# 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.

# Path between Vertices

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

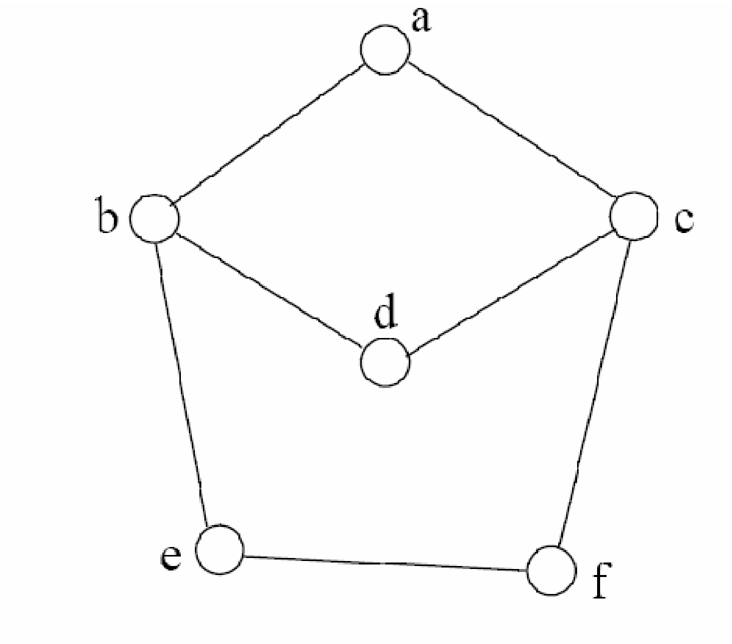
*Note: a path is allowed to go through the same vertex or the same edge any number of times!*

- 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

# Breadth-First Search: Simplified

*create a queue Q*

*v.visited = true*

*Q.push(v)*
*while Q is non-empty*
  *remove the head u of Q*
  *mark and enqueue all (unvisited) neighbours of u*

The time complexity of BFS is O(V+E) because:

1. It visits each vertex once, contributing V to the complexity.
2. It checks each edge once when moving from one vertex to its adjacent vertices, contributing E to the complexity.

# 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***

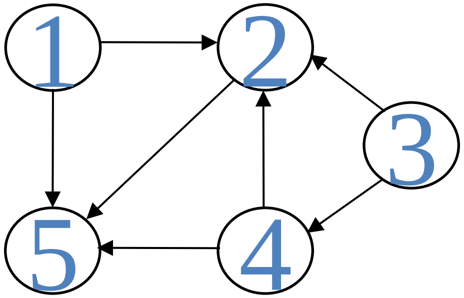
*(Why?)*

O(V)

***What will be the running time?***

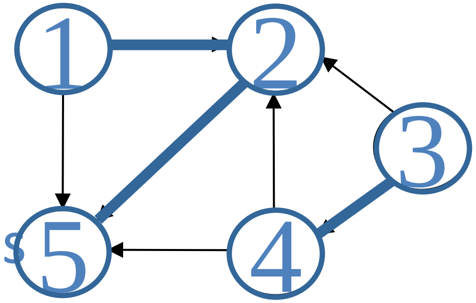
**Total running time: O(V+E)**

# 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

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

0 WHITE d[u] GRAY f[u] BLACK 2V

# 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?**

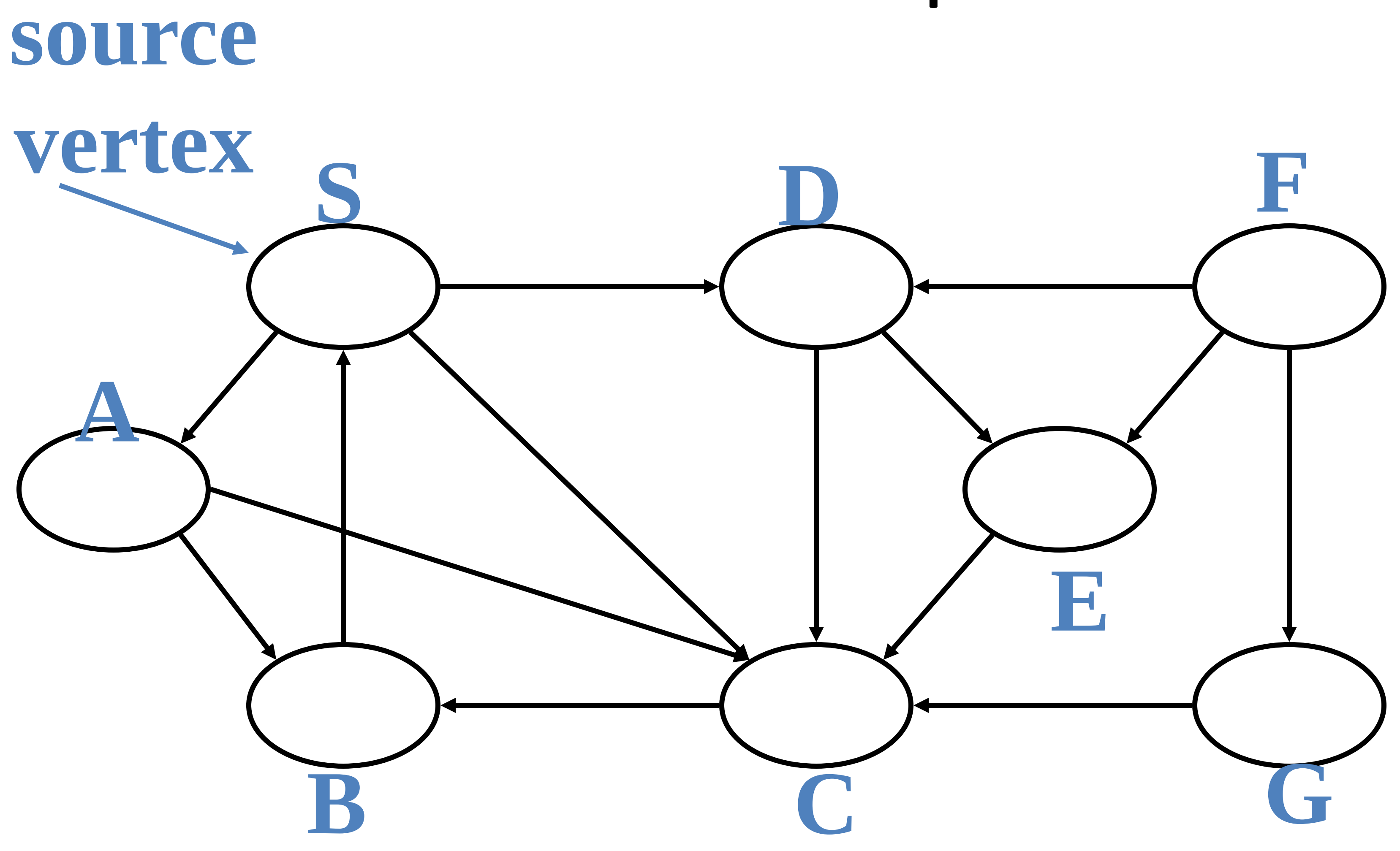
# Depth-First Search: The Code

```
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);
}
```

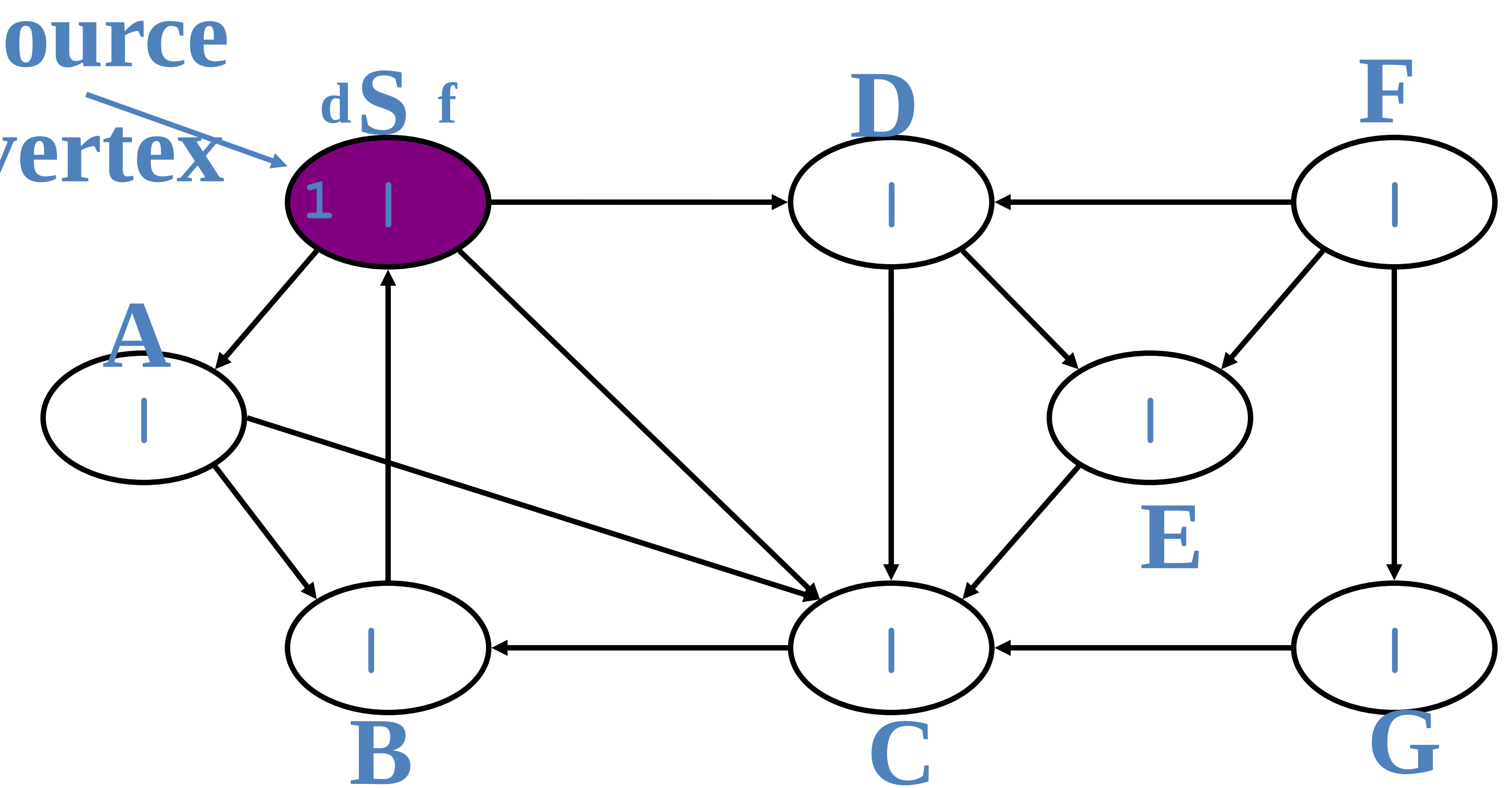
```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
        if(color[v] == WHITE){
            prev[v]=u;
            DFS_Visit(v);
    }}
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

**Will all vertices eventually be colored black?**

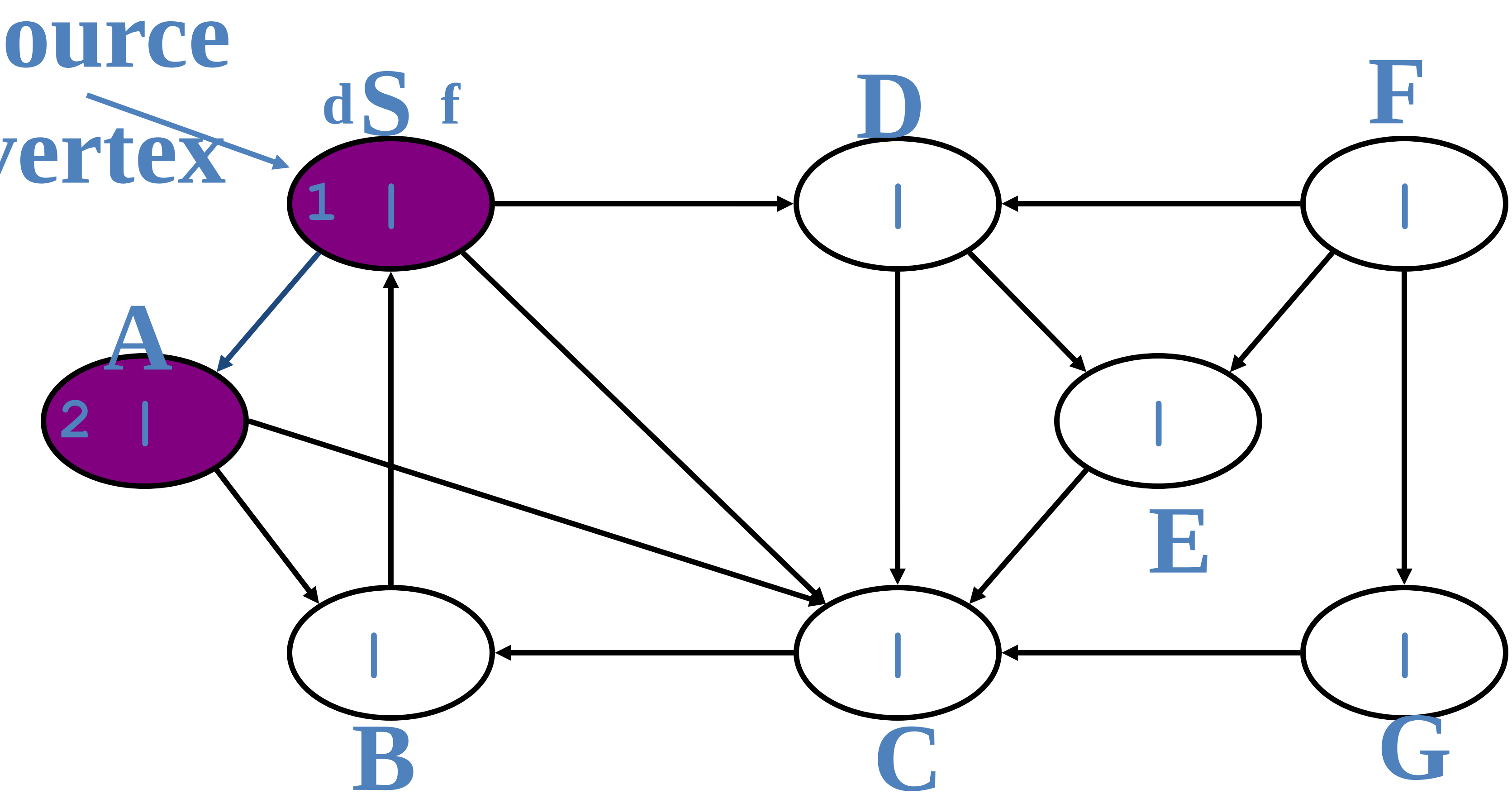
# DFS Example



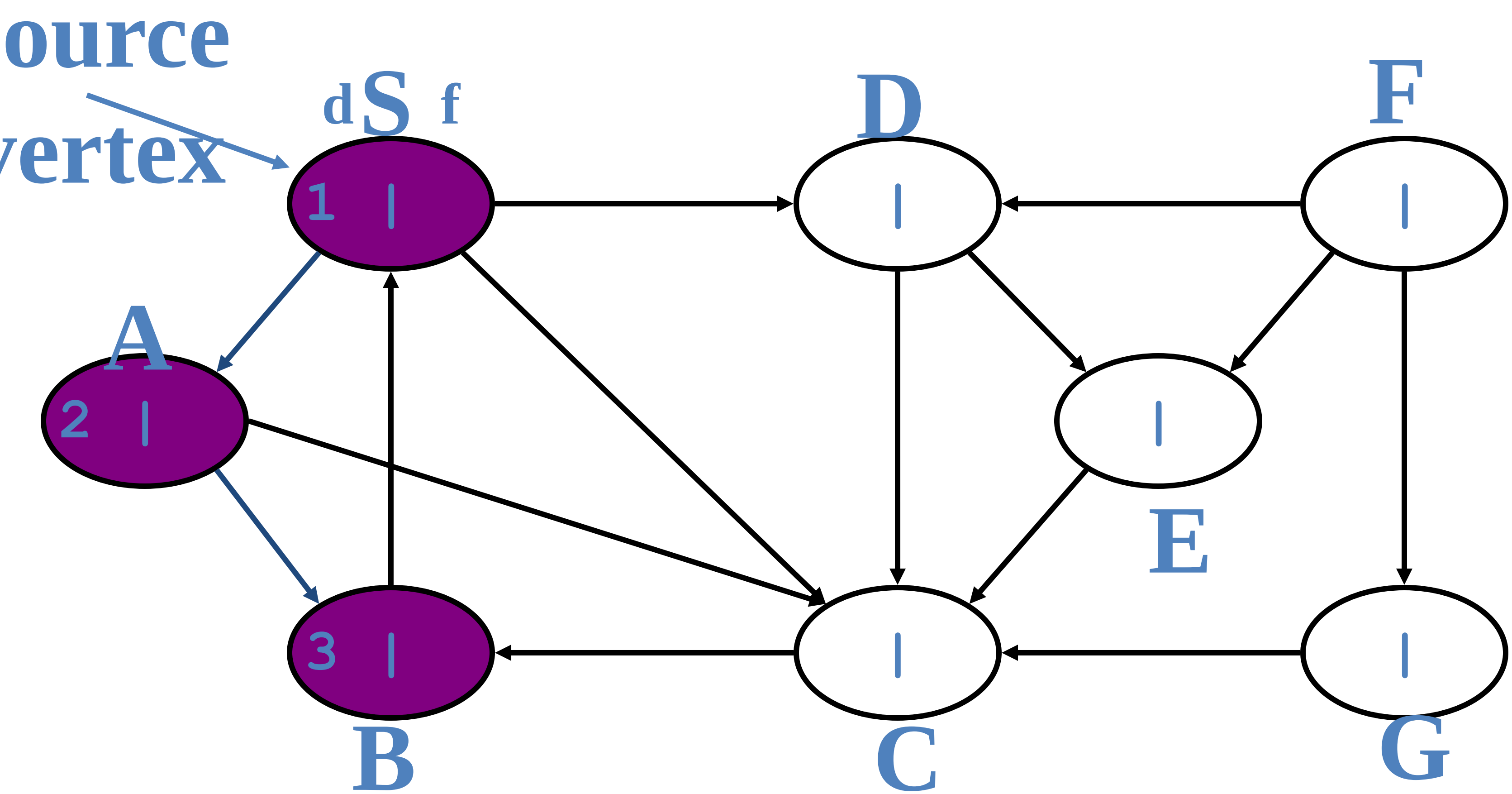
# DFS Example



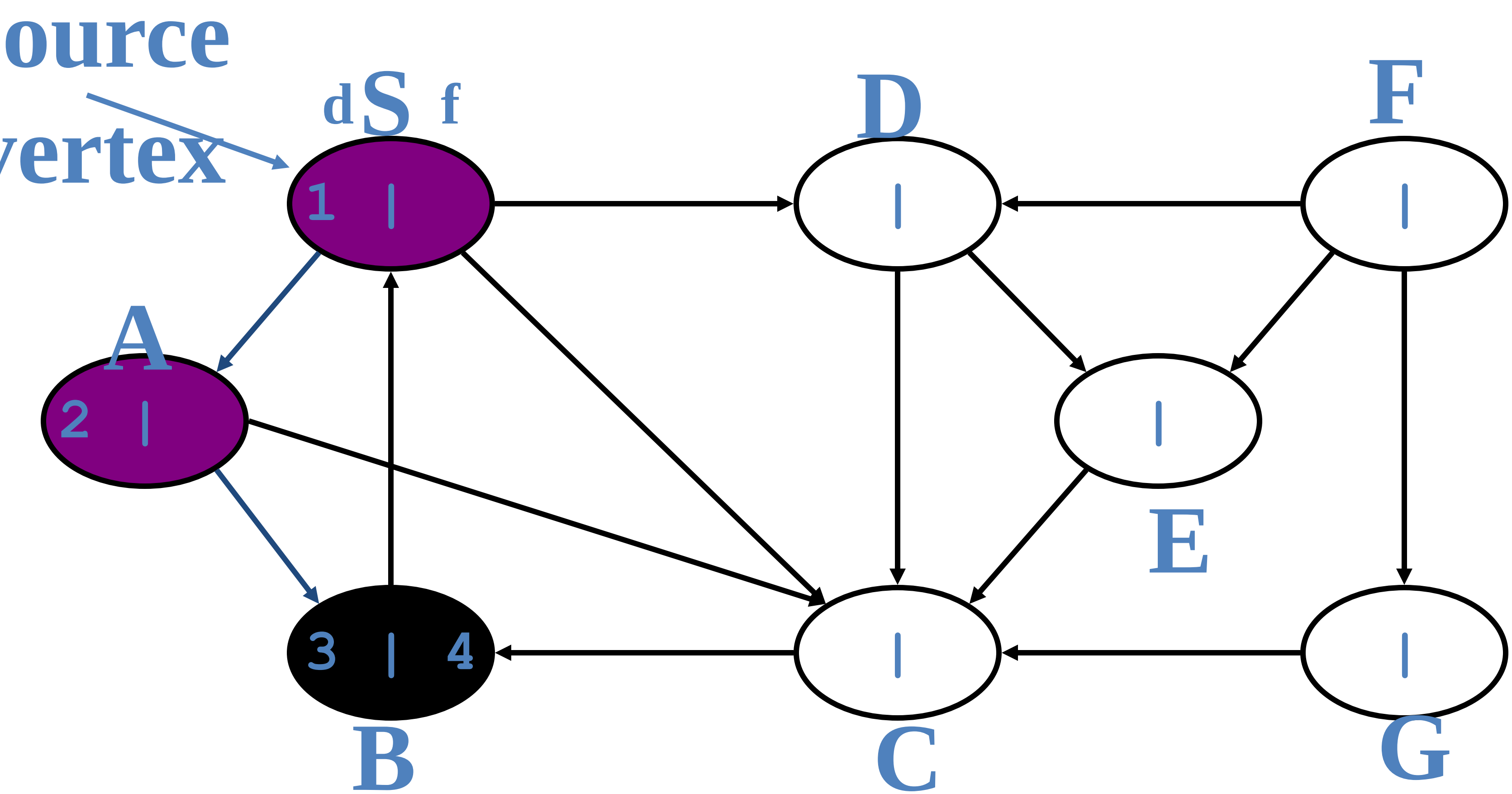
# DFS Example



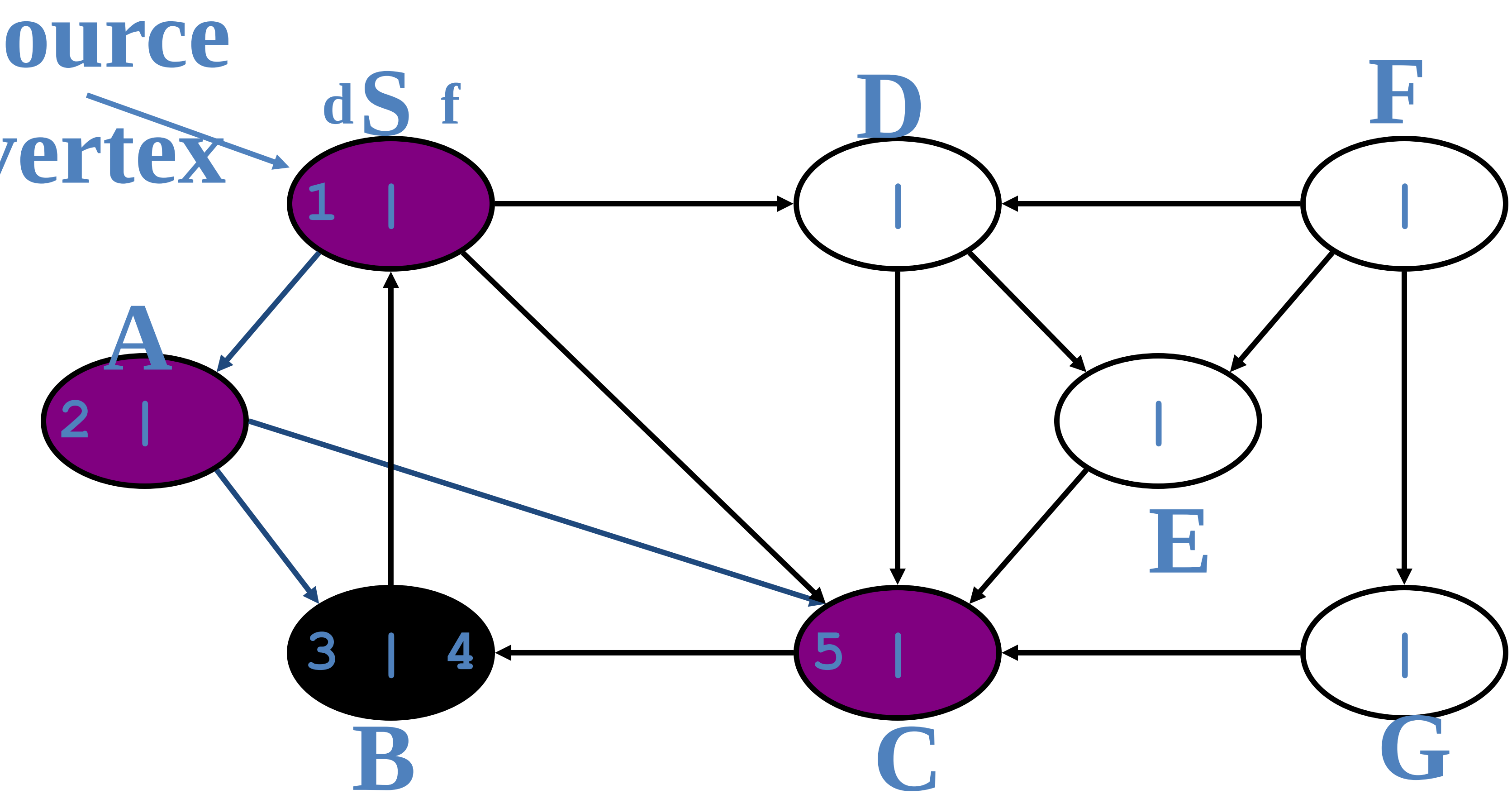
# DFS Example



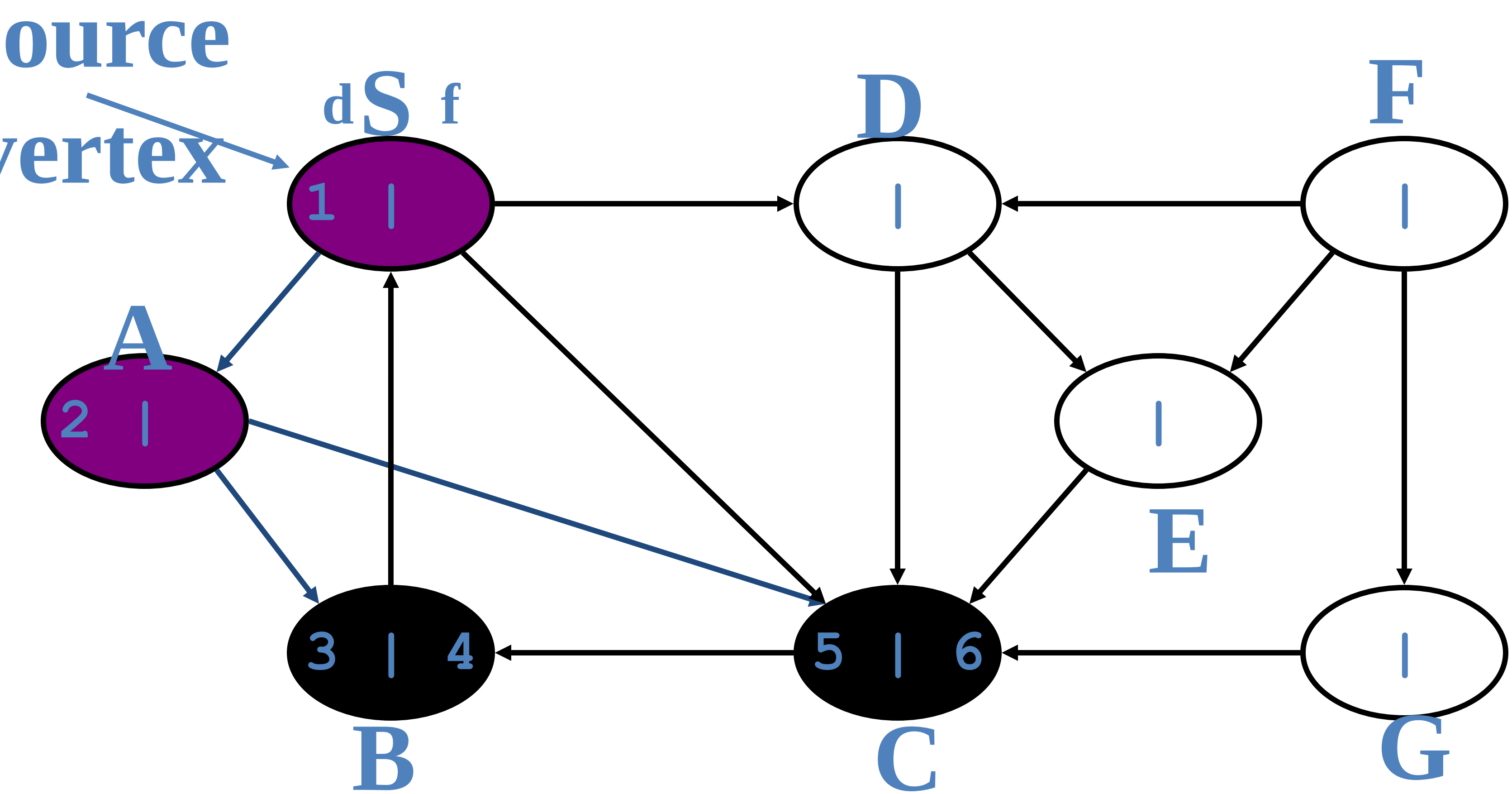
# DFS Example



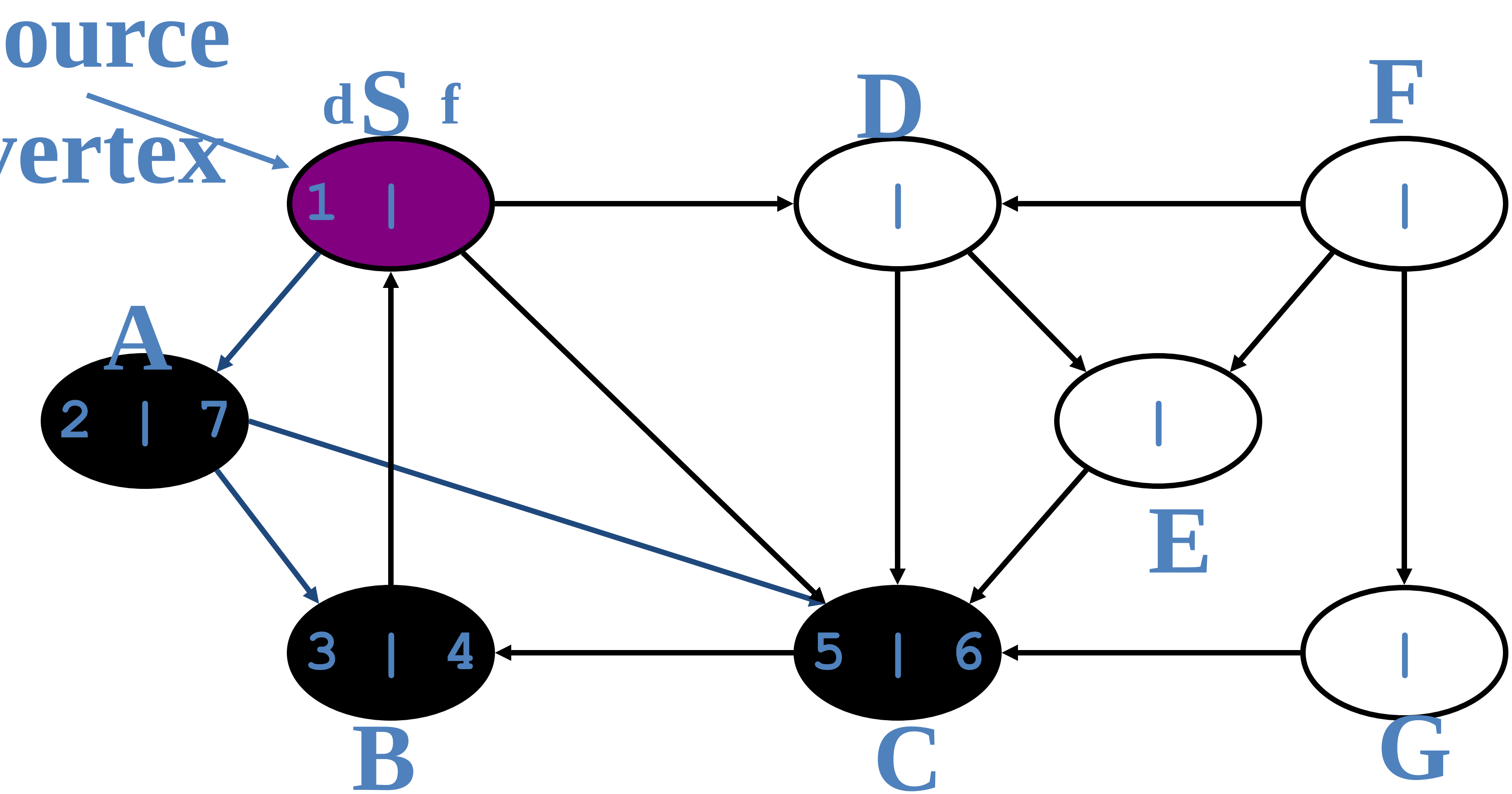
# DFS Example



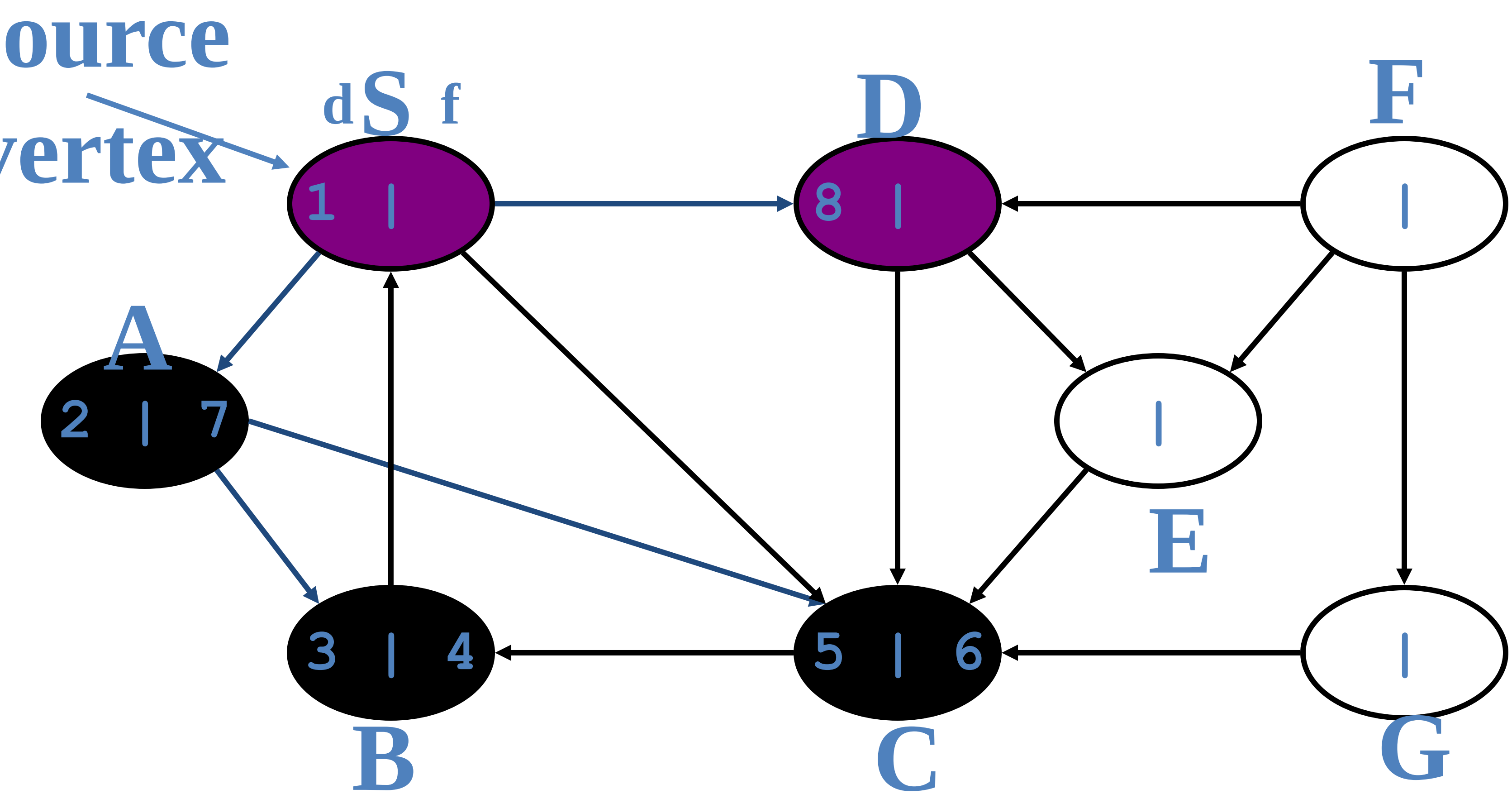
# DFS Example



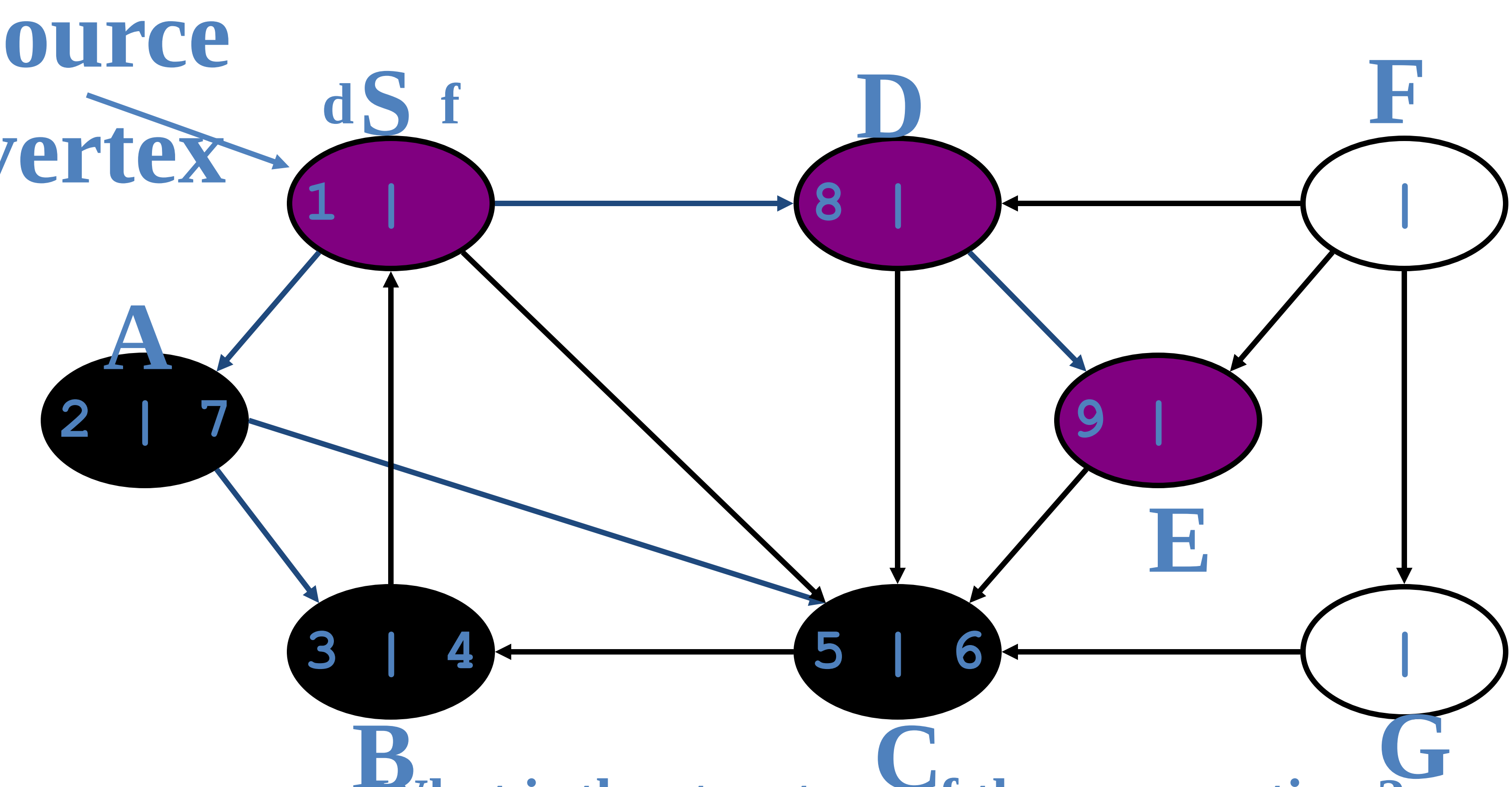
# DFS Example



# DFS Example

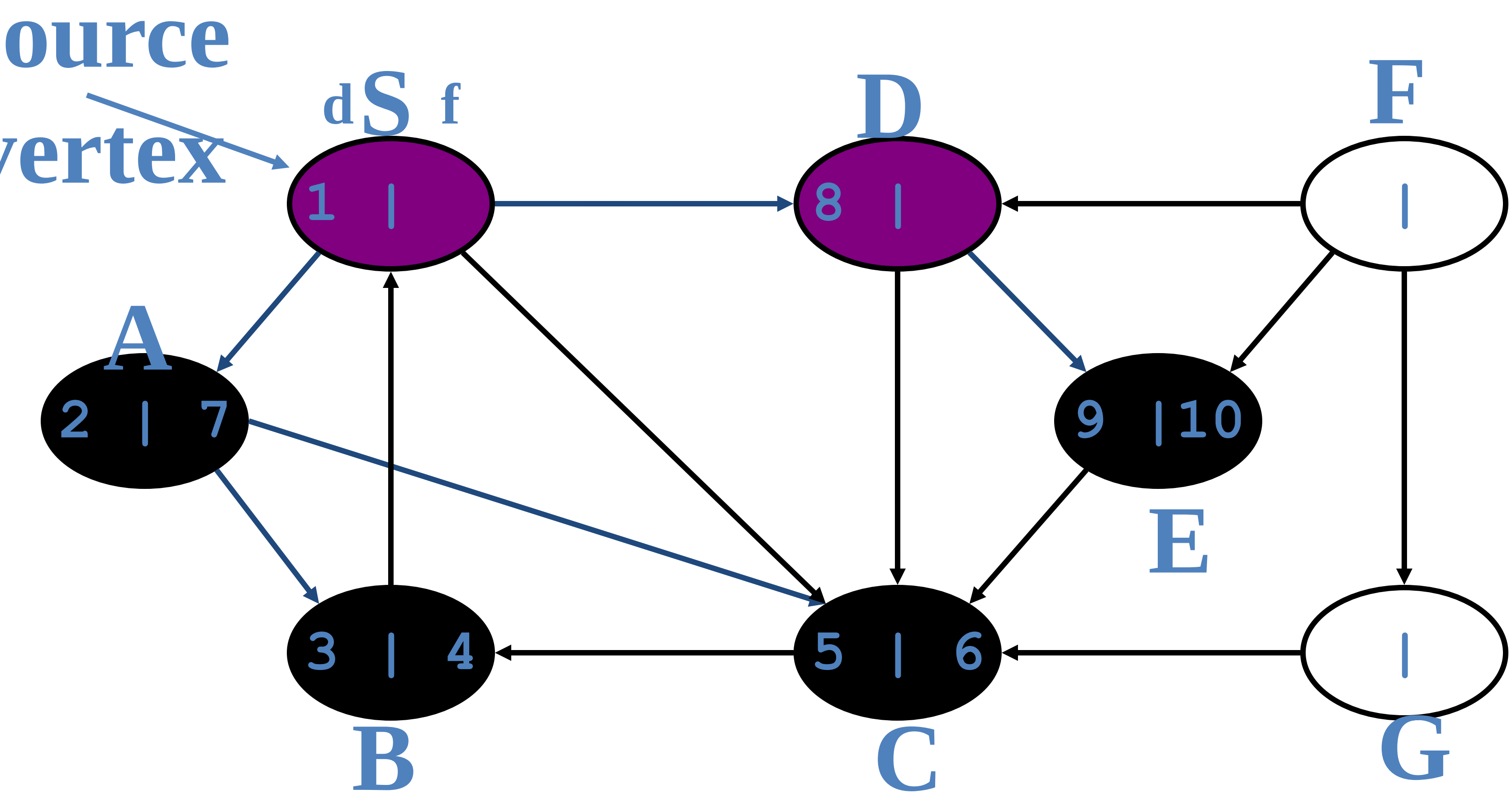


# DFS Example

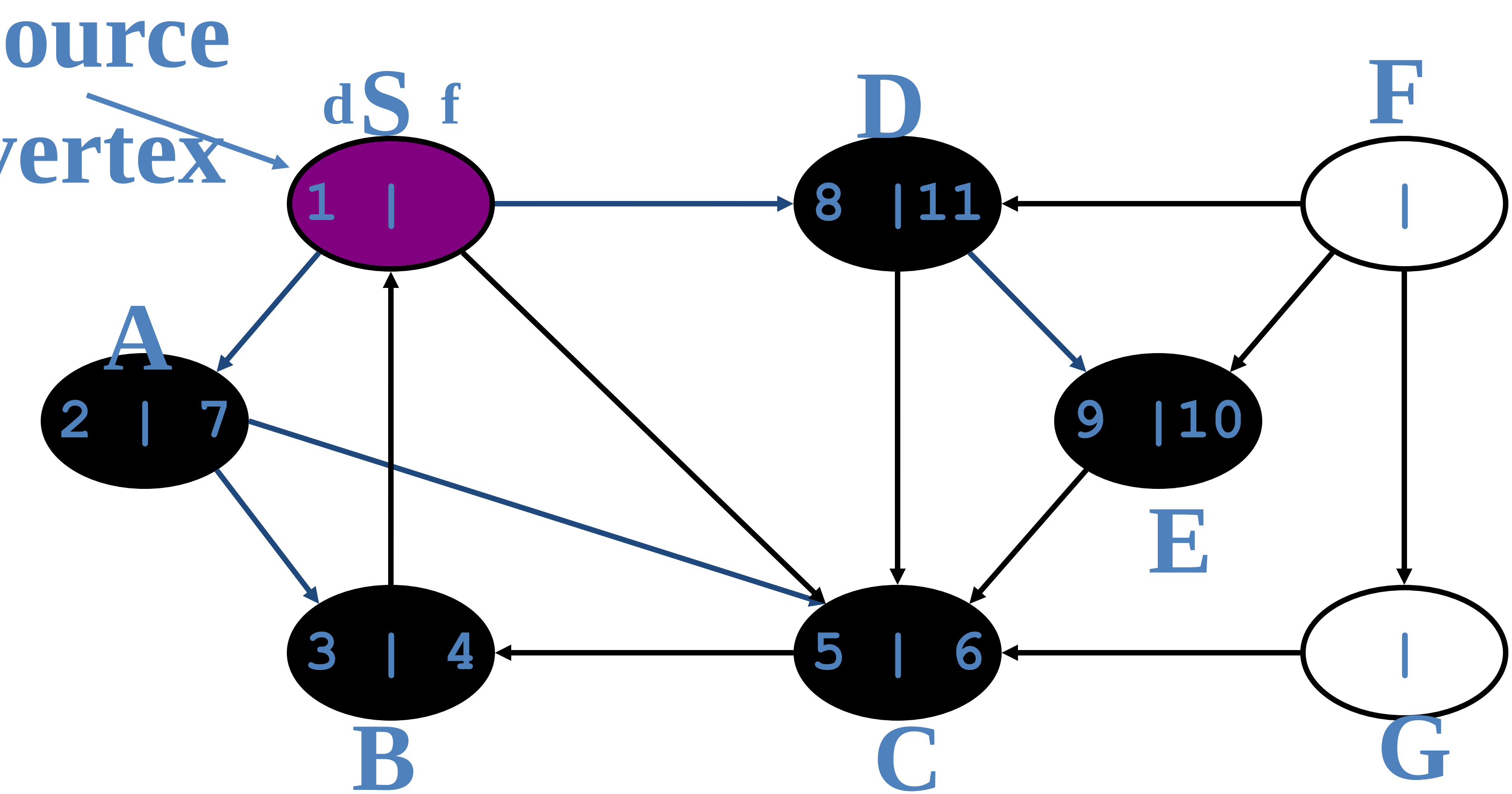



**What is the structure of the grey vertices?**
**What do they represent?**

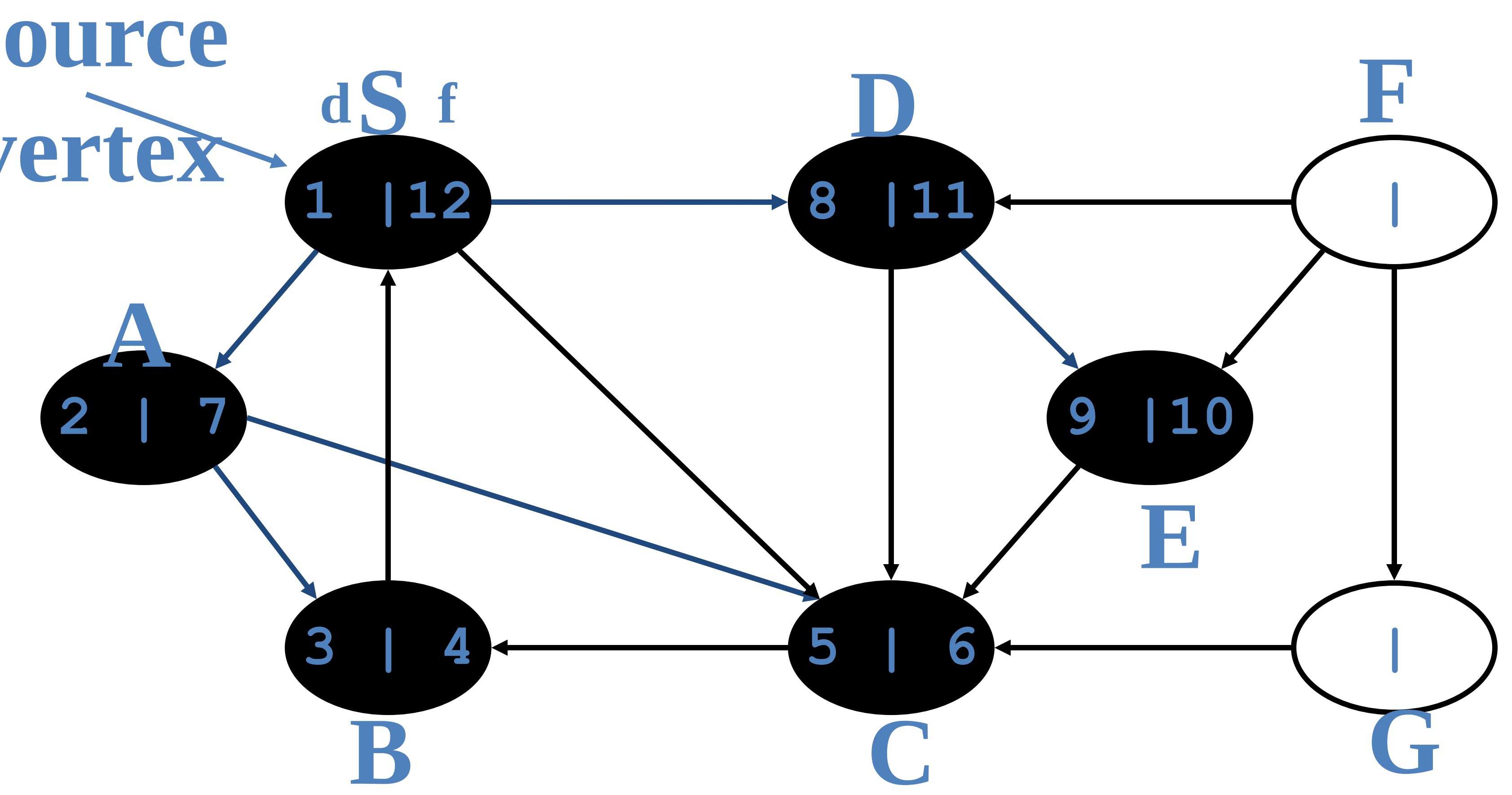
# DFS Example



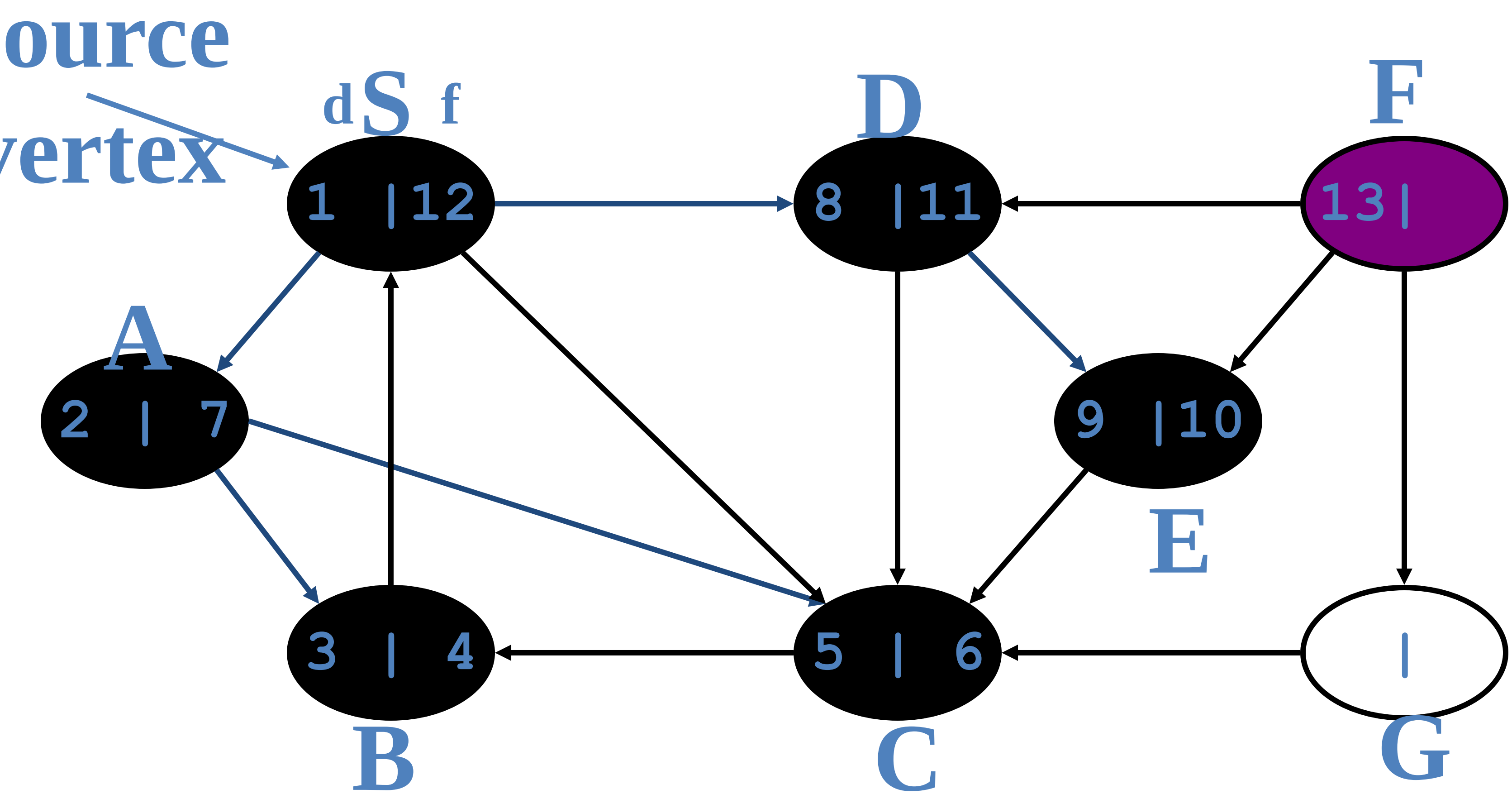
# DFS Example



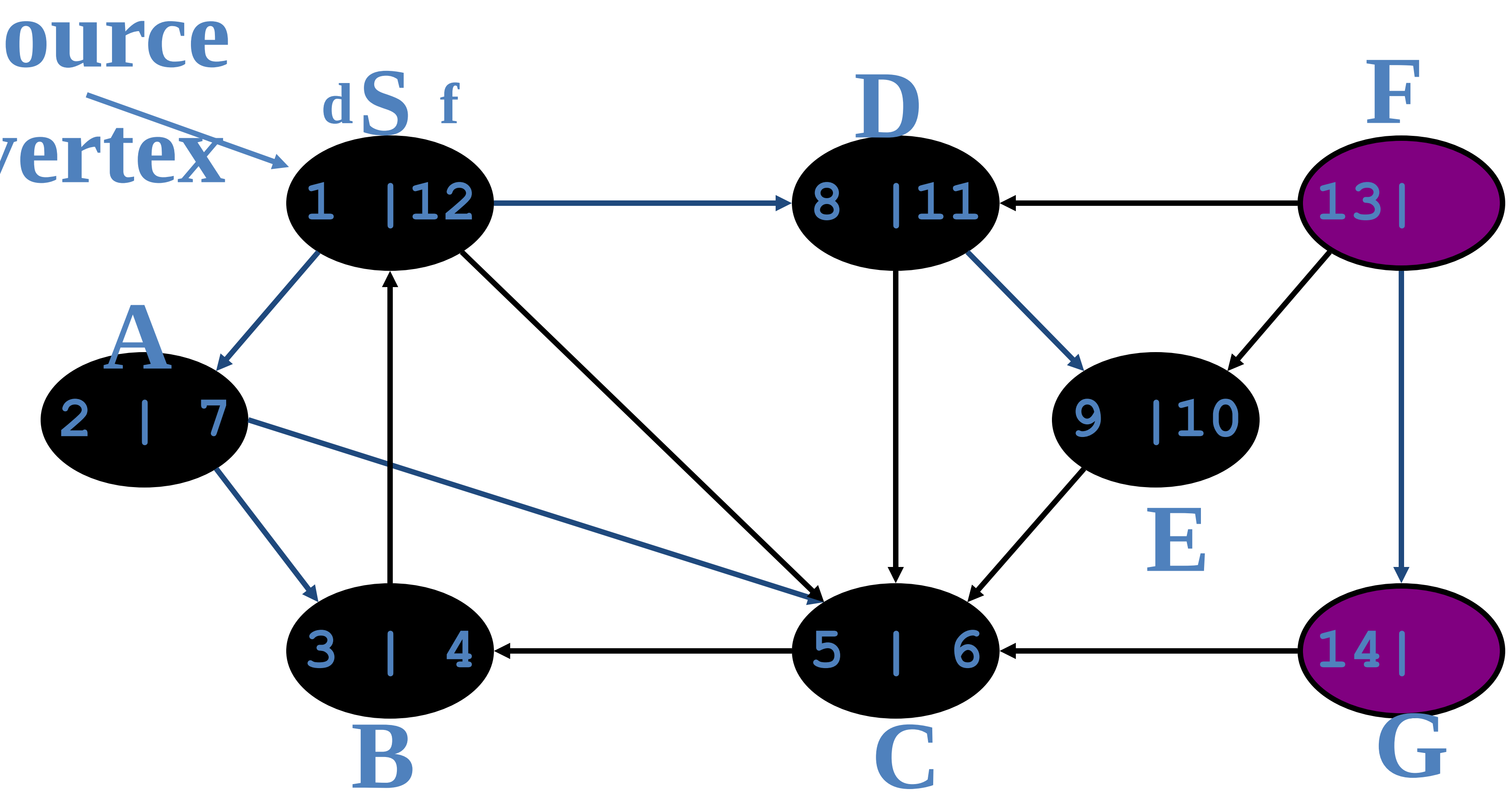
# DFS Example



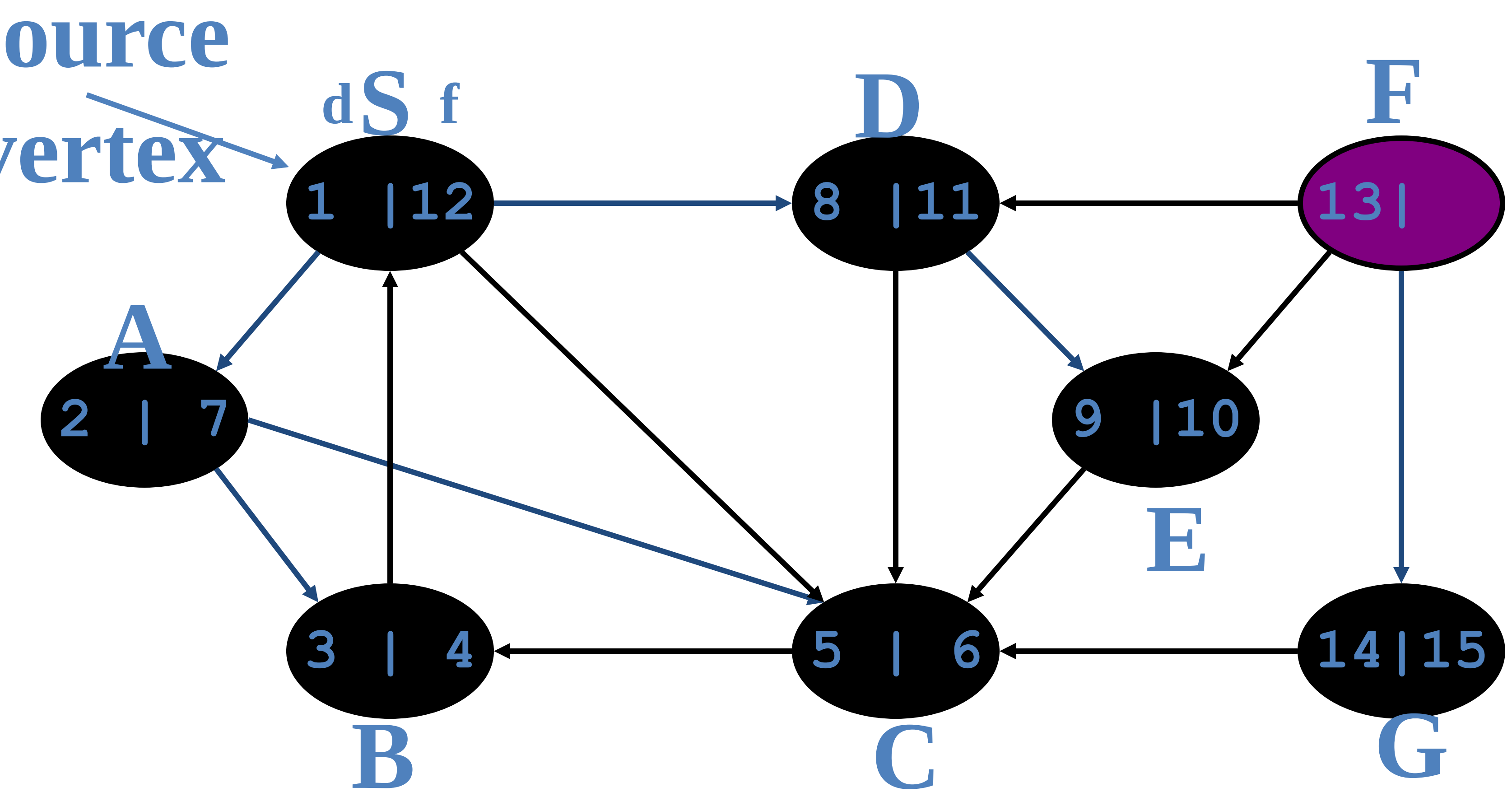
# DFS Example



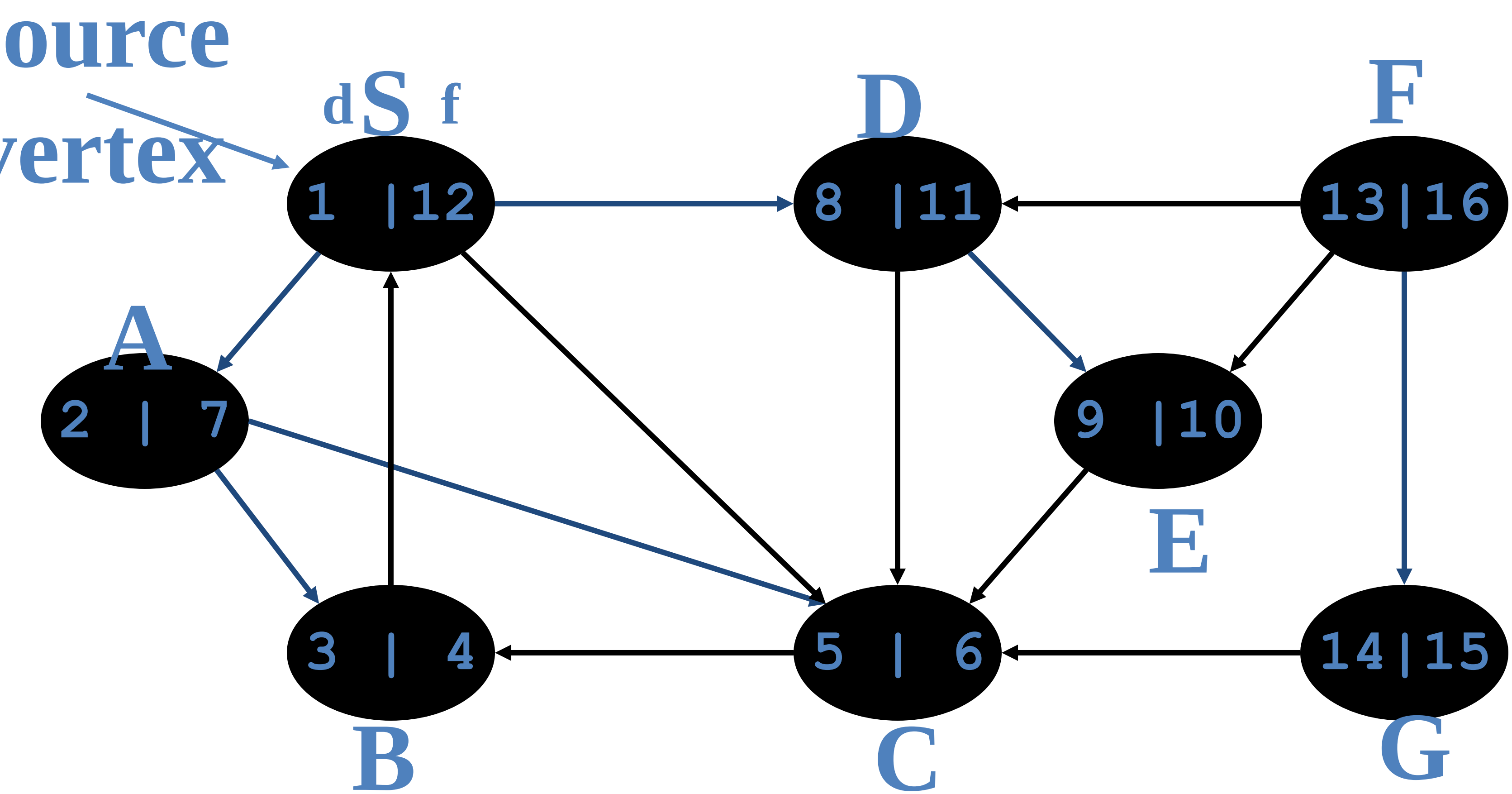
# DFS Example



# 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);
}
```

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

```
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;
}
```

**BUT, there is actually a tighter bound.**

**How many times will DFS_Visit() actually be called?**

# 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;
}
```

**So, running time of DFS = O(V+E)**

# DFS Complexity

- While iterating with this technique, we move over **each node and edge exactly once**, and once we are over a node that has already been visited then we backtrack, which means we are pruning possibilities that have already been marked. So hence the overall complexity is reduced from exponential to linear.

**Pseudocode for DFS:**

- DFS(Graph, vertex)

```
DFS(Graph, vertex)
    vertex.visited = true
    for each v1 ∈ Graph.Adj[vertex]
        if v1.visited ==  false
            DFS(Graph, v1)
```

- The time complexity of DFS is also O(V+E) due to:

1. It visits each vertex once, contributing V to the complexity.
2. It checks each edge once when moving from one vertex to its adjacent vertices, contributing E to the complexity.

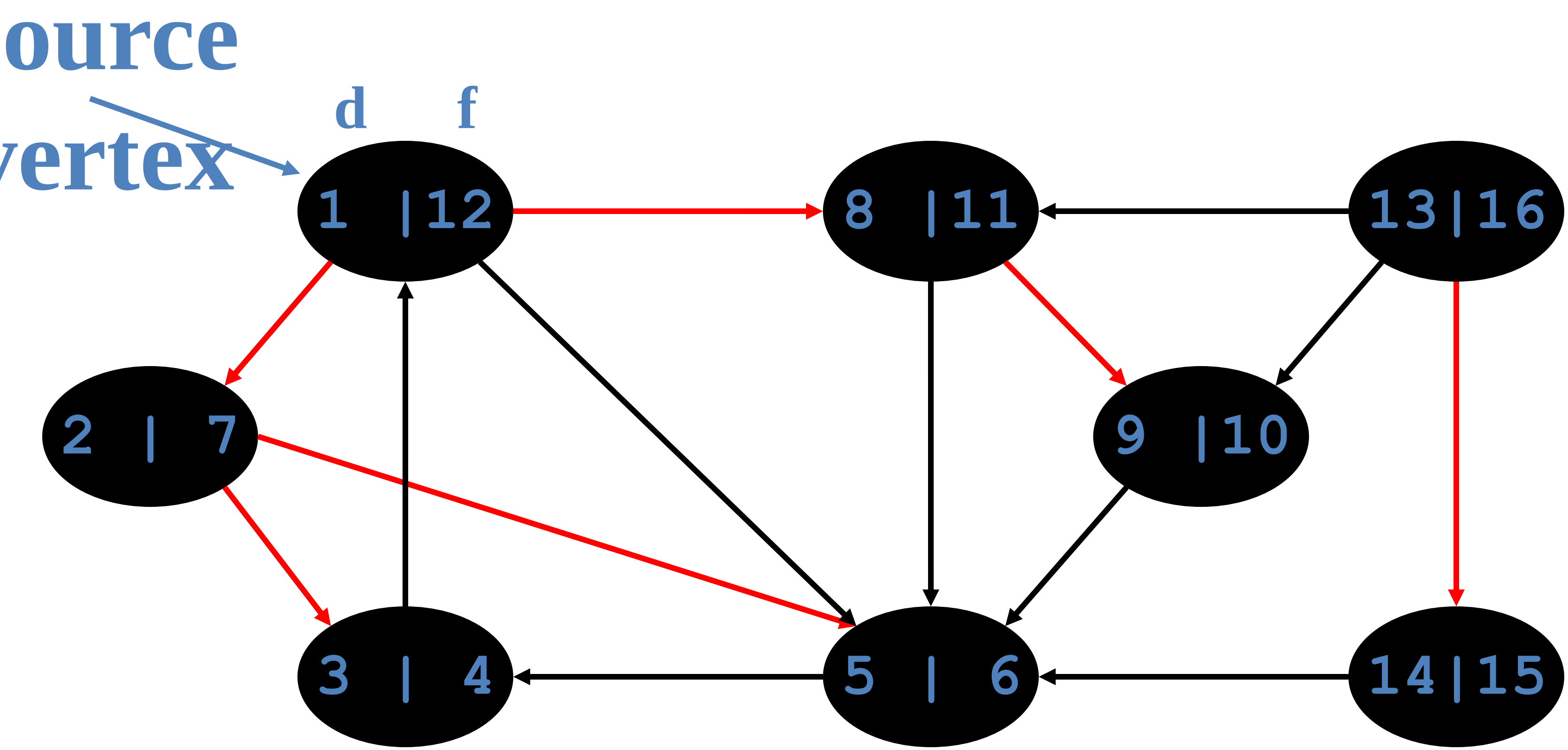
# Depth-First Sort Analysis

- 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

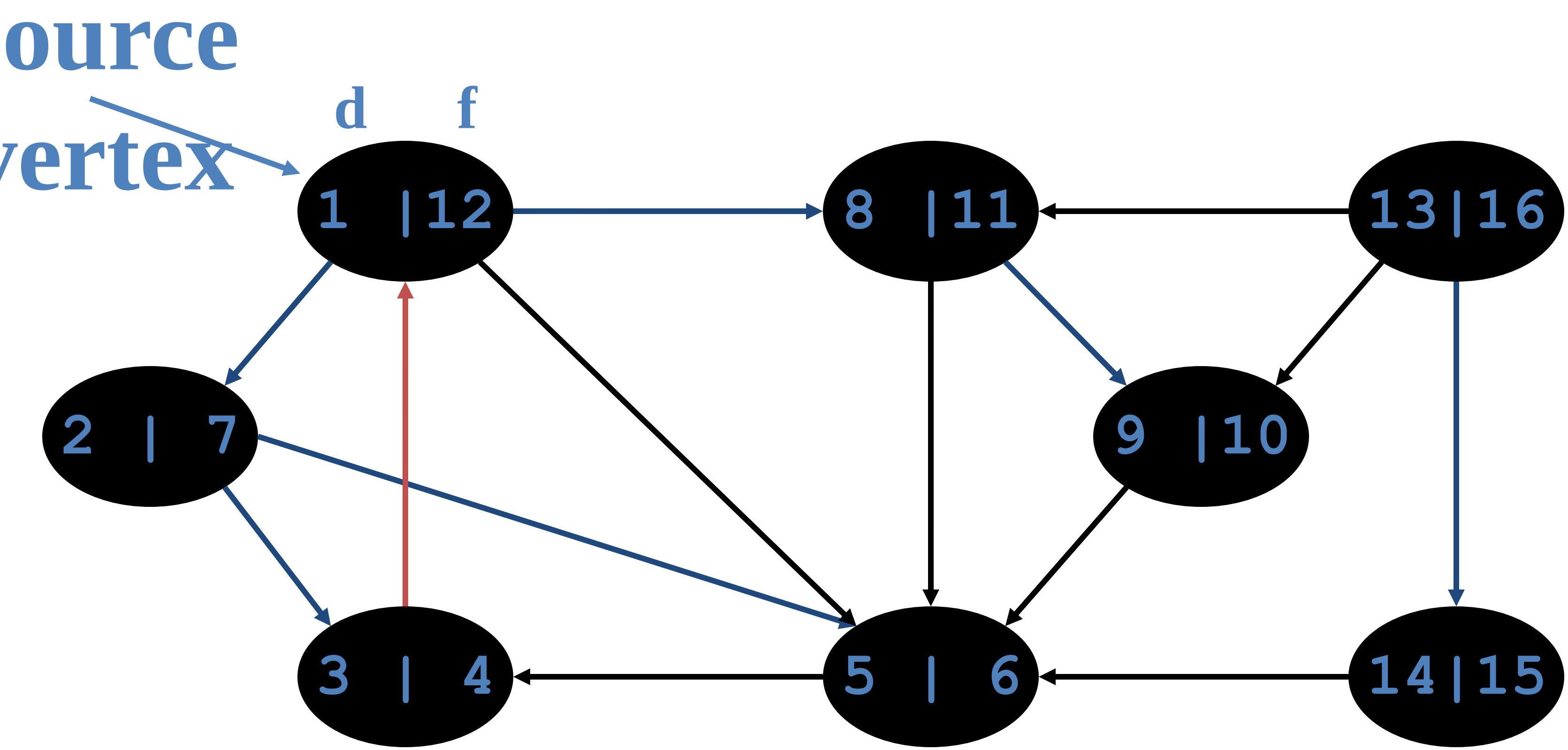



Tree edges

# 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

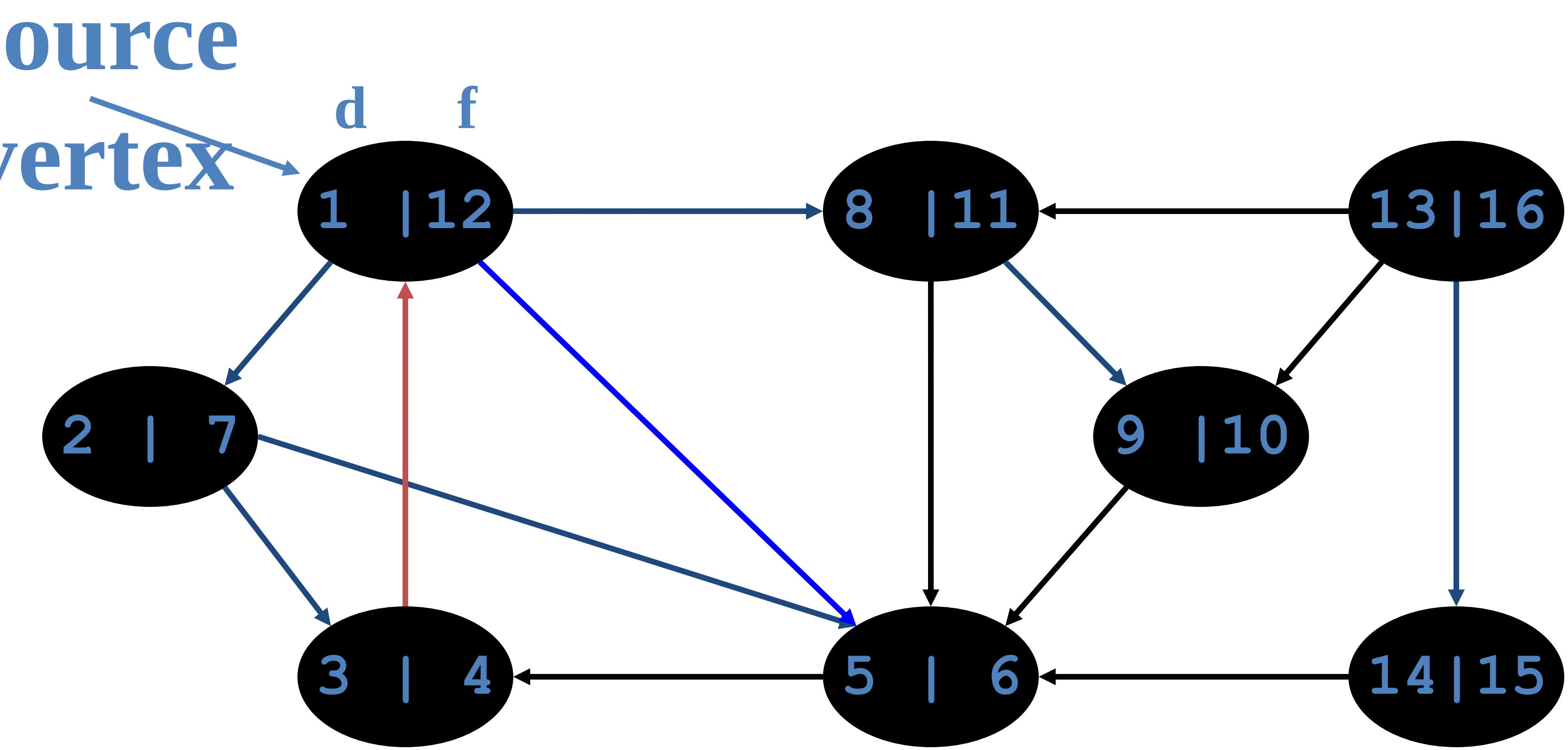



**Tree edges** **Back edges**

# 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

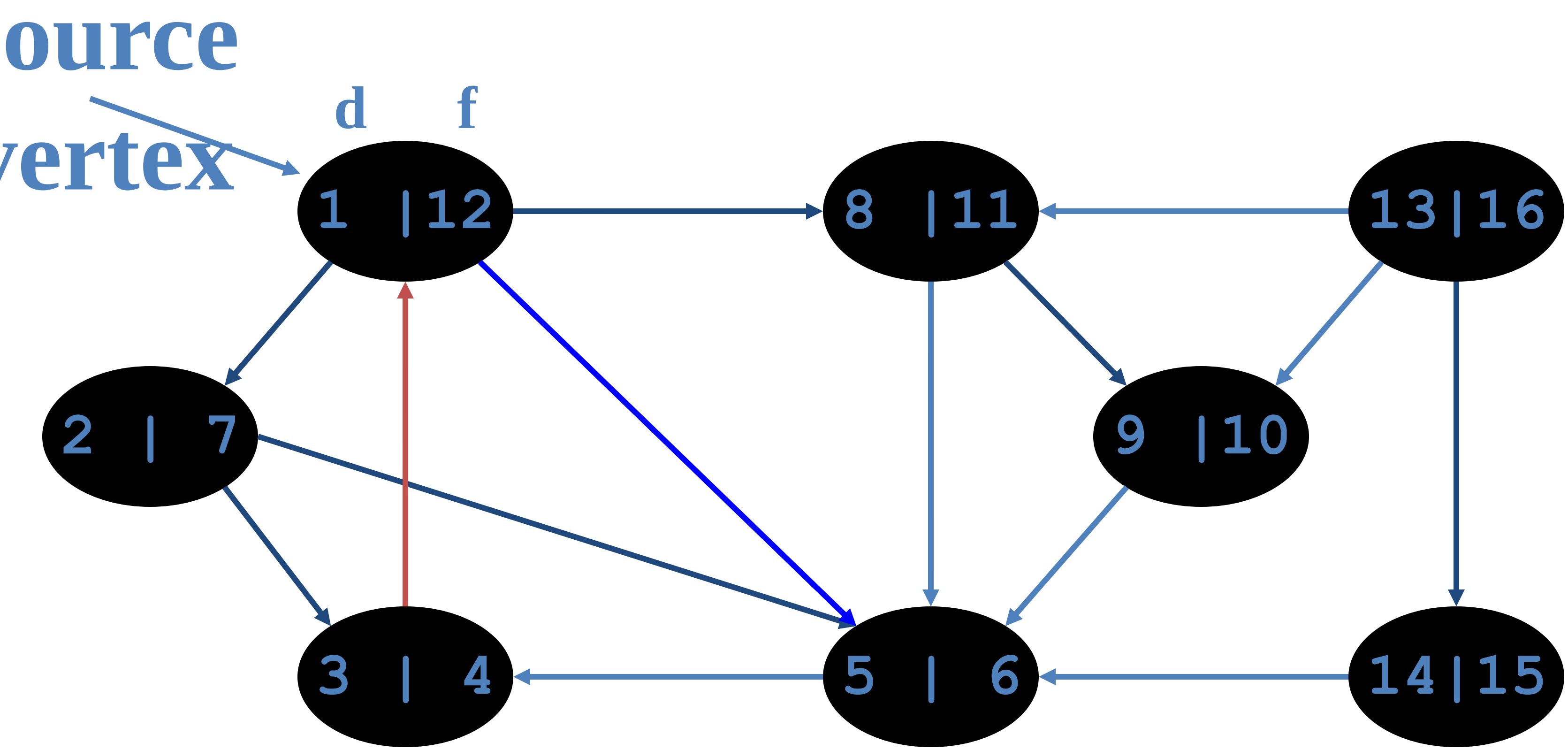



**Tree edges** **Back edges** **Forward edges**

# 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**

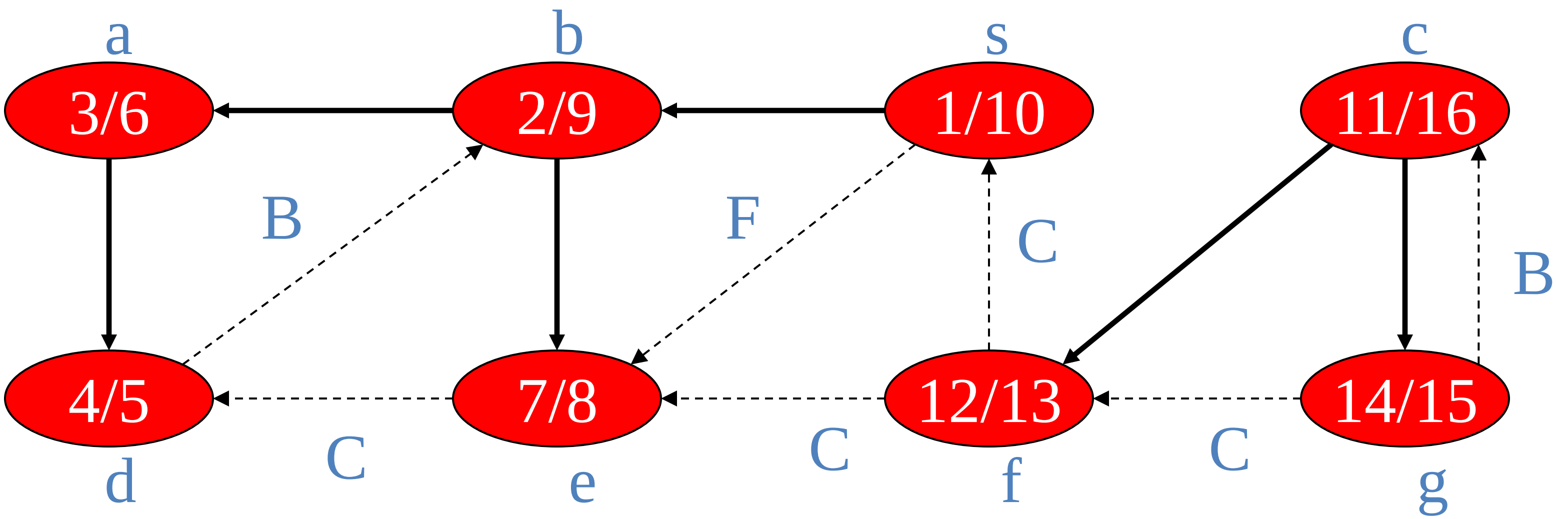
# DFS: Kinds of edges

- 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

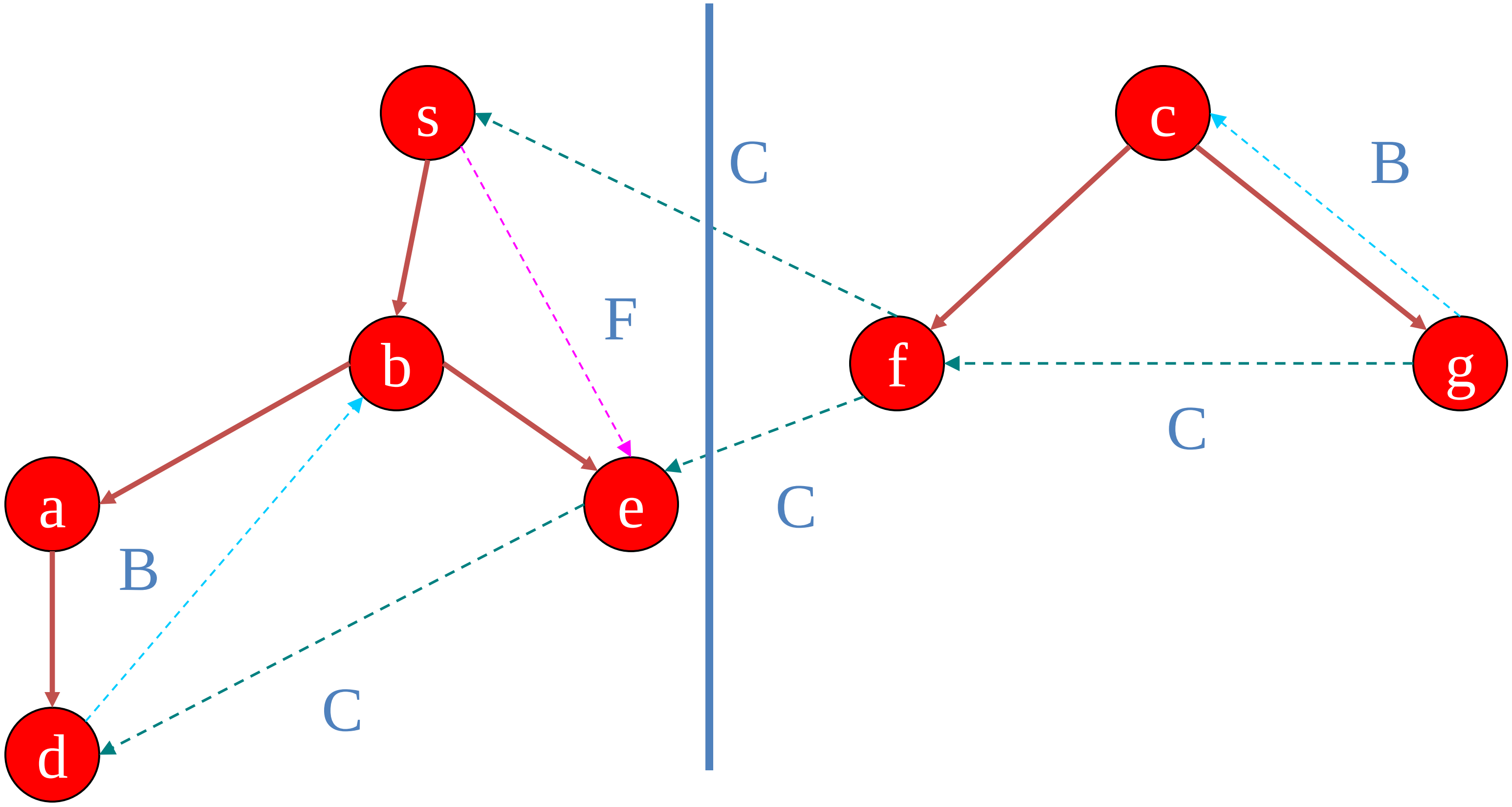
# More about the edges

- 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 - Timestamps



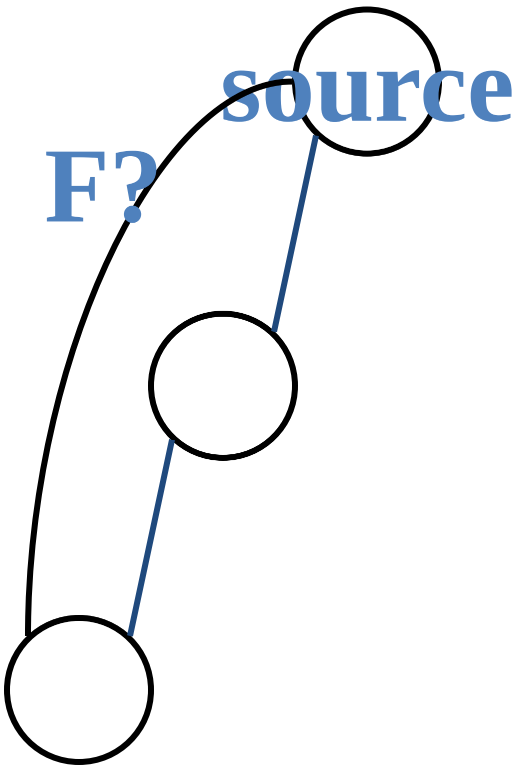
# Depth-First Search: Detect Edge

```
    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;
}
```

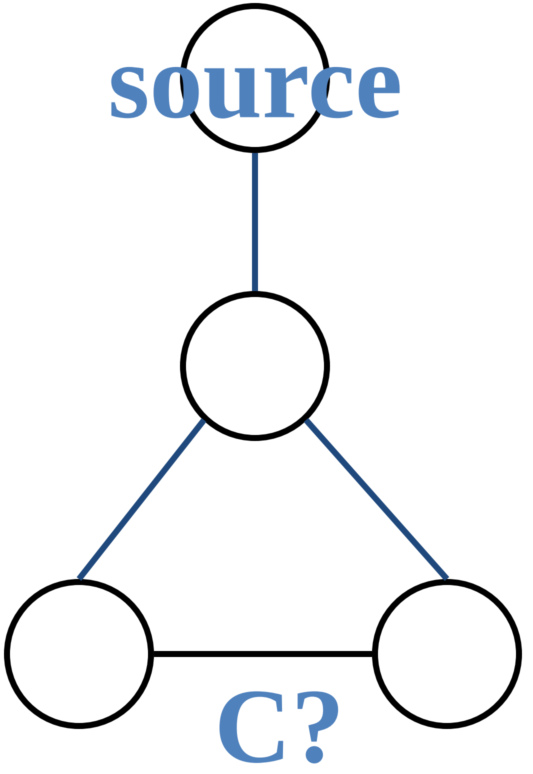
# DFS: Kinds Of Edges

- 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 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



# 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 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