人工智能实验八

18340146 计算机科学与技术 宋渝杰

任务：

从无信息搜索和启发式搜索算法中分别选择一个策略解决迷宫问题。

类型一：一致代价、迭代加深、双向搜索

类型二：A*、IDA*

报告要求：

- 需要有对实现的策略的原理解释
- 需要策略的实验效果（四个方面的算法性能）的对比和分析
- 思考题回答自己对不同策略优缺点，适用场景的理解和认识

算法原理：

本次实验我选择了**迭代加深**和 **IDA*** 进行实验，当然它们都基于深度优先搜索：

深度优先搜索（DFS）： 图遍历的一种算法，在本次实验中，具体的步骤如下：

1. 访问起点 'S'
2. 依次（任意）从顶点的未被访问的邻接点出发，对图进行深度优先遍历（本次实验中，如果该邻接节点是路径 '0'，则继续对该邻接节点的一个未被访问的邻接节点进行搜索；如果是障碍 '1' 或之前搜索过的节点，则停止当前搜索，对另一个邻接节点进行搜索；如果是终点 'E'，则结束搜索），直到访问到终点或者图中和起点有路径相通的顶点都被访问

深度优先搜索的性能：

- 完备性：是：本问题属于状态空间有限，且可以对重复路径进行剪枝的问题
- 最优性：否：深度优先搜索不能确保找到最优解
- 时间复杂度：$O(b^m)$，其中 b 是图节点的最大分支（本实验 $b=4$），m 是从顶点出发的最长路径长度
- 空间复杂度：$O(bm)$

迭代加深搜索（IDS）： 因为深度优先搜索可能会在搜索过程中经过一条很长的无效路径，因此该算法提供一个限度 limit，控制每次深度优先搜索最大深度。在本次实验中，具体的步骤如下：

1. 设定一个合适的 limit 初始值（本次实验中，设定为起点和终点的曼哈顿距离）
2. 以该 limit 值进行深度受限搜索（即控制深度优先搜索的深度不能超过 limit）
3. 当某次限度为 limit 的深度优先搜索无法找到解时，将 limit = limit+1，重新进行深度受限搜索，直到找到一个解为止

迭代加深搜索的性能：

- 完备性：是：本问题属于状态空间有限，且可以对重复路径进行剪枝的问题
- 最优性：是：迭代加深搜索由于控制限度每次 +1，可以确保找到最优解
- 时间复杂度：$O(b^d)$，其中 d 是最短路径长度
- 空间复杂度：$O(bd)$

迭代加深 A*（IDA*）： 相当于启发式 IDS，具体的算法步骤如下：

1. 设定一个合适的 limit 初始值（本次实验中，设定为起点和终点的曼哈顿距离）
2. 以该 limit 值进行深度受限搜索，但对于深度优先搜索的第二步，修改"依次（任意）从顶点的未被访问的邻接点出发，对图进行深度优先遍历"为：**计算所有邻接点的估价函数值，优先搜索估价函数值低的节点**，且在某次搜索过程中，如果该节点的估价函数值大于 limit，则马上进行剪枝
3. 当某次限度为 limit 的深度优先搜索无法找到解时，将 limit = limit+1，重新进行深度受限搜索，直到找到一个解为止

IDA* 的估价函数：$f(x) = h(x) + g(x)$，其中 $g(x)$ 为从起点到节点 x 付出的实际代价，$h(x)$ 为从节点 x 到终点的最优路径的估计代价

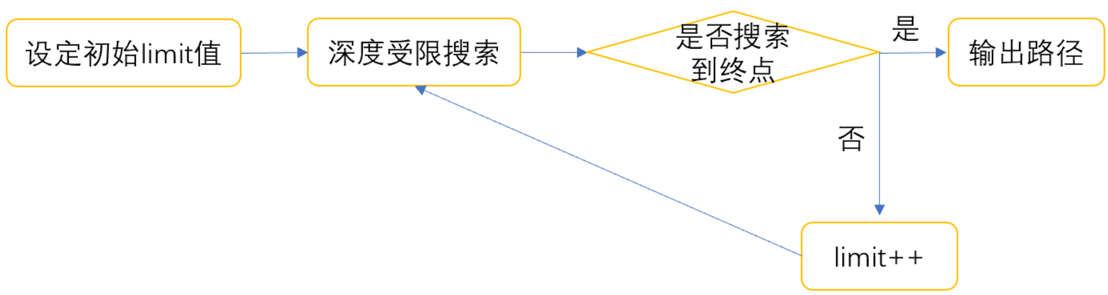
本次实验 $h(x)$ 采取曼哈顿距离：$h(x) = \mathrm{abs}(x_x - e_x) + \mathrm{abs}(x_y - e_y)$

迭代加深 A* 的性能：

- 完备性：是：本问题属于状态空间有限，且可以对重复路径进行剪枝的问题
- 最优性：是：迭代加深 A* 由于控制限度每次 +1，可以确保找到最优解
- 时间复杂度：$O(b^d)$
- 空间复杂度：$O(bd)$

伪代码/流程图：

迭代加深搜索（IDS）：



迭代加深 A*（IDA*） 相比于 IDS 仅仅是修改了深度受限搜索的具体细节，整体框架和 IDS 相同

代码展示：

下面仅展示关键代码，全部代码请移步 18340146_songyujie_lab8.cpp 文件

读取测试集和验证集：

```
int sx, sy, ex, ey; // start end 坐标
string maze[20]; // 迷宫地图

freopen("MazeData.txt", "r", stdin);
while (cin >> maze[n]) {
    m = maze[n].length();
    for (j=0; j<m; j++) {
        if (maze[n][j] == 'S') {
            sx = n;
            sy = j;
        }
```

```
        if (maze[n][j] == 'E') {
            ex = n;
            ey = j;
        }
    }
    n++;
}
```

DFS/IDS:

```
void DFS(int x, int y, int g) { // 普通 DFS 时 limit = 10000000（无穷大）
    num++; // 记录搜索的次数
    if (maze[x][y] == 'E') {
        stop = 1; // 结束标识符
        cout << "len: " << g << endl; // 输出路径长度
        cout << "Road: " << endl; // 输出路径
        for (int i=0; i<n; i++) cout << maze[i] << endl;
        return;
    }
    if (maze[x][y] == '0') maze[x][y] = 'R';
    if (stop == 0 and (maze[x][y-1] == '0' or maze[x][y-1] == 'E') and g+1 <=
limit) DFS(x, y-1, g+1); // 四个方向 DFS
    if (stop == 0 and (maze[x+1][y] == '0' or maze[x+1][y] == 'E') and g+1 <=
limit) DFS(x+1, y, g+1);
    if (stop == 0 and (maze[x][y+1] == '0' or maze[x][y+1] == 'E') and g+1 <=
limit) DFS(x, y+1, g+1);
    if (stop == 0 and (maze[x-1][y] == '0' or maze[x-1][y] == 'E') and g+1 <=
limit) DFS(x-1, y, g+1);
    maze[x][y] = '0'; // 回溯
}

void IDS(int x, int y) { // IDS
    while (stop == 0) {
        DFS(x, y, 0);
        limit++; // 限制++
    }
}
```

IDA*:

```
struct node{
    int x, y, g, h; // 横纵坐标、f = g+h
};

int cmp(node a, node b) {
    return a.g+a.h < b.g+b.h; // f 升序排列
}

int h(int x, int y) { // 曼哈顿距离
    return abs(x-ex)+abs(y-ey);
}

void IDA_DFS(int x, int y, int g) {
    num++; // 记录搜索的次数
    if (maze[x][y] == 'E') {
        stop = 1; // 结束标识符
```

```
        cout << "len: " << g << endl; // 输出路径长度
        cout << "Road: " << endl; // 输出路径
        for (int i=0; i<n; i++) cout << maze[i] << endl;
        return;
    }
    if (maze[x][y] == 'O') maze[x][y] = 'R';
    node Node[4];
    Node[0].x = x-1; Node[0].y = y; Node[0].g = g+1; Node[0].h = h(x-1, y); // 四个节点
    Node[1].x = x+1; Node[1].y = y; Node[1].g = g+1; Node[1].h = h(x+1, y);
    Node[2].x = x; Node[2].y = y-1; Node[2].g = g+1; Node[2].h = h(x, y-1);
    Node[3].x = x; Node[3].y = y+1; Node[3].g = g+1; Node[3].h = h(x, y+1);
    sort(Node, Node+4, cmp); // 按照估价函数 f 升序排序
    for (int i=0; i<4; i++) {
        if (stop == 0 and (maze[Node[i].x][Node[i].y] == 'O' or maze[Node[i].x][Node[i].y] == 'E') and Node[i].g+Node[i].h <= limit)
            IDA_DFS(Node[i].x, Node[i].y, Node[i].g);
    }
    maze[x][y] = 'O'; // 回溯
}

void IDA_star(int x, int y) { // IDA_star
    while (stop == 0) {
        IDA_DFS(x, y, 0);
        limit++; // 限制++
    }
}
```

DFS、IDS、IDA* 运行代码:

```
// DFS
stop = num = 0;
limit = 1e7; // 普通 DFS 时 limit 设为无穷大
cout << "Normal DFS:" << endl;
QueryPerformanceCounter(&t1); // 计时
DFS(sx, sy, 0);
QueryPerformanceCounter(&t2);
time = (double)(t2.QuadPart-t1.QuadPart)/(double)tc.QuadPart;
printf("Time: %.4fms\n", time*1000);
cout << "Search count: " << num << endl << endl;

// IDS
stop = num = 0;
limit = abs(sx-ex)+abs(sy-ey); // 默认为起点和终点的曼哈顿距离
cout << "IDS:" << endl;
QueryPerformanceCounter(&t1);
IDS(sx, sy);
QueryPerformanceCounter(&t2);
time = (double)(t2.QuadPart-t1.QuadPart)/(double)tc.QuadPart;
printf("Time: %.4fms\n", time*1000);
cout << "Search count: " << num << endl << endl;

// IDA_star
stop = num = 0;
limit = abs(sx-ex)+abs(sy-ey); // 默认为起点和终点的曼哈顿距离
cout << "IDA_star:" << endl;
QueryPerformanceCounter(&t1);
```

```
IDA_star(sx, sy);
QueryPerformanceCounter(&t2);
time = (double)(t2.QuadPart-t1.QuadPart)/(double)tc.QuadPart;
printf("Time: %.4fms\n", time*1000);
cout << "Search count: " << num << endl << endl;
```

实验结果以及分析：

三种算法（DFS、IDS、IDA*）的实验结果如下：

DFS：

```
Normal DFS:
len: 152
Road:
111111111111111111111111111111111111
1RRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRRS1
1R1111111111111111111111101111111101
1R11RRR1RRR1000000111111100011000001
1R11R1R1R1R10111101111111110110111111
1R11R1R1R1R1RRR000000000011011000001
1R11R1R1R1R1R1R111100111000011111101
1R10R1R1R1RRR1RRRR110111111110000001
1R11R1R1R11111111R110000000011011111
1R11R1RRR11RRRRRRR111111111011000001
1RRRR111111R11111110000001101111101
111111RRRRRR1RRRRRRR111101101000001
1RRRRRR111111R11111R100001101101111
1R1111111RRRRRR1RRRRR0011111011000001
1RRRRRRRR111111R111111111111011001101
11111111111RRRRRR000000000000011111101
1ERRRRRRRRR1111111111111111100000000 1
111111111111111111111111111111111111
Time: 19.5014ms
Search count: 154
```

IDS：

```
IDS:
len: 68
Road:
111111111111111111111111111111111111
1000000000000000000000000RRRRRRRRRS1
1011111111111111111111111R1111111101
1011000100010000001111111RRR11000001
101101010101011110111111111R11011111
10110101010100000000RRRRR11R11000001
10110101010101011110R111RRRR11111101
10100101010001000011R111111110000001
10110101011111111011RRRRRRRR11011111
101101000110000000111111111R11000001
100001111110111111100000011R11111101
111111000000100000001111011R10000001
100000011111101111101000011R11011111
101111110000000100000001111R11000001
100000000111111011111111111R11001101
1111111111RRRRRRRRRRRRRRRRRR11111101
1ERRRRRRRRR1111111111111111000000001
111111111111111111111111111111111111
Time: 24.7595ms
Search count: 11457
```

IDA*:

```
IDA_star:
len: 68
Road:
111111111111111111111111111111111111
1000000000000000000000000RRRRRRRRRS1
1011111111111111111111111R1111111101
1011000100010000001111111RRR11000001
101101010101011110111111111R11011111
10110101010100000000RRRRR11R11000001
10110101010101011110R111RRRR11111101
10100101010001000011R111111110000001
10110101011111111011RRRRRRRR11011111
101101000110000000111111111R11000001
100001111110111111100000011R11111101
111111000000100000001111011R10000001
100000011111101111101000011R11011111
101111110000000100000001111R11000001
100000000111111011111111111R11001101
1111111111RRRRRRRRRRRRRRRRRR11111101
1ERRRRRRRRR1111111111111111000000001
111111111111111111111111111111111111
Time: 26.0594ms
Search count: 3243
```

可以看出朴素 DFS 虽然找到的解不是最优解，但是从时间和搜索递归次数来看都是表现最好的方案。IDA* 的搜索递归次数（3243）优于 IDS（11457），而运行时间方面多次实验发现两者各有胜负。

结果解释如下：

- 本次实验是小型地图，而 IDS 和 IDA* 的优势更多地体现在最长路径长度 m 远远大于最优路径长度 d 的情况下。
- IDA* 算法中涉及估价函数的计算以及排序，算法时间复杂度方面常数略大，导致虽然 IDA* 的搜索递归次数远小于 IDS，但是大部分测试中 IDA* 是稍微慢于 IDS 的
- 策略的实验效果（四个方面的算法性能）的对比和分析已在原理叙述中讲述

思考题：

这些策略的优缺点是什么？它们分别适用于怎样的场景？

- 一致代价搜索：**优点：**能确保到搜索第一次到某一个点是沿着最优的路径搜索到的，确保了最优性。**缺点：**和 BFS 一样需要边界队列，要保存指数级的节点数量，空间复杂度大。**适用：**对时间要求大而对空间要求不大，或者最长路径长度远大于最短路径长度的场景
- 迭代加深搜索：**优点：**解决了 DFS 容易在一条错路上浪费太多时间的问题，而且通过控制深度限制，确保了最优性。**缺点：**在限制缓慢+1增大的时候，许多路径被重复搜索，会浪费不少时间。**适用：**对时间要求不大而对空间要求很大（大型搜索问题），或者最长路径长度远大于最短路径长度的场景
- 双向搜索：**优点：**双向 BFS 搜索树会比朴素 BFS 搜索树小，时间和空间方面都比朴素 BFS 优。**缺点：**需要知道终点的坐标，不知道终点坐标则完全失效，终点坐标有多个时存在多颗搜索树，时间和空间方面反而可能劣化。**适用：**终点唯一且知道坐标的场景
- A^* 搜索：**优点：**相当于启发式 Dijkstra 算法，在估值函数设计合理时能起到加速搜索的效果。**缺点：**不解决 BFS 空间复杂度大的问题，而且需要计算估值函数和排序，需要知道终点坐标且算法时间常数略大。**适用：**终点唯一且知道坐标的场景，空间要求不大
- IDA^* 搜索：**优点：**相当于启发式迭代加深搜索，在估值函数设计合理时能实现很快的剪枝，起到加速搜索的效果。**缺点：**和迭代加深搜索一样，在限制缓慢+1增大的时候，许多路径被重复搜索，会浪费不少时间。而且也需要计算估值函数和排序，需要知道终点坐标且算法时间常数略大。**适用：**比迭代加深搜索更适合大型搜索问题，终点唯一且知道坐标的场景