数据结构与算法设计

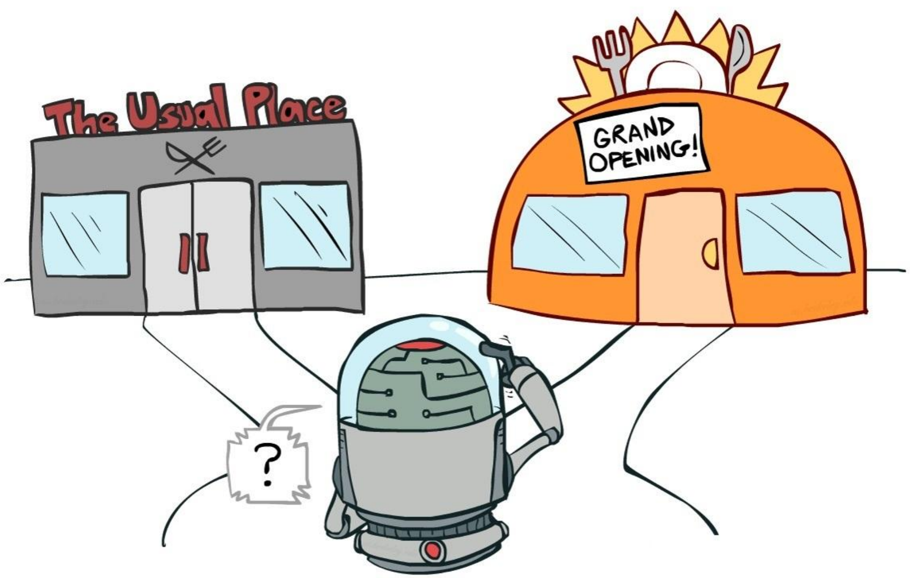
周　可

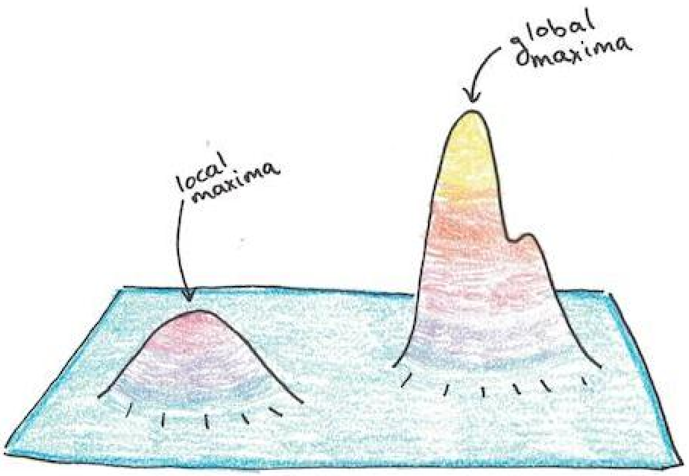
Mail：zhke@hust.edu.cn

华中科技大学，武汉光电国家研究中心

课程引入：

Exploration-Exploitation dilemma





Explore or Exploit: The Hidden Decision that Guides Your Life ——Scott H. Young

1. **Graph Representation**
2. Graph Searching (BFS, DFS)
3. Dijkstra Algorithm
4. MST

Background

In many applications, a tree (the most complex data structure we have learnt so far) is not enough.

(1) Think about the network where the vertices are all the Internet routers in the world.

(2) Sometimes we might need a structure more complex than a single graph. But for most of the applications, a graph should be enough.

Definition

An (undirected) graph is a pair <V,E>.

(1) V is the set of vertices, $|V|=n$.

(2) E is the set of edges, $|E|=m$.

(3) So the size of a graph is $|V|+|E|$.

Definition

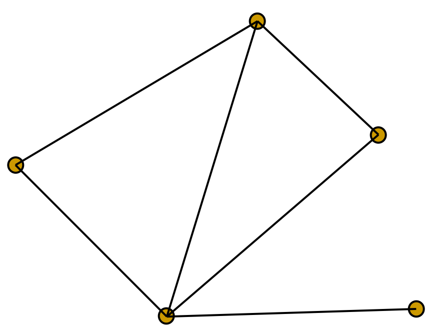
An (undirected) graph is a pair <V,E>.

(1) V is the set of vertices, |V|=n.

(2) E is the set of edges, |E|=m.

(3) So the size of a graph is |V|+|E|.

In the graph on our right, n=5, m=6.



Graph Basics,1

In a graph G of n vertices, how many edges could we have?

Graph Basics,1

In a graph G of n vertices, how many edges could we have?

Minimum: 0

Maximum: n(n-1)/2, how could that happen?

Graph Basics,1

In a graph G of n vertices, how many edges could we have?

Minimum: 0

Maximum: n(n-1)/2, how could that happen?

When G is a complete graph!

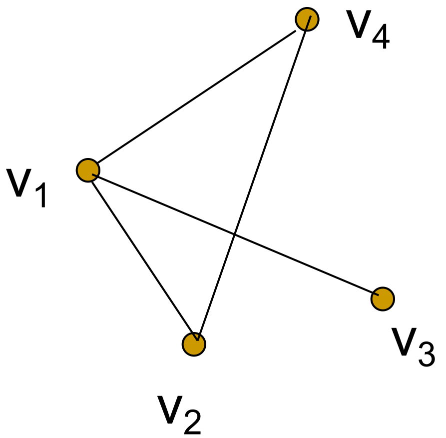
Graph Basics,2

The degree of a vertex v in G, deg(v), is the number of edges adjacent to v.

$deg(v_1) = 3$, $deg(v_2)=2$

$deg(v_3)=1$, $deg(v_4)=2$

Can you see some pattern?

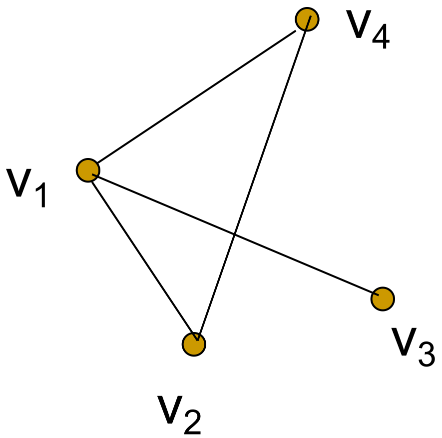


Graph Basics,2

The degree of a vertex v in G, deg(v), is the number of edges adjacent to v.

$deg(v_1)$ = 3, $deg(v_2)$=2

$deg(v_3)$=1, $deg(v_4)$=2



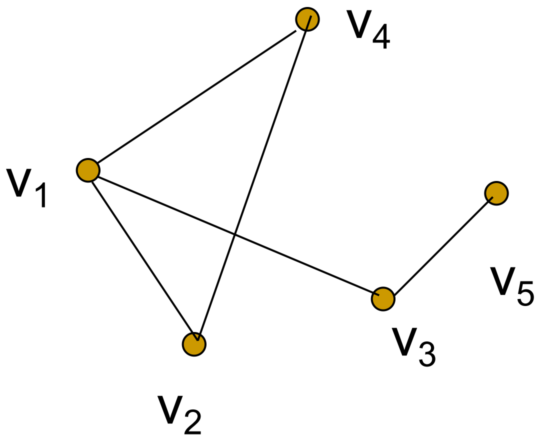
Can you see some pattern?

$\sum_v deg(v)$ is even. Why?

Graph Representation

A graph is a data structure, so we need to consider how to represent it.

Adjacency matrix: a_{ij} = 1 if v_i is adjacent to v_j

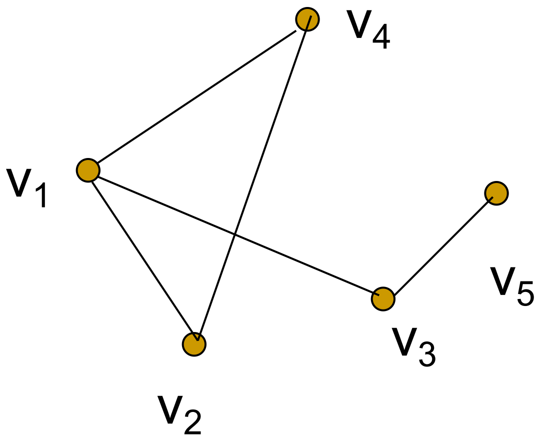


Graph Representation

A graph is a data structure, so we need to consider how to represent it.

Adjacency matrix: a_{ij} = 1 if vi is adjacent to vj otherwise it is 0.

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	0
v_2	1	0	0	1	0
v_3	1	0	0	0	1
v_4	1	1	0	0	0
v_5	0	0	1	0	0



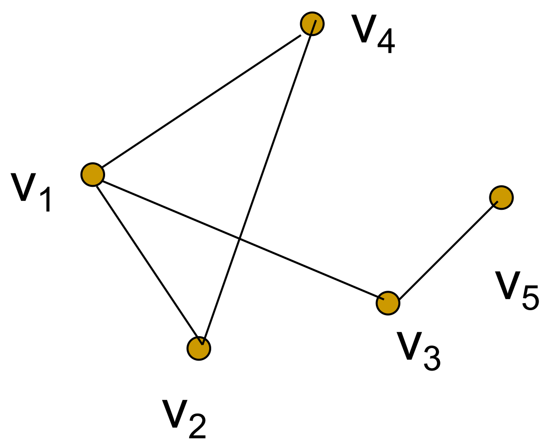
Graph Representation

A graph is a data structure, so we need to consider how to represent it.

Adjacency matrix: a_{ij} = 1 if vi is adjacent to vj

otherwise it is 0.

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	0
v_2	1	0	0	1	0
v_3	1	0	0	0	1
v_4	1	1	0	0	0
v_5	0	0	1	0	0



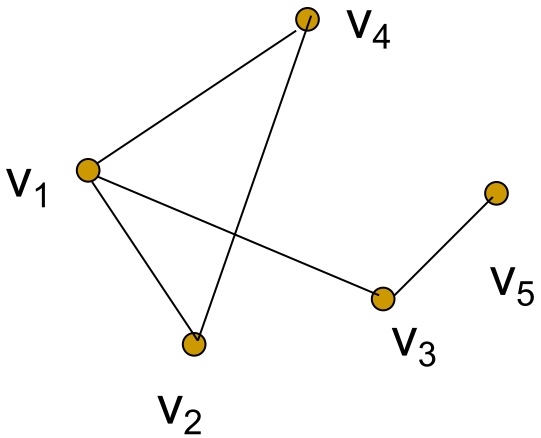
What property does this matrix have?

Graph Representation

A graph is a data structure, so we need to consider how to represent it.

Adjacency matrix: a_{ij} = 1 if vi is adjacent to vj otherwise it is 0.

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	0
v_2	1	0	0	1	0
v_3	1	0	0	0	1
v_4	1	1	0	0	0
v_5	0	0	1	0	0

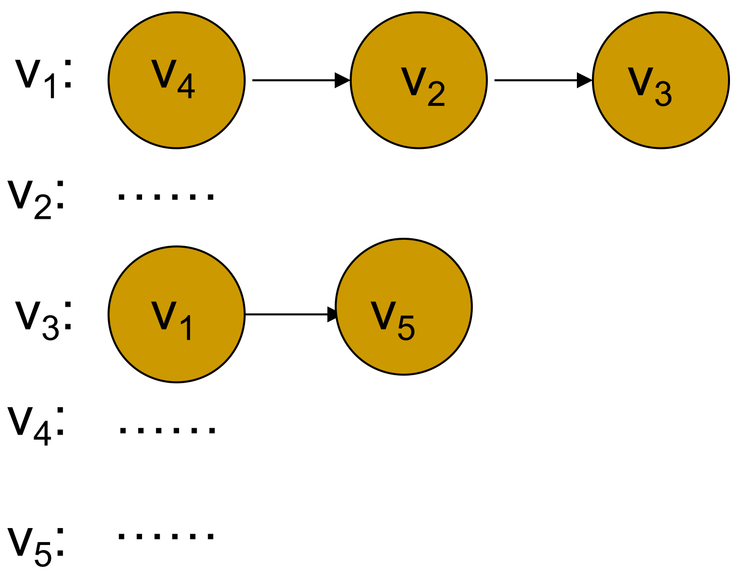


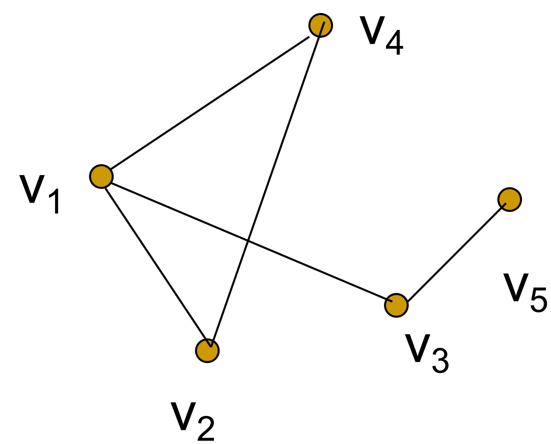
What is the cost of this representation?

Graph Representation

A graph is a data structure, so we need to consider how to represent it.

Adjacency list: for each v_i maintain the list of vertices adjacent to v_i.

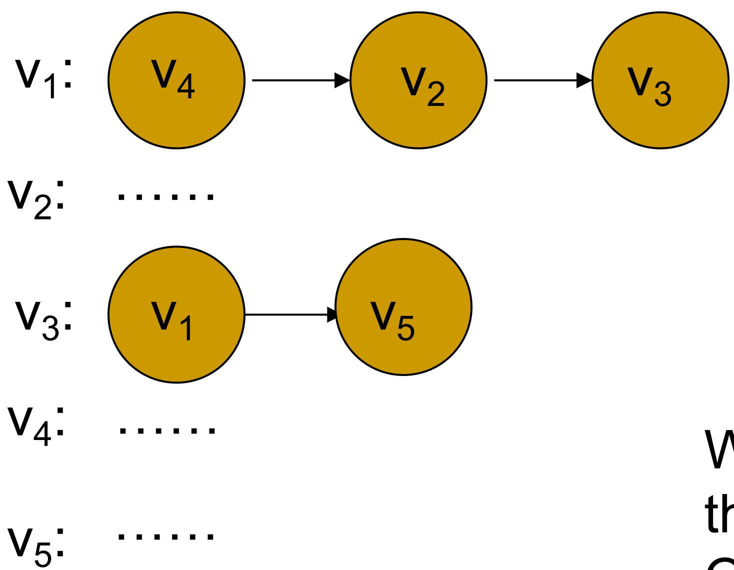


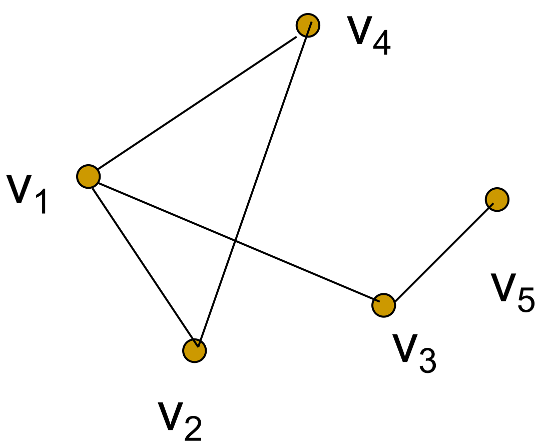


Graph Representation

A graph is a data structure, so we need to consider how to represent it.

Adjacency list: for each v_i maintain the list of vertices adjacent to v_i.



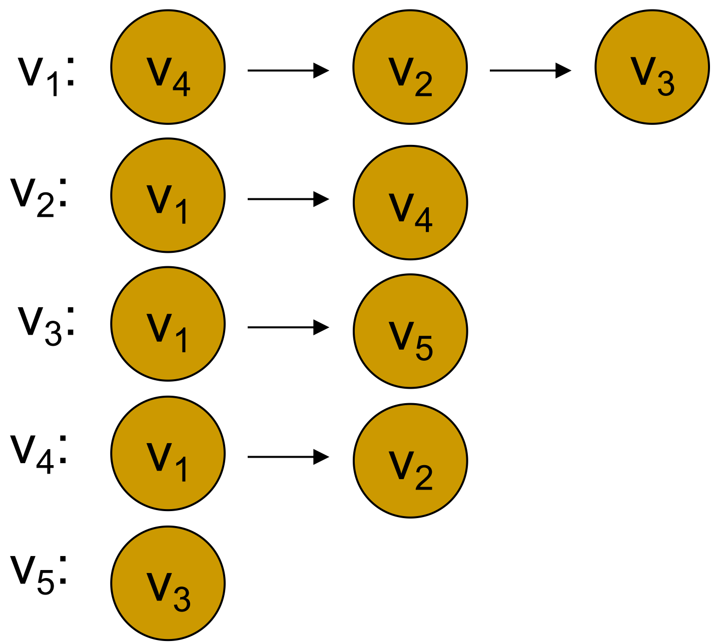


What is the cost of this representation?
$O(|V|+|E|)=O(n+m)$

Graph Representation

	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	0
v_2	1	0	0	1	0
v_3	1	0	0	0	1
v_4	1	1	0	0	0
v_5	0	0	1	0	0

Adjacency matrix



Adjacency list

两者比较：优缺点； 适用场合

1. Graph Representation

2. Graph Searching (BFS, DFS)

3. Dijkstra Algorithm

4. MST

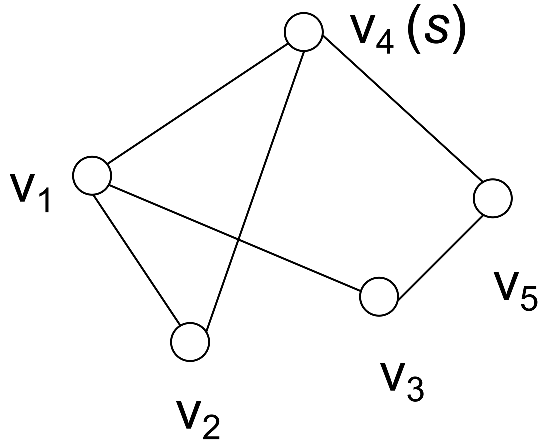
Graph Searching

Given a graph G of n vertices, we must know how to explore the graph.

This is similar to the preorder, inorder, and postorder traversals of a tree.

Graph Searching---BFS

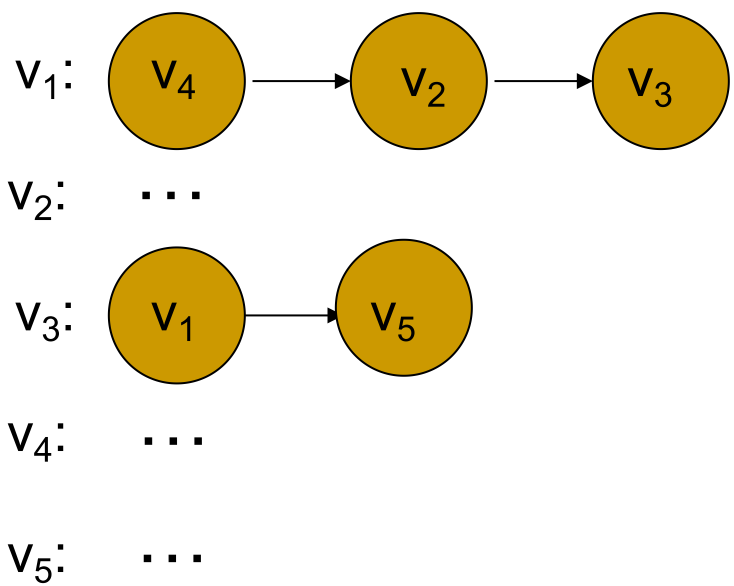
Given a graph G = <V,E> and a source vertex s, breadth-first search explores the edges of G to visit every vertex reachable from s.

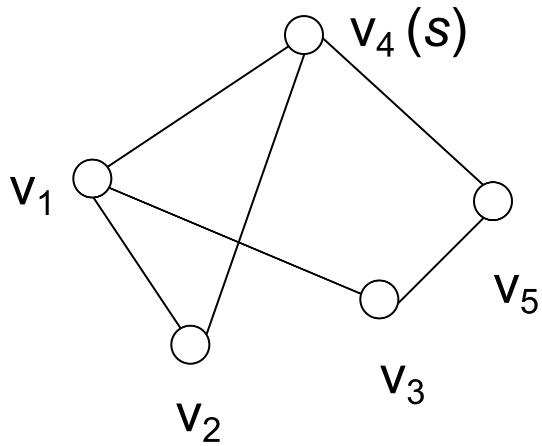


Graph Searching---BFS

Given a graph G = <V,E> and a source vertex s, breadth-first search explores the edges of G to visit every vertex reachable from s.

Assume that the adjacency list representation of G is given.





Graph Searching---BFS

Given a node *u*, *π[u]* is the parent of *u*,

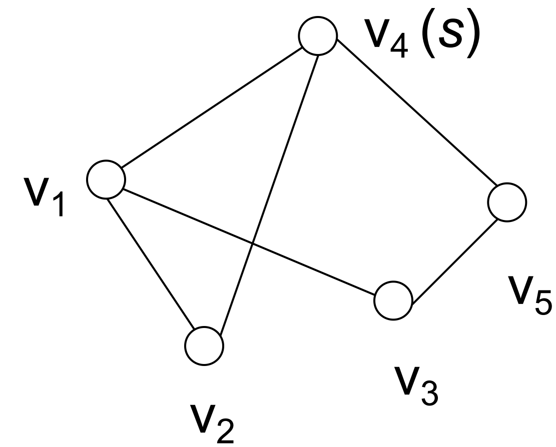
For illustration purpose, the color of *u* is denoted as *color[u]*. (Initially, *color[u]=white*.)

d[u] is the distance from *s* to *u*.

Example. After we have done with

the right example,

$d[v_2]=1$ and $d[v_3]=2$.

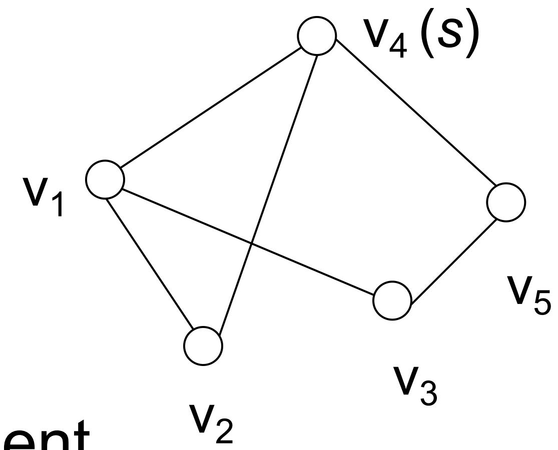


Graph Searching---BFS

Given a node *u, π[u]* is the parent of *u,*

For illustration purpose, the color of *u* is denoted as *color[u].* (Initially, *color[u]=white.*)

d[u] is the distance from *s* to *u.*



Idea: Initially, all the nodes have white color. Once a node is visited its color changes to nonwhite (yellow). A node has a **red color** only if all its incident vertices have nonwhite color (yellow or red).

广度优先搜索：“逐层搜索”法

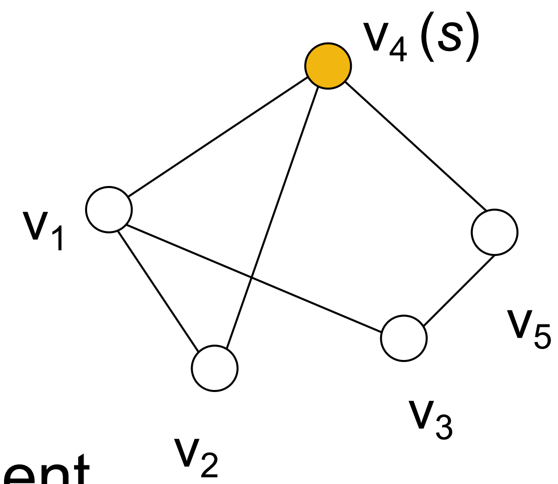
Graph Searching---BFS

Given a node *u, π[u]* is the parent of *u,*

For illustration purpose, the color of *u* is denoted as *color[u].* (Initially, *color[u]=white.*)

d[u] is the distance from *s* to *u.*

Idea: Initially, all the nodes have white color. Once a node is visited its color changes to nonwhite (**yellow**). A node has a **red color** only if all its incident vertices have nonwhite color (yellow or red).



广度优先搜索：“逐层搜索”法

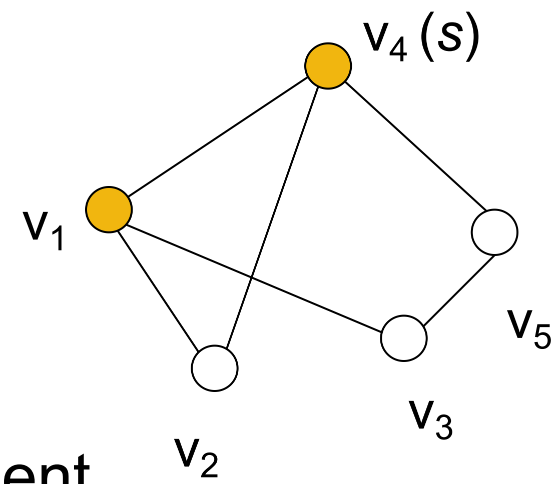
Graph Searching---BFS

Given a node *u, π[u]* is the parent of *u,*

For illustration purpose, the color of *u* is denoted as *color[u].* (Initially, *color[u]=white.*)

d[u] is the distance from *s* to *u.*

Idea: Initially, all the nodes have white color. Once a node is visited its color changes to nonwhite (**yellow**). A node has a **red color** only if all its incident vertices have nonwhite color (yellow or red).



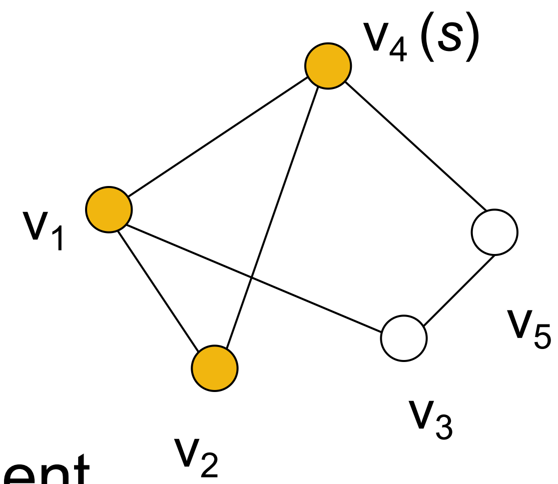
广度优先搜索：“逐层搜索”法

Graph Searching---BFS

Given a node *u*, *π[u]* is the parent of *u*,

For illustration purpose, the color of *u* is denoted as *color[u]*. (Initially, *color[u]=white.*)

d[u] is the distance from *s* to *u.*



Idea: Initially, all the nodes have white color. Once a node is visited its color changes to nonwhite (**yellow**). A node has a **red color** only if all its incident vertices have nonwhite color (yellow or red).

广度优先搜索：“逐层搜索”法

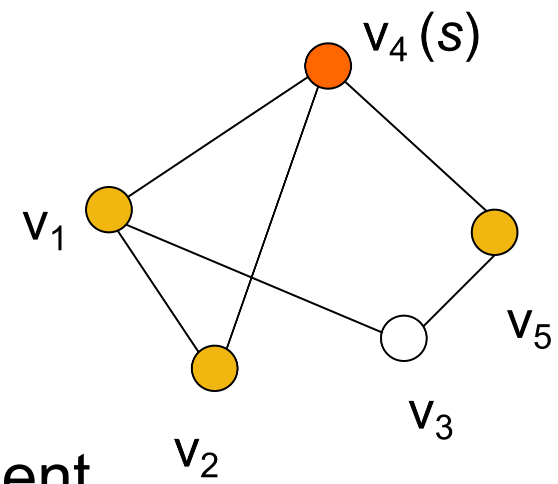
Graph Searching---BFS

Given a node *u*, *π[u]* is the parent of *u*,

For illustration purpose, the color of *u* is denoted as *color[u]*. (Initially, *color[u]=white*.)

d[u] is the distance from *s* to *u*.

Idea: Initially, all the nodes have white color. Once a node is visited its color changes to nonwhite (**yellow**). A node has a **red color** only if all its incident vertices have nonwhite color (yellow or red).



广度优先搜索：“逐层搜索”法

Breadth-First Search

For each u in V(G)-{s}

do color[u] $\leftarrow$ white

d[u] $\leftarrow \infty$

π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow \emptyset$

ENQUEUE(Q,s)

While Q $\neq \emptyset$

do $u \leftarrow$ DEQUEUE(Q)

for each v in Adj[u]

do if color[v] = white

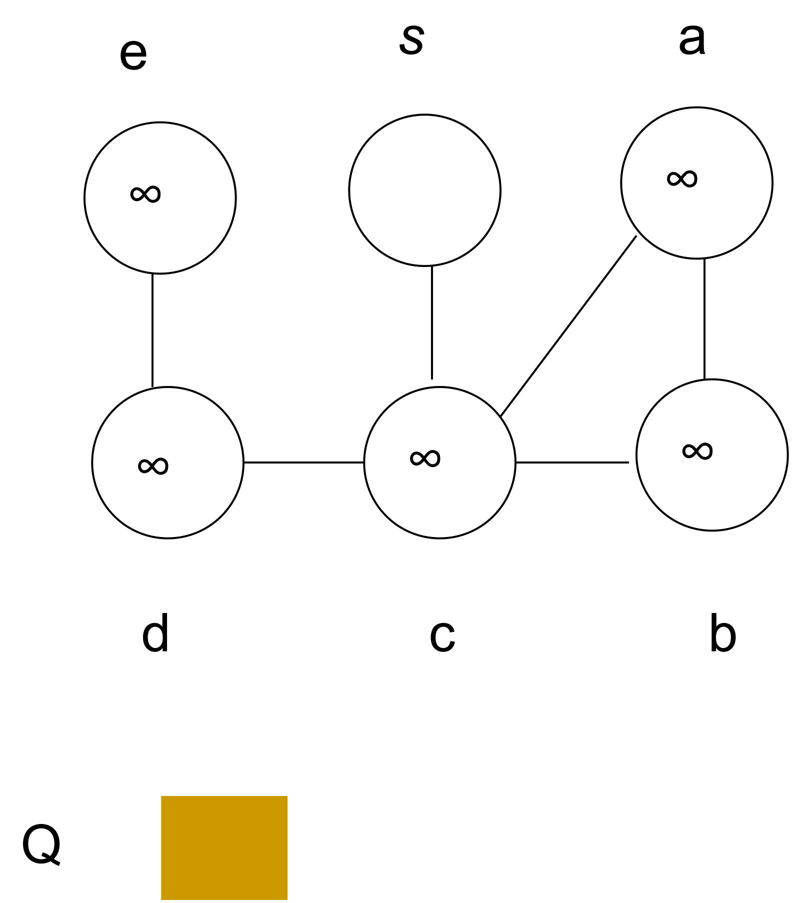
then color[v] $\leftarrow$ yellow

d[v] $\leftarrow$ d[u] + 1

π[v] $\leftarrow u$

ENQUEUE(Q,v)

color[u] $\leftarrow$ red



Breadth-First Search

For each u in V(G)-{s}

do color[u] $\leftarrow$ white

d[u] $\leftarrow \infty$

π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow$ Ø

ENQUEUE(Q,s)

While Q $\neq$ Ø

do $u \leftarrow$ DEQUEUE(Q)

for each v in Adj[u]

do if color[v] = white

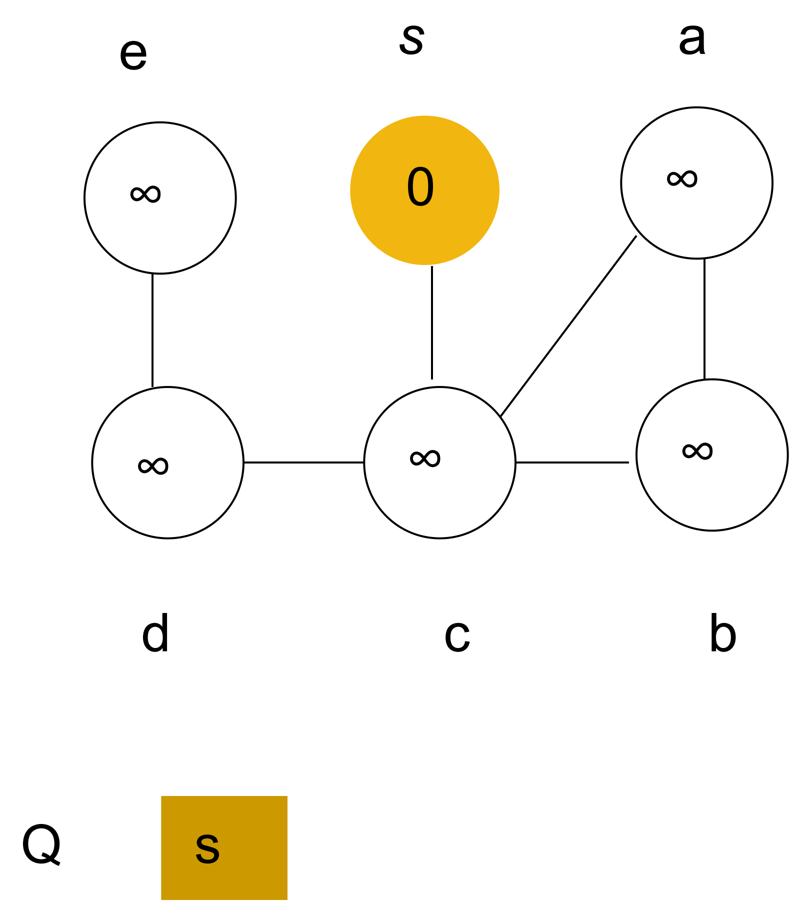
then color[v] $\leftarrow$ yellow

d[v] $\leftarrow$ d[u] + 1

π[v] $\leftarrow u$

ENQUEUE(Q,v)

color[u] $\leftarrow$ red



Breadth-First Search

For each u in V(G)-{s}

do color[u] $\leftarrow$ white

d[u] $\leftarrow \infty$

π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow \varnothing$

ENQUEUE(Q,s)

While Q $\neq \varnothing$

do $u \leftarrow$ DEQUEUE(Q)

for each v in Adj[u]

do if color[v] = white

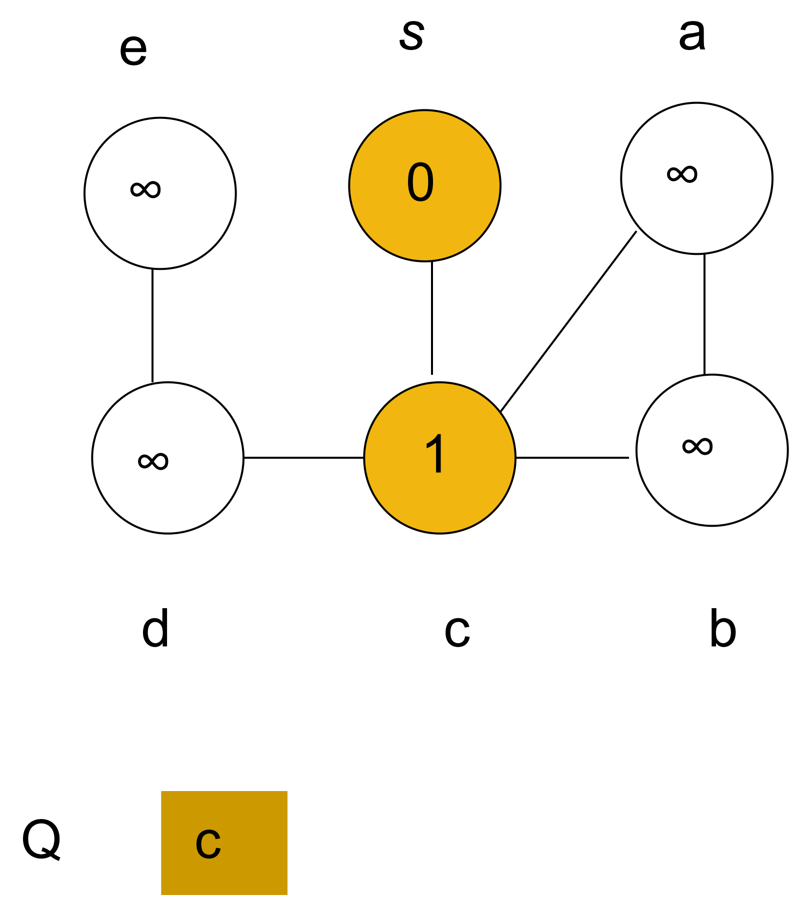
then color[v] $\leftarrow$ yellow

d[v] $\leftarrow$ d[u] + 1

π[v] $\leftarrow$ u

ENQUEUE(Q,v)

color[u] $\leftarrow$ red



Breadth-First Search

For each *u* in V(G)-{s}

do color[*u*] ← white

d[*u*] ← ∞

π[*u*] ← NIL

color[s] ← yellow, d[s] ← 0, π[s] ← NIL

Q ← Ø

ENQUEUE(Q,s)

While Q ≠ Ø

do *u* ← DEQUEUE(Q)

for each v in Adj[*u*]

do if color[v] = white

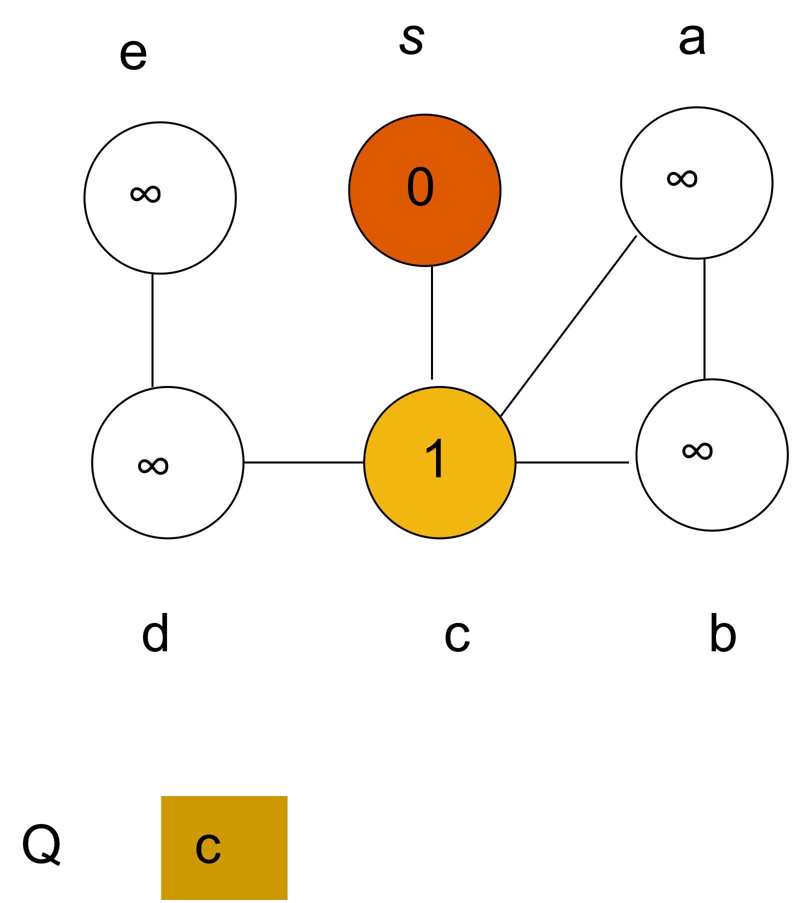
then color[v] ← yellow

d[v] ← d[*u*] + 1

π[v] ← *u*

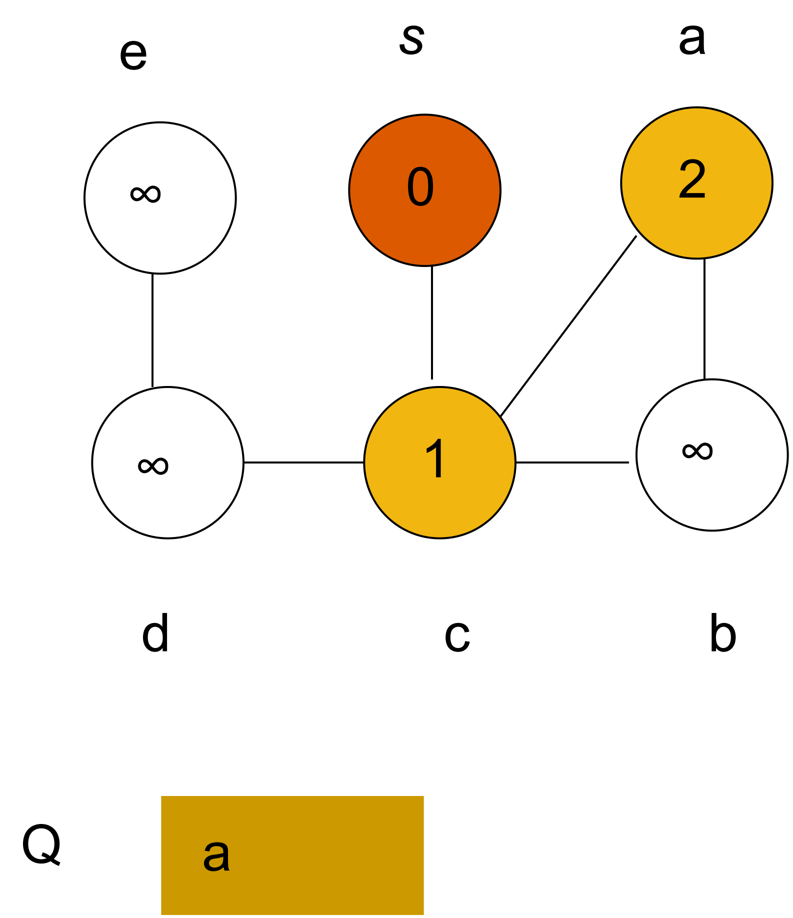
ENQUEUE(Q,v)

color[*u*] ← red



Breadth-First Search

```
For each u in V(G)-{s}
   do color[u] ← white
        d[u] ← ∞
        π[u] ← NIL
color[s] ← yellow, d[s] ← 0, π[s] ← NIL
Q ← Ø
ENQUEUE(Q,s)
While Q ≠ Ø
   do u ← DEQUEUE(Q)
        for each v in Adj[u]
              do if color[v] = white
                       then color[v] ← yellow
                                d[v] ← d[u] + 1
                                π[v] ← u
                                ENQUEUE(Q,v)
        color[u] ← red
```



Breadth-First Search

For each u in V(G)-{s}

    do color[u] ← white

        d[u] ← ∞

        π[u] ← NIL

color[s] ← yellow, d[s] ← 0, π[s] ← NIL

Q ← Ø

ENQUEUE(Q,s)

While Q ≠ Ø

    do u ← DEQUEUE(Q)

        for each v in Adj[u]

            do if color[v] = white

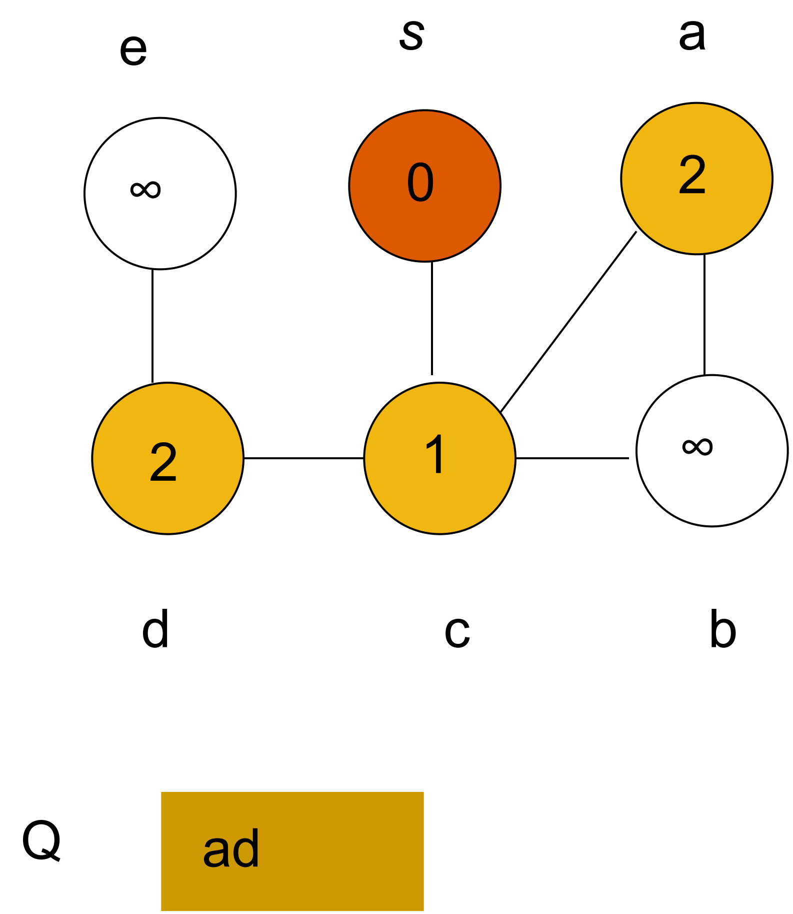
                then color[v] ← yellow

                    d[v] ← d[u] + 1

                    π[v] ← u

                    ENQUEUE(Q,v)

    color[u] ← red



Breadth-First Search

For each u in V(G)-{s}

do color[u] ← white

d[u] ← ∞

π[u] ← NIL

color[s] ← yellow, d[s] ← 0, π[s] ← NIL

Q ← Ø

ENQUEUE(Q,s)

While Q ≠ Ø

do u ← DEQUEUE(Q)

for each v in Adj[u]

do if color[v] = white

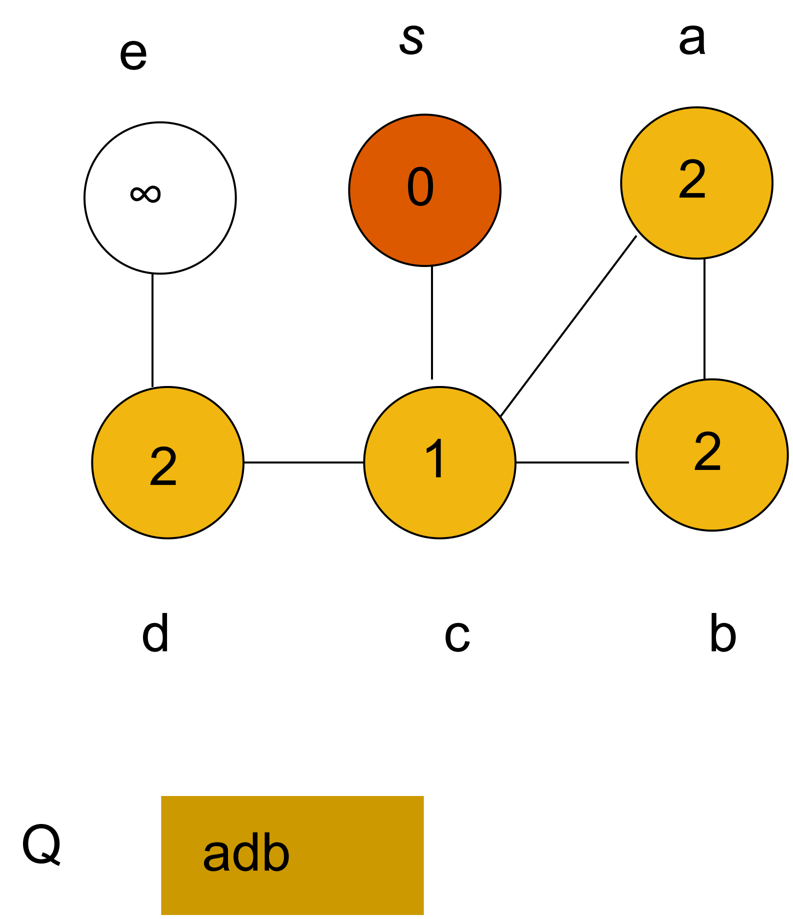
then color[v] ← yellow

d[v] ← d[u] + 1

π[v] ← u

ENQUEUE(Q,v)

color[u] ← red



Breadth-First Search

For each u in V(G)-{s}

  do color[u] $\leftarrow$ white

    d[u] $\leftarrow \infty$

    π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow$ Ø

ENQUEUE(Q,s)

While Q ≠ Ø

  do $u \leftarrow$ DEQUEUE(Q)

    for each v in Adj[u]

      do if color[v] = white

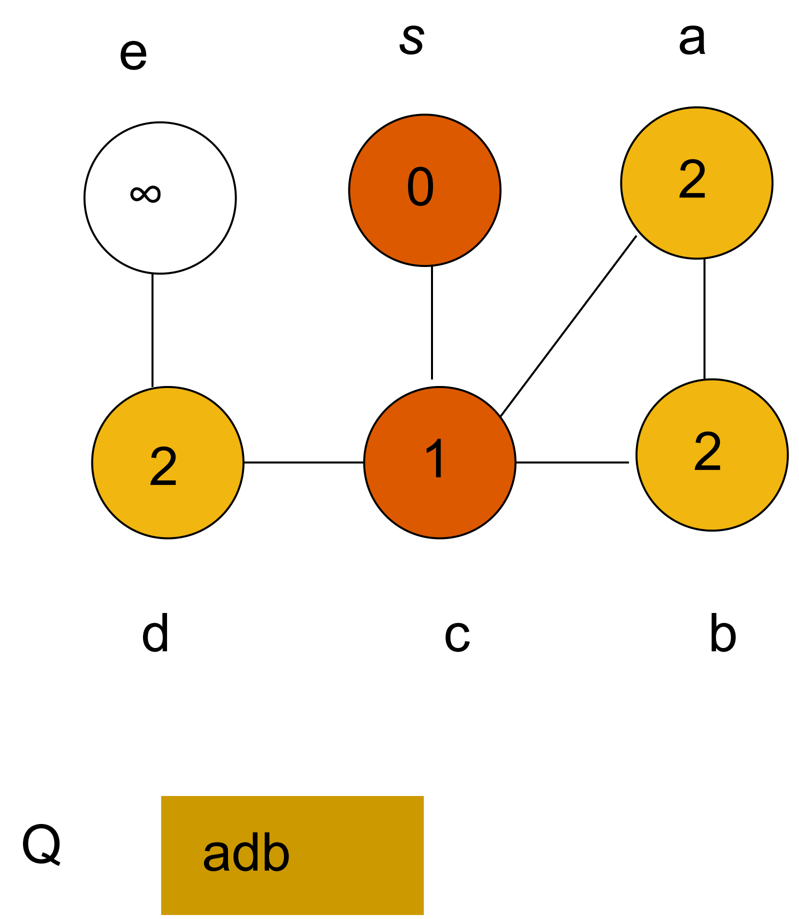
        then color[v] $\leftarrow$ yellow

          d[v] $\leftarrow$ d[u] + 1

          π[v] $\leftarrow u$

          ENQUEUE(Q,v)

    color[u] $\leftarrow$ red



Breadth-First Search

For each u in V(G)-{s}

  do color[u] $\leftarrow$ white

    d[u] $\leftarrow \infty$

    π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow \emptyset$

ENQUEUE(Q,s)

While Q $\neq \emptyset$

  do $u \leftarrow$ DEQUEUE(Q)

    for each v in Adj[u]

      do if color[v] = white

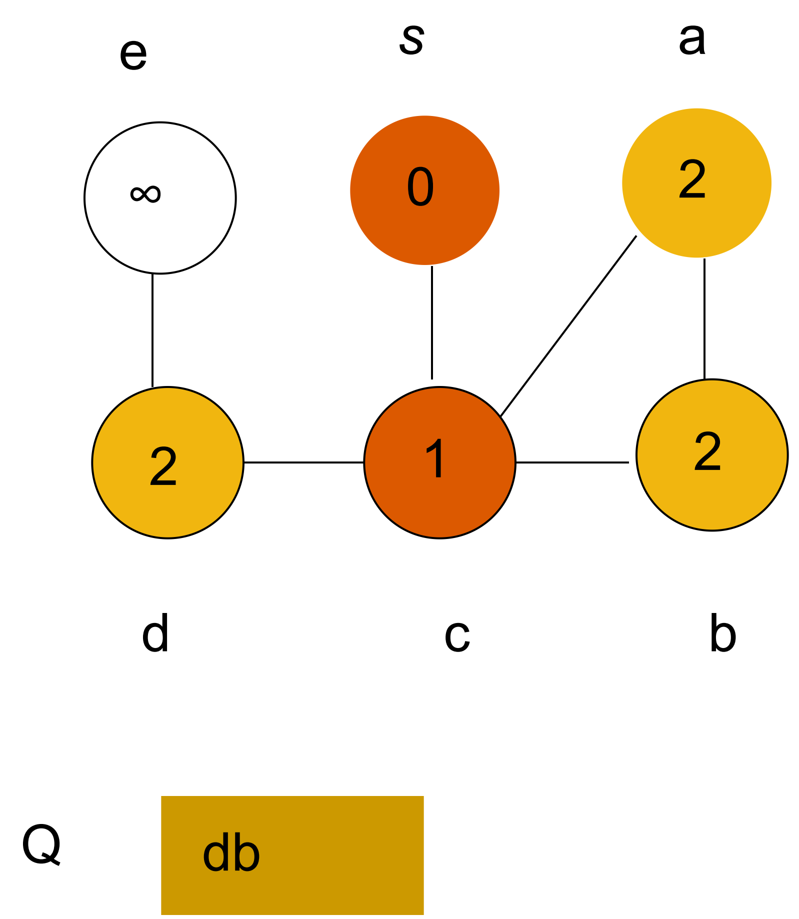
        then color[v] $\leftarrow$ yellow

          d[v] $\leftarrow$ d[u] + 1

          π[v] $\leftarrow u$

          ENQUEUE(Q,v)

    color[u] $\leftarrow$ red



Breadth-First Search

For each u in V(G)-{s}

  do color[u] $\leftarrow$ white

    d[u] $\leftarrow \infty$

    π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow \varnothing$

ENQUEUE(Q,s)

While Q $\neq \varnothing$

  do $u \leftarrow$ DEQUEUE(Q)

    for each v in Adj[u]

      do if color[v] = white

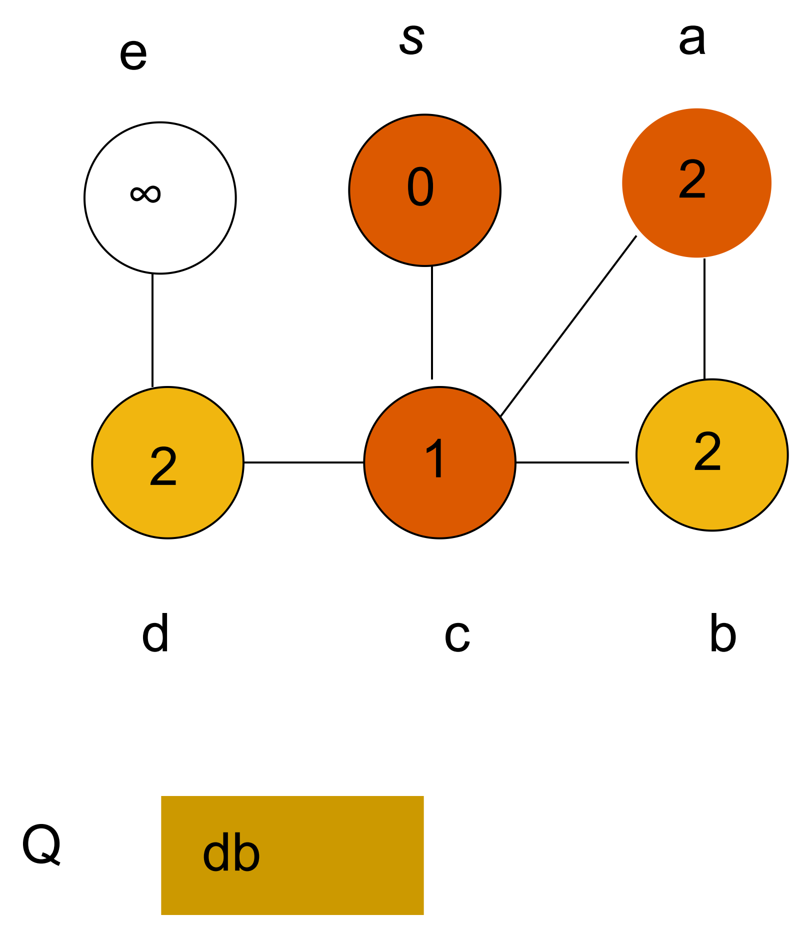
        then color[v] $\leftarrow$ yellow

          d[v] $\leftarrow$ d[u] + 1

          π[v] $\leftarrow$ u

          ENQUEUE(Q,v)

  color[u] $\leftarrow$ red



Breadth-First Search

For each u in V(G)-{s}

  do color[u] ← white

    d[u] ← ∞

    π[u] ← NIL

color[s] ← yellow, d[s] ← 0, π[s] ← NIL

Q ← Ø

ENQUEUE(Q,s)

While Q ≠ Ø

  do u ← DEQUEUE(Q)

    for each v in Adj[u]

      do if color[v] = white

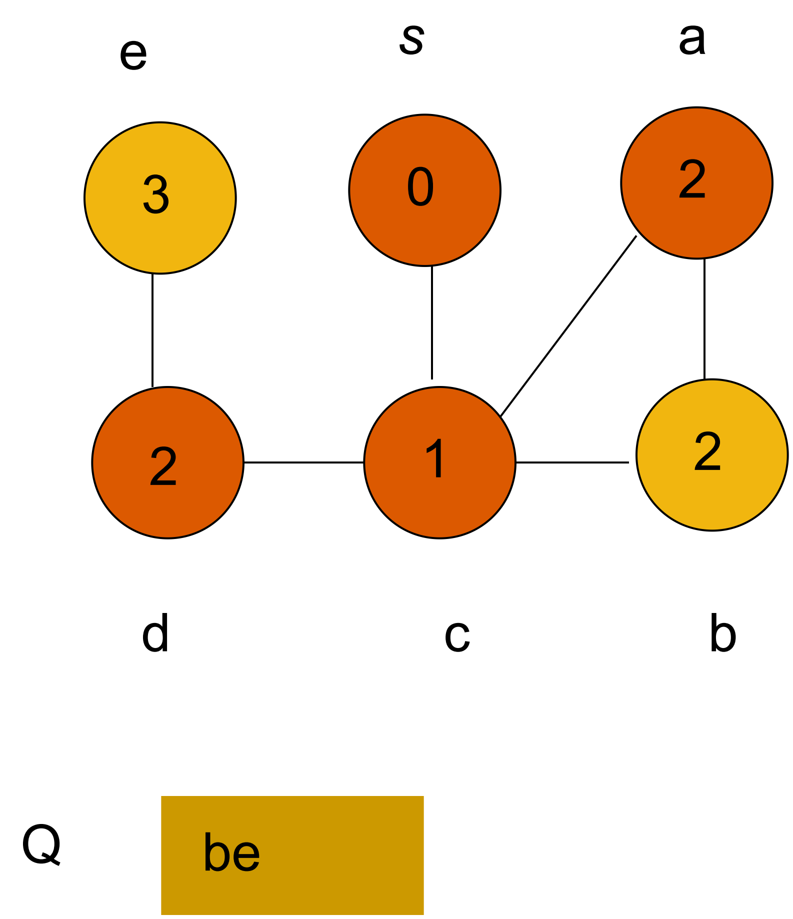
        then color[v] ← yellow

          d[v] ← d[u] + 1

          π[v] ← u

          ENQUEUE(Q,v)

  color[u] ← red



Breadth-First Search

For each *u* in V(G)-{s}

do color[*u*] ← white

d[*u*] ← ∞

π[*u*] ← NIL

color[s] ← yellow, d[s] ← 0, π[s] ← NIL

Q ← Ø

ENQUEUE(Q,s)

While Q ≠ Ø

do *u* ← DEQUEUE(Q)

for each v in Adj[*u*]

do if color[v] = white

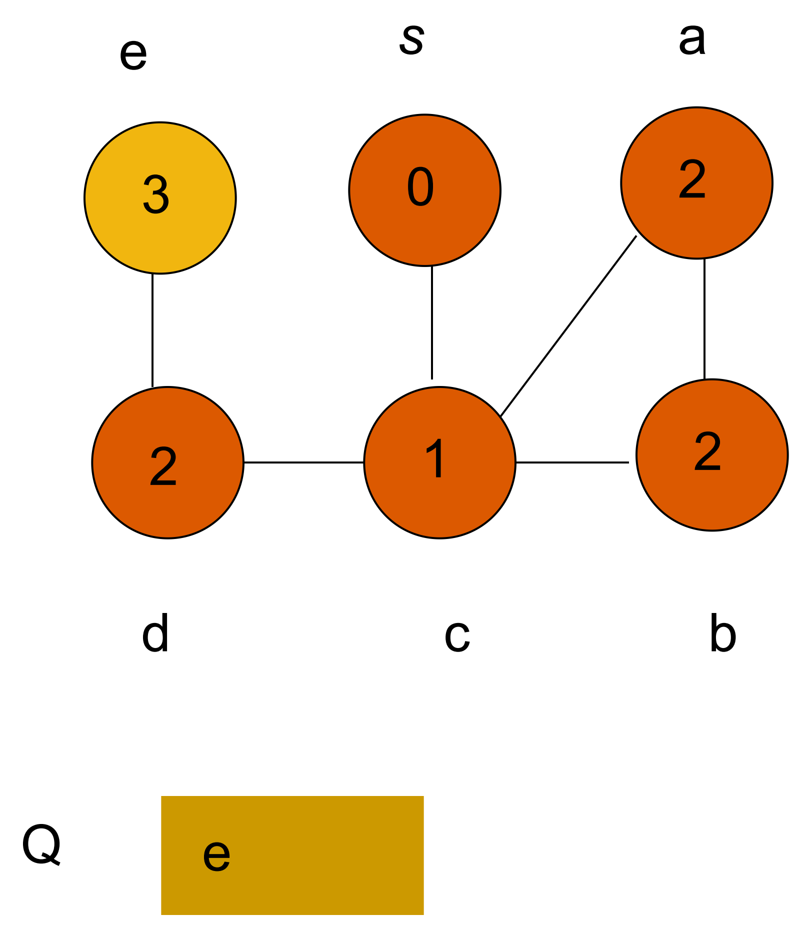
then color[v] ← yellow

d[v] ← d[*u*] + 1

π[v] ← *u*

ENQUEUE(Q,v)

color[*u*] ← red



Breadth-First Search

For each u in V(G)-{s}

  do color[u] $\leftarrow$ white

    d[u] $\leftarrow \infty$

    π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow \emptyset$

ENQUEUE(Q,s)

While Q $\neq \emptyset$

  do $u \leftarrow$ DEQUEUE(Q)

    for each v in Adj[*u*]

      do if color[v] = white

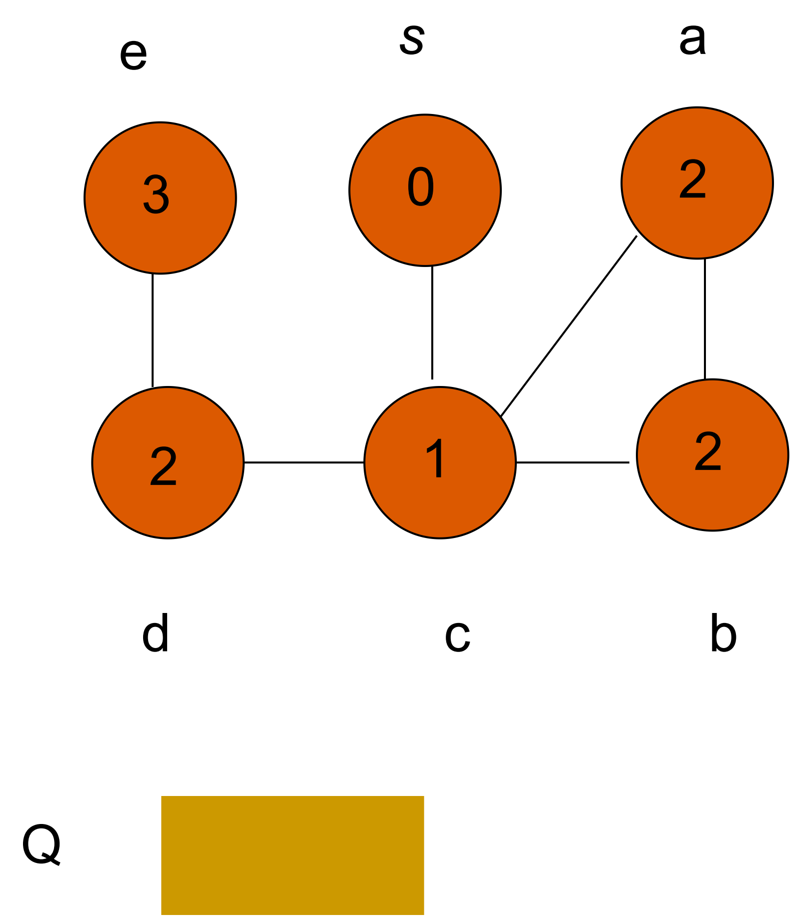
        then color[v] $\leftarrow$ yellow

          d[v] $\leftarrow$ d[u] + 1

          π[v] $\leftarrow$ u

          ENQUEUE(Q,v)

  color[u] $\leftarrow$ red



Breadth-First Search

```
For each u in V(G)-{s}
  do color[u] ← white
     d[u] ← ∞
     π[u] ← NIL
color[s] ← yellow, d[s] ← 0, π[s] ← NIL
Q ← Ø
ENQUEUE(Q,s)
While Q ≠ Ø
  do u ← DEQUEUE(Q)
     for each v in Adj[u]
         do if color[v] = white
               then color[v] ← yellow
                    d[v] ← d[u] + 1
                    π[v] ← u
                    ENQUEUE(Q,v)
     color[u] ← red
```

Why BFS works?

Breadth-First Search

For each u in V(G)-{s}

  do color[u] ← white

    d[u] ← ∞

    π[u] ← NIL

color[s] ← yellow, d[s] ← 0, π[s] ← NIL

Q ← Ø

ENQUEUE(Q,s)

While Q ≠ Ø

  do u ← DEQUEUE(Q)

    for each v in Adj[u]

      do if color[v] = white

        then color[v] ← yellow

          d[v] ← d[u] + 1

          π[v] ← u

          ENQUEUE(Q,v)

    color[u] ← red

What is the running time of BFS?

Breadth-First Search

For each u in V(G)-{s}

  do color[u] $\leftarrow$ white

    d[u] $\leftarrow \infty$

    π[u] $\leftarrow$ NIL

color[s] $\leftarrow$ yellow, d[s] $\leftarrow$ 0, π[s] $\leftarrow$ NIL

Q $\leftarrow \emptyset$

ENQUEUE(Q,s)

While Q $\neq \emptyset$

  do $u \leftarrow$ DEQUEUE(Q)

    for each v in Adj[u]

      do if color[v] = white

        then color[v] $\leftarrow$ yellow

          d[v] $\leftarrow$ d[u] + 1

          π[v] $\leftarrow u$

          ENQUEUE(Q,v)

    color[u] $\leftarrow$ red

What is the running time of BFS?

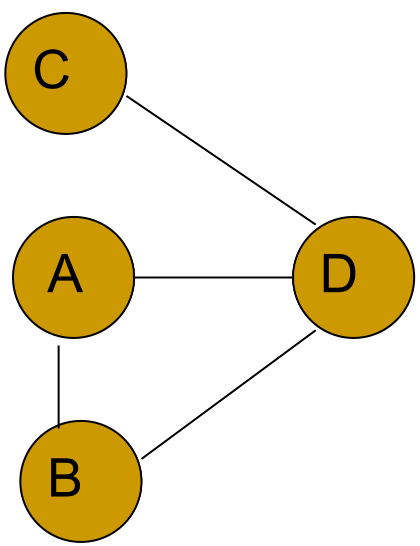
O(|V|+|E|).

Graph Searching --- DFS

DFS: 结点检测中，当有新的结点到达，就终止对原来结点的检测，开始新结点的检测，新结点被检测后，再恢复对原结点的检测。

Depth-First Search is one of the most powerful searching methods in Artificial Intelligence.

When one is exploring a maze or playing chess, DFS is usually unconsciously used.

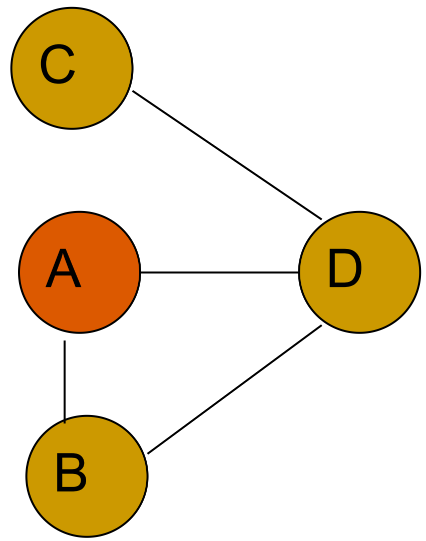


Graph Searching --- DFS

Depth-First Search is one of the most powerful searching methods in Artificial Intelligence.

When one is exploring a maze or playing chess, DFS is usually unconsciously used.

If we start at A …

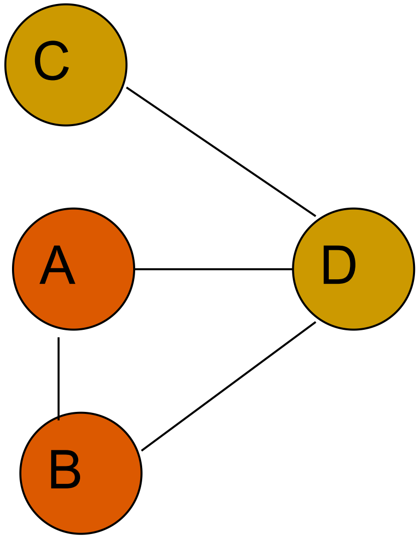


Graph Searching --- DFS

Depth-First Search is one of the most powerful searching methods in Artificial Intelligence.

When one is exploring a maze or playing chess, DFS is usually unconsciously used.

If we start at A …

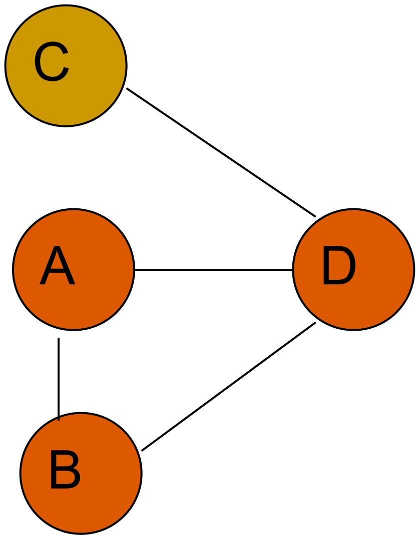


Graph Searching --- DFS

Depth-First Search is one of the most powerful searching methods in Artificial Intelligence.

When one is exploring a maze or playing chess, DFS is usually unconsciously used.

If we start at A …

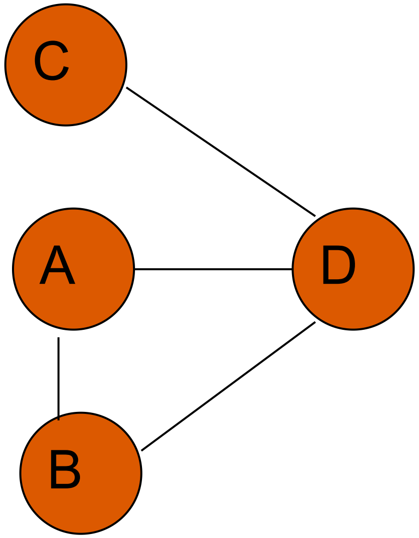


Graph Searching --- DFS

Depth-First Search is one of the most powerful searching methods in Artificial Intelligence.

When one is exploring a maze or playing chess, DFS is usually unconsciously used.

If we start at A …

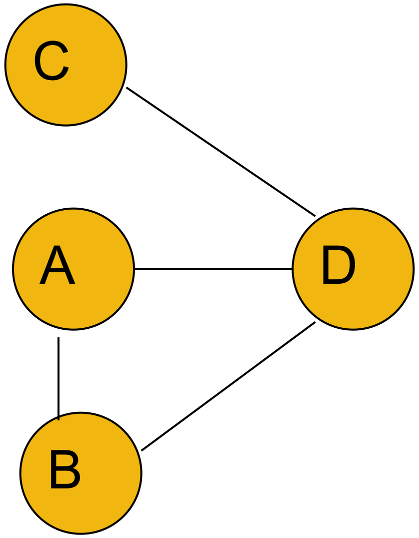


Graph Searching --- DFS

Depth-First Search is one of the most powerful searching methods in Artificial Intelligence.

When one is exploring a maze or playing chess, DFS is usually unconsciously used.

What if we start at D?

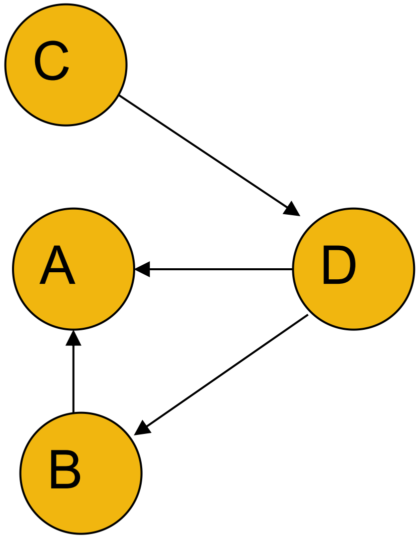


Graph Searching --- DFS

Depth-First Search is one of the most powerful searching methods in Artificial Intelligence.

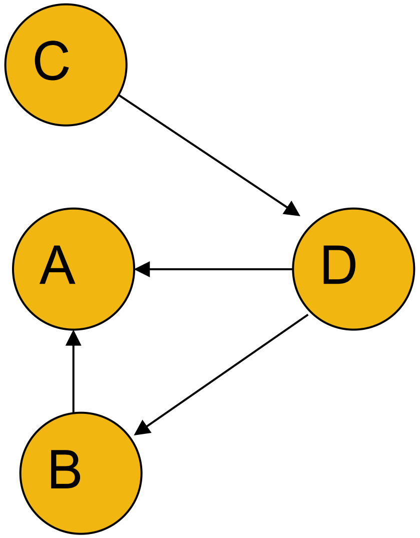
When one is exploring a maze or playing chess, DFS is usually unconsciously used.

Or what if the edges are directed?



Directed Graph

A directed graph is the same as an undirected graph except that the edges are directed, i.e., a directed edge <u,v> does not imply that <v,u> is also an edge of the graph. (In an undirected graph, (u,v) and (v,u) denote the same edge.)

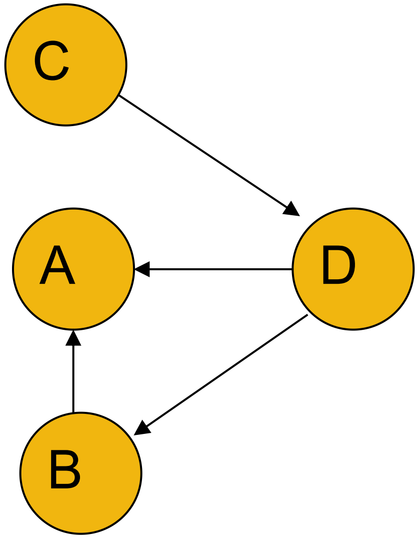


Depth-First Search

A directed graph is the same as an undirected graph except that the edges are directed, i.e., a directed edge <u,v> does not imply that <v,u> is also an edge of the graph. (In an undirected graph, (u,v) and (v,u) denote the same edge.)

Q: What are the vertices adjacent

to D (or what are the neighbors of D)?



Depth-First Search

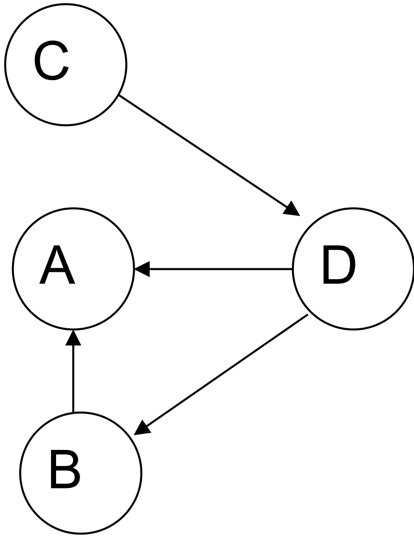
Given a node *v, π[v]* is the predecessor of *v.*

Initially, all the nodes are White. When a node

is discovered, it is colored **Yellow** and when

All its neighbors have all been discovered, it

is changed to **Red**.



Each *v* has two timestamps: d[v] is the time v is discovered (v is colored Yellow), f[v] is the time when all the neighbors of v are discovered (v is then colored Red.)

Depth-First Search

Given a node *v*, *π[v]* is the predecessor of *v.*

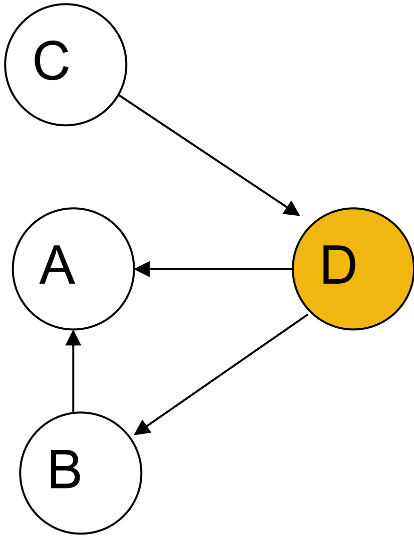
Initially, all the nodes are White. When a node

is discovered, it is colored **Yellow** and when

All its neighbors have all been discovered, it

is changed to **Red**.

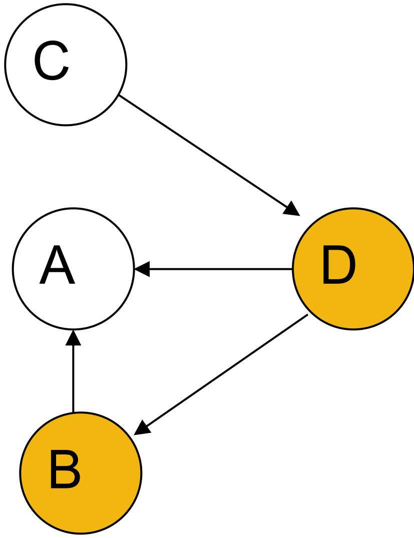
Each *v* has two timestamps: d[v] is the time v is discovered (v is colored Yellow), f[v] is the time when all the neighbors of v are discovered (v is then colored Red.)



Depth-First Search

Given a node *v, π[v]* is the predecessor of *v.*

Initially, all the nodes are White. When a node is discovered, it is colored **Yellow** and when All its neighbors have all been discovered, it is changed to **Red**.



Each *v* has two timestamps: d[v] is the time v is discovered (v is colored Yellow), f[v] is the time when all the neighbors of v are discovered (v is then colored Red.)

Depth-First Search

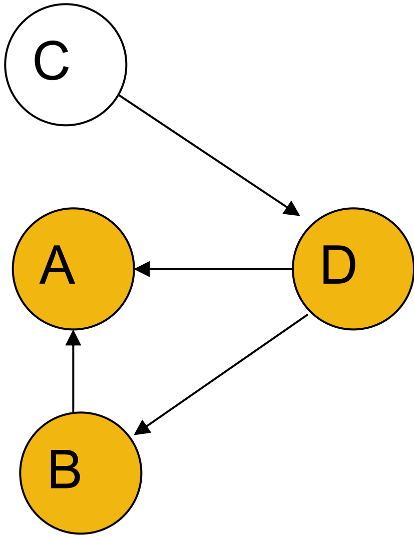
Given a node *v, π[v]* is the predecessor of *v.*

Initially, all the nodes are White. When a node

is discovered, it is colored **Yellow** and when

All its neighbors have all been discovered, it

is changed to **Red**.



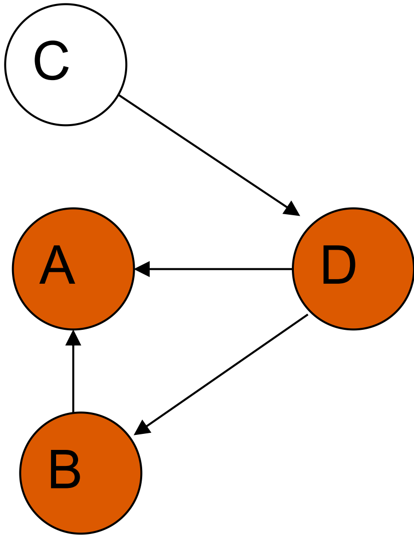
Each *v* has two timestamps: d[v] is the time v is discovered (v is colored Yellow), f[v] is the time when all the neighbors of v are discovered (v is then colored Red.)

Depth-First Search

Given a node *v, π[v]* is the predecessor of *v.*

Initially, all the nodes are White. When a node is discovered, it is colored **Yellow** and when All its neighbors have all been discovered, it is changed to **Red**.

Each *v* has two timestamps: d[v] is the time v is discovered (v is colored Yellow), f[v] is the time when all the neighbors of v are discovered (v is then colored Red.)

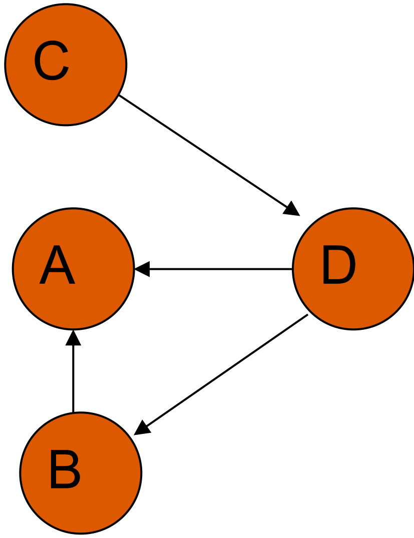


Depth-First Search

Given a node *v*, *π[v]* is the predecessor of *v.*

Initially, all the nodes are White. When a node is discovered, it is colored **Yellow** and when All its neighbors have all been discovered, it is changed to **Red**.

Each *v* has two timestamps: d[v] is the time v is discovered (v is colored Yellow), f[v] is the time when all the neighbors of v are discovered (v is then colored Red.)



Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1

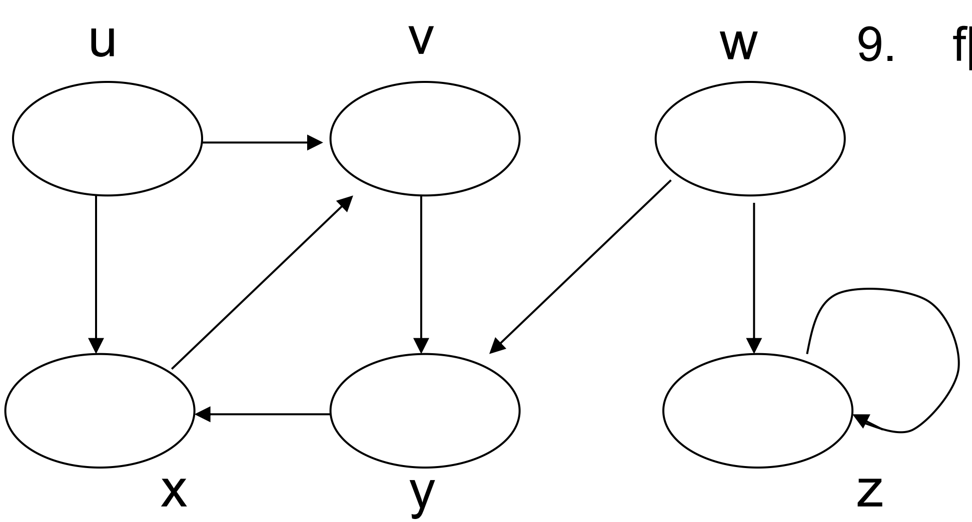
Depth-First Search

DFS(G)

```
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

DFS-visit(u)

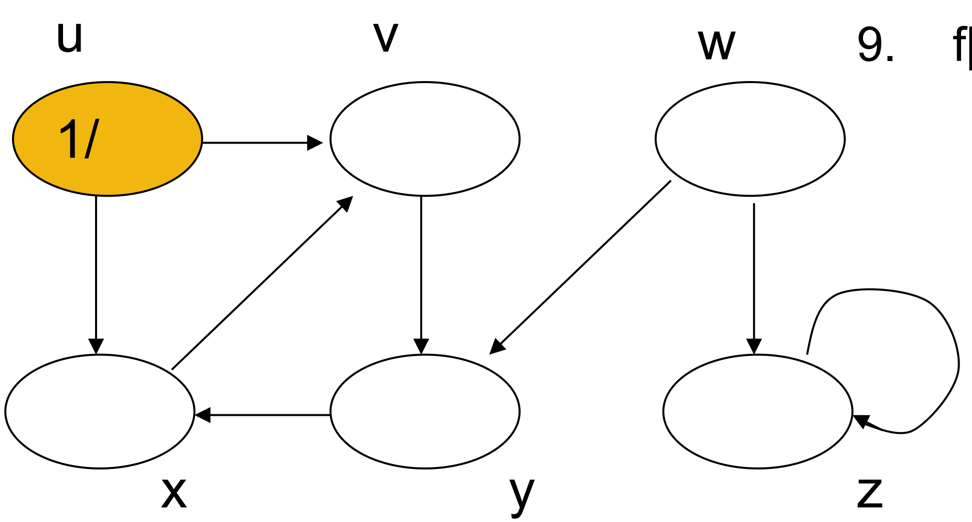
```
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



Depth-First Search

```
DFS(G)
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

```
DFS-visit(u)
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



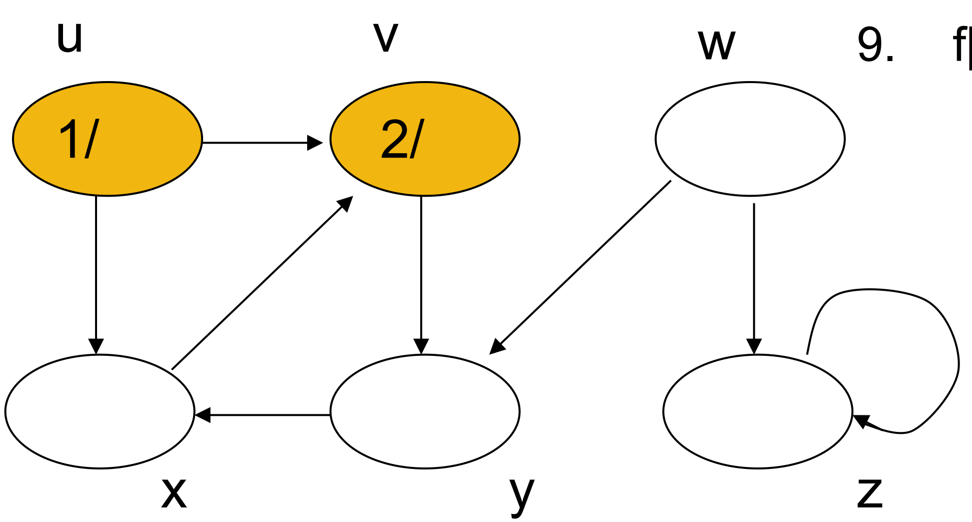
Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1



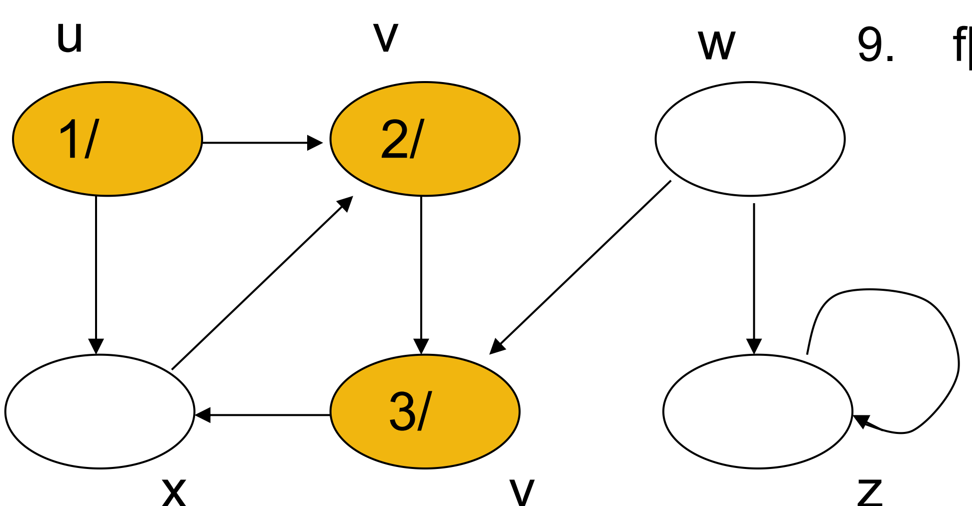
Depth-First Search

DFS(G)

```
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

DFS-visit(u)

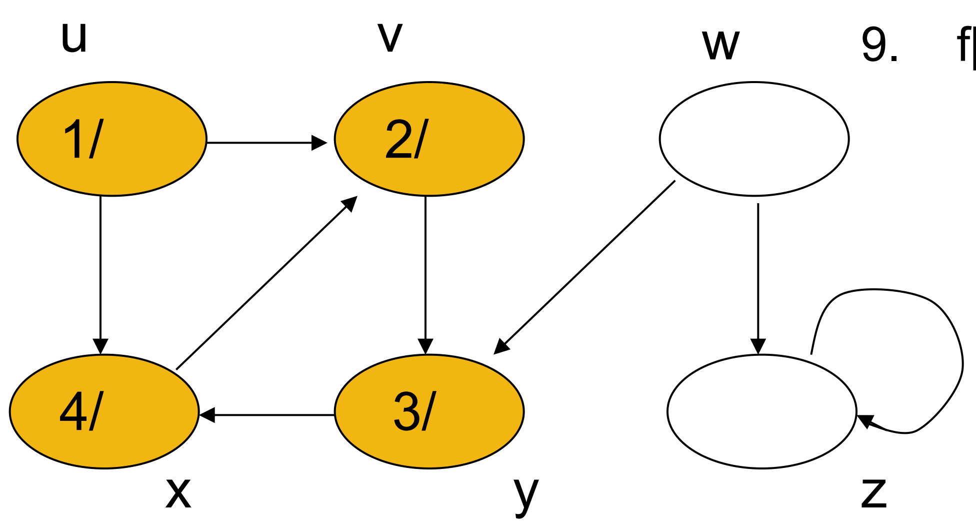
```
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



Depth-First Search

```
DFS(G)
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

```
DFS-visit(u)
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



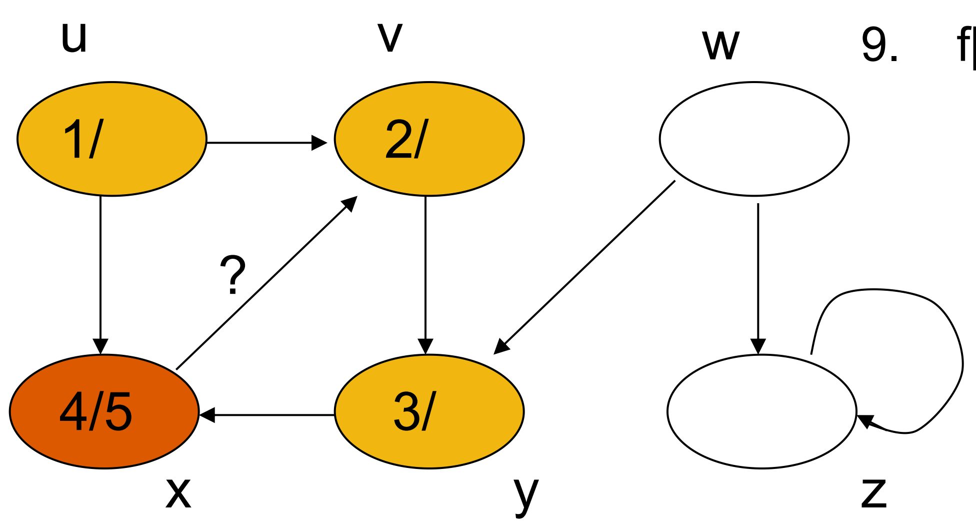
Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1



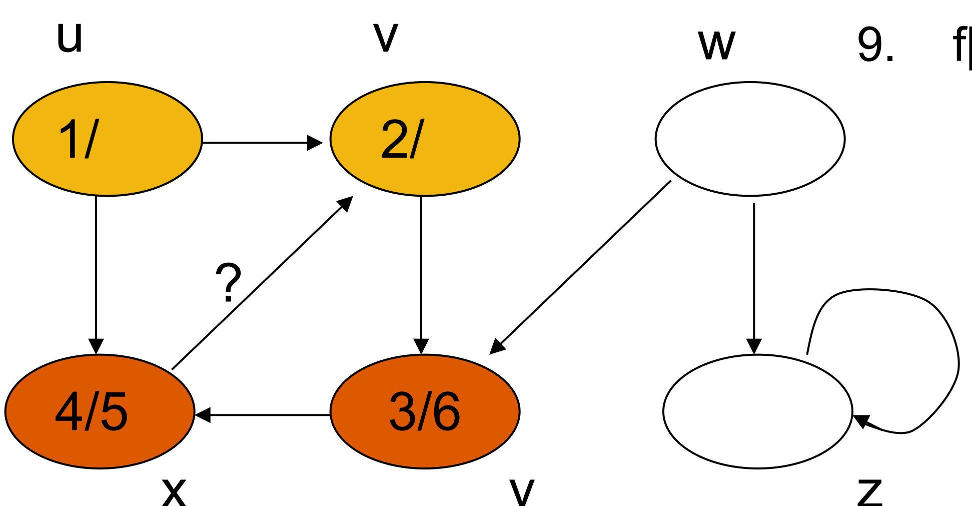
Depth-First Search

DFS(G)

```
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

DFS-visit(u)

```
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



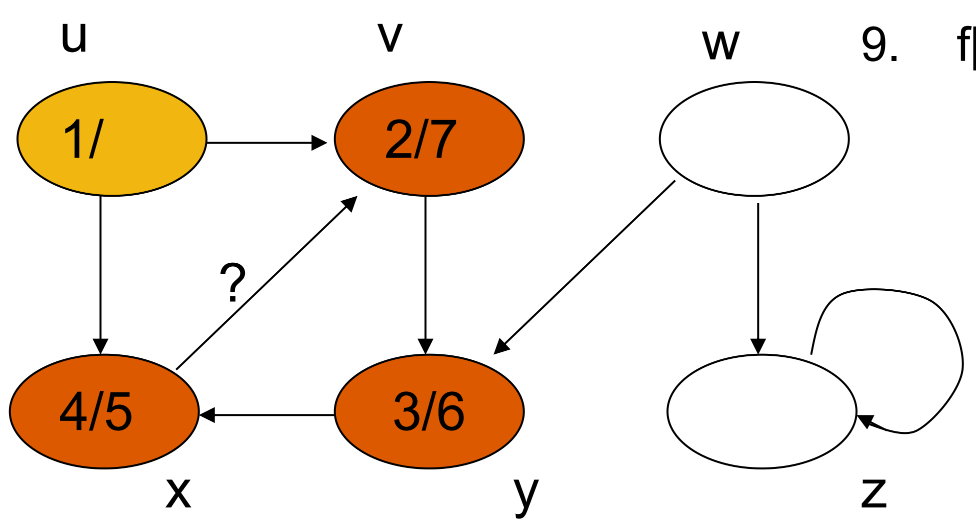
Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1



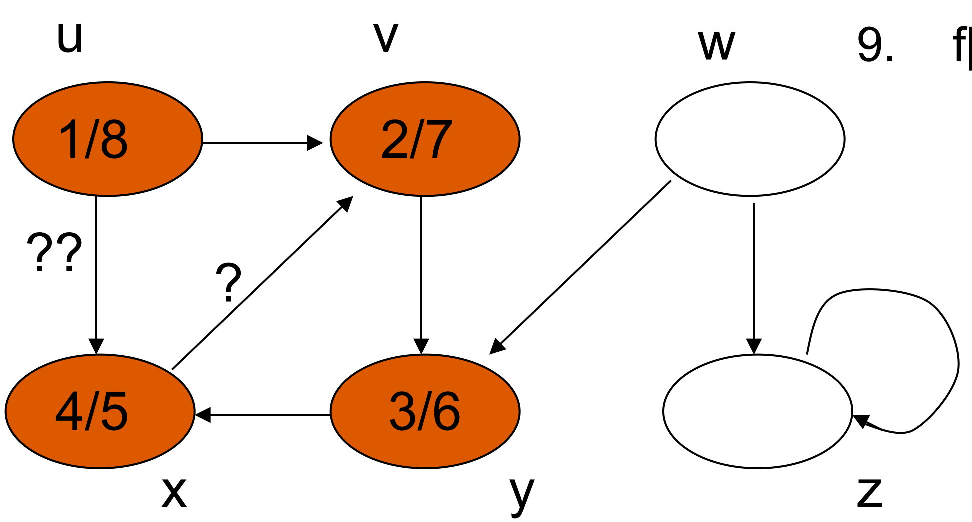
Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1



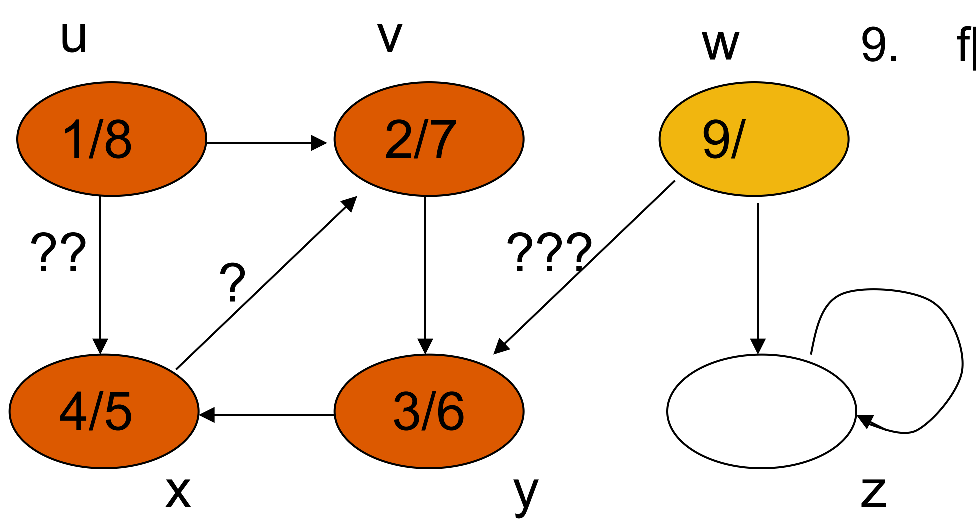
Depth-First Search

DFS(G)

```
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

DFS-visit(u)

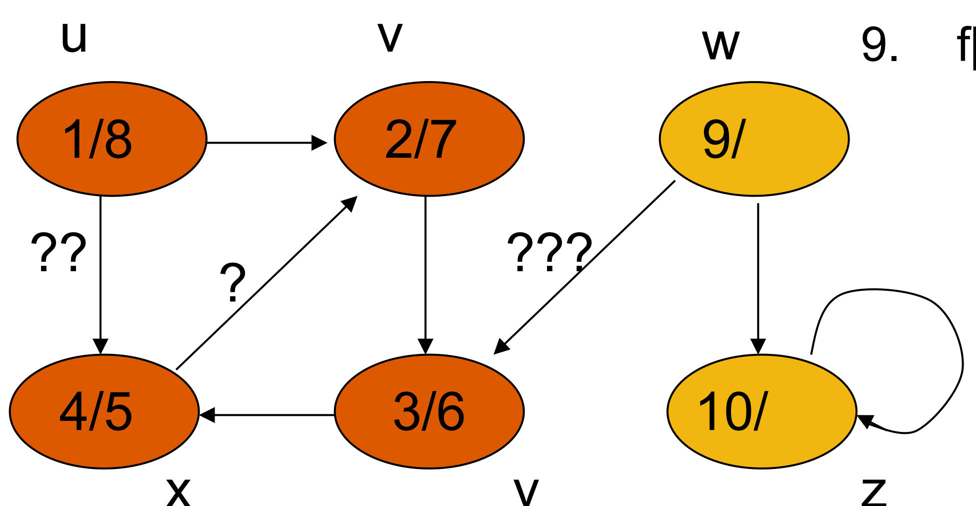
```
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



Depth-First Search

```
DFS(G)
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

```
DFS-visit(u)
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



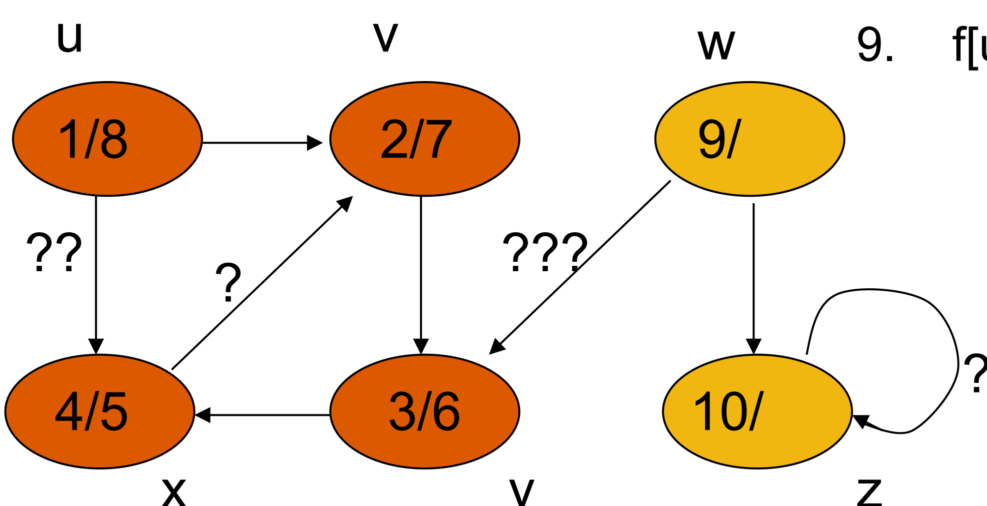
Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1



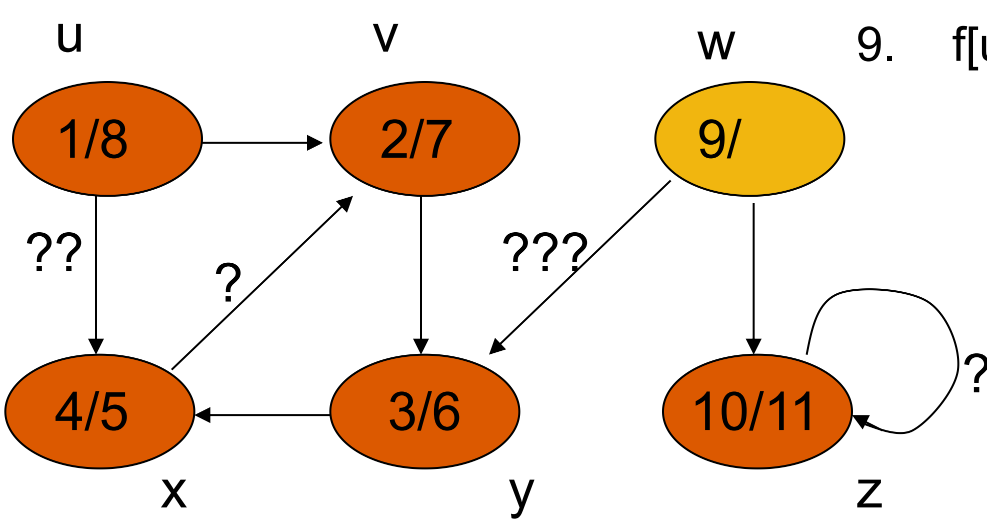
Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1



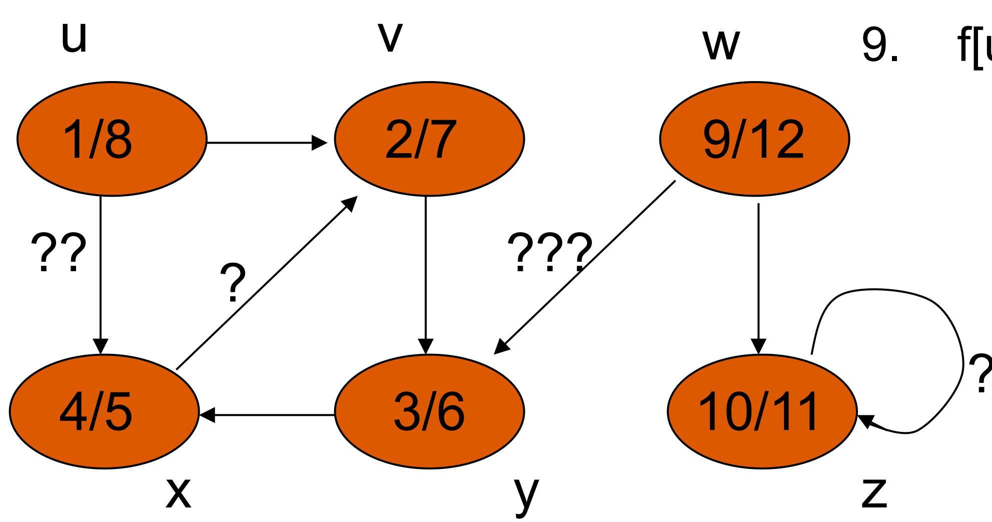
Depth-First Search

DFS(G)

1. For each vertex u in V[G]
2. do color[u] ← White
3. π[u] ← NIL
4. time ← 0
5. For each vertex u in V[G]
6. do if color[u] = White
7. then DFS-visit(u)

DFS-visit(u)

1. color[u] ← Yellow
2. time ← time +1
3. d[u] ← time
4. For each vertex v in Adj[u]
5. do if color[v] = White
6. then π[v] ← u
7. DFS-visit(v)
8. color[u] ← Red
9. f[u] ← time ← time +1



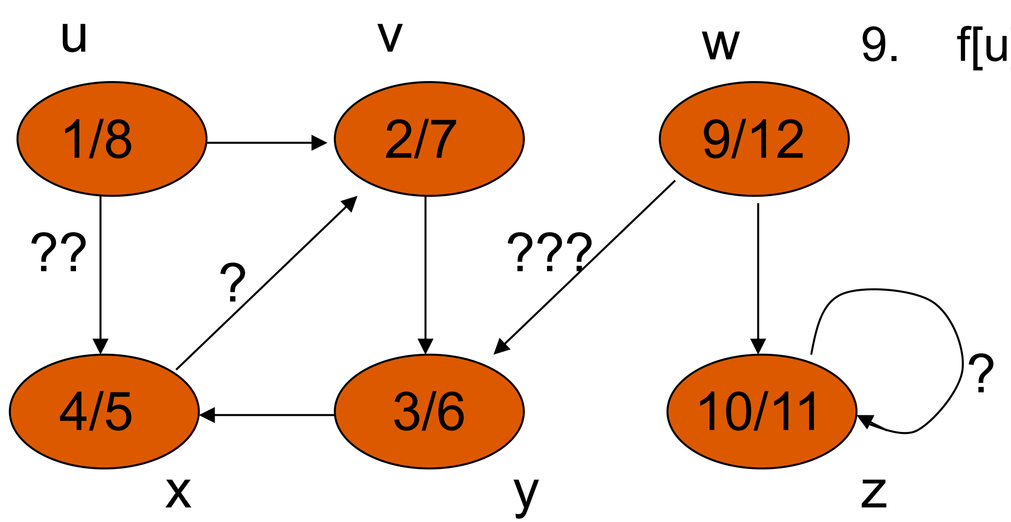
Depth-First Search

DFS(G)

```
For each vertex u in V[G]
    do color[u] ← White
        π[u] ← NIL
time ← 0
For each vertex u in V[G]
    do if color[u] = White
        then DFS-visit(u)
```

DFS-visit(u)

```
color[u] ← Yellow
time ← time +1
d[u] ← time
For each vertex v in Adj[u]
    do if color[v] = White
        then π[v] ← u
            DFS-visit(v)
color[u] ← Red
f[u] ← time ← time +1
```



Running time?

O(V+E).

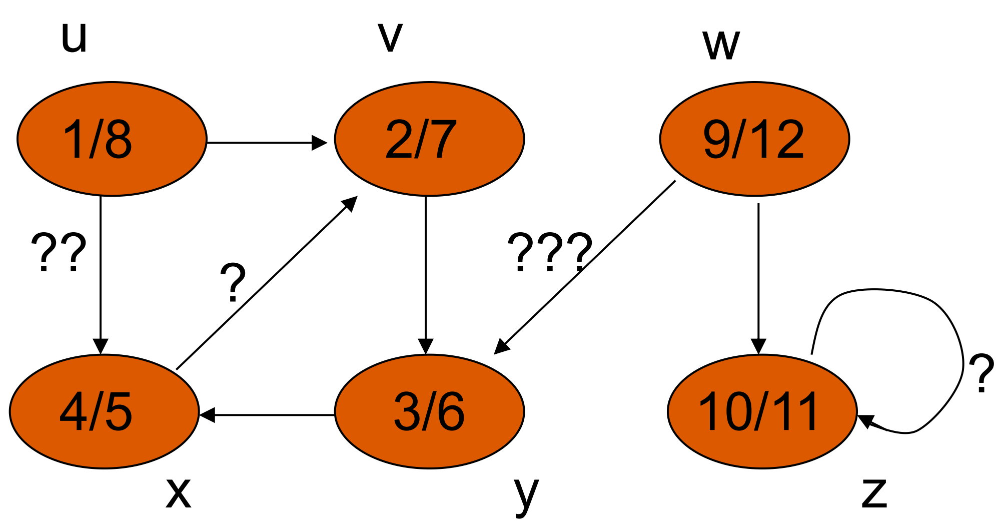
Depth-First Search

结点的发现时间和完成时间具有括号化结构。

左括号 “(u”表示结点u的发现，右括号 “u)”表示结点u的完成。

Parenthesis theorem.

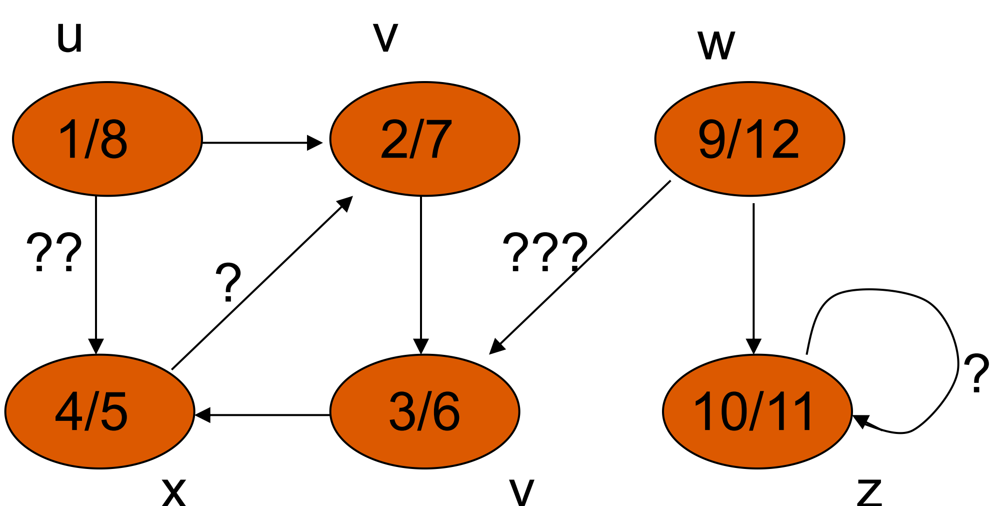
1. If (d[u],f[u]) and (d[v],f[v]) are disjoint, then u and v are not in the same tree.
2. If (d[u],f[u]) is contained in (d[v],f[v]), then u is a descendant of v, and vice versa.



Depth-First Search

There are 4 kinds of edges when we run DFS.

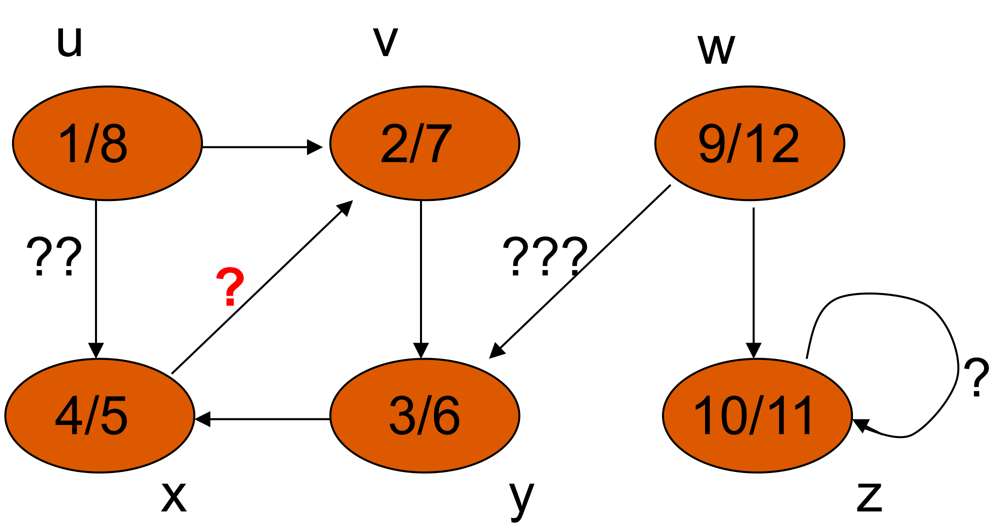
1. Tree edges →.
2. Back edges →?.
3. Forward edges →??.
4. Cross edges →??? (all other edges except the first 3 types)



Depth-First Search

How do we make use of these edges to decide whether a directed graph has a cycle?

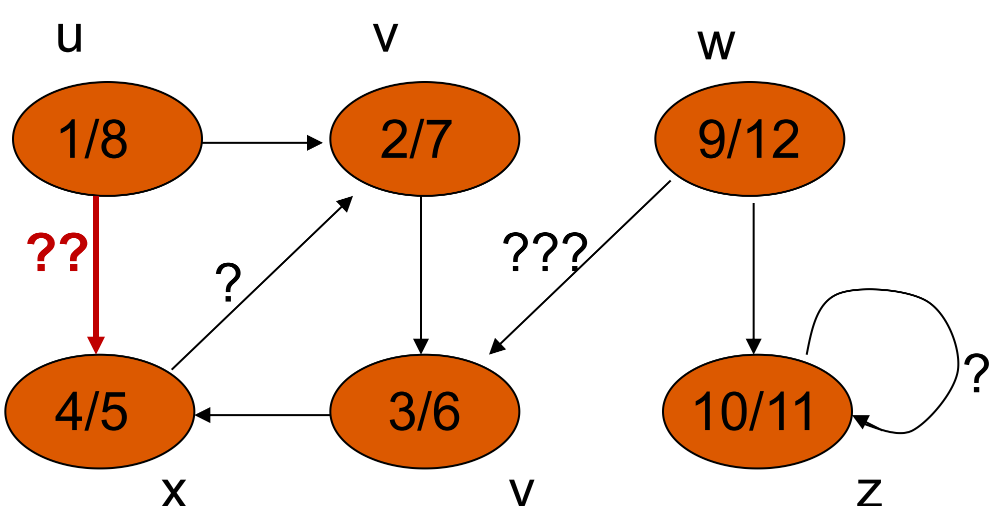
1. Tree edges →.
2. Back edges →?.
3. Forward edges →??.
4. Cross edges →??? (all other edges except the first 3 types)



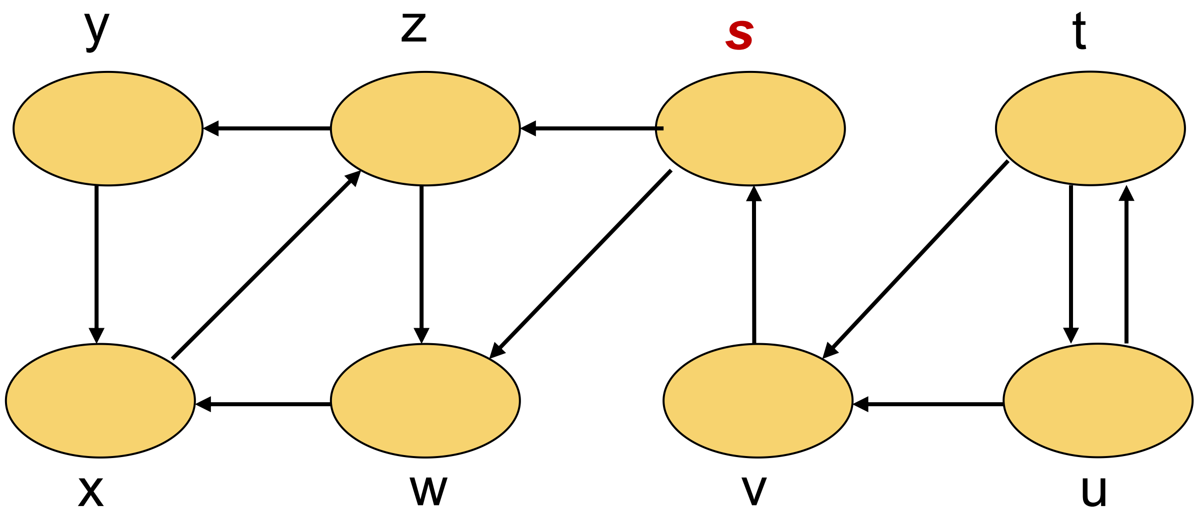
Depth-First Search

How do we make use of forward edges in network communication?

1. Tree edges →.
2. Back edges →?.
3. Forward edges →??.
4. Cross edges →??? (all other edges except the first 3 types)

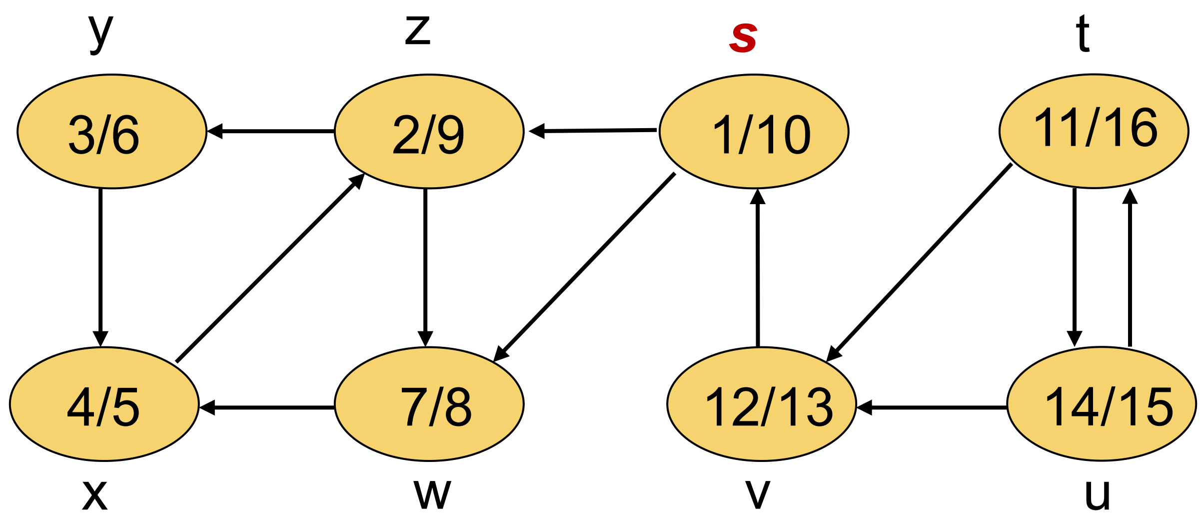


DFS搜索结果?

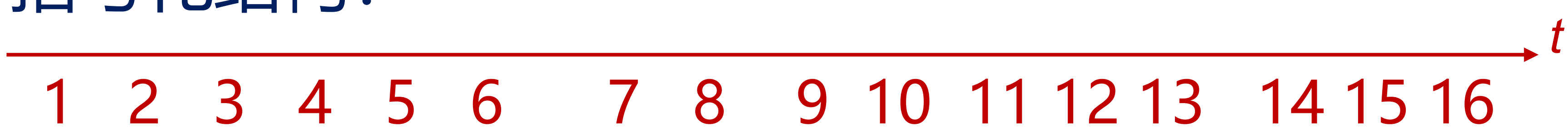


括号化结构?

DFS搜索结果:



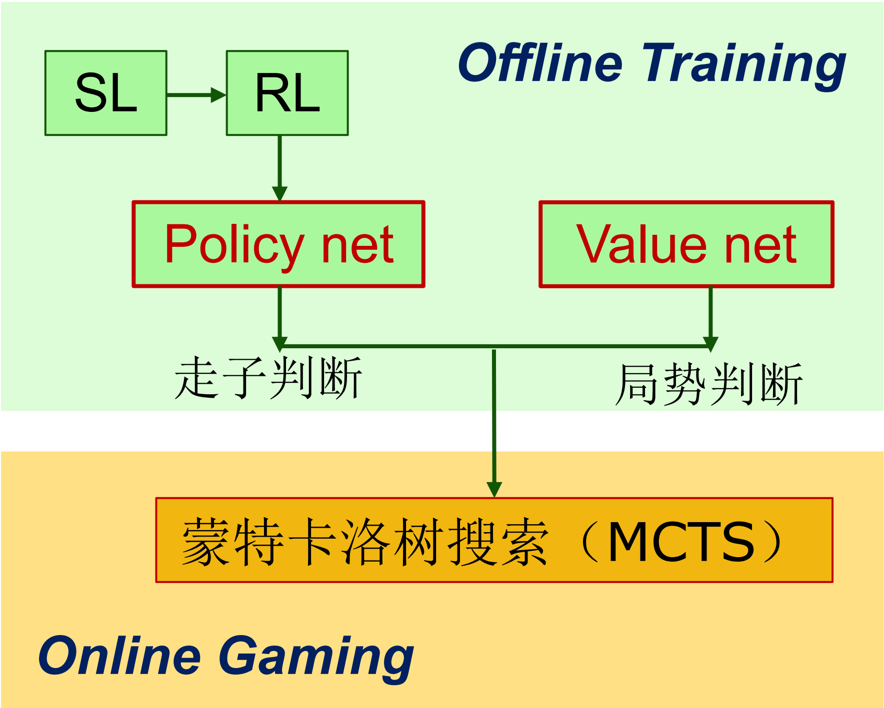
括号化结构:



(s (z (y (x, x) y) (w, w) z) s) (t (v, v) (u, u) t)

理解：递归调用/堆栈操作（push→pop）

拓展：AI算法AlphaGo为何能够战胜人类?



AlphaGo工作原理示意图

任何完全信息博弈都是一种搜索。搜索复杂度取决于搜索空间的宽度和深度。

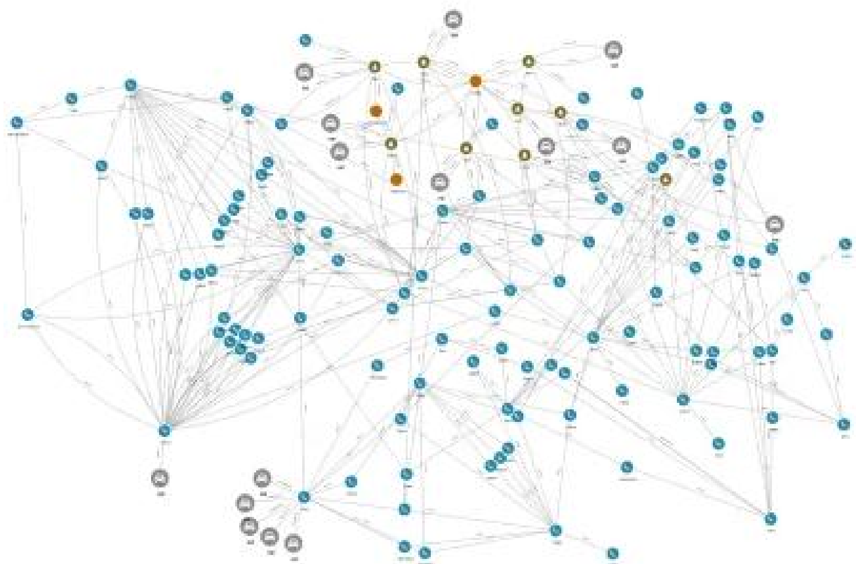
围棋：宽度约为250，深度约为150，总搜索空间约为250^{150}。

- Policy net（策略网络）：减少搜索宽度
- Value net（价值网络）：减少搜索深度

图注：

- SL（Supervised Learning，监督学习）：模仿人类
- RL（Reinforcement Learning，强化学习）：自我进化

拓展：图的应用——图谱



- 知识图谱、社交图谱、数据相关性图谱等
- 表示数据之间的关系，如相似性、访问相关性等（挖掘深层关系）
- 数据处理→元数据处理（提高效率、降低成本）

Thank You!

Q&A