ACM Template

DUT ACM Lab

2021 年 9 月 24 日

目录

第一章　STL 使用

1.1 set 与 multiset

1.2 map 与 unordered_map

第二章　图论

2.1　二分图

2.1.1　二分图最大匹配

2.1.1.1　实现

第一种做法

匈牙利算法，复杂度 $\Theta(nm)$

```
const int mx_n = 1005;
bool mp[mx_n][mx_n];
bool vis[mx_n];
int pre[mx_n];

bool dfs(cint loc) {
    for(int i=n+1; i<=n+m; i++) {
        if(mp[loc][i] && !vis[i]) {
            vis[i] = 1;
            if(!pre[i] || dfs(pre[i])) {
                pre[i] = loc;
                return 1;
            }
        }
    }
    return 0;
}

int main() {
    int ans = 0;
    for(int i=1; i<=n; i++) {
        memset(vis, 0, sizeof vis);
        ans += dfs(i);
    }
    cout << ans << endl;
    return 0;
}
```

第二种做法

转化为网络流，复杂度依赖选择

2.1.1.2 性质

最大独立集 = n - 最大匹配
最小点覆盖 = n - 最大独立集

2.2 树

2.2.1 树的重心

2.2.1.1 实现

对于树上的每一个点，计算其所有子树中最大的子树节点数，这个值最小的点就是这棵树的重心。

```
void dfs(cint loc, cint fa) {
    int pre = 0;
    son[loc] = 1;
    for(int v: to[loc]) {
        if(v != fa) {
            dfs(v, loc);
            pre = max(pre, son[v]);
            son[loc] += son[v];
        }
    }
    pre = max(pre, n-son[loc]);
    if(pre < st) {
        st = pre; # 最大儿子的最小值
        ans = loc; # 最大儿子的编号
    }
}
```

2.2.1.2 性质

1. 一棵树最多有两个重心，且相邻
2. 以树的重心为根时，所有子树的大小都不超过整棵树大小的一半
3. 在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离
4. 把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上
5. 树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到它们的距离和一样

2.2.1.3 一点证明

2.2.1.3.1 1 首先证明如果树有两个重心，则它们必相邻

设两点分别为 a ，b ，且它们之间的简单路径经过的点的个数不为 0 ，即它们不相邻
不妨认为 a 在树中的深度大于 b
那么，点 a 向上的子树大小一定**大于**点 b 向上的子树，同时**大于**点 b 向下不经过点 a 的子树
同理，点 b 向下经过点 a 的子树一定**大于**点 a 向下的子树
所以，最大值仅由这两棵子树决定，而只要两点不相邻，上述总是成立的
不难发现，这两种子树，在 a 或 b 沿着两点间简单路径移动时会减小，不符合重心的定义
再证最多只有两个重心
显然，如果树的重心大于两个，至少有一对重心无法相邻

2.2.2 点分治

2.2.2.1 思路

如果在统计树上信息时，可以将子树内的信息单独统计，将多个子树的信息合并统计，那么可以考虑树分治。

如果对于每一次分治，复杂度为 $\Theta(N)$ ，那么 k 次递归的复杂度就是 $\Theta(kN)$

如果能保证递归次数为 log 级别，那么复杂度就会是 $\Theta(N\log N)$ ，而从重心分值就可以保证这种性质

2.2.2.2 实现

预定义部分

```
int h[10010], nx[20020], to[20020], w[20020], cnt; // 链式前向星数组
int son[10010]; // 经过处理后的每个点的儿子个数
bool vis[10010]; // 该点是否在分治时作为子树的根
int id; // 当前所处理的树的重心
int snode; // 当前所处理树的节点数量

void add(cint f, cint t, cint co) {
    nx[++cnt] = h[f];
    h[f] = cnt;
    to[cnt] = t;
    w[cnt] = co;
}
```

统计以某点为根且不跨越其余根的子树大小

```
int gsiz(cint loc, cint fa) {
    int sum = 1;
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            sum += gsiz(to[i], loc);
        }
    return sum;
}
```

寻找树的重心

```
void gp(cint loc, cint fa) {
    int pre = 0;
    son[loc] = 1;
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            gp(to[i], loc);
            pre = max(pre, son[to[i]]);
            son[loc] += son[to[i]];
            if(id) return;
        }
    pre = max(pre, snode - son[loc]);
    // 树的重心可能有两个，此处任取了一个
    if(pre <= snode/2) {
```

```
        id = loc;
        return;
    }
}
```

统计跨越重心的答案

```
void check(cint loc, cint fa) {
    // 统计跨越重心的答案
    for(int i=h[loc]; i; i=nx[i])
        if(to[i] != fa && !vis[to[i]]) {
            check(to[i], loc);
        }
}
```

合并子树

```
void update(cint loc, cint fa) {
    // 合并子树
    for(int i=h[loc]; i; i=nx[i]) {
        if(to[i] != fa && !vis[to[i]]) {
            update(to[i], loc);
        }
    }
}
```

解决问题

```
void sol(cint loc) {
    vis[loc] = 1;
    // 初始化计算答案与合并子树时需要用到的东西
    for(int i=h[loc]; i; i=nx[i]) {
        if(!vis[to[i]]) {
            check(to[i], loc, w[i]);
            update(to[i], loc, w[i]);
        }
    }
    for(int i=h[loc]; i; i=nx[i]) {
        if(!vis[to[i]]) {
            snode = gsiz(to[i], loc);
            id = 0;
            gp(to[i], loc);
            sol(id);
        }
    }
}
```

主函数里的一点东西

```
int main() {
```

```
    snode = n;
    gp(1, 1);
    sol(id);
}
```

第三章　数论

第四章　计算几何

第五章　博弈理论

第六章　数据结构

6.1　树状数组

```
int bnode[mx_n];

int lowbit(int &x) { return x&-x; }

void add(int x, cint co) {
    while(x <= mx_n) {
        bnode[x] += co;
        x += lowbit(x);
    }
}

int query(int x) {
    int ans = 0;
    while(x) {
        ans += bnode[x];
        x -= lowbit(x);
    }
    return ans;
}
```

注意，树状数组无法直接处理 **0** ，需要处理一下

第七章　典型题型

7.1　最长上升子序列

7.1.1　做法

第一种：

dp，复杂度 $\Theta(N^2)$ ，优点是可以记录子序列

```
// Nope
```

第二种：

贪心，复杂度 $\Theta(N \log N)$ ，优点是复杂度低

```
int mx[mx_n];
int r = 0;
memset(mx, 0x3f, sizeof mx);
mx[0] = 0;
for(int i=1; i<=m; i++) {
    int id = lower_bound(mx, mx+r+1, c[i]) - mx;
    mx[id] = c[i];
    r = max(r, id);
}
# 最后 mx 数组合法的最大的下标就是答案
```

第八章　杂项