第十九章 深度优先搜索

1. 搜索算法

- 为什么需要搜索算法?
  现在人们开车去旅行时，总会在出发前使用导航软件，导航软会推荐合适的路线， 这背后有一套导找路线的方法，就是搜索算法。 搜索是计算机程序设计中一种最基本、最常用的算法。如果一个问题，无法找到明显的数学规律使用类似递推、贪心、 动态规划等方法求解，搜索便是经常使用的方法。 搜索算法是直接基于计算机高速运算的特点而使用的求解方法。 它的基本思路是：从问题的初始 状态出发，按照一定的策略，有序推进，不断深入，一一验证， 最终搜索整个解空间，从而我到最终解。 按照推进的策略，搜索一般分为深度优先搜索（DFS，Depth-First Search）和广度优先搜索 （BFS，Breadth-First Search）

https://www.bilibili.com/video/BV1yG4y1v7Ux/?spm_id_from=333.337.search-card.all.click&vd_source=24a2ef9dd2cbe3420c470e405b5ad871

- DFS和BFS的区别 DFS和BFS 是两种常用的搜索算法。它们在搜索时有一些不同之处

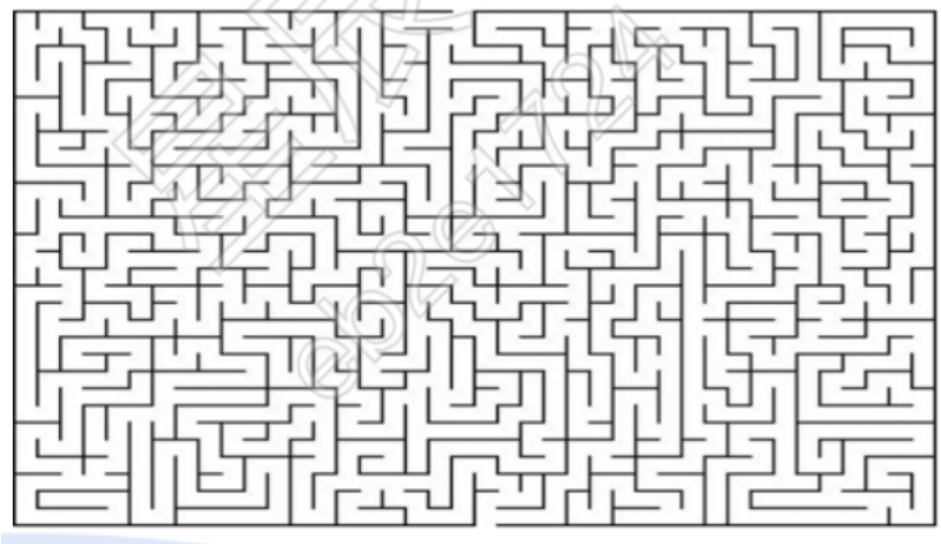
1. 遍历方式
   DFS：从起始节点开始，枚举所有可能的路径，如果某条路径可走，就一直沿着该路径向下遍历。直到终点或者无路可走，然后回到上一个节点，继续遍历其他路径。 BFS：从起始节点开始，先遍历所有与起始节点直接相连的节点，然后遍历与这些节点相连的节点，以此类推，直到找到目标节点或遍历完整个解空间。
2. 遍历顺序
   DFS：在搜索过程中，DFS会沿着某个可走的路径一直向下搜索， 直到终点或无路可走，然后再回到上一个节点继续遍历其他路径
   BFS：在搜索过程中，BFS会同时遍历所有可走的路径，路径之间不存在先后顺序。
   一般，DFS 适用于查找路径 （方案）总数的问题，而BFS 适用于查找最优路径 （方案） 的问题。
   选择使用哪种算法取决于具体的问题和应用场景。

2. 深度优先搜索算法

2.1 DFS的应用场景

深度优先搜索算法在很多领域，包括图遍历、 路径搜索、状态空间搜索等问题中都有广泛的应用它的简单性和易实现性使得它成为解决许多问题的有力工具
因发明“深度优先搜索算法”，约翰·霍普克洛夫特与罗伯特·塔扬在1986年共同获得计算机领域的最高奖：图灵奖

- 问题引入 我们一般如何走迷宫?
  进入迷营后，如果遇到分路口，选择一个方向前进，直到无路可走，这时返回上一个分路口选择其它还没走过的方向前进.此类推，直到找到出口



- 示例

  有一个 5 行 4 列的方格迷宫，0 表示可以通过，1 表示不可以通过，每一步可以向上、下、左、右任意方向移动一步，请计算从左上角（1,1）位置移动到右下角（5,4）位置，最少移动多少步。

实现过程就是从起点开始沿着一条路径一直搜索下去，直到终点或者无路可走， 回溯到上个节点继续搜索其他可以路径。 依次类推，直到所有路径均被搜索一遍结束
示例中共有2条不同的路径可以到达终点，最短路径为7。 算法分析：

1. 从起点开始，每到一个格子，往上、下、左、右4个方向进行探索，如果某个方向可以走就向该方向走一步，然后继续做相同的探索
2. 如果当前点的4个方向都已经探索完成，则回到上一个格子继续探索其他的路径
3. 如果到达了终点， 比较并更新最短路径。 算法实现：

vis[i][j] 用来标记点（i，j）是否在当前路径上。
a[i][j] 表示点（i，j）是否可通过，0 表示可以通过，1 表示不可以通过。
在迷宫中探索的四个方向，可以使用两个偏移量数组 dx 和 dy 来设置。

```
#include<iostream>
using namespace std;
int dx[4] = {1, 0, -1, 0};          //  四个方向的行偏移量
int dy[4] = {0, -1, 0, 1};          //  四个方向的列偏移量
int n, m, ans, a[15][15];
bool vis[15][15];                   // vis  数组记录小格子的访问状态
```

	−1, 0	
0, −1	x, y	0, +1
	+1, 0	

```
void dfs(int x , int y ,int step)  // step 表示到达终点的步数
{
    if(x == n && y ==m) //如果到达右下角
    {
        ans = min(ans, step); //更新最短路径
        return;
    }
    int tx, ty;
    for(int x =0;i<4;++i)  //探索四个方向
    {
        tx = x + dx[i];
        ty = y + dy[i];
        if( tx >=1 && tx <=n && ty >=1 && ty <= m)  //如果探索的小个子在迷宫阵内
        {
            if(!vis[x][y] && a[tx][ty] ==0) //若果当前小个子未走过并且可以走
            {
                vis[tx][ty] = true; //将当前的小个子标记为已走过
                dfs(tx, ty, step +1);//探索该方向，当（tx，ty）在迷宫范围内的时候，可走且未被走过。进入点（tx，ty）继续探索
                vis[tx][ty] = false;// 回溯时，恢复改点的原始状态，当从点（tx，ty）返回的时候，要将vis[tx][ty]设成false避免对后续路径产生影响
            }
        }
    }
}
```

完整的程序如下：

```
#include<iostream>
using namespace std;

inx dx[4] = {1, 0, -1, 0};   //四个方向的行偏移量
int dy[4] = {0, -1, 0 , 1};  //四个方向的列偏移量
int n, m ans, a[15][15];
bool vis[15][15];
{
    if(x == n && y ==m) //如果到达右下角
    {
        ans = min(ans, step); //更新最短路径
        return;
    }
    int tx, ty;
    for(int x =0;i<4;++i)  //探索四个方向
    {
        tx = x + dx[i];
        ty = y + dy[i];
        if( tx >=1 && tx <=n && ty >=1 && ty <= m)  //如果探索的小个子在迷宫阵内
        {
            if(!vis[x][y] && a[tx][ty] ==0) //若果当前小个子未走过并且可以走
            {
                vis[tx][ty] = true; //将当前的小个子标记为已走过
                dfs(tx, ty, step +1);//探索该方向，当（tx，ty）在迷宫范围内的时候，可走且未被走过。进入点（tx，ty）继续探索
                vis[tx][ty] = false;// 回溯时，恢复改点的原始状态，当从点（tx，ty）返回的时候，要将vis[tx][ty]设成false避免对后续路径产生影响
            }
        }
    }
}

int main()
{
    cin >> n >> m;
    for(int i = 1; i<= n;i++ )
    {
        for(int j = 1; j<=m; j++)
            cin >> a[i][j];
    }
    ans = 0x3f3f3f3f;
    vis[1][1] = true;    //找最短路径
    dfs(1, 1, 0);        //标记为已访问
    cout << ans;         //调用函数
    return 0;
}
```

2.2深度优先搜索

深度优先搜索尝试一条路径走到底，然后在回溯去探索其他路径，深度优先搜索该怎么实现呢，有递归和非递归两种表现方式。

DFS 算法-递归形式

- DFS算法框架一

```
void dfs(int k)
{
    if(是解)
    {
        输出解;
        return;
    }
    for(int i = 1; i <= 搜索分支可能数; i++)
    {
        if(如果当前分枝不满足条件)
            continue;
        保存当前状态;
        dfs( k + 1 );
        恢复保存当前的状态;  //回溯
    }
}
```

- DFS算法框架2二

```
void dfs(int k)
{
    for(int i = 1; i <= 搜索分支可能数; i++)
    {
        if(如果当前分枝不满足条件)
            continue;
        保存当前状态;
        if(是解)
        {
            输出解;
        }
        else
        {
            dfs( k + 1 );
        }
        恢复保存当前的状态;  //回溯
    }
}
```

DFS算法非递归形式 (使用较少)

1. 将根节点放入栈（stack）中
2. while(!stack.empty()) //当栈不为空
   从stack 中取出第一个节点， 并检验它是否为目标
   如果找到目标， 则结束搜导并回传结果
   否则将它某一个尚未检验过的直接子节点加入stack中
3. 如果不存在未检测过的直接子节点
   将上一级节点加入 stack中
4. 重复步骤3。
5. 若stack 为空，表示整张图都检查过了一亦即图中没有欲搜导的目标。 结束搜导并回传“找不到目标”。