目录

存图

链式前向星

```
//链式前向星
struct Edge{
    int to,w,next;
}edge[Maxn];
int cnt=0;
int head[Maxn];
void init(){
    for(int i=0;i<Maxn;++i){
```

```
        edge[i].next=-1;
        head[i]=-1;
    }
    cnt=0;
}
//加的单边
void addedge(int u,int v,int w){
    edge[cnt].next=head[u];
    edge[cnt].to=v;
    edge[cnt].w=w;
    head[u]=cnt++;
}
//遍历节点u的所有可到达的节点
for(int i=head[u];~i;i=edge[i].next){

}
```

邻接表

```
//邻接表
struct Edge{
    int u,v,w;
    Edge(int a,int b,int c){
        u=a,v=b;w=c;
    }
};
vector<Edge>edge[Maxn];
//初始化
for(int i=1;i<=n;i++){
    edge[i].clear();
}
//存边
edge[a].push_back(edge(a,b,c));
//遍历节点u可到达的节点
for(int i=0;i<edge[u].size();i++){

}
```

二分图

题目

- Codeforces-1105E-Helping Hiasat

无向图的最大独立集

概念

"任意两点之间都没有边相连"的点集被称为无向图的独立集,包含点数最多的一个就是图的最大独立集

无向图的最大团

概念

"任意两点之间都有一条边相连"的子图被称为无向图的团,点数最多的团被称为图的最大团

实现

调用函数:

```
//邻接矩阵存图
int n;  //点的个数
int mp[N][N];
int cnt[N],vis[N];  //cnt[i]当前最大团的节点数,vis记录当前最大团的节点
int ans,group[N];   //ans记录最大团的节点数,group记录答案最大团的所有节点
bool dfs(int u, int dep)    //当前点u,搜索深度dep
{
    int j;
    for(int i=u+1;i<=n;i++)
    {
        if(cnt[i]+dep<=ans) //剪枝
            return false;
        if(mp[i][u])
        {
            for (j = 0; j < dep; j++)   //判断是否与当前最大团中各个点相连
                if (!matrix[i][vis[j]])
                    break;
            if(j == dep)
            {
                vis[dep]=i;
                if(dfs(i,dep+1))return true;
            }
        }
    }
    if(dep > ans)
    {
        for(int i=0;i<dep;i++)group[i]=vis[i];
        ans=dep;
    }
    return false;
}
```

调用方式:

```
ans = 0
for(int i=n;i>=1;i--)
{
    vis[0]=i;
    dfs(i,1);
```

```
        cnt[i]=ans;
    }
```

定理

- 无向图的最大团等于其补图的最大独立集
- $G'=(V,E')$为$G=(V,E)$的补图.其中$E'=\left{ (x,y)\notin E\right}$

求补图

```
//n*m的图用邻接矩阵存储
for(int i=1;i<=n;i++)
    for(int j=1;j<=m;j++)
        mp[i][j]^=1;
```

最短路

题目

- P1462 通往奥格瑞玛的道路
- AC-wings 通信线路

最短路常用算法

Dijkstra

- 适用条件：无负权边
  - 虽说如此，但也并不是有负权就严格不能用，具体情况具体分析。比如，如果原图保证是有向无环，那么还是可以使用的。
- 算法思想： 贪心
  - 选择某个距离原点最近的点为中介，更新所有其他点。重复这个过程。
  - 基于原理，可以使用堆进行优化，快速完成得到距离原点最近的点。
- 复杂度：mlog(n)

SPFA

- 可以处理负权边
- 算法思想：迭代
  - 尝试以每个点为中介，更新其余点到起点的距离
- 复杂度： km，一般k为较小常数，最坏情况nm
- 无负权边的时候，可以采用堆优化，此时和堆优化的Dijkstra完全一致。

Folyd

- 适用条件：可以求出任意两点之间的最短路
- 算法思想：dp
  - 最外层枚举可利用的前k个节点，之后利用新加入的节点来更新其他节点之间的最短路

- 其实和SPFA很像，只不过前者固定了原点
- 应用：
  - 传递闭包
- 复杂度： n^3

实现

- Dijkstra

```
struct edge
{
    int pos, val;
    edge( int pos = 0, int val = 0 ) : pos(pos), val(val) {}
    bool operator < ( const edge &e ) const
    {
        return val > e.val;
    }
};
vector< edge > G[maxn];
int dis[maxn];
bool vis[maxn];
void Dijkstra( int s )
{
    memset( dis, 0x3f, sizeof( dis ) );
    priority_queue< edge > q;
    dis[s] = 0; q.push({s,dis[s]});
    while( !q.empty() )
    {
        auto tp = q.top( ); q.pop( );
        if ( vis[tp.pos] ) continue;
        //要在这时候判断是否访问过，因为一个点可能会加入队列很多次，要取权值最小的一次，所以每次更新都要入队，而不是在队列里就不入队
        inq[tp.pos] = 1;
        for ( auto v : G[tp.pos] )
        {
            if( dis[v.pos] > dis[tp.pos] + v.val )
            {
                dis[v.pos] = dis[tp.pos] + v.val;
                q.push( {v.pos, dis[v.pos]} );
            }
        }
    }
}
```

- SPFA

```
struct node
{
    int to;
    long long val;
```

```
    node( int to = 0, long long val = 0 ) : to(to), val(val) {}
};
vector< node > G[maxn];
long long dis[maxn];
bool inq[maxn];
void SPFA( int s )
{
    memset( inq, 0, sizeof( inq ) );
    for( int i = 1; i < maxn; ++ i) dis[i] = 1E18;
    dis[s] = 0, inq[s] = 1;
    queue< int > q; q.push(s);
    while( !q.empty() )
    {
        int x = q.front(); q.pop();
        inq[x] = 0;
         //这里和堆优化的区别就显现出来了，堆优化版本只会入队一次，而SPFA则不是
        for( auto to : G[x] )
        {
            if( dis[to.to] > dis[x] + to.val )
            {
                dis[to.to] = dis[x] + to.val;
                if( !inq[to.to] ) q.push(to.to), inq[to.to] = 1;
            }
        }
    }
}
```

- Floyd

```
//最开始dp就是原图
for( int k = 1; k <= n; ++ i )
    for( int i = 1; i <= n; ++ i )
        for( int j = 1; j <= n; ++ j )
            dp[i][j] = min( dp[i][j], dp[i][k] + dp[k][j] );
```

常见套路

- 与最短路有关的题常与二分相关联
    - 常见问法，在满足某个调价的约束条件下，另一个条件最大或者最小。答案具有单调性。
- 弗洛伊德经常和具有传递性关系的题目结合。或者需要知道利用某个几个节点时任意两点间的最短路，即历史状态。
- 有时候可以考虑补图，反图(每条边的方向取反)等。
    - 反图主要处理的就是顺序问题。比如一个图，每个点都有一个点权。若想知道到某个点的所有路径中点权最大和最小的点的点权，并且要求点权最小的点在点权最大的点之后出现，这时候就可以建立一个反图，然后从终点向起点走。
    - 补图主要是在处理与二分图相关的问题时来进行考虑。
- 要分析题目性质，有时候的最短路可以基于已知信息计算出来。

LCA

求法

- 倍增
- Tarjan
  - Tarjan主要是利用DFS顺寻。
  - 节点分为三种，分别是正在访问，未访问，已经访问完成并且经过回溯的点
  - 刚开始每个人的祖先都是自己
  - 在孩子里进行递归，递归完成后将孩子的father置成自己。这样就保证了只有回溯完成后father才会变动
  - 然后进行询问操作
    - 如果要询问的点已经回溯完成，那么他的father就是和当前点的LCA。因为只有回溯完成的点father才会变更，并且由于dfs的原因，待询问点与当前点LCA一定是已经访问并且还未回溯的点，他的father并没有发生变化。
    - 否则不进行处理。
  - 回溯完成。

实现

- Tarjan

```
int Get( int x ){ return x == fa[x] ? x : fa[x] = Get( fa[x] ); }
void Tarjan( int s )
{
    vis[s] = 1;
    for( auto to : G[s] )
        if( !vis[to] )
        {
            Tarjan(to);
            fa[to] = s;
        }
    for( auto to : query[s] )
        if( vis[to] == 2 )
        {
            dx[to] += 1;
            dx[s] += 1;
            dx[Get(to)] -= 2;
        }
    vis[s] = 2;
}
```

- 倍增

```
void DFS( int s, int last, int depth, int distance )
{
    vis[s] = 1;
    fa[s][0] = last;
    dis[s] = distance;
    dep[s] = depth;
```

```
    for( int i = 1; (1 << i) <= depth; ++ i )
        fa[s][i] = fa[fa[s][i-1]][i -1];

    for( auto v : G[s] )
        if( !vis[v.to] )
            DFS( v.to, s, depth + 1, distance + v.w);
}
int Query( int x, int y )
{
    if( dep[x] < dep[y] ) swap(x, y);
    while( dep[x] > dep[y] ) x = fa[x][lg[dep[x] - dep[y]] - 1];
    if( x == y ) return x;
    for( int k = lg[dep[x]] - 1; k >= 0; k -- )
        if( fa[x][k] != fa[y][k] ) x = fa[x][k], y = fa[y][k];
    return fa[x][0];
}
```

最小生成树

算法

- Kruskal
    - $mlogm$
- Prime
    - $O(n^2)$
    - 主要应用于稠密图，尤其是完全图的最小生成树求解

实现

- KrusKal

```
int fa[maxn];
void Init()
{
    for( int i = 1; i <= n; ++ i ) fa[i] = i;
}
int GetFa( int x ) { return  x == fa[x] ? x : fa[x] = GetFa(fa[x]); }
bool Connect( int u, int v )
{
    u = GetFa(u), v = GetFa(v);
    if( u == v ) return false;
    fa[u] = v;
    return true;
}
int KrusKal()
{
    int ans = 0;
    for( auto i : e ) if( Connect( i.u, i.v ) ) ans += i.w;
    return ans;
}
```

- Prime

```
double Prime()
{
    double ans = 0;
    for( int i = 1; i <= n; ++ i ) dis[i] = INF;
    dis[1] = 0;
    for( int i = 1; i <= n; ++ i )
    {
        int x = 0;
        for( int j = 1; j <= n; ++ j )
            if( !vis[j] && ( x == 0 || dis[x] > dis[j] ) )
                x = j;
        ans += dis[x];
        vis[x] = 1;
        for( int j = 1; j <= n; ++ j )
           if( !vis[j] ) dis[j] = min( dis[j], Calc( v[x], v[j] ) );
    }
    return ans;
}
```

题目

- P1991 无线通讯网
  - 最小生成树，但其中可以选择其中几条边把其边权变为0，问最大边权最小是多少
  - 一般这种既要最大又要最小的问题都可以用二分
  - 关键就是把那些边去掉
    - 贪心的想肯定是去掉边权最大的几条
    - 这样问题就转化为，枚举最大距离，当出现一条边权大于枚举的距离时有几个连通块，就要使用几次去边操作
- P1265 公路修建
  - 由于是完全图，所以Kruskal会MLE
  - 考虑用Prime
    - 但是空间依然不够用
    - 其实不用存下来每两点之间的距离，只要在更新时计算就好了

欧拉图

定义

- 欧拉路径
  - 图G中的一个路径经过每个边恰好一次
- 欧拉回路
  - 一个回路时欧拉路径
- 欧拉图
  - 具有欧拉回路的图。据偶欧拉路径但不具有欧拉回路的图成为半欧拉图。

定理(前提均是图联通)

- 欧拉回路
  - 无向图：无奇数度顶点
  - 有向图：每个点的入度等于出度
- 欧拉路径
  - 无向图：奇度数顶点为0或2，两个奇数度顶点一个为起点另一个为终点。
  - 有向图：可以存在两个点，入度不等于出度，其中一个入度比出度大1，另一个出度比入度大1

题目

- P1341 无序字母对
  - 题意：给定n对各不相同的字母，要求使用n+1个字母构造一个字符串，使给定的n对字母相邻(可以交换顺序)，如果有多个答案，输出字典序最小的方案
  - 发现就是找一条字典序最小的欧拉路径。
  - 本来用vector存图，但发现不好删边。为什么要删边呢，因为欧拉路径中一个点可以经过多次，只是每条边只能经过一次。
  - 要找字典序最小，那么每次寻找的时候就先从字典序小的路径开始找。
    - 但这里有个问题，如果

    ```
        void DFS( int s )
           {
               for( int i = 'A'; i <= 'z'; ++ i )
                   if( G[s][i]){ G[s][i] --; G[i][s] --; DFS(i); }
               ans += s;
           }
    ```

    然后reverse ans就是对的,但是

    ```
    void DFS( int s )
    {
        ans += s;
        for( int i = 'A'; i <= 'z'; ++ i )
            if( G[s][i]){ G[s][i] --; G[i][s] --; DFS(i); }
    }
    ```

    就是错的，为什么呢?

    - 比如

      > 7 ab ac cd da ae ef fa

      前者输出abcdaefab，后者输出abcdaefa，后者显然不对。。。因为从a到b再回来，答案应该是aba但是如果按照后边的写法，就会是ab了，换句话说，a出现的时机不对，本应该回溯时就加进答案，但后一种写法并没有。