# Data Structure and Algorithms
# CSE 2202

Department of Computer Science and Engineering

University of Dhaka

# Recommended Textbooks

- **Cormen, T.H., Leiserson, C.E., Rivest, R.L. and Stein, C., 2022. Introduction to algorithms. MIT press.**
- **Goodrich, M.T., Tamassia, R. and Goldwasser, M.H., 2013. Data structures and algorithms in Python. John Wiley & Sons Ltd.**

# Time complexity

- Amount of time needed for the algorithm to finish
- Best case
- Average case
- Worst case
- Not actual time: related to size of input.
- Big O notation

# Graph

- Graph is probably the data structure that has the closest resemblance to our daily life.

- There are many types of graphs describing the relationships in real life.
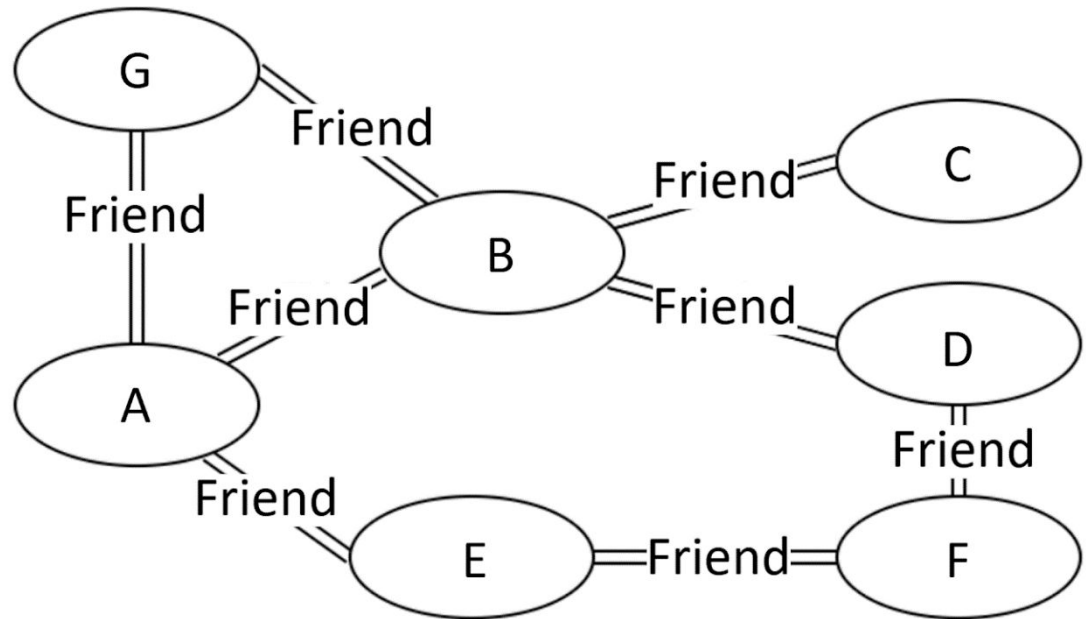
# Graph Variations

- Variations:
  - A *connected graph* has a path from every vertex to every other
  - In an *undirected graph:*
    - Edge (u,v) = edge (v,u)
    - No self-loops
  - In a *directed* graph:
    - Edge (u,v) goes from vertex u to vertex v, notated u→v

# Graph Variations
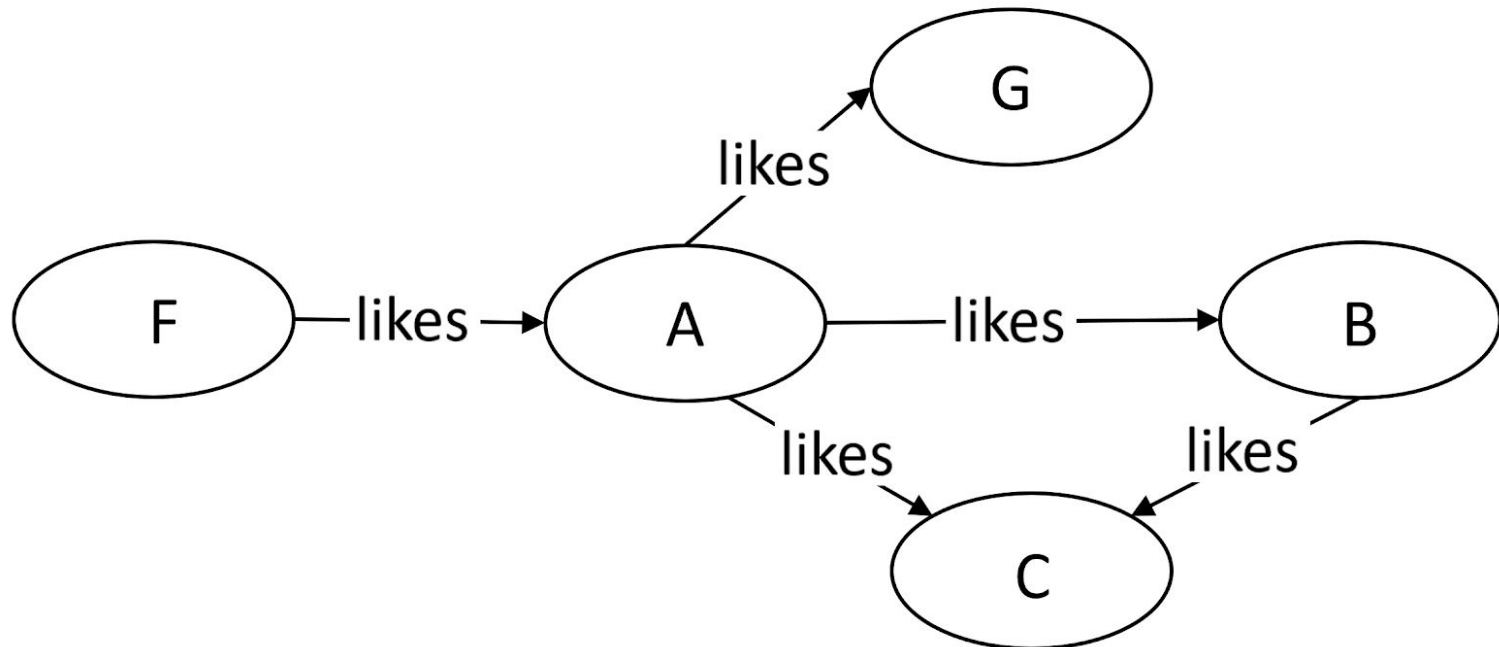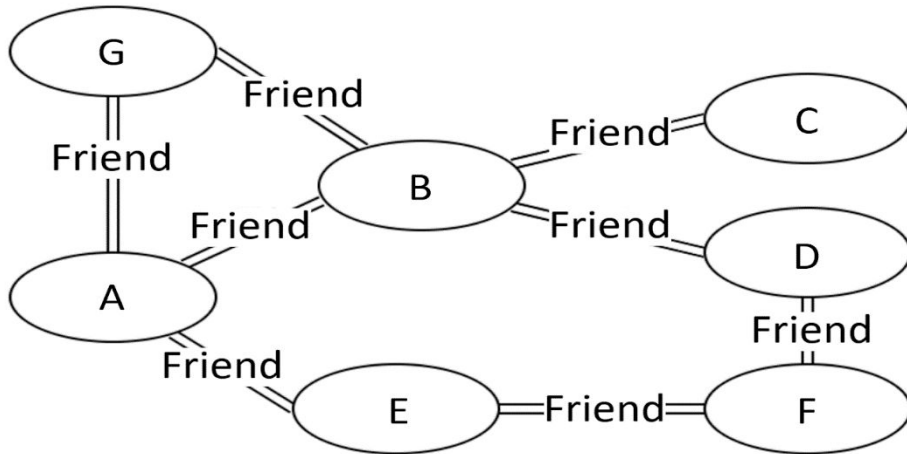
# Graph Variations

- 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
  - A *multigraph* allows multiple edges between the same vertices
    - E.g., the call graph in a program (a function can get called from multiple points in another function)
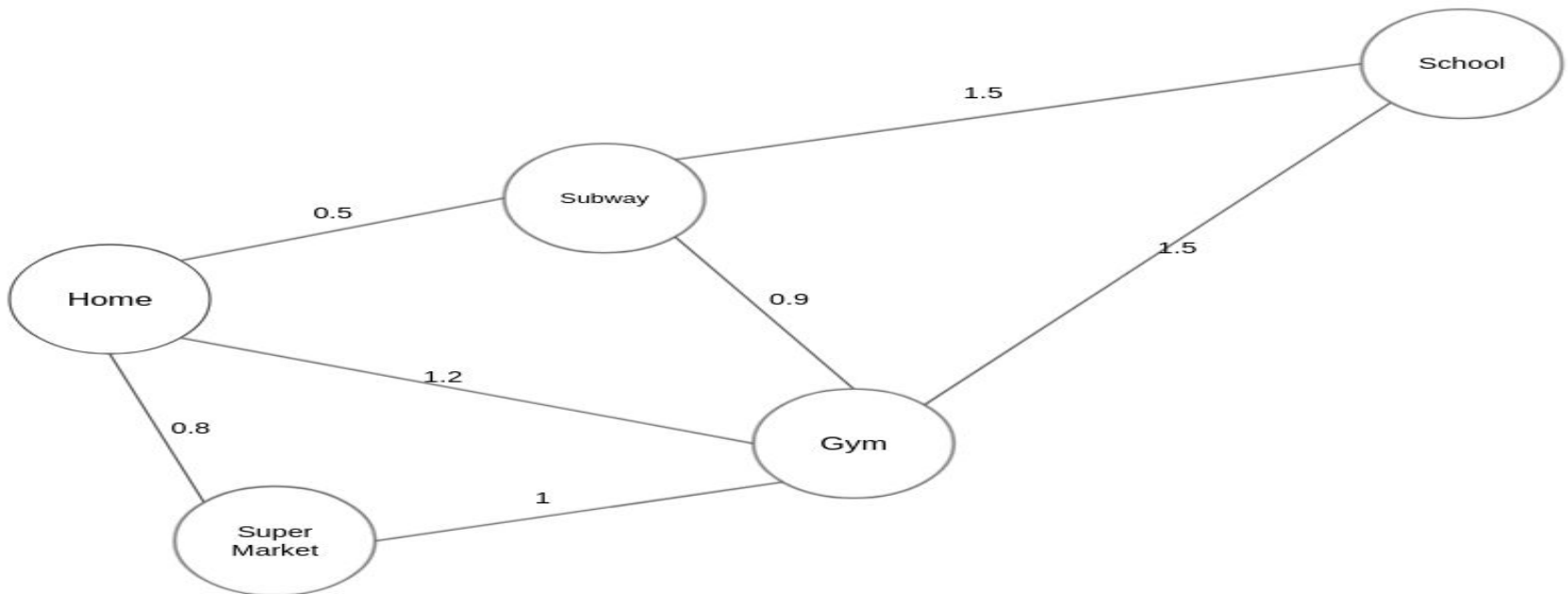
# Graph - Definition

- 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
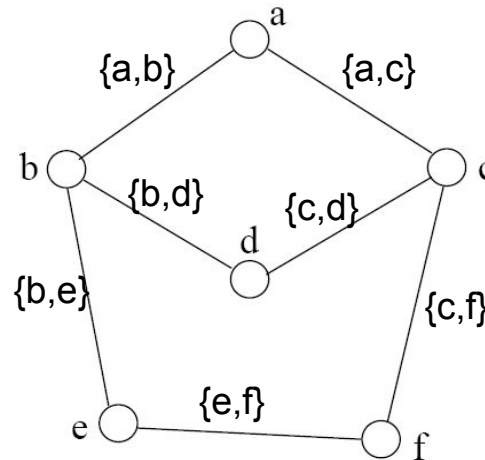


$$V = \{a, b, c, d, e, f\}$$

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

**An undirected graph**

# Definition

- **Path:** the sequence of vertices to go through from one vertex to another.
  - a path from A to C is [A, B, C], or [A, G, B, C], or [A, E, F, D, B, C].
- **Path Length:** the number of edges in a path.
- **Cycle:** a path where the starting point and endpoint are the same vertex.
  - [A, B, D, F, E] forms a cycle. Similarly, [A, G, B] forms another cycle.
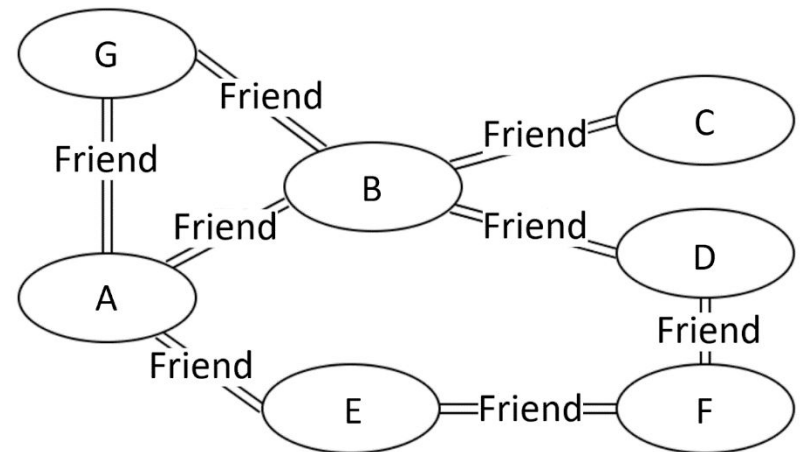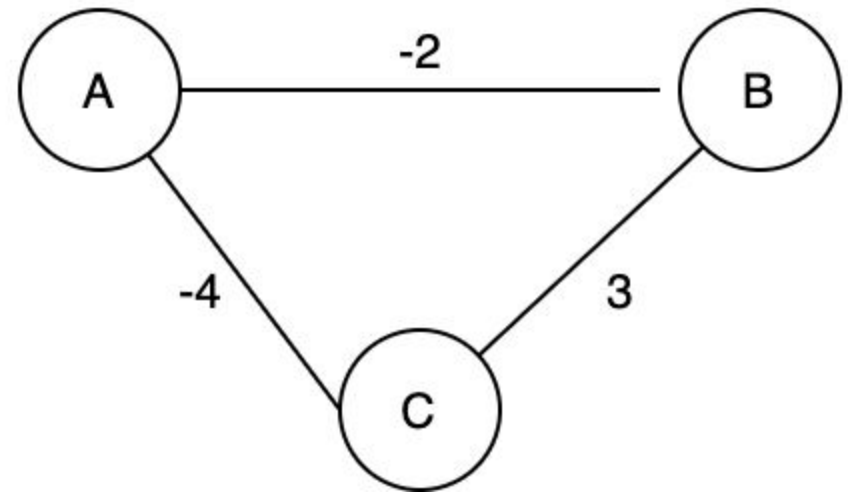
# Definition

- **Degree of a Vertex:** the term "degree" applies to unweighted graphs. The degree of a vertex is the number of edges connecting the vertex.

  - the degree of vertex A is 3 because three edges are connecting it.

- **In-Degree:** "in-degree" is a concept in directed graphs. If the in-degree of a vertex is d, there are d directional edges incident to the vertex.

  - In Figure 2, A's indegree is 1, i.e., the edge from F to A.

- **Out-Degree:** "out-degree" is a concept in directed graphs. If the out-degree of a vertex is d, there are d edges incident from the vertex.

  - A's outdegree is 3, i,e, the edges A to B, A to C, and A to G.
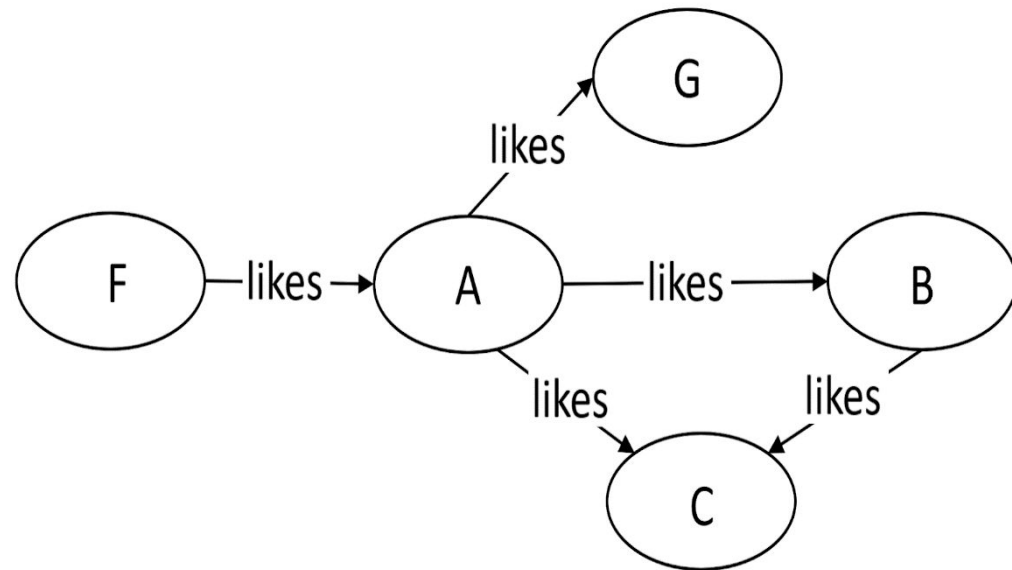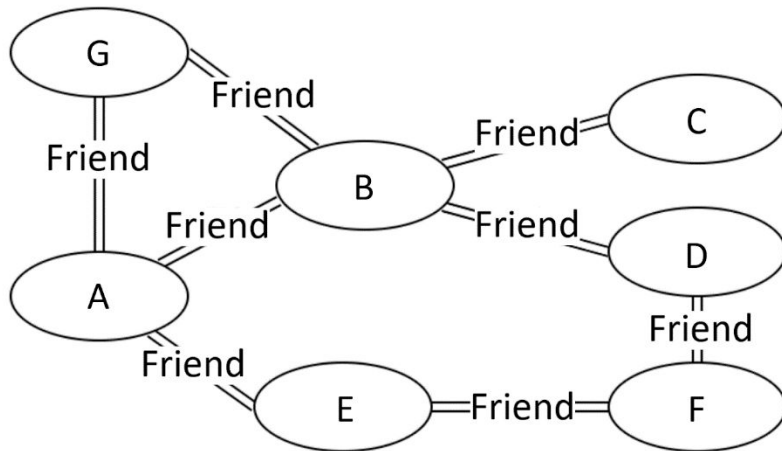
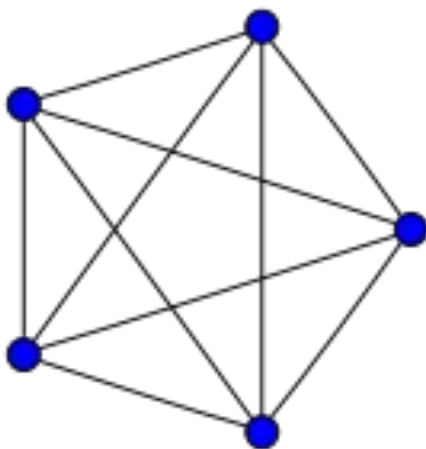# Definition

- **Connectivity**: if there exists at least one path between two vertices, these two vertices are connected.

  - A and C are connected because there is at least one path connecting them.

- **Negative Weight Cycle:** In a "weighted graph", if the sum of the weights of all edges of a cycle is a negative value, it is a negative weight cycle.

  - In the Figure the sum of weights is -3.

# Definition

- Complete Graph
  - a complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge.
  - A complete digraph is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction).
  - How many edges are there in an N-vertex complete graph?

# Definition

- Bipartite Graph

- a bipartite graph (or bigraph) is a graph whose vertices can be divided into two disjoint and independent sets
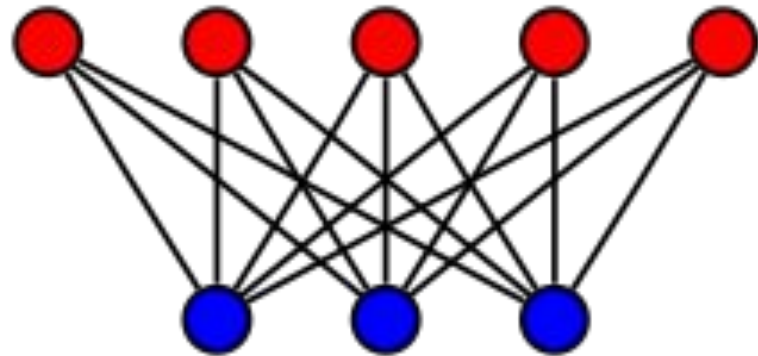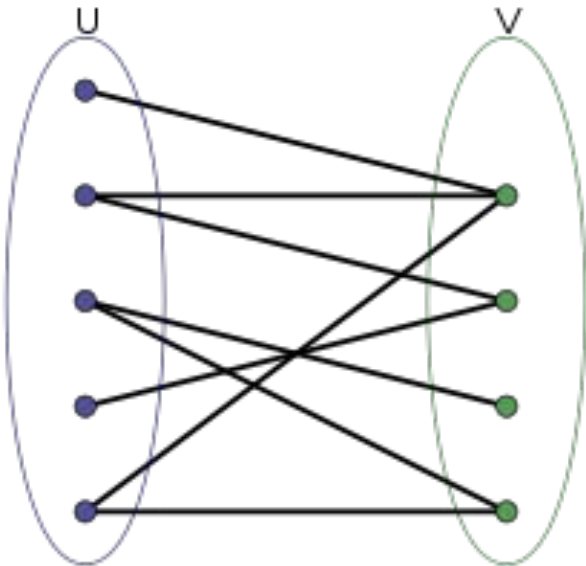
# Definition

- We will typically express running times in terms of |E| and |V| (often dropping the |'s)
  - If $|E| \approx |V|^2$ the graph is *dense*
  - If $|E| \approx |V|$ the graph is *sparse*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense

# 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

# Adjacency Matrix
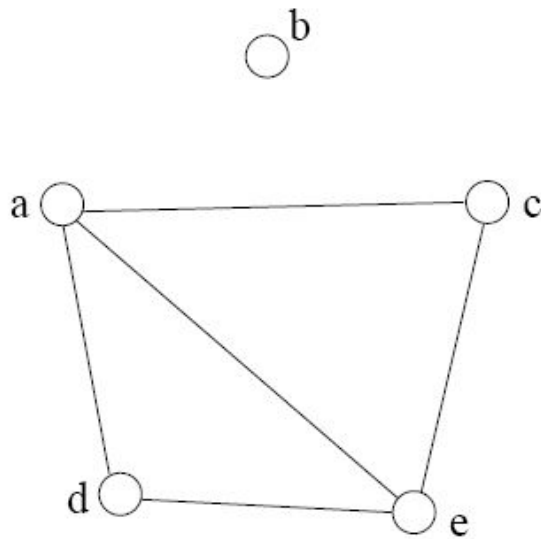


|   | a | b | c | d | e |
|---|---|---|---|---|---|
| a | 0 | 0 | 1 | 1 | 1 |
| b | 0 | 0 | 0 | 0 | 0 |
| c | 1 | 0 | 0 | 0 | 1 |
| d | 1 | 0 | 0 | 0 | 1 |
| e | 1 | 0 | 1 | 1 | 0 |

- |

# Simple Questions on Adjacency Matrix

- Is there a direct link between A and B?
- What is the indegree and outdegree for a vertex A?
- 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
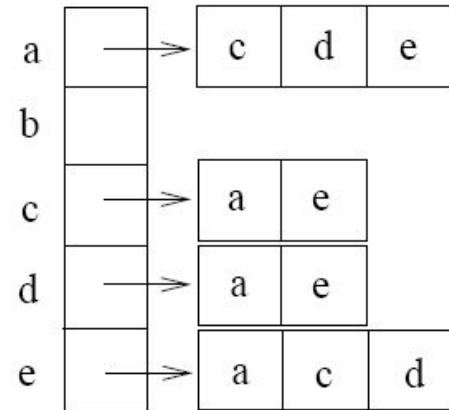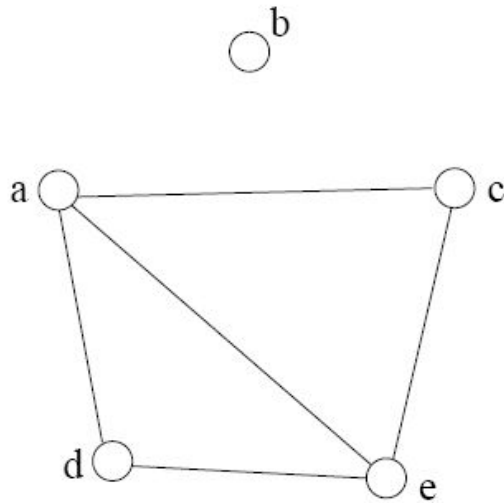


- If the graph is not dense, in other words, sparse, a better solution is an adjacency list

# Adjacency Matrix Example



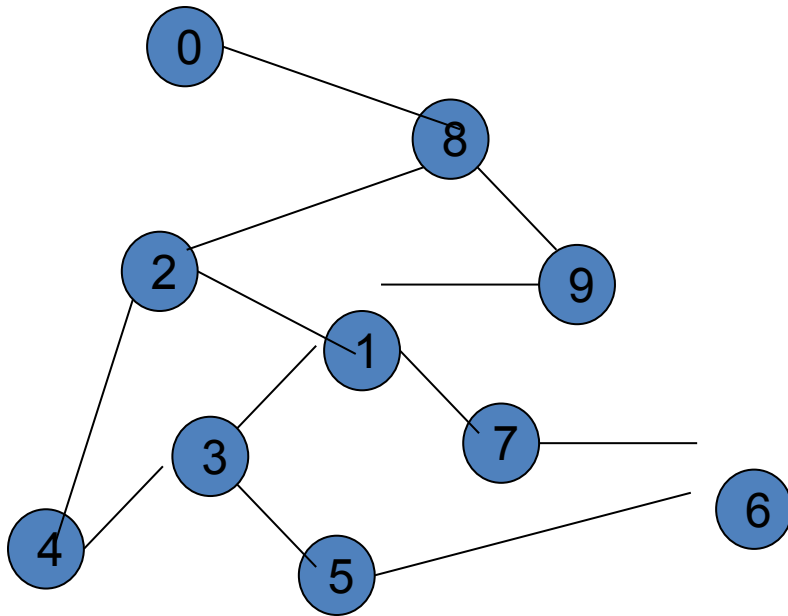|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 2 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 1 | 1 | 0 | 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
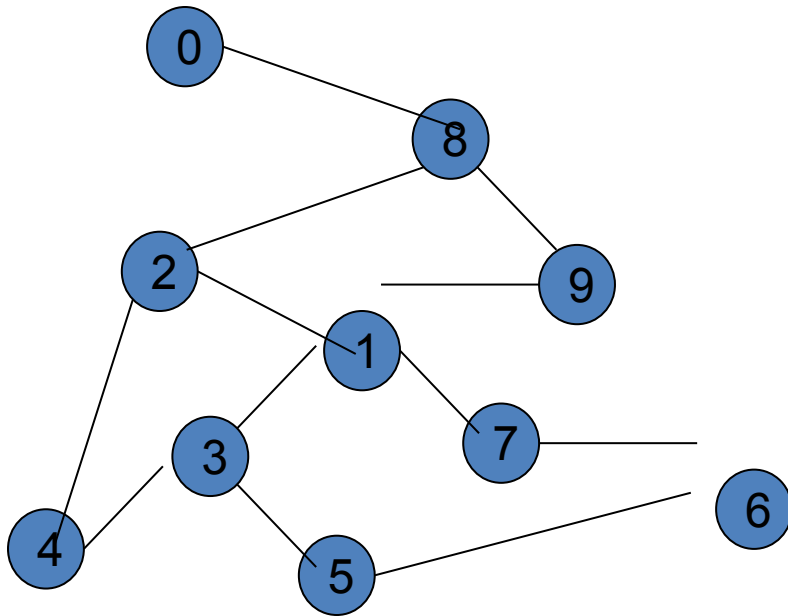
# Storage of Adjacency List

- 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 $(v_0, v_1, v_2, \ldots v_k)$ such that:
  - For $0 \le i < k$, $\{v_i, v_{i+1}\}$ is an edge

  *Note: a path is allowed to go through the same vertex or the same edge any number of times!*
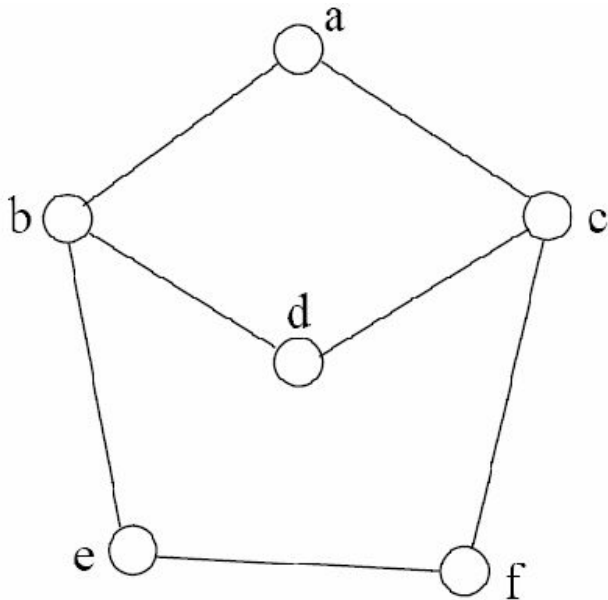
- The length of a path is the number of edges on the path

# 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}

# Exercises on Graph

- CLRS – Chapter 22 – elementary Graph Algorithms
- Exercise you have to solve: (Page 593)
  - 22.1-5 (Square)
  - 22.1-6 (Universal Sink)

# 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)
    - Find the shortest paths in an unweighted graph
  - Depth-First Search (DFS)
    - Topological sort
    - Find strongly connected components

# 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
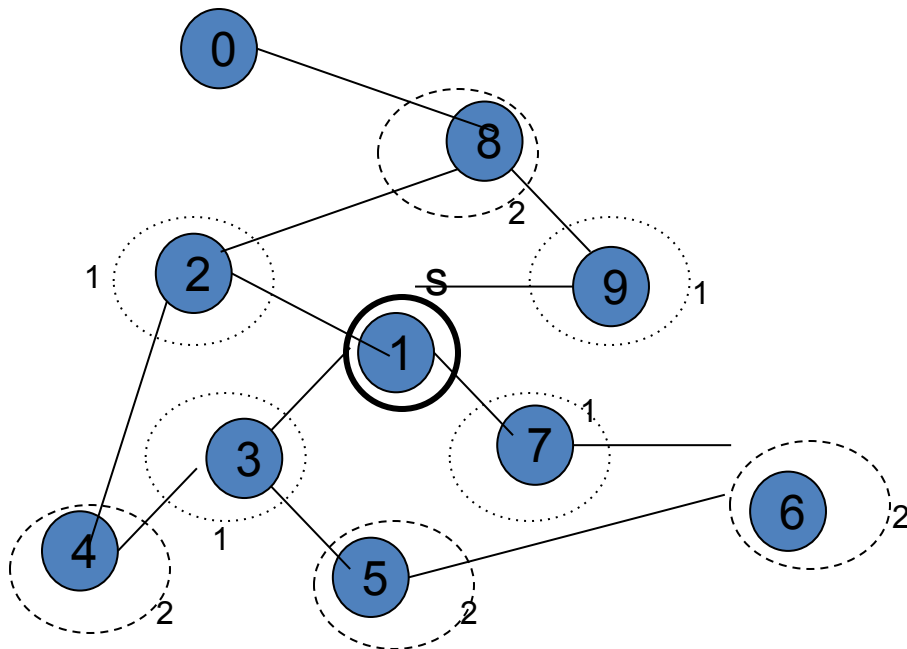


Example

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
- Goal: methodically explore every vertex and every edge
- Ultimately: build a tree on the graph
  - Pick a vertex as the root
  - Choose certain edges to produce a tree
  - Note: might also build a *forest* if graph is not connected

# Breadth-First Search

- "Explore" a graph, turning it into a tree
  - One vertex at a time
  - Expand frontier of explored vertices across the *breadth* of the frontier
- Builds a tree over the graph
  - Pick a *source vertex* to be the root
  - Find ("discover") its children, then their children, etc.

# Breadth-First Search

- Every vertex of a graph contains a color at every moment:
  - White vertices have not been discovered
    - All vertices start with white initially
  - Grey vertices are discovered but not fully explored
    - They may be adjacent to white vertices
  - Black vertices are discovered and fully explored
    - They are adjacent only to black and gray vertices
- Explore vertices by scanning adjacency list of grey vertices

# Breadth-First Search: The Code

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
   for each vertex u ∈
 V-{s}
 {
     color[u]=WHITE;
 prev[u]=NIL;
 d[u]=inf;
 }
 color[s]=GRAY;
d[s]=0; prev[s]=NIL;
Q=empty;
ENQUEUE(Q,s);
```
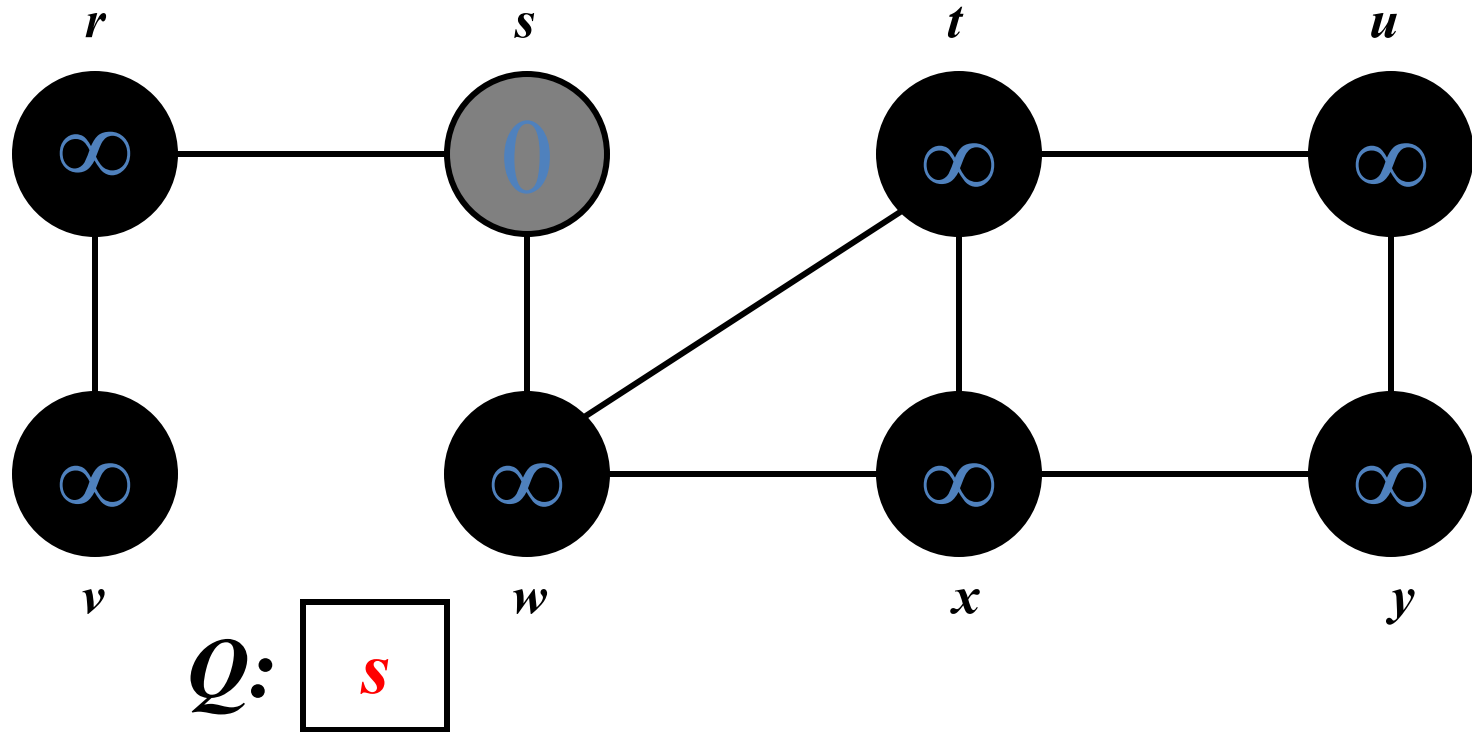
```
While(Q not empty)
{
  u = DEQUEUE(Q);
  for each v ∈ adj[u]{
    if (color[v] == WHITE){
        color[v] = GREY;
        d[v] = d[u] + 1;
        prev[v] = u;
        Enqueue(Q, v);
    }
  }
  color[u] = BLACK;
 }
}
```
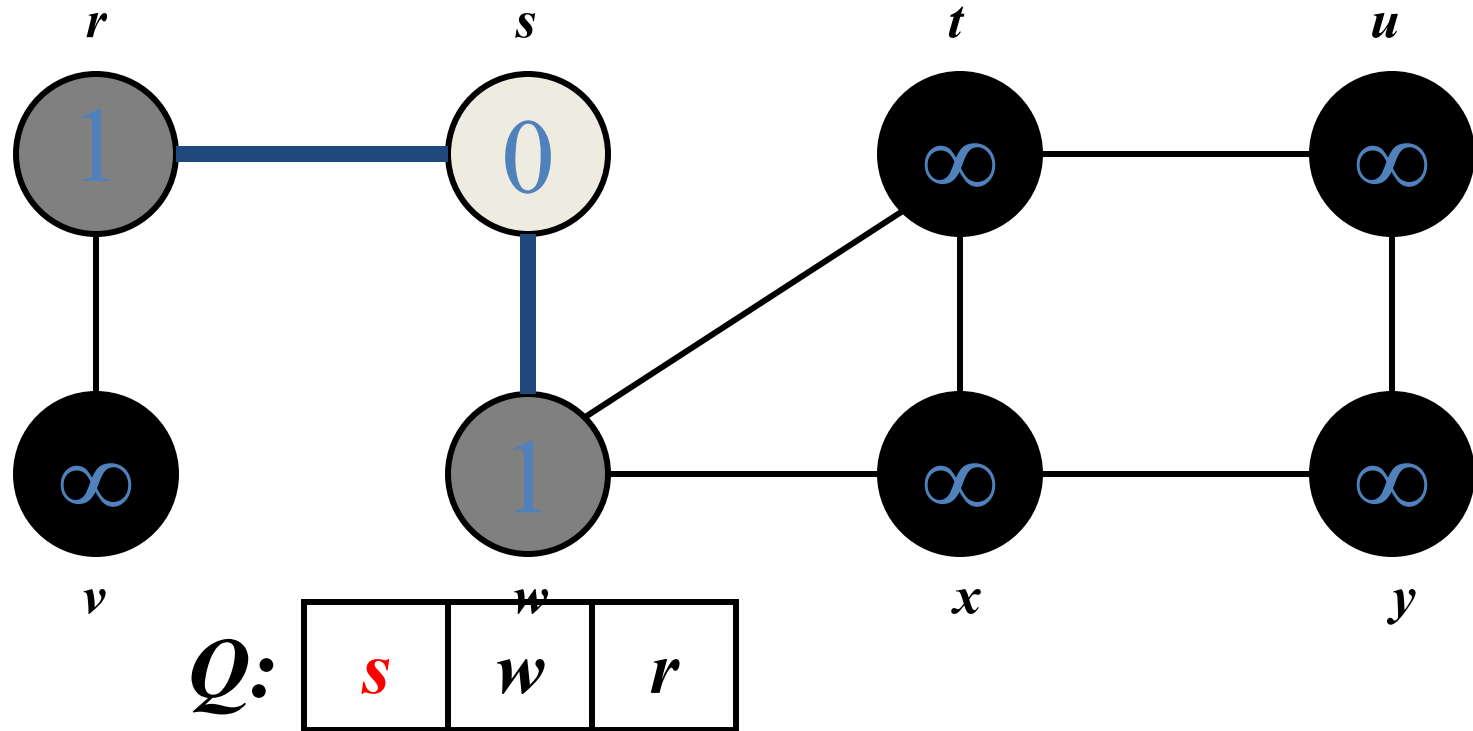
# Breadth-First Search: Example
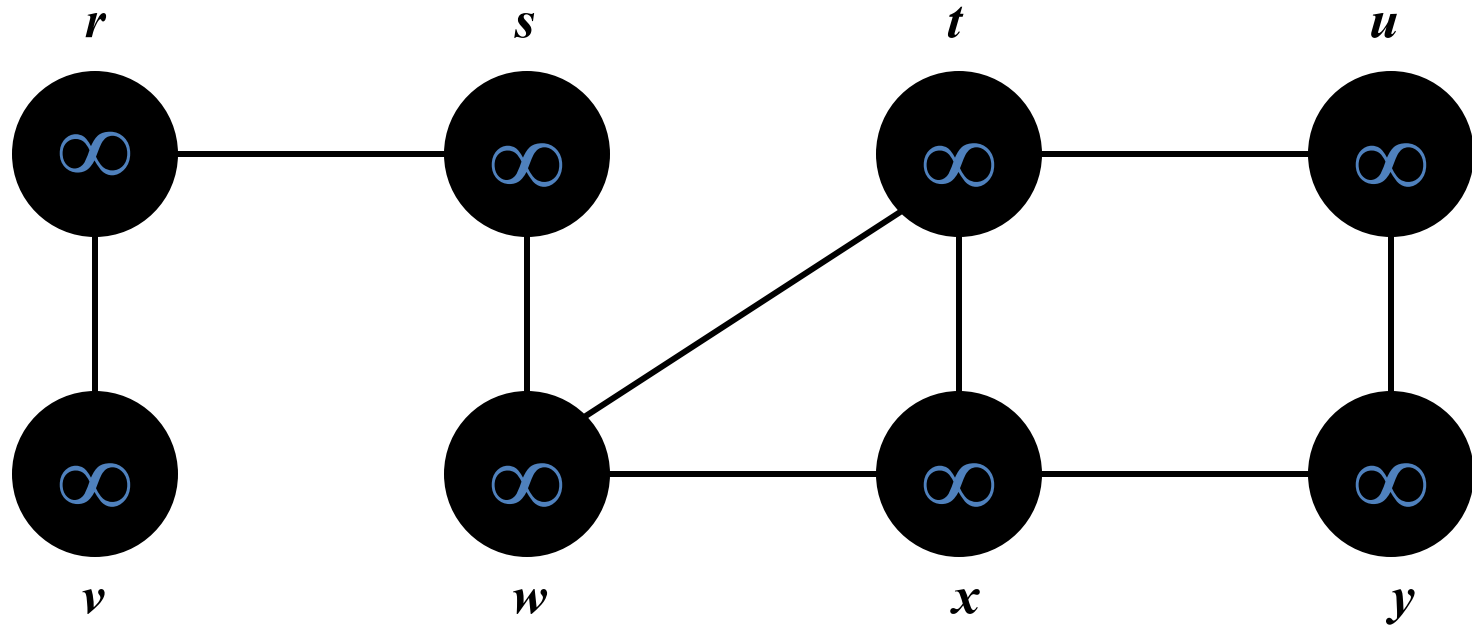


| Vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| color | W | W | W | W | W | W | W | W |
| d | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| prev | nil | nil | nil | nil | nil | nil | nil | nil |

# Breadth-First Search: Example



Q: | s |

| vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| Color | W | G | W | W | W | W | W | W |
| d | ∞ | 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ |
| prev | nil | nil | nil | nil | nil | nil | nil | nil |

# Breadth-First Search: Example



| vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| Color | G | B | W | W | W | G | W | W |
| d | 1 | 0 | ∞ | ∞ | ∞ | 1 | ∞ | ∞ |
| prev | s | nil | nil | nil | nil | s | nil | nil |

# Breadth-First Search: Example
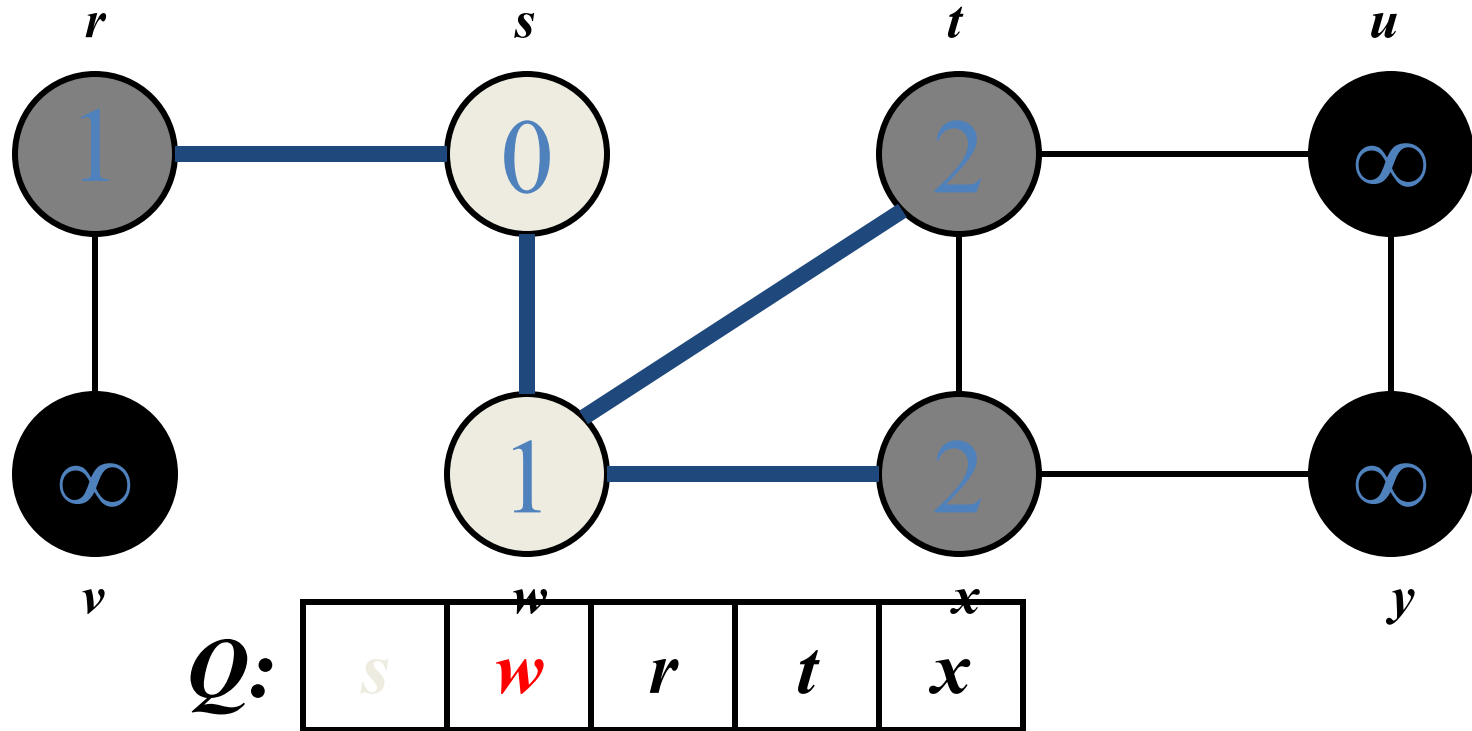


| vertex | r | s | t | u | V | w | X | y |
|--------|---|---|---|---|---|---|---|---|
| **Color** | G | B | **G** | W | W | **B** | **G** | W |
| **d** | 1 | 0 | **2** | ∞ | ∞ | **1** | **2** | ∞ |
| **prev** | s | nil | **w** | nil | nil | **s** | **w** | nil |

# Breadth-First Search: Example
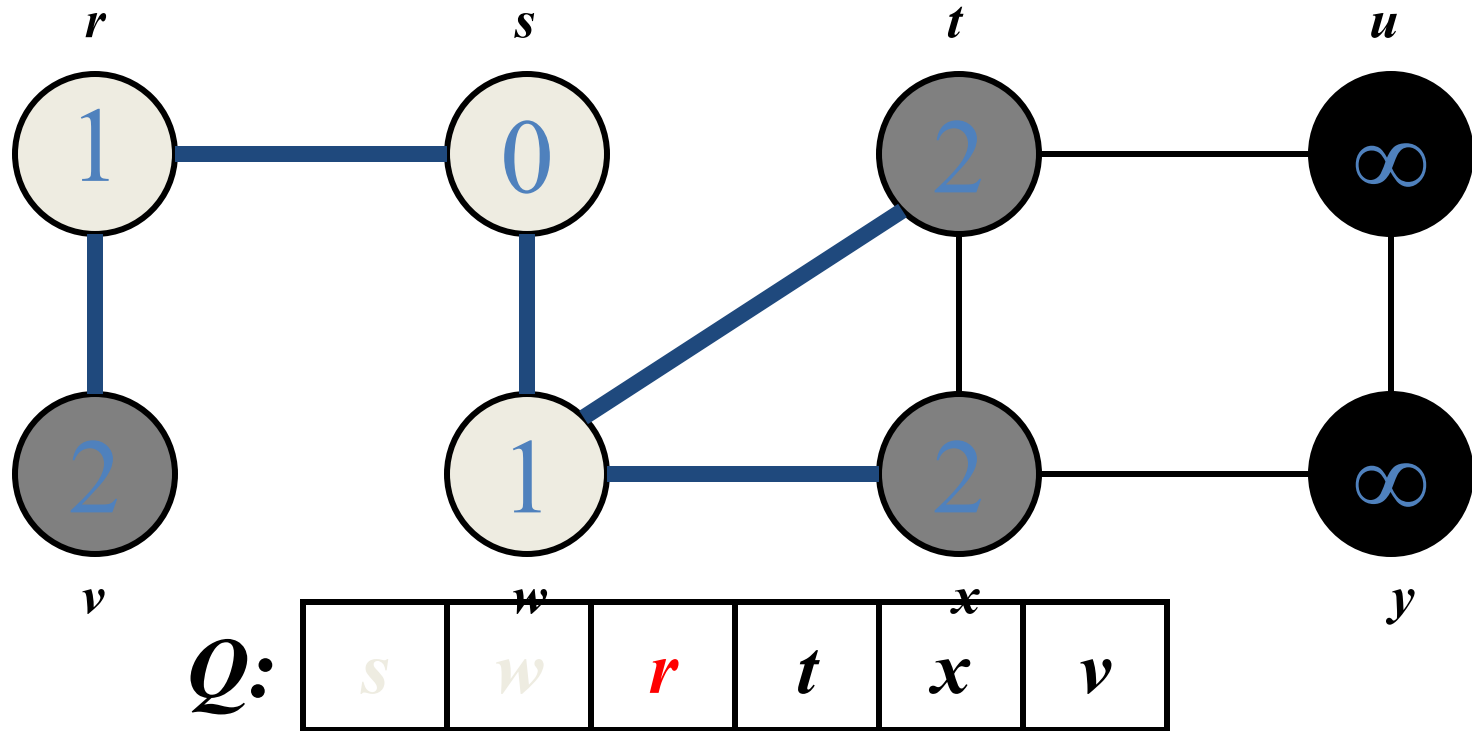


| vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| Color | B | B | G | W | G | B | G | W |
| d | 1 | 0 | 2 | ∞ | 2 | 1 | 2 | ∞ |
| prev | s | nil | w | nil | r | s | w | nil |

# Breadth-First Search: Example
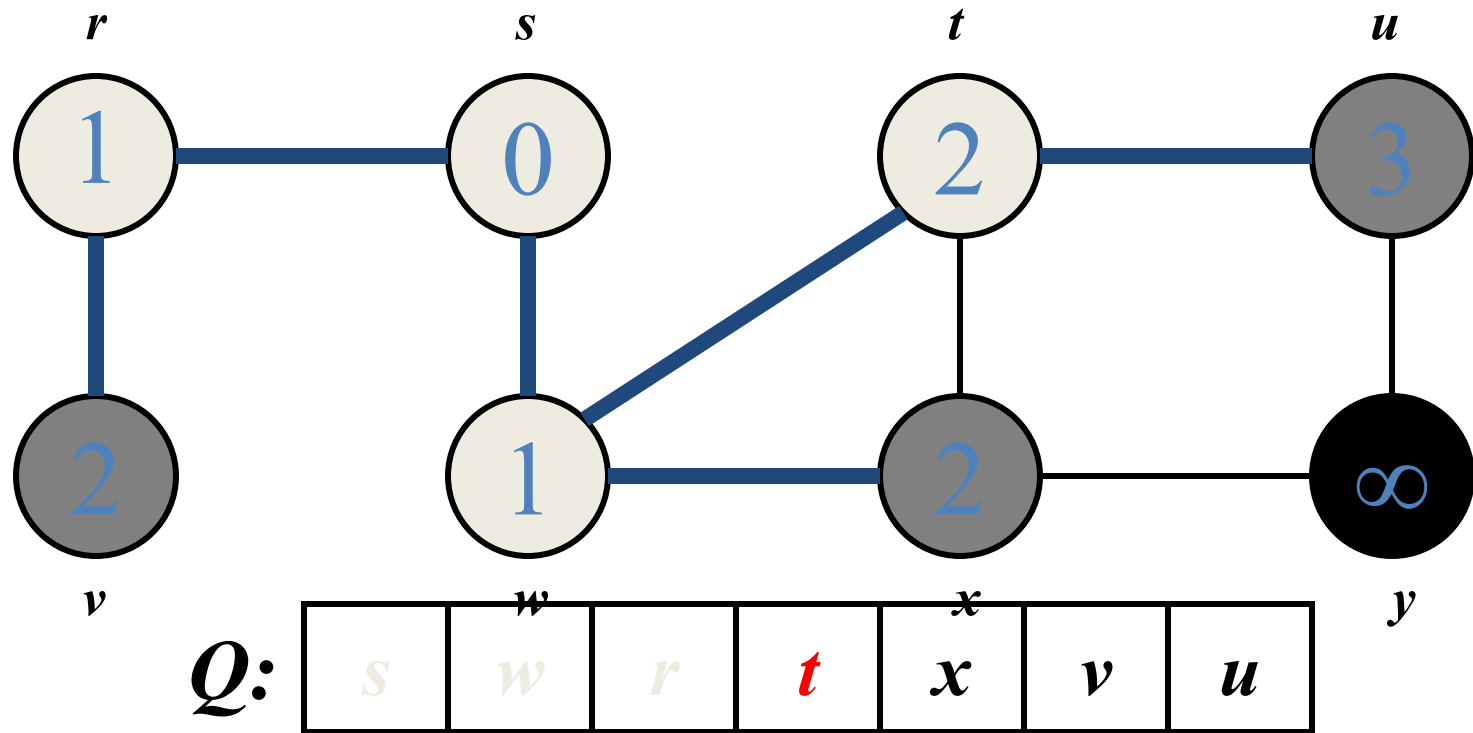


| vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| **Color** | B | B | **B** | **G** | G | B | G | W |
| **d** | 1 | 0 | **2** | **3** | 2 | 1 | 2 | ∞ |
| **prev** | s | nil | **w** | **t** | r | s | w | nil |

# Breadth-First Search: Example
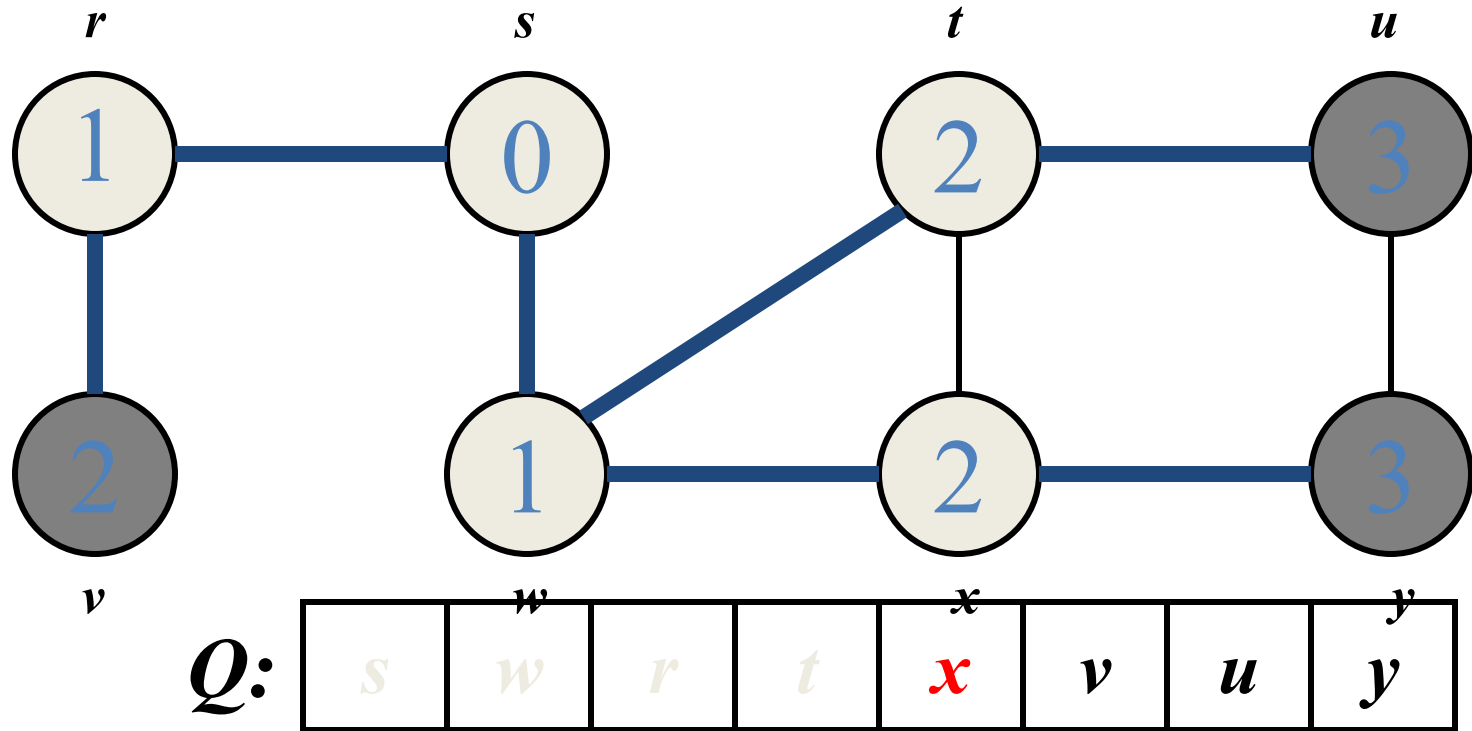


Q: | s | w | r | t | **x** | v | u | y |

| vertex | r | s | t | u | v | w | x | y |
|---|---|---|---|---|---|---|---|---|
| **Color** | B | B | B | G | G | B | **B** | **G** |
| **d** | 1 | 0 | 2 | 3 | 2 | 1 | **2** | **3** |
| **prev** | s | nil | w | t | r | s | **w** | **x** |

# Breadth-First Search: Example
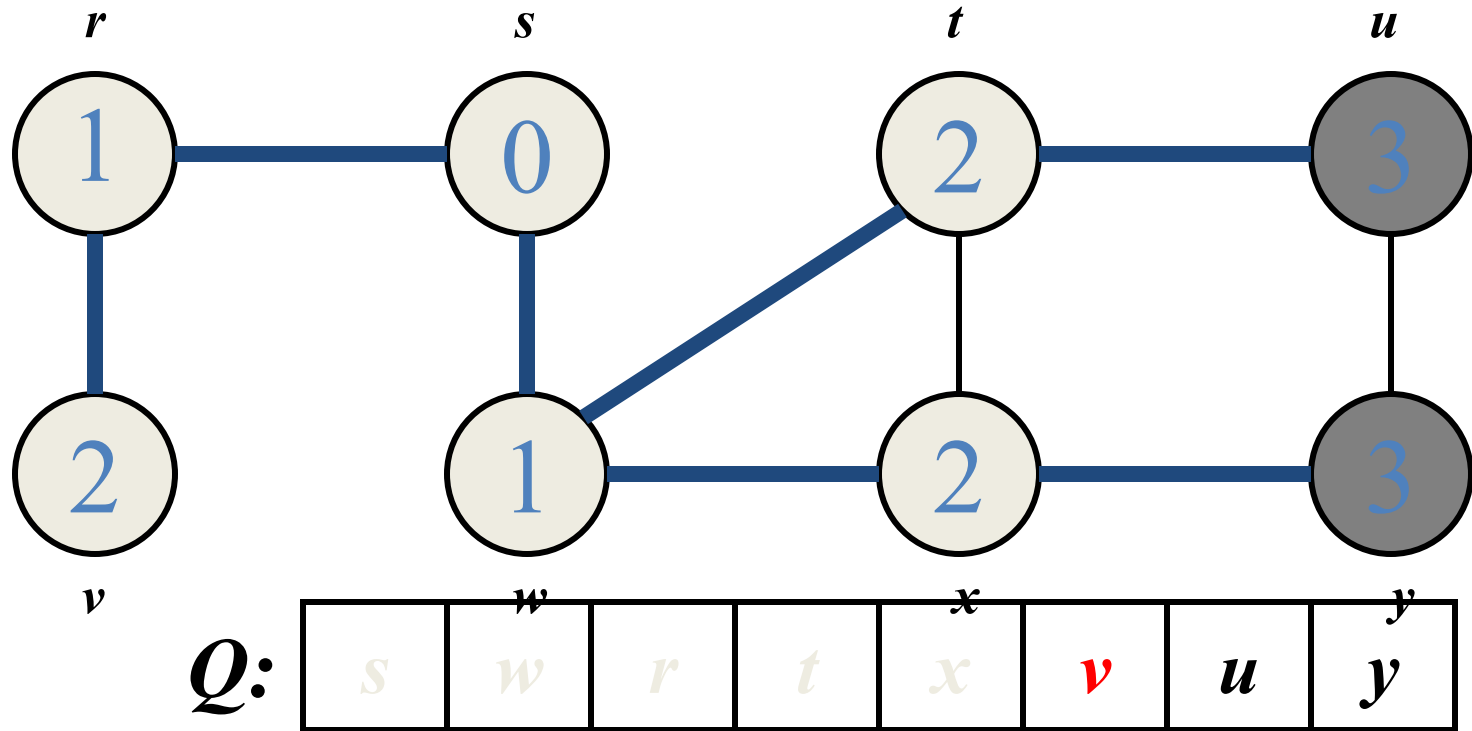


Q: | s | w | r | t | x | *v* | *u* | *y* |

| vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| **Color** | B | B | B | G | **B** | B | B | G |
| **d** | 1 | 0 | 2 | 3 | **2** | 1 | 2 | 3 |
| **prev** | s | nil | w | t | **r** | s | w | x |

# Breadth-First Search: Example
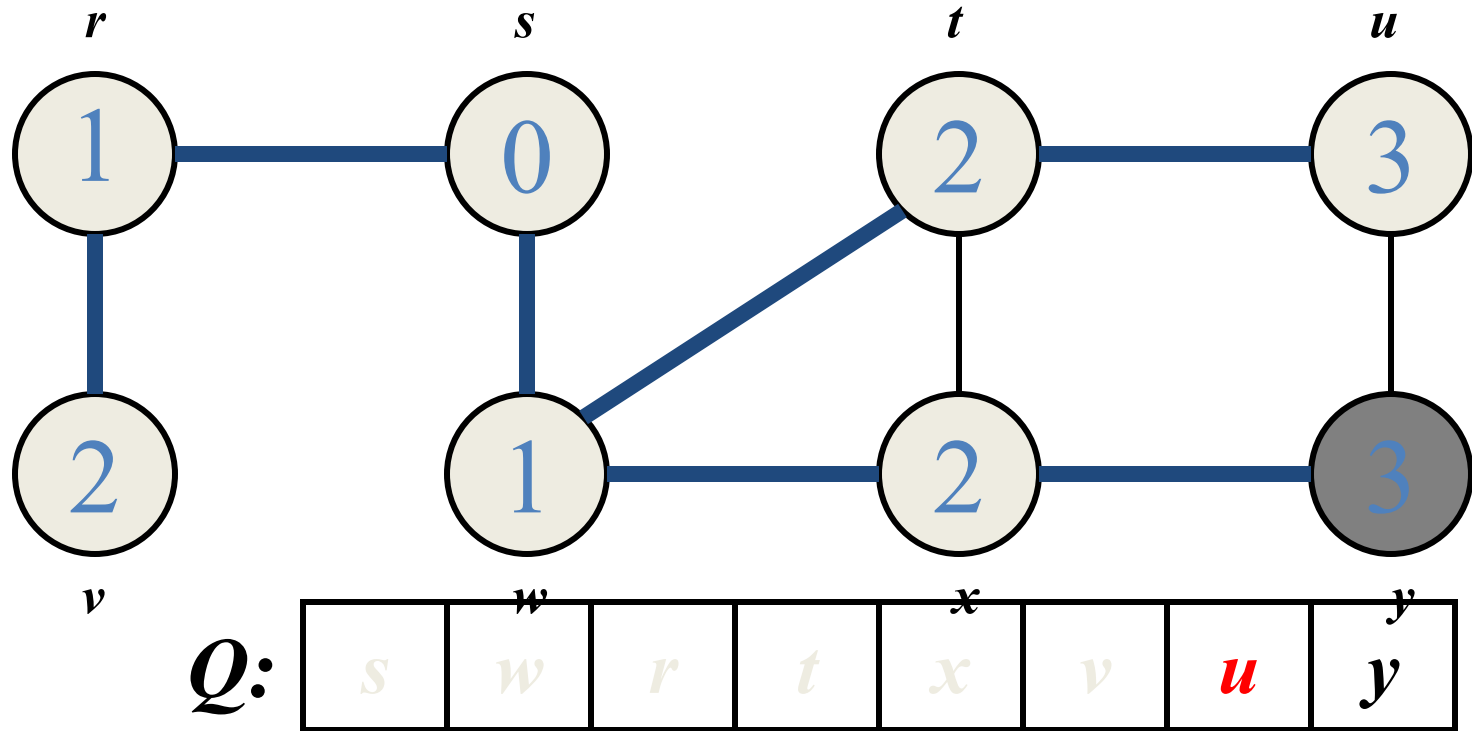


Q: | s | w | r | t | x | v | **u** | **y** |

| vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| **Color** | B | B | B | **B** | B | B | B | G |
| **d** | 1 | 0 | 2 | **3** | 2 | 1 | 2 | 3 |
| **prev** | s | nil | w | **t** | r | s | w | x |

# Breadth-First Search: Example
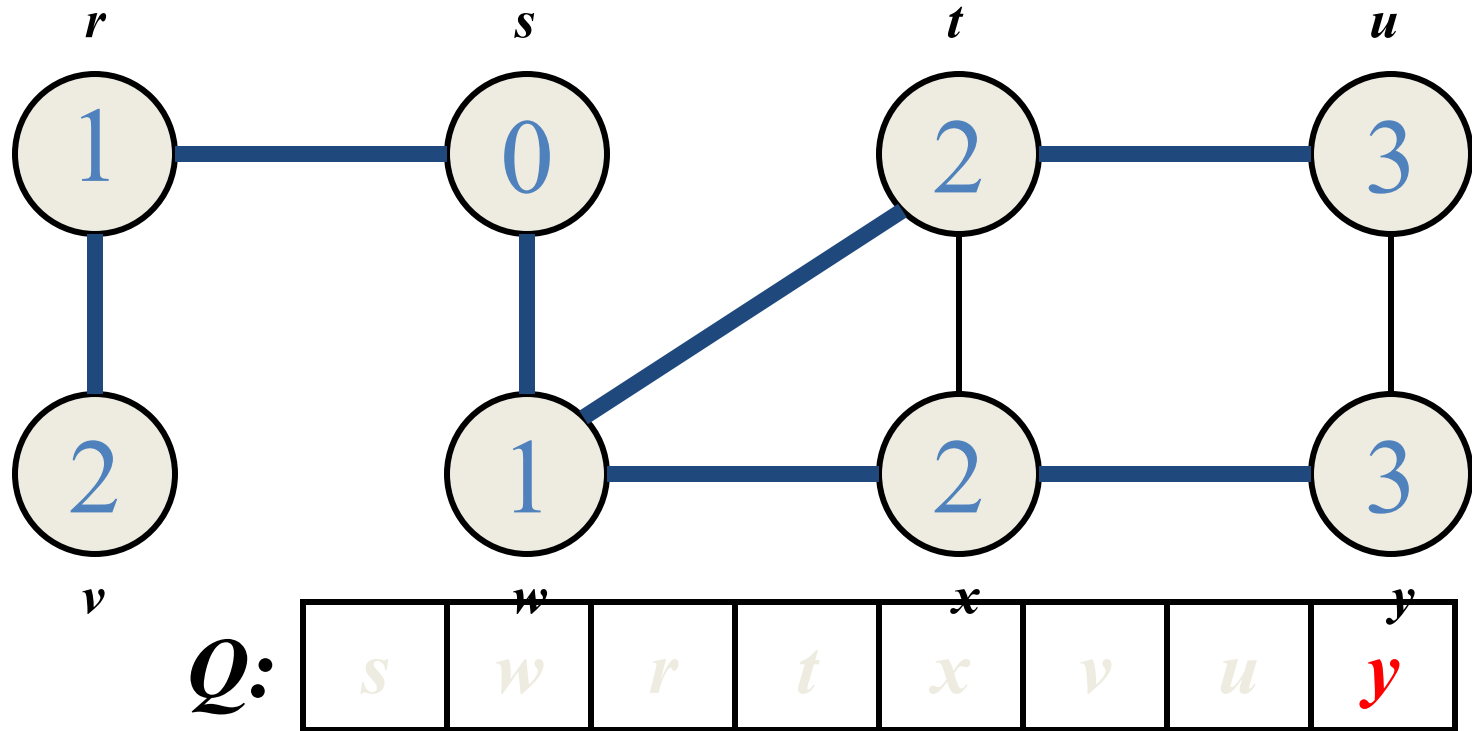


$Q$: | $s$ | $w$ | $r$ | $t$ | $x$ | $v$ | $u$ | $y$ |

| vertex | r | s | t | u | v | w | x | y |
|--------|---|---|---|---|---|---|---|---|
| **Color** | B | B | B | G | B | B | B | **B** |
| **d** | 1 | 0 | 2 | 3 | 2 | 1 | 2 | **3** |
| **prev** | s | nil | w | t | r | s | w | **x** |

# BFS: The Code (again)

```
BFS(G) // starts from here
{
   for each vertex u ∈
  V-{s}
  {
     color[u]=WHITE;
 prev[u]=NIL;
 d[u]=inf;
 }
 color[s]=GRAY;
d[s]=0; prev[s]=NIL;
Q=empty;
ENQUEUE(Q,s);
```

```
While(Q not empty)
{
  u = DEQUEUE(Q);
  for each v ∈ adj[u]{
    if (color[v] == WHITE){
        color[v] = GREY;
        d[v] = d[u] + 1;
        prev[v] = u;
        Enqueue(Q, v);
    }
  }
  color[u] = BLACK;
 }
}
```

# Breadth-First Search: Print Path

```
Print-Path(G, s, v)
{
   if(v==s)
    print(s)
    else if(prev[v]==NIL)
    print(No path);
   else{
    Print-Path(G,s,prev[v]);
    print(v);
   }
}
```

# BFS: Complexity

```
Data: color[V], prev[V],d[V]

BFS(G) // starts from here
{
   for each vertex u ∈
  V-{s}
  {
     color[u]=WHITE;
  prev[u]=NIL;
  d[u]=inf;
  }
  color[s]=GRAY;
  d[s]=0; prev[s]=NIL;
  Q=empty;
  ENQUEUE(Q,s);
```

O(V)

```
While(Q not empty)
 {
   u = DEQUEUE(Q);
   for each v ∈ adj[u]{
    if(color[v] == WHITE){
        color[v] = GREY;
        d[v] = d[u] + 1;
        prev[v] = u;
        Enqueue(Q, v);
     }
   }
  color[u] = BLACK;
 }
}
```

*u = every vertex, but only once*
*(Why?)*

O(V)

*What will be the running time?*

**Total running time: 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 $\infty$ if v not reachable from s
  - Proof given in the book (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 bipartite-ness of a graph.
- Find cycle in a graph.
- Find the connectedness of a graph.
- And many more.

# Exercises on BFS

- CLRS – Chapter 22 – elementary Graph Algorithms
- Exercise you have to solve: (Page 602)
  - 22.2-7 (Wrestler)
  - 22.2-8 (Diameter)
  - 22.2-9 (Traverse)

# 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
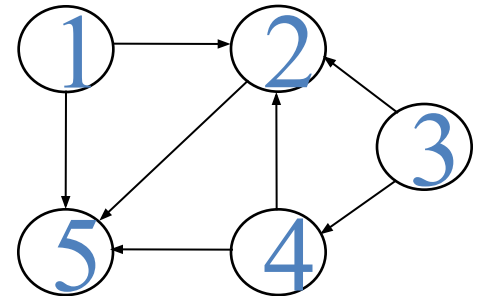
# Depth-First Search

- Search "deeper" in the graph whenever possible
- Edges are explored out of the most recently discovered vertex **v** that still has unexplored edges
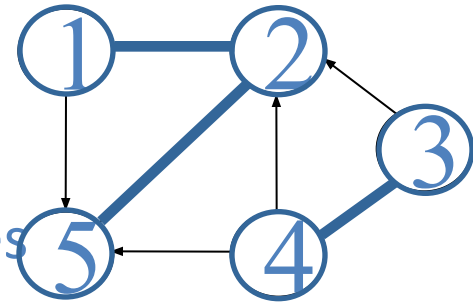
- After all edges of v have been explored, the search "backtracks" from the parent of **v**
- The process continues until all vertices reachable from the original source have been discovered
- If undiscovered vertices remain, choose one of them as a new source and repeat the search from that vertex
- DFS creates a "depth-first forest"

# DFS Additional Data Structures

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

$$1 \leq d[u] < f[u] \leq 2|V|$$

GRAY

W
HI

d

f

BL
AC

0

2
V
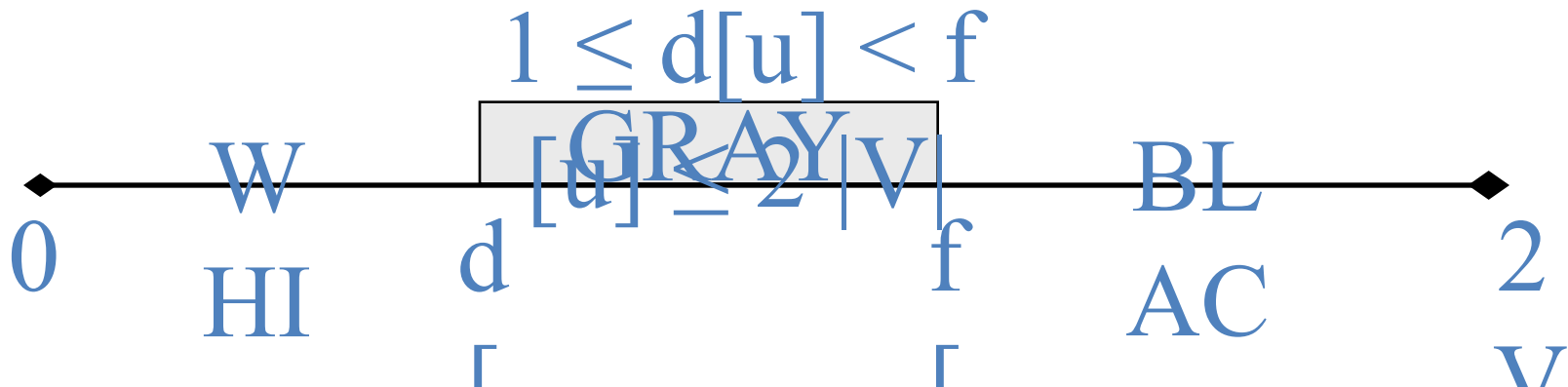
# Depth-First Search: The Code

```
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;
}
```
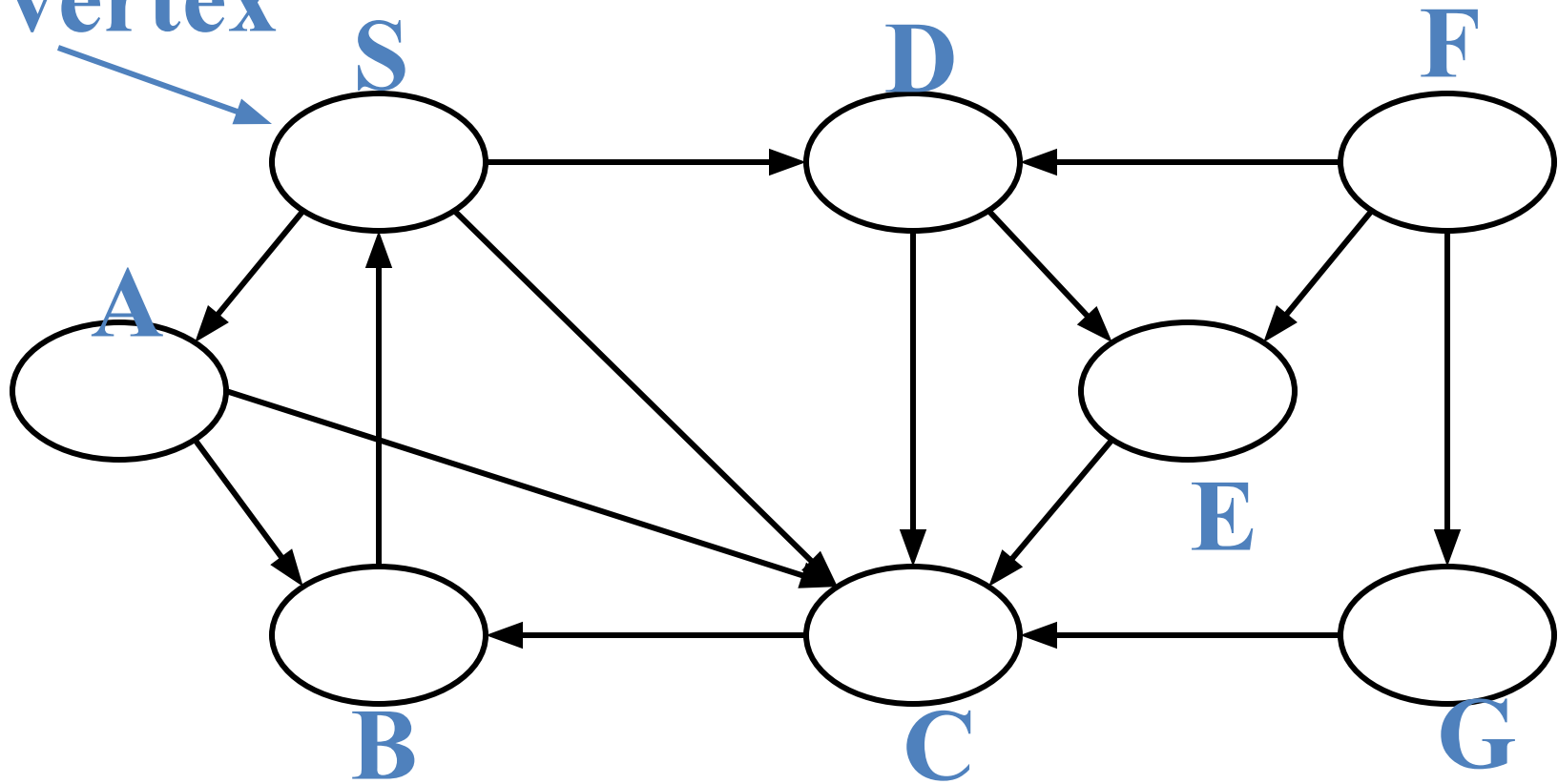
# Depth-First Search: The Code

```
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;
}
```
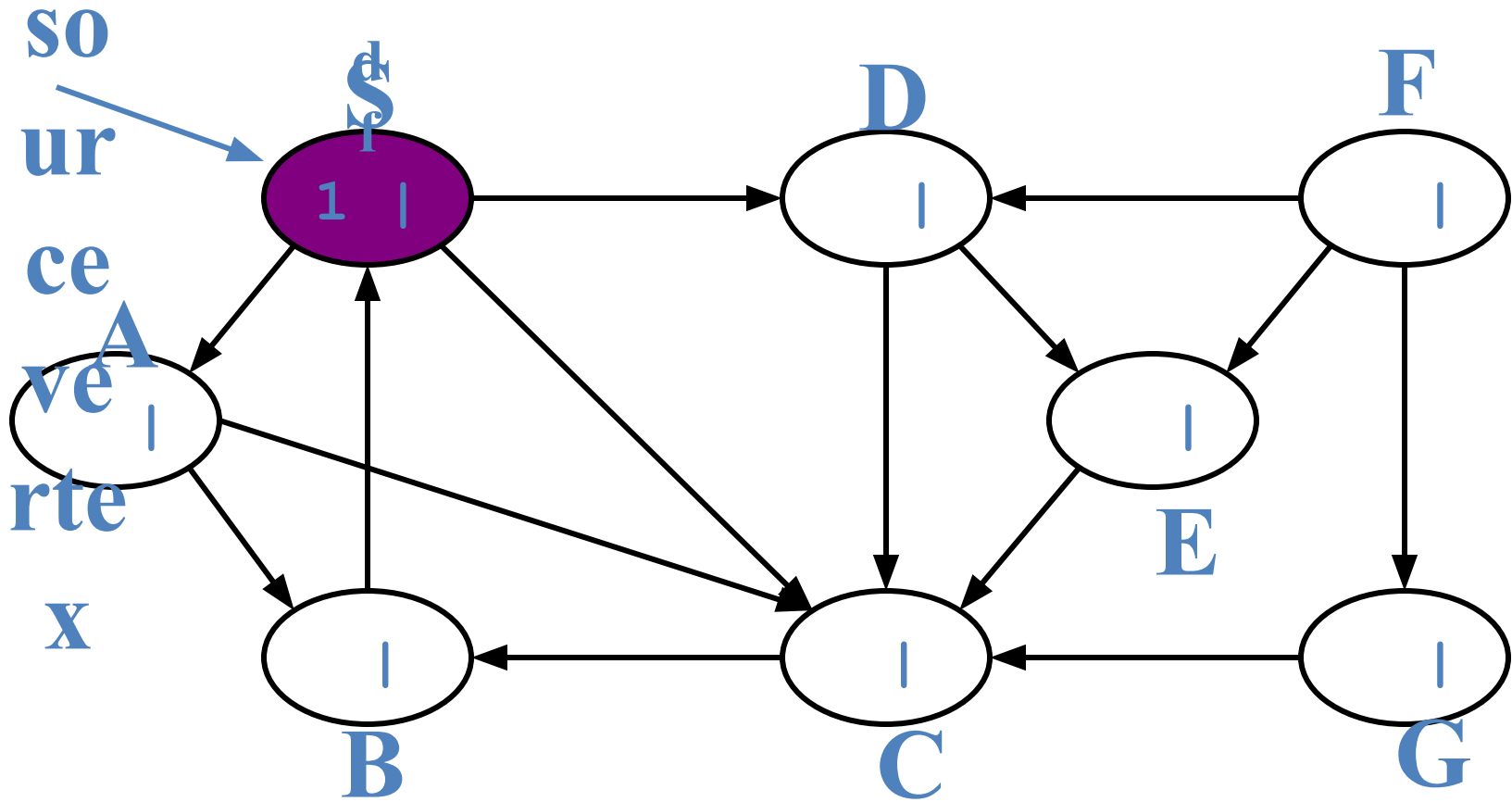
## What does d[u] represent?

# 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?**
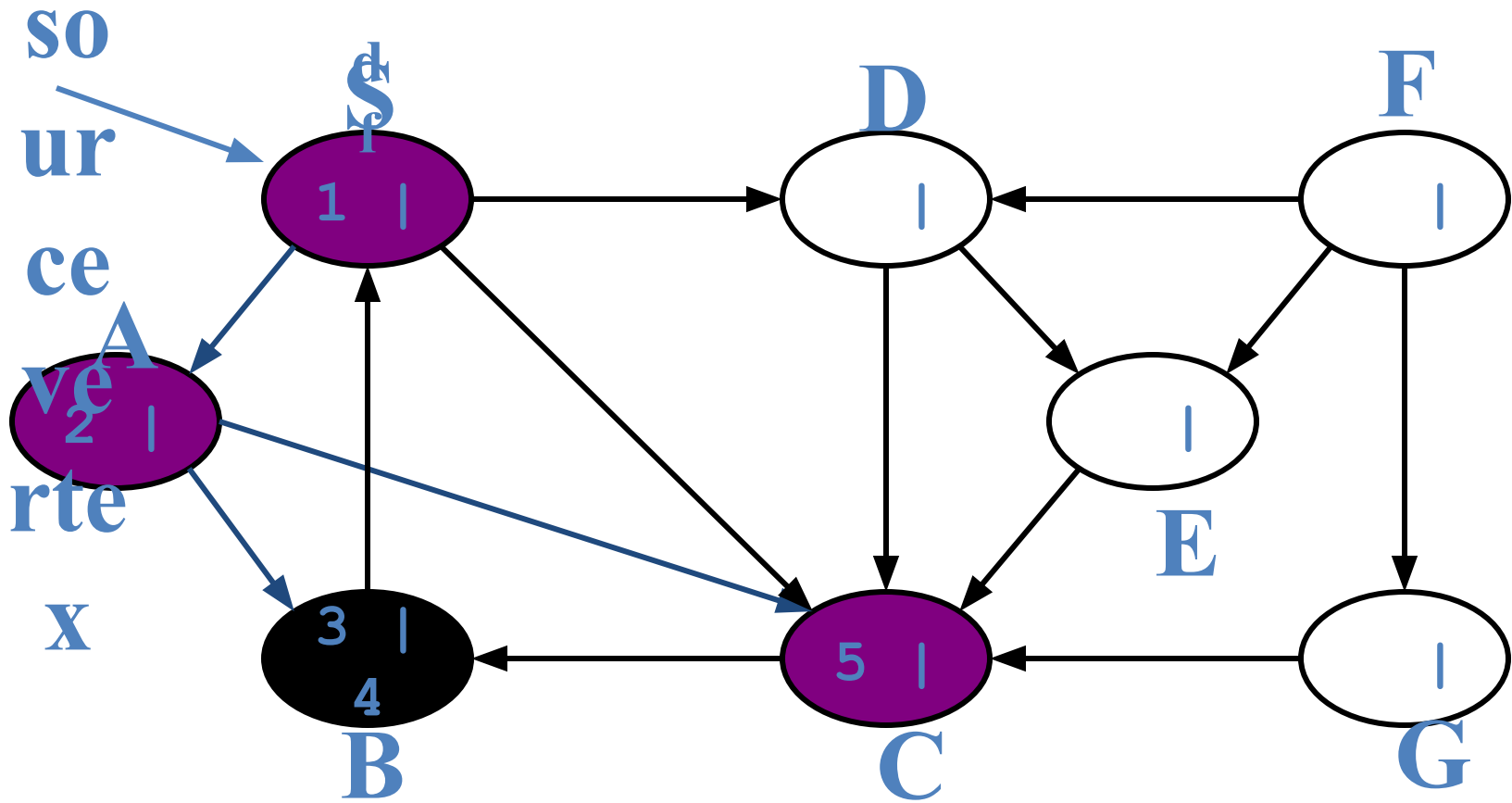
# DFS Example

**source vertex**

# DFS Example



source vertex

S d f 1 |

A |

B |

D |

E |

C |

F |

G |

# DFS Example

# DFS Example

# DFS Example



source vertex

S d f 1 |

A 2 |

B 3 | 4

D |

F |

E |

C |

G |

# DFS Example
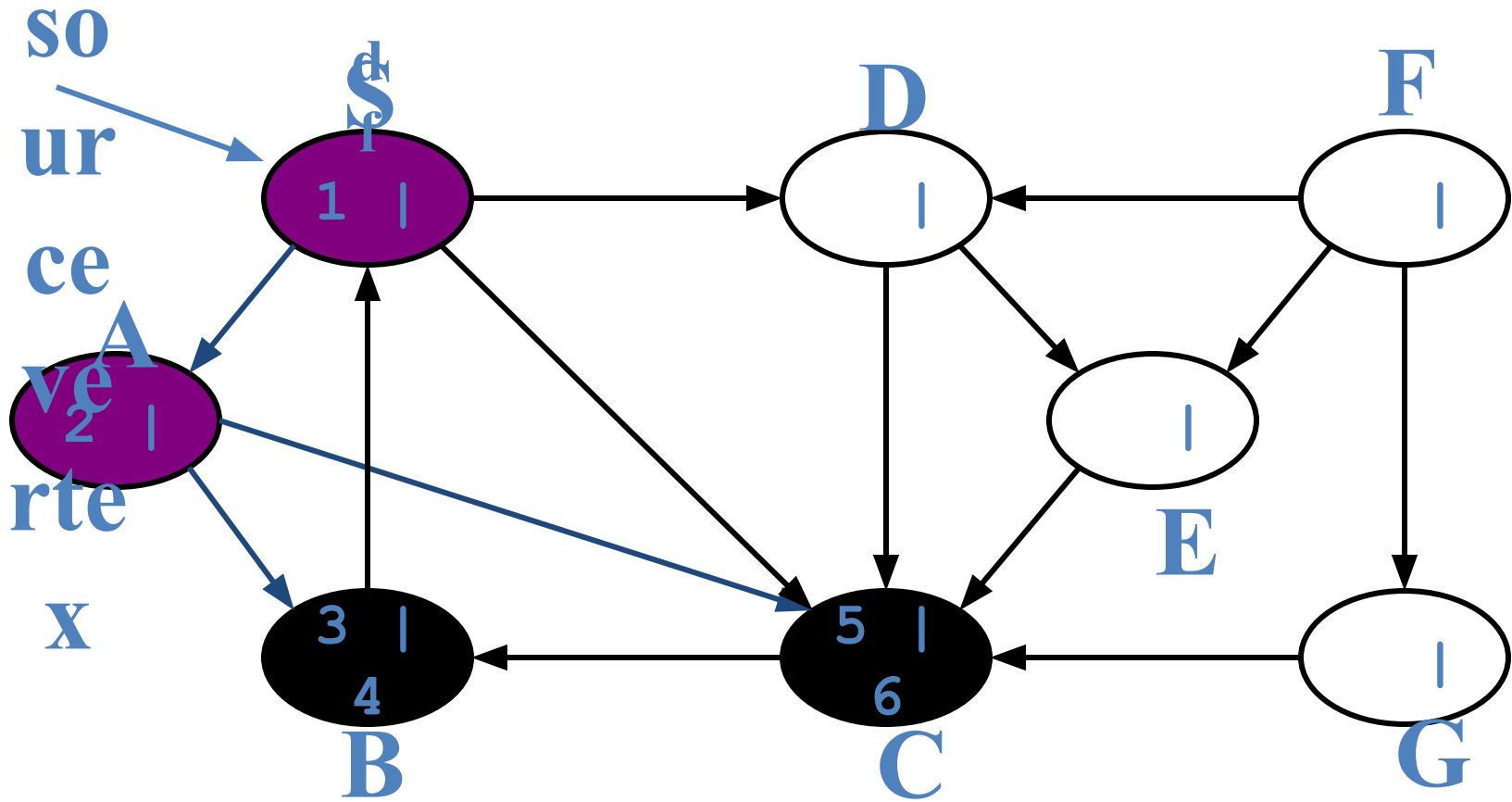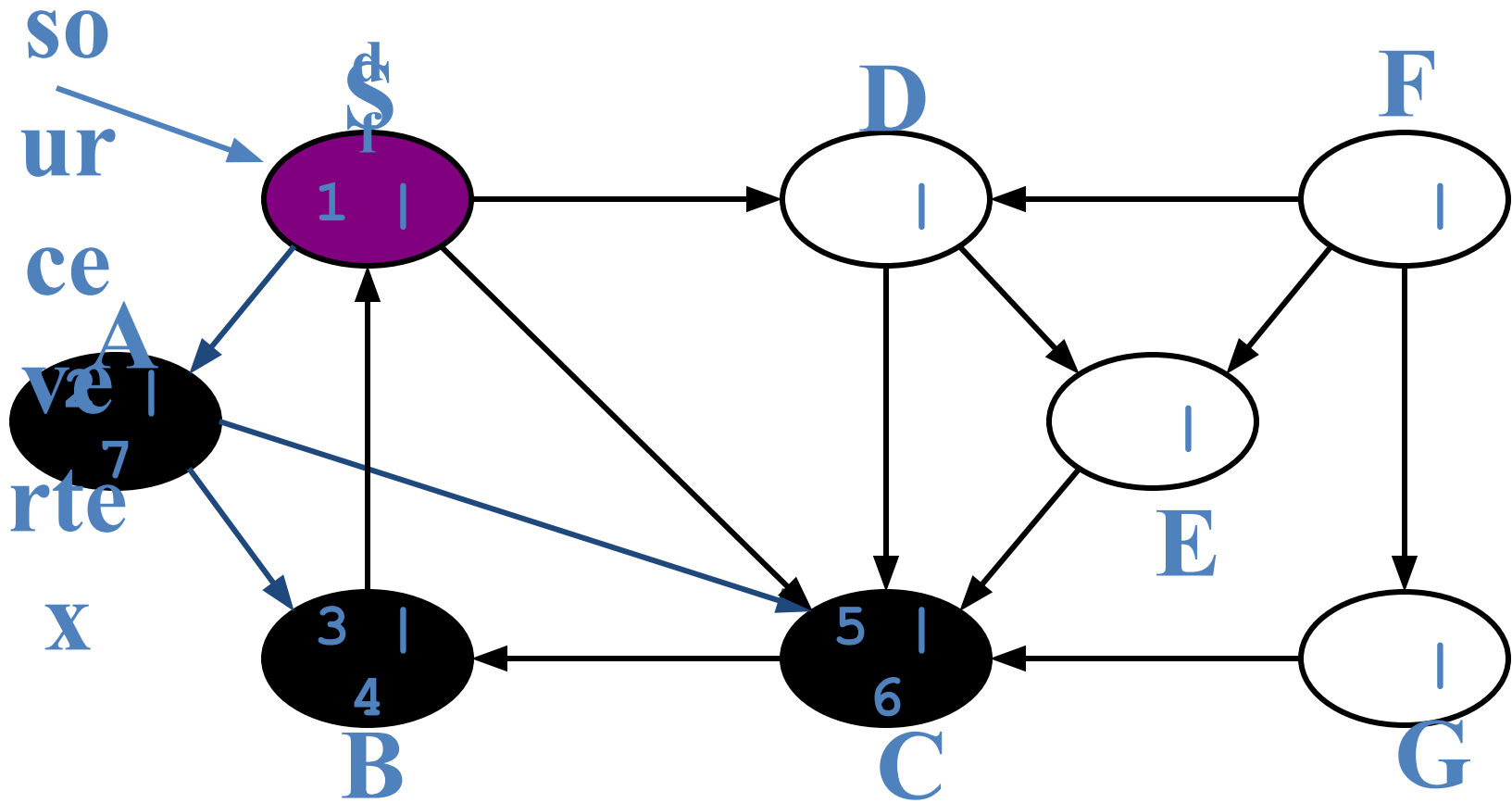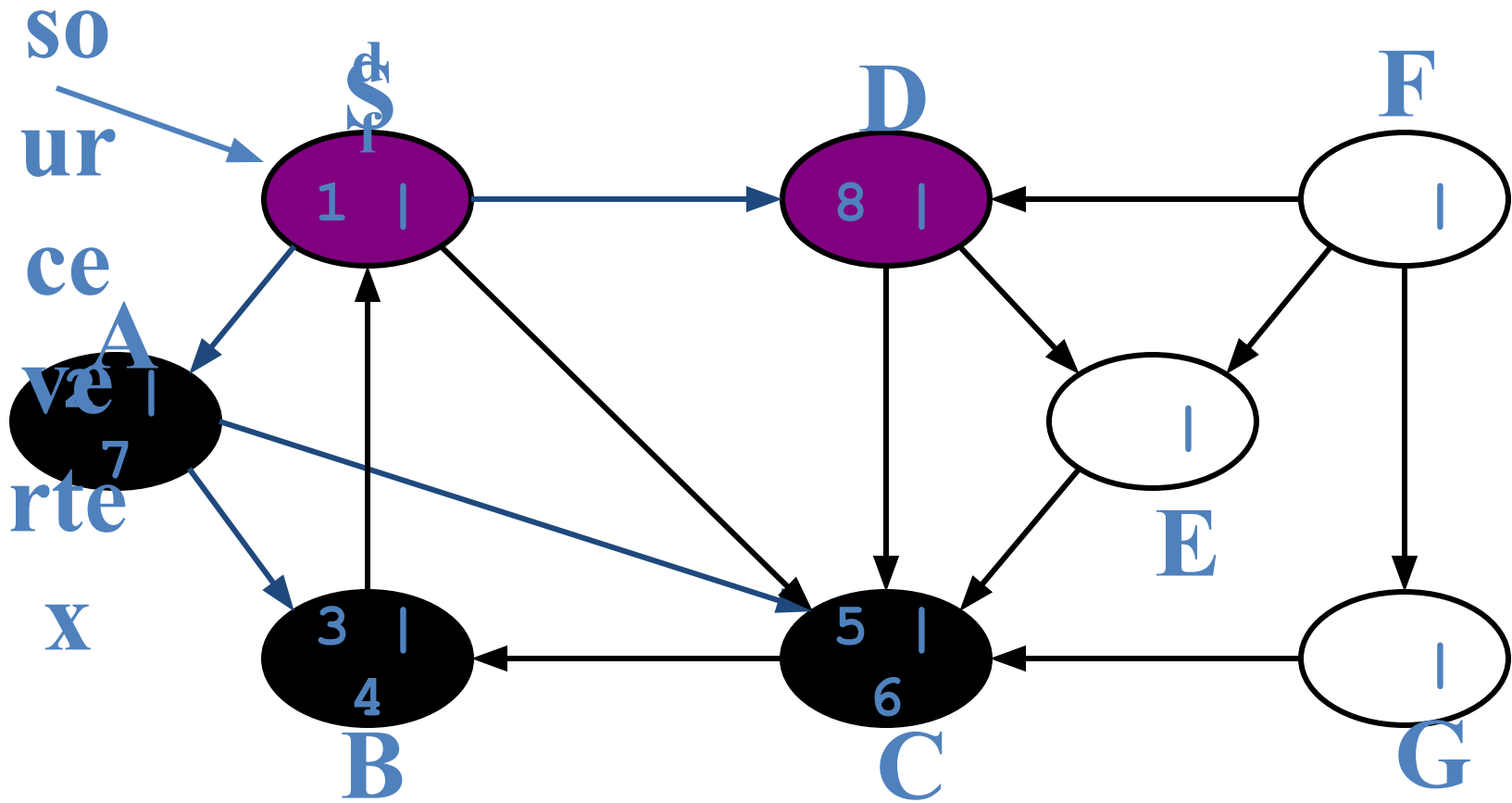


source vertex

S d f
1 |

A d f
2 |

B 3 |
4

D |

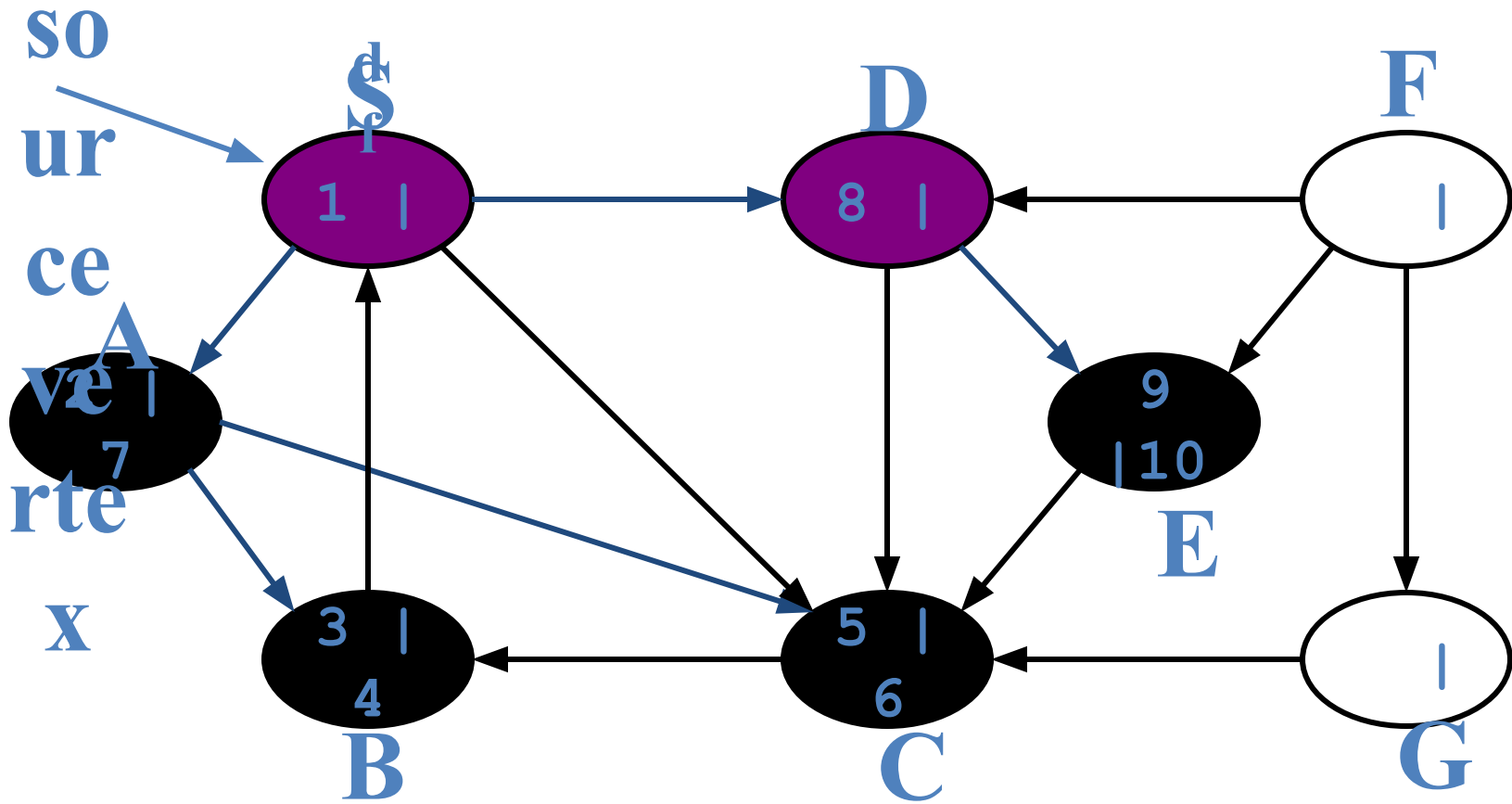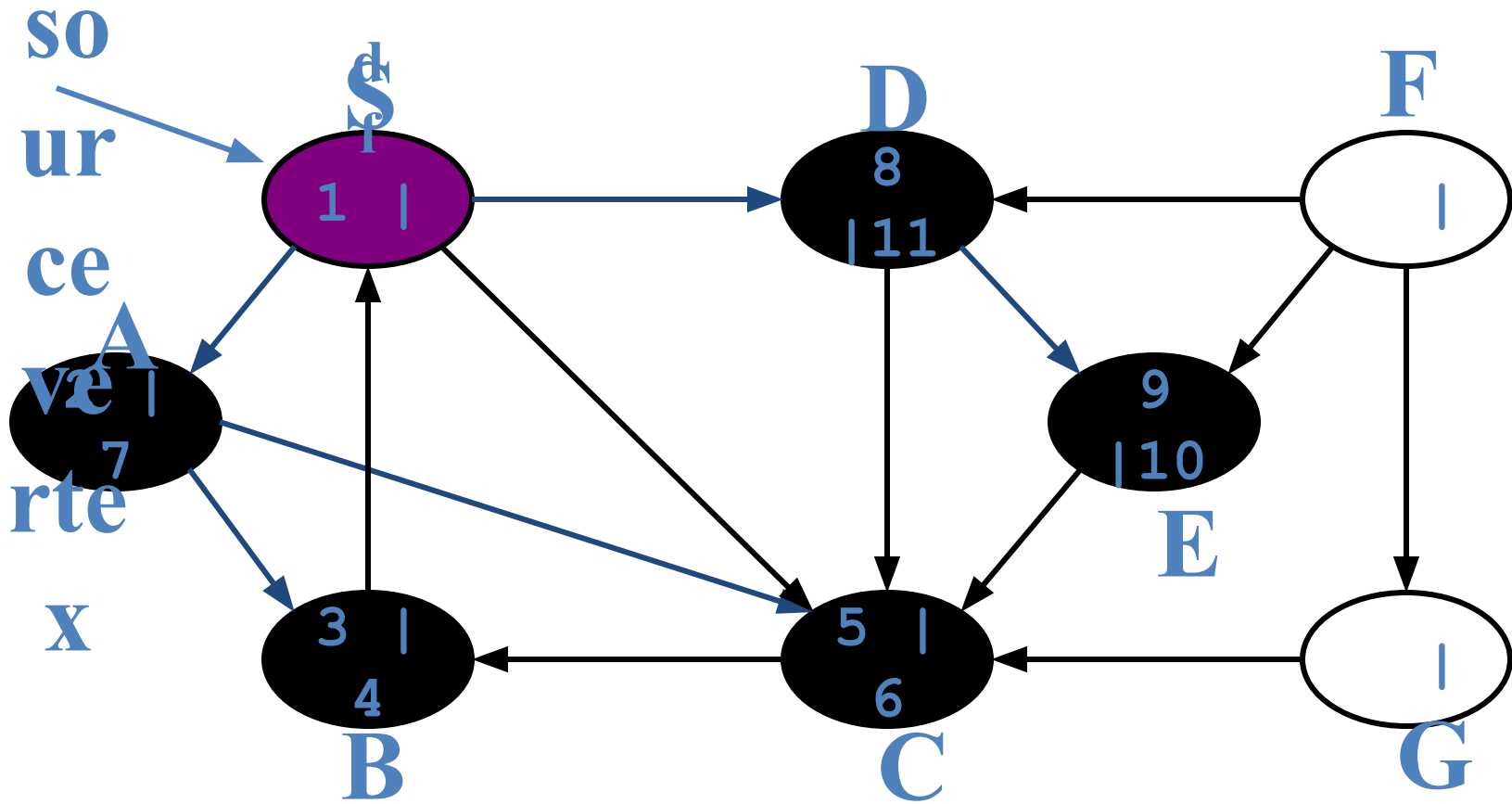E |

F |

C 5 |

G |

# DFS Example

# DFS Example

# DFS Example

# DFS Example



source

d f
**S**
1 |

**D**
8 |

**F**
|

**A**
v e
2 |
7

**B**
3 |
4

**C**
5 |
6

**E**
9 |

**G**
|

**What is the structure of the grey vertices?**
**What do they represent?**

# DFS Example

# DFS Example


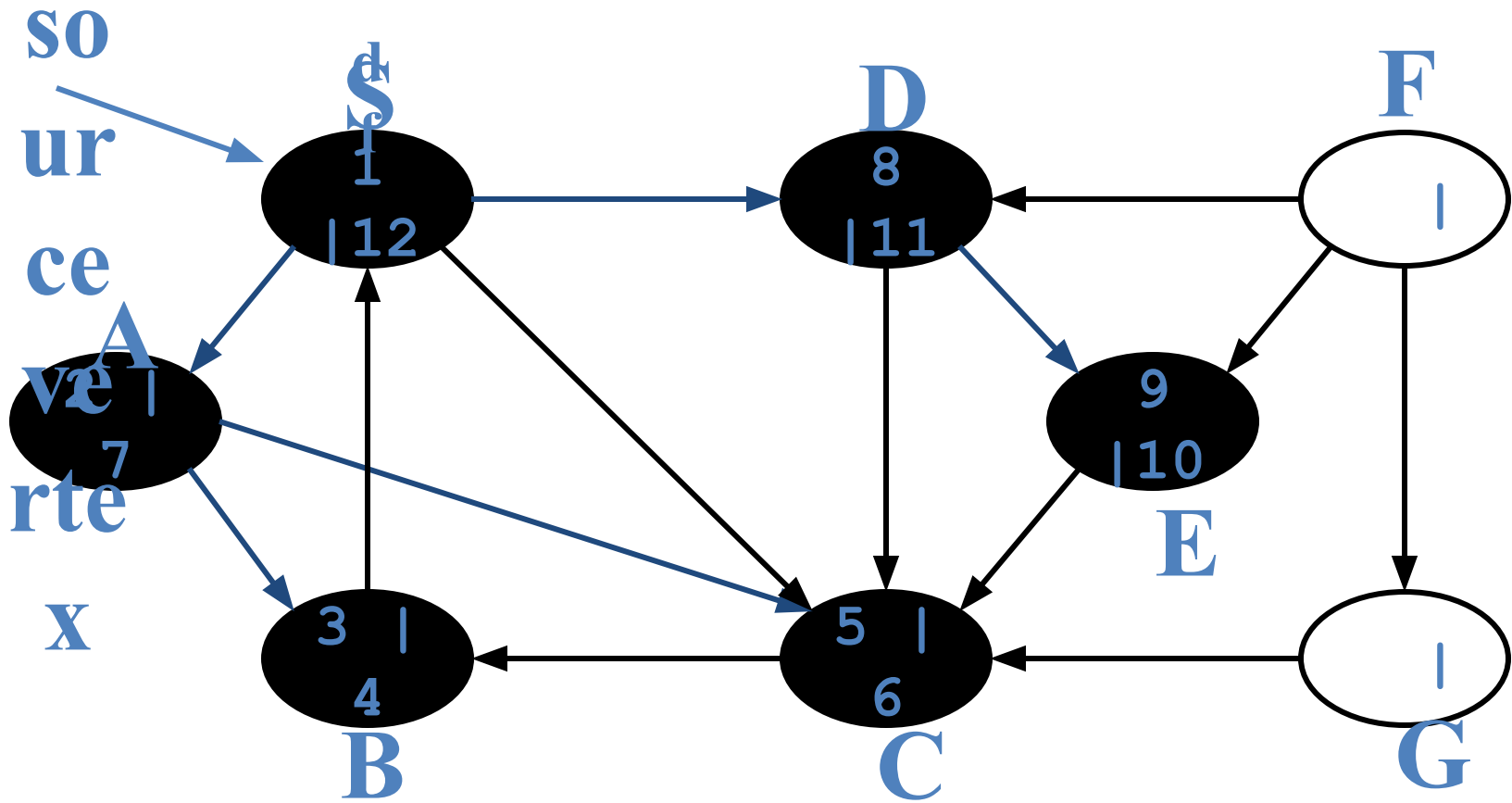
source vertex

S d | f  1 |

A 2 | 7

B 3 | 4

D 8 | 11

E 9 | 10

C 5 | 6

F |

G |

# DFS Example

# DFS Example

# DFS Example

# DFS Example



source vertex

S d 1 |12
A e d 2 | 7
B 3 | 4
D 8 |11
E 9 |10
F 13|
C 5 | 6
G 14|15

# DFS Example



source vertex

S d 1 |12

A e 2 | 7

B 3 | 4

D 8 |11

E 9 |10

C 5 | 6

F 13| 16

G 14| 15
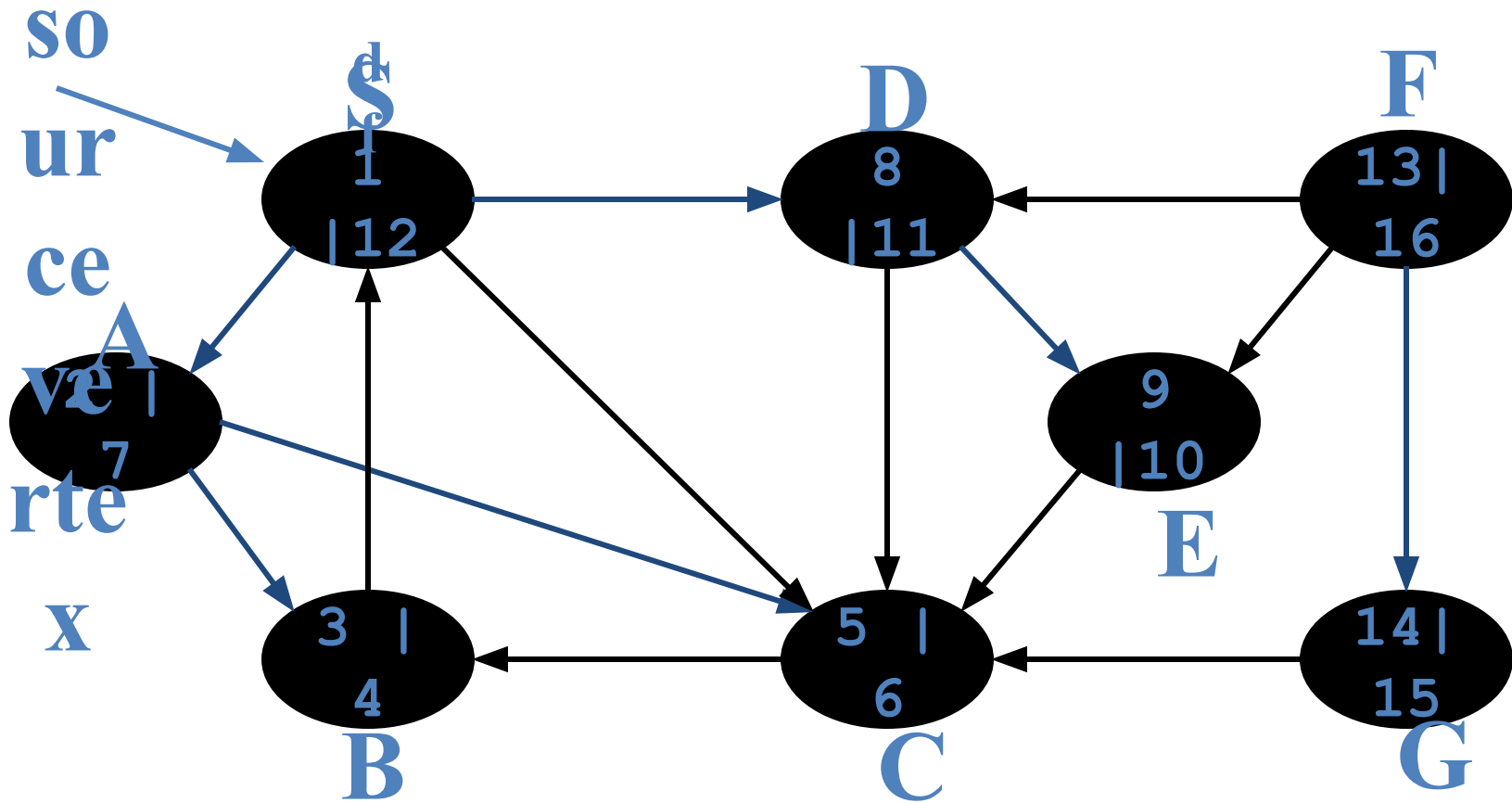
# Depth-First Search: The Code

```
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);
}
```

O(V)

O(V)

```
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;
}
```

O(V)

**Running time: O($V^2$) because call DFS_Visit on each vertex, and the loop over Adj[] can run as many as |V| times**

# Depth-First Search: The Code

```
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;
}
```

**BUT, there is actually a tighter bound.**
**How many times will DFS_Visit() actually be called?**

# Depth-First Search: The Code

```
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;
}
```

**So, 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
  - Important to be comfortable with this kind of reasoning and analysis

# 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?*
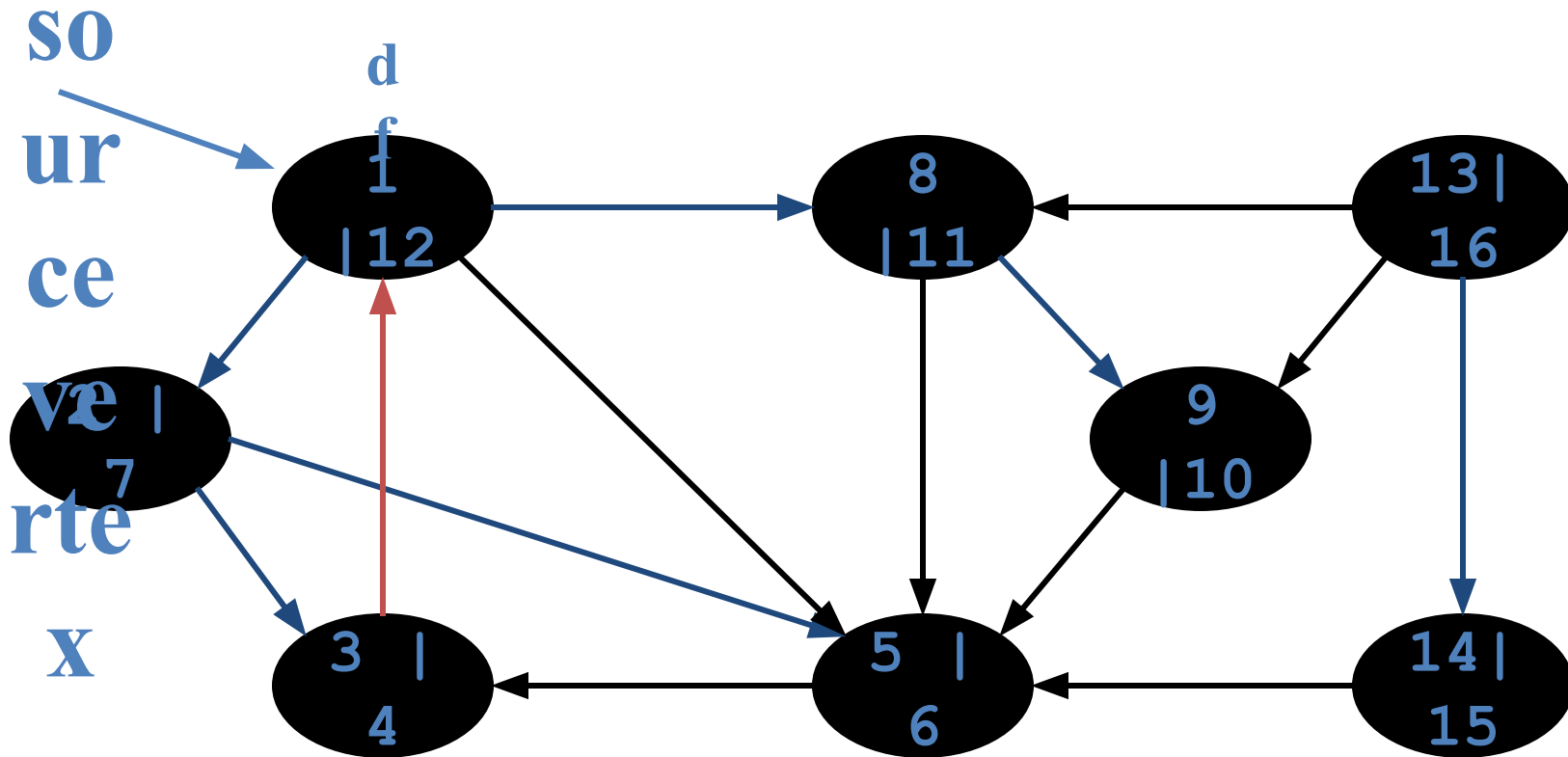      - *No*

# DFS Example
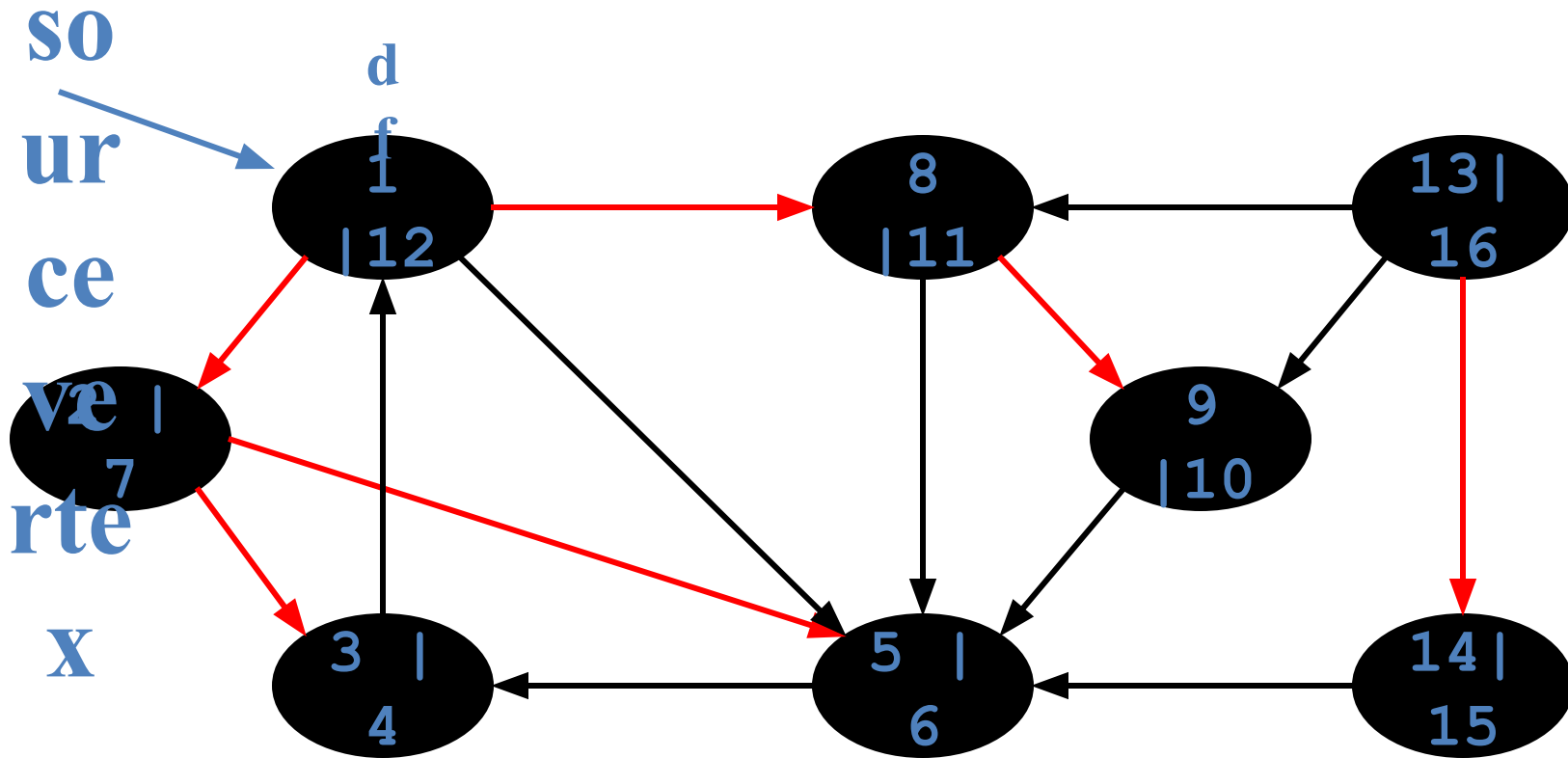


**Tree edges**

# 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.
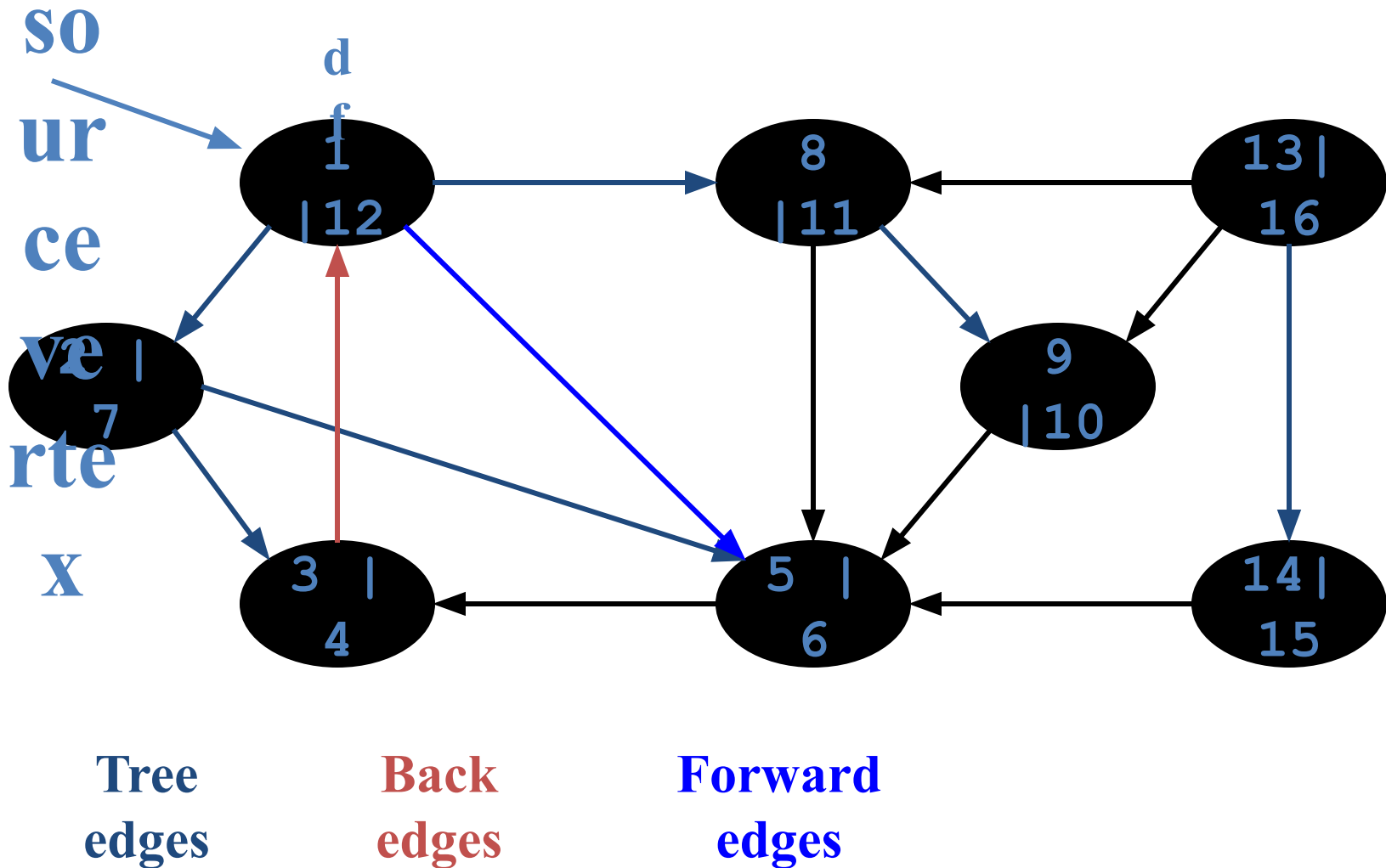
# DFS Example



source vertex

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
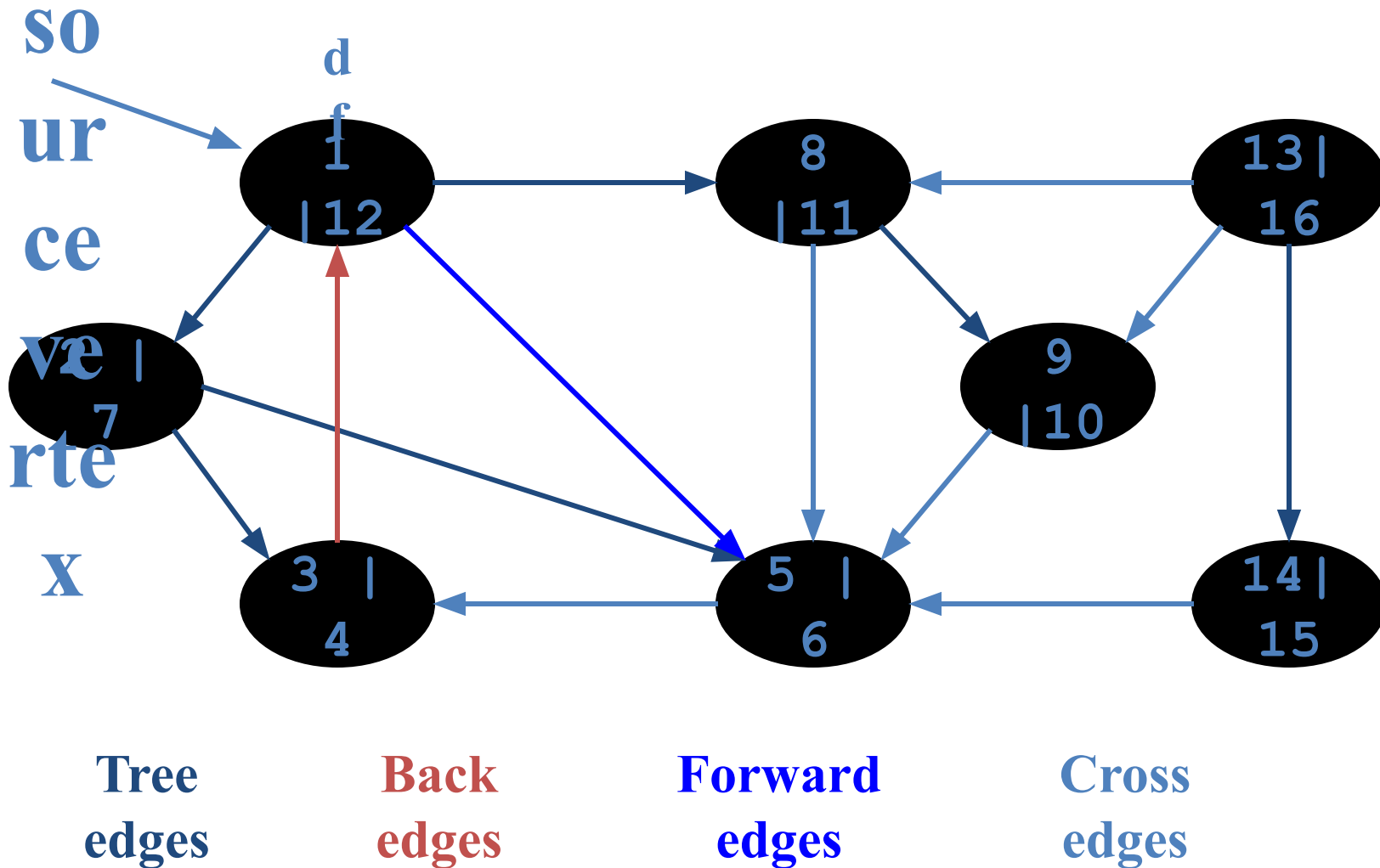    - From grey node to black node

# DFS Example



source vertex

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

# 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
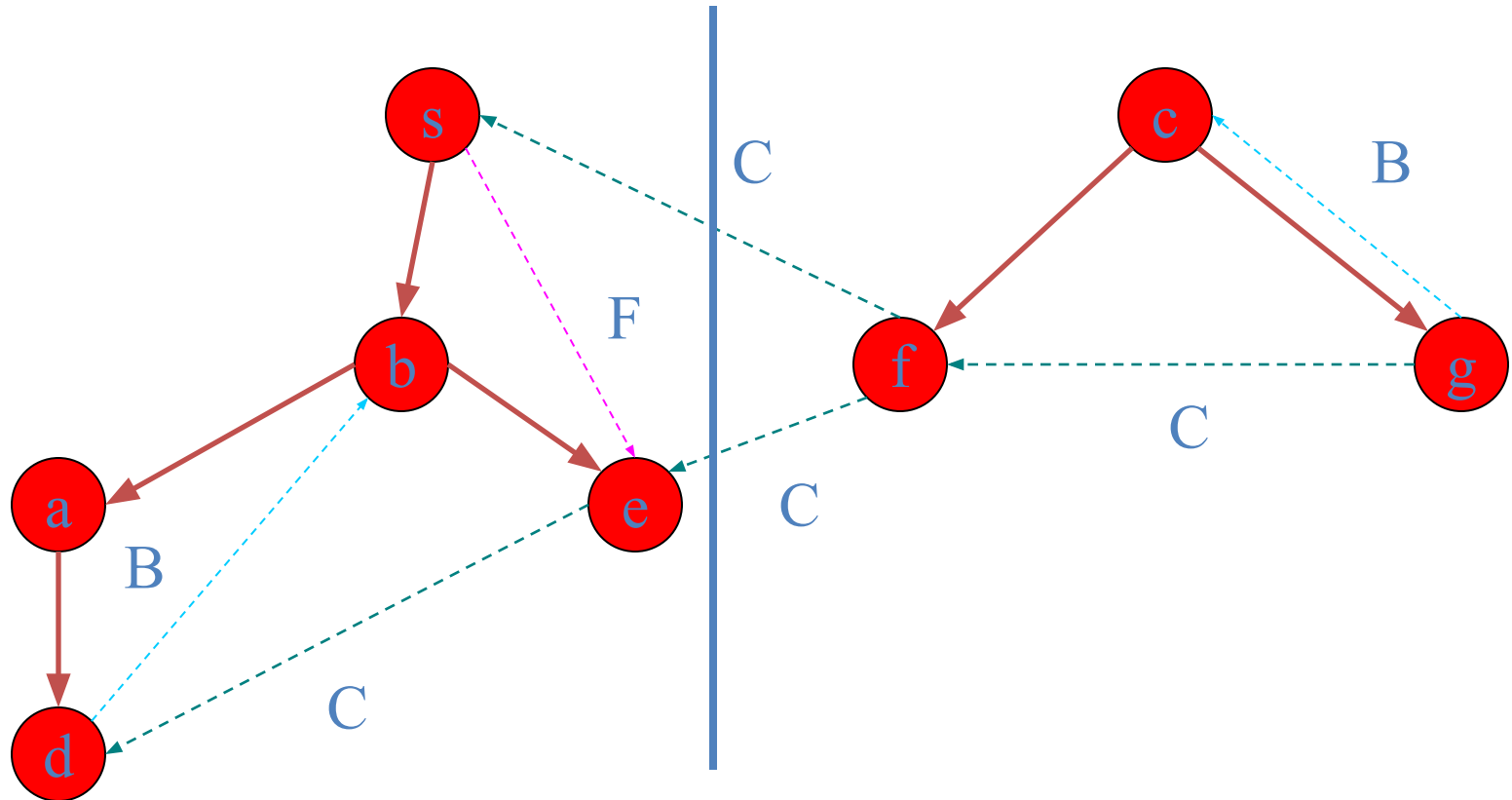
# More about the edges

- 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
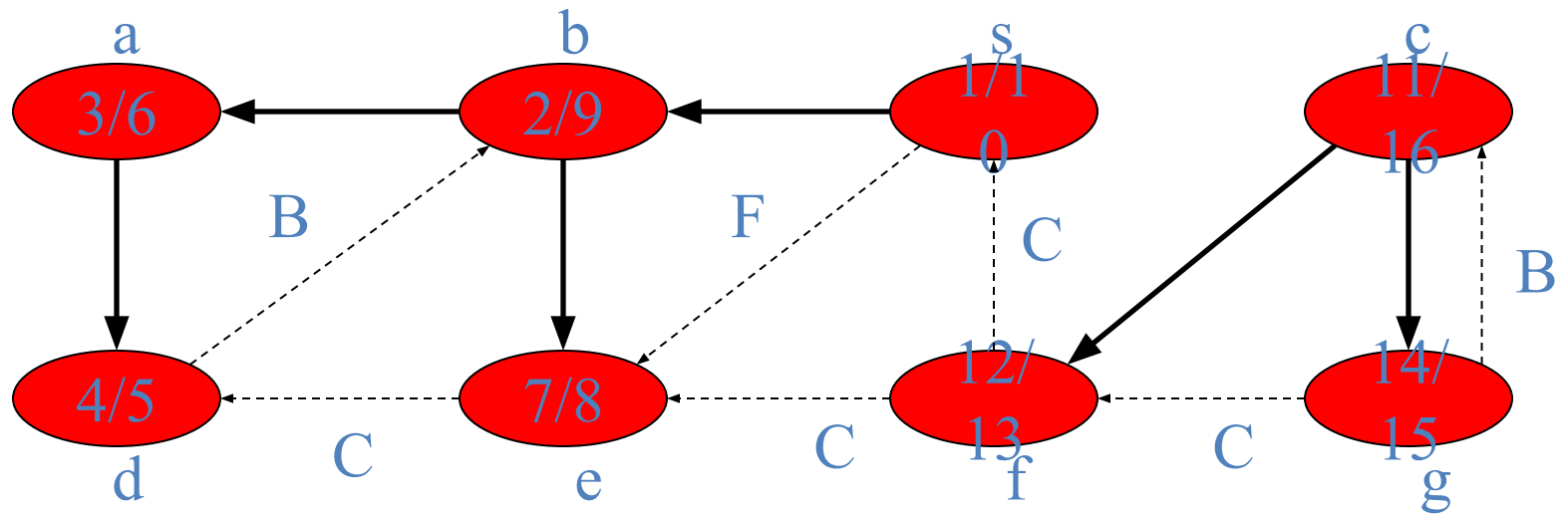
# Depth-First Search - Timestamps

# 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);

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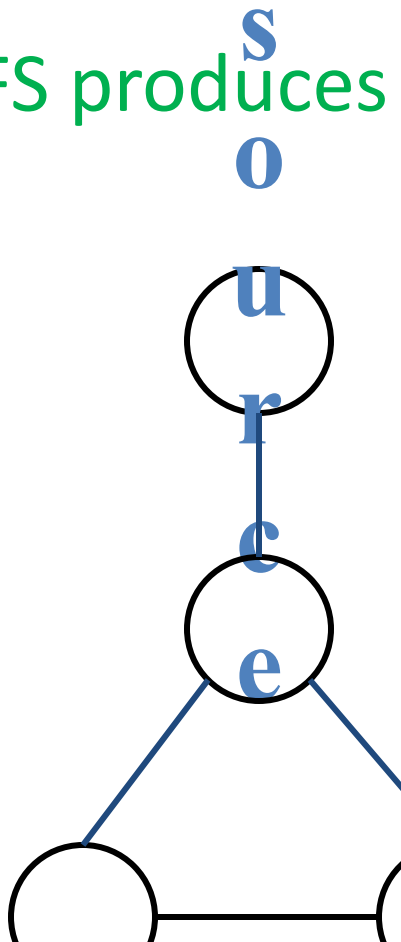
```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
    detect edge type using
    "color[v]"
        if(color[v] == WHITE){
      prev[v]=u;
          DFS_Visit(v);
    }}
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

# 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?*)
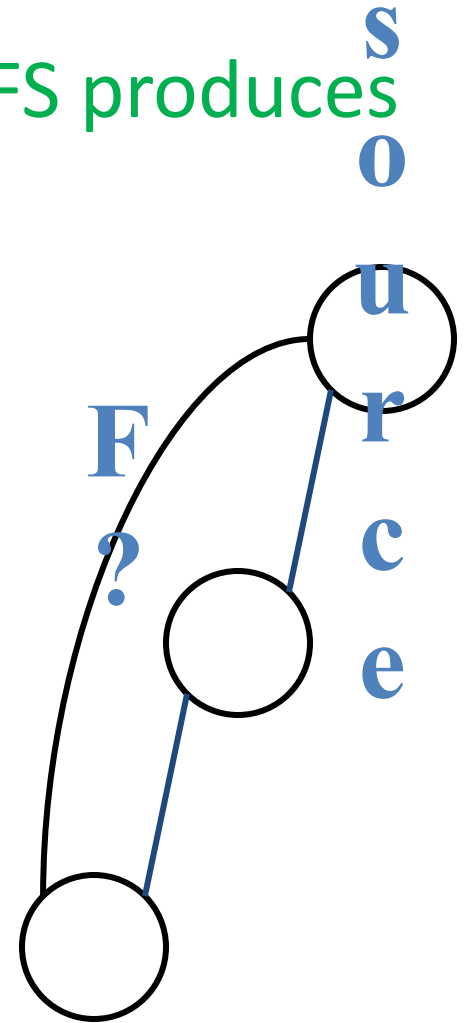
**source**

**F?**

# 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 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

**s
o
u
r
c
e**

**C
?**
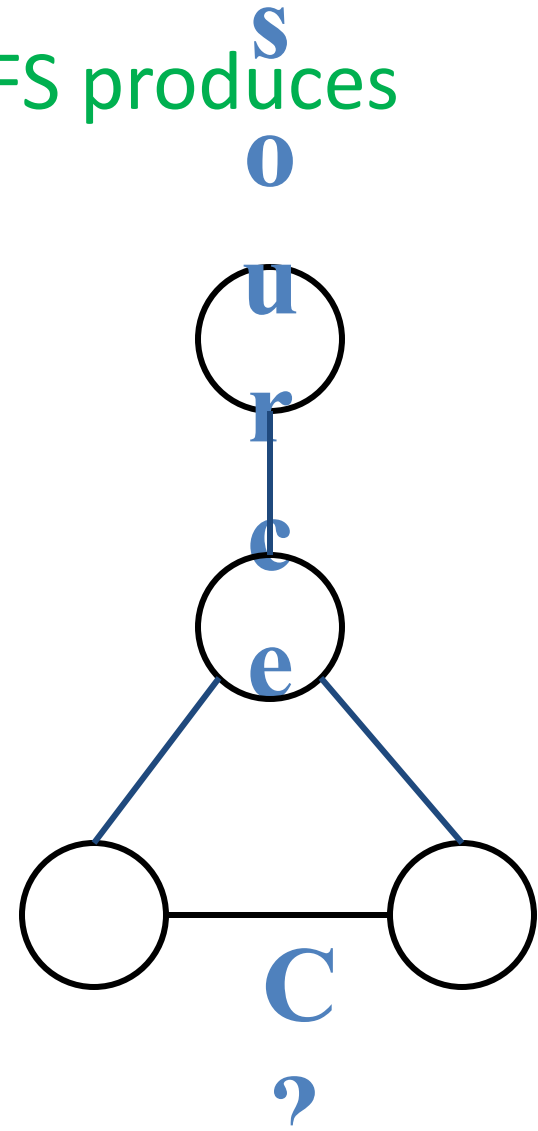
# DFS And Graph Cycles

- 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
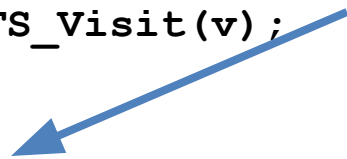- Thus, can run DFS to find whether a graph has a cycle

# 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;
}
```
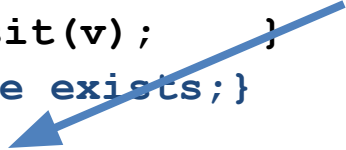
# DFS And Cycles

## What will be the running time?

```
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 for undirected graph to detect cycle?*
- A: O(V+E)
- We can actually determine if cycles exist in O(V) time
  - How??

# 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)

# Exercises on DFS

- CLRS Chapter 22 (Elementary Graph Algorithms)

- Exercise: (Page
  - 22.3-5 –Detect edge using d[u], d[v], f[u], f[v]
  - 22.3-12 – Connected Component
  - 22.3-13 – Singly connected

# Some applications of BFS and DFS

- Topological Sort (Topic of Next Lecture)
- Euler Path (Topic of Next Lecture)
- Dictionary Search
- Mathematical Problem
- Grid Traversal

# The idea of State/Node

- Parameters describing a scenario
- Useless Parameters
  - If value of the parameter change doesn't affect the outcome
  - If value of the parameter can be derived from other parameters
- Useful Parameter
  - Not useless!!!

# Example - States

- Grid Problem, direction change takes time
- Grid Problem, blocks alternating