深度优先搜索

dbywsc

2025/7

目录

定义

深度优先搜索 (Depth First Search) 简称 **DFS** ，原本是图论中的概念，但是我们经常利用其思想解决一些别的问题，在此处指的是一种搜索算法。

DFS 的本质是**递归** + **回溯** ，即通过函数自己调用自己的方式实现暴力搜索的目的。它的基本逻辑是把能走的路先走到底，如果走不下去了就回过头走别的路，常被用来解决一些与“选择状态”，“选择方案”有关的问题。

全排列问题 – I

P1706 全排列问题

要求我们输出 1 n 的全排列，由于 n 最大为 9 ，因此通过循环的方式最多需要进行 9^9 次运算，无法通过本题。

可以定义函数 $dfs(int\ step)$ ，其中参数 $step$ 表示当前在处理第几个数字；定义数组 $st[1..n]$ 和数组 $num[1..n]$ 分别存储当前数字在本次排列中有没有被标记和存储当前排列。那么在函数中，我们可以做一次 1 n 的遍历，如果 st_i 没有被标记过，说明它没有在本排列中出现过，因此标记 $st_i = true$，我们认为它是本轮 dfs 中要处理的数字，因此把它存储到当前排列中：$num_{step} = i$。之后进行下一步的搜索 $dfs(step + 1)$ ，之后我们要还原现场，进行下一轮的判断：$st_i = false$ 。

当某次 dfs 把 1 n 都搜完了，也就是 $step > n$ 时，我们就得到了一个排列，将它输出，并且退出这一次的 dfs ，让它不再往下搜。

全排列问题 – II

```
void dfs(int step) {
    if(step > n) {
        for(int i = 1; i <= n; i++) std::cout << "    " << num[i];
        std::cout << std::endl;
        return;
    }
    for(int i = 1; i <= n; i++)
        if(!st[i]) {
            st[i] = true; num[step] = i;
            dfs(step + 1);
            st[i] = false;
        }
}
void solve(void) {
    std::cin >> n;
    dfs(1);
}
```

全排列问题 – III

上述代码的时间复杂度是 $O(n!)$，在本题的规模之内远小于 9^9。
通过上面的代码，我们能够总结出 *DFS* 所需的几个条件：
1. 初始状态：搜索时我们需要从一个最初始的状态出发。
2. 递归：满足条件时，我们要通过递归的方式搜索下一层状态。
3. 回溯：递归后要清理现场，保证不影响到下次搜索。
4. 退出条件：当任务完成后，我们要及时的退出，避免永无止境[1]的搜索。

[1]事实上，函数递归时会占用程序的**栈空间**，当栈空间被占满后仍然没有退出递归时就会引发**段错误**(Segmentation fault)

DFS 的优化 – I

P10483 小猫爬山

显然，这道题要求我们尽可能的往一辆车里塞更多的猫，但是这道题的规模是 $n \leq 18$ ，如果再像全排列一样每次都做 n 次枚举是无法通过本题的，因此可以先做一个优化：对所有小猫的重量排一个序，拍完序后如果第 i 只小猫放不到车里面，那么在它之后的所有小猫也放不到车里面，所以就可以直接退出了。[2]

此外，本题要求我们求最小的话费，因此在某一次搜索中，如果当前方案的话费已经超过了目前保存下来的其他方案的最小花费，那么答案一定不会在这个方案上，我们也没有必要继续搜索下去了，可以直接退出。这样的优化就叫做**剪枝**。

[2]这样的思路我们就叫做 **贪心**

DFS 的优化 – II

```
void dfs(int step, int tot) {
    if(tot >= ans) return;  //剪枝
    if(step > n) {
        ans = tot;  //由于 tot >= ans 的情况已经被剪了，
        return;     //所以此时的 tot 是必定小于 ans 的
    }
    for(int i = 0; i < tot; i++)
        if(g[i] + c[step] <= w) {
            g[i] += c[step];
            dfs(step + 1, tot);
            g[i] -= c[step];
        }
    g[tot] = c[step];
    dfs(step + 1, tot + 1);
    g[tot] = 0;
}
void solve(void) {
    std::cin >> n >> w;  for(int i = 1; i <= n; i++) std::cin >> c[i];
    std::sort(c + 1, c + 1 + n); dfs(1, 0);
    std::cout << ans;
}
```

DFS 解决迷宫问题 – I

搜索算法更直观的一种用法是解决走迷宫的问题[3]
P1605 迷宫
很显然，我们可以定义函数 $dfs(int\ x,\ int\ y)$ ，最开始的状态显然就是 $dfs(int\ sx,\ int\ sy)$ ，之后我们可以往上下左右四个方向走，如果走到了当前的位置是障碍物，或者当前的位置已经超出了 $n \times m$ 的迷宫范围就退出。每次如果走到了 (fx, fy) 就说明有了一种方案到达终点，令 $ans + 1$ 。
对于迷宫的存储，我们可以使用二维数组。

[3]虽然 DFS 可以解决这种问题，但是受制于栈内存的限制，大家其实不太喜欢用 DFS 写这种问题，这里列出来是为了让大家更好的理解算法。

DFS 解决迷宫问题 – II

```
void dfs(int x, int y) {
    if(x < 1 || x > n || y < 1 || y > m || G[x][y]) return;
    if(x == fx && y == fy) {
        ans++;
        return;
    }
    G[x][y] = true;
    dfs(x + 1, y); dfs(x - 1, y); dfs(x, y + 1); dfs(x, y - 1);
    G[x][y] = false;
}
void solve(void) {
    std::cin >> n >> m >> t;
    std::cin >> sx >> sy >> fx >> fy;
    while(t--) {
        int x, y; std::cin >> x >> y;
        G[x][y] = true;
    }
    dfs(sx, sy); std::cout << ans;
}
```

dfs 是搜索算法的基本功，同时在图论算法中占据着重要的地位。有非常多的算法都是以 *dfs* 为原型发明出来的。
由于递归的概念过于抽象，因此大家在一开始学习时一定会难以理解。想要学会 *dfs* 的最好方式就是勤加练习，并且仔细的思考解决问题的每一个步骤，随着题量的增加，之前的一些困惑自然也就迎刃而解了。
P2089 烤鸡
P2404 自然数的拆分问题
P1036 [NOIP 2002 普及组] 选数
B3622 枚举子集（递归实现指数型枚举）
P2036 [COCI 2008/2009 #2] PERKET
P1596 [USACO10OCT] Lake Counting S
P1219 [USACO1.5] 八皇后 Checker Challenge