



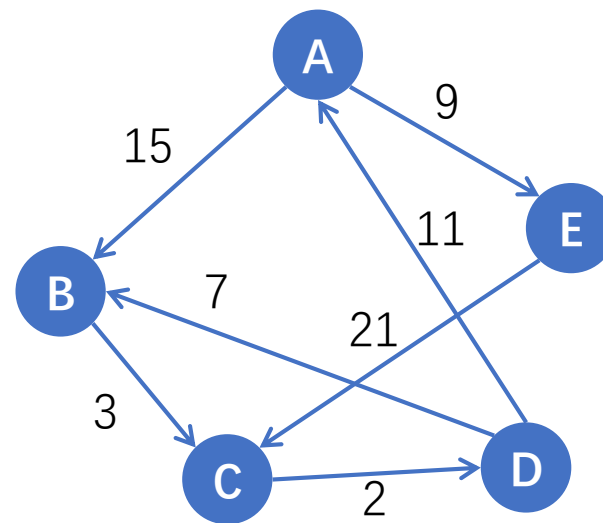
图论专题

湖南师大附中 许力



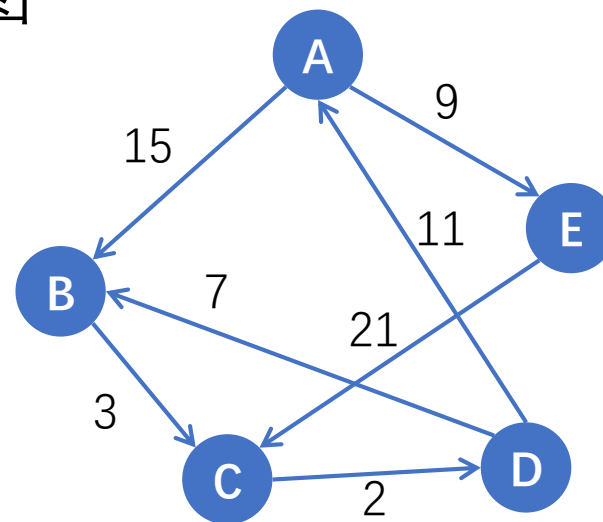
图的定义

- 图是由顶点的集合和边的集合构成的数据结构
- 图G是一个由两个集合V, E组成的二元组(V,E)。其中V表示顶点集, E表示边集。E中的元素是二元组数对, 用(u,v)表示, 其中 $u,v \in V$



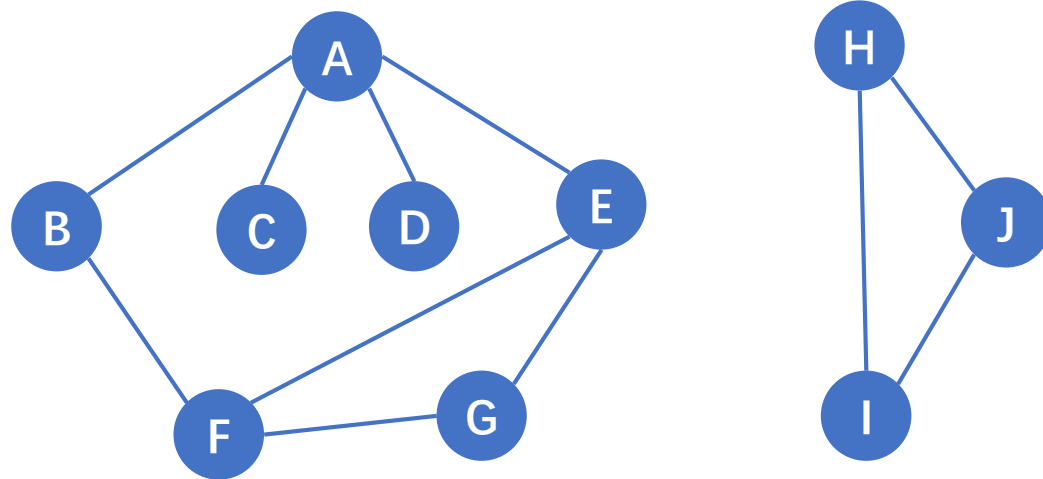
图的补充定义

- 图中的边有方向为有向图，反之则为无向图
- 边可能有权值（不一定非负），无边权可视为1
- 图中的每对顶点间都有边相连，为完全图
- 图中的每对顶点间至少有一条路径，为连通图



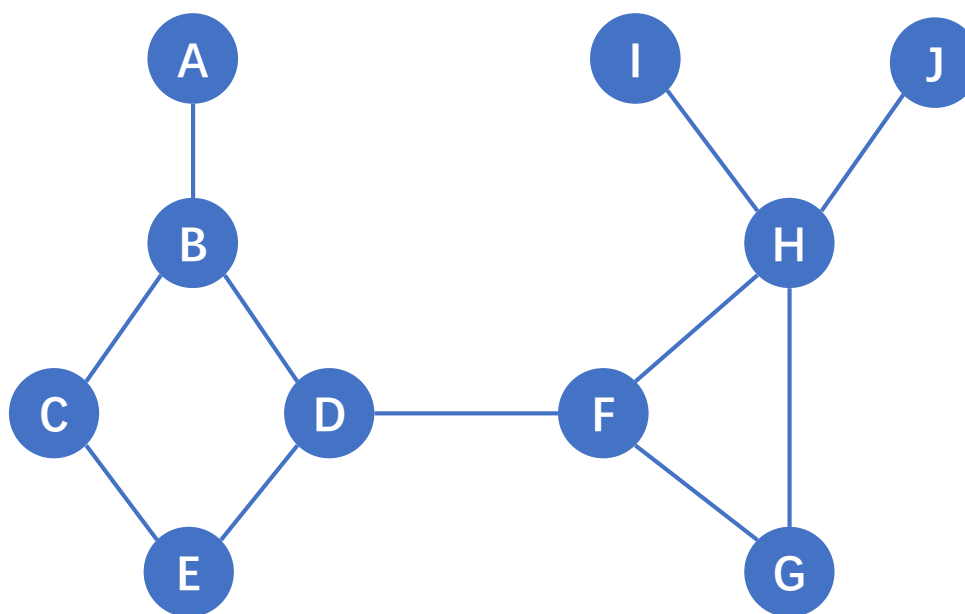
图的连通

- 图中的每对顶点间至少有一条路径，为连通图，反之为非连通图
- 非连通图的极大连通子图，为连通分量
- 有向非连通图的极大强连通子图，为强连通分量



割点/关节点

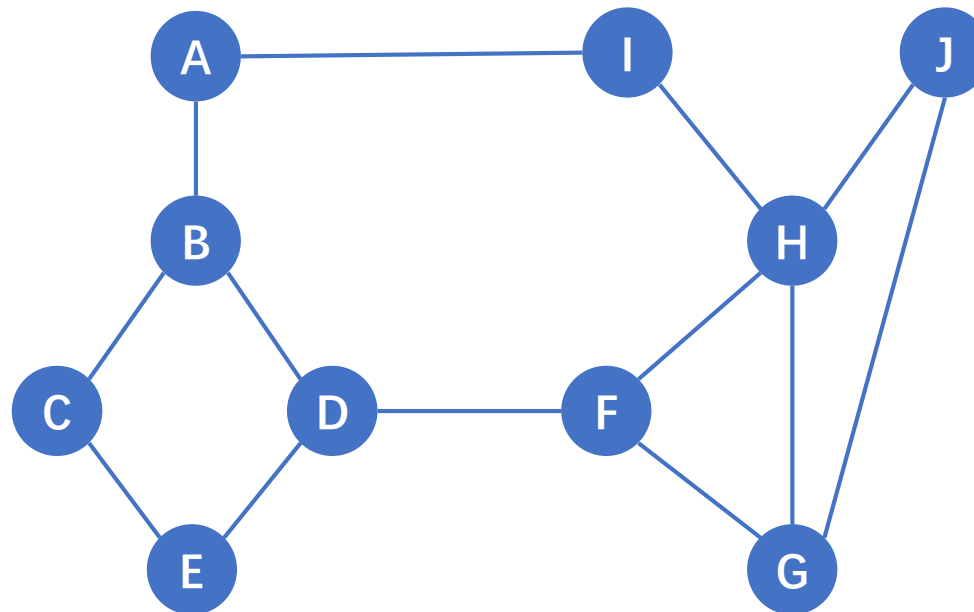
- 无向连通图中，当删去顶点 v 及所有依附于 v 的边后，图的连通分量增加，则顶点 v 为割点



B/D/F/H

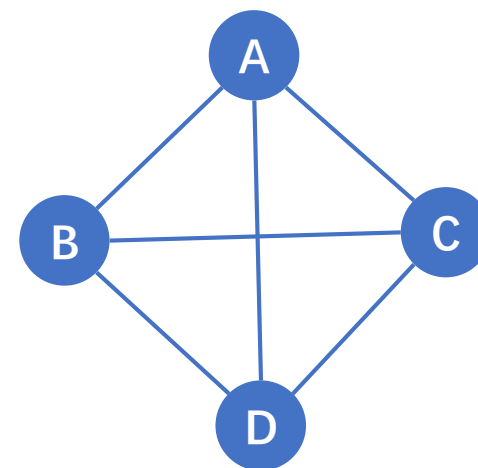
重连通图

- 没有割点的连通图叫做重连通图
- 在重连通图上每对顶点间至少有两条路径
- 比如：在表示通信网络的连通图中不希望有割点



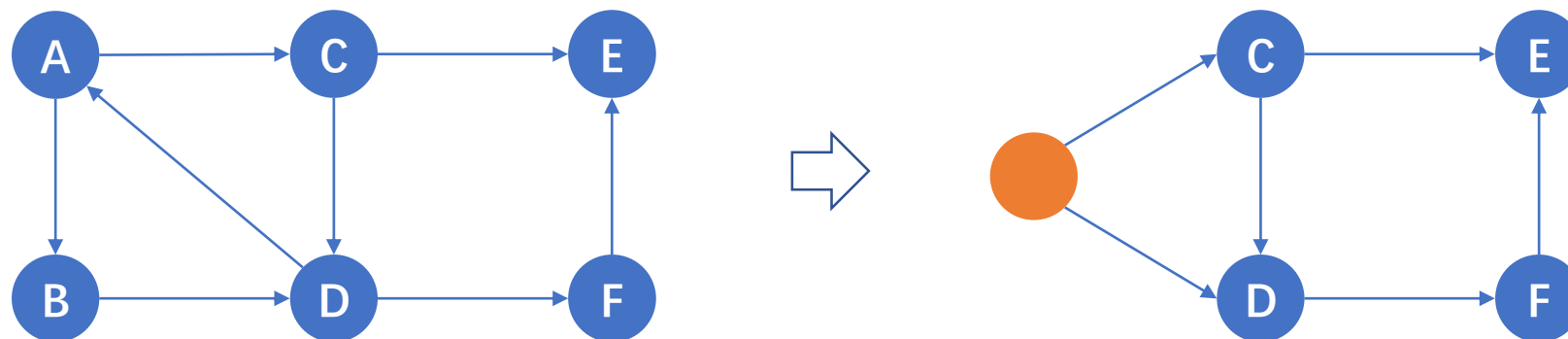
图的其他补充定义

- 和顶点相连的边的数量即为该顶点的度，有向图还分出度、入度
- 一条路径上除了起点和终点外，其他顶点都不相同，为简单路径
- 起点和终点都相同的路径，为环/回路
- 回路中的每个顶点仅被访问一次，为汉密尔顿回路
- 回路中的每条边仅被访问一次，为欧拉回路



有向无环图 (DAG)

- 顾名思义
- 有向无环图的构造可以通过强连通分量缩点来完成



有向无环图 (DAG)

- 动态规划的模型就是DAG，因为：
 1. 要有明确的转移方向（有向）
 2. 要满足无后效性（环会破坏无后效性）

图的存储和遍历

- 详见《搜索专题》

图模型的构建

- 在很多问题描述中，有时能一眼看出这是图论问题，有时不能
- 图论题的难点，反而常常不在算法本身上

例题

- 一个人带了一只狼、一只羊和一筐白菜想要过河
- 河上有一只小船，每次人只能带一样东西。如果人不在时狼就会吃掉羊，羊就要吃白菜
- 问怎样安排渡河，才能做到既把所有东西都带过河，而且在河上往返次数最少？

分析

抽象化:

- 设: M表示人, W表示狼, S表示羊, V表示白菜
- 刚开始可能出现的所有排列:

{WMSV}、{MWS}、{MWV}、{MSV}、{WSV}、{WM}、{MS}、
{MV}、{WS}、{WV}、{SV}、{M}、{S}、{V}、{W}、 $\{\emptyset\}$

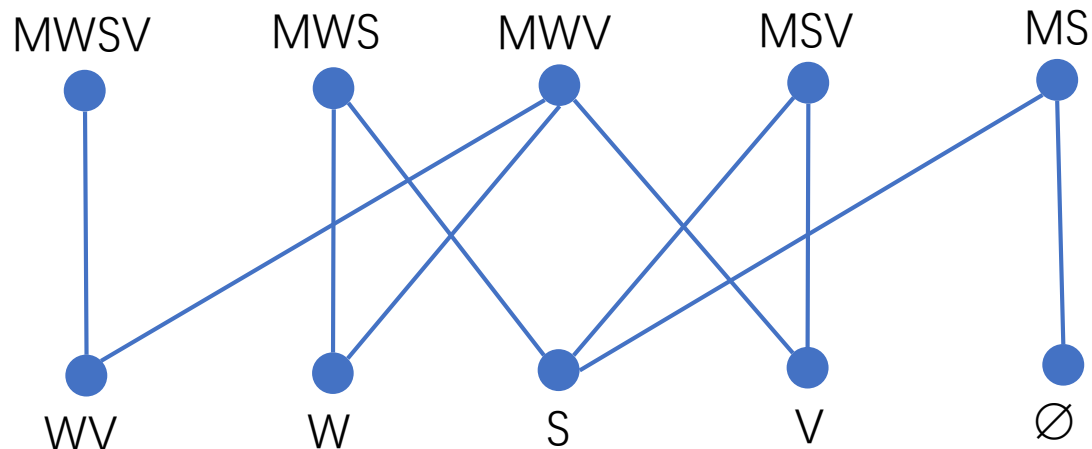
分析

$\{WSV\}$ 、 $\{WS\}$ 、 $\{SV\}$ 、 $\{MW\}$ 、 $\{MV\}$ 、 $\{M\}$

- 以上六种情况是可能出现冲突的，排除掉

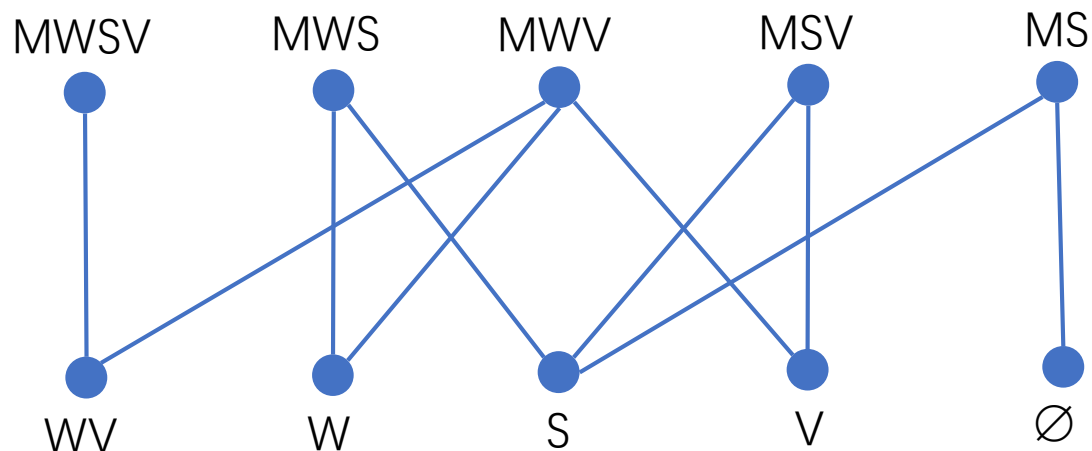
分析

- 将剩下10种情况作为图的顶点，按如下规则连边：如果情况A经过一次渡河可以变成情况B，就在顶点A、B间连一条边



分析

- 问题的求解目标就转化为：在下图中找一条连接顶点MWSV与 \emptyset ，并且包含边数最少的路径
- 把图的边长设为1，渡河问题转化为求顶点MWSV到顶点 \emptyset 的最短路径问题



图的权值

- 图是由点、边、权三大要素构成。有的权表示容量或是流量，它们的运算特征是“串联求最值，并联求和”，即一条路径上最大或是最小的权决定了整条路径的权，而求解目标则是求图中或是两点之间所有路径的权值和
- 有的权表示长度或是时间等等，它们的运算特征是“串联求和，并联求最值”，即一条路径的权由这条路径上每条边的权相加得到，求解目标往往是求图中或是两点之间所有路径的权的最优值

目录

- 最短路
- 最小生成树
- 强连通分量
- 拓扑序
- 二分图与二分图匹配
- 差分约束
- 图的连通性问题

最短路

最短路

- 在图论问题中有一类非常常见的问题：在带权图中需要知道两点之间的最短距离（如果是无权图，也可以视为权值固定为1）。

常用的算法有：

1. Floyd算法
2. Dijkstra算法
3. SPFA算法

松弛操作

- 讲最短路之前，先介绍一下松弛操作，因为松弛操作是大部分最短路算法实现的基础

```
//d[i]为到源点顶点i的最短路估计值  
//w[][]为顶点间的边权值  
  
if (d[u] + w[u][v] < d[v])  
    d[v] = d[u] + w[u][v]
```

松弛操作

- 以上就是一段家喻户晓的代码，随题意会有各种形态上的变化
- 松弛操作其实就是不断更新源点到顶点 v 的最短路径估计值，且能执行松弛操作的都是与顶点 u 相连的点。也就是顶点 u 与顶点 v 之间有边

Floyd算法

- Floyd算法也叫Floyd-Warshall算法，是一种求图中所有点对之间最短路的算法，所以也被称为多源最短路径算法，不过时间复杂度高达 $O(n^3)$
- Floyd算法基于邻接矩阵存储图，并直接更新该矩阵，算法结束以后，将得到一个新的邻接矩阵。这个邻接矩阵中存的是点对之间的最短路

Floyd算法

- Floyd算法的基本思想是：任意两个点对 (i,j) 之间的最短路，要么经过了某个点 k 从而变得更短，要么无需经过点 k 直接相连更短
- 所以先枚举这个点 k

参考代码

- 其实就是个复杂度 $O(n^3)$ 的DP

```
//n为顶点数
for (int k = 1; k < n; ++ k)
    for (int i = 1; i < n; ++ i)
        for (int j = 1; j < n; ++ j)
            if (dis[i][j] > dis[i][k] + dis[k][j])
                dis[i][j] = dis[i][k] + dis[k][j];
```

Floyd算法小结

- 以上代码默认为：如果 (i,j) 没有边，那么 $\text{dis}(i,j)$ 为 inf ，直接在原矩阵上更新，从原来的记录边的矩阵转变为记录最短路的矩阵
- 注意在枚举了新的中间点 k 以后，所有的起、止点 i 、 j 都要重新枚举，所以 k 的循环要放在最外面
- Floyd算法基于动态规划的思想，其本质是通过某个 k 点是否可以使两个点的距离变得更短

Dijkstra算法

- Floyd算法是求所有点对间的最短路，可是实际上你碰到的问题90%以上都是要求某个符合条件的点对间的最短路
- 这个时候Floyd算法显然太浪费了，而且 $O(n^3)$ 的复杂度应付1000个点以上的图就很困难了
- 求某个指定源点出发的最短路的Dijkstra算法，相应地被称为单源最短路径算法

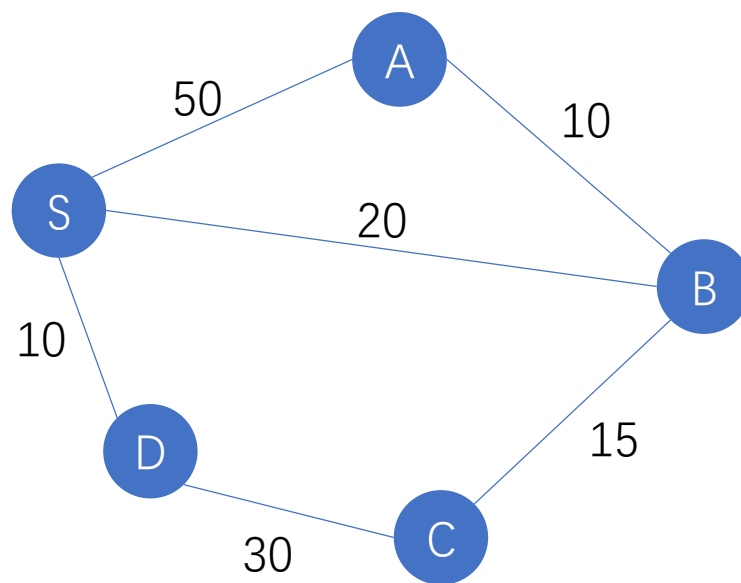
Dijkstra算法

1. 用邻接表储存距离，设一个 visited 数组，标记某个点是否已找到最短路；设一个 dis 数组，储存起点到某点之间的最短路长度
 - 此时起点 $dis=0$ ，其他点 $dis=inf$
2. For 循环遍历每个点，包括起点（允许无序）
3. 在所有没有被访问过的点中找到一个顶点 u 使 $dis[u]$ 最小（最开始当然是起点）
 - 这一步可以堆优化加速

Dijkstra算法

4. 将 `visited[u]` 标记已经确定的最短路
5. For 循环遍历与 `u` 点相连（需要额外储存）且没有确定最短路径的点 `v`
6. 如果 `dis[u]` 加上从 `u` 到 `v` 的边权比 `dis[v]` 小，则更新 `dis[v]`（松弛操作）

Dijkstra算法演示

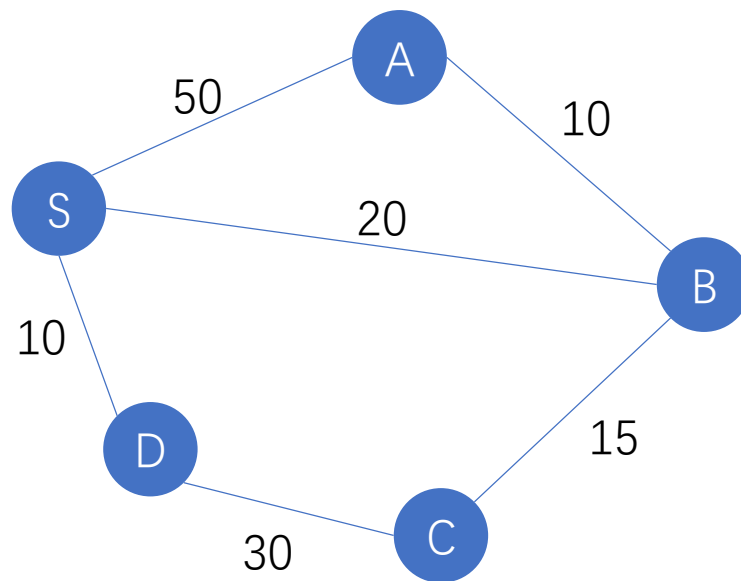


dis

0	50	20	inf	10
---	----	----	-----	----

 { S }

Dijkstra算法演示

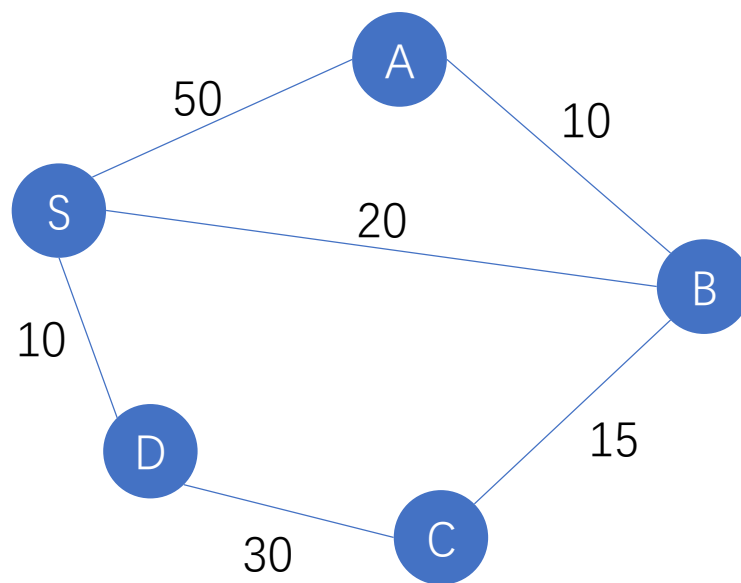


dis

0	50	20	40	10
---	----	----	----	----

 $\{ S, D \}$

Dijkstra算法演示

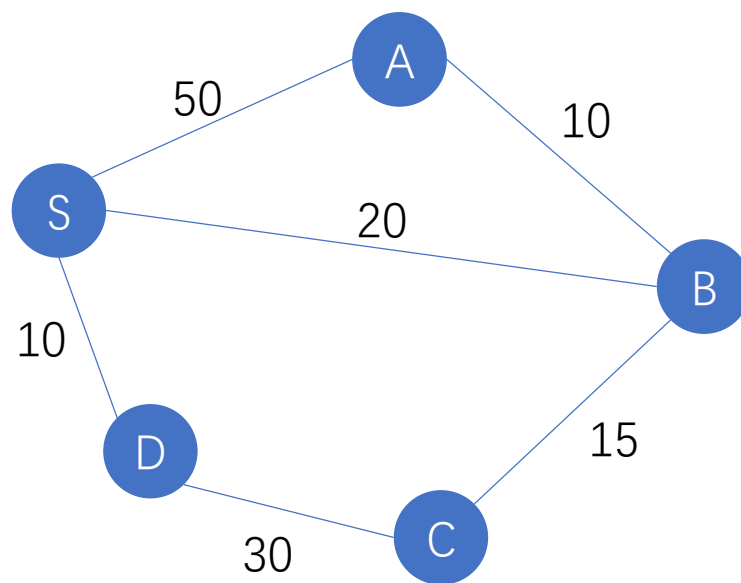


dis

0	30	20	35	10
---	----	----	----	----

{ S, D, B }

Dijkstra算法演示

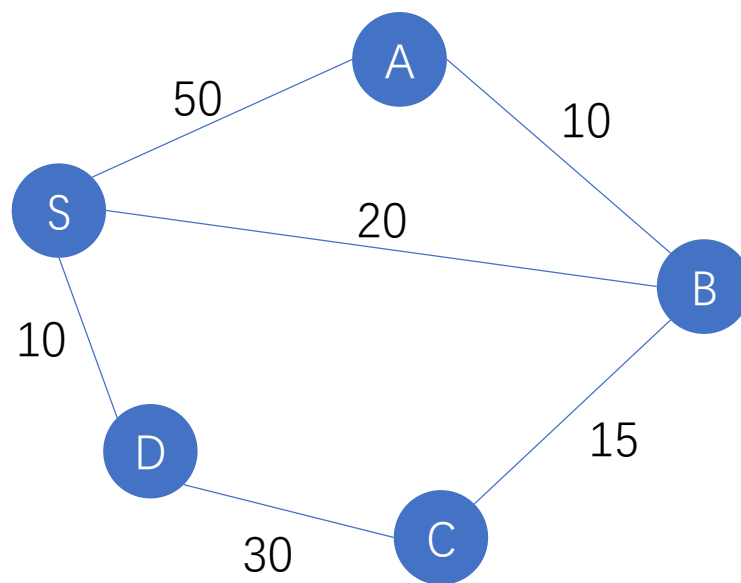


dis

0	30	20	35	10
---	----	----	----	----

{ S, D, B, A }

Dijkstra算法演示



dis

0	30	20	35	10
---	----	----	----	----

{ S, D, B, A, C }

Dijkstra算法

- Dijkstra 算法建立在一个前提上： Dijkstra 要保证已经拓展到的点得到的已经是目前为止的最短路，它不会在以后再被更新，这样才能继续往下拓展
- 这样的做法有一个很大的缺陷： Dijkstra 是不能处理负边的
- 除去不能处理负边这一点外， Dijkstra 还是很稳定而高效的算法

参考代码

```
int dijkstra(int a, int b)
{
    bool visited[1001];
    memset(visited, false, sizeof(visited));
    int now;
    for (int i = 1; i <= n; ++ i) dis[i] = INF;
    dis[a] = 0;
    for (int i = 1; i <= n; ++ i)
    {
        int minx = INF;
        for (int j = 1; j <= n; ++ j)
            if (!visited[j] && w[a][j] < minx)
                minx = dis[j], now = j;
        visited[now] = true;
        for (int j = 1; j <= n; ++ j)
            if (!visited[j] && (dis[j] > dis[now] + w[now][j]))
                dis[j] = dis[now] + w[now][j]; // 松弛操作
    }
    return dis[b];
}
```

Bellman-Ford算法

- Bellman-Ford算法也是单源最短路算法，不算太常用。但因为SPFA算法是在其基础上优化而来，所以先介绍一下

Bellman-Ford算法

- Bellman-Ford算法的基本思想是：两点间如果有最短路，那么这条路径上的每个节点最多经过一次。也就是说，这条最短路不超过 $n-1$ 条边

Bellman-Ford算法

- 证明这个结论：
- 如果一个节点经过了两次，那么我们走了一个环
 - 如果这个环的权值和为正，显然不划算
 - 如果是负环，那么经过该节点的最短路不存在
 - 如果是零环，去掉不影响最优值

Bellman-Ford算法

- 最短路是符合最优子结构性质的，路径边数上限为 k 时的最短路由边数上限为 $k-1$ 时的最短路“加一条边”来求
- 而根据刚才的结论，最多只需要迭代 $n-1$ 次就可以求出最短路

Bellman-Ford算法

- 考虑长度不超过 $1, 2, 3 \dots k-1$ 的路，标号为 $d[]$ 。长度为 k 的最短路可以由不超过 $1, 2, \dots k-1$ 的路增加一条边得到：

```
for (i = 1; i <= m; ++ i) dis[i] = INF; // 初始化dis数组
dis[1] = 0;

for (k = 1; k <= n - 1; ++ k)
    for (i = 1; i <= m; ++ i)
        if (dis[v[i]] > dis[u[i]] + w[i]) // 松弛操作
            dis[v[i]] = dis[u[i]] + w[i];
```

SPFA算法

- SPFA算法实际上是Bellman-Ford算法的一个改进版本
- Bellman-Ford算法中间的每一步都是对于所有的边进行松弛操作。但是真的有必要对所有的边进行松弛操作吗？

SPFA算法

- 答案是否定的
- 事实上，每一次只需要对路径的权值在上一次改变的点所连接的边进行松弛操作就可以了
- 因此，可以用一个队列来记录需要松弛的节点，然后仿照Bellman-Ford来不断的迭代就可以了

SPFA算法

1. 建立一个先进先出的优先队列，初始时队列里只有起始点
2. 再建立一个数组记录起始点到所有点的最短路径（该数组的初始值赋为inf，该点到它本身的路径赋为0）
3. 然后执行松弛操作，用队列里的点去刷新起始点到所有点的最短路，如果刷新成功且被刷新点不在队列中，则把该点加入到队列
4. 重复执行直到队列为空

SPFA算法演示

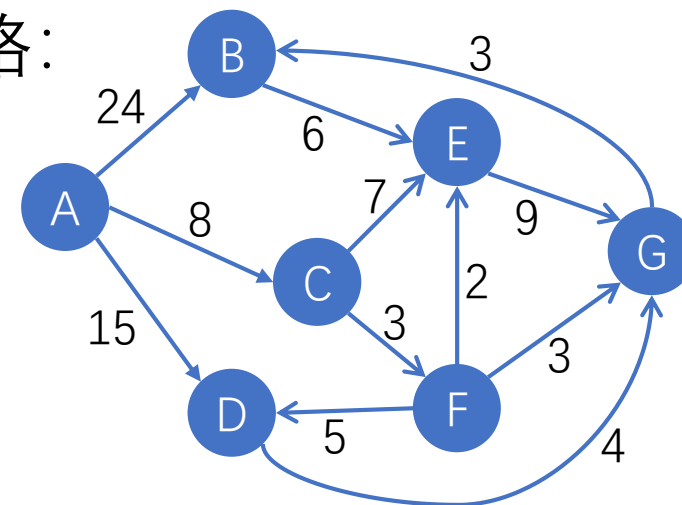
- 求A到G的最短路。
- 首先建立源点A到其余各点的最短路径表格：

	A	B	C	D	E	F	G
d[i]	0	∞	∞	∞	∞	∞	∞

- A出队，对B、C、D点进行松弛操作：

	A	B	C	D	E	F	G
d[i]	0	24	8	15	∞	∞	∞

- 三个点的最短路估值都变小，B、C、D入队



SPFA算法演示

- B出队，对E点进行松弛操作：

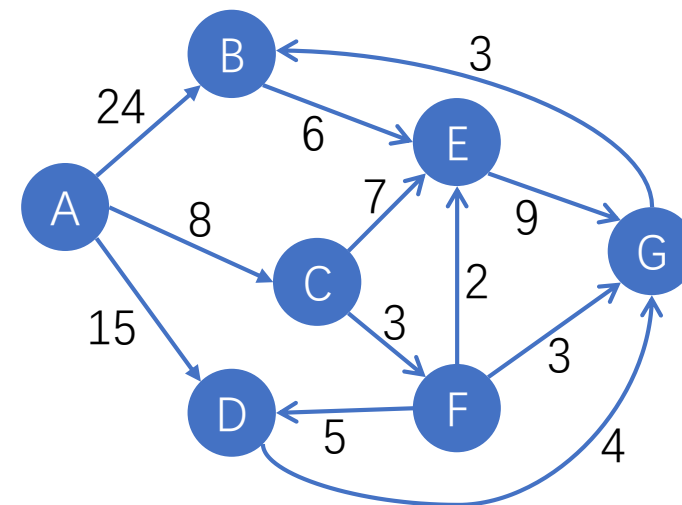
	A	B	C	D	E	F	G
d[i]	0	24	8	15	30	∞	∞

E的最短路估值变小，E入队

- C出队，对E、F点进行松弛操作：

	A	B	C	D	E	F	G
d[i]	0	24	8	15	15	11	∞

- E、F的最短路估值变小，F入队。此时队列中的元素为D、E、F



SPFA算法演示

- D出队，对G进行松弛操作：

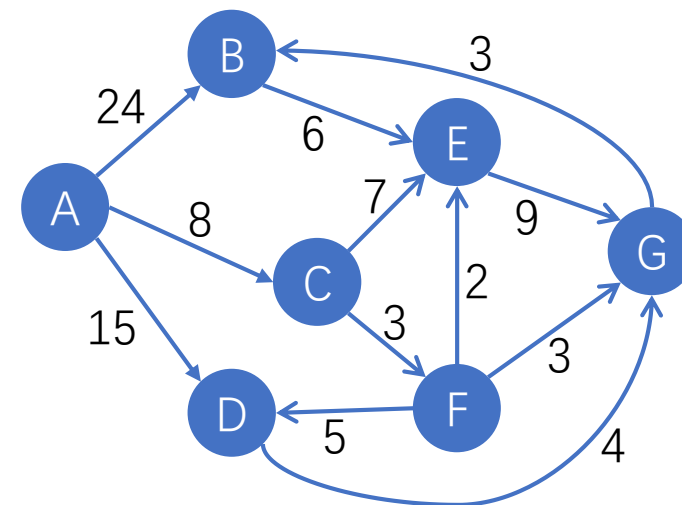
	A	B	C	D	E	F	G
d[i]	0	24	8	15	15	11	19

G的最短路估值变小，G入队

- E出队，对G进行松弛操作：

	A	B	C	D	E	F	G
d[i]	0	24	8	15	15	11	19

- G的最短路估值未变小。此时队列中的元素为F、G



SPFA算法演示

- F出队，对D、E、G进行松弛操作：

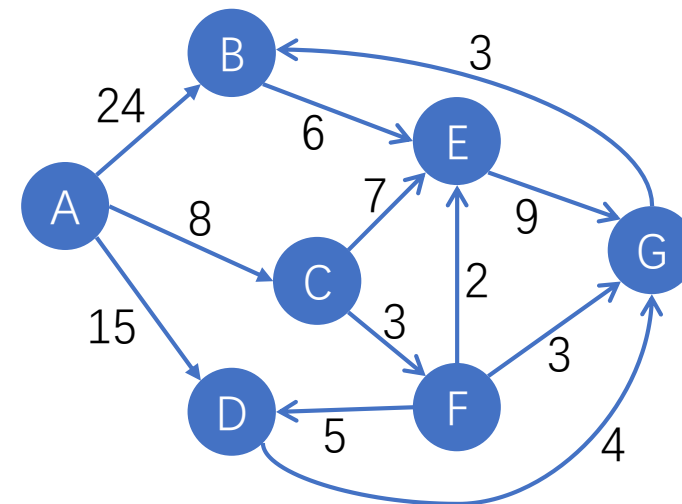
	A	B	C	D	E	F	G
d[i]	0	24	8	15	13	11	14

E、G的最短路估值变小，E入队。

- G出队，对B点进行松弛操作：

	A	B	C	D	E	F	G
d[i]	0	17	8	15	13	11	14

- B点的最短路估值变小，B点入队。此时队列中的元素为E、B



SPFA算法演示

- E出队，对G进行松弛操作：

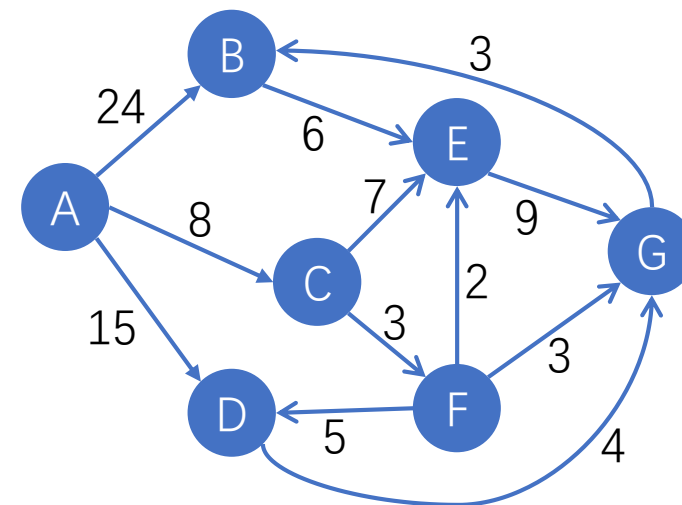
	A	B	C	D	E	F	G
d[i]	0	17	8	15	13	11	14

G的最短路估值未变小。此时队列中的元素为B

- B出队，对E点进行松弛操作：

	A	B	C	D	E	F	G
d[i]	0	17	8	15	13	11	14

- E点的最短路估值未变小，此时队列空。最终A到G的最短路为14



参考代码

```
int spfa(int a, int b)
{
    queue<int> que;
    bool exist[1001];
    memset(exist, false, sizeof(exist));
    for (int i = 1; i <= n; ++i) dis[i] = INF;
    dis[a] = 0;
    int now = a;
    que.push(now);
    do {
        now = que.front();
        que.pop();
        exist[now] = false;
        for (int i = 1; i <= n; ++i)
        {
            if (dis[i] > dis[now] + w[now][i])
                dis[i] = dis[now] + w[now][i];
            if (!exist[i])
            {
                que.push(i);
                exist[i] = true;
            }
        }
    } while (!que.empty());
    return (dis[b]);
}
```

处理负环

- 既然图中有可能出现负边，那就有可能产生负环(就是存在某条回路，而这条回路上的边权值和为负)
- 如果图中有负环的话，是不存在最短路的。因为每在环中跑一圈，路径总长会变得更小

处理负环

- Dijkstra算法因为不能处理负边，所以也不能判断是否有负环存在
- 有一种直观的想法是如果一个点总是被更新，那么这个图很可能有负环
- 在SPFA时，如果发现一个点被加入队列中超过 n 次，则这个图中肯定存在负环

处理负环

- 当题目明确要求只是要你判断是否有负环（而不要求在不存在负环的情况下求最短路）时，跑一遍SPFA比较浪费
- 还可以有一种简单的DFS判断方法？

SPFA算法小结

- SPFA与Dijkstra在某些方面有相同之处，都是用已扩展的去更新未扩展的。但是SPFA做了一个折中：
- Dijkstra认为扩展以后就已经是最短路了，因此它必须暴力或者用堆来保证每次扩展新的节点出来，都满足这个性质

SPFA算法小结

- SPFA与Dijkstra在某些方面也有相同之处，都是用已扩展的去更新未扩展的。但是SPFA做了一个折中：
- 而SPFA不同，它允许一个点被多次更新，在一个点被更新以后，显然它有可能再去更新别的点，那么我们把它加入队列，并且打个“它在队列中”的标记（避免它还没从队列中出来就再次被更新，导致加入队列两次）

最短路算法的比较

	Folyd	Dijkstra	Bellman-Ford	SPFA
时间复杂度	$O(V^3)$	$O(V^2)$	$O(EV)$	$\approx O(2E)$
代码量与难度	简易	较大	适中	适中
所需数据结构	无	堆	无	优先队列
可否处理负边	可以	否	可以	可以
实用性	效率低, 不常用	效率适中, 常用	效率低, 不常用	效率适中, 常用

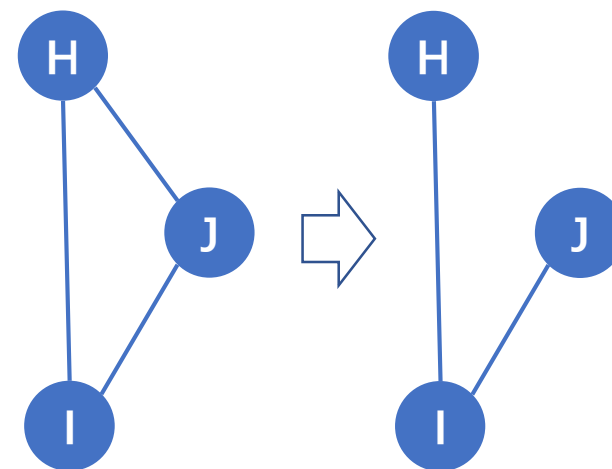
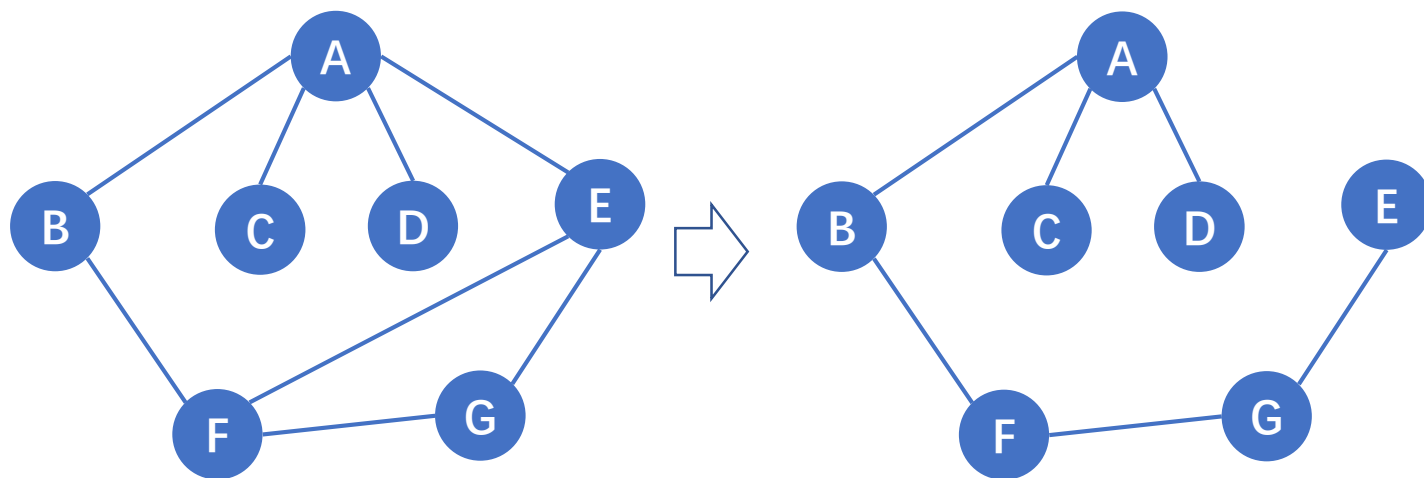
- SPFA算法的时间复杂度 $O(kE)$, k 为所有顶点进队的平均次数, 可以期望在完全图中 $k \leq 2$

最小生成树 (MST)

生成树

- 非连通图的极大连通子图中，全部顶点 (m 个) 加上顶点遍历过程中经过的边 ($m-1$ 条) 构成的子图，称为该图的生成树
- 生成树中是不能有回路的

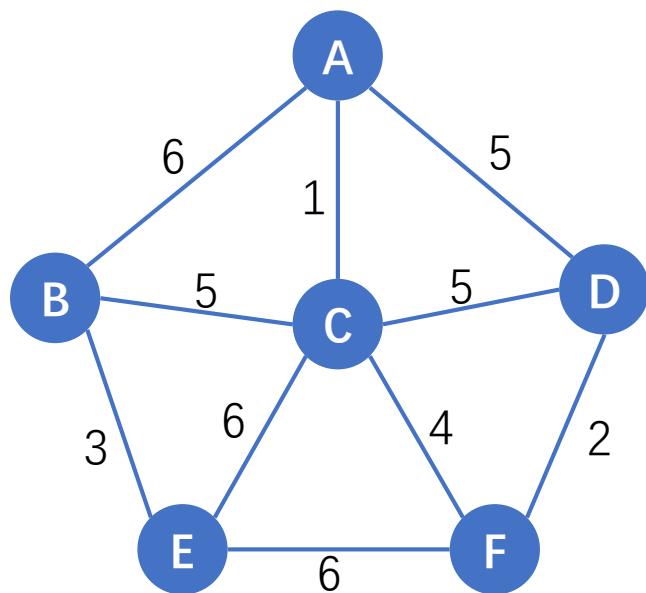
生成树



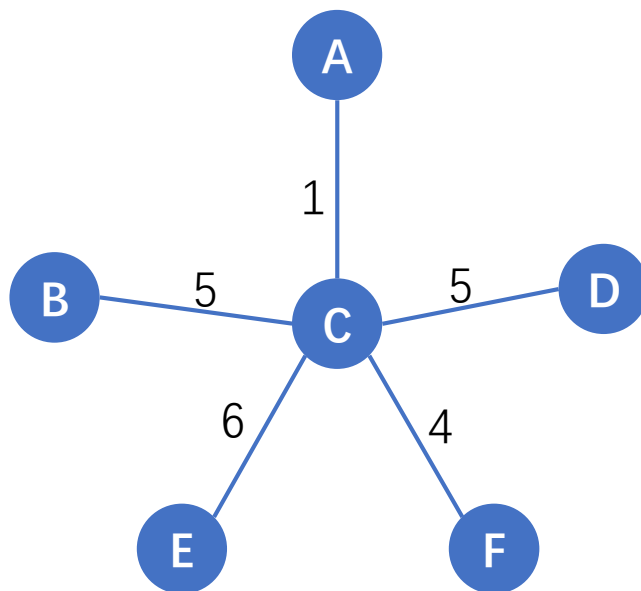
最小生成树 (MST)

- 对于一个带权图，在其所有的生成树中，树上的边权和最小的那一棵称为最小生成树
- 最小生成树有可能不唯一

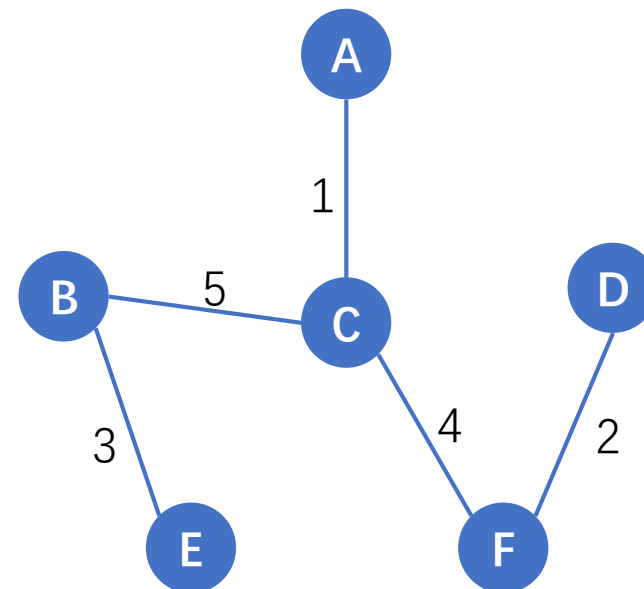
最小生成树 (MST)



带权图



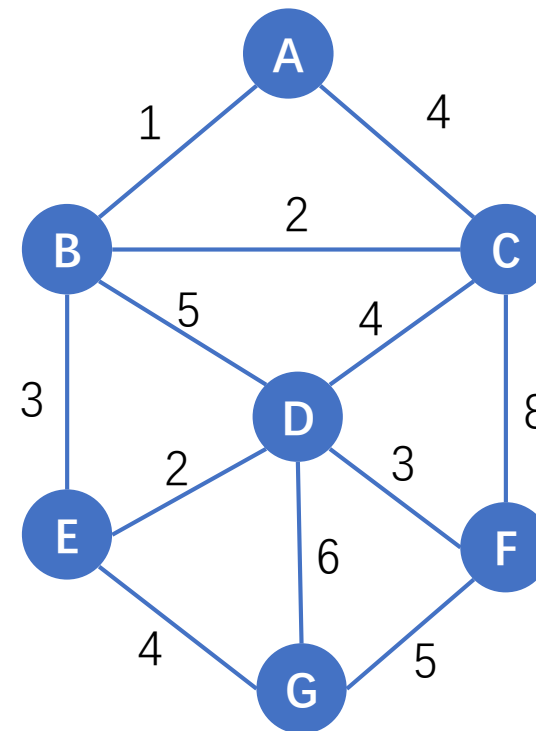
生成树



最小生成树

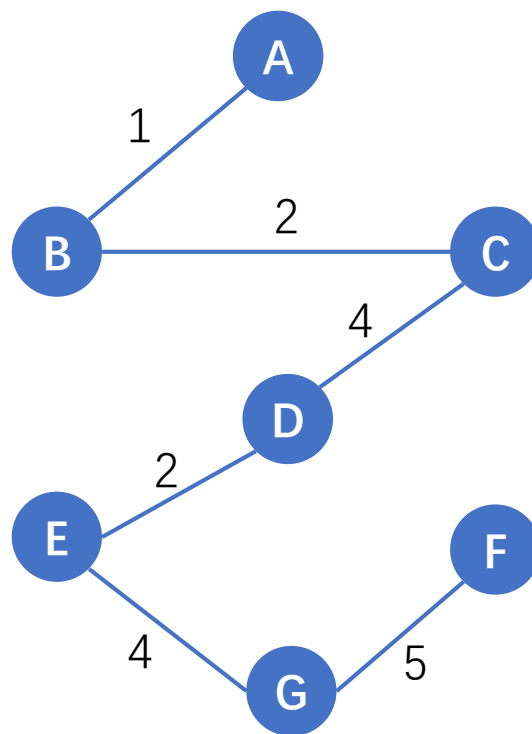
提问

- 该图为一双向连通带权图，画出从顶点A出发的：
 1. 深度优先生成树
 2. 广度优先生成树
 3. 最小生成树



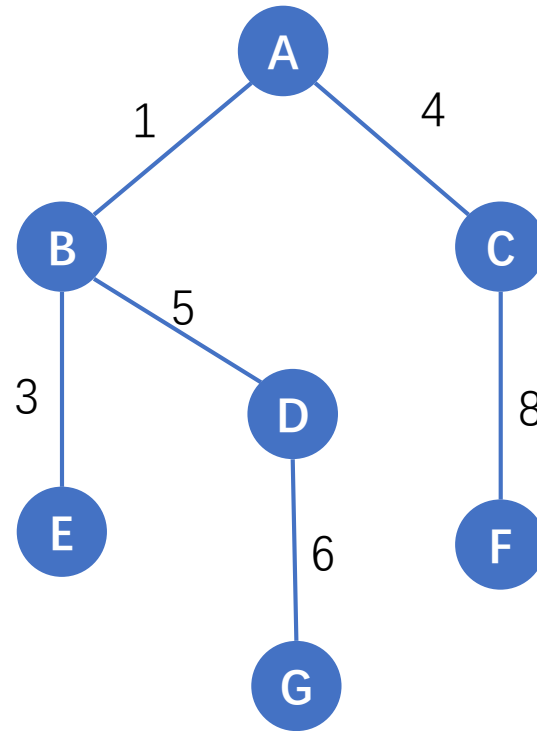
回答

- 深度优先生成树



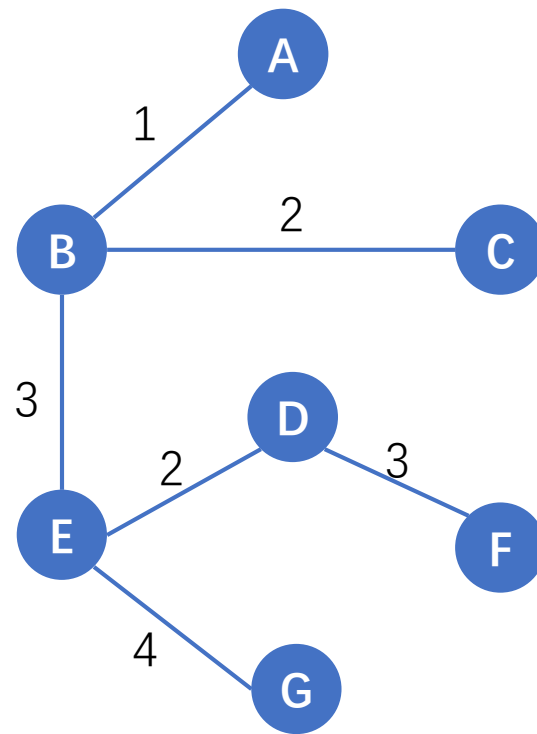
回答

- 广度优先生成树



回答

- 最小生成树



哪些题属于MST问题

- 假设要在 n 个城市之间建立通讯联络网，则连通 n 个城市只需要修建 $n-1$ 条线路，如何在最节省经费的前提下建立这个通讯网。
诸如此类

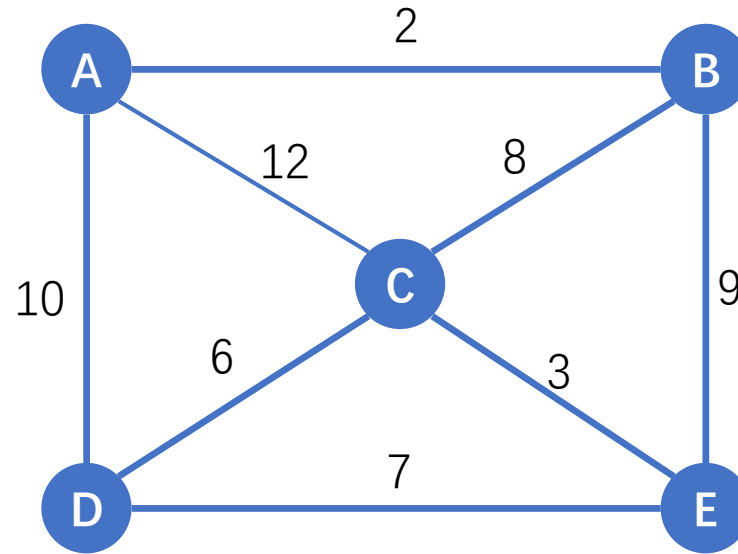
最小生成树

- 常用的算法有：

1. Prim算法

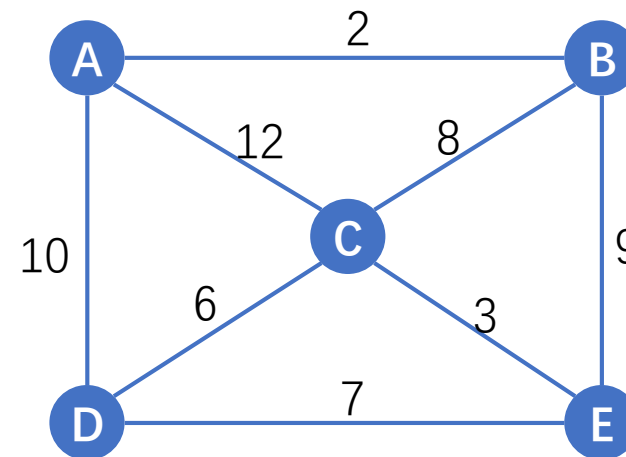
2. Kruskal算法

Prim算法演示



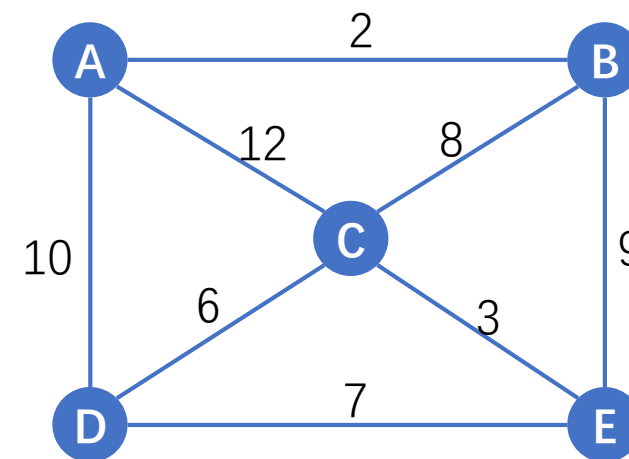
Prim算法

1. 建立两个顶点集合V1和V2，分别表示已加入和未加入生成树的顶点集合
2. 从V2中选择第一个顶点A加入到V1
3. 计算A到V2中其他所有顶点的权：
 - $\text{cost}[A][B]=2$
 - $\text{cost}[A][C]=12$
 - $\text{cost}[A][D]=10$
 - $\text{cost}[A][E]=\text{inf}$



Prim算法

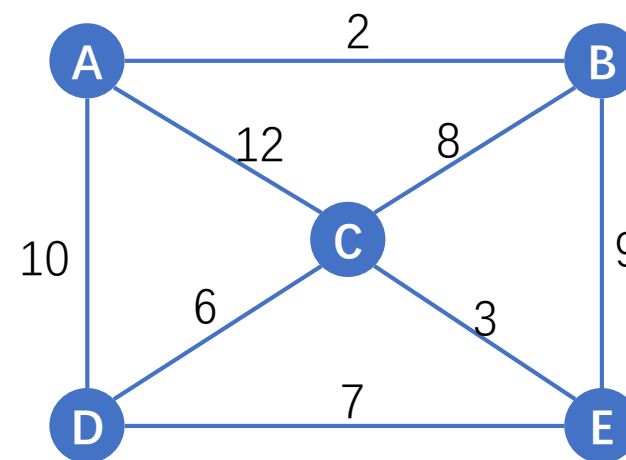
4. 选择边[A][B]加入到生成树, V1和V2中更新B
– mincost += cost[A][B] = 2



Prim算法

5. 从B出发重复这一过程。但是在计算B与V2中剩余顶点的边权时，是在刚才计算A与剩余顶点边权的基础上做更新。即如果从B到某个顶点的权比从A到该顶点的权更小时才更新

- $\text{cost}[B][C]=8$ ，更新 $\text{cost}[A][C]$;
- $\text{cost}[B][D]=\text{inf}$ ， $\text{cost}[A][D]$ 不更新;
- $\text{cost}[B][E]=9$ ，更新 $\text{cost}[A][E]$

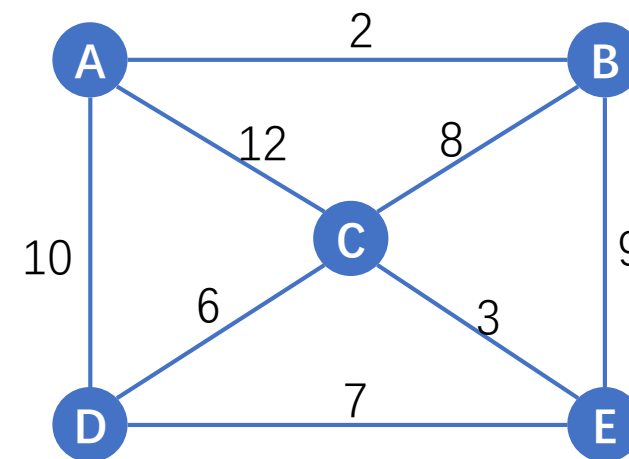


Prim算法

6. 显然最小的是 $\text{cost}[A][C]$, 那么选择边 $[B][C]$ 加入到生成树

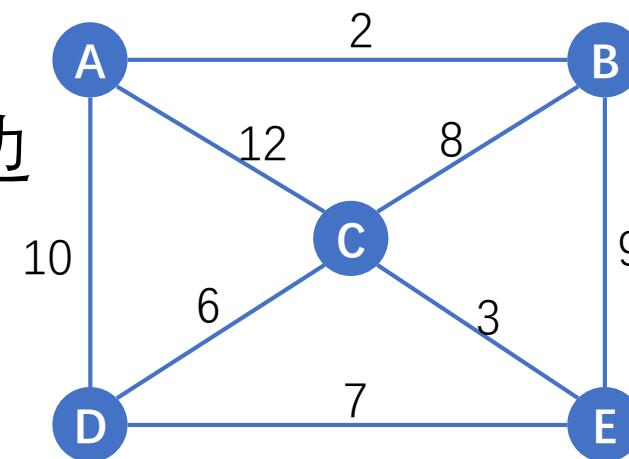
– $\text{mincost} += \text{cost}[B][C] = 2 + 8$

7. C入V1, 重复直到V2空



正确性

- 为什么这样连边是对的？
- Prim算法每次从已有的集合V1中扩展出一条边和一个顶点，同时也是从V1向外扩展代价最小的边
- 如果以后继续扩展，这个代价也不会变得更小
- 所以这条边必定是最终的最小生成树中的一条边



优化

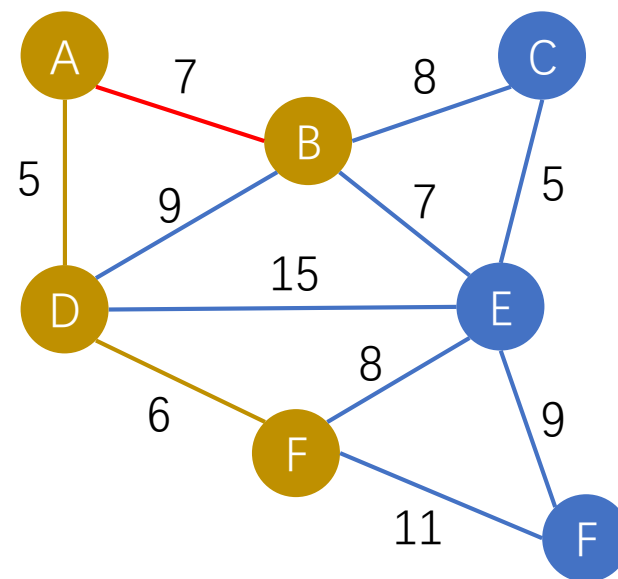
- 在选取最小边这一步时，如果暴力查找所有点对，复杂度是 $O(n^2)$ 的
- 常见的优化是用堆优化，可以将复杂度降低到 $O(n \log n)$

Prim算法和Dijkstra算法

- 看到这里你一定联想到了什么
- Prim算法和求最短路的Dijkstra算法十分相似。区别只在一个地方：
 - 在Prim算法里，我们计算的是一个集合到另一个集合的一些点的距离，从中取出最短的边
 - 在Dijkstra算法里，由于我们要算的是某个点到其他点的最短路，因此在取出候选点的时候，需要计算原先的最短路

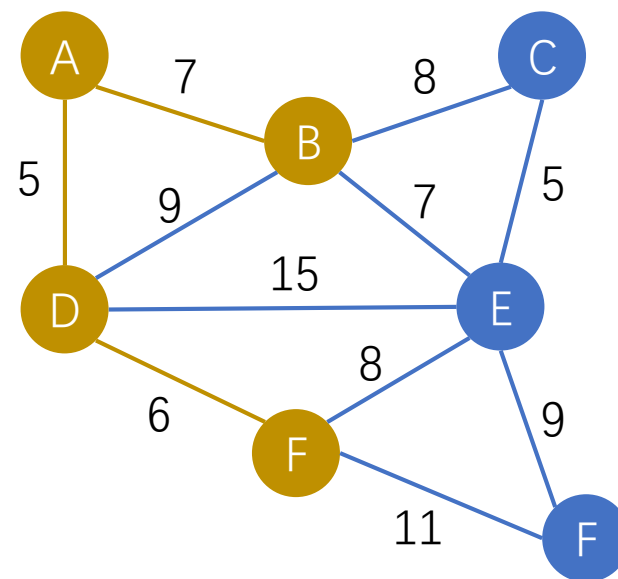
Prim算法和Dijkstra算法

- 以右图为例：
- 在Prim中，我们选了(A,B)这条边，因为它是{A,D,F}这个点集离B最短的一条边

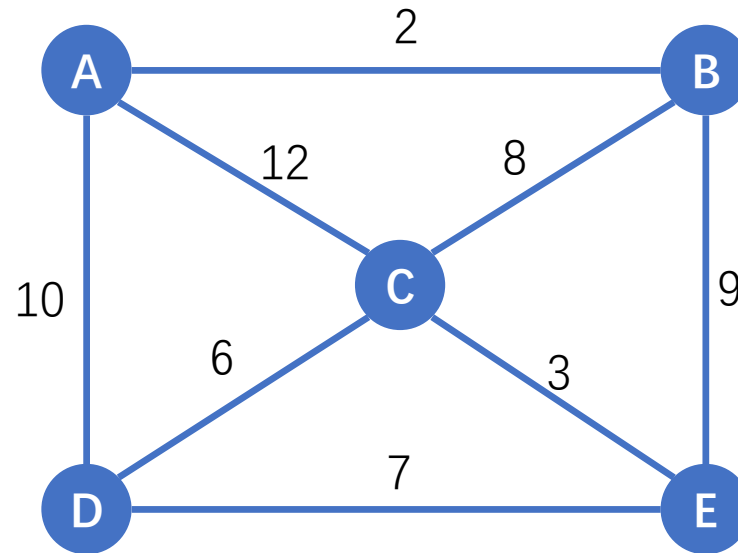


Prim算法和Dijkstra算法

- 以右图为例：
- 但是如果以顶点D为源点，
(D,B)之间的最短路应该是9而不是12
- 也就是Dijkstra的时候，我们将顶点A加入集合中更新其他点的时候，需要将A本身的最短路考虑进来

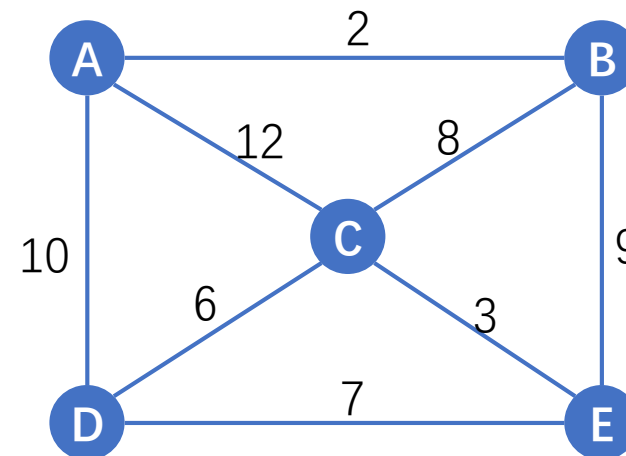


Kruskal算法演示



Kruskal算法

1. 开始的时候使所有的顶点单独构成集合V2，另有空顶点集合V1
2. 对这n个顶点的边按权值进行排序，从权值最小的边开始：如果该条边的两个顶点分别在集合V1、V2中，则选入，并更新集合V1、V2
3. 重复直到遍历所有的边



正确性

- 证明Kruskal算法的正确性：
- 假如选取的某条边 E_1 不在最后的最小生成树中，但这条边连接的两个连通分量（具体到这里就是两个顶点集合）还是要通过某条边 E_2 连接起来的（如果不是这样就不能构成生成树）

正确性

- 证明Kruskal算法的正确性：
- 那这另一条边E2和我们选取的E1会和这两个连通分量一起共同构成一个环
- 环中一定会存在一条权值比E1大的边E3（E3有可能就是E2，也可能不是）可替代E1（因为在这假设中E1不是最小的），使得分量仍保持连通。这样得到的包括E3的生成树就不是最小了
- 所以E1一定是最后的最小生成树中的一条边

优化

- 如何快速有效地判断所选边的顶点是否分属两个集合？
- 借助并查集
- Kruskal效率高，而且好写

Prim算法和Kruskal算法

- Prim算法为单一源头开始不断寻找连接的最短边，从而向外扩展，其中间结果是一棵树
- Kruskal算法则是不断寻找最短边，然后不断将集合合并，其中间结果是森林

Prim算法和Kruskal算法

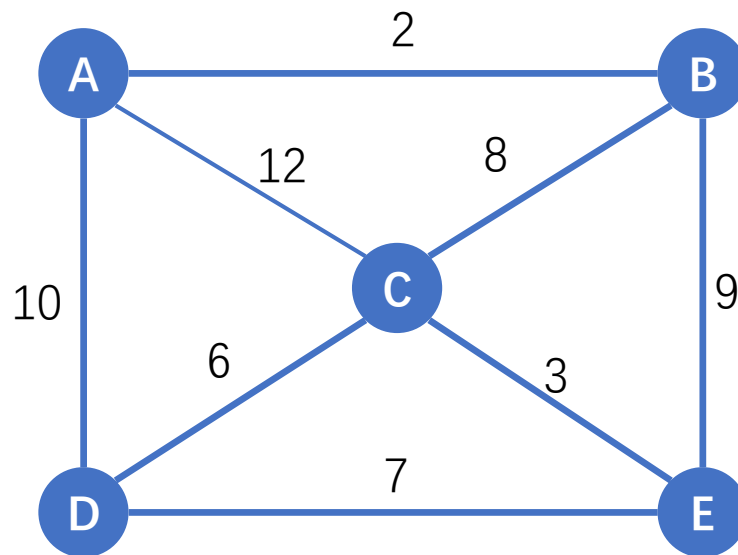
- Prim算法在 $O(n^2)$ 的复杂度下可以用邻接矩阵实现, 在 $O(n\log n)$ 的复杂度下用链式前向星实现
- Kruskal算法则需要定义一个struct保存每条边的信息

Prim算法和Kruskal算法

- 两种算法都基于贪心，一般不会出现测试数据针对卡某一种算法的情况
- Kruskal是基于边的算法，在处理稀疏图的时候效率更高

破圈算法演示

- 此外还有一个略冷门的破圈法



课后练习

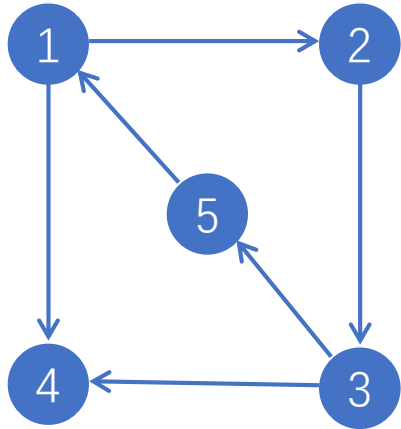
- luogu 1195 口袋的天空
- luogu 1991 无线通讯网
- luogu 1111 修复公路

强连通分量 (SCC)

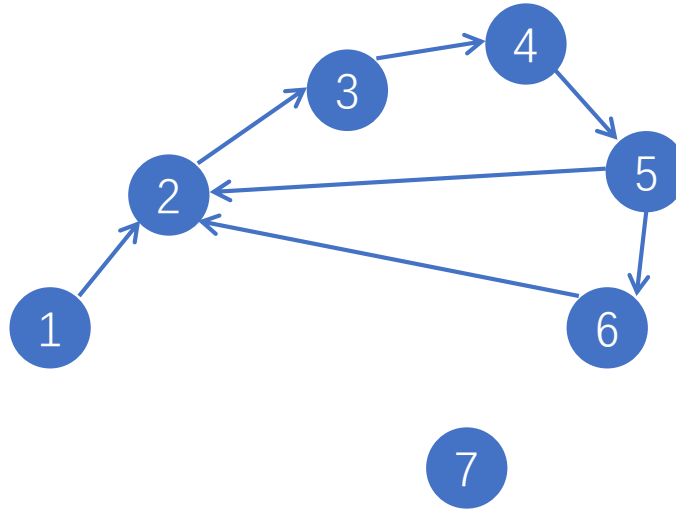
强连通分量 (SCC)

- 前面给的定义：有向非连通图的极大强连通子图，为强连通分量
- 在有向图中，如果顶点 (u,v) 间有一条从 u 到 v 的路径，同时还有一条从 v 到 u 的路径，则称两个顶点 u 和 v 是强连通的
- 如果有向图中任一对顶点都满足这样的性质，就称为强连通图
- 如果有向图的子图(由部分顶点和边构成的图)满足这样的性质，称为强连通分量

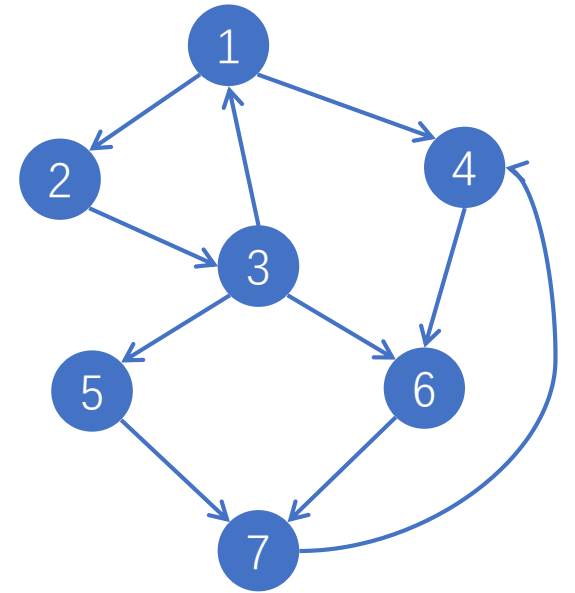
强连通分量 (SCC)



{1、2、3、5}、{4}



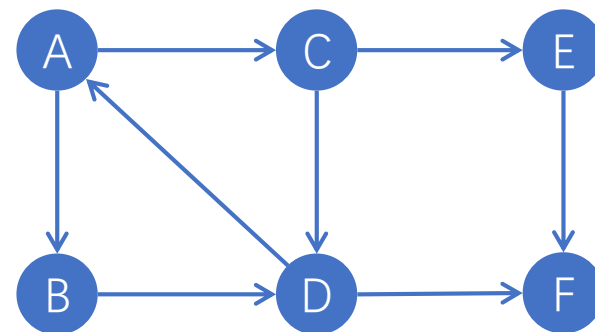
{1}、{2、3、4、5、6}、{7}



{1、2、3}、{4、6、7}、{5}

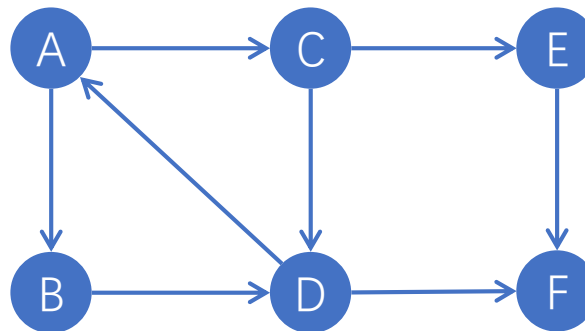
强连通分量怎么求

- 暴力怎么求?
- DFS 第一次
- 将有向边方向反过来 DFS 第二次
- 取重合点 (交集)
- 时间复杂度复杂度 $O(V^2+E)$



Tarjan算法

- Tarjan算法可以求出有向图的所有极大强连通分量
- 其主要做法是通过DFS寻找强连通分量，复杂度可以达到 $O(V+E)$
- 在树上做DFS是很方便的，因为你要么往儿子节点走，要么往父亲节点回溯
- 但是图呢？



Robert Tarjan，计算机科学家，以LCA，强连通分量等算法闻名。

Tarjan算法

- 一个原则就是：可以访问的点就去访问，已经访问过的点就跳过
- 如此一来,我们就可以把图当成树一样的DFS
- 此时，每个强连通分量相当于DFS树中的一棵子树

Tarjan算法

- DFS时，把当前树中未处理的节点加入一个栈，回溯时可以判断栈顶到栈中的节点是否为一个强连通分量
- Tarjan算法的实质是将图转成树，然后在树上解决问题

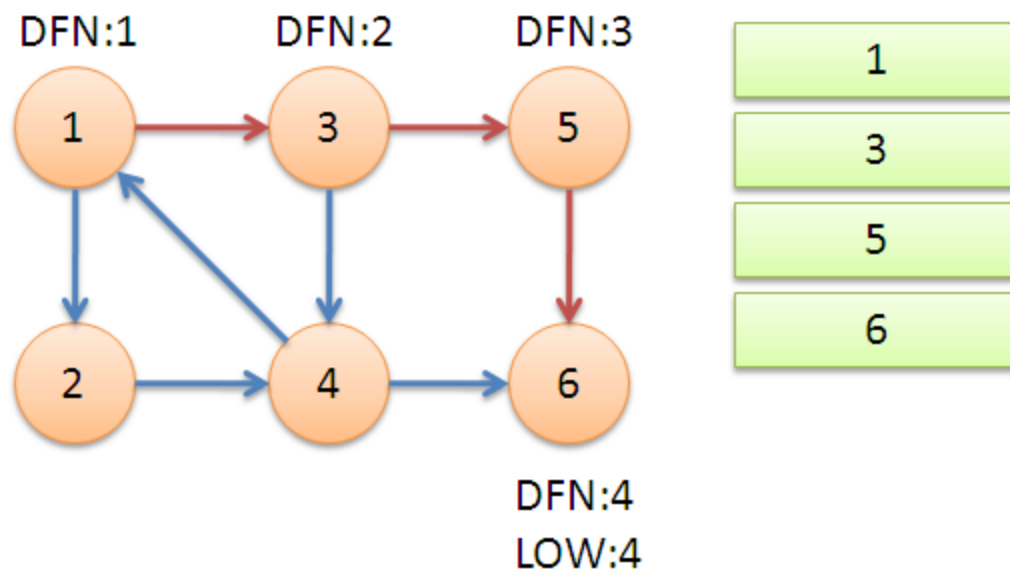
Tarjan算法

- 我们定义DFN[u]为节点u搜索的次序编号（时间戳），Low[u]是一个标记数组，为u或u的子树能够追溯到的最早的栈中节点的次序编号

$$\text{Low}[u] = \min \{ \text{Low}[u], \text{Low}[v] \} \text{ (不管 } v \text{ 有没有dfs过, 但要求 } v \text{ 必须在栈内) }$$

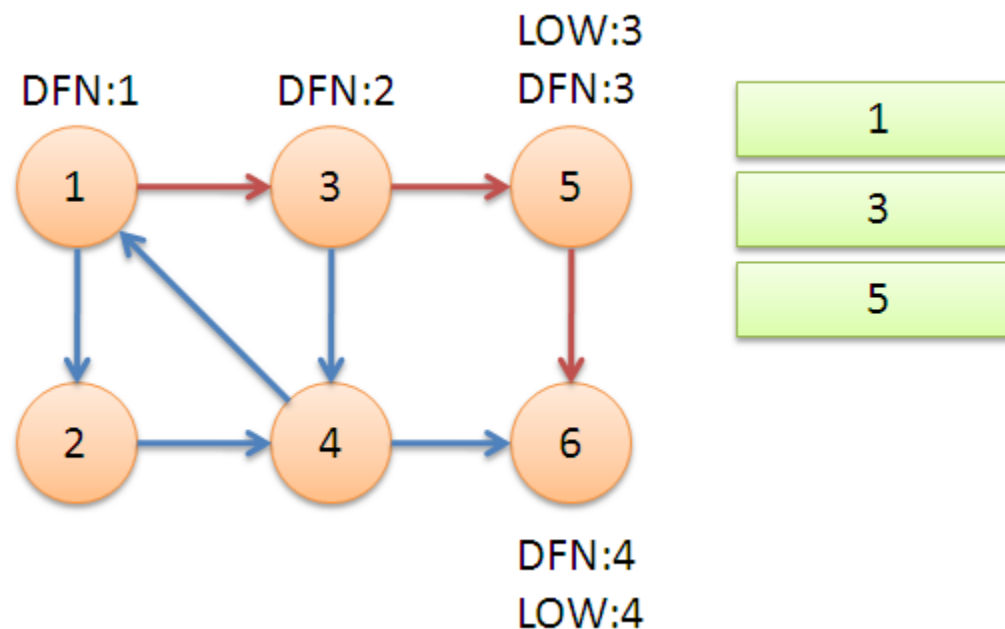
- 刚访问到某节点u的时候，初始化Low[]=DFN[]。DFS回溯时维护Low[]
- 维护之后，当DFN[u]=Low[u]时，以u为根的搜索子树上所有节点构成一个强连通分量

Tarjan算法演示



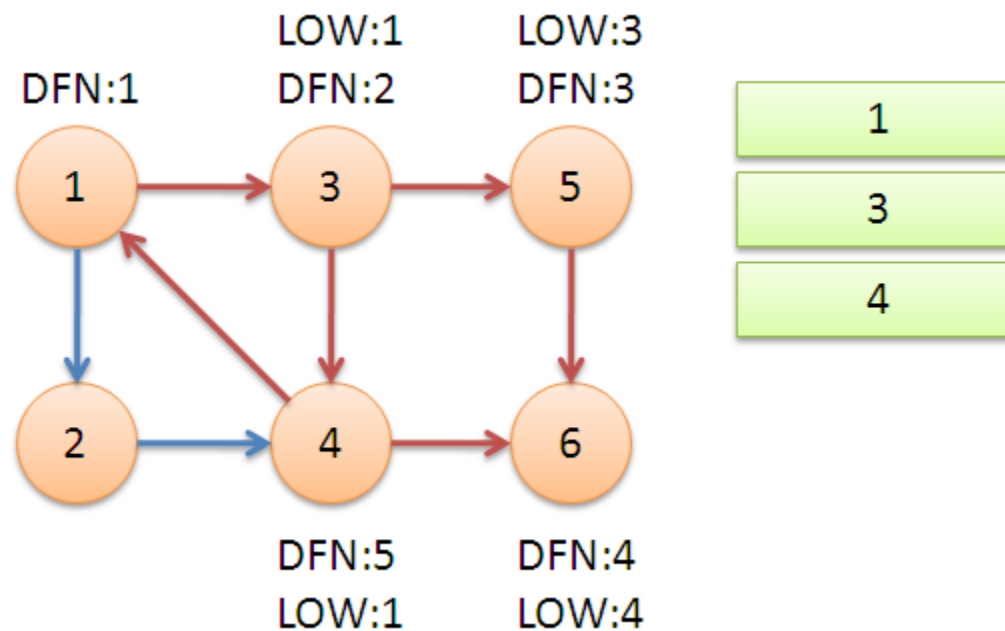
- 节点6退栈, 退栈时检查发现 $DFN(4)=LOW(4)$, 节点6是一个强连通分量

Tarjan算法演示



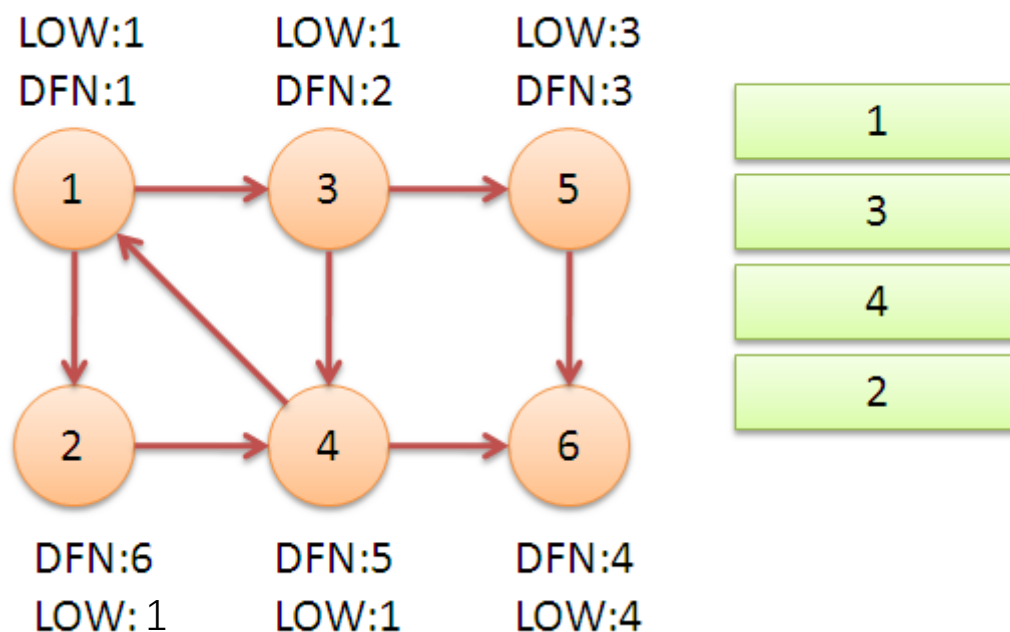
- 节点5退栈, 退栈时检查发现 $DFN(3)=LOW(3)$, 节点5也是一个强连通分量。

Tarjan算法演示



- 返回节点3，搜索到节点4，进栈
- 发现节点4有边连向节点1，而节点1还在栈中，所以 $LOW[4] = DFN[1] = 1$
- 而节点3有边连向节点4，节点4在栈中，所以 $LOW[3] = LOW[4] = 1$

Tarjan算法演示



- 返回节点1，搜索到节点2，进栈
- 发现节点2有边连向节点4，而节点4还在栈中，所以 $LOW[2]=DFN[4]=1$
- 返回节点1，节点1的 $DFN[1]=LOW[1]=1$ ，全部出栈，1、3、4、2为第三个强连通分量

参考代码

```
void tarjan(int u)
{
    DFN[u] = Low[u] = ++ index //为节点u 设定编号和Low初值
    Stack.push(u)             //节点u进栈
    for each(u, v) in E       //枚举每条边
        if (v is not visited) //如果节点v 未被访问过
            tarjan(v)          //继续向下找
            Low[u] = min(Low[u], Low[v])
        else if (v in Stack)   //如果节点v 还在栈内
            Low[u] = min(Low[u], DFN[v])
    if (DFN[u] == Low[u])      //如果节点u 是强连通分量的根
        do
            v = Stack.pop     //v退栈, 为该强连通分量的一个顶点
            print v
        while (u == v)
}
```

Tarjan算法小结

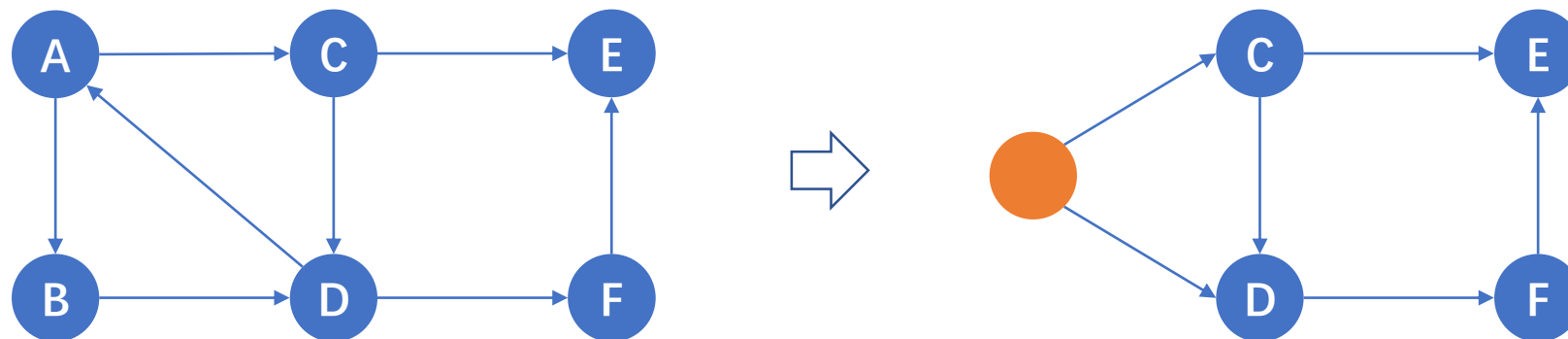
- Tarjan算法基于这样的原理：在任何深度优先搜索中，同一强连通分量内的所有节点均在同一棵深度优先搜索树中
- Low[]数组实际上记录的是某个节点在其所属的强连通分量所对应搜索树的根结点的DFN值。而且在栈中，根节点一定是位于最下方的

Tarjan算法小结

- 强连通分量必定是由环或者若干个环构成。所以当发现搜索的下一个点已经在栈中时，就将这一条路径的Low值统一，也就是这条路径上的点属于同一个强连通分量
- 如果遍历完整个搜索树后某个点的DFN值和Low值相等，则意味着它是该搜索树的根，这时它以及它以上直到栈顶的所有节点共同构成一个强连通分量

求强连通分量的意义

- Tarjan算法的本质，相当于在有向图中找有向环
- 如果我们把有向图中所有强连通分量都视作一个节点，然后重新构图，新的图会是一个有向无环图（DAG）



求强连通分量的意义

- 怎么实现呢？可以把属于同一个强连通分量的点都染色（打上相同的标记）
- 我们可以直接在Tarjan的时候，所有退栈处理的点都做染色处理

求强连通分量的意义

- 更进一步地，还可以利用Tarjan算法求出的Low[]和DFN[]来求割边/割点

抢掠计划

- 给定一张 n 个点 m 条边的有向图，每个点有点权，有一些点被标记为终止节点
- 从1号点出发，沿着图中的边走，最后走到一个终止节点，途中经过的点的点权和为你的收益。如果经过了一个点多次，收益也只能计算一次
- 求最大的收益。保证能从1号点走到某个终止节点

$n, m \leq 500,000$

分析

- 如果我们走到了某个强连通分量，那么我们就可以走遍这个强连通分量中的每个节点，然后从任意一条边走出这个强连通分量
- 那么我们把每个强连通分量缩成一个点，并把这些点的点权和作为新的点的点权
- 这样我们得到了一个有向无环图，然后可以DP

分析

- 设 $f[i]$ 表示走到 i 号节点的最大收益，有：

$$f[i] = w[i] + \max\{f[j], (j, i) \in E\}$$

- 直接按照拓扑序DP

Tarjan拓展

- Tarjan算法针对的是有向图
- 可以用于求无向图中的双连通分量？
- 只需要作一点改动：不允许从一个点走回其父亲

拓扑序

什么是拓扑序

- 日常生活中，常有这样的例子：某些事情的完成，依赖于一些先决条件，有相对严格的先后关系
- 比如先修了某些课程才能选修下一个模块，或者先铺好了道路才能完成亮化绿化等等
- 这时候我们可以构图，以先后顺序来确定某条边的起点和终点。换句话说，在这样的图里，先完成的事情所对应的顶点，其遍历顺序一定是在后完成的事情所对应顶点的前面

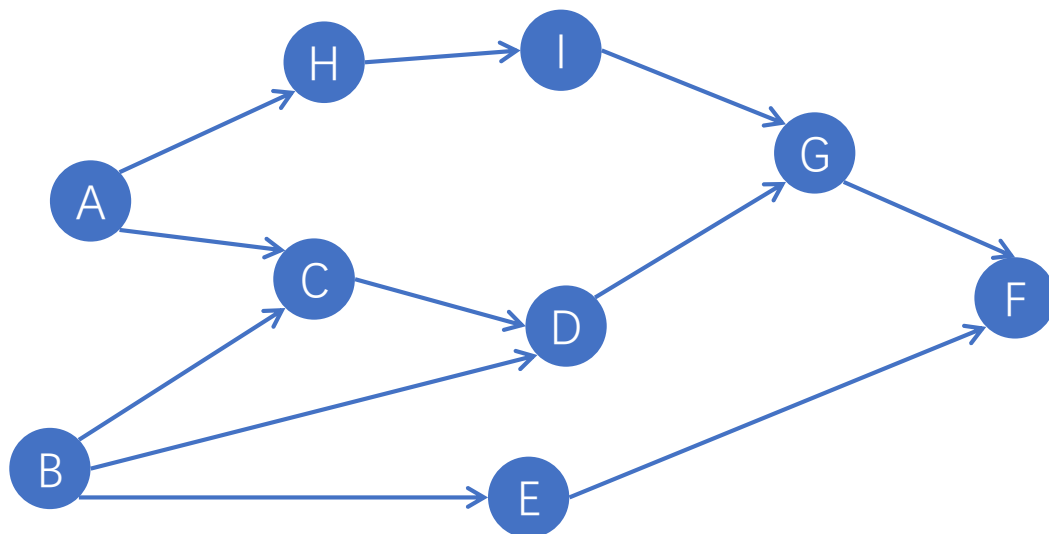
拓扑序

- 一个有向无环图，即可以被称为拓扑图
- 有向无环图具备一个性质，它定义了一个相对严格的遍历次序，即为拓扑序

拓扑序

- 拓扑序的重要意义， 在于为图上的DP提供明确的转移方向
- 拓扑序可能不唯一， 但不影响DP的正确性

拓扑序



(A-B-C-D-E-H-I-G-F)

(B-E-A-H-C-I-D-G-F)

复习一下

- 对于一个有向图来说，一个点有入度和出度两个属性
- 从这个点往别的点连一条边，那么这个点出度加1；别的点往这个点连一条边，那么这个点入度加1
- 对于无向图来说，则不存在出度和入度，统称为度

怎么求拓扑序

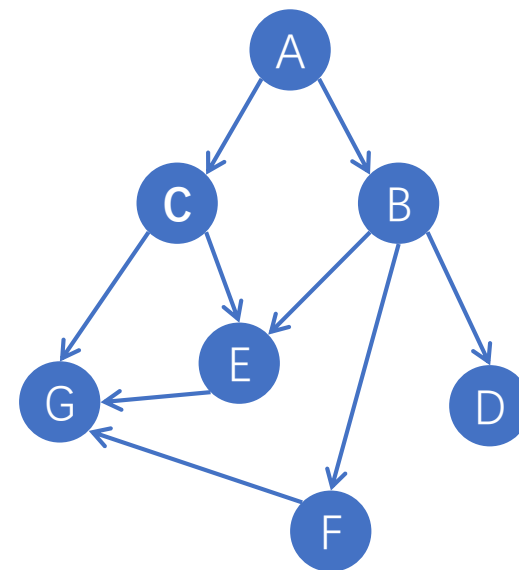
1. 从入度为0的点中任选一个，然后把它放入已排序集合中，同时访问所有以它为起点的边，将对应终点的入度减1
2. 重复1，直到不存在一个入度为0的点，若此时所有点都已进入已排序集合，则拓扑排序完成，得到了一个拓扑序。否则这个图不是一个拓扑图（即它必然存在环）

演示

a b h c d g f e

算法实现(邻接矩阵)

- 执行步骤:
- 1、找到全为零的第j列，输出j;
- 2、将第j行的全部元素置为零;
- 3、找到全为零的第k列，输出k;
- 4、将第k行的全部元素置为零;
- 反复执行3、4，直至所有元素输出完毕。

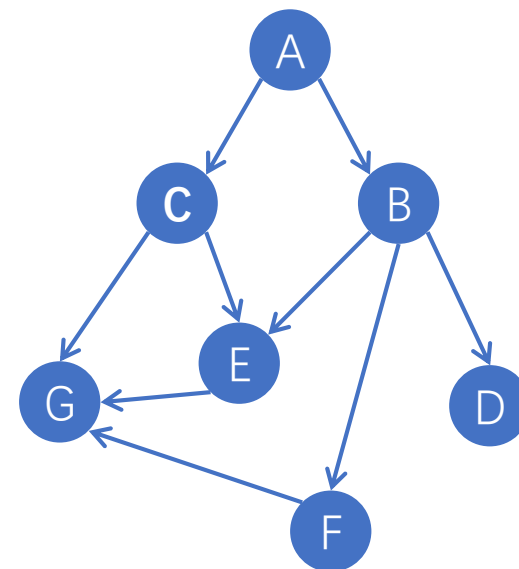


	A	B	C	D	E	F	G
A	0	1	1	0	0	0	0
B	0	0	0	1	1	1	0
C	0	0	0	0	1	0	1
D	0	0	0	0	0	0	0
E	0	0	0	0	0	0	1
F	0	0	0	0	0	0	1
G	0	0	0	0	0	0	0

算法实现(邻接表)

- 执行步骤:
- 1、用一个数组记录每个节点的入度, 将入度为0的节点进栈;
- 2、将栈中入度为0的结点V输出;
- 3、根据邻接表找到节点V的所有的邻接节点, 并将这些邻接节点的入度减1。如果某一节点的入度变为0, 则进栈;
- 4、反复执行 2、3, 直至栈空为止。

• Ps: 这里也可以用优先队列, 注意先后顺序。



栈		入度	
		A	0
		B	1
		C	1
		D	1
		E	2
		F	1
		G	3
	0		

旅行计划

- 编号1~n的城市，有m条道路相连
- 现在要制定一条旅行路线，使得线路上除了第一个城市，每个城市都在路线前一个城市的东面，并且满足这个前提下还希望游览的城市尽量多

$n \leq 10w$, $m \leq 20w$

分析

- 有向无环图中的最长路问题
- 具体做法就是求裸的拓扑序，从入度为0的边开始
- 还可以反向处理

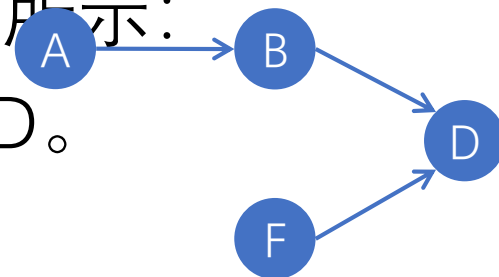
士兵排队

- 【问题描述】

有 n 个士兵，编号依次为A、B、C，……。

队列训练时，指挥官要把一些士兵从高到矮依次排成一行。但现在指挥官不能直接获得每个人的身高信息，只能获得“ p_1 比 p_2 高”这样的比较结果（ $p_1, p_2 \in \{A, \dots, Z\}$ ），记为 $p_1 > p_2$ 。

例如 $A > B$ ， $B > D$ ， $F > D$ 。士兵的身高关系如图所示：



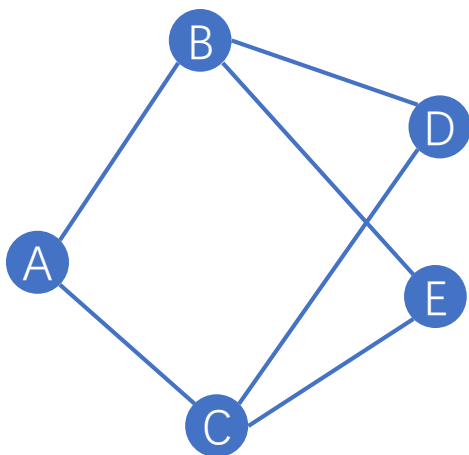
对应的排队方案有三个：AFBD、FABD、ABFD。

现在要输出所有士兵的排队序列。

二分图与二分图匹配

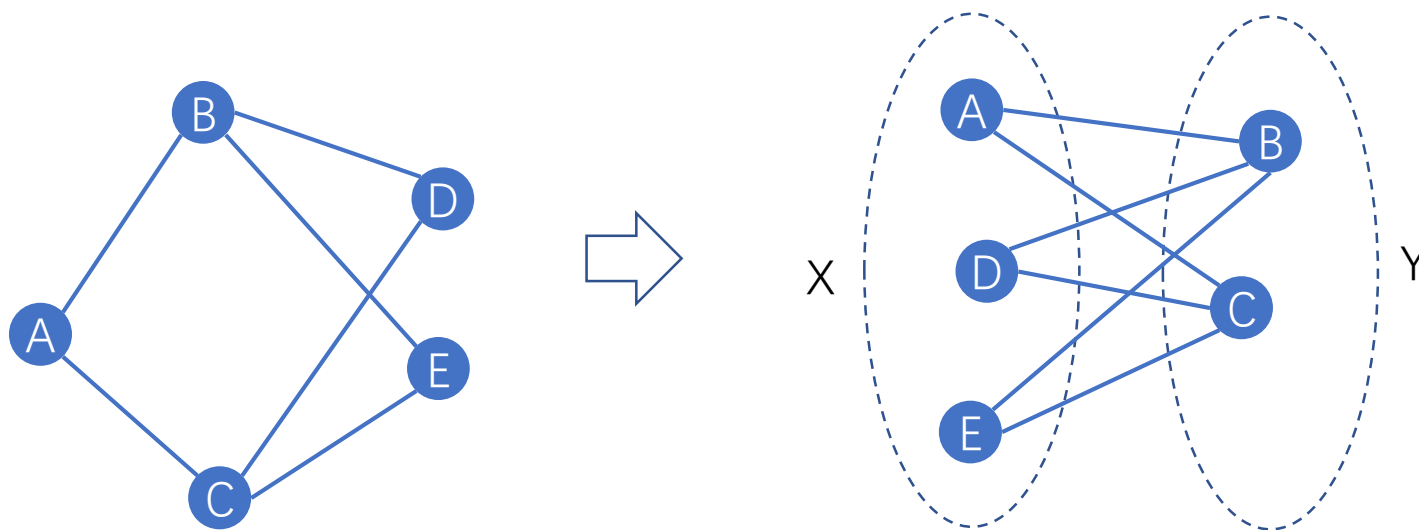
二分图

- 若图 G 的顶点集可划分为两个非空子集 X 和 Y ，即 $V=X \cup Y$ 且 $X \cap Y = \emptyset$ ，并使得对于任意的边 $(u,v) \in E$ ，都有 $u \in X$ 且 $u \notin Y$ ， $v \in Y$ 且 $v \notin X$ （或者反之），则称这个图 G 是二分图



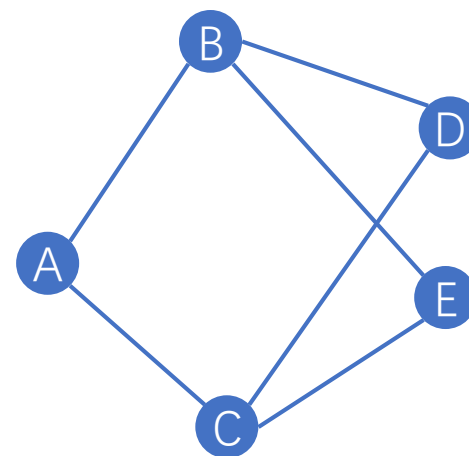
二分图

- 通俗地说，就是对于一个二分图，能够把它的所有顶点分成两个点集，使得两个点集之间有边相连，但是点集内部的点均没有边相连



二分图判定

- 二分图怎么判定呢？
- 利用DFS对图进行黑白染色（打标记）
- 如果某个点染成黑色，那么与其相连的点必定染成白色，反之同理
- 如果在染色过程中发现邻接点有相同颜色(标记)的，则不是二分图。这个做法称之为二分图染色



二分图判定

1. 图中无环

- 如果图中没有环，那么它的各顶点一定可以二分，因为可以按遍历顺序把它们轮流放入两个点集
- 具体代码实现的时候直接BFS遍历就好

二分图判定

2. 图中有环

- 图中如果有环，而环的边数为偶数的话，那所有顶点还是可以二分的
- 而如果环的边数为奇数，就势必会剩一条边，其对应的顶点无法二分

二分图判定

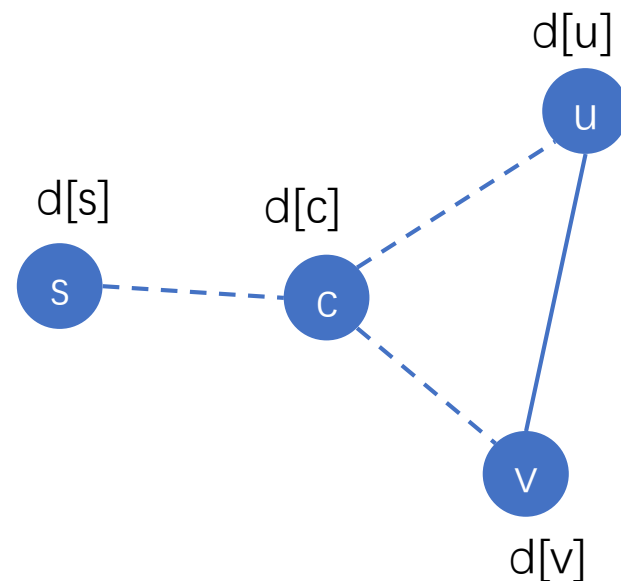
- 一张图是二分图的充要条件是图中不存在奇环
- 如果图中无环，可以视为环的边数为0，而0不属于奇数

判断奇环

- 有一个方法可以通过快速判断奇环的存在而得出结论，从而不必对整张图剩余的部分继续染色

快速判断奇环

- 利用DFS预处理初始节点到每个节点的距离 $d[]$ ，设初始节点为 s ，当前访问节点为 v 。这里边权可以看作1
- u 是 v 的相邻节点， c 是从 s 到 u 和从 s 到 v 的交叉点
- 如果当前节点 v 的相邻节点 u 已被标记，那么势必存在一个包含 u 、 v 、 c 在内的环



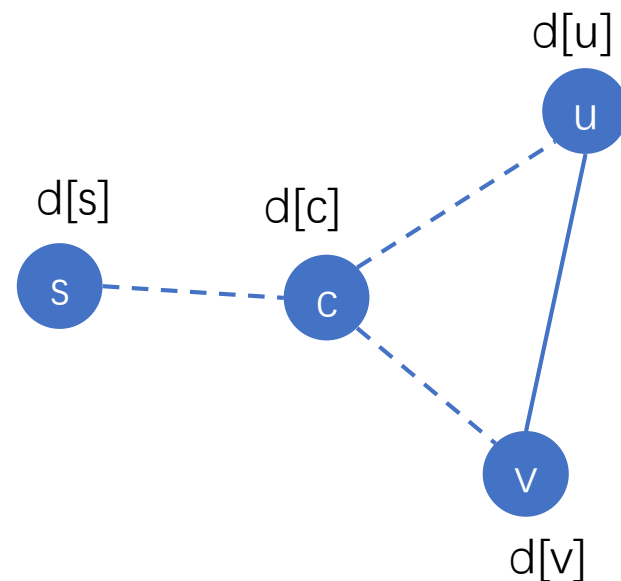
快速判断奇环

- $d[v]-d[u]$ 是从 c 到 v 和从 u 到 c 距离的差值，环 $c-\cdots-v-u-\cdots-c$ 的边长可以表示为：

$$d[v]+d[u]-2\times d[c]+1$$

- 化一化可以得到：

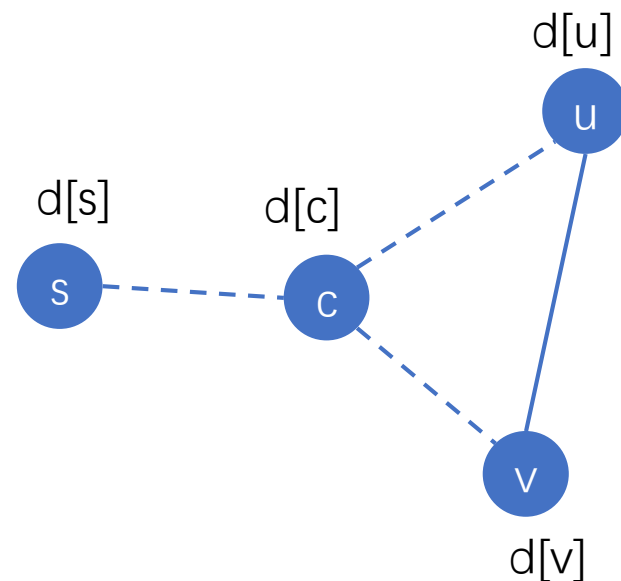
$$2\times(d[u]-d[c])+(d[v]-d[u]+1)$$



快速判断奇环

$$2 \times (d[u] - d[c]) + (d[v] - d[u] + 1)$$

- 这个表达式的奇偶性完全由后面的部分决定
- 所以如果 $d[v] - d[u] + 1$ 是奇数，那么这个环是奇环，这个图就不是二分图



关押罪犯

- 有两座监狱一共关押着 n 名罪犯，编号分别为 $1 \sim n$ 。很多罪犯之间积怨已久，随时可能爆发冲突。我们用怨气值来表示某两名罪犯之间的仇恨程度。如果两名怨气值为 c 的罪犯被关押在同一监狱，他们俩之间会发生摩擦，并造成影响力为 c 的冲突事件
- 每年年末警察局会将本年内监狱中的所有冲突事件按影响力从大到小排成一个列表。在详细考察了 n 名罪犯间的矛盾关系后，警察局准备将罪犯们在两座监狱内重新分配，以求产生的冲突事件影响力都较小
- 应如何分配罪犯，才能使影响力最大的冲突事件的影响力最小？这个最小值是多少？

分析

- 要求最大值最小，典型的二分；要求将罪犯关押在两个监狱里，容易想到二分图模型
- 可以按照影响力大小排序后，二分答案，将大于答案的都连边（小于的不影响），然后黑白染色判断是否能构成二分图

邮差

- 有一个辛苦的邮递员，他每天的工作便是给全小镇的人送信。因为小镇由很多街道组成，所以邮递员要将所有的街道都走一遍。
- 输入一个带权图 G ，问最短行程和方案。

例题： 邮差

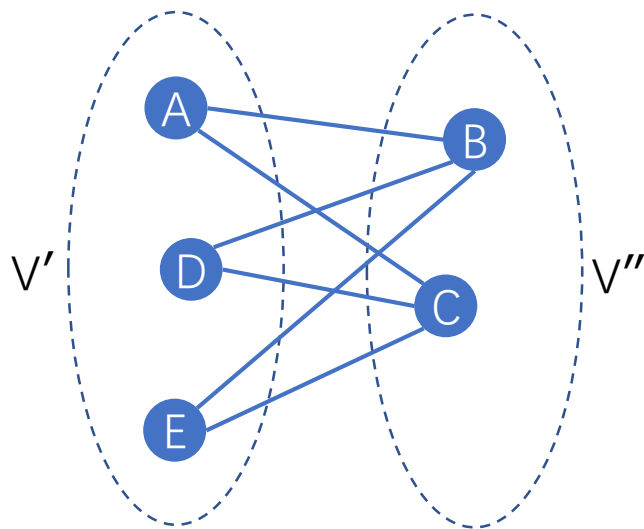
- 一笔画？ 对， 就是欧拉图。
- 如果不是欧拉图， 那么必定存在偶数个奇点。
- 将所有偶点删去， 将图转化为只包含奇点的图。
- 再求最小生成树。

匹配

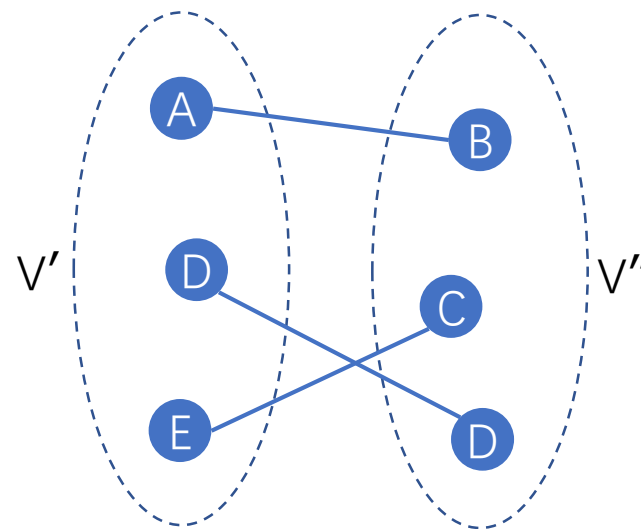
- 匹配是指这样一些边的集合：任意两条边没有公共端点
- 也就是只存在一对一的关系，不存在一对多或者多对一的关系

二分图匹配

- 如下图一就不是二分图匹配，图二则是二分图匹配



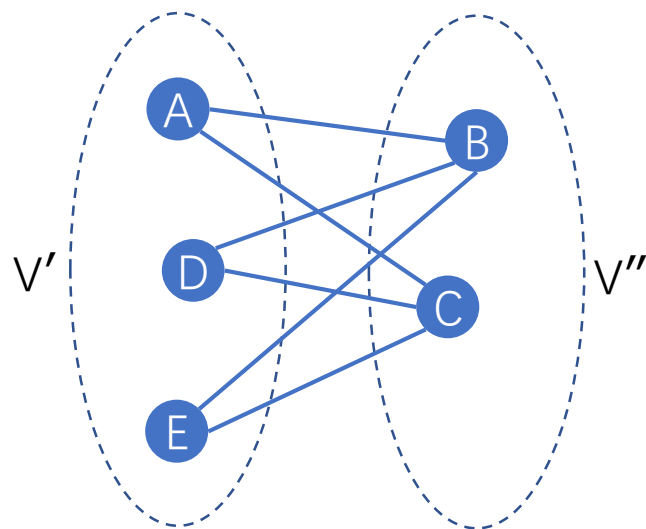
图一



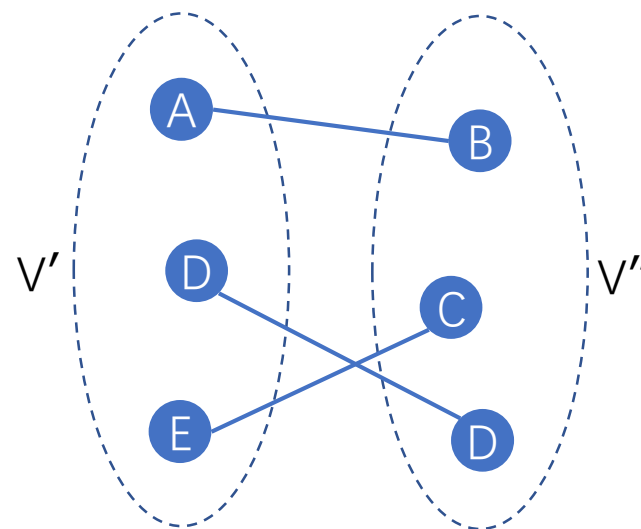
图二

二分图匹配

- 匹配边数最多，称为二分图最大匹配
- 图二因为所有的点都已匹配上，还可以进一步称之为完全匹配



图一



图二

Hall定理

- 二分图完全匹配存在性判定定理
- 设有二分图 $G = \langle V_1 \cup V_2, E \rangle$ ，假设 V_1 点集个数 $\leq V_2$ 点集个数，则从 V_1 中任取 k 个不同的点，它们连向 V_2 中的点的个数不少于 k

二分图最大匹配

- 二分图最大匹配，如果不带权值，一般使用Hungary算法
- 如果连边带权值，则是要求出一种匹配方案，使得匹配集合中的边权之和最大或最小，这时需要使用Kuhn-Munkres算法
- Hungary算法也可以是看作是求边权固定为1的最大带权匹配

Hungary算法

- 在介绍Hungary算法之前，先介绍几个基本概念：

1. 未覆盖点

– 匹配未涉及到的点，也就是被“孤立”了

Hungary算法

- 在介绍Hungary算法之前，先介绍几个基本概念：

2. 交错路

- 对于一个匹配边集，如果存在一条长度为奇数的路径，满足第奇数条边不属于这个匹配，而第偶数条边属于这个匹配，那么这条路径称为交错路
- 或者这么理解：图中任意两条相邻的边，一条属于匹配边集，另一条不属于

Hungary算法

- 在介绍Hungary算法之前，先介绍几个基本概念：

3. 可增广路

- 起点终点都是未覆盖点的交错路。如果找不到可增广路，即是达到了最大匹配

Hungary算法

1. 首先从任意一个未被配对的点 u 开始，从点 u 的连边中任选一条开始配对。如果此时 v 没有配对，则配对成功，找到了一条增广路，继续尝试下一个点 u
2. 这一过程中如果发现 v 已经被匹配了，就要从 u 的连边中再重新选一条找到增广路；如果不能，就需要拆除原有的匹配关系

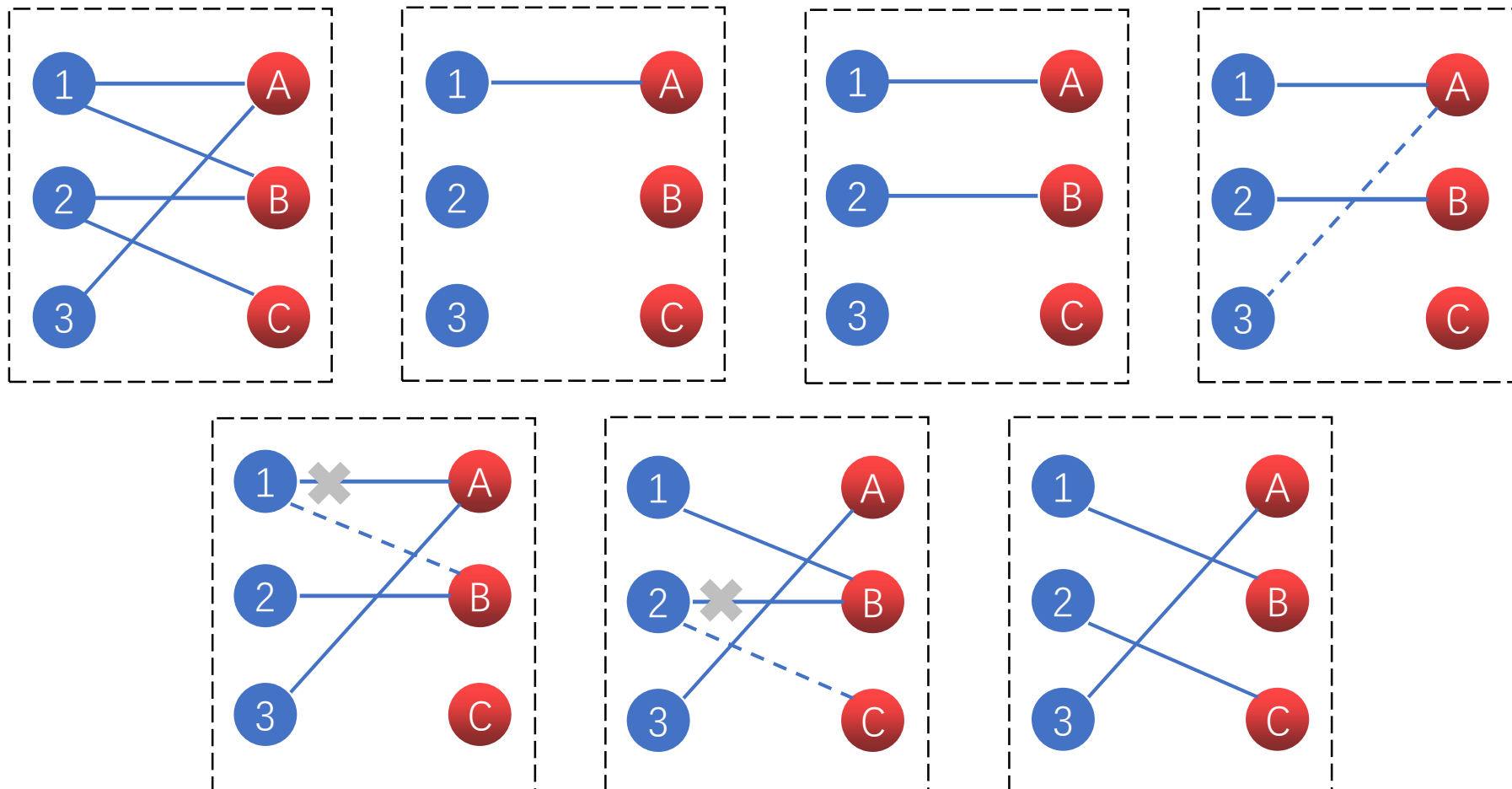
Hungary算法

3. 要更新原有的配对关系，可以用`match[]`数组来记录配对关系。
比如点`u`和`v`配对，就记作`match[u]==v && match[v]==u`
4. 配对的过程用DFS/BFS都可以

Hungary算法

5. 接下来对剩下的点一一配对，直到所有的点都尝试完毕，找不到新的增广路为止
- 算法复杂度为 $O(VE)$

Hungary算法演示



参考代码

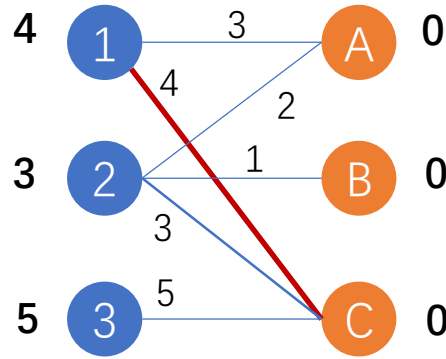
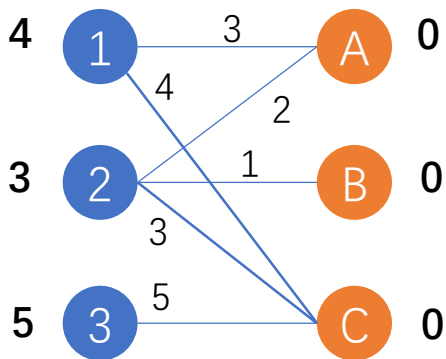
```
int dfs(int u)
{
    int i;
    for (i = 1; i <= n; ++ i)
    {
        if (flag[i] == 0 && e[u][i] == 1)
        {
            flag[i] = 1;    // 标记i点被访问过
            if (match[i] == 0 || dfs(match[i]))
                // 如果点i没有被配对, 或者找到了新的配对
            {
                match[i] = u;    // 更新配对关系
                match[u] = i;
                return 1;
            }
        }
    }
    return 0;
}
```

KM算法

- KM算法的基本思想是：通过给顶点赋最大权，然后再利用Hungary算法进行最大匹配
- 匹配的基本原则是：从起始顶点出发，只与权值相同的边匹配，若是找不到边，则该路径（包含冲突路径）起始顶点权值-1，终止顶点权值+1，再进行匹配
- 重复这一过程直至完成最大匹配

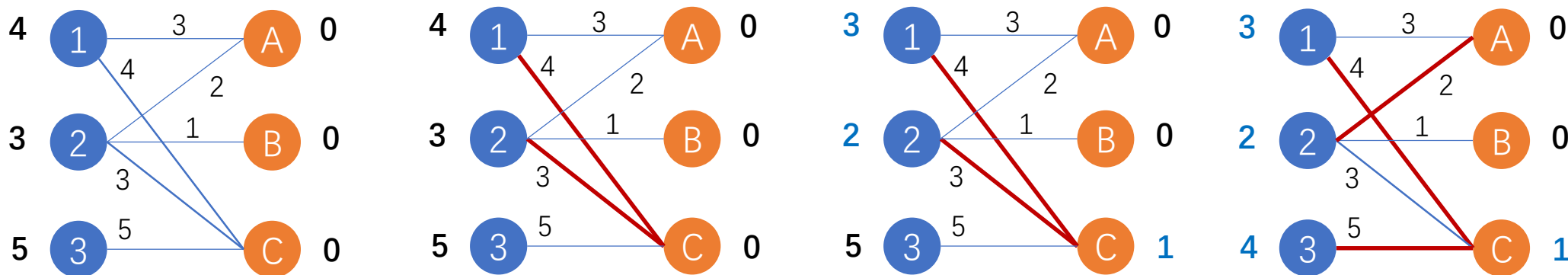
KM算法

- 1号点因为权值为4，所以只能匹配C点，因为只有C点和1号点的权值之和等于它们之间的边权值



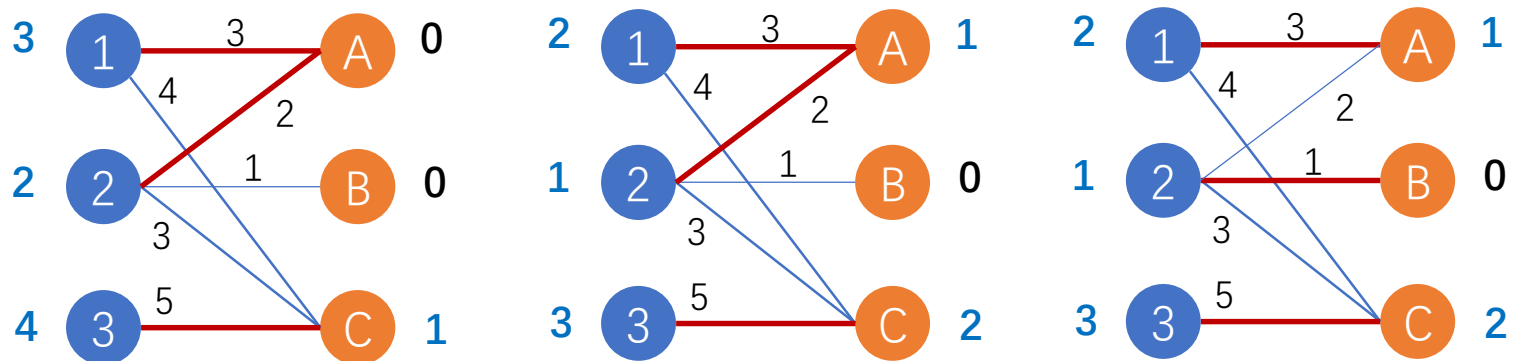
KM算法

- 然后继续为2号点匹配，此时应匹配C点，出现冲突。解决的办法是修改标号如下图
- 修改标号后，2号点匹配A点 ($2+0=2$)
- 继续匹配3号点，此时无法匹配 ($5+1>5$)，3号点修改标号。修改之后，3号点匹配C点 ($4+1=5$)，出现冲突



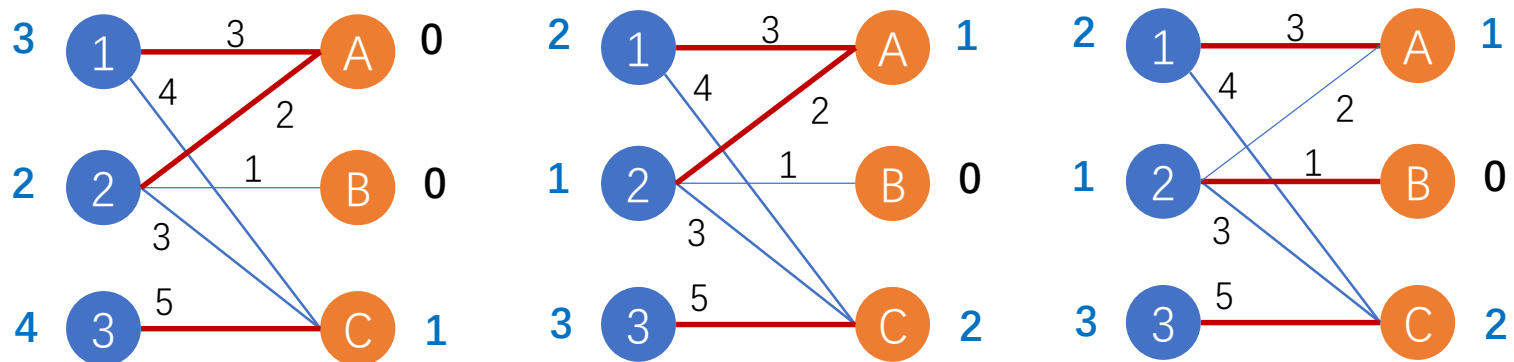
KM算法

- 与3号点冲突的1号点，换为匹配A点，因与2号点冲突而失败
- 修改相关的1号点、2号点、3号点、C点、A点标号
- 此时3号点匹配C点 ($3+2=5$)，与之冲突的1号点匹配A点 ($2+1=3$)，与1号点冲突的2号点匹配B点 ($1+0=1$)，此时匹配完全成功



KM算法

- 整个过程递归实现
- KM算法相当于不断寻找原问题的等价子图，然后在此基础上使用Hungary算法
- KM算法只能求带权二分图最大匹配，若是要求最小，需要把权值取反
- 时间复杂度 $O(n^3)$ （ n 是点数）



带权二分图最大匹配

- 其实这个问题，更主流的做法是最小费用最大流

二分图匹配的拓展

- 最小点覆盖：用最少的点覆盖所有的边
- 定理：最小点覆盖=最大匹配数

二分图匹配的拓展

- 最大独立集：最多的点使得点两两间无边
- 定理：设图 $G=(V,E)$ ，最大独立集 U ，最大匹配边集 M ，有：
$$|U|=|V|-|M|$$

二分图匹配的拓展

- 最小路径覆盖：用最少的不相交路径覆盖所有的点
- 定理：最小路径覆盖 = 顶点 V - 最大匹配数

差分约束

差分约束

- 如果一个系统由n个变量和m个约束条件组成，其中每个约束条件形如：


$$x_j - x_i \leq b_k \quad (i, j \in [1, n], \quad k \in [1, m])$$

则称其为差分约束系统

- 通俗地说，差分约束是求解关于一组变量的特殊不等式组的方法


差分约束

- 比如有如下一组含有5个变量，7个约束条件的不等式组


$$\begin{cases} x_1 - x_5 \leq -1 \\ x_2 - x_5 \leq 1 \\ x_3 - x_1 \leq 5 \\ x_4 - x_1 \leq 4 \\ x_4 - x_3 \leq -1 \\ x_5 - x_3 \leq -3 \\ x_5 - x_4 \leq -3 \end{cases}$$

差分约束

- 这些不等式有个特点：全都是两个未知数的差小于等于某个常数（大于等于也可以，左右乘以-1即可）
- 这样的不等式组就称作差分约束系统
- 所以差分约束系统实际上就是求一组不等式的一个可行解



$$\begin{aligned}x_1 - x_5 &\leq -1 \\x_2 - x_5 &\leq 1 \\x_3 - x_1 &\leq 5 \\x_4 - x_1 &\leq 4 \\x_4 - x_3 &\leq -1 \\x_5 - x_3 &\leq -3 \\x_5 - x_4 &\leq -3\end{aligned}$$

这些解有什么特点

- 这个不等式组要么无解，要么就有无数组解
- 因为如果有一组解 $\{X_1, X_2, \dots, X_n\}$ 的话，那么对于任何一个常数 k ， $\{X_1 + k, X_2 + k, \dots, X_n + k\}$ 肯定也是一组解，因为任何两个数同时加一个数之后，它们的差是不变的，那么这个差分约束系统中的所有不等式都不会被破坏

差分约束

- 那么差分约束系统和图论有什么关联?


$$\begin{aligned}x_1 - x_5 &\leq -1 \\x_2 - x_5 &\leq 1 \\x_3 - x_1 &\leq 5 \\x_4 - x_1 &\leq 4 \\x_4 - x_3 &\leq -1 \\x_5 - x_3 &\leq -3 \\x_5 - x_4 &\leq -3\end{aligned}$$

差分约束和最短路


- 松弛操作的代码:

$$d[v] \leq d[u] + w[u][v]$$

移项得:

$$d[v] - d[u] \leq w[u][v]$$

- $d[u]$ 和 $d[v]$ 是从源点分别到顶点 u 和顶点 v 的最短路径估计值,
 $w[u][v]$ 是边 (u,v) 的权值



$$\begin{aligned} x_1 - x_5 &\leq -1 \\ x_2 - x_5 &\leq 1 \\ x_3 - x_1 &\leq 5 \\ x_4 - x_1 &\leq 4 \\ x_4 - x_3 &\leq -1 \\ x_5 - x_3 &\leq -3 \\ x_5 - x_4 &\leq -3 \end{aligned}$$

差分约束和最短路

- 显然右边的不等式就形如：

$$d[v] - d[u] \leq w[u][v]$$

- 于是我们可以把一个差分约束系统转化成一张图，每个未知数 x_i 对应图中的一个顶点 v_i ，把所有的不等式都看成图中的一条边


$$\begin{aligned} x_1 - x_5 &\leq -1 \\ x_2 - x_5 &\leq 1 \\ x_3 - x_1 &\leq 5 \\ x_4 - x_1 &\leq 4 \\ x_4 - x_3 &\leq -1 \\ x_5 - x_3 &\leq -3 \\ x_5 - x_4 &\leq -3 \end{aligned}$$

差分约束和最短路

- 对于不等式

$$X_i - X_j \leq c$$

移项:

$$X_i \leq X_j + c$$

就可以看成边 (V_j, V_i) , 权值为 c

- 最后, 我们在这张图上求一次单源最短路, 这些三角形不等式就会全部都满足了

$$x_1 - x_5 \leq -1$$

$$x_2 - x_5 \leq 1$$

$$x_3 - x_1 \leq 5$$

$$x_4 - x_1 \leq 4$$

$$x_4 - x_3 \leq -1$$

$$x_5 - x_3 \leq -3$$

$$x_5 - x_4 \leq -3$$

差分约束

- 需要注意的是，差分约束必须保证所有的不等式具有同样的不等号才能正确运行！
- 这样我们就将问题图形化，可以将求解不等式组的问题转化为在一个有向图中求最短路的问题
- 用Dijkstra或者SPFA都可以，不过要注意对无解的情况做特判

源点

- 求单源最短路，必然需要一个源点，然后再求这个源点到其他点的最短路。但是对于差分约束系统，源点在哪里呢？
- 我们不妨自己造一个

源点

- 以上面的不等式组为例，我们就再新加一个未知数 x_0
- 然后对原来的每个未知数都对 x_0 加一个不等式
(这个不等式当然也要和其它不等式形式相同，即两个未知数的差小于等于某个常数)

源点

- 我们索性就全都写成 $X_n - X_0 \leq 0$ ，于是这个差分约束系统中就多出了下列不等式：

$$X_1 - X_0 \leq 0$$

$$X_2 - X_0 \leq 0$$

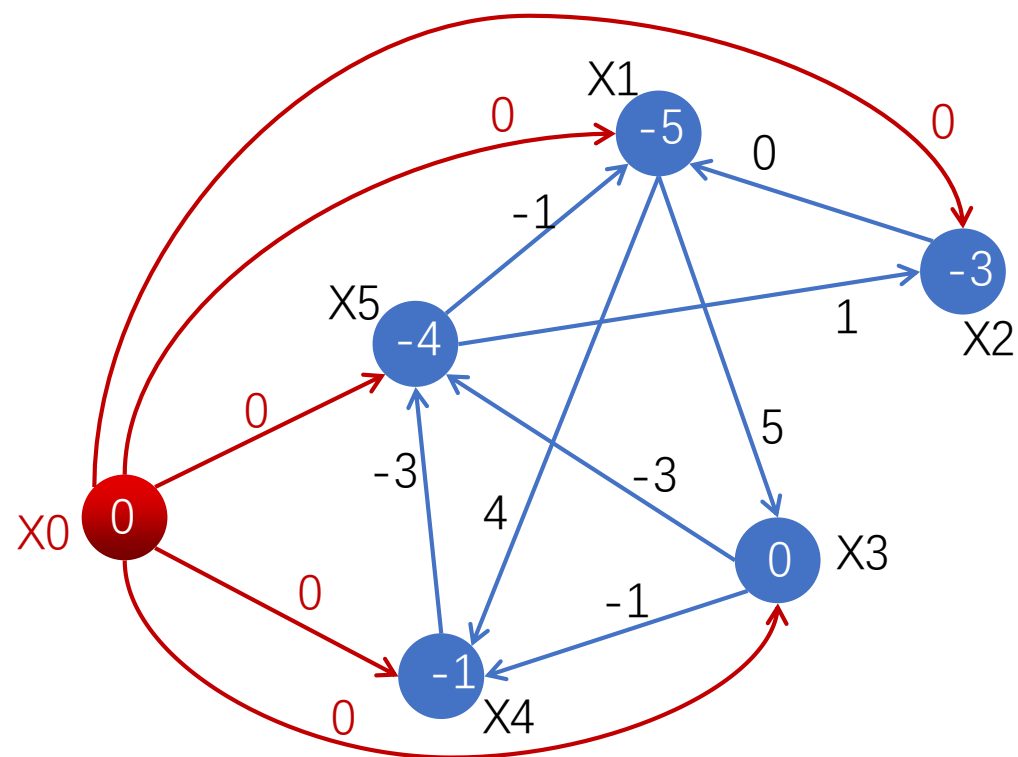
$$X_3 - X_0 \leq 0$$

$$X_4 - X_0 \leq 0$$

$$X_5 - X_0 \leq 0$$

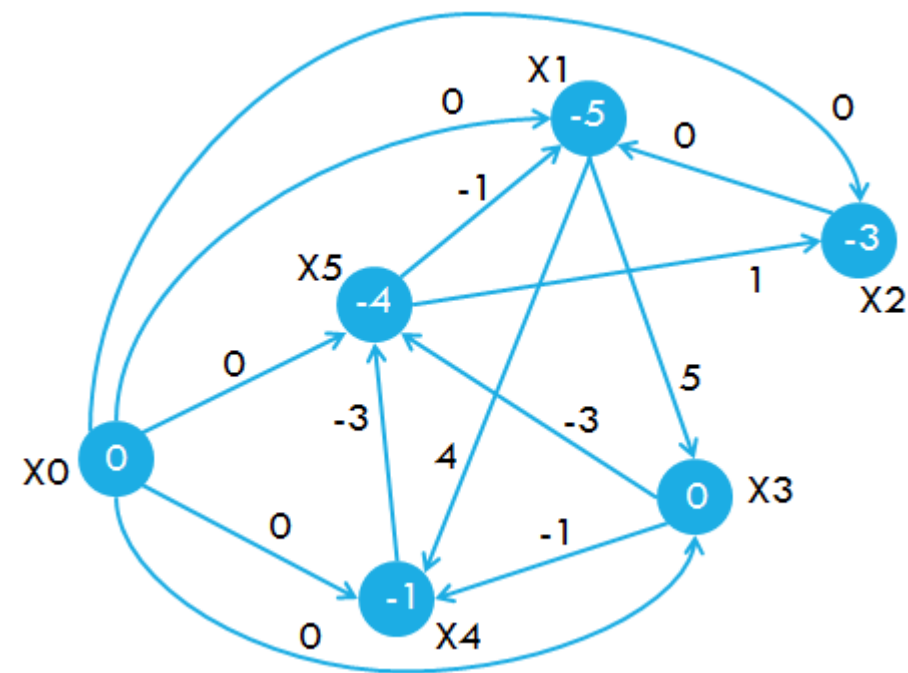
建图

- 最后建的图如下



建图

- 图中的每一条边都代表差分约束系统中的一个不等式
- 现在以 x_0 为源点，求单源最短路径
- 最终得到的 x_0 到 x_n 的最短路径长度就是 x_n 的一组解



建图

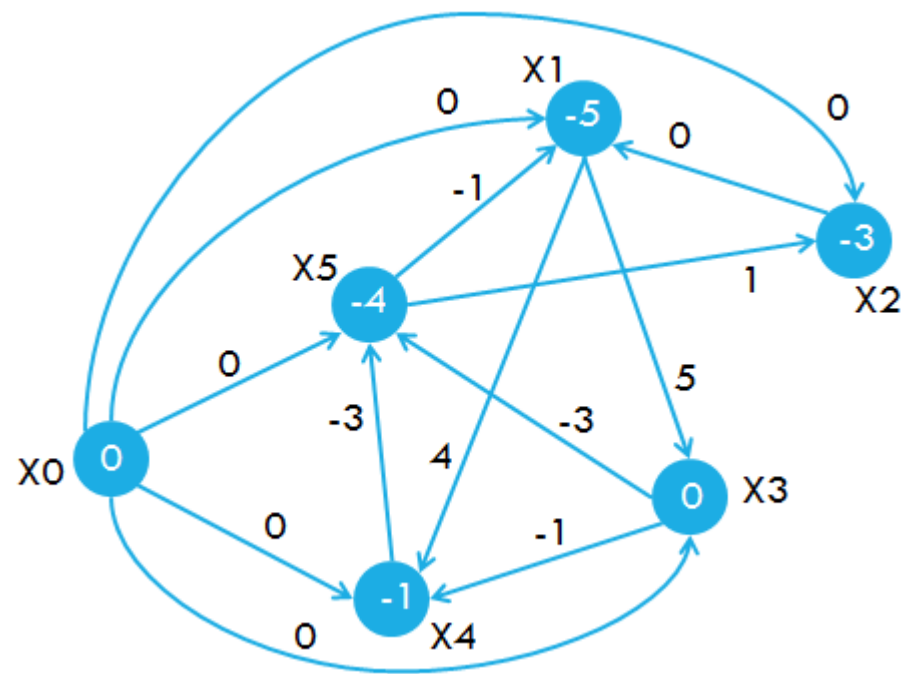
• 从图中可以看到，这组解是：

$\{-5, -3, 0, -1, -4\}$

当然把每个数都加上10

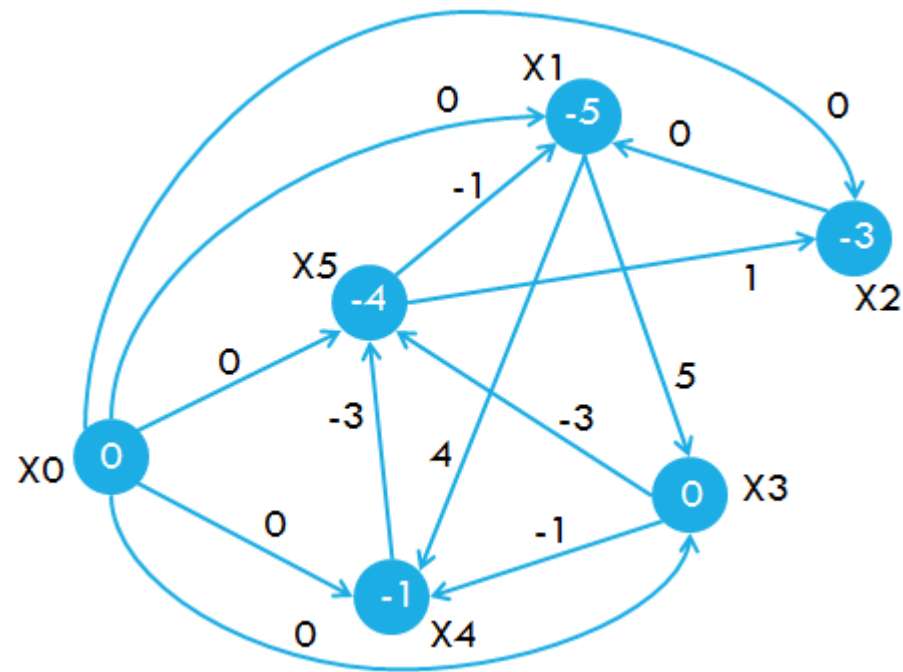
也是一组解

$\{5, 7, 10, 9, 6\}$



建图

- 也有可能出现无解的情况，也就是从源点到某一个顶点不存在最短路径（图中存在负环）



补充说明

- 对于这张图来说，它代表的一组解其实是 $\{0, -5, -3, 0, -1, -4\}$ ，也就是说 X_0 的值也在这组解当中
- 但是 X_0 的值是无可争议的，既然是以它为源点求的最短路径，那么源点到它的最短路径长度当然是0了
- 因此，实际上我们解的这个差分约束系统存在一个隐含条件：

$$X_0 = 0$$

补充说明

- 也就是说在两组不等式组组成的差分约束系统的前提下，再把其中的一个未知数的值定死
- 这样的情况在实际问题中是很常见的。比如一个问题表面上给出了一些不等式，但还隐藏着一些不等式，比如所有未知数都大于等于0或者都不能超过某个上限之类的
- 比如上面的第二组不等式组就规定了所有未知数都 ≤ 0

差分约束的应用

1. 判断是否存在可行解
2. 判断满足约束条件的最小/大解(即求建图后所有点的下/上界, 最长/短路径)。建立的方程为

$$X_{ai} - X_{bi} \geq B_i \text{ 或者 } X_{ai} - X_{bi} \leq B_i$$

建立一条从点 b_i 到 a_i 的边, 边的权值为 B_i

Intervals

- 有一个序列，题目用 n 个整数组合 $[a_i, b_i, c_i]$ 来描述它， $[a_i, b_i, c_i]$ 表示在该序列中处于 $[a_i, b_i]$ 这个区间的整数至少有 c_i 个。
- 如果存在这样的序列，请求出满足题目要求的最短的序列长度是多少，如果不存在则输出 -1

分析

- 这个题目转化为差分约束系统如下：
- 如果 $i \in s$ ，则 $t[i]=1$ ，否则 $t[i]=0$ 。令 $s[i] = \sum t[j] (j = 0 \sim i)$ ，这样题目的条件就可以用下面的式子表示：

$$s[b[i]] - s[a[i]-1] \geq c[i]$$

$$s[i] - s[i+1] \geq -1$$

$$s[i+1] - s[i] \geq 0$$

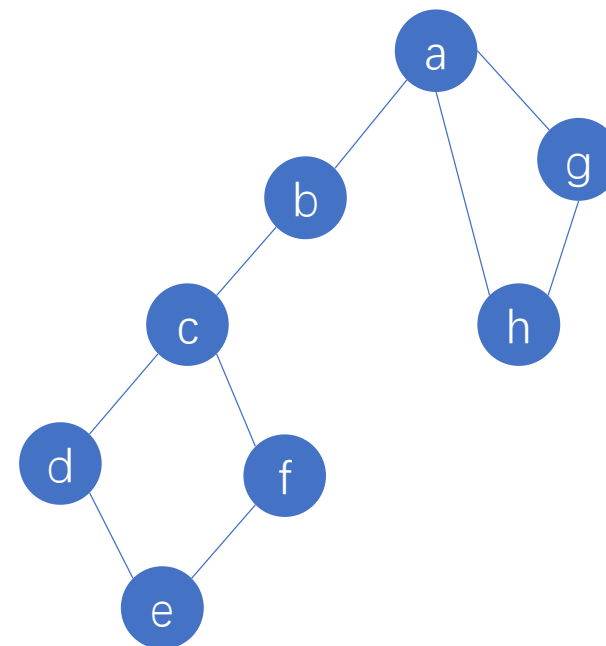
分析

- 注意后面两个式子是 s 自然具有的性质
- 我们要求的就是 $s[n] - s[-1]$ 的最小值（因为题目都是非负的）

图的连通性问题

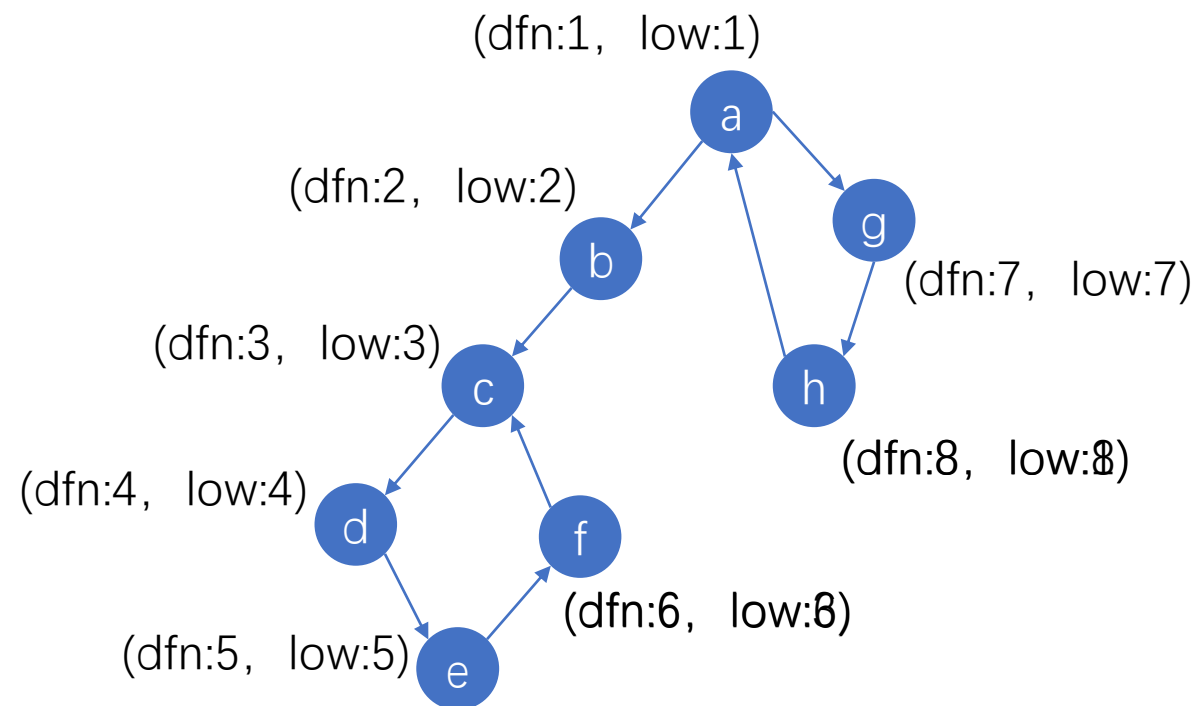
割点和割边/桥

- 无向连通图中，如果删掉某点，图变成不连通，则称该点为割点
c、b、a
- 无向连通图中，如果删掉某边，图变成不连通，则称该边为割边
(a,b)、(b,c)



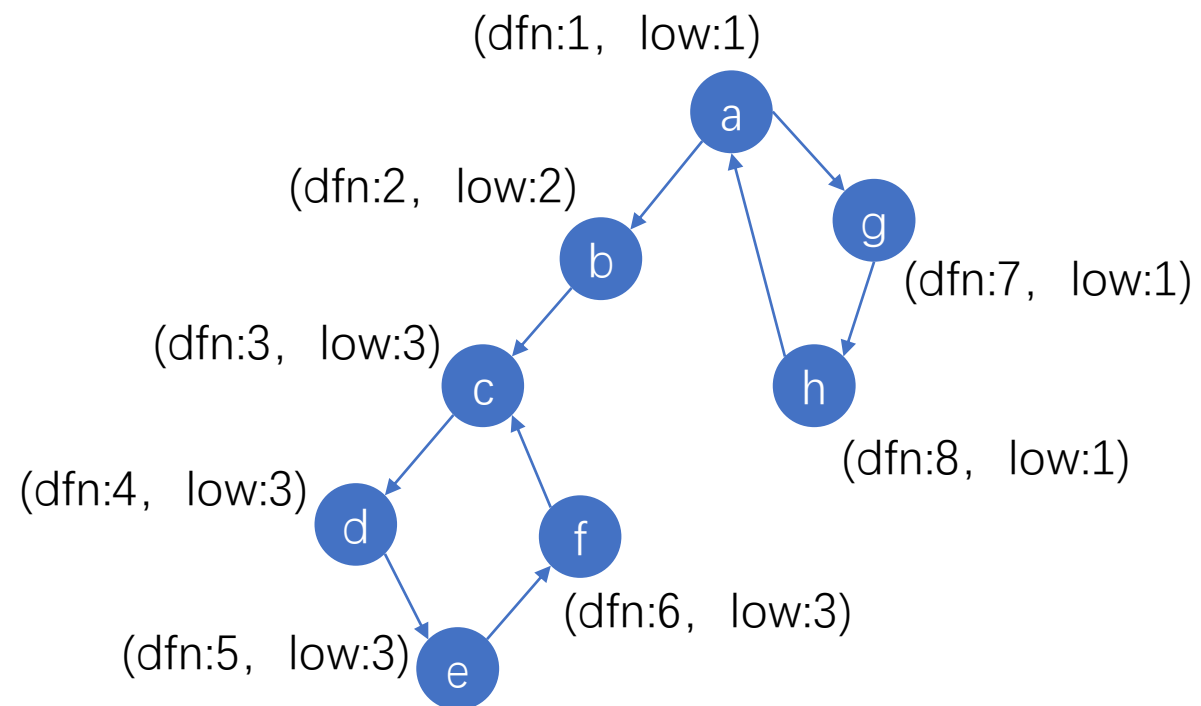
求割点/割边

- 求割点和桥还是Tarjan算法
- 思路和有向图求SCC相似
- 先复习一下Tarjan求SCC
 - c、d、e、f
 - b
 - a、g、h



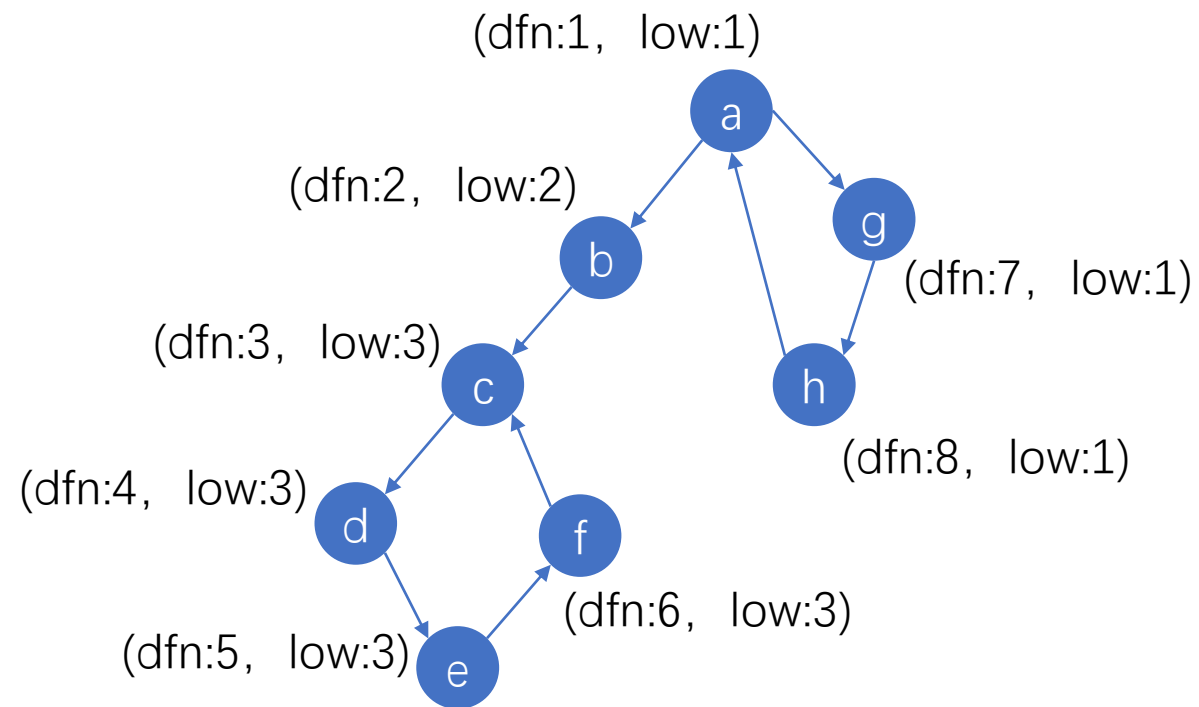
求割点/割边

- $dfn[u]$ 和前面的定义相同, $low[u]$ 定义为u或者u的子树中能够通过非父子边追溯到的最早节点的DFS开始时间(dfn 值)



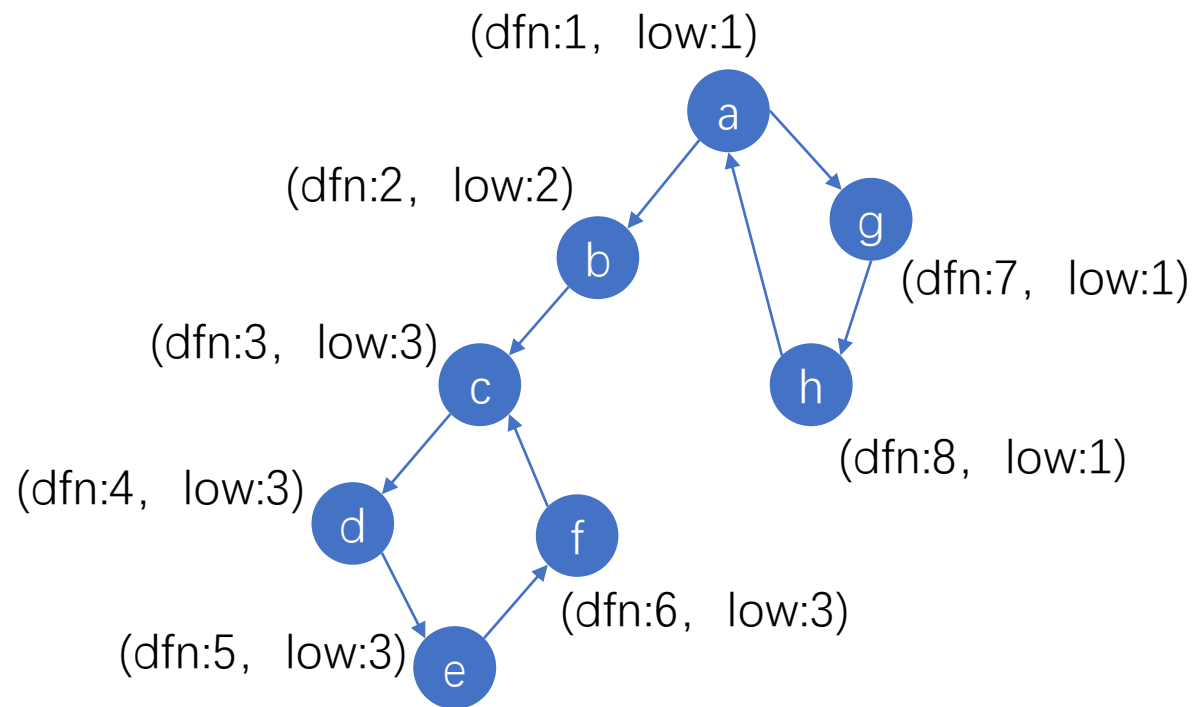
求割点/割边

- 一个节点 u 是割点，当且仅当：
 1. u 为树根，且有超过一棵子树；
 2. u 不为树根，且存在 (u,v) 的树枝边（DFS搜索树中的父子边），使得 $\text{dfn}(u) \leq \text{low}(v)$
- 因为这说明 v 的子孙不能够通过其他边到达 u 的祖先，这样去掉 u 之后，图必然分裂为两个子图



求割点/割边

- 一条边(u,v)是桥，当且仅当：
 1. (u,v)是树枝边，且满足 $\text{dfn}(u) < \text{low}(v)$ ，前提是其没有重边
- 理由同前页
- 非树枝边不可能是桥；有重边，也不可能是桥

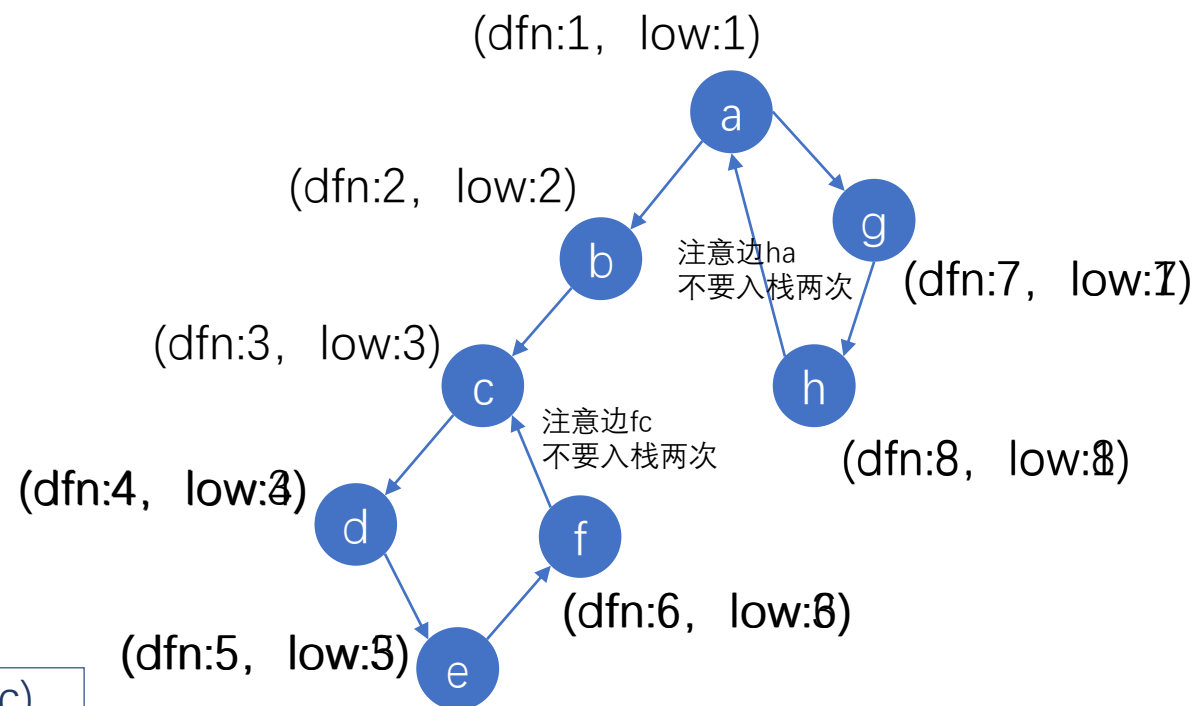


点双连通分量

- 如果一个无向连通图中，不存在割点，则称为点双连通图
- 于是判断一张图是否是点双连通图，只需要Tarjan找割点
- 无向连通图中，不含有割点的连通分量，称为点双连通分量

求点双连通分量

- 在求割点的过程中就能顺便把每个点双连通分量求出
- 建一个栈用于存储当前双连通分量。在搜索图时，每找到一条树枝边或反向边(连到树中祖先的边)，就把这条边加入栈中（要防止访问过的边重复入栈）
- 如果遇到某树枝边(u,v) 满足 $\text{dfn}(u) \leq \text{low}(v)$ ，说明u是一个割点，此时把边从栈顶一个个取出，直到遇到了边(u,v)，取出的这些边与其关联的点，组成一个点双连通分量



(f,c)
(e,f)
(d,e)
(b,d)
(g,b)
(a,g)

- { (f,c)、(e,f)、(d,e)、(c,d) }
- { (b,c) }
- { (a,b) }
- { (h,a)、(g,h)、(g,a) }

边双连通分量

- 如果一个无向连通图中，任一对顶点间至少存在两条无公共边的路径，则称为边双连通图
- 无向连通图中，不含有桥的连通分量，称为边双连通分量
- 桥不属于任何一个边双连通分量，除了桥以外的边，属于且仅属于一个边双连通分量
- 点双连通分量一定是边双连通分量（两点一边的特例图除外），反之则不一定

求边双连通分量

- 既然桥不属于任何边双连通分量，而其他的边均属于且仅属于一个边连通分量
- 那么Tarjan求出桥后，把桥删除，得到的每个连通块，均是边双连通分量

课后练习

- 抢掠计划: [luogu3627](#)
- 旅行计划: [luogu1137](#)
- 关押罪犯: [luogu1525](#)
- Intervals: [poj1201](#)
- Car的旅行路线: [luogu1027](#)
- 神经网络: [luogu1038](#)
- 糖果: [luogu3275](#)

车站分级

2013普及组P4

一条单向的铁路上，依次有编号为 $1, 2, \dots, n$ 的 n 个火车站。每个火车站都有一个级别，最低为 1 级。现有若干趟车次在这条线路上行驶，每一趟都满足如下要求：如果这趟车次停靠了火车站 x ，则始发站、终点站之间所有级别大于等于火车站 x 的都必须停靠。（注意：起始站和终点站自然也算作事先已知需要停靠的站点）

例如，下表是 5 趟车次的运行情况。其中，前 4 趟车次均满足要求，而第 5 趟车次由于停靠了 3 号火车站（2 级）却未停靠途经的 6 号火车站（亦为 2 级）而不满足要求。

车站编号	1		2		3		4		5		6		7		8		9
车站级别	3		1		2		1		3		2		1		1		3
车次																	
1	始	→	→	→	停	→	→	→	停	→	终						
2					始	→	→	→	停	→	终						
3	始	→	→	→	→	→	→	→	停	→	→	→	→	→	→	→	终
4							始	→	停	→	停	→	停	→	停	→	终
5					始	→	→	→	停	→	→	→	→	→	→	→	终

现有 m 趟车次的运行情况（全部满足要求），试推算这 n 个火车站至少分为几个不同的级别。 $1 \leq n, m \leq 1000$

车站分级_{luogu1983}

- 拓扑排序

寻找道路（提高P2难度）

在有向图 G 中，每条边的长度均为 1，现给定起点和终点，请你在图中找一条从起点到终点的路径，该路径满足以下条件：

1. 路径上的所有点的出边所指向的点都直接或间接与终点连通。
2. 在满足条件 1 的情况下使路径最短。

注意：图 G 中可能存在重边和自环，题目保证终点没有出边。

请你输出符合条件的路径的长度。

$$0 < n \leq 10,000, 0 < m \leq 200,000$$

分析

- 这题最重要的就是反过来从终点向起点思考

逛公园

2017提高组D1P3

策策同学特别喜欢逛公园。公园可以看成一张 N 个点 M 条边构成的有向图，且没有自环和重边。其中1号点是公园的入口， N 号点是公园的出口，每条边有一个非负权值，代表策策经过这条边所要花的时间。

策策每天都会去逛公园，他总是从1号点进去，从 N 号点出来。

策策喜欢新鲜的事物，他不希望有两天逛公园的路线完全一样，同时策策还是一个特别热爱学习的好孩子，他不希望每天在逛公园这件事上花费太多的时间。如果1号点到 N 号点的最短路长为 d ，那么策策只会喜欢长度不超过 $d + K$ 的路线。

策策同学想知道总共有多少条满足条件的路线，你能帮帮他吗？

为避免输出过大，答案对 P 取模。

如果有无穷多条合法的路线，请输出 -1 。

N	M	K
100000	200000	50

逛公园 *luogu3953*

- 最短路+DP