图的概念是什么？

顶点和边的集合

图有哪两种常用的表示方法呢？

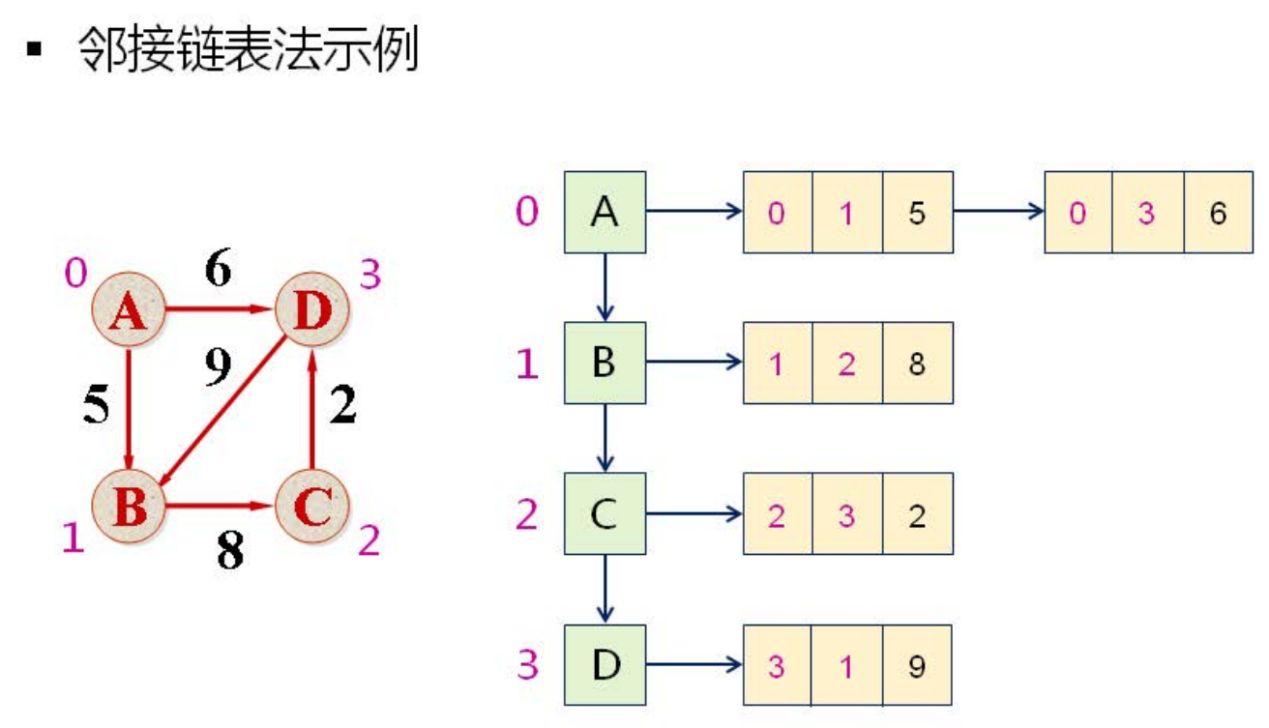
一为邻接矩阵，二为邻接链表。

邻接矩阵怎么表示呢？

数组嘛！一个N维的节点数组和一个N×N的二维的边数组

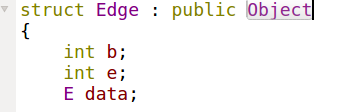
那邻接链表怎么表示呢？

1. 所有顶点存在一个链表中。
2. 顶点是一个自定义的数据结构，里面包含顶点的值（指针）以及一个边集链表，该链表表示始发于该顶点的边
3. 每一条边包含三个信息：起点、终点、权值

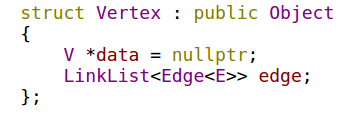


具体代码中的定义：

边：

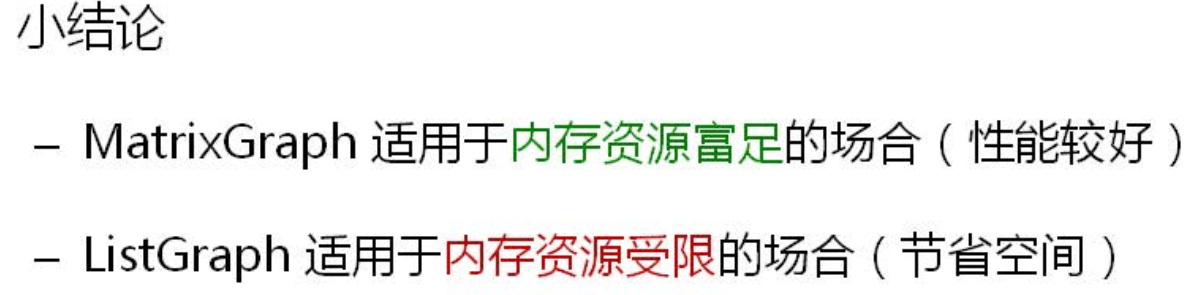


顶点：



图中的数据结构：







下面是两个常见的版块

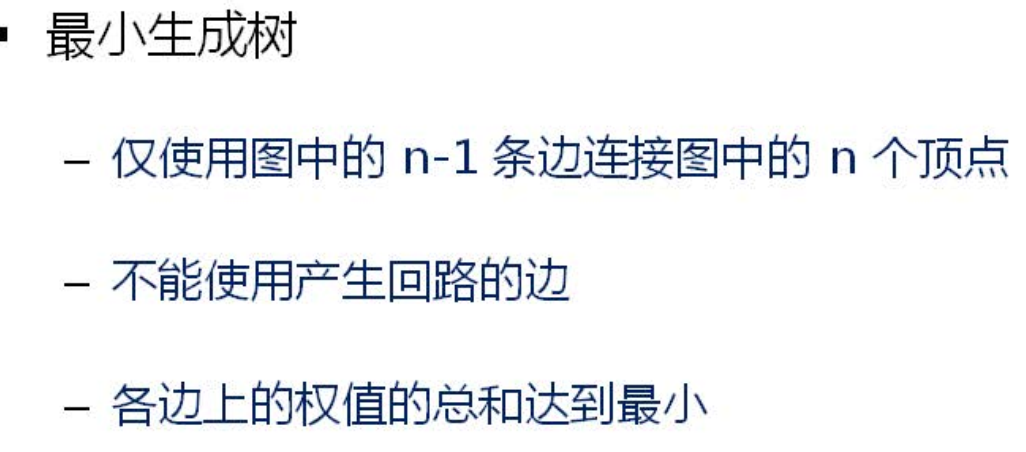
1. 最小生成树
2. 最短路径

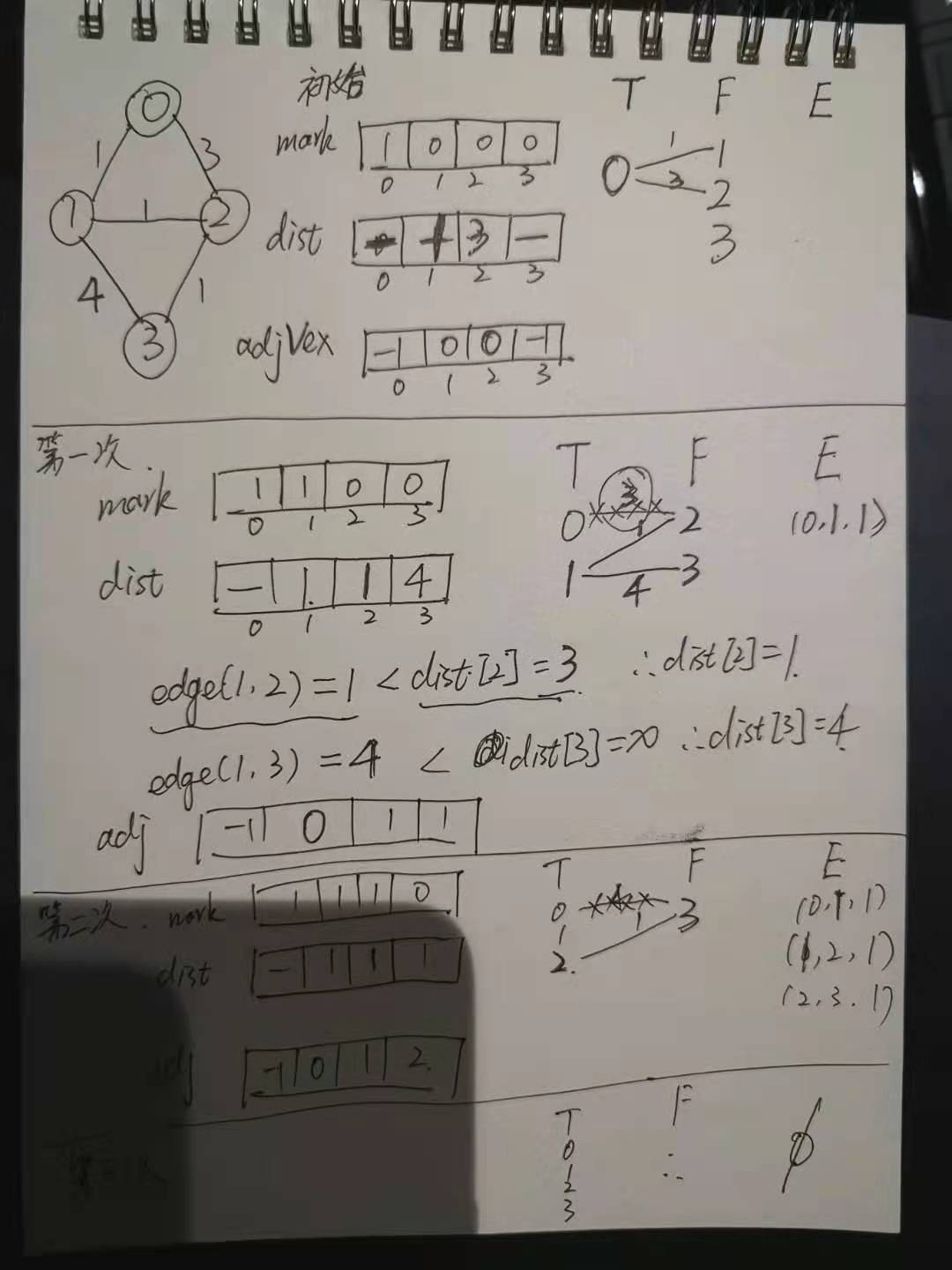
最小生成树掌握prim算法和kruskal算法

最短路径掌握dijkstra算法和floyd算法

最小生成树问题描述：

图中有n个顶点，如何选择n-1条边，使得所有顶点两两可达，且n-1条边的权重之和最小





伪代码：

vector<Edge> prim(graph) {  
 vector<Edge> ret;  
 int n = graph.vCount();  
 vector<bool> mark(n, false);  
 vector<Edge> dist(n, INF);  
 vector<int> adj\_vertex(n, -1);  
   
 mark[0] = true;  
 for(auto adj : graph.adjOf(0)) {  
 adj\_vertex[adj] = 0;  
 dist[adj] = graph.getEdge(0, adj);  
 }  
   
 bool end = false;  
 for(int i=0; i<n && !end; ++i) {  
 int k = -1;  
 Edge m = INF;  
 for(int j=0; j<n; ++j) {  
 if(!mark[j] && dist[j] < m) {  
 m = dist[j];  
 k = j;  
 }  
 }  
 if(k==-1) end = true;  
 if(!end) {  
 ret.push\_back(m);  
 mark[k] = true;  
 for(auto adj : graph.adjOf(k)) {  
 if(!mark[adj] && graph.getEdge(k, adj) < dist[adj]) {  
 dist[adj] = graph.getEdge(k, adj);  
 adj\_vertex[adj] = k;  
 }  
 }  
 }  
 }  
   
 return ret;  
}

白话：

数据结构：

Mark数组 - 标记是否已经使用

Dist数组 - 记录目前可用的边

Adj\_vertex - 可用边对应的前驱顶点

初始化：随便选一个点，该点标记为已选，然后把它的相邻顶点找出来，更新dist数组和adj\_vertex数组

循环，如果没有观测到结束标记，遍历n次。每次循环中：

1. 从dist里面找到最小权重的边，注意排除掉已选点所对应的。
2. 找到最小权重边对应的前驱顶点。
3. 如果没找到，则说明所有点均已选，程序结束
4. 如果找到，则这条边进入ret集合，并把这个点加入已选。
5. 遍历它的邻居，如果该邻居未选且对应的edge值小于dist数据里面的值，则更新dist数组，同时更新adj\_vertex数组

完毕。

接下来是kruskal算法。

对于所有的边，按权值低到高排序。然后开始遍历每一条边，得到起点b和终点e。这里并查集出场啦，如果b和e已经联通，则continue；如果未联通，则该条边加入结果集合。

完事！

## 1135 最低成本联通所有城市

接下来要掌握一下leetcode的1135题

1. kruskal + 并查集解法：自己写的

class UF {

private:

    int count\_;

    vector<int> parent\_;

    vector<int> size\_;

    int find(int x) {

        while(x != parent\_[x]) {

            parent\_[x] = parent\_[parent\_[x]];

            x = parent\_[x];

        }

        return x;

    }

public:

    UF(int n) {

        count\_ = n;

        parent\_.resize(n);

        size\_.resize(n);

        for(int i=0; i<n; ++i) {

            parent\_[i] = i;

            size\_[i] = 1;

        }

    }

    int count() {return count\_;}

    void nodeUnion(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        if(root\_p == root\_q) return;

        if(size\_[p] > size\_[q]) {

            parent\_[root\_q] = root\_p;

            size\_[p] += size\_[q];

        }

        else {

            parent\_[root\_p] = root\_q;

            size\_[q] += size\_[p];

        }

        --count\_;

    }

    bool isConnected(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        return root\_p == root\_q;

    }

};

class Solution {

public:

    int minimumCost(int N, vector<vector<int>>& connections) {

        vector<vector<int>> ret\_edges;

        UF uf(N+1);

        sort(connections.begin(), connections.end(),

            [](const vector<int>& a, const vector<int>& b) {

            return a[2] < b[2];

        });

        int n = connections.size();

        for(int i=0; i<n && ret\_edges.size() < N-1; ++i) {

            int b = connections[i][0];

            int e = connections[i][1];

            if(!uf.isConnected(b, e)) {

                ret\_edges.push\_back(connections[i]);

                uf.nodeUnion(b, e);

            }

        }

        if(ret\_edges.size() != N-1) return -1;

        int ret = 0;

        for(const auto& edge : ret\_edges) {

            ret += edge[2];

        }

        return ret;

    }

};

1. prim写法， 来自101，优先级队列，但是不能输出路径

class Solution {

public:

    int minimumCost(int N, vector<vector<int>>& connections) {

        // 基本数据结构pair<int,int> - 权重， 后继， 基于这个结构建立查询表graph和优先级队列

        vector<vector<pair<int, int>>> graph(N, vector<pair<int,int>>());

        for(const auto& conn : connections) {

            graph[conn[0]-1].push\_back(make\_pair(conn[2], conn[1]-1));

            graph[conn[1]-1].push\_back(make\_pair(conn[2], conn[0]-1));

        }

        priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

        unordered\_set<int> visited;

        int costs = 0;

        // 小技巧， 在进入循环以前，优先级队列先推入{0, 0}，这样就可以把所有操作统一放到循环里面

        pq.emplace(0,0);

        while(!pq.empty()) {

            auto [cost, city2] = pq.top(); pq.pop();

            if(!visited.count(city2)) {

                costs += cost;

                visited.insert(city2);

                for(const auto& c : graph[city2]) pq.push(c);

            }

        }

        return N == visited.size() ? costs : -1;

    }

};

下面是大名鼎鼎的Dijkstra算法

它解决的是起点到所有节点的最短路径

数据结构：vector<Edge> dist; 集合S、V-S

思路：贪心算法

1. 初始：dist元素初始化为无穷大。起点v0加入S，寻找所有v0为起点的边，更新相应dist。
2. 循环：当S != V时

2-1 在V-S集合，即未选择集合里面选出一个dist最小的路径，记录该路径对应的顶点k

2-2 把上述顶点k加到S集合里面

2-3 对于每一个V-S集合里的元素，看看经过k能否使得dist变小

完事

问题是如何记录路径呢？我们可以增加一个数组，记录每个节点的前驱节点：



伪代码：

Vector<int> dijkstra(int i, int j) {  
 vector<int> ret;  
   
 int N; // N为节点总数目  
 vector<int> dist(N, INT\_MAX); // 记录每个节点的距离或代价  
 vector<bool> mark(N, false); // 记录每个节点是否已经“敲定”  
 vector<int> path(N, -1); // 记录每个节点的前驱节点  
   
 for(int k=0; i<N; ++k) {  
 if(isAdjacent(i,k)) {  
 dist[k] = getEdge(i, k);  
 path[k] = i;  
 }  
 }  
   
 mark[i] = true;  
 for(int k=0; k<N; ++k) {  
 // 寻找最小边和对应顶点  
 int m = INT\_MAX;  
 int u = -1;  
 for(int i=0; i<N; ++i) {  
 if(!mark[i] && dist[i] < m) {  
 m = dist[i];  
 u = i;  
 }  
 }  
 if(u == -1) break;  
 // 把u设为已访问  
 mark[u] = true;  
 // 检查u能否使V-S集合中的节点对应路径变短  
 for(int w=0; w<N; ++w) {  
 if(!mark[w] && isAdjacent(u,w) && dist[w] > dist[u] + getEdge(u,w)) {  
 dist[w] = dist[u] + getEdge(u,w);  
 path[w] = u;  
 }  
 }  
 // 利用栈来恢复路径  
 stack<int> s;  
 s.push(j);  
 for(int k=path[j]; k != -1; k=path[k]) {  
 s.push(k);  
 }  
 while(!s.empty()) {  
 ret.push\_back(s.top());  
 s.pop();  
 }  
 return ret;  
 }  
}

最后是一次性求出所有最短路径的floyd算法：

的使用是关键，白话其定义，如下：

节点i到节点j，在使用节点[0,k]作为中继条件下的最短路径，可以得到其递推公式

Ak[i][j] = min(Ak-1[i][j], Ak-1[i][k], Ak-1[k][j])

其base case为：A-1[i][j] = getEdge(i,j)

可以理解为一个动态规划问题

核心伪代码：

for (int i=0; i<vCount(); ++i)  
{  
 for (int j=0; j<vCount(); ++j)  
 {  
 dist[i][j] = isAdjacent(i, j) ? (path[i][j] = j, getEdge(i, j)) : LIMIT;  
 }  
}  
  
for (int k=0; k<vCount(); ++k)  
{  
 for (int i=0; i<vCount(); ++i)  
 {  
 for (int j=0; j<vCount(); ++j)  
 {  
 if (dist[i][j] > (dist[i][k] + dist[k][j]))  
 {  
 dist[i][j] = dist[i][k] + dist[k][j];  
 path[i][j] = path[i][k];  
 }  
 }  
 }  
}

接下来一鼓作气，复习下A\*算法：

OpenList ： 待检查方格列表

CloseList ： 已选择方格列表

步骤如下：

1. 把起始点加入OpenList
2. 重复以下步骤：
3. 从OpenList里面取出代价最小的节点（可以使用二叉堆）
4. 把这个节点移动到CloseList
5. 对这个节点的相邻8个节点进行如下操作

* 如果已经在CloseList，或者越界，或者障碍物，忽略它
* 如果不OpenList中，把它加入OpenList，把当前节点设置为它的父亲，记录该方格的权重
* 如果已在OpenList中，则计算一下经由当前方格到达这里能否更好？如果能的话，更新权重和父节点

1. 停止条件

如果终点在openlist中，则找到了

如果openlist空了，则失败了

## 882 细分图中的可到达节点

/\*

 这道hard题的整体思路就浮出水面了

 它是利用Dijkstra算法遍历以求得可到达节点的最短路径，同时在这个过程中记录更新used数组。

 这种就是在遍历框架中加些小操作的问题

跟普通的Dijkstra算法的区别在于：1、没有设置V表和S表；2、遍历中统计相邻节点关系

 \*/

class Solution {

public:

    using pii = pair<int,int>;

    // maxMoves - 移动距离上限     n - 节点数

    int reachableNodes(vector<vector<int>>& edges, int maxMoves, int n) {

        int ans = 0;

        // 后继表示法表示的graph, pii-first后继节点 pii-second:权重

        vector<vector<pii>> graph(n);

        for(auto& edge : edges) {

            int u = edge[0], v = edge[1], w = edge[2];

            graph[u].push\_back({v, w});

            graph[v].push\_back({u, w});

        }

        // dist数组 : 从起点0到起点i的最短距离

        vector<int> dist(n, maxMoves+1);// 最多走M步，所以M+1有无穷大的意义

        dist[0] = 0;

        // used数组 : used[{i,j}] = u  表示从起点i到终点j经历了u个小节点

        map<pii, int> used;

        // 优先级队列 待选择列表 first-权重 second-该边对应的节点

        priority\_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;

        pq.emplace(0,0);

        while(!pq.empty()) {

            pii top = pq.top(); pq.pop();

            int d = top.first, node = top.second;

            if(dist[node] < d) continue; // 疑点1：为什么这样做？为什么等于不continue

            dist[node] = d;

            ++ans;

            // 和邻居的关系只能在下面这段代码来体现

            for(auto pair : graph[node]) {

                int nei = pair.first;

                int weight = pair.second;

                used[{node, nei}] = min(weight, maxMoves-d);

                int d2 = d + 1 + weight;

                if(d2 < dist[nei]) {

                    dist[nei] = d2;

                    pq.emplace(d2, nei);

                }

            }

        }

        for(auto edge : edges) {

            int u = edge[0], v = edge[1], w = edge[2];

            ans += min(used[{u,v}] + used[{v,u}], w);

        }

        return ans;

    }

};

## 785 判断二分图

广度优先搜索

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        vector<int> color(n);

        queue<int> q;

        for(int i=0; i<n; ++i) {

            if(color[i] == 0) {

                color[i] = 1;

                q.push(i);

                while(!q.empty()) {

                    int node = q.front(); q.pop();

                    for(auto adj : graph[node]) {

                        if(color[adj] == 0) {

                            color[adj] = color[node] == 2 ? 1 : 2;

                            q.push(adj);

                        }

                        else {

                            if(color[adj] == color[node]) {

                                return false;

                            }

                        }

                    }

                }

            }

        }

        return true;

    }

};

/\*

思路：遍历所有节点，对每一个未染色的节点进行广度优先搜索，搜索未染色节点进行染色，并在邻居处判断是否出现错误。

也就是：一遍遍历涂色，一遍检查错误

\*/

下面是深度优先解法：

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        vector<int> color(n,0);

        for(int i=0; i<n; ++i) {

            if(color[i] == 0) {

                color[i] = 1;

                if(!dfs(graph, i, color)) return false;

            }

        }

        return true;

    }

    bool dfs(vector<vector<int>>& graph, int node, vector<int>& color) {

        for(auto adj : graph[node]) {

            if(color[adj] == 0) {

                color[adj] = color[node] == 2 ? 1 : 2;

                if(!dfs(graph, adj, color)) return false;

            }

            else {

                if(color[adj] == color[node]) {

                    return false;

                }

            }

        }

        return true;

    }

};

/\*

思路：遍历所有节点，对每一个未染色的节点进行深度优先搜索，搜索未染色节点进行染色，并在邻居处判断是否出现错误。

也就是：一遍遍历涂色，一遍检查错误

\*/

上面用的是先赋值再递归的方法，把判断放在当前节点和邻居节点的关系上；其实还有另一种方法，就是直接递归，把判断放在当前节点的值是否会出现冲突上——就是说，我给你设置一个值作为参数，看你会不会自相矛盾。链接：<https://leetcode-cn.com/problems/is-graph-bipartite/solution/bfs-dfs-bing-cha-ji-san-chong-fang-fa-pan-duan-er-/>

代码如下：

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        vector<int> color(n,0);

        for(int i=0; i<n; ++i) {

            if(color[i] == 0) {

                if(!dfs(graph, i, 1, color)) return false;

            }

        }

        return true;

    }

    bool dfs(vector<vector<int>>& graph, int node, int given\_color, vector<int>& color) {

        if(color[node] != 0) {

            return given\_color == color[node];

        }

        color[node] = given\_color;

        for(auto adj : graph[node]) {

            if(!dfs(graph, adj, -given\_color, color)) return false;

        }

        return true;

    }

};

还有第三种方法：并查集

class UF {

private:

    int count\_;

    vector<int> parent\_;

    vector<int> size\_;

    int find(int x) {

        while(parent\_[x] != x) {

            parent\_[x] = parent\_[parent\_[x]];

            x = parent\_[x];

        }

        return x;

    }

public:

    UF(int n) {

        count\_ = n;

        parent\_.resize(n);

        size\_.resize(n);

        for(int i=0; i<n; ++i) {

            parent\_[i] = i;

            size\_[i] = 1;

        }

    }

    void nodeUnion(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        if(root\_p == root\_q) return;

        if(size\_[root\_p] > size\_[root\_q]) {

            parent\_[root\_q] = root\_p;

            size\_[root\_p] += size\_[root\_q];

        }

        else {

            parent\_[root\_p] = root\_q;

            size\_[root\_q] += size\_[root\_p];

        }

        --count\_;

    }

    bool isConnected(int p, int q) {

        int root\_p = find(p);

        int root\_q = find(q);

        return (root\_p == root\_q);

    }

    int count() {

        return count\_;

    }

};

class Solution {

public:

    bool isBipartite(vector<vector<int>>& graph) {

        int n = graph.size();

        UF uf(n);

        for(int i=0; i<n; ++i) {

            auto& adjs = graph[i];

            for(auto adj : adjs) {

                if(uf.isConnected(i, adj)) {

                    return false;   // 如果当前节点已经跟邻居一个集合（颜色）了，则错误

                }

                uf.nodeUnion(adjs[0], adj);

            }

        }

        return true;

    }

};

/\*

并查集写法，把每个节点的后继节点都联通起来，并且判断当前节点是否已经和邻居“勾结“

\*/

## 210 课程表II

拓扑排序类问题，步骤如下：

1. 建立邻接表，统计入度
2. 广度优先入度为0的节点。注意在找邻居的时候入度--。表示该节点已删除。

class Solution {

public:

    vector<int> findOrder(int numCourses, vector<vector<int>>& prerequisites) {

        vector<vector<int>> graph(numCourses);

        vector<int> indgree(numCourses, 0);

        vector<int> res;

        for(const auto& prerequisite : prerequisites) {

            graph[prerequisite[1]].push\_back(prerequisite[0]);

            ++indgree[prerequisite[0]];

        }

        queue<int> q;

        for(int i=0; i<numCourses; ++i) {

            if(indgree[i] == 0) {

                q.push(i);

            }

        }

        while(!q.empty()) {

            int u = q.front();

            q.pop();

            res.push\_back(u);

            for(auto v : graph[u]) {

                --indgree[v];

                if(indgree[v] == 0) {

                    q.push(v);

                }

            }

        }

        for(auto x : indgree) {

            if(x!=0) return vector<int>();

        }

        return res;

    }

};

## 1059 从始点到终点的所有路径

class Solution {

public:

    bool leadsToDestination(int n, vector<vector<int>>& edges, int source, int destination) {

        vector<vector<int>> graph(n);

        for(auto edge : edges) {

            graph[edge[0]].push\_back(edge[1]);

        }

        vector<bool> visited(n, false);

        return dfs(graph, source, destination, visited);

    }

    // 从start节点出发能否所有终点都到达target

    bool dfs(vector<vector<int>>& graph, int start, int destination, vector<bool>& visited) {

        if(visited[start]) return false;

        if(graph[start].empty() && start != destination) return false;

        visited[start] = true;

        for(auto adj : graph[start]) {

            if(!dfs(graph, adj, destination, visited)) return false;

        }

        visited[start] = false; // 神奇，加了这句话就对了

        return true;

    }

};

/\*

这个题我写的很好

难点在于对环路的处理，对于这种bool类型的递归或者回溯，一个比较好用的办法就是找true的情形或者false的情形进行排除

我就是挑出了所有false的情形。

同时回溯必不可少，因为要寻找多条路，所以visited既要拿来判断环路，又可以用来回溯。

\*/