A\*算法

1，实验代码  
/\*

//A\*算法对象类

\*/

#include <vector>

#include <math.h>

#include <list>

#include <iostream>

using namespace std;

const int kCost1 = 10; //直移一格消耗

const int kCost2 = 14; //斜移一格消耗

struct Point

{

int x, y; //点坐标，这里为了方便按照C++的数组来计算，x代表横排，y代表竖列

int F, G, H; //F=G+H

Point \*parent; //parent的坐标，这里没有用指针，从而简化代码

Point(int \_x, int \_y) :x(\_x), y(\_y), F(0), G(0), H(0), parent(NULL) //变量初始化

{

}

};

class Astar

{

public:

void InitAstar(std::vector<std::vector<int> > &\_maze);

std::list<Point \*> GetPath(Point &startPoint, Point &endPoint, bool isIgnoreCorner);

private:

Point \*findPath(Point &startPoint, Point &endPoint, bool isIgnoreCorner);

std::vector<Point \*> getSurroundPoints(const Point \*point, bool isIgnoreCorner) const;

bool isCanreach(const Point \*point, const Point \*target, bool isIgnoreCorner) const; //判断某点是否可以用于下一步判断

Point \*isInList(const std::list<Point \*> &list, const Point \*point) const; //判断开启/关闭列表中是否包含某点

Point \*getLeastFpoint(); //从开启列表中返回F值最小的节点

//计算FGH值

int calcG(Point \*temp\_start, Point \*point);

int calcH(Point \*point, Point \*end);

int calcF(Point \*point);

private:

std::vector<std::vector<int> > maze;

std::list<Point \*> openList; //开启列表

std::list<Point \*> closeList; //关闭列表

};

void Astar::InitAstar(std::vector<std::vector<int> > &\_maze)

{

maze = \_maze;

}

int Astar::calcG(Point \*temp\_start, Point \*point)

{

int extraG = (abs(point->x - temp\_start->x) + abs(point->y - temp\_start->y)) == 1 ? kCost1 : kCost2;

int parentG = point->parent == NULL ? 0 : point->parent->G; //如果是初始节点，则其父节点是空

return parentG + extraG;

}

int Astar::calcH(Point \*point, Point \*end)

{

//用简单的欧几里得距离计算H，这个H的计算是关键，还有很多算法，没深入研究^\_^

return sqrt((double)(end->x - point->x)\*(double)(end->x - point->x) + (double)(end->y - point->y)\*(double)(end->y - point->y))\*kCost1;

}

int Astar::calcF(Point \*point)

{

return point->G + point->H;

}

Point \*Astar::getLeastFpoint()

{

if (!openList.empty())

{

auto resPoint = openList.front();

for (auto &point : openList)

if (point->F<resPoint->F)

resPoint = point;

return resPoint;

}

return NULL;

}

Point \*Astar::findPath(Point &startPoint, Point &endPoint, bool isIgnoreCorner)

{

openList.push\_back(new Point(startPoint.x, startPoint.y)); //置入起点,拷贝开辟一个节点，内外隔离

while (!openList.empty())

{

auto curPoint = getLeastFpoint(); //找到F值最小的点

openList.remove(curPoint); //从开启列表中删除

closeList.push\_back(curPoint); //放到关闭列表

//1,找到当前周围八个格中可以通过的格子

auto surroundPoints = getSurroundPoints(curPoint, isIgnoreCorner);

for (auto &target : surroundPoints)

{

//2,对某一个格子，如果它不在开启列表中，加入到开启列表，设置当前格为其父节点，计算F G H

if (!isInList(openList, target))

{

target->parent = curPoint;

target->G = calcG(curPoint, target);

target->H = calcH(target, &endPoint);

target->F = calcF(target);

openList.push\_back(target);

}

//3，对某一个格子，它在开启列表中，计算G值, 如果比原来的大, 就什么都不做, 否则设置它的父节点为当前点,并更新G和F

else

{

int tempG = calcG(curPoint, target);

if (tempG<target->G)

{

target->parent = curPoint;

target->G = tempG;

target->F = calcF(target);

}

}

Point \*resPoint = isInList(openList, &endPoint);

if (resPoint)

return resPoint; //返回列表里的节点指针，不要用原来传入的endpoint指针，因为发生了深拷贝

}

}

return NULL;

}

std::list<Point \*> Astar::GetPath(Point &startPoint, Point &endPoint, bool isIgnoreCorner)

{

Point \*result = findPath(startPoint, endPoint, isIgnoreCorner);

std::list<Point \*> path;

//返回路径，如果没找到路径，返回空链表

while (result)

{

path.push\_front(result);

result = result->parent;

}

// 清空临时开闭列表，防止重复执行GetPath导致结果异常

openList.clear();

closeList.clear();

return path;

}

Point \*Astar::isInList(const std::list<Point \*> &list, const Point \*point) const

{

//判断某个节点是否在列表中，这里不能比较指针，因为每次加入列表是新开辟的节点，只能比较坐标

for (auto p : list)

if (p->x == point->x&&p->y == point->y)

return p;

return NULL;

}

bool Astar::isCanreach(const Point \*point, const Point \*target, bool isIgnoreCorner) const

{

if (target->x<0 || target->x>maze.size() - 1

|| target->y<0 || target->y>maze[0].size() - 1

|| maze[target->x][target->y] == 1

|| target->x == point->x&&target->y == point->y

|| isInList(closeList, target)) //如果点与当前节点重合、超出地图、是障碍物、或者在关闭列表中，返回false

return false;

else

{

if (abs(point->x - target->x) + abs(point->y - target->y) == 1) //非斜角可以

return true;

else

{

//斜对角要判断是否绊住

if (maze[point->x][target->y] == 0 && maze[target->x][point->y] == 0)

return true;

else

return isIgnoreCorner;

}

}

}

std::vector<Point \*> Astar::getSurroundPoints(const Point \*point, bool isIgnoreCorner) const

{

std::vector<Point \*> surroundPoints;

for (int x = point->x - 1; x <= point->x + 1; x++)

for (int y = point->y - 1; y <= point->y + 1; y++)

if (isCanreach(point, new Point(x, y), isIgnoreCorner))

surroundPoints.push\_back(new Point(x, y));

return surroundPoints;

}

int main()

{

//初始化地图，用二维矩阵代表地图，1表示障碍物，0表示可通

vector<vector<int>> maze = {

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },

{ 1, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 1 },

{ 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1 },

{ 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1 },

{ 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1 },

{ 1, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1 },

{ 1, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1 },

{ 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 }

};

Astar astar;

astar.InitAstar(maze);

//设置起始和结束点

Point start(1, 1);

Point end(6, 10);

//A\*算法找寻路径

list<Point \*> path = astar.GetPath(start, end, false);

//打印

for (auto &p : path)

cout << '(' << p->x << ',' << p->y << ')' << endl;

system("pause");

return 0;

}