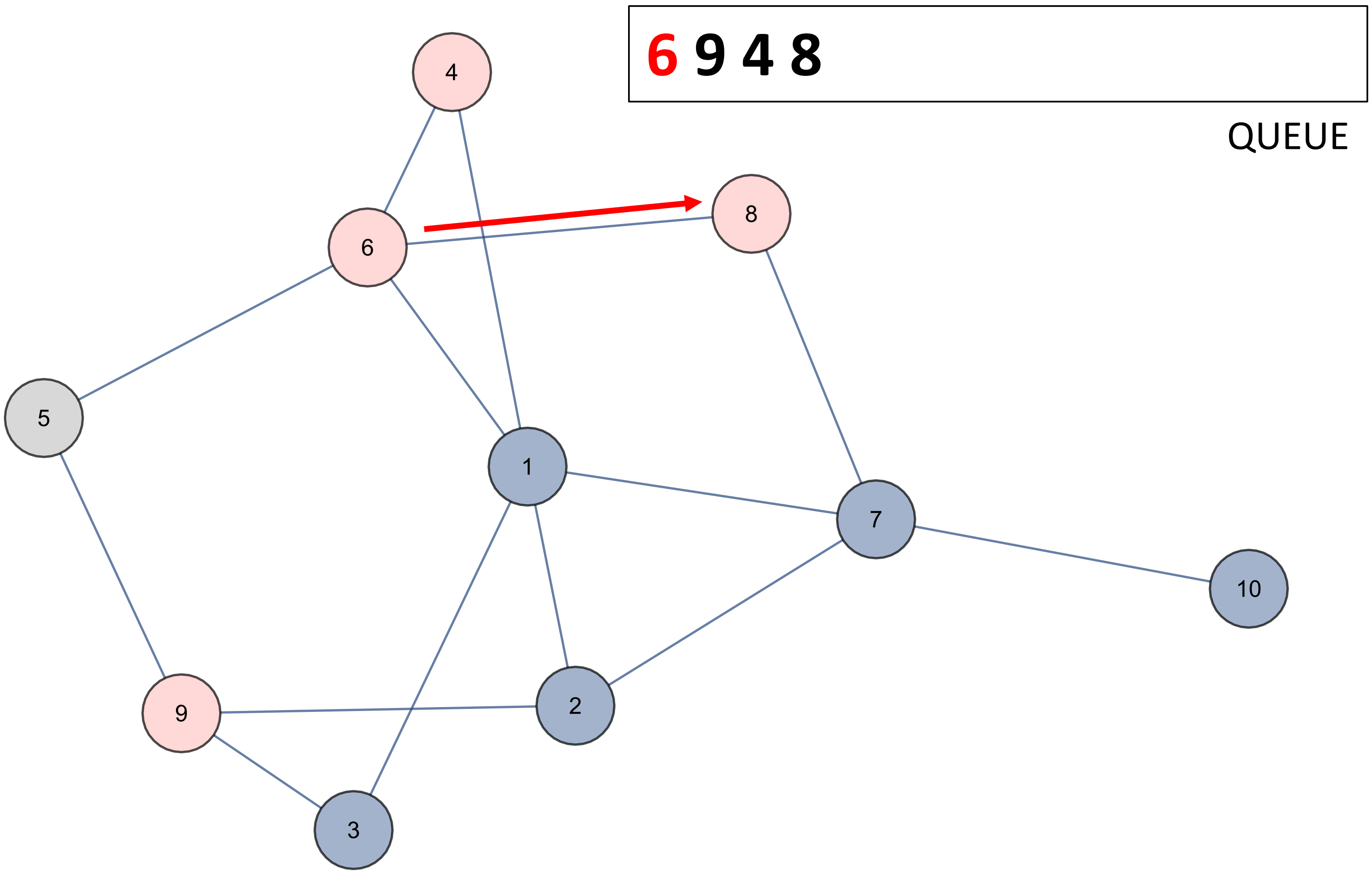
Breadth First Search



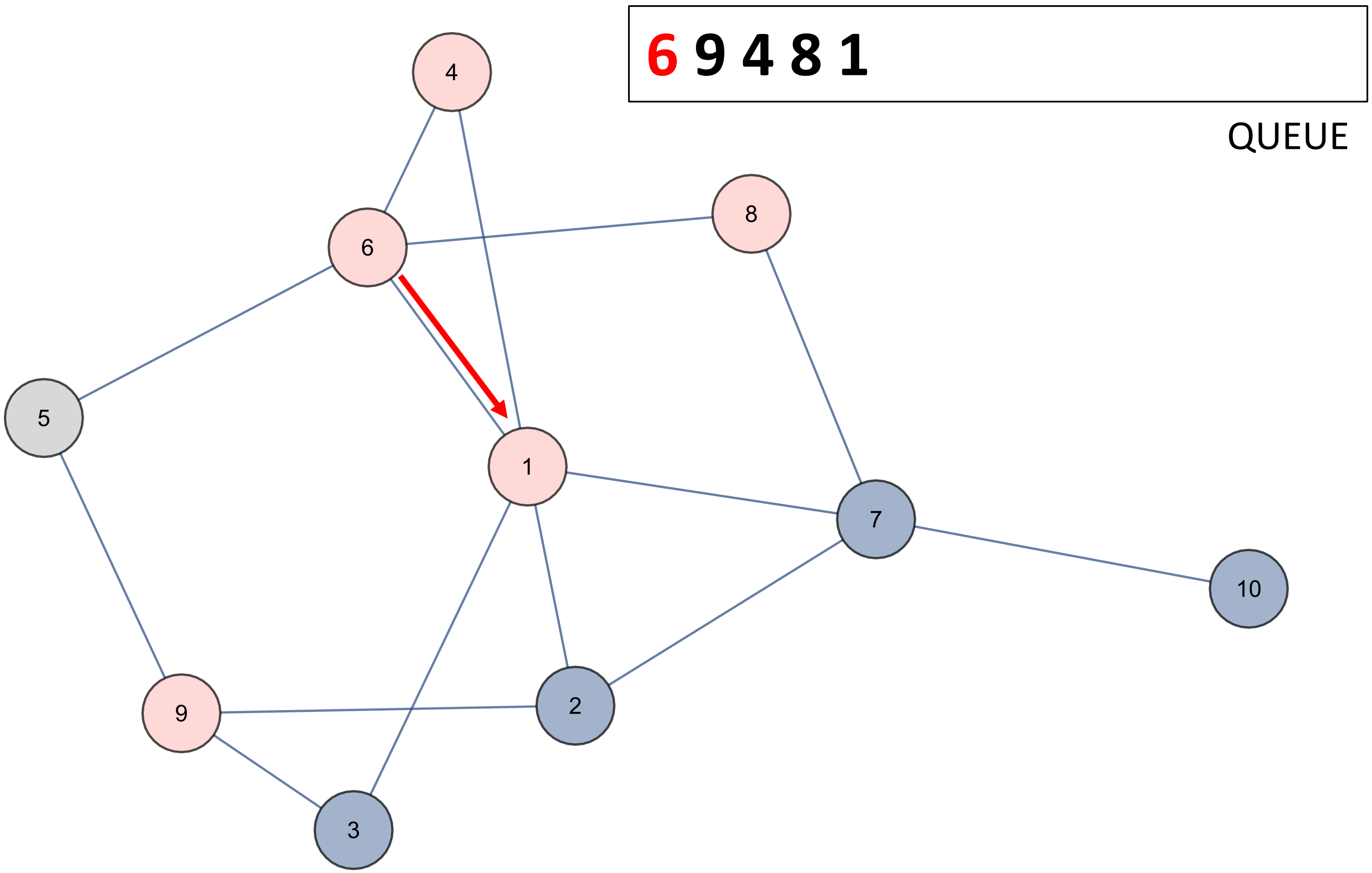
6 9 4

QUEUE

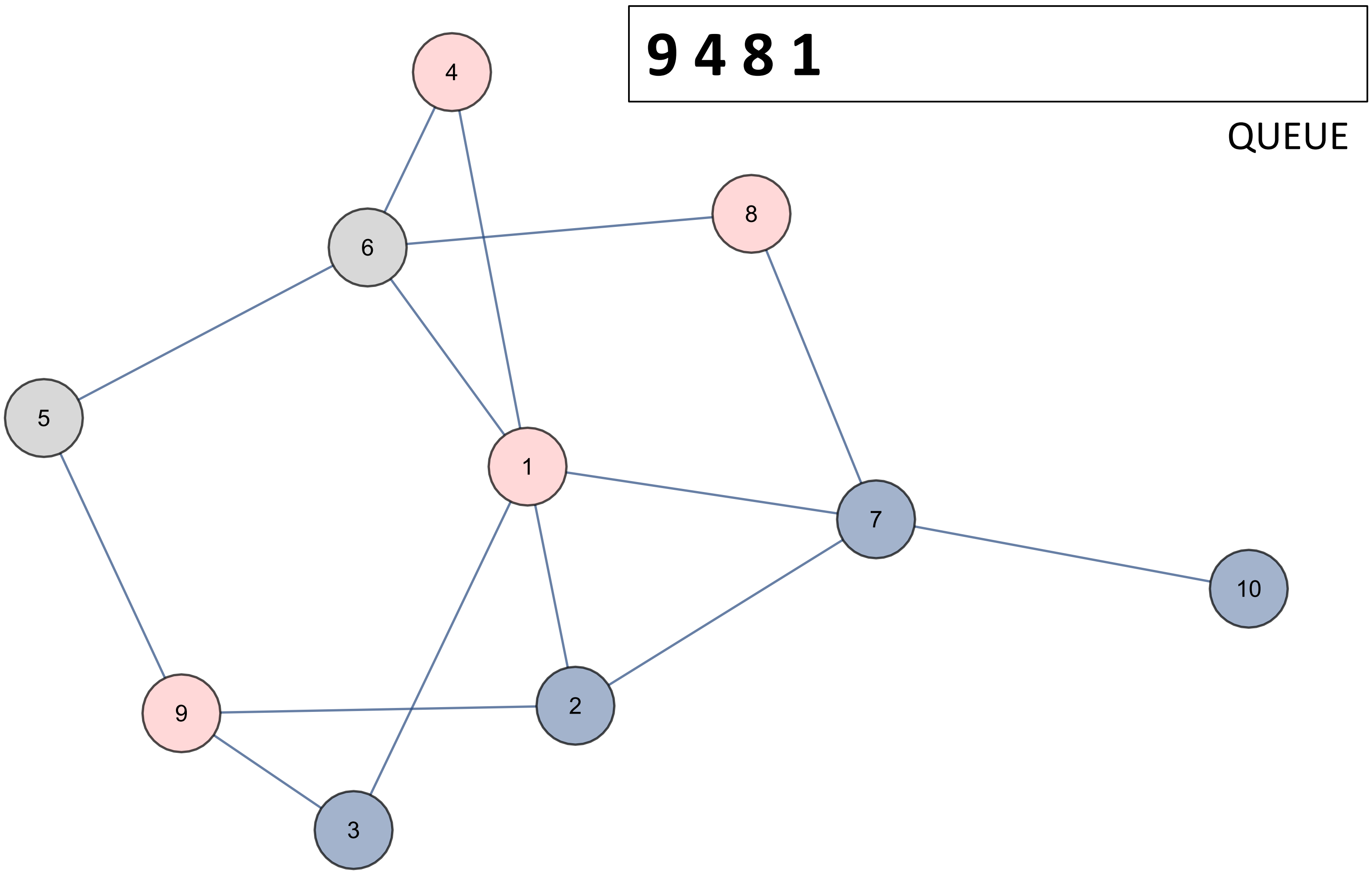
Breadth First Search



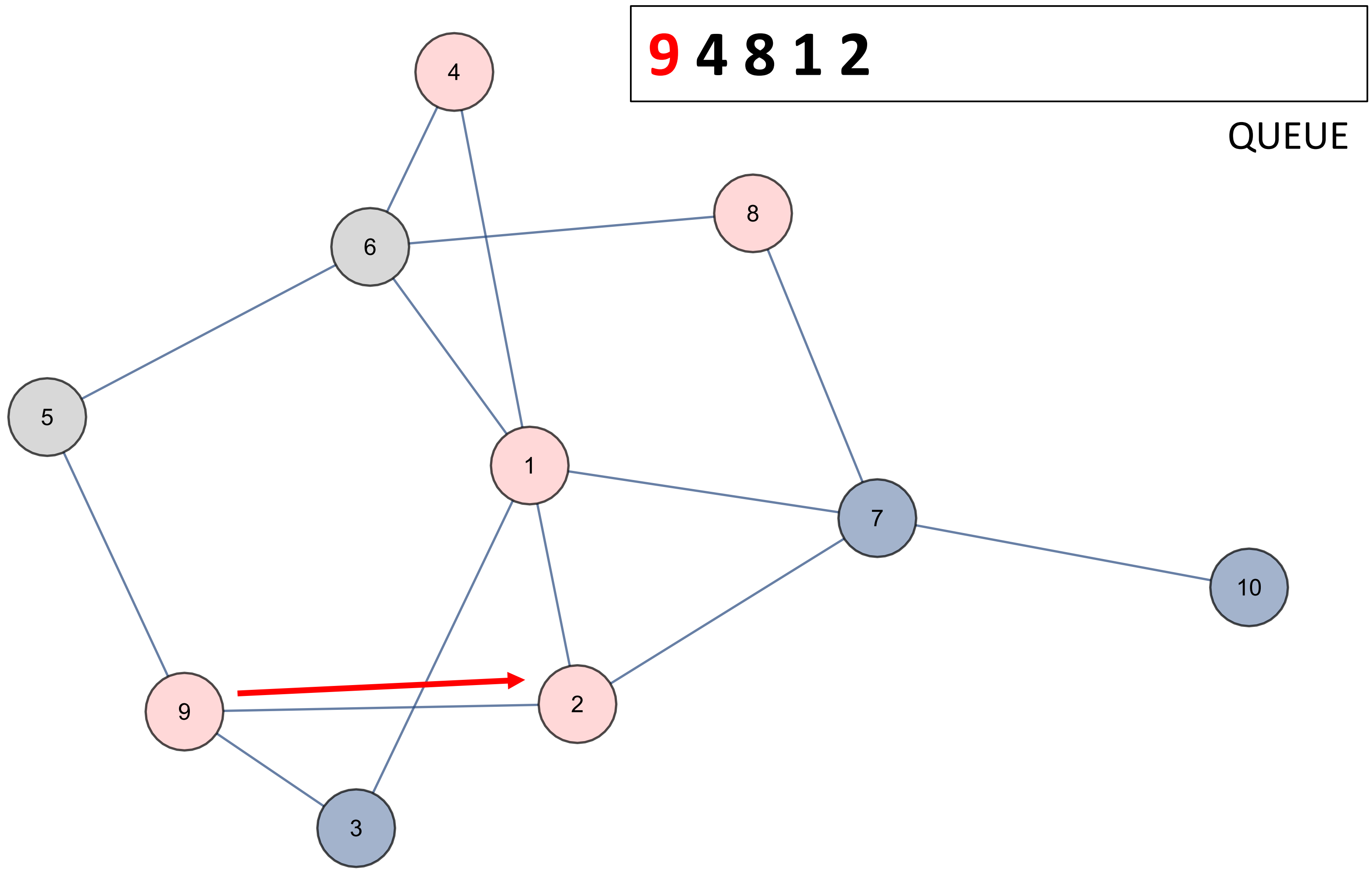
Breadth First Search



Breadth First Search



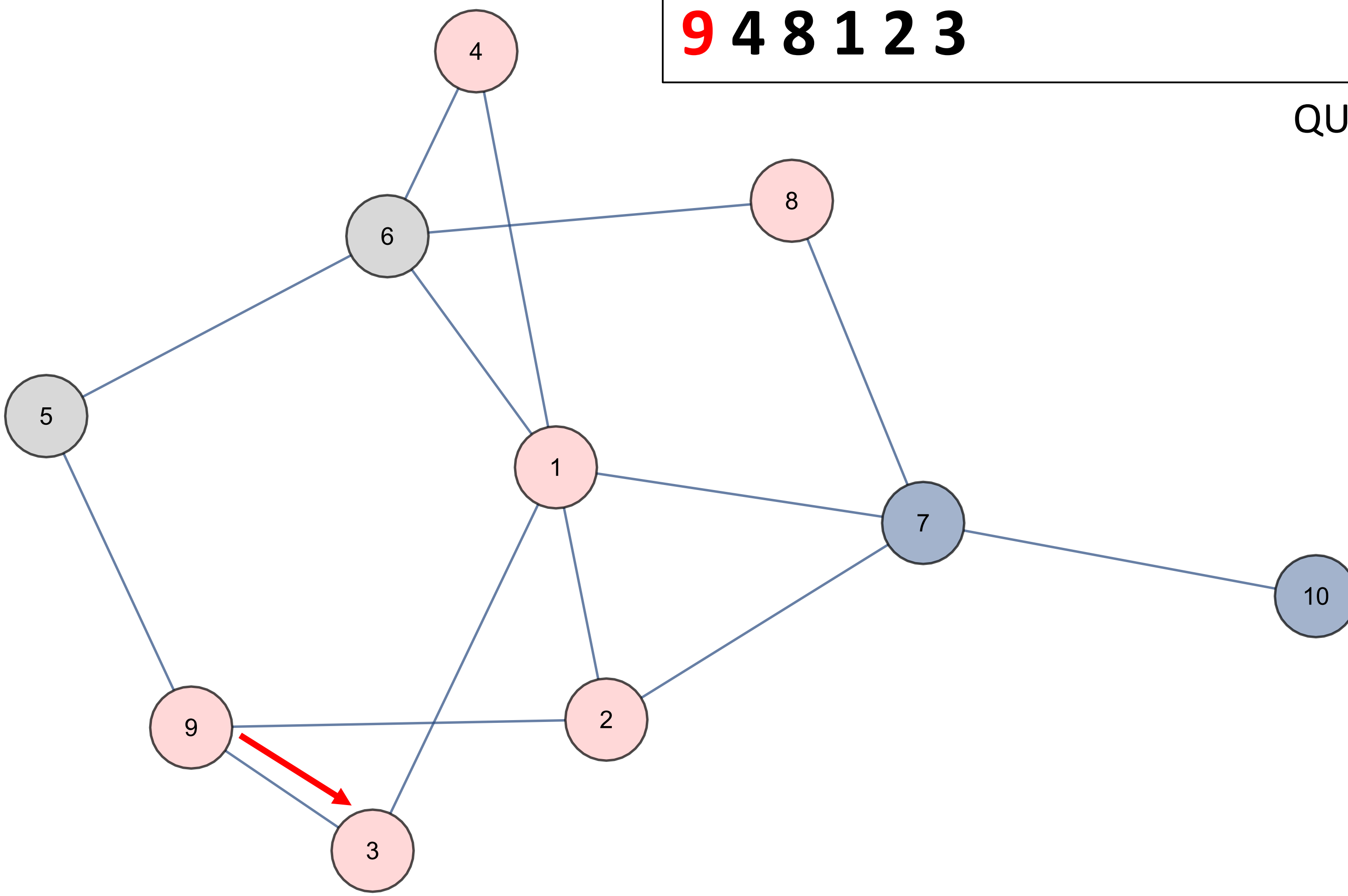
Breadth First Search



Breadth First Search

9 4 8 1 2 3

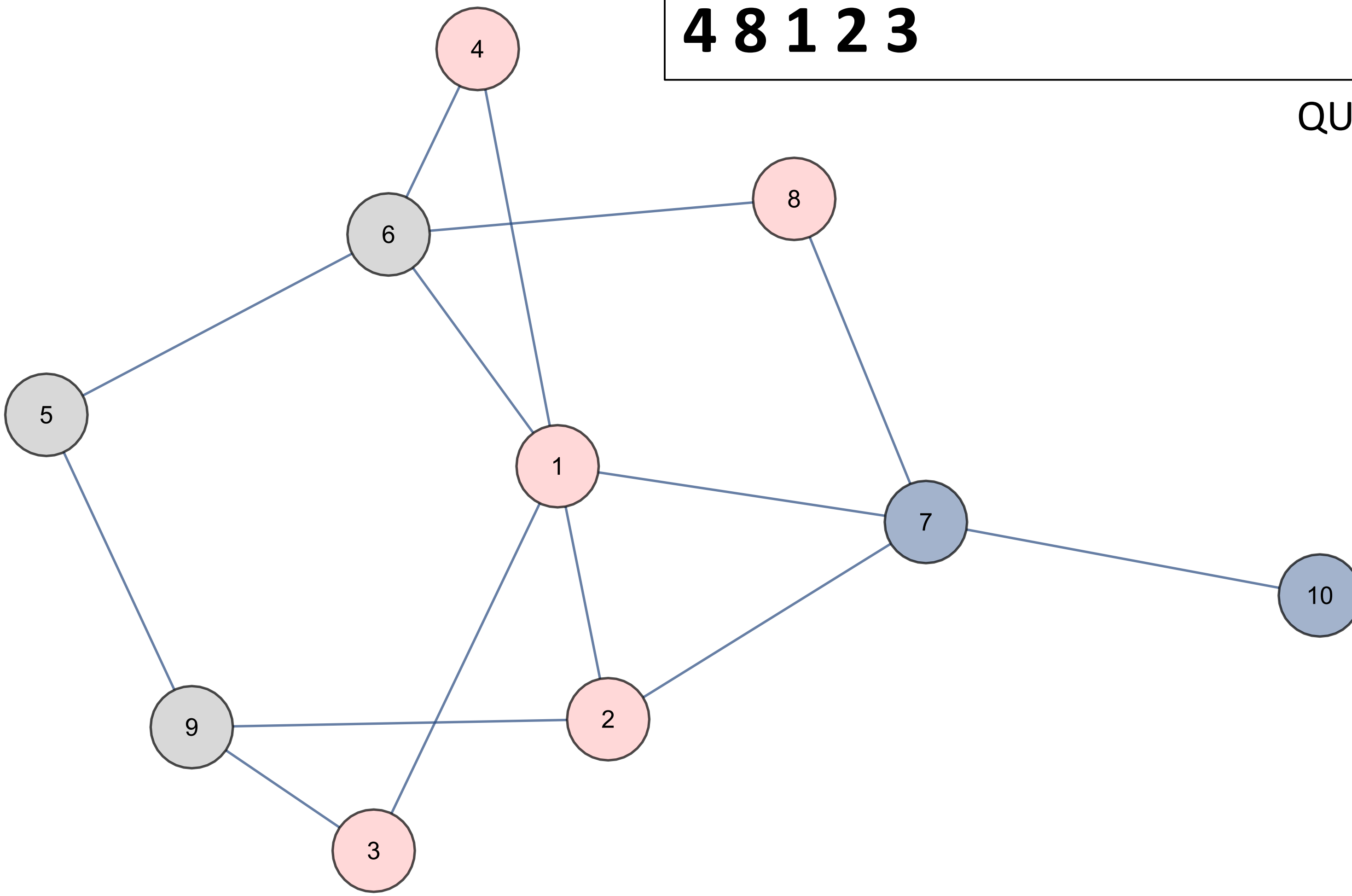
QUEUE



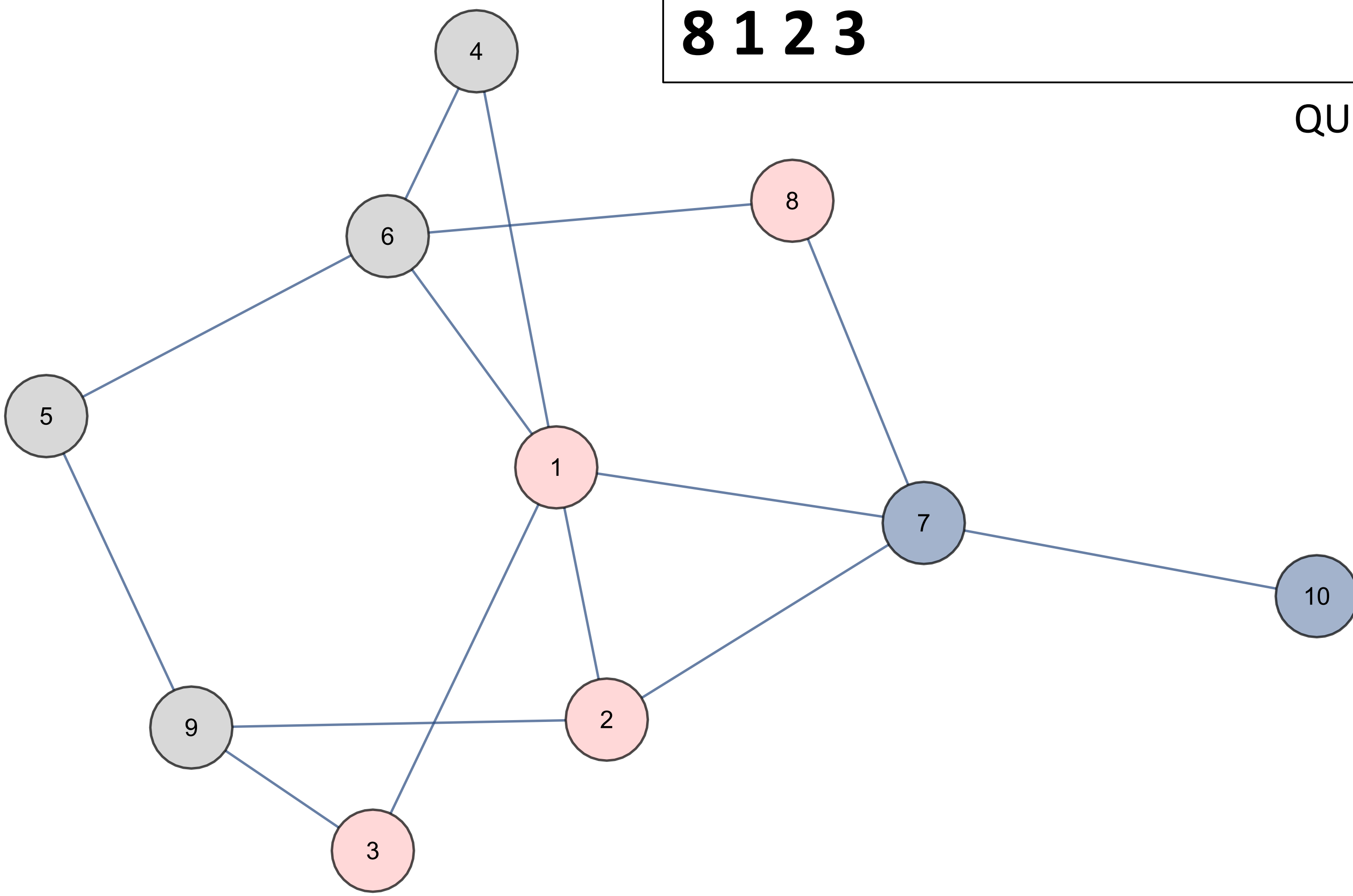
Breadth First Search

4 8 1 2 3

QUEUE



Breadth First Search



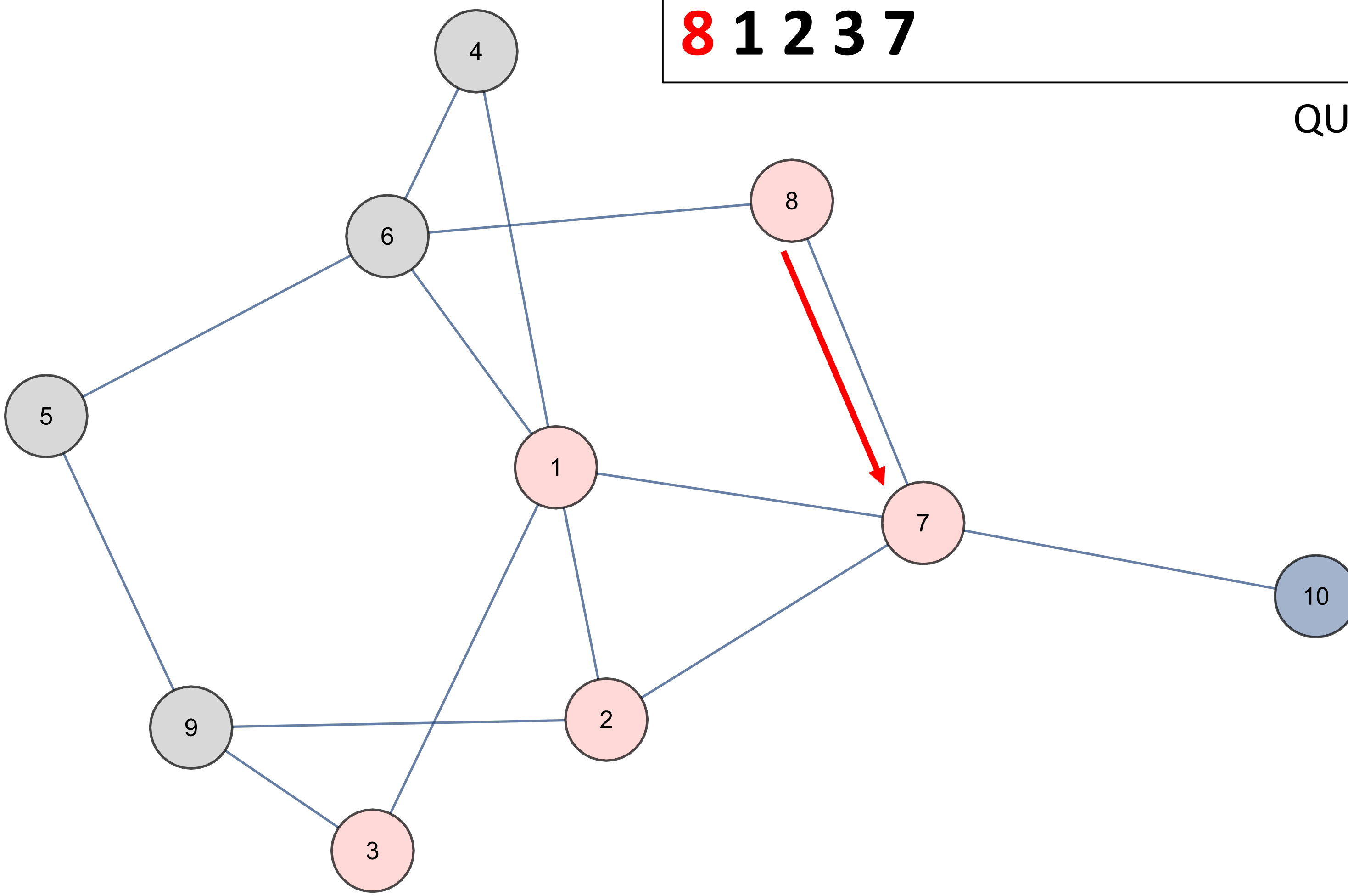
8 1 2 3

QUEUE

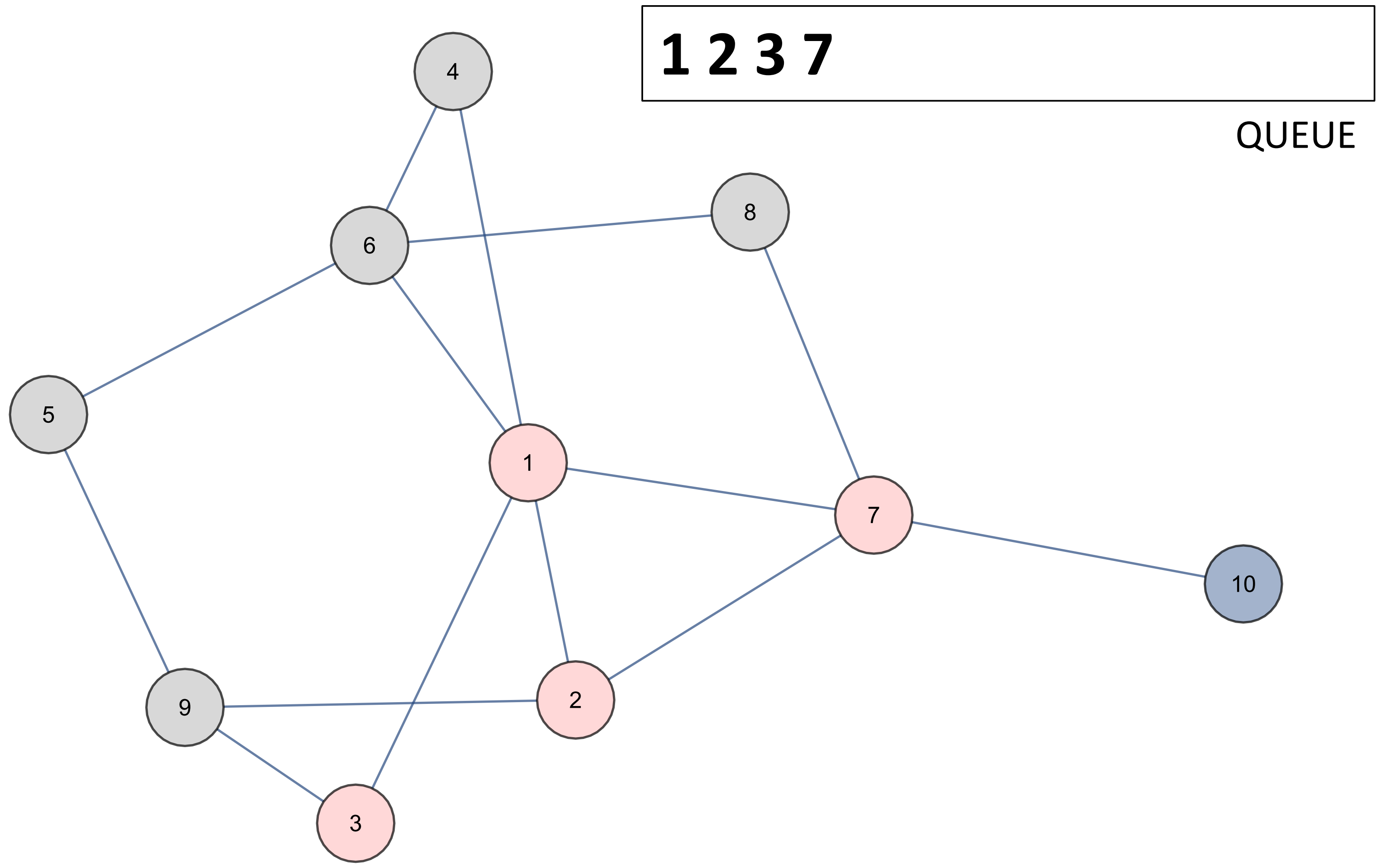
Breadth First Search

8 1 2 3 7

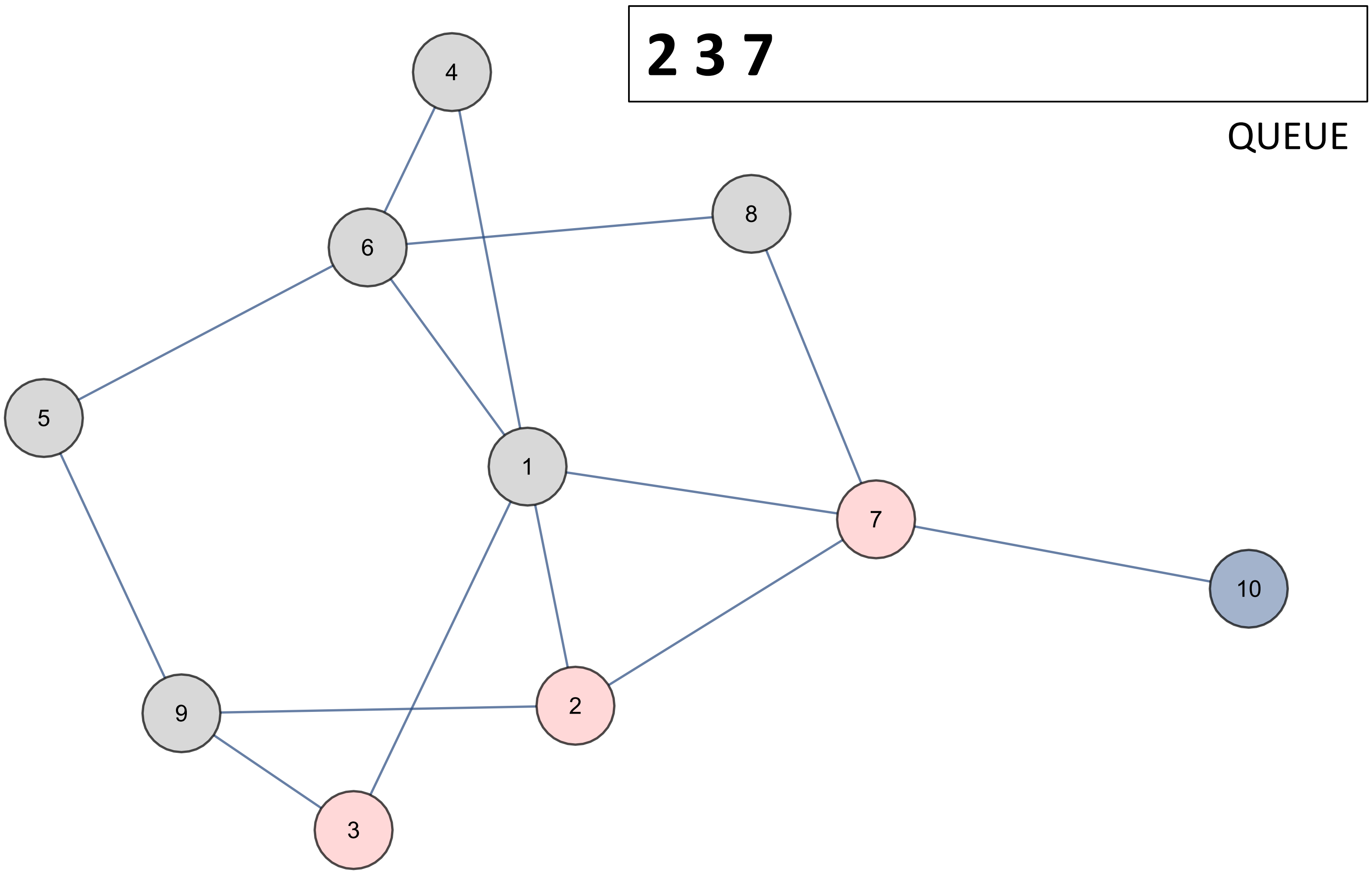
QUEUE



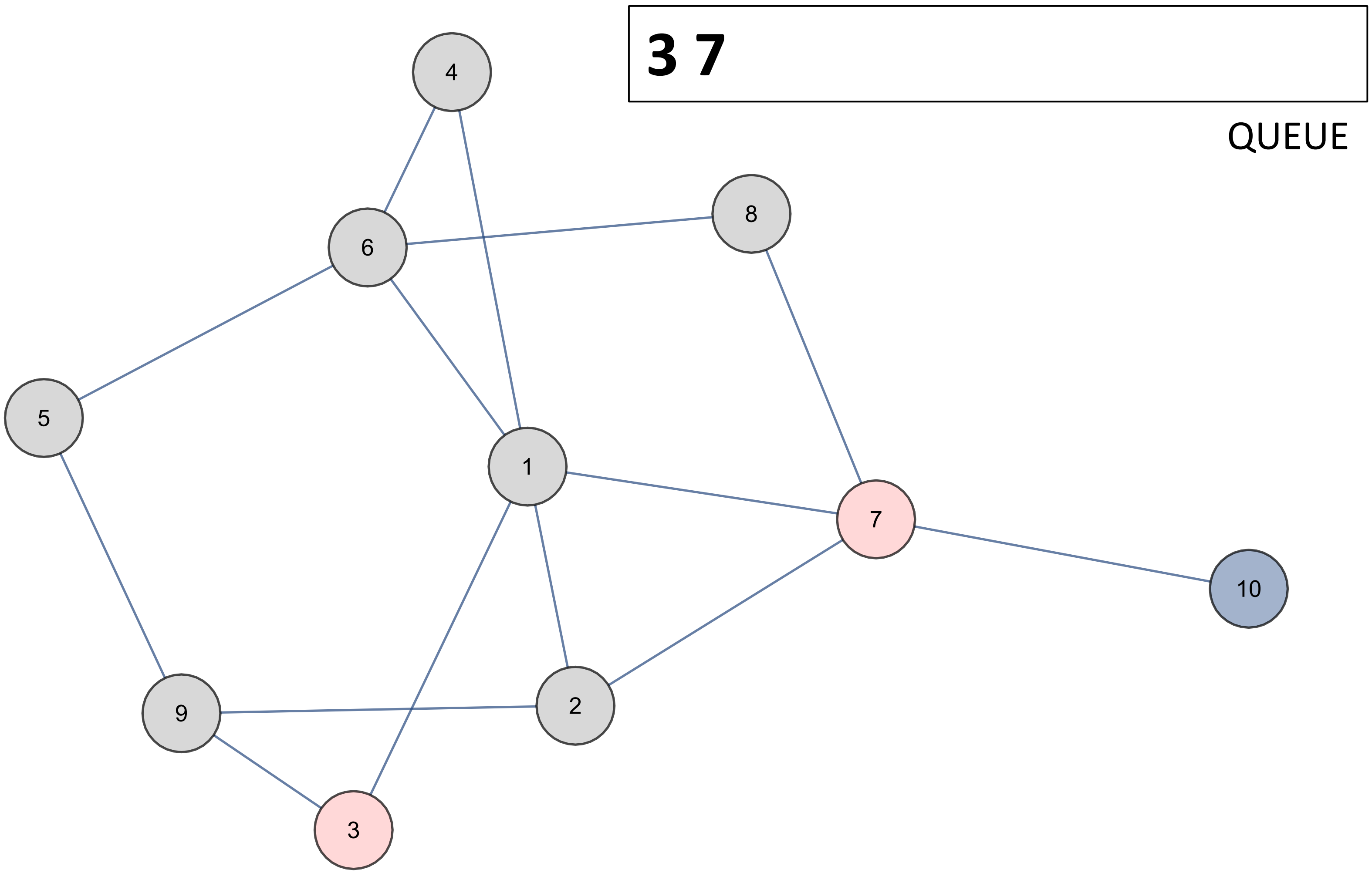
Breadth First Search



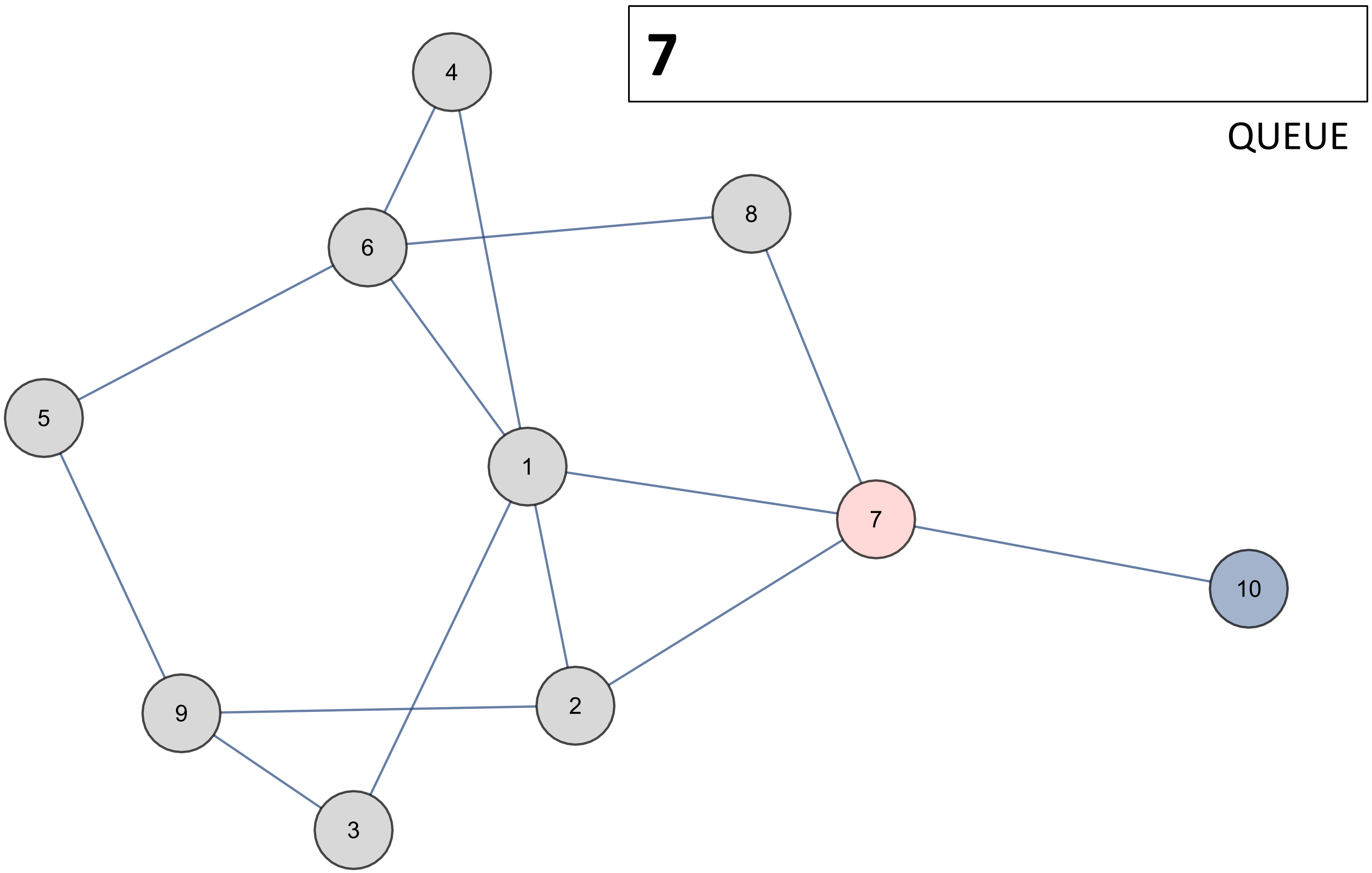
Breadth First Search



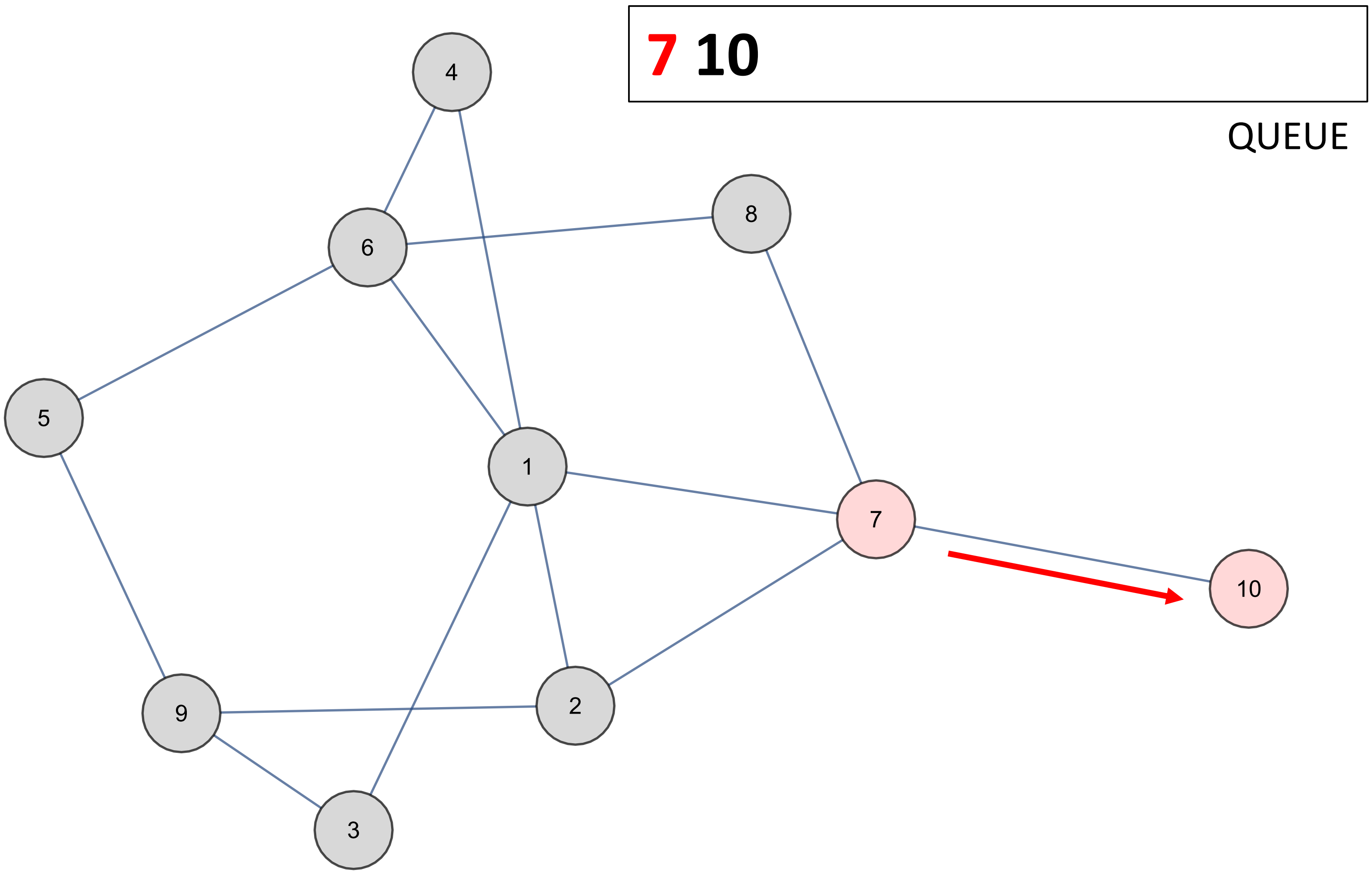
Breadth First Search



Breadth First Search



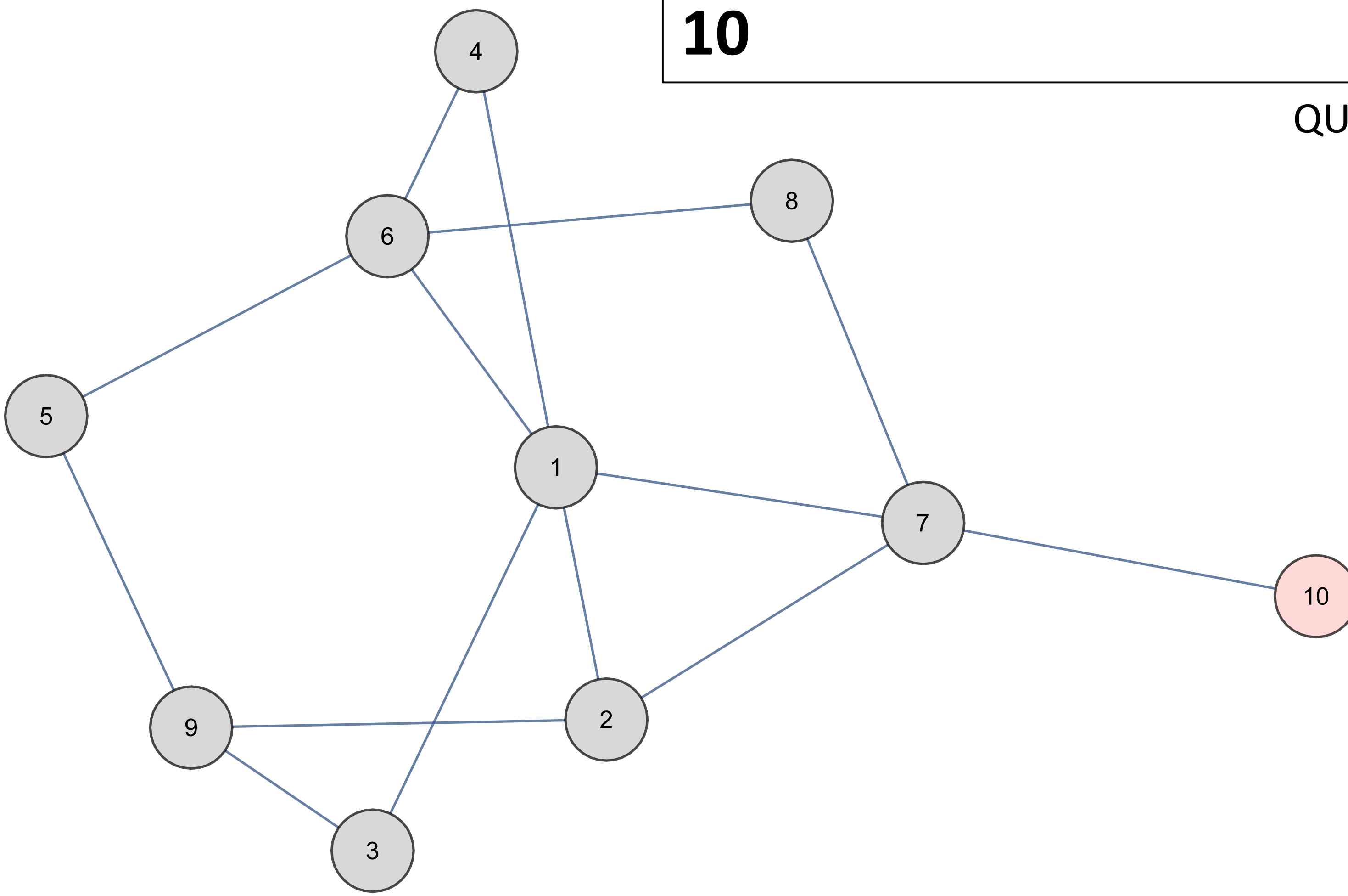
Breadth First Search



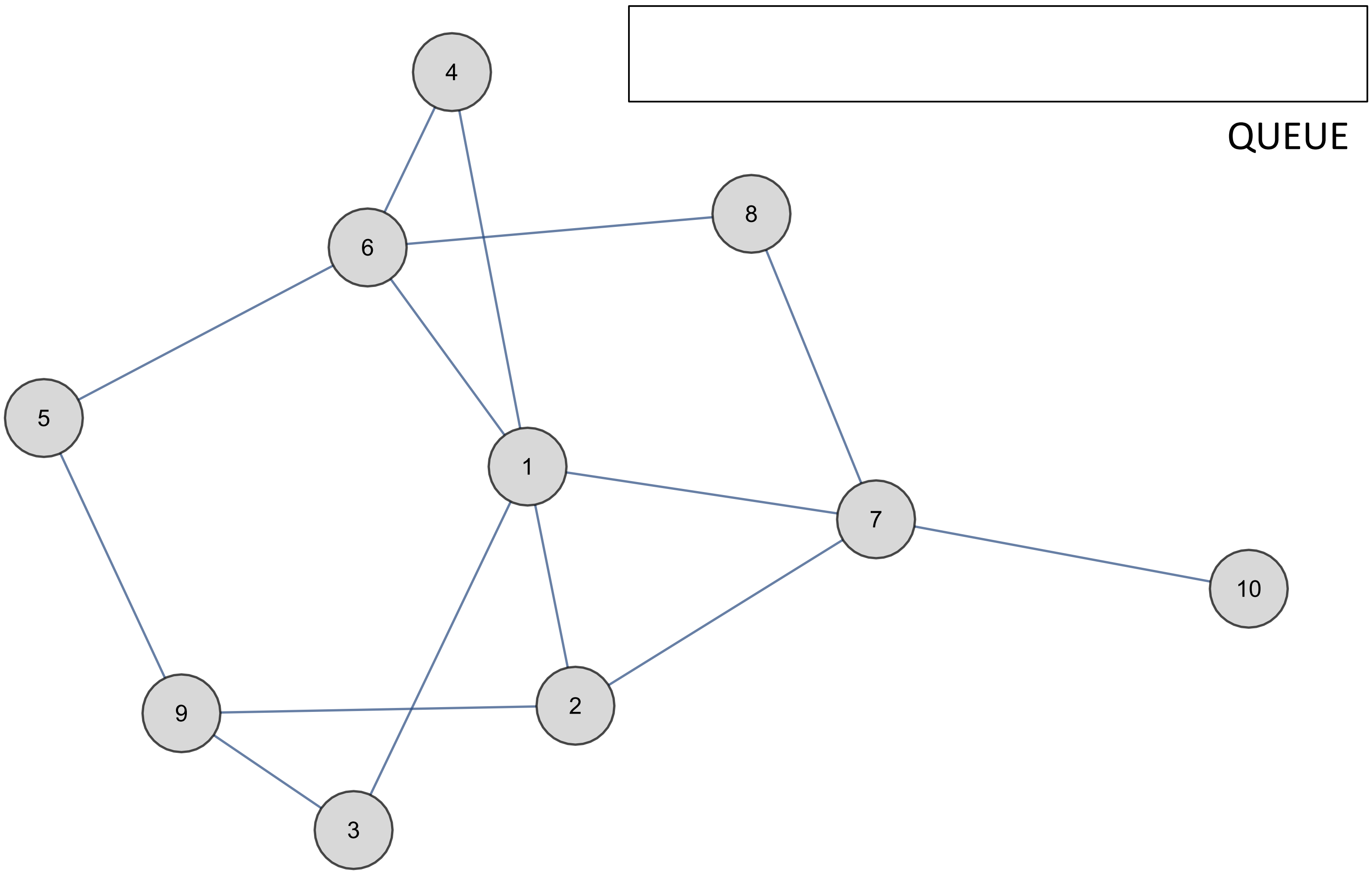
Breadth First Search

10

QUEUE



Breadth First Search



Breadth First Search

```
# Global/class scope variables
n = number of nodes in the graph
g = adjacency list representing unweighted graph

# s = start node, e = end node, and 0 ≤ e,s < n
function bfs(s, e):

  # Do a BFS starting at node s
  prev = solve(s)

  # Return reconstructed path from s -> e
  return reconstructPath(s, e, prev)
```

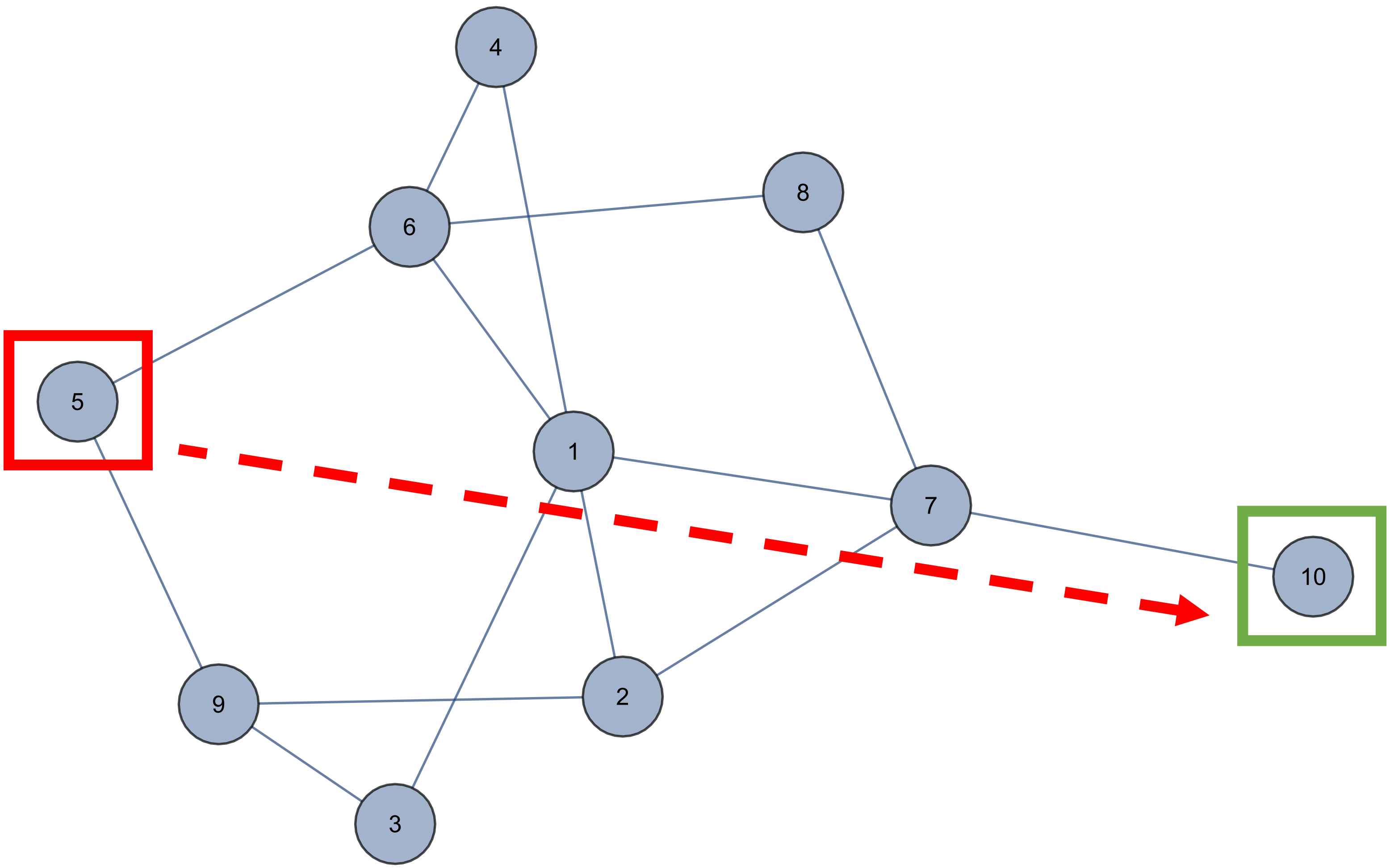
Breadth First Search

```
function solve(s):
  q = queue data structure with enqueue and dequeue
  q.enqueue(s)

  visited = [false, …, false] # size n
  visited[s] = true

  prev = [null, …, null] # size n
  while !q.isEmpty():
    node = q.dequeue()
    neighbours = g.get(node)

    for(next : neighbours):
      if !visited[next]:
        q.enqueue(next)
        visited[next] = true
        prev[next] = node
  return prev
```

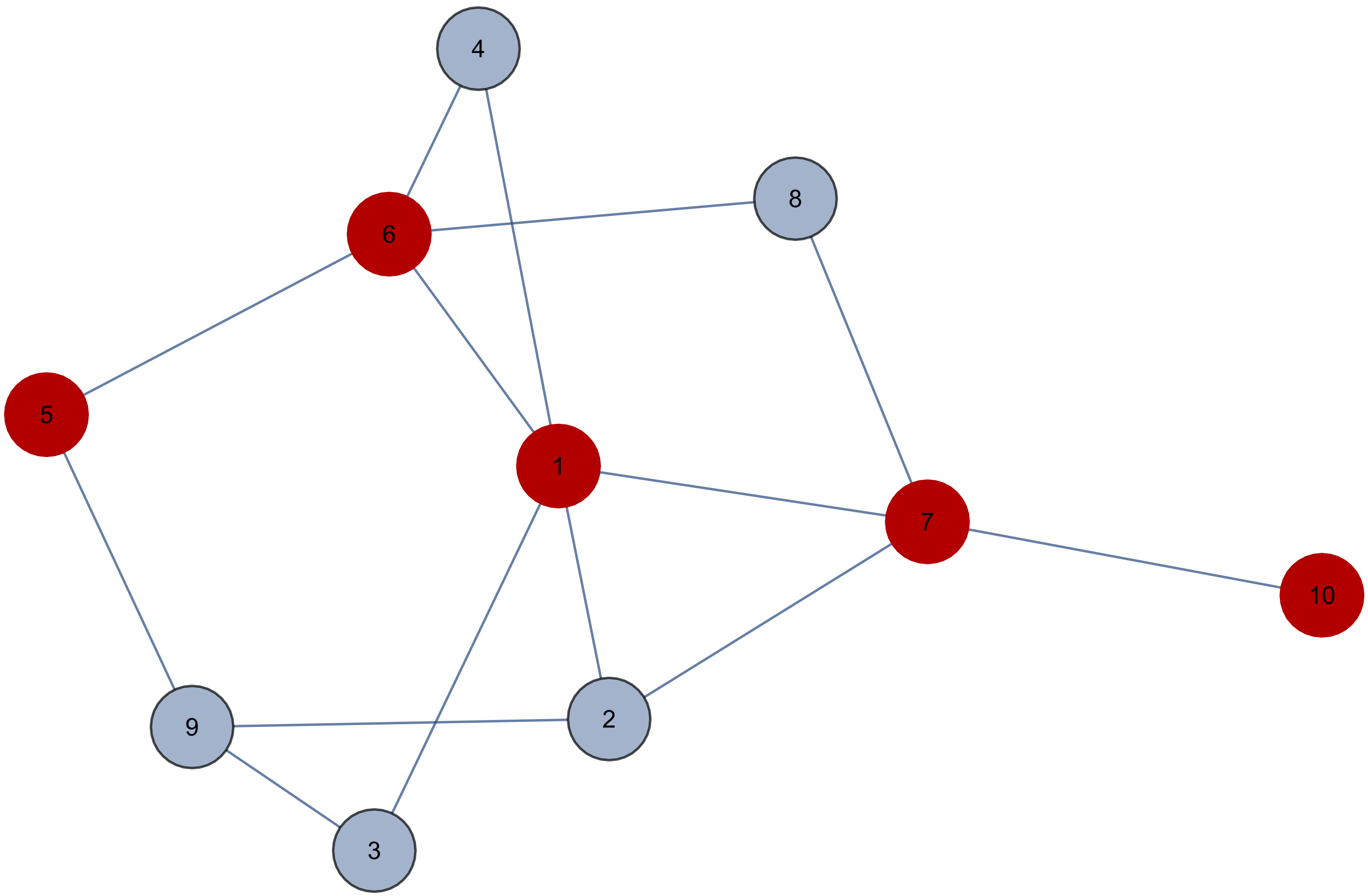
Breadth First Search

```
function reconstructPath(s, e, prev):

  # Reconstruct path going backwards from e
  path = []
  for(at = e; at != null; at = prev[at]):
    path.add(at)

  path.reverse()

  # If s and e are connected return the path
  if path[0] == s:
    return path
  return []
```

Shortest Path Problem

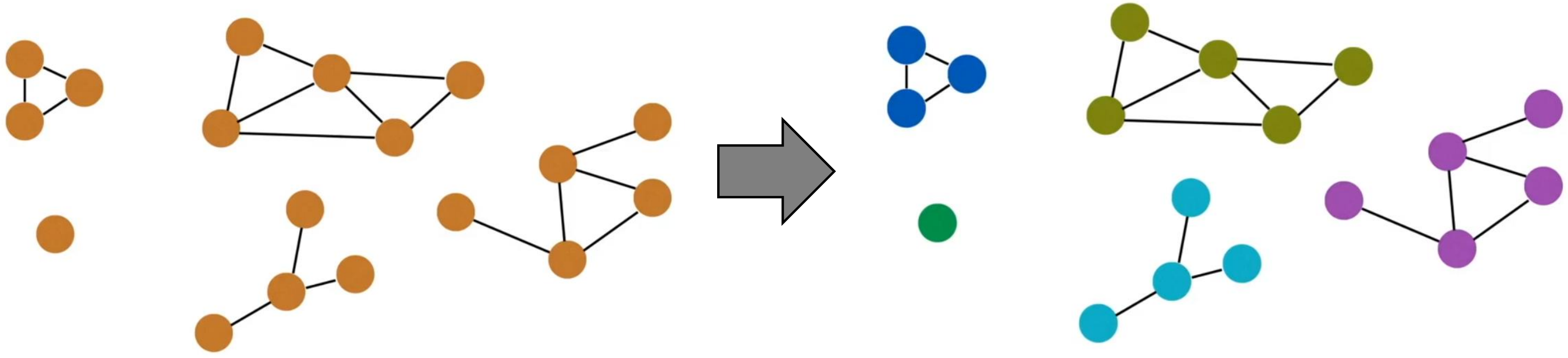


Shortest Path Problem



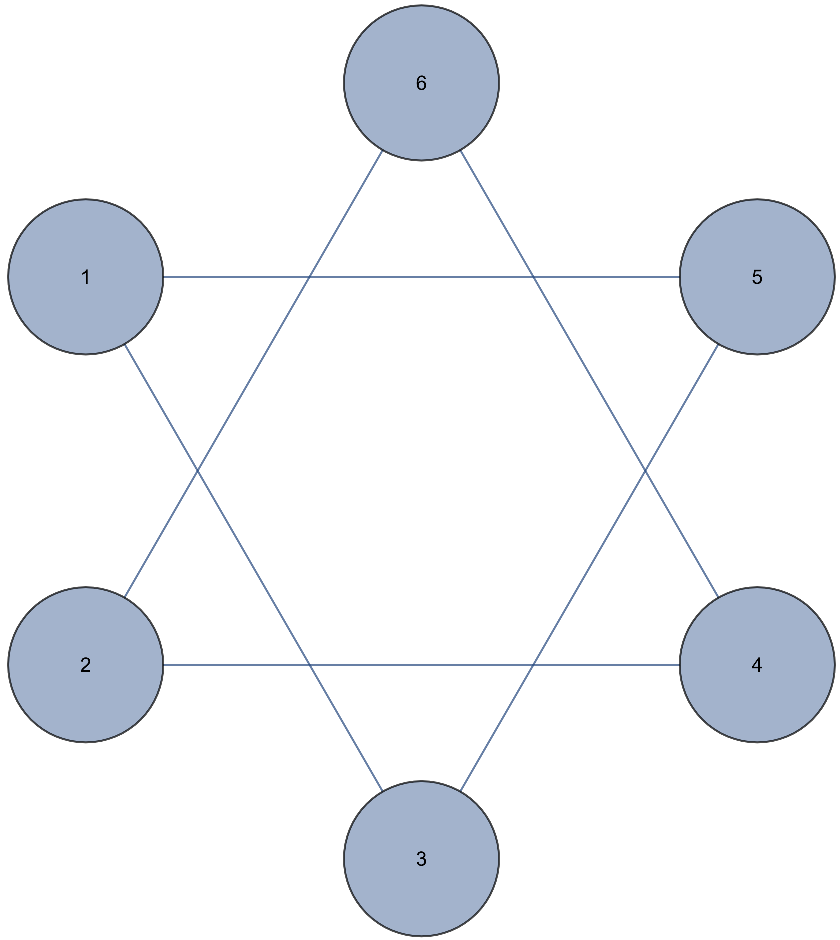
Connected Components

In graph theory, a **component**, sometimes called a **connected component**, of an undirected graph is a subgraph in which any two vertices are connected to each other by paths, and which is connected to no additional vertices in the supergraph. For example, the graph shown in the illustration has three components. A vertex with no incident edges is itself a component. A graph that is itself connected has exactly one component, consisting of the whole graph.

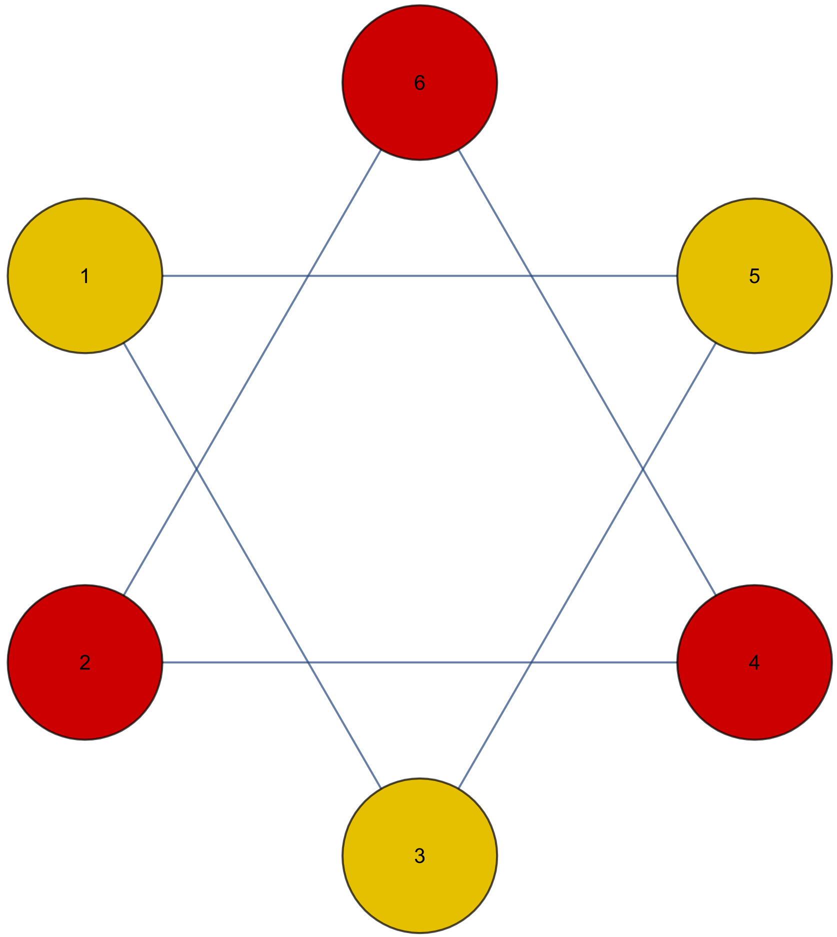


A graph with 5 components.

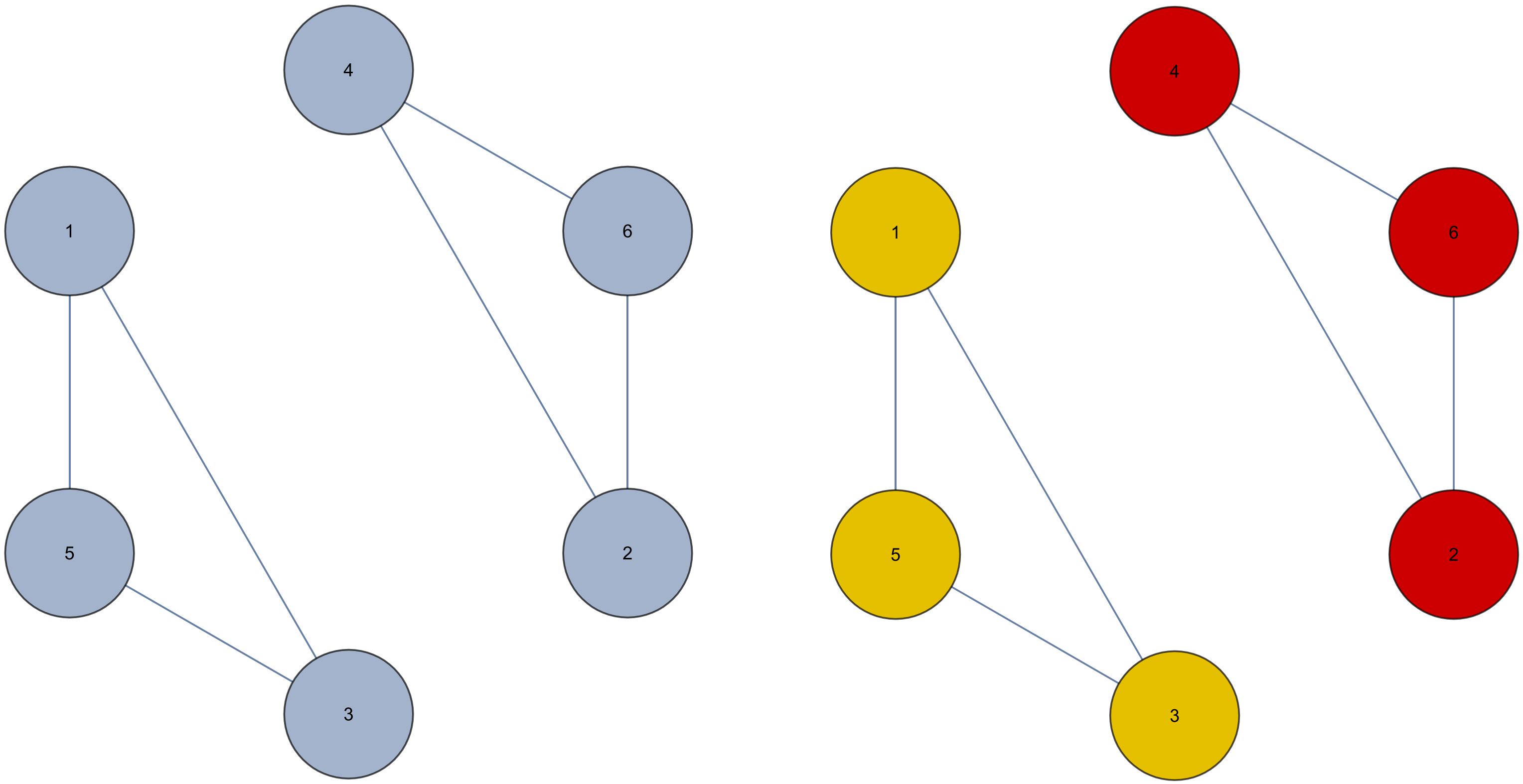
Connected Components



Connected Components



Connected Components

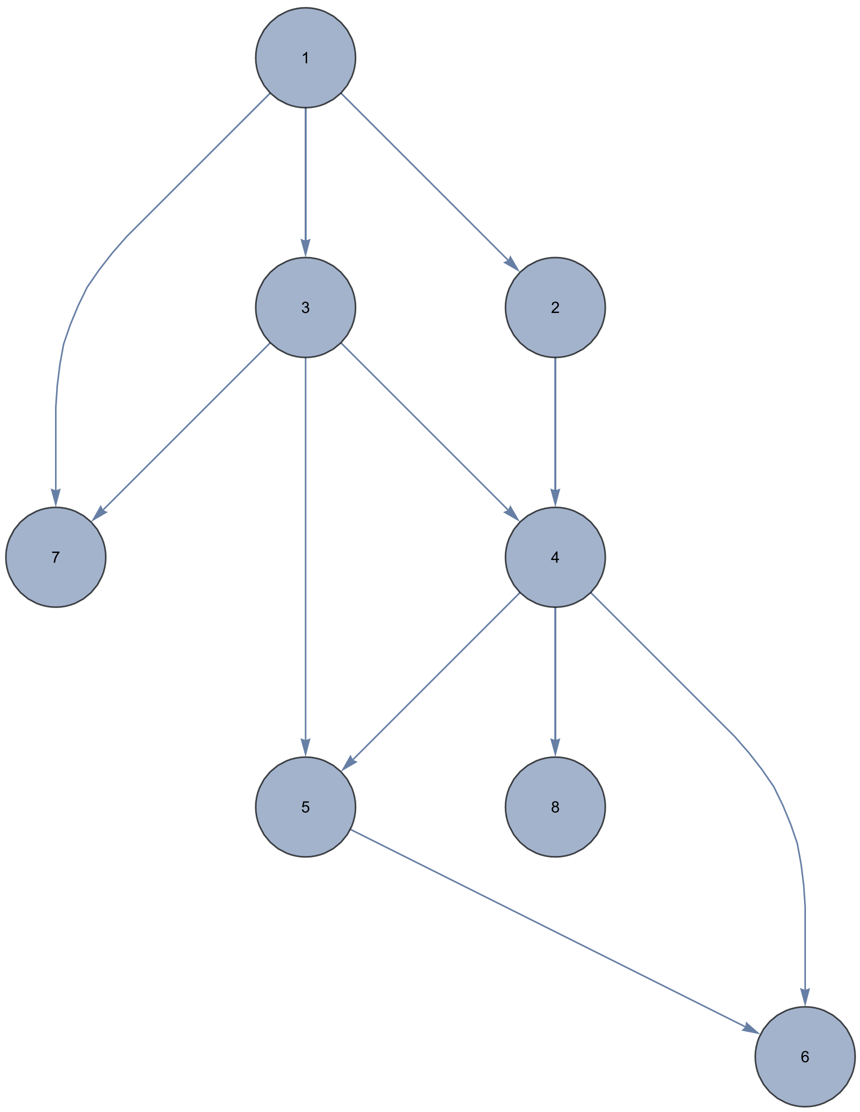


Connected Components

```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n

function findComponents():
  for (i = 0; i < n; i++):
    if !visited[i]:
      count++
      dfs(i)
  return (count, components)

function dfs(at):
  visited[at] = true
  components[at] = count
  for (next : g[at]):
    if !visited[next]:
      dfs(next)
```

Topological Sort

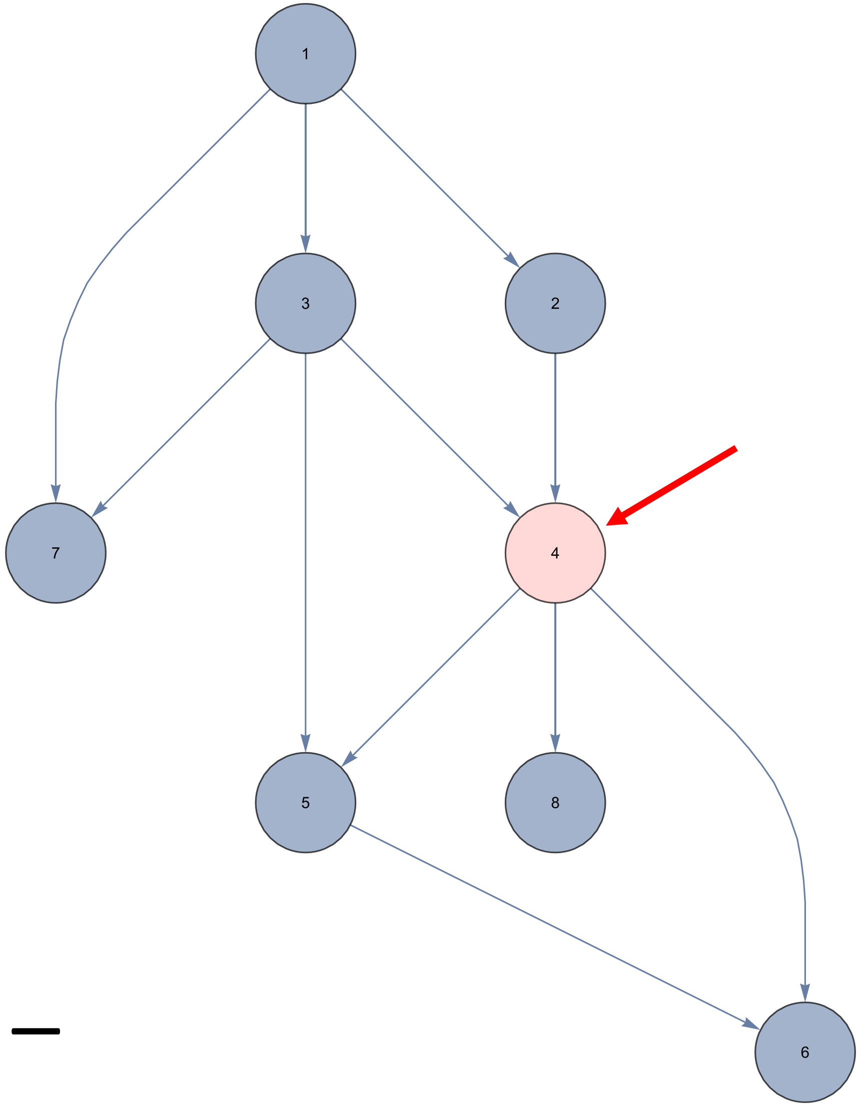
In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

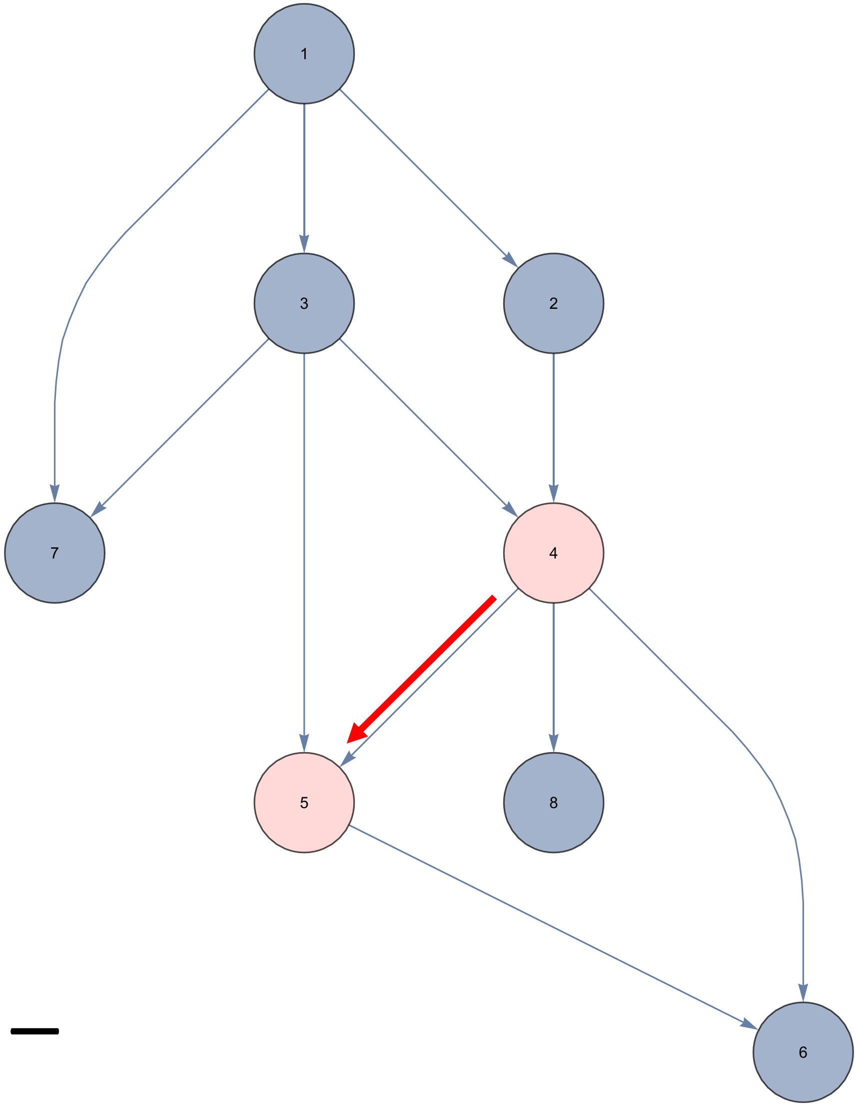
_ _ _ _ _ _ _ _



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

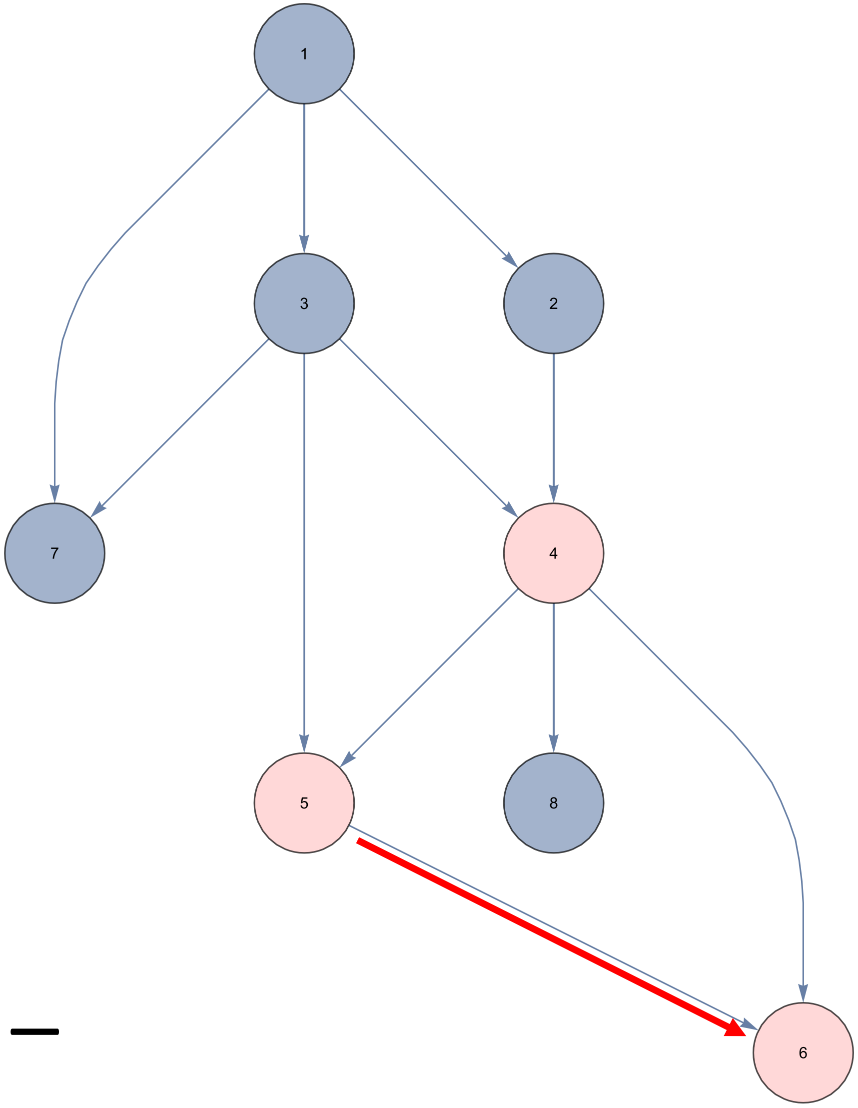
_ _ _ _ _ _ _ _



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

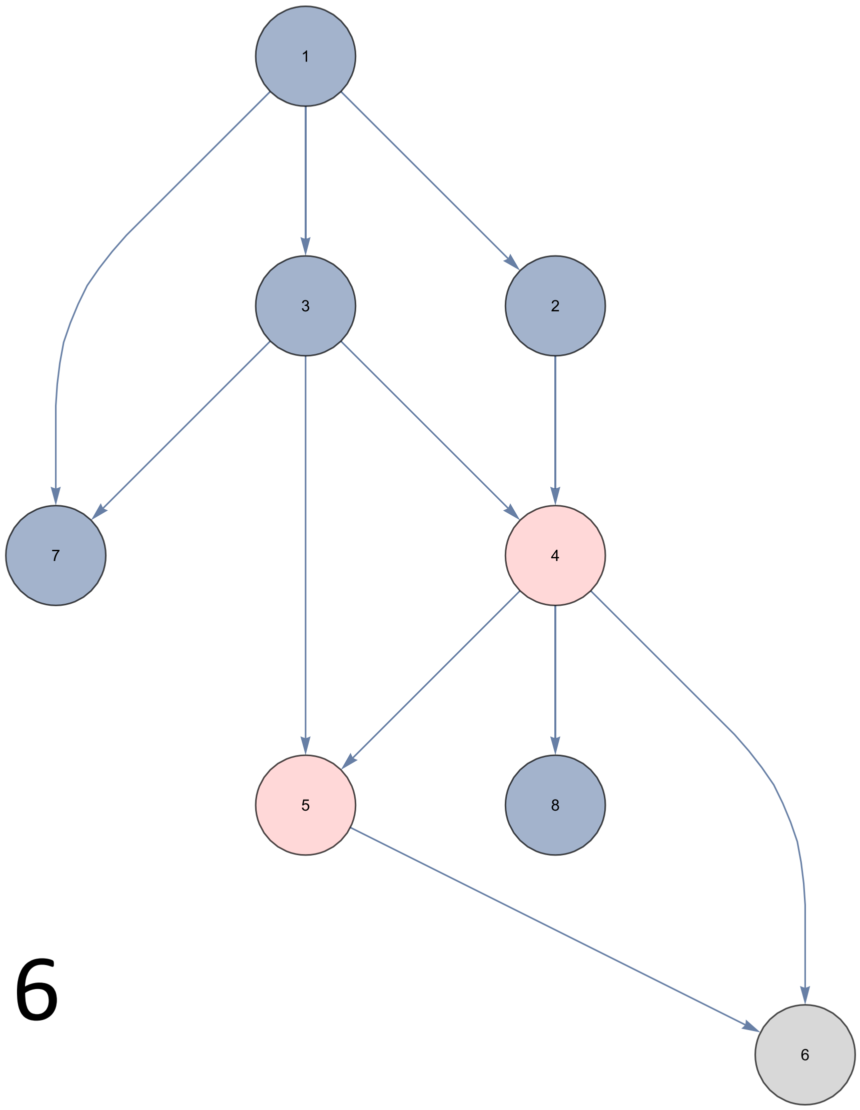
_ _ _ _ _ _ _ _



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

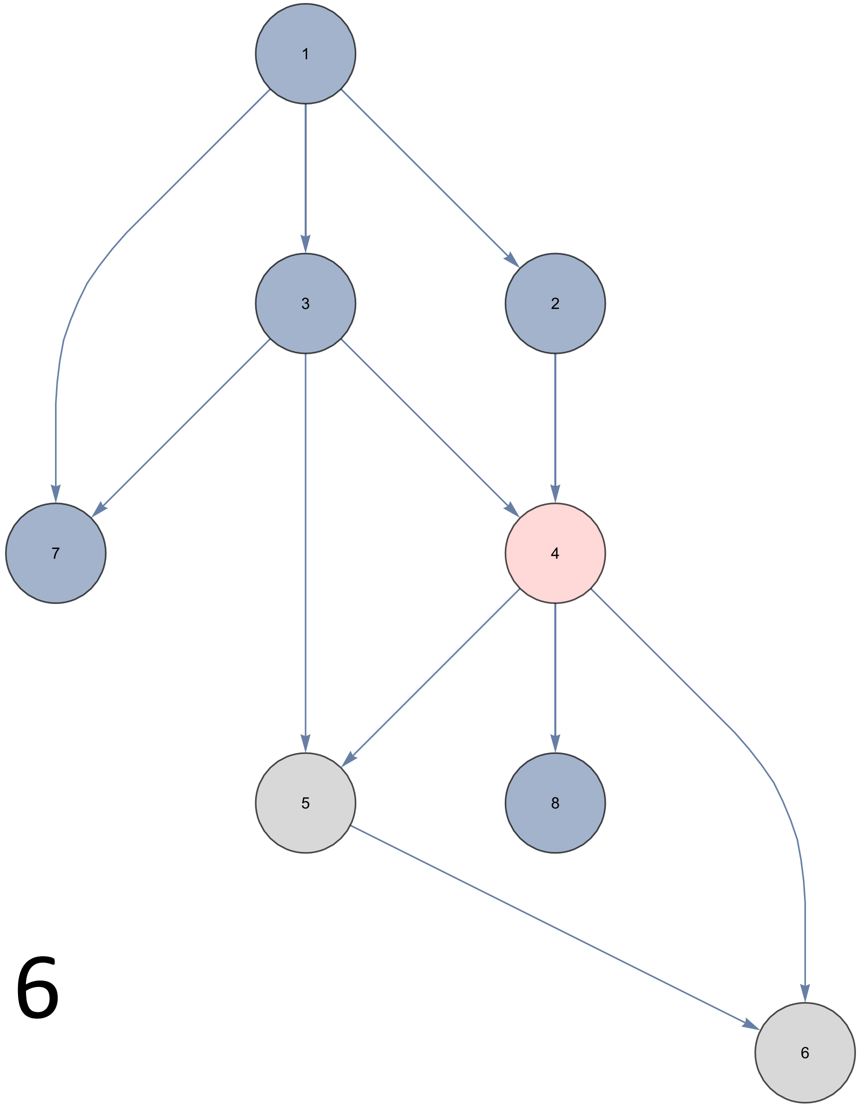
_ _ _ _ _ _ _ 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

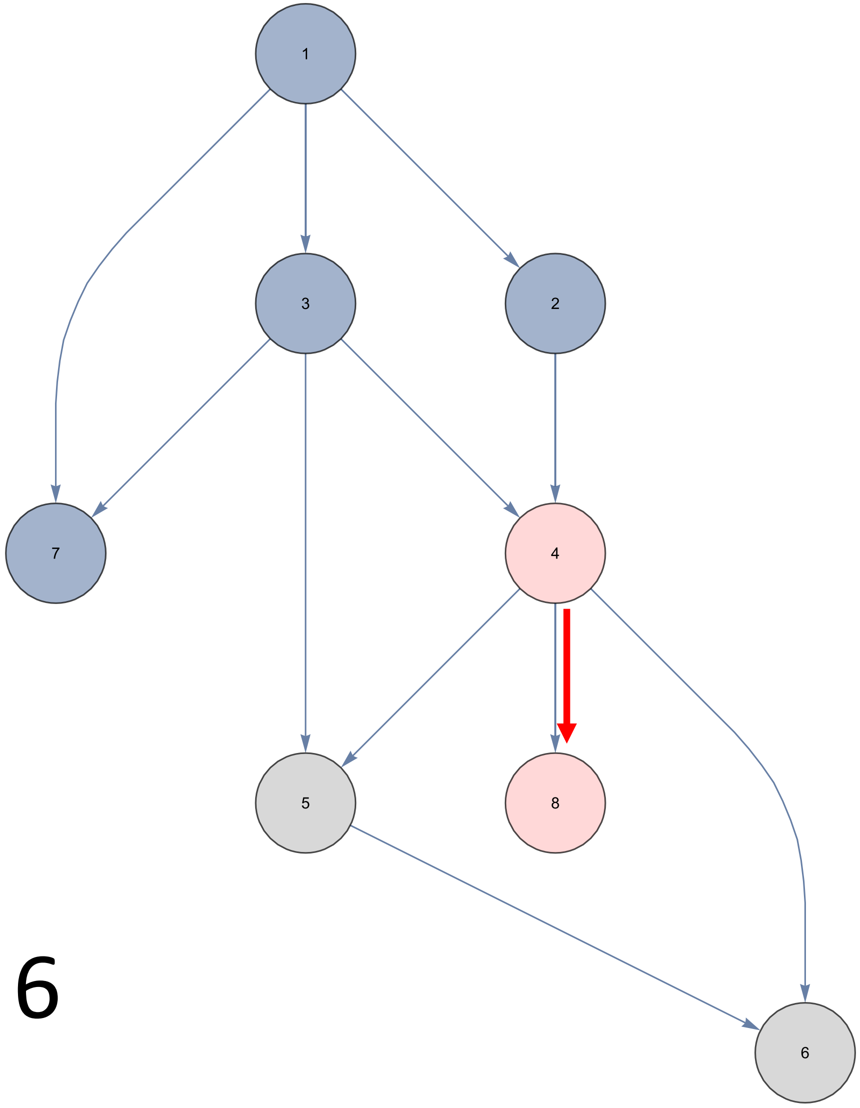
_ _ _ _ _ _ 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

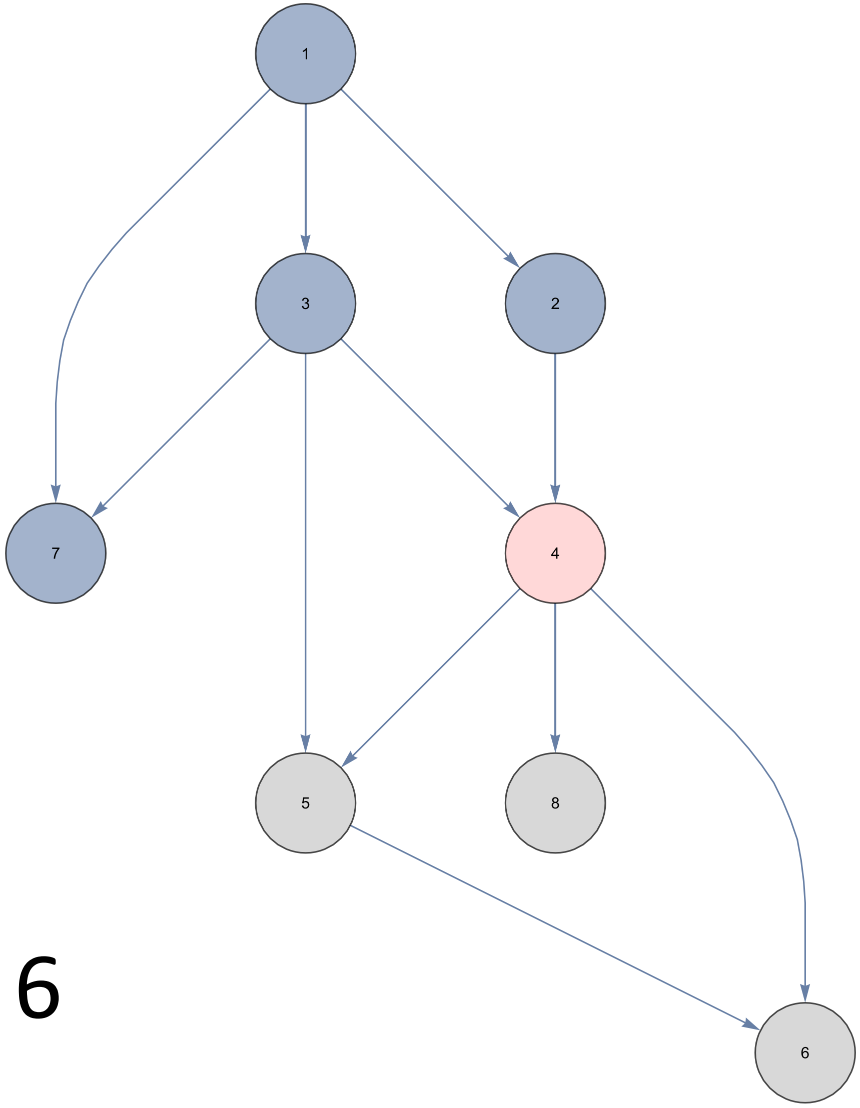
_ _ _ _ _ _ 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

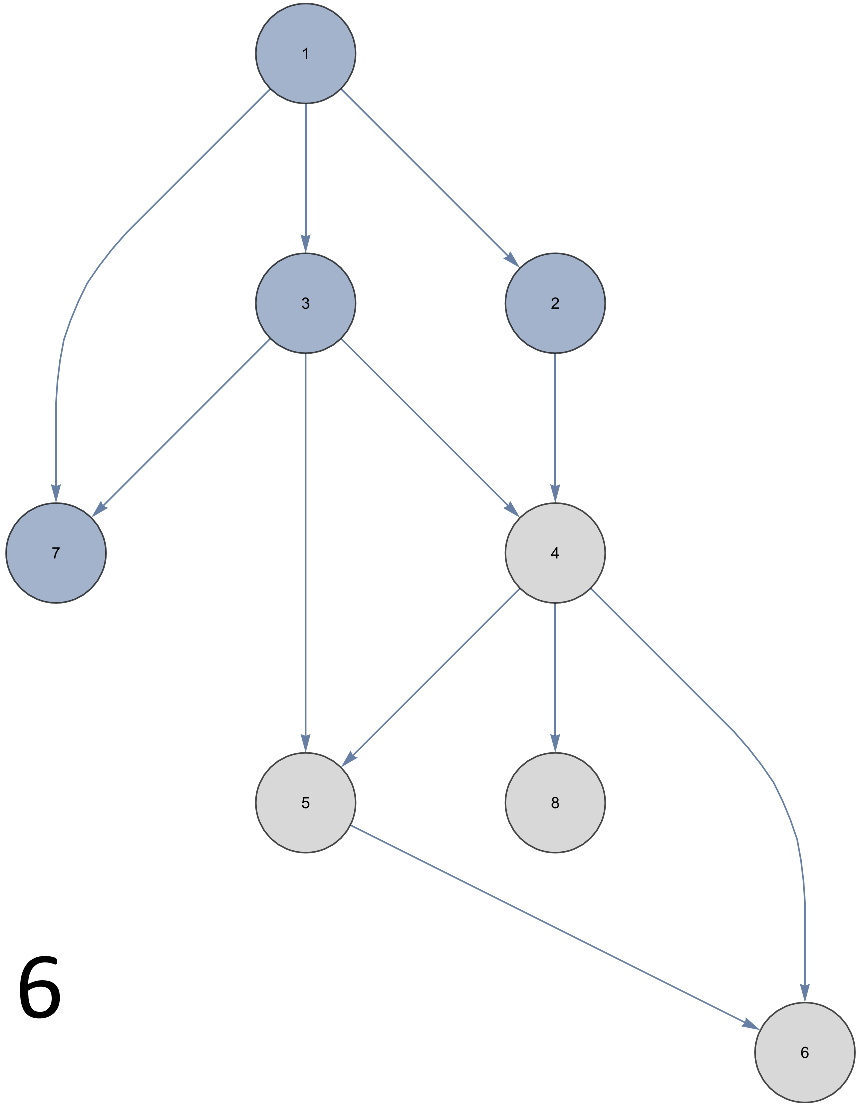
_ _ _ _ _ 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

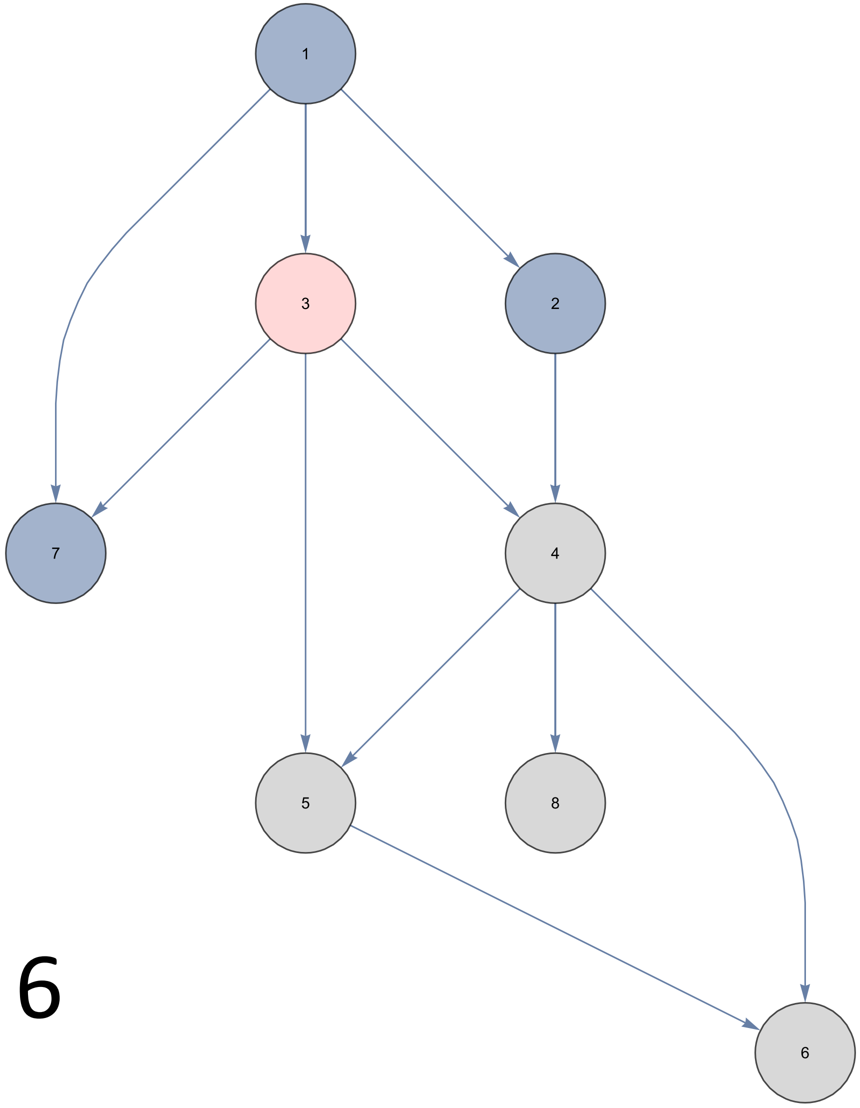
_ _ _ _ 4 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

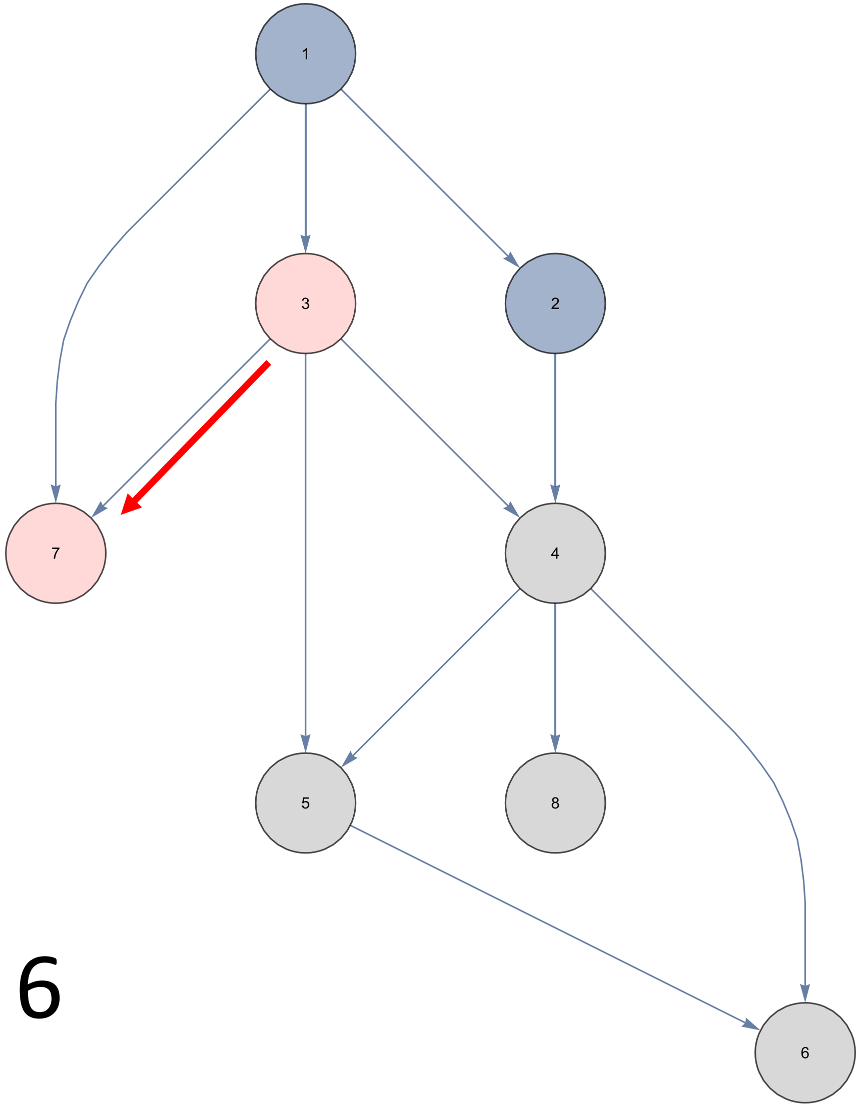
_ _ _ _ 4 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

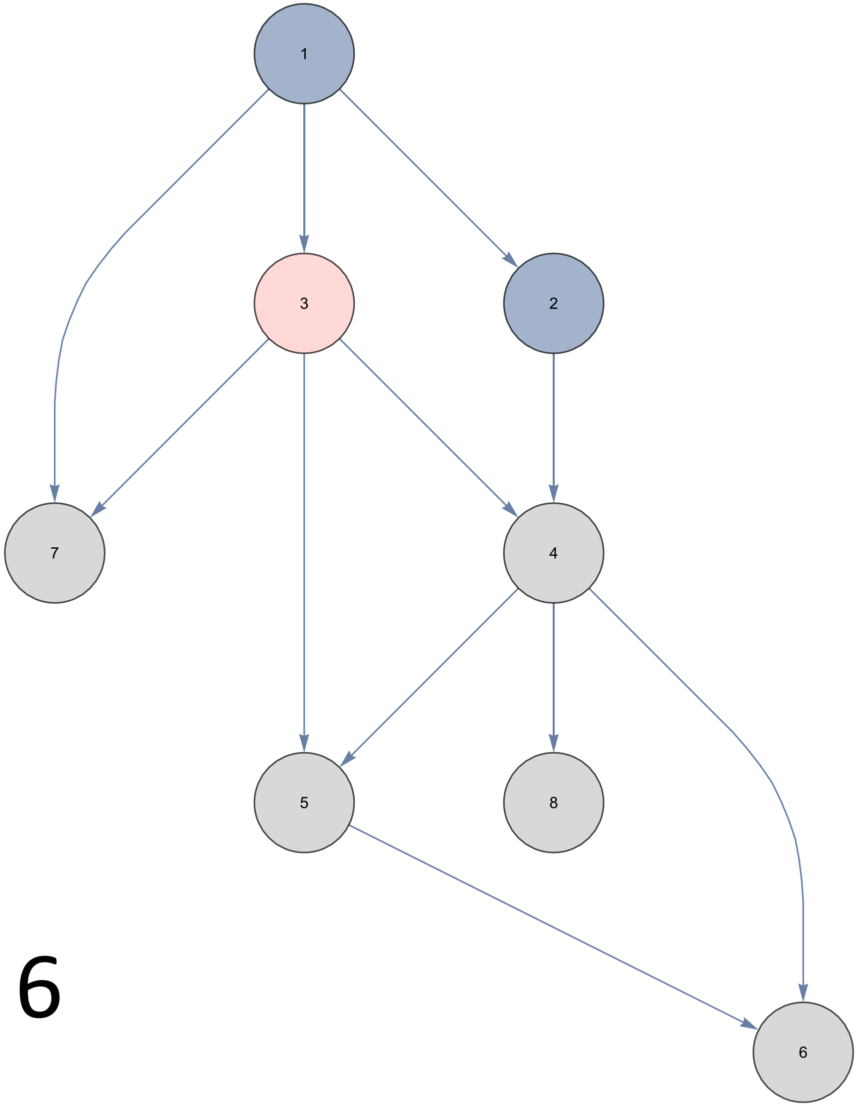
_ _ _ _ 4 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

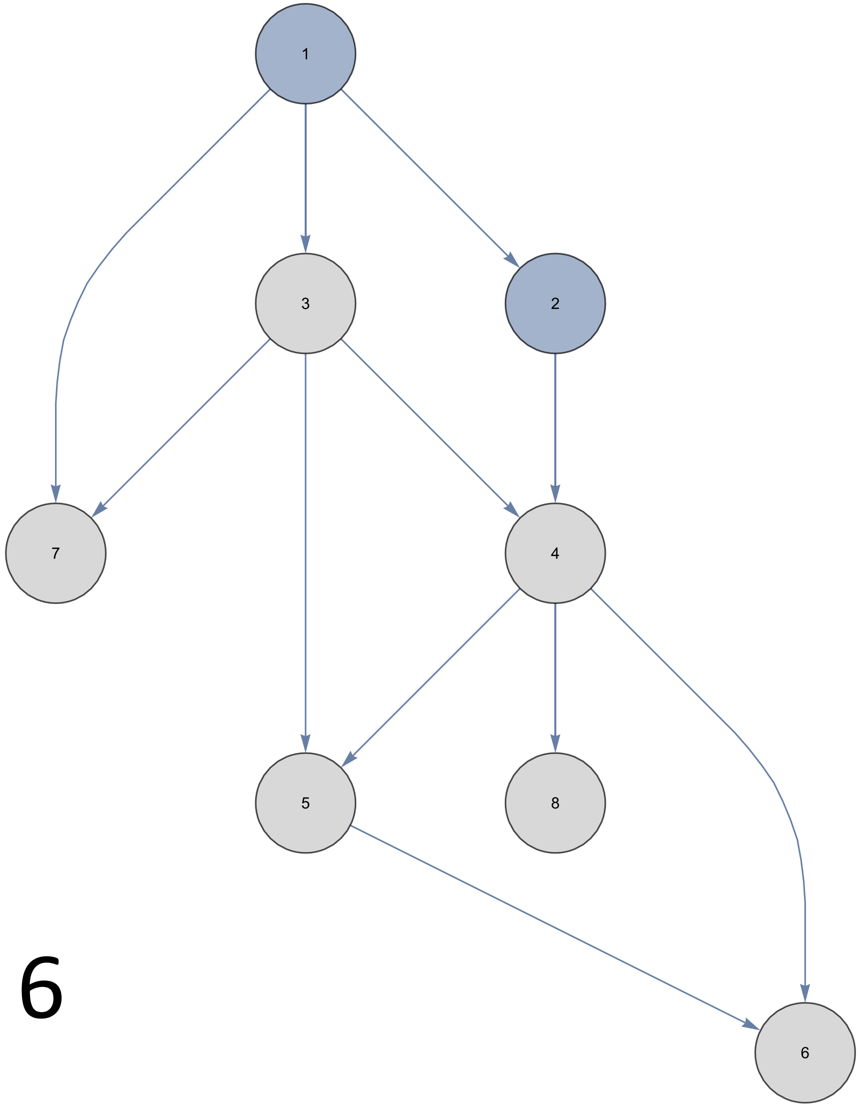
_ _ _ 7 4 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

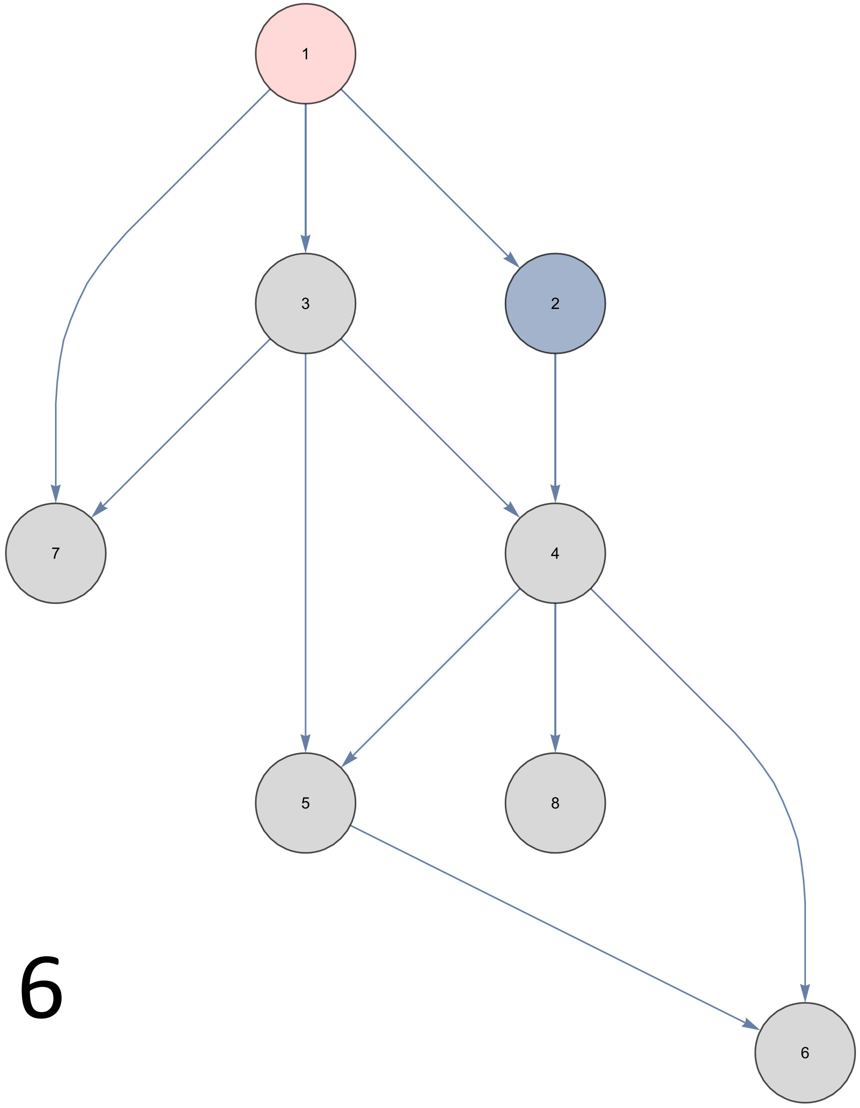
_ _ 3 7 4 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

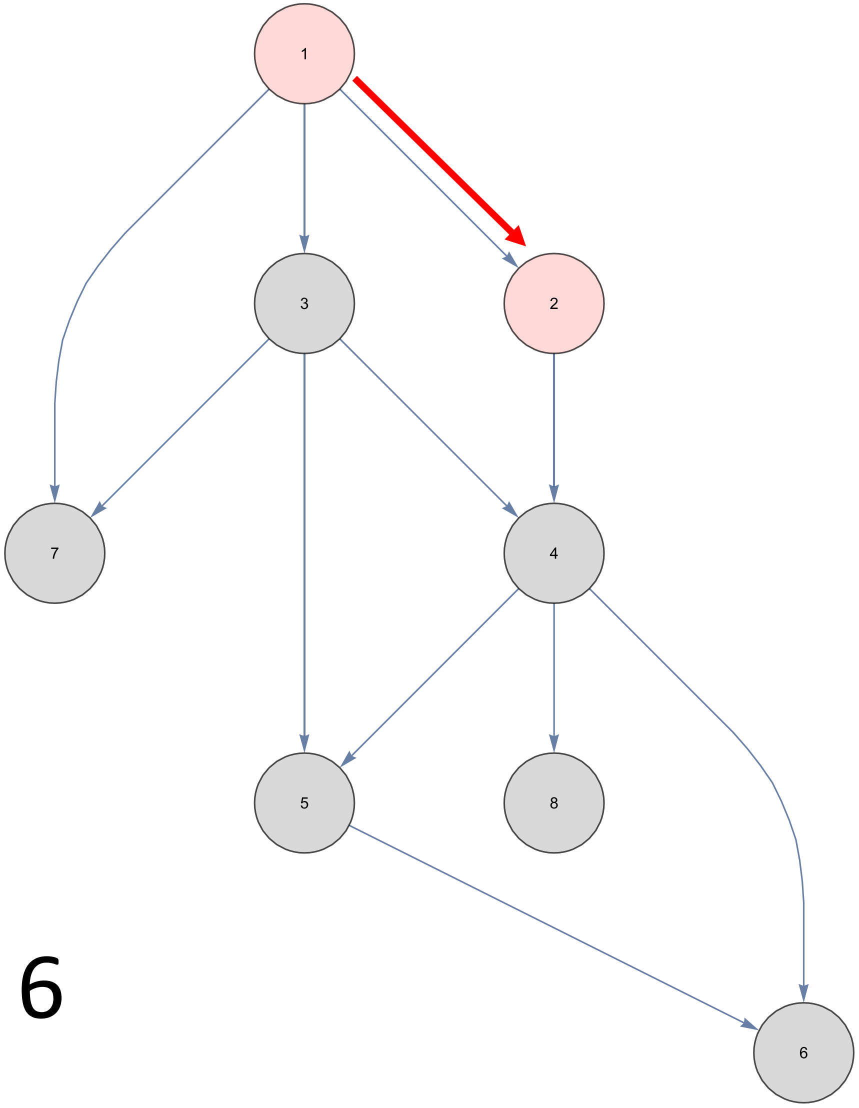
_ _ 3 7 4 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

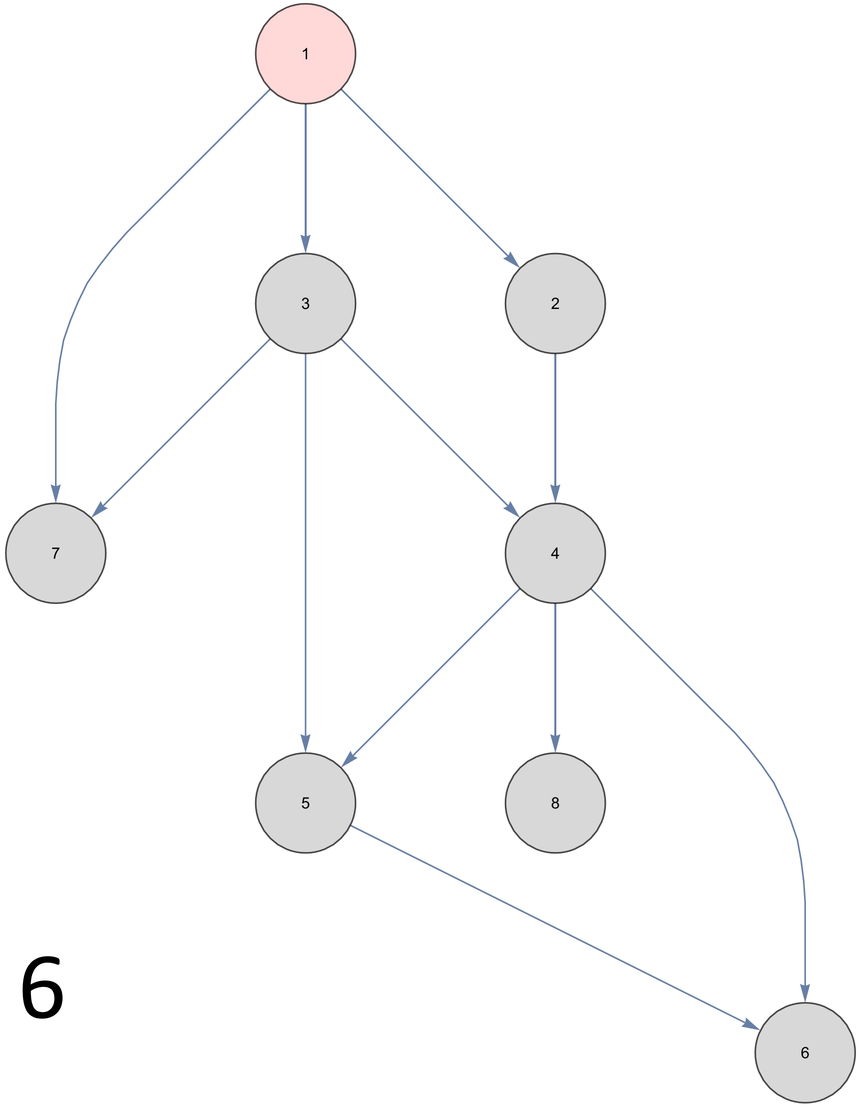
_ _ 3 7 4 8 5 6



Topological Sort

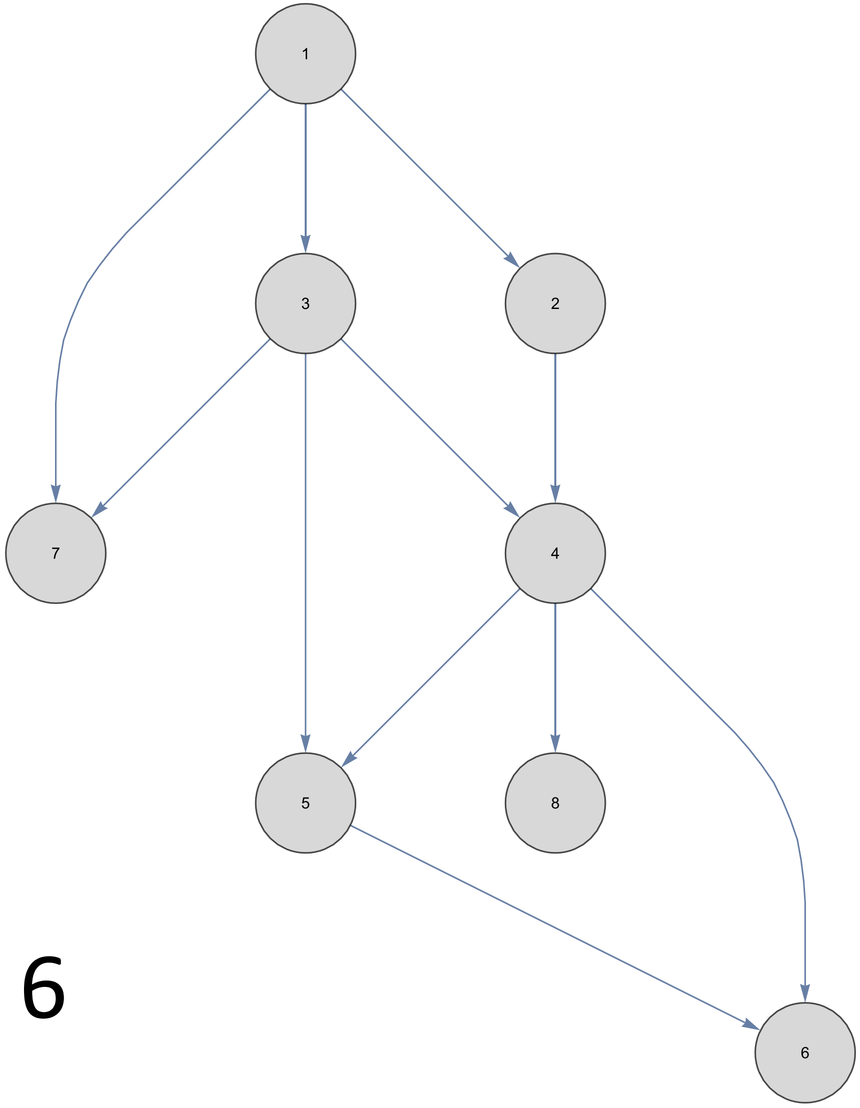
In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.

_ 2 3 7 4 8 5 6



Topological Sort

In computer science, a **topological sort** or **topological ordering** of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.



1 2 3 7 4 8 5 6

Topological Sort

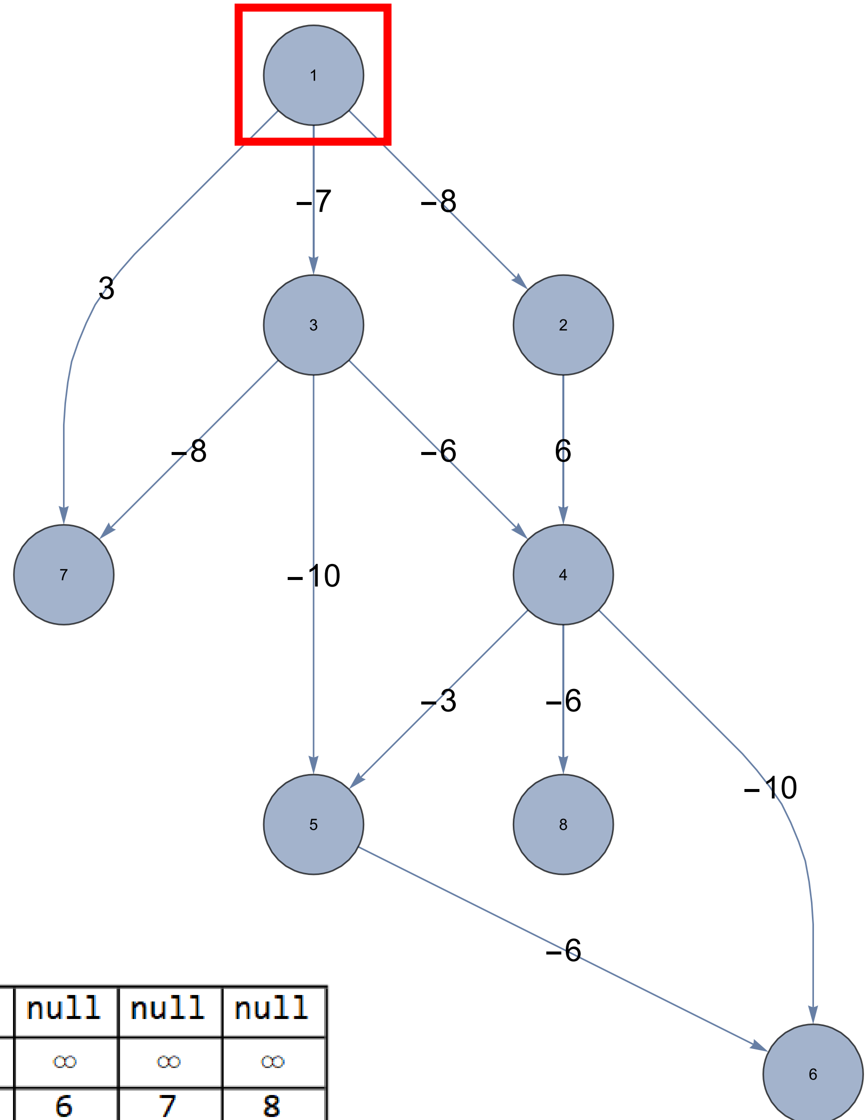
```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  N = graph.numberOfNodes()
  V = [false,…,false] # Length N
  ordering = [0,…,0]  # Length N
  i = N - 1 # Index for ordering array

  for(at = 0; at < N; at++):
    if V[at] == false:
      visitedNodes = []
      dfs(at, V, visitedNodes, graph)
      for nodeId in visitedNodes:
        ordering[i] = nodeId
        i = i - 1
  return ordering
```

Topological Sort

```
# Execute Depth First Search (DFS)
function dfs(at, V, visitedNodes, graph):

  V[at] = true

  edges = graph.getEdgesOutFromNode(at)
  for edge in edges:
    if V[edge.to] == false:
      dfs(edge.to, V, visitedNodes, graph)

  visitedNodes.add(at)
```

SSSP on DAG

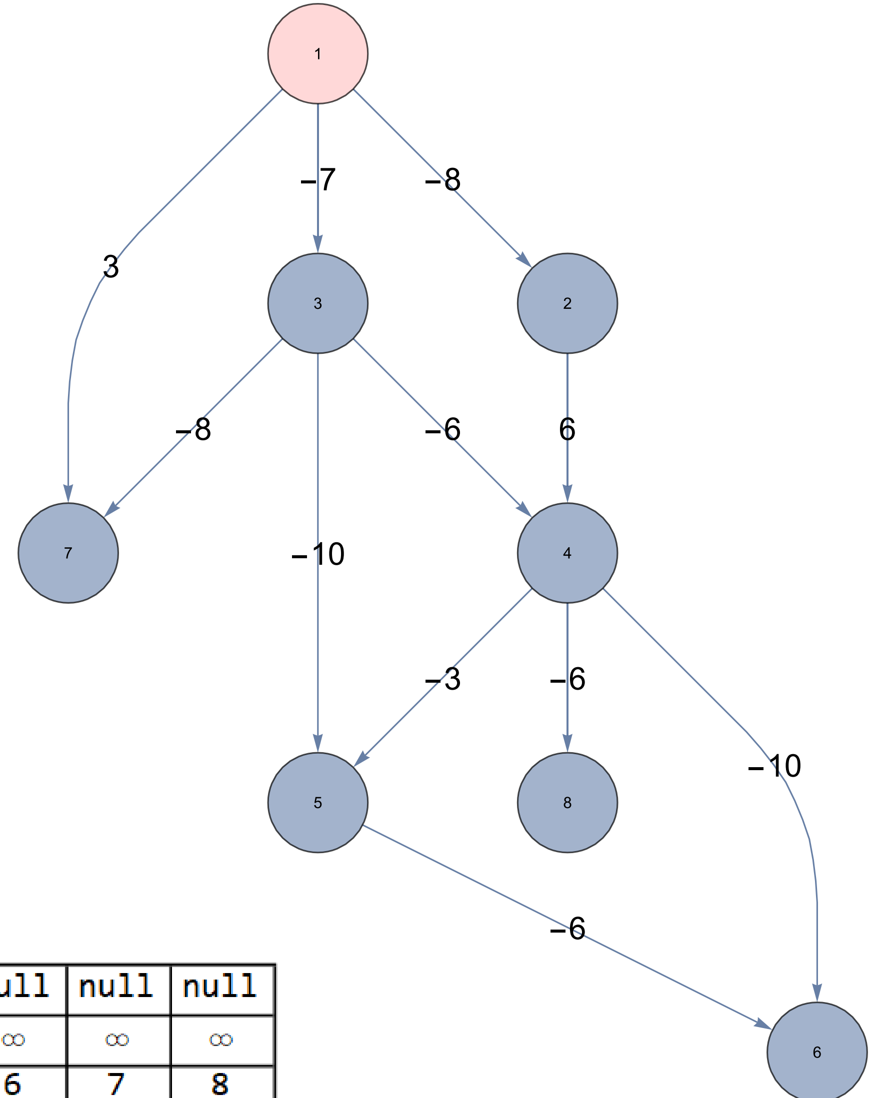
single source shortest path
on directed acyclic graph



from	null	null	null	null	null	null	null	null
distance	∞	∞	∞	∞	∞	∞	∞	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

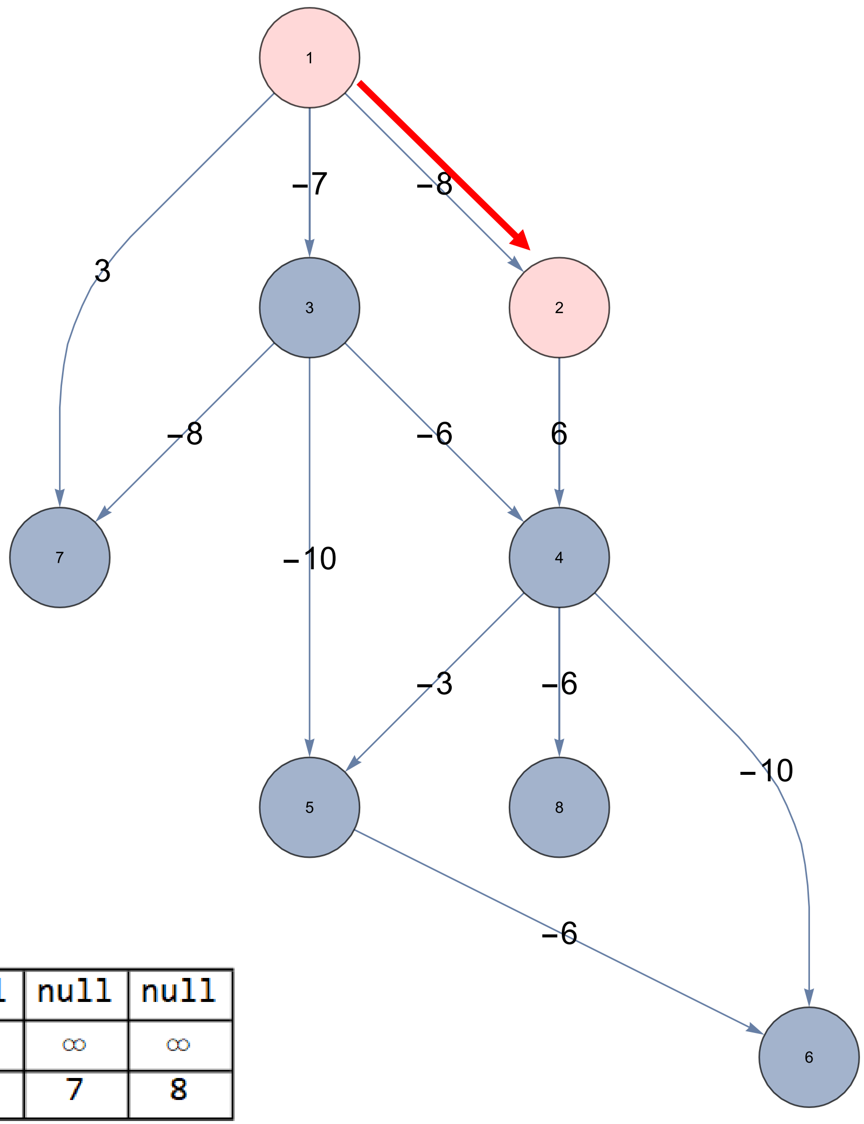
single source shortest path
on directed acyclic graph



from	1	null	null	null	null	null	null	null
distance	0	∞	∞	∞	∞	∞	∞	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

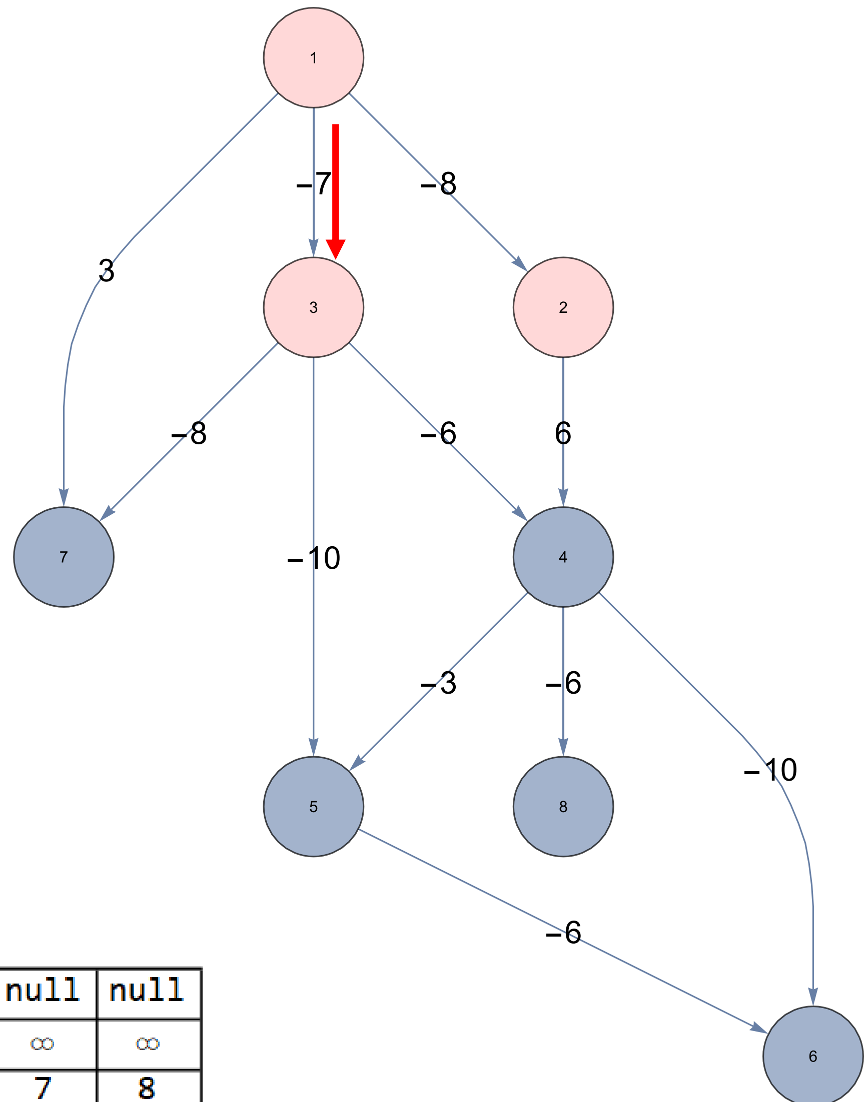
single source shortest path
on directed acyclic graph



from	1	1	null	null	null	null	null	null
distance	0	−8	∞	∞	∞	∞	∞	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

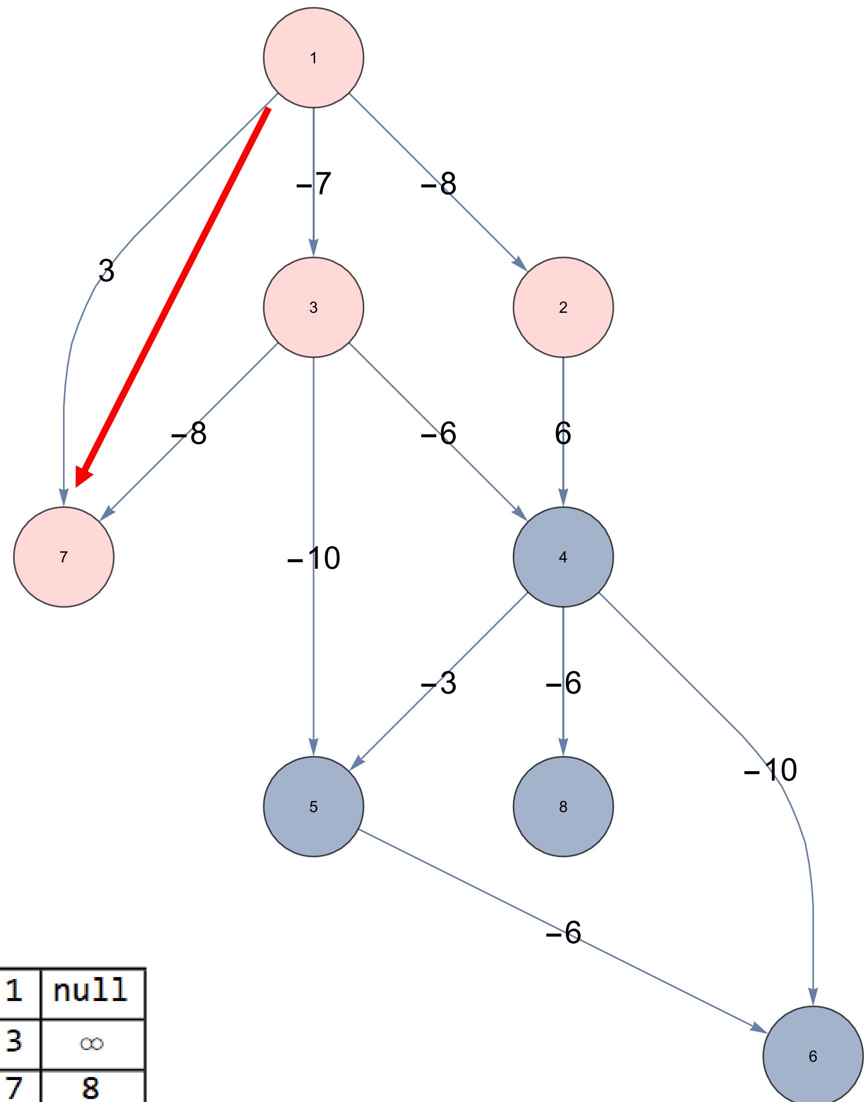
single source shortest path
on directed acyclic graph



from	1	1	1	null	null	null	null	null
distance	0	−8	−7	∞	∞	∞	∞	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

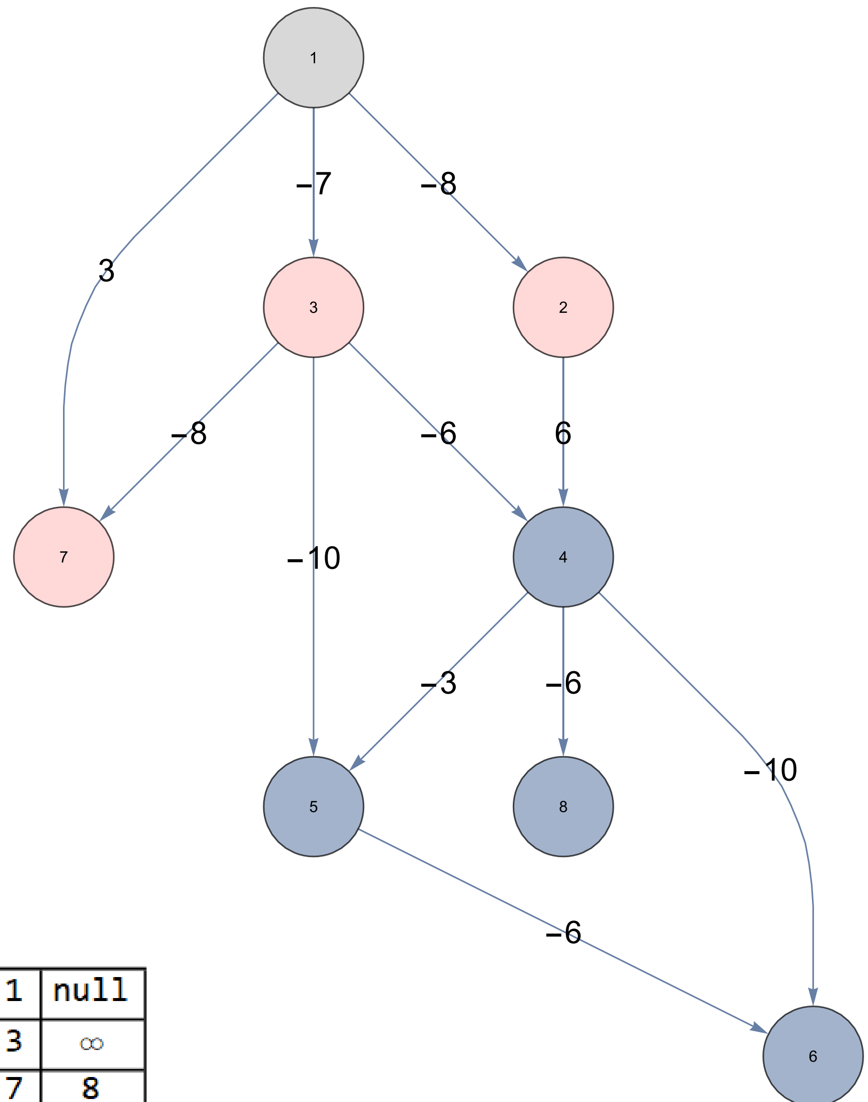
single source shortest path
on directed acyclic graph



from	1	1	1	null	null	null	1	null
distance	0	−8	−7	∞	∞	∞	3	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

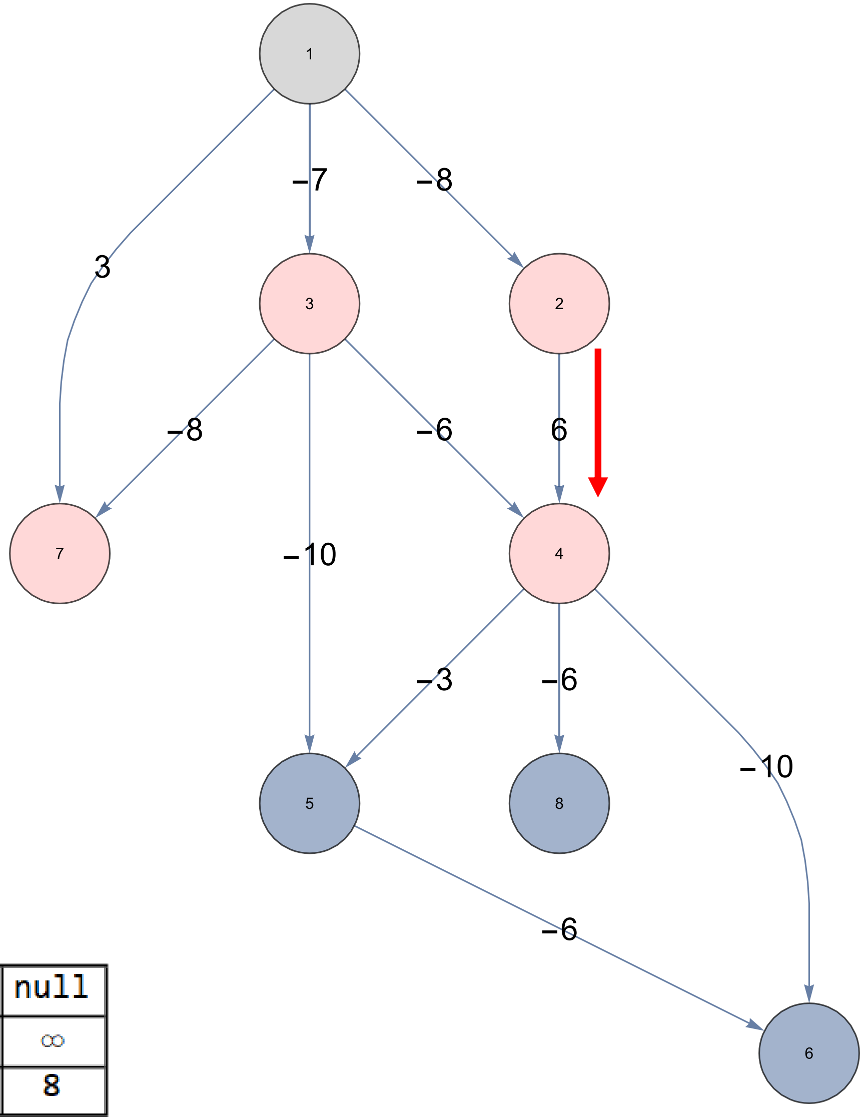
single source shortest path
on directed acyclic graph



from	1	1	1	null	null	null	1	null
distance	0	−8	−7	∞	∞	∞	3	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

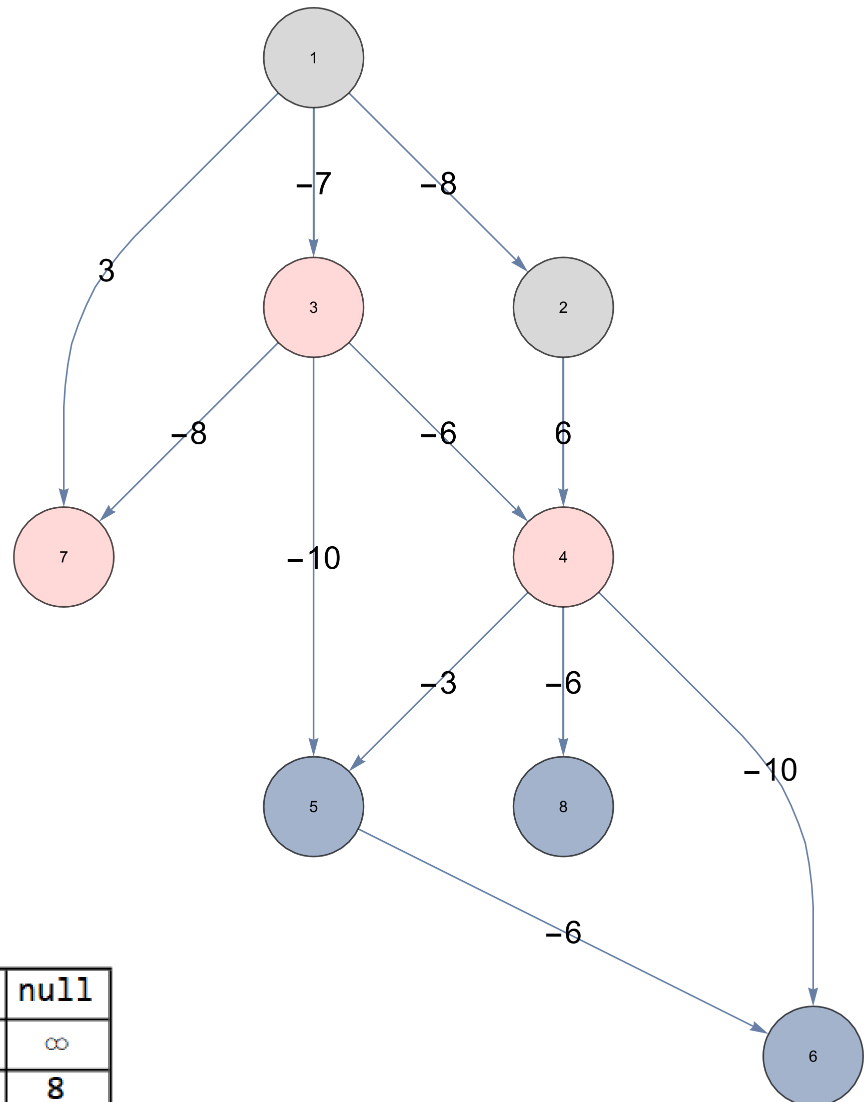
single source shortest path
on directed acyclic graph



from	1	1	1	2	null	null	1	null
distance	0	−8	−7	−2	∞	∞	3	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

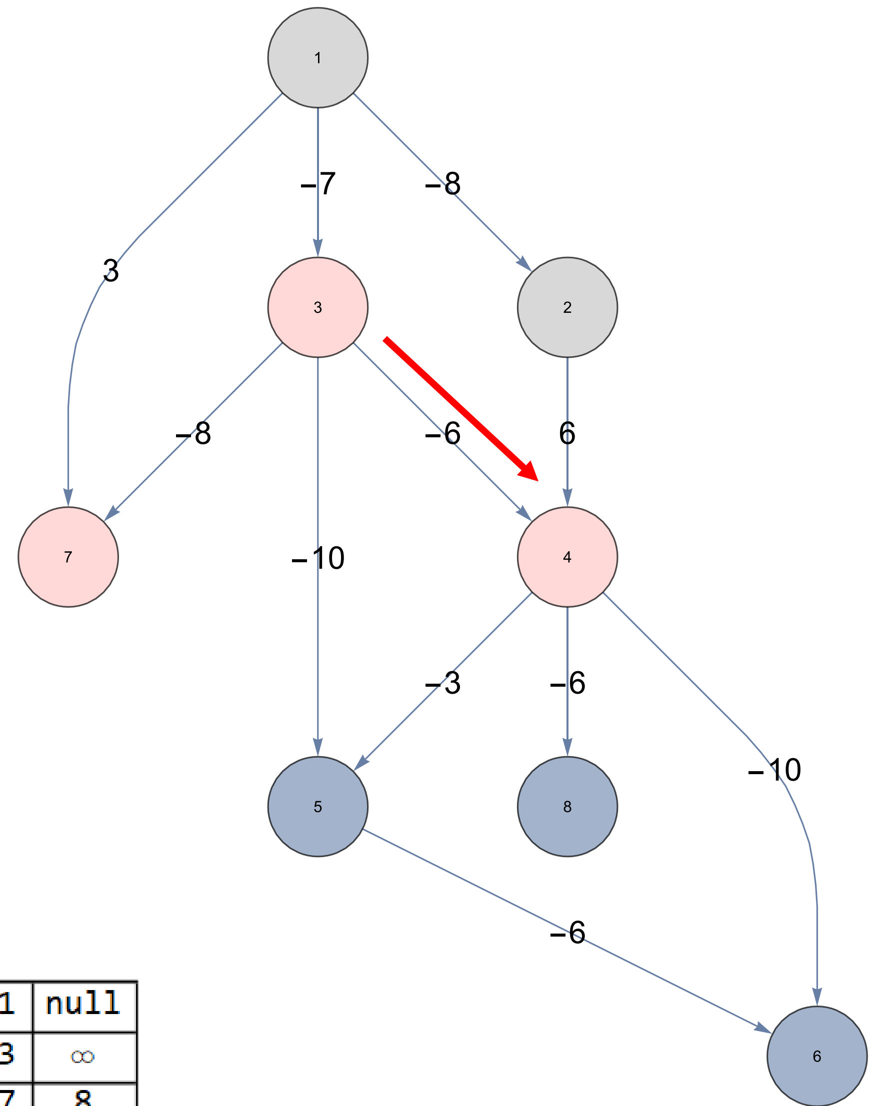
single source shortest path
on directed acyclic graph



from	1	1	1	2	null	null	1	null
distance	0	−8	−7	−2	∞	∞	3	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

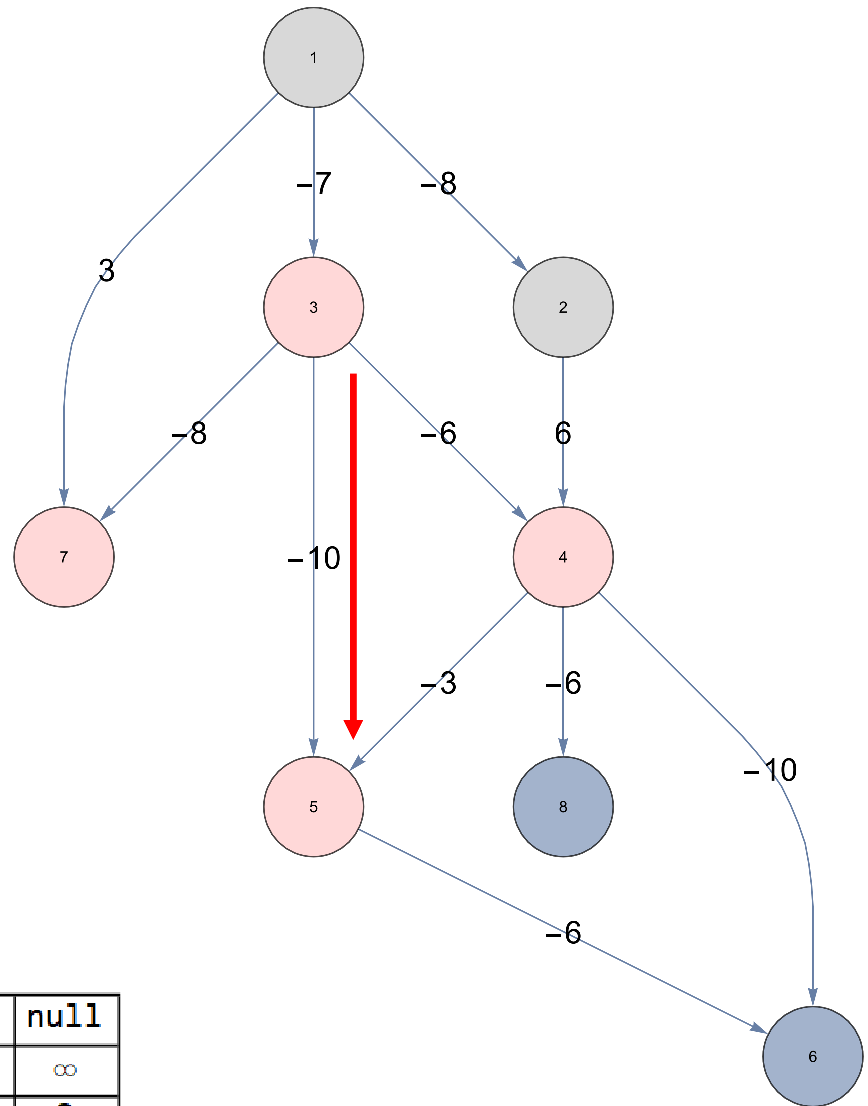
single source shortest path
on directed acyclic graph



from	1	1	1	3	null	null	1	null
distance	0	−8	−7	−13	∞	∞	3	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

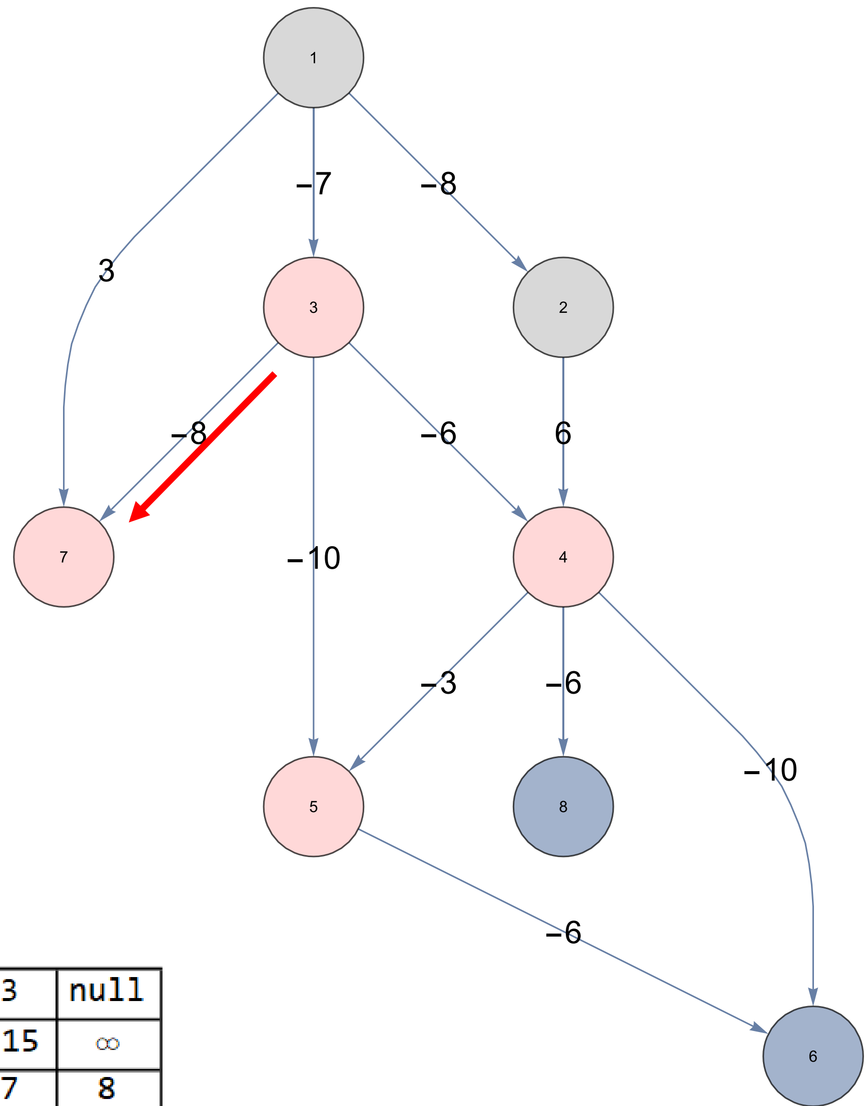
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	null	1	null
distance	0	−8	−7	−13	−17	∞	3	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

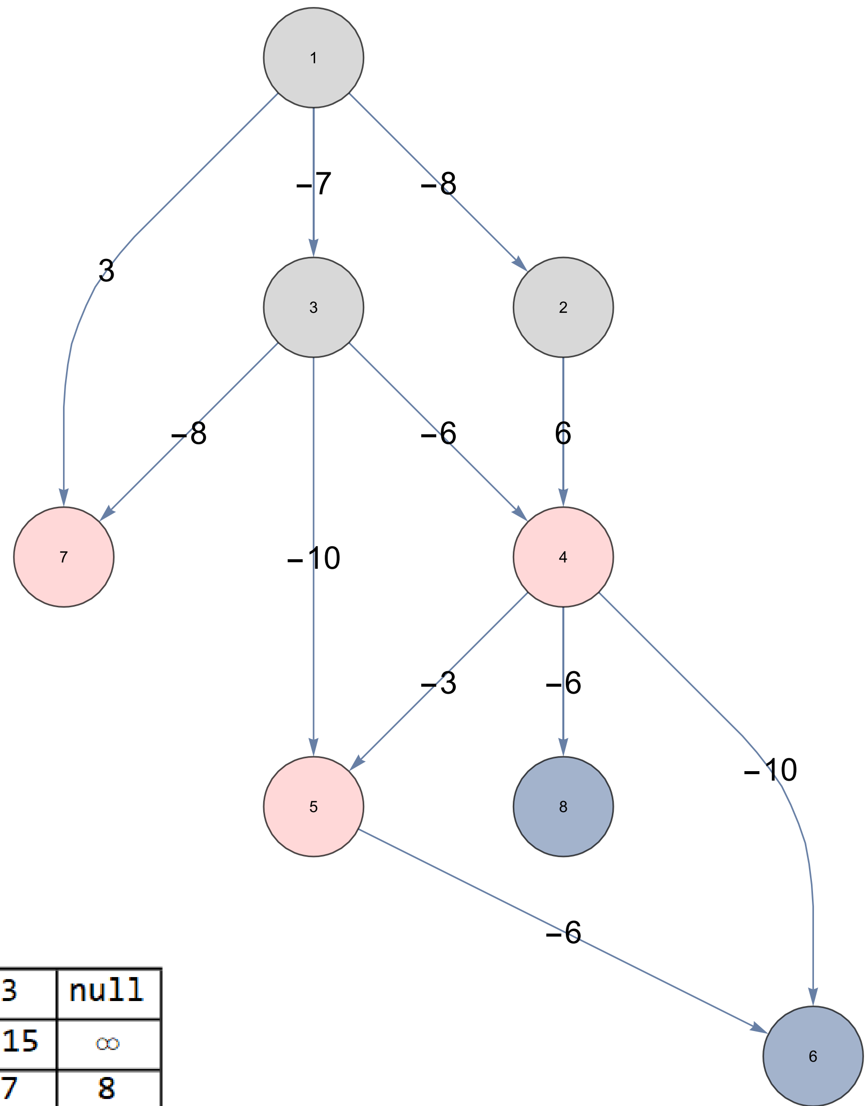
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	null	3	null
distance	0	−8	−7	−13	−17	∞	−15	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

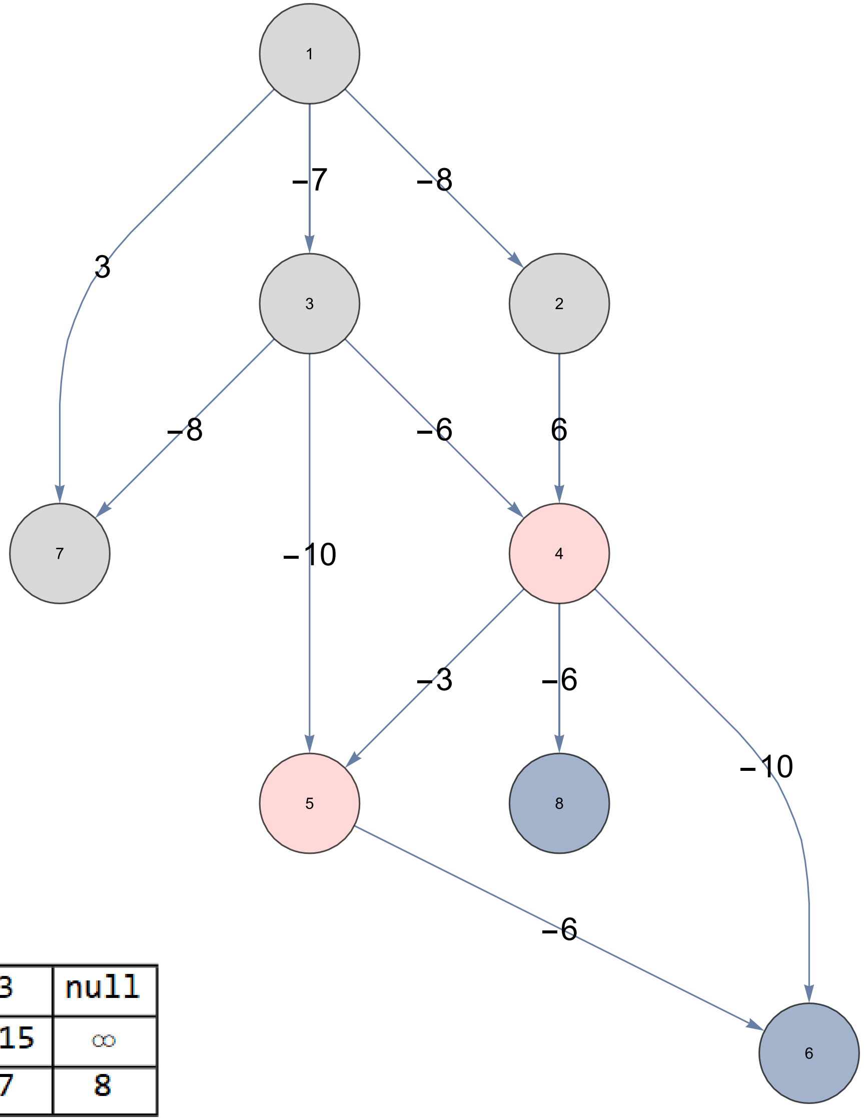
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	null	3	null
distance	0	−8	−7	−13	−17	∞	−15	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

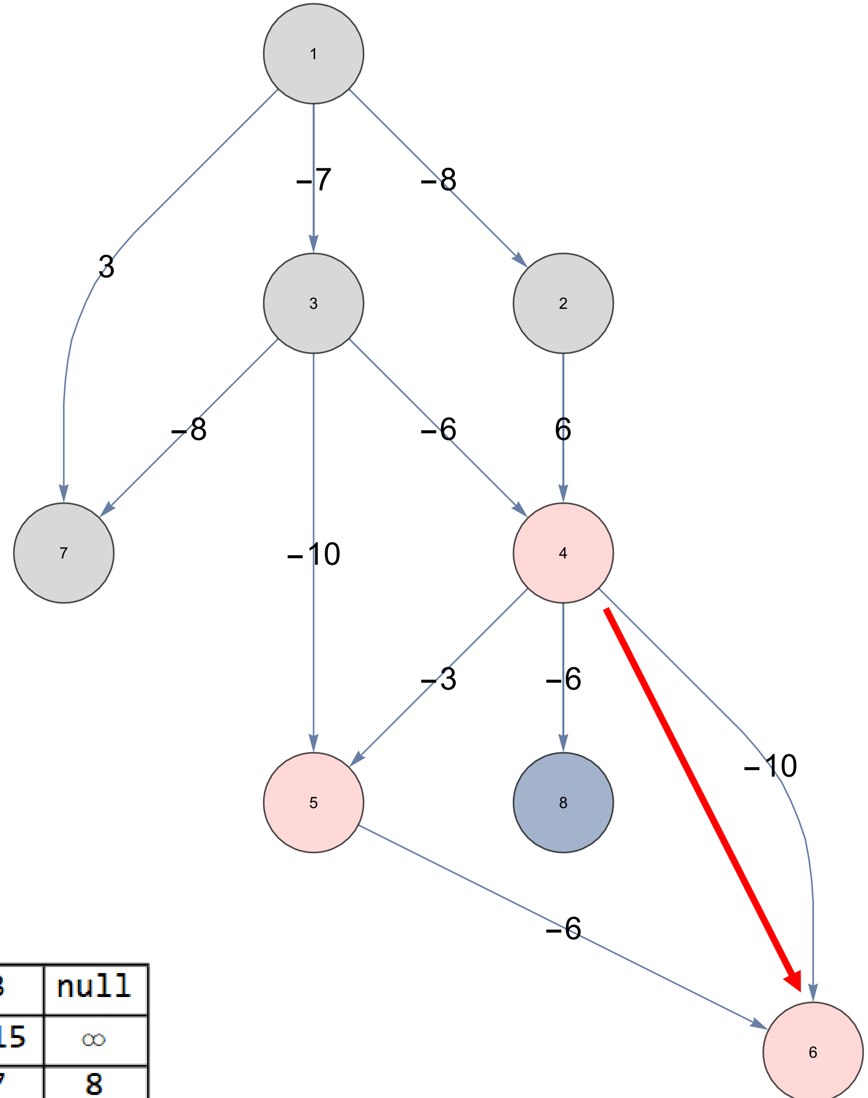
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	null	3	null
distance	0	−8	−7	−13	−17	∞	−15	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

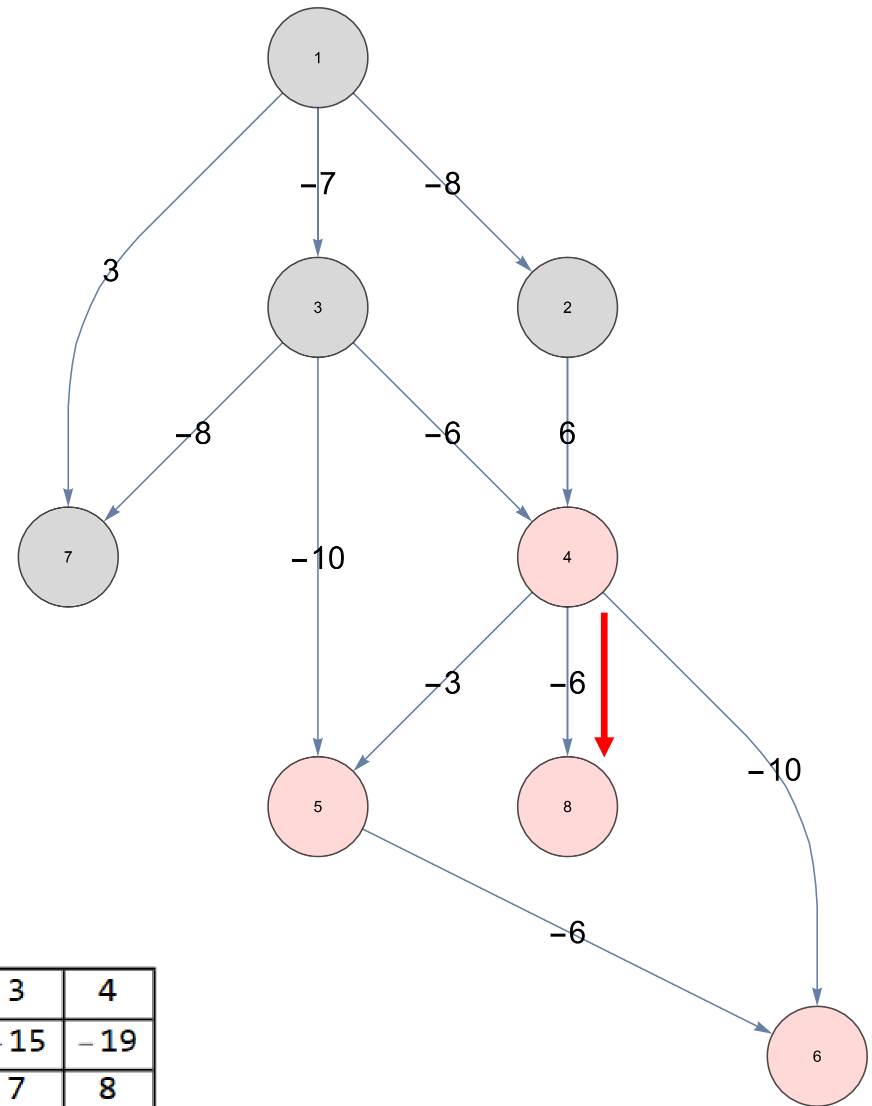
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	null
distance	0	−8	−7	−13	−17	−23	−15	∞
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

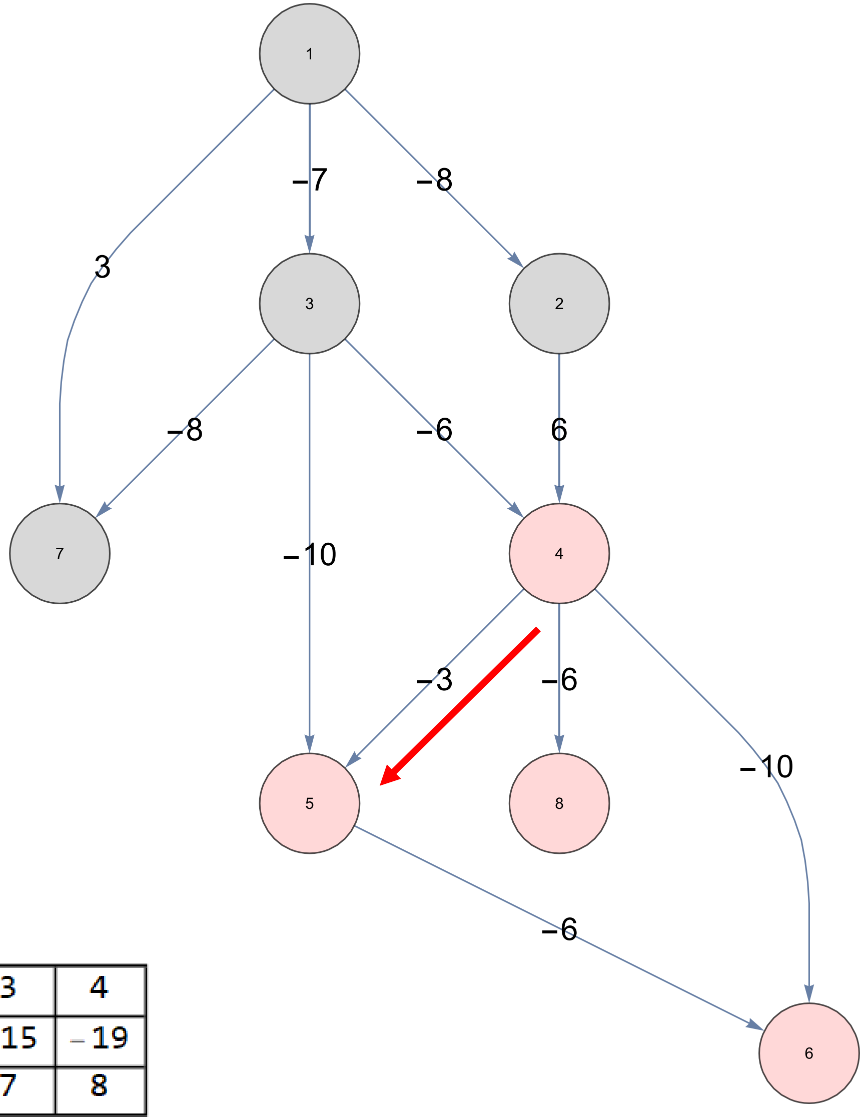
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	4
distance	0	−8	−7	−13	−17	−23	−15	−19
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

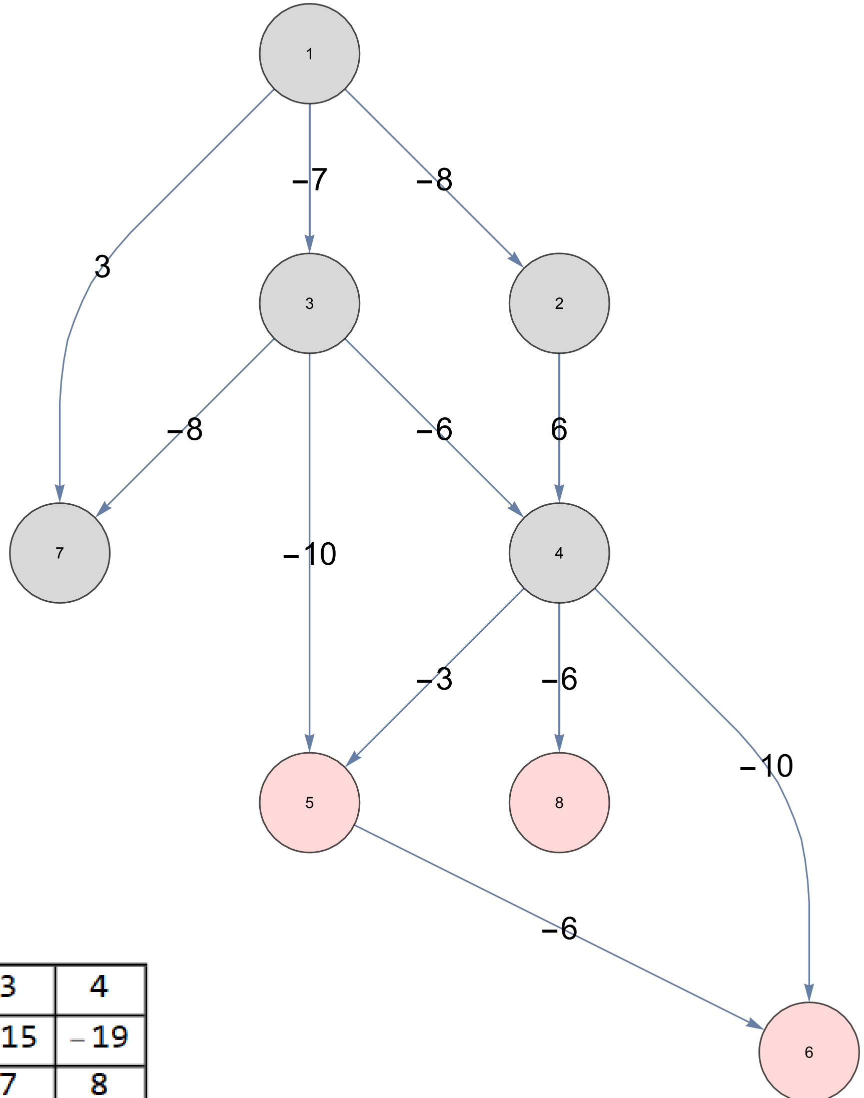
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	4
distance	0	−8	−7	−13	−17	−23	−15	−19
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

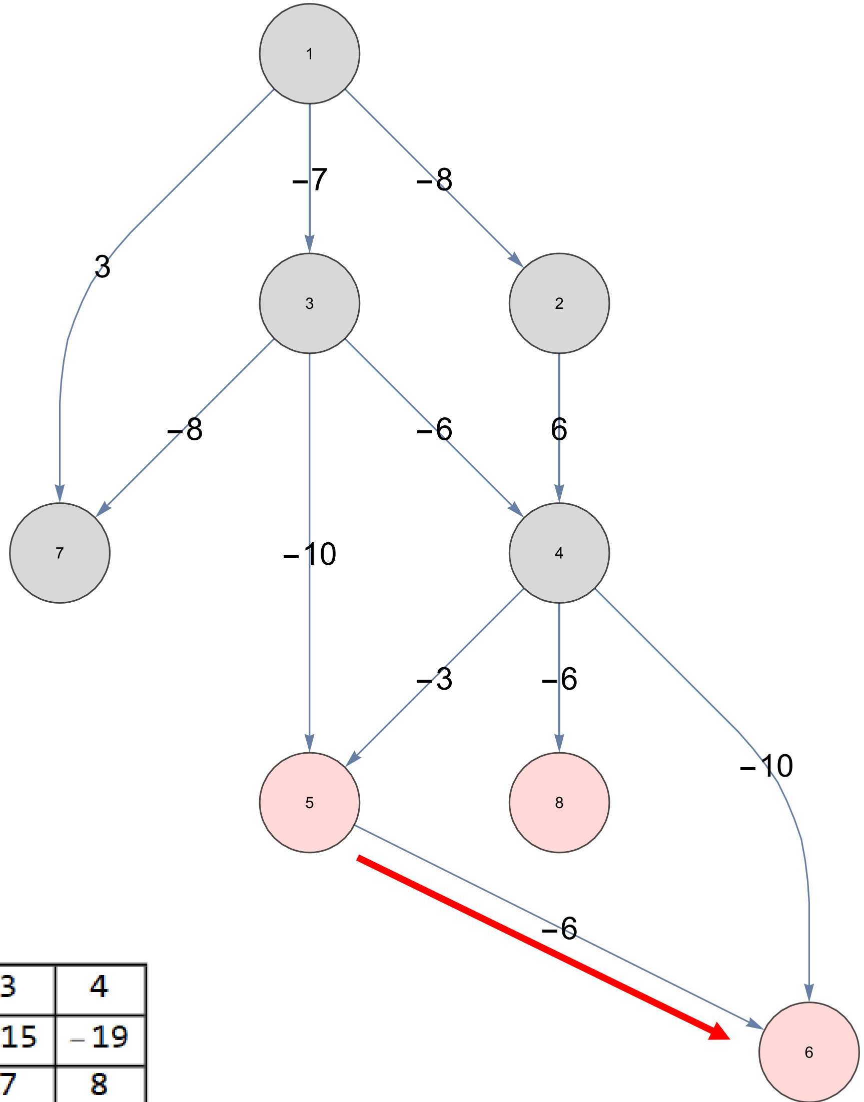
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	4
distance	0	−8	−7	−13	−17	−23	−15	−19
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

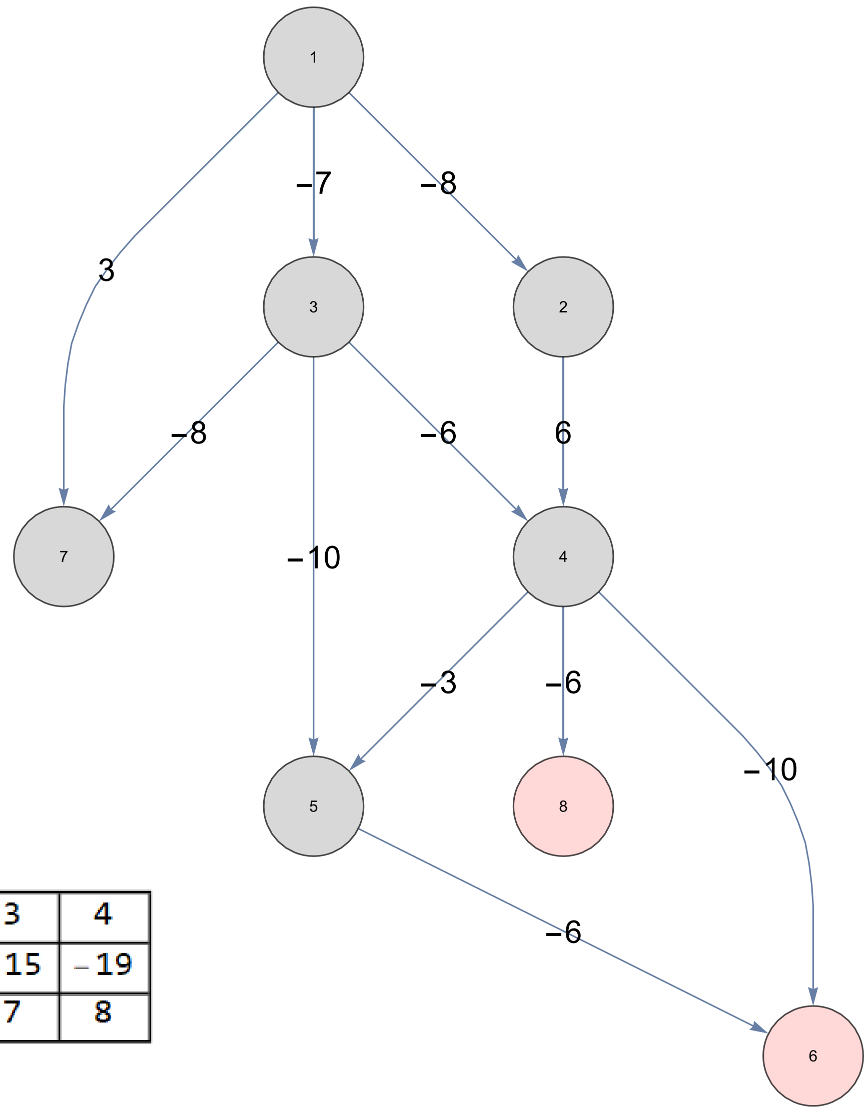
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	4
distance	0	−8	−7	−13	−17	−23	−15	−19
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

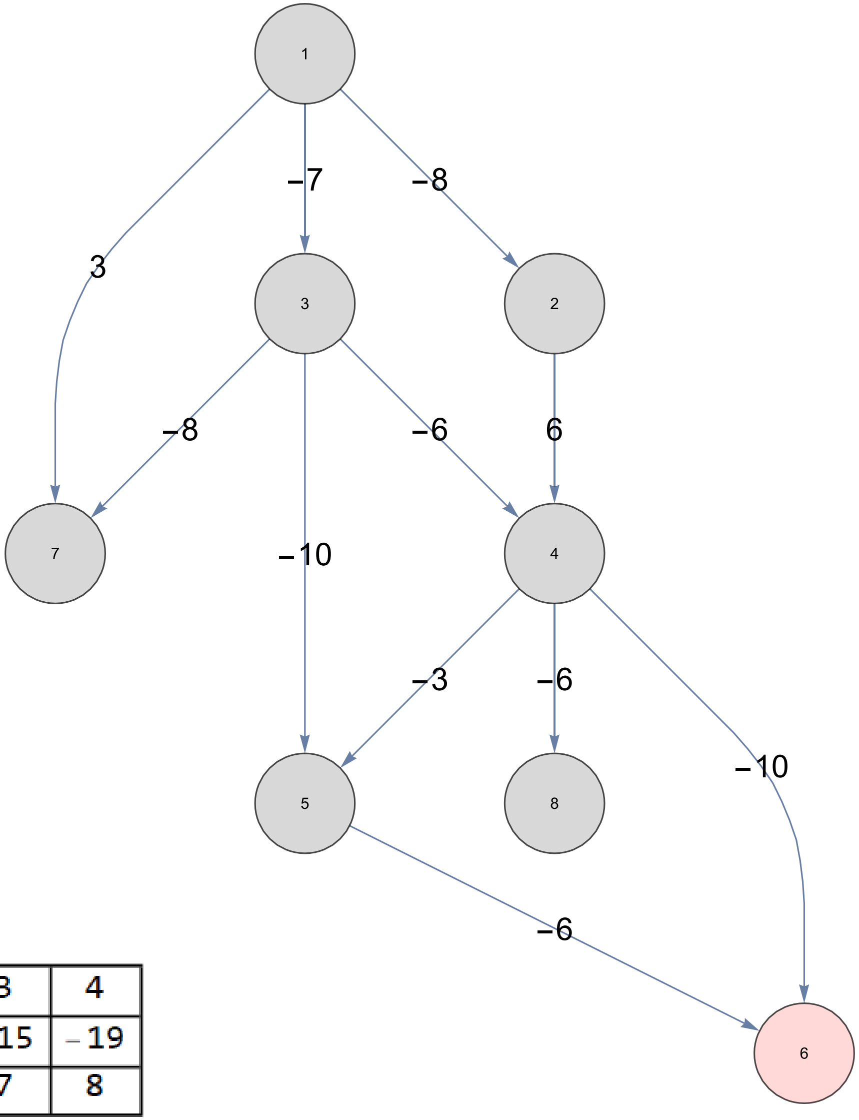
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	4
distance	0	−8	−7	−13	−17	−23	−15	−19
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

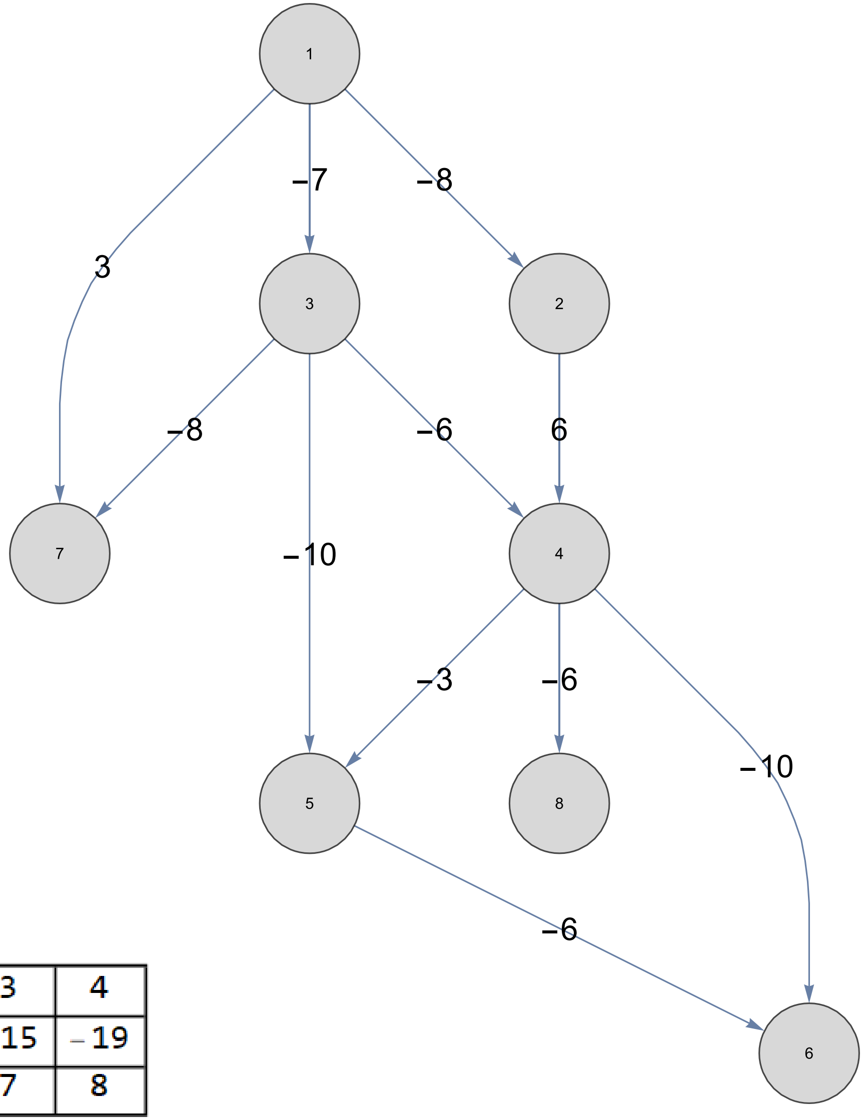
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	4
distance	0	−8	−7	−13	−17	−23	−15	−19
vertex	1	2	3	4	5	6	7	8

SSSP on DAG

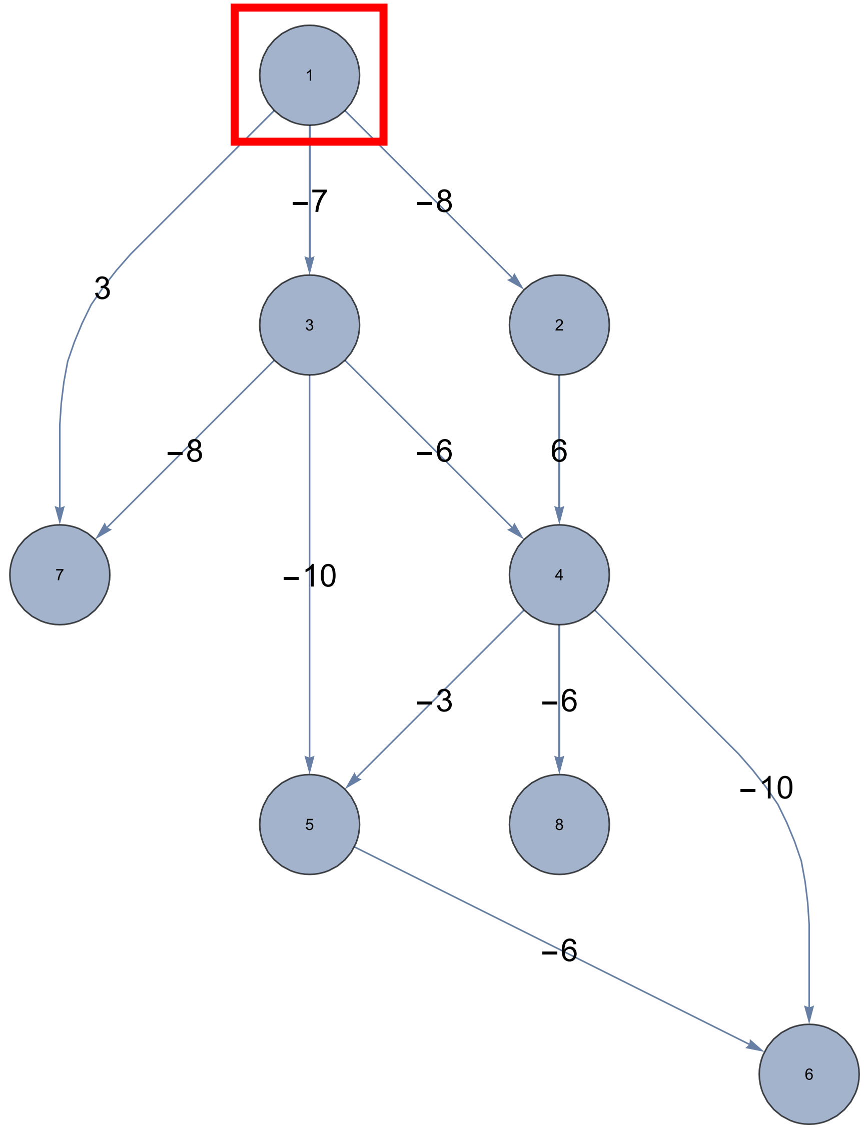
single source shortest path
on directed acyclic graph



from	1	1	1	3	3	4	3	4
distance	0	−8	−7	−13	−17	−23	−15	−19
vertex	1	2	3	4	5	6	7	8

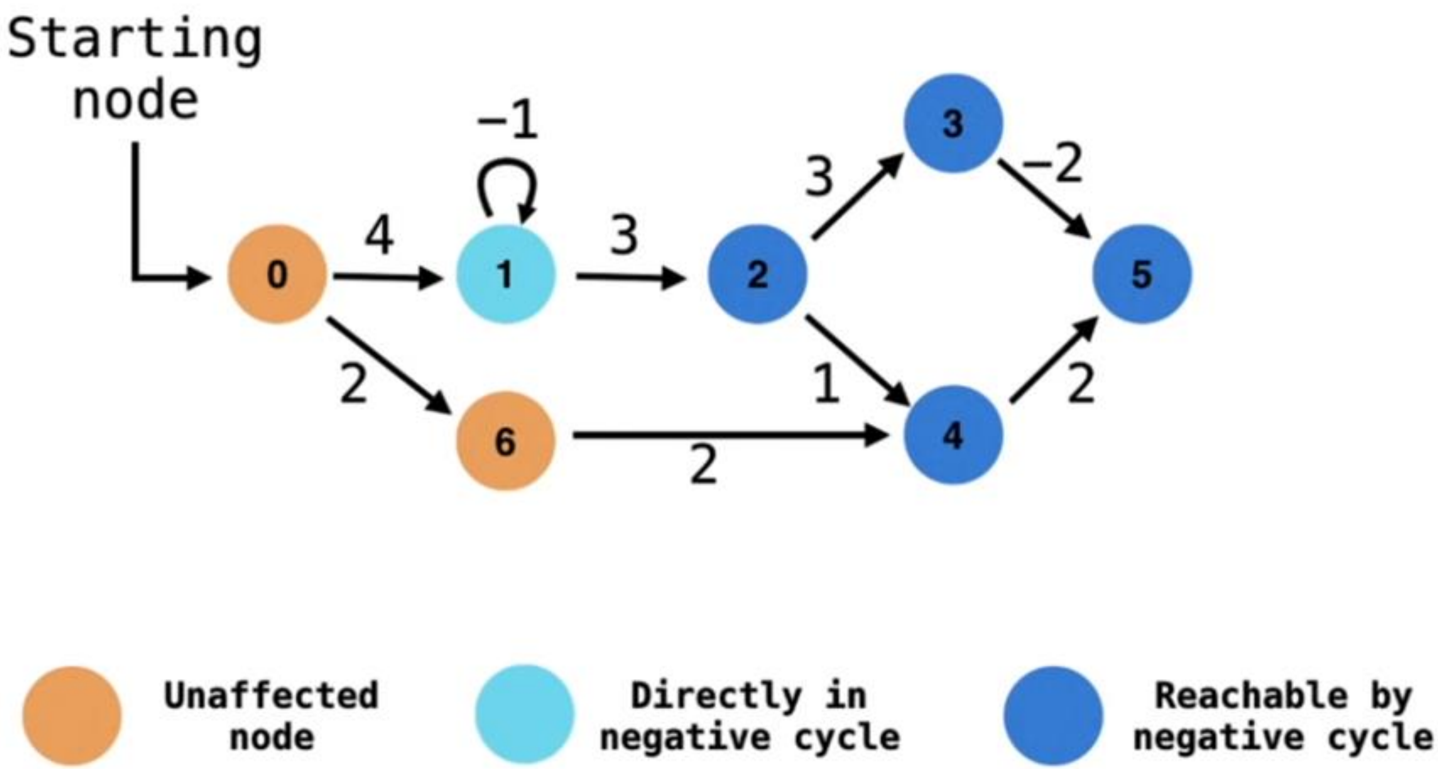
SSLP on DAG

single source longest path
on directed acyclic graph



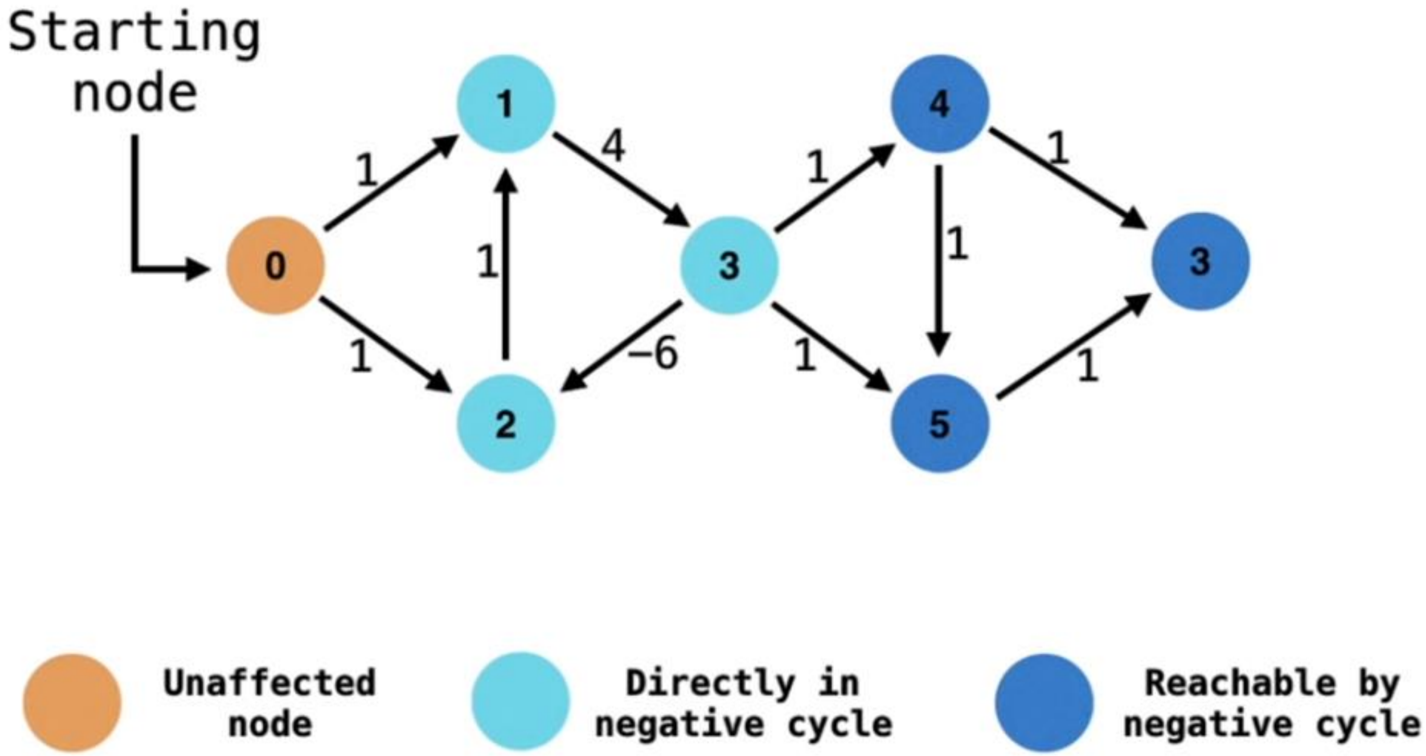
Negative Cycles

Negative cycles can manifest themselves in many ways…

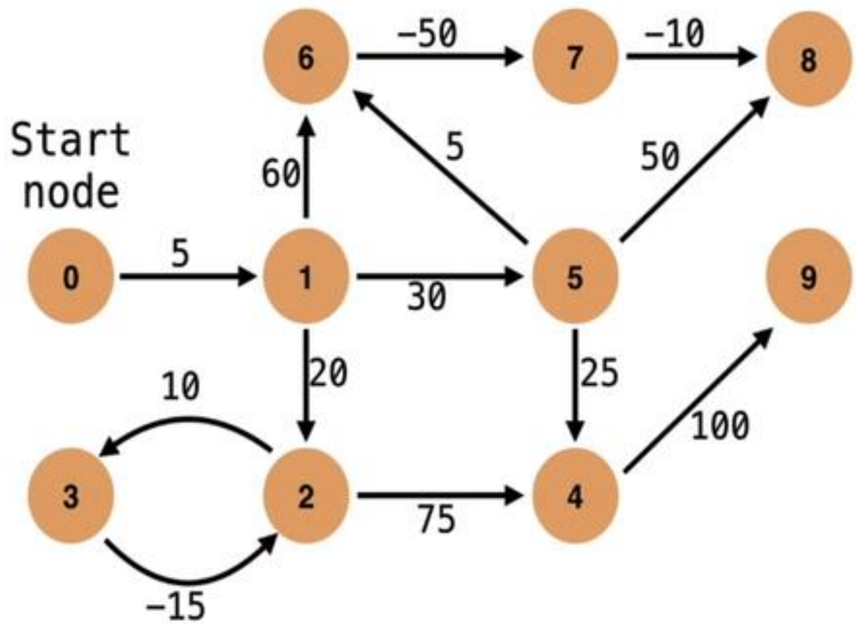


Negative Cycles

Negative cycles can manifest themselves in many ways…



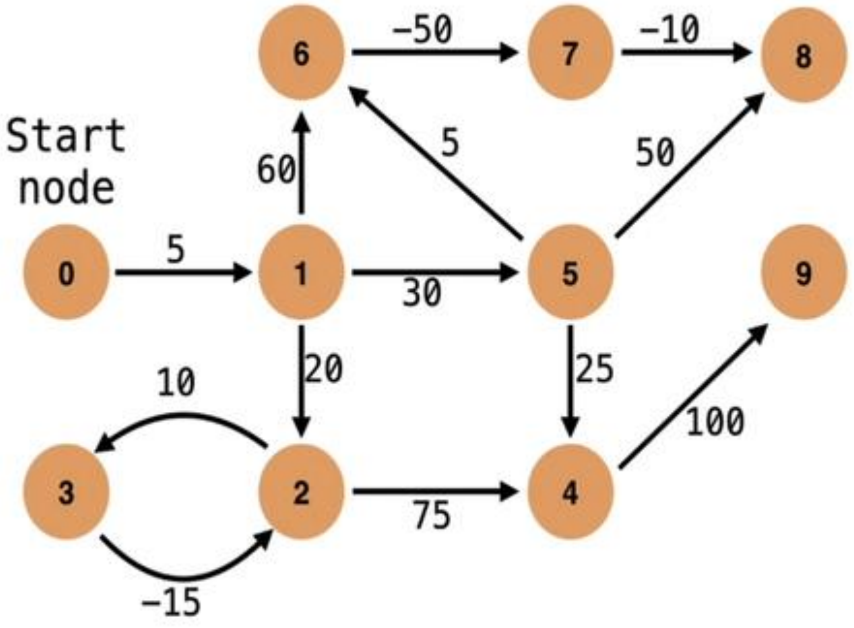
Negative Cycles



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

NOTE: The edges do not need to be chosen in any specific order.

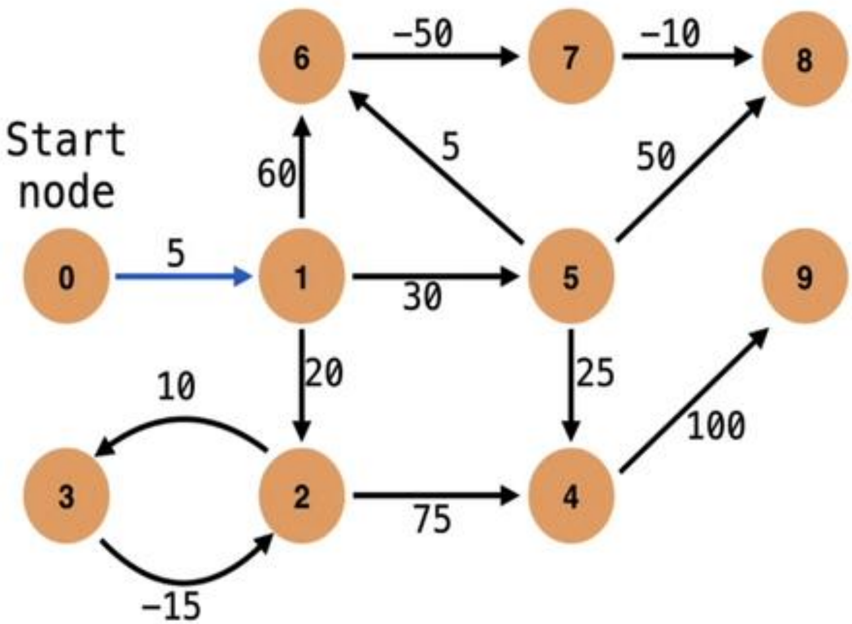
Negative Cycles



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

NOTE: The edges do not need to be chosen in any specific order.

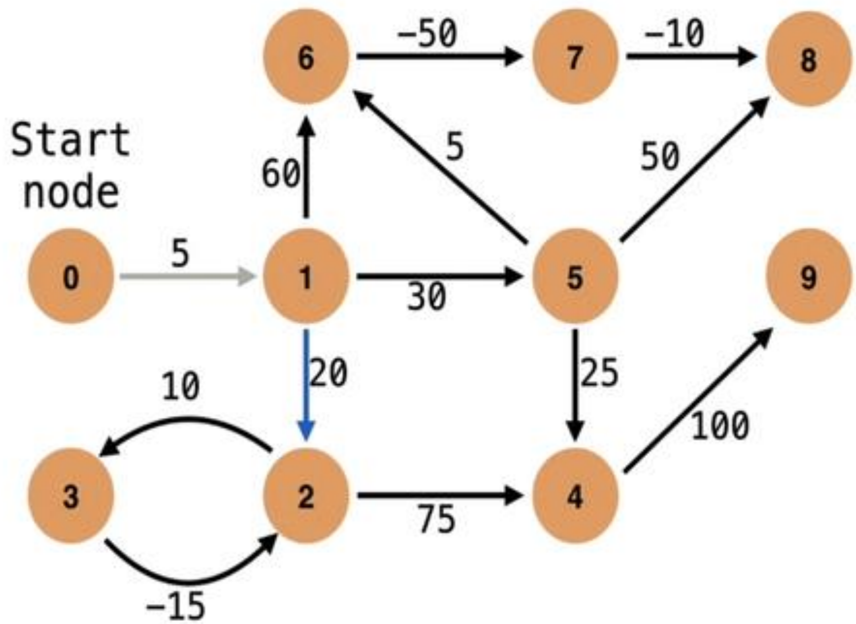
Negative Cycles



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

NOTE: The edges do not need to be chosen in any specific order.

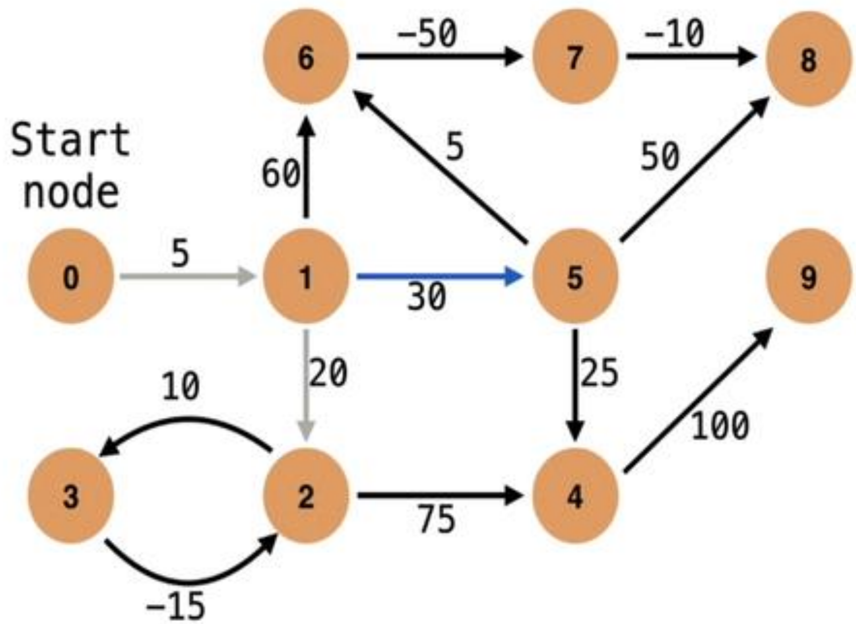
Negative Cycles



0	0
1	5
2	25
3	∞
4	∞
5	∞
6	∞
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

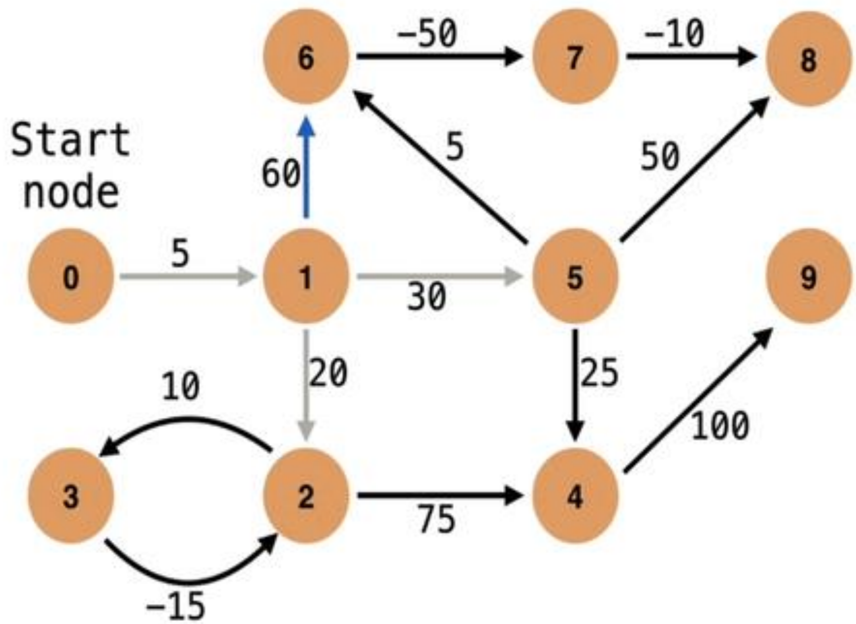
Negative Cycles



0	0
1	5
2	25
3	∞
4	∞
5	35
6	∞
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

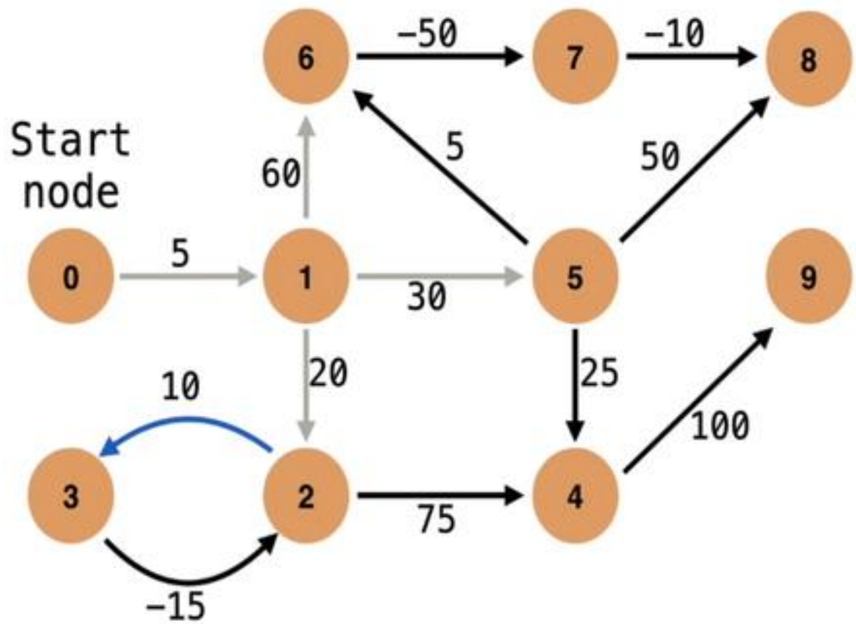
Negative Cycles



0	0
1	5
2	25
3	∞
4	∞
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

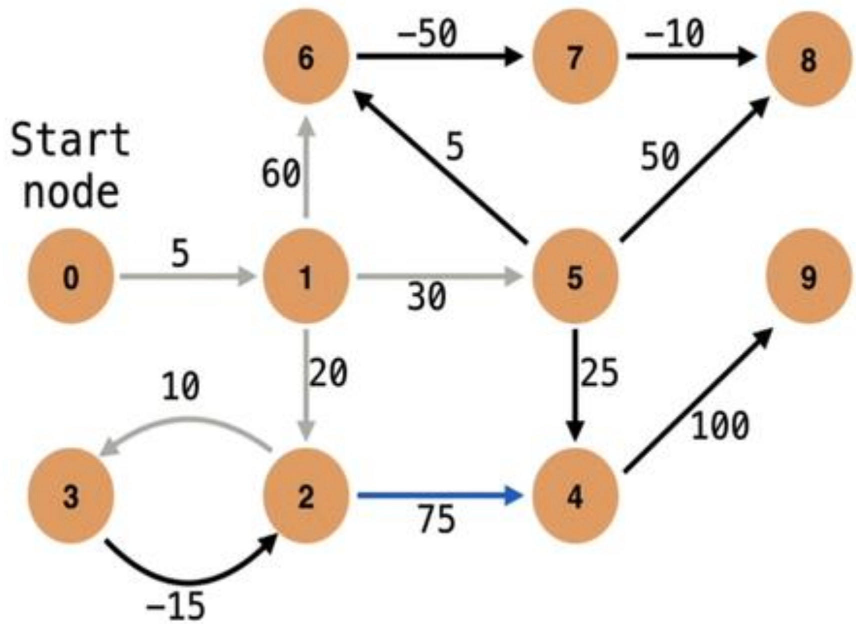
Negative Cycles



0	0
1	5
2	25
3	35
4	∞
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

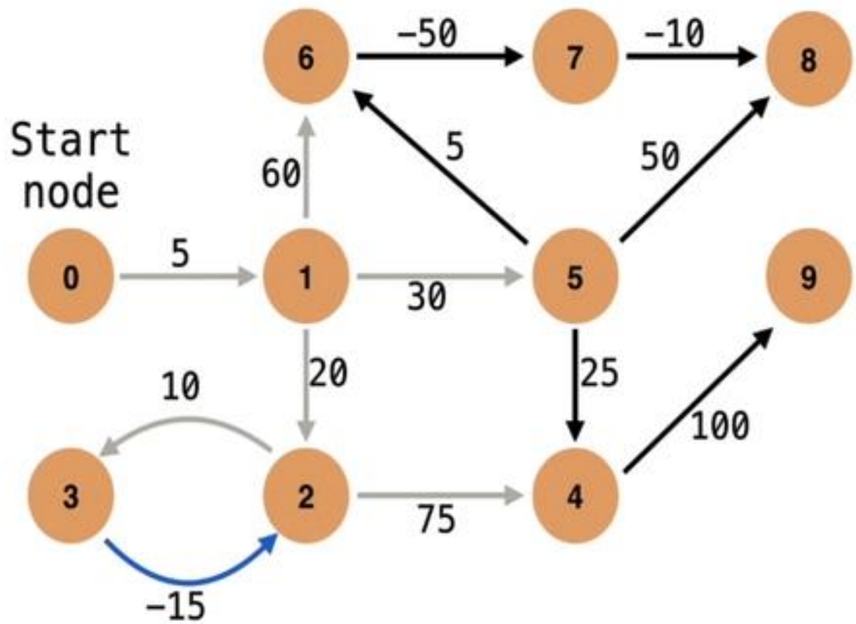
Negative Cycles



0	0
1	5
2	25
3	35
4	100
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

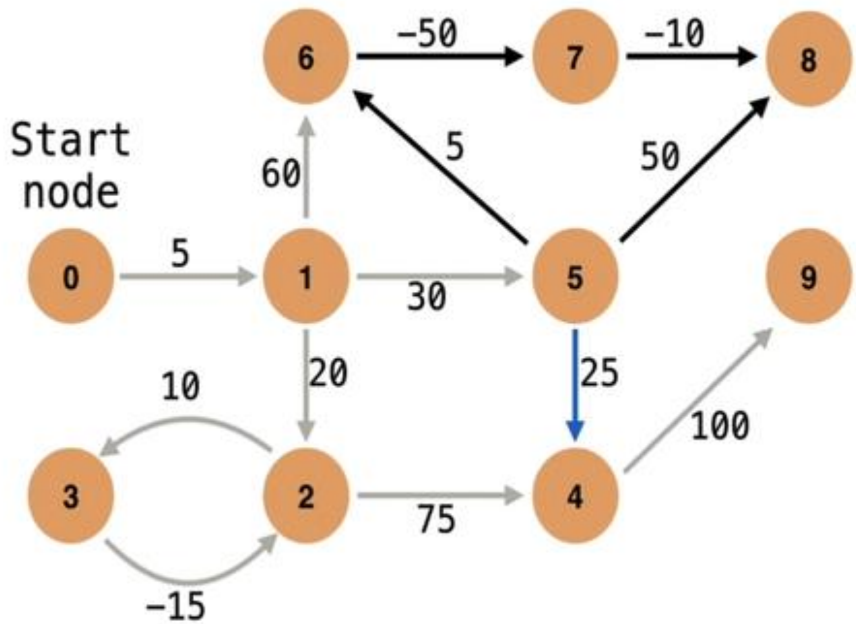
Negative Cycles



0	0
1	5
2	20
3	35
4	100
5	35
6	65
7	∞
8	∞
9	∞

NOTE: The edges do not need to be chosen in any specific order.

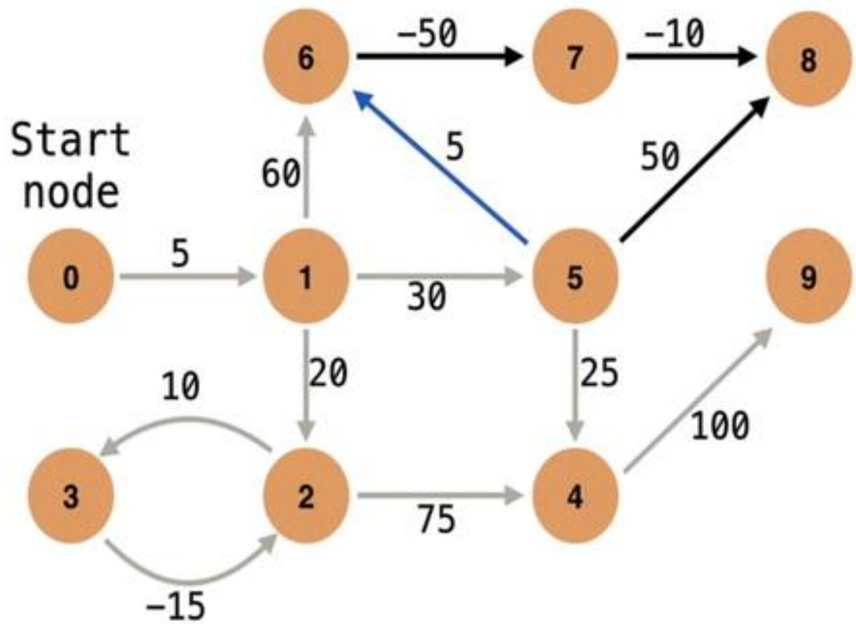
Negative Cycles



0	0
1	5
2	20
3	35
4	60
5	35
6	65
7	∞
8	∞
9	200

NOTE: The edges do not need to be chosen in any specific order.

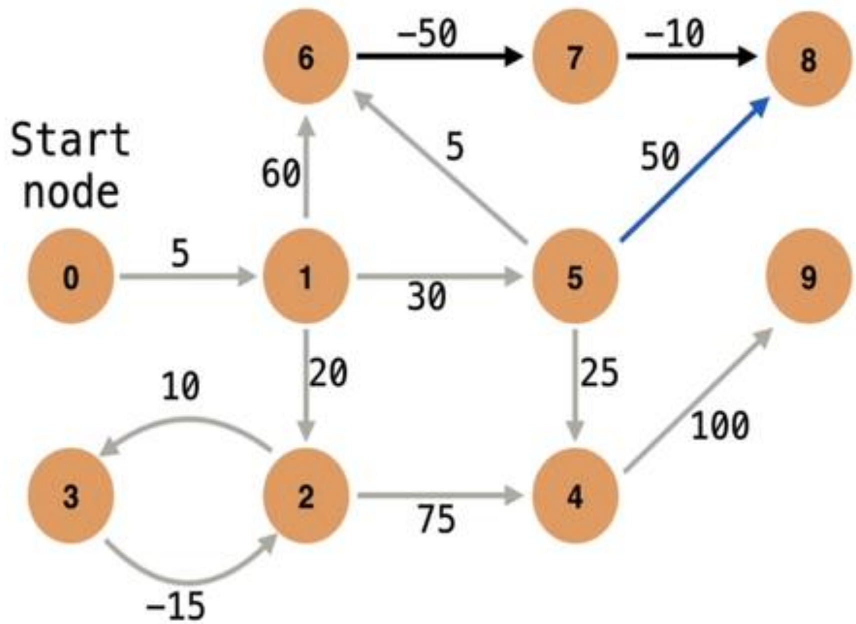
Negative Cycles



0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	∞
8	∞
9	200

NOTE: The edges do not need to be chosen in any specific order.

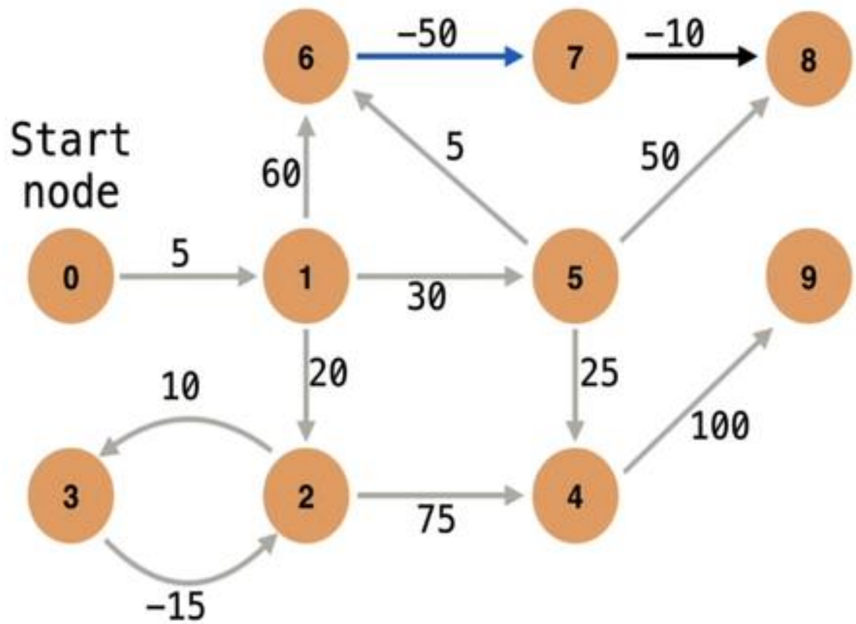
Negative Cycles



0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	∞
8	85
9	200

NOTE: The edges do not need to be chosen in any specific order.

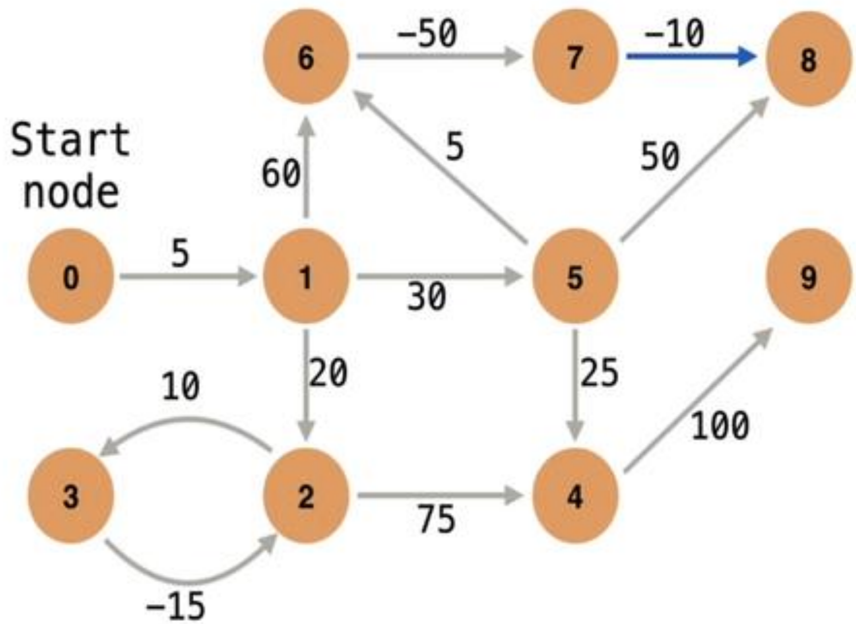
Negative Cycles



0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	85
9	200

NOTE: The edges do not need to be chosen in any specific order.

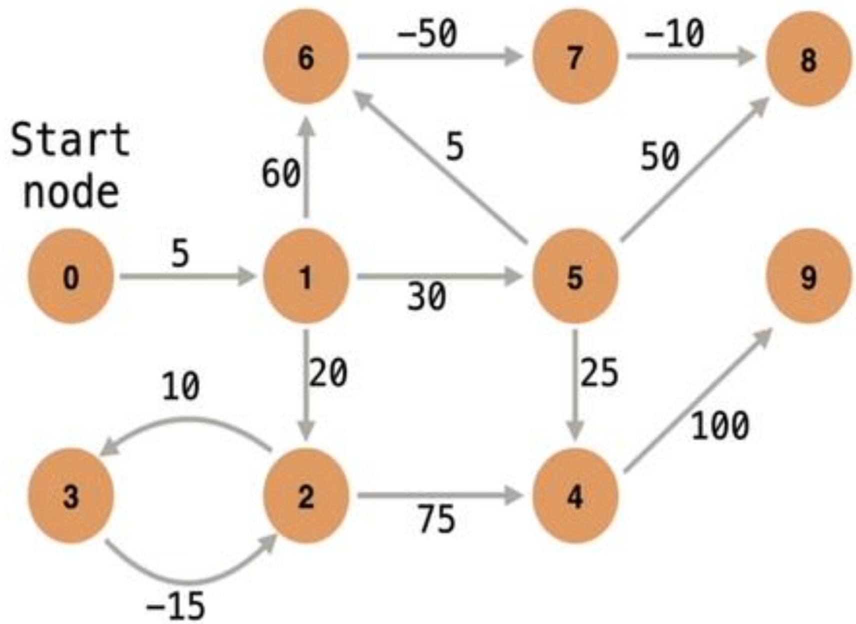
Negative Cycles



0	0
1	5
2	20
3	35
4	60
5	35
6	40
7	-10
8	-20
9	200

NOTE: The edges do not need to be chosen in any specific order.

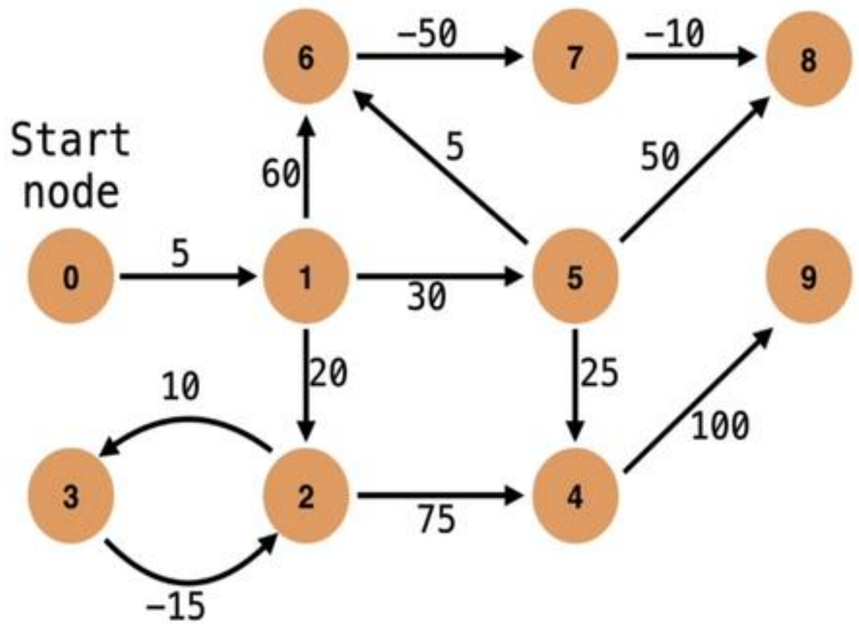
Negative Cycles



0	0
1	5
2	15
3	30
4	60
5	35
6	40
7	-10
8	-20
9	160

Iteration 2 complete, 7 more to go…
Let’s fast-forward to the end…

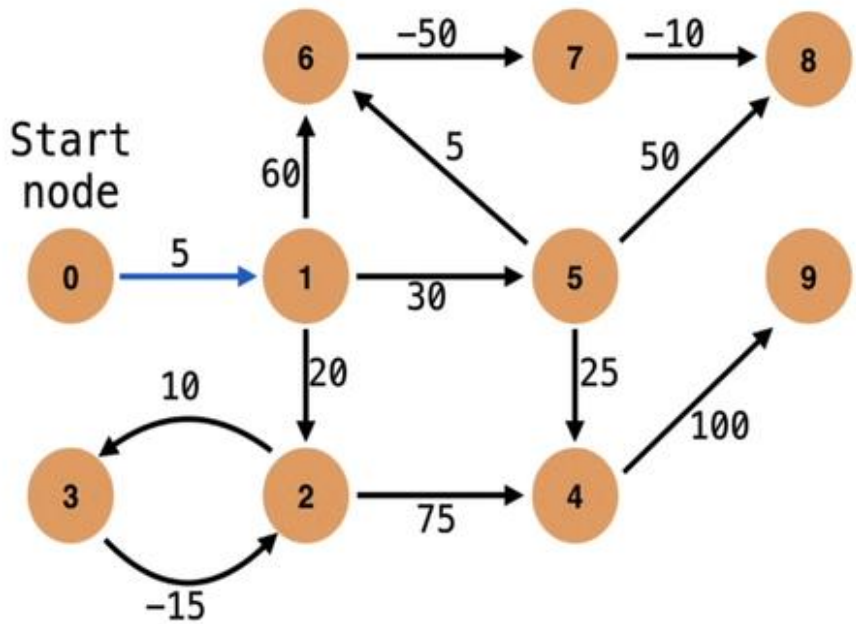
Negative Cycles



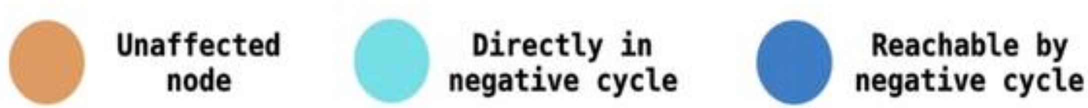
0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160

We're finished with the SSSP part. Now let's detect those negative cycles. If we can relax an edge then there's a negative cycle.

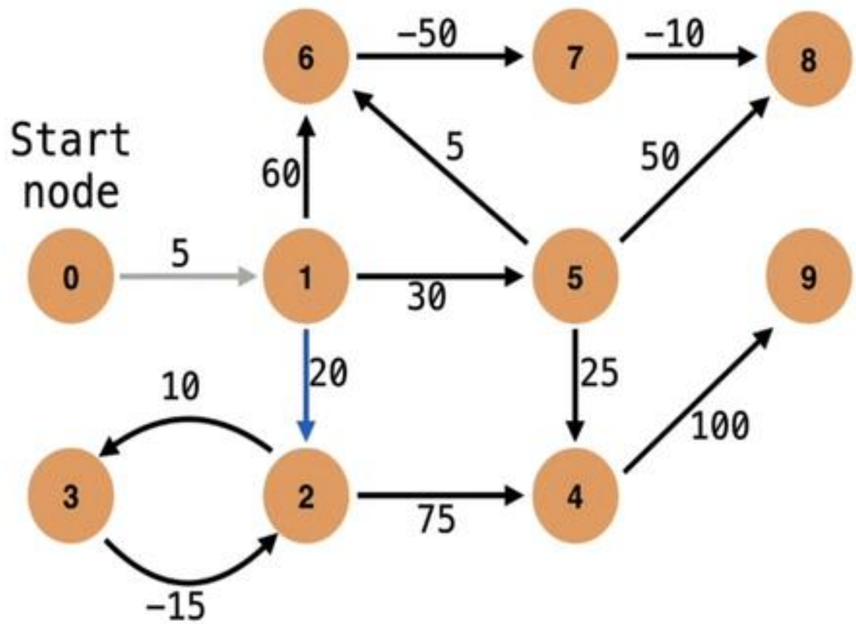
Negative Cycles



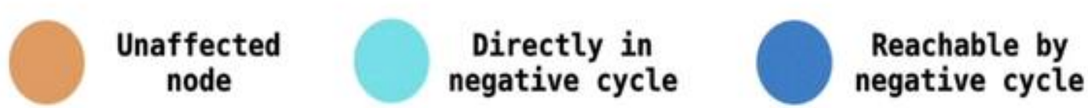
0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160



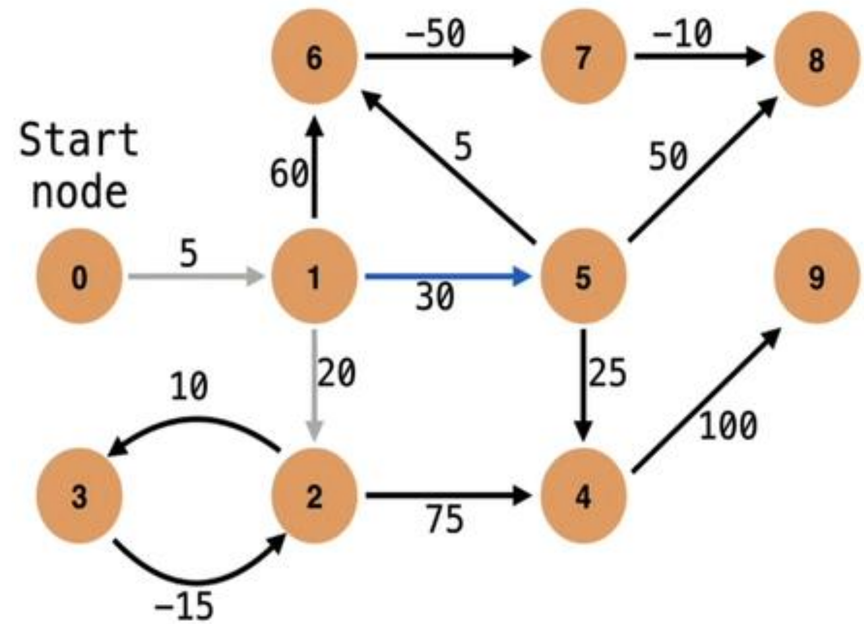
Negative Cycles



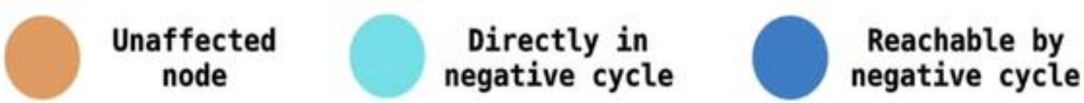
0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160



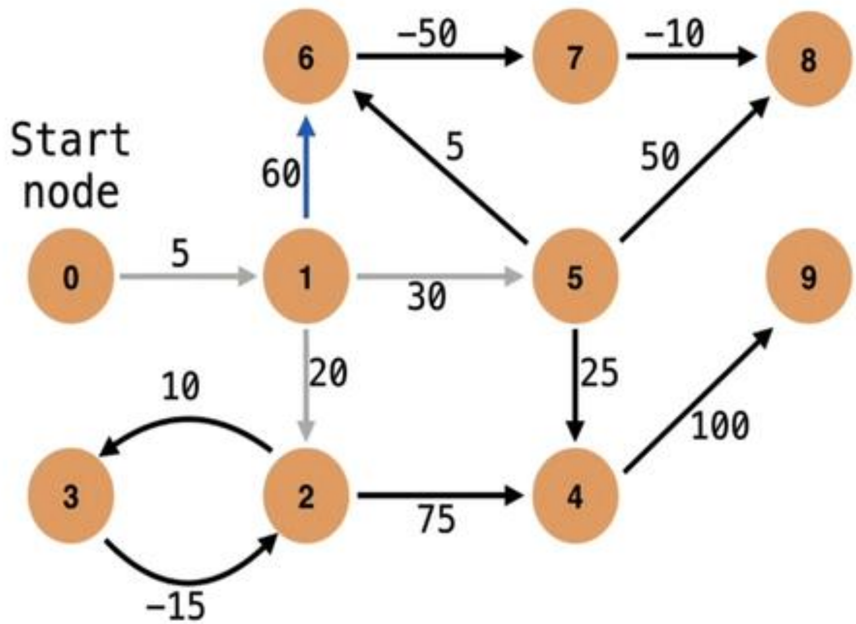
Negative Cycles



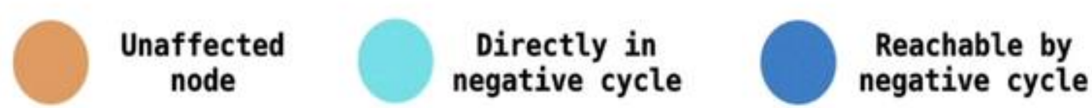
0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160



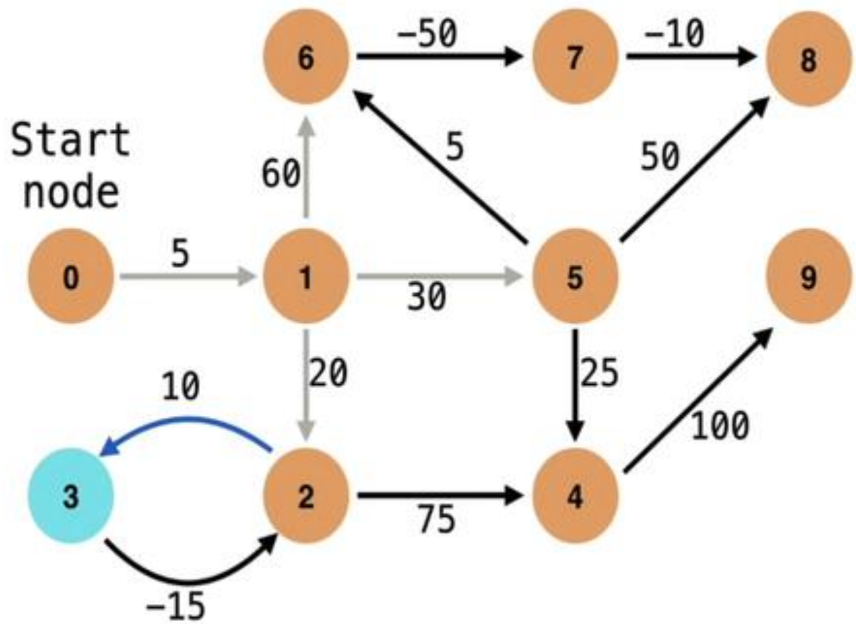
Negative Cycles



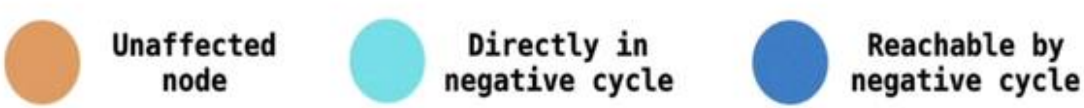
0	0
1	5
2	-20
3	-5
4	60
5	35
6	40
7	-10
8	-20
9	160



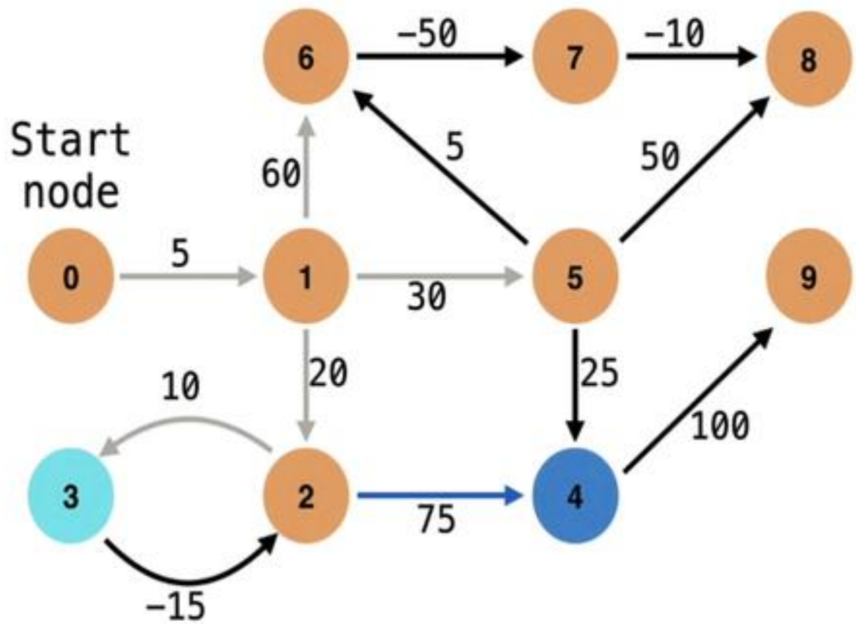
Negative Cycles



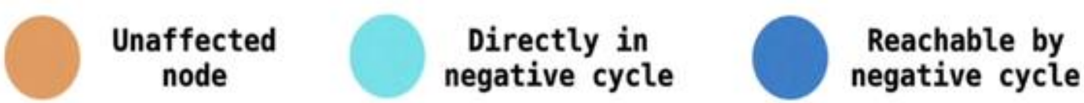
0	0
1	5
2	-20
3	-∞
4	60
5	35
6	40
7	-10
8	-20
9	160



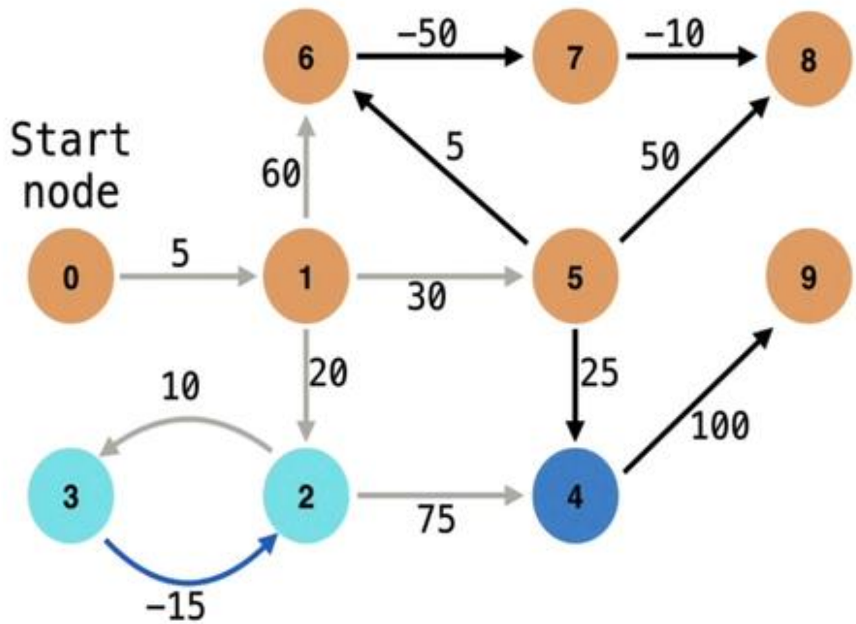
Negative Cycles



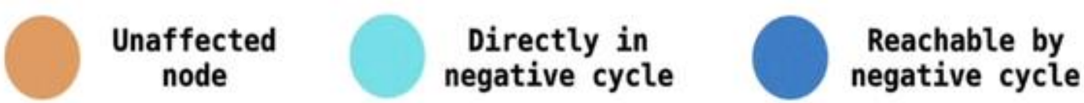
0	0
1	5
2	-20
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	160



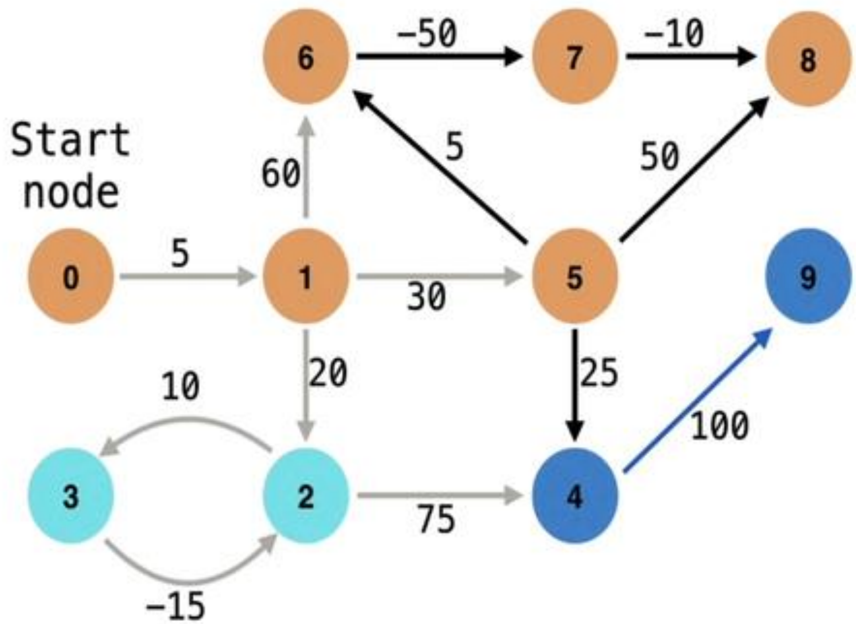
Negative Cycles



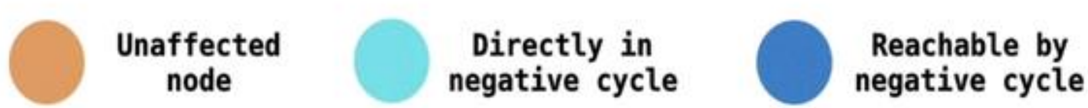
0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	160



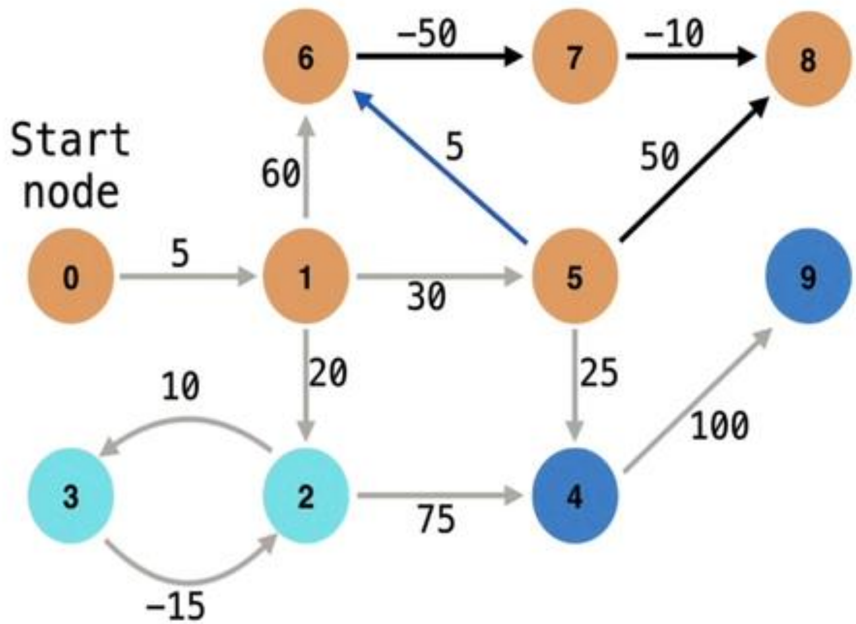
Negative Cycles



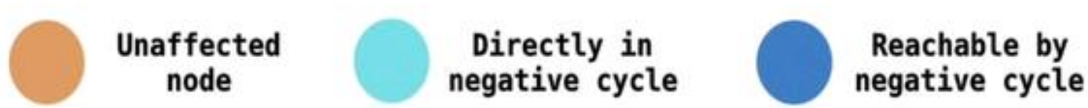
0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞



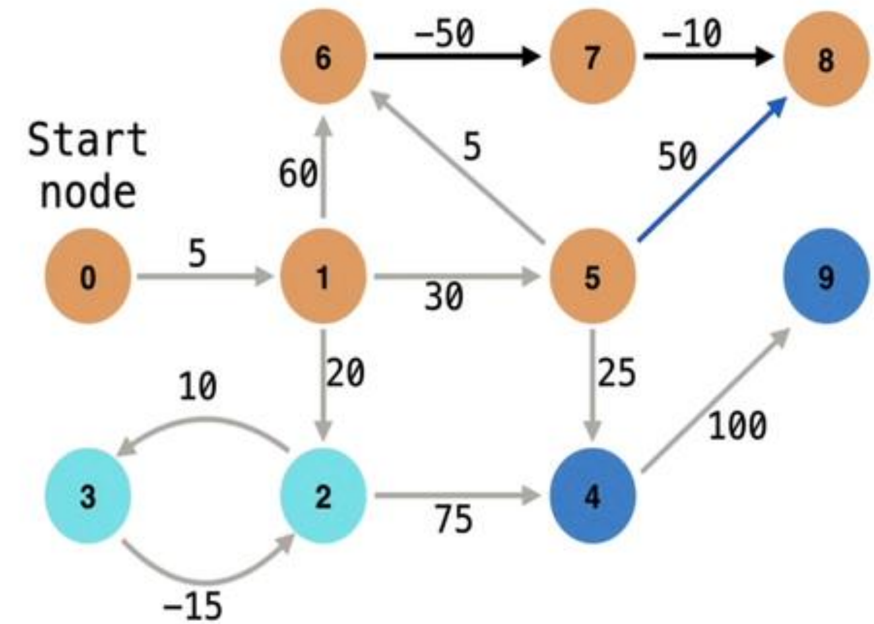
Negative Cycles



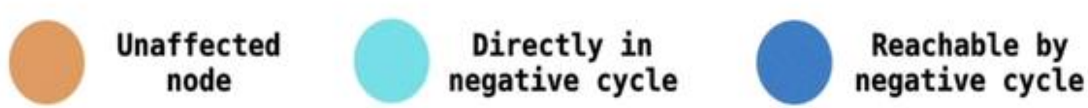
0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞



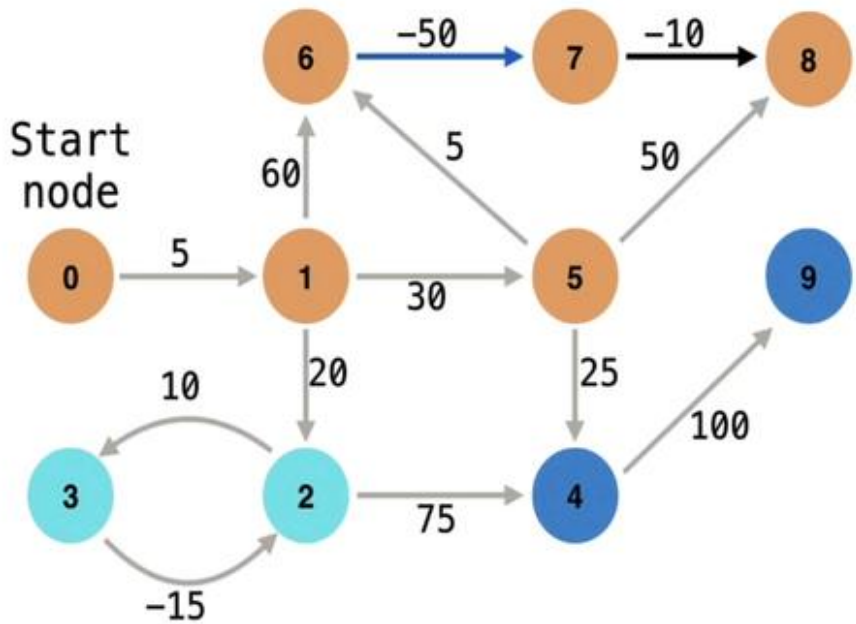
Negative Cycles



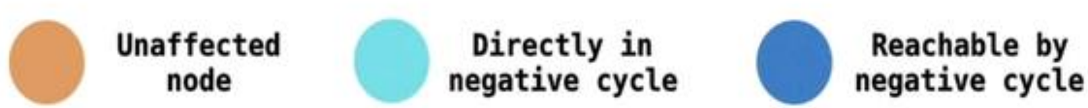
0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞



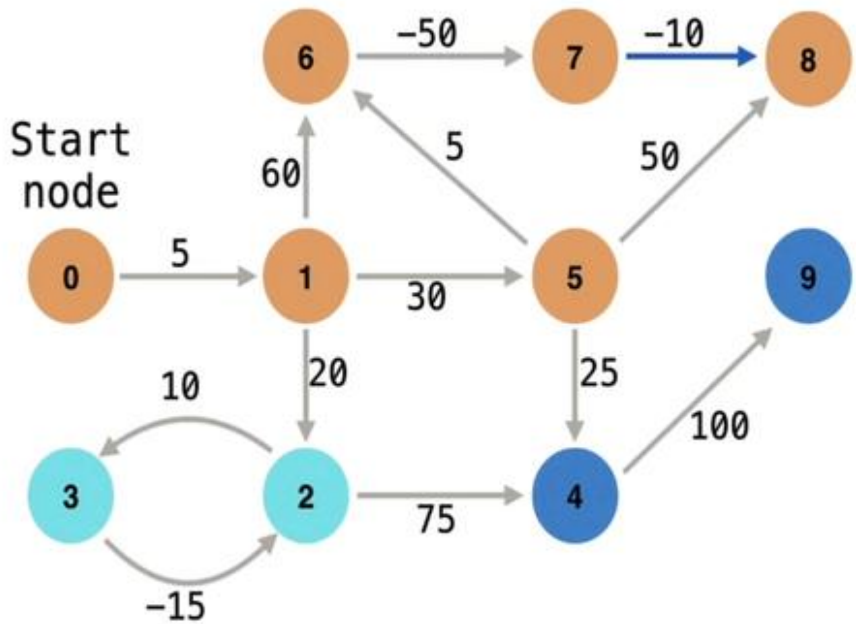
Negative Cycles



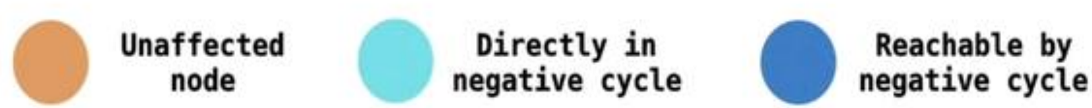
0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞



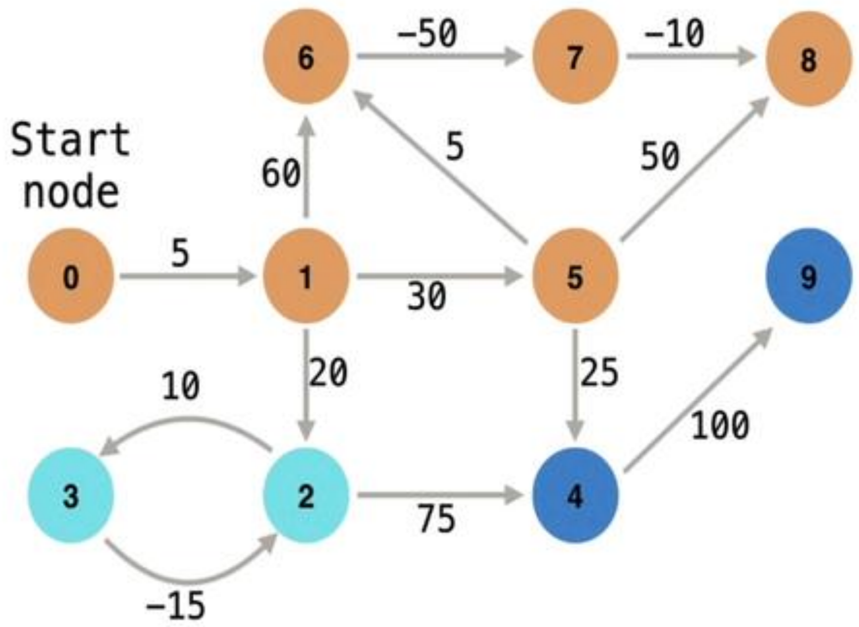
Negative Cycles



0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞



Negative Cycles



0	0
1	5
2	-∞
3	-∞
4	-∞
5	35
6	40
7	-10
8	-20
9	-∞

Repeat this for another 8 iterations in order to ensure the cycles fully propagate. In this example, we happened to detect all cycles on the first iteration, but this was a coincidence.

Dijkstra algorithm
Bellman-Ford Algorithm
Floyd-Warshall algorithm
…

The basic DFS and BFS algorithm is the building block of some more complicated algorithms.

References

https://github.com/williamfiset/Algorithms/tree/master/src/main/java/com/williamfiset/algorithms/graphtheory