图 Graph

1 基本信息

1.1 图的介绍

无向图

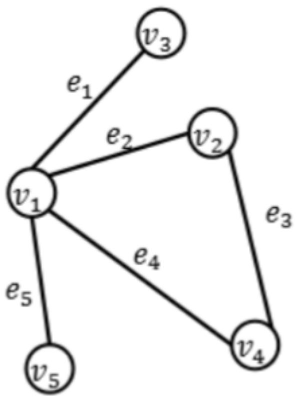
无向图是一对(V,E)，其中：

- V是一组元素，每一个元素称为一个节点
- E是一组无序对$\{u,v\}$，其中u和v都是节点

一个节点也可以称为一个顶点（vertex）。我们称V为图的顶点集或节点集，称E为边集

无向图节点u的**度（degree）**是指节点u上连的边的数量

如图所示



$V=\{v_1,v_2,v_3,v_4,v_5\}$

$E=\{e_1,e_2,e_3,e_4,e_5\}$

有向图

无向图是一对(V,E)，其中：

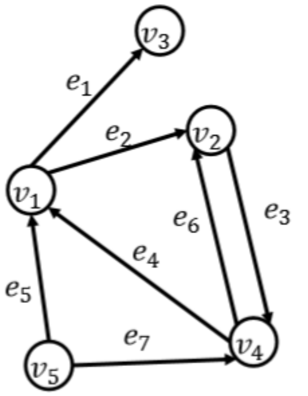
- V是一组元素，每一个元素称为一个节点
- E是一组有序对$\{u,v\}$，其中u和v都是节点，我们说有一条从u到v的有向边

有向边(u,v)是u的一条出边，v的一条入边

据此，v是u的一个出邻居，u是v的入邻居

有向图中节点u的**出度**是指节点u的出边，节点u的入度是指节点u的入边

如图所示



$V = \{v_1, v_2, v_3, v_4, v_5\}$

$E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$

$e_3 = \{v_2, v_4\}$

$e_6 = \{v_4, v_2\}$

1.2 图的定义

设$G = (V, E)$是一个图

路径Path

G中的一条**路径**是指一系列节点$(v_1, v_2, ..., v_k)$使得

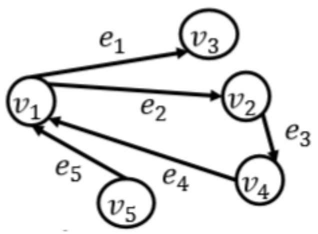
- 对于任何$i \in [1, k]$存在一条边在v_i和v_{i+1}之间

环Cycle

G中的一个环是一条路径$(v_1, v_2, ..., v_k)$使得

- $k \geq 4$且$v_1 = v_k$

如图所示

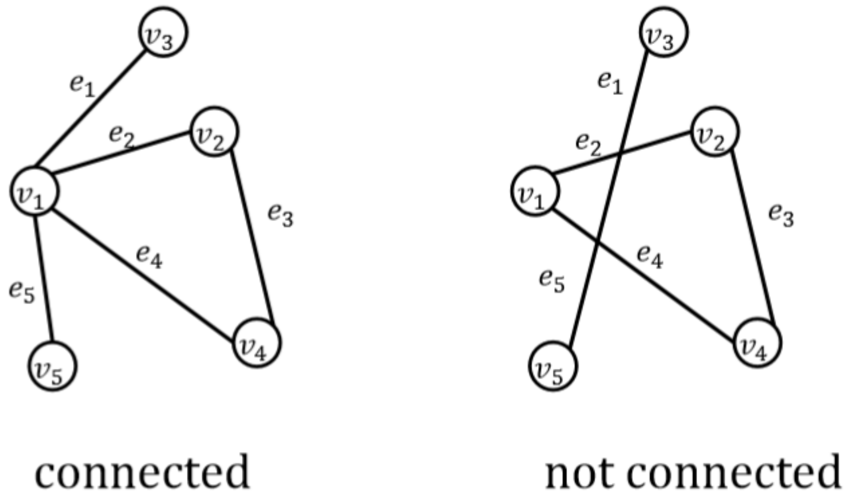


路径：(v_5, v_1, v_2, v_4)

环：(v_1, v_2, v_4, v_1)

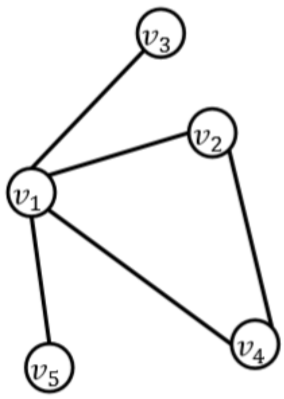
连通图Connected Graph——无向图

如果对于任意两个不同的顶点u和v, $G = (V, E)$有一条从u到v的路径，那么我们说无向图$G = (V, E)$是连通的

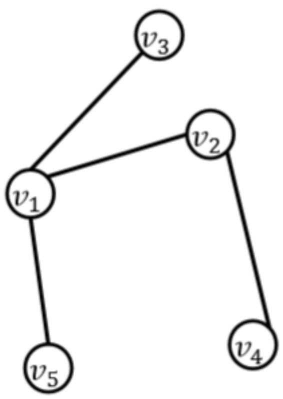


图、树、森林

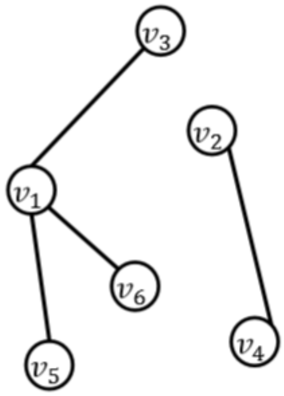
图（Graph）：一对点集和边集



树（Tree）：**没有环的无向连通图**



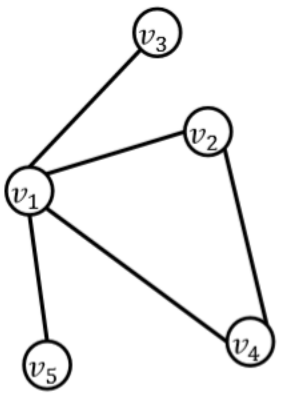
森林（Forest）：一些树的集合

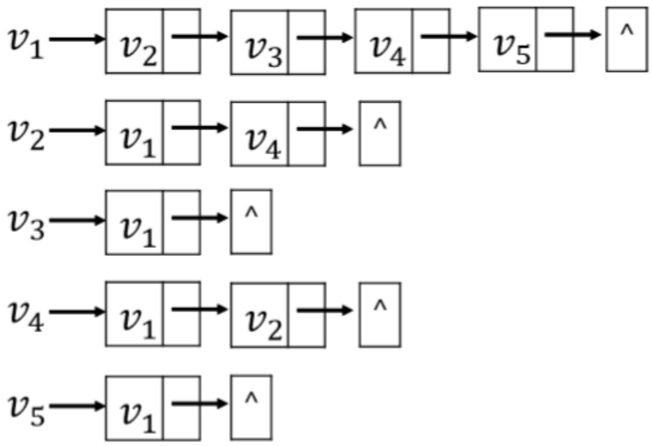


1.3 图的表示

邻接链表Adjacency List——无向图（空间：O(|V|+|E|)）

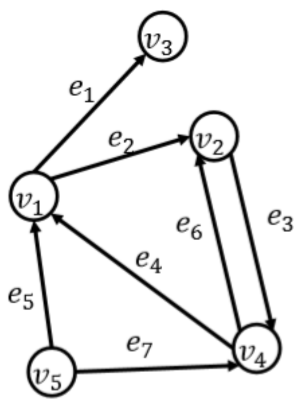
每个节点$u \in V$有连接一个链表，枚举所有与u相连的结点

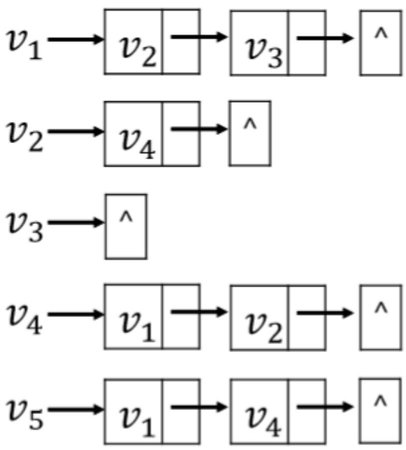




邻接链表Adjacency List——有向图（空间：O(|V|+|E|)）

每个节点$u \in V$有连接一个链表，枚举所有u的出邻居

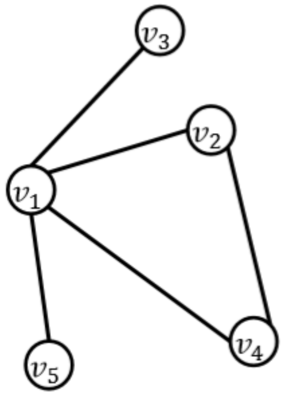




邻接矩阵Adjacency Matrix——无向图（空间：O(|V²|)）

一个$|V| * |V|$的矩阵A，其中如果(u, v)是E的一条边，那么$A[u, v] = 1$

A一定是对称的

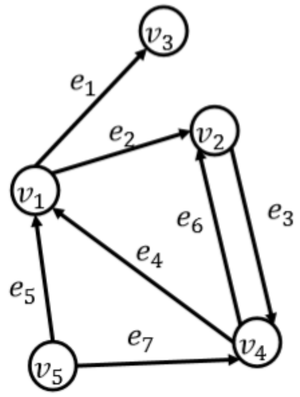


	v_1	v_2	v_3	v_4	v_5
v_1	0	1	1	1	1
v_2	1	0	0	1	0
v_3	1	0	0	0	0
v_4	1	1	0	0	0
v_5	1	0	0	0	0

邻接矩阵Adjacency Matrix——有向图（空间：O(|V²|))

一个$|V|*|V|$的矩阵A，其中如果(u,v)属于E且是u的出边，那么$A[u,v]=1$

A不一定是对称的



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

2 图的遍历

2.1 无边权最短路径介绍——有向图

让$G(V,E)$是一个有向图，G中的一条**路径**是指一系列节点$(v_1,v_2,...,v_k)$使得

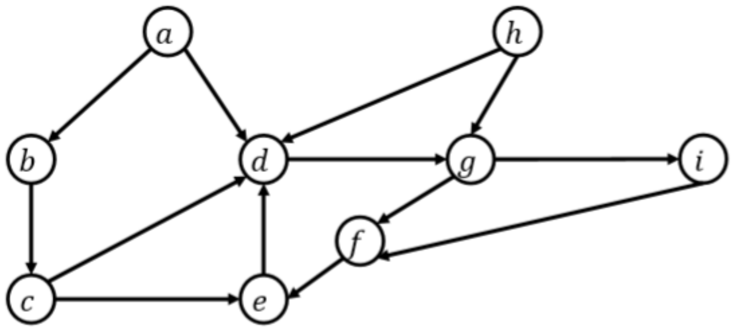
- 对于任何$i\in[1,k]$存在一条边从v_i到v_{i+1}
- 有时候我们也记为$v_1\rightarrow v_2\rightarrow,...,\rightarrow v_k$

从v_1到v_k路径的长度为$k-1$

给定两个顶点$u,v\in V$，从u到v的最短路径是从u到v的路径中长度最短的那一个

如果从u到v没有路径，我们说u，v不可达

例



从a到g有几条路径

$a \to b \to c \to d \to g$
$a \to b \to c \to e \to d \to g$
$a \to d \to g$

其中

$a \to d \to g$是最短路径，长度为2

注意

从a出发，h不可达

单源最短路径Single Source Shortest Path——有向图无边权

让$G = (V, E)$是一个单位边权的有向图，s是V中的一个节点

单源最短路径SSSP问题是找到s与任何一个除了s的节点之间的最短路径（除非不可达）

2.2 广度优先搜索BFS——SSSP问题——有向图无边权

思路

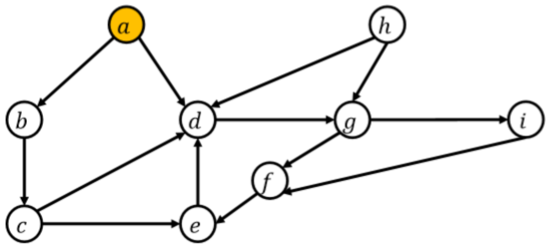
1. 将图中所有的节点都标识为**白色**（尚未访问），然后创建一个空的BFS树T
2. 创建一个队列Q将源节点s入队，并将其标识为黄色（在队列中）
3. 使s成为树T的根
4. 重复5，6，7，8直到Q是空的
5. 　离队Q中第一个节点v
6. 　对于v的每一个出邻居u，如果仍表示为白色
7. 　　把u入队Q，把u标识为黄色
8. 　　在BFS树中让u成为v的子节点
9. 把v标识为红色（离队）

伪代码——BFS

```
Algorithm:BFS(Graph = (V,E), Node src)

Color all the Node in V WHITE
Create Queue Q
Create BFS Tree
src = tree.root
src.color = YELLOW
Q.enqueue(src)
while(Q is not empty)
    v = Q.dequeue()
    for(each out-neighbour u in v's outEdge)
        if(u.color == WHITE)
            Q.enqueue(u)
            u.color = YELLOW
            src.child.add(u)
    v.color = RED

```

示例

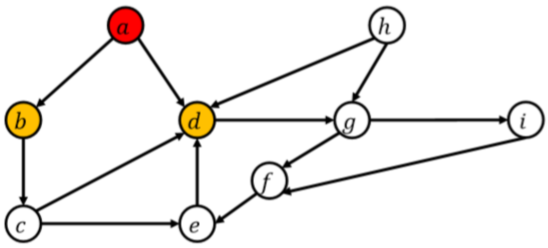
- Suppose that source vertex is a.



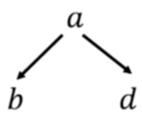
BFS tree

- Q = (a)

- After de-queuing a:

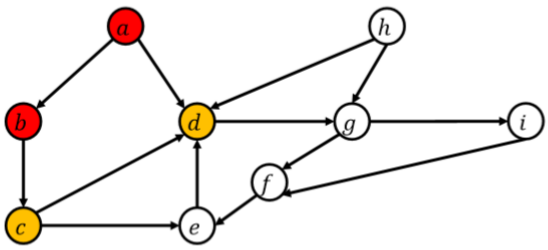


BFS tree

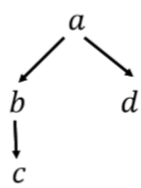


- Q = (b, d)

- After dequeuing b:

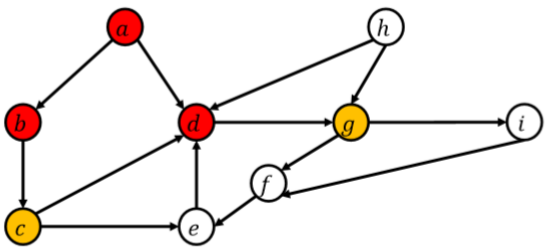


BFS tree

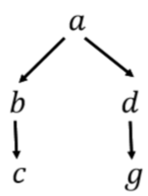


- Q = (d, c)

- After dequeuing d:

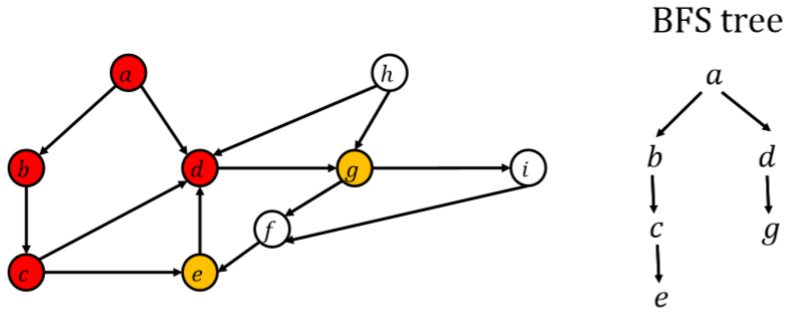


BFS tree



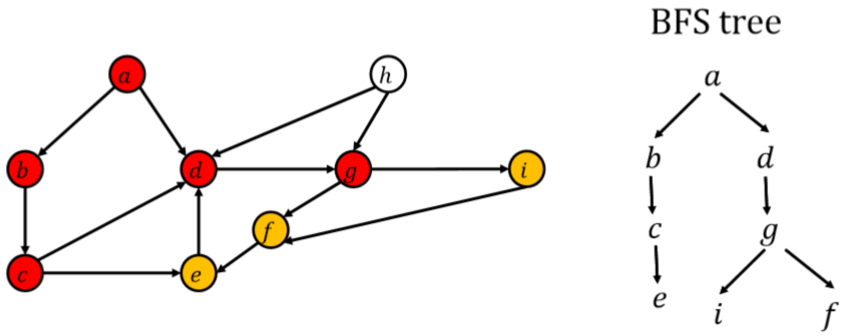
- Q = (c, g)

- After dequeuing c:



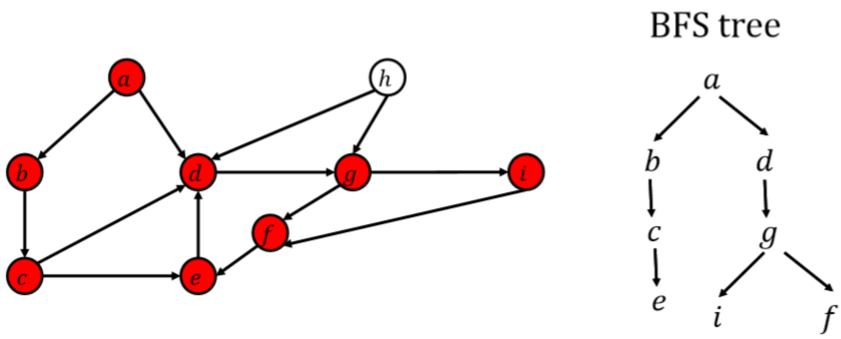
- Q = (g, e)
- d is not enqueue again as it is red now

- After dequeuing g:



- Q = (e, i, f)

- After dequeuing e, i, f



- Q = ()

此时BFS结束，注意h仍然标识为白色，可以得到a不可到达h

SSSP问题的解

从a到任意节点x的最短路径就是BFS树中从根a到任意一个节点的路径

时间复杂度分析——BFS（O(|V|+|E|)）

当一个节点v离队时，我们花$O(1+d^+(v))$的时间处理它，其中d^+v是v的出度

每个节点只会入队和离队一次

所有的出度加起来有 $\sum _{i=1}^k d^{+}v_i = E$

因此BFS的时间复杂度为$O(|V|+|E|)$

2.3 深度优先搜索DFS——SSSP问题——有向图无边权

介绍

DFS算法是一种非常强大的算法，能很好地解决几个经典问题

我们将看到这样一个问题：检测输入**图是否包含环**

DFS将会在$O(|V|+|E|)$的时间内解决这个问题

能使用DFS解决的问题，基本都是最优解

与BFS一样，DFS算法也输出一棵树，称为DFS树

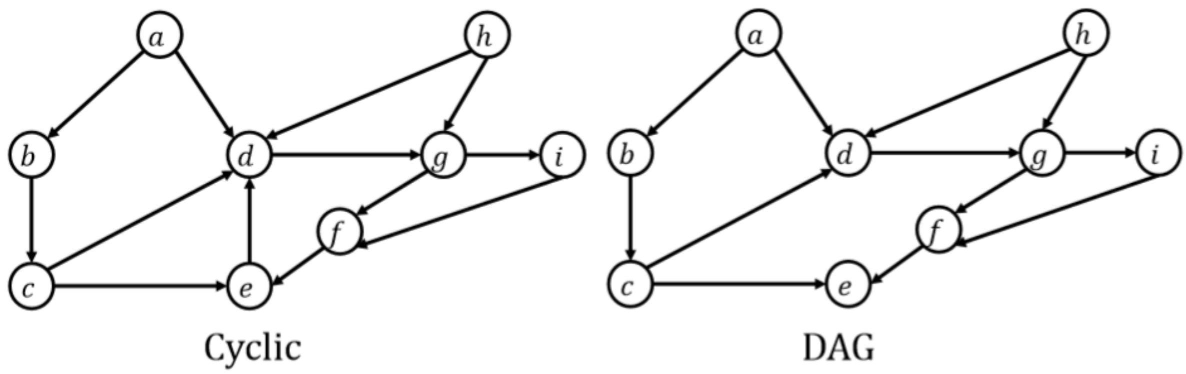
这棵树包含了关于输入图的重要信息，允许我们决定输入图是否有环

有向无环图DAG（Directed Acyclic Graph）定义

如果一个有向图不包含环，我们称之为有向无环图(DAG)。否则G是有向有环图

DAG是计算机科学中非常重要的概念，如**spark、tensorflow**等

如图所示，左图为有向有环图，右图为DAG



思路

1. 将图中所有的节点都标识为**白色**（尚未访问），然后创建一个空的DFS树T
2. 创建一个队列S，随便挑一个节点v压栈，并将其标识为黄色（在栈中）
3. 使v成为树T的根
4. 重复5，6，7，8，9，10，11直到S是空的
5. 查看(peak)S中第一个节点v（先不出栈）
6. 对于v的所有出邻居u，判断是否还有标识为白色的出邻居
7. 如果查到了第一个白色的出邻居u
8. 把u压栈，并把它标识成黄色
9. 在BFS树中让u成为v的子节点
10. 如果任何一个出邻居都不是标识为白色的
11. 离队v，把它标识为红色（离队）

注意

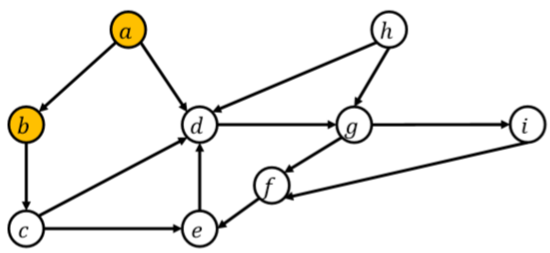
上述操作结束后，如果节点集合V中还有其它节点是白色的，则从里面随便挑一个节点v，继续按照上述方法构造一个新的DFS树

伪代码——DFS

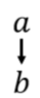
```
Algorithm:DFS(Graph = (V,E))

Color all the Node in WHITE
Create Stack S
Create DFS Tree
Ramdom pick one node v
v.color = YELLOW
tree.root = v

S.push(v)
while(S is not empty)
    v = S.peak()
    hasWhiteNeighbour = false
    Node u
    for(each out-neighbour u in v's outEdge)
        if(u.color == WHITE)
            hasWhiteNeighbour = true
            u = thisNode
    if(hasWhiteNeighbour = true)
        S.push(u)
        u.color = YELLOW
        v.child = u
    else if(hasWhiteNeighbour = false)
        v = S.pop()
        v.color = RED

```

示例

- Top of stack: a, which has white out-neighbors b, d. Suppose we access b first. Push b into S

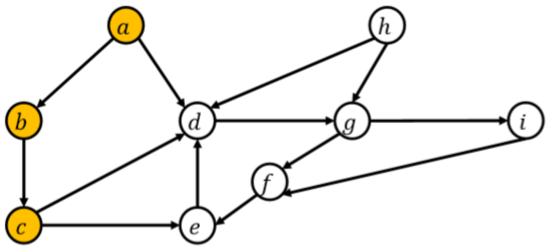


DFS tree



- S = (a, b).

- Top of stack: b, which has white out-neighbors c. Push c into S

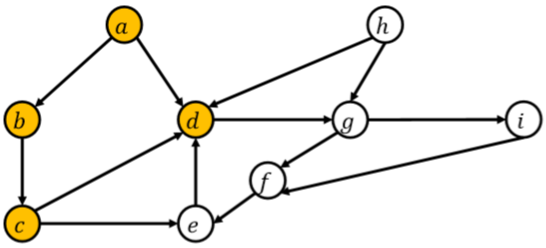


DFS tree

$$a \downarrow b \downarrow c$$

- S = (a, b, c).

- Top of stack: c, which has white out-neighbors d and e. Suppose we access d first. Push d into S

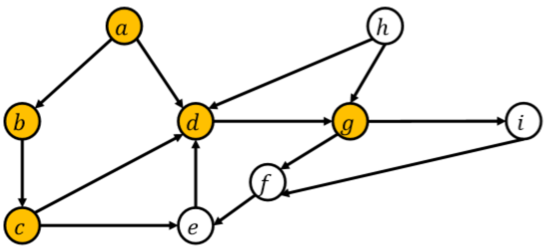


DFS tree

$$a \downarrow b \downarrow c \downarrow d$$

- S = (a, b, c, d).

- Top of stack: d, which has white out-neighbors g. Push g into S



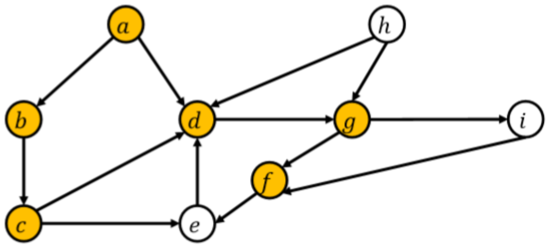
DFS tree

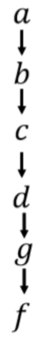
$$a \downarrow b \downarrow c \downarrow d \downarrow g$$

- S = (a, b, c, d, g).

- Top of stack: g, which has white out-neighbors f and i. Suppose we access f first. Push f into S

DFS tree

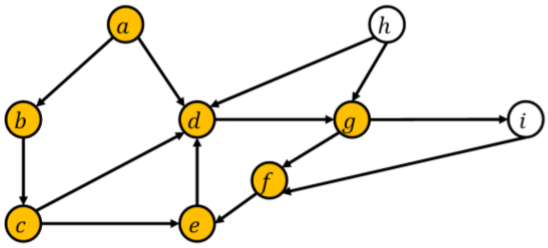


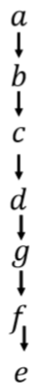


- S = (a, b, c, d, g, f).

- Top of stack: f, which has white out-neighbors e. Push e into S

DFS tree

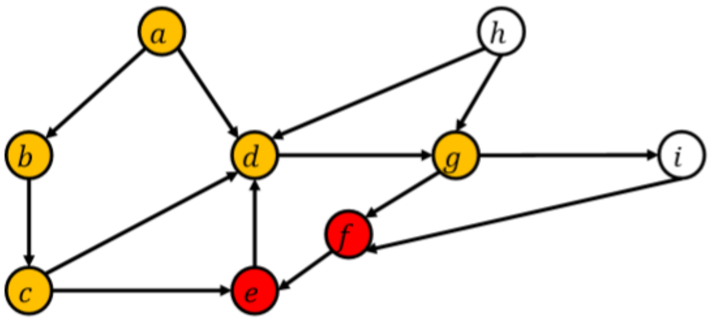


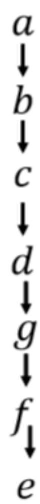


- S = (a, b, c, d, g, f, e).

- Top of stack: e, e has no white out-neighbors. So pop it from S, and color it red. Similarly for s.

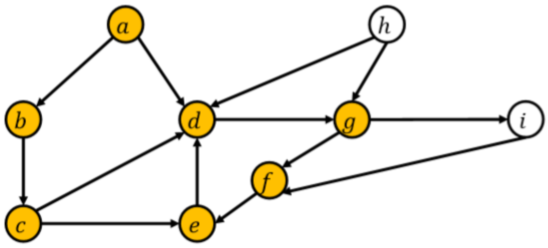
DFS tree



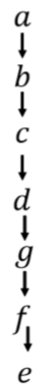


- S = (a, b, c, d, g).

- Top of stack: f, which has white out-neighbors e. Push e into S



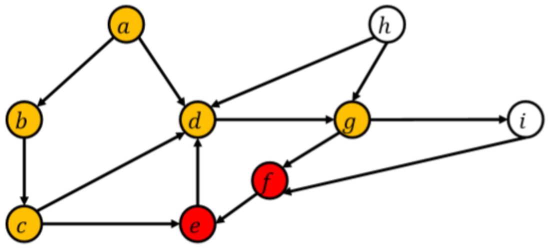
DFS tree

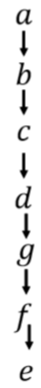


- S = (a, b, c, d, g, f, e).

- Top of stack: e, e has no white out-neighbors. So pop it from S, and color it red. Similarly for s.

DFS tree

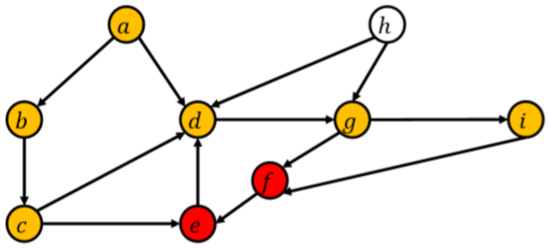


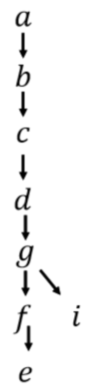


- S = (a, b, c, d, g).

- Top of stack: g, which still has white out-neighbors i. Push i into S

DFS tree

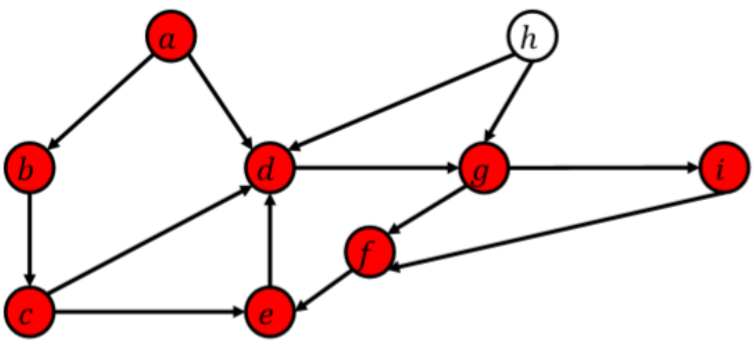


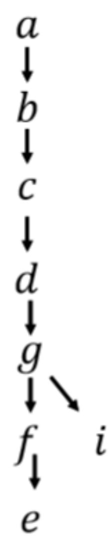


- S = (a, b, c, d, g, i).

- After popping i, g, d, c, b, a

DFS tree

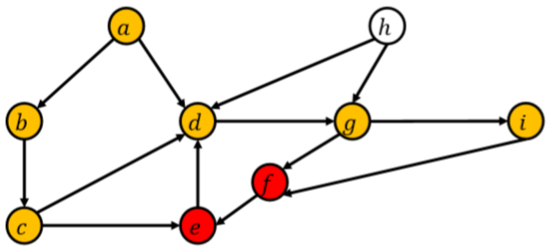


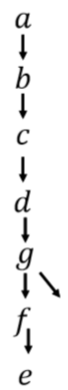


- S = ().

- Top of stack: g, which still has white out-neighbors i. Push i into S

DFS tree

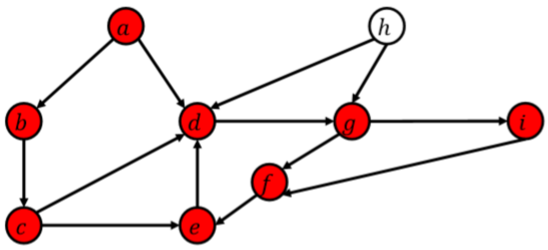


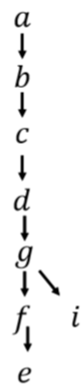


- S = (a, b, c, d, g, i).

- After popping i, g, d, c, b, a

DFS tree

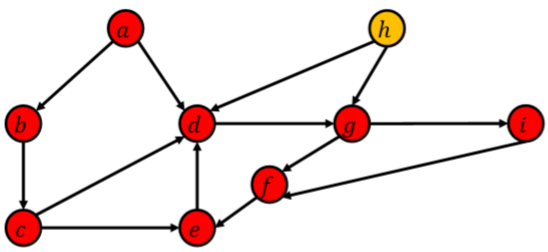


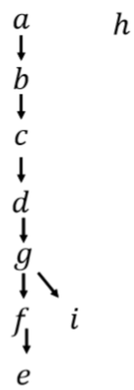


- S = ().

- Now there is still a white vertex h. So we perform another DFS starting from h

DFS forest

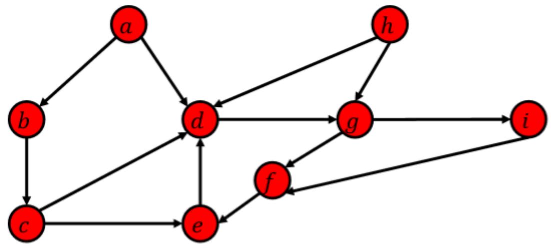


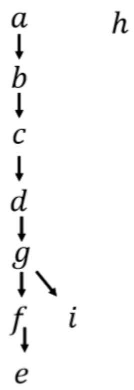


- S = (h).

- Pop h. The end.

DFS forest





- S = ( ).
- Note that we have created a DFS-forest,

Which consists of 2 DFS-trees.

时间复杂度分析——DFS（O(|V|+|E|)

当一个节点v出栈时，我们花$O(1+d^{+}(v))$的时间处理它，其中$d^{+}v$是v的出度

每个节点只会压栈和出栈一次

所有的出度加起来有 $\sum _{i=1}^k d^{+}v_i = E$

因此DFS的时间复杂度为$O(|V|+|E|)$

有向图是否有环的判断思路——O(|V|+|E|)

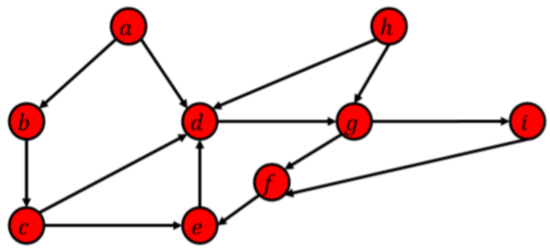
假设我们已经跑完了DFS，得到了DFS森林T

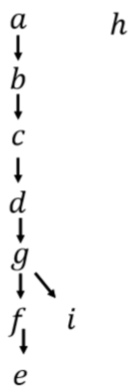
设(u,v)为G中的一条边（记住这条边是从u指向v的）

它可以分为

- 前向边Forward edge：u是v的祖先
- 后向边Backword edge：u是v的后代
- 交叉边Cross edg：不是上述两种情况的（在树的同一层，在不同的树）

DFS Forest





- Forward edge:
  - (a,b),(a,d),(b,c),(c,d),(c,e),(d,g),(g,f),(g,i),(f,e)
- Backward edge: (e,d)
- Cross edge: (i,f),(h,d),(h,g)

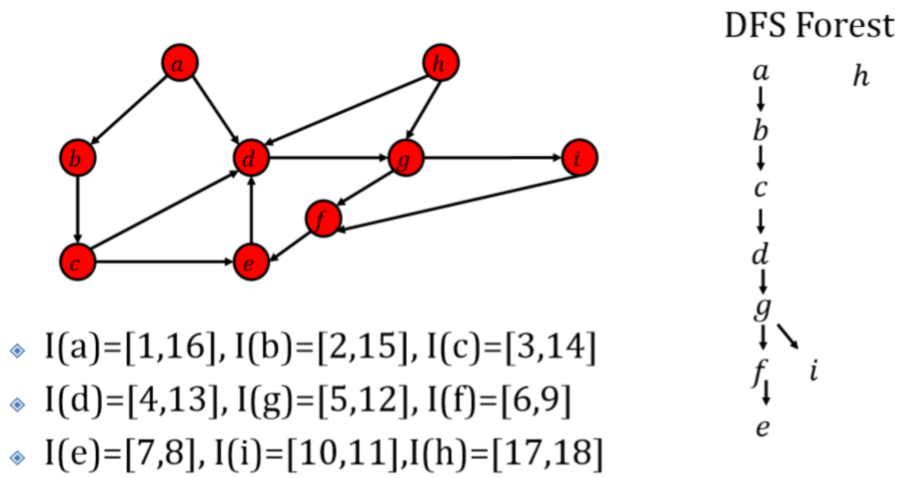
如何通过O(1)代价确定每条边(u,v)的类型?

维护计数器c，它最初是0，每次在栈上执行push或pop操作时，我们将c加1

对于每个节点v，定义

- 它的发现时间$d\text{-}tm(v)$是v压栈之后c的值

- 它的结束时间$f-tm(v)$是v出栈之后c的值
- 定义$I(v) = [d-time(v)， f-tm(v)]$



括号定理

- 如果u是v在T的DFS-tree中的祖先，那么$I(u)$包含$I(v)$
- 如果u是v在T的DFS-tree中的后代，那么$I(u)$就包含在$I(v)$中
- 否则，I(u)和I(v)是不相交的

括号定理的证明

运用到栈先进后出FIFO的特点

有向图是否有环

设T为任意的DFS森林

G包含一个环的条件当且仅当存在关于T的后向边

如果没有找到后向，则判定G为DAG

2.4 拓扑排序——DAG——使用DFS解

拓扑排序定义

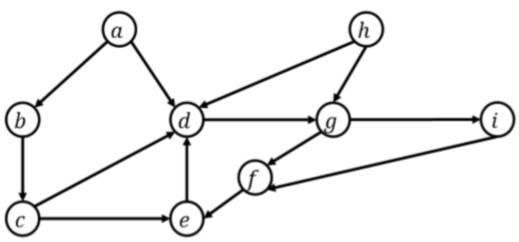
设$G = (V, E)$是有向无环图DAG

G的拓扑顺序是V中的顶点的顺序，使得对于任何边(u, v)，它必须在顺序上保持u在v之前

拓扑排序**不唯一**

每一个DAG都有自己的拓扑排序

如图所示



两种可能的拓扑排序

- h, a, b, c, d, g, i, f, e

- a, h, b, c, d, g, i, f, e

思路

创建一个空列表L

在G上运行DFS，每当节点v变成红色时（即从出栈时），将它添加到L

输出与L相反的顺序

❗时间复杂度分析——拓扑排序（O(|V|+|E|)）

很明显，因为它只是在拓扑排序中增加了一小部分，总体时间复杂度不变

3 有边权最短路径算法Shortest Path Algorithm(SP)

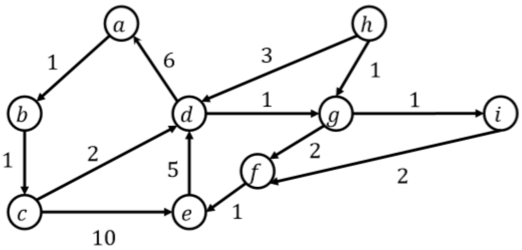
3.1 带边权图定义——有向图有边权

设$G=(V,E)$是有向图，设w是将E中的每条边映射为正整数的函数。

具体来说,对于每个$e \in E, w(e)$是一个正整数，我们称之为e的边权

有向带边权图定义为一对(G,w)

如图所示



3.2 最短路径——有向图有边权

考虑一个由有向图G=(V,E)和函数w定义的有向带边权图

考虑G中的一个路径$(v_1,v_2),(v_2,v_3)\dots,(v_l,v_{l+1})$对于一些整数$l \ge 1$我们定义路径的长度为

$$\sum_{i=1}^{l} w(v_i, v_{i+1})$$

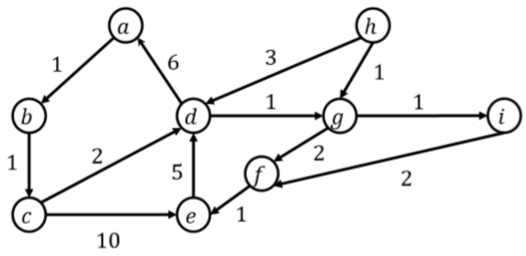
给两个顶点$u, v \in V$,从u到v的最短路径是从所有u到v的路径中长度最短的那一个

如果u不能到达v，那么u到v的最短路径是∞

性质

如果$v_1 \to v_2 \to \cdots \to v_{l+1}$ 是从 v_1 to v_{l+1}的最短路径， 那么对于任何i和j满足$1 \le i \le j \le l+1$, $v_i \to v_{i+1} \to \cdots \to v_j$ 是从 v_i 到 v_j的最短路径

例



$c \rightarrow e$ 的长度是10

$c \rightarrow d \rightarrow g \rightarrow f \rightarrow e$ 的长度是6

第二条路线是最短路径

3.3 Dijkstra 算法

引入

利用子序列属性，我们的算法将生成一棵最短路径树，它编码了从源顶点s开始的所有最短路径

对每个节点$v \in V$,我们将保持一个值\$dist (v)\$表示到目前位置从s到的到v的最短路径

在算法的最后，我们将确保每个\$dist(v)\$都等于从s到v的精确最短路径

我们算法的核心操作是动态规划，给定一条边(u, v)

- 如果\$dist(v)<dist(u)+w(u,v)\$什么都不做
- 否则，\$dist(v)=dist(u)+w(u,v)\$

思路

1. 初始化建立数组\$parent(v)=null\$
2. 初始化建立数组\$dist(v)=\infin\$
3. 源节点s的\$dist(s)=0\$
4. 建立集合\$S\$，初始化存放所有的节点
5. 重复操作直到S为空
6. 从\$S\$中找到\$dist(u)\$最小的顶点u
7. 对于u的每一条出边
8. 如果\$dist(v)>dist(u)+w(u,v)\$
9. 设置\$dist(v)=dist(u)+w(u,v)\$

伪代码——Dijkstra

```
Algorithm:Dijkstra(Graph = (V,E),Node src)

Create parent[V.len], each element = null
Create dist[V.len], each element = ∞
Create set S = V
dist[src] = 0

while(S is not empty)
    u = the Node which has the min dist in S
    S.remove(u)
    for(each out-neighbour v in u's outEdge)
        if(dist[v] > dist[u] + w[u][v])
            dist[v] = dist[u] + w[u][v]

```

❗时间复杂度分析——Dijkstra（O[(|V|+|E|) * log|V|])

4 最小生成树Minimum Spanning Tree(MST)

4.1 带边权图定义——无向图有边权

设$G=(V,E)$是无向图，设w是将E中的每条边映射为正整数的函数

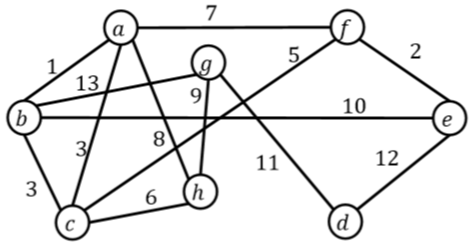
具体来说,对于每个$e\in E,w (e)$是一个正整数，我们称之为e的边权

无向带边权图定义为一对(G,w)

我们将G中顶点u和v之间的边表示为$\{u,v\}$，而不是(u,v)，以强调u,v的顺序无关紧要

我们认为G是连通的，即V中的任意两个顶点之间存在一条路径

例



4.2 生成树定义

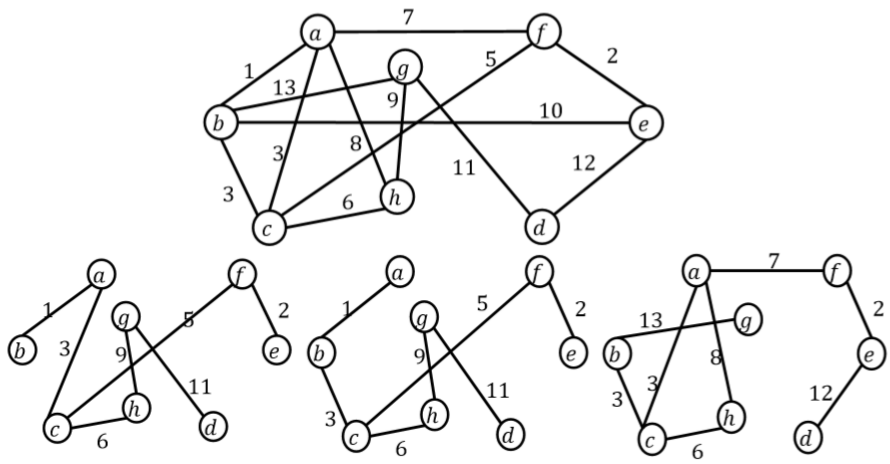
记住树被定义为没有环的连通无向图

给定$G=(V,E)$的连通无向带边权图(G,w)，生成树T是满足以下条件的树

- T包含V的所有节点
- T中的所有边都属于G

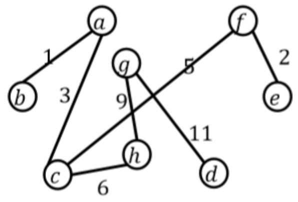
T的代价（cost）定义为T中所有边权之和(注意T必须有$|V|-1$条边)

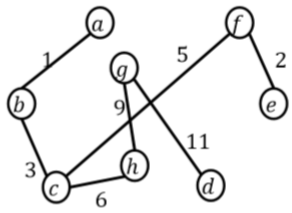
例



最小生成树

代价最小的生成树





4.3 Prim算法——解最小生成树

代码

```
//最小生成树
class SpanningTree {

    ArrayList<Edge> edges;//所有的边的集合
    ArrayList<Node> nodes;//所有的点的集合
    int totalCost = 0;//最小生成树的总cost


    Set<Node> S = new HashSet<>();//已经在最小生成树里面的节点
    Edge lightExtensionEdge;//当前最短的出边


    //最小可拓展边 -> 用最小堆实现
    PriorityQueue<Edge> best_Ext = new PriorityQueue<>(new Comparator<Edge>() {
        @Override
        public int compare(Edge o1, Edge o2) {
            return o1.weight - o2.weight;
        }
    });

    /**
     * @param edges                边哈希集合
     * @param nodes                节点哈希集合
     * @param smallestWeightEdge 最小的边权 -> 用于初始化
     */
    public SpanningTree(ArrayList<Edge> edges, ArrayList<Node> nodes, Edge smallestWeightEdge) {
        this.edges = edges;
        this.nodes = nodes;

        lightExtensionEdge = smallestWeightEdge;
        lightExtensionEdge.color = Color.RED;       //已经加入
        lightExtensionEdge.u.color = Color.RED;     //已经加入
        lightExtensionEdge.v.color = Color.RED;     //已经加入
        S.add(lightExtensionEdge.u);
        S.add(lightExtensionEdge.v);
        totalCost += lightExtensionEdge.weight;
    }


```

```java
    //最小生成树的代价
    public int treeCost() {
        return totalCost;
    }


    public void Span() {
        while (S.size() != nodes.size()) {
            Node u = lightExtensionEdge.u;
            Node v = lightExtensionEdge.v;

            //加入当前u和v没有访问的所有边
            for (int i = 0; i < u.edges.size(); i++) {
                if( (u.edges.get(i).color != Color.RED) ){
                    best_Ext.add(u.edges.get(i));
                }
            }

            for (int i = 0; i < v.edges.size(); i++) {
                if( (v.edges.get(i).color != Color.RED)){
                    best_Ext.add(v.edges.get(i));
                }
            }

            //如果最小堆中弹出来的边的两个Node在Set中都有了的话，就不要这个边了，直到有一个Node不在Set里面
            while(true){
                Edge edge = best_Ext.poll();
                if(!S.contains(edge.v) || !S.contains(edge.u)){
                    lightExtensionEdge = edge;
                    break;
                }
            }

            totalCost += lightExtensionEdge.weight;
            lightExtensionEdge.color = Color.RED;
            u = lightExtensionEdge.u;
            v = lightExtensionEdge.v;
            S.add(u);
            S.add(v);
            u.color = Color.RED;
            v.color = Color.RED;
        }
    }
}

class Node {
    ArrayList<Edge> edges = new ArrayList<>();//邻接矩阵，无向图的边
    int index;//编号
    Color color = Color.WHITE;//初始尚未访问

    public Node(int index) {
        this.index = index;
    }


}

```

```java
//无向图的边
class Edge {
    Node u;
    Node v;
    int weight;
    Color color = Color.WHITE;//边是否已经加入Spanning Tree

    public Edge(Node u, Node v, int weight) {
        this.u = u;
        this.v = v;
        this.weight = weight;
    }

}

enum Color {
    RED, WHITE;
}
```

时间复杂度分析——Prim（O((|V|+|E|) log |V|))

使用斐波那契堆，在本课程中不会讲到，我们可以把运行时间提高到O(|V| log |V| + |E|)