A*搜索和IDA*搜索问题实验报告

PB16111245王立峰

实验目的

利用A*算法和IDA*算法实现迷宫问题。

实验要求

迷宫大小设置为 18*25，所以输入为大小为 18*25 二维数组 A[18][25]数组的下标代表点的 位置，数组的值代表该位置是否可通行，0 表示可通行，1 表示不可通行。本实验中入口和出口 的位置设为(1,0) 和(16,24)；需要的迷宫矩阵input.txt已经给出。

实验内容

下面分别讨论A*和IDA*

A*

A*算法的核心主要在于启发函数的选择和使用上，对于本次实验的迷宫问题， 我选择了使用曼哈顿距离来构造启发函数。**f()=g()+h()**函数中，**g()**函数表示当前节点已走的路径长度，而**h()**表示曼哈顿距离，用来预测当前节点到目标节点的大致距离，所以**f()**函数表示起始节点的大致距离（包含了预测的情况），可以大致用来启发搜索的路径选择，从而选择一条路径代价相对较小的路径。

另外，需要设置open列表和closed列表用来保存节点在路径选择中的情况。

实验主要数据结构

```
struct Node{
    int x;//节点的x坐标
    int y;//节点的y坐标
    char value;//节点的性质，('0'表示可通行，'1'表示不可通行)
    Node *parent;//当前节点的父节点指针
    int f;//节点的f()
    int g;//节点的g()
    int h;//节点的h()
    int openflag;//节点是否加入open列表的标志
    int closedflag;//节点是否加入closed的标志
    char direction;//节点由父节点往哪个方向前进得出来的
```

```
    int pathflag;//节点是否在最终路径的标志
};//存储节点信息的数据结构

struct compare
{
    bool operator()(Node a, Node b) {
        return a.f > b.f;
    }
};//优先级队列中用来排序的函数

int steps=0;//总步数
Node MapNode[XRANGE][YRANGE];//迷宫地图
Node S, E;//起始节点
priority_queue <Node, vector<Node>, compare > open_list;//open列表，优先级队列
queue <Node> closed_list;//closed列表，普通队列
vector <char> path_direction;//保存路径节点的方向
```

主要的函数

```
void readFileSet();
//此函数主要是用来读取文件并将其信息写入__MapNode__中。
void initNode(Node node);
//初始化节点信息
void checkSE(Node start, Node end);
//检测输入的起始节点是否合法
int ManhanttanDistance(Node x, Node y);
//求曼哈顿距离
Node FLowestOpen(priority_queue <Node, vector<Node>, compare> open_list);
//从优先级队列中取出最小元素
bool reachableAdjacentNodes(Node m, char direction);
//检测当前节点的下一步往四个方向是否可达，判断是否为墙或边界
void Astar(Node start, Node end);
//核心的A*搜索算法
```

分函数说明

下面对其中几个重要函数说明

```
bool reachableAdjacentNodes(Node m, char direction) {
    if (direction == 'R') {
        if ((m.y == YRANGE - 1) || (MapNode[m.x][m.y + 1].value == '1')) {
            return false;
        }
        else
            return true;
    }
    else if (direction == 'L') {
        if ((m.y == 0) || (MapNode[m.x][m.y - 1].value == '1')) {
            return false;
        }
        else
            return true;
    }
    else if (direction == 'D') {
        if ((m.x == XRANGE - 1) || (MapNode[m.x + 1][m.y].value == '1')) {
            return false;
        }
        else
            return true;
```

```
    }
    else if (direction == 'U') {
        if ((m.x == 0) || (MapNode[m.x - 1][m.y].value == '1')) {
            return false;
        }
        else
            return true;
    }
}
```

分别对R, L, D, U四个方向检测可达性，即当x或y达到边界值时或者下一步是墙时，则不可达，其余都是可达的。

```
void Astar(Node start, Node end) {
    Node currentNode;
    open_list.push(start);
    start.openflag = 1;
    while (open_list.empty() != true) {
        currentNode = FLowestOpen(open_list);
        closed_list.push(currentNode);
//      currentNode.closedflag = 1;
        MapNode[currentNode.x][currentNode.y].closedflag = 1;
        open_list.pop();
        if (end.closedflag==1) {//if we have added end to closedlist
            break;
        }

    A:  if (reachableAdjacentNodes(currentNode, 'R')) {
            //check east adjacent node is in closedlist
            if (MapNode[currentNode.x][currentNode.y + 1].closedflag == 1) {
                goto B;
            }
            else {
                // east adjacent node is not in openlist
                if (MapNode[currentNode.x][currentNode.y + 1].openflag == 0) {
                    MapNode[currentNode.x][currentNode.y + 1].openflag = 1;
                    MapNode[currentNode.x][currentNode.y + 1].parent =
&MapNode[currentNode.x][currentNode.y];
                    MapNode[currentNode.x][currentNode.y + 1].g = MapNode[currentNode.x]
[currentNode.y].g + 1;
                    MapNode[currentNode.x][currentNode.y + 1].f = MapNode[currentNode.x]
[currentNode.y + 1].g +
                        ManhanttanDistance(MapNode[currentNode.x][currentNode.y + 1], end);
                    open_list.push(MapNode[currentNode.x][currentNode.y + 1]);
                    MapNode[currentNode.x][currentNode.y + 1].direction = 'R';
                }
                else {//east adjacent node is in openlist
                    if (currentNode.f >= MapNode[currentNode.x][currentNode.y + 1].f) {
                        continue;
                    }
                }
            }
        }
    B:  if (reachableAdjacentNodes(currentNode, 'L')) {
            //check west adjacent node is in closedlist
            if (MapNode[currentNode.x][currentNode.y - 1].closedflag == 1) {
                goto C;

            }
            else {
```

```
                //check west adjacent node is in openlist
                if (MapNode[currentNode.x][currentNode.y - 1].openflag == 0) {
                    MapNode[currentNode.x][currentNode.y - 1].openflag = 1;
                    MapNode[currentNode.x][currentNode.y - 1].parent =
&MapNode[currentNode.x][currentNode.y];
                    MapNode[currentNode.x][currentNode.y - 1].g = MapNode[currentNode.x]
[currentNode.y].g + 1;
                    MapNode[currentNode.x][currentNode.y - 1].f = MapNode[currentNode.x]
[currentNode.y - 1].g +
                        ManhanttanDistance(MapNode[currentNode.x][currentNode.y - 1], end);
                    open_list.push(MapNode[currentNode.x][currentNode.y - 1]);
                    MapNode[currentNode.x][currentNode.y - 1].direction = 'L';
                }
                else {
                    if (currentNode.f >= MapNode[currentNode.x][currentNode.y - 1].f) {
                        continue;
                    }
                }
            }
        }
    C:  if (reachableAdjacentNodes(currentNode, 'D')) {
            //check south adjacent node is in closedlist
            if (MapNode[currentNode.x + 1][currentNode.y].closedflag == 1) {
                goto D;
            }
            else {
                //check south adjacent node is in openlist
                if (MapNode[currentNode.x + 1][currentNode.y].openflag == 0) {
                    MapNode[currentNode.x + 1][currentNode.y].openflag = 1;
                    MapNode[currentNode.x + 1][currentNode.y].parent =
&MapNode[currentNode.x][currentNode.y];
                    MapNode[currentNode.x + 1][currentNode.y].g = MapNode[currentNode.x]
[currentNode.y].g + 1;
                    MapNode[currentNode.x + 1][currentNode.y].f = MapNode[currentNode.x + 1]
[currentNode.y].g +
                        ManhanttanDistance(MapNode[currentNode.x + 1][currentNode.y], end);
                    open_list.push(MapNode[currentNode.x + 1][currentNode.y]);
                    MapNode[currentNode.x + 1][currentNode.y].direction = 'D';
                }
                else {
                    if (currentNode.f >= MapNode[currentNode.x + 1][currentNode.y].f) {
                        continue;
                    }
                }
            }
        }
    D:  if (reachableAdjacentNodes(currentNode, 'U')) {
            //check north adjacent node is in closedlist
            if (MapNode[currentNode.x - 1][currentNode.y].closedflag == 1) {
                cout << endl;
                continue;
            }
            else {
                //check north adjacent node is in openlist
                if (MapNode[currentNode.x - 1][currentNode.y].openflag == 0) {
                    MapNode[currentNode.x - 1][currentNode.y].openflag = 1;
                    MapNode[currentNode.x - 1][currentNode.y].parent =
&MapNode[currentNode.x][currentNode.y];
                    MapNode[currentNode.x - 1][currentNode.y].g = MapNode[currentNode.x]
[currentNode.y].g + 1;
```

```
                    MapNode[currentNode.x - 1][currentNode.y].f = MapNode[currentNode.x]
[currentNode.y].g +
                        ManhanttanDistance(MapNode[currentNode.x - 1][currentNode.y], end);
                    open_list.push(MapNode[currentNode.x - 1][currentNode.y]);
                    MapNode[currentNode.x - 1][currentNode.y].direction = 'U';
                }
                else {
                    if (currentNode.f >= MapNode[currentNode.x - 1][currentNode.y].f) {
                        continue;
                    }
                }
            }
        }
    }
}
```

Astar()函数的基本思想就是从起始节点出发，把当前节点加入到closed列表中，从当前节点往四个方向判断，如果下个节点已在closed列表中，就不用考虑了，说明已经走过了，在判断其是否在open列表中，如果不在，则加入open列表，如果已在，则判断当前节点和下一节点的**f()**哪个更大，如果当前的更大或相等，则可以直接结束本次循环，然后选用下一节点作为当前节点。

时间复杂度和空间复杂度分析

A*算法的时间复杂度不高，有些类似广度优先，但空间开销较大，需要维护多个列表等数据结构，而且需要不停更新等。

IDA*

IDA*算法本质上就是一种迭代加深的搜索算法 ，不过采用的截断值是启发函数，总体来说就是一个搜索深度会加深的深度优先搜索，而这个深度的限制与启发函数有关。

主要数据结构

与A*基本差不多，不过没有了open列表和closed列表。

```
struct Node {
    int x;
    int y;
    char value;
//  Node *parent;
    int f;
    int g;
    int h;
    int flag;
    char direction;
};

struct Path {
    int x;
    int y;
};//记录路径

vector <Node> temp_path;
Node MapNode[XRANGE][YRANGE];
Node S, E;
Path path[XRANGE*YRANGE] = { 0 };
int steps;
```

主要函数

主要几个函数也与A*相同，唯一不同就是IDAstar()函数。

```
void readFileSet();
void initNode(Node node);
void checkSE(Node start, Node end);
int ManhanttanDistance(Node x, Node y);
bool reachableAdjacentNodes(Node m, char direction);
void IDAstar(Node currentNode, int bound, int level);
```

分函数说明

```
void IDAstar(Node currentNode, int bound, int level) {
    if (level == bound) {
        if (currentNode.x == E.x && currentNode.y == E.y) {
            temp_path.push_back(currentNode);
            path[level].x = currentNode.x;
            path[level].y = currentNode.y;
            return;
        }
        else {
            IDAstar(currentNode, bound + 1, 1);
        }
    }
    else {
        if (ManhanttanDistance(currentNode, E) + level > bound) {
            return;
        }
        else {
            if (reachableAdjacentNodes(currentNode, 'R')) {
                if (MapNode[currentNode.x][currentNode.y + 1].flag == 1) {
                    goto B;
                }
                path[level].x = currentNode.x;
                path[level].y = currentNode.y;
                MapNode[currentNode.x][currentNode.y + 1].flag = 1;
                steps = steps + 1;
                IDAstar(MapNode[currentNode.x][currentNode.y + 1], bound, level + 1);
                steps = steps + 1;
                temp_path.push_back(currentNode);
            }
        B:  if (reachableAdjacentNodes(currentNode, 'L')) {
                if (MapNode[currentNode.x][currentNode.y - 1].flag == 1) {
                    goto C;
                }
                path[level].x = currentNode.x;
                path[level].y = currentNode.y;
                MapNode[currentNode.x][currentNode.y - 1].flag = 1;
                steps = steps + 1;
                IDAstar(MapNode[currentNode.x][currentNode.y - 1], bound, level + 1);
                steps = steps + 1;
                temp_path.push_back(currentNode);
            }
        C:  if (reachableAdjacentNodes(currentNode, 'D')) {
                if (MapNode[currentNode.x + 1][currentNode.y].flag == 1) {
                     goto D;
                }

                path[level].x = currentNode.x;
                path[level].y = currentNode.y;
```

```
                MapNode[currentNode.x + 1][currentNode.y].flag = 1;
                steps = steps + 1;
                IDAstar(MapNode[currentNode.x + 1][currentNode.y], bound, level + 1);
                steps = steps + 1;
                temp_path.push_back(currentNode);
            }
        D:  if (reachableAdjacentNodes(currentNode, 'U')) {
                if (MapNode[currentNode.x - 1][currentNode.y].flag == 1) {
                    return;
                }
                path[level].x = currentNode.x;
                path[level].y = currentNode.y;
                MapNode[currentNode.x - 1][currentNode.y].flag = 1;
                steps = steps + 1;
                IDAstar(MapNode[currentNode.x - 1][currentNode.y], bound, level + 1);
                steps = steps + 1;
                temp_path.push_back(currentNode);
            }
        }
    }
}
```

此函数中主要是根据设定的深度阈值bound和level来进行迭代加深搜索，当当前搜索深度level达到上限时，若果仍然没有发现解的话，则这时就要加深bound,使得最大搜索搜索深度加深，从而找到解。至于没有达到bound就找到解的，就和普通深度优先搜索一样进行搜索就行了，在其中，可以利用启发函数来进行剪枝，以减小搜索时间成本。

时间复杂度和空间复杂度分析

IDA*算法空间成本不大，只要在回溯树上操作就OK了，但其时间成本比较高，尤其是剪枝策略会深度影响最终的时间成本。

实验结果

	A*(input.txt)	A*(input2.txt)	IDA*(input.txt)	IDA*(input2.txt)
步数	39	118	39	*
时间	0.041	0.363	0.000	*

相应的输出文件和结果等可在实验文件中查看。