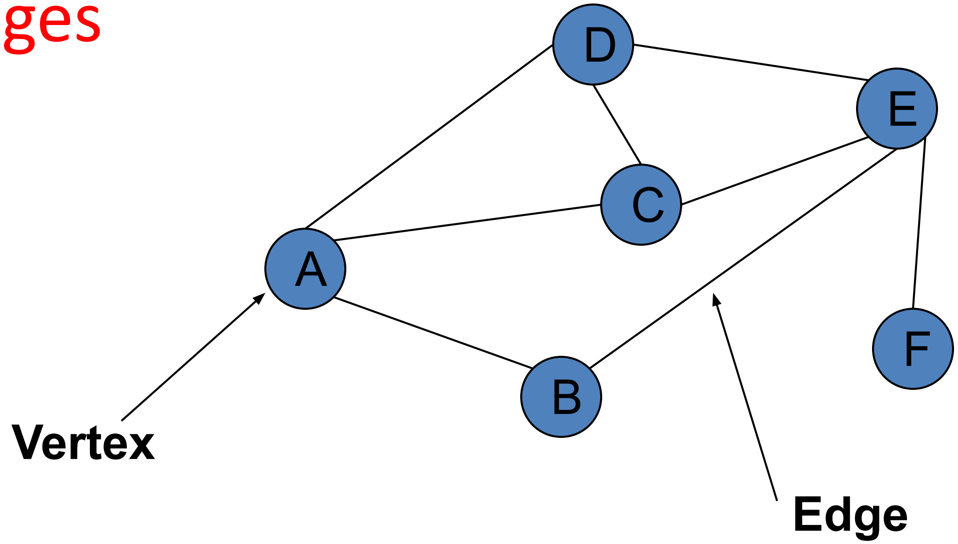
Graph
BFS & DFS

Graphs

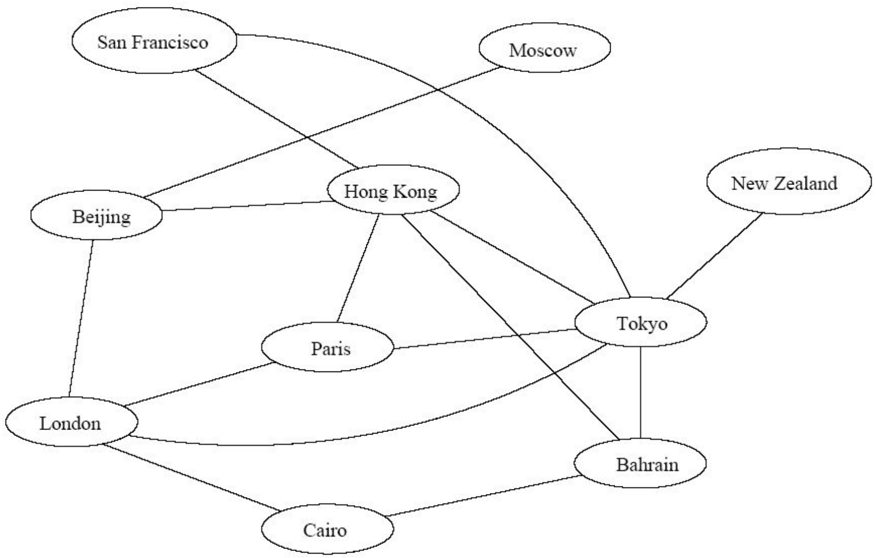
- Extremely useful tool in modeling problems
- Consist of:
  - Vertices
  - Edges



Vertices can be considered "sites" or locations.

Edges represent connections.

Application

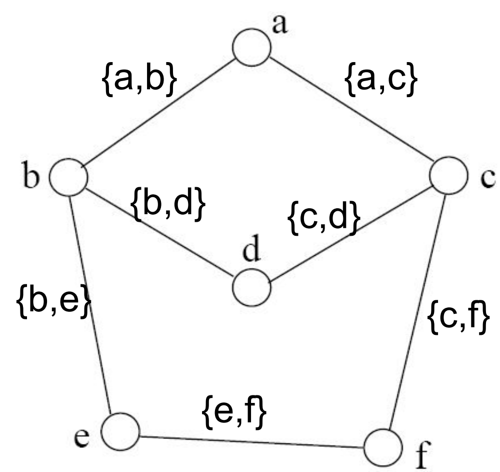


Air flight system

- Each vertex represents a city
- Each edge represents a direct flight between two cities
- A query on direct flights = a query on whether an edge exists
- A query on how to get to a location = does a path exist from A to B
- We can even associate costs to edges (weighted graphs), then ask "what is the cheapest path from A to B"

Definition

- A graph G=(V, E) consists a set of vertices, V, and a set of edges, E.
- Each edge is a pair of *(v, w)*, where v, w belongs to V
- If the pair is unordered, the graph is undirected; otherwise it is directed



$$V = \{a, b, c, d, e, f\}$$

$$E = \{\{a, b\}, \{a, c\}, \{b, d\}, \{c, d\}, \{b, e\}, \{c, f\}, \{e, f\}\}$$

An undirected graph

Definition

- Connected Components
- Bipartite Graph
- Path
- Cycles

Graph Variations

- Variations:
  - A *connected graph* has a path from every vertex to every other
  - In an *undirected graph:*
    - Edge (u,v) = edge (v,u)
  - In a *directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated $u \rightarrow v$

Graph Variations

- More variations:
  - A *weighted graph* associates weights with either the edges or the vertices
    - E.g., a road map: edges might be weighted w/ distance

Graph Representation

- Two popular computer representations of a graph. Both represent the vertex set and the edge set, but in different ways.

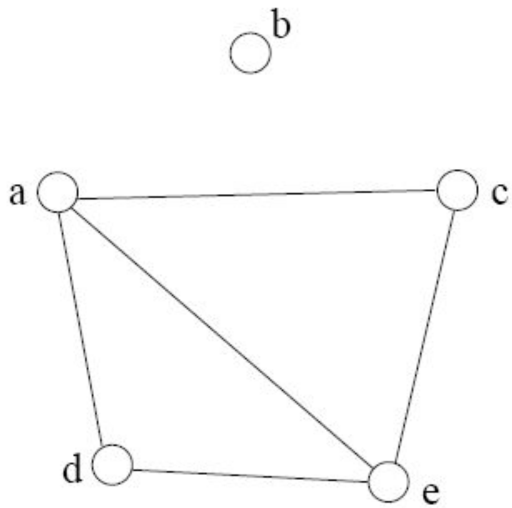
1. Adjacency Matrix

   Use a 2D matrix to represent the graph

2. Adjacency List

   Use a 1D array of linked lists (We will implement it using vector in c++)

Adjacency Matrix



	a	b	c	d	e
a	0	0	1	1	1
b	0	0	0	0	0
c	1	0	0	0	1
d	1	0	0	0	1
e	1	0	1	1	0

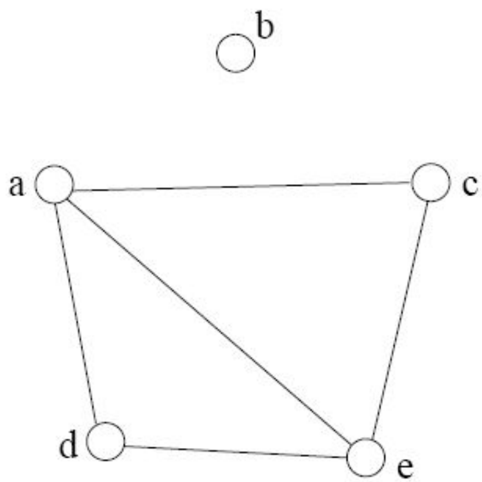
- 2D array A[0..n-1, 0..n-1], where ***n*** is the number of vertices in the graph
- Each row and column is indexed by the vertex id
  - e,g a=0, b=1, c=2, d=3, e=4
- *A[i][j]=1* if there is an edge connecting vertices *i* and *j*; otherwise, *A[i][j]=0*
- The storage requirement is $\Theta(n^2)$. It is not efficient if the graph has few edges. An adjacency matrix is an appropriate representation if the graph is dense: $|E|=\Theta(|V|^2)$
- We can detect in O(1) time whether two vertices are connected.

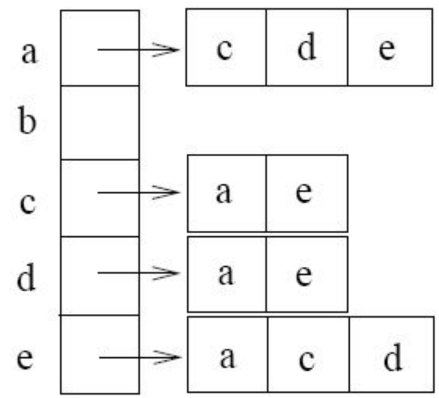
Simple Questions on Adjacency Matrix

Can you solve the following problems?

- Is there a direct link between A and B?
- How many nodes are directly connected to vertex A?
- Is it an undirected graph or directed graph?
- Suppose ADJ is an NxN matrix. What will be the result if we create another matrix ADJ2 where ADJ2=ADJxADJ?

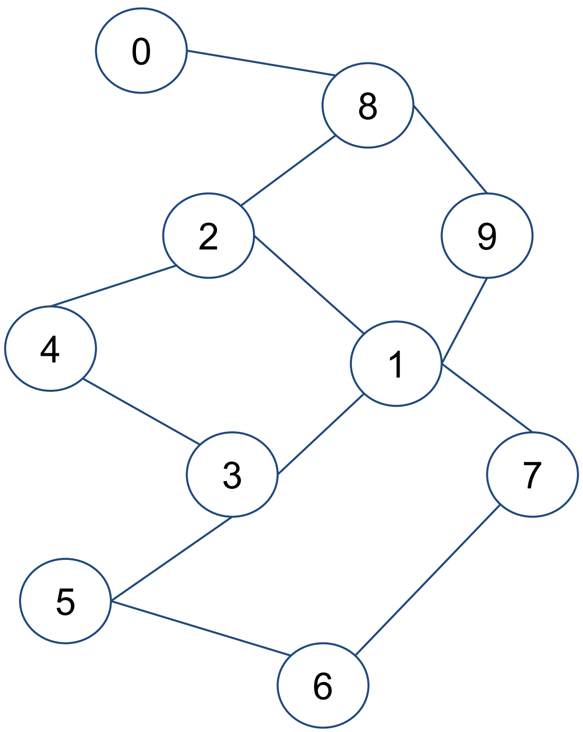
Adjacency List





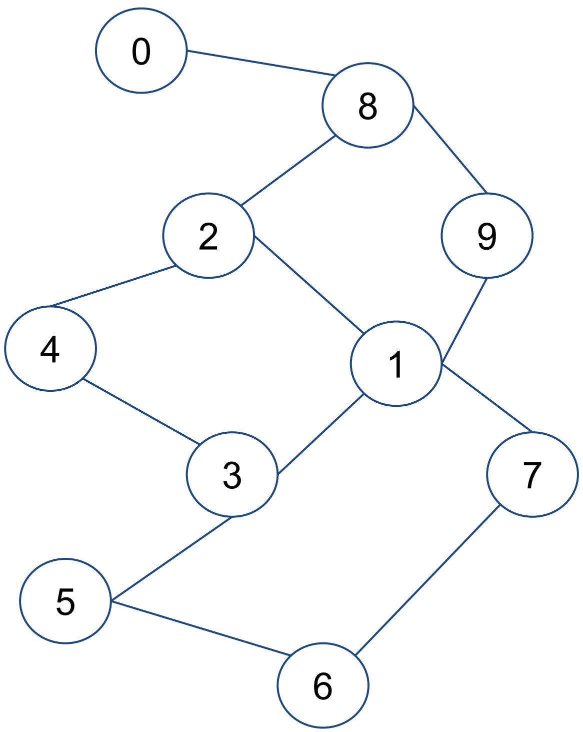
- If the graph is not dense, in other words, sparse, a better solution is an adjacency list
- The adjacency list is an array A[1..n] of lists, where n is the number of vertices in the graph.
- Each array entry is indexed by the vertex id
- Each list *A[i]* stores the ids of the vertices adjacent to vertex *i*

Adjacency Matrix Example



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

Adjacency List Example



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

Storage of Adjacency List

- The array takes up $\Theta(n)$ space
- Define degree of v, $\deg(v)$, to be the number of edges incident to v. Then, the total space to store the graph is proportional to:

$$\sum_{\text{vertex } v} \deg(v)$$

- An edge $e=\{u,v\}$ of the graph contributes a count of 1 to $\deg(u)$ and contributes a count 1 to $\deg(v)$
- Therefore, $\Sigma_{\text{vertex } v}\deg(v) = 2m$, where m is the total number of edges
- In all, the adjacency list takes up $\Theta(n+m)$ space
  - If $m = O(n^2)$ (i.e. dense graphs), both adjacent matrix and adjacent lists use $\Theta(n^2)$ space.
  - If $m = O(n)$, adjacent list outperform adjacent matrix

- However, one cannot tell in $O(1)$ time whether two vertices are connected

Adjacency List vs. Matrix

- **Adjacency List**
  - More compact than adjacency matrices if graph has few edges
  - Requires more time to find if an edge exists

- **Adjacency Matrix**
  - Always require n^2 space
    - This can waste a lot of space if the number of edges are sparse
  - Can quickly find if an edge exists

Path between Vertices

- A path is a sequence of vertices (v_0, v_1, v_2,… v_k) such that:
  - For $0 \leq i < k$, $\{v_i, v_{i+1}\}$ is an edge

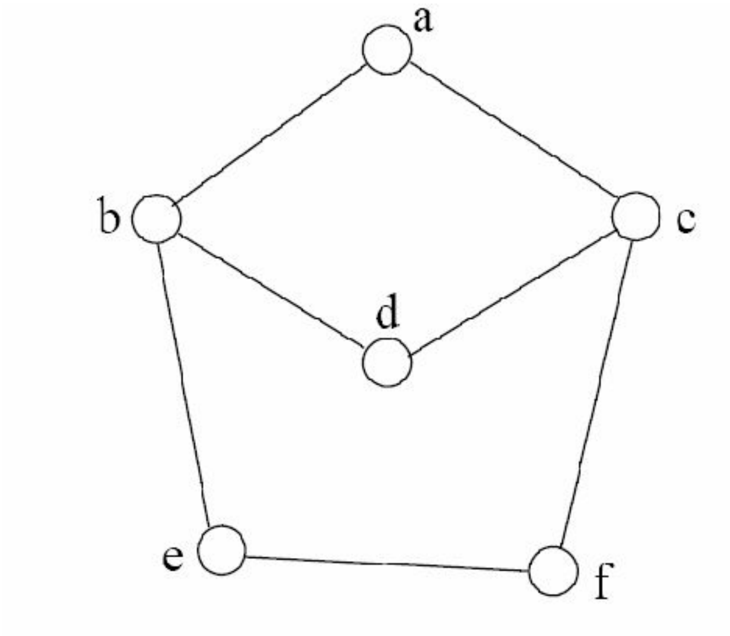
  Note: a path is allowed to go through the same vertex or the same edge any number of times!

- The length of a path is the number of edges on the path

Types of paths

- A path is simple if and only if it does not contain a vertex more than once.
- A path is a cycle if and only if $v_0 = v_k$
  - The beginning and end are the same vertex!
- A path contains a cycle as its sub-path if some vertex appears twice or more

Path Examples



Are these paths?

Any cycles?

What is the path's length?

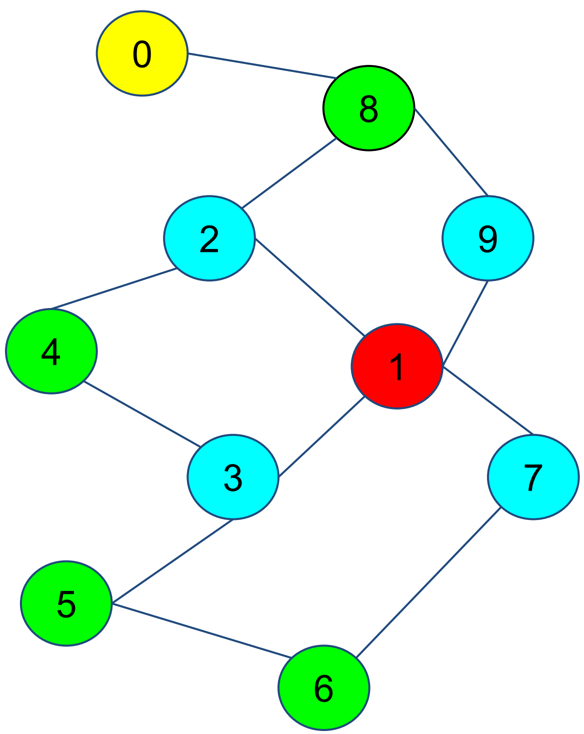
1. {a,c,f,e}
2. {a,b,d,c,f,e}
3. {a, c, d, b, d, c, f, e}
4. {a,c,d,b,a}
5. {a,c,f,e,b,d,c,a}

Graph Traversal

- Application example
  - Given a graph representation and a vertex **s** in the graph
  - Find paths from **s** to other vertices
- Two common graph traversal algorithms
    - Breadth-First Search (BFS)
      - Finds the shortest paths in an unweighted graph
    - Depth-First Search (DFS)

BFS and Shortest Path Problem

- Given any source vertex ***s***, BFS visits the other vertices at increasing distances away from s. In doing so, BFS discovers paths from s to other vertices
- What do we mean by "distance"? The number of edges on a path from s



Example

Consider s=vertex 1

Nodes at distance 1:
2, 3, 7, 9

Nodes at distance 2:
8, 6, 5, 4

Nodes at distance 3:
0

Graph Searching

- Given: a graph G = (V, E), directed or undirected and a source S.
- Goal: Find the shortest path to every node of the graph, from S

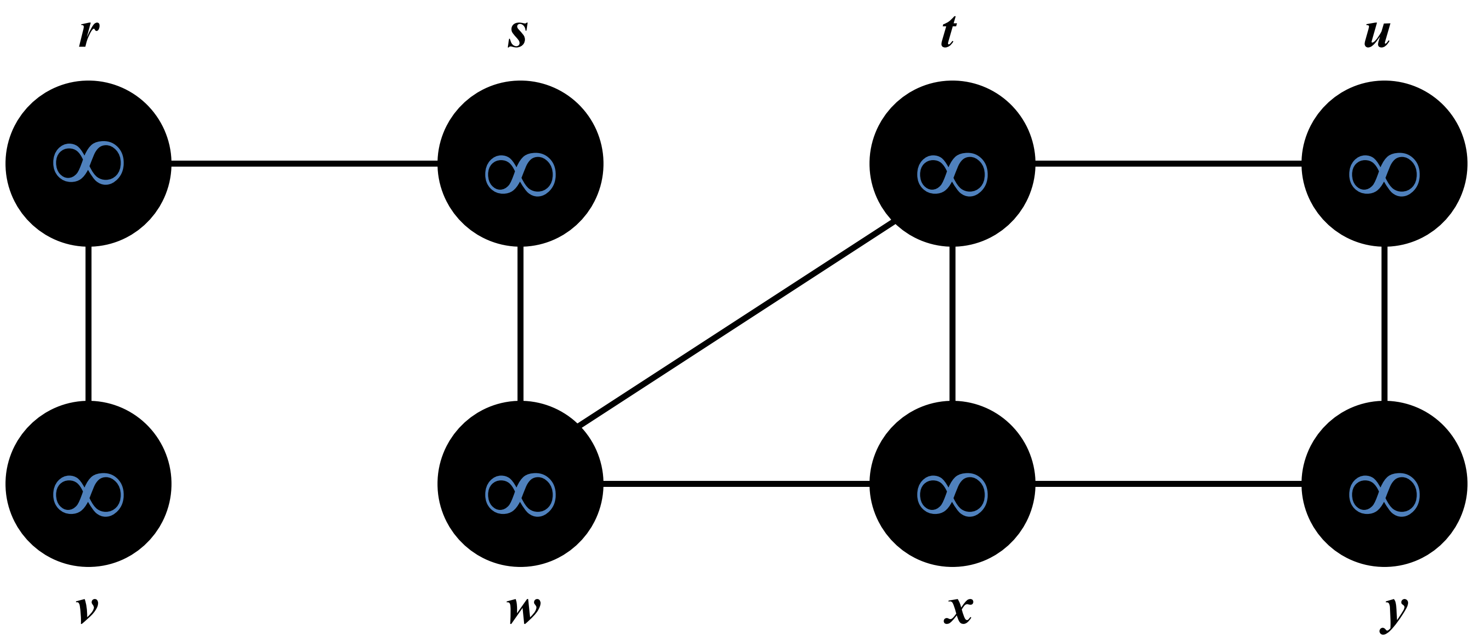
Breadth-First Search

- “Explore” a graph, starting from the source
  - Maintains a queue of nodes that are yet to be explored.
  - “Expand” (dequeue i.e. pop from the queue) one node at a time
  - Enqueue (i.e. push) all nodes adjacent to the one we are expanding (if they have never been enqueued before)

Breadth-First Search: The Code

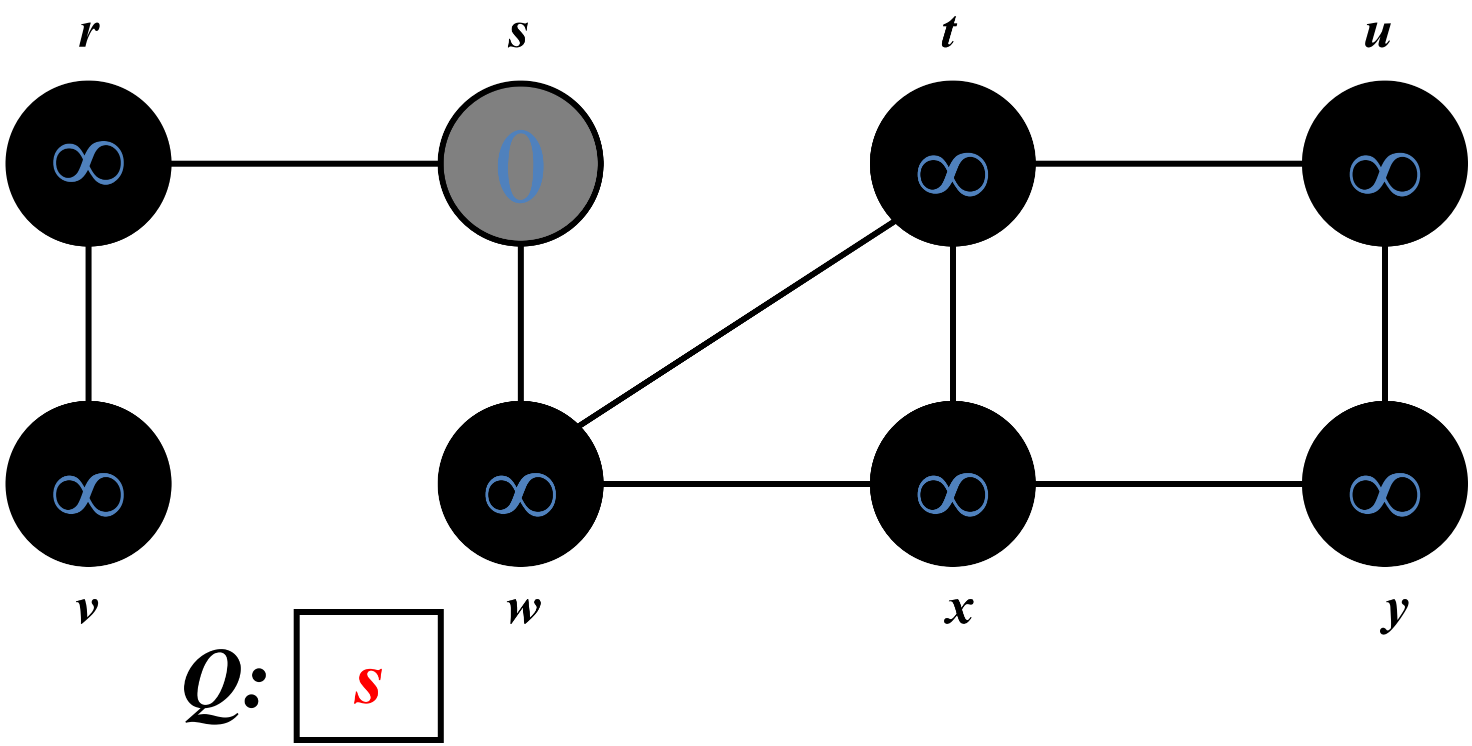
```
vector <int> adj[NODES];
queue <int> q;
int dis[NODES], par[NODES];
void bfs(int s){
    for(int i=1; i<=n; i++){
        dis[i]=-1;
    }
    dis[s]=0;
    par[s]=s;
    q.push(s);
    int u, v;
    while(!q.empty()){
        u=q.front();
        q.pop();
        for(int i=0; i<adj[u].size(); i++){
            v=adj[u][i];
            if(dis[v]==-1){
                dis[v]=1+dis[u];
                par[v]=u;
                q.push(v);
            }
        }
    }
}
```

Breadth-First Search: Example



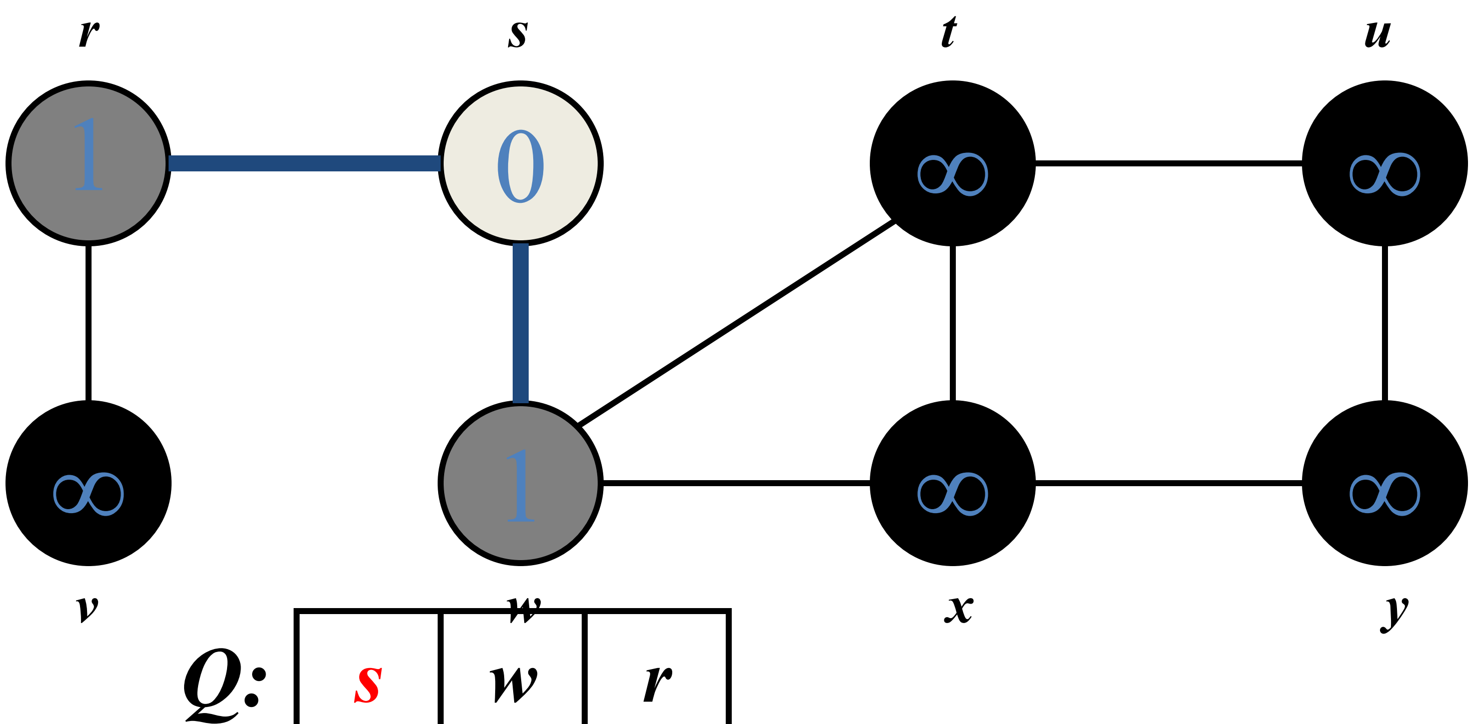
Vertex	r	s	t	u	v	w	x	y
d	∞	∞	∞	∞	∞	∞	∞	∞
par	nil	nil	nil	nil	nil	nil	nil	nil

Breadth-First Search: Example



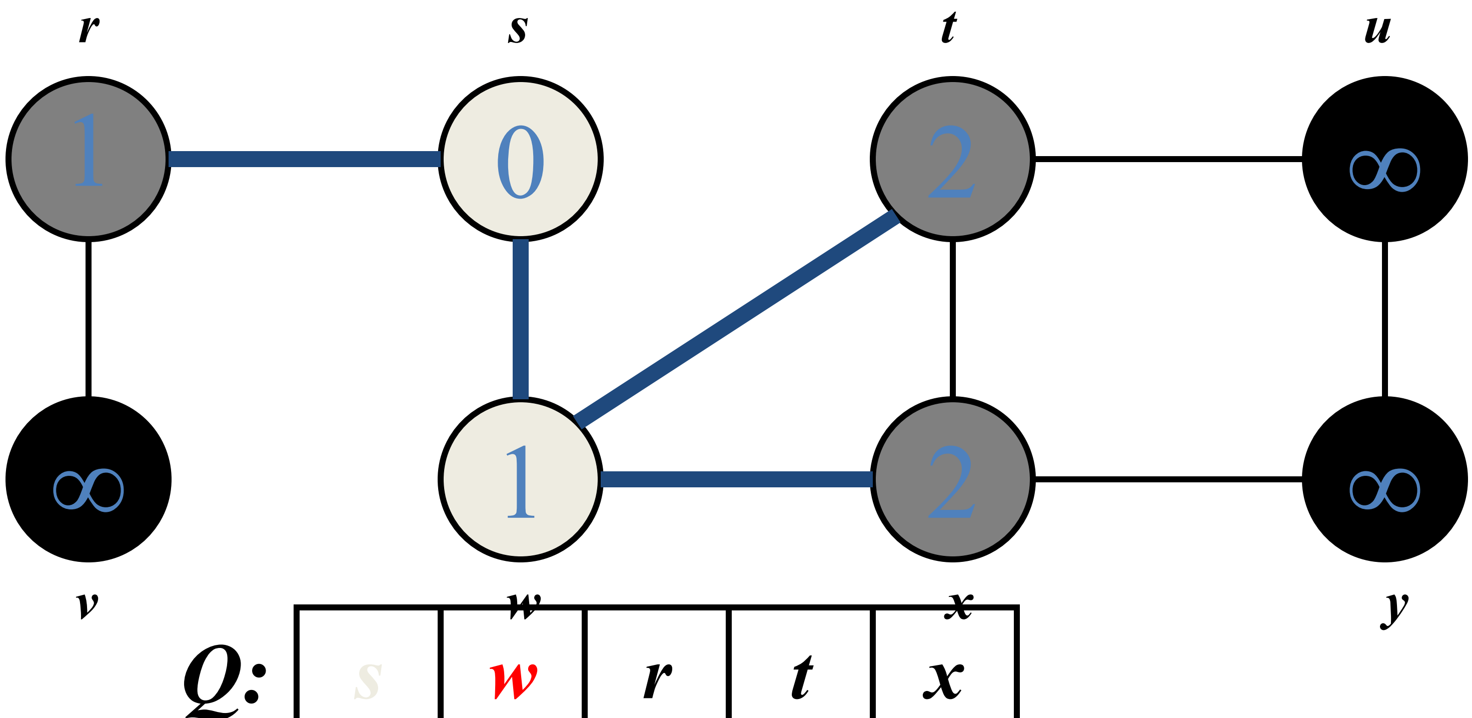
vertex	r	s	t	u	v	w	x	y
d	∞	**0**	∞	∞	∞	∞	∞	∞
par	nil	**nil**	nil	nil	nil	nil	nil	nil

Breadth-First Search: Example



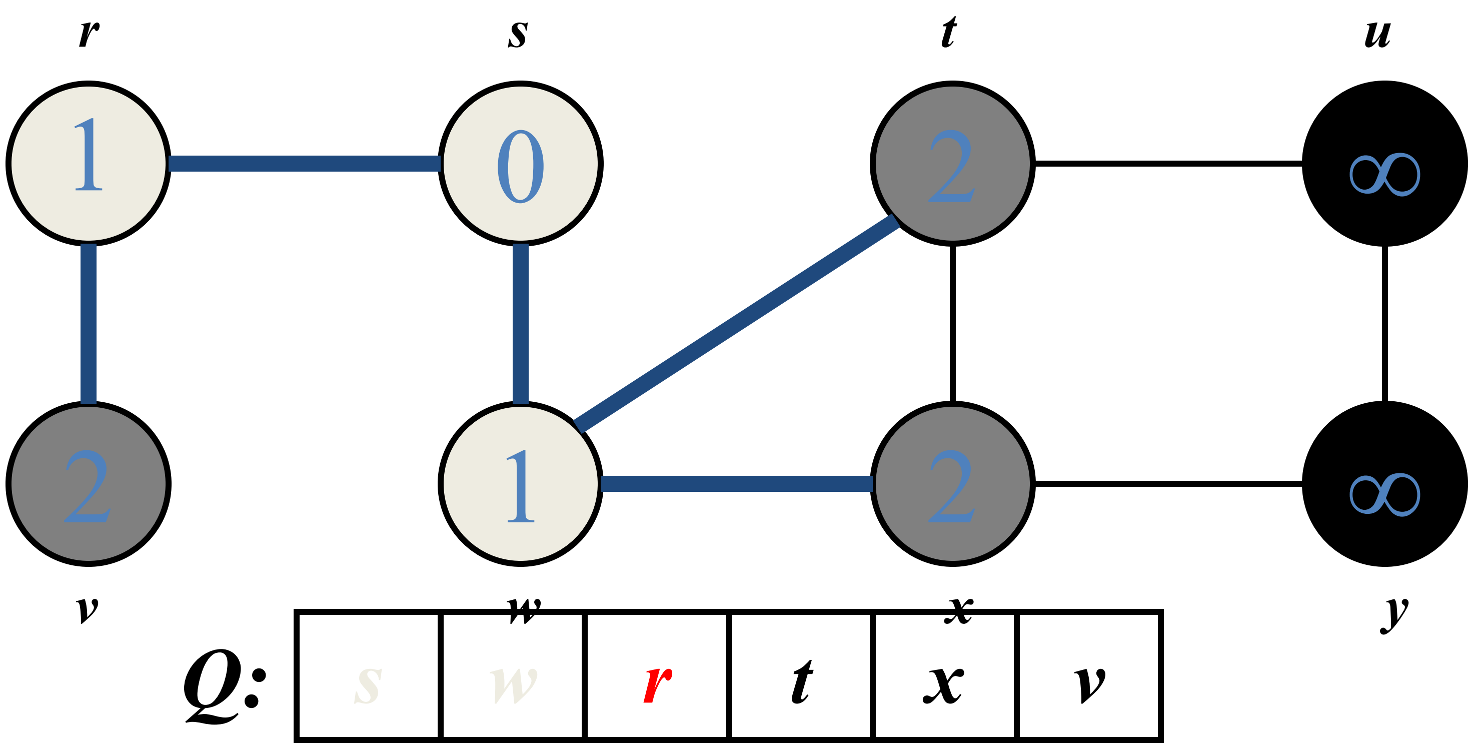
vertex	r	s	t	u	v	w	x	y
d	1	0	∞	∞	∞	1	∞	∞
par	s	nil	nil	nil	nil	s	nil	nil

Breadth-First Search: Example



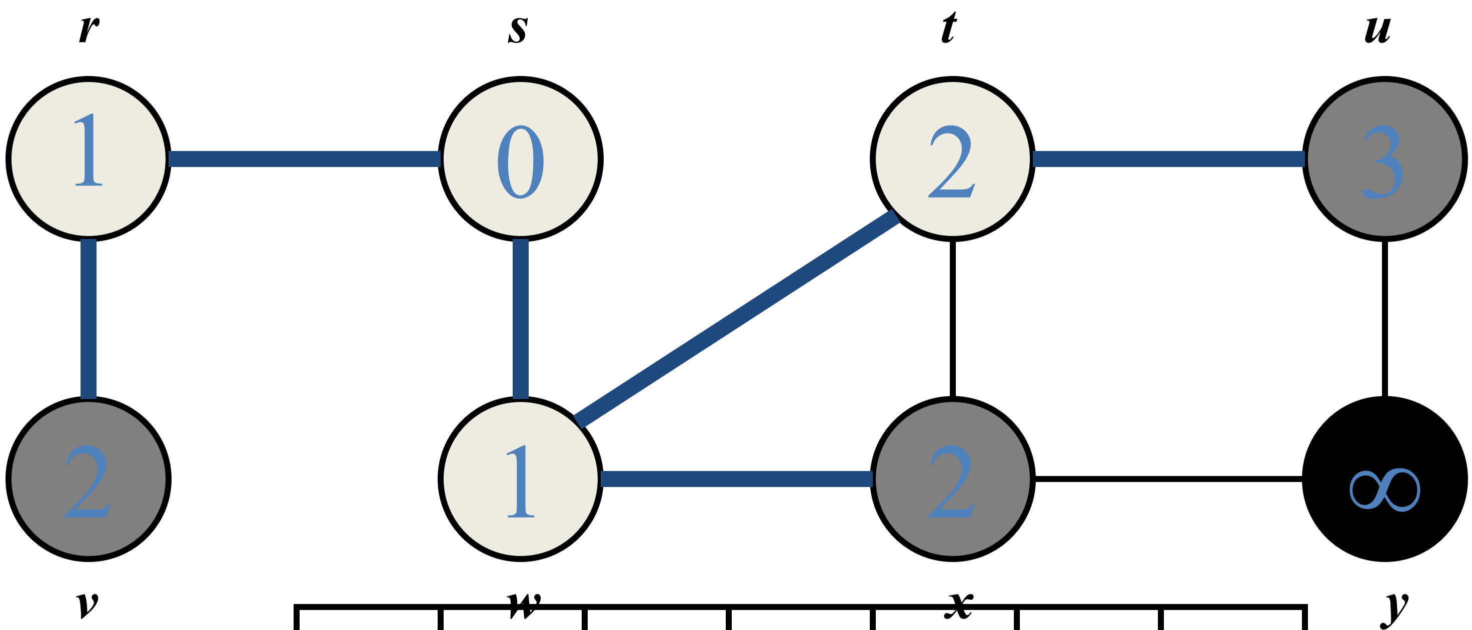
vertex	r	s	t	u	V	w	X	y
d	1	0	**2**	∞	∞	**1**	**2**	∞
par	s	nil	**w**	nil	nil	**s**	**w**	nil

Breadth-First Search: Example



vertex	r	s	t	u	v	w	x	y
d	**1**	0	2	∞	**2**	1	2	∞
par	**s**	nil	w	nil	r	s	w	nil

Breadth-First Search: Example

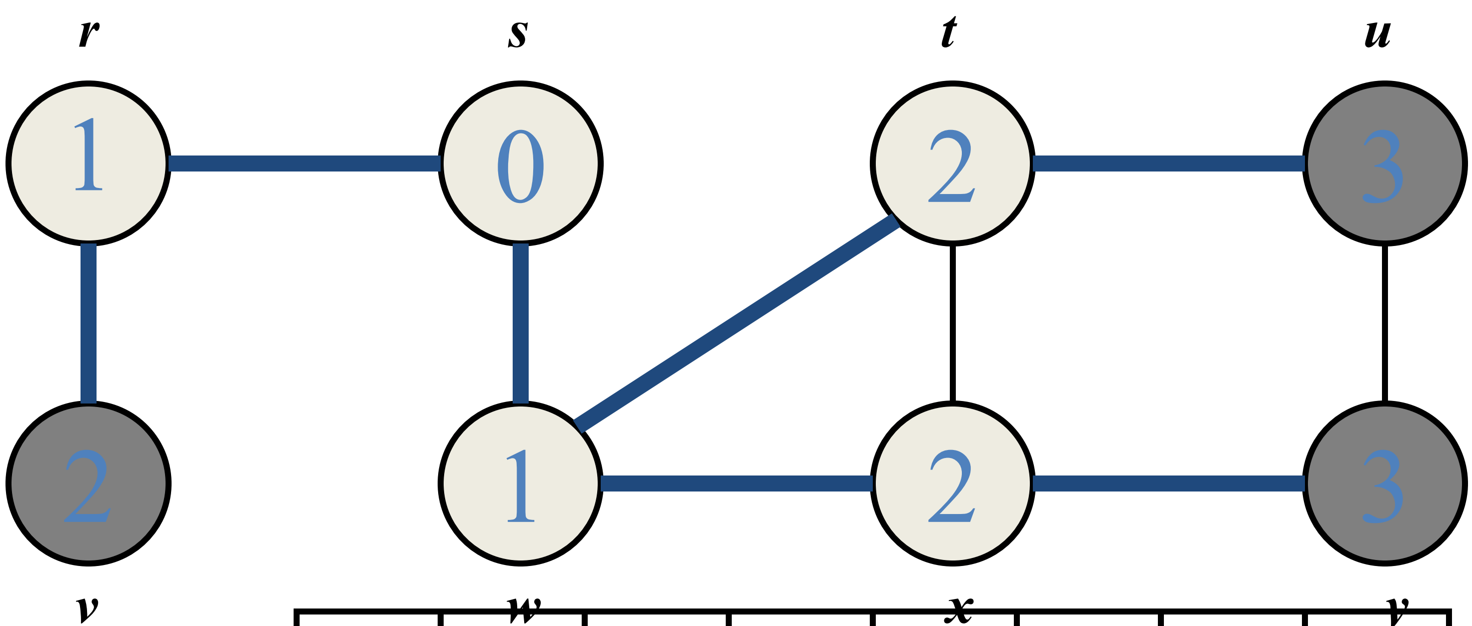


Q:

s	*w*	*r*	*t*	*x*	*v*	*u*

vertex	r	s	t	u	v	w	x	y
d	1	0	2	3	2	1	2	∞
prev	s	nil	w	t	r	s	w	nil

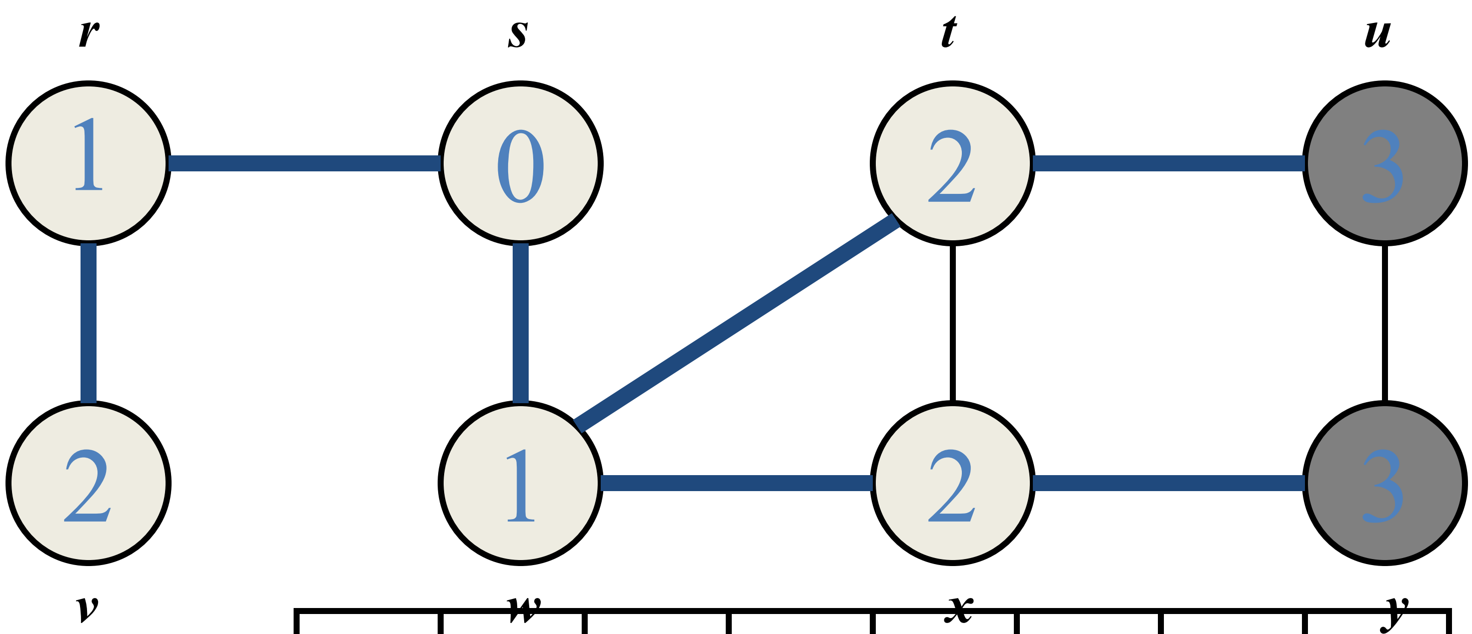
Breadth-First Search: Example



Q:							
s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
d	1	0	2	3	2	1	**2**	3
prev	s	nil	w	t	r	s	**w**	x

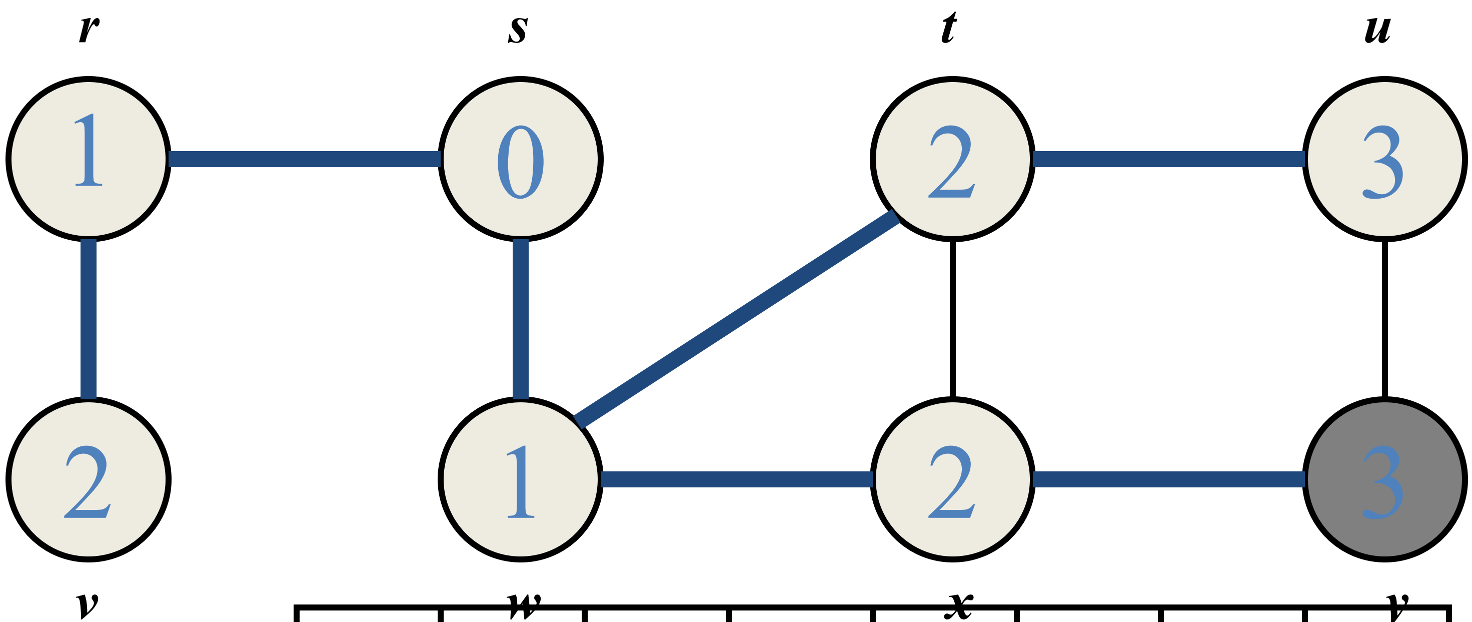
Breadth-First Search: Example



Q:							
s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
d	1	0	2	3	**2**	1	2	3
prev	s	nil	w	t	**r**	s	w	x

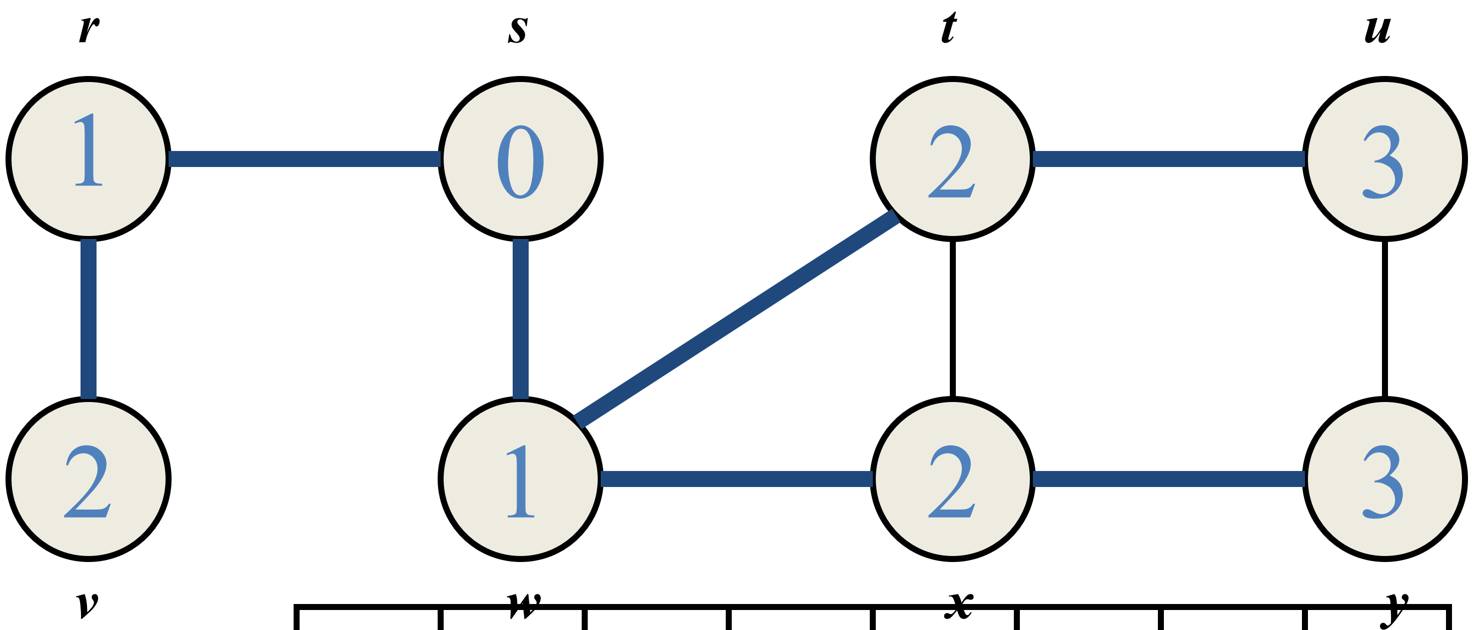
Breadth-First Search: Example



Q:							
s	*w*	*r*	*t*	*x*	*v*	*u*	*y*

vertex	r	s	t	u	v	w	x	y
d	1	0	2	**3**	2	1	2	3
prev	s	nil	w	**t**	r	s	w	x

Breadth-First Search: Example



Q:

s	w	r	t	x	v	u	y

vertex	r	s	t	u	v	w	x	y
d	1	0	2	3	2	1	2	3
prev	s	nil	w	t	r	s	w	x

Breadth-First Search: Printing the Shortest Path

```
void path_print(int curr, int s){
    if(curr!=s){
        path_print(par[curr], s);
    }
    printf("%d\n", curr);
}
```

BFS: Complexity

- Every node is dequeued once [Complexity: O(V)]
- Every node is enqueued once [Complexity: O(V)]
- Every edge is considered once (twice in undirected graphs) [Complexity: O(E)]
- Total Complexity: O(V+E)

Breadth-First Search: Properties

- BFS calculates the *shortest-path distance* to the source node
  - Shortest-path distance $\delta(s,v)$ = minimum number of edges from s to v, or ∞ if v not reachable from s
  - Proof given in the book (Cormen et. al. p. 472-5)
- BFS builds *breadth-first tree*, in which paths to root represent shortest paths in G
  - Thus can use BFS to calculate shortest path from one vertex to another in O(V+E) time

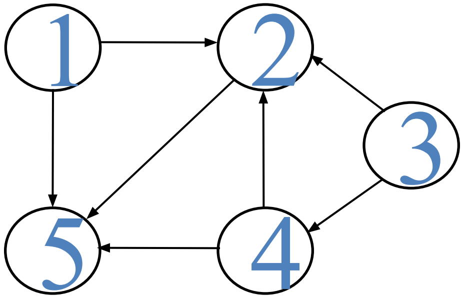
Application of BFS

- Find the shortest path in an undirected/directed unweighted graph.
- Find the bipartiteness of a graph.
- Find the connectedness of a graph.

BFS – Questions

- Find the shortest path between “A” and “B” (with path)? When will it fail?
- Find the most distant node from start node “A”
- How can we detect that there exists no path between A and B using BFS?
- Print all of those nodes that are at distance 2 from source vertex “S”.
- How can we modify BFS algorithm to check the bipartiteness of a graph?
- Is it possible to answer that there exists more than one path from “S” to “T” with minimum path cost?

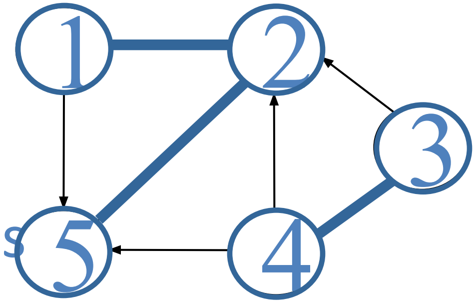
Depth-First Search



- **Input:**
  - G = (V, E) (No source vertex given!)
- **Goal**:
  - Explore the edges of G to "discover" every vertex in V starting at the most current visited node
  - Search may be repeated from multiple sources
- **Output:**
  - 2 **timestamps** on each vertex:
    - d[v] = discovery time
    - f[v] = finishing time (done with examining v's adjacency list)
  - Depth-first forest

Depth-First Search

- Search "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex v that still has unexplored edges



- After all edges of v have been explored, the search "backtracks" from the parent of v
- The process continues until all vertices reachable from the original source have been discovered
- If undiscovered vertices remain, choose one of them as a new source and repeat the search from that vertex
- DFS creates a "depth-first forest"

DFS Additional Data Structures

- Global variable: time-stamp
  - Incremented when nodes are discovered or finished
- color[u]
  - White before discovery, gray while processing and black when finished processing
- prev[u] – predecessor of u
- d[u], f[u] – discovery and finish times

Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{                                    Initialize
   for each vertex u ∈ V
   {
      color[u] = WHITE;
       prev[u]=NIL;
       f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if(color[v] == WHITE){
         prev[v]=u;
         DFS_Visit(v);}
    }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

Depth-First Search: The Code

```
Data: color[V], time,
   prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
      prev[u]=NIL;
      f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if(color[v] == WHITE){
         prev[v]=u;
         DFS_Visit(v);}
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

What does `d[u]` represent?

Depth-First Search: The Code

```
Data: color[V], time,
   prev[V],d[V], f[V]
DFS(G) // where prog starts
{
    for each vertex u ∈ V
    {
        color[u] = WHITE;
         prev[u]=NIL;
         f[u]=inf; d[u]=inf;
    }
    time = 0;
    for each vertex u ∈ V
      if (color[u] == WHITE)
          DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
        if(color[v] == WHITE){
            prev[v]=u;
            DFS_Visit(v);}
    }
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

What does `f[u]` represent?

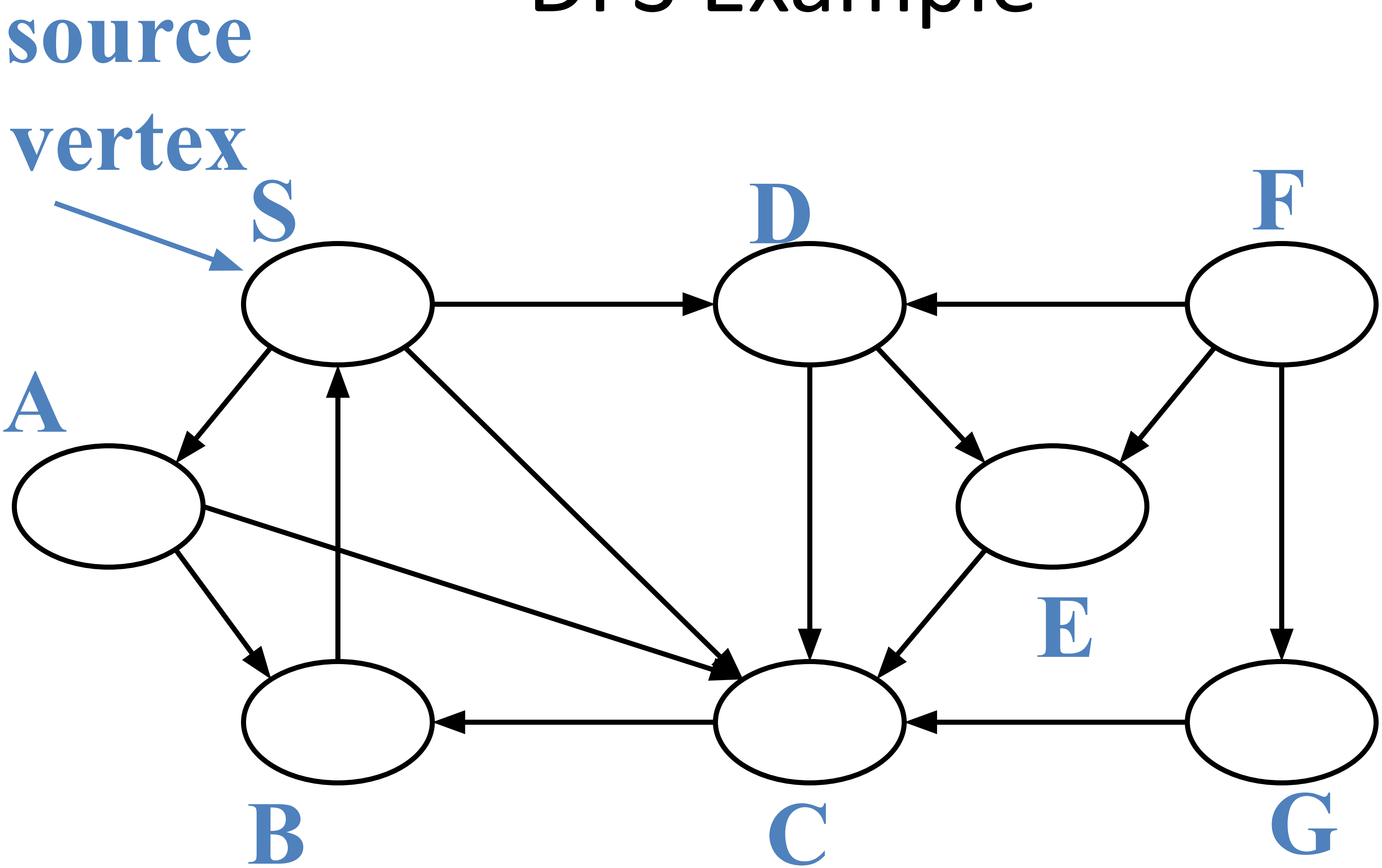
Depth-First Search: The Code

```
Data: color[V], time,
   prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
      prev[u]=NIL;
      f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
        DFS_Visit(u);
}
```

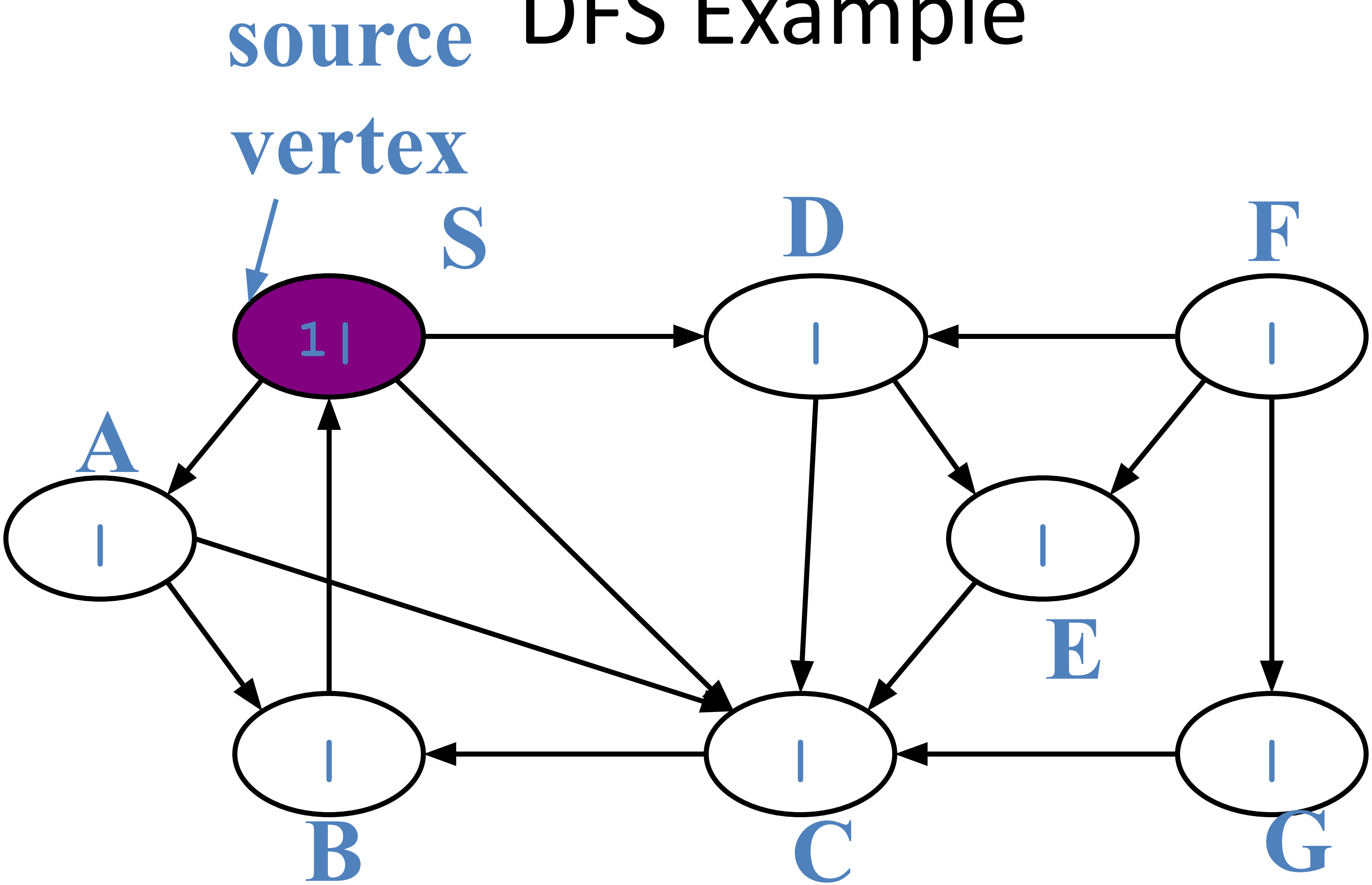
```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if(color[v] == WHITE){
         prev[v]=u;
         DFS_Visit(v);
   }}
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

Will all vertices eventually be colored black?

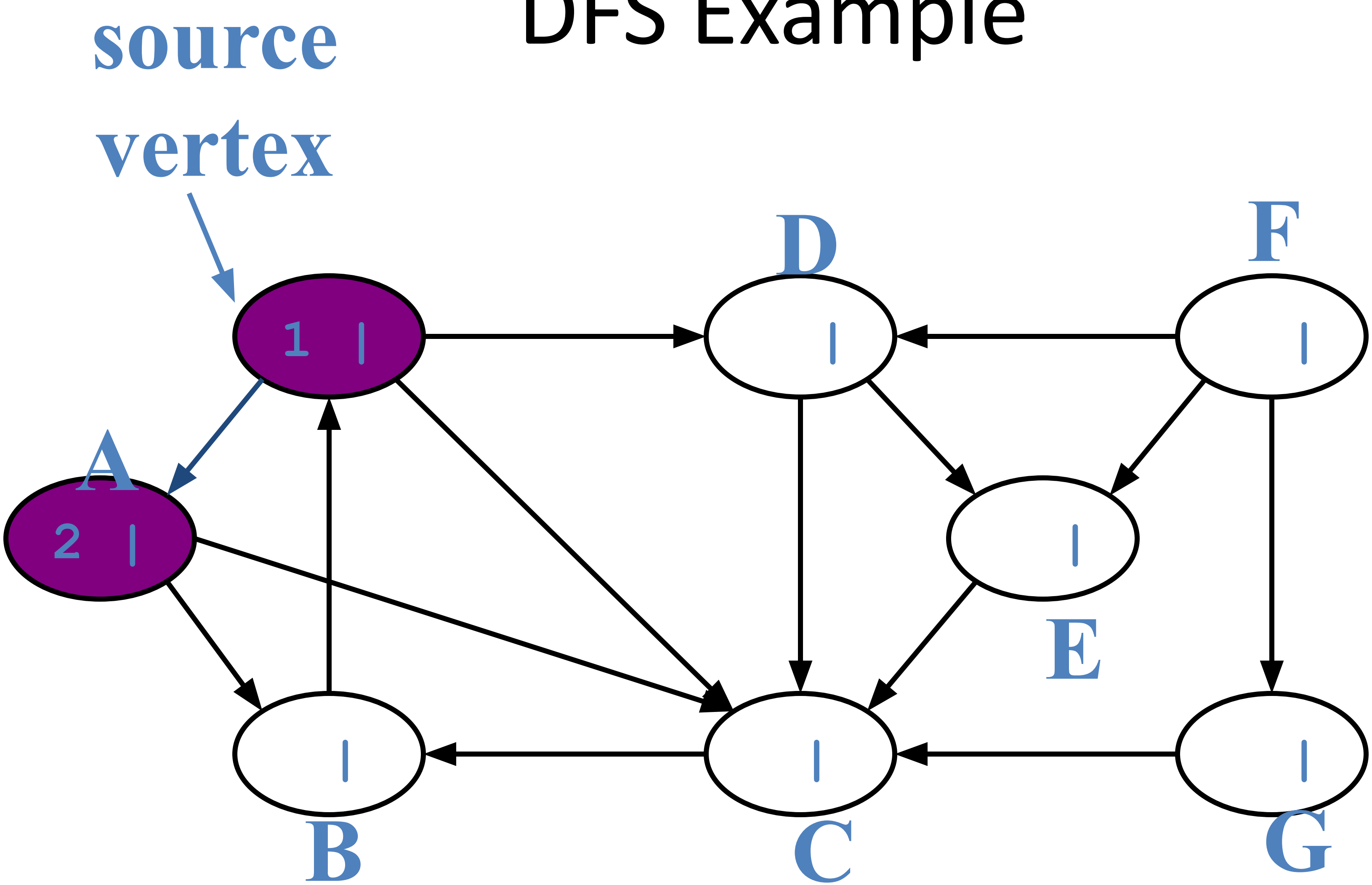
DFS Example



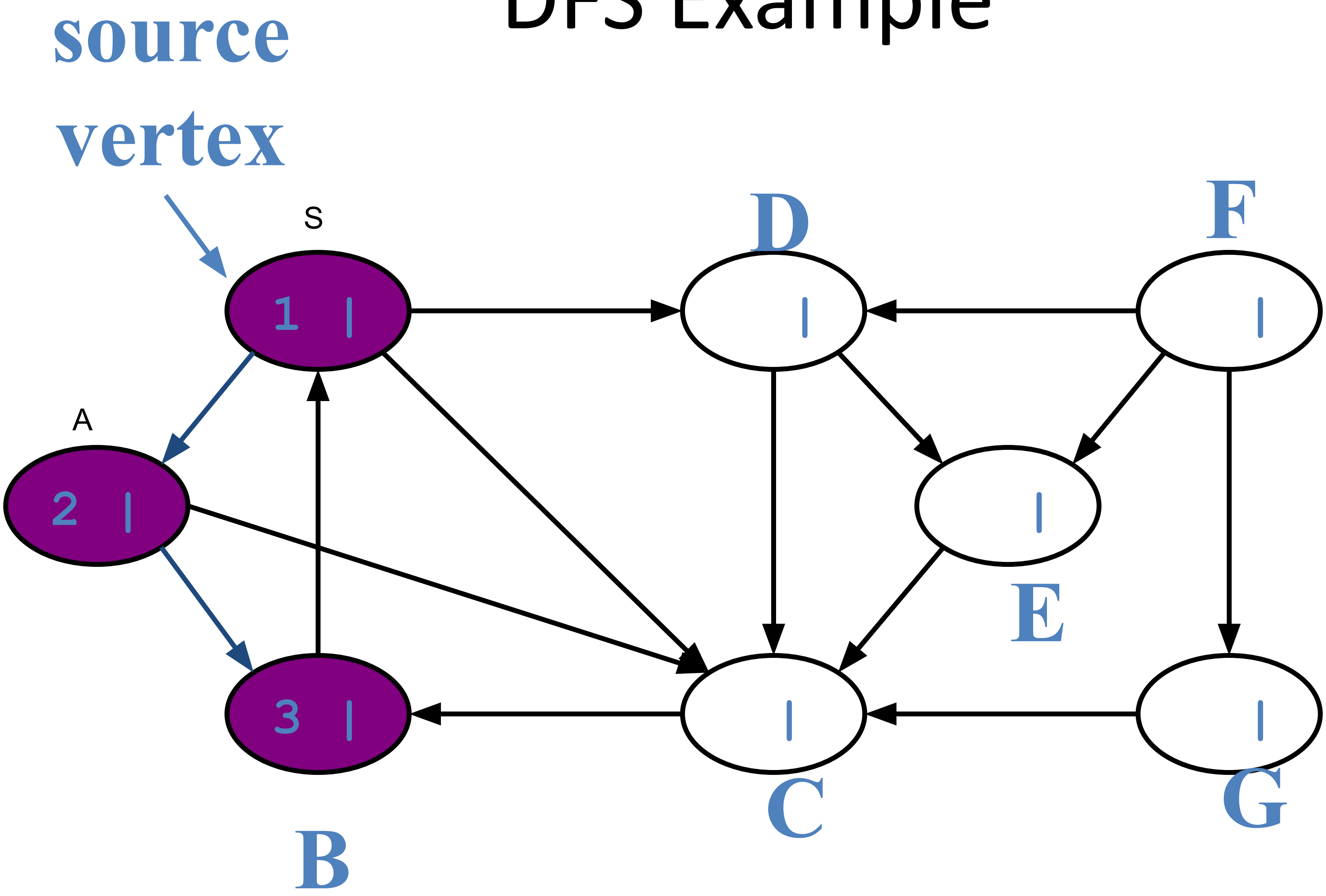
DFS Example



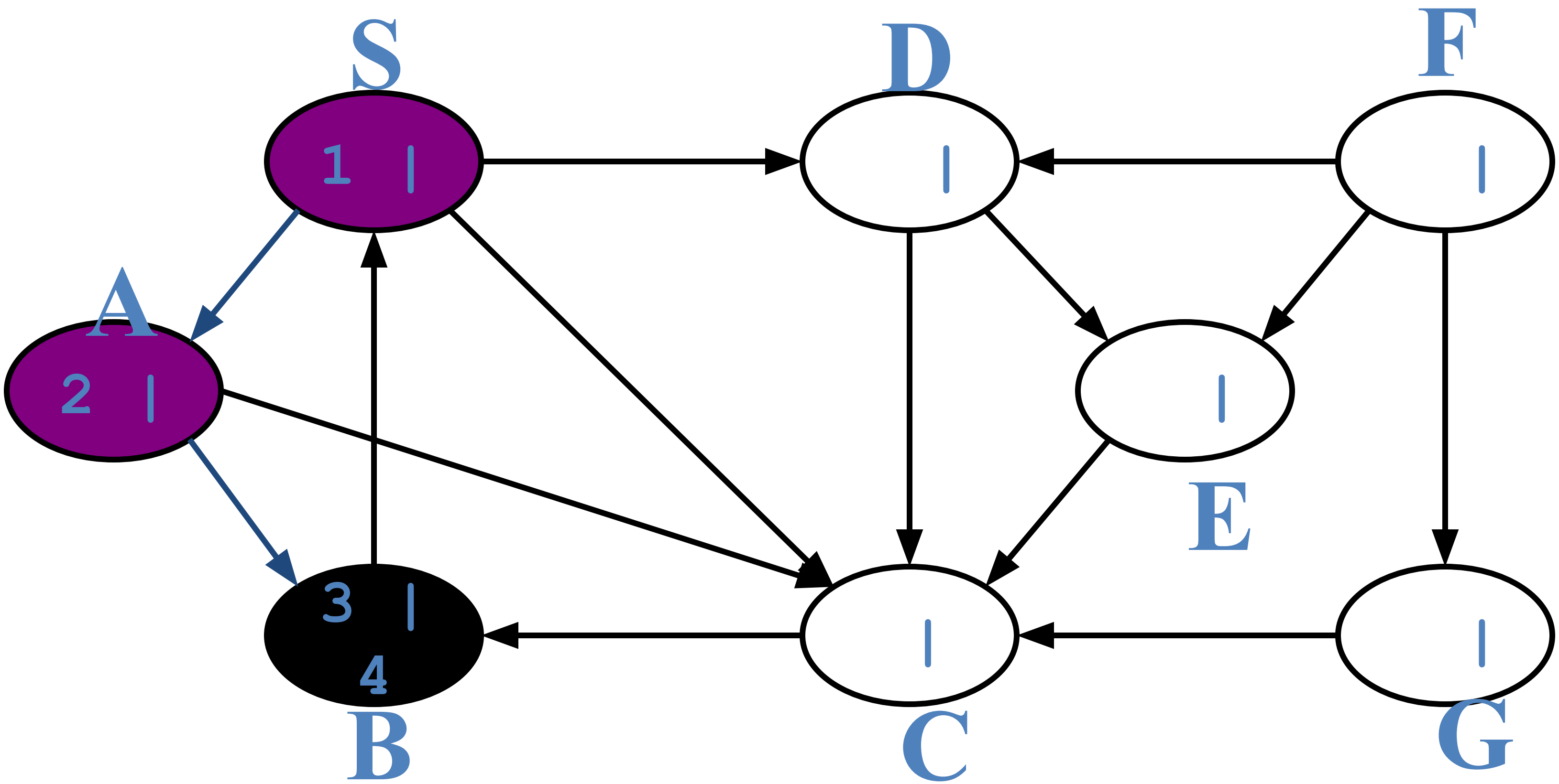
DFS Example



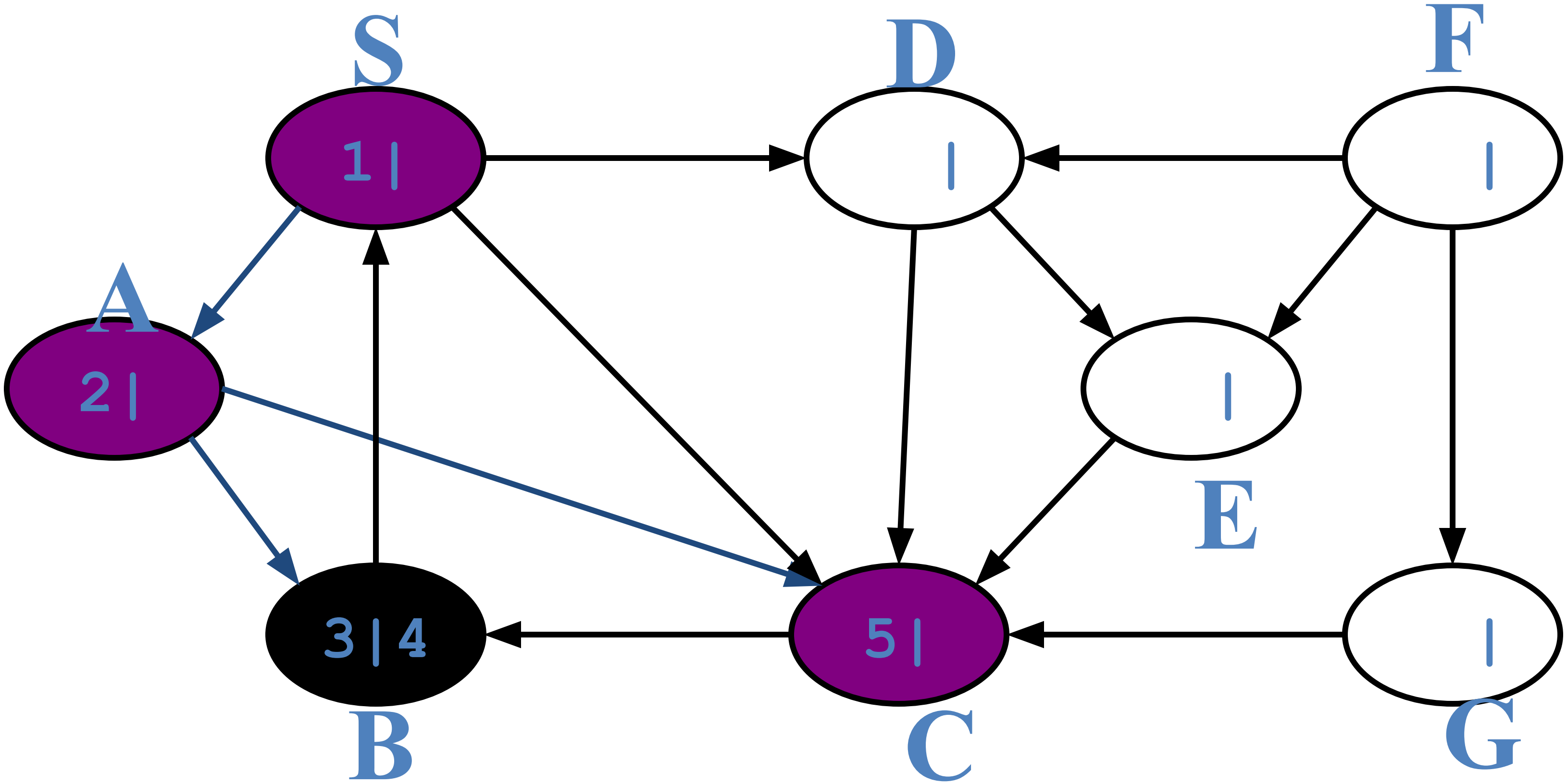
DFS Example



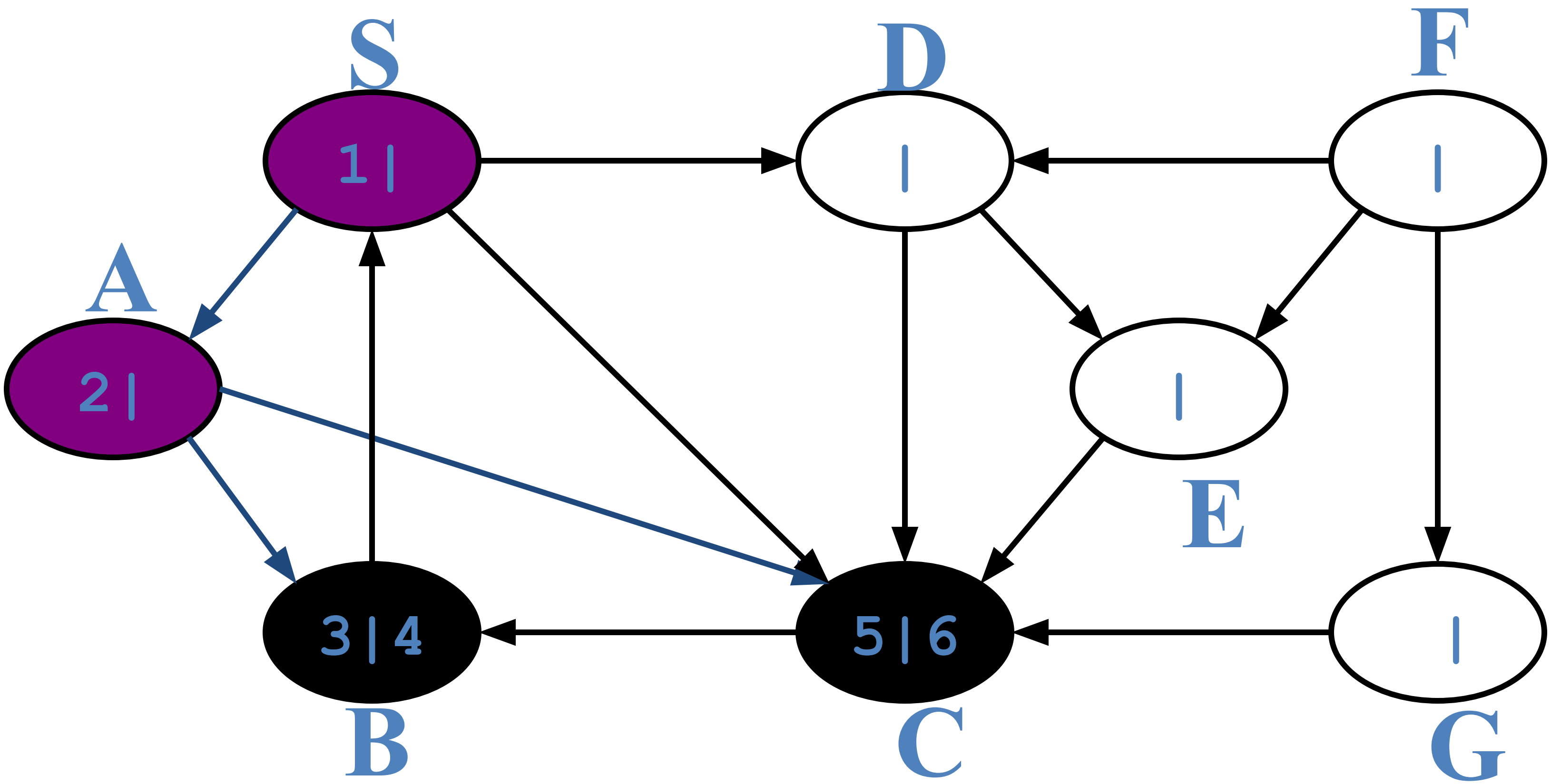
DFS Example



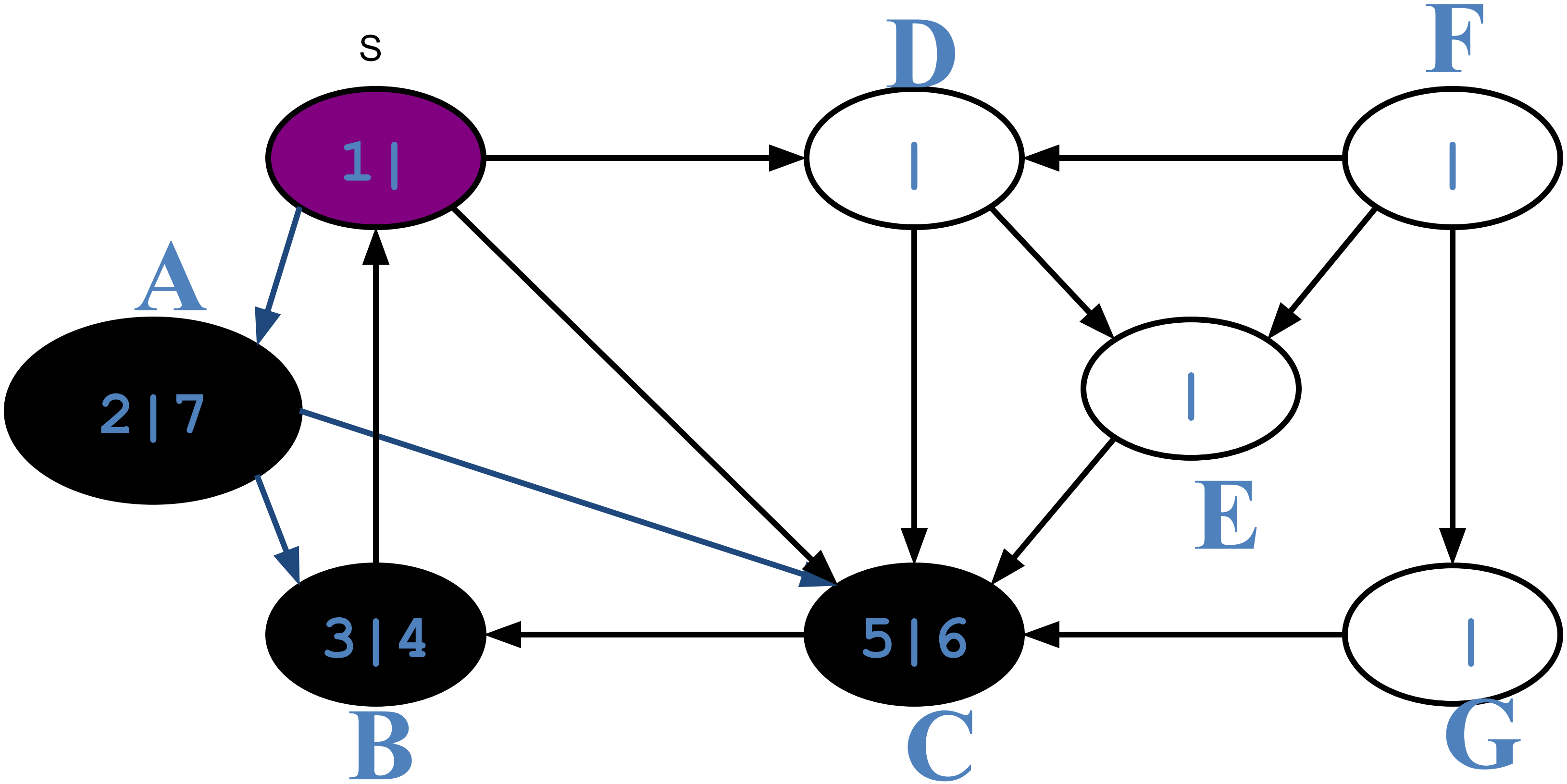
DFS Example



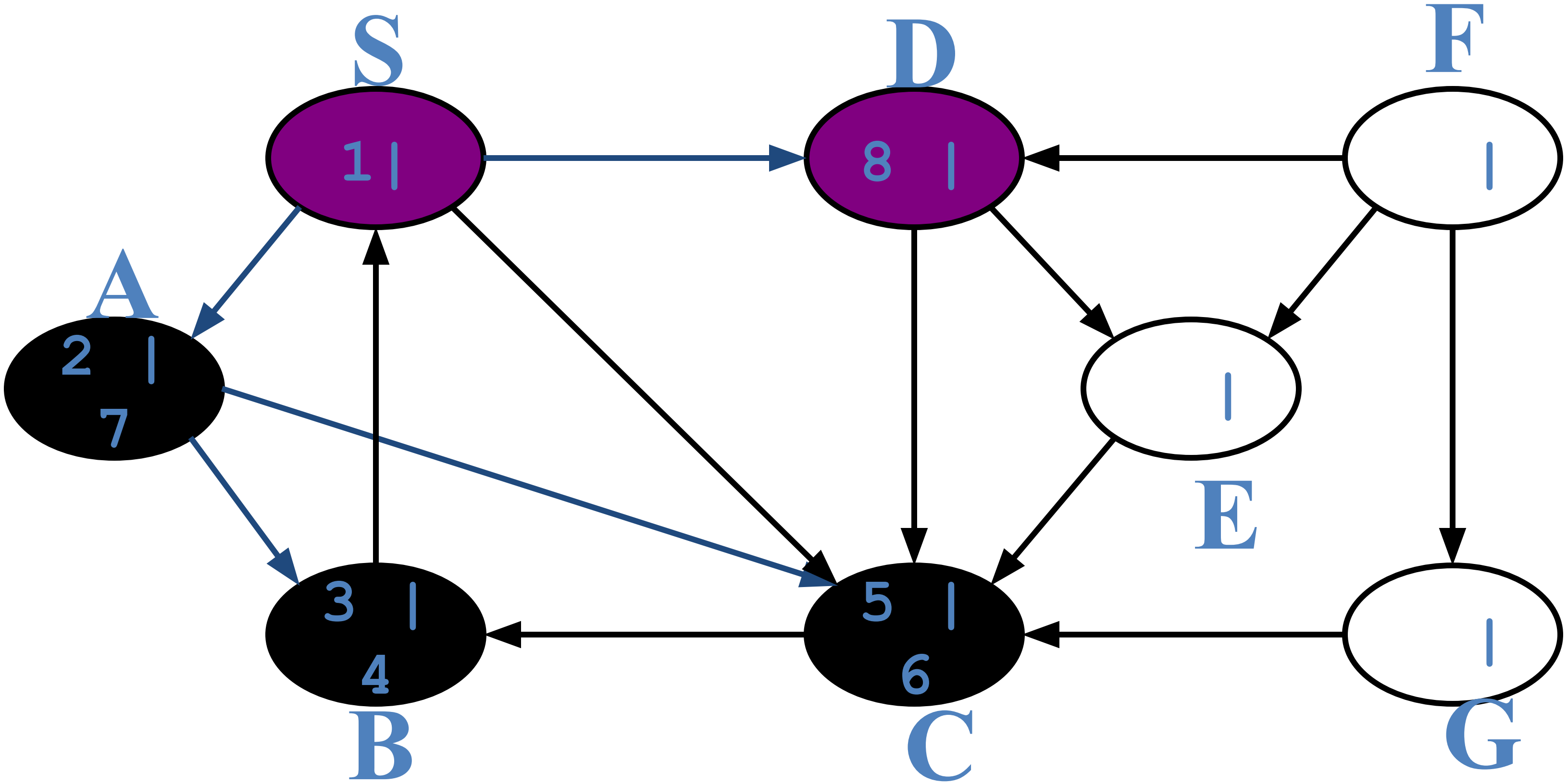
DFS Example



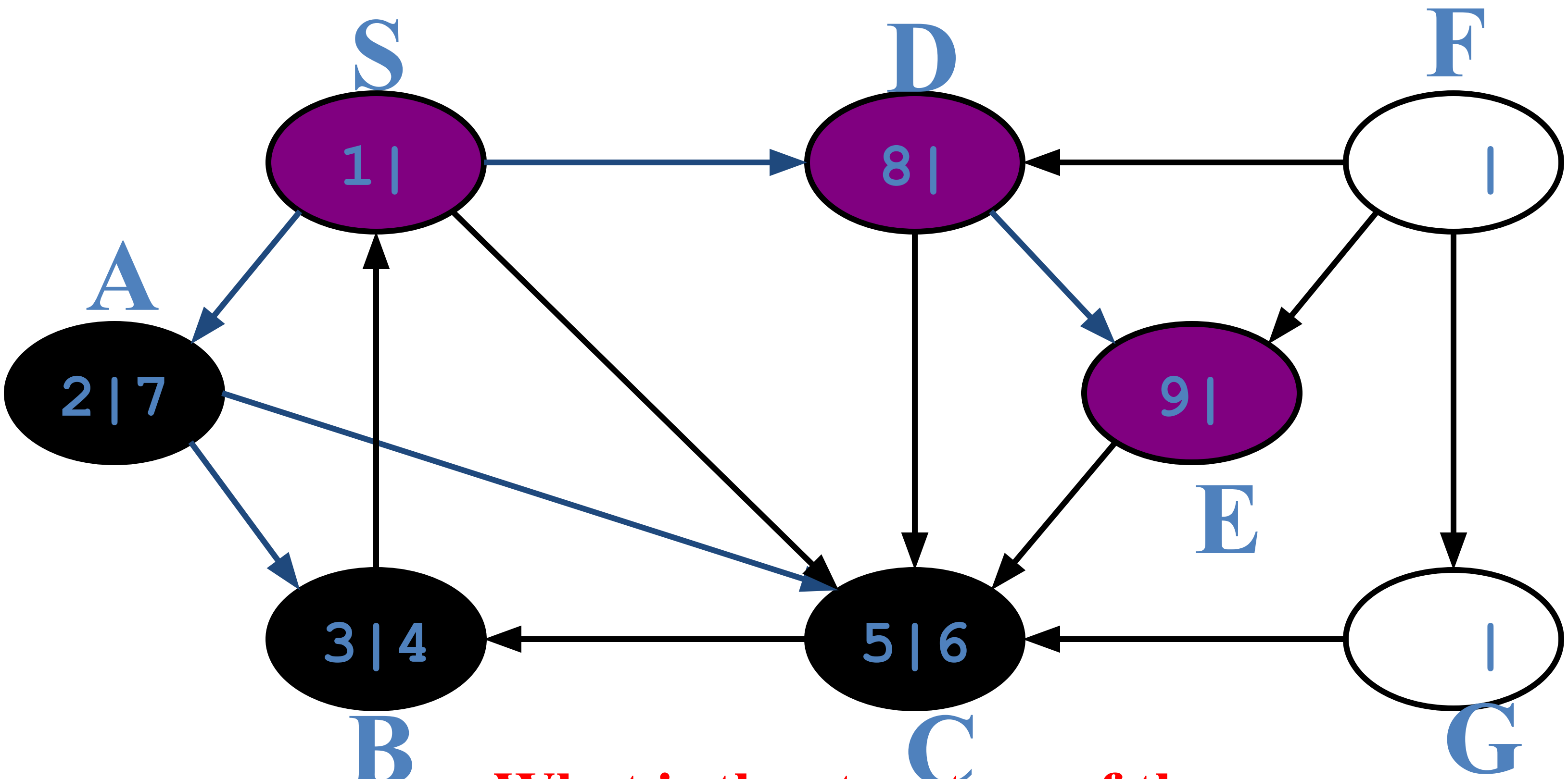
DFS Example



DFS Example



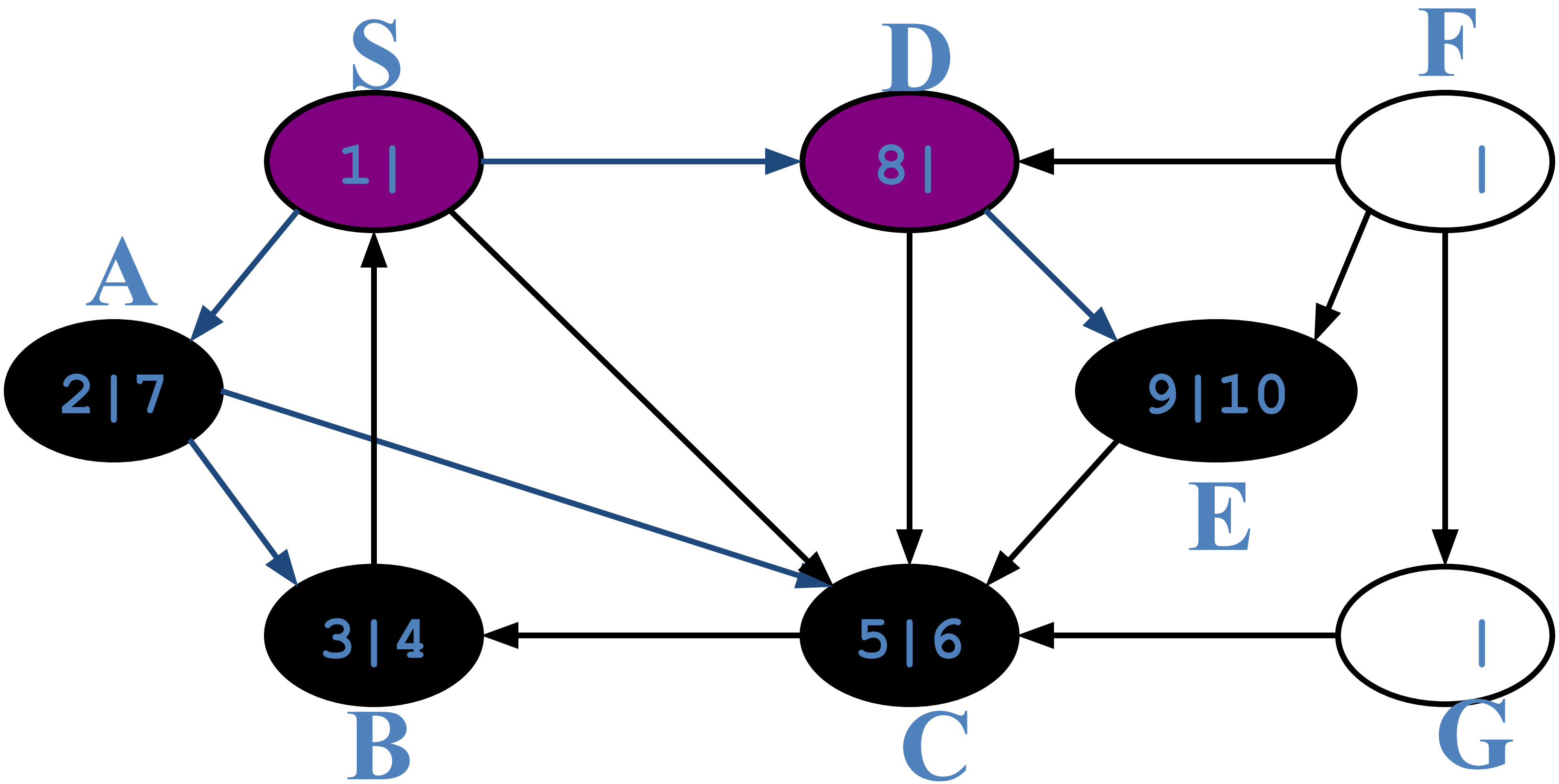
DFS Example



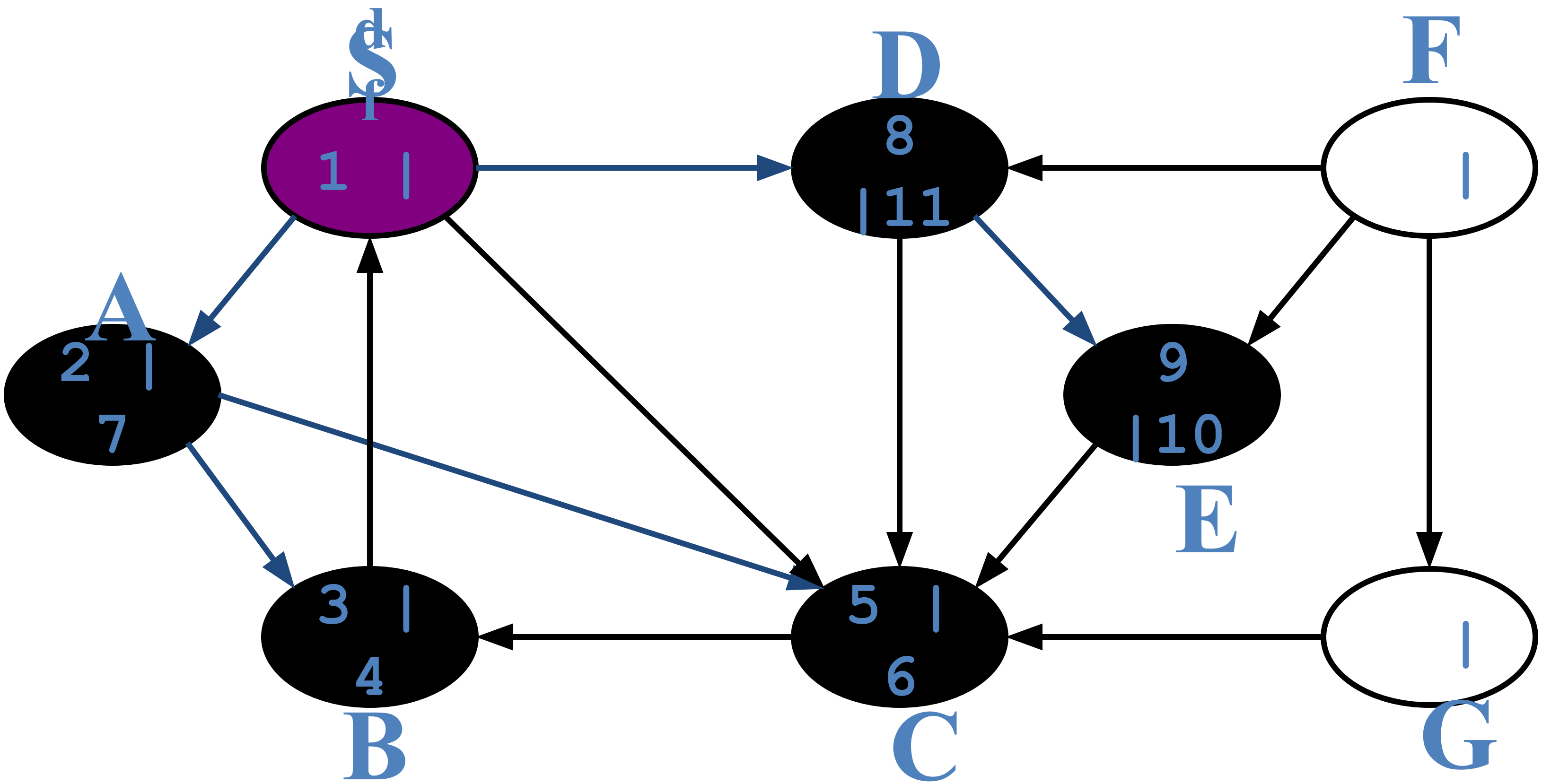
What is the structure of the grey vertices?

What do they represent?

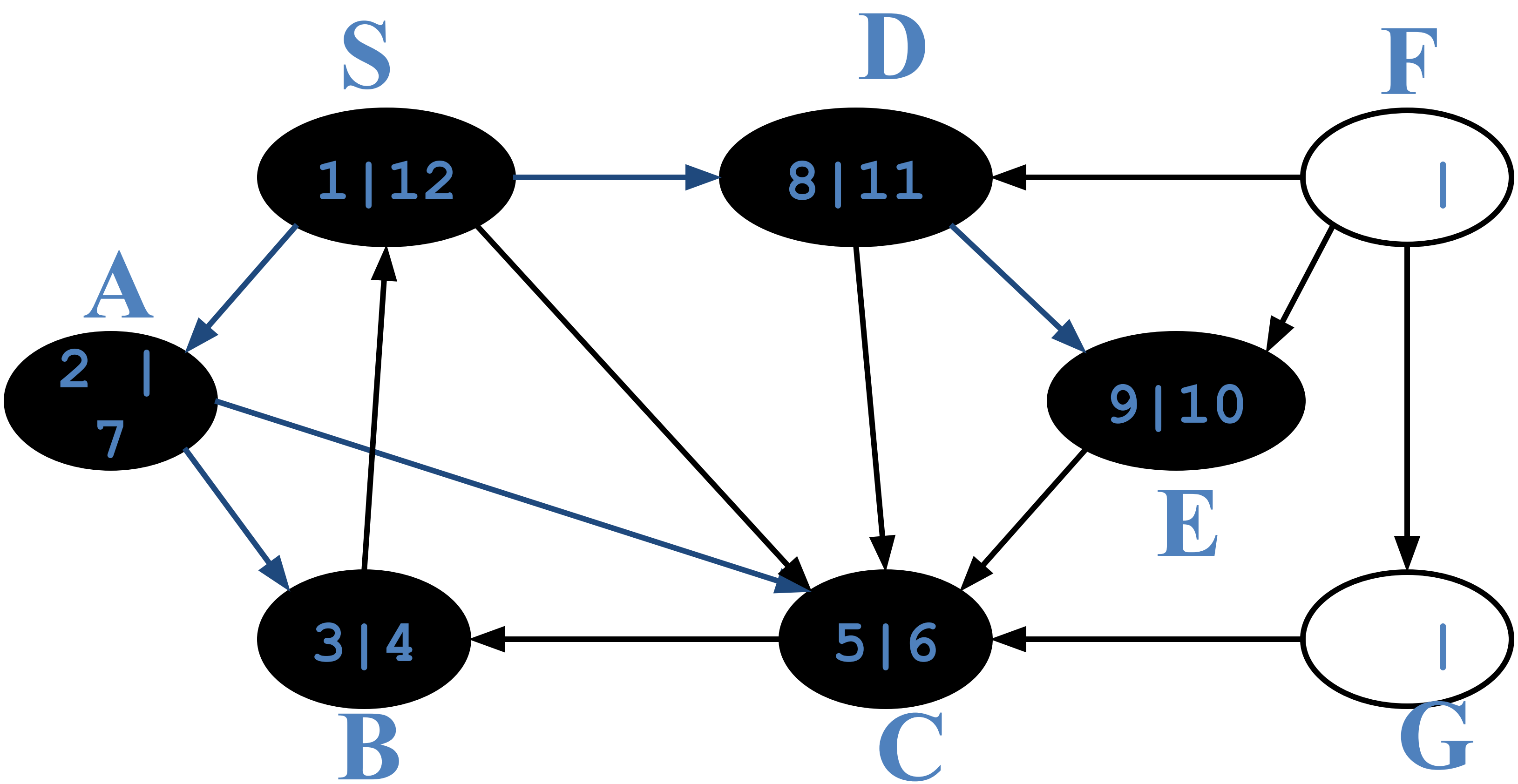
DFS Example



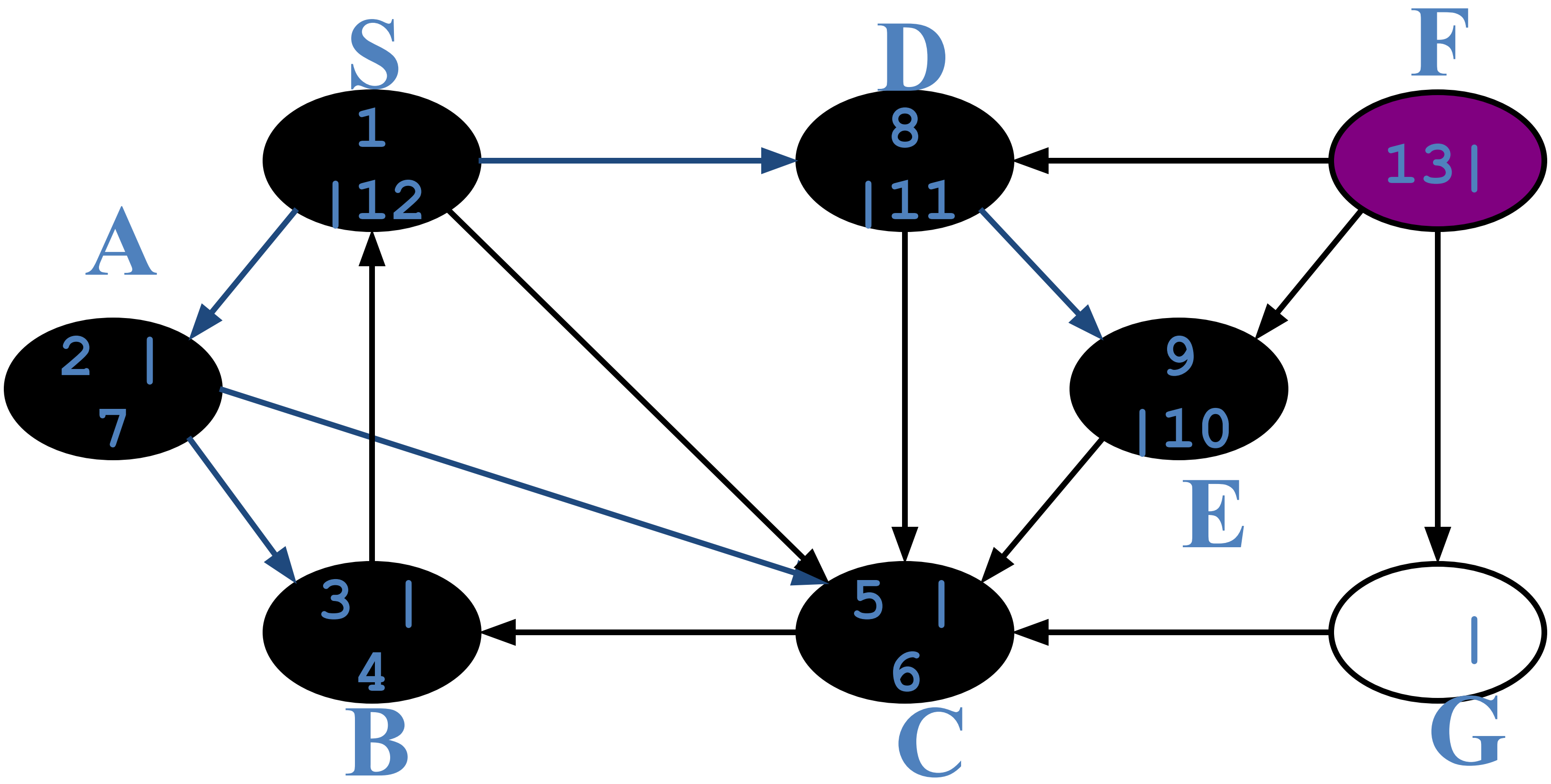
DFS Example



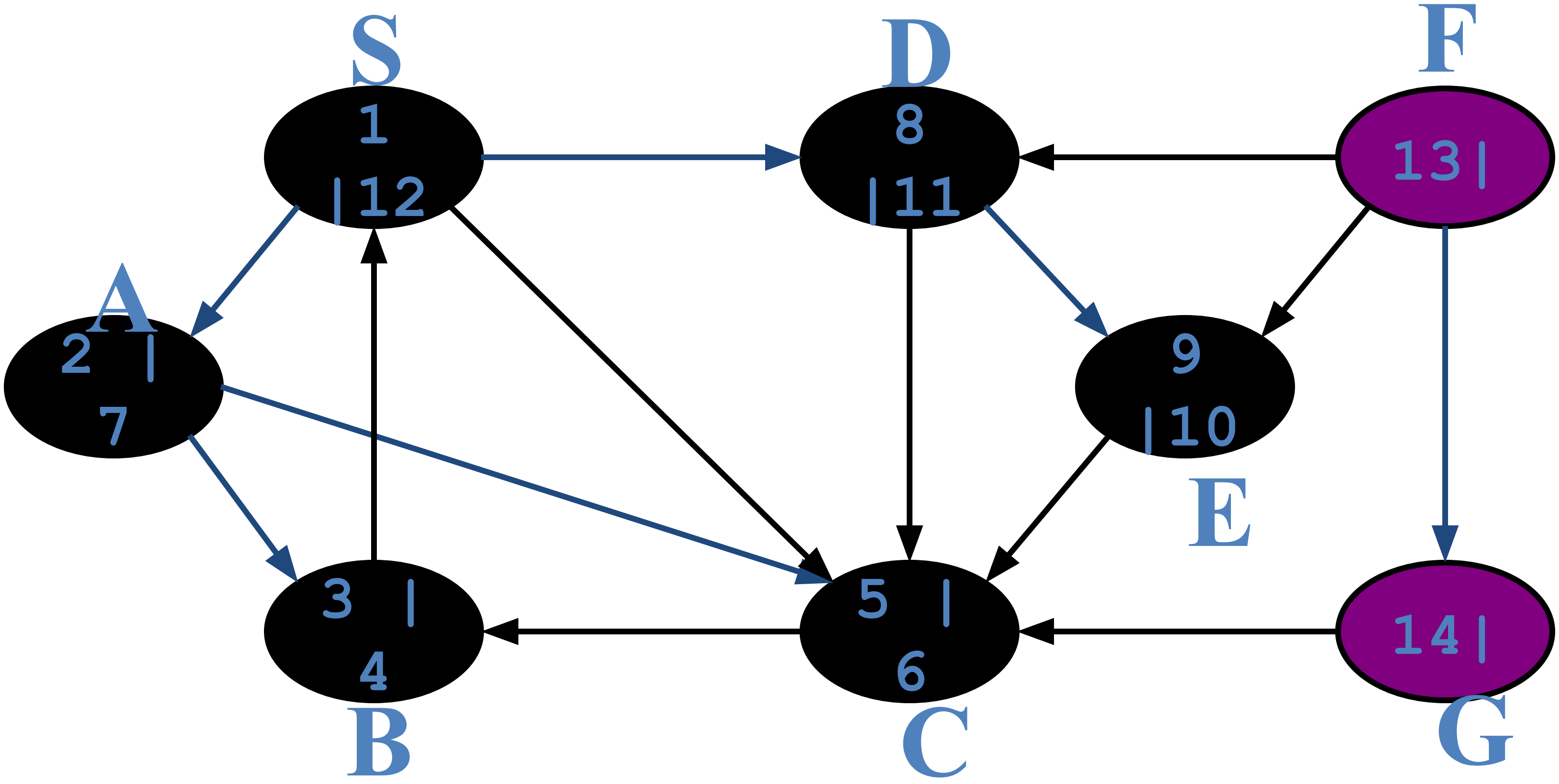
DFS Example



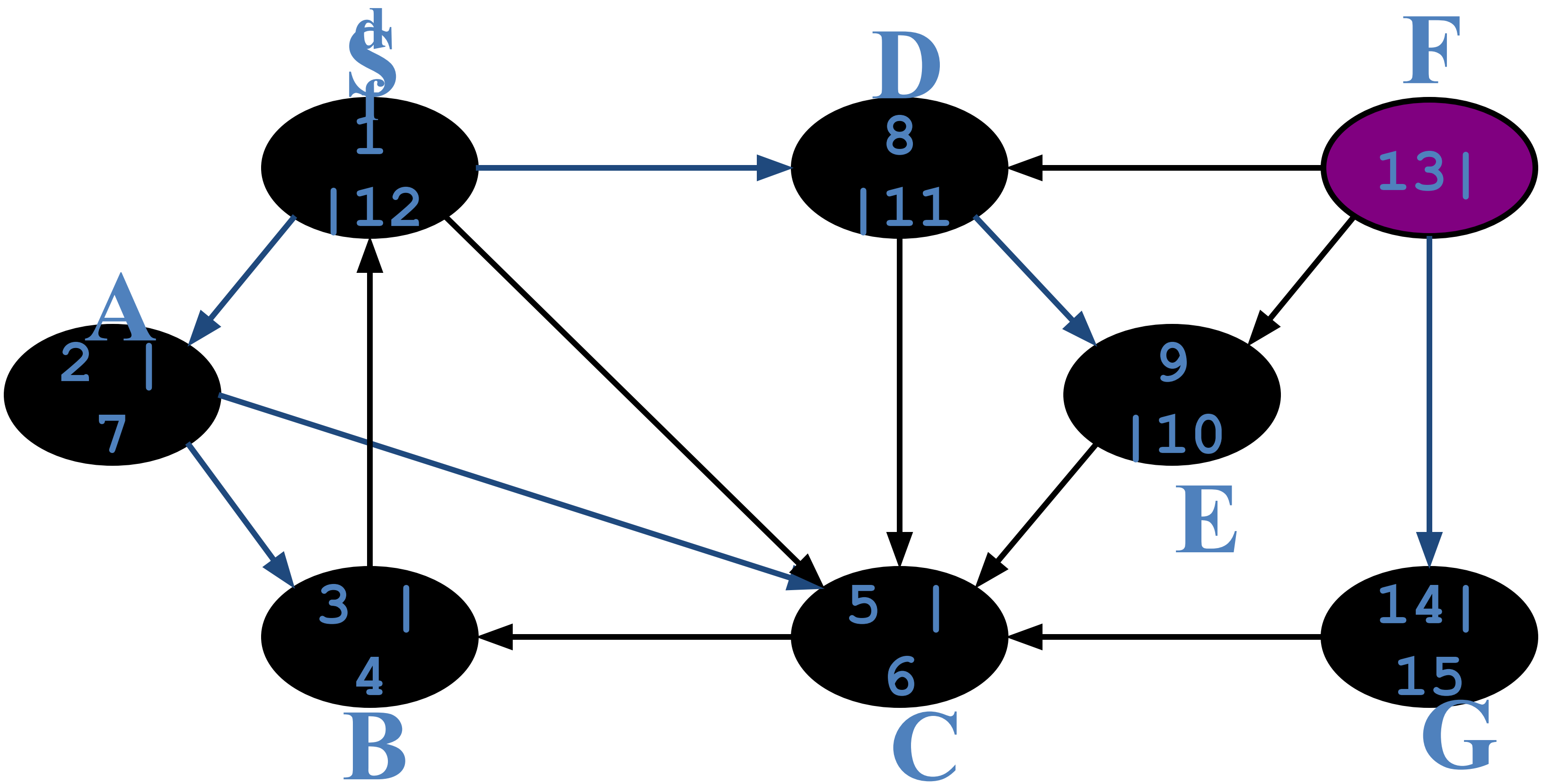
DFS Example



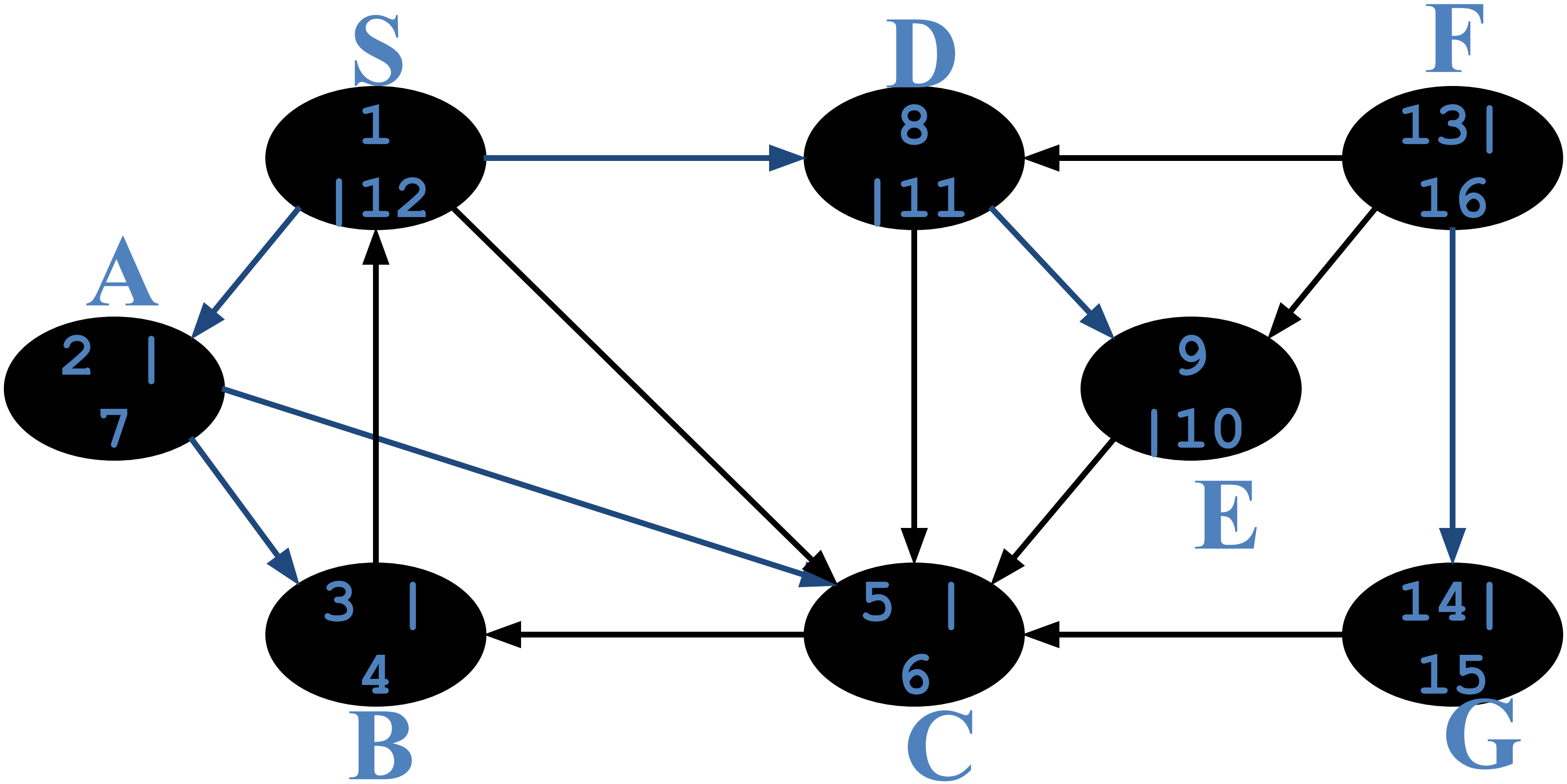
DFS Example



DFS Example



DFS Example



Depth-First Search: The Code

```
Data: color[V], time,
   prev[V],d[V], f[V]
DFS(G)  // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
       prev[u]=NIL;
       f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
      if (color[v] == WHITE)
          prev[v]=u;
          DFS_Visit(v);
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

What will be the running time?

Depth-First Search: The Code

```
Data: color[V], time,
  prev[V],d[V], f[V]
DFS(G) // where prog starts
{
    for each vertex u ∈ V
    {
        color[u] = WHITE;
         prev[u]=NIL;
         f[u]=inf; d[u]=inf;
    }
    time = 0;
    for each vertex u ∈ V
      if (color[u] == WHITE)
          DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
        if (color[v] == WHITE)
            prev[v]=u;
            DFS_Visit(v);
    }
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

Running time of DFS = O(V+E)

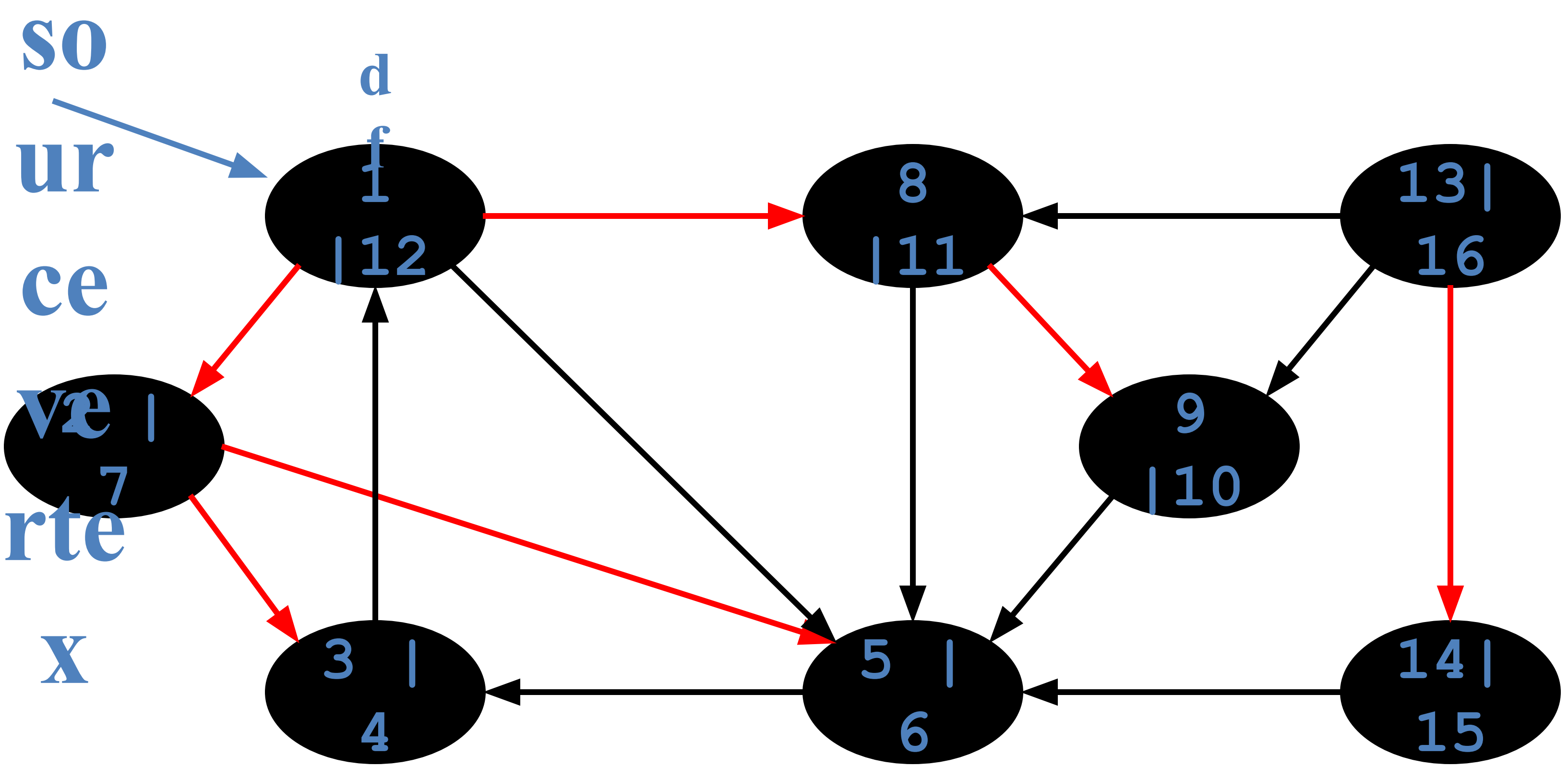
Depth-First Sort Analysis

- This running time argument is an informal example of *amortized analysis*
  - “Charge” the exploration of edge to the edge:
    - Each loop in DFS_Visit can be attributed to an edge in the graph
    - Runs once per edge if directed graph, twice if undirected
    - Thus loop will run in O(E) time, algorithm O(V+E)
      - Considered linear for graph, b/c adj list requires O(V+E) storage

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
    - The tree edges form a spanning forest
    - *Can tree edges form cycles? Why or why not?*
      - *No*

DFS Example

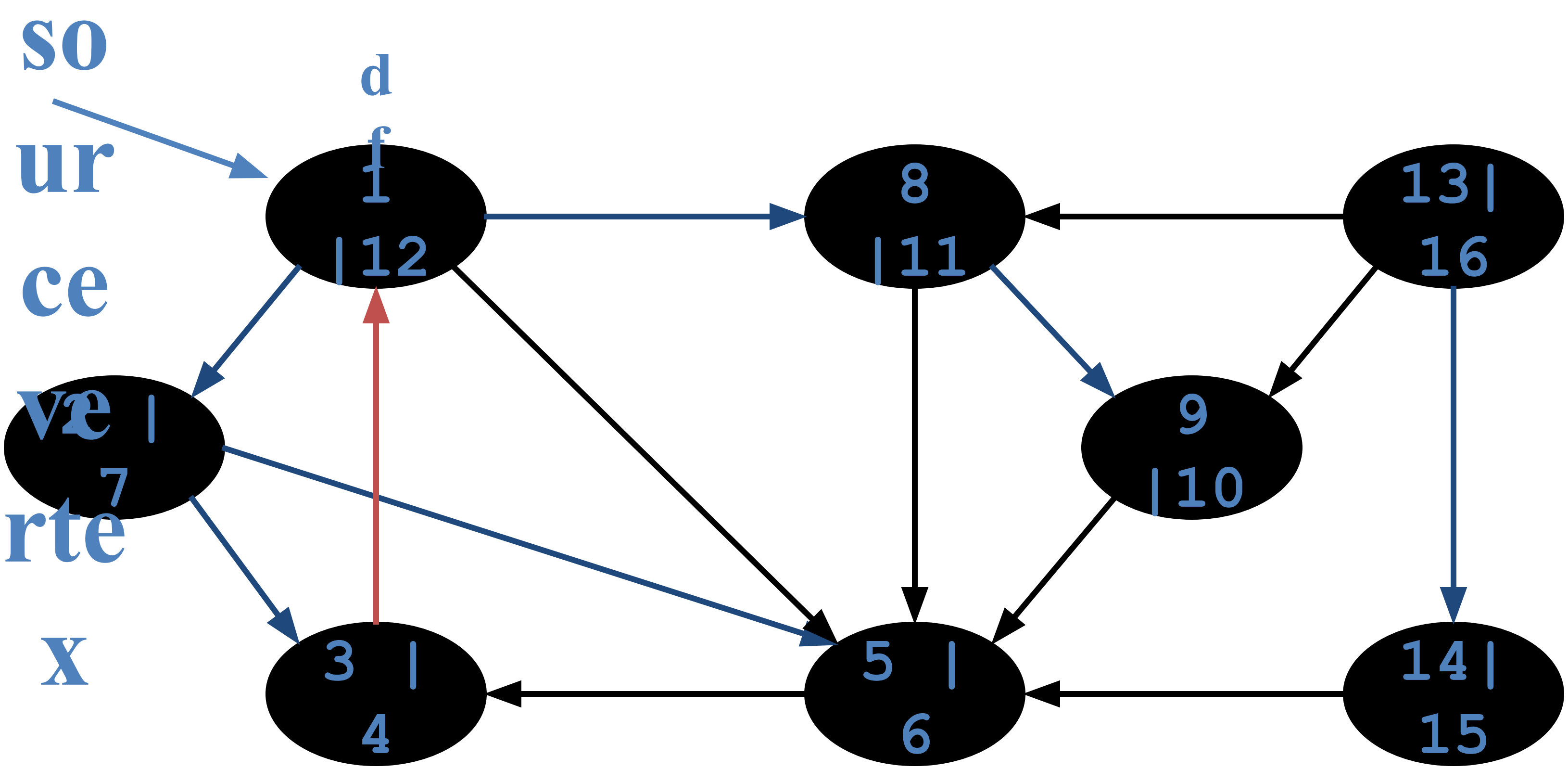


Tree edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
    - Encounter a grey vertex (grey to grey)
    - Self loops are considered as to be back edge.

DFS Example



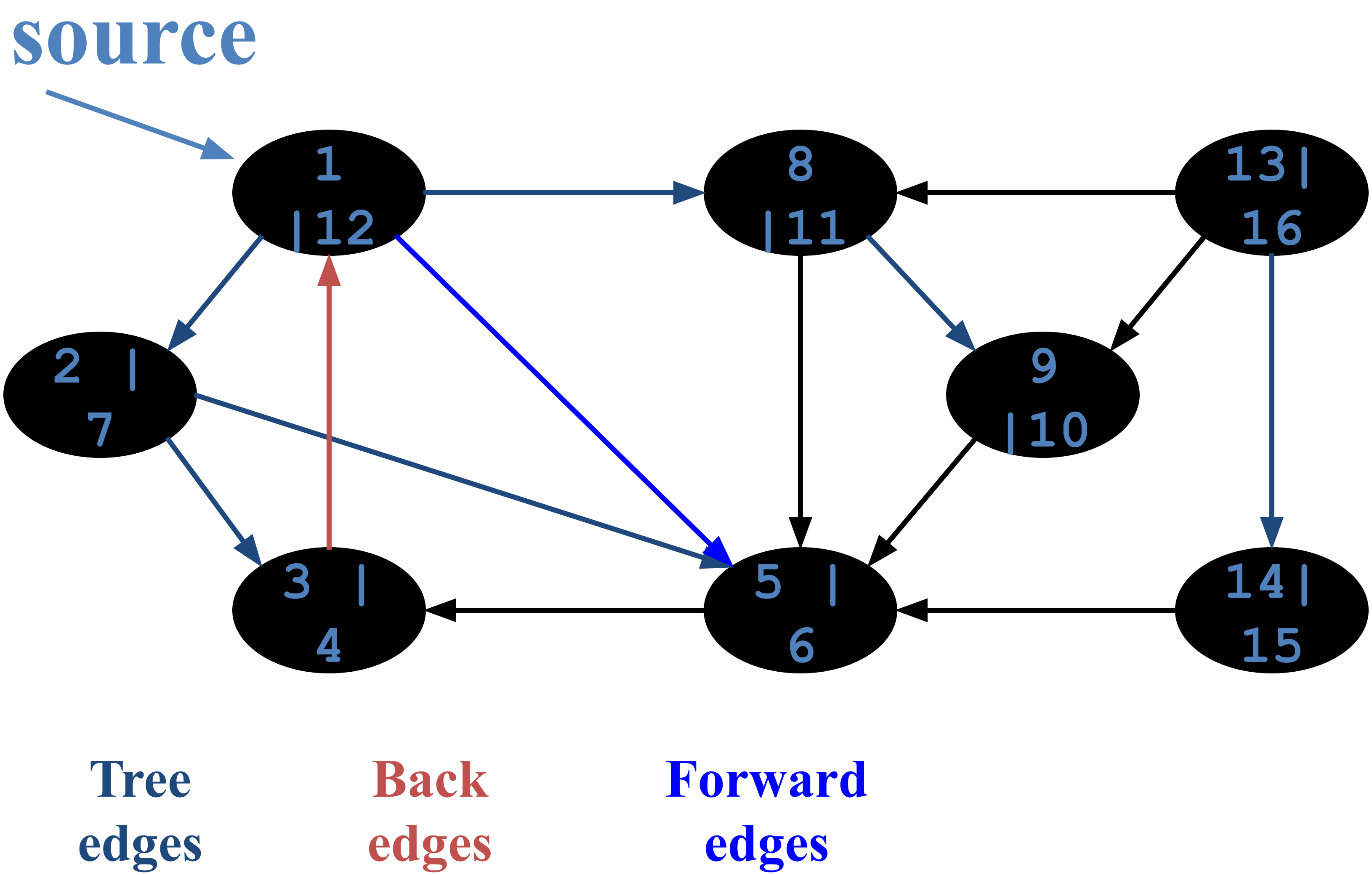
Tree edges

Back edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
    - Not a tree edge, though
    - From grey node to black node

DFS Example

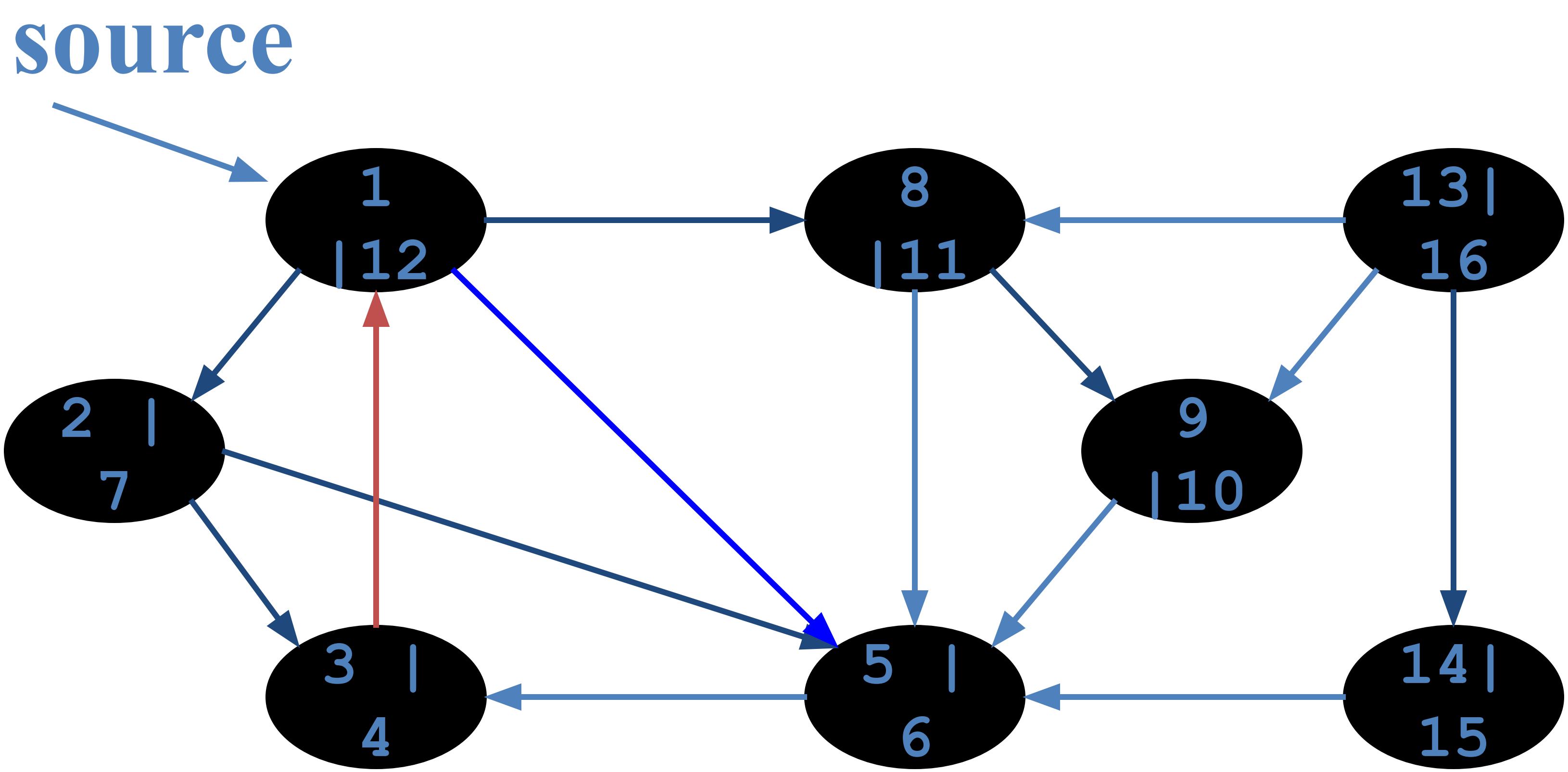


source
1
|12
8
|11
13|
16
2 |
7
9
|10
3 |
4
5 |
6
14|
15
Tree
edges
Back
edges
Forward
edges

DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  – *Tree edge*: encounter new (white) vertex
  – *Back edge*: from descendent to ancestor
  – *Forward edge*: from ancestor to descendent
  – *Cross edge*: between a tree or subtrees
    - From a grey node to a black node

DFS Example



Tree edges **Back edges** **Forward edges** **Cross edges**

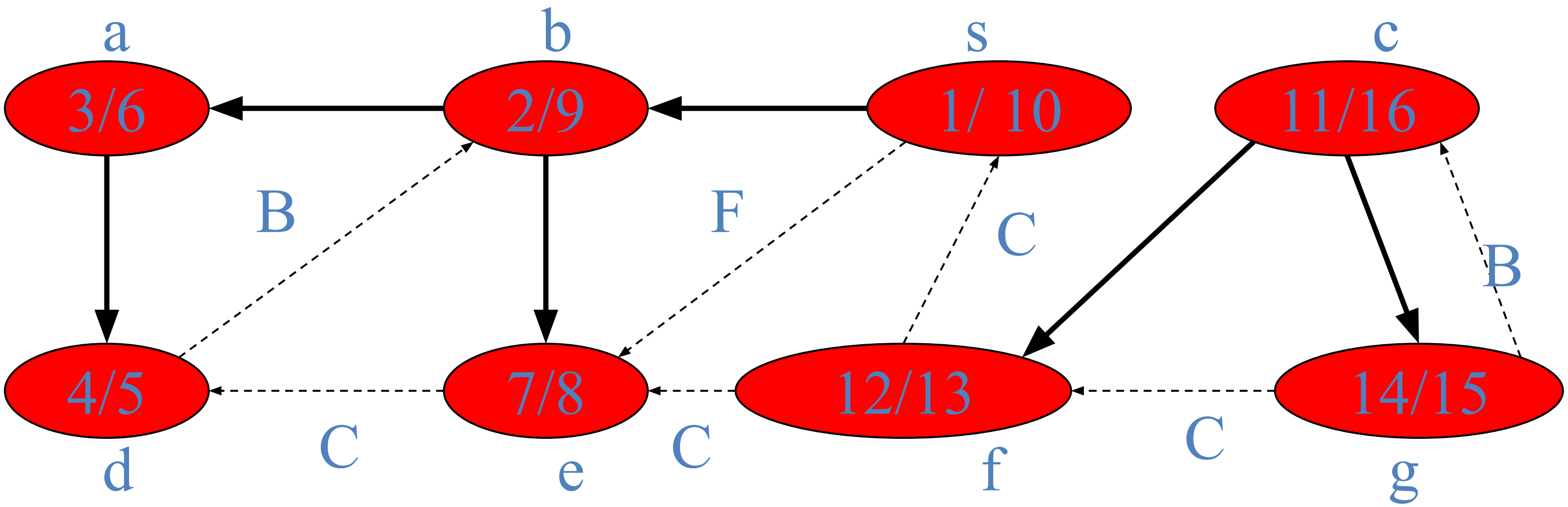
DFS: Kinds of edges

- DFS introduces an important distinction among edges in the original graph:
  - *Tree edge*: encounter new (white) vertex
  - *Back edge*: from descendent to ancestor
  - *Forward edge*: from ancestor to descendent
  - *Cross edge*: between a tree or subtrees
- Note: tree & back edges are important; most algorithms don't distinguish forward & cross

More about the edges

- Let (u,v) is an edge.
  - If (color[v] = WHITE) then (u,v) is a tree edge
  - If (color[v] = GRAY) then (u,v) is a back edge
  - If (color[v] = BLACK) then (u,v) is a forward/cross edge
    - Forward Edge: d[u]<d[v]
    - Cross Edge: d[u]>d[v]

Depth-First Search - Timestamps



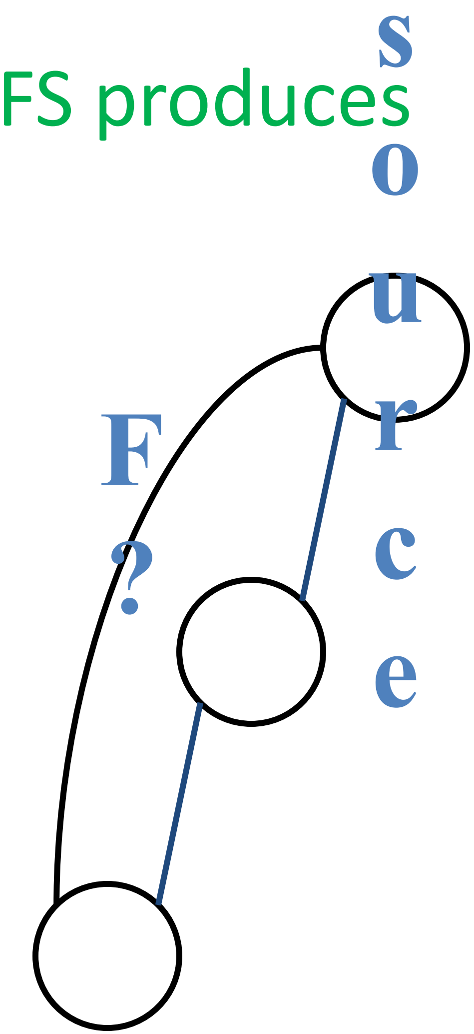
Depth-First Search: Detect Edge

```
Data: color[V], time,
   prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
       prev[u]=NIL;
       f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
        DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
     detect edge type using
    "color[v]"
       if(color[v] == WHITE){
          prev[v]=u;
          DFS_Visit(v);
    }}
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

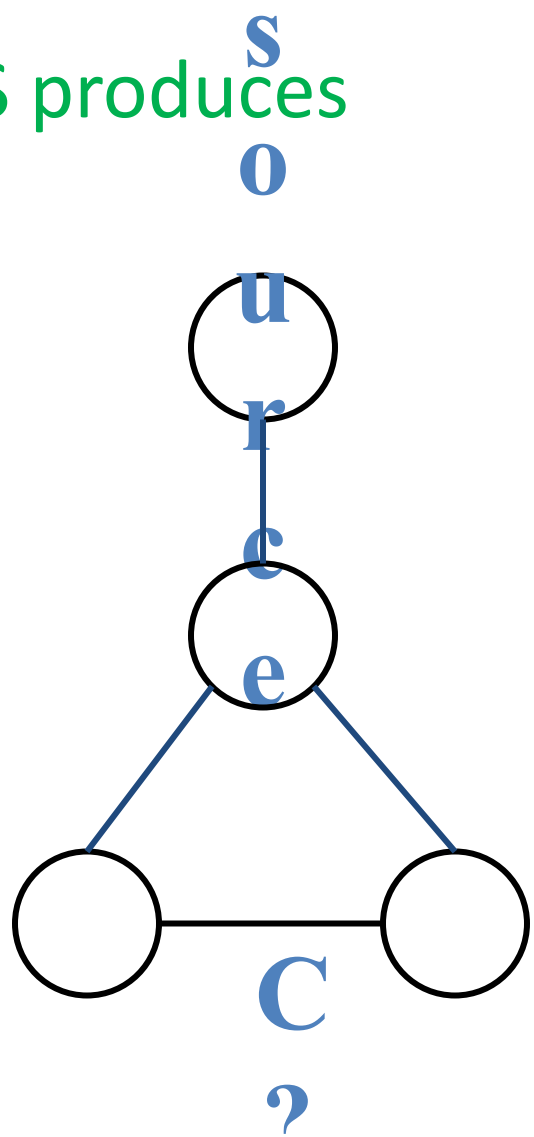
DFS: Kinds Of Edges

- Thm 22.10: If G is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a forward edge
    - But F? edge must actually be a back edge (*why?*)



DFS: Kinds Of Edges

- Thm 23.9: If G is undirected, a DFS produces only tree and back edges
- Proof by contradiction:
  - Assume there's a cross edge
    - But C? edge cannot be cross:
    - must be explored from one of the vertices it connects, becoming a tree vertex, before other vertex is explored
    - So in fact the picture is wrong…both lower tree edges cannot in fact be tree edges



DFS And Graph Cycles

- Thm: An undirected graph is *acyclic* iff a DFS yields no back edges
  - If acyclic, no back edges (because a back edge implies a cycle
  - If no back edges, acyclic
    - No back edges implies only tree edges (*Why?*)
    - Only tree edges implies we have a tree or a forest
    - Which by definition is acyclic
- Thus, can run DFS to find whether a graph has a cycle

DFS And Cycles

How would you modify the code to detect cycles?

```
Data: color[V], time,
      prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
       prev[u]=NIL;
       f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
       if (color[v]==WHITE){
          prev[v]=u;
          DFS_Visit(v);
       }
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

DFS And Cycles

What will be the running time?

```
Data: color[V], time,
      prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
        prev[u]=NIL;
        f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
DFS_Visit(u)
{
   color[u] = GREY;
   time = time+1;
   d[u] = time;
   for each v ∈ Adj[u]
   {
       if (color[v]==WHITE){
          prev[v]=u;
          DFS_Visit(v);     }
    else {cycle exists;}
   }
   color[u] = BLACK;
   time = time+1;
   f[u] = time;
}
```

DFS And Cycles

- *What will be the running time?*
- A: O(V+E)

DFS And Cycles

- *What will be the running time for undirected graph to detect cycle?*
- A: O(V+E)
- We can actually determine if cycles exist in O(V) time:
  - In an undirected acyclic forest, $|E| \leq |V| - 1$
  - So count the edges: if ever see $|V|$ distinct edges, must have seen a back edge along the way

DFS And Cycles

- *What will be the running time for directed graph to detect cycle?*
- A: O(V+E)

Reference

- Cormen –
  - Chapter 22 (Elementary Graph Algorithms)
- Exercise –
  - 22.3-5 –Detect edge using d[u], d[v], f[u], f[v]
  - 22.3-13 – Connected Component
  - 22.3-13 – Singly connected