西安电子科技大学

人工智能概论大作业

题　目：基于深度优先搜索算法的迷宫问题探索

姓　名：XXX

学　号：2300920XXXX

专　业：

摘要

本文以图论中的“树”理论为基础，通过对深度优先搜索算法思想的理解和应用，结合数据结构中的“队列”，成功实现了对迷宫问题的解决。本研究的目标是将抽象的问题具体化，并为解决迷宫问题提供有效的算法。在论文中，详细介绍了深度优先搜索算法的核心代码框架，并展示了该算法的实际应用效果。结果表明，深度优先搜索算法在解决迷宫问题中具有良好的性能和可行性。

关键词： 迷宫问题 深度优先搜索 算法

一．绪论

1. 背景知识

1） 迷宫问题

1.图的存储结构

图的存储结构又称图的表示，是用来表示图中各个顶点及其关系的方法，其最常用的方法是邻接矩阵和邻接表。

邻接矩阵是一个二维数组，其中行和列分别表示图中的顶点，数组中的元素表示两个顶点之间是否存在边或弧的关系。如果存在边或弧，则相应位置的元素为1，否则为0。邻接矩阵的优点是可以快速判断两个顶点之间是否存在关系，但缺点是当图中顶点较多时，矩阵的空间复杂度较高。

邻接表是由链表构成的数组，数组的每个元素对应一个顶点，链表中存储该顶点与其他顶点之间的关系。邻接表的优点是对于稀疏图来说可以节省空间，但缺点是在判断两个顶点之间是否存在关系时需要遍历链表。

无论采用什么存储方式，其目标总是相同的，不仅要存储图中各个顶点的信息，同时还要存储顶点之间的所有关系。

2.图的遍历

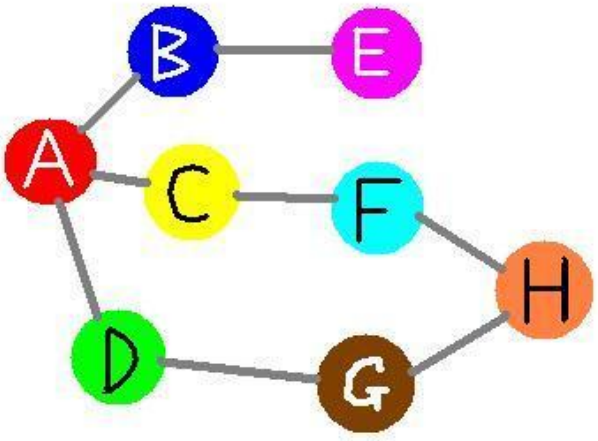
图的遍历是指按照某种顺序访问图中的每一个顶点，确保每个顶点恰好被访问一次。图的遍历是图论中的基本操作，常用于解决各种图相关的问题，如搜索、连通性检测等。常见的图遍历算法有深度优先搜索

（Depth-First Search, DFS）和广度优先搜索（Breadth-First Search, BFS）。

深度优先搜索（Depth-First Search, DFS）是一种用于遍历或搜索树或图的算法。它的基本思想是从根节点（或任意一个起点）开始，尽可能深地搜索图中的每个分支。当到达图的某个节点时，如果它还有未被访问的相邻节点，就继续向下访问；如果没有未被访问的相邻节点，则回溯到上一个节点，继续访问该节点的其他未被访问的相邻节点。这一过程持续到所有节点都被访问过为止。

广度优先搜索（Breadth-First Search, BFS）是一种用于遍历或搜索树或图的算法。与深度优先搜索（DFS）不同，BFS从根节点（或任意一个起点）开始，逐层向外扩展，首先访问所有直接相邻的节点，然后访问这些相邻节点的相邻节点，以此类推，直到所有节点都被访问过。

本实验中用到的是深度优先搜索遍历。即首先访问初始点**V**，并将其标记为已经访问过，接着访问**V**的所有未被访问过的邻接点，顺序任意，并均标记为已访问过，以此类推，直到图中所有和初始点**V**有路径相通的顶点都被访问过为止。往深处一直搜索，走过的点都被记录下来，直到找到目标或者是触发终止条件最后回溯到走过的某一个点，在这个点上面继续进行深度搜索，最后一个图中的所有的点都能被遍历到，再用深度搜索需要注意的点就是我们在回溯的时候我们记录的点要记得还原成没有记录过的状态。



图

深度优先搜索（DFS）

举例说明之：下图是一个无向图，如果我们从 A 点发起深度优先搜索（以下的访问次序并不是唯一的，第二个点既可以是 B 也可以是 C,D），则我们可能得到如下的一个访问过程：A->B->E（没有路了！回溯到 A)->C->F->H->G->D（没有路，最终回溯到 A,A 也没有未访问的相邻节点，本次搜索结束）。简要说明深度优先搜

索的特点：每次深度优先搜索的结果必然是图的一个连通分量。深度优先搜索可以从多点发起。如果将每个节点在深度优先搜索过程中的"结束时间"排序（具体做法是创建一个 list，然后在每个节点的相邻节点都已被访问的情况下，将该节点加入 list 结尾，然后逆转整个链表)，则我们可以得到所谓的"拓扑排序",即 topological sort。

深度优先遍历图的方法是，从图中某顶点 v 出发：

（1）访问顶点 v；

（2）依次从 v 的未被访问的邻接点出发，对图进行深度优先遍历；直至图中和 v 有路径相通的顶点都被访问；

（3）若此时图中尚有顶点未被访问，则从一个未被访问的顶点出发，重新进行深度优先遍历，直到图中所有顶点均被访问过为止。

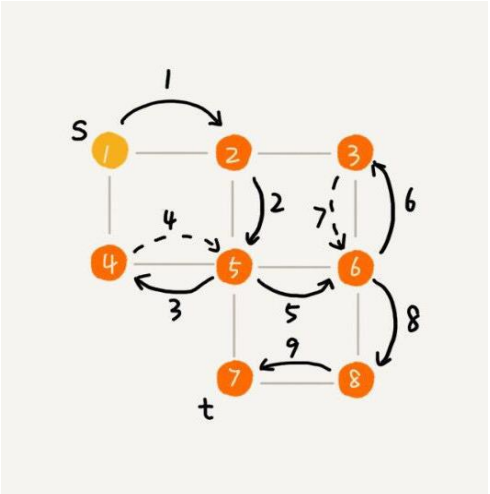
2. 背景介绍

1） 相关理论

深度优先搜索最直观的例子就是“走迷宫”。

假设你站在迷宫的某个岔路口，然后想找到出口。你随意选择一个岔路口来走，走着走着发现是条死路，你就会回退到上一个岔路口，重新选择一条路继续走，直到最终找到出口。这种走法就是一种深度优先搜索策略。

走迷宫的例子很好理解，现在我们把这个思想用到图中，来找一个顶点到另一个顶点的路径。用一个图来说明深度优先搜索的过程，搜索的起始点是 s，终止顶点是 t，我们希望在图中寻找一条从顶点 s 到顶点 t 的路径。图中的实现箭头表示遍历，虚线箭头表示回退。从图中可以看出，深度优先搜索找出来的路径，并不是顶点 s 到顶点 t 的最短路径。实际上，深度优先搜索用的是一种比较著名的算法思想——回溯思想。



```
“
    const int TREE_SIZE = 9;
    std::stack<node*> visited, unvisited;
    node nodes[TREE_SIZE];
    node* current;
    for( int i=0; i<TREE_SIZE; i++) //初始化树
        {
            nodes[i].self = i;
            int child = i*2+1;
            if( child<TREE_SIZE ) //Left child
            nodes[i].left = &nodes[child];
            else nodes[i].left = NULL;
            child++;
            if( child<TREE_SIZE ) //Right child
            nodes[i].right = &nodes[child];
            else       nodes[i].right = NULL;
        }
     unvisited.push(&nodes[0]); //先把0放入UNVISITED stack
     while(!unvisited.empty()) //只有UNVISITED不空
     {
         current=(unvisited.top()); //当前应该访问的
         unvisited.pop();
         if(current->right!=NULL)
         unvisited.push(current->right); // 把右边压入 因为右边的访问次序是在左边之后
         if(current->left!=NULL)
         unvisited.push(current->left);
         visited.push(current);
         cout<<current->self<<endl;
      }
”
```

2） 问题引入

无论是深度优先搜索还是广度优先搜索，其本质都是将图的二维顶点结构线性化的过程，并将当前顶点相邻的未被访问的顶点作为下一个顶点，广度优先搜索采用队列作为数据结构。

本实验的目的是设计一个程序,实现手动或者自动生成一个 N*M 矩阵的迷宫，寻找一条从入口点到出口点的通路具体实验内容如下：

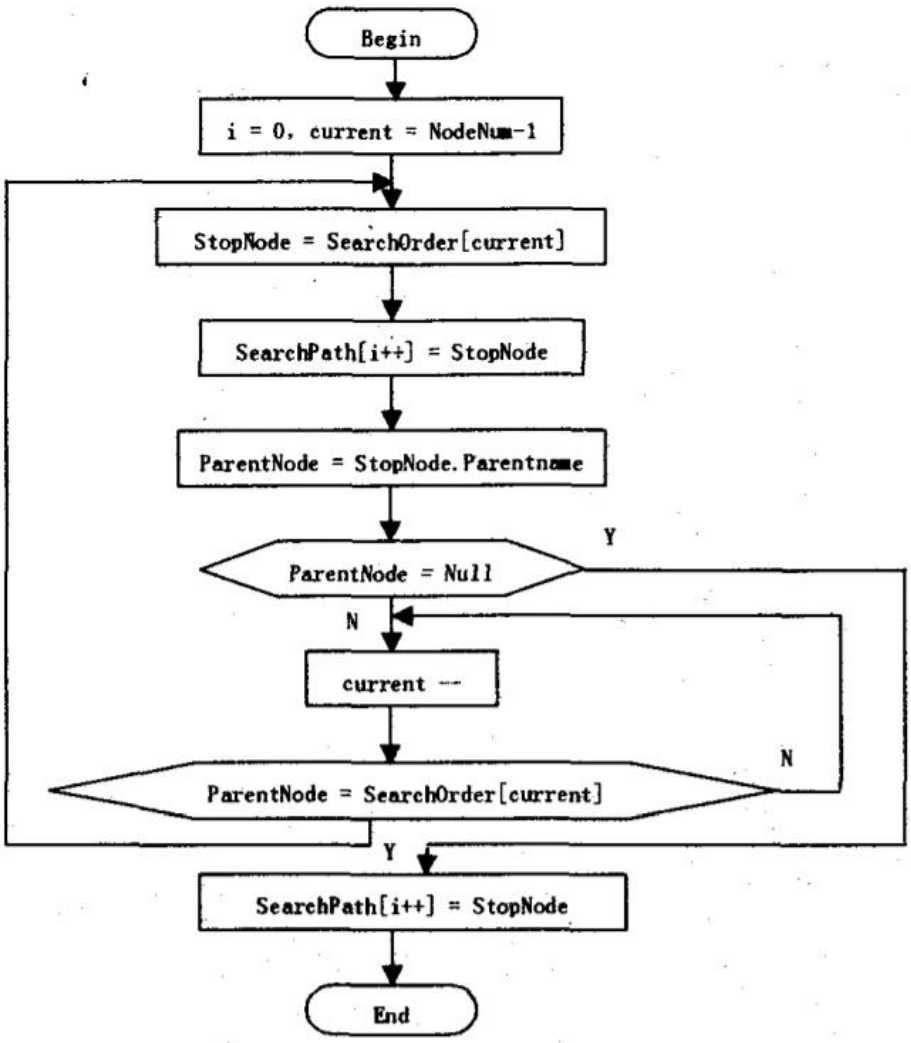
给定一个大小为N*M的迷宫。迷宫由通道和墙壁组成,每一步可以向邻接的上下左右四格的通道移动。请求出从起点到终点所需的最小步数(起点一定可以到达终点)。

二．算法介绍

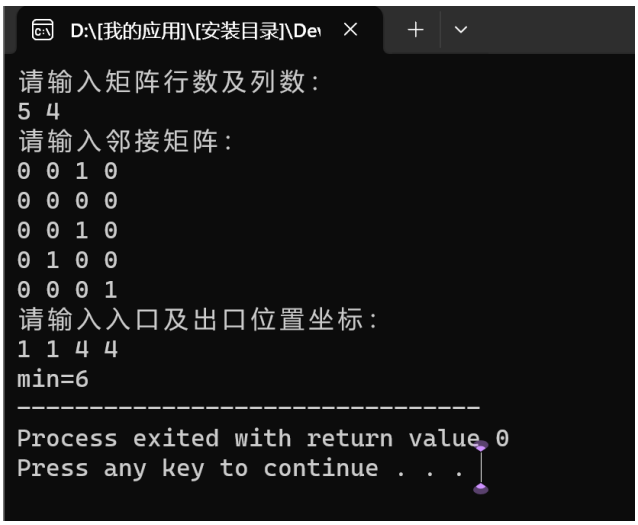
1.算法的基本思路

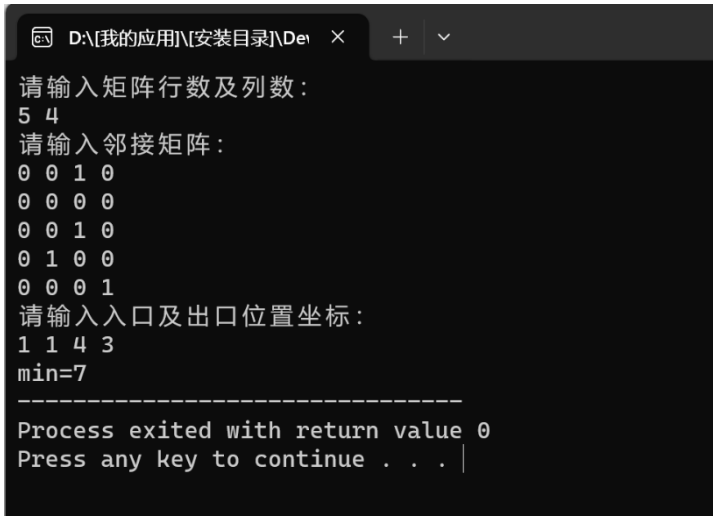
分析上述过程，在对搜索过程中．在图中每个顶点至多调用一次，因为一旦某个顶点标志成已被访问，就不再从它出发进行搜索。因此。搜索过程的实质上是对每个顶点查找其邻接点的过程。其耗费的时间取决于所采用的存贮结构，当用二维数组表示的邻接矩阵作图的存贮结构时，查找每个顶点的邻接点所需的时间为$O(n^2)$，其中n为图中顶点数；而当以邻接表作图的存贮结构时，找邻接点所需时间为O(e)，其中e为无向图中边的数或有向图中弧的数。因此，当以邻接表作存贮结构时。搜索顺序算法的时间复杂度为 O(n+e)，搜索路径生成算法的时间复杂度为O(n)。

2.算法的实现方法（流程图）



三．实验过程





四．实验结论

1.结论

(1)深度优先搜索遍历:深度优先搜索是一个递归的过程。首先访问一个顶点并标记为已访问，然后从该顶点的任意一个未被访问的邻接点出发进行深度优先搜索遍历。当所有邻接点均被访问过时，就退回到上一个顶点 Vk，然后从 Vk 的另一个未被访问过的邻接点出发进行深度优先搜索遍历。如此循环执行，直到退回到初始点并且没有未被访问过的邻接点为止。

(2)广度优先搜索遍历:广度优先搜索的过程如下：首先访问初始点 Vi 并标记为已访问，然后依次访问所有未被访问过的邻接点，其访问顺序可以是任意的，假设依次为 Vi1, Vi2, ..., Vin 并标记为已访问。接着按照 Vi1, Vi2, ..., Vin 的次序访问每一个顶点的所有未被访问过的邻接点，并将它们标记为已访问。依次类推，直到图中所有和初始点有路径相通的顶点都被访问过为止。在设计迷宫问题时，我们要考虑使用二维数组。例如，我们选择 maze[n+2][n+2] 来表示迷宫，而不是用 maze[N][N] 来表示。

2.优缺点

• 深度优先搜索：

1）优点：

1.简单易实现：深度优先搜索算法相对简单直观，易于实现。

2. 空间消耗少：在搜索过程中只需要保存当前路径上的节点，不需要保存整个搜索空间，因此占用空间相对较少。

2）缺点:

1. 可能陷入局部最优解：由于深度优先搜索沿着一条路径尽可能深地搜索，因此有可能错过最优解而陷入局部最优解。这是因为深度优先搜索没有全局信息，只依赖于当前路径进行决策。

2. 不适用于非连通图：对于非连通图，深度优先搜索需要额外的处理来确保所有节点都被访问到。

3. 可能无法找到最优解：深度优先搜索并不保证找到最优解，尤其是在搜索空间很大且分支较多的情况下，可能导致搜索时间很长。

• 广度优先搜索：

1） 优点：

1. 找到最短路径：在无权图或者权值较小的图中，广度优先搜索可以找到从起始节点到目标节点的最短路径。由于广度优先搜索逐层扩展，所以当首次访问到目标节点时，就是最短路径。

2.不会陷入局部最优解：由于广度优先搜索逐层扩展，因此可以确保找到的解是最优解之一。这是因为广度优先搜索会先访问离起始节点最近的节点，然后逐层扩展到离起始节点更远的节点。

3.可以用于生成树和拓扑排序：广度优先搜索可以用来生成树（BFS树），并且在有向无环图中进行拓扑排序。

2） 缺点：内存耗费量大（需要开大量的数组单元用来存储状态）。

3.算法比较

1）深度优先算法占内存少但速度较慢，广度优先算法占内存多但速度较快，在距离和深度成正比的情况下能较快地求出最优解。

2）深度优先与广度优先的控制结构和产生系统很相似，唯一的区别在于对扩展节点选取上。由于其保留了所有的前继节点，所以在产生后继节点时可以去掉一部分重复的节点，从而提高了搜索效率。

3）这两种算法每次都扩展一个节点的所有子节点，而不同的是，深度优先下一次扩展的是本次扩展出来的子节点中的一个，而广度优先扩展的则是本次扩展的节点的兄弟点。在具体实现上为了提高效率,所以采用了不同的数据结构。

五。 参考文献

[1] 九度1335闯迷宫．九度,2013-12-31。

[2] 严蔚敏，吴伟民 著．数据结构（C语言版）：清华大学出版社，2012-05-01。
[3] 谭浩强．C语言程序设计．北京：清华大学出版社，1999。
[4] 苟帅，张俊平．C++Builder 5．0编程员指南．北京：希望电子出版社,2000。
[5] 王晓东．算法设计与分析．北京：清华大学出版社，2004。

附录：代码

```
#include <stdio.h>
void DFS(int,int,int);
int edge[50][50],visited[50][50],min=999;
//定义一个数组edge用于储存迷宫地图，地图中0代表可达，1代表不可达
//定义一个数组visited用来标记已经遍历过的点，min代表走出迷宫需要的最少步数
int n,m,startx,starty,endx,endy;//定义出口和入口的横纵坐标
int next[4][2]={ {0,1},{1,0},{0,-1},{-1,0} };//定义一个数组next用来表示下一步的方向

int main()
{
    int i,j;//定义i代表迷宫地图每行的点数，j代表每列的点数
  printf("请输入矩阵行数及列数：\n");
     scanf ("%d%d",&n,&m);//输入 n行 m列 的迷宫地图
  printf("请输入邻接矩阵：\n");
    for (i=1;i<=n;i++)
    {
        for (j=1;j<=m;j++)
            scanf ("%d",&edge[i][j]);
    }//输入迷宫地图，数组的下标从1开始
    printf("请输入入口及出口位置坐标：\n") ;
    scanf ("%d%d%d%d",&startx,&starty,&endx,&endy);//输入入口和出口的横纵坐标
    DFS(startx,starty,0);//深度优先搜索函数
    printf ("min=%d",min);
    return 0;
}

void DFS(int x,int y,int count)//传入当前位置坐标和已经走过的步数
{
```

```
        if (x==endx&&y==endy)
        {
            if (count<min)//如果已经到达出口，判断该路径是否为当前最短路径
                min=count;
            return;//递归结束，返回上一步
        }
        int i,nextx,nexty;//下一步要走的位置坐标
        for (i=0;i<4;i++)//利用循环来搜索所有可能的路径
        {
            nextx=x+next[i][0];//计算下一步的位置坐标
            nexty=y+next[i][1];
            if (nextx<1||nextx>n||nexty<1||nexty>m)//判断是否越界，未越界则继续执行
                continue;
            if (edge[nextx][nexty]!=1&&visited[nextx][nexty]!=1)
            {
                visited[nextx][nexty]=1;//如果下一步没越界，继续判断该位置是否为障碍物和是否已经遍历
                //若满足条件则标记已经遍历
                DFS(nextx,nexty,count+1);//递归查找下一步骤，并将步数加一
                visited[nextx][nexty]=0;//全部遍历过后，将路径的标记全部取消
                //走到出口执行该语句
                //从出口进行回溯，最终回到入口
            }
        }
}
```