香港中文大學（深圳）
The Chinese University of Hong Kong

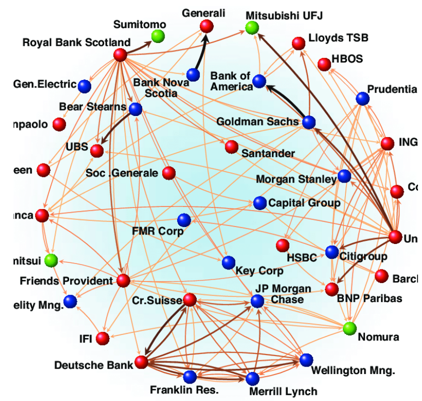
CSC3100 Data Structures
Lecture 19: Graphs, BFS, DFS

Yixiang Fang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

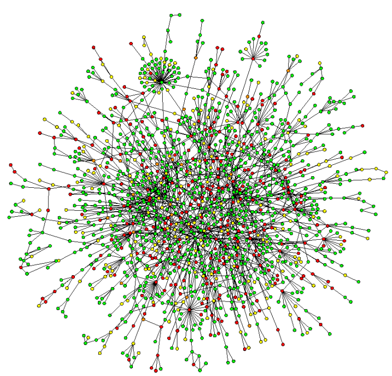
Examples of graphs

- Graph: a fundament data structure to represent objects and their relationships

Social networks



Financial networks



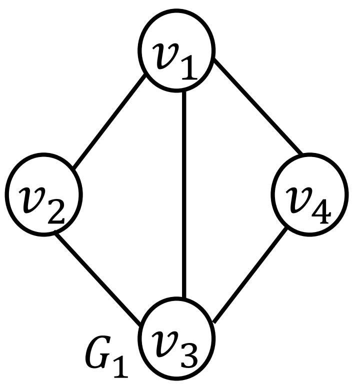
Protein interaction networks

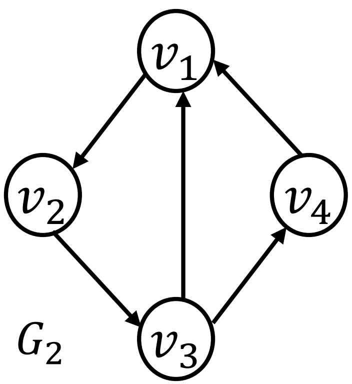
Road networks

Graph: definition

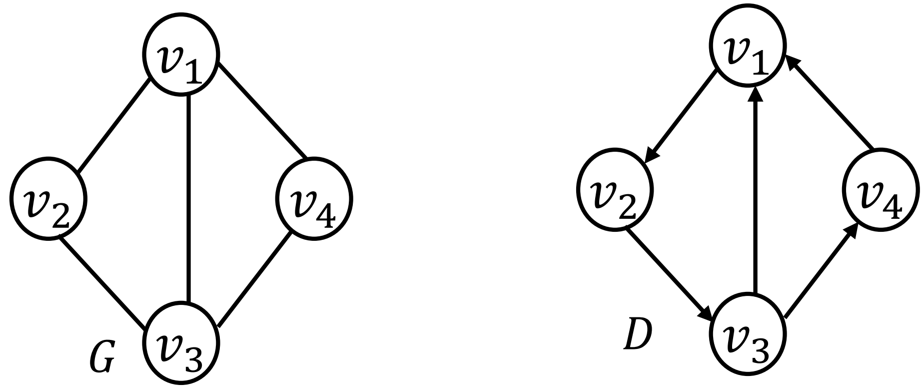
- A graph G is defined as a pair of (V, E) where:
  - V is the set of objects, each of which is called a node (or a vertex)
  - E is the set of edges, where each edge is a pair of two node u and v

- Notations
  - n: the number of nodes, i.e., $|V|$
  - m: the number of edges, i.e., $|E|$





Undirected/directed graphs



- A graph is an undirected graph, if there is no order in an edge
  - We use (u, v) to represent an edge in an undirected graph
    - (u, v) and (v, u) are the same edge
  - In many social networks, e.g., Facebook
- A graph is a directed graph, if there is an order in an edge
  - We use $\langle u, v \rangle$ to represent an edge in a directed graph
    - $\langle u, v \rangle$ and $\langle v, u \rangle$ represent two different edges
  - In some social networks, e.g., Twitter

Terminologies

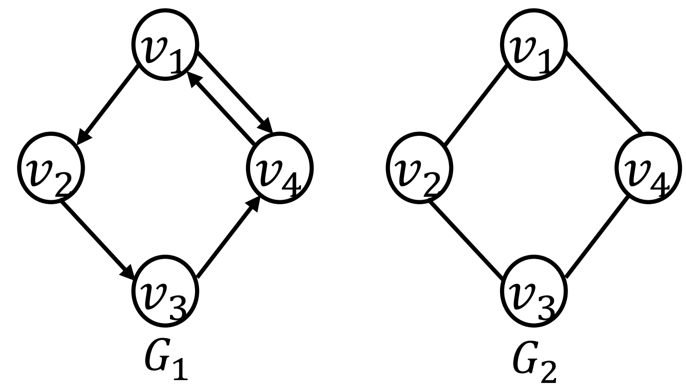
- Neighbor: v is called a neighbor of u is there is an edge between v and u
  - In directed graphs, v is called the out-neighbor of u is there is an edge $\langle u, v\rangle$; v is called the in-neighbor of u is there is an edge $\langle v, u\rangle$

- Degree: the degree $d(v)$ of a node v is the number of neighbors of this node v
  - In directed graphs, the out-degree $d_{out}(v)$ of a node v is the number of out-neighbors of this node; the in-degree $d_{in}(v)$ of a node v is the number of in-neighbors of this node

- A graph is connected if there is a path from every vertex to every other vertex
  - A tree is a connected, acyclic "undirected" graph

An example

- In G_1, the degree of v_1 is 3, the out-degree of v_1 is 2, and the in-degree of v_1 is 1

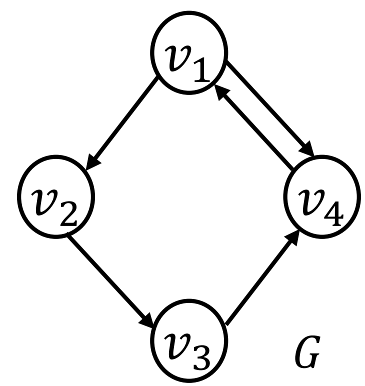


- In G_2, the degree of v_3 is 2
- The number of edges in G_1 is 5
  - The degrees of v_1, v_2, v_3 and v_4 are 3, 2, 2, and 3, respectively
  - In G_1, $\sum_{v \in V} d(v) = 3 + 2 + 2 + 3 = 10 = 2 \cdot 5 = 2 \cdot m$
  - In G_1, $\sum_{v \in V} d_{out}(v) = 2 + 1 + 1 + 1 = 5 = m$, $\sum_{v \in V} d_{in}(v) = 1 + 1 + 1 + 2 = 5 = m$
- How about G_2?

Terminologies

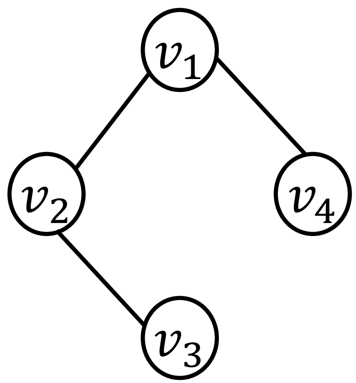
- A path from node u to node v in a graph G is a sequence of nodes, $(u, v_1, \dots, v_k, v)$ such that there exist a sequence of edges
  - $((u, v_1), (v_1, v_2), \dots, (v_k, v))$ if G is an undirected graph
  - $(\langle u, v_1\rangle, \langle v_1, v_2\rangle, \dots, \langle v_k, v\rangle)$ if G is a directed graph
- A simple path is a path in which all nodes except the first and last are distinct
- A cycle is a simple path in which the first and the last nodes are the same

- Example:
  - (v_1, v_3) is not a path
  - $(v_1, v_2, v_3, v_4, v_1, v_4)$ is a path but not a simple path
  - (v_1, v_2, v_3, v_4) is a simple path but not a cycle
  - $(v_1, v_2, v_3, v_4, v_1)$ is a simple path and a cycle



Graph representation: adjacency list

- Adjacency list for undirected graph
  - Each node $v \in V$ is associated with a linked list that stores all neighbors of v; we map an ID for each node



G

$v_1 \rightarrow head \leftrightarrow v_2 \leftrightarrow v_4 \leftrightarrow tail$

$v_2 \rightarrow head \leftrightarrow v_1 \leftrightarrow v_3 \leftrightarrow tail$

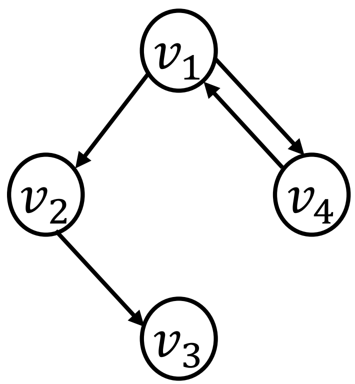
$v_3 \rightarrow head \leftrightarrow v_2 \leftrightarrow tail$

$v_4 \rightarrow head \leftrightarrow v_1 \leftrightarrow tail$

  - Space: $O(n+m)$, where n is the number of nodes and m is the number of edges

Graph representation: adjacency list

- Adjacency list for directed graph
  - Each node $v \in V$ is associated with a linked list that stores all out-neighbors of v



G

$v_1 \rightarrow head \leftrightarrow v_4 \leftrightarrow v_2 \leftrightarrow tail$

$v_2 \rightarrow head \leftrightarrow v_3 \leftrightarrow tail$

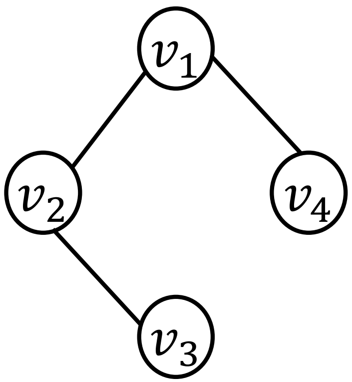
$v_3 \rightarrow head \leftrightarrow tail$

$v_4 \rightarrow head \leftrightarrow v_1 \leftrightarrow tail$

  - Space: $O(n + m)$, where n is the number of nodes and m is the number of edges

Graph representation: adjacency matrix

- Adjacency matrix for undirected graph
  - A $n \times n$ two dimensional matrix A where $A[u][v] = 1$ if $(u, v) \in E$, or 0 otherwise



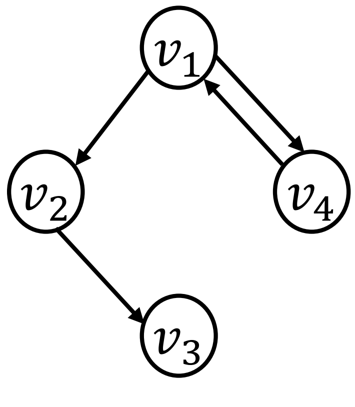
G

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	1	0	1	0
v_3	0	1	0	0
v_4	1	0	0	0

  - A is symmetric
  - Space: $O(n^2)$ where n is the number of nodes

Graph representation: adjacency matrix

- Adjacency matrix for directed graph
  - A $n \times n$ two dimensional matrix A where $A[u][v] = 1$ if $\langle u, v \rangle \in E$, or 0 otherwise



G

	v_1	v_2	v_3	v_4
v_1	0	1	0	1
v_2	0	0	1	0
v_3	0	0	0	0
v_4	1	0	0	0

  - A may not be symmetric
  - Space: $O(n^2)$ where n is the number of nodes

Comparison: Adjacency List/Matrix

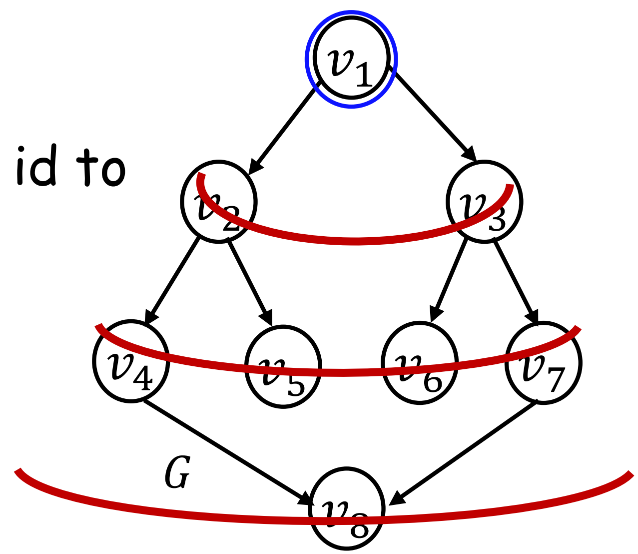
- Adjacency list:
  - Space: $O(n+m)$, save space if the graph is sparse, i.e., $m \ll n^2$
  - Check the existence of an edge (u,v): $O(k)$ time where k is the number of neighbors of v
  - Retrieve the neighbors of a node: $O(k)$ time
  - Add/delete a node: $O(n+m)$
  - Add/delete an edge: $O(k)$

- Adjacency matrix:
  - Space consumption: $O(n^2)$
  - Check the existence of an edge (u,v): $O(1)$ time
  - Retrieve the neighbors of a node: $O(n)$ time
  - Add/delete a node: $O(n^2)$, (create a new matrix)
  - Add/delete an edge: $O(1)$

Breadth-First Search (BFS)

- Intuition of BFS
  - Given a source node s, always visit nodes that are closer to the source s first before visiting the others

- The result is not unique, if we do not define an order among out-going edges from a node
  - Possible results
    - $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$
    - $v_1, v_3, v_2, v_7, v_6, v_5, v_4, v_8$
  - If we impose an order by going from smaller id to larger id, then the result will be unique



BFS steps

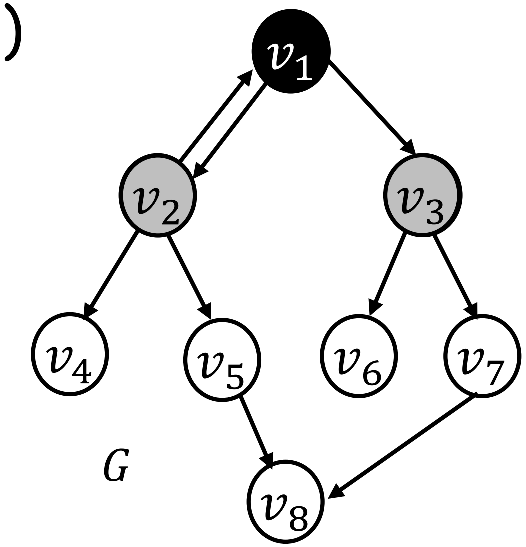
- At the beginning, color all nodes to be white
- Create a queue Q, enqueue the source s to Q, and color the source to be gray (meaning s is in the queue)
- Repeat the following until queue Q is empty
  - Dequeue from Q, let the node be v
  - For every out-neighbor u of v that is still white
    - Enqueue u into Q, and color u to gray (to indicate u is in queue)
  - Color v to be black (meaning v has finished)

- Example:
  - Assume the source is v_1

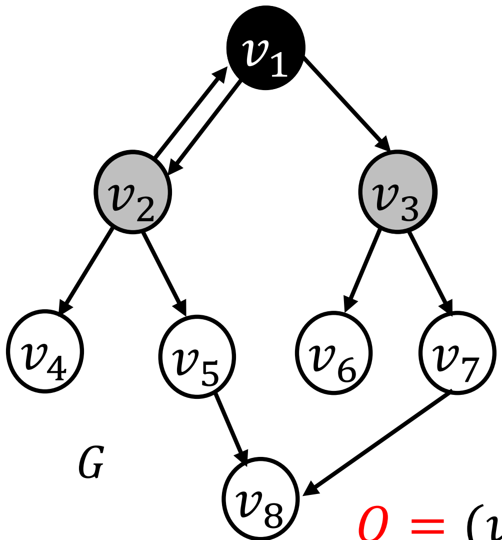
$$Q = (v_1)$$

After dequeuing v_1

$$Q = (v_2, v_3)$$

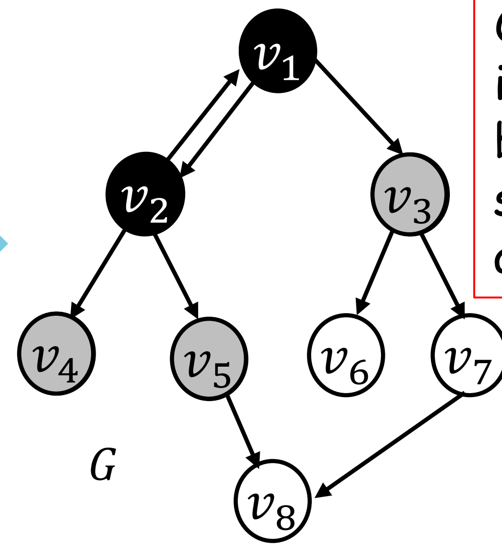


BFS: running example



$Q = (v_2, v_3)$

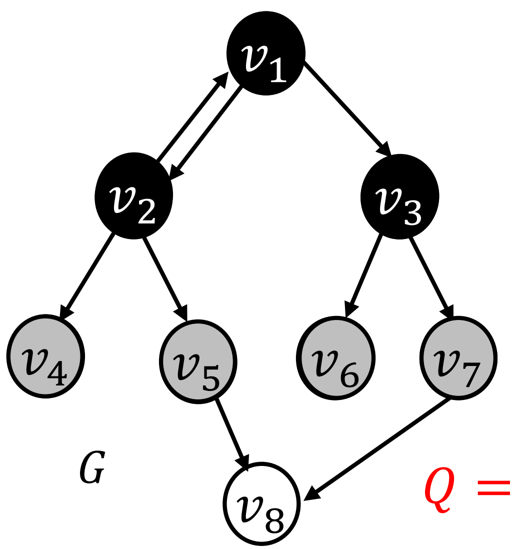
Dequeue from Q



Out-neighbor of v_2 includes v_1, v_4, v_5, but v_1 is not white, so only v_4 and v_5 are enqueued into Q

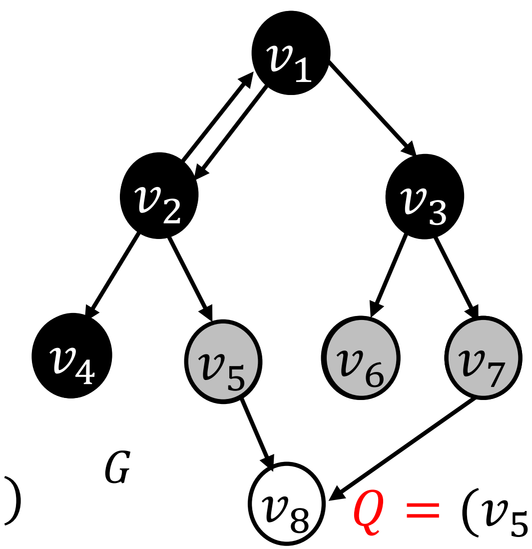
$Q = (v_3, v_4, v_5)$

Dequeue from Q



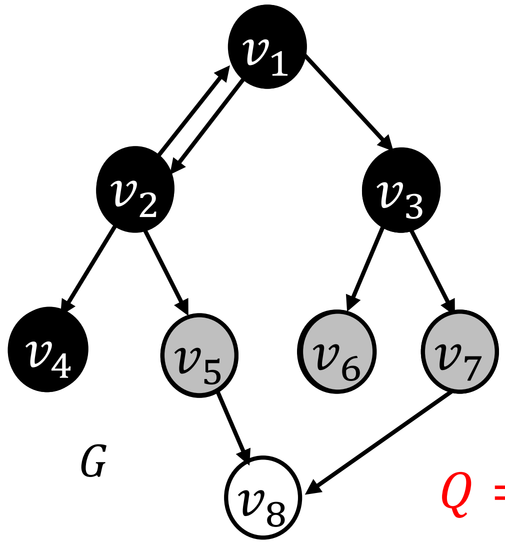
$Q = (v_4, v_5, v_6, v_7)$

Dequeue from Q



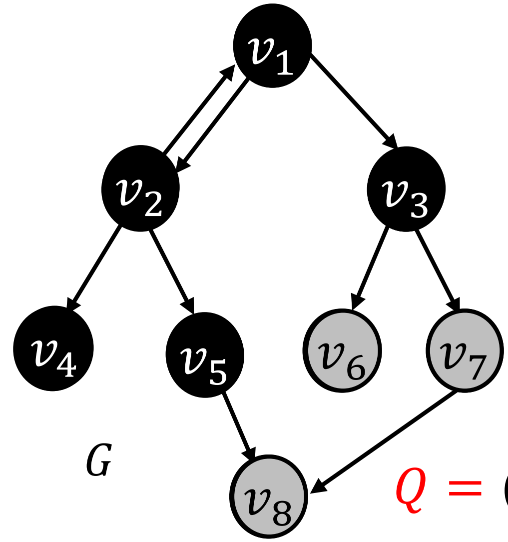
$Q = (v_5, v_6, v_7)$

BFS: running example



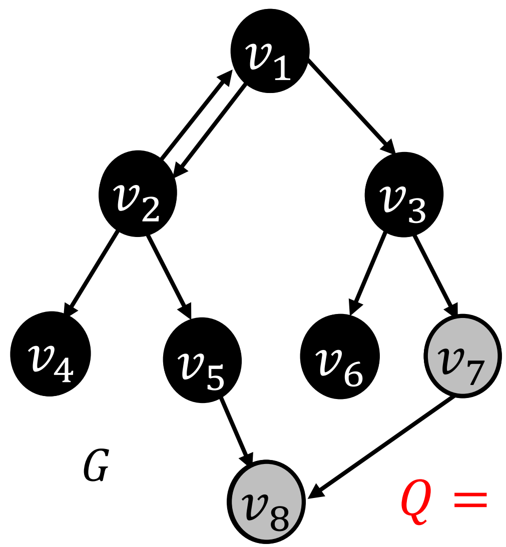
$Q = (v_5, v_6, v_7)$

Dequeue from Q



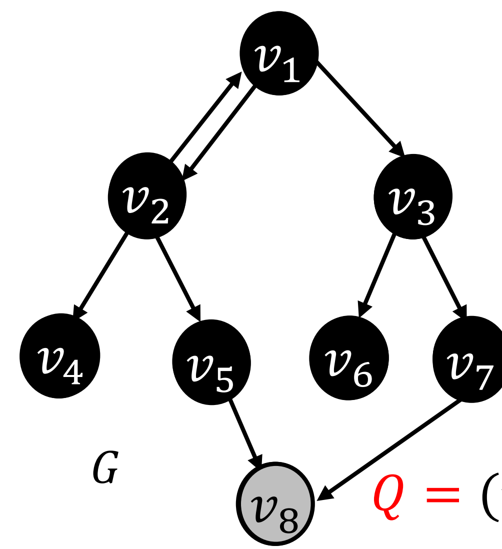
$Q = (v_6, v_7, v_8)$

Dequeue from Q



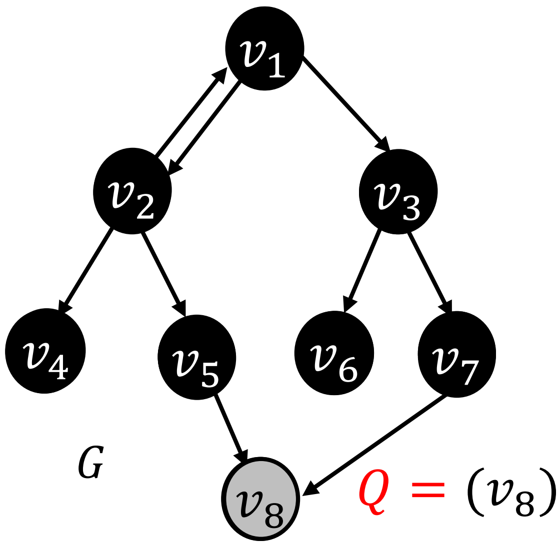
$Q = (v_7, v_8)$

Dequeue from Q

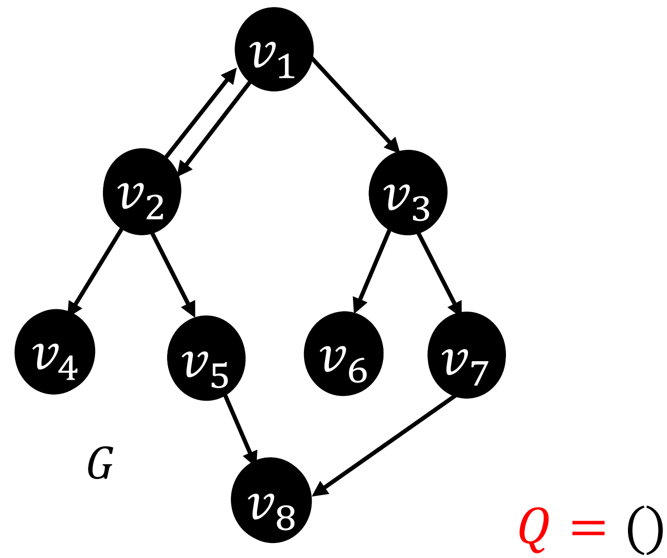


$Q = (v_8)$

BFS: running example



Dequeue from Q

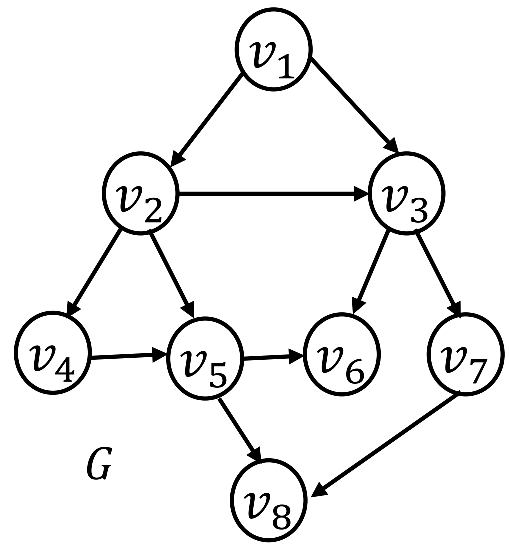


Now Q is empty

BFS finishes

Practice

- Given the following graph G, show the process of the BFS if the source is v_2
  - You may use 0 to denote the color to be white, 1 to denote the color to be gray, and 2 to denote the color to be black



BFS: implementation

Algorithm 1: *BFS(G, s)*

```
color← allocate an array of size G.n, initialize with all zeros
// Use 0 : white, 1 : gray, and 2: black
Q ← an empty queue
Q.enqueue(s)
color[s] ← 1
while !Q.isEmpty()
 v ← Q.dequeue
 for u ∈ out-neighbor of v
    if color[u]=0
       Q.enqueue(u)
       color[u]=1
 color[v] ← 2
 print v
free the array color if necessary
```

```
//adjacency matrix to store the graph
for u = 0 to G.n-1
  if G.adjmatrix[v][u]==1 and color[u]==0
     …

//adjacency list to store the graph
linkedlist_node = G[v].head.next
while linkedlist_node != G[v].tail
   u =linkedlist_node.element
   …
   linkedlist_node = linkedlist_node.next
```

BFS: time complexity

- When a node u is dequeued,
  - We examine all of its neighbors (check their color), enqueue them and color them to gray if they are white
  - After that, we color u as black
  - This incurs $c(1 + d_{out}(u))$ costs for node u where c is a constant (if we use adjacency list to represent the graph)

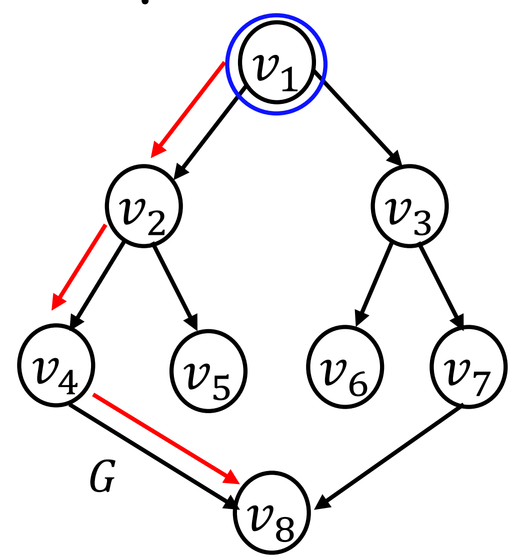
- Each node is dequeued at most once
  - Why? we enqueue a node at most once
  - If it is in the queue, its color is gray and we will not further enqueue it

- Therefore, the total running time with adjacency list representation is:
  - $\sum_{u \in V} c(1 + d_{out}(u)) = c(n + m) = O(n + m)$

Depth-First Search (DFS)

- Going along one path until we cannot go further

- Imposing an order to make the traversal unique: from smaller id to larger id
  - Visiting order: $v_1, v_2, v_4, v_8, v_5, v_3, v_6, v_7$



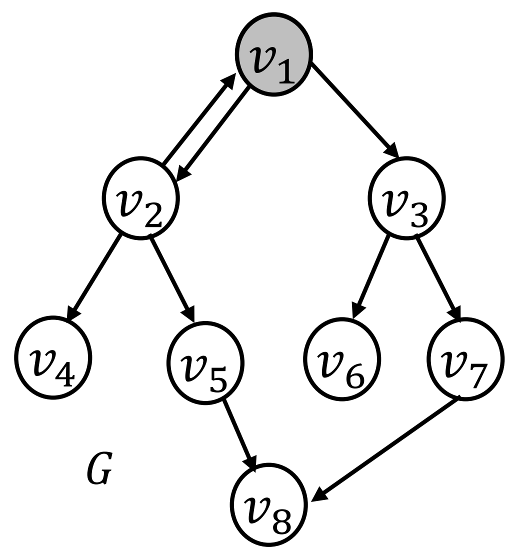
- We still focus on directed graph
  - Extension to undirected graph will be straightforward

Depth-First Search (DFS)

- Initialization:
  - At the beginning, color all nodes to be white
  - Create a stack S, push the source s to S, and color the source to be gray (meaning s is in the stack)

- Example:
- Assume that v_1 is the source

$S = $ | v_1 | |

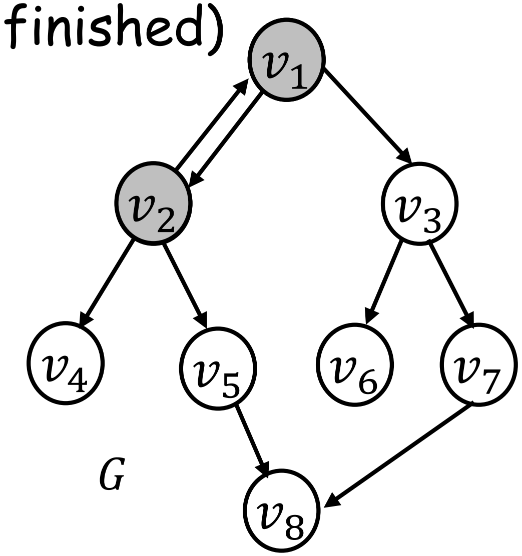


Depth-First Search (DFS)

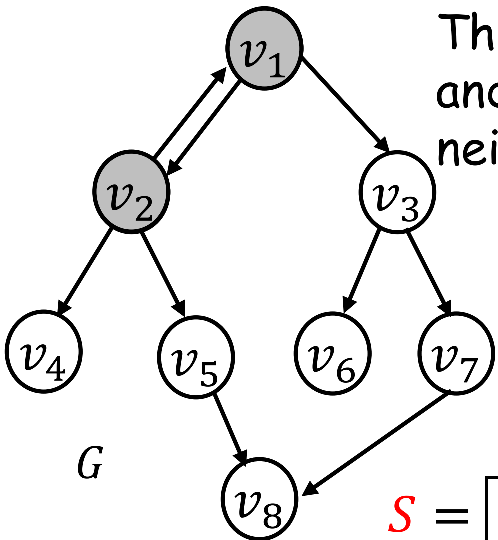
- Repeat the following until S is empty
  - Get the top node, denoted as v, on stack S, **do not** pop v
  - If v still has white out-neighbors
    - Let u be such a white out-neighbor of v
    - Push u to S, and color u to gray
  - Otherwise (v has no white out-neighbors)
    - Pop v and color it as black (meaning that v has finished)

$S = \boxed{v_1}$

$S = \boxed{v_1}\boxed{v_2}$



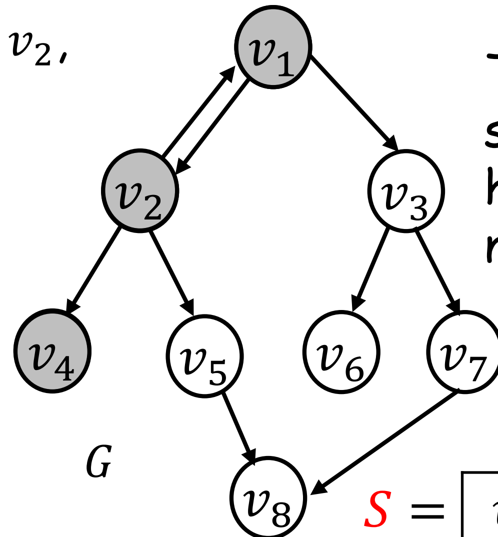
DFS: running example



The top of the stack is v_2, and it has white out-neighbor v_4

$S =$

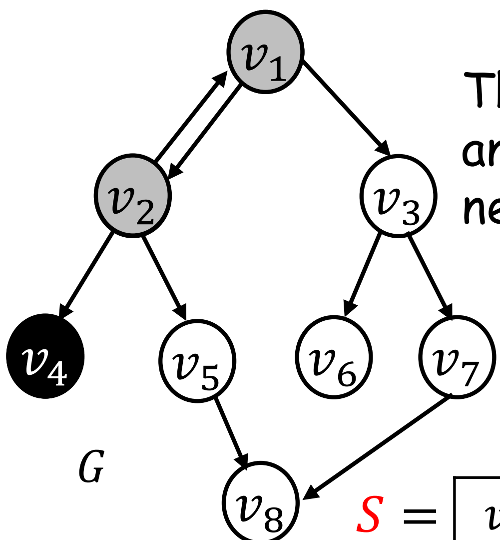
v_1	v_2	



The top of the stack is v_4, and it has no white out-neighbor

$S =$

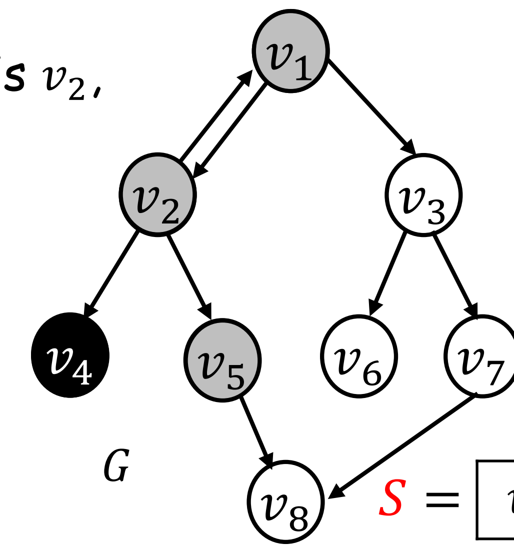
v_1	v_2	v_4	



The top of the stack is v_2, and it has white out-neighbor v_5

$S =$

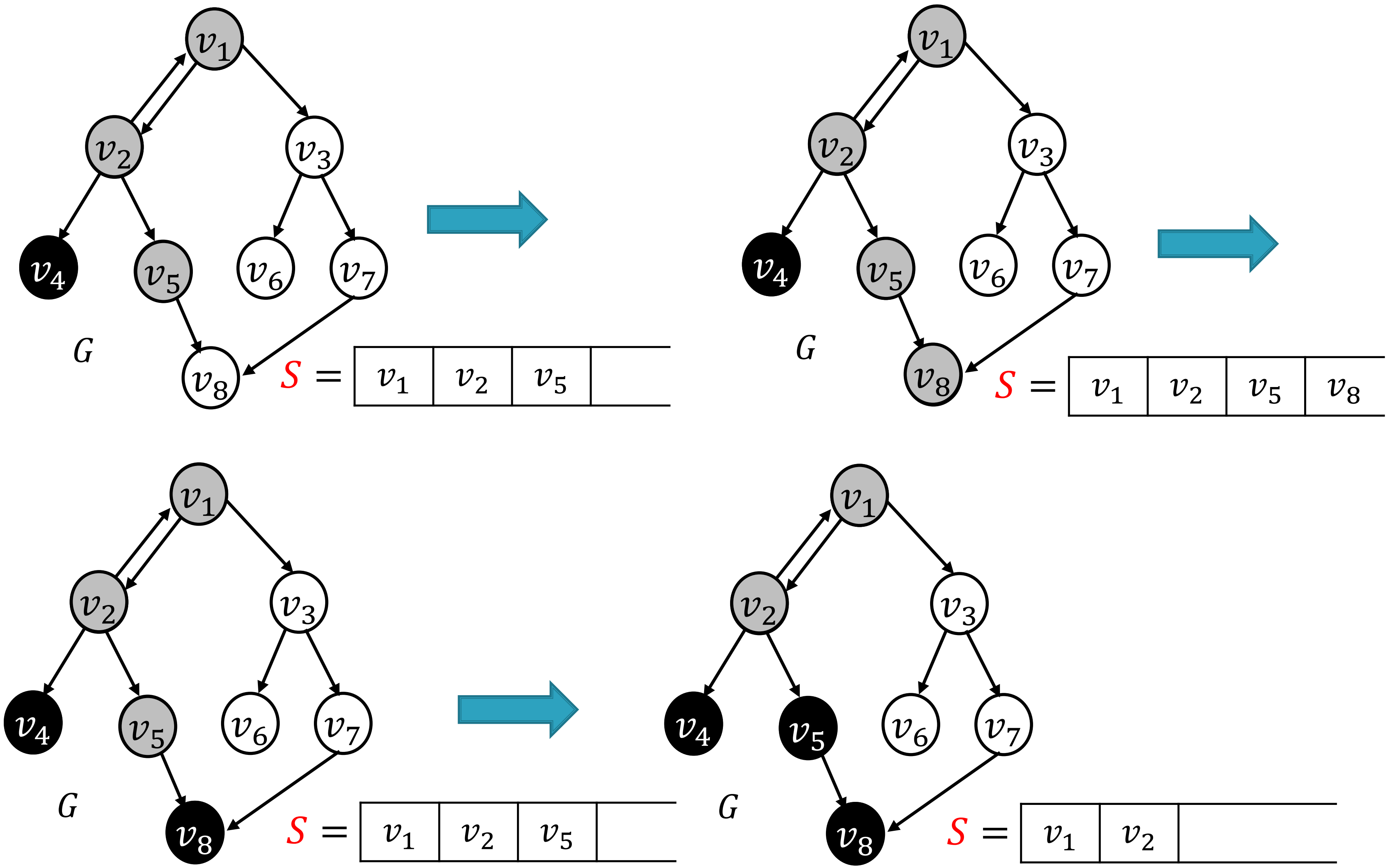
v_1	v_2	



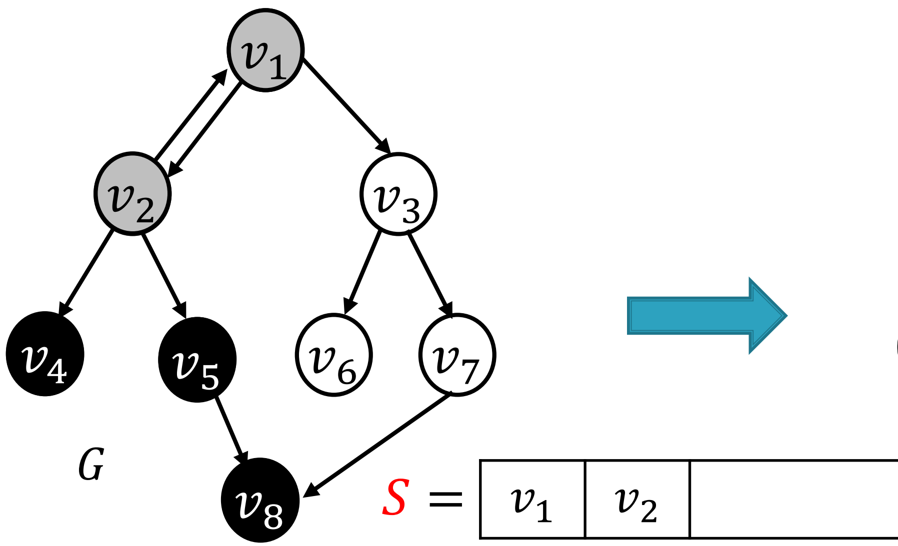
$S =$

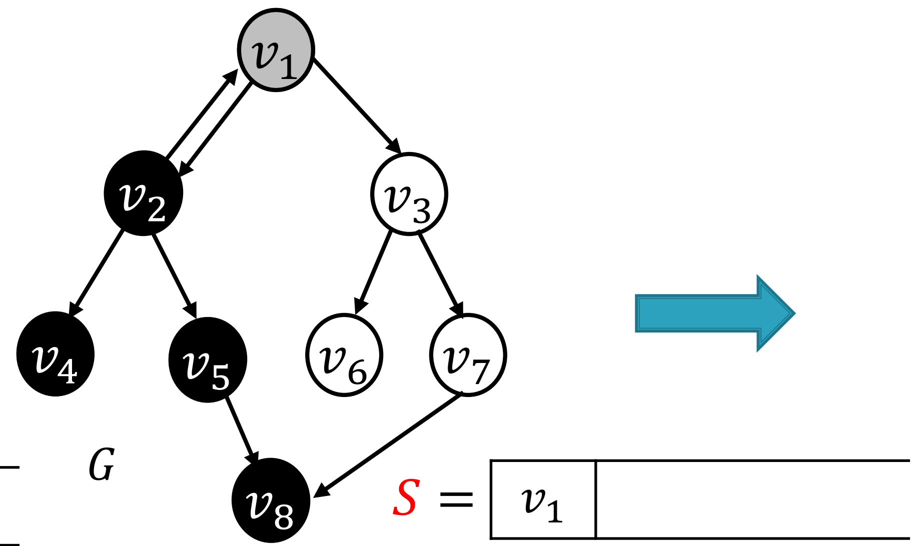
v_1	v_2	v_5	

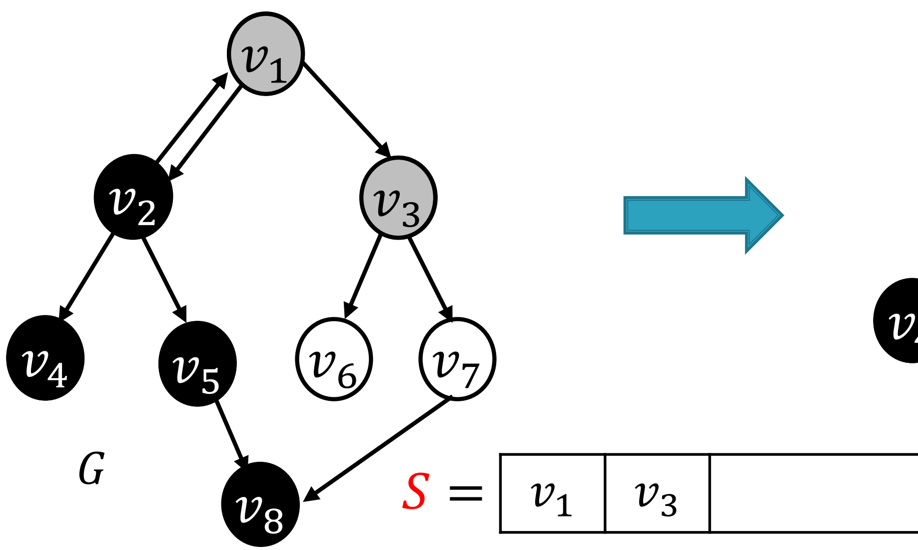
DFS: running example

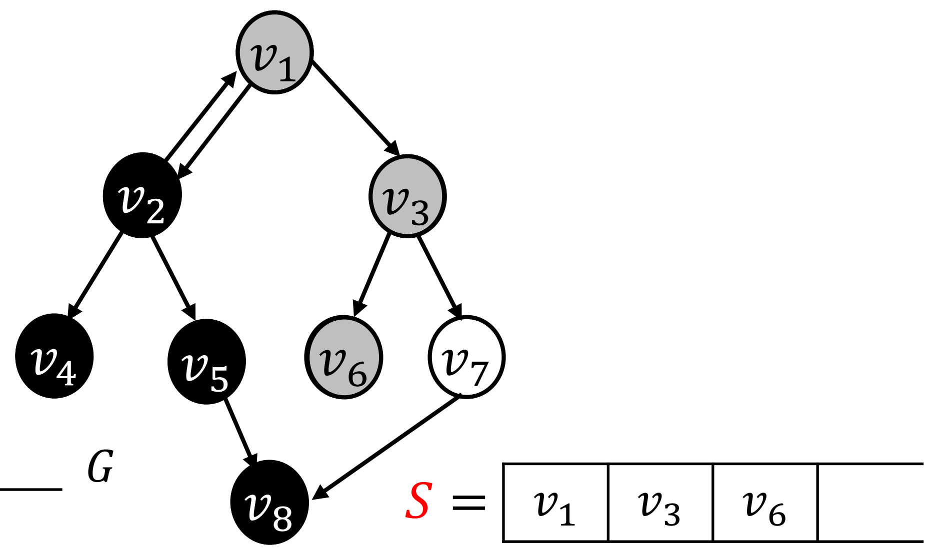


DFS: running example

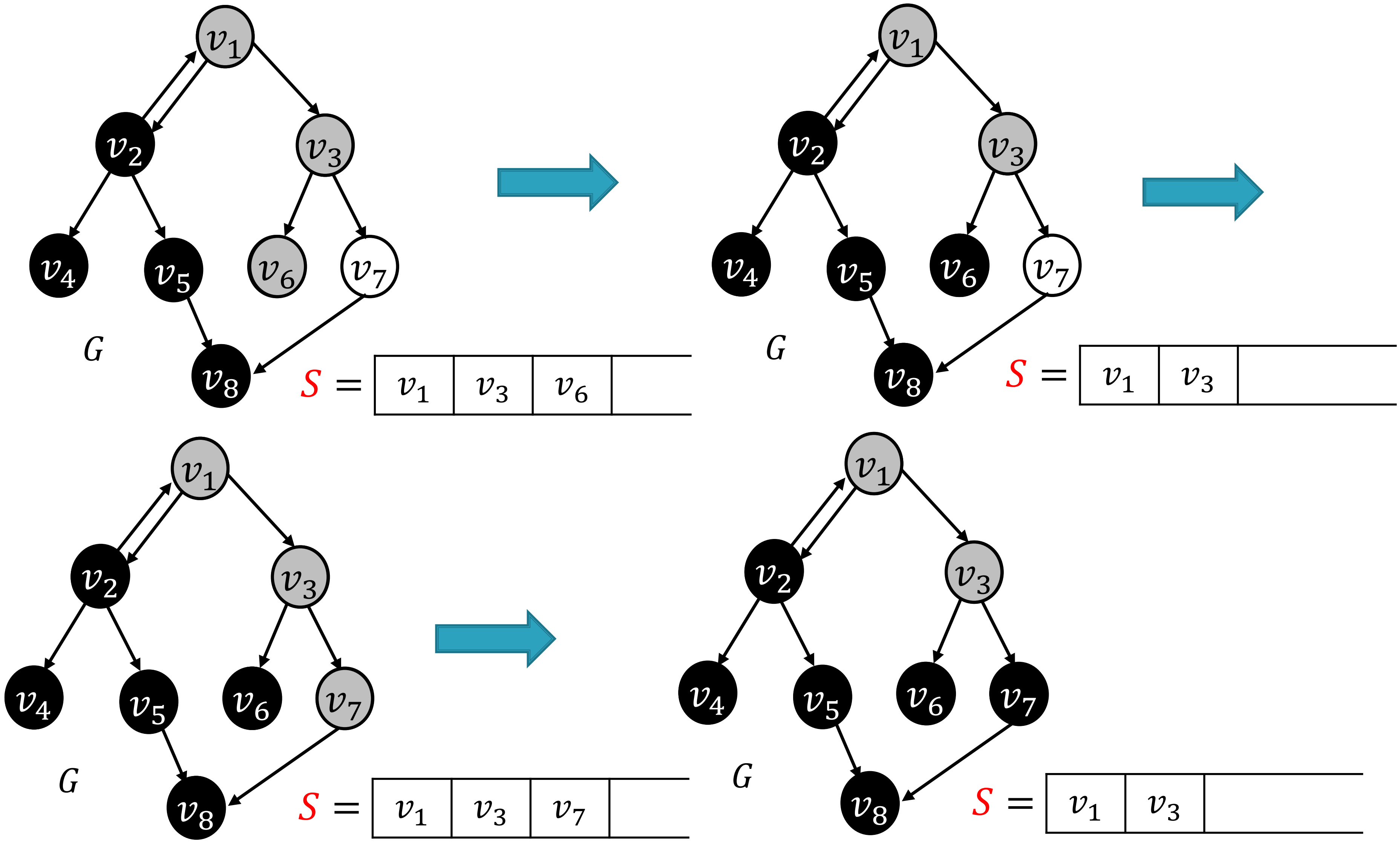




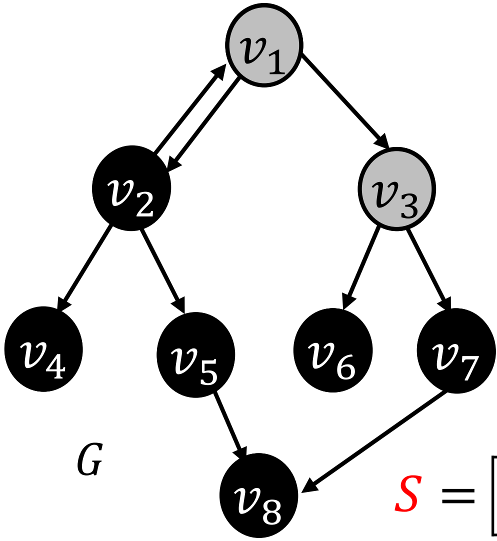




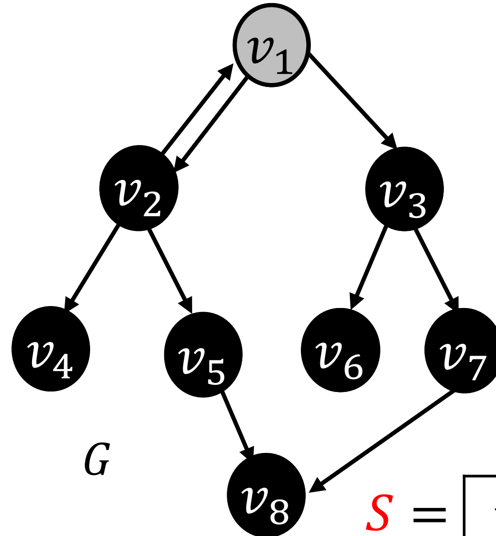
DFS: running example



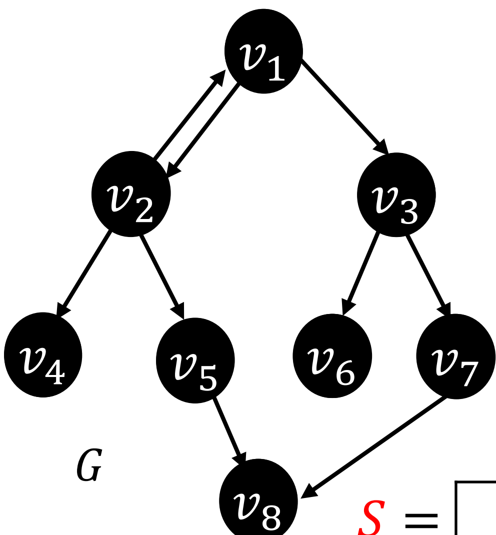
DFS: running example



$S =$ | v_1 | v_3 |



$S =$ | v_1 |

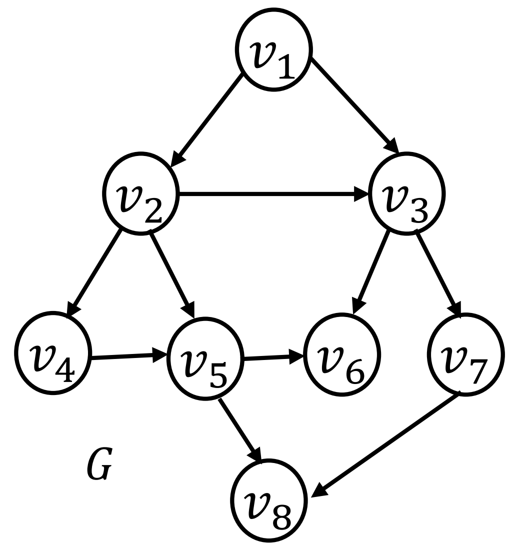


$S =$

Stack empty, DFS from source v_1 finishes

Practice

- Given the following graph G, show the process of the DFS if the source is v_2
  - You may use 0 to denote the color to be white, 1 to denote the color to be gray, and 2 to denote the color to be black



DFS: implementation

Algorithm 1: ***DFS(V, E, s)***

```
color← initialize an array of size n with all zero values
// Use 0 : white, 1 : gray, and 2: black
S ← an empty stack
S.push(s)
color[s] ← 1
while !S.isEmpty()
   v ← S.top()
   if v still has white-neighbor u
         S.push(u)
         color[u]=1
   else
      color[v] ← 2
      S.pop()
Free color array if necessary
```

DFS: complexity analysis

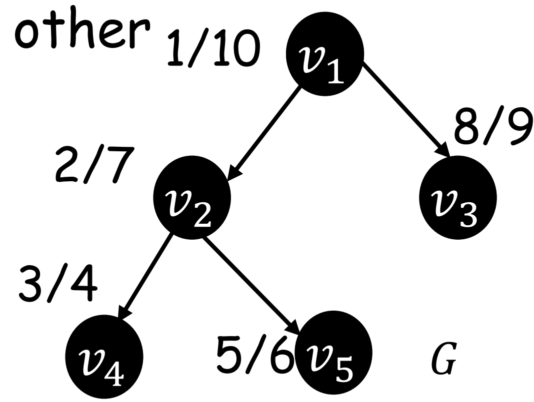
- When a node v get popped from the stack?
  - None of its out-neighbors is a white node
  - We may need to repeated check if node v has white out-neighbor
    - We need to check $d_{out}(v)$ times in the worst case
    - Cost: $d_{out}(v) \cdot d_{out}(v)$?
    - Can we do better?
      - We record the position checked last time
      - All nodes in previous positions will not be white
      - Cost: $O(d_{out}(v))$

- As each node is popped at most once, the total time cost is
  - $\sum_{u \in V} c(1 + d_{out}(u)) = c(n + m) = O(n + m)$

Properties of DFS

- Let $u.d$ and $u.f$ to indicate their first discovery time and their finish time, respectively, and denote $I(u)$ as the interval $[u.d, u.f]$

- We will only have three cases for two nodes u and v
  - $I(u) \subset I(v)$, u is the descendant of v
  - $I(v) \subset I(u)$, v is the descendant of u
  - $I(u) \cap I(v) = \emptyset$, neither one is the descendant of the other
  - Example:
    - $I(v_2)$: $[2,7]$, $\mathrm{I}(v_4) = [3,4]$
      - v_4 is a descendant of v_2 in the DFS tree



- We can check if a node u is a descendant of another node v in $O(1)$ time
  - If there is no such property, we need to retrieve the path using the prev array

Recommended reading

- Reading materials
  - Textbook Chapters 22.1-22.3
- Next lecture
  - Graph minimum spanning trees, Textbook Chapter 23