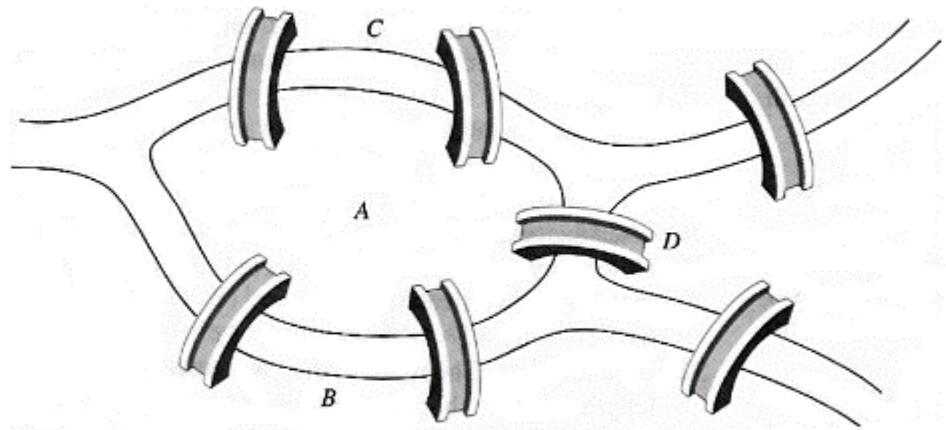
Brute Force and Exhaustive Search II

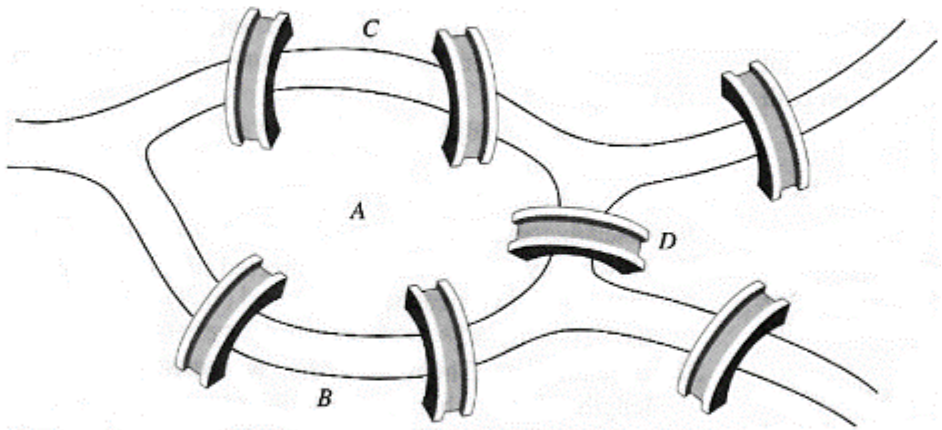
Murat Osmanoglu

Graph Theory



- Königsberg was a city in Germany in 18th century. There was a river named Pregel that divided the city into four distinct regions.

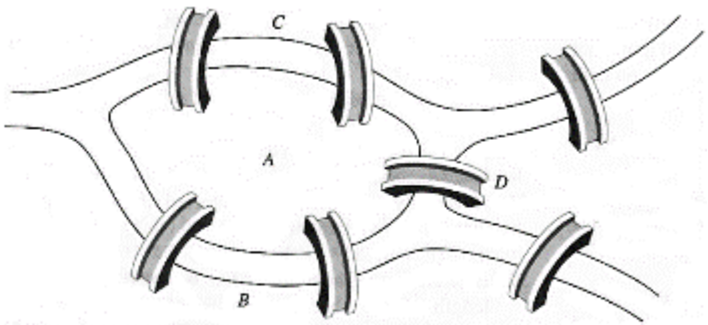
Graph Theory



- Königsberg was a city in Germany in 18th century. There was a river named Pregel that divided the city into four distinct regions.
- There was a natural question for the people of Königsberg :

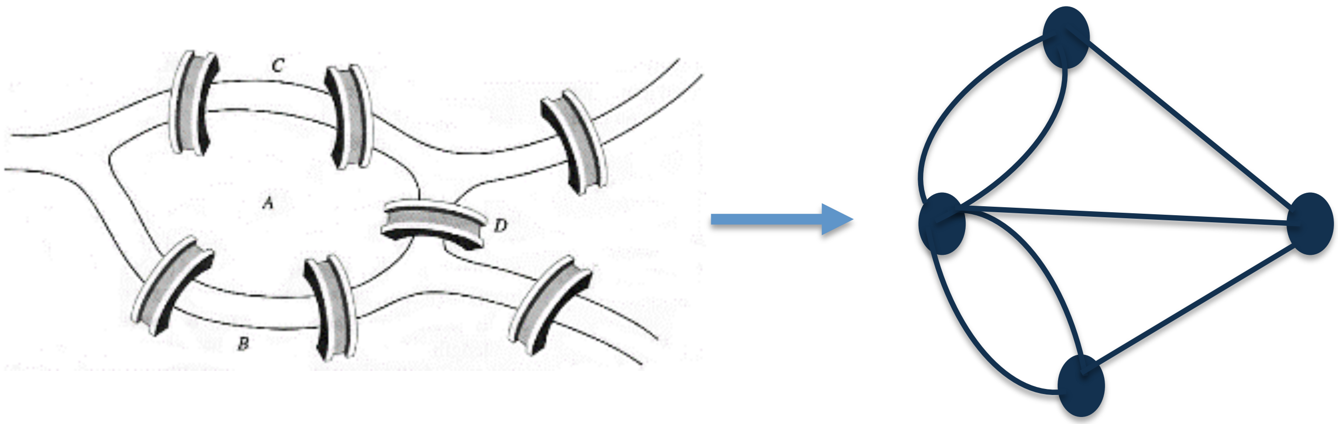
'Is it possible to take a walk around the city that crosses each bridge exaactly once?'

Graph Theory



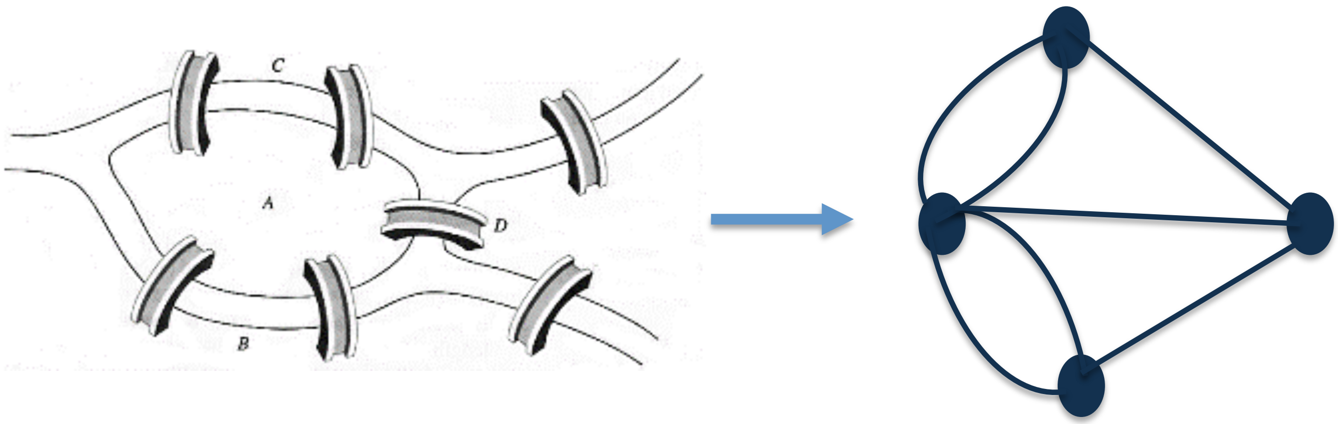
- The problem was solved by Swiss mathematician Leonard Euler. His works are considered as the beginning of Graph Theory.

Graph Theory



- The problem was solved by Swiss mathematician Leonard Euler. His works are considered as the beginning of Graph Theory.
- Euler represented four distinct lands with four points (or nodes), and seven bridges with seven lines connecting those points.

Graph Theory



- The problem was solved by Swiss mathematician Leonard Euler. His works are considered as the beginning of Graph Theory.

- Euler represented four distinct lands with four points (or nodes), and seven bridges with seven lines connecting those points.

'Can you find a path that includes every edge exactly once?'
'Is the given graph traversable?'

Graph Theory

$$G = (V, E)$$

set of nodes (or vertices)

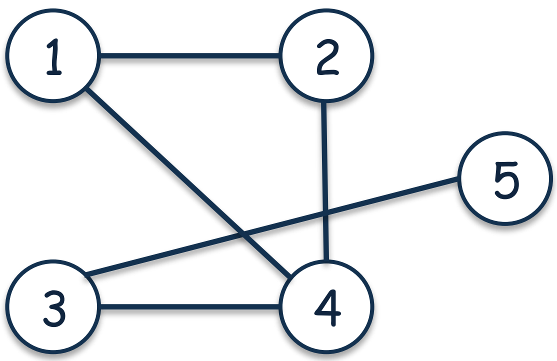
set of edges (or arc)

Graph Theory

$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



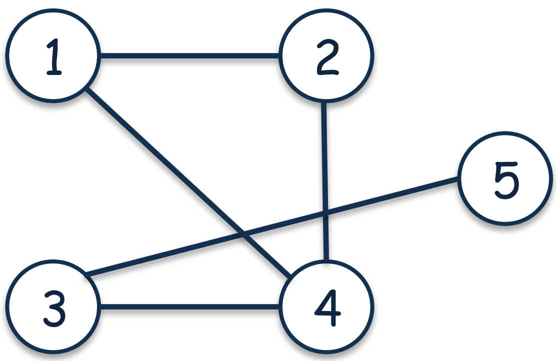
undirected graph

Graph Theory

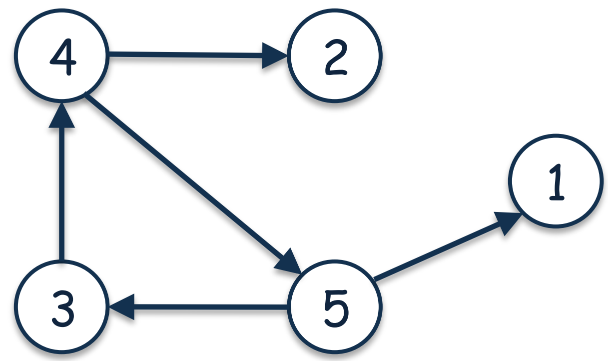
$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



undirected graph



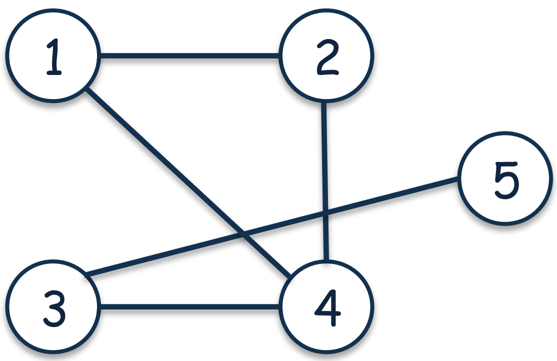
directed graph

Graph Theory

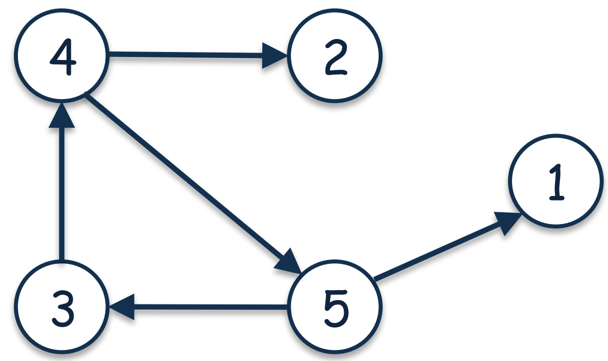
$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



undirected graph



directed graph

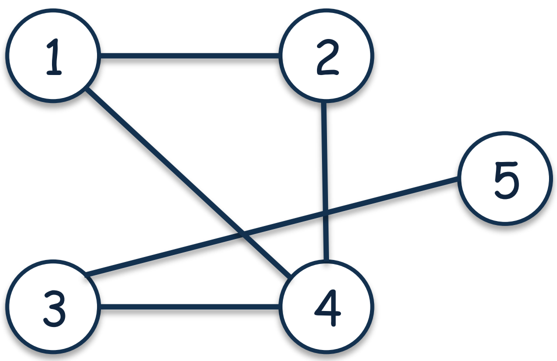
deg(v)= # of edges at that vertex

Graph Theory

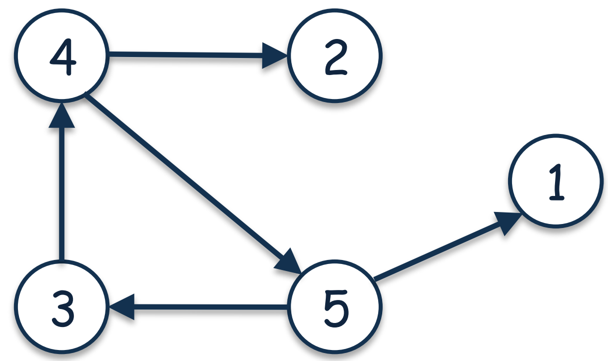
$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



undirected graph



directed graph

deg(v)= # of edges at that vertex

$deg^{in}(v)$ = # of incoming edges

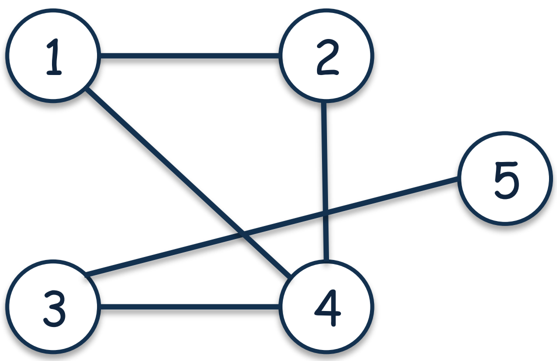
$deg^{out}(v)$ = # of outgoing edges

Graph Theory

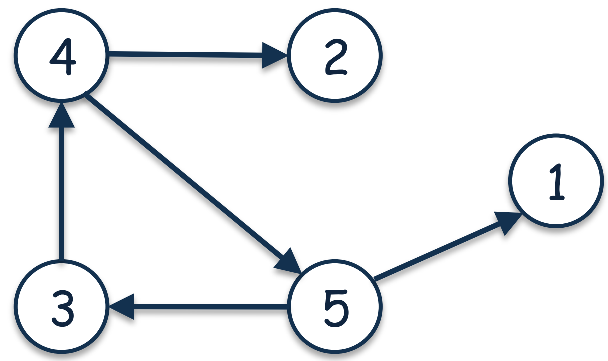
$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)



undirected graph



directed graph

deg(v)= # of edges at that vertex

$$\Sigma \deg(v) = 2|E|$$

$\deg^{in}(v)$ = # of incoming edges

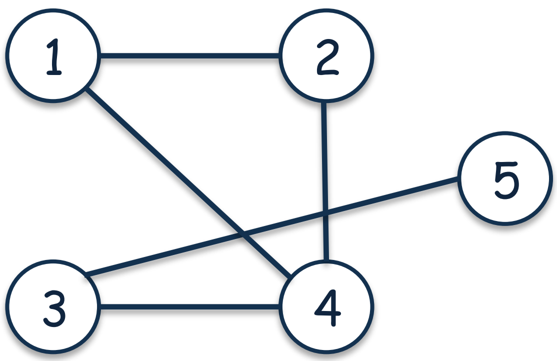
$\deg^{out}(v)$ = # of outgoing edges

Graph Theory

$$G = (V, E)$$

set of nodes (or vertices)

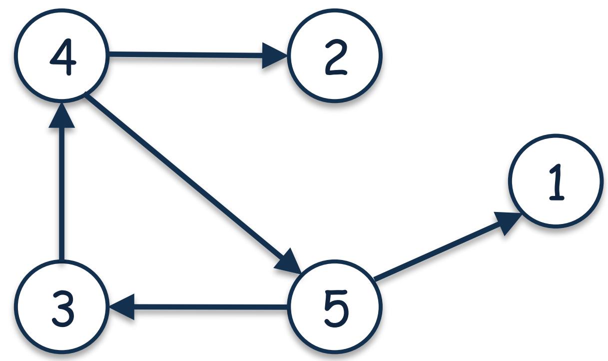
set of edges (or arc)



undirected graph

deg(v)= # of edges at that vertex

$$\Sigma \ \mathrm{deg}(v) = 2\,|E|$$



directed graph

deg^{in} (v) = # of incoming edges

deg^{out} (v) = # of outgoing edges

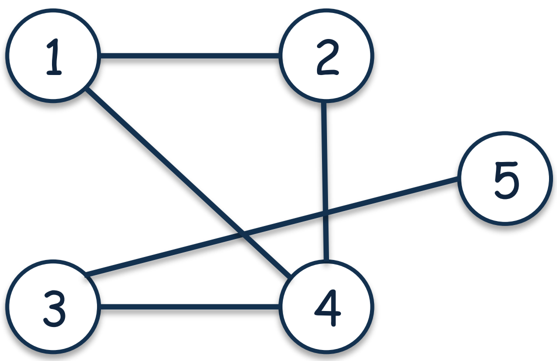
$$\Sigma \ \mathrm{deg}^{in}(v) = \Sigma \ \mathrm{deg}^{in}(v) = |E|$$

Graph Theory

$$G = (V, E)$$

set of nodes (or vertices)

set of edges (or arc)

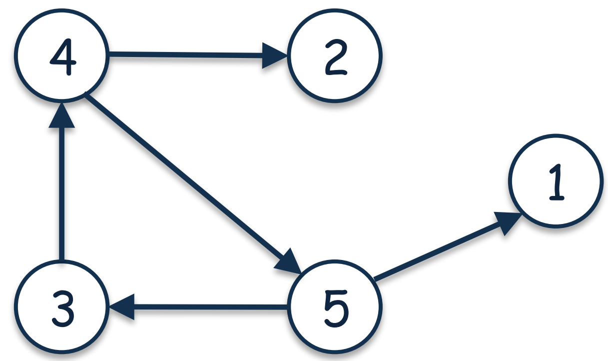


undirected graph

deg(v)= # of edges at that vertex

$$\Sigma \ deg(v) = 2\ |E|$$

- a vertex v is called odd vertex if deg(v) is odd
- a vertex v is called even vertex if deg(v) is even



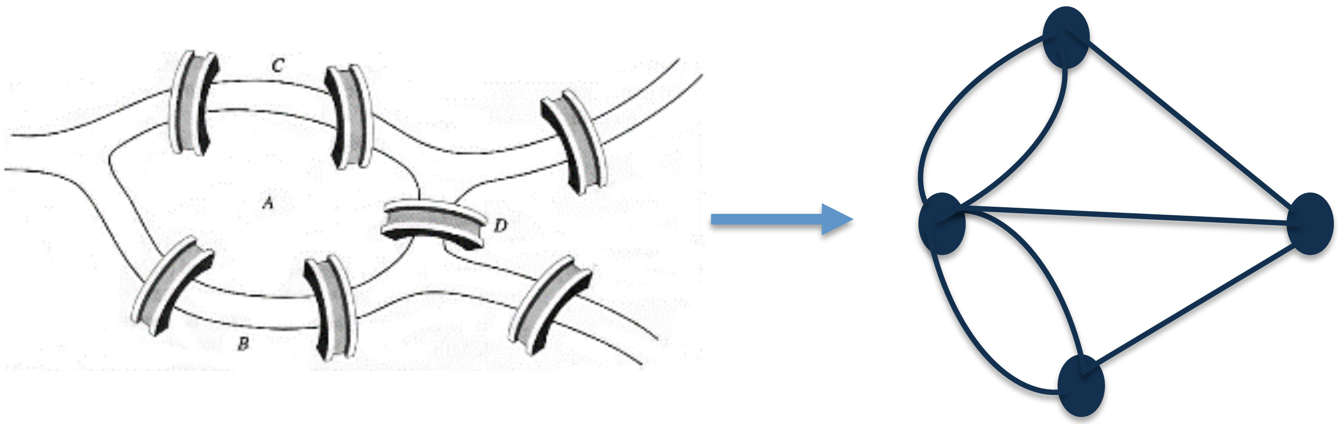
directed graph

deg^{in} (v) = # of incoming edges

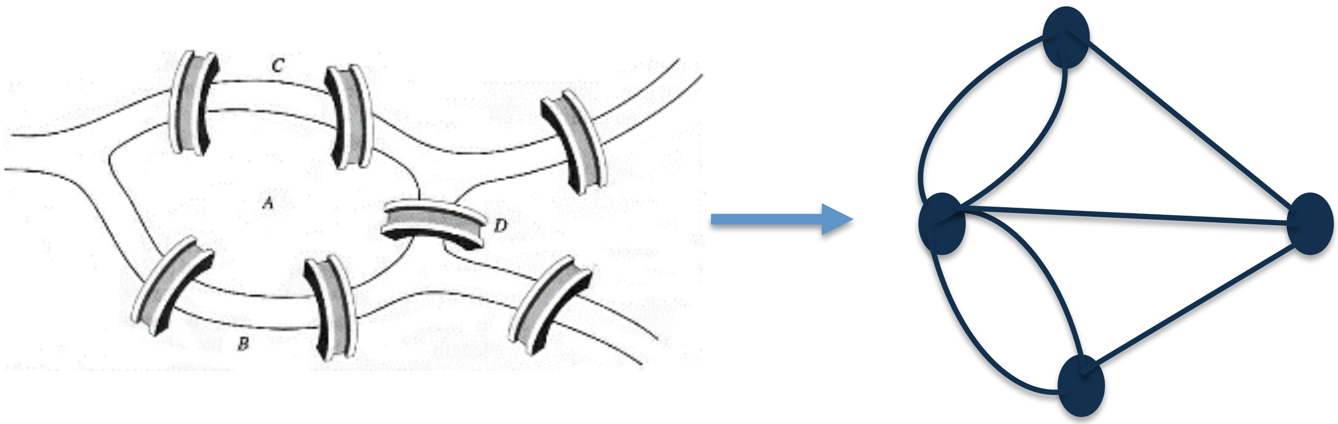
deg^{out} (v) = # of outgoing edges

$$\Sigma \ deg^{in}(v) = \Sigma \ deg^{in}(v) = |E|$$

Graph Theory

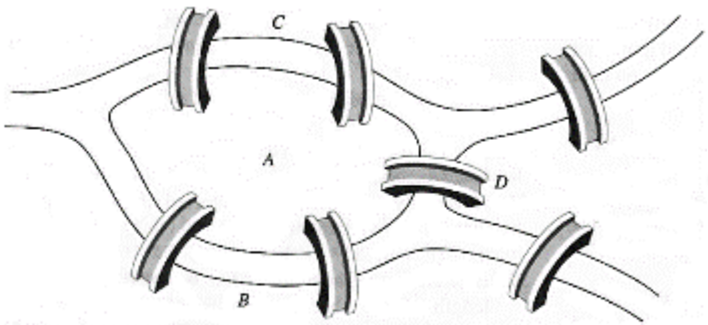


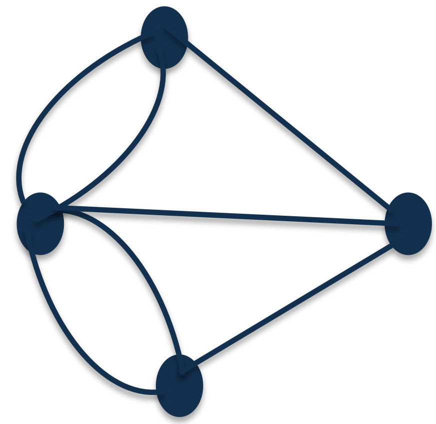
Graph Theory



- Euler showed that a graph can be traversable if it has no odd vertex or exactly two odd vertices.

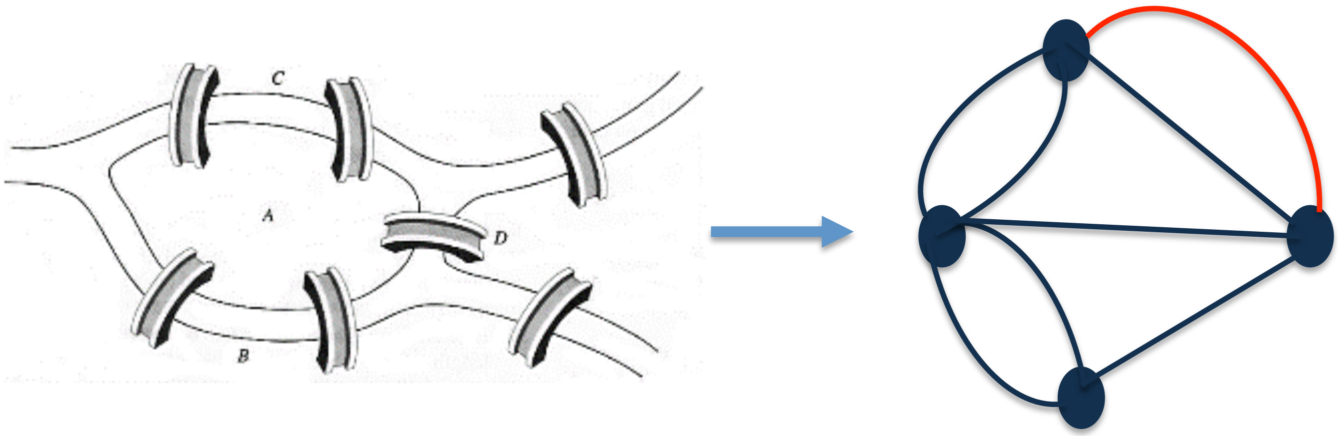
Graph Theory





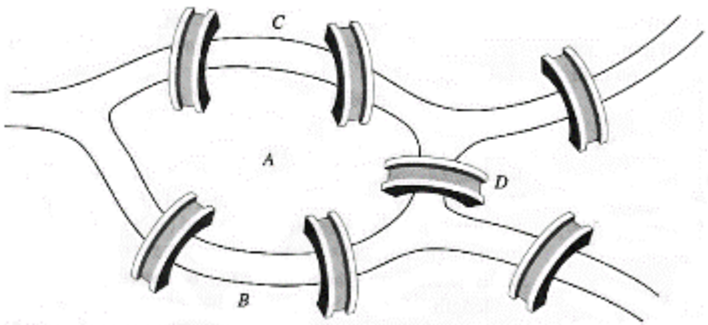
- Euler showed that a graph can be traversable if it has no odd vertex or exactly two odd vertices.
- Königsberg graph is not traversable since it has four odd vertices.

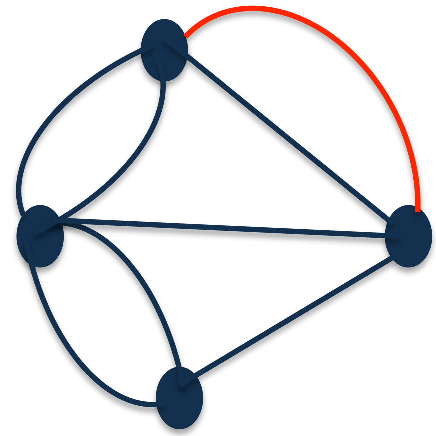
Graph Theory



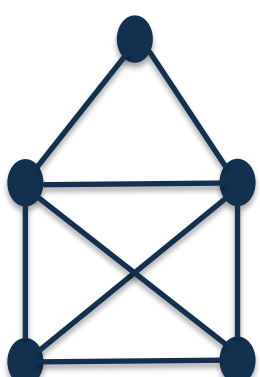
- Euler showed that a graph can be traversable if it has no odd vertex or exactly two odd vertices.
- Königsberg graph is not traversable since it has four odd vertices.

Graph Theory



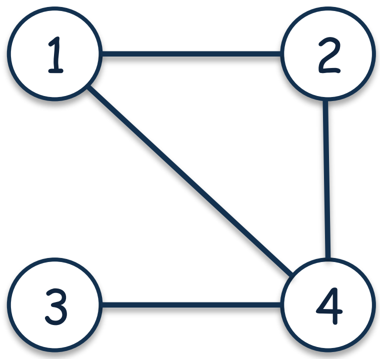


- Euler showed that a graph can be traversable if it has no odd vertex or exactly two odd vertices.
- Königsberg graph is not traversable since it has four odd vertices.

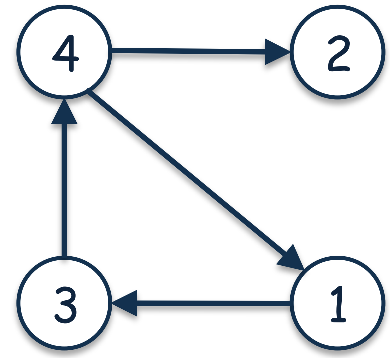


Can you draw an envelope without lifting your pen from the paper?

Graph Theory

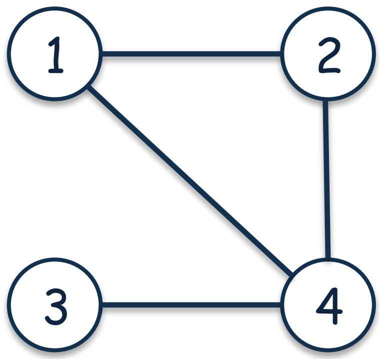


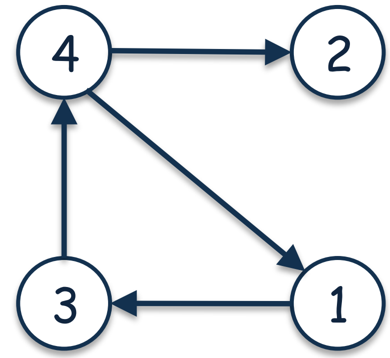
Adjacency List



Adjacency List

Graph Theory





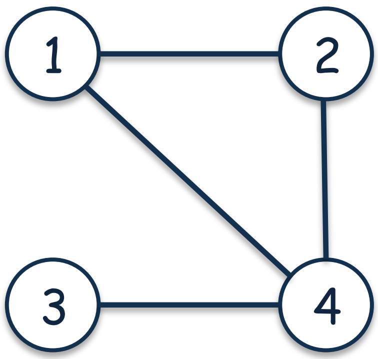
Adjacency List

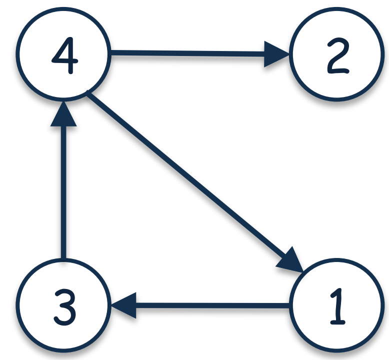
1 - 2,4
2 - 1,4
3 - 4
4 - 1,2,3

Adjacency List

1 - 3
2 -
3 - 4
4 - 1,2

Graph Theory





Adjacency List

1 - 2,4
2 - 1,4
3 - 4
4 - 1,2,3

Adjacency Matrix

	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

Adjacency List

1 - 3
2 -
3 - 4
4 - 1,2

Adjacency Matrix

	1	2	3	4
1	0	0	1	0
2	0	0	0	0
3	0	0	0	1
4	1	1	0	0

Graph Theory

Adjacency List | **Adjacency Matrix**

Graph Theory

	Adjacency List	Adjacency Matrix
• retrieving all neighbors of a given node u	O(deg(u))	O(|V|)

Graph Theory

	Adjacency List	Adjacency Matrix
• retrieving all neighbors of a given node u	O(deg(u))	O(|V|)
• given nodes u and v, checking if u and v are adjacent	O(deg(u))	O(1)

Graph Theory

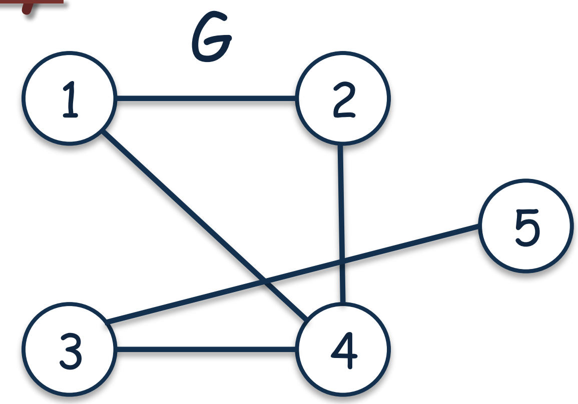
	Adjacency List	Adjacency Matrix
• retrieving all neighbors of a given node u	O(deg(u))	O(|V|)
• given nodes u and v, checking if u and v are adjacent	O(deg(u))	O(1)
• space	O(|E|+|V|)	$O(\|V\|^2)$

Graph Theory

	Adjacency List	Adjacency Matrix
• retrieving all neighbors of a given node u	$O(deg(u))$	$O(\|V\|)$
• given nodes u and v, checking if u and v are adjacent	$O(deg(u))$	$O(1)$
• space	$O(\|E\|+\|V\|)$	$O(\|V\|^2)$

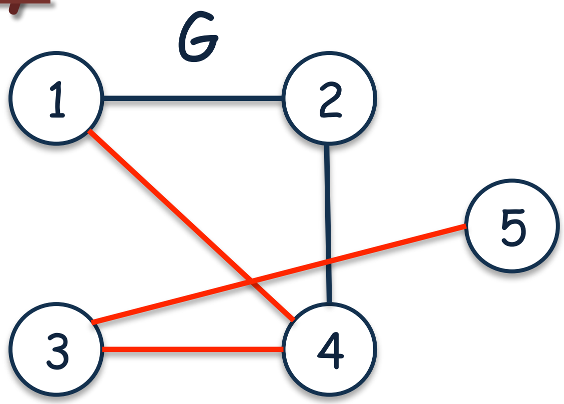
If graph is sparse, use adjacency list;
if graph is dense, use adjacency matrix

Graph Theory



- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct

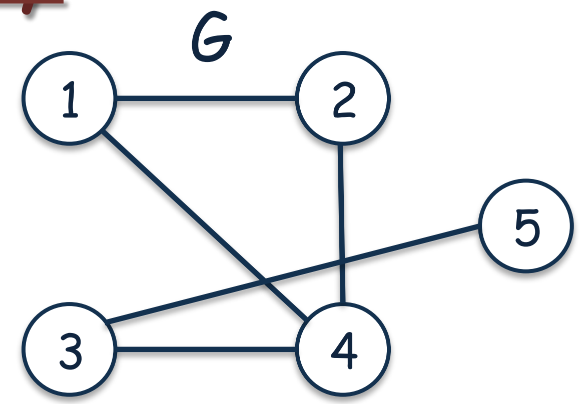
Graph Theory



5, 3, 4, 1 is a simple path in G

- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct

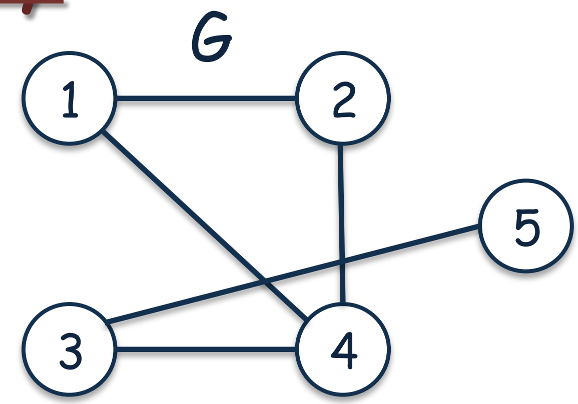
Graph Theory



- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct

- nodes u and v are called connected if there is a path between them. A graph is connected if there is a path between every pair of nodes

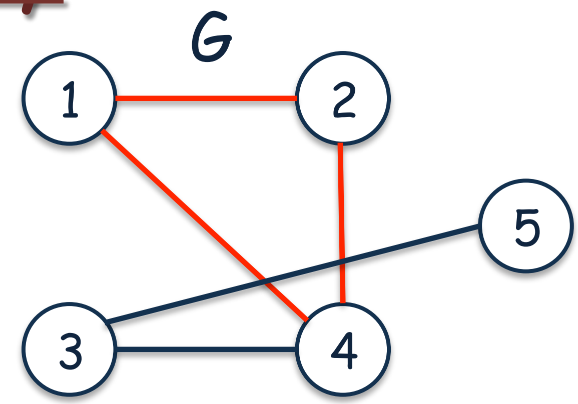
Graph Theory



- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct
- nodes u and v are called connected if there is a path between them. A graph is connected if there is a path between every pair of nodes
- a cycle is a path $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. A cycle is simple if first k-1 nodes are distinct

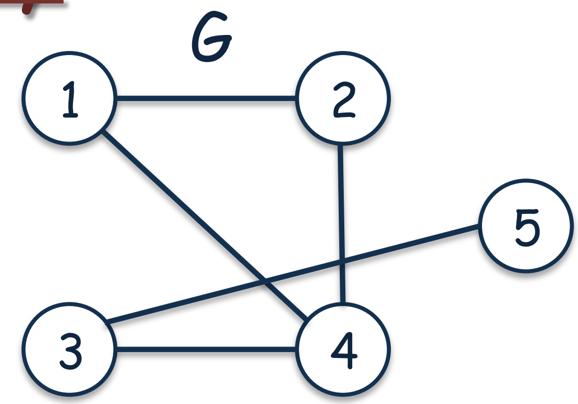
Graph Theory

4, 1, 2, 4 is a simple cycle in G



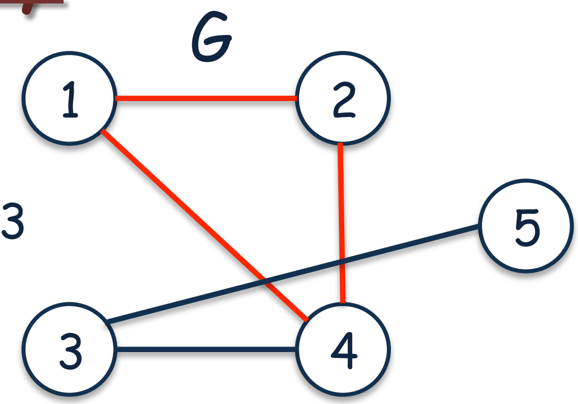
- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct
- nodes u and v are called connected if there is a path between them. A graph is connected if there is a path between every pair of nodes
- a cycle is a path $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. A cycle is simple if first k-1 nodes are distinct

Graph Theory



- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct
- nodes u and v are called connected if there is a path between them. A graph is connected if there is a path between every pair of nodes
- a cycle is a path $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. A cycle is simple if first k-1 nodes are distinct
- length of a path is the number of edges in the path

Graph Theory



4, 1, 2, 4 is a simple cycle with length 3

- a path in a graph is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that (v_i, v_j) is an edge in the graph.
  a path is simple if all nodes are distinct
- nodes u and v are called connected if there is a path between them. A graph is connected if there is a path between every pair of nodes
- a cycle is a path $v_1, v_2, \ldots, v_k$ such that $v_1 = v_k$. A cycle is simple if first k-1 nodes are distinct
- length of a path is the number of edges in the path

Graph Traversal

- One of the most fundemantal graph problems is to traverse every edge and every vertex in a graph

Graph Traversal

- One of the most fundemantal graph problems is to traverse every edge and every vertex in a graph
- Explore the graph in a systematic way :
    - make sure that each edge visited at most twice
    - don't miss anything

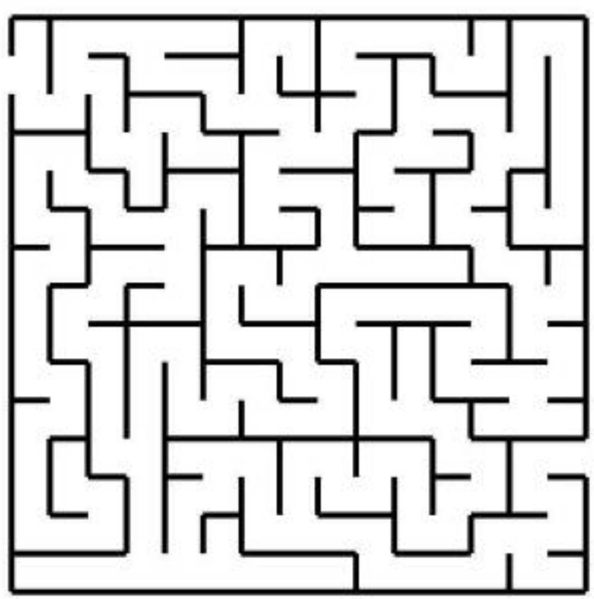
Graph Traversal

- One of the most fundemantal graph problems is to traverse every edge and every vertex in a graph

- Explore the graph in a systematic way :

  make sure that each edge visited at most twice

  don't miss anything

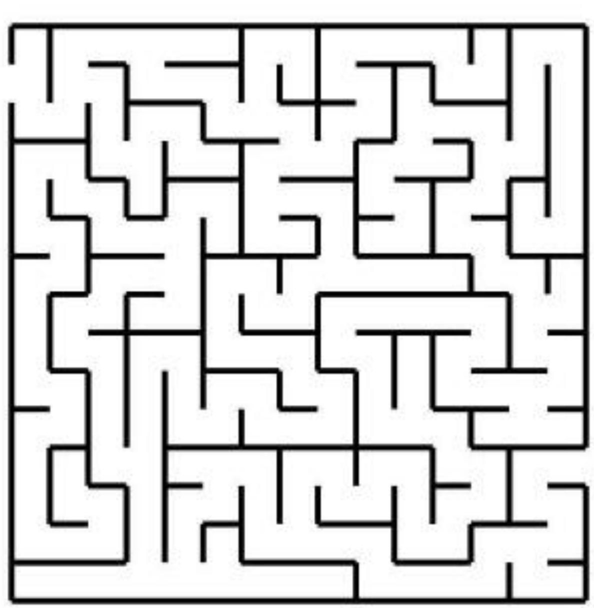


Graph Traversal

- One of the most fundemantal graph problems is to traverse every edge and every vertex in a graph
- Explore the graph in a systematic way :

  make sure that each edge visited at most twice

  don't miss anything



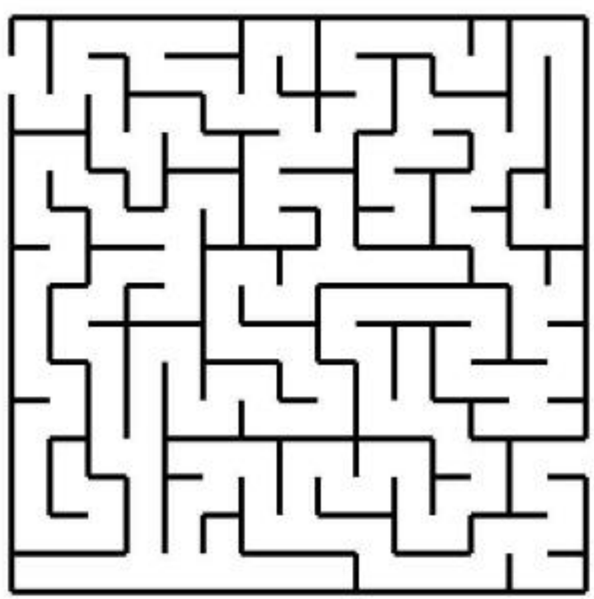
- each vertex denotes a junction and each edge denotes a hallway

Graph Traversal

- One of the most fundemantal graph problems is to traverse every edge and every vertex in a graph
- Explore the graph in a systematic way :

  make sure that each edge visited at most twice

  don't miss anything



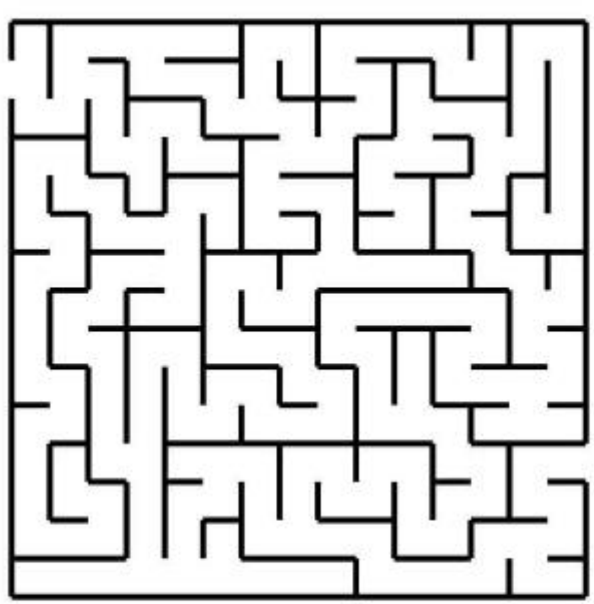
- each vertex denotes a junction and each edge denotes a hallway
- any traversal algorithm can be sufficient to get us out of any maze

Graph Traversal

- One of the most fundemantal graph problems is to traverse every edge and every vertex in a graph
- Explore the graph in a systematic way :

  make sure that each edge visited at most twice

  don't miss anything



- each vertex denotes a junction and each edge denotes a hallway
- any traversal algorithm can be sufficient to get us out of any maze
- For efficieny, make sure you don't get stuck (visiting same place over and over again)

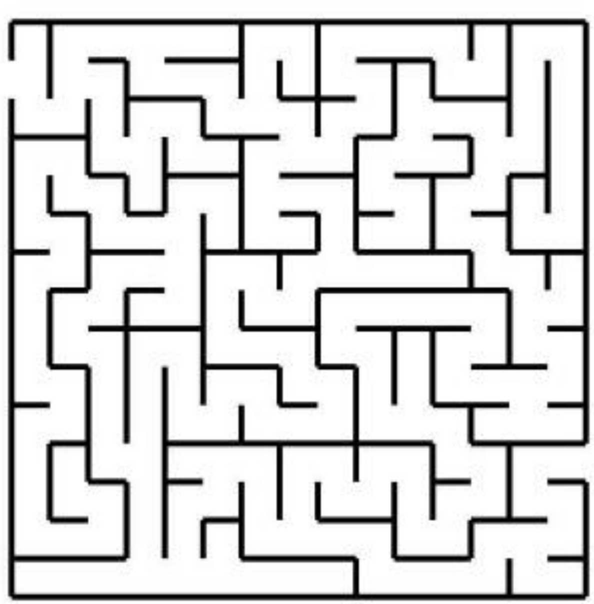
Graph Traversal

- One of the most fundemantal graph problems is to traverse every edge and every vertex in a graph

- Explore the graph in a systematic way :

  make sure that each edge visited at most twice

  don't miss anything



- each vertex denotes a junction and each edge denotes a hallway

- any traversal algorithm can be sufficient to get us out of any maze

- For efficieny, make sure you don't get stuck (visiting same place over and over again)

- For correctness, we do the traversal in a way that we get out of the maze

Graph Traversal

the key idea

Graph Traversal

the key idea

- mark each vertex when you first visit it
- keep track of what you haven't yet completely explored

Graph Traversal

the key idea

- mark each vertex when you first visit it
- keep track of what you haven't yet completely explored

possible three states for each vertex

Graph Traversal

the key idea

- mark each vertex when you first visit it
- keep track of what you haven't yet completely explored

possible three states for each vertex

undiscovered discovered processed

Graph Traversal

the key idea

- mark each vertex when you first visit it
- keep track of what you haven't yet completely explored

possible three states for each vertex

undiscovered	discovered	processed
initial state for a vertex	the vertex has been visited but all of its incident edges have not been checked out	the vertex and all of its incident edges have been visited

Graph Traversal

the key idea

- mark each vertex when you first visit it
- keep track of what you haven't yet completely explored

possible three states for each vertex

undiscovered	discovered	processed
initial state for a vertex	the vertex has been visited but all of its incident edges have not been checked out	the vertex and all of its incident edges have been visited

state of each vertex changes from left to right

Breadth First Search

- instead of going deep in a graph, just go in cross-wise fashion

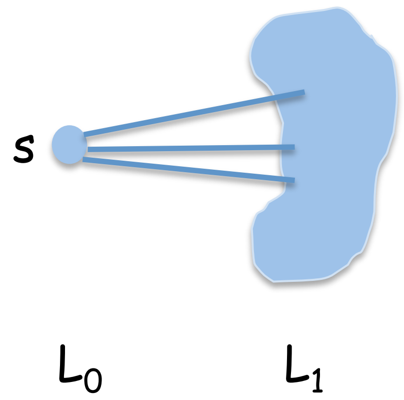
Breadth First Search

- instead of going deep in a graph, just go in cross-wise fashion
- explore the graph outward from a starting point (a node s) in all possible directions

Breadth First Search

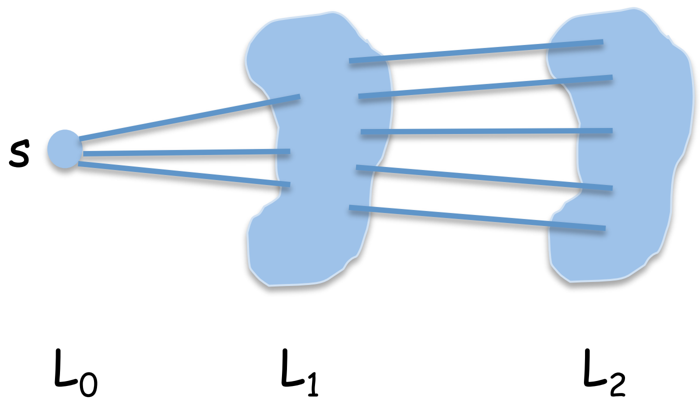
- instead of going deep in a graph, just go in cross-wise fashion

- explore the graph outward from a starting point (a node s) in all possible directions – add one layer of nodes at a time



Breadth First Search

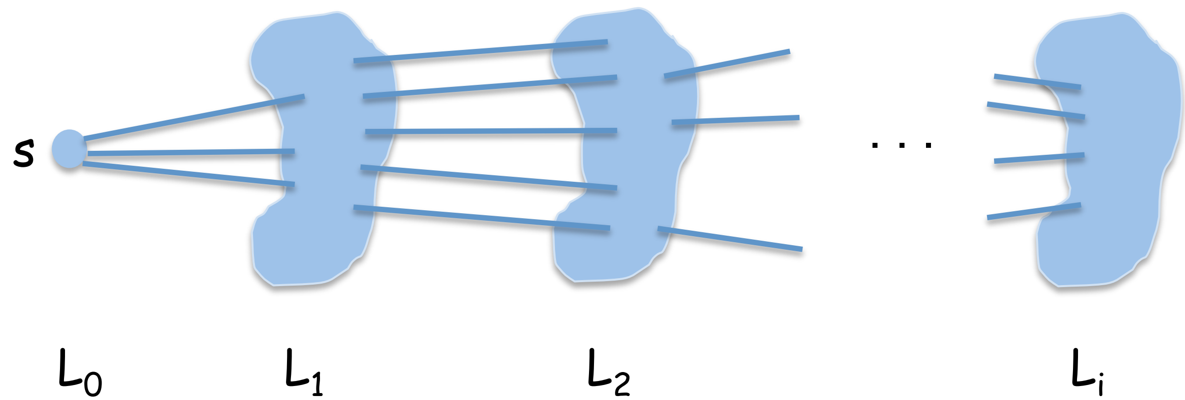
- instead of going deep in a graph, just go in cross-wise fashion
- explore the graph outward from a starting point (a node s) in all possible directions – add one layer of nodes at a time



Breadth First Search

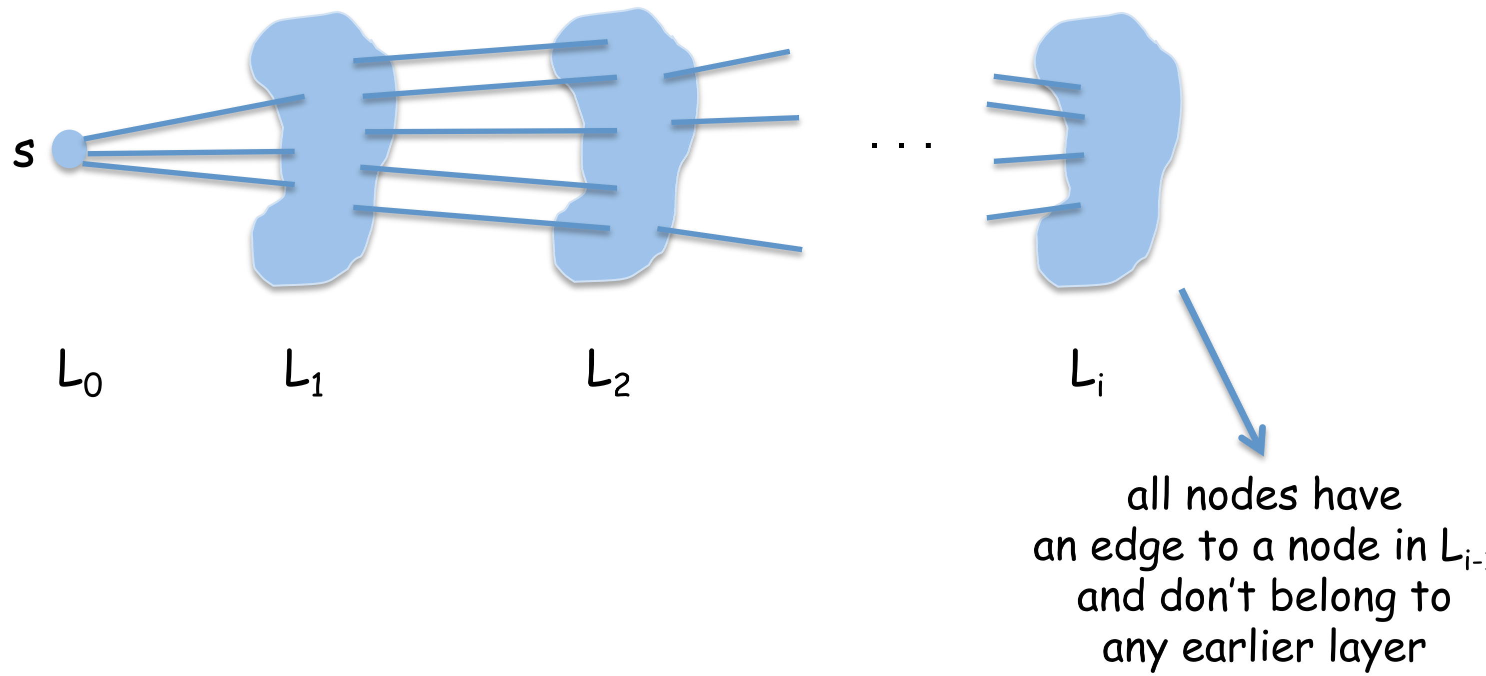
- instead of going deep in a graph, just go in cross-wise fashion

- explore the graph outward from a starting point (a node s) in all possible directions – add one layer of nodes at a time



Breadth First Search

- instead of going deep in a graph, just go in cross-wise fashion
- explore the graph outward from a starting point (a node s) in all possible directions – add one layer of nodes at a time



Breadth First Search

- every node u is associated with three parameters :

distance parent color

Breadth First Search

- every node u is associated with three parameters :

distance

The length of the shortest path from s to u

parent

u's predecessor on the shortest path from s to u

color

shows the state of u
white : undiscovered
gray : discovered
Black : processed

Breadth First Search

- every node u is associated with three parameters :

distance

The length of the shortest path from s to u

parent

u's predecessor on the shortest path from s to u

color

shows the state of u
white : undiscovered
gray : discovered
Black : processed

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
 . . .
```

Breadth First Search

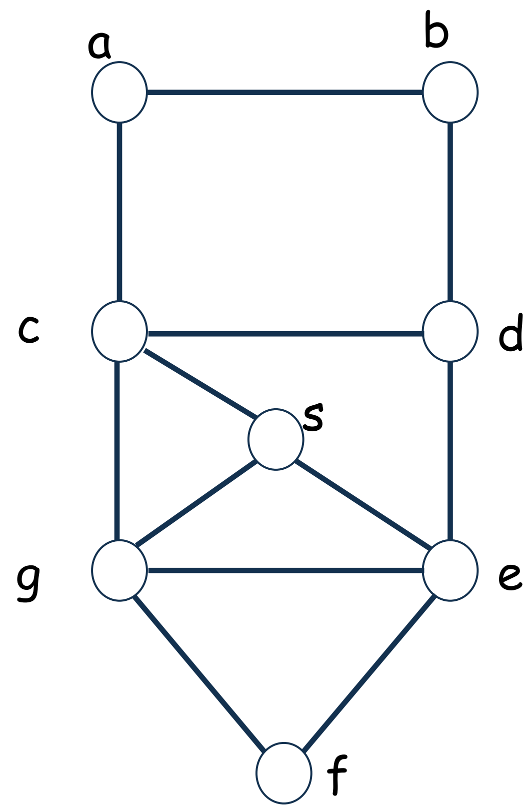
BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

Breadth First Search

BFS(G,s)

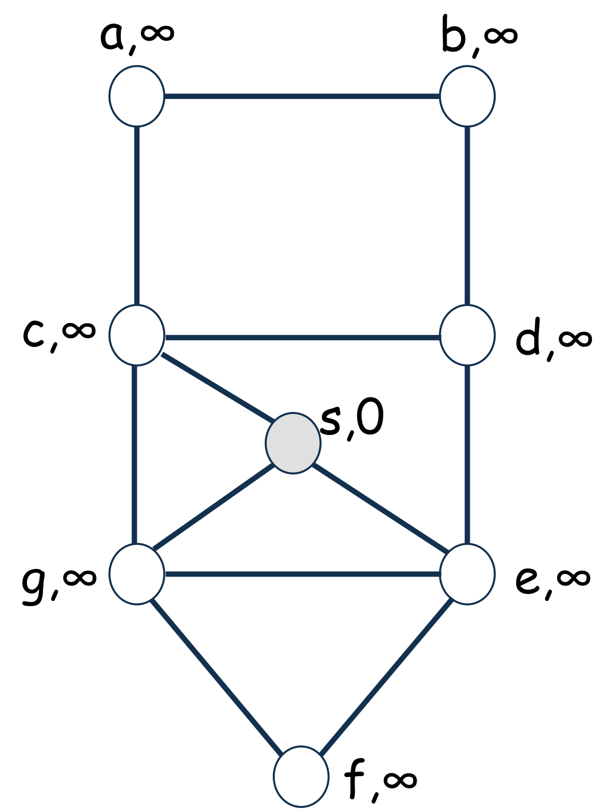
```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v ∈ Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```



Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

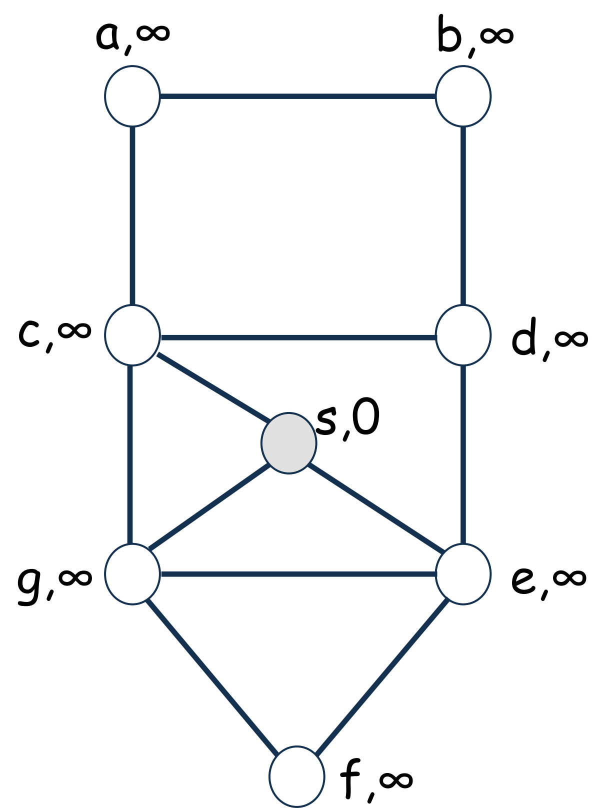


Q = { }

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v ∈ Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

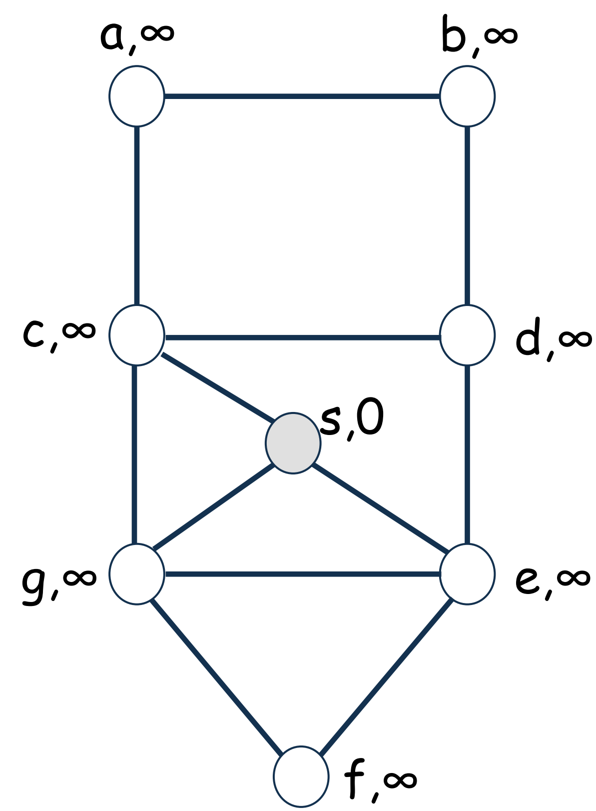


Q = {s}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

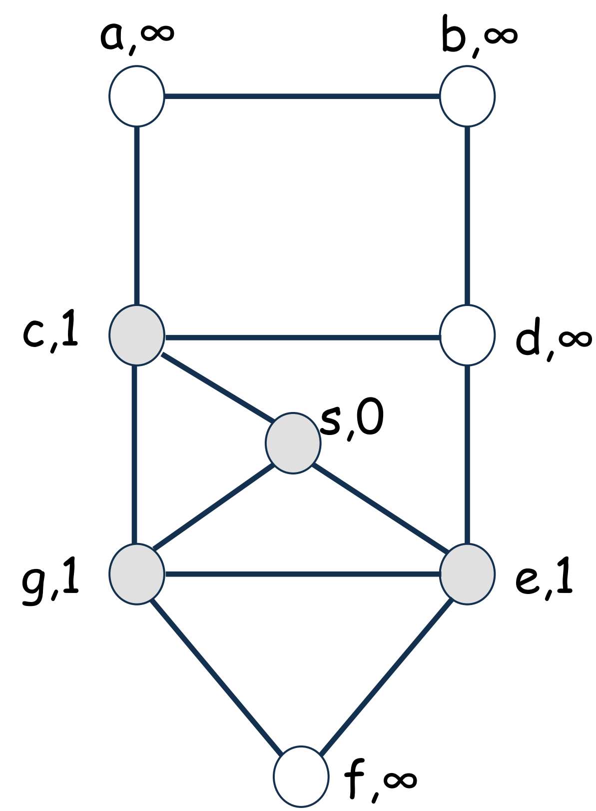


s   Q = { }

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

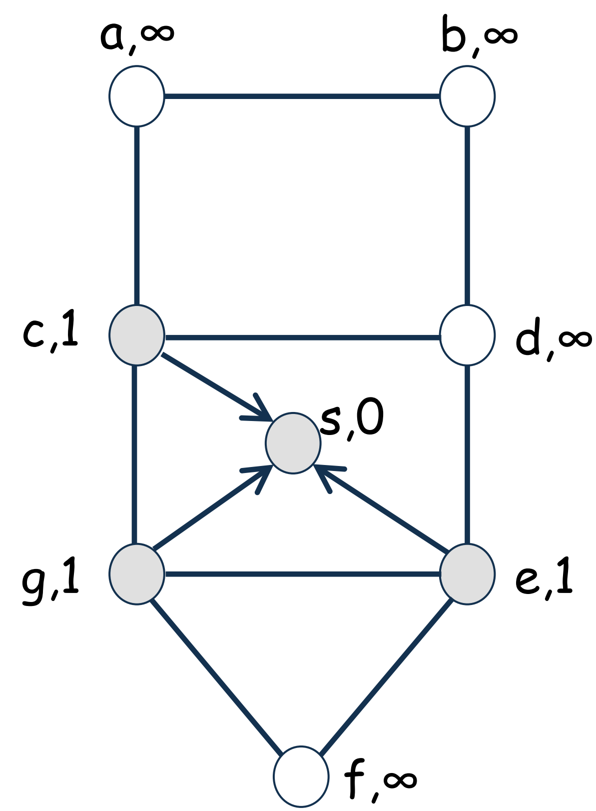


s Q = { }

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

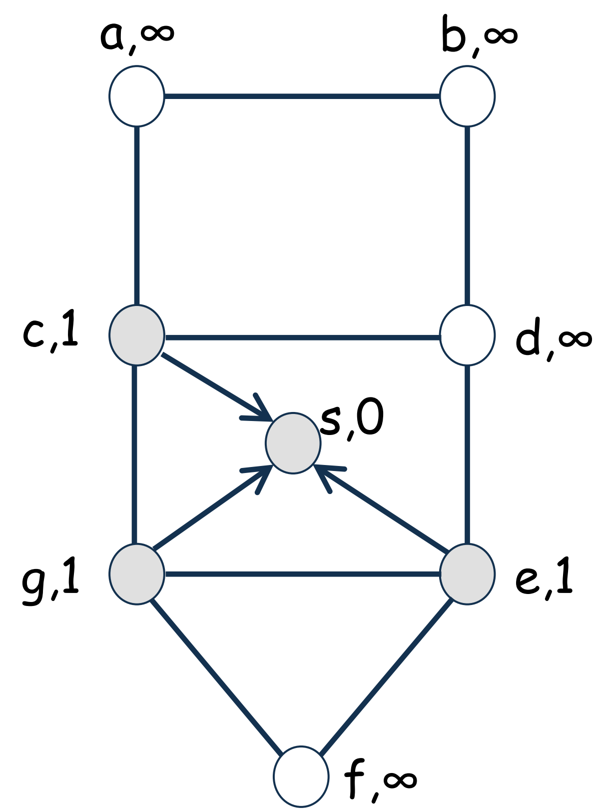


s Q = { }

Breadth First Search

BFS(G,s)

```
for each vertex u of V
     u.color = white
     u.dis = ∞
     u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
    u = Dequeu(Q)
    for each v∈Adj(u)
        if v.color = white
            v.color = gray
            v.dis = u.dis + 1
            v.par = u
            Enqueue(Q,v)
    u.color = black
```

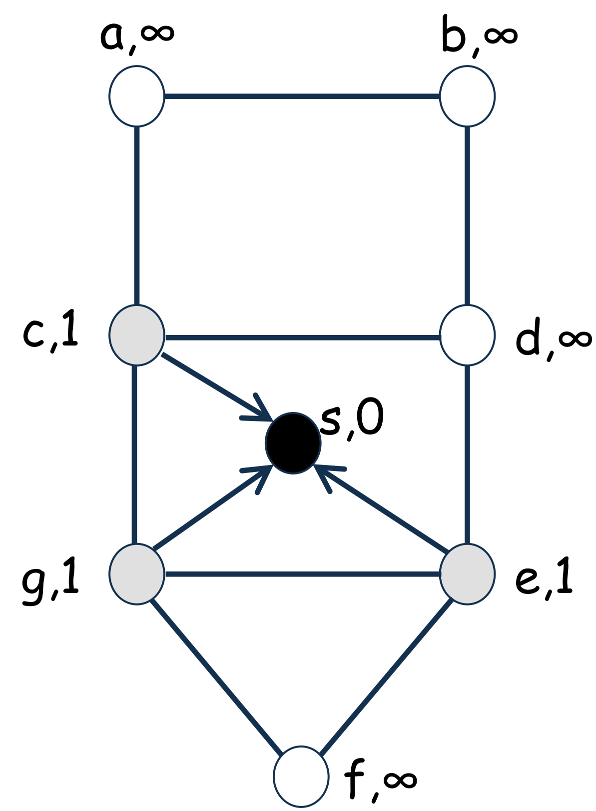


s Q = {c g e}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

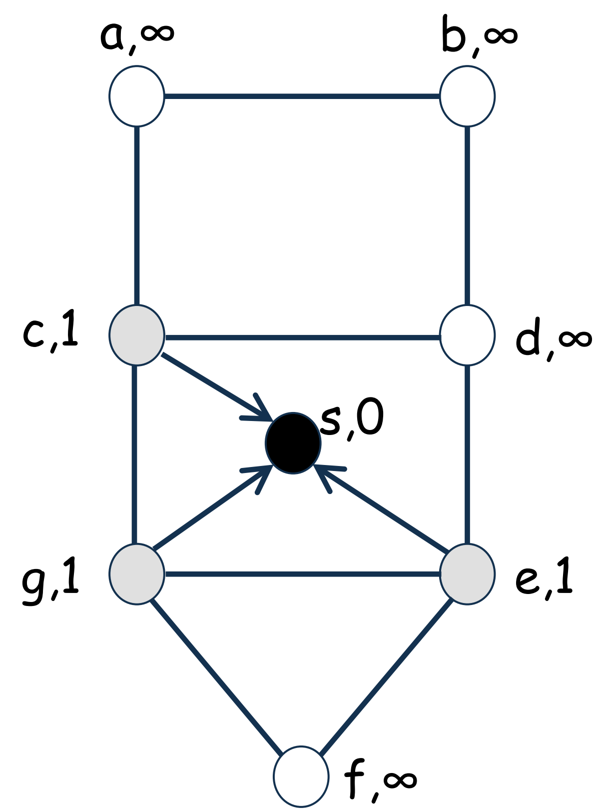


s Q = {c g e}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

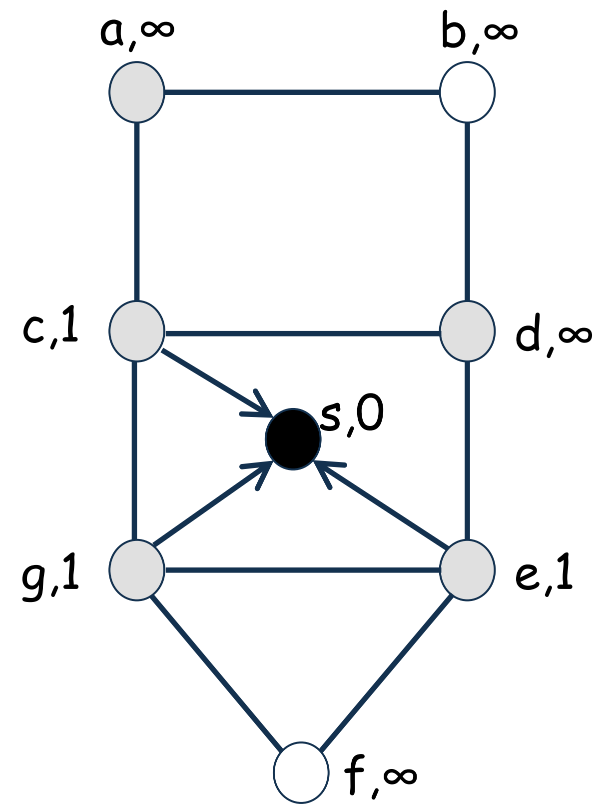


c   Q = {g e}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

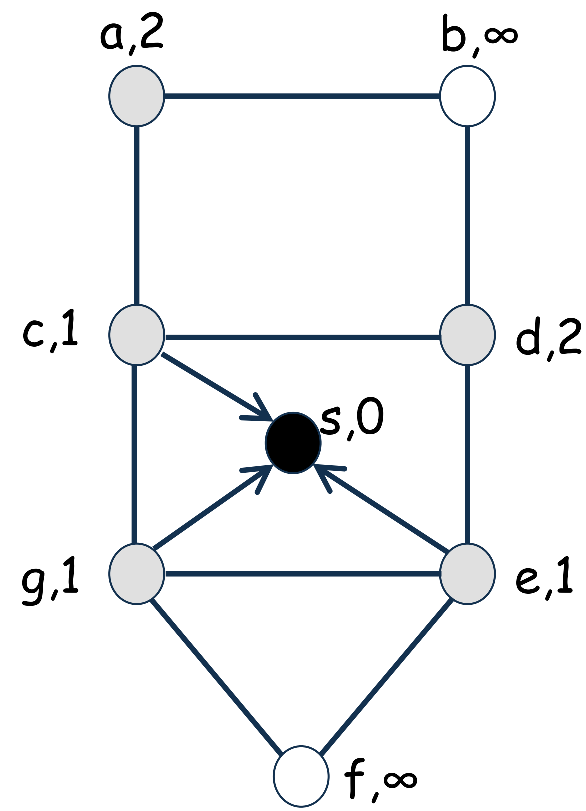


c   Q = {g e}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

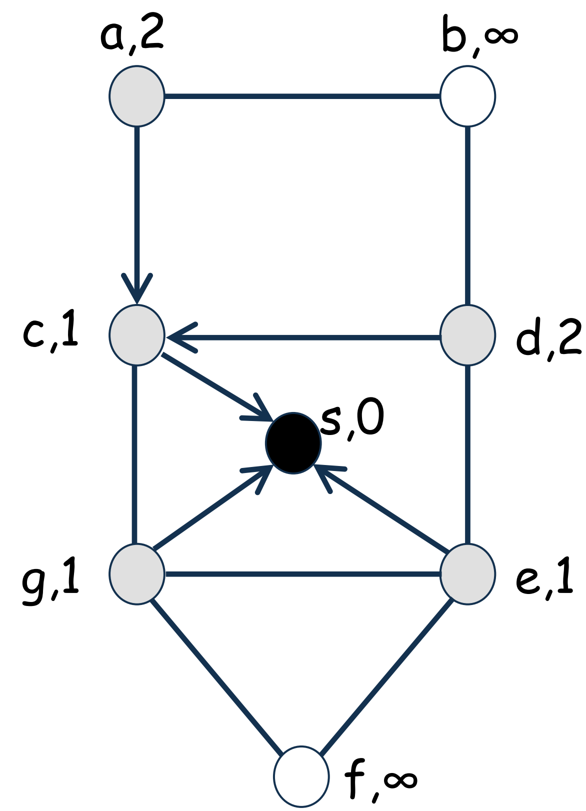


c Q = {g e}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

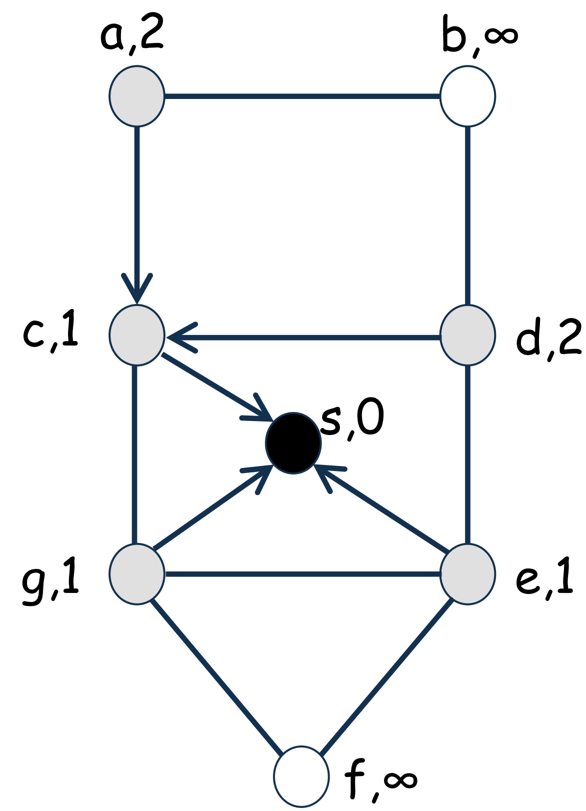


c Q = {g e}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

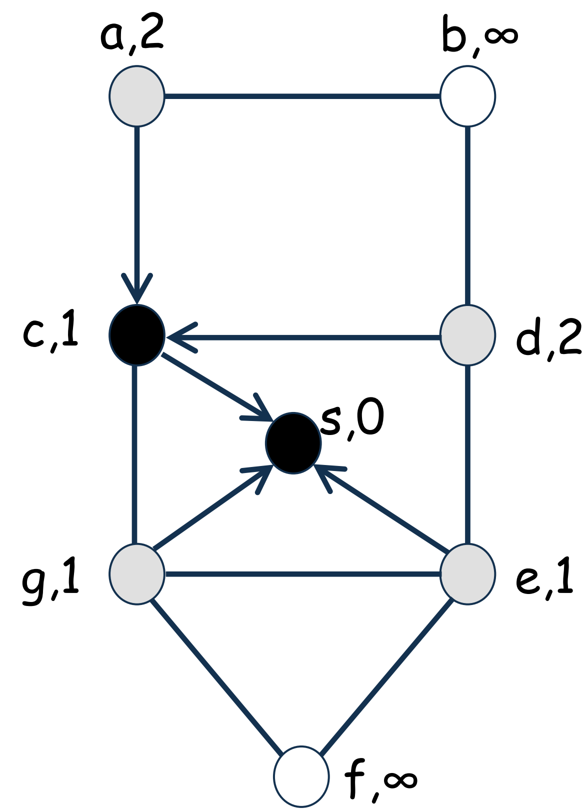


c   Q = {g e a d}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

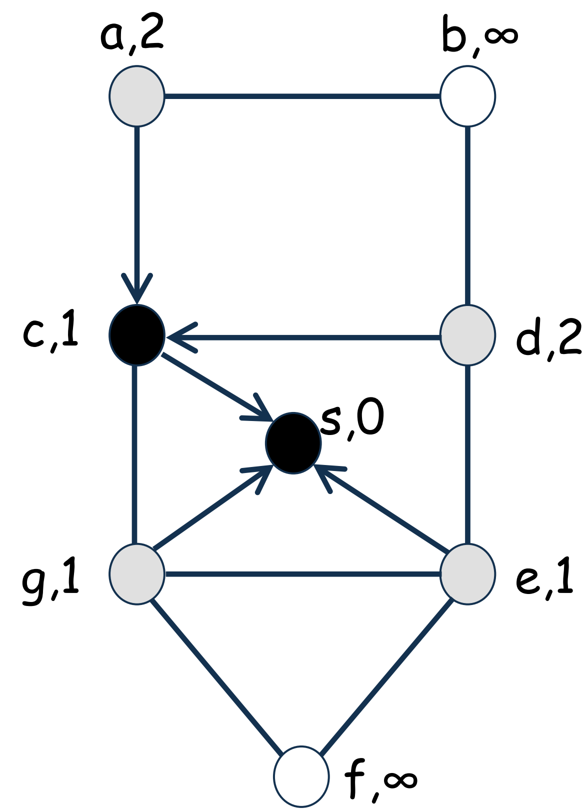


c   Q = {g e a d}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

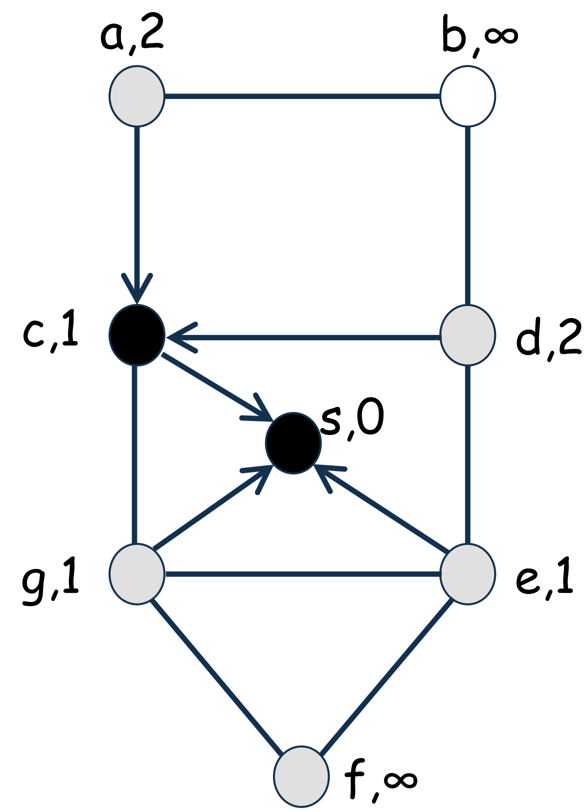


g   Q = {e a d}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

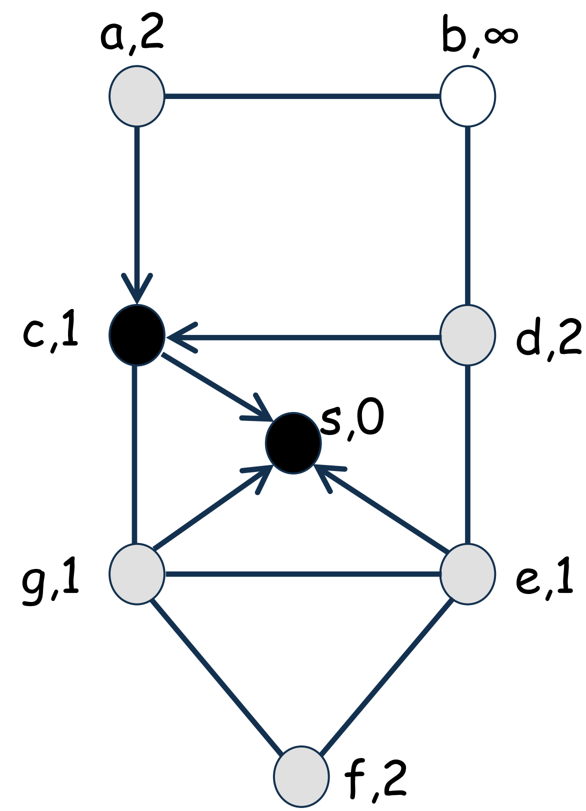


g Q = {e a d}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

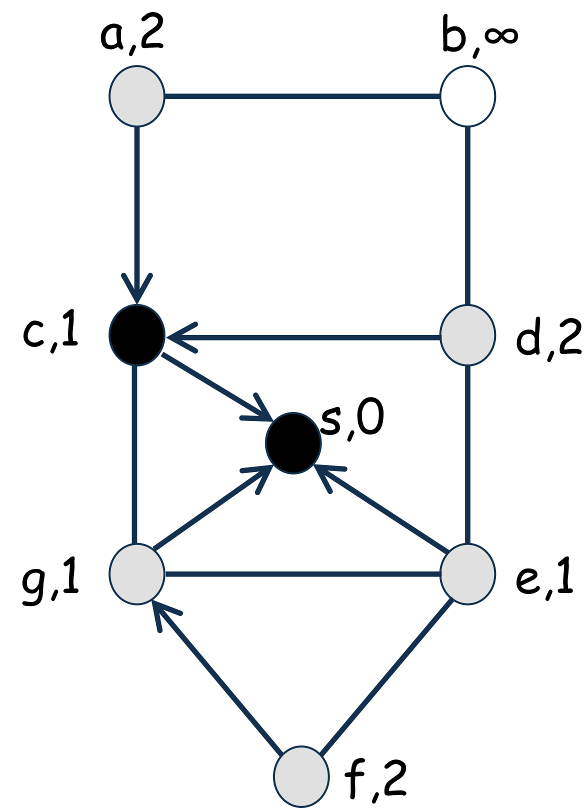


g Q = {e a d}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

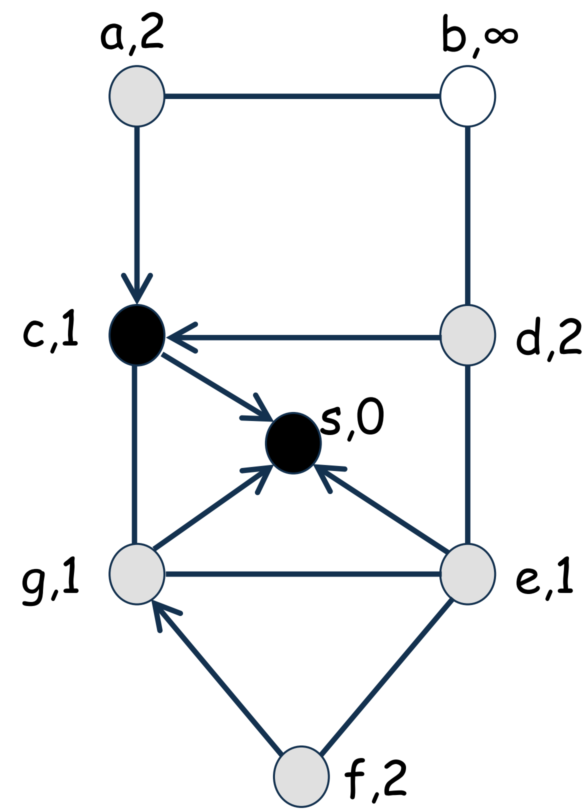


g Q = {e a d}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

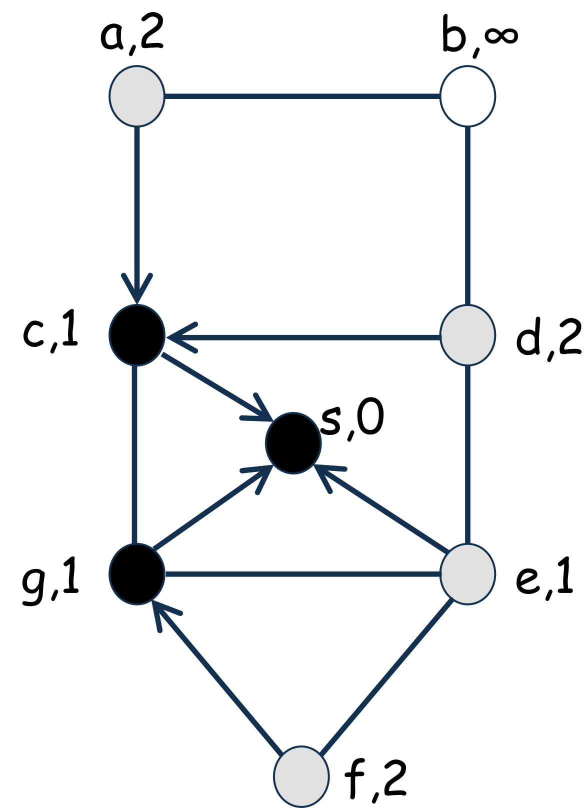


g Q = {e a d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

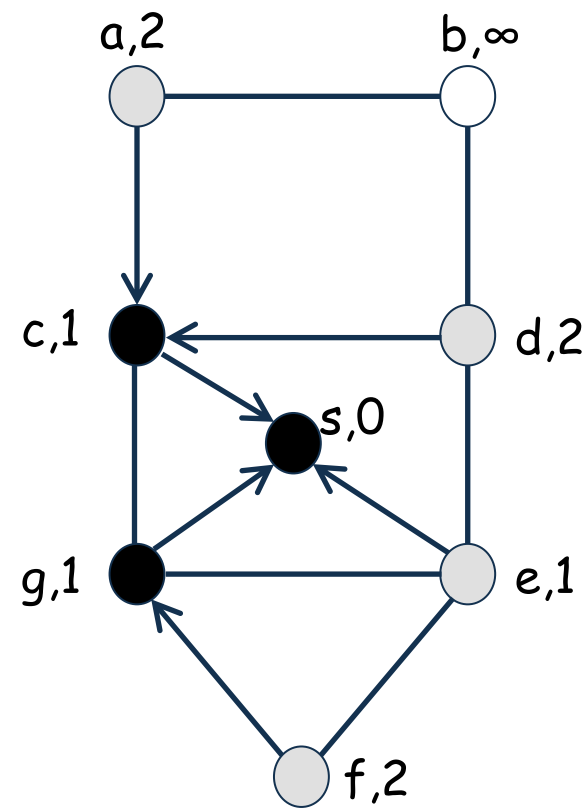


g Q = {e a d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
     u.color = white
     u.dis = ∞
     u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
    u = Dequeu(Q)
    for each v∈Adj(u)
        if v.color = white
            v.color = gray
            v.dis = u.dis + 1
            v.par = u
            Enqueue(Q,v)
    u.color = black
```

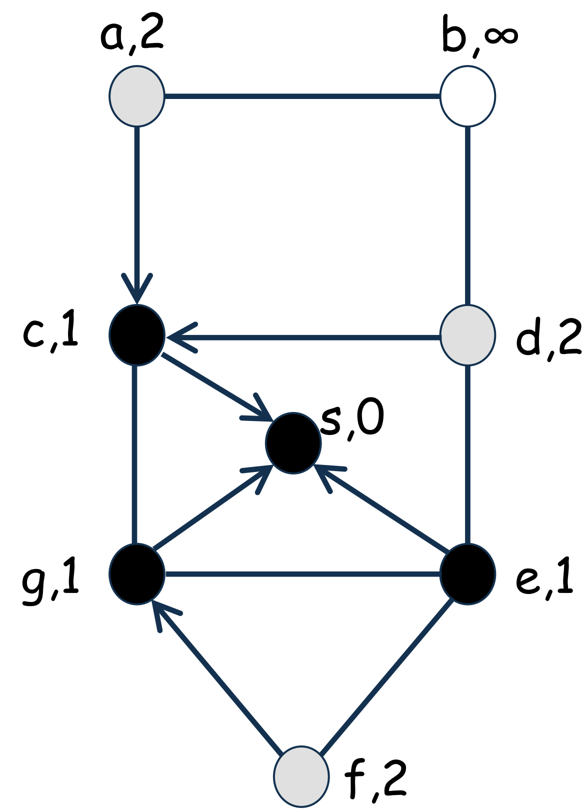


e Q = {a d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

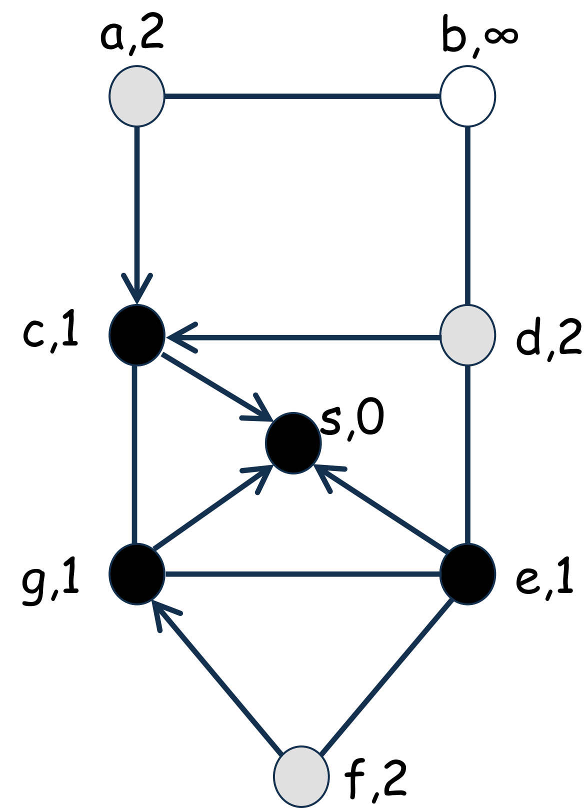


e Q = {a d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

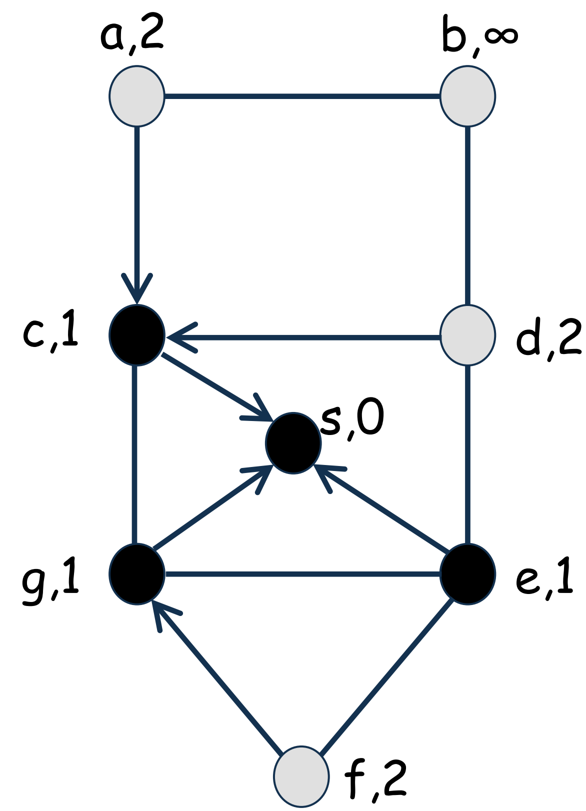


a Q = {d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

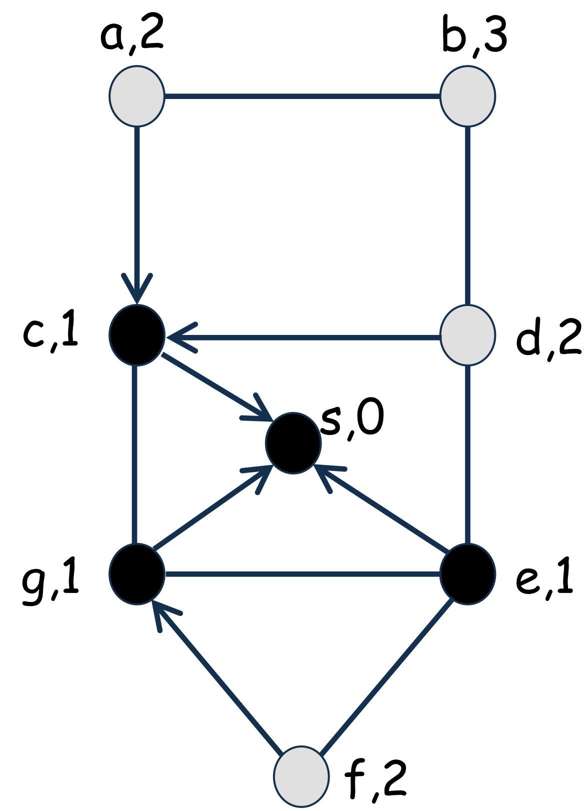


a   Q = {d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

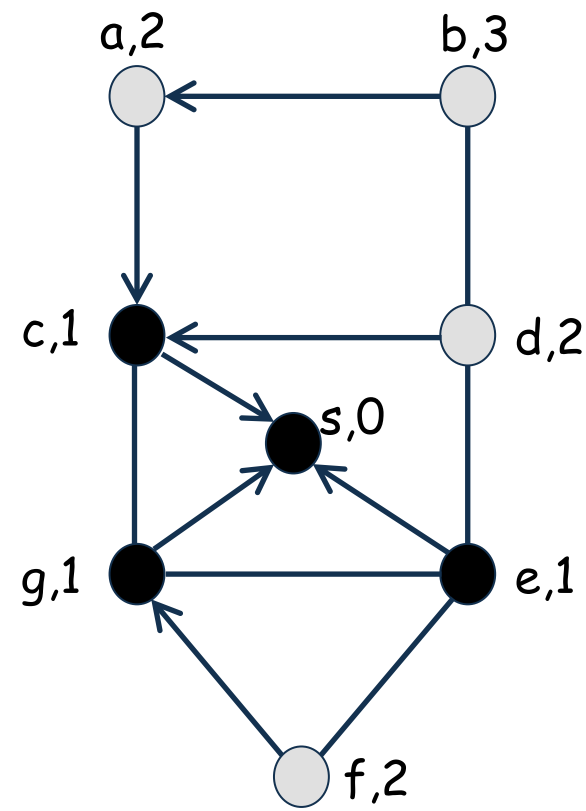


a Q = {d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

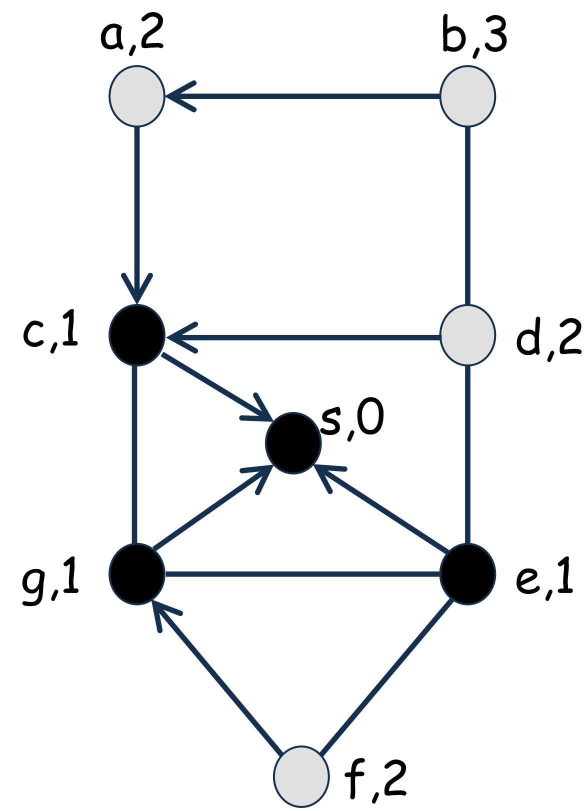


a Q = {d f}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

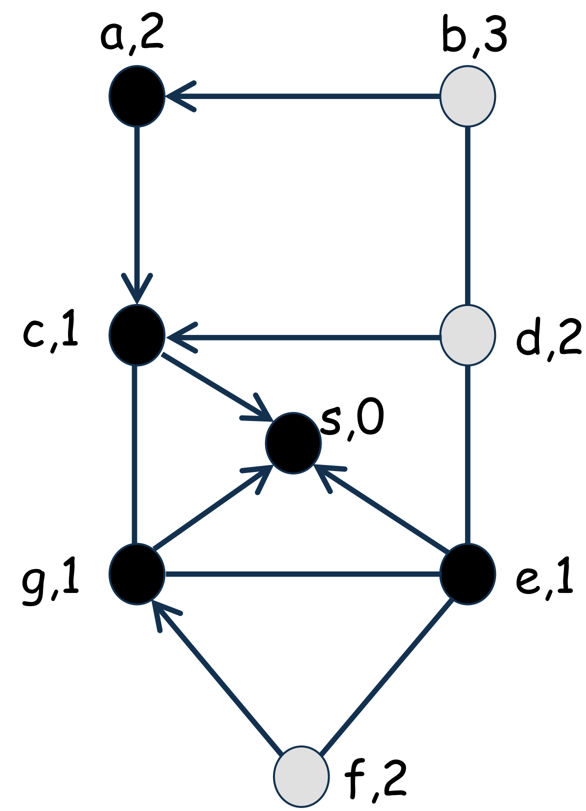


a Q = {d f b}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

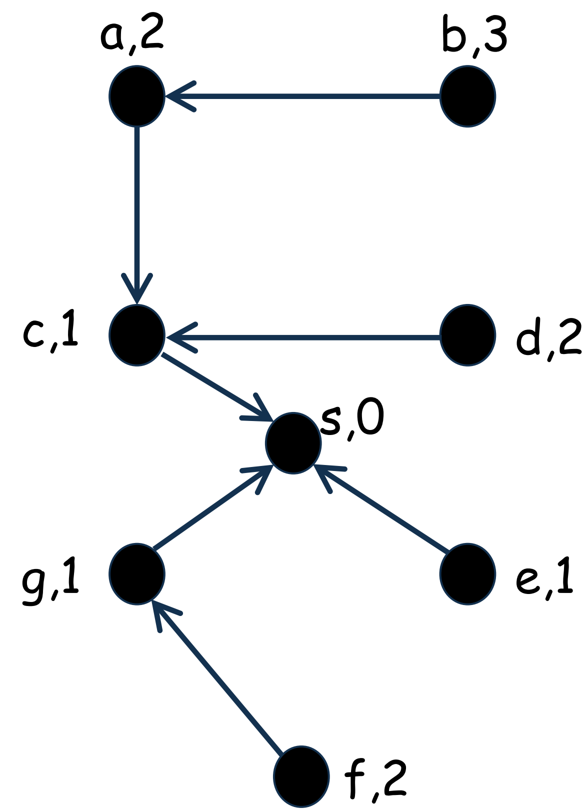


a Q = {d f b}

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
        u = Dequeu(Q)
        for each v∈Adj(u)
                if v.color = white
                        v.color = gray
                        v.dis = u.dis + 1
                        v.par = u
                        Enqueue(Q,v)
        u.color = black
```

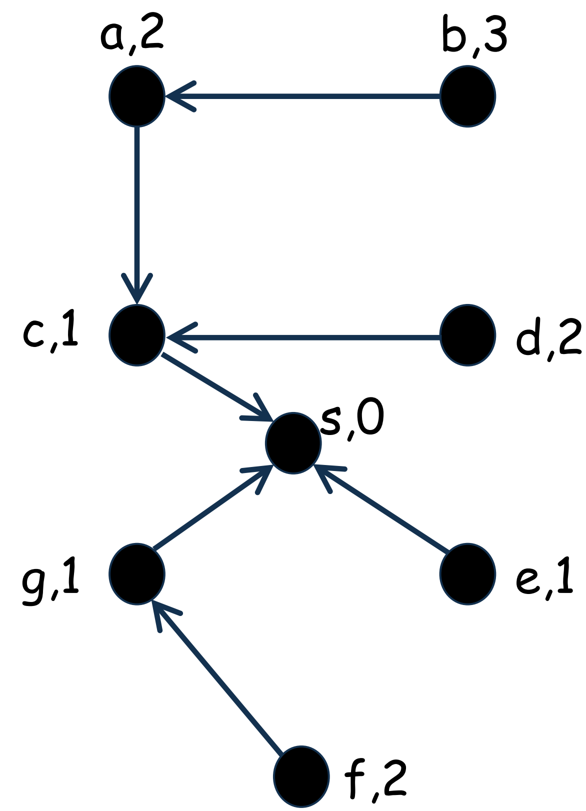


b Q = { }

Breadth First Search

BFS(G,s)

```
for each vertex u of V          ⎫
        u.color = white         ⎬ O(|V|)
        u.dis = ∞               ⎪
        u.par = nil             ⎭
s.color = gray                  ⎫
s.dis = 0                       ⎬ O(1)
initialize an empty queue Q     ⎪
Enqueu(Q,s)                     ⎭
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```



b Q = { }

Breadth First Search

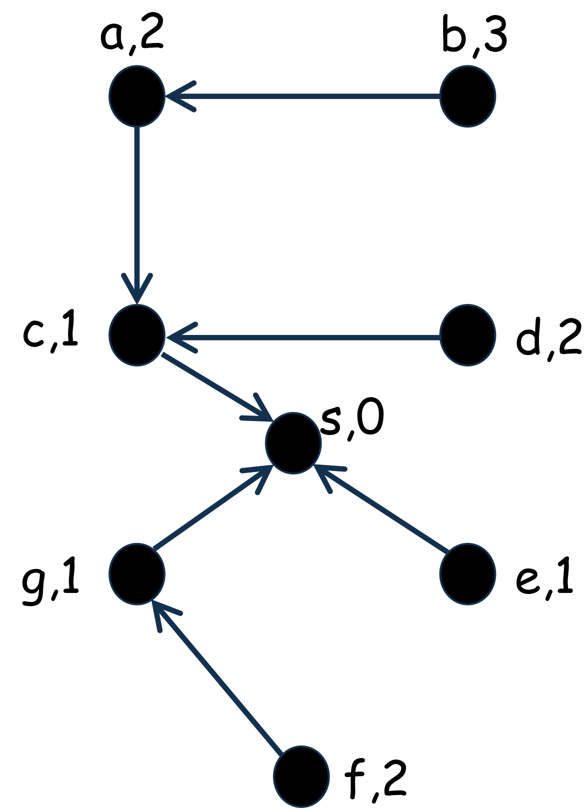
BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

O(|V|)

O(1)

O(|E|)



b Q = { }

Breadth First Search

BFS(G,s)

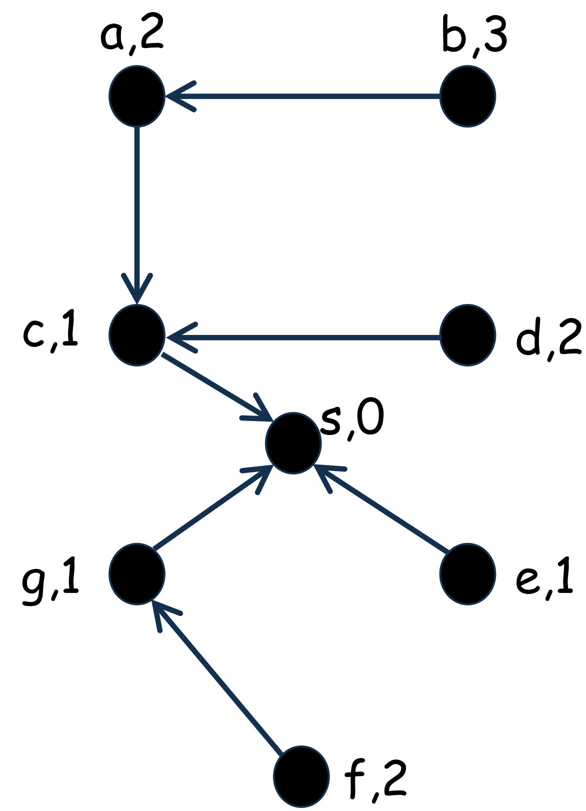
```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
      u = Dequeu(Q)
      for each v∈Adj(u)
            if v.color = white
                  v.color = gray
                  v.dis = u.dis + 1
                  v.par = u
                  Enqueue(Q,v)
      u.color = black
```

$\left.\right\}$ O(|V|) (first four lines)

$\left.\right\}$ O(1) (s.color through Enqueu(Q,s))

$\left.\right\}$ O(|E|) (for each loop)

total O(|V| + |E|)



b Q = { }

Breadth First Search

BFS(G,s)

```
for each vertex u of V
        u.color = white
        u.dis = ∞
        u.par = nil
s.color = gray
s.dis = 0
initialize an empty queue Q
Enqueu(Q,s)
while Q ≠ ∅
        u = Dequeu(Q)
        for each v∈Adj(u)
                if v.color = white
                        v.color = gray
                        v.dis = u.dis + 1
                        v.par = u
                        Enqueue(Q,v)
        u.color = black
```

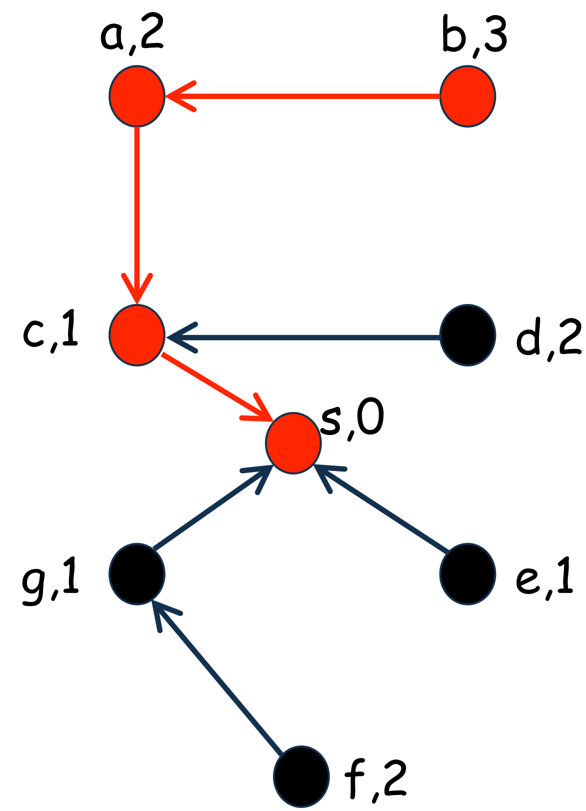
O(|V|)

O(1)

O(|E|)

total O(|V| + |E|)

parent pointer used to find the shortest path



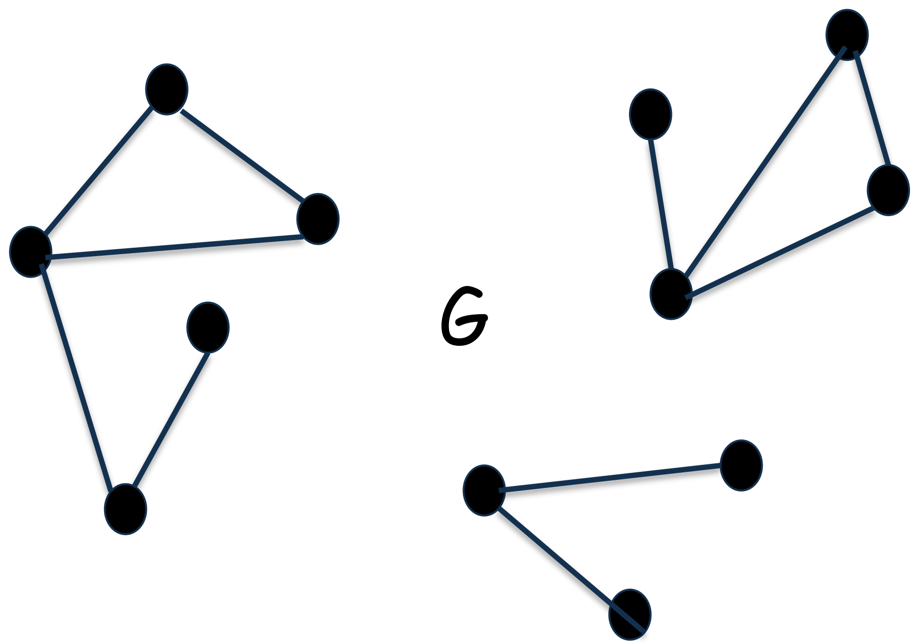
b Q = { }

Connected Components

- a connected component is a maximal subgraph where there is a path between any two nodes of it

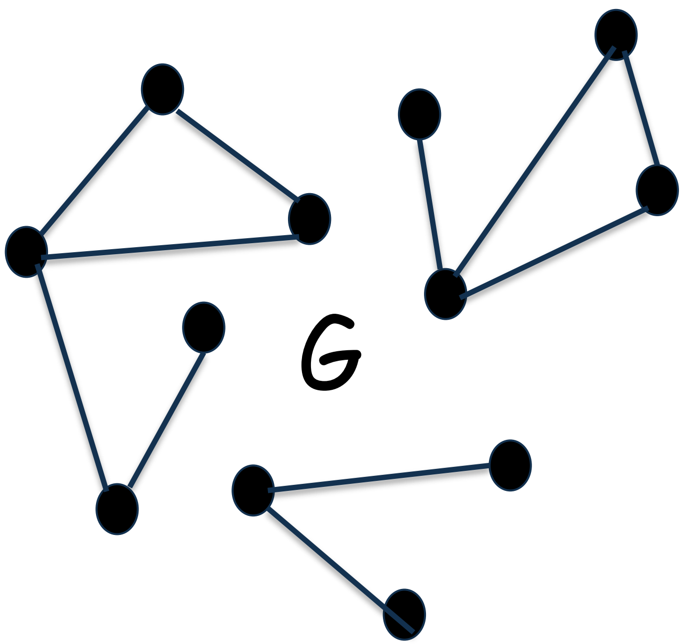
Connected Components

- a connected component is a maximal subgraph where there is a path between any two nodes of it
- a graph can be made up of seperate connected components



Connected Components

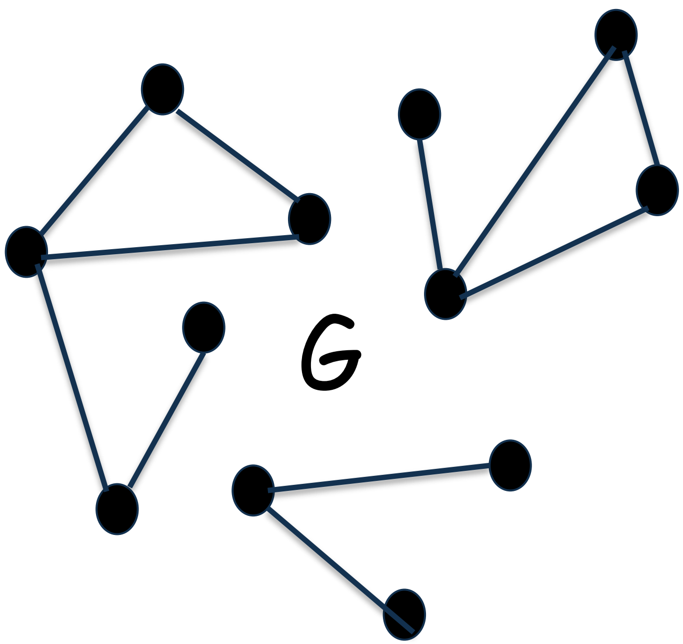
- a connected component is a maximal subgraph where there is a path between any two nodes of it
- a graph can be made up of seperate connected components



- find the number of connected components of a given graph (use BFS)

Connected Components

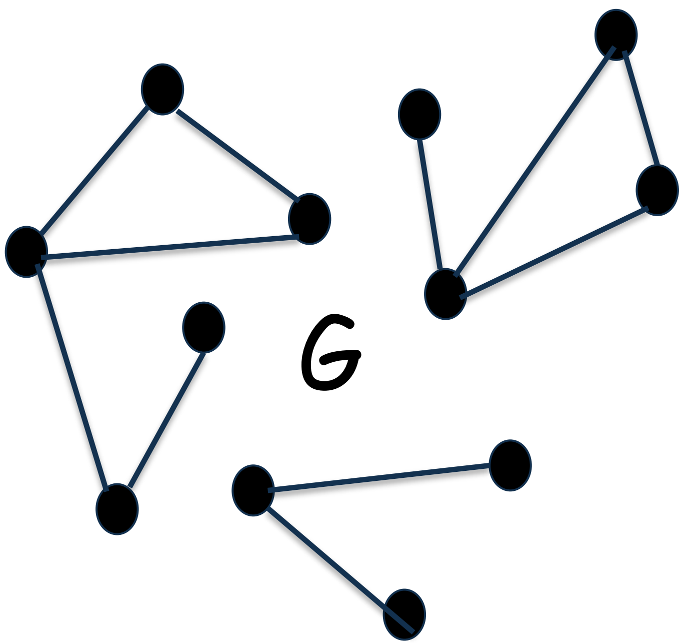
- a connected component is a maximal subgraph where there is a path between any two nodes of it
- a graph can be made up of seperate connected components



- find the number of connected components of a given graph (use BFS)
- start from the first vertex; any vertex we have discovered during this search should be part of same component.

Connected Components

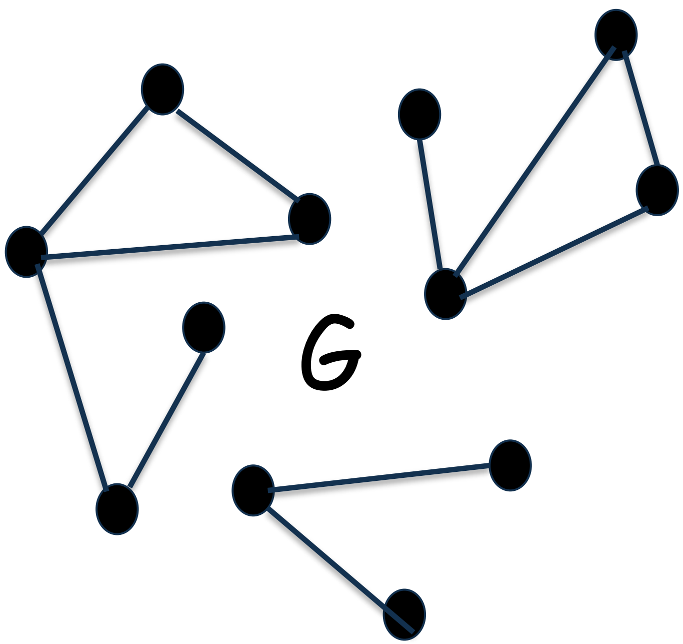
- a connected component is a maximal subgraph where there is a path between any two nodes of it
- a graph can be made up of seperate connected components



- find the number of connected components of a given graph (use BFS)
- start from the first vertex; any vertex we have discovered during this search should be part of same component.
- so, repeat the process with an undiscovered vertex.

Connected Components

- a connected component is a maximal subgraph where there is a path between any two nodes of it
- a graph can be made up of seperate connected components

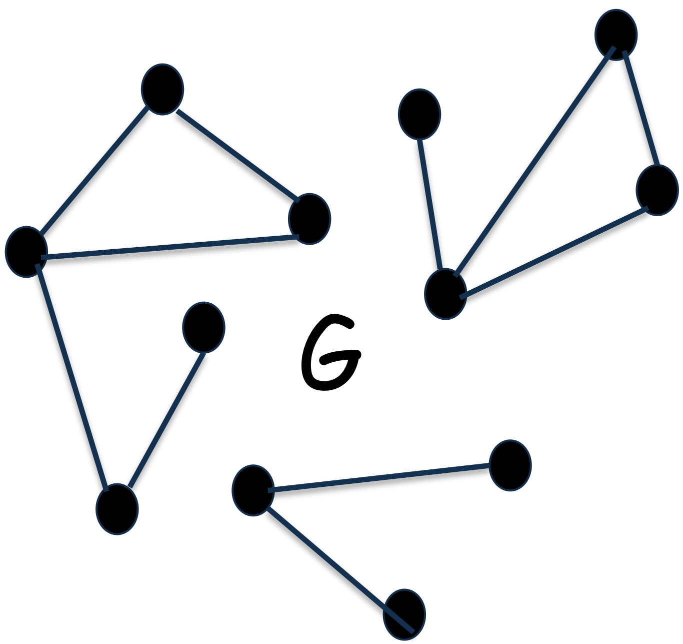


- find the number of connected components of a given graph (use BFS)
- start from the first vertex; any vertex we have discovered during this search should be part of same component.
- so, repeat the process with an undiscovered vertex.

```
int num = 0
for (i=1 to n)
      if (v_i has not been discovered)
            num = num + 1
            BFS(G, v_i)
return num
```

Connected Components

- a connected component is a maximal subgraph where there is a path between any two nodes of it
- a graph can be made up of seperate connected components



- It would be useful to clasify each vertex by which connected component it belongs
- When we run BFS on G from v, we mark each vertex as being owned by v.
- If we iterate through all vertices, each vertex will be marked by its owner that represents a different connected component

Depth First Search

- instead of going cross-wise, just go deep in the graph

Depth First Search

- instead of going cross-wise, just go deep in the graph
- when we process a node u, we pick each neighbor v of u in order

Depth First Search

- instead of going cross-wise, just go deep in the graph

- when we process a node u, we pick each neighbor v of u in order

  at the time of checking each v, we check again each neighbor of v in order

  only after processing all descendants of v, we pass to the next neighbor of u

Depth First Search

- instead of going cross-wise, just go deep in the graph
- when we process a node u, we pick each neighbor v of u in order

  at the time of checking each v, we check again each neighbor of v in order

  only after processing all descendants of v, we pass to the next neighbor of u

- the process continues until all verices reachable from the source have been discovered
- if any undiscovered vertices remain, choose one of them as new source and repeat the process

Depth First Search

- every node u is associated with four parameters :

Depth First Search

- every node u is associated with four parameters :

discovery finish parent color

Depth First Search

- every node u is associated with four parameters :

discovery

the time we have
discovered the node u

finish

the time we have
processed the node u

parent

u's predecessor on
the shortest path
from s to u

color

shows the state of u
white : undiscovered
gray : discovered
Black : processed

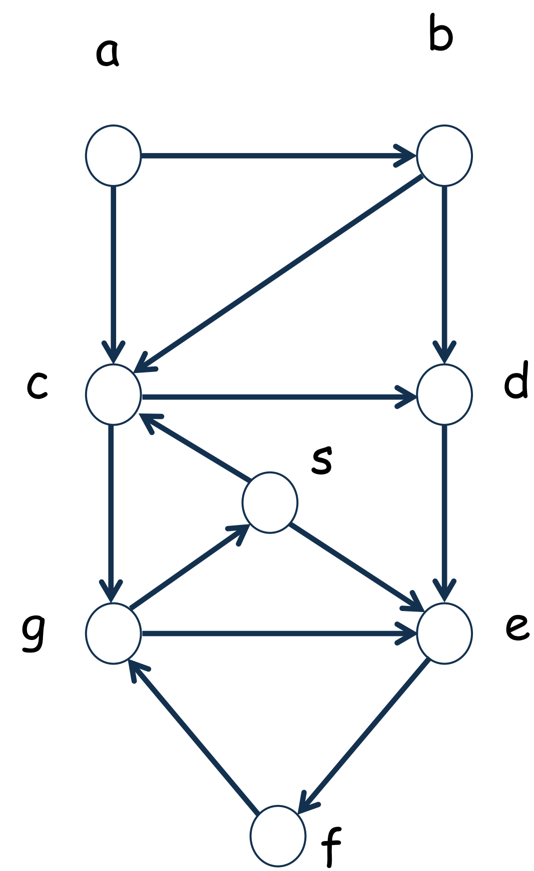
Depth First Search

<u>DFS(G)</u>

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
     if (v.color = white)
          v.par = u
          DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



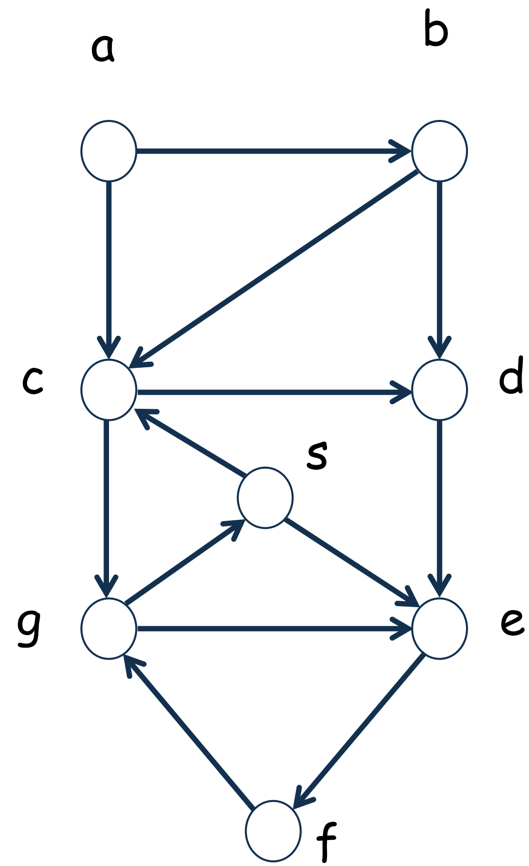
Depth First Search

<u>DFS(G)</u>

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```

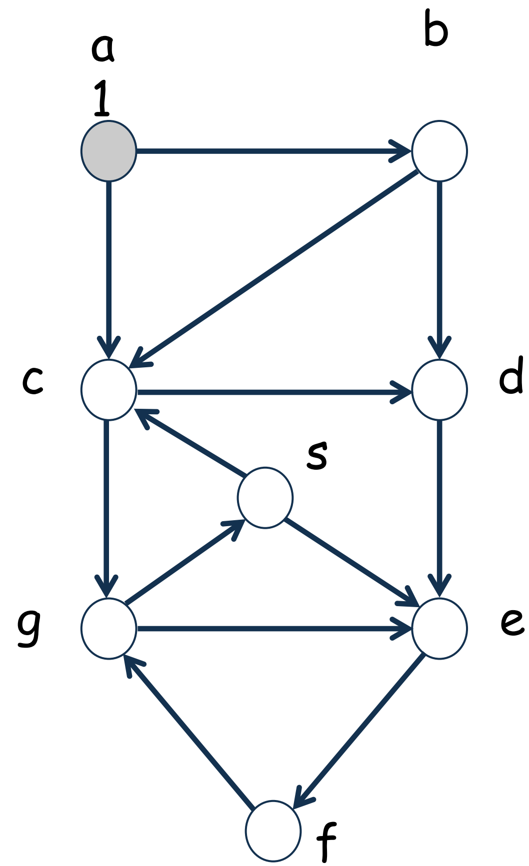


time = 0

Depth First Search

```
DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
            DFS_Visit(u)
```

```
DFS_Visit(u)
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
    if (v.color = white)
        v.par = u
        DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



time = 1

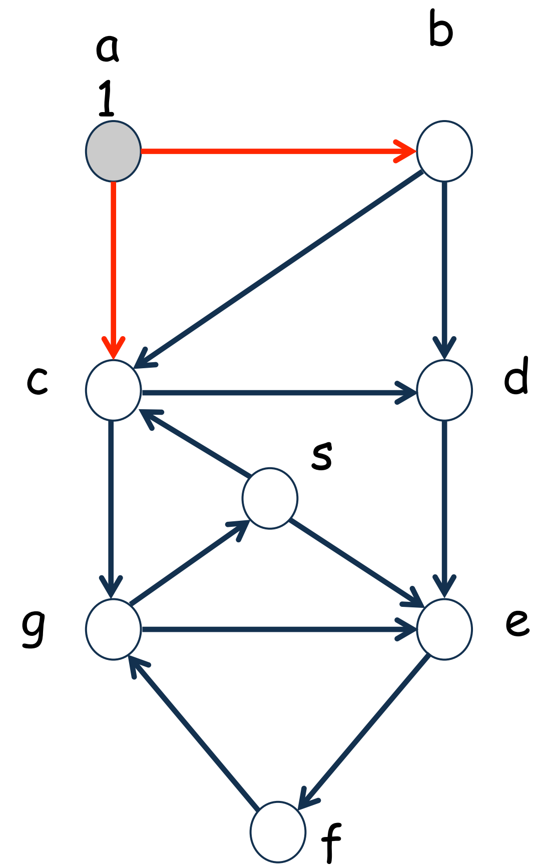
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
                DFS_Visit(u)
```

DFS_Visit(u)

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



time = 1

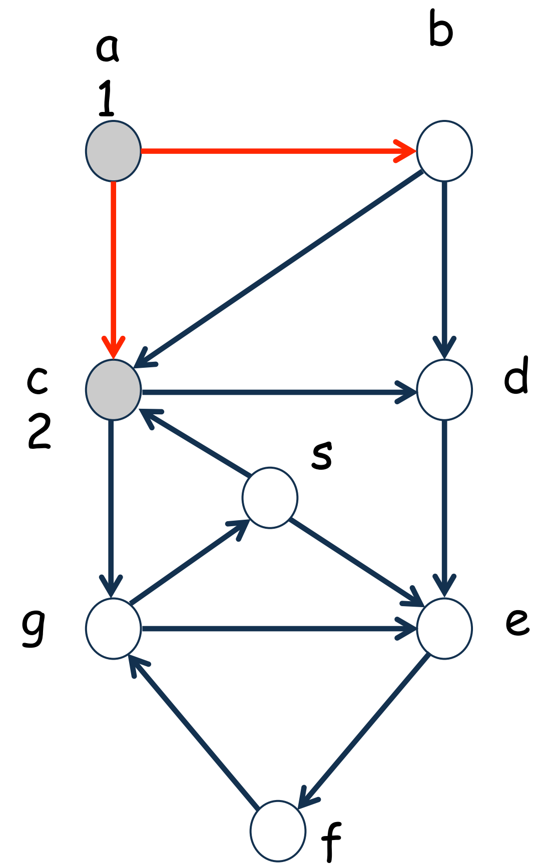
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

DFS_Visit(u)

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



time = 2

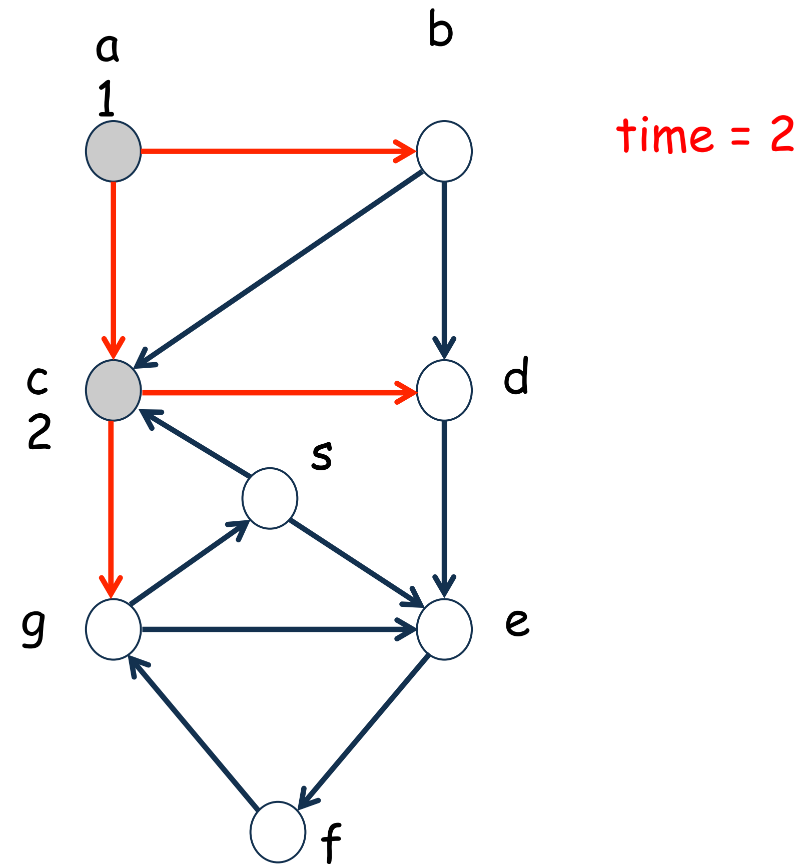
Depth First Search

<u>DFS(G)</u>

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
     if (v.color = white)
          v.par = u
          DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



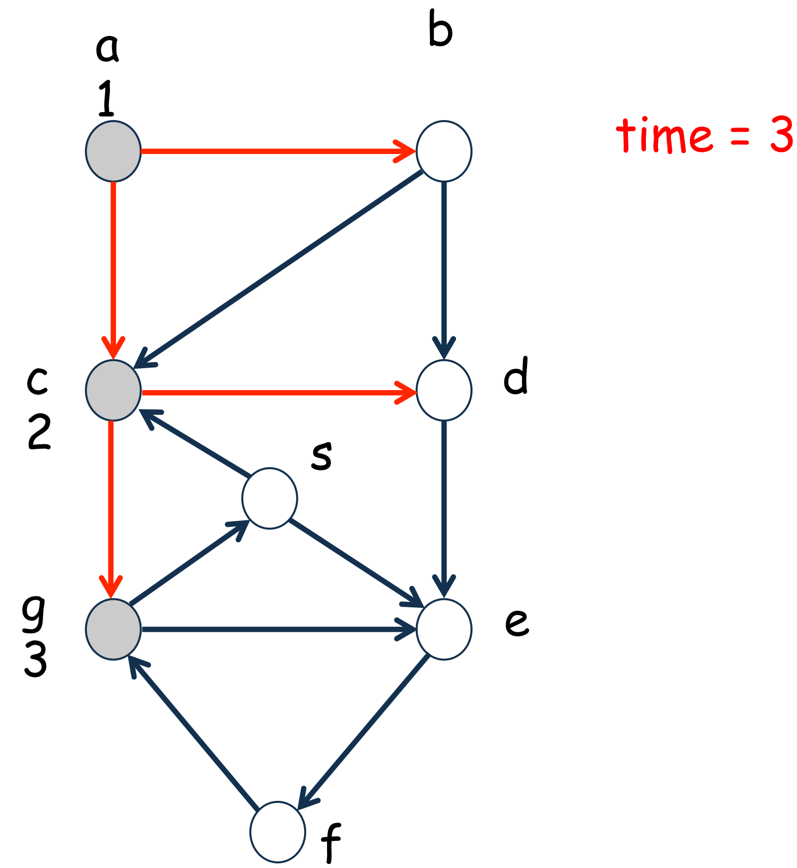
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

DFS_Visit(u)

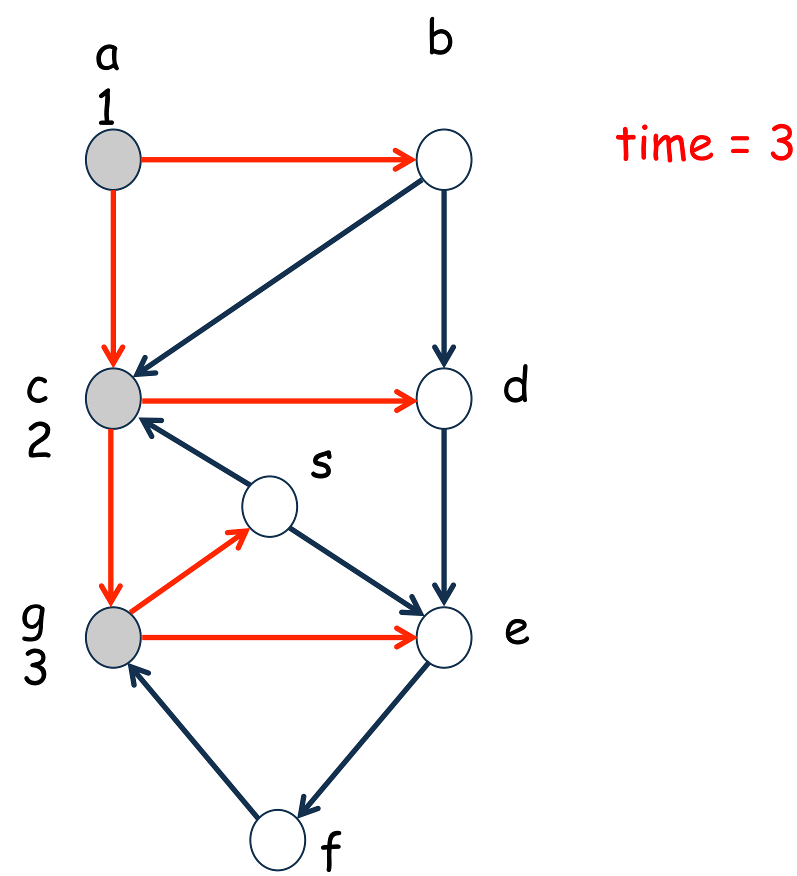
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



Depth First Search

<u>DFS(G)</u>
```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



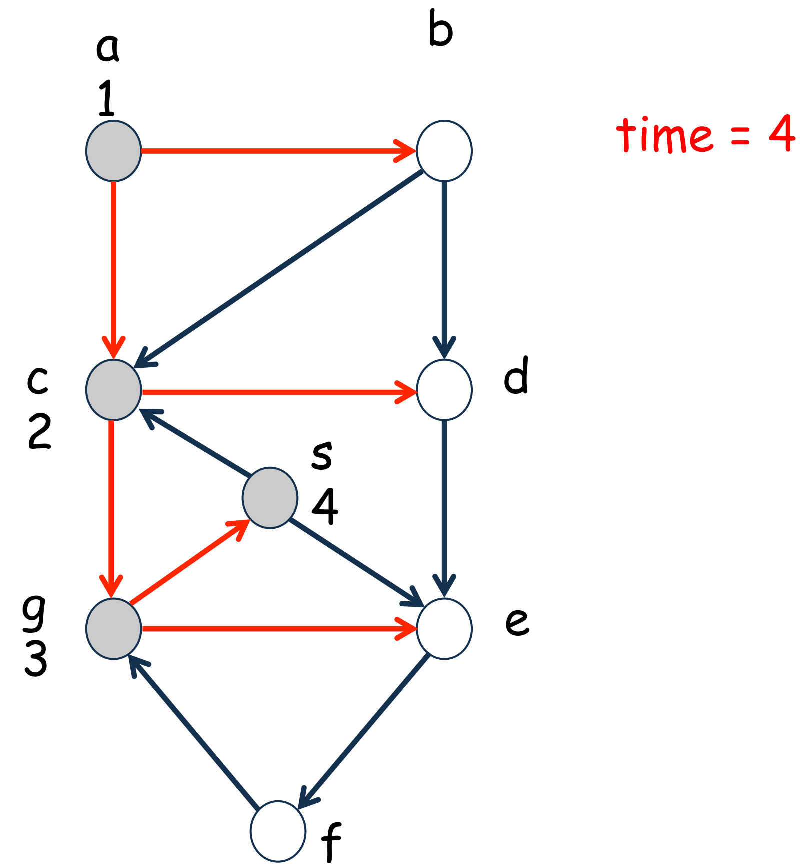
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
                DFS_Visit(u)
```

DFS_Visit(u)

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



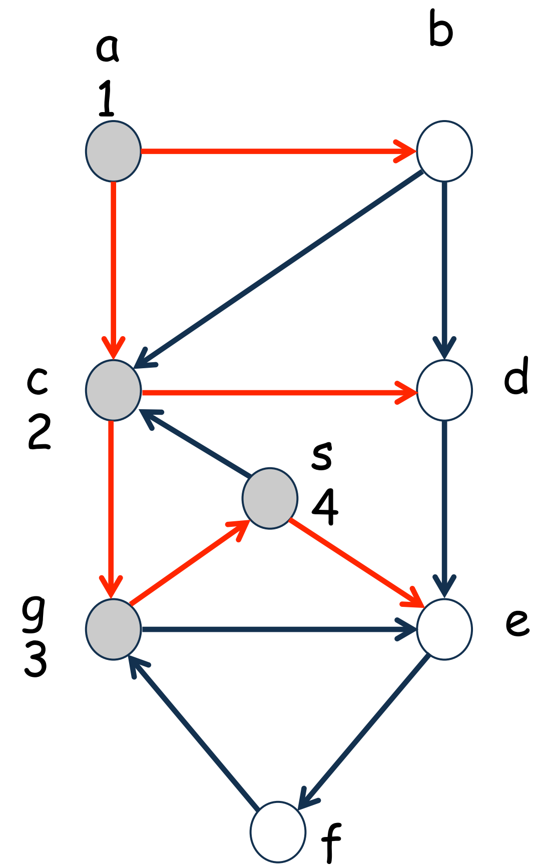
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

DFS_Visit(u)

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



time = 4

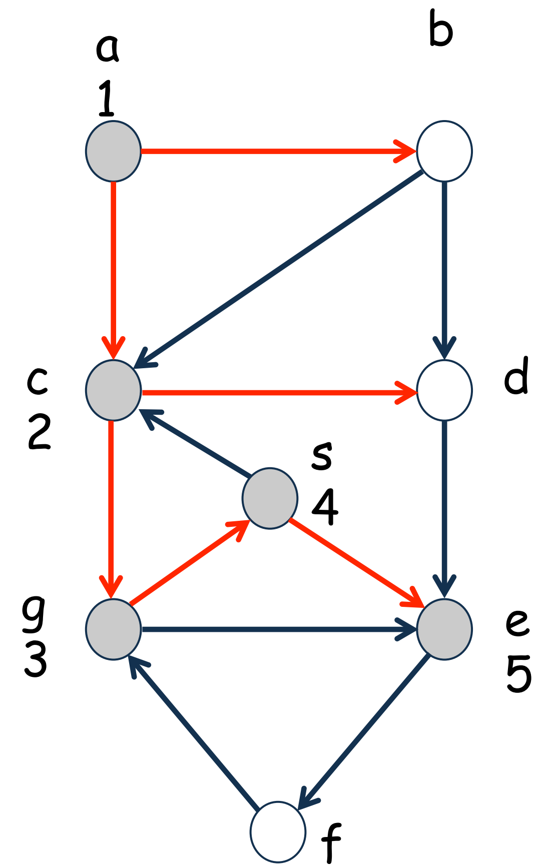
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
            DFS_Visit(u)
```

DFS_Visit(u)

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```

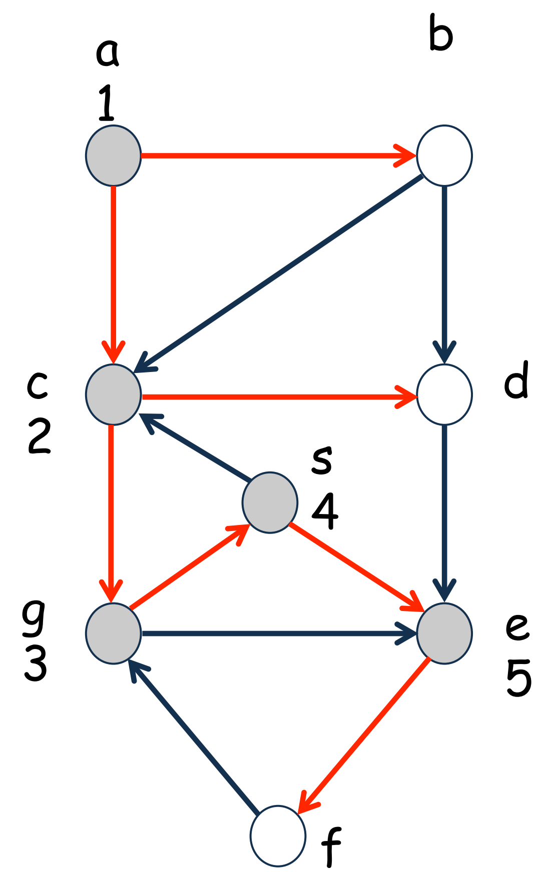


time = 5

Depth First Search

<u>DFS(G)</u>
```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```

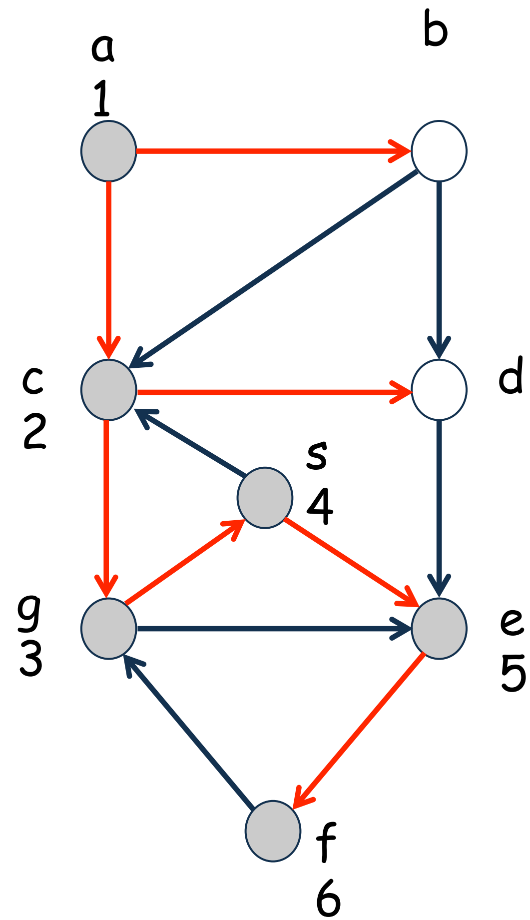


time = 5

Depth First Search

DFS(G)
for each vertex u of V
    u.color = white
    u.par = nil
time = 0
for each vertex u of V
    if u.color = white
        DFS_Visit(u)

DFS_Visit(u)
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
    if (v.color = white)
        v.par = u
        DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time



time = 6

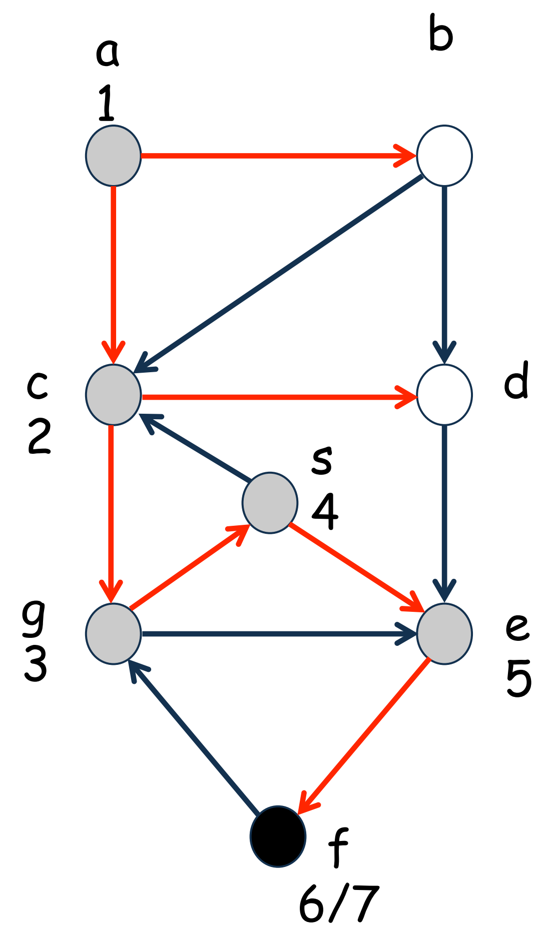
Depth First Search

<u>DFS(G)</u>

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```

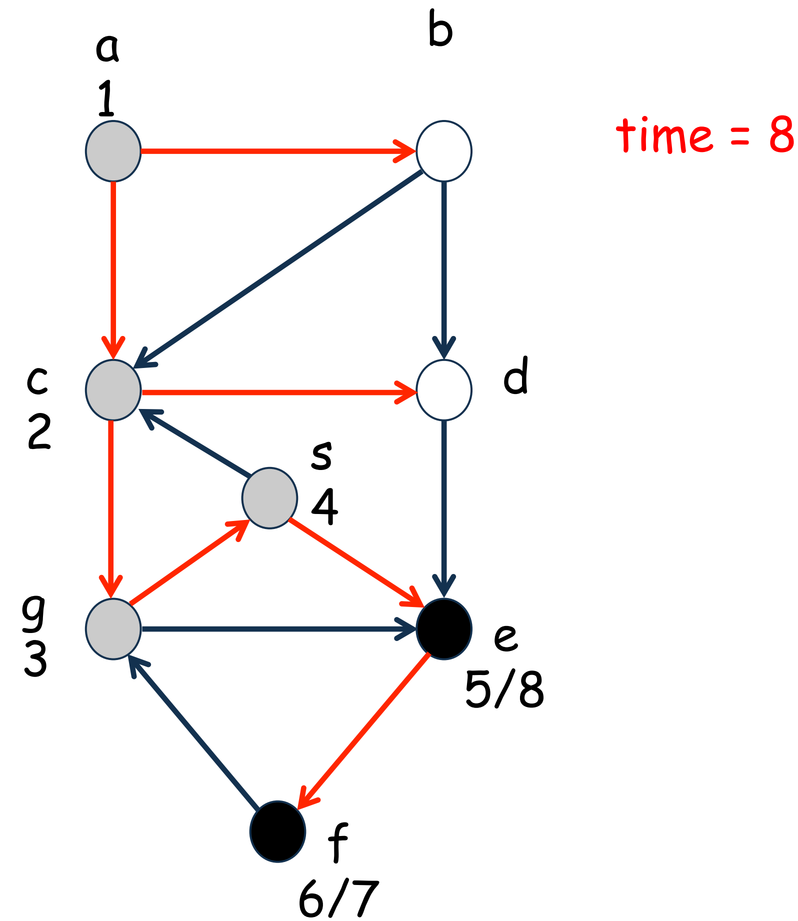


time = 7

Depth First Search

DFS(G)
for each vertex u of V
    u.color = white
    u.par = nil
time = 0
for each vertex u of V
    if u.color = white
        DFS_Visit(u)

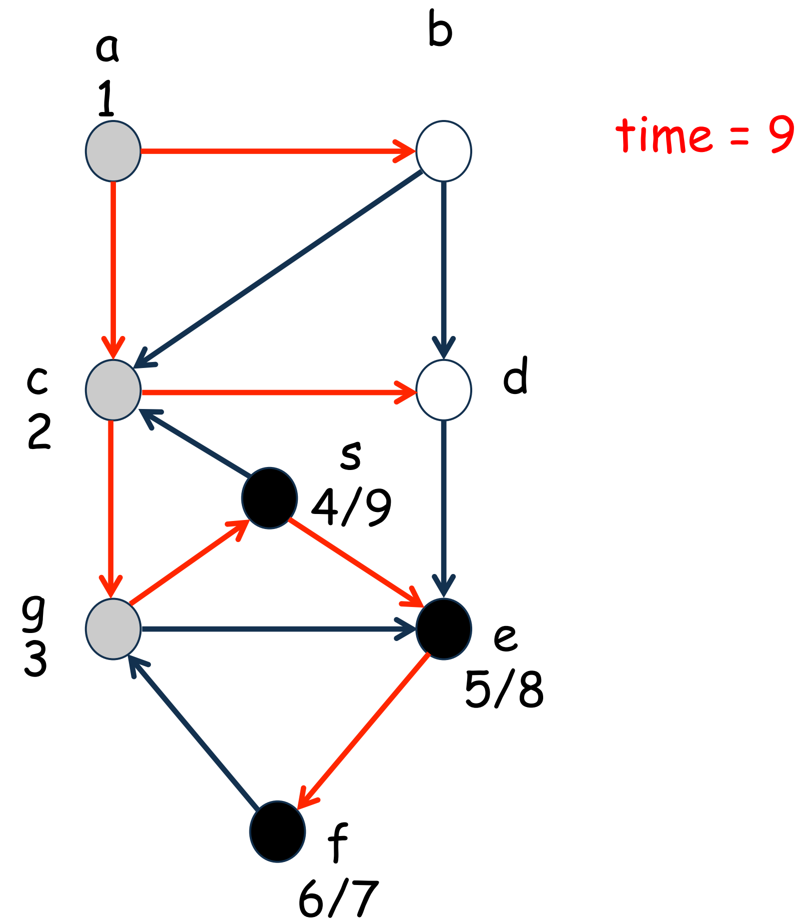
DFS_Visit(u)
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
    if (v.color = white)
        v.par = u
        DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time



Depth First Search

DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
            DFS_Visit(u)

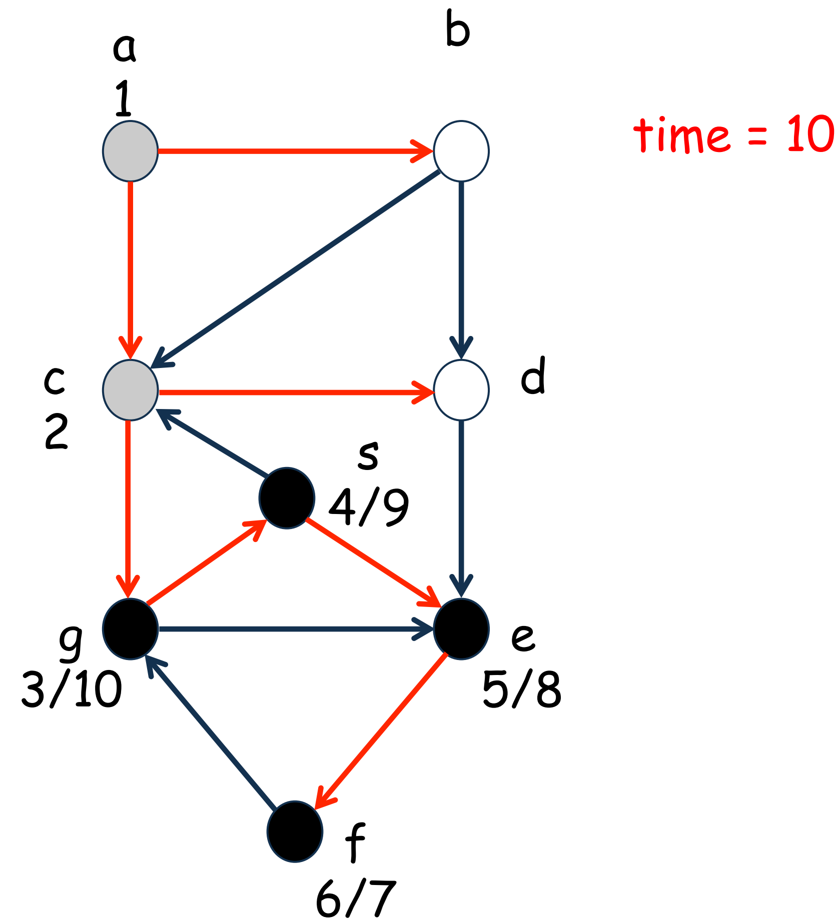
DFS_Visit(u)
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time



Depth First Search

DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
                DFS_Visit(u)

DFS_Visit(u)
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
        if (v.color = white)
                v.par = u
                DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time



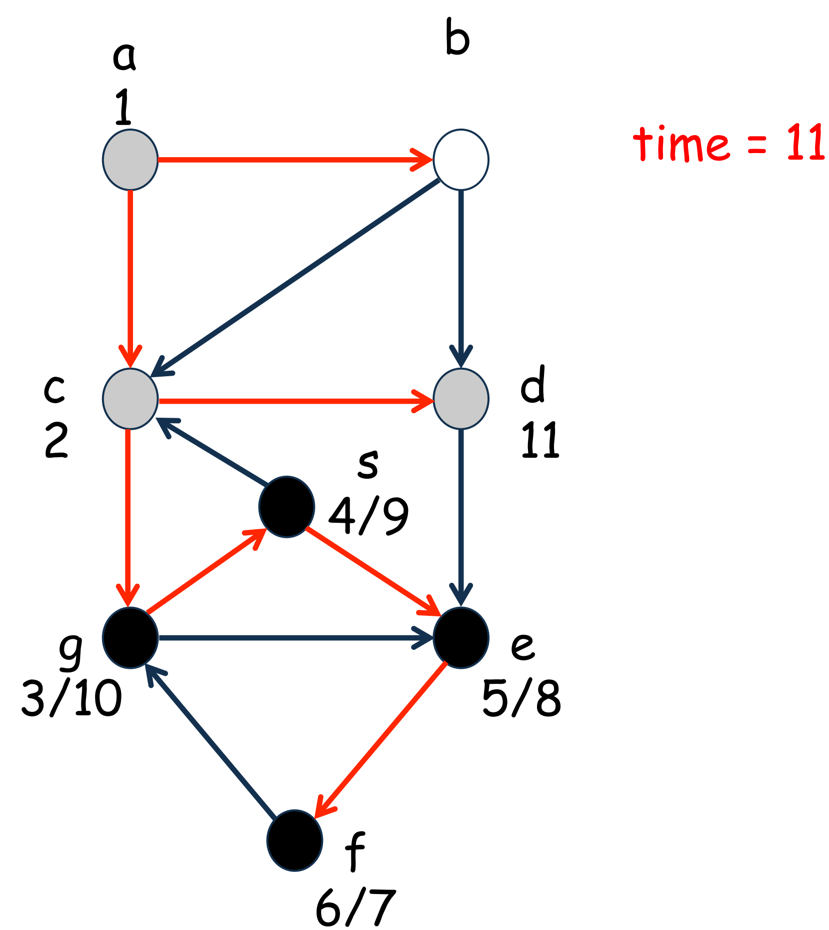
Depth First Search

<u>DFS(G)</u>

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>

```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



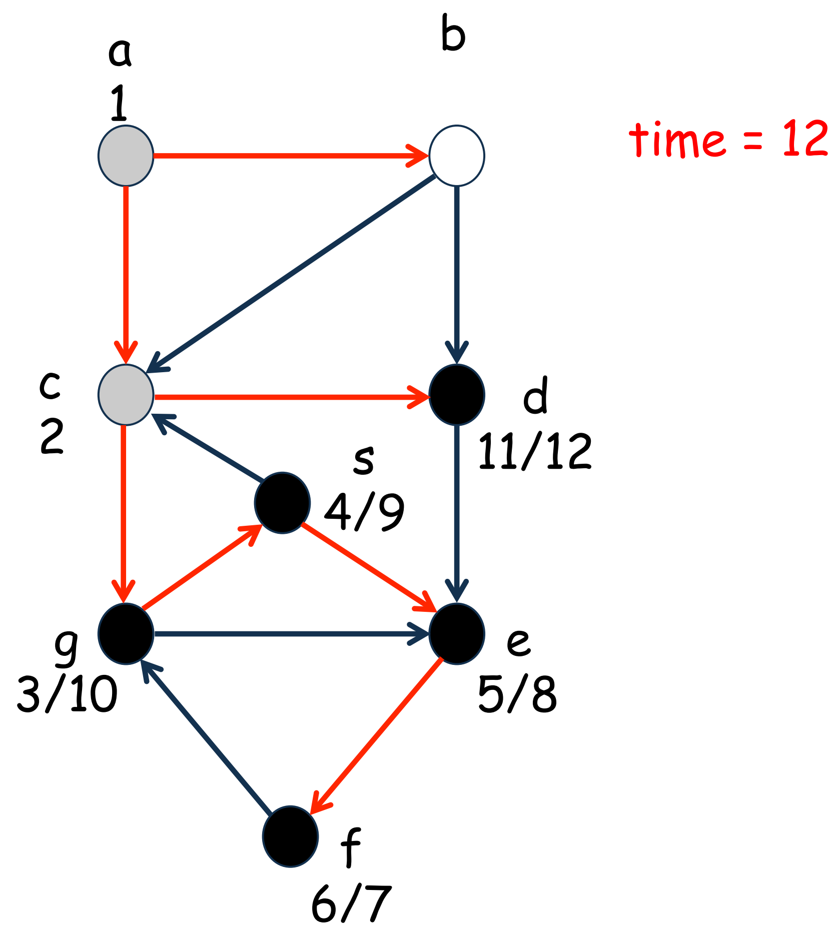
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
            DFS_Visit(u)
```

DFS_Visit(u)

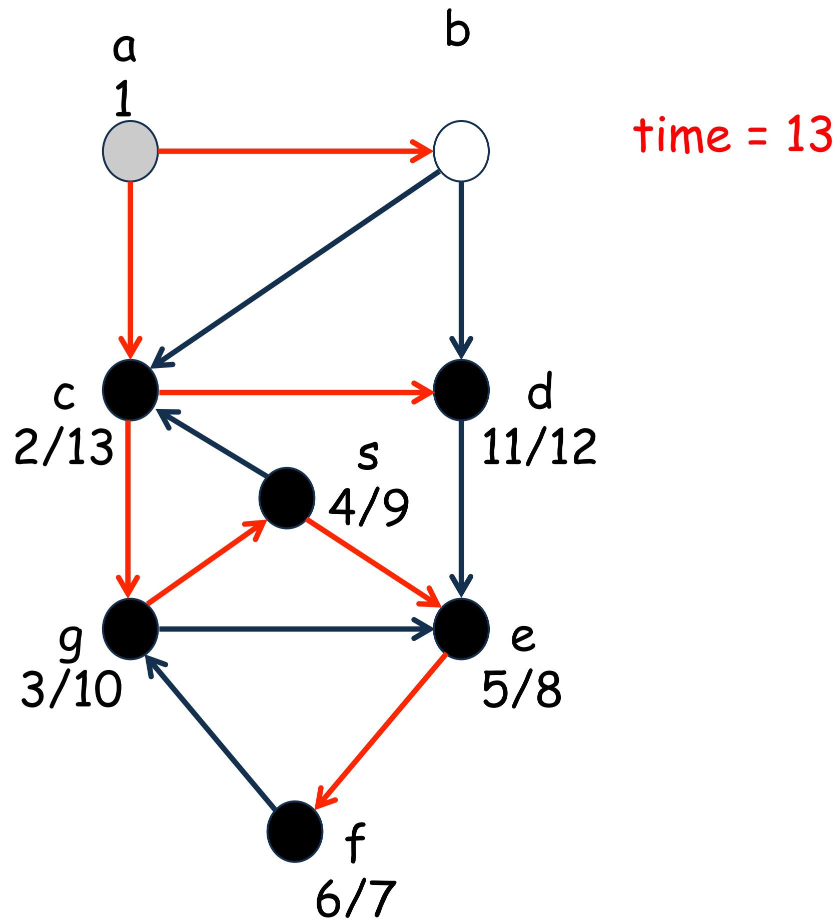
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



Depth First Search

DFS(G)
```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
            DFS_Visit(u)
```

DFS_Visit(u)
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



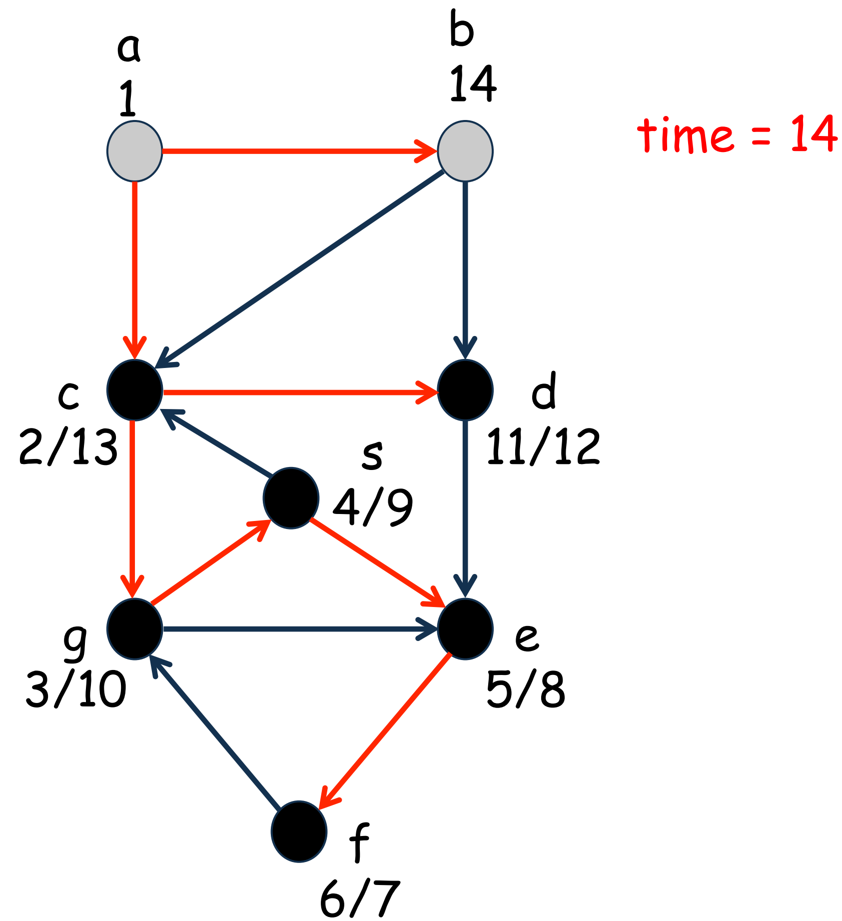
Depth First Search

DFS(G)

```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

DFS_Visit(u)

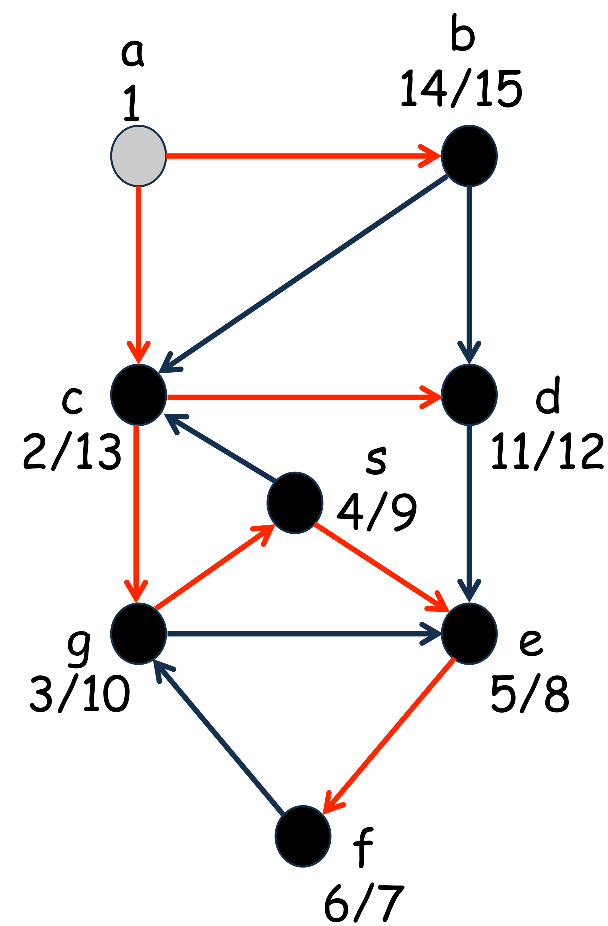
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
      if (v.color = white)
            v.par = u
            DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



Depth First Search

DFS(G)
```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
                DFS_Visit(u)
```

DFS_Visit(u)
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
        if (v.color = white)
                v.par = u
                DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```

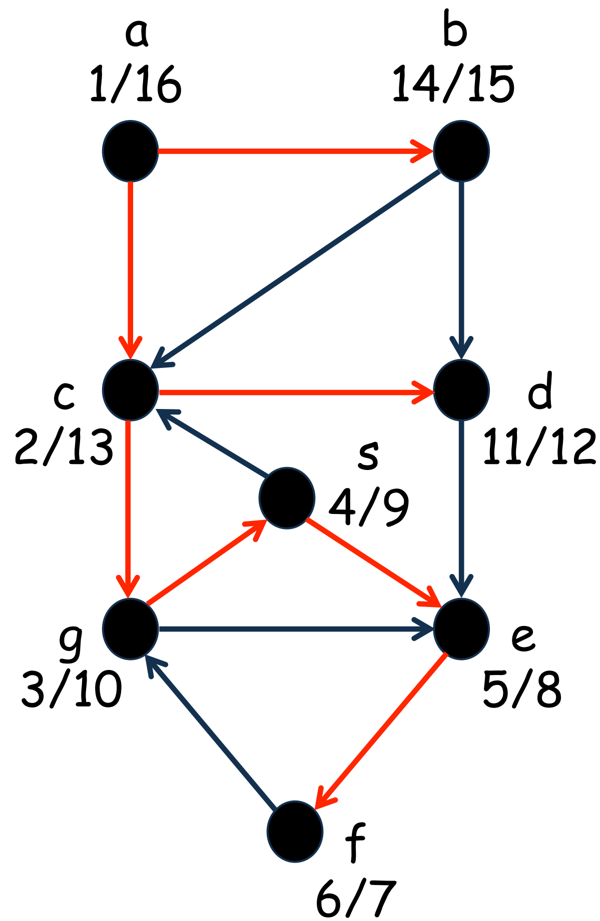


time = 15

Depth First Search

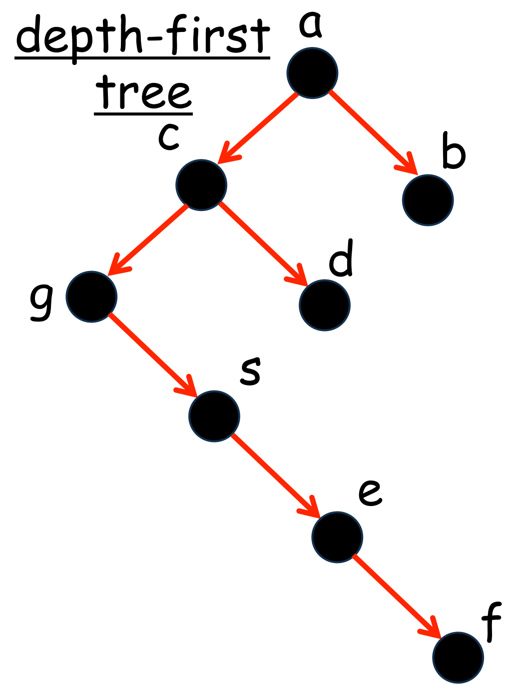
DFS(G)
```
for each vertex u of V
        u.color = white
        u.par = nil
time = 0
for each vertex u of V
        if u.color = white
            DFS_Visit(u)
```

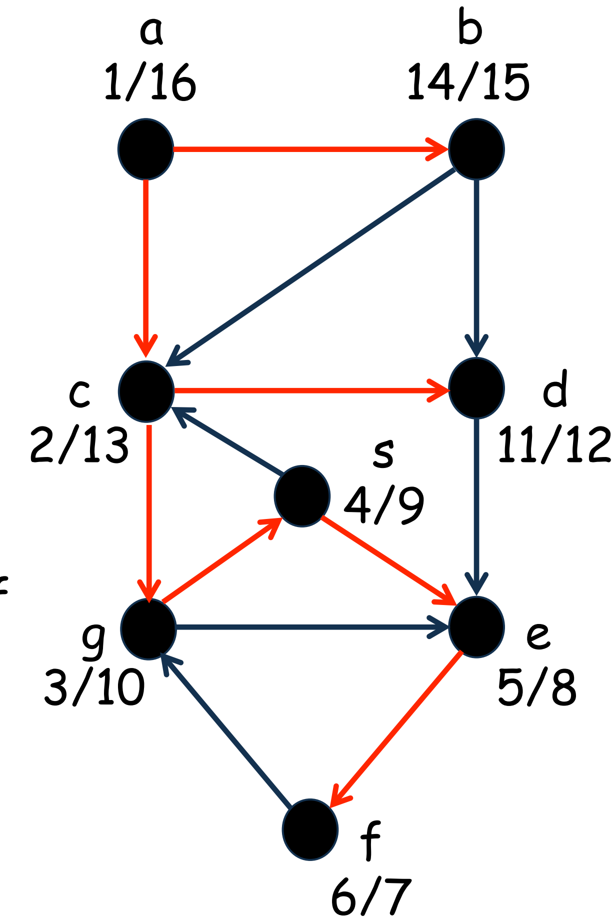
DFS_Visit(u)
```
u.color = gray
time = time + 1
u.dis = time
for each v in Adj(u)
     if (v.color = white)
          v.par = u
          DFS_Visit(v)
u.color = black
time = time + 1
u.fin = time
```



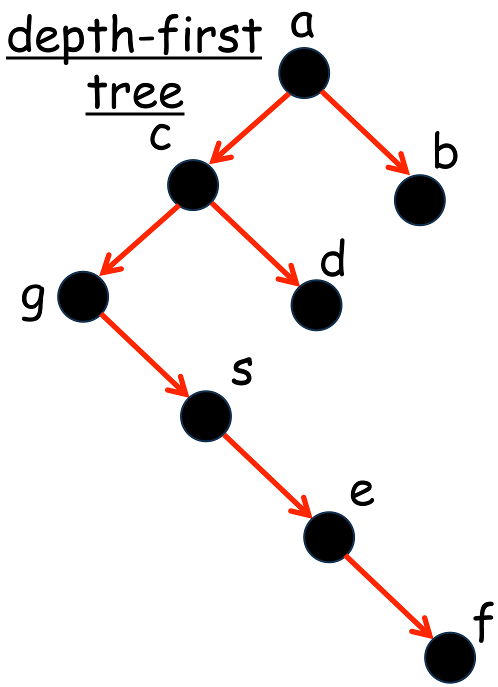
time = 15

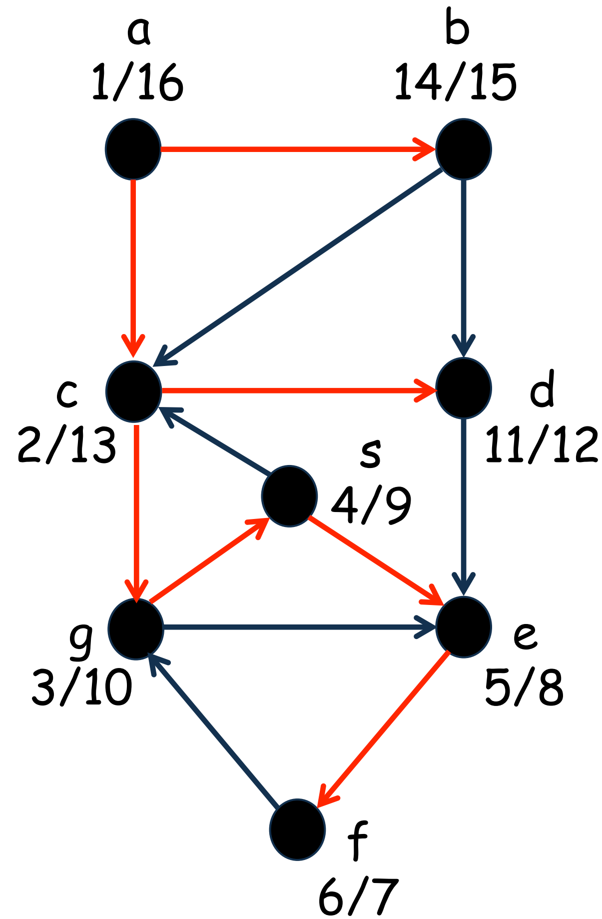
Depth First Search





Depth First Search





tree-edge

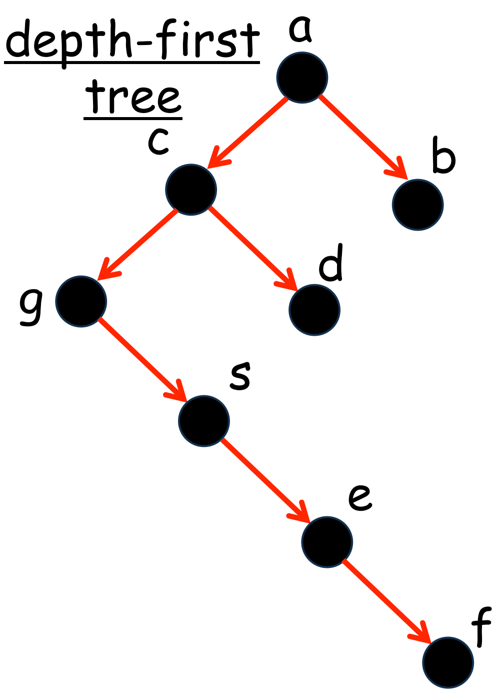
(u,v) is called tree-edge if it was first discovered by exploring edge

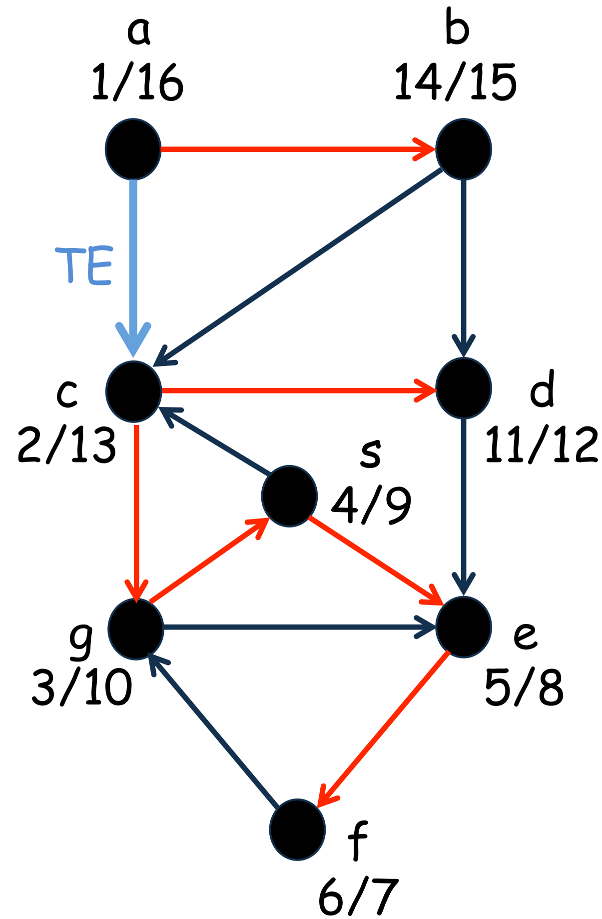
Depth First Search

tree-edge

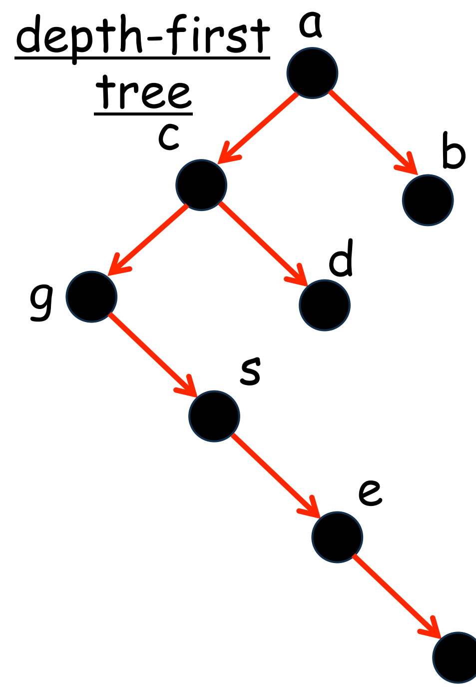
(u,v) is called tree-edge if it was first discovered by exploring edge

depth-first tree



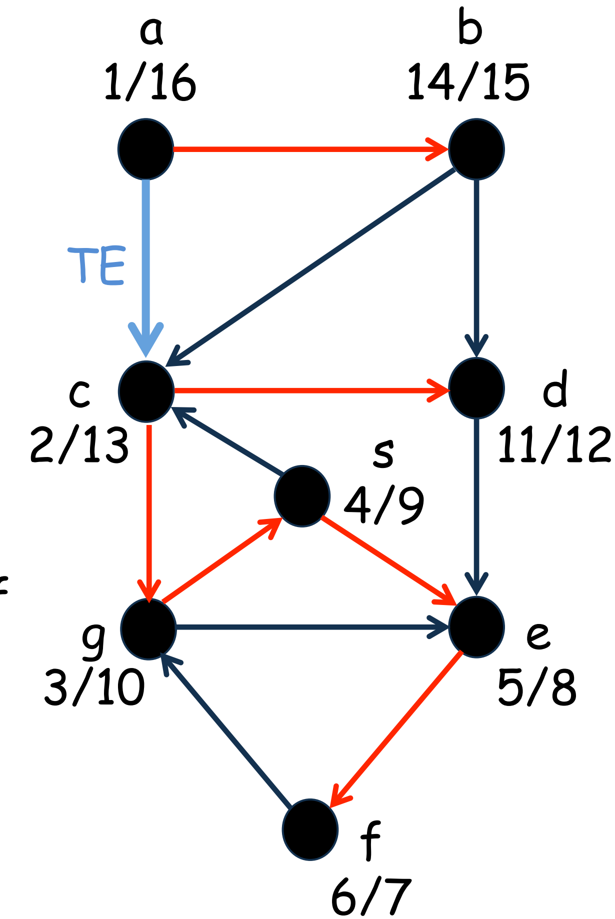


Depth First Search

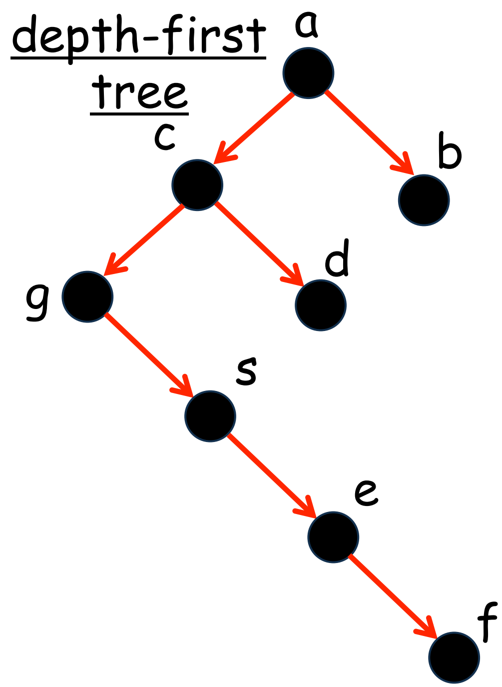


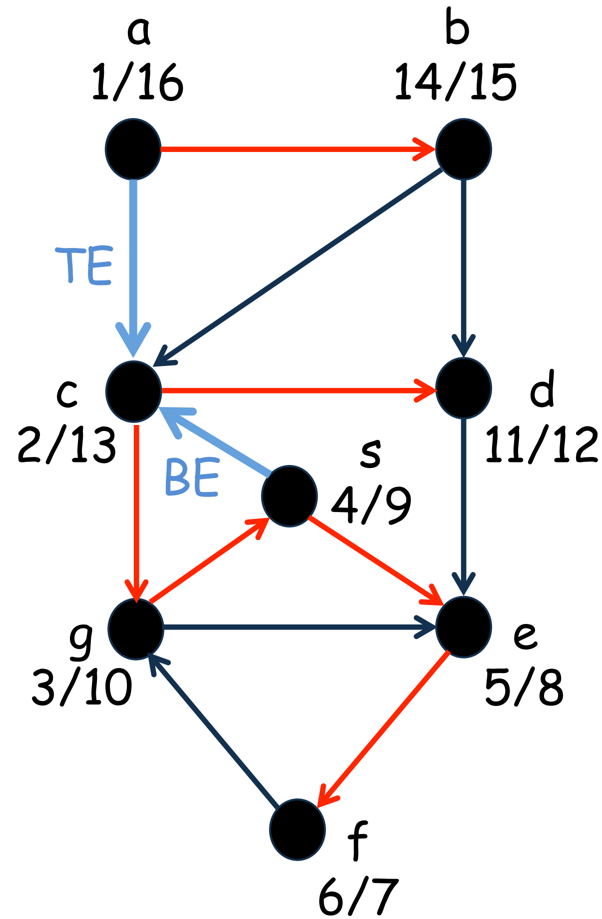
tree-edge
(u,v) is called tree-edge if it was first discovered by exploring edge

back-edge
(u,v) is called back-edge if it's connecting vertex u to an ancestor v in depth-first tree



Depth First Search

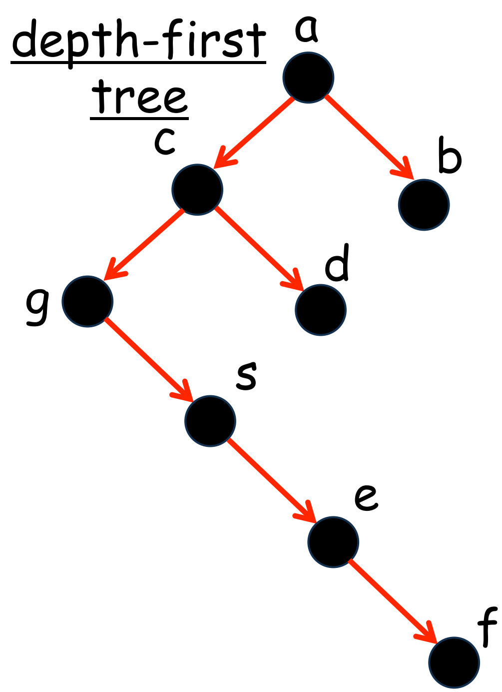


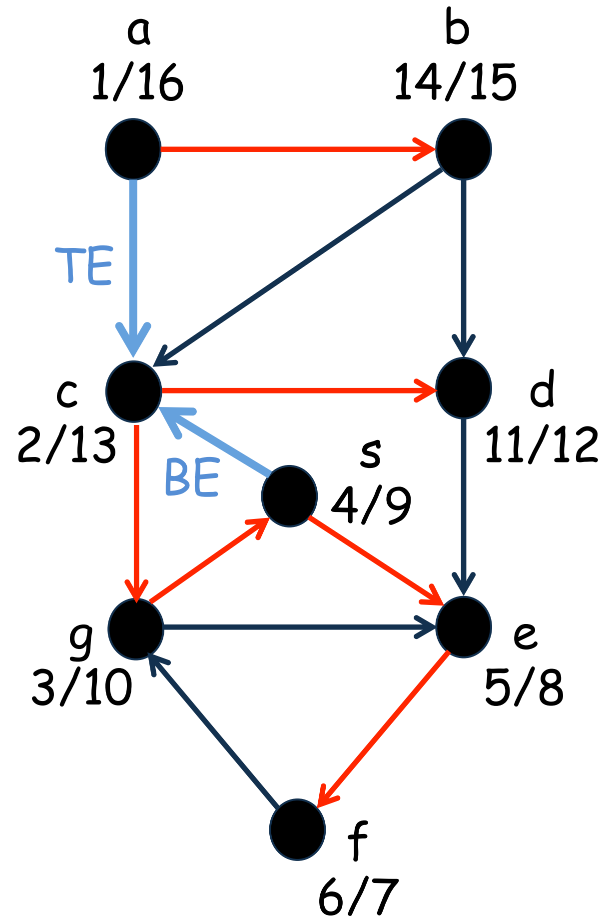


tree-edge
(u,v) is called tree-edge if it was first discovered by exploring edge

back-edge
(u,v) is called back-edge if it's connecting vertex u to an ancestor v in depth-first tree

Depth First Search



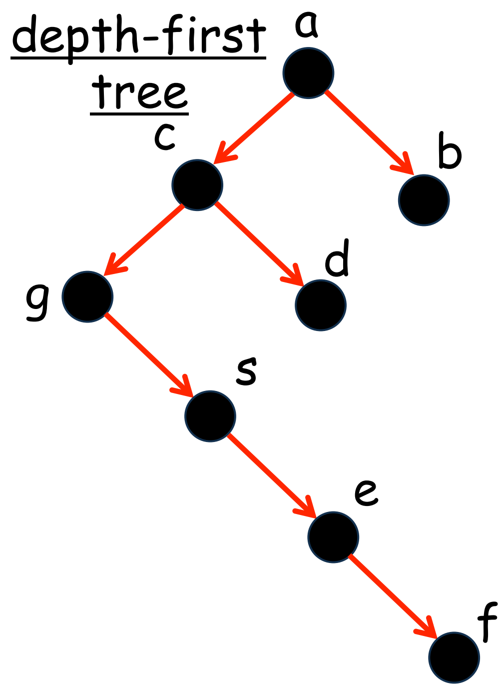


tree-edge
(u,v) is called tree-edge if it was first discovered by exploring edge

back-edge
(u,v) is called back-edge if it's connecting vertex u to an ancestor v in depth-first tree

forward-edge
(u,v) is called forward-edge if it's a nontree-edge connecting vertex u to a decendant v in depth-first tree

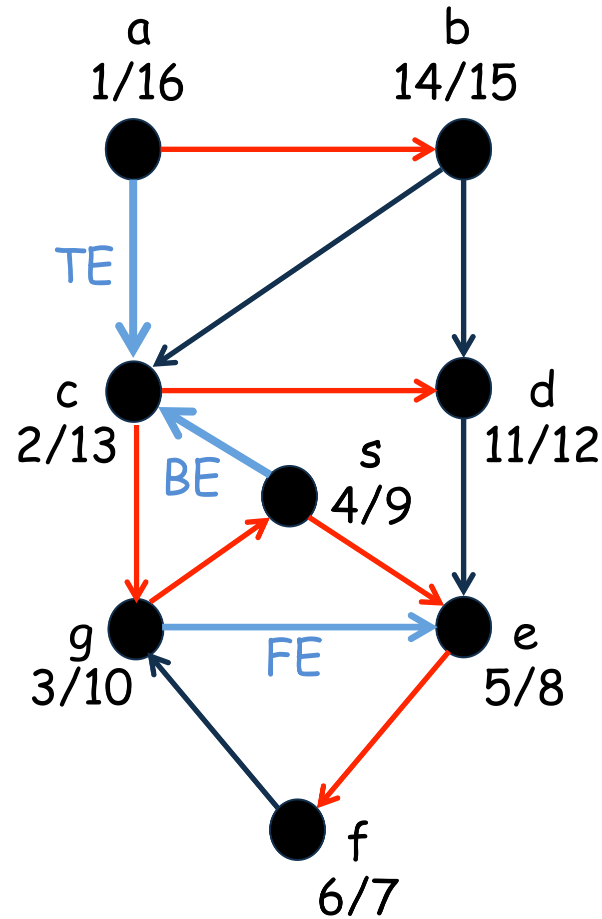
Depth First Search



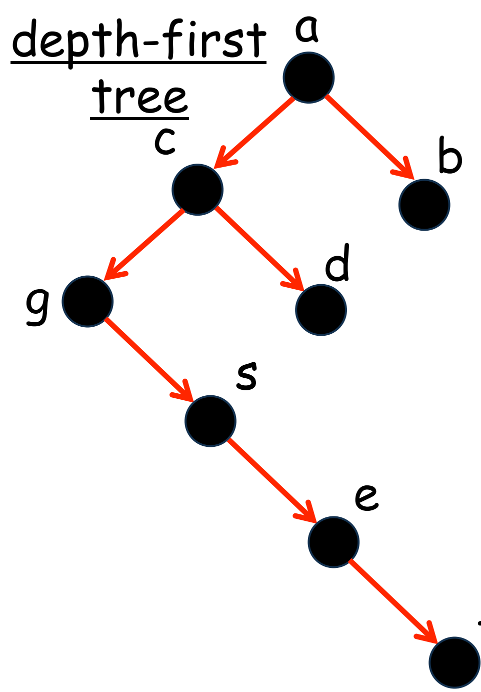
tree-edge
(u,v) is called tree-edge if it was first discovered by exploring edge

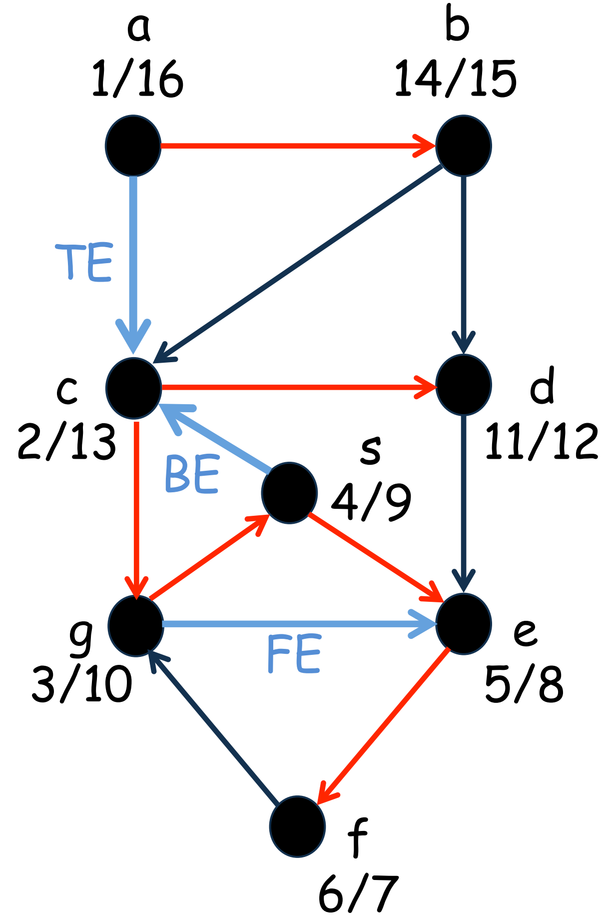
back-edge
(u,v) is called back-edge if it's connecting vertex u to an ancestor v in depth-first tree

forward-edge
(u,v) is called forward-edge if it's a nontree-edge connecting vertex u to a decendant v in depth-first tree



Depth First Search





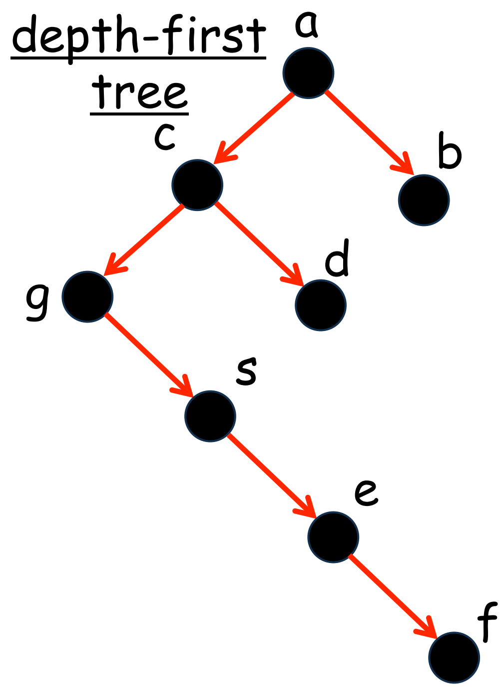
<u>tree-edge</u>
(u,v) is called tree-edge if it was first discovered by exploring edge

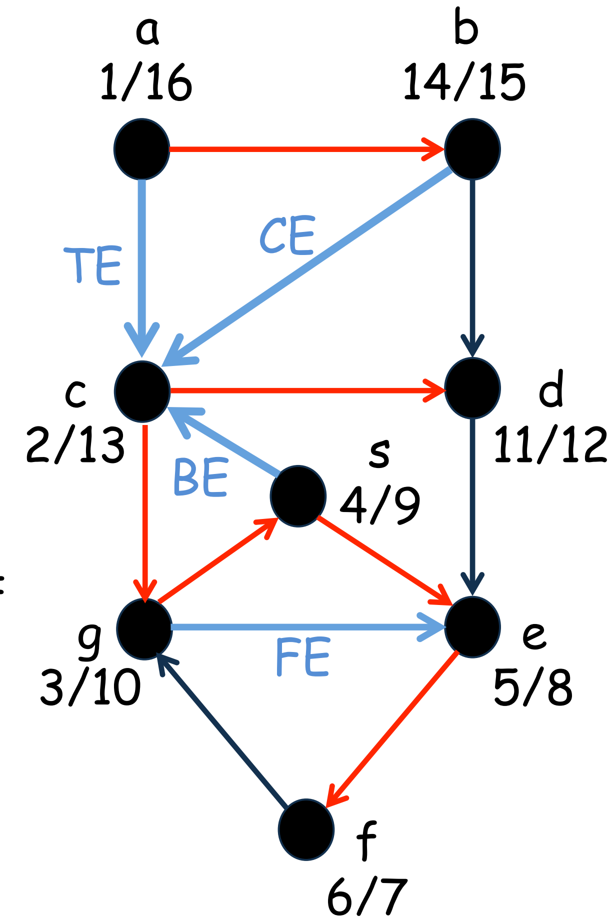
<u>back-edge</u>
(u,v) is called back-edge if it's connecting vertex u to an ancestor v in depth-first tree

<u>forward-edge</u>
(u,v) is called forward-edge if it's a nontree-edge connecting vertex u to a decendant v in depth-first tree

<u>cross-edge</u>
(u,v) is called cross-edge if it's connecting vertex u to vertex v such that there is no ancestor/ descendant relation between them

Depth First Search





tree-edge

(u,v) is called tree-edge if it was first discovered by exploring edge

back-edge

(u,v) is called back-edge if it's connecting vertex u to an ancestor v in depth-first tree

forward-edge

(u,v) is called forward-edge if it's a nontree-edge connecting vertex u to a decendant v in depth-first tree

cross-edge

(u,v) is called cross-edge if it's connecting vertex u to vertex v such that there is no ancestor/ descendant relation between them

Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph

Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph

```
DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
       if u.color = white
            DFS_Visit(u)

DFS_Visit(u)
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
            output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```

Cycle Detection

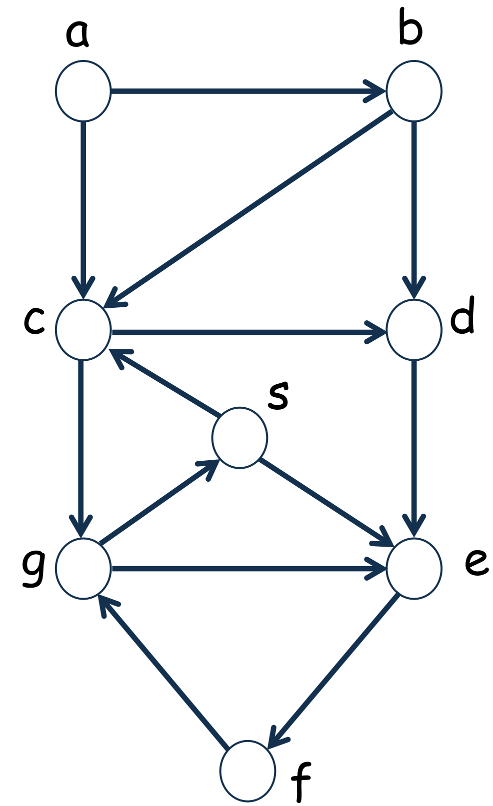
- There is a cycle in the graph only if there is back edge in the graph

<u>DFS(G)</u>

```
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
       if u.color = white
              DFS_Visit(u)
```

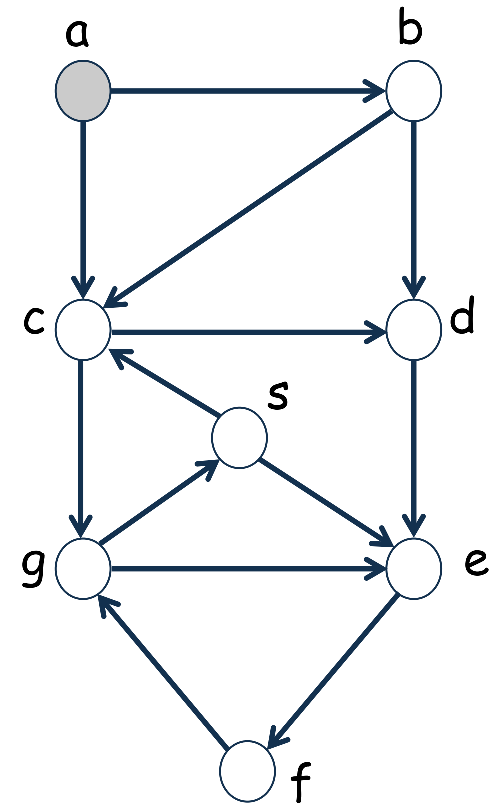
<u>DFS_Visit(u)</u>

```
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
             output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```



Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph

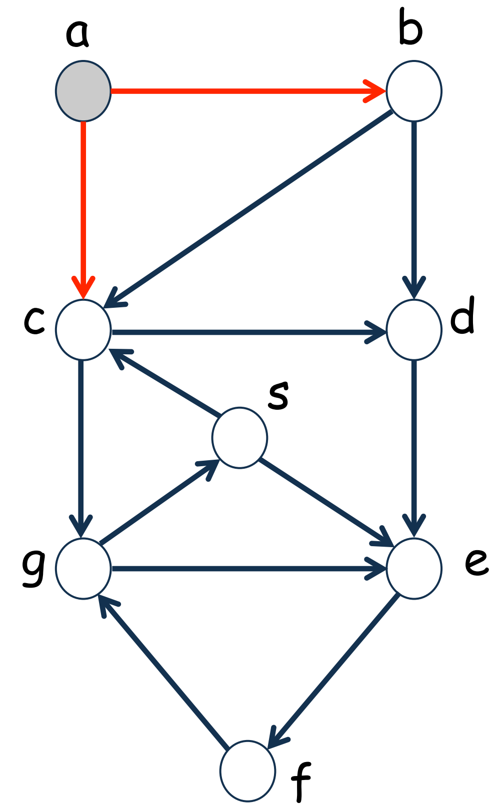



```
DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
       if u.color = white
             DFS_Visit(u)


DFS_Visit(u)
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
            output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```

Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph

```
DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
       if u.color = white
             DFS_Visit(u)


DFS_Visit(u)
u.color = gray
for each v in Adj(u)
     if (v.color = gray)
            output 'cycle found'
     else
           v.par = u
           DFS_Visit(v)
u.color = black
```

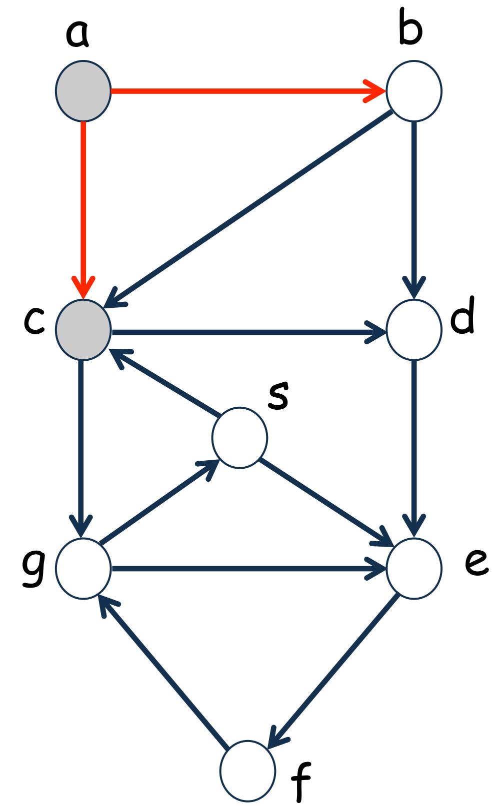


Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph

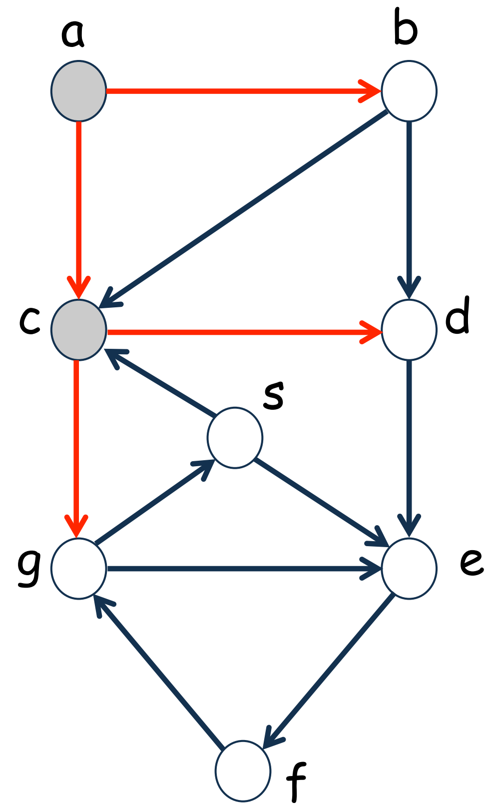
<u>DFS(G)</u>
```
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

<u>DFS_Visit(u)</u>
```
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
            output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```



Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph




```
DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
        if u.color = white
              DFS_Visit(u)
```

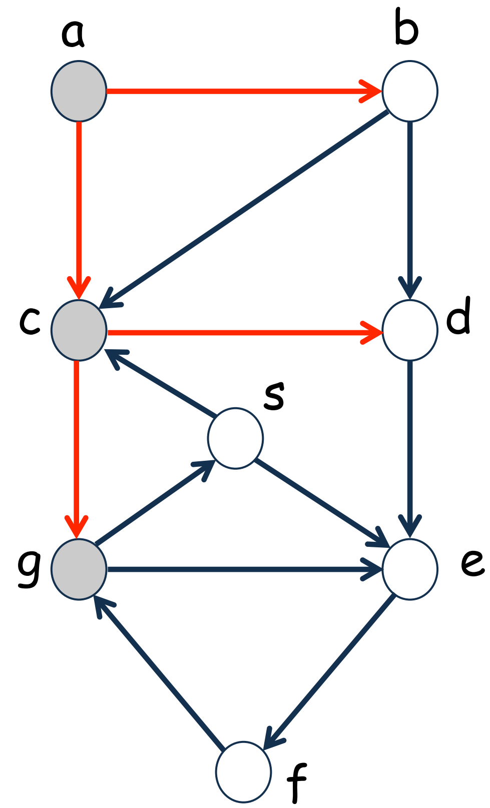
```
DFS_Visit(u)
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
             output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```

Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph

<u>DFS(G)</u>
```
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
        if u.color = white
               DFS_Visit(u)
```

<u>DFS_Visit(u)</u>
```
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
             output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```



Cycle Detection

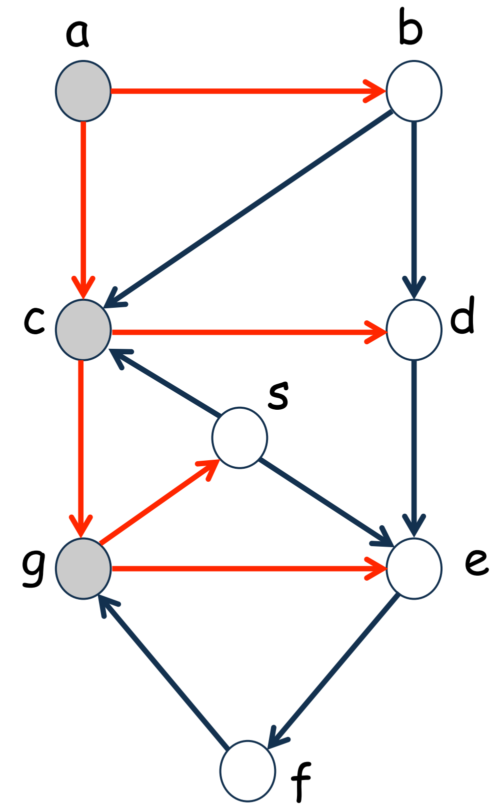
- There is a cycle in the graph only if there is back edge in the graph

<u>DFS(G)</u>

```
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
       if u.color = white
             DFS_Visit(u)
```

<u>DFS_Visit(u)</u>

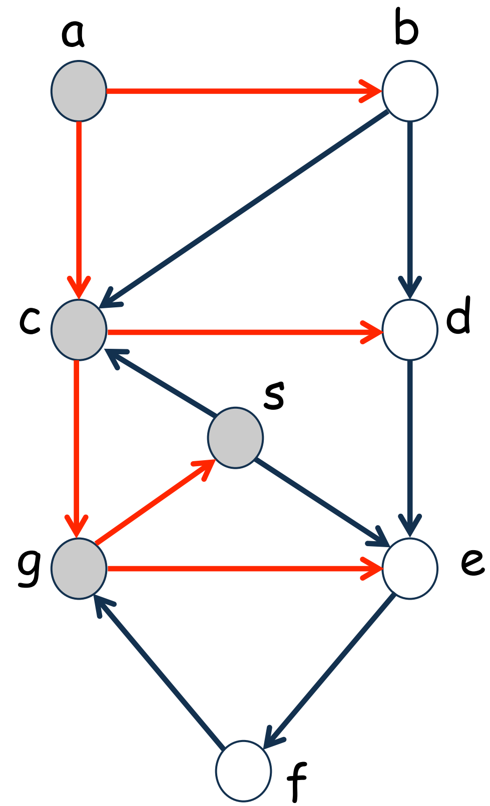
```
u.color = gray
for each v in Adj(u)
     if (v.color = gray)
            output 'cycle found'
     else
           v.par = u
           DFS_Visit(v)
u.color = black
```



Cycle Detection

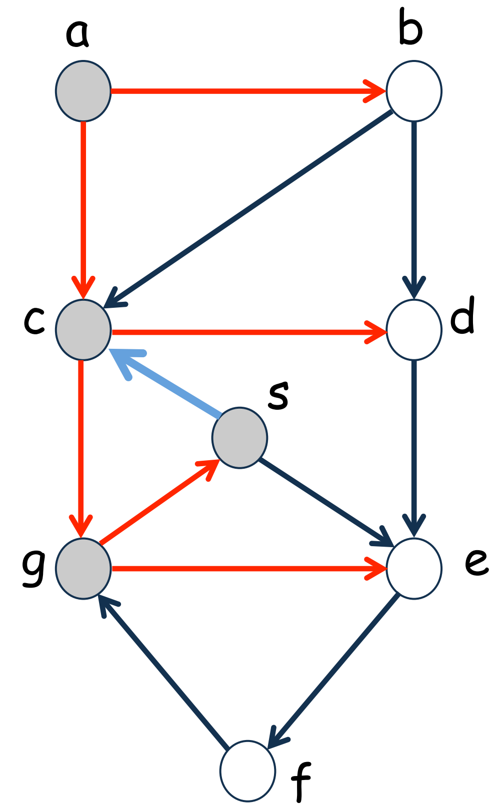
- There is a cycle in the graph only if there is back edge in the graph

```
DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
       if u.color = white
             DFS_Visit(u)

DFS_Visit(u)
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
             output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```



Cycle Detection

- There is a cycle in the graph only if there is back edge in the graph




```
DFS(G)
for each vertex u of V
        u.color = white
        u.par = nil
for each vertex u of V
        if u.color = white
              DFS_Visit(u)

DFS_Visit(u)
u.color = gray
for each v in Adj(u)
      if (v.color = gray)
            output 'cycle found'
      else
            v.par = u
            DFS_Visit(v)
u.color = black
```