setup

In[3]:=
```
options = Sequence[VertexSize → Large, VertexLabels → Placed[Automatic, {1 / 2, 1 / 2}], ImageSize → Large]
```

Out[3]= $\text{Sequence}\left[\text{VertexSize} \to \text{Large}, \text{VertexLabels} \to \text{Placed}\left[\text{Automatic}, \left\{\frac{1}{2}, \frac{1}{2}\right\}\right], \text{ImageSize} \to \text{Large}\right]$

Tang Anke

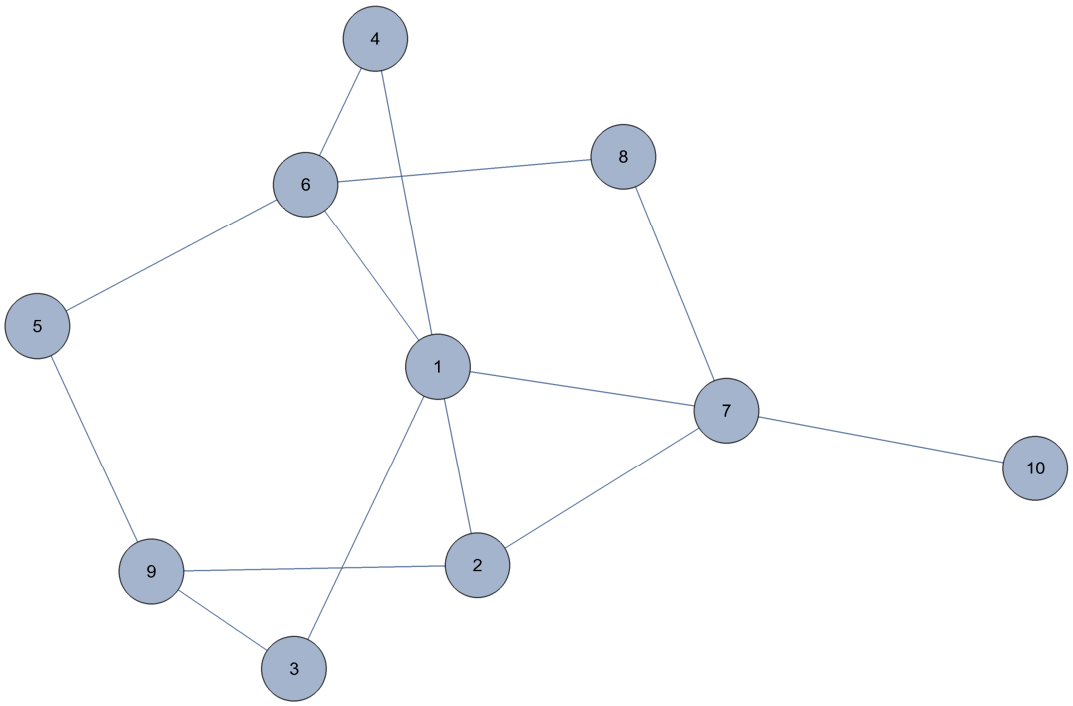
depth first search (DFS)

---

basic DFS algorithm

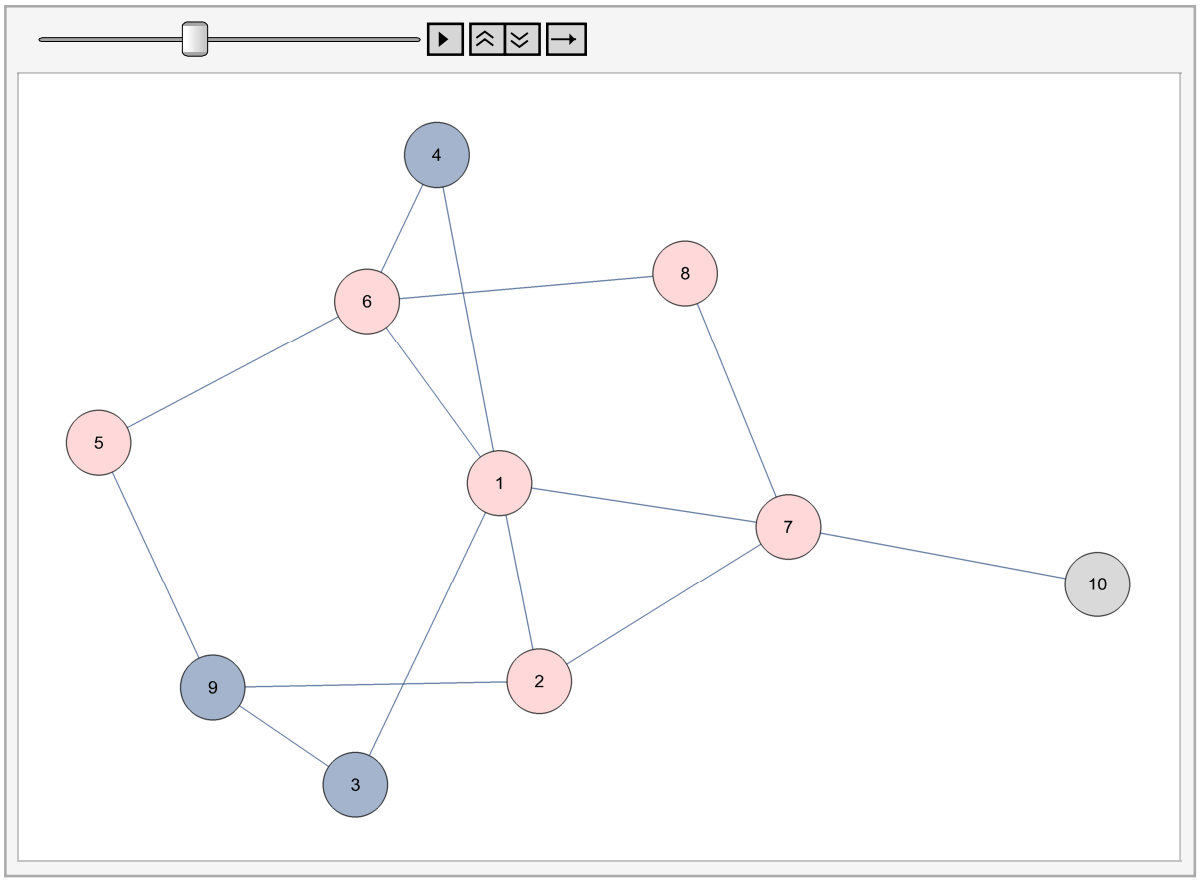
In[ ]:=
```
g = RandomGraph[{10, 14}, options]
```

Out[ ]=



In[ ]:=
```
ListAnimate[
 Module[{visit = Function[# -> LightRed], visited = Function[# → LightGray]},
  GraphPlot[  , VertexStyle → #, options] & /@ FoldList[Append, {}, Join[visit /@ {5, 6, 8, 7, 10},
    visited /@ {10}, visit /@ {2, 1, 4}, visited /@ {4}, visit /@ {3, 9}, visited /@ {9, 3, 1, 2, 7, 8, 6, 5}]]
 ]
]
```

Out[ ]=



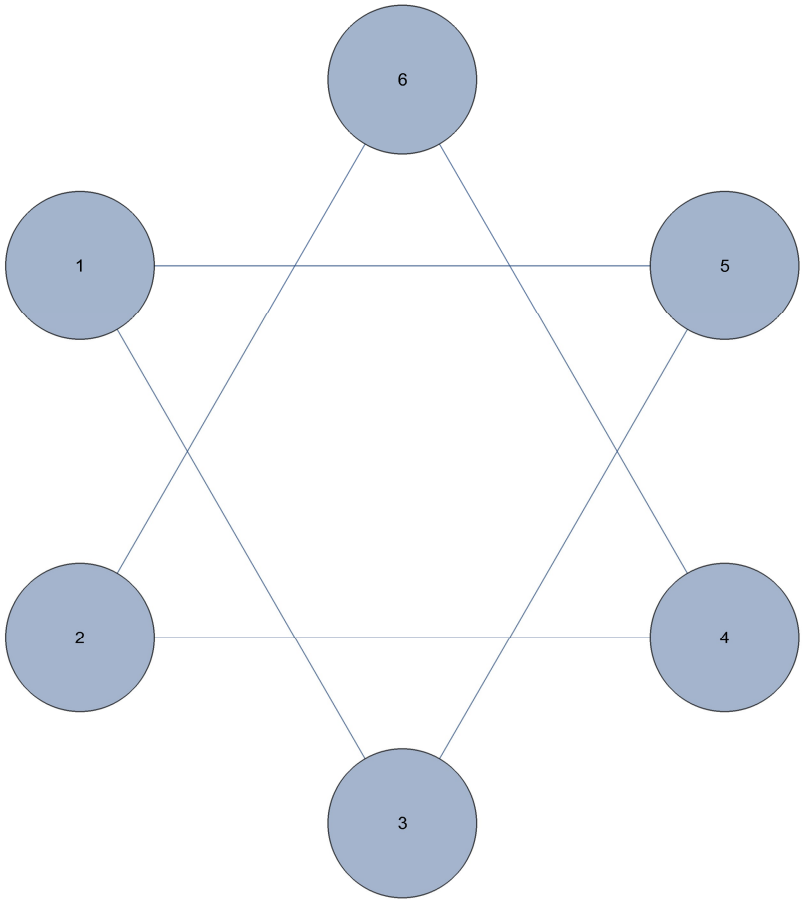

```python
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
visited = [false, …, false] # size n

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)
```

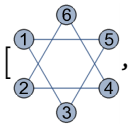
applications

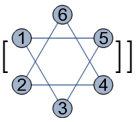
connected components

In[ ]:=
```
Graph[ , options, GraphLayout → {"CircularEmbedding", "OptimalOrder" → False}]
```

Out[ ]=

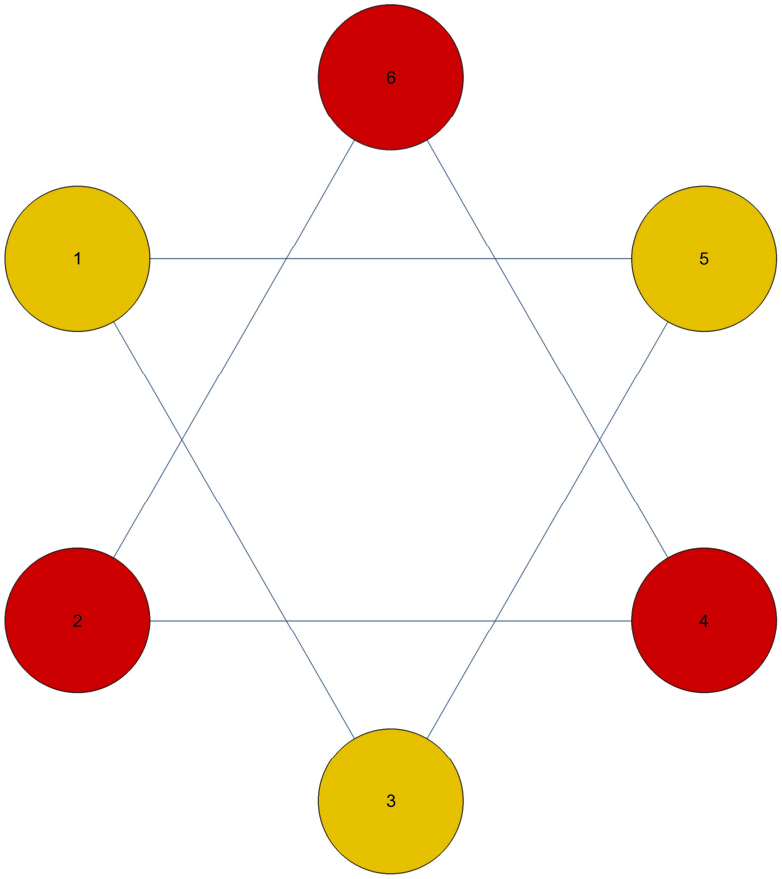


In[ ]:= HighlightGraph[

, ConnectedComponents[

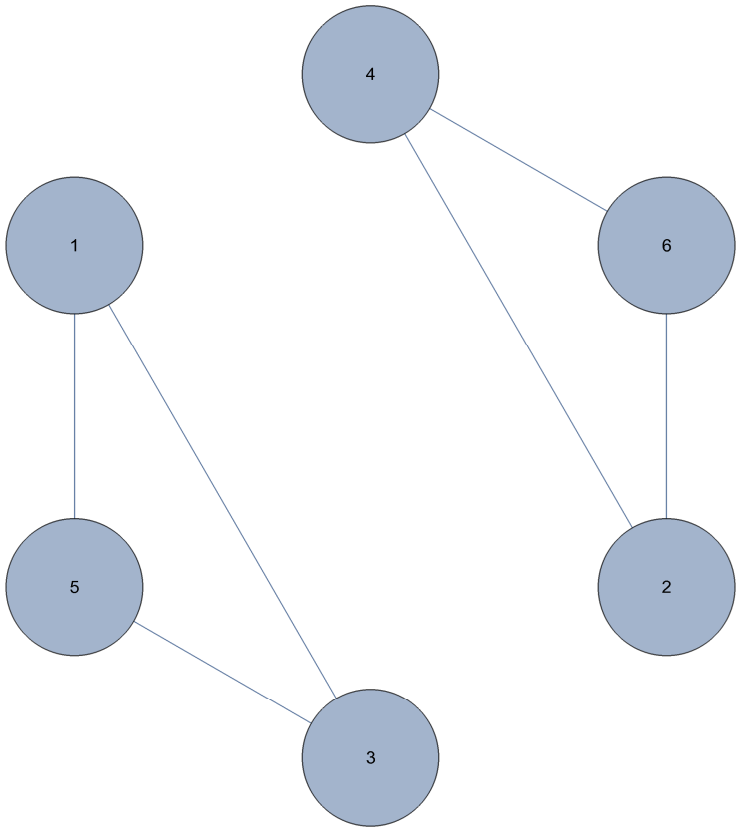
]]

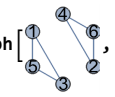
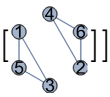
Out[ ]=



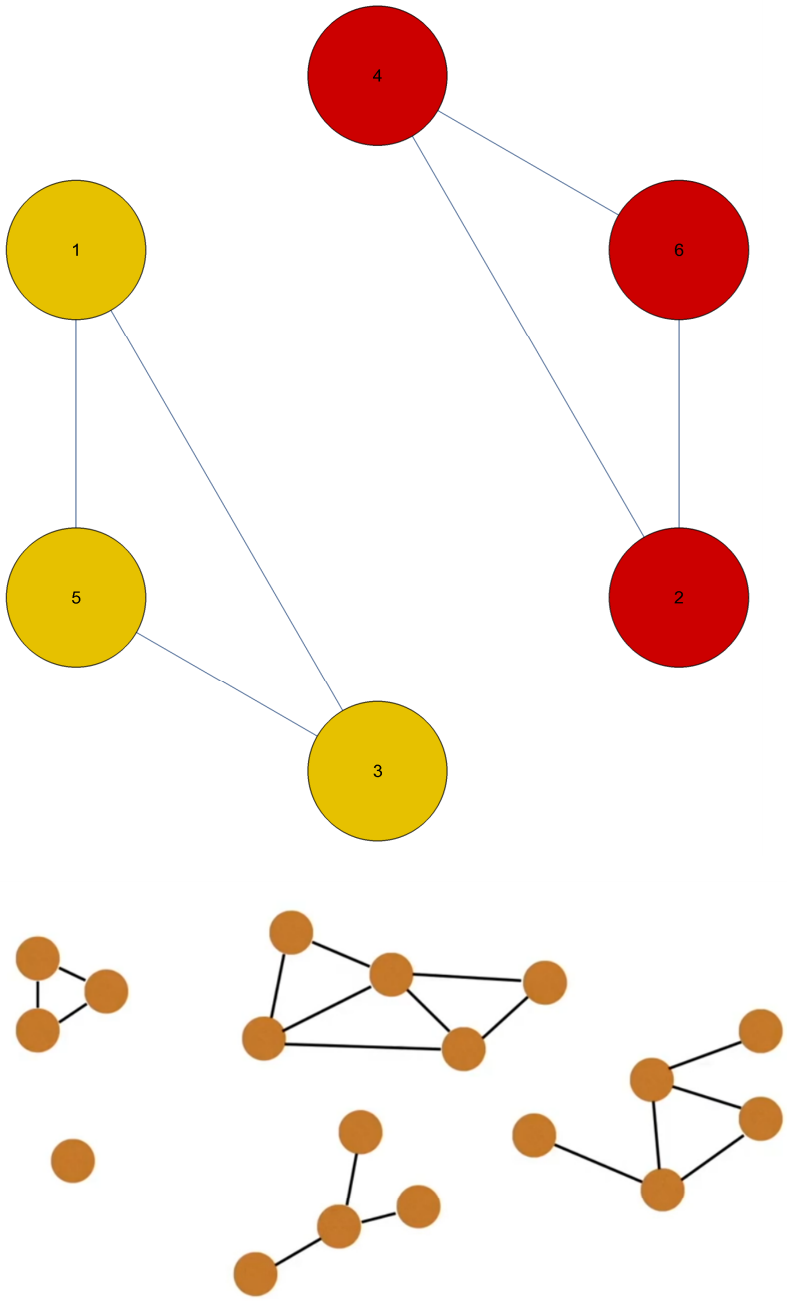
In[•]:= `Graph[ , options, GraphLayout → {"CircularEmbedding", "OptimalOrder" → True}]`

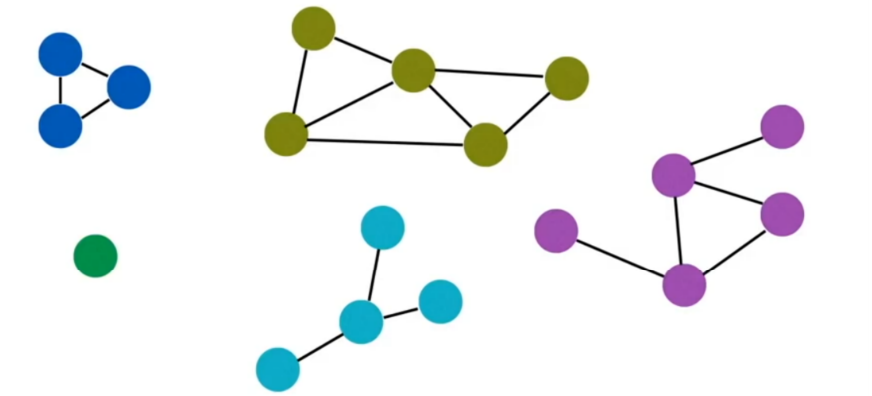
Out[•]=



In[ ]:= HighlightGraph[ , ConnectedComponents[]]

Out[ ]=





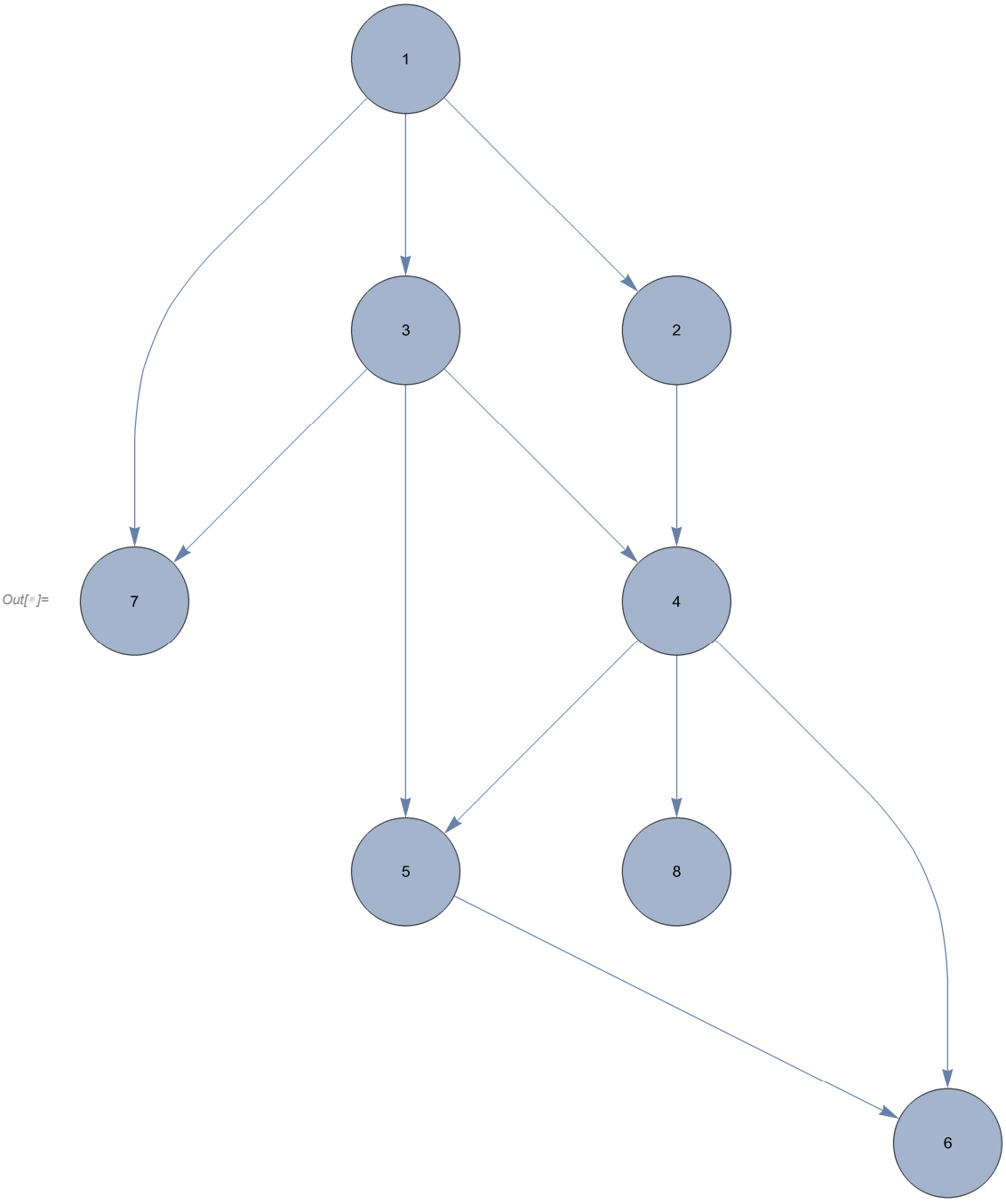
```
# Global or class scope variables
n = number of nodes in the graph
g = adjacency list representing graph
count = 0
components = empty integer array # size n
visited = [false, …, false] # size n
```

```
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)
```

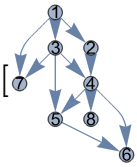
topological sort

In[ ]:=
```
Graph[Range[8], {1 → 2, 1 → 3, 2 → 4, 3 → 4, 3 → 5, 4 → 5, 1 → 7, 3 → 7, 4 → 8, 4 → 6, 5 → 6},
 options, GraphLayout → "LayeredDigraphEmbedding"]
```

Out[ ]=

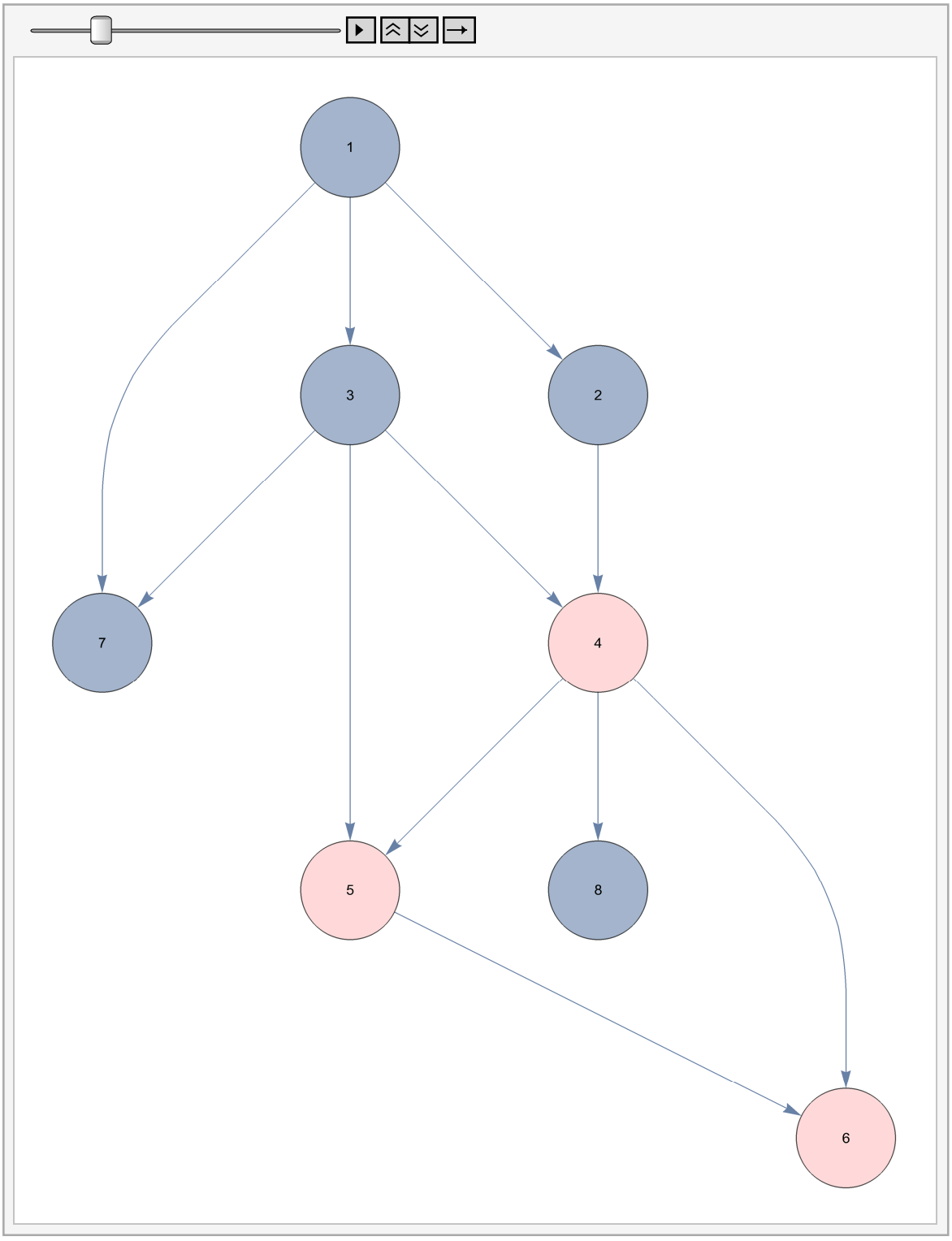


```
In[ ]:= ListAnimate[
 Module[{visit = Function[# -> LightRed], visited = Function[# → LightGray]},
  GraphPlot[
```




```
   , VertexStyle → #, options] & /@
   FoldList[Append, {}, Join[visit /@ {4, 5, 6}, visited /@ {6, 5}, visit /@ {8},
     visited /@ {8, 4}, visit /@ {3, 7}, visited /@ {7, 3}, visit /@ {1, 2}, visited /@ {2, 1}]]
 ]
]
```

Out[ ]=

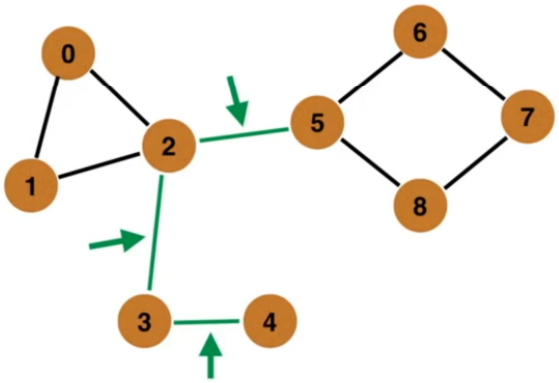


{6,5,8,4,7,3,2,1}

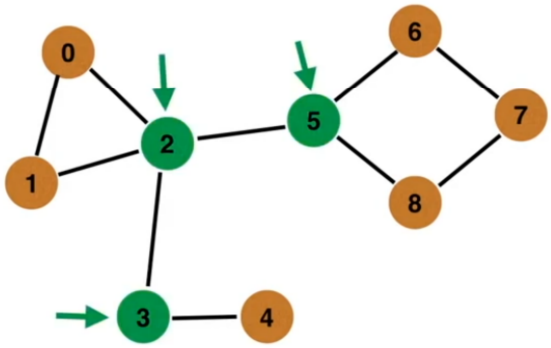
```
# Assumption: graph is stored as adjacency list
function topsort(graph):

  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

# 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)
```

find bridges and articulation points

A **bridge / cut edge** is any edge in a graph whose removal increases the number of connected components.

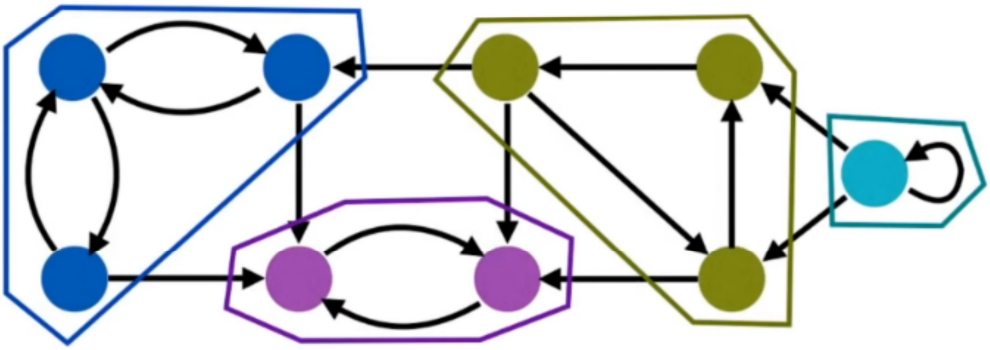


An **articulation point / cut vertex** is any node in a graph whose removal increases the number of connected components.



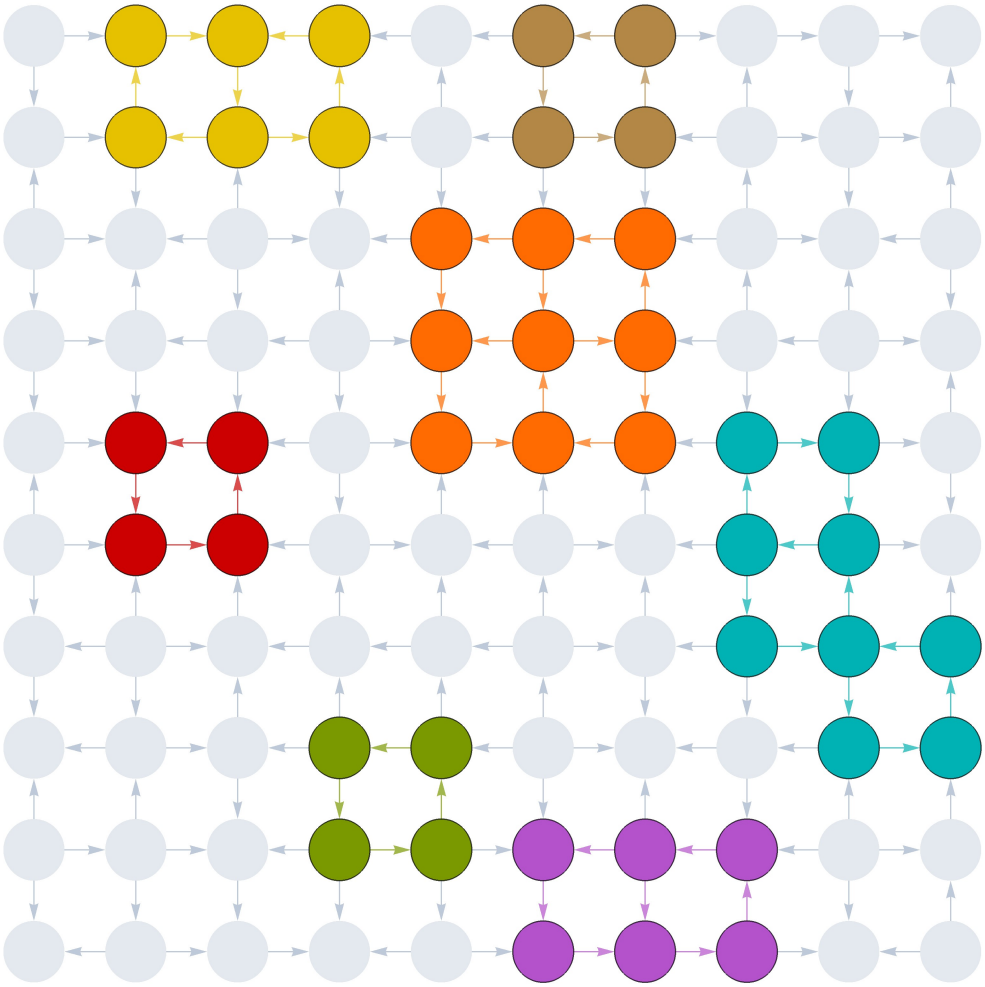
Tarjan's algorithm for finding strongly connected components

Strongly Connected Components (SCCs) can be thought of as **self-contained cycles** within a **directed graph** where every vertex in a given cycle can reach every other vertex in the same cycle.



In[ ]:=
```
g = DirectedGraph[GridGraph[{10, 10}], "Random", VertexSize → 0.6, GraphHighlightStyle → "DehighlightFade"];
HighlightGraph[g, Select[ConnectedGraphComponents[g], VertexCount[#] > 1 &]]
```

Out[ ]=



find Eulerian path (directed graph)

FindSpanningTree

...

What else can DFS do?

We can augment the DFS algorithm to:

- Compute a graph's minimum spanning tree.
- Detect and find cycles in a graph.
- Check if a graph is bipartite.
- Find strongly connected components.
- Topologically sort the nodes of a graph.
- Find bridges and articulation points.
- Find augmenting paths in a flow network.
- Generate mazes.

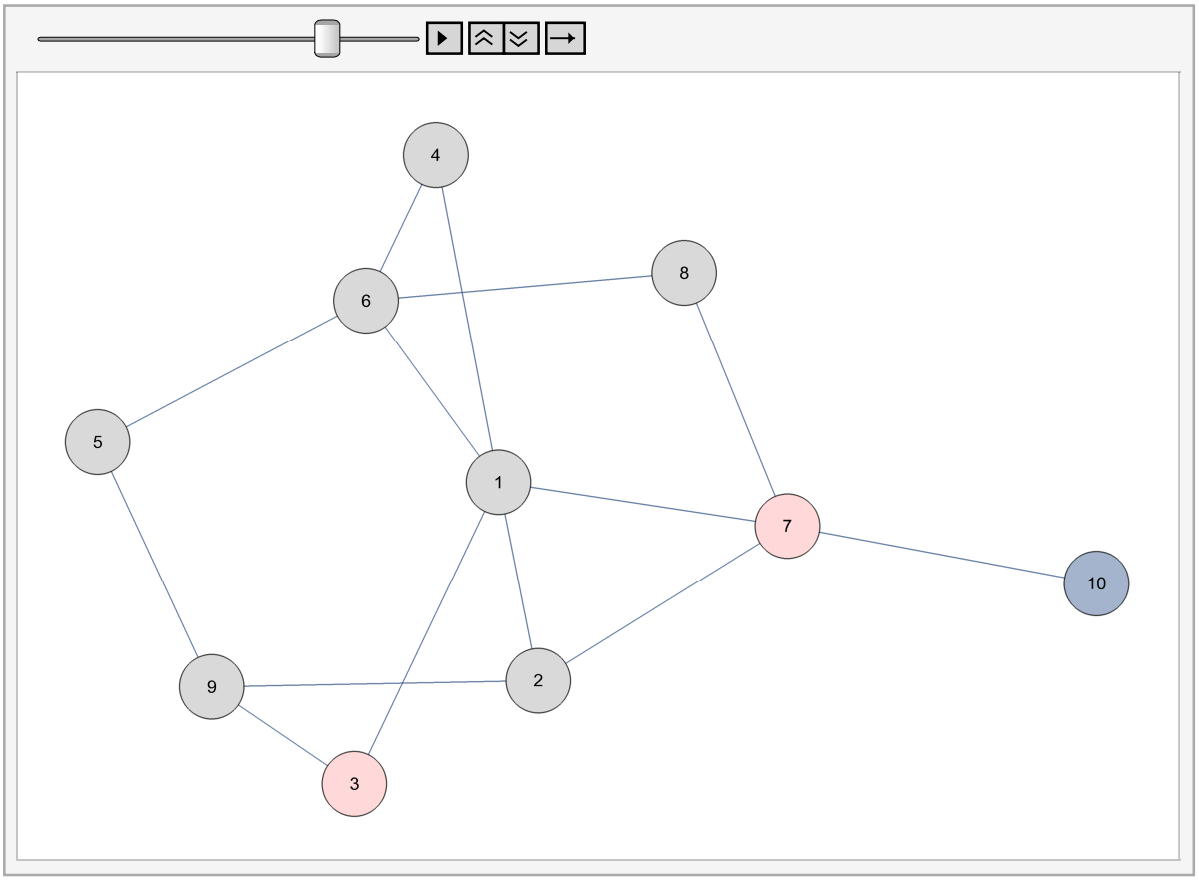
breadth first search (BFS)

---

algorithm

```
In[ ]:= ListAnimate[
          Module[{visit = Function[# -> LightRed], visited = Function[# → LightGray]},
           GraphPlot[  , VertexStyle → #, options] & /@
            FoldList[Append, {}, Join[visit /@ {5, 6, 9}, visited /@ {5}, visit /@ {4, 8, 1}, visited /@ {6}, visit /@ {2, 3},
              visited /@ {9, 4}, visit /@ {7}, visited /@ {8, 1, 2, 3}, visit /@ {10}, visited /@ {7, 10}]]
          ]
         ]
```

Out[•]=




```
# 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)
```

```
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

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 []
```

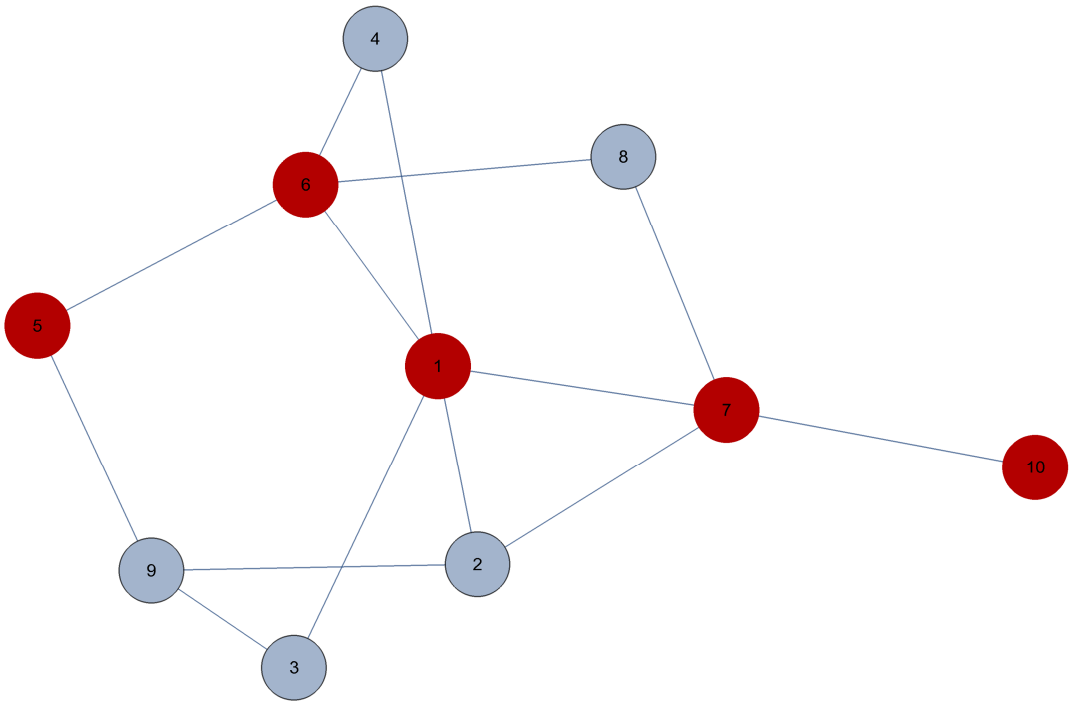
---

applications

shortest path problem (unweighted graph)

O(V+E)

Out[•]=

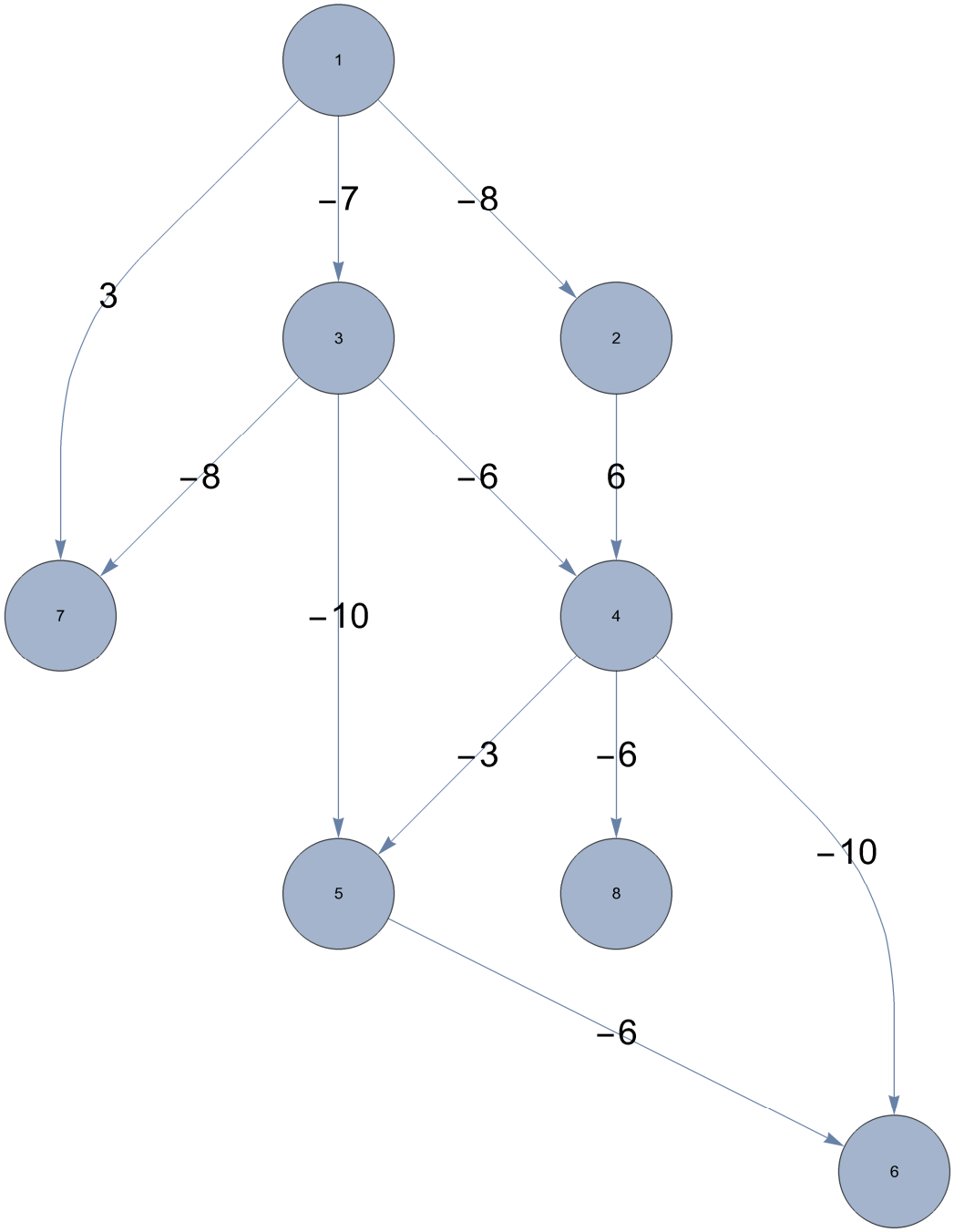


single source shortest path (SSSP) on directed acyclic graph (DAG)

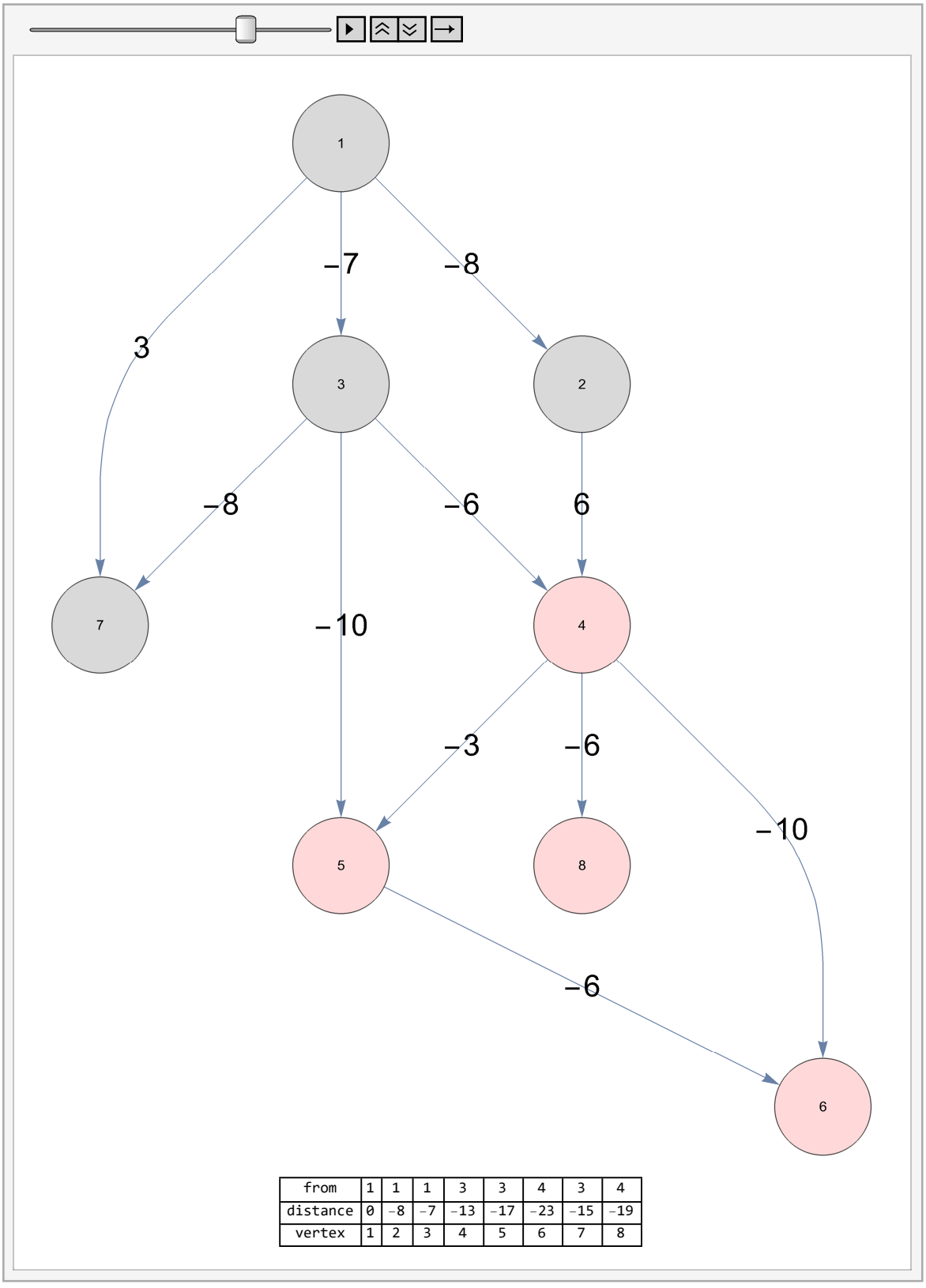
```
In[ ]:= Module[{weights = RandomInteger[{-10, 10}, 11],
    edges = {1 ↔ 2, 1 ↔ 3, 2 ↔ 4, 3 ↔ 4, 3 ↔ 5, 4 ↔ 5, 1 ↔ 7, 3 ↔ 7, 4 ↔ 8, 4 ↔ 6, 5 ↔ 6}},
   Echo[weights];
   Graph[Range[7], edges, EdgeWeight → weights, options,
    EdgeLabels → Thread[Rule[edges, weights]], EdgeLabelStyle → Directive[20]]
  ]
```

```
» {-8, -7, 6, -6, -10, -3, 3, -8, -6, -10, -6}
```

Out[ ]=



Out[ ]=



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

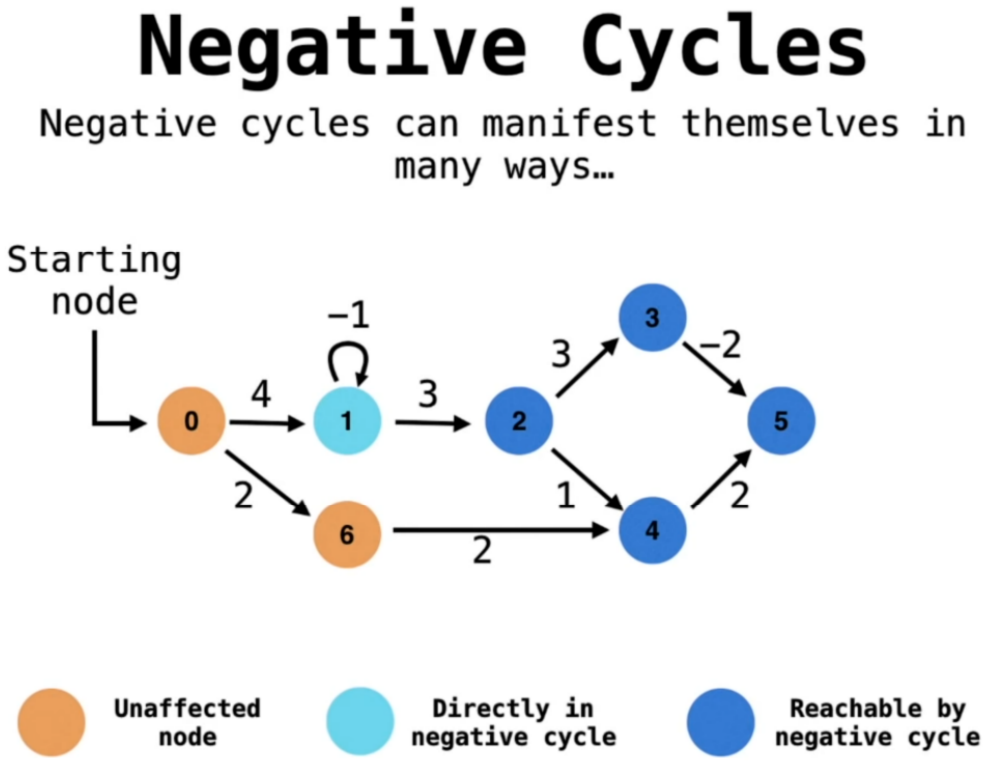
longest path on DAG

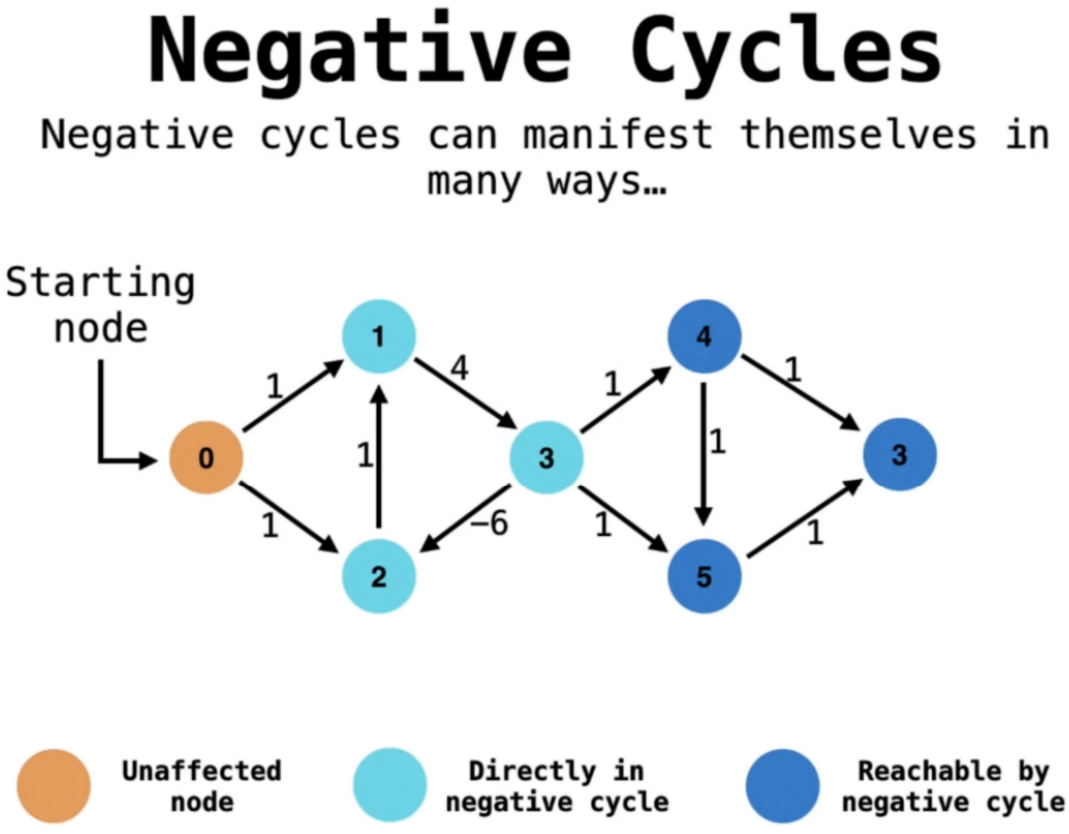
Dijkstra algorithm

O((V+E)logV)

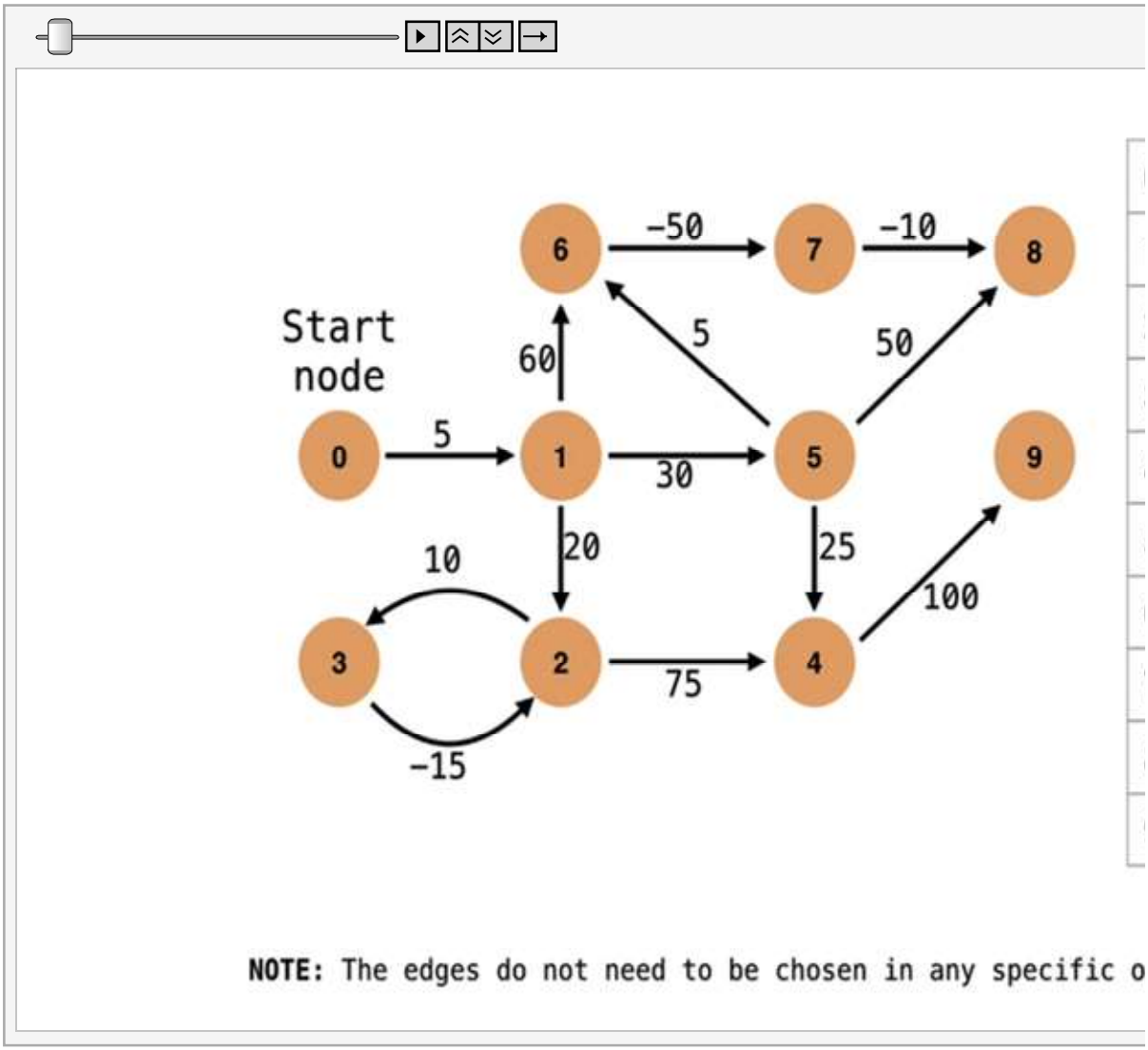
Bellman - Ford Algorithm

O(EV)





Out[ ]=



Floyd-Warshall algorithm (all pairs shortest path (APSP))

$O(V^3)$

...