深度优先搜索

简介

深度优先搜索算法（DFS），是图算法的一种，其过程简要来说是对每一个可能的分支路径深入到不能再深入为止，而且每个节点只能访问一次。

连通图的深度优先遍历类似于树的先根遍历

DFS算法描述

1. 从图中v_0出发，访问v_0。
2. 找出v_0的第一个未被访问的邻接点，访问该顶点。以该顶点为新顶点，重复此步骤，直至刚访问过的顶点没有未被访问的邻接点为止。
3. 返回前一个访问过的仍有未被访问邻接点的顶点，继续访问该顶点的下一个未被访问领接点。
4. 重复2,3步骤，直至所有顶点均被访问，搜索结束。

从这四点可以看出**深度优先搜索是一个递归过程**

DFS空间复杂度

若有v个顶点、E条边：

- 用邻接表储存图，有O(V+E)
- 用邻接矩阵储存图，有O(V^2)

深度优先搜索的优化技巧

1. **优化搜索顺序**：在一些搜索问题中，搜索树的各个层次，各个分支之间的顺序是不固定的。不同的搜索顺序会产生不同的搜索树形态，其规模大小也相差甚远。
2. **排除等效冗余**：在搜索过程中，如果我们能够判定从搜索树的当前节点上沿着某几条不同分支到达的子树是等效的，那么只需要对其中的一条分支执行搜索。
3. **可行性剪枝（上下界剪枝）**：该方法判断继续搜索能否得出答案，如果不能直接回溯。　　在搜索过程中，即使对当前状态进行检查，如果发现分支已经无

法到达递归边界，就执行回溯。
4. **最优性剪枝**：最优性剪枝，是一种重要的搜索剪枝策略。它记录当前得到的最优值，如果当前结点已经无法产生比当前最优解更优的解时，可以提前回溯。
5. **记忆化**：可以记录每个状态的搜索结果，再重复遍历一个状态时直接检索并返回。这好比我们对图进行深度优先遍历时，标记一个节点是否已经被访问过。

DFS核心代码

储存图的方式：

- 邻接表
- 邻接矩阵

实现遍历过程的方式：

- 通过栈维护
- 递归

邻接矩阵储存图，通过递归遍历

```
struct G {
    int arr[MAX_N][MAX_M];
    int num;
} G;

int vis[MAX_N] = {0};

void DFS(int v) {
    vis[v] = 1; // 访问过
    for (int i = 0; i < G.num; i++) {
        if (G.arr[v][i] && vis[i] == 0) { //递归调用
            DFS(i);
        }
    }
    return ;
}

```

邻接矩阵储存图，通过非递归遍历

```cpp
struct G {
    int arr[MAX_N][MAX_M];
    int num;
} G;

int vis[MAX_N] = {0};

void DFS(int v){
    stack<int> s;
    s.push(v);
    vis[v] = 1; // 访问过

    while (!s.empty()) {
        int i = s.top(), j;
        for (j = i; j < G,num; j++) {
            if(G.arr[i][j] && G.vis[j] == 0) { // 找到i的邻接点
                G.vis[j] = 1;
                s.push(j); // 继续找j的邻接点
                break;
            }
        }
        if(j == G.num){ // 没找到i的邻接点
            s.pop();
        }
    }
    return ;
}
```

广度优先搜索

简介

广度优先算法（Breadth-First Search），简称BFS，是一种图形搜索演算法。简单的说，BFS是从根节点开始，沿着树的宽度遍历树的节点，如果发现目标，则演算终止。

BFS算法描述

1. 访问顶点 v_0

2. 访问 v_0 的所有未被访问的邻接点v_1, v_2 ... v_n

3. 依次从这些邻接点（在步骤 2 中访问的顶点）出发，访问它们的所有未被访问的邻接点，依此类推，直到图中所有访问过的顶点的邻接点都被访问

可以看出BFS是一个层序遍历的过程，一层一层的向下遍历。需要一个辅助队列来维护这种遍历方式。

空间复杂度

因为所有节点都必须被储存

1. O(V + E)，V 是节点的数目，而 E 是图中边的数目。
2. O(B ^ M)，B 是最大分支系数，而 M 是树的最长路径长度。

由于对空间的大量需求，因此BFS并不适合解非常大的问题。

记忆化搜索

因为搜索不能够很好地处理重叠子问题，动态规划虽然比较好地处理了重叠子问题，但是在有些拓扑关系比较复杂的题目面前，又显得无奈。记忆化搜索便将两种方法结合到一起，能更好的去处理问题。

记忆化搜索 = 搜索的形式 + 动态规划的思想

动态规划与记忆化搜索

- 动态规划就是一个最优化问题，先将问题分解为子问题，并且对于这些分解的子问题自身就是最优的才能在这个基础上得出我们要解决的问题的最优方案。动态规划不同于贪心算法，因为贪心算法是从局部最优来解决问题，而动态规划是全局最优的。
- 动态规划的一种变形就是记忆化搜索，就是根据动归方程写出递归式，然后在函数的开头直接返回以前计算过的结果，当然这样做也需要一个存储结构记下前面计算过的结果，所以又称为记忆化搜索。
- 用一个数组或者其他的存储结构存储得到的子问题的解。这样就可以省很多时间，也就是典型的空间换时间

记忆化搜索递归式动态规划

记忆化搜索的思想

在搜索过程中，会有很多重复计算,如果我们能记录一些状态的答案，就可以减少重复搜索量。

记忆化搜索的适用范围

根据记忆化搜索的思想，它是解决重复计算，而不是重复生成，也就是说，这些搜索必须是在搜索扩展路径的过程中分步计算的题目，也就是“搜索答案与路径相关”的题目，而不能是搜索一个路径之后才能进行计算的题目，必须要分步计算，并且搜索过程中，一个搜索结果必须可以建立在同类型问题的结果上，也就是类似于动态规划解决的问题。
其问题表达，不是单纯生成一个走步方案，而是生成一个走步方案的代价等，而且每走一步，在搜索树/图中生成一个新状态，都可以精确计算出到此为止的费用，也就是，可以分步计算，这样才可以套用已经得到的答案 。

记忆化搜索的核心实现

1. 首先，要通过一个表记录已经存储下的搜索结果，一般用哈希表实现
2. 状态表示，由于是要用哈希表实现，所以状态最好可以用数字表示，常用的方法是把一个状态连写成一个p进制数字，然后把这个数字对应的十进制数字作为状态
3. 在每一状态搜索的开始，高效的使用哈希表搜索这个状态是否出现过，如果已经做过，直接调用答案，回溯
4. 如果没有，则按正常方法搜索

记忆化搜索是类似于动态规划的，不同的是，它是倒做的“递归式动态规划”

斐波那契数列（记忆化版）

```
int f[MAX_N] = {0};

int func(int n) {
    if (f[n] != 0) return f[n];
    if (n <= 2) return (f[n] = 1);
    else return (f[n] = func(n - 2) + func(n - 1));
}
```

剪枝

简介

在搜索算法中优化中，剪枝，就是通过某种判断，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”，故称剪枝。应用剪枝优化的核心问题是设计剪枝判断方法，即确定哪些枝条应当舍弃，哪些枝条应当保留的方法。

剪枝优化三原则

搜索算法，绝大部分需要用到剪枝。然而，不是所有的枝条都可以剪掉，这就需要通过设计出合理的判断方法，以决定某一分支的取舍。在设计判断方法的时候,需要遵循一定的原则。

正确性

枝条不是爱剪就能剪的。如果随便剪枝，把带有最优解的那一分支也剪掉了的话，剪枝也就失去了意义。所以，剪枝的前提是一定要保证不丢失正确的结果。

准确性

在保证了正确性的基础上，我们应该根据具体问题具体分析，采用合适的判断手段，使不包含最优解的枝条尽可能多的被剪去，以达到程序“最优化”的目的。可以说，剪枝的准确性，是衡量一个优化算法好坏的标准。

高效性

设计优化程序的根本目的，是要减少搜索的次数，使程序运行的时间减少。但为了使搜索次数尽可能的减少，我们又必须花工夫设计出一个准确性较高的优化算法，而当算法的准确性升高，其判断的次数必定增多，从而又导致耗时的增多，这便引出了矛盾。因此，如何在优化与效率之间寻找一个平衡点，使得程序的时间复杂度尽可能降低,同样是非常重要的。倘若一个剪枝的判断效果非常好，但是它却需要耗费大量的时间来判断、比较，结果整个程序运行起来也跟没有优化过的没什么区别，这样就太得不偿失了。

分类

可行性剪枝

该方法判断继续搜索能否得出答案，如果不能直接回溯。

最优性剪枝

最优性剪枝，又称为上下界剪枝，是一种重要的搜索剪枝策略。它记录当前得到的最优值，如果当前结点已经无法产生比当前最优解更优的解时，可以提前回溯。

剪枝策略的寻找的方法

- 微观方法：从问题本身出发，发现剪枝条件
- 宏观方法：从整体出发，发现剪枝条件。
- 注意提高效率，这是关键，最重要的。

总之，剪枝策略，属于算法优化范畴；通常应用在DFS 和 BFS 搜索算法中；剪枝策略就是寻找过滤条件，提前减少不必要的搜索路径。

练习题

P1605 迷宫

代码

```
#include <stdio.h>

int map[6][6] = {0};
int vis[6][6] = {0};
int dir[4][2] = {-1, 0, 1, 0, 0, 1, 0, -1};

int n, m, ans = 0;
int startx, starty, endx, endy;

void dfs(int x, int y) {
    if (x == endx && y == endy) {
        ans++;
        return ;
    }
    for (int i = 0; i < 4; i++) {
        int now_x = x + dir[i][0], now_y = y + dir[i][1];
        if (now_x > m || now_x < 1 || now_y > n || now_y < 1)
continue;
        if (vis[now_y][now_x]) continue;
        vis[now_y][now_x] = 1;
        dfs(now_x, now_y);
        vis[now_y][now_x] = 0;
    }
    return ;
}

```

```cpp
int main() {
    int t;
    scanf("%d %d %d", &n, &m, &t);
    scanf("%d %d %d %d", &startx, &starty, &endx, &endy);
    vis[starty][startx] = 1;
    for (int i = 0; i < t; i++) {
        int x, y;
        scanf("%d %d", &x, &y);
        vis[y][x] = 1;
    }
    dfs(startx, starty);
    printf("%d\n", ans);
    return 0;
}
```

P1443 马的遍历

所用知识

广度优先搜索

代码

```cpp
#include<iostream>
#include<cstdio>
#include<queue>
using namespace std;

struct Node {
    int x, y;
} node, temp;

int dir[8][2] = {
    {1, 2}, {2, 1}, {-1, 2}, {-2, 1},
    {-1, -2}, {-2, -1}, {1, -2}, {2, -1}
}; // 方向数组
int arr[505][505] = {0}; // 用作标记和记录步数
int n, m, sx, sy;

void init() {
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            arr[i][j] = -1;
```

```
        }
    }
    arr[sx][sy] = 0;
    return ;
}

void bfs() {
    queue<Node> q;
    node.x = sx;
    node.y = sy;
    q.push(node);
    while (!q.empty()) {
        temp = q.front();
        q.pop();
        for (int i = 0; i < 8; i++) {
            int nx = temp.x + dir[i][0], ny = temp.y + dir[i][1];
            if (nx < 1 || nx > n || ny < 1 || ny > m) continue; //判断越界
            if (arr[nx][ny] == -1) { // 判断是否走过
                node.x = nx;
                node.y = ny;
                q.push(node);
                arr[nx][ny] = arr[temp.x][temp.y] + 1; // 步数加 1
            }
        }
    }
    return ;
}

int main() {
    scanf("%d%d%d%d", &n, &m, &sx, &sy);
    init();
    bfs();
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= m; j++) {
            printf("%-5d", arr[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

P1219 八皇后

解题思路

- 开三个标记数组分别标记**列**和每个点所在位置的**两条对角线的平行线**
- 开一个int型数组存满足条件的结果，在递归过程中输出前三组即可（即是字典序）

判断在一条对角线的平行线的方法：

- 到左边界的距离+到下边界的距离的和相等的点在同一条主对角线的平行线上
- 到左边界的距离+到上边界的距离的和相等的点在同一条副对角线的平行线上

代码

```
#include <iostream>

using namespace std;

const int MAX_N = 15;

int ans = 0, n;
int vis_column[MAX_N] = {0}; // 标记列
int vis_MDL[MAX_N << 1] = {0}; // 标记主对角线平行线
int vis_ADL[MAX_N << 1] = {0}; // 标记副对角线平行线
int str[MAX_N] = {0}; // 存满足条件的解

// 输出解
void output() {
    for (int i = 0; i < n; i++) {
        if (i != 0) cout << " ";
        cout << str[i] + 1;
    }
    cout << endl;
    return ;
}

void dfs(int cnt) {
    if (cnt == n) { // 遍历到n行结束
        ans++;
        if (ans < 4) output();
        return ;
    }
    for (int i = 0; i < n; i++) {
```

```
        if (vis_column[i] || vis_MDL[cnt + i] || vis_ADL[n - 1 - cnt + i]) continue; // 判断列、主副对角线平行线是否满足
        str[cnt] = i;
        vis_column[i] = vis_MDL[cnt + i] = vis_ADL[n - 1 - cnt + i] = 1; // 标记
        dfs(cnt + 1);
        vis_column[i] = vis_MDL[cnt + i] = vis_ADL[n - 1 - cnt + i] = 0; // 取消标记
    }
    return ;
}

int main() {
    cin >> n;
    dfs(0);
    cout << ans << endl;
    return 0;
}
```

P1464 Function

所用知识

深搜、记忆化

思路解析

- 如果a，b，c中有一个值小于等于0，那么w(a, b, c)的值为1
- 如果a，b，c中有一个值大于20，那么w(a, b, c)的值为w(20, 20, 20)
- 如果a<b<c，那么w(a, b, c)=w(a, b, c-1) + w(a, b-1, c-1) - w(a, b-1, c)
- 否则w(a, b, c)=w(a-1, b, c) + w(a-1, b-1, c) + w(a-1, b, c-1) - w(a-1, b-1, c-1)

将每次函数的返回值存起来即可，如果之前存过直接返回

代码

```
#include <iostream>
#include <cstdio>
#include <cstring>

using namespace std;

int dp[25][25][25] = {0};
```

```
8
9  int w(int a, int b, int c) {
10     if (a <= 0 || b <= 0 || c <= 0) return 1;
11     if (a > 20 || b > 20 || c > 20) return w(20, 20, 20);
12     if (dp[a][b][c]) return dp[a][b][c];
13     if (a < b && b < c) {
14         dp[a][b][c] = w(a, b, c - 1) + w(a, b - 1, c - 1) -
   w(a, b - 1,c);
15         return dp[a][b][c];
16     }
17     dp[a][b][c] = (w(a - 1, b, c) + w(a - 1, b - 1, c) + w(a
   - 1, b, c - 1) - w(a - 1, b - 1, c - 1));
18     return dp[a][b][c];
19 }
20
21 int main(){
22     int a, b, c;
23     memset(dp, 0, sizeof(dp));
24     while(scanf("%d%d%d", &a, &b, &c) != EOF) {
25         if (a == -1 && b == -1 && c == -1) break;
26         printf("w(%d, %d, %d) = %d\n", a, b, c, w(a, b, c));
27     }
28     return 0;
29 }
```

P1433 吃奶酪

所用知识

深度优先搜索、剪枝

思路解析

1. 以（0，0）为起点
2. 打表，将每一点到其他点的距离先计算出来存到num数组中，节约时间
3. 采用最优性剪枝，如果当前路径长度比当前最短的路径要长时，回溯

代码

```
1 #include <iostream>
2 #include <cmath>
3 #include <cstdio>
4 #include <cfloat>
```

```cpp

using namespace std;

int n, vis[1001] = {0};
double x[105] = {0}, y[105] = {0}, num[1005][1005] = {0}; // num数组存两点间距离
double ans = DBL_MAX; // double型最大值 头文件<cfloat>

void dfs(int k, int ind, double sum) {
    if (sum > ans) return; // 当前路径长度比当前最短的路径长，回溯

    if (k == n) {
        // 走完所有点，更新最短路径
        ans = min(ans, sum);
        return ;
    }
    for (int i = 1; i <= n; i++) {
        if(!vis[i]) {
            vis[i] = 1; // 标记为走过
            dfs(k + 1, i, sum + num[ind][i]);
            vis[i] = 0; // 回溯到此处时，取消标记
        }
    }
}

// 打表，获得任意两点间距离
void get_num() {
    for(int i = 0; i <= n; i++) {
        for(int j = 0; j <= n; j++) {
            num[i][j] = sqrt((x[i] - x[j]) * (x[i] - x[j]) + (y[i] - y[j]) * (y[i] - y[j]));
        }
    }
    return ;
}

int main() {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> x[i] >> y[i];
    }
    get_num();
    dfs(0, 0, 0.0);
    printf("%.2lf\n",ans);
    return 0;
```

```
47  }
```