Graph Algorithms

DSA, Fall 2022

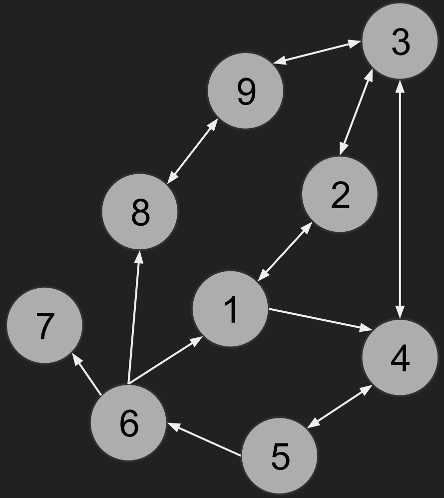
Graphs

A graph is a set of things (vertices) connected together (edges)

- if connections have a direction — directed graph
- if connections do not have a direction — undirected graph

Can be used to represent

- networks of computers with communication links
- networks of friends with "friend of" or "follows" connectivity
- cities and flights between them
- etc…



Definitions

A directed graph is a pair (V, E) consisting of:

- A finite set V of vertices
- A finite set E of edges of the form (u, v) connecting vertex u to vertex v

Edge (u, v) — u is the source, v is the target — written $u \to v$

For undirected graphs, take edges to be pairs {u, v}

v is reachable from u if there exists vertices $v_1, \ldots, v_k$ such that
$u \to v_1,\ v_1 \to v_2,\ \ldots,\ v_{k-1} \to v_k,$ and $v_k \to v$

Graph Representations

Need to represent both vertices and edges

- For simplicity, assume vertices are 0, 1, …, n

Most algorithms involve querying the graph more than updating the graph

Adjacency list

attach to each vertex a list of connected vertices

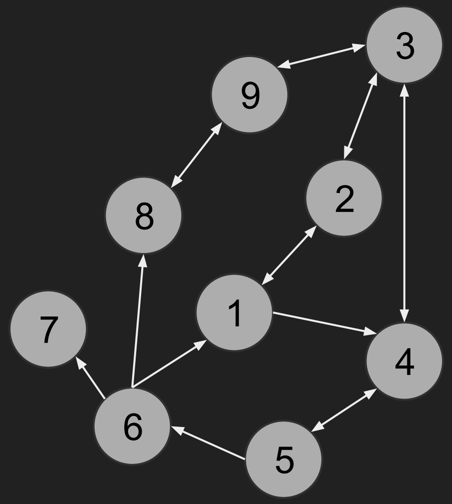
Adjacency matrix

matrix indexed by vertices with 1 at (i, j) when there's an edge from i to j

Adjacency List Representation

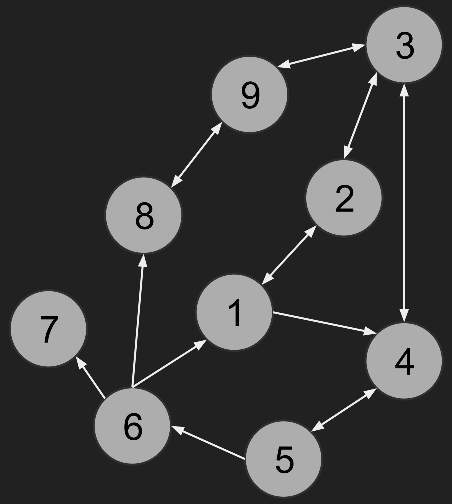
```
type Graph struct {
    vertices int
    edges []*Edge
}

type Edge struct {
    target int
    next *Edge
}
```



Adjacency Matrix Representation

```
type Graph struct {
    vertices int
    edges [][]int
}
```



Choice of Representation

The sparser a graph (small number of edges vs vertices), the more benefit from the adjacency list representation

- maximum number of edges in a (directed) graph is $|V|^2$
- a graph is sparse if $|E| << |V|^2$
- most practical graphs are sparse

Space for adjacency list representation $O(|V| + |E|)$
Space for adjacency matrix representation $O(|V|^2)$

Both pretty straightforwardly generalize to weighted graphs representations

Core Algorithms — Search

Given a starting vertex u, search for all reachable vertices from u

- construct a path from u to each reachable vertex
- it's basically a tree rooted at u with reachables vertices as nodes

Two basic approaches

- systematically look at all vertices distance 1, 2, 3, ... from starting vertex
- blaze through from starting vertex, backtracking when you get stuck

Search Skeleton

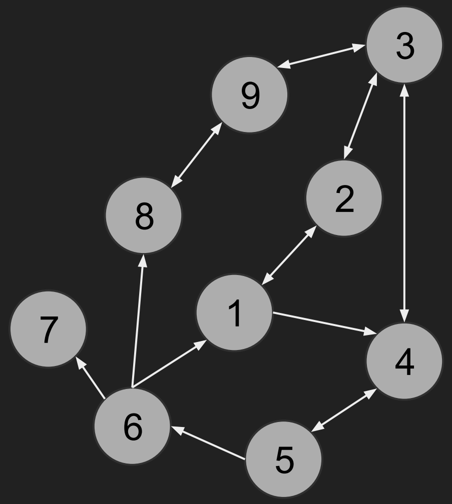
```
color every vertex gray
color[start] ← red
while there is a red vertex:
    pick a red vertex v
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
```

How to pick next red vertex to look at?

Breadth-First Search

Add red vertices into a QUEUE

```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```

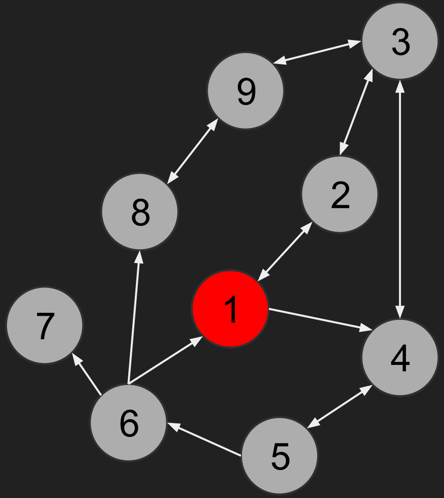


Breadth-First Search

Add red vertices into a QUEUE

QUEUE: 


```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```

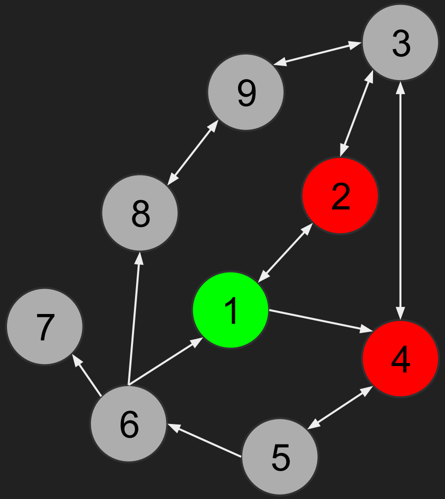


Breadth-First Search

Add red vertices into a QUEUE

QUEUE: 4 2

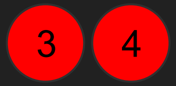
```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```



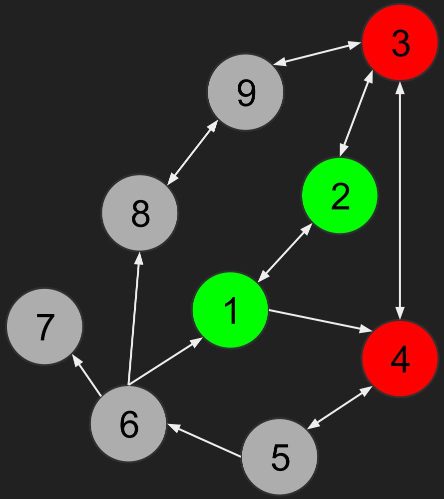
Breadth-First Search

Add red vertices into a QUEUE

QUEUE:




```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```

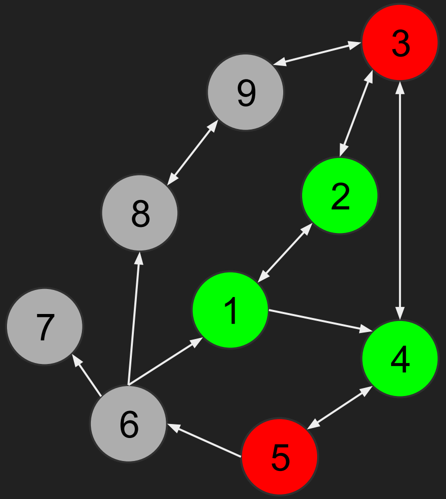


Breadth-First Search

Add red vertices into a QUEUE

QUEUE: 5 3

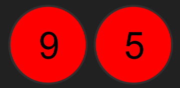
```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```



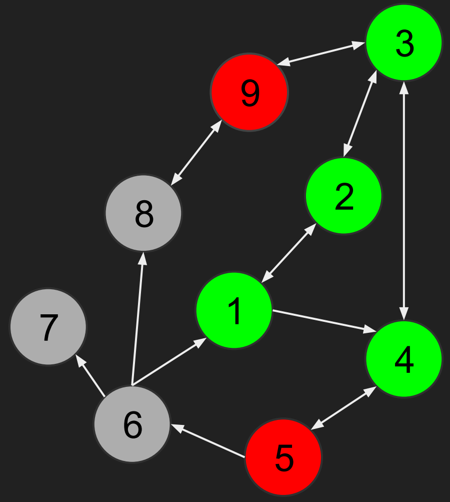
Breadth-First Search

Add red vertices into a QUEUE

QUEUE:



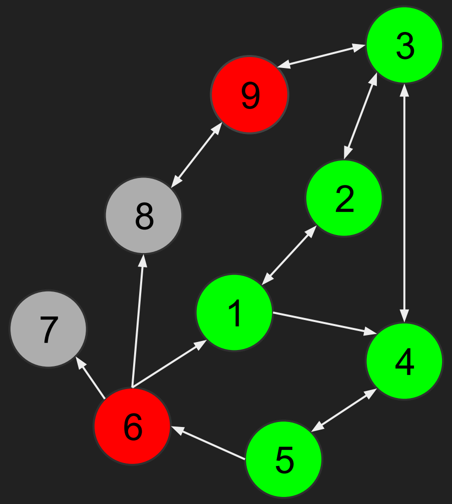

```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```



Breadth-First Search

Add red vertices into a QUEUE


```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```

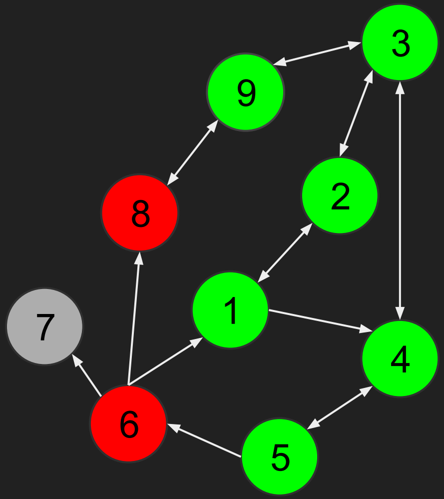


Breadth-First Search

Add red vertices into a QUEUE

QUEUE: 8 6

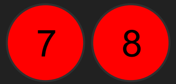
```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```



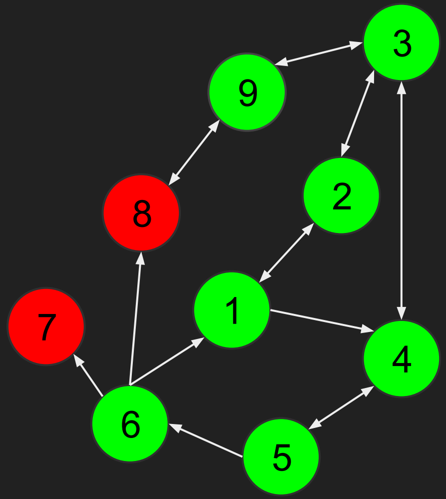
Breadth-First Search

Add red vertices into a QUEUE

QUEUE:



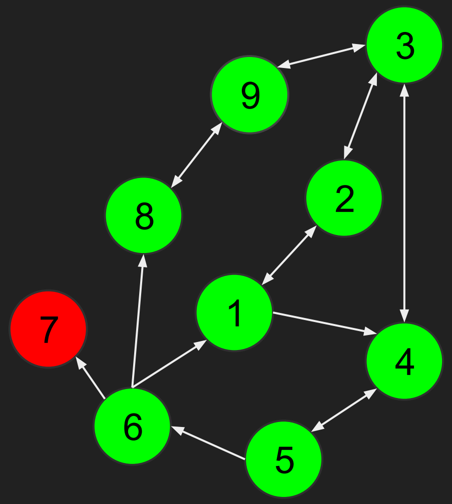

```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```



Breadth-First Search

Add red vertices into a QUEUE


```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```

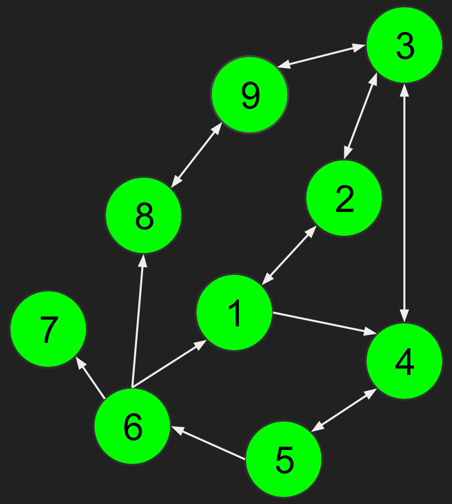


Breadth-First Search

Add red vertices into a QUEUE

QUEUE:

```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```



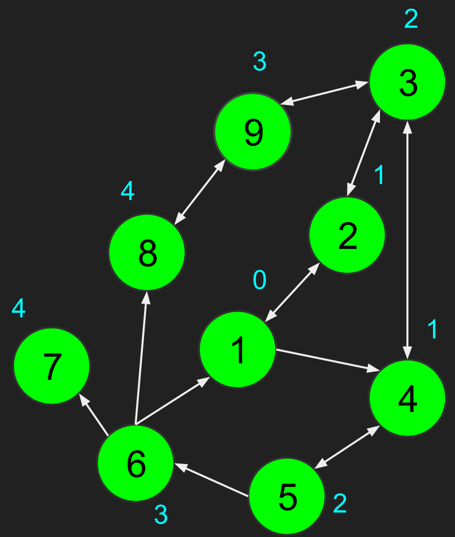
Application — Shortest Paths

A feature of BFS is that it searches vertices in order of "closeness" from the start vertex

- "closeness" defined in terms of number of edges to follow to reach vertex

Application — Shortest Paths

```
for every vertex u: color[u] ← gray
for every vertex u: dist[u] ← ∞
color[start] ← red
ENQUEUE(Q, start)
dist[start] ← 0
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            dist[u] ← dist[v] + 1
            ENQUEUE(Q, u)
```



Application — Shortest Paths

A feature of BFS is that it searches vertices in order of "closeness" from the start vertex

- "closeness" defined in terms of number of edges to follow to reach vertex

Can also construct the shortest path from start vertex for every reachable vertex

- shortest path is not unique

Application — Shortest Paths

```
for every vertex u: color[u] ← gray
for every vertex u: dist[u] ← ∞   parent[u] ← nil
color[start] ← red
ENQUEUE(Q, start)
dist[start] ← 0
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            dist[u] ← dist[v] + 1    parent[u] ← v
            ENQUEUE(Q, u)
```

When dist[u] < ∞ following parent[u] up to start yields a path from start to u

Depth-First Search

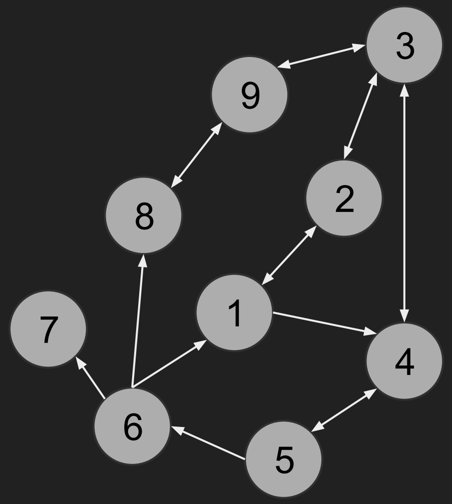
Add red vertices into a STACK

```
color every vertex gray
color[start] ← red
ENQUEUE(Q, start)
while not ISEMPTY(Q):
    v ← DEQUEUE(Q)
    color[v] ← green
    for every adjacent vertex u of v:
        if color[u] = gray:
            color[u] ← red
            ENQUEUE(Q, u)
```

Depth-First Search

Add red vertices into a STACK

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

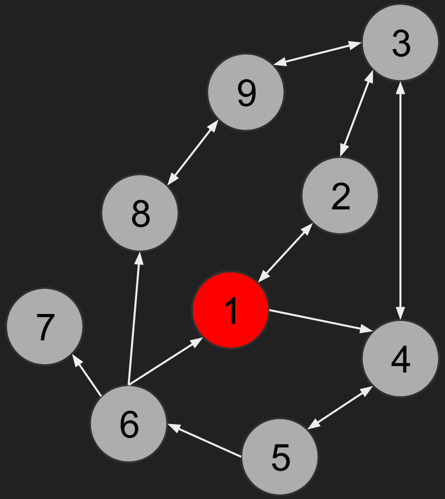


Depth-First Search

Add red vertices into a STACK

STACK: 1

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

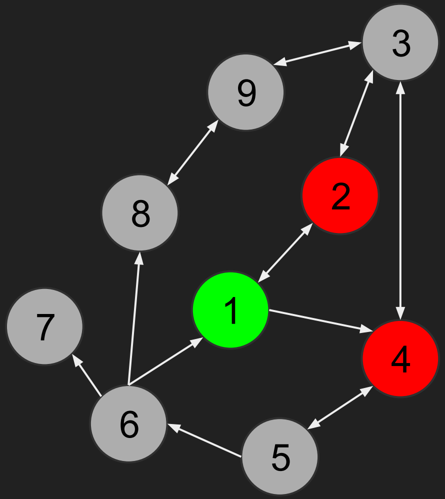


Depth-First Search

Add red vertices into a STACK

STACK: 


```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

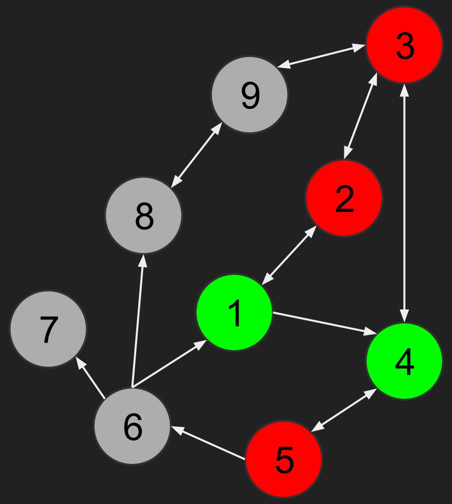


Depth-First Search

Add red vertices into a STACK

STACK:


```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

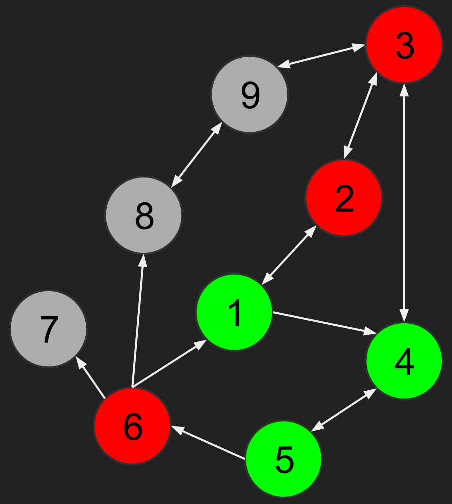


Depth-First Search

Add red vertices into a STACK

STACK:

color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)

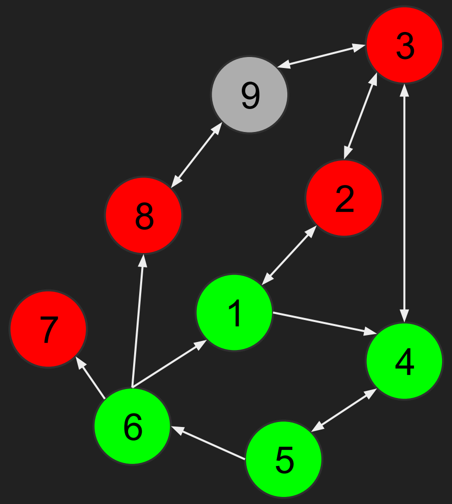


Depth-First Search

Add red vertices into a STACK

STACK: 8 7 3 2

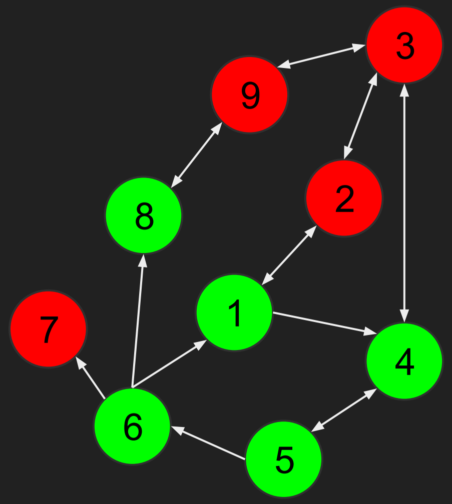
```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```



Depth-First Search

Add red vertices into a STACK

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

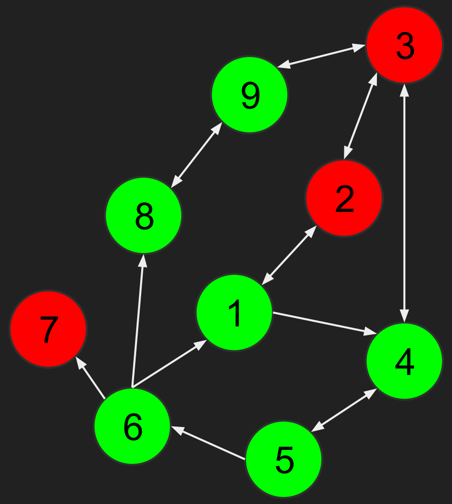


Depth-First Search

Add red vertices into a STACK

STACK: 3 7 3 2

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

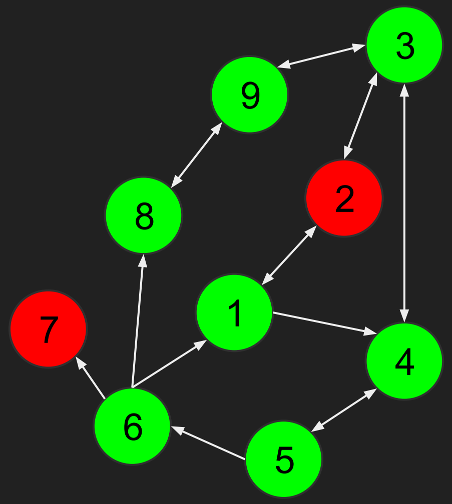


Depth-First Search

Add red vertices into a STACK

STACK: 2 7 3 2

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

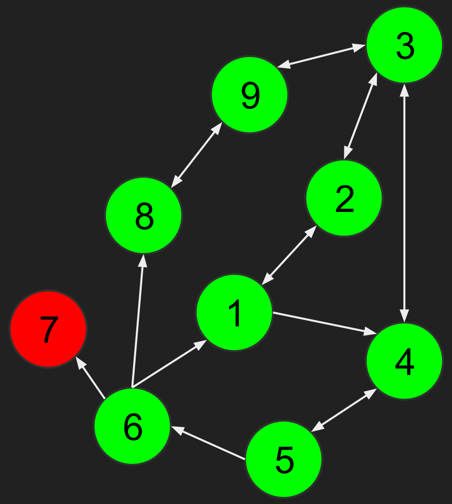


Depth-First Search

Add red vertices into a STACK

STACK: 7 3 2

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

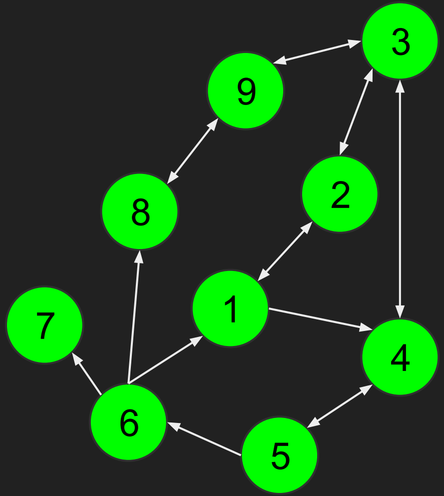


Depth-First Search

Add red vertices into a STACK

STACK: 3 2

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

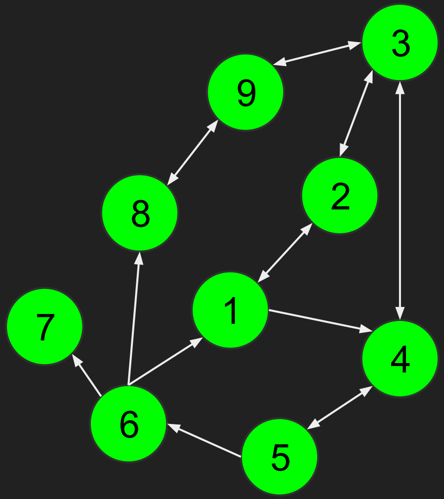


Depth-First Search

Add red vertices into a STACK

STACK: 2

```
color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)
```

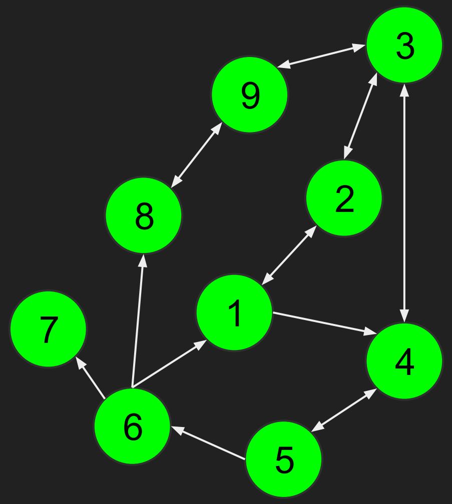


Depth-First Search

Add red vertices into a STACK

STACK:

color every vertex gray
color[start] ← red
PUSH(T, start)
while not ISEMPTY(T):
    v ← POP(T)
    if color[v] != green:
        color[v] ← green
        for every adjacent vertex u of v:
            if color[u] != green:
                color[u] ← red
                PUSH(T, u)



Application — Topological Sort

An acyclic directed graph is a directed graph with no cycles

- no path from a vertex to itself

Given an acyclic directed graph G, the topological sort of G is an ordering of the vertices of G such that when there's an edge u → v in G, then u comes before v in the ordering

Classic: vertices of G are tasks, u → v means u must be done before v, then a topological sort is a way to schedule tasks so that required tasks are done first

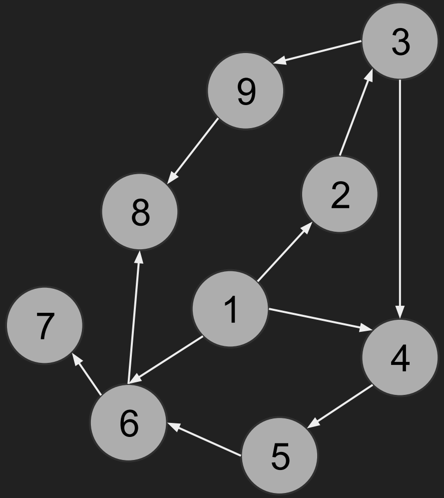
Practical: vertices are modules, u → v means v depends on u, a topological sort is an order of loading modules so you don't get errors

Application — Topological Sort

Starting point: recursive version of DFS
Blue vertices represent vertices being worked on

```
color every vertex gray
DFS(G, start)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
```

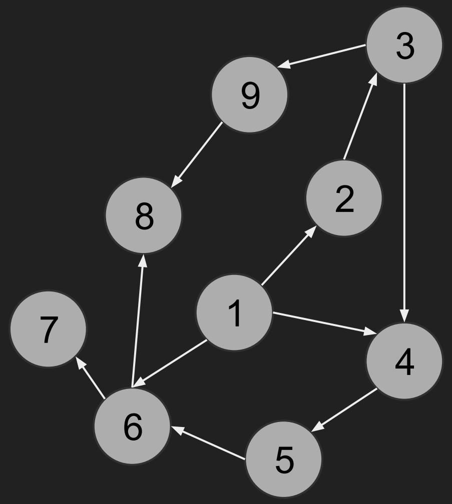


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

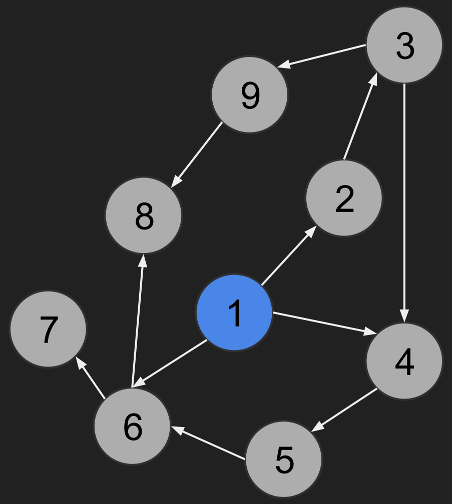


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

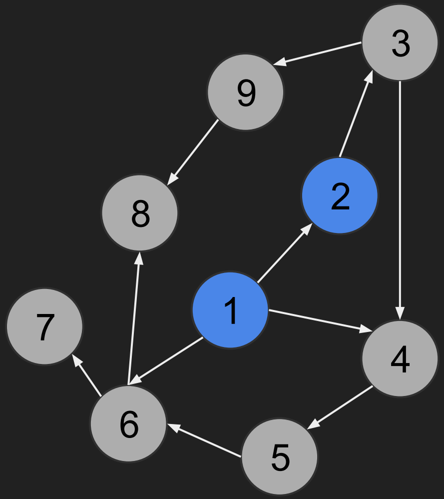


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

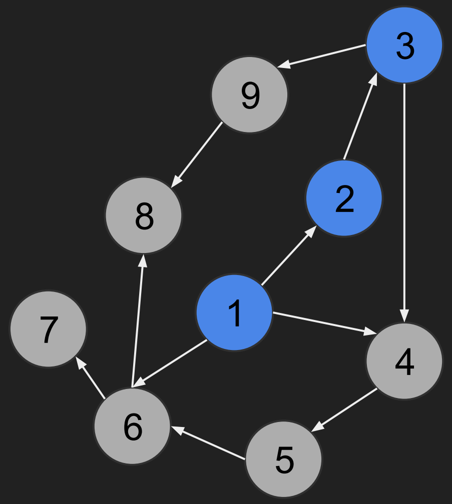


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

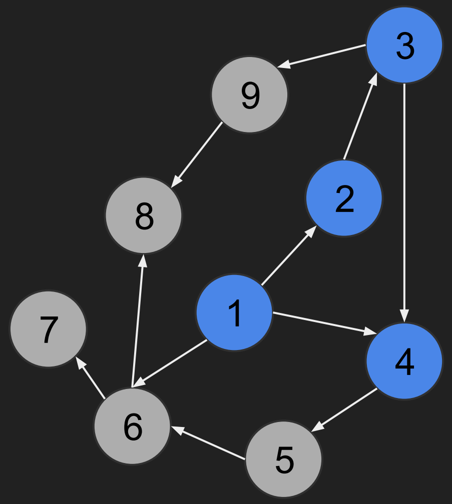


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

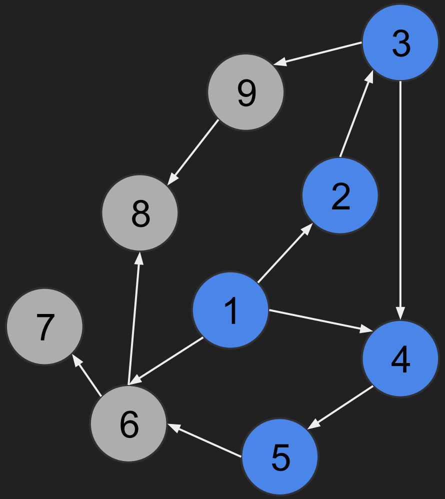


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

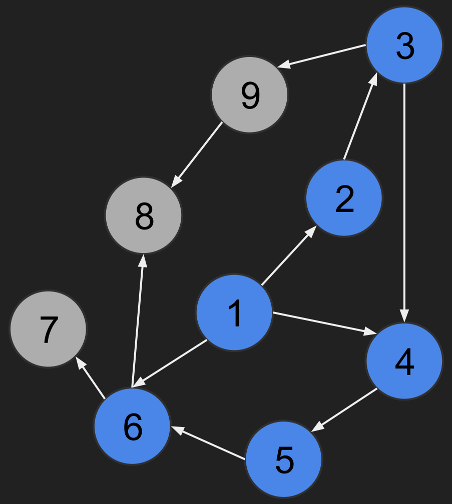


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

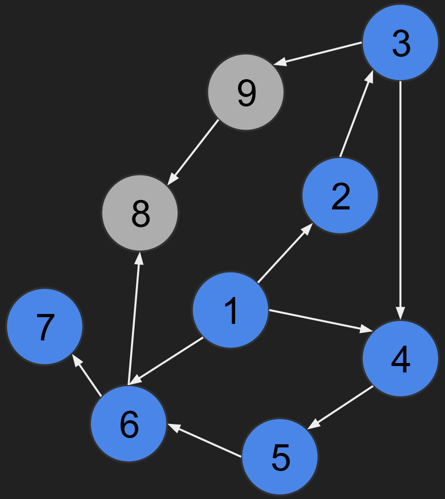


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT:

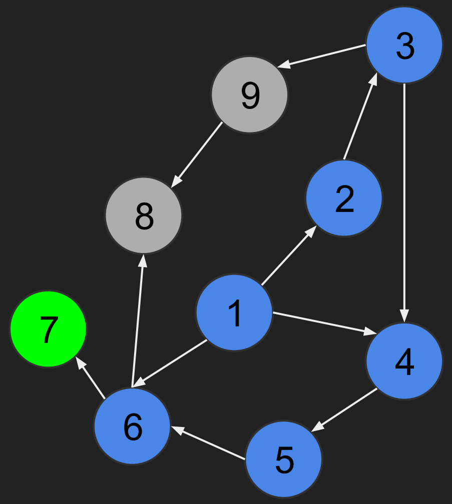


Application — Topological Sort

OUTPUT: 7

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

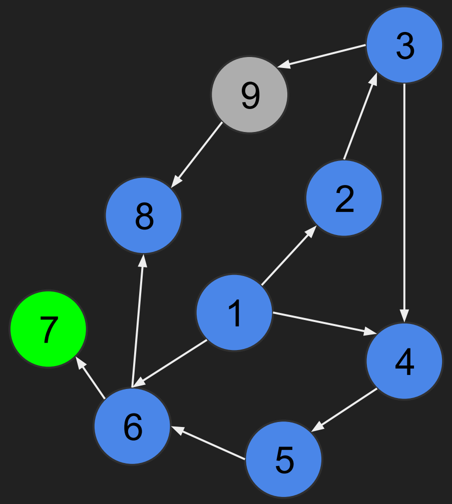


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7

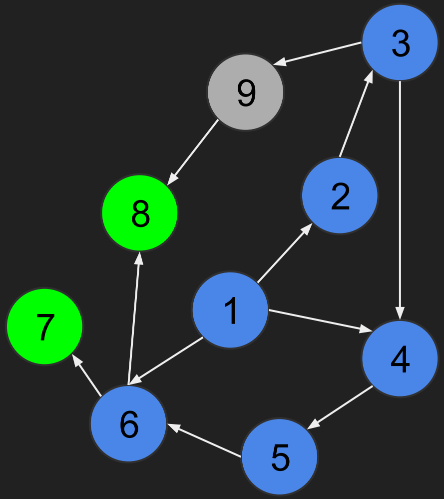


Application — Topological Sort

color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)

OUTPUT: 7 8

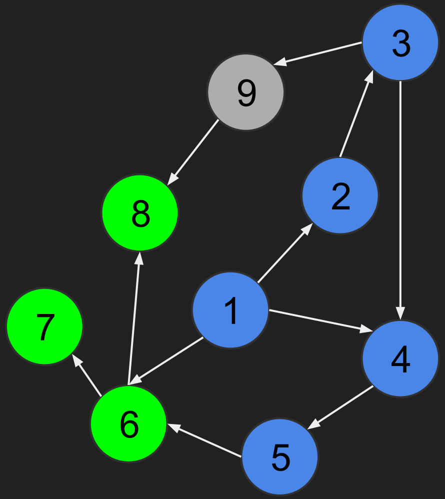


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7 8 6

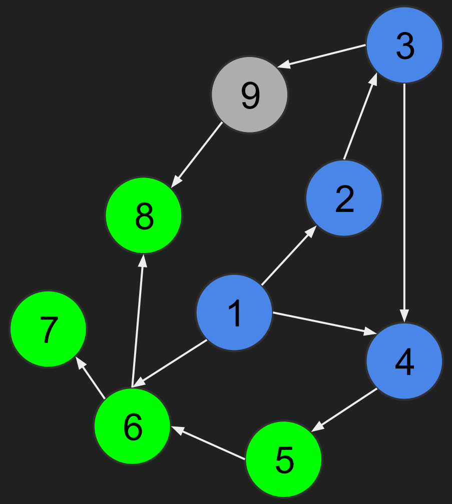


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7 8 6 5

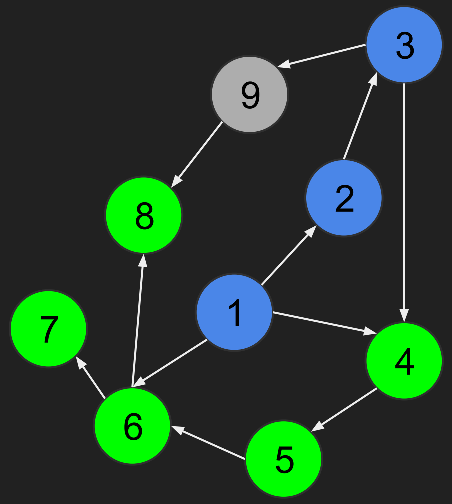


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7 8 6 5 4

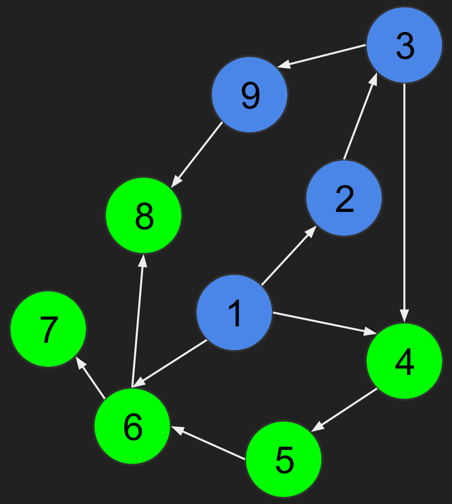


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7 8 6 5 4

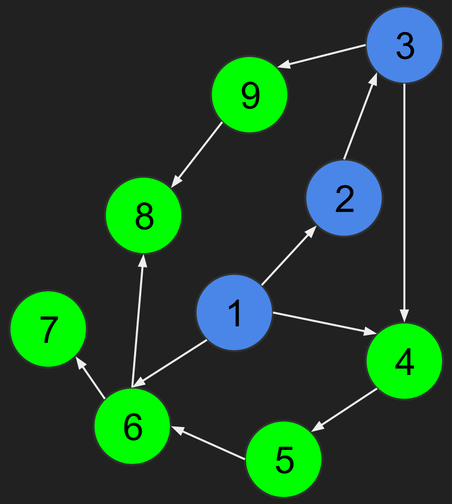


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7 8 6 5 4 9

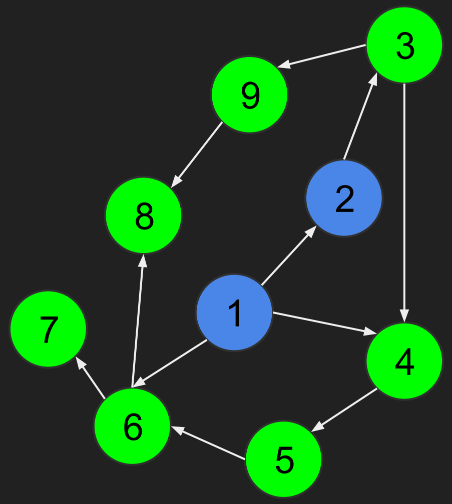


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7 8 6 5 4 9 3

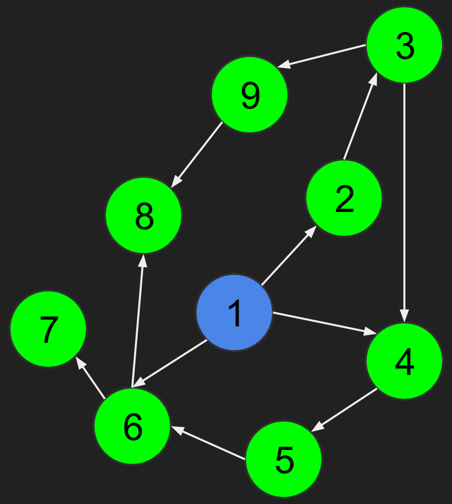


Application — Topological Sort

OUTPUT: 7 8 6 5 4 9 3 2

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

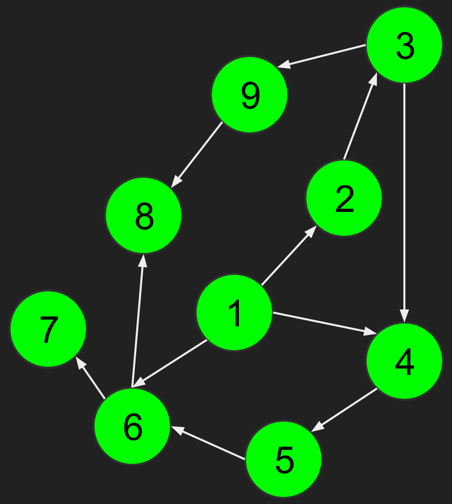


Application — Topological Sort

```
color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)
```

OUTPUT: 7 8 6 5 4 9 3 2 1



Application — Topological Sort

color every vertex gray
for every vertex u with no edge into it:
    DFS(G, u)

DFS(G, v) :=
    color[v] ← blue
    for every adjacent vertex u of v:
        if color[u] = blue:
            error "cycle in graph"
        if color[u] = gray:
            DFS(G, u)
    color[v] ← green
    output(v)

OUTPUT: 7 8 6 5 4 9 3 2 1

Reverse to get a topological sort

1 2 3 9 4 5 6 8 7