Algorithms

DFS

Last Class's Topic

- Graph Representation
  - Adjacency Matrix
  - Adjacency List
- BFS – Breadth First Search

Breadth-First Search: The Code

```
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;
 }
}
```

3

Breadth-First Search: Print Path

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

```
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);
   }
}
```
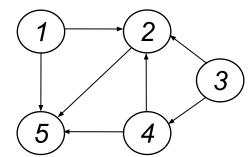
BFS – Questions

- 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
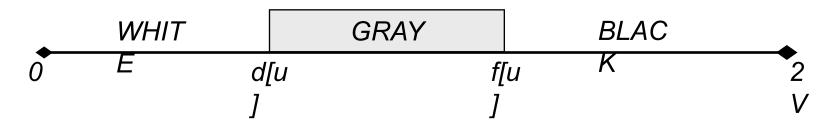   - f[v] = finishing time (done with examining v's adjacency list)
  - Depth-first forest

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] – 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

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

WHIT	GRAY	BLAC
E		K

0   d[u]   f[u]   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
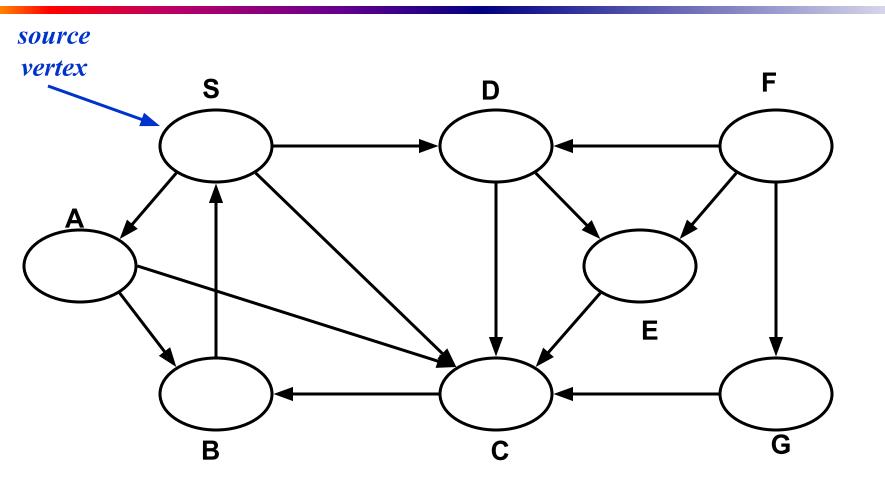
```
Data: color[V], time,
    prev[V],d[V], f[V]
DFS(G) // where prog starts
{
    for each vertex u ∈ V
    {
        color[u] = WHITE;
    prev[u]=NIL;
    f[u]=inf; d[u]=inf;
    }
    time = 0;
    for each vertex u ∈ V
      if (color[u] == WHITE)
          DFS_Visit(u);
}
```
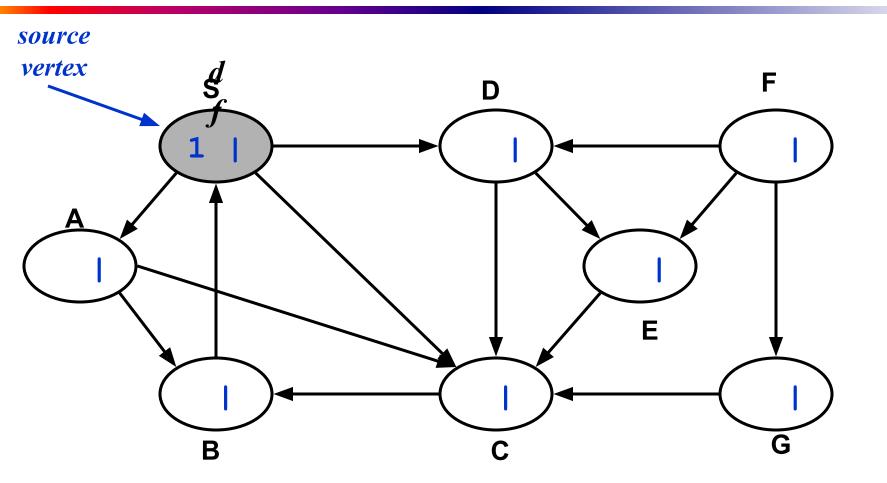
```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
        if(color[v] == WHITE){
      prev[v]=u;
          DFS_Visit(v);}
    }
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

What does `d[u]` *represent?*

Depth-First Search: The Code
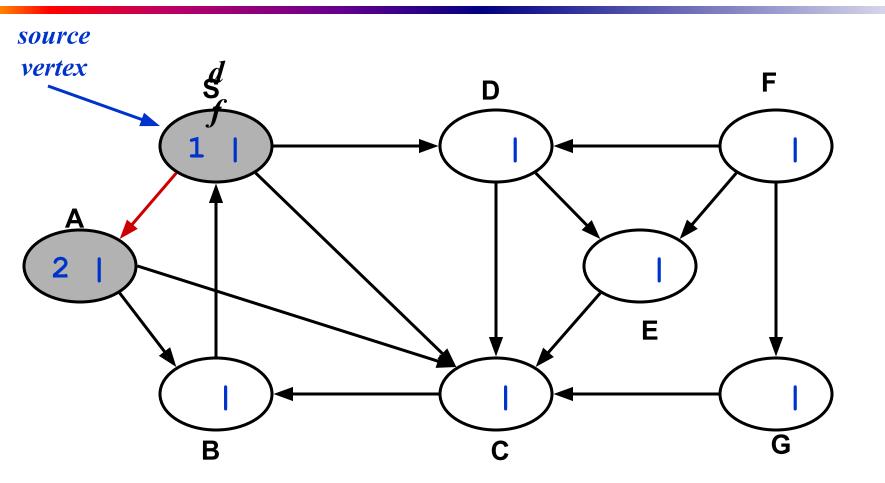
```
Data: color[V], time,
    prev[V],d[V], f[V]
DFS(G) // where prog starts
{
    for each vertex u ∈ V
    {
        color[u] = WHITE;
    prev[u]=NIL;
    f[u]=inf; d[u]=inf;
    }
    time = 0;
    for each vertex u ∈ V
      if (color[u] == WHITE)
          DFS_Visit(u);
}
```
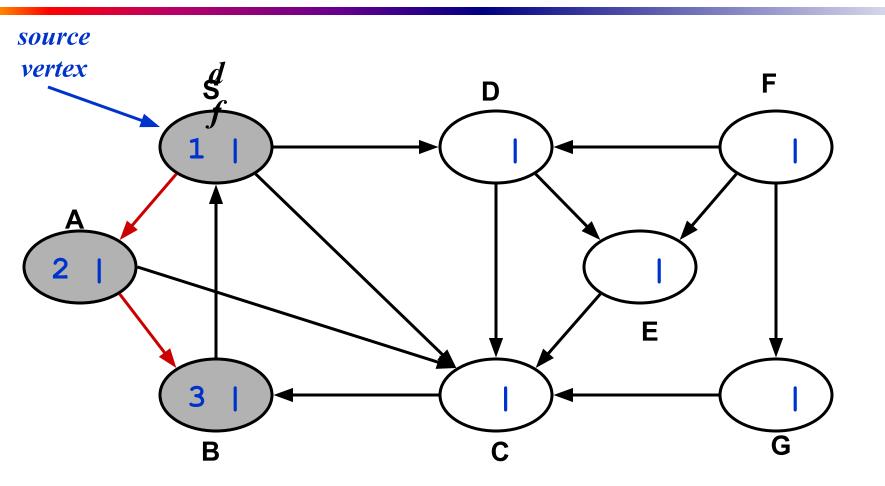
```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
        if(color[v] == WHITE){
      prev[v]=u;
          DFS_Visit(v);}
    }
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

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;
}
```
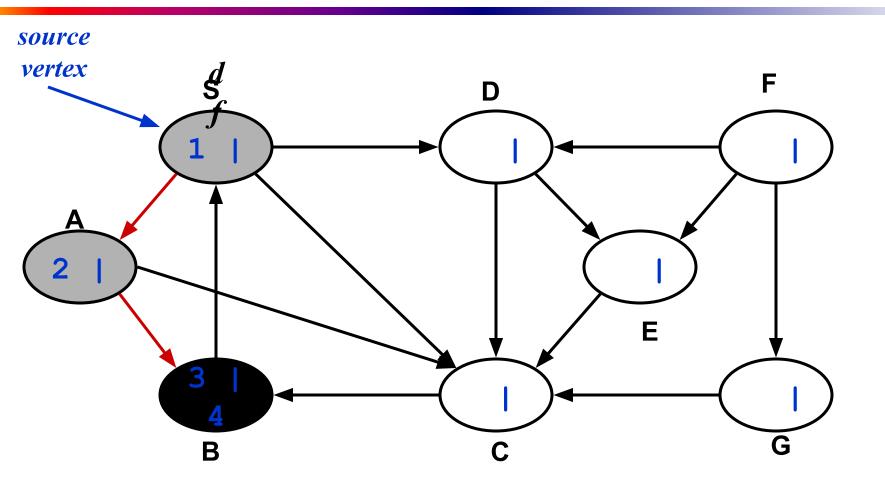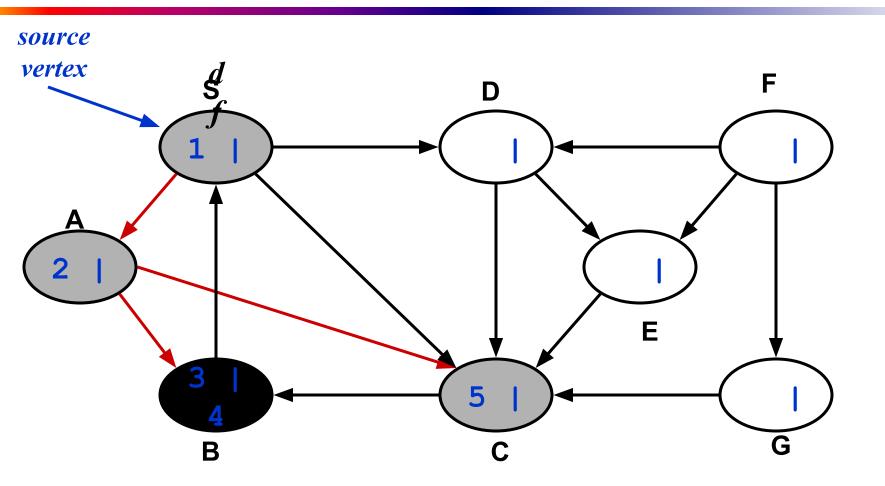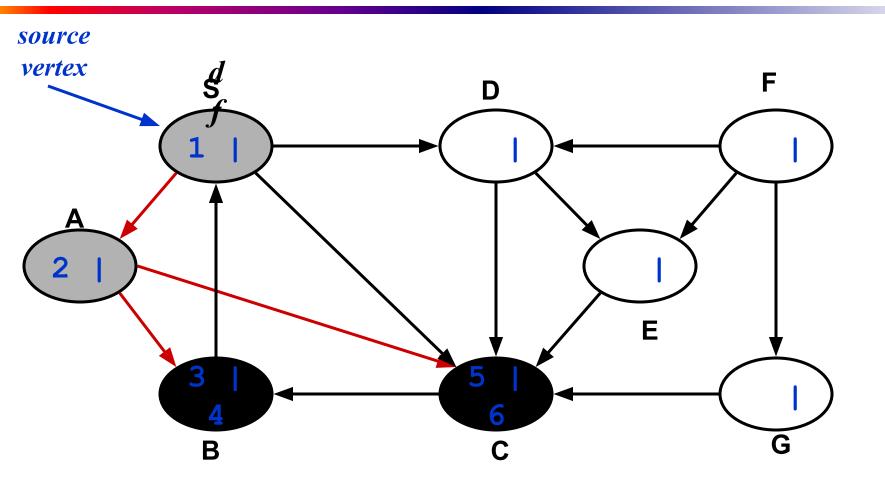
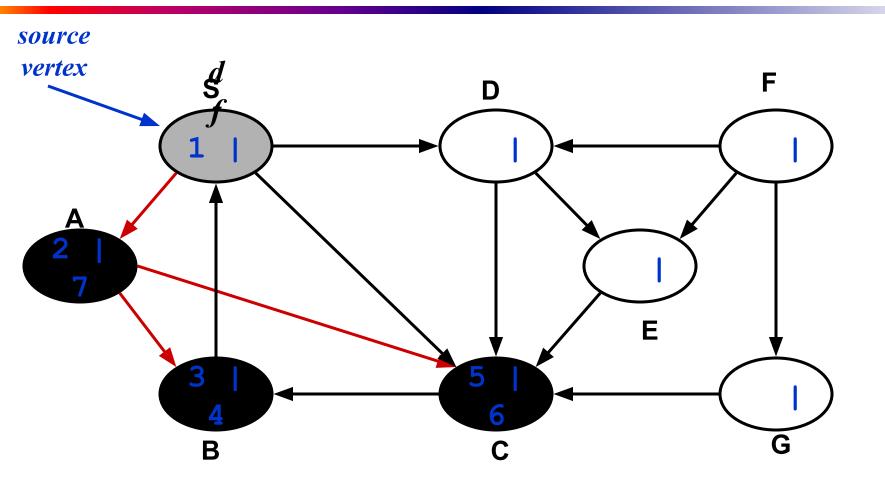*Will all vertices eventually be colored black?*

DFS Example

DFS Example

DFS Example

DFS Example

DFS Example

source vertex

DFS Example

DFS Example

source vertex

DFS Example

DFS Example

source vertex

d
s f

D   F

1 | 8 | |

A

2 |
7

|

E

3 | 5 | |
4   6

B   C   G

21

DFS Example

source vertex

d
s
f

A

2 | 7

D

8 |

F

|

9 |

E

3 | 4

B

5 | 6

C

1 |

G

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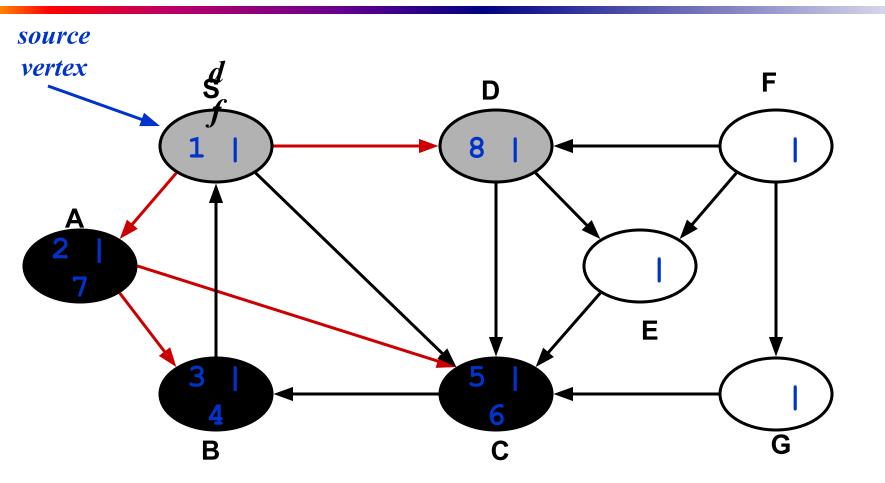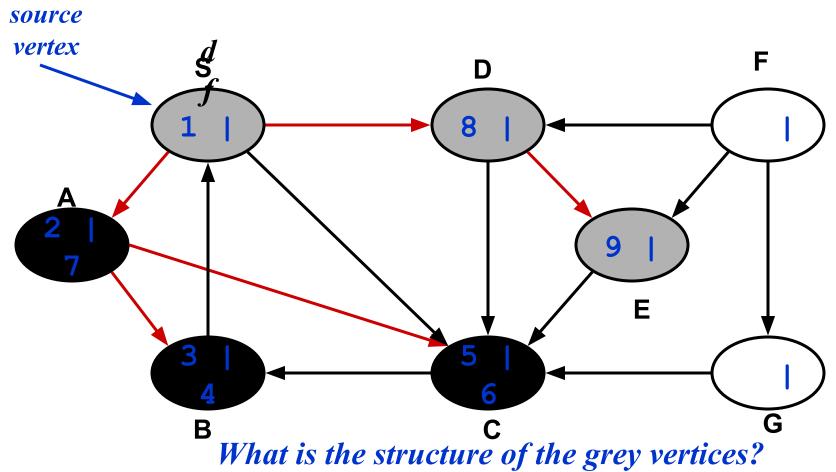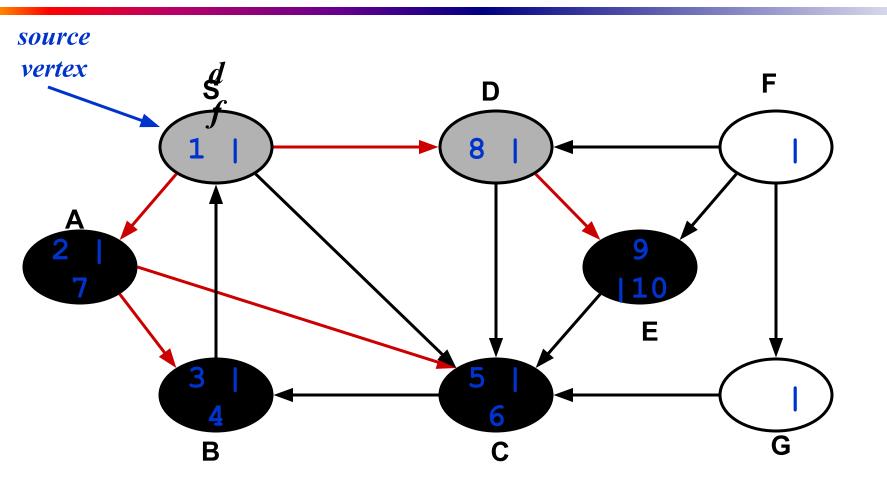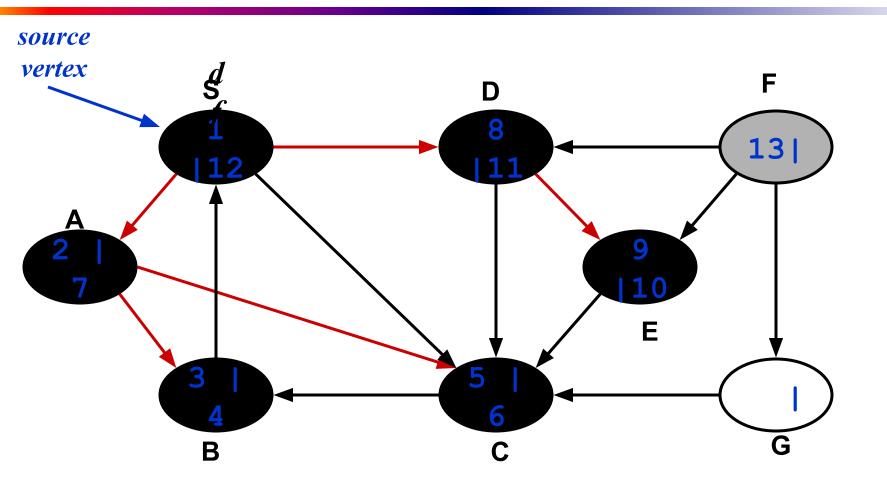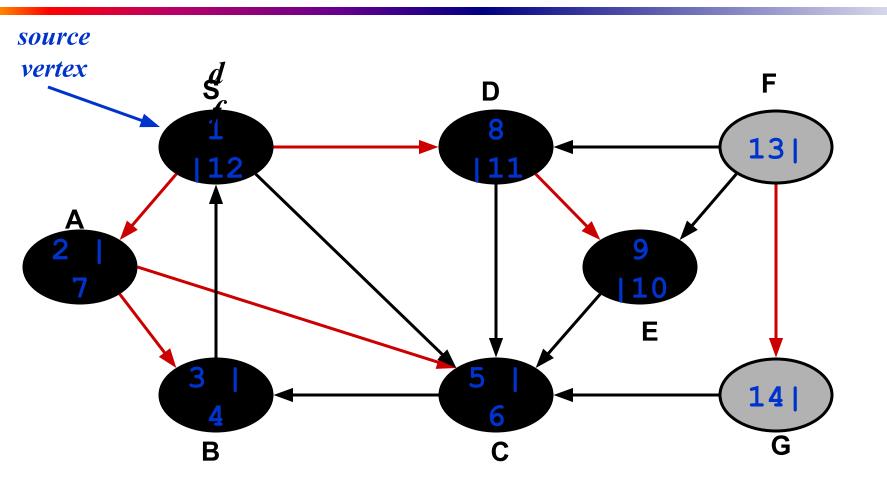
What is the structure of the grey vertices?
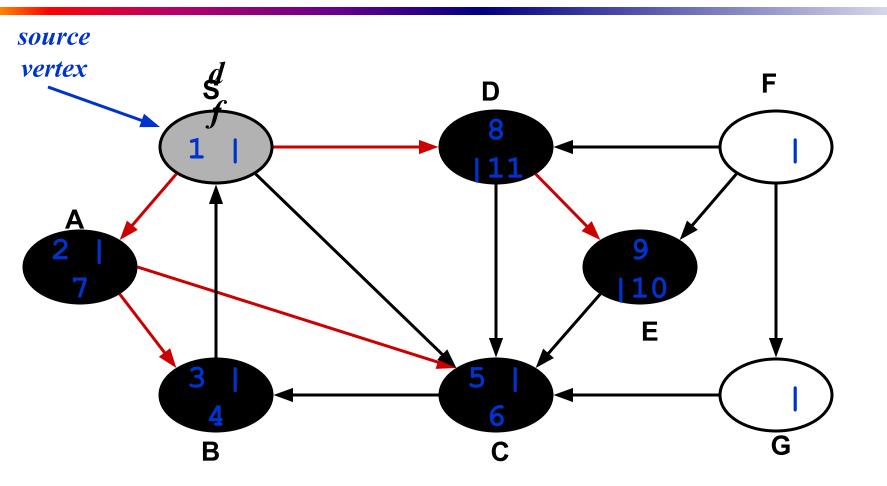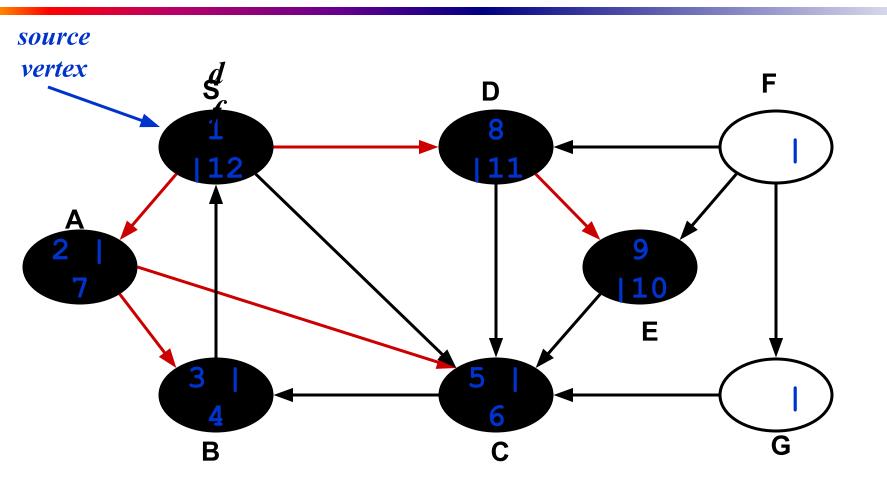*What do they represent?*

DFS Example

source vertex

d
S
f

| 1 | |

| 8 | |

F | |

A
| 2 | |
7

| 9 |
|10

E

D

| 3 | |
4

B

| 5 | |
6

C

| |

G

23

DFS Example

DFS Example

DFS Example

DFS Example

source vertex

27

DFS Example

source vertex

S *d* *f*
1 |12

D 8 |11

F 13|

A 2 | 7

B 3 | 4

C 5 | 6

E 9 |10

G 14| 15

28

DFS Example

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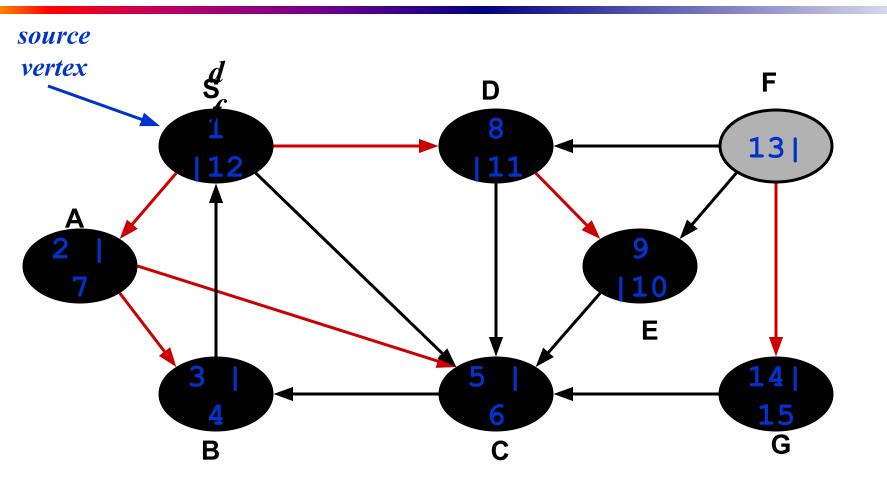
```
Data: color[V], time,
    prev[V],d[V], f[V]
DFS(G) // where prog starts
{
   for each vertex u ∈ V
   {
      color[u] = WHITE;
   prev[u]=NIL;
   f[u]=inf; d[u]=inf;
   }
   time = 0;
   for each vertex u ∈ V
     if (color[u] == WHITE)
         DFS_Visit(u);
}
```

```
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?

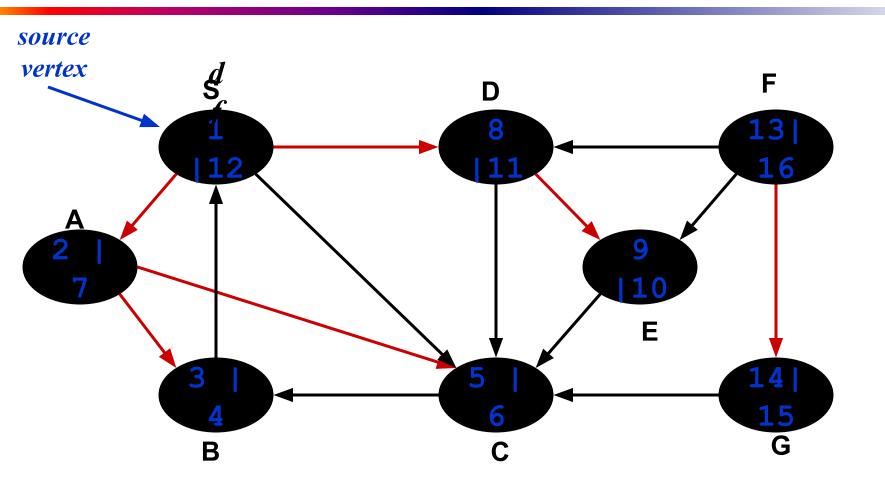# Depth-First Search: The Code

```
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)

worst case complexity

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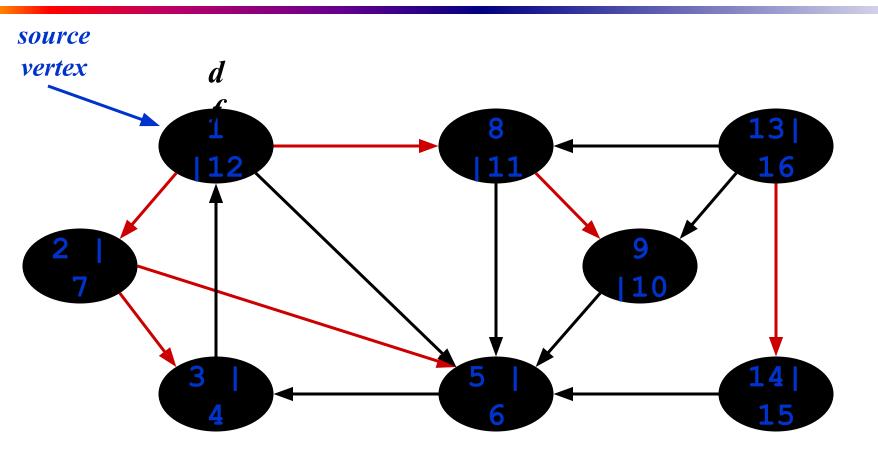
```
DFS_Visit(u)
{
    color[u] = GREY;
    time = time+1;
    d[u] = time;
    for each v ∈ Adj[u]
    {
        if (color[v] == WHITE)
      prev[v]=u;
          DFS_Visit(v);
    }
    color[u] = BLACK;
    time = time+1;
    f[u] = time;
}
```

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

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
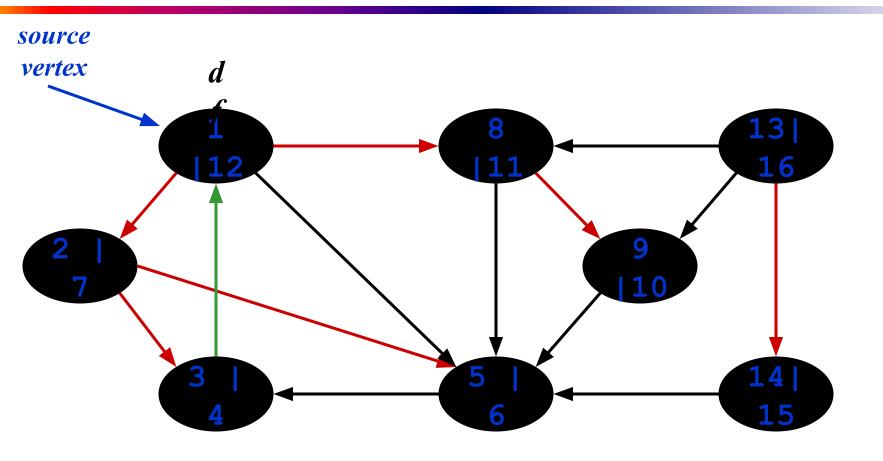
DFS: Kinds of edges

- 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

36

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)
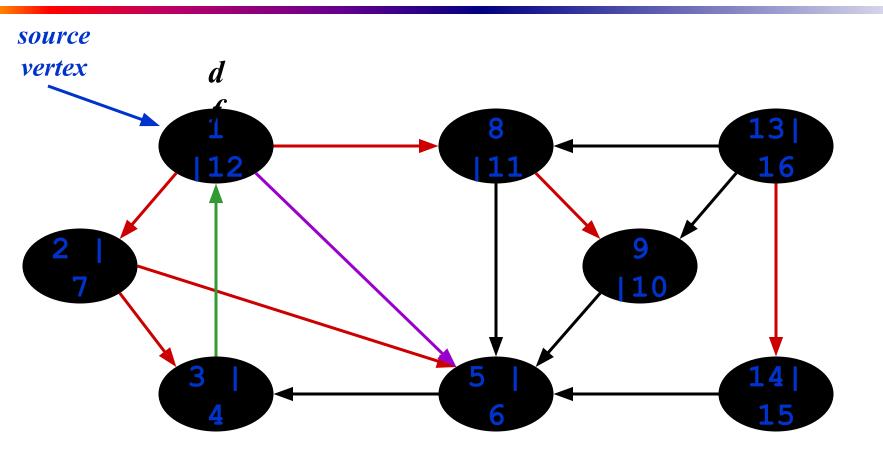   - Self loops are considered as to be back edge.

DFS Example

source vertex

d
f

1 |12

8 |11

13| 16

2 | 7

9 |10

3 | 4

5 | 6

14| 15

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
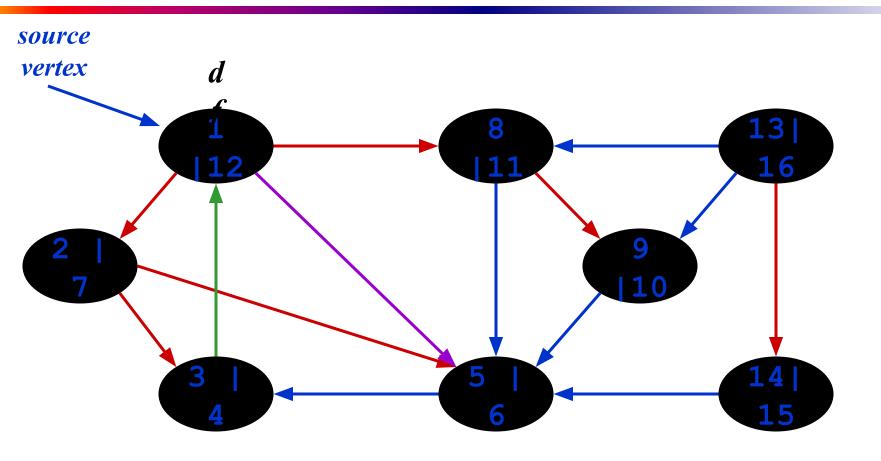   - From grey node to black node

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
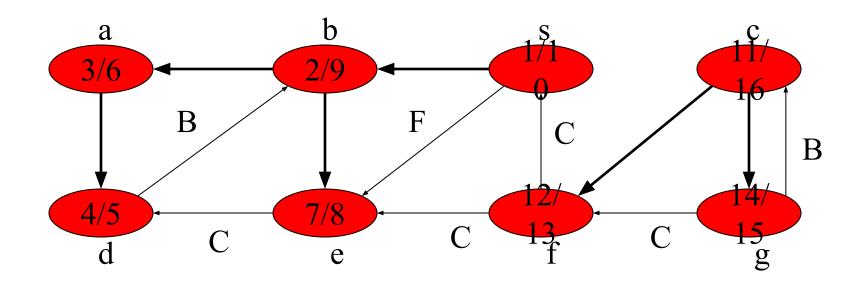   - From a grey node to a black node

DFS Example

source vertex

d

f

1|12   8|11   13|16

2|7

9|10

3|4   5|6   14|15

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
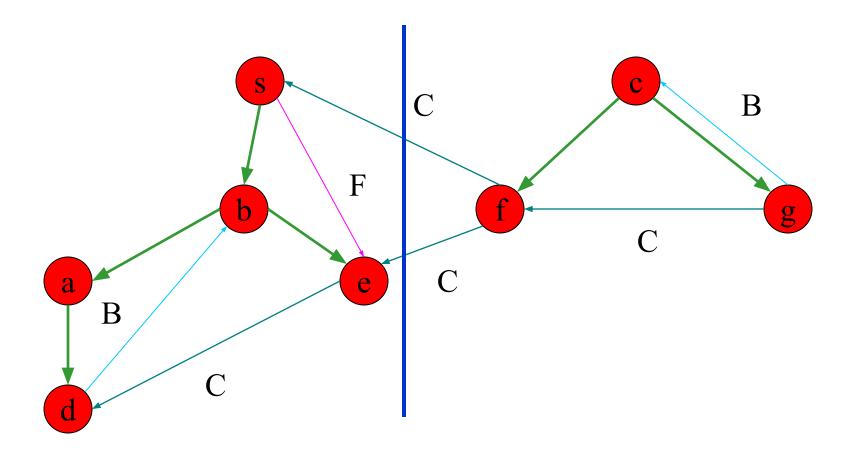- Note: tree & back edges are important; most algorithms don't distinguish forward & cross

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: $d[u]<d[v]$: In same forest
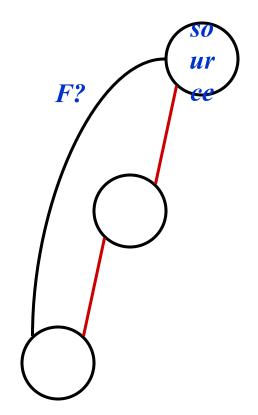   - Cross Edge: $d[u]>d[v]$: In another forest

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;
}
```
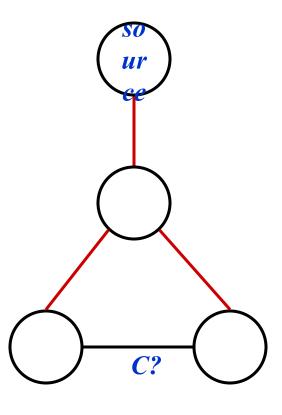
DFS: Kinds Of Edges

- 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?*)

so ur ce

F?

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

so ur ce

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?

```
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)

- 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)

Reference

- Cormen –
  - Chapter 22 (Elementary Graph Algorithms)
- Exercise –
  - 22.3-4 –Detect edge using d[u], d[v], f[u], f[v]
  - 22.3-11 – Connected Component
  - 22.3-12 – Singly connected