搜索、树与图论

Saya

洛谷
www.luogu.com.cn

DFS(搜索)

DFS 为深度优先搜索。

算法大致框架：

```
void dfs(int k) {
    if (到达终止状态) {
        // 统计答案
        return ;
    }
    for (所有可能的选择) {
        // 存储下这种选择
        dfs(k + 1); // 递归下一层
        // 回溯，删除刚刚的选择
    }
}
```

P10448 组合型枚举

从 $1 \sim n$ 这 n 个整数中随机选出 m 个，输出所有可能的选择方案

对于所有数据，满足 $0 \leq m \leq n, n + (n - m) \leq 25$。

Solution

套用 P2 中 DFS 的模板，依次枚举第 i 个数应该选择什么。

但是由于是组合型枚举，输出时要求按升序输出，所以第 $i+1$ 次选择的数，应该大于第 i 次选择的数，所以我们额外记录一个变量 las，每次枚举时从 las+1 枚举到 n。

代码

```
int n, m, ans[30];
void dfs(int k, int las) {
    if (k == m + 1) {
        for (int i = 1; i <= m; i++) cout << ans[i] << " ";
        cout << "\n";
        return ;
    }
    for (int i = las + 1; i <= n; i++) {
        ans[k] = i;
        dfs(k + 1, i);
        ans[k] = 0;
    }
}
```

P5662 纪念品

小伟突然获得一种超能力，他知道未来 T 天 N 种纪念品每天的价格。某个纪念品的价格是指购买一个该纪念品所需的金币数量，以及卖出一个该纪念品换回的金币数量。

每天，小伟可以进行以下两种交易无限次：

1.任选一个纪念品，若手上有足够金币，以当日价格购买该纪念品；

2.卖出持有的任意一个纪念品，以当日价格换回金币。

每天卖出纪念品换回的金币可以立即用于购买纪念品，当日购买的纪念品也可以当日卖出换回金币。当然，一直持有纪念品也是可以的。

问初始有 M 个金币，T 天时最多会有多少个。

Solution

由于每天可以无数次地买卖同一种纪念品，所以我们可以有一种贪心策略：我每天买入的物品，第二天早上一定全部卖出去。

因此只需要计算当天的钱，在第二天早上最多会变成多少。

即执行 T-1 次搜索，枚举所有可能的购买方案，选出赚钱数最多的一种。

这样就可以获得60的部分分。

代码

```
void dfs(int k, int s, int d) {
    if (k == n + 1) {
        ans = max(ans, res + s);
        return ;
    }
    if (p[d][k] < p[d + 1][k]) {
        for (int c = 1; c * p[d][k] <= s;
c++) {
            res += c * p[d + 1][k];
            dfs(k + 1, s - c * p[d][k], d);
            res -= c * p[d + 1][k];
        }
    }
    dfs(k + 1, s, d);
}
```

```
for (int i = 1; i < T; i++) {
    res = 0, ans = m;
    dfs(1, m, i);
    m = ans;
}
```

图的存储

邻接矩阵

e[i][j]=1/0 表示节点 i 与节点 j 之间是否有连边

动态数组存储每个节点的出边

```
vector<int> adj[N];
<u,v> : adj[u].push_back(v);
vector<pair<int, int> > adj[N]
<u,v,w> : adj[u].push_back({v, w});
```

DFS(图论)

DFS 全称是 Depth First Search，中文名是深度优先搜索，是一种用于遍历或搜索树或图的算法。所谓深度优先，就是说每次都尝试向更深的节点走。

算法大致框架：

```
void dfs(int u) {
    vis[u] = true;
    for (int v : adj[u])
        if (!vis[v])
            dfs(v);
}
```

P3958 奶酪

有一块高为 h，长度和宽度都为无穷大的奶酪，下表面高度 z 记作 0。在奶酪中有 n 个半径为 r 的球形空洞，给定这些空洞球心的坐标。问能否在不破坏奶酪的情况下，从下表面出发，经过空洞到达上表面。

对于所有数据，$n \leq 10^3$。

Solution

把每个空洞当成图上的一个点，如果两个空洞相交或相切，则在它们之间建一条双向边。

然后从每个与下表面相交或相切的点出发进行 DFS，看看是否能到达与上表面相交或相切的点。

代码

```
void dfs(int k){
    vis[k] = true;
    if(z[k] >= h) {
        flag = true;
        return;
    }
    for(int v : t[k])
        if(!vis[v])
            dfs(v);
}
```

```
for(int i = 1; i <= n; i++)
    if(z[i] <= r){
        dfs(i);
        if(flag) {
            puts("Yes");
            break;
        }
    }
if(!flag) puts("No");
```

BFS

BFS 全称是 Breadth First Search，中文名是宽度优先搜索，也叫广度优先搜索。所谓宽度优先。就是每次都尝试访问同一层的节点。 如果同一层都访问完了，再访问下一层。

可以在边权为 1 的图上求最短路。

算法大致框架：

```
queue<类型> q;
// 初始化所有点的dis为一个极大值
dis[s]=0; q.push(s); // 放入出发点，可以为多个
while (!q.empty()) {
    auto u = q.front(); q.pop();
    for (auto v : adj[u]) { // 枚举所有从u出发能直接到达的点v
        if (dis[v] > dis[u] + 1) {
            dis[v] = dis[u] + 1;
            q.push(v);
        }
    }
}
```

P10491 The Chivalrous Cow B

给定一张 $n \times m$ 的地图，K表示初始位置，*表示障碍，H表示目标位置。问从初始位置K，按照象棋中马的走法，不经过障碍的话最少需要几步到达目的地H。

数据保证有解。

$\mathrm{n} \leq 150, m \leq 150$。

Solution

问最少几步/在边权为 1 的图上求最短路常用 BFS。

先预处理出两个数组

```
const int dx[] = {-1, -2, -2, -1, 1, 2, 2, 1};
const int dy[] = {-2, -1, 1, 2, -2, -1, 1, 2};
```

表示每一步在横竖位置上移动的距离。

然后依次计算1/2/3/4/……分别能走到哪些位置。具体方法是使用一个队列queue<pair<int, int>> q存储走过的点，每次从队列头部取出一个点计算它能走到哪些点，走到还没走过的点就把它加进队列里，记录下走到这个点需要几步。

代码

```
while (!q.empty()) {
    int ux = q.front().first, uy = q.front().second; q.pop();
    for (int i = 0; i < 8; i++) {
        int nx = ux + dx[i], ny = uy + dy[i];
        if (1 <= nx && nx <= n && 1 <= ny && ny <= m && s[nx][ny] != '*'
&& dis[nx][ny] > dis[ux][uy] + 1) {
            dis[nx][ny] = dis[ux][uy] + 1;
            q.push({nx, ny});
        }
    }
}
cout << dis[tx][ty] << '\n';
```

P7775 VUK

给定一张 $n \times m$ 的地图，+ 表示树，V 是初始位置，J 是目标位置。求从初始位置到目标位置，路程中离树的距离的最小值最大是多少。

对于所有数据，$1 \leq n, m \leq 500$。

Solution

首先，预处理出每个点 (x,y) 到达最近的树的距离 dis[x][y]。

树有多棵 → 多源BFS

初始时把所有点的dis初始化为一个极大值，然后把树所在的点的dis修改为0，并放入初始的队列中。把它们同时作为起点出发跑BFS。

这样每个点只会被从离它最近的树出发的最短路更新一次。

代码

```
queue<pair<int, int>> q;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= m; j++) {
        cin >> s[i][j];
        if (s[i][j] == '+') {
            q.push({i, j});
            dis[i][j] = 0;
        }
        if (s[i][j] == 'V') sx = i, sy = j;
        if (s[i][j] == 'J') tx = i, ty = j;
    }
while (!q.empty()) {
    int ux = q.front().first, uy = q.front().second; q.pop();
    for (int i = 0; i < 4; i++) {
        int nx = ux + dx[i], ny = uy + dy[i];
        if (1 <= nx && nx <= n && 1 <= ny && ny <= m && dis[nx][ny] >
dis[ux][uy] + 1) {
            dis[nx][ny] = dis[ux][uy] + 1;
            q.push({nx, ny});
        }
    }
}
```

Solution

求距离最小值最大是多少 → 二分答案

选取 $[l,r]$ 的中点 mid，判断是否能不经过离树距离小于 mid 的点，然后从初始位置出发到达目标位置。DFS

```
void dfs(int x, int y, int d) {
    vis[x][y] = true;
    for (int i = 0; i < 4; i++) {
        int nx = x + dx[i], ny = y + dy[i];
        if (1 <= nx && nx <= n && 1 <= ny && ny <= m && !vis[nx][ny]
&& dis[nx][ny] >= d) dfs(nx, ny, d);
    }
}
bool check(int d) {
    memset(vis, 0, sizeof(vis));
    dfs(sx, sy, d);
    return vis[tx][ty];
}
```

二分部分代码

```
int L = 0, R = min(dis[sx][sy], dis[tx][ty]), ans = 0;
while (L <= R) {
    int mid = (L + R) / 2;
    if (check(mid)) ans = mid, L = mid + 1;
    else R = mid - 1;
}
```

树

- 有根树/无根树
- 内向树/外向树
- 完全二叉树/满二叉树
- 父亲/儿子/祖先/后代/子树

P5018 对称二叉树

一棵有点权的有根树如果满足以下条件，则被轩轩称为对称二叉树：

1.二叉树；

2.将这棵树所有节点的左右子树交换，新树和原树对应位置的结构相同且点权相等。

现在给出一棵二叉树，希望你找出它的一棵子树，该子树为对称二叉树，且节点数最多。请输出这棵子树的节点数。

$n \leq 10^6$。

Solution

枚举每个点，单独考虑以该点为根的子树，自上而下地尝试交换左右子树，看看左右子树是否对称。

以 i 为根的子树与以 j 为根的子树对称的条件是：

- $val[i] == val[j]$
- 以 $r[i]$ 为根的子树与以 $l[j]$ 为根的子树对称
- 以 $l[i]$ 为根的子树与以 $r[j]$ 为根的子树对称

若该子树是对称二叉树，则统计其节点数，用以更新答案。

在判定一个子树是否是对称二叉树时，最多只会遍历它左右子树中较小子树节点数级别的状态。这和启发式合并的过程类似，所以总时间复杂度为 $O(n \log n)$。

代码

```
bool dfs(int a, int b) {
    if (a == -1 && b == -1) return true;
    if (a == -1 || b == -1) return false;
    if (v[a] != v[b]) return false;
    if (!dfs(r[a], l[b])) return false;
    if (!dfs(l[a], r[b])) return false;
    return true;
}
void dfs1(int k) {
    if (k != -1) ans++;
    else return ;
    dfs1(l[k]);
    dfs1(r[k]);
}
```

```
for (int i = 1; i <= N; i++)
    if (dfs(i, i)) {
        ans = 0;
        dfs1(i);
        Max = max(Max, ans);
    }
```

P5658 括号树

给定一棵 n 个节点的树，树上每个节点有一个 (或) ，把从根到树上每个点 i 的路径上每个节点的括号取出来，就会串成一个括号串 s_i。

请你对于每个 i，计算 s_i 有多少个子串是合法的括号序列。

树上前缀和

普通一维前缀和：$sum[i] = a[1] + a[2] + \cdots + a[i]$

$O(n)$ 求法：依次对于 $i = 1,2,3,\cdots,n$，执行 $sum[i] = sum[i-1] + a[i]$

树上前缀和：$sum[v] = \sum a[u]$(u 为 v 的祖先，包括 v 本身)

$O(n)$ 求法：自树根开始自上而下地进行 DFS，一遍历到某个节点 u，就执行 $sum[u] = sum[fa[u]] + a[u]$

Solution

我们先计算 s_i 有多少个后缀为合法的括号序列，记该数量为 f_i，然后再做树上前缀和，就得到了 s_i 子串中合法括号序列的数量。

要计算 s_i 有多少个后缀为合法的括号序列，可以分类讨论：

- 如果节点 i 上是左括号，则数量为 0。
- 如果节点 i 上是右括号，则找到根到节点i 路径上与其匹配的最近的左括号，假设这个括号在节点 j 上，则 $f_i = f_{fa_j} + 1$。

而维护这个最近的左括号，则需要用到我们昨天讲过的栈。从上往下计算时，遇到左括号则放入栈中，遇到右括号则弹出与之匹配的左括号。从下往上回溯时，遇到左括号则出栈，遇到右括号则把与之匹配的左括号放回栈中。

代码

```
void dfs(int u, int dep) {
    pos[dep] = u;
    int Last = -1;
    if (s[u] == ')' && tot > 0 && s[pos[Stack[tot]]] == '(')
        p[u] = Stack[tot], Last = Stack[tot--] - 1;
    else Stack[++tot] = dep;

    if (s[u] == ')' && Last != -1) ans[u] = ans[pos[Last]] + 1;
    for (int v : adj[u]) dfs(v, dep + 1);
    if (!p[u]) tot--;
    else Stack[++tot] = p[u], p[u] = 0;
}
void calc(int u) {
    res ^= 1ll * u * ans[u];
    for (int v : adj[u]) ans[v] += ans[u], calc(v);
}
```

P7073 表达式

给定一个逻辑表达式（运算符只包含 &、|、！），并给出其中每个变量的 0/1 取值。

q 次询问，每次询问把某个变量取反后整个表达式的结果是多少（询问之间独立，即询问完后修改会被撤销）。

逻辑表达式以后缀表达式形式给出。

后缀表达式

后缀表达式的定义如下：

1.如果 E 是一个操作数，则 E 的后缀表达式是它本身。

2.如果 E 是 $E1\ op\ E2$ 形式的表达式，其中 op 是任何二元操作符，且优先级不高于 E1 、E2 中括号外的操作符，则 E 的后缀式为 E1′E2′op，其中 E1′ 、E2′ 分别为 E1、E2 的后缀式。

3.如果 E 是 E1 形式的表达式，则 E1 的后缀式就是 E 的后缀式。

Solution

根据后缀表达式建立表达式树，只考虑对二元运算符（&、|）和初始变量建立节点，遇到！符号我们直接在节点上打一个取反标记。

根据后缀表达式建树相对简单。设置一个节点栈，遇到变量直接放入节点栈；遇到二元运算符则新建一个节点，并从节点栈中取出两个节点，分别设置为它的左右儿子，更新新节点的运算结果，并将其存入节点栈中。

代码

```
for (int i = 0; i < len; i++) {
    if ('0' <= s[i] && s[i] <= '9') id = id * 10 + s[i] - '0';
    else {
        if (id) stk[++tp] = id, id = 0;
        if (s[i] == '&' || s[i] == '|') {
            c[++nodeid] = s[i];
            int x = stk[tp--], y = stk[tp--];
            if (a[nodeid] == '&')
                a[nodeid] = ((a[x] ^ rev[x]) & (a[y] ^ rev[y]));
            else a[nodeid] = ((a[x] ^ rev[x]) | (a[y] ^ rev[y]));
            ls[nodeid] = x, rs[nodeid] = y, stk[++tp] = nodeid;
        }
        if (s[i] == '!') rev[stk[tp]] ^= 1;
    }
}
```

Solution

建立出表达式树后，我们对这棵树自上而下的进行 DFS。

如果当前节点符号为 `&`，并且它的其中一棵子树最后的运算结果为`0`，那么它另一棵子树所有变量的取值无论如何变，都不会改变整体的运算结果。

如果当前节点符号为 `|`，并且它的其中一棵子树最后的运算结果为`1`，那么它另一棵子树所有变量的取值无论如何变，都不会改变整体的运算结果。

我们可以预处理出哪些变量取反后，整个表达式的运算结果也会取反，哪些取反后表达式结果不变。

代码

```
void dfs(int u) {
    if (c[u] == '&') {
        if ((a[rs[u]] ^ rev[rs[u]]) == 1) {
            if (ls[u] <= n) cge[ls[u]] = true;
            else dfs(ls[u]);
        }
        if ((a[ls[u]] ^ rev[ls[u]]) == 1) {
            if (rs[u] <= n) cge[rs[u]] = true;
            else dfs(rs[u]);
        }
    }
    else {
        if ((a[rs[u]] ^ rev[rs[u]]) == 0) {
            if (ls[u] <= n) cge[ls[u]] = true;
            else dfs(ls[u]);
        }
        if ((a[ls[u]] ^ rev[ls[u]]) == 0) {
            if (rs[u] <= n) cge[rs[u]] = true;
            else dfs(rs[u]);
        }
    }
}
```

P8815 逻辑表达式

给出一个含有 0,1,(,),&,| 的逻辑表达式，计算该逻辑表达式的值，并且计算 a&b 和 a|b 的"短路"各出现了多少次。（只记录最外层的短路，若外层已经短路，则内层的短路不计。例如 1 |（0&1），外层已经短路，则里面的0&1不记录）

短路:在计算 a&b 时，a 的值为0;

在计算 a|b 时，b 的值为0。

Solution

可以先使用栈，依照中缀表达式建出表达式树。

设置两个栈，一个节点栈、一个运算符栈。从左到右遍历中缀表达式。如果遇到数字，则创建一个节点并将其压入节点栈。如果遇到运算符，根据运算符的优先级进行处理。

Solution

如果当前运算符的优先级高于栈顶运算符，将当前运算符压入运算符栈。

如果当前运算符的优先级低于或等于栈顶运算符，从运算符栈中弹出所有优先级高于或等于当前运算符的元素，并同时弹出相应的两个节点连起来组成一棵子树。然后再将运算符入栈。

如果遇到左括号，将其压入运算符栈。

如果遇到右括号，从运算符栈中弹出元素（同时弹出相应节点连接）直到遇到左括号，并将这些元素连接起来形成一个子树。

代码

```
for (int i = 0; i < s.size(); i++) {
    if (s[i] == '0' || s[i] == '1') {
        a[++nodeid] = s[i] - '0'; id.push(nodeid);
    }
    else {
        if (s[i] == '(') op.push(s[i]);
        else if (s[i] == ')') {
            while (op.top() != '(') {
                c[++nodeid] = op.top(); op.pop();
                int r = id.top(); id.pop();
                int l = id.top(); id.pop();
                ls[nodeid] = l, rs[nodeid] = r;
                id.push(nodeid);
            }
            op.pop();
        }
        else if (s[i] == '&') {
            while (op.size() > 0 && op.top() == '&') {
                c[++nodeid] = op.top(); op.pop();
                int r = id.top(); id.pop();
                int l = id.top(); id.pop();
                ls[nodeid] = l, rs[nodeid] = r;
                id.push(nodeid);
            }
            op.push(s[i]);
        }
        else {
            while (op.size() > 0 && op.top() != '(') {
                c[++nodeid] = op.top(); op.pop();
                int r = id.top(); id.pop();
                int l = id.top(); id.pop();
                ls[nodeid] = l, rs[nodeid] = r;
                id.push(nodeid);
            }
            op.push(s[i]);
        }
    }
}
while (op.size() > 0) {
    c[++nodeid] = op.top(); op.pop();
    int r = id.top(); id.pop();
    int l = id.top(); id.pop();
    ls[nodeid] = l, rs[nodeid] = r;
    id.push(nodeid);
}
```

Solution

然后遍历表达式树，遇到叶子节点则直接返回值，否则先遍历左子树再遍历右子树，根据左子树的结果和当前节点的符号判断是否发生了“短路”，并计算结果返回。

代码

```
int dfs(int u) {
    if (c[u] == '&') {
        int lans = dfs(ls[u]);
        if (lans == 0) {
            ans1++;
            return 0;
        }
        else return lans & dfs(rs[u]);
    }
    else if (c[u] == '|') {
        int lans = dfs(ls[u]);
        if (lans == 1) {
            ans2++;
            return 1;
        }
        else return lans | dfs(rs[u]);
    }
    else return a[u];
}
```