# 图

## 1. 图的存储

### 1.1 邻接矩阵

根据题目数据范围，2000顶点数以内可用

直接开2维数组 / vector即可

### 1.2 邻接表

```
vector<pair<int,int>> edge[100010]; //带权 // 常数取决于n的范围
vector<int> edge[100010]; // 无权
struct EDGE{
    int next, to, w;
}edge[100010];
int main(void)
{
    // 读入
    cin >> n >> m;
    for(int i=1;i<=m;i++){
        int u,v,w;
        cin >> u >> v >> w;
        edge[u].push_back(<v,w>);
    }

    // 遍历
    for(int i=0;i<edge[now].size();i++){
        int from = now, to = edge[now][i].first, weight = edge[now][i].second;
    }
    return 0;
}
```

## 2. 图的基本搜索

复杂度O(n+m)

对于无向图只能遍历到当前连通分量内的顶点和边

对于有向图会更少

算法本身和普通搜索一样，重点在于搜索过程中维护的信息

连通分量计数

### 2.1 DFS

```
vector<int> edges[100010];
int vis[100010] = {0};

void dfs(int now){ // 根据具体题目需要再带其他参数
    if(vis[now]) return;
```

```
    // if 终止条件 维护xxx信息
    vis[now] = 1;
    // 遍历
    for(int i=0;i<edge[now].size();i++){
        int from = now, to = edge[now][i].first, weight = edge[now][i].second;
        // 可能需要维护的信息
        dfs(to);
        // 可能需要回溯的信息
    }
}
```

## 2.2 BFS

```
queue<int> q;
int vis[100010] = {0};
vector<int> edges[100010];
void bfs(int st, int ed){
    q.push(st);
    vis[st] = 1;
    while(q.size()){
        int now = q.front();
        q.pop();
        // if(now == ed) 可能需要维护的终止信息
        // 遍历
        for(int i=0;i<edge[now].size();i++){
            int from = now, to = edge[now][i].first, weight = edge[now]
[i].second;
            if(!vis[to]){
                // do something
                q.push(to);
                vis[to] = 1;
            }
        }
    }
}
```

# 3. 拓扑排序

**有向无环图（DAG）：**

能拓扑排序的图，一定是有向无环图；有向无环图，一定能拓扑排序；

用拓扑排序判定即可

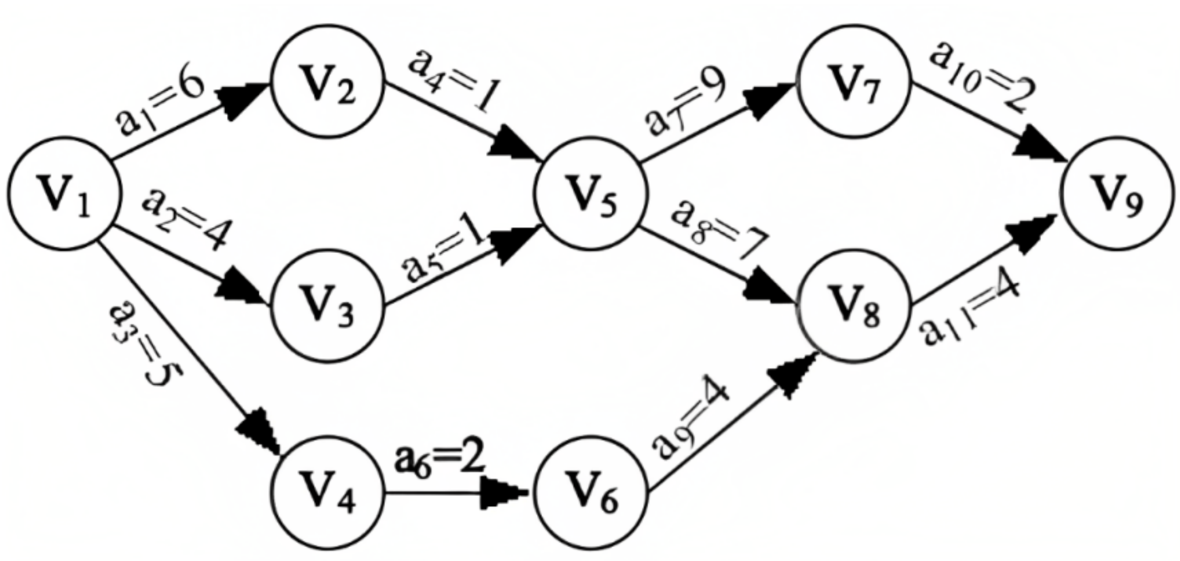
复杂度线性

```
int n, m;
vector<int> edges[100010];
int in[100010];  // 提前存储每个结点的入度（通常在读入时同时记录）

queue<int> q;
vector<int> sort_res; // 存放拓扑排序的排序结果
int toposort(){
```

```
    for(int i=1;i<=n;i++){
        if (in[i] == 0) q.push(i);
    }
    while(q.size()){
        int now = q.front(); // q.top()
        q.pop();
        sort_res.push_back(now);
        for(int i=0;i<edges[now].size();i++){
            int to = edges[now][i];
            in[to]--;
            if(in[to] == 0){ // 如果更新入度后 入度变为0 放入队列
                q.push(to);
            }
        }
    }
    if (sort_res.size() == n) return 1; // 是有向无环图
    return 0; // 不是有向无环图 nlogn
}
```

应用场景 除了有向无环图 还有**AOV网络** 求最早可以完成的时间等




**求字典序最大 / 最小的拓扑排序**

把上述算法中普通队列换成优先队列。

例如让求字典序最小，把下面

```
struct node{
    int node_idx; // 顶点编号 相当于上面放进队列的值
    char ch; // 顶点的字符
}
// 在 priority_queue 里，operator() 返回 true 表示 “a 的优先级低于 b”
struct cmp {
    bool operator()(const node& a, const node& b) const {
        return a.ch > b.ch;  // a的字符更大时 表示a的优先级低于b
    }
};
priority_queue<node, vector<node>, cmp> que; // 优先
```

# 4. 最小生成树

## 4.1 Prims 算法

适合稠密图 朴素实现复杂度O(n^2) 堆优化后变为O(mlogn)

思路：任选起点作为初始连通块 --> 每次与连通块相邻边中权重最小的 --> 把该边和一个新的顶点加入连通块 --> 直到所有顶点都在连通块内

```
// O(n^2)的实现
// 堆优化在于找到与连通块最近的点时不用遍历所有顶点
//Prim算法需要的两个数组
int dis[N]; // dis[i]: 顶点i到连通块的距离
bool vis[N];
void Prim(){
    //默认从1开始，则从2起，我们需要把dis设置为极大值
    for (int i=2;i<=n;i++){
        dis[i]=INF;
    }
    //首先从1开始，设置与1相连的所有边的权值
    for (int i=head[1];i;i=edge[i].next){
        int v=edge[i].to;
        dis[v]=min(dis[v],edge[i].w);//可能存在重边
    }
    int now=1;//当前点为1
    //注意：i少遍历一次，因为1省略了
    for (int i=1;i<n;i++){
        vis[now]=true;
        int minF=INF;
        //遍历所有的点，找到未遍历过的，并且边权值最小的那一条边
        for (int j=1;j<=n;j++){
            if (!vis[j] && dis[j]<minF){
                minF=dis[j]; //这一条边的权值
                now=j; //当前这条边的一个顶点
            }
        }
        ans+=minF; //边权相加
        //遍历所有与now相连的边
        for (int j=head[now];j;j=edge[j].next){
            int v=edge[j].to; //边的另一个顶点
            if (!vis[v] && dis[v]>edge[j].w){
                //更新dis数组中这边
                dis[v]=edge[j].w;
            }
        }
    }
    for (int i=1;i<=n;i++){
        if (dis[i]==INF){
            cout<<"impossible\n";
            return;
        }
    }
    cout<<ans;
```

```
}
```

### 4.2 kruskal 算法

适合稀疏图 复杂度O(mlogm)

思路：所有边按边权升序排序 --> 遍历所有边，判断它的两个顶点是否已连通 --> 是则不加这条边，否则加入 --> 更新顶点连通状态

这里的顶点连通状态用并查集表示

**有空可以先讲一下并查集**

```
int parent[N];
int sz[N];
//并查集初始化
void init(){
    for (int i=1;i<=n;i++){
        parent[i]=i;
    }
}
// 查找连通块：并查集的查找+路径压缩+按rank合并
// 食物链 A B C n
void find_set(int x){
    if (x!=parent[x]){
        parent[x]=find_set(parent[x]);
    }
    return parent[x];
}
void kruskal(){
    //将所有的边按照权值从小到达排序
    comp(edge+1,edge+1+m,comp); //注意此处为m边数，不是顶点数
    init();
    int cnt=0;//记录边的数量
    int ans=0;//记录最后的最小生成树的权值之和
    for (int i=1;i<=m;i++){
        //获得边与边权
        int a=edge[i].a,b=edge[i].b=edge[i].w;
        //查找某一条边的对应的两个顶点
        int pa=find_set(a),pb=find_set(b);
        //如果这两个顶点不属于同一个集合中，则我们直接把他们合并即可。
        if (pa!=pb){
            //则合并到同一连通块
            parent[pa]=pb;
            //ans加上此边权
            ans+=w;
            //遍历的边数+1
            cnt++;
        }
        //直到最后遍历到了最后一个顶点
        if (cnt==n-1) break;
    }
    return ans;
}
```

# 5. 最短路

## 5.1 dij

适用于没有负权边的情况

思路：维护"已确定距离"的顶点集合，每次找到集合外离起点距离最小的顶点，加入该集合，并更新这个顶点的邻居的距离

**O(n^2) 的朴素版本**

```
struct edge {
  int v, w;
};

vector<edge> e[MAXN];
int dis[MAXN], vis[MAXN];
// dis[i]: 目前顶点i到起点的最近距离

void dijkstra(int n, int s) { // s --> e
  // 初始化距离为无穷大 只有dis[s] = 0
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0;

  for (int i = 1; i <= n; i++) {
    // 遍历n轮 每轮从没有访问过的顶点中找到目前离s最近的 记作u
    int u = 0, mind = 0x3f3f3f3f;
    for (int j = 1; j <= n; j++)
      if (!vis[j] && dis[j] < mind)
        u = j, mind = dis[j];
    // 把u放入“已知距离集合” 更新u的邻居的距离
    vis[u] = true; s --> u -->
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w;
      if (dis[v] > dis[u] + w) dis[v] = dis[u] + w;
    }
  }
}
```

**O(nlogm)的堆优化版本**

```
struct edge {
  int v, w;
};

// 优先队列中 dis小的更优先
// operator() 返回 true 表示 “a 的优先级低于 b”
// 即 dis更大的node优先级更低
struct node {
  int dis, u;
  bool operator>(const node& a) const { return dis > a.dis; }
};
```

```
vector<edge> e[MAXN];
int dis[MAXN], vis[MAXN];
// 优化"寻找u"的过程

priority_queue<node, vector<node>, greater<node>> q;

void dijkstra(int n, int s, int e) {
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  memset(vis, 0, (n + 1) * sizeof(int));
  dis[s] = 0;

  q.push({0, s});
  while (!q.empty()) {
    // 取优先队列的队首为u 即队列中dis最小的node
    int u = q.top().u;
    q.pop();
    if (vis[u]) continue;

    // 以下和O(n^2)版本一样 更新u的邻居
    vis[u] = 1;
    for (auto ed : e[u]) {
      int v = ed.v, w = ed.w; // v w
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        q.push({dis[v], v});
      }
    }
  }
  return dis[e];
}
```

### 5.2 floyd

100 - 500

搭配邻接矩阵存储法使用

时间复杂度O(n^3)

```
void floyd(){
    for (k = 1; k <= n; k++) {
      for (x = 1; x <= n; x++) {
        for (y = 1; y <= n; y++) {
          f[k][x][y] = min(f[k - 1][x][y], f[k - 1][x][k] + f[k - 1][k][y]);
        }
      }
    }
}
```

### 5.3 Bellman–Ford以及负环判断

SPFA 网格图

最短路计算适用于没有负权环的情况

松弛 u --> v  dis[v] = dis[u] + w

```
struct Edge {
  int u, v, w;
};

vector<Edge> edge;

int dis[MAXN], u, v, w;
constexpr int INF = 0x3f3f3f3f;

bool bellmanford(int n, int s) {
  // 初始化距离为无穷大 只有dis[s] = 0
  memset(dis, 0x3f, (n + 1) * sizeof(int));
  dis[s] = 0;
  bool flag = false;  // 判断一轮循环过程中是否发生松弛操作
  // 最多成功松弛n轮 否则说明有负环
  for (int i = 1; i <= n; i++) { // i
    flag = false;
    // 每次遍历所有边
    for (int j = 0; j < edge.size(); j++) {
      u = edge[j].u, v = edge[j].v, w = edge[j].w; // u-->v w
      if (dis[u] == INF) continue; // dis[u]  --> dis[v] = dis[u]+w;
      // 无穷大与常数加减仍然为无穷大
      // 因此最短路长度为 INF 的点引出的边不可能发生松弛操作
      // 如果使用边j可以松弛 则更新距离
      if (dis[v] > dis[u] + w) {
        dis[v] = dis[u] + w;
        flag = true;
      }
    }
    // 没有可以松弛的边时就停止算法
    if (!flag) {
      break;
    }
  }
  // 第 n 轮循环仍然可以松弛时说明 s 点可以抵达一个负环 （负环判断）
  return flag;
}
x --> v_i
```

需要注意的是，以 S 点为源点跑 Bellman–Ford 算法时，如果没有给出存在负环的结果，只能说明从 S 点出发不能抵达一个负环，而不能说明图上不存在负环。

因此如果需要判断整个图上是否存在负环，最严谨的做法是建立一个超级源点，向图上每个节点连一条权值为 0 的边，然后以超级源点为起点执行 Bellman–Ford 算法。

### 5.4

**这里有空可以讲一下非简单路径计数**

简单路径 非简单路径

s t --> k 路径

邻接矩阵 A[i] [j] A[i] [j] 1-hop

A^k --> [i] [j]

n * n  --> n^3

n^2logn  A^n 矩阵快速幂

# 6. 一些也许有用的思想技巧

## 6.1 引入顶点

一个例子：给定一个顶点集合和一个顶点t，要求顶点集合中到t距离最小的顶点到t的距离。

当然也可以在反向图上从t出发来做。 dij

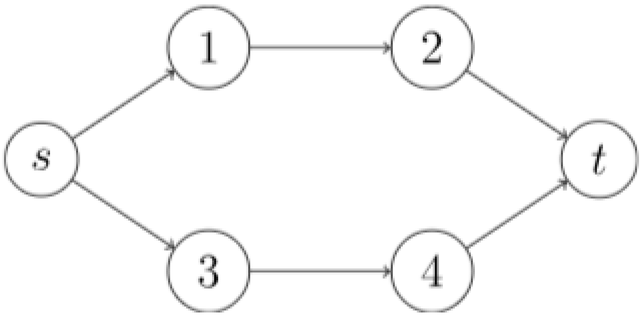
这里想说的是，可以人为添加一个源顶点s，从s到顶点集合中所有顶点建权重为0的边，再从s出发，求到t的最短路。

## 6.2 拆点

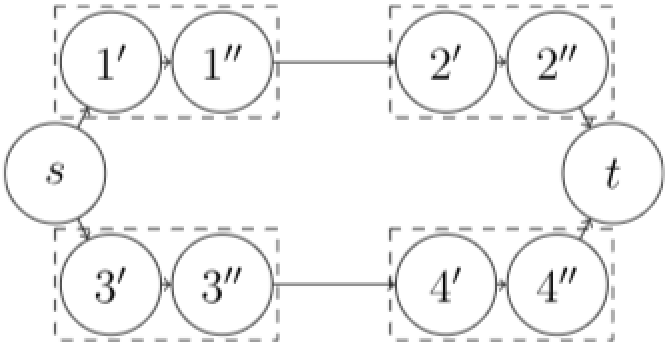
通常权重是在边上的，如果在点上（例如每个点有点权，让算最短路）

可以把问题转化为边带权的情况

如果原图是这样：




拆点之后的图是这个样子：




### 6.3 差分约束

例题 https://www.luogu.com.cn/problem/P1993

概括就是，n个变量，之间有相等和不等关系（x1 > x2+c、x1 < x2+c、x1 = x2等）

问是否存在一组n个变量的赋值，满足所有关系，存在的话输出

图解可以参考 https://zhuanlan.zhihu.com/p/104764488

根据不等关系建图 --> 建超级源点求到所有点最短路 +c

dis[v] <= dis[u] + w