数据结构与算法

DATA STRUCTURE

第一十九讲 图

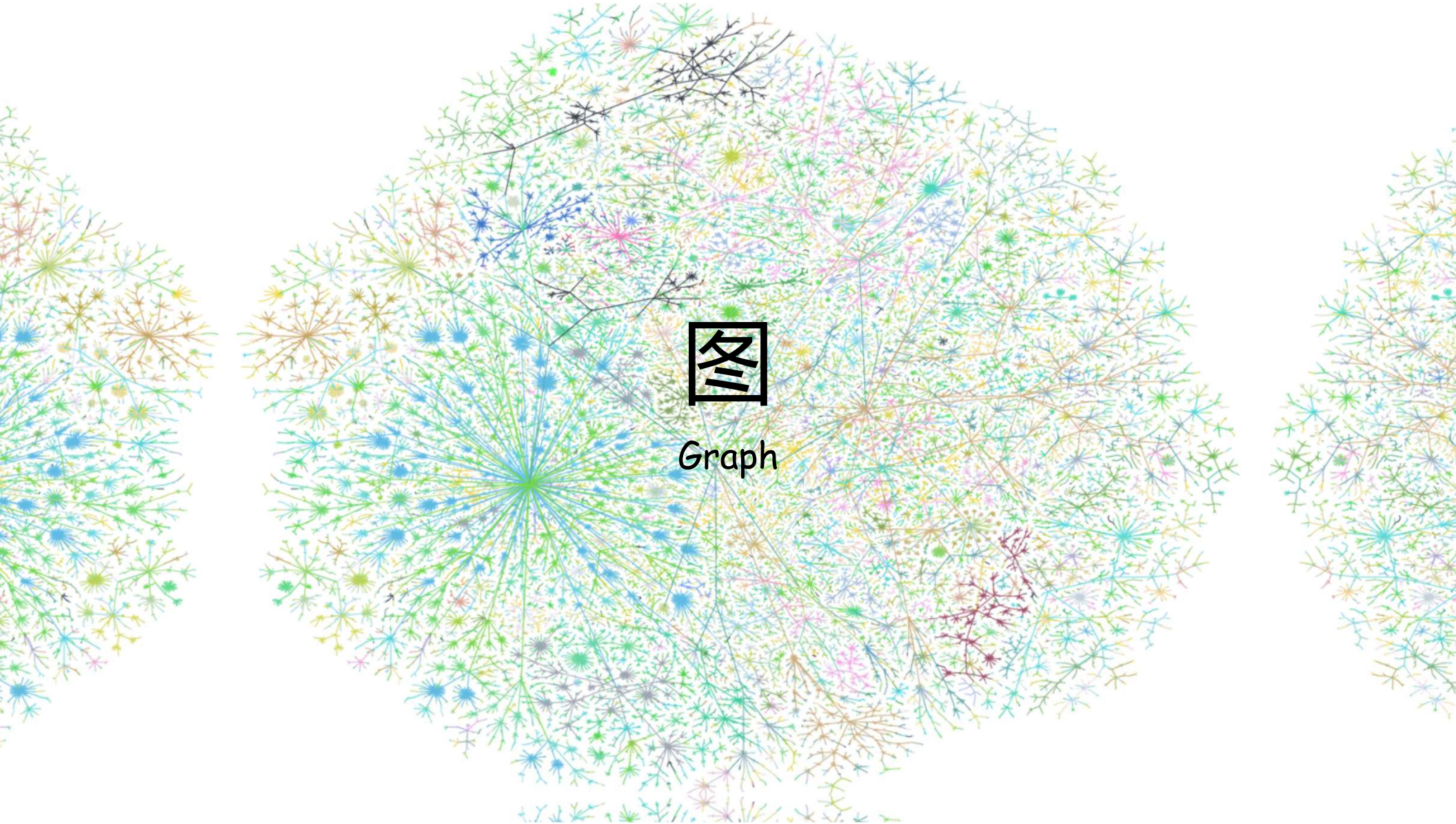
胡浩栋

信息管理与工程学院

2018 - 2019 第一学期

课堂内容

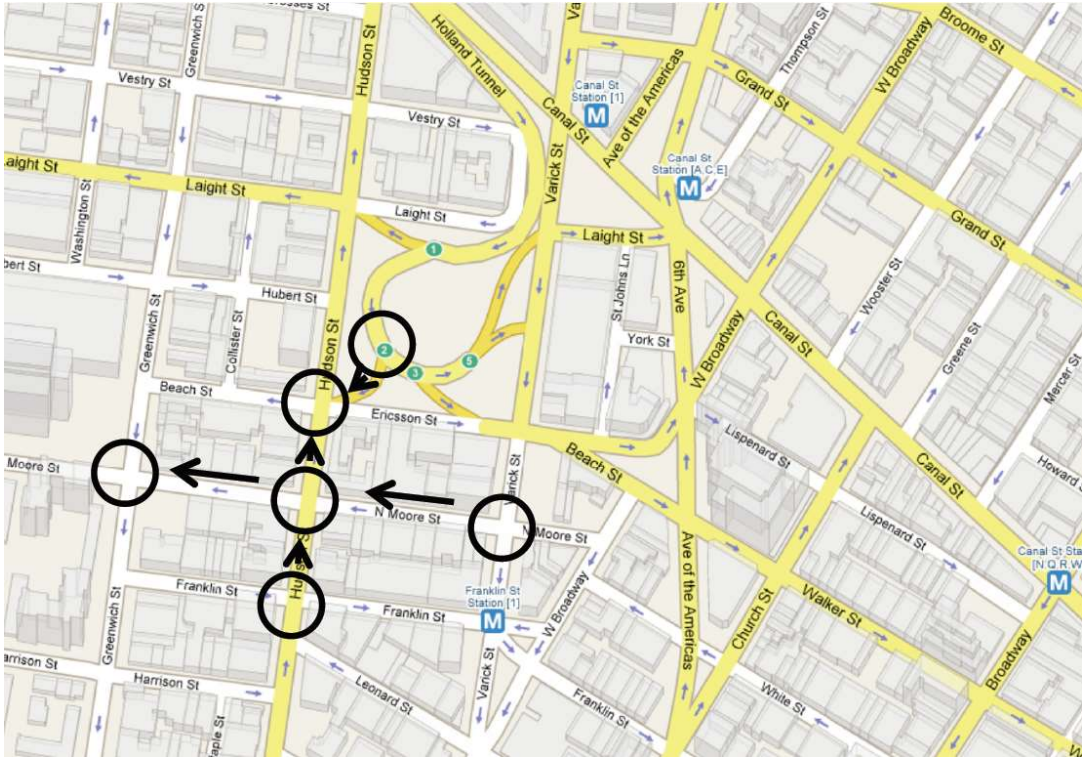
- 图的概念
- DFS
- BFS



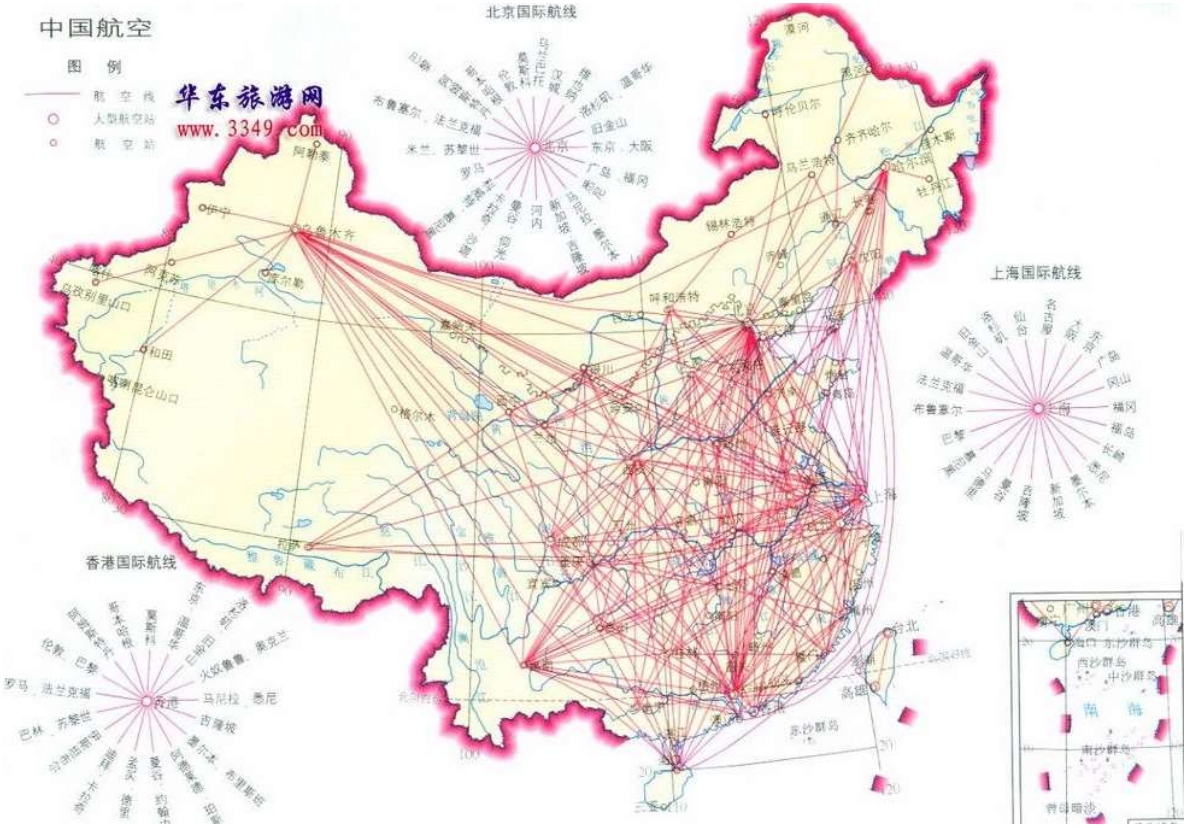
图

Graph

车导航地图


Vestry St
Greenwich St
Hudson St
Holland Tunnel
Canal St
Varick St
Ave of the Americas
Canal St Station [1]
Canal St Station [A,C,E]
Grand St
Thompson St
W Broadway
Broome St
Laight St
Washington St
Hubert St
Collister St
St Johns Ln
6th Ave
York St
Wooster St
Greene St
Mercer St
Beach St
Ericsson St
Lispenard St
Moore St
N Moore St
Franklin St
Franklin St Station [1]
Harrison St
Leonard St
Church St
Walker St
White St
Broadway
Canal St Station [N,Q,R,W]

飞行路线图



全球商品运输图

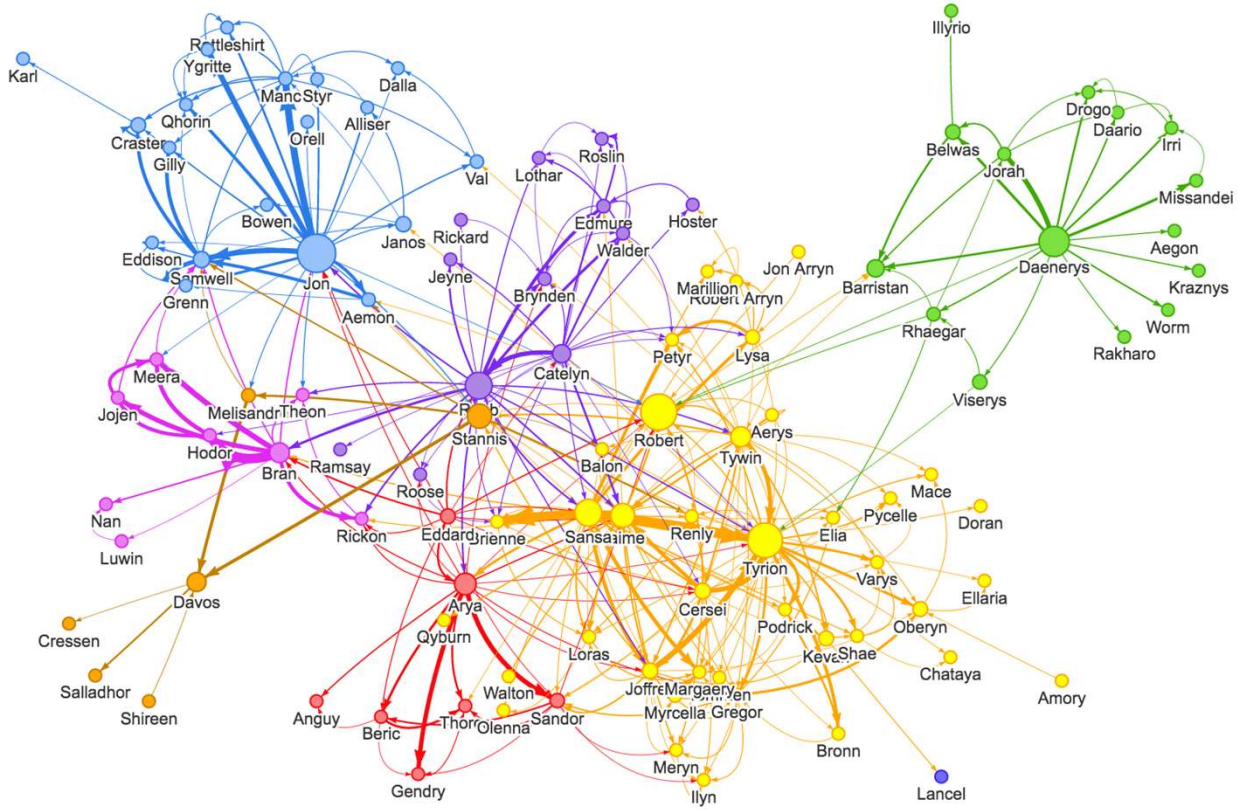
Soybeans

Water

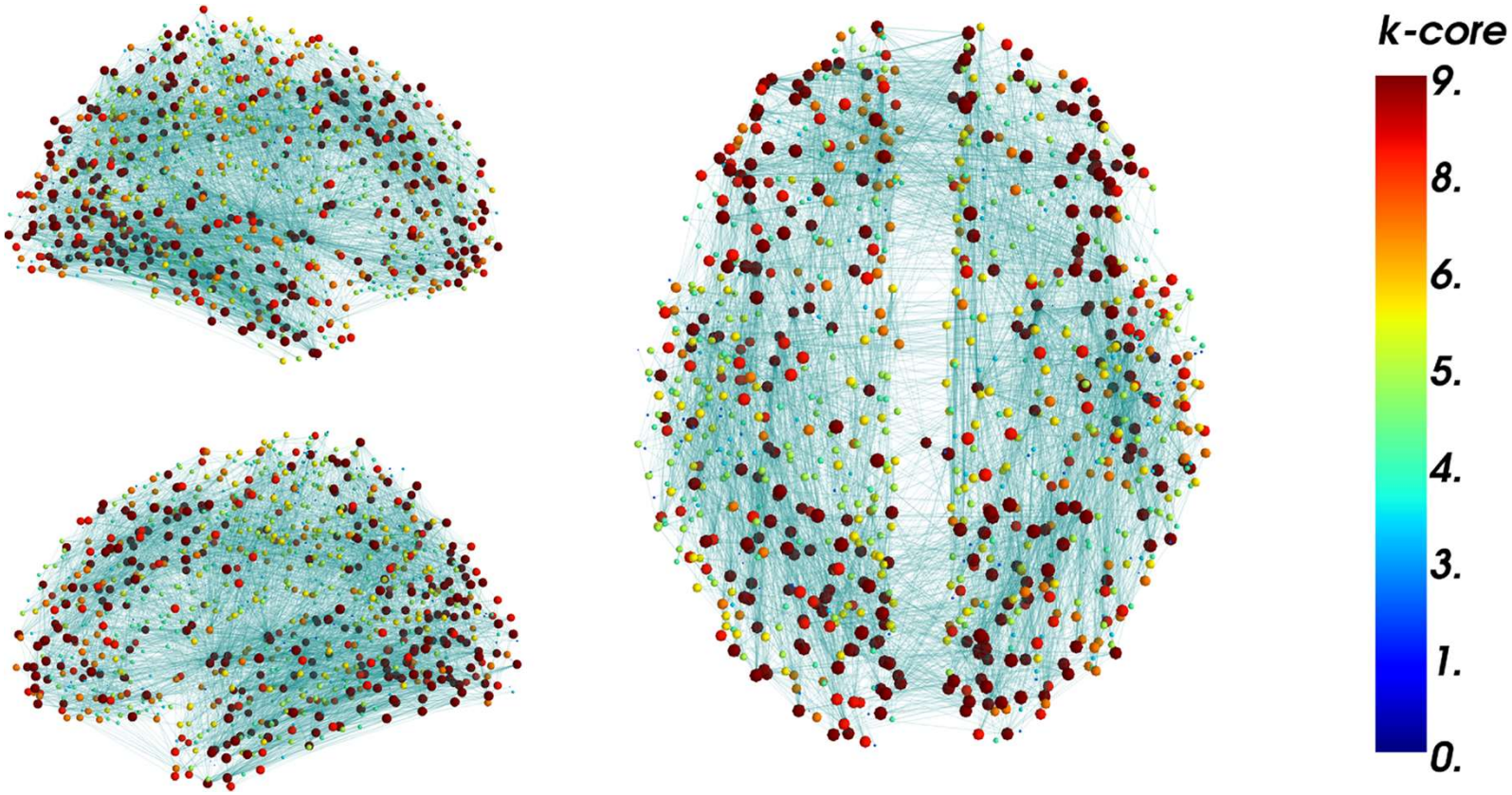
社交网络图

关系图

Game of Thrones 角色关系

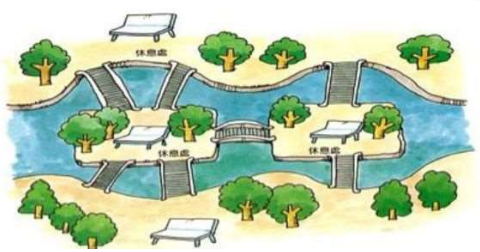


大脑神经元连接图



图

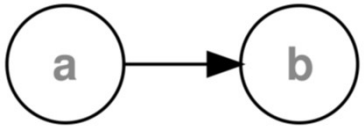
- 有各种各样图
- 绝大部分都可以抽象成点与点之间的关系
- 最早来源于欧拉的七桥问题



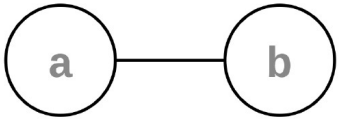
图的定义

- 图是一种数学结构，通过点、边的方式对事物之间关系的抽象。
  - 顶点集合Vertices
  - 边集合Edges
  - 图就是 $G = (V, E)$

- 根据边的方向分为
  - 有向图



  - 无向图



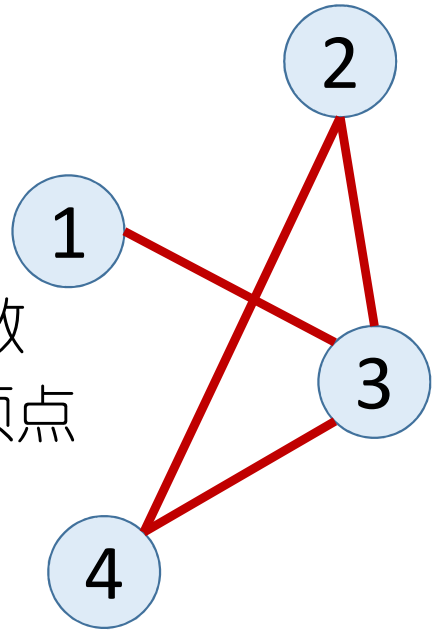

```
struct Node
{
    int id;
    int key;
    std::vector<Edge*> arcs;

    bool active;
}
```

```
struct Edge
{
    int id;
    int weight;
    Node *U;
    Node *V;
    bool active;
}
```

```
class MyGraph
{
private:
    std::vector<Node *> _nodes;
    std::vector<Edge *> _edges;
};
```

无向图

- 图$G = (V, E)$中的边没有方向
- 顶点的度(degree)：与它相连的边的条数
- 邻接顶点(neighbor)：与它有边相连的顶点

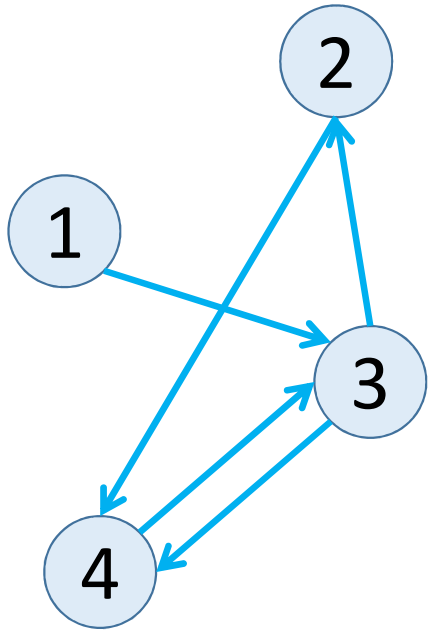


$G = (V, E)$

- 例子
  - V = {1,2,3,4}
  - E = { {1,3}, {2,4}, {3,4}, {2,3} }
  - 顶点4的度是2
  - 顶点4的邻接顶点是顶点2和顶点3

有向图

- 图$G = (V, E)$中的边集合E是有方向
- 顶点v的入度：以v为终点的有向边条数
- 顶点v的出度：以v为起点的有向边条数
- 邻接顶点neighbor：与它有边相连的顶点

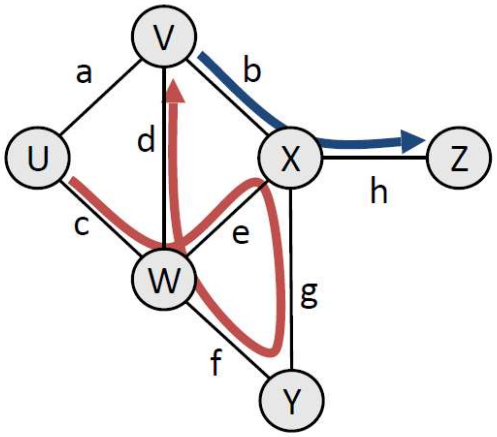


$G = (V, E)$

- 例子
  - V = {1,2,3,4}
  - E = { (1,3), (2,4), (3,4), (4,3), (3,2) }
  - 顶点4的入度是2
  - 顶点4的出度是1
  - 顶点4的入边的邻接顶点是2，3
  - 顶点4的出边的邻接顶点是3

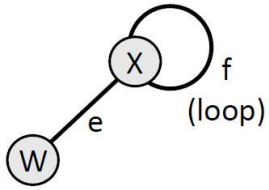
路径path

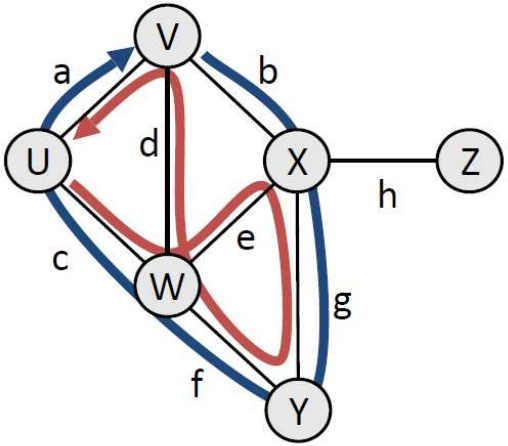
- 顶点u 到v的路径就是边的有序序列，沿着这个序列可以从u到v
- 既可以用顶点表示，也可用边表示
  - 顶点表示：{U, W, X, Y, W, V}
  - 边表示：　{c, e, g, f, d}
- 路径长度就是序列中边的数目



Cycle和loop

- 环路（cycle）：一条路径的起点和终点是同一节点
- 无环路图（acyclic graph）：图里不包含任何cycle
- 自环（loop）：一条边的起点和终点是同一个节点。一般图不允许有自环



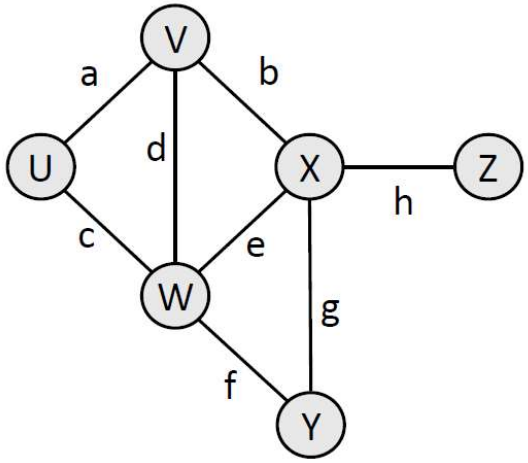


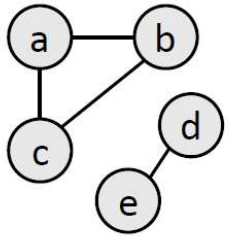
可达性和连通性

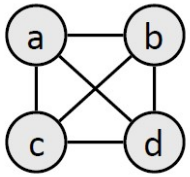
- 可达的reachable：顶点V到顶点Y是可达的，如果他们之间有条路径

- 连通的connected：如果两个顶点之间是可达的

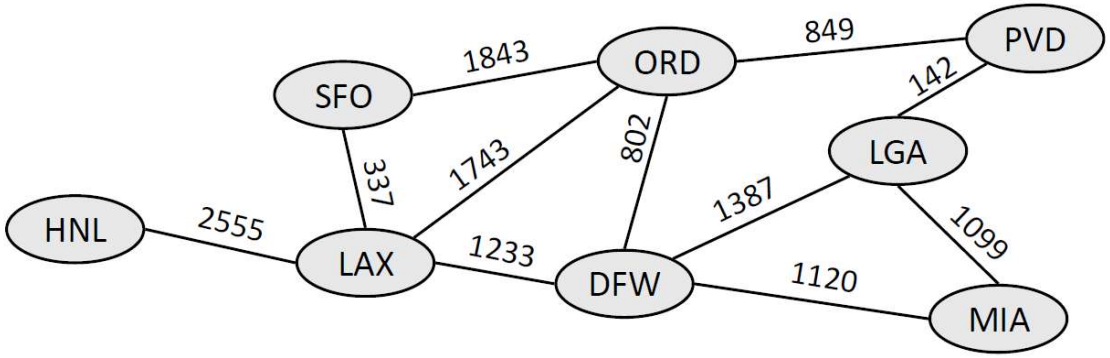
- 完全的complete：如果图里每个顶点都有边连接其余顶点







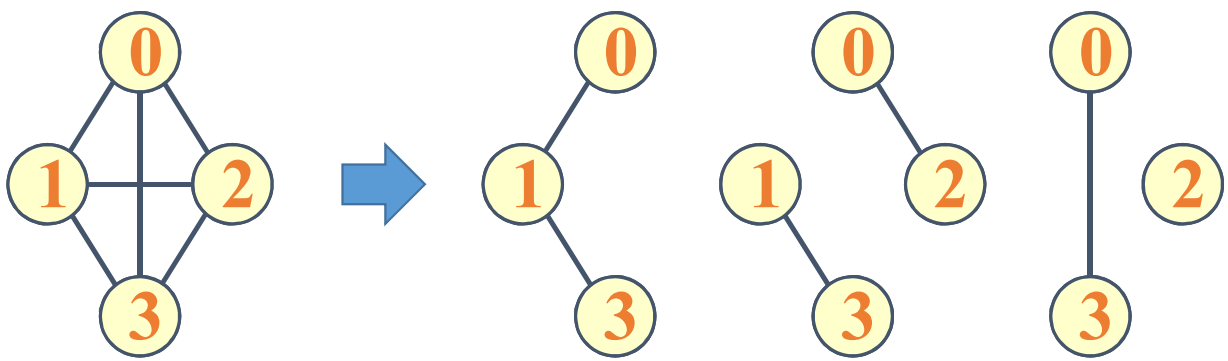
加权图



图的边具有与它相关的数，称之为权weight

子图

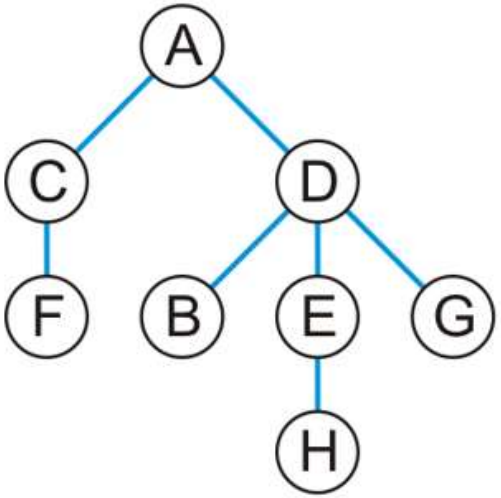
- 设有两个图$G=(V,E)$和$G'=(V',E')$。若$V' \in V$且$E' \in E$，则称图G'图G的子图。



树

- 树是一种特殊的图
  - 选一个节点作为根节点
  - 根节点的邻接顶点为子节点
  - 递归定义以子节点为根节点为子树

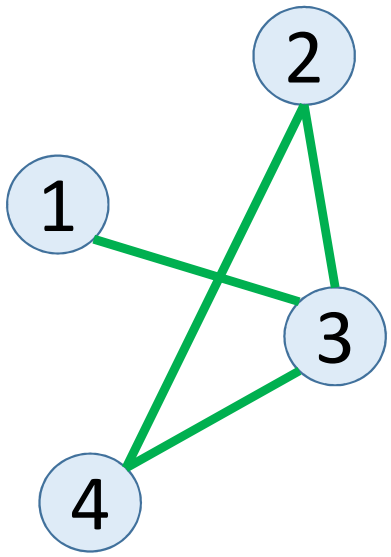
  - 只有一个父节点
  - 不允许有环路



图的表达

- 邻近矩阵

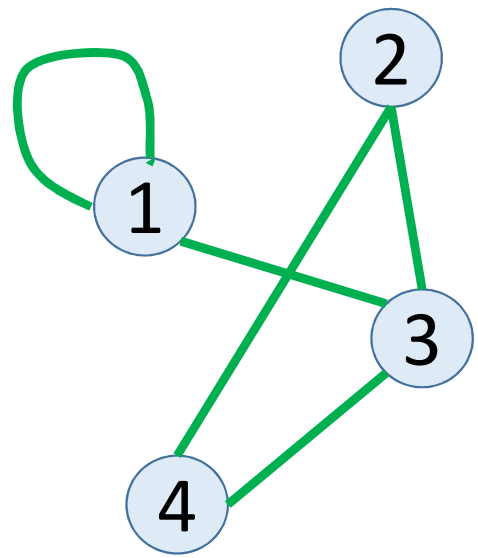
$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 0 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 \\
4 & 0 & 1 & 1 & 0
\end{array}
$$



图的表达

- 邻近矩阵

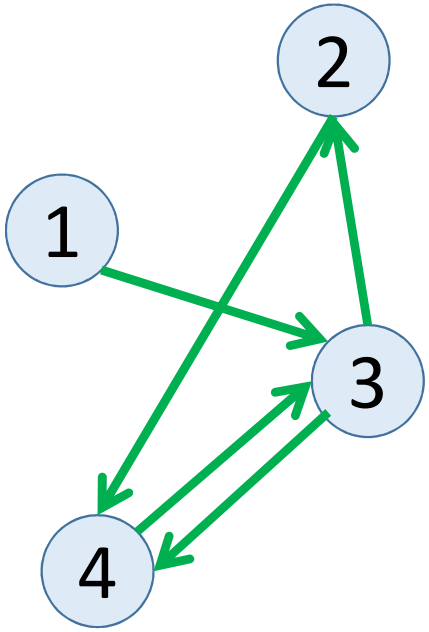
$$
\begin{array}{c|cccc}
 & 1 & 2 & 3 & 4 \\
\hline
1 & 1 & 0 & 1 & 0 \\
2 & 0 & 0 & 1 & 1 \\
3 & 1 & 1 & 0 & 1 \\
4 & 0 & 1 & 1 & 0
\end{array}
$$



图的表达

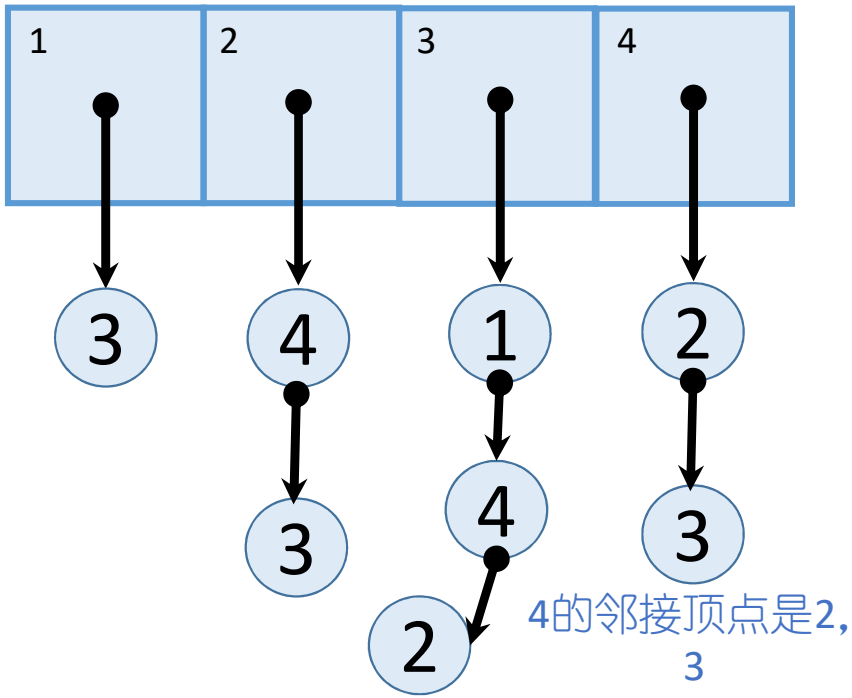
- 邻近矩阵—有向图

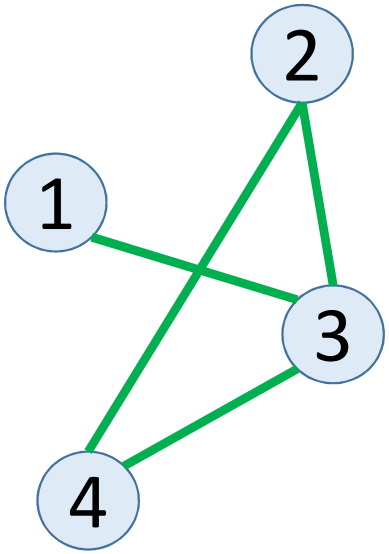
$$
\begin{array}{cc}
 & \text{终点} \\
 & \begin{array}{cccc} 1 & 2 & 3 & 4 \end{array} \\
\text{起点}\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \end{array} & \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\end{array}
$$



图的表达

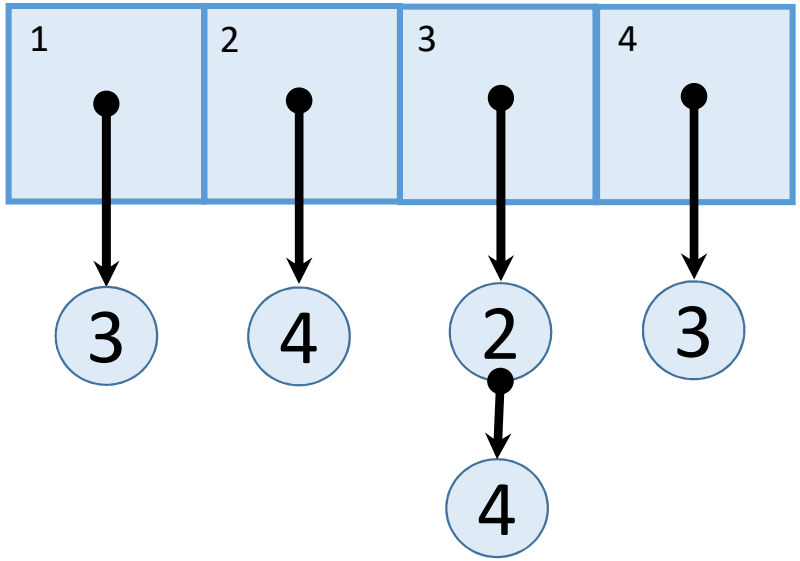
- 链表

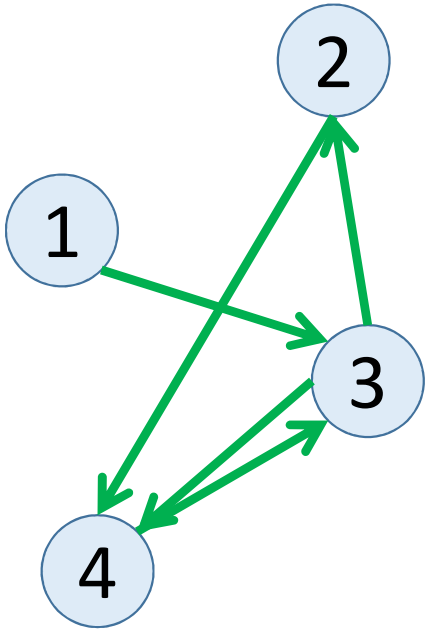




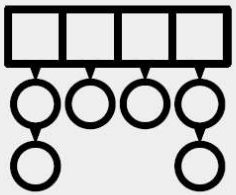
图的表达

- 链表





Trade-off

For G = (V, E)	$\begin{bmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix}$	
Edge Membership Is e = {u, v} in E?	$O(1)$	$O(\deg(u))$ **or** $O(\deg(v))$
Neighbor Query What are the neighbors of u?	$O(\lvert V \rvert)$	$O(\deg(v))$
Space requirements	$O(\lvert V \rvert^2)$	$O(\lvert V \rvert + \lvert E \rvert)$

图数据结构

- 还可以存的信息：
  - 属性：点/边ID，顶点值，边权重，是不是active
- 常用的操作，比如查询
  - 某条边是不是属于边集合E?
  - 顶点v的邻接顶点有那些?

点/边的结构体

```
struct Node
{
    int id;
    int key;
    // outgoing edges
    std::vector<Edge*> arcs;
    // incoming edges
    std::vector<Edge*> inArcs;

    bool active;

    Node(int value = 0)
    {
        id = getId();
        key = value;
        active = true;
    }
    int getId()
    {
        static int s_id = 0;
        return s_id++;
    }
};
```

```
struct Edge
{
    int id;
    int weight;
    Node *U;
    Node *V;
    bool active;

    Edge(Node *src, Node *dst, int value = 0)
    {
        id = getId();
        weight = value;
        U = src;
        V = dst;
        active = true;
    }
    int getId()
    {
        static int s_id = 0;
        return s_id++;
    }
};
```

用getId()自动产生连续id，之后可以用id对应点/边

MyGraph类

- 注意用id对应点/边数组下标，可以很大程度上简化相关算法，不过得保证匹配。
- 比如加点时候，push_back()加到最后一个，然后assert数组大小等于点id+1
- 如果需要删除点，不用直接删除，而是标明不用。这样即避免了数组下标对应的问题，而且可以恢复点。

```
class MyGraph
{
public:
    // build graph with vertices of given size
    MyGraph(int sz);

    // Release node/edge memory
    ~MyGraph();

    void AddEdge(int uid, int vid);
    void PrintGraph();

private:
    // TODO: implement Copy constructor
    MyGraph(const MyGraph &other);
    // TODO: Implement assignment overloading
    MyGraph& operator =(const MyGraph &rhs);
    // a node with id is stored at index, equals to id.
    // When deleting a node, just mark it as not active, instead of deletion.
    std::vector<Node *> _nodes;
    std::vector<Edge *> _edges;
};
```

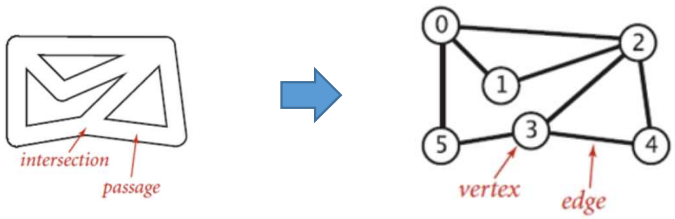
```
MyGraph::MyGraph(int sz)
{
    for (int i = 0; i < sz; i++)
    {
        Node *nd = new Node();
        _nodes.push_back(nd);
        assert(nd->id + 1 == (int)_nodes.size());
    }
}
```

```
void MyGraph::PrintGraph()
{
    for (auto nd : _nodes)
    {
        cout << "Node #" << nd->id << ": ";
        for (auto arc : nd->arcs)
        {
            cout << "#" << arc->V->id << ", ";
        }
        cout << endl;
    }
}
```

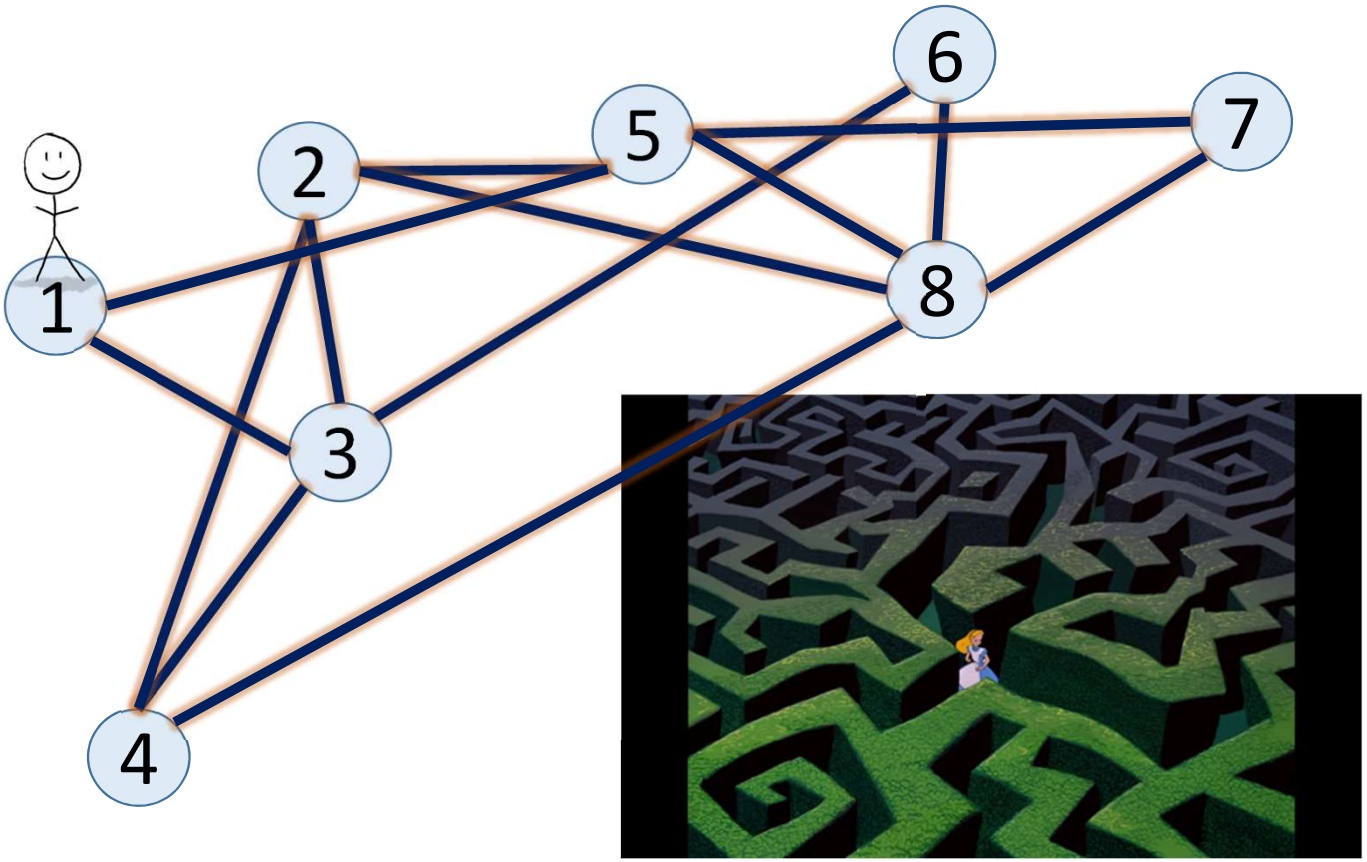
图的遍历

遍历的方法

- 深度优先Depth First Search
  - 递归
  - 栈迭代
- 广度优先Breadth First Search
  - 队列迭代

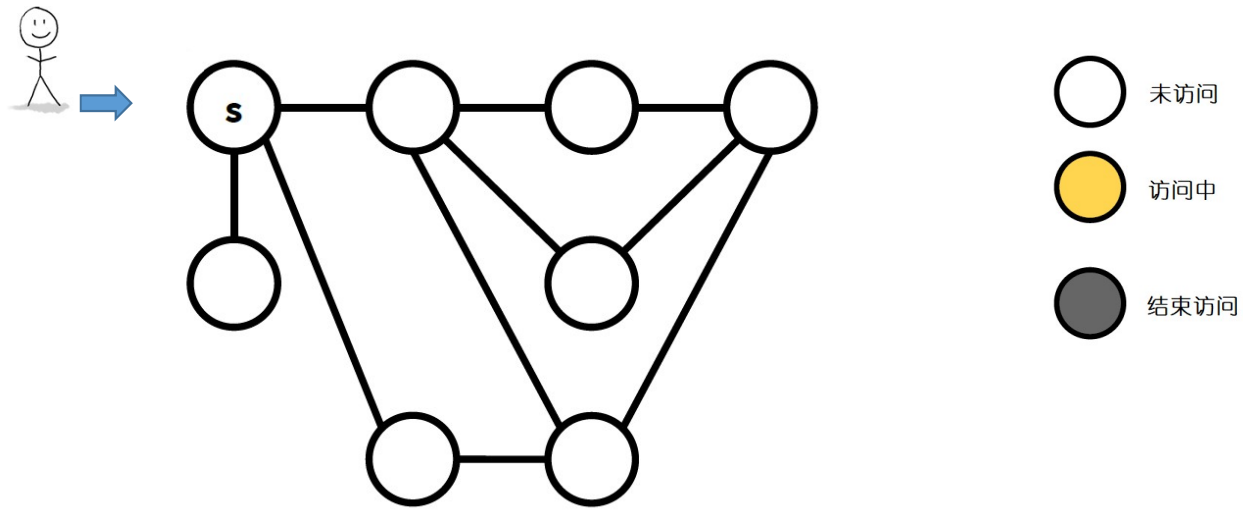


迷宫找路—深度优先



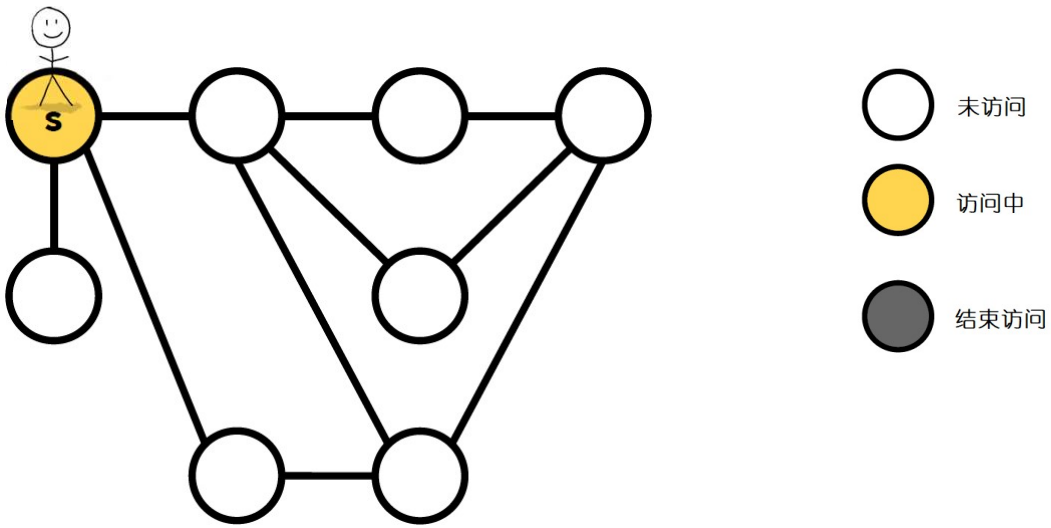
DFS例子

迷宫路径探索



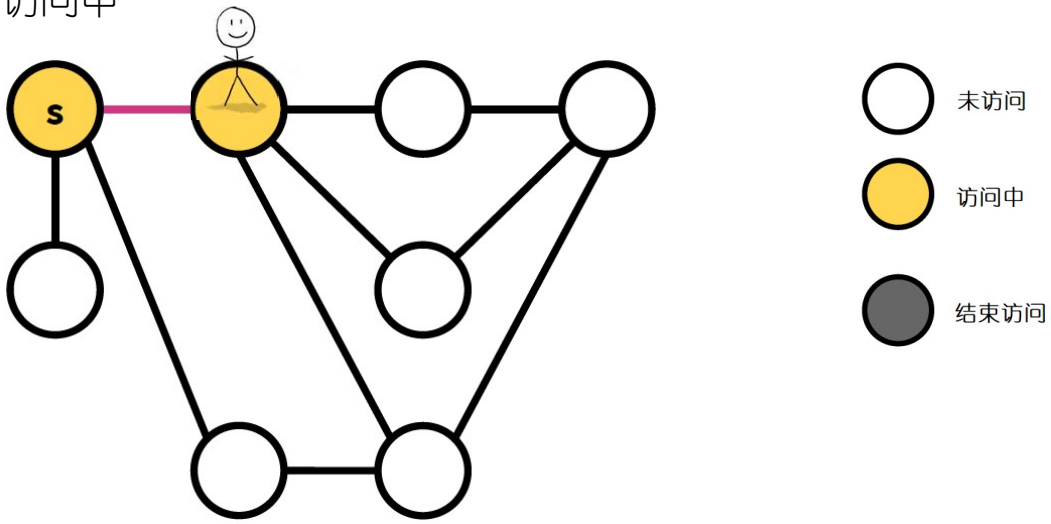
DFS例子

迷宫路径探索，开始访问s



DFS例子

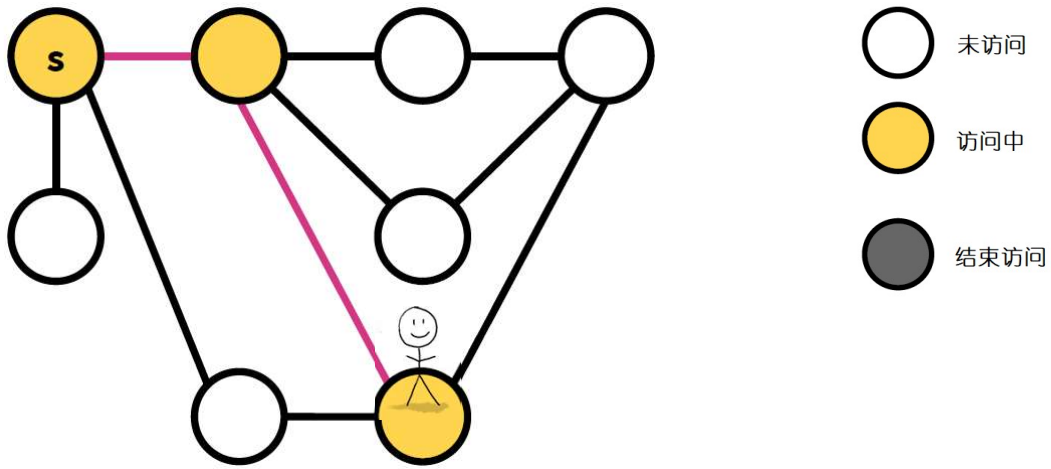
迷宫路径探索
先访问一条边，邻接顶点设为访问中



递归栈：S

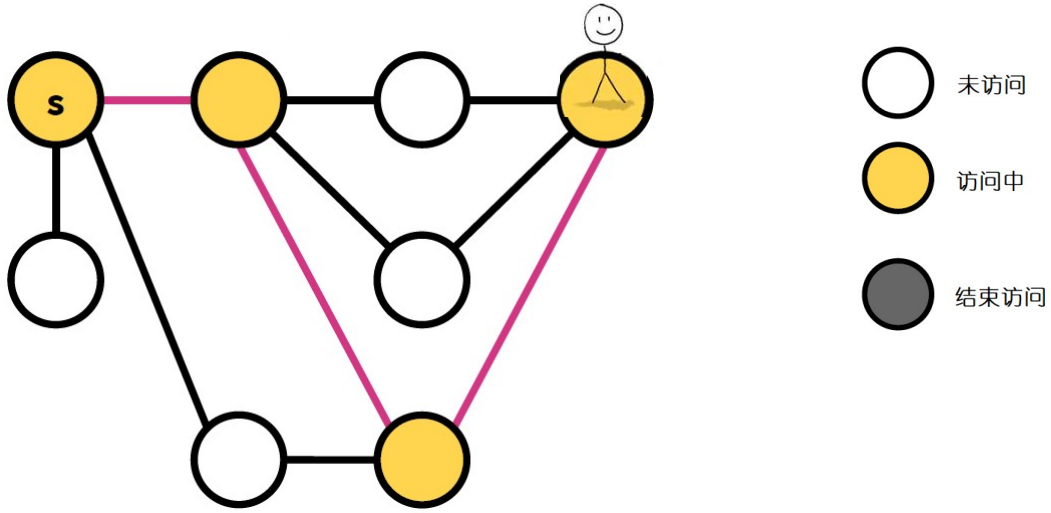
DFS例子

迷宫路径探索
继续访问邻接顶点



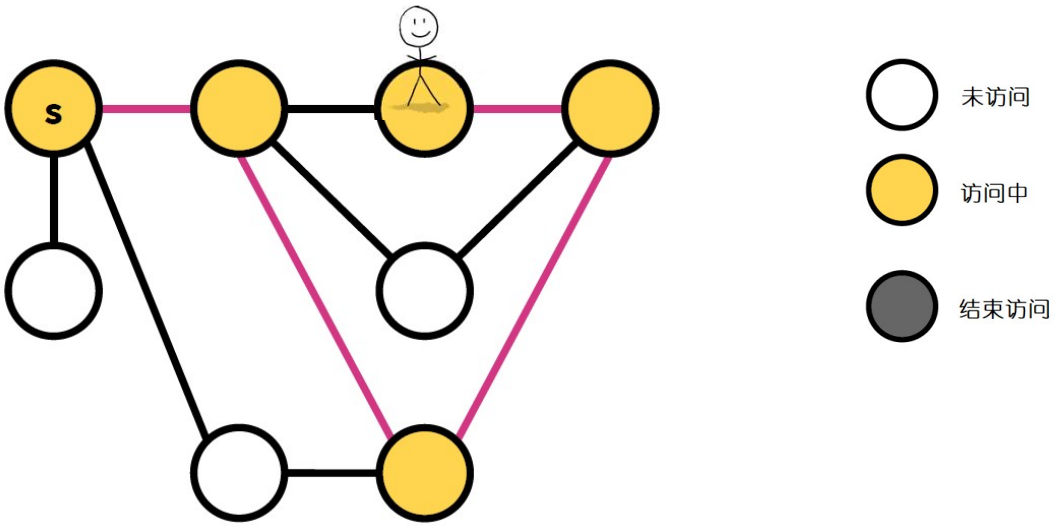
DFS例子

迷宫路径探索
继续访问邻接顶点



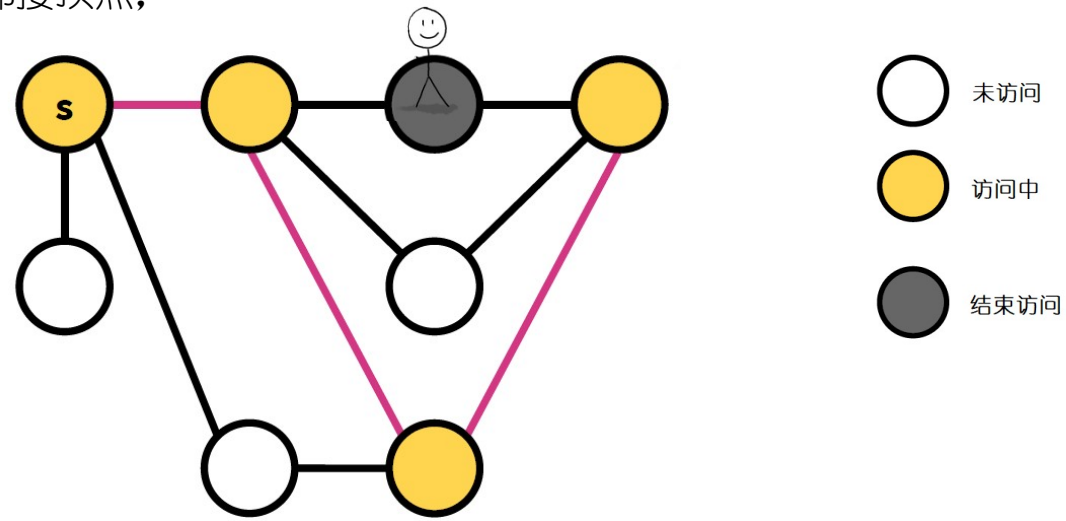
DFS例子

迷宫路径探索
继续访问邻接顶点



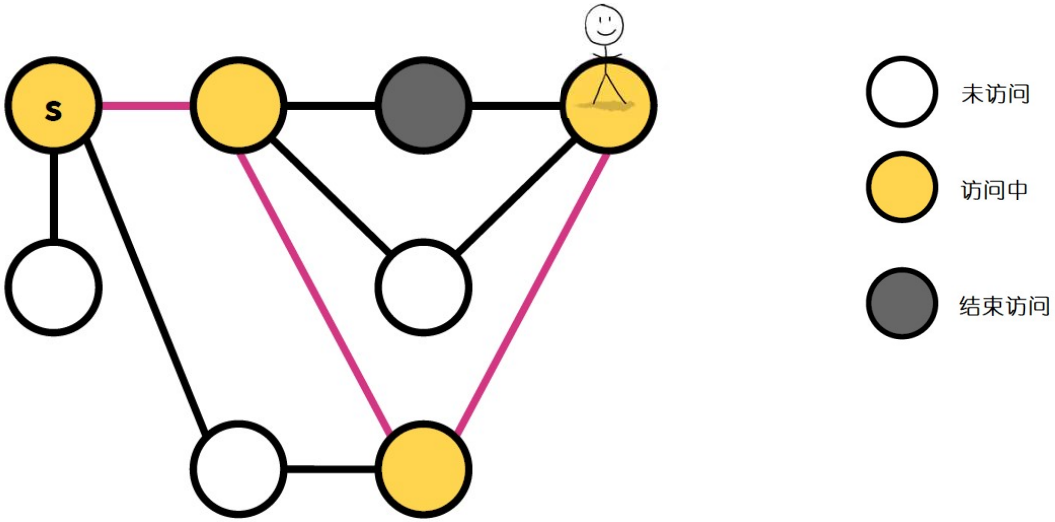
DFS例子

迷宫路径探索
注意当前节点没有未访问的邻接顶点，
所以设置为访问结束



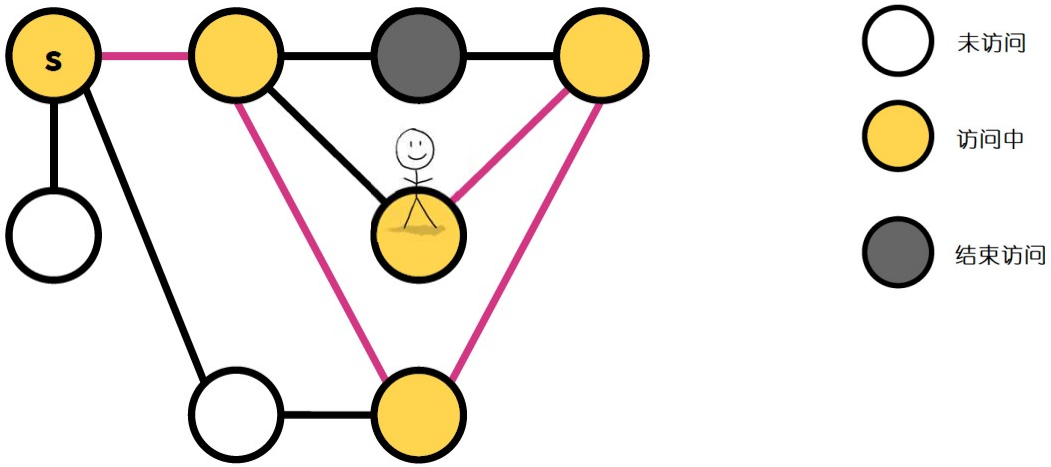
DFS例子

迷宫路径探索
返回后，找未访问节点



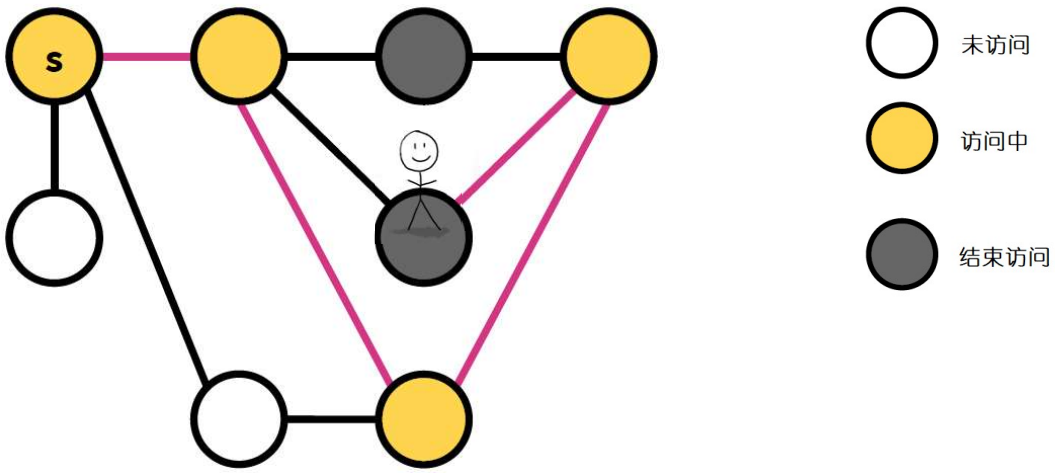
DFS例子

迷宫路径探索
继续访问邻接顶点



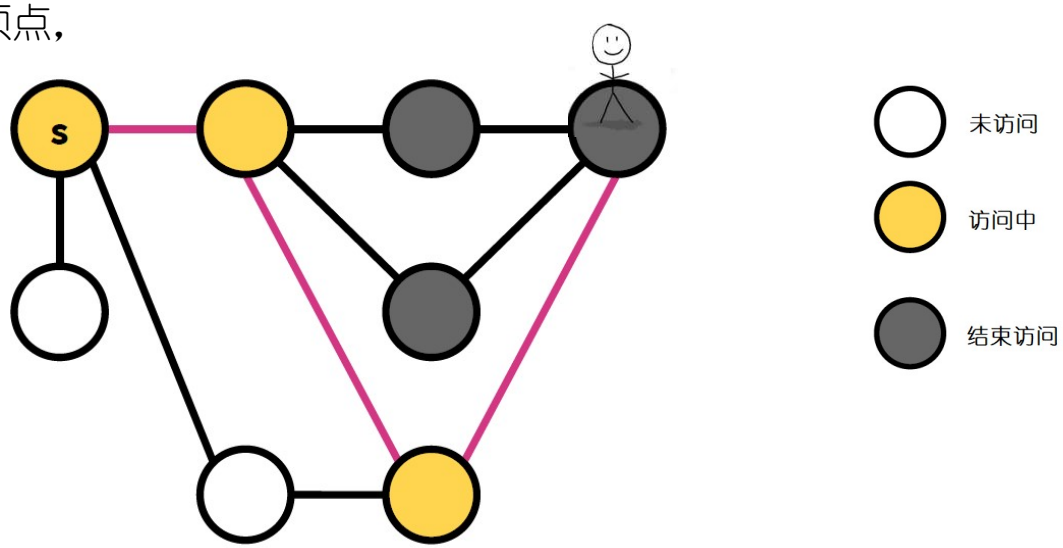
DFS例子

迷宫路径探索
当前节点没有未访问的邻接顶点，
所以设置为访问结束



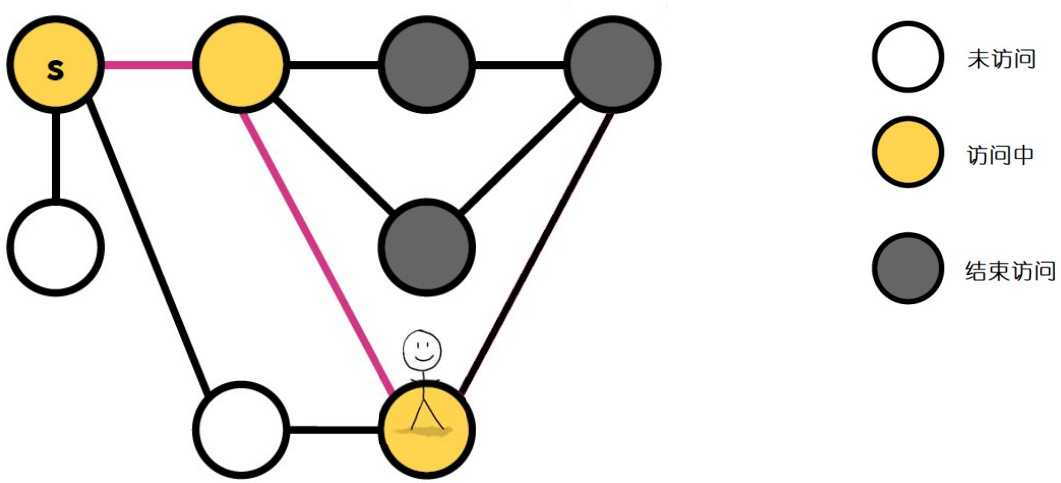
DFS例子

迷宫路径探索
返回后，没有未访问的邻接顶点，
标志为访问结束



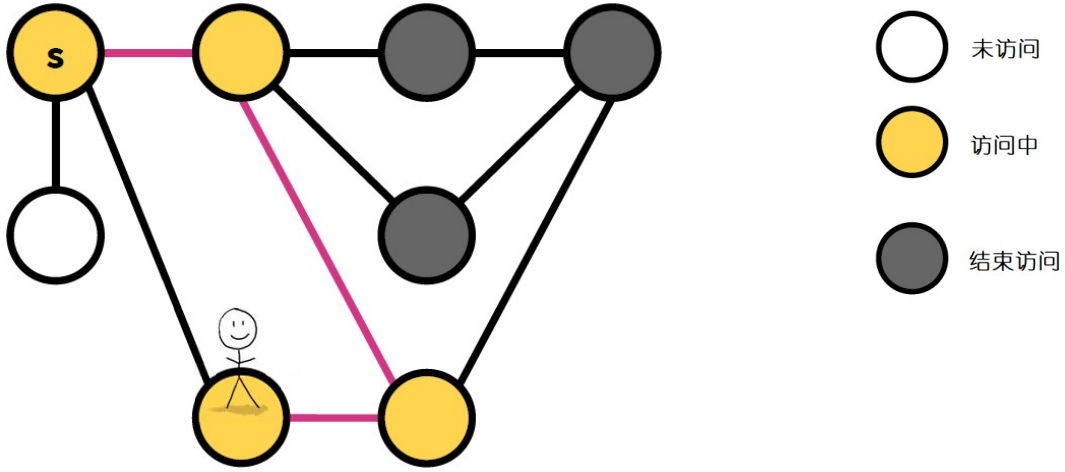
DFS例子

迷宫路径探索
返回后，找未访问节点



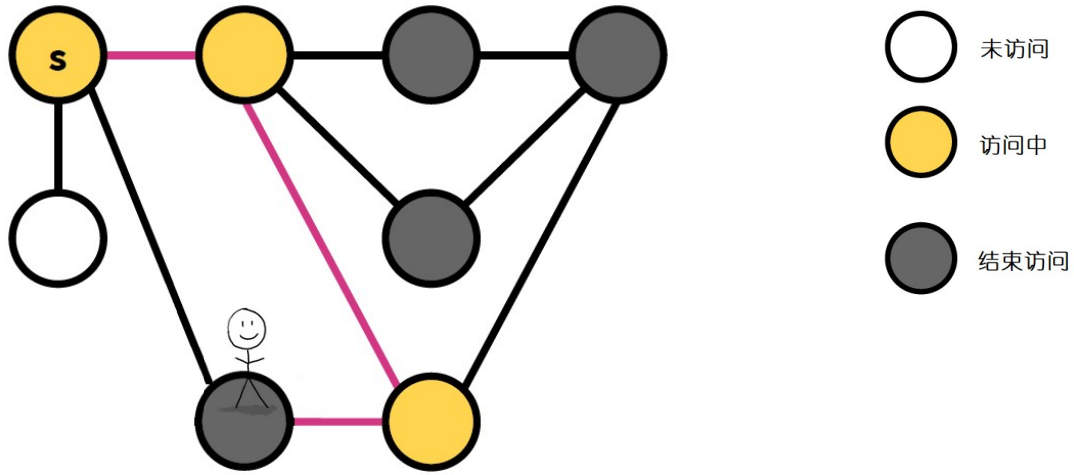
DFS例子

迷宫路径探索
继续访问邻接顶点



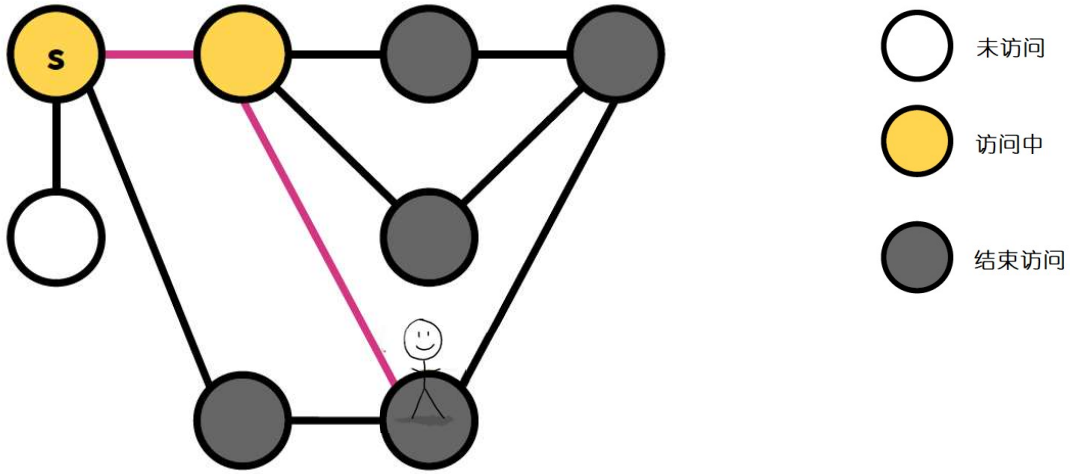
DFS例子

迷宫路径探索
当前节点没有未访问的邻接顶点，
所以设置为访问结束



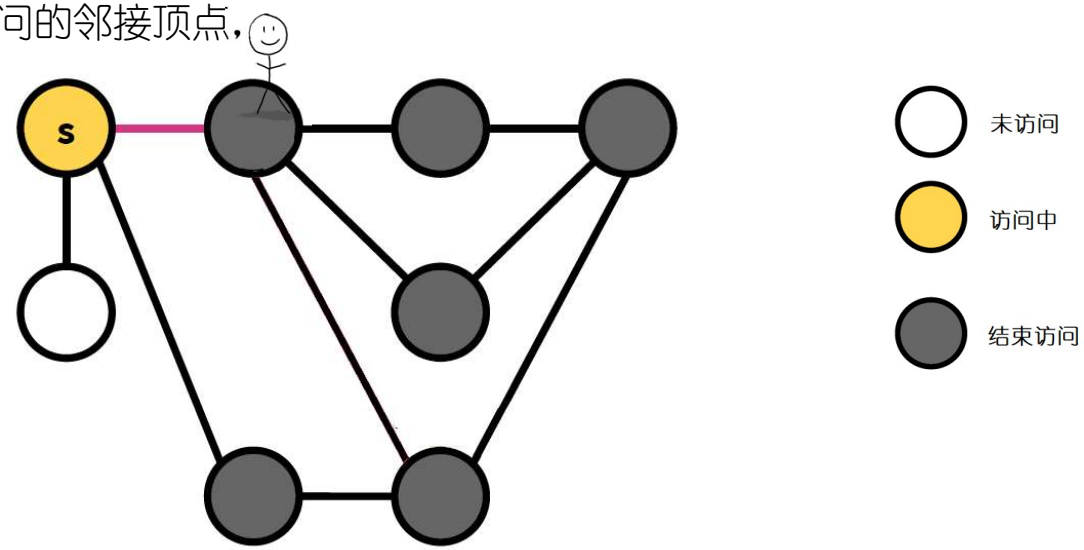
DFS例子

迷宫路径探索
返回后，当前节点没有未访问的邻接顶点，
标志为访问结束



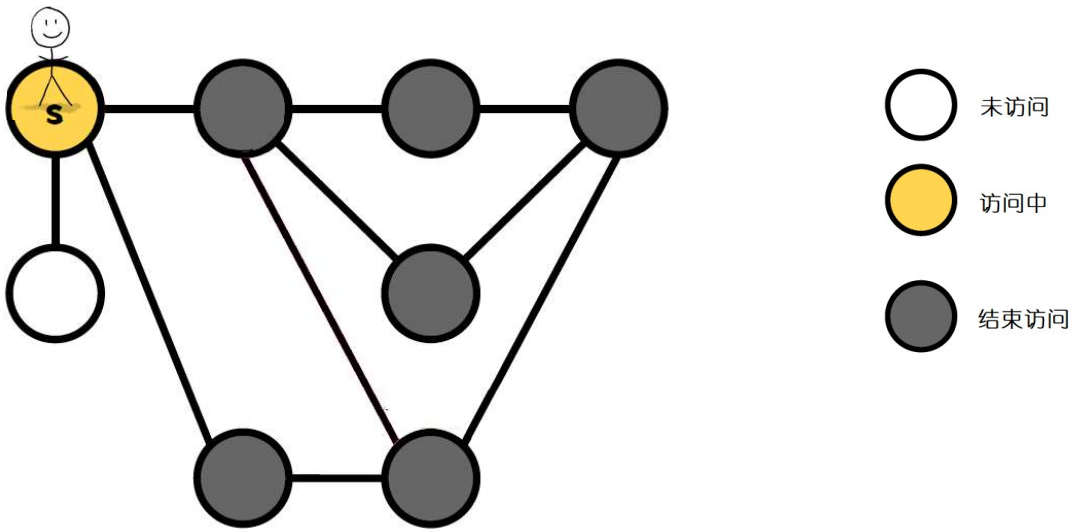
DFS例子

迷宫路径探索
返回后，当前节点没有未访问的邻接顶点，
标志为访问结束



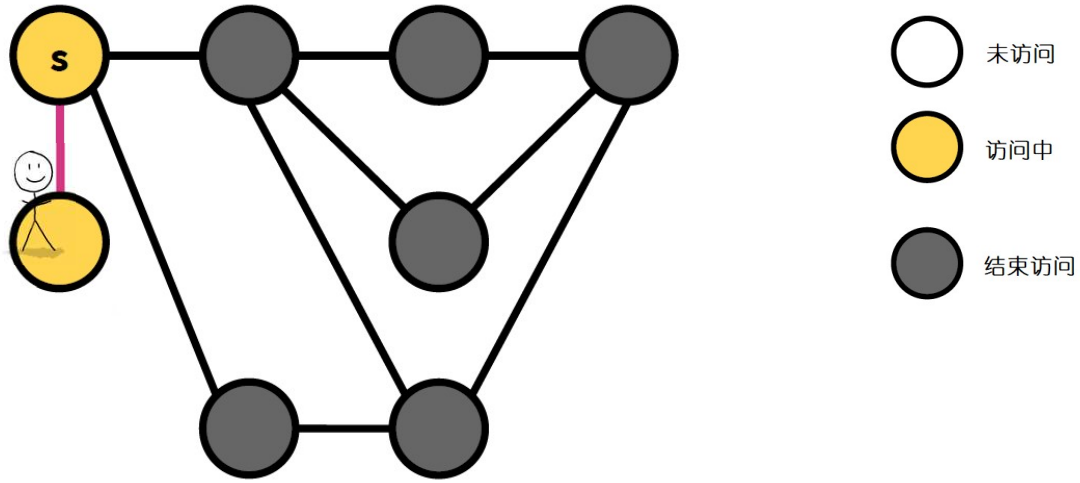
DFS例子

迷宫路径探索
返回后，找未访问节点



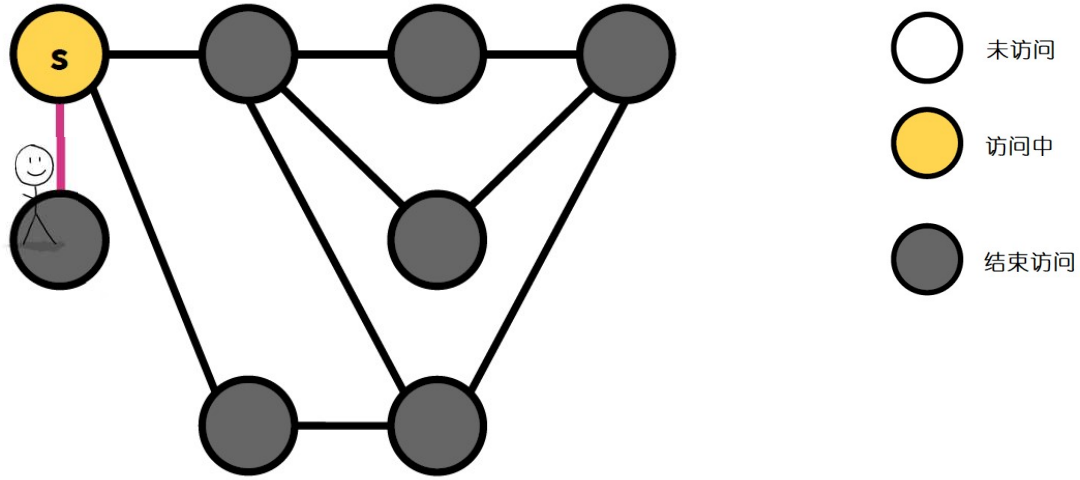
DFS例子

迷宫路径探索
继续访问邻接顶点



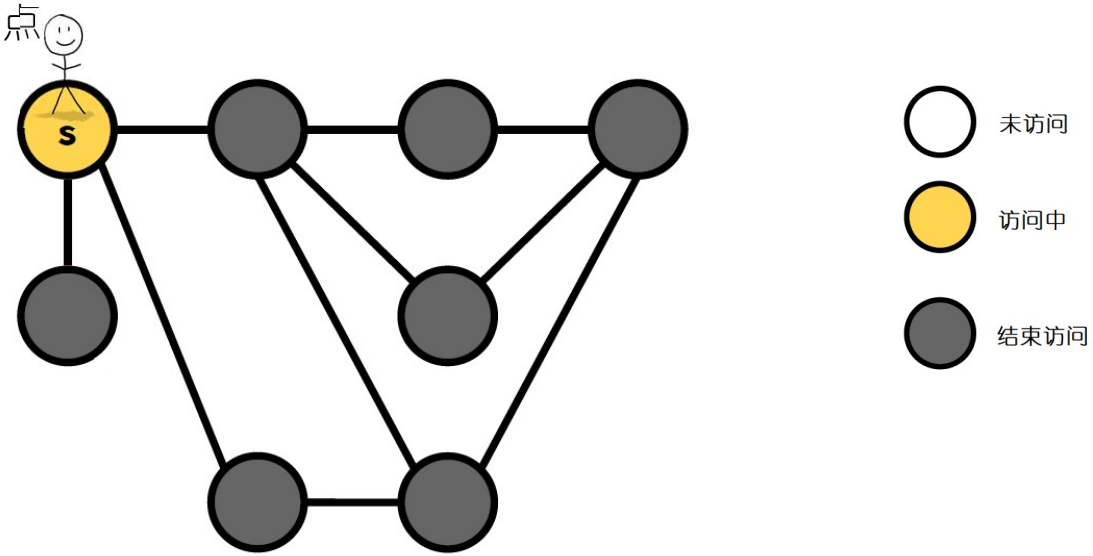
DFS例子

迷宫路径探索
当前节点没有未访问的邻接顶点，
所以设置为访问结束



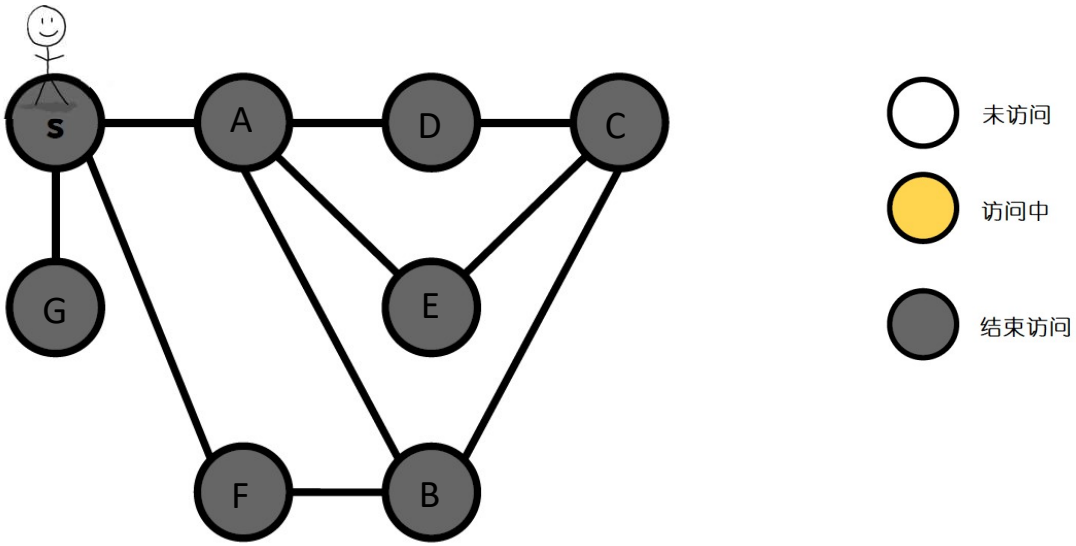
DFS例子

迷宫路径探索
返回后，看有没有未访问节点



DFS例子

迷宫路径探索
当前节点没有未访问的邻接顶点，
所以设置为访问结束



DFS递归算法

DFS(Node u)

u.status =

foreach（v是u的邻接顶点）

if v.status ==

DFS(v)

u.status =

Runtime: O(|V|+|E|)

```
void MyGraph::Dfs(int start)
{
    cout << "\n Start DFS search:" << endl;
    bool *visited = new bool[_nodes.size()]();
    DfsInternal(start, visited);

    cout << endl;
    delete [] visited;
}

void MyGraph::DfsInternal(int uid, bool *visited)
{
    visited[uid] = true;
    cout << "#" << uid << ", ";

    for (auto arc : _nodes.at(uid)->arcs)
    {
        int vid = arc->V->id;
        if (!visited[vid])
        {
            DfsInternal(vid, visited);
        }
    }
}
```

DFS迭代算法

DFS_Iterative(Node u)

stack stk

stk.push(u)

while（S不空）

u = stk.pop()

if u.status == ○

u.status = ●

foreach（v是u的邻接顶点）

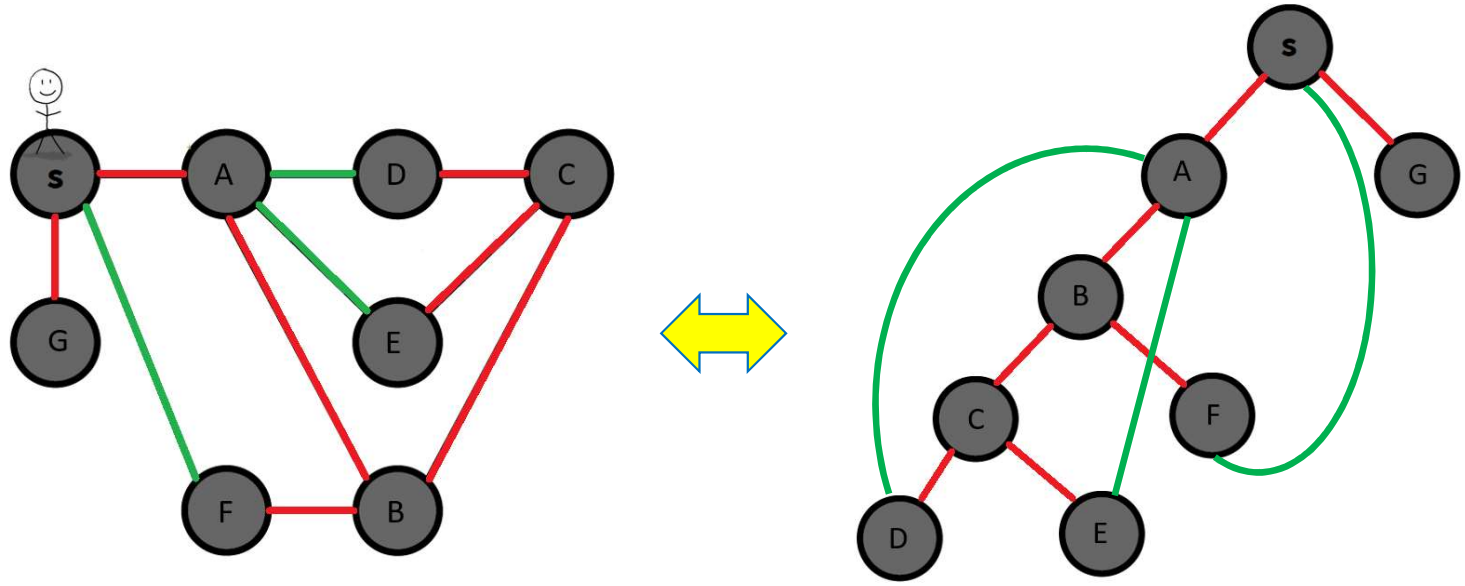
stk.push(v)

Runtime: O(|V|+|E|)

```
void MyGraph::Dfs_Iterative(int start)
{
    cout << "\n Start DFS iterative search:" << endl;
    bool *visited = new bool[_nodes.size()]();

    stack<int> stk;
    stk.push(start);

    while(!stk.empty())
    {
        int uid = _nodes.at(stk.top())->id;
        stk.pop();
        if (!visited[uid])
        {
            visited[uid] = true;
            cout << "#" << uid << ", ";

            for (auto arc : _nodes.at(uid)->arcs)
            {
                stk.push(arc->V->id);
            }
        }
    }

    cout << endl;
    delete [] visited;
}
```

为什么叫depth first

- 算法过程隐含了一个DFS树



DFS算法+始末时间

DFS(Node u, Time currentTime)

u.start = currentTime

currentTime += 1

u.status =

foreach （v是u的邻接顶点）

if v.status ==

currentTime = **DFS**(v, currentTime)

currentTime +=1

u.end = currentTime

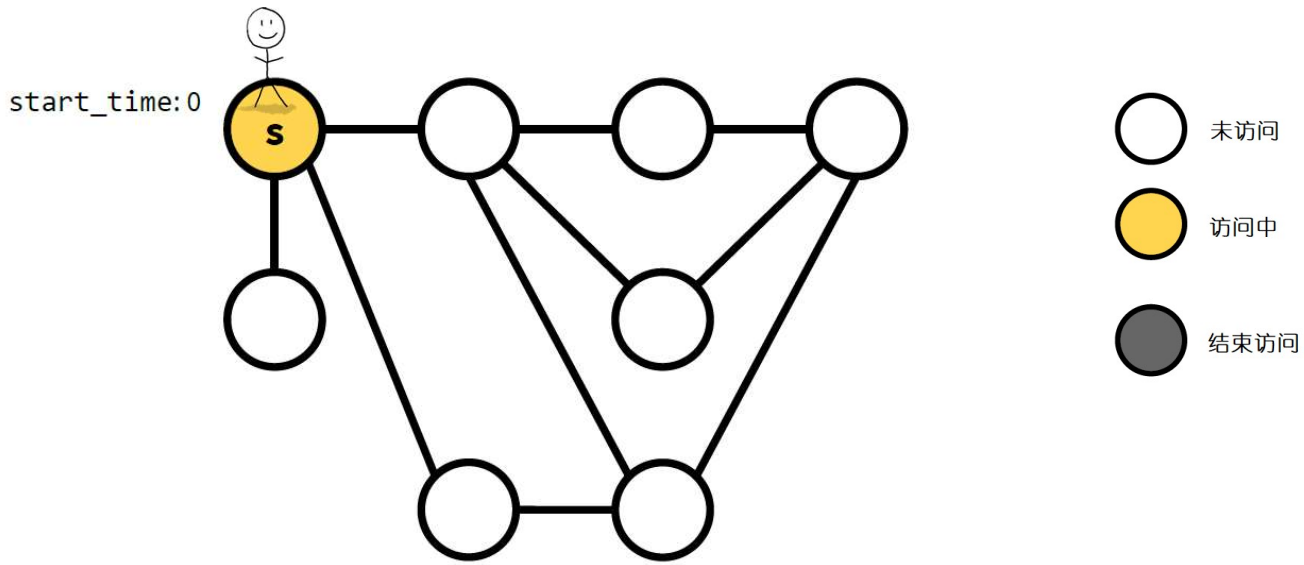
u.status =

return currentTime

```
void MyGraph::DfsTime(int start)
{
    cout << "\n Start DFS Time search:" << endl;
    bool *visited = new bool[_nodes.size()]();
    pair<int, int> *travelTimes = new pair<int, int>[_nodes.size()];

    DfsTimeInteral(start, 0, visited, travelTimes);

    delete [] visited;
    delete [] travelTimes;
}

int MyGraph::DfsTimeInteral(int uid, int currentTime,
                            bool *visited, pair<int, int> *travelTimes)
{
    visited[uid] = true;
    travelTimes[uid].first = currentTime++;
    cout << "#" << uid << ", ";

    for (auto arc : _nodes.at(uid)->arcs)
    {
        int vid = arc->V->id;
        if (!visited[vid])
        {
            currentTime = DfsTimeInteral(vid, currentTime, visited, travelTimes);
            currentTime++;
        }
    }

    travelTimes[uid].second = currentTime;
    return currentTime;
}
```

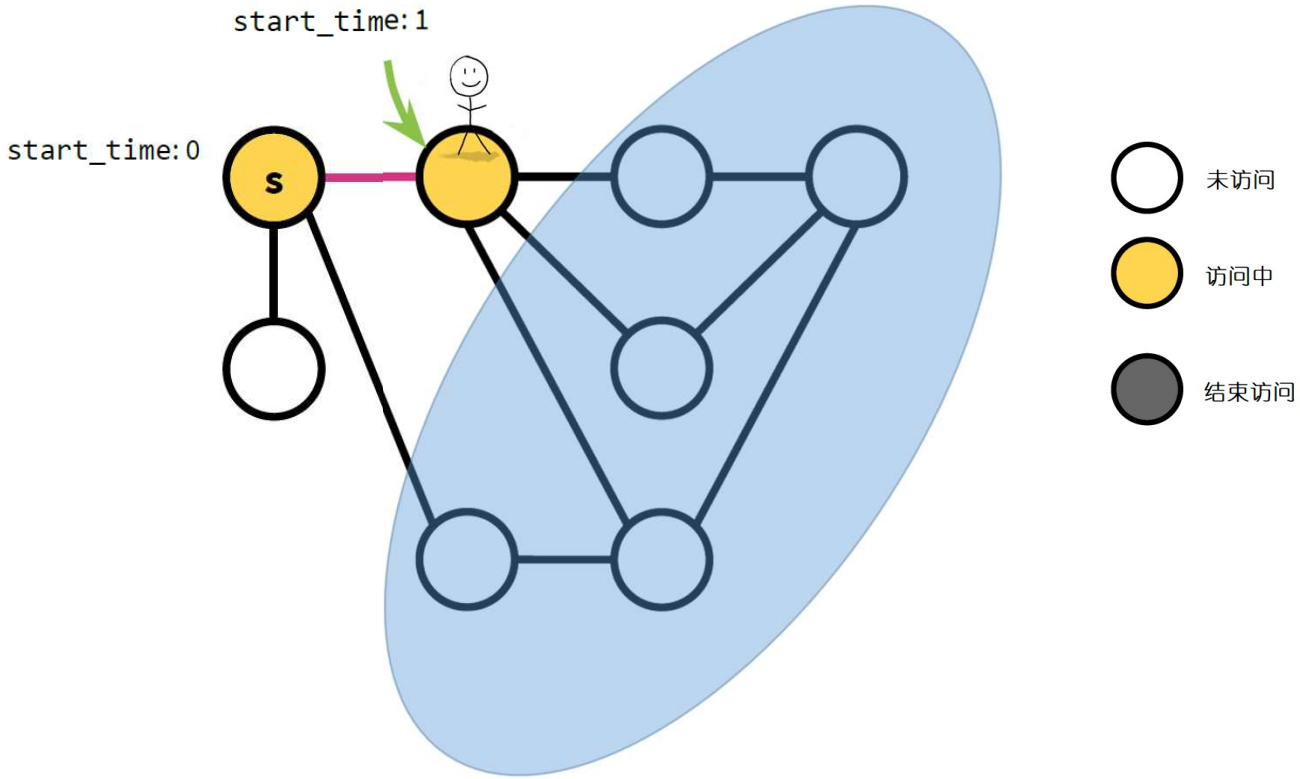
DFS算法+始末时间

加入始末时间



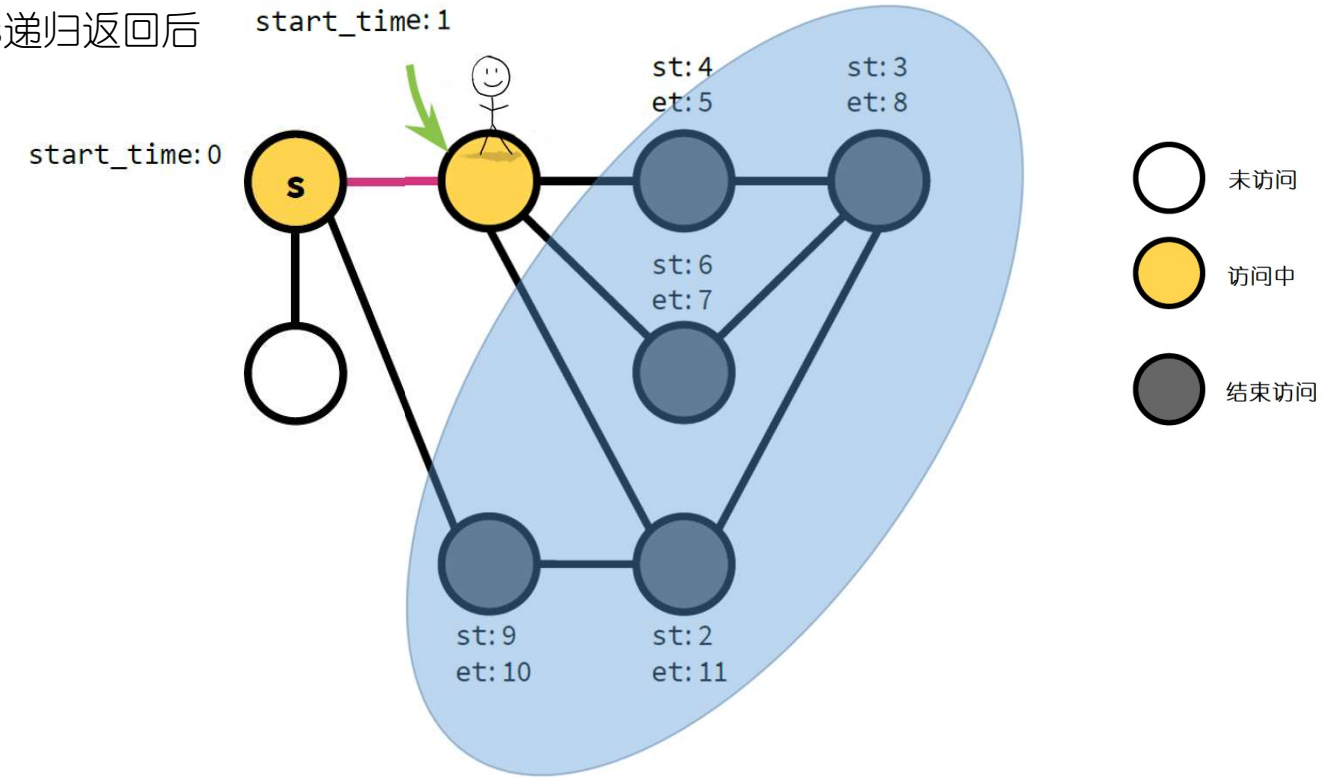
DFS算法+始末时间

递归的过程



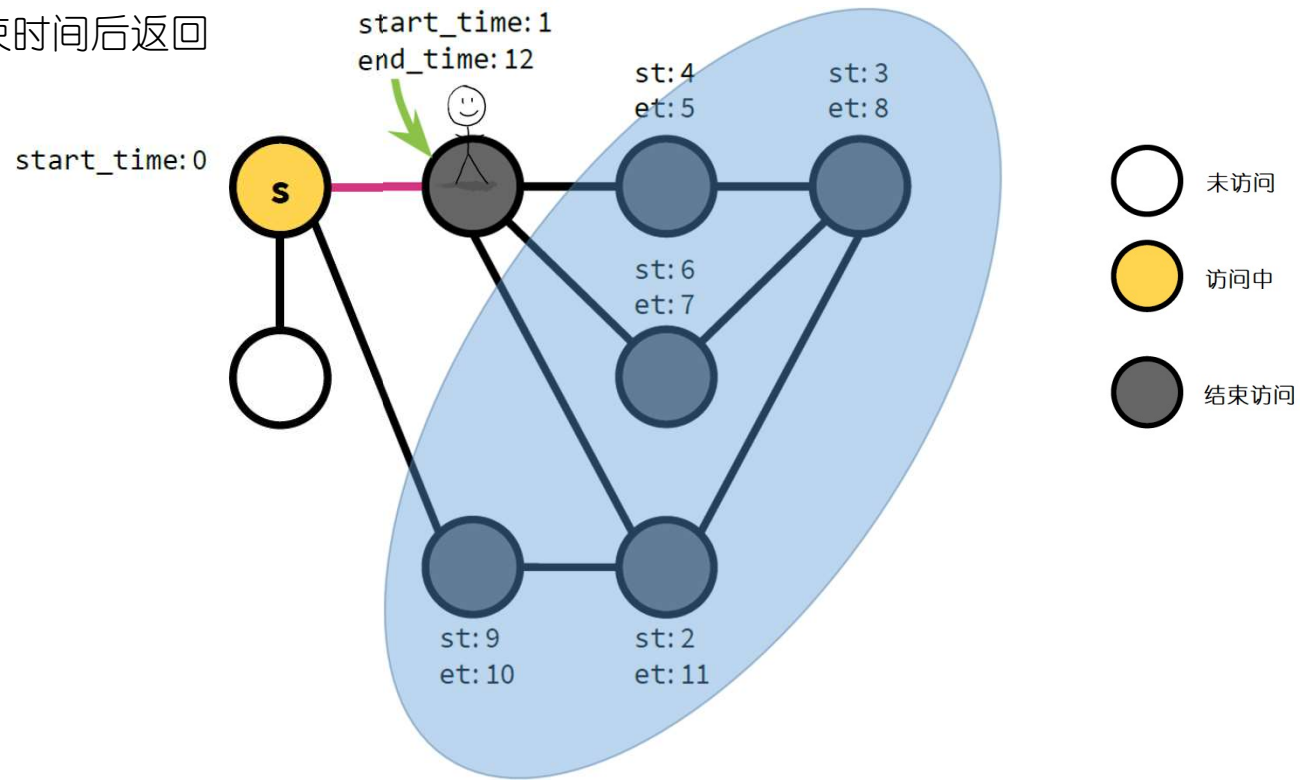
DFS算法+始末时间

当前节点在dfs递归返回后



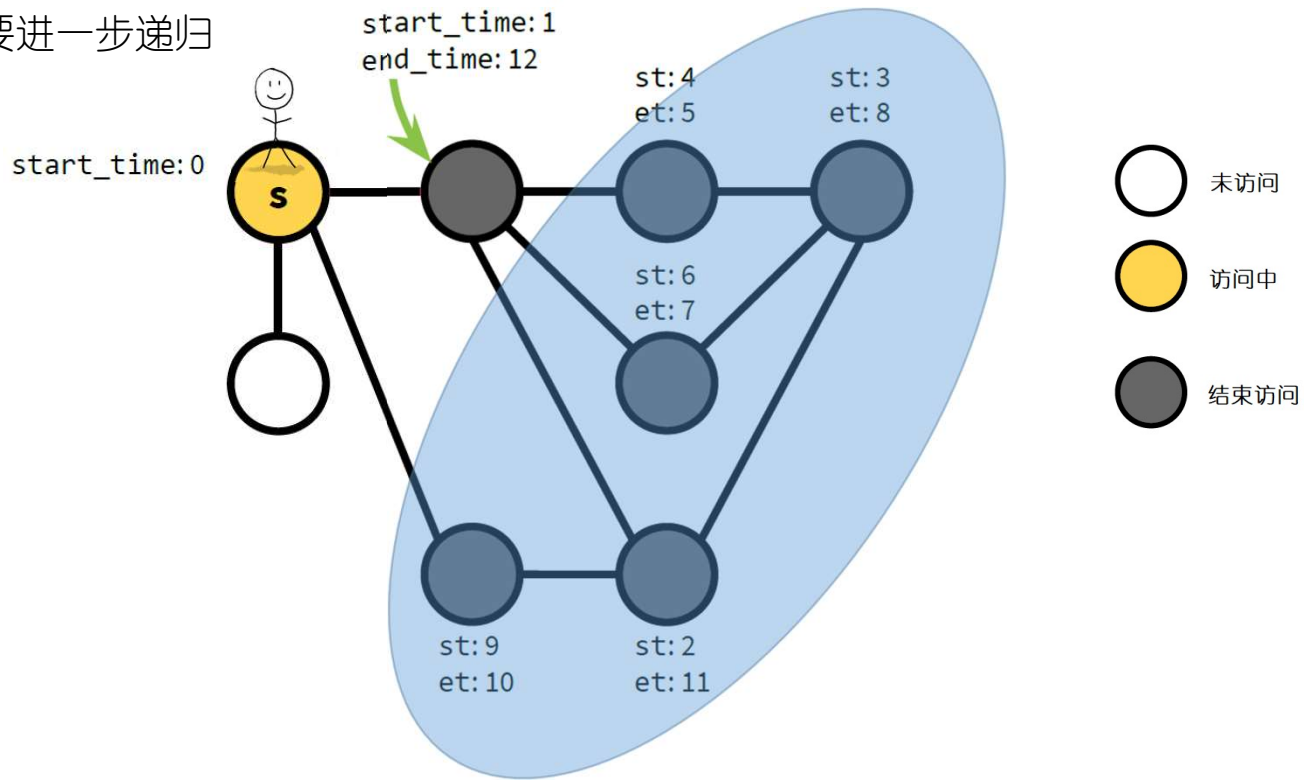
DFS算法+始末时间

设置自己的结束时间后返回



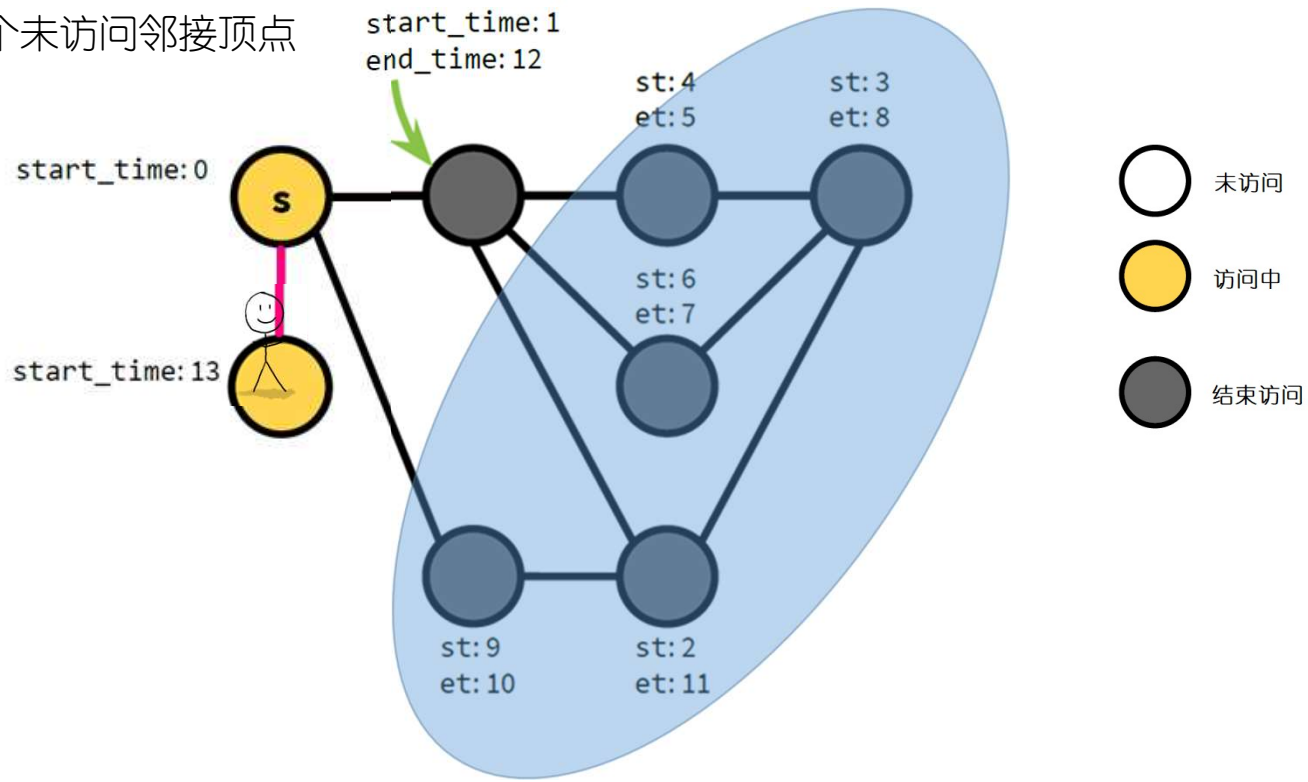
DFS算法+始末时间

其父节点还需要进一步递归



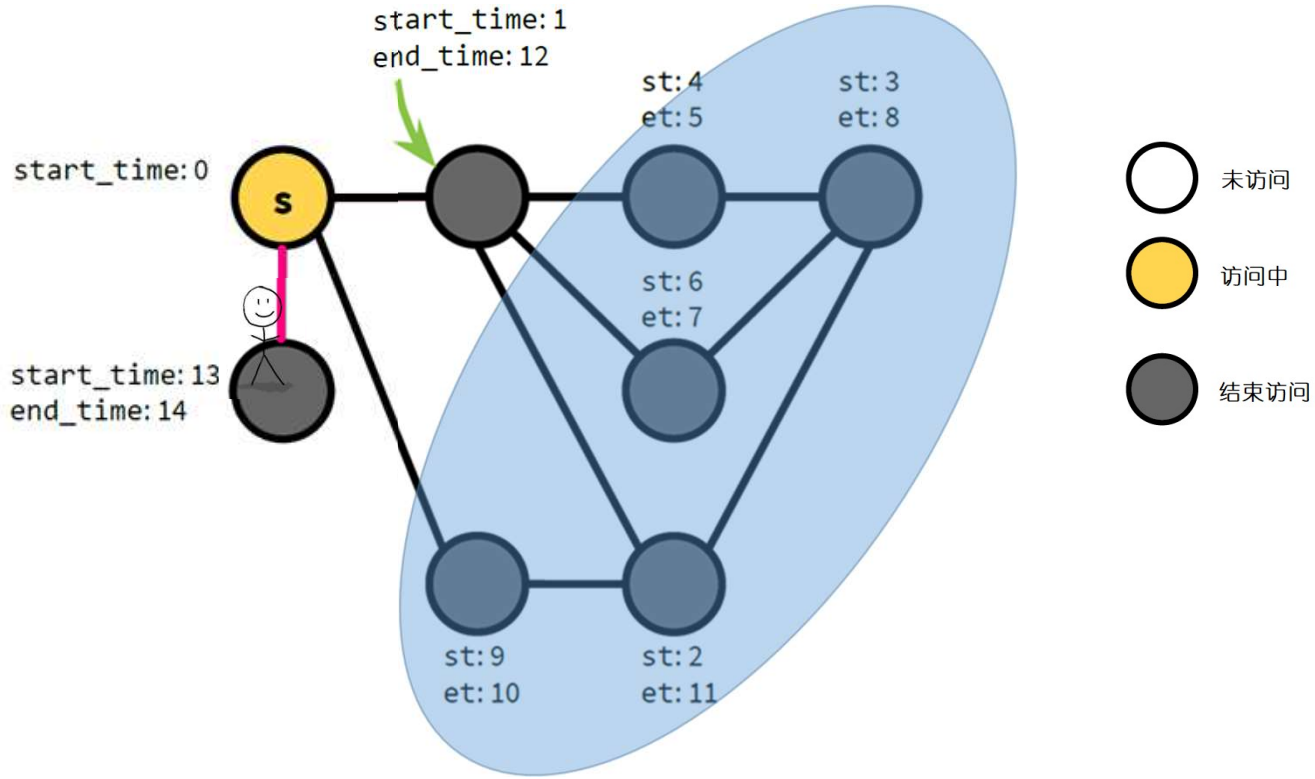
DFS算法+始末时间

父节点还有一个未访问邻接顶点



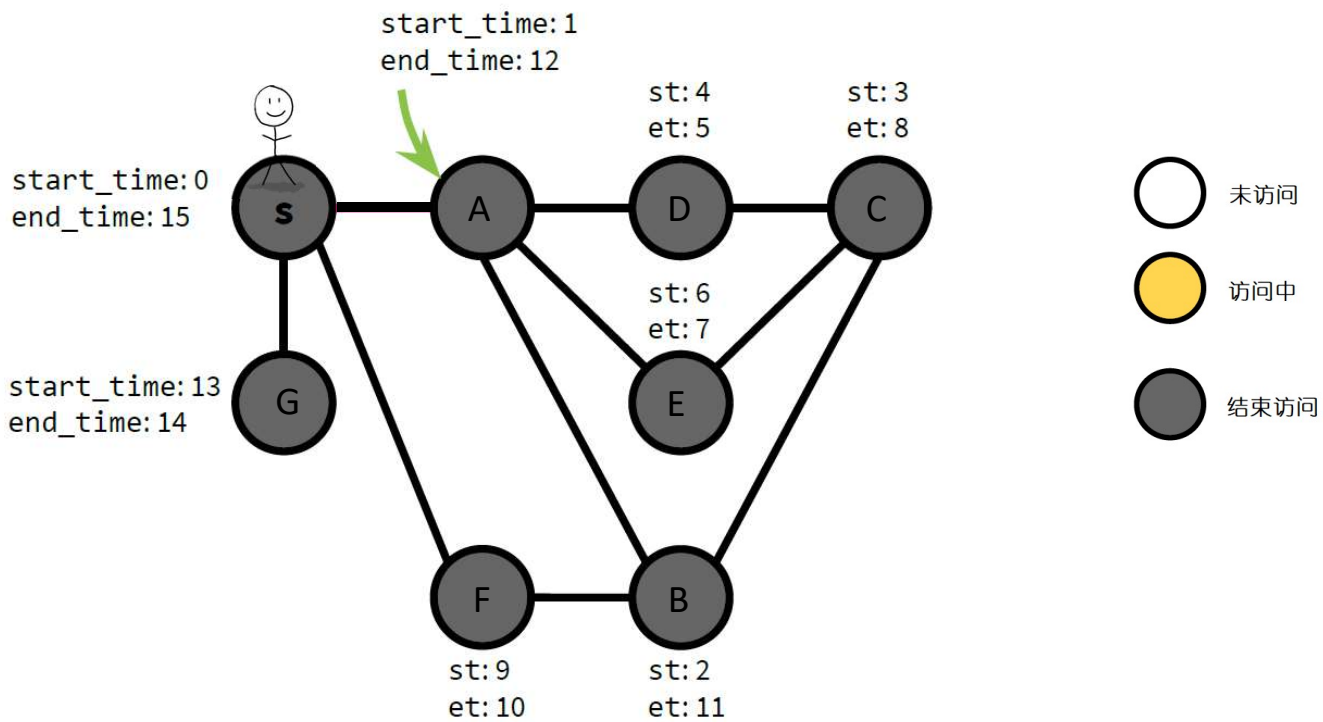
DFS算法+始末时间

结束后返回



DFS算法+始末时间

全部结束



注意点

- 在有向图中，一遍dfs不见得能访问到所有节点
- 在剩下的未访问的节点中再选一个继续dfs
  - 始末时间继续累加
  - 访问过的节点不要再访问

Q&A

Thanks!