南開大學
Nankai University

人工智能学院

人工智能技术实验报告

实验二　A* 算法

姓名：石若川

学号：2111381

专业：智能科学与技术

2023 年 10 月 6 日

目录

1 问题简述

A* 算法一种典型的启发式搜索算法，是静态路网中求解最短路径最有效的方法之一。启发式搜索又称为有信息搜索，它是利用问题拥有的启发信息来引导搜索，达到减少搜索范围、降低问题复杂度的目的，通常拥有更好的性能。A* 算法关键部分为启发函数：

$$F = G + H$$

其中，G 为从起点 A 移动到目前方格的代价，H 是该方格到终点 B 的估算成本，F 为寻找下一个遍历节点的依据。

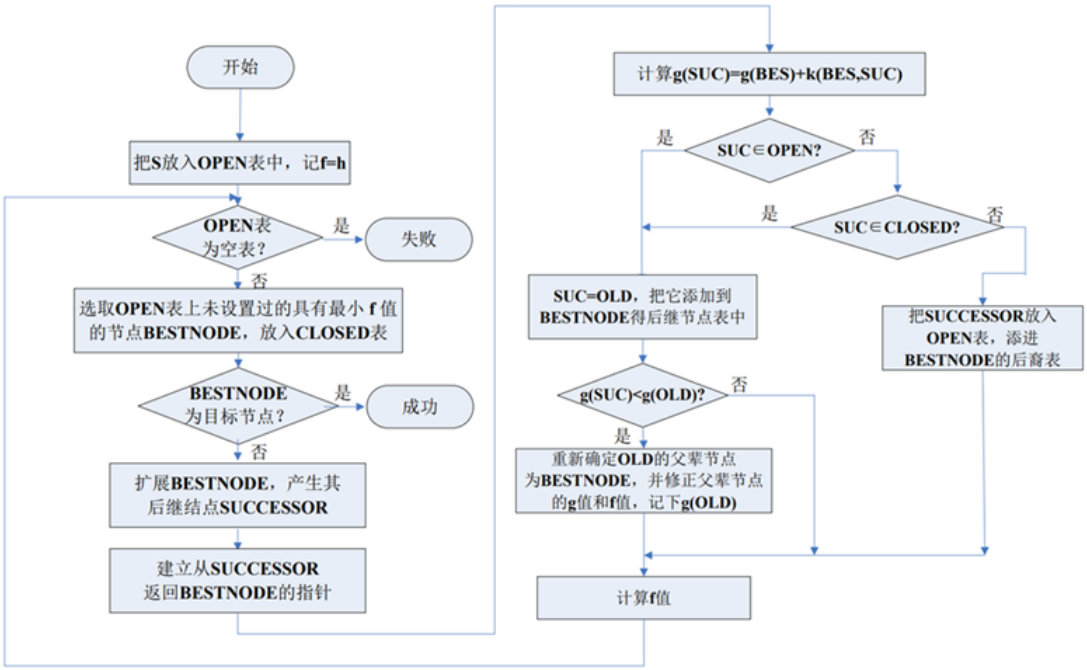


图 1.1: 流程图

算法实现流程：

1. 创建 open list、close list，以起始点 A 开始，将起始点放入 close list；
2. 使用循环获取最佳路径：
   (a) 未找到目标点时，对目前的节点 curr，获取附近节点 neighbor；
   (b) neighbor 中节点为八个方向，根据有无墙体、是否可到达添加，同时计算 G、H 的值；
   (c) 查看 neighbor 中的节点是否在 open list/close list 中，如果不在 open list 中，将该节点添加到 open list 中；否则，更新 open list 中该点的 G 值；
   (d) 查找 open list 中的节点，找出最小 F 值，将该节点作为下一次循环的 curr；
   (e) 循环。
3. 通过父节点回溯最佳路径并将结果存入 list 类型的表中，得到最佳路径表。

2 实验目的

1. 握图搜索算法思路和流程，理解无信息搜索与有信息搜索的区别；
2. 对于启发式搜索，理解 A* 算法估值函数的选取对算法性能的影响；
3. 对于启发式搜索，理解 A* 算法求解流程和搜索顺序；
4. 深入理解对图形化界面设计。（选做）

3 实验内容

1. 实现 A* 算法，以带有障碍物的二维地图为基础，求寻找目标点的最短路径；
2. 将搜索区域划分为方形网格，每个网格内的方块分为可通过和不可通过的，给定起点和终点，要求寻找从起点到终点的最短路径。
3. 对实验进行图形化界面设计，展示搜索过程和最优路径。（选做）

4 编译环境

编译环境为 Windows 11 下 Python3.6，图形化界面利用了 Tkinter 进行可视化。

5 实验步骤

5.1 A* 算法

实验中由于需要求出 F 值最小的结点，因此利用最小堆的结构进行实现。堆是一个有序的树状结构，通常是一棵二叉树。在最小堆中，每个节点的值小于或等于其子节点的值；在最大堆中，每个节点的值大于或等于其子节点的值。堆在 A* 算法中，可以确保每次处理的节点都是最有希望的节点，以提高搜索效率。

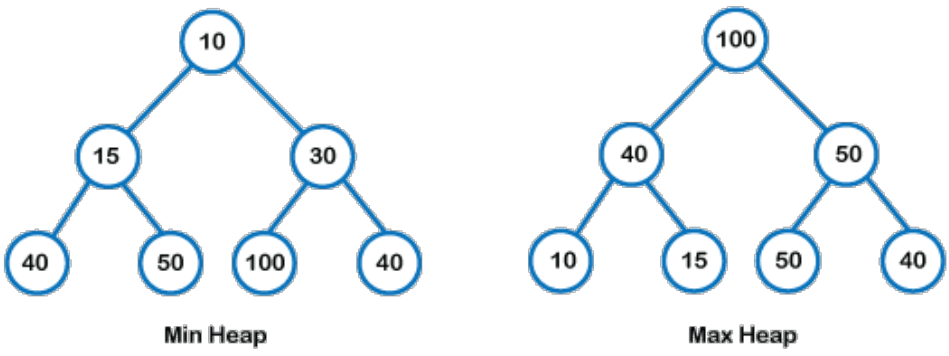


图 5.2: 堆的结构示意图

下面是如何在提供的代码中使用堆来实现 A* 算法的关键部分：

1. 首先创建一个空的堆（最小堆）来存储待考虑的节点：

```
heap = []
```

2. 在 A* 算法的主循环中，选择具有最小估价代价（F 值）的节点作为当前节点。这是通过以下操作实现的：

```
pMin = heapq.heappop(heap)
```

这一行代码将堆中具有最小 f 值的节点弹出，并将其作为当前节点进行处理。

3. 将当前节点从 open list 中取出，添加到 close list 中：

```
open_list.remove((pMin.x, pMin.y))
close_list.add((pMin.x, pMin.y))
```

4. 遍历当前节点的相邻节点，并对每个相邻节点进行如下处理：

   - 如果相邻节点在关闭列表中，则跳过该相邻节点。
   - 计算从起点到该相邻节点的代价（G 值）和启发式估价代价（H 值），然后计算总代价（F 值）。
   - 如果相邻节点不在开放列表中，或者新的 G 值比原来的 G 值更小，就更新相邻节点的 G 值、F 值，并将当前节点设置为相邻节点的父节点。
   - 最后，将相邻节点添加到堆中：

在计算 G 值时有多种距离度量方法，如曼哈顿距离、欧氏距离等。欧氏距离的计算公式如下：

$$d_{12} = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

实验中，节点与上下左右四个相邻结点的距离设为 10，左上、左下、右上、右下四个相邻节点的距离设为 14。代码如下：

```
if (xCur, yCur) not in close_list and self.points[xCur][yCur].state !=
    PointState.BARRIER.value:
    # 在open list中
    if (xCur, yCur) in open_list:
        # 起点到pMin再到pCur的代价比起点到pCur的代价小
        if ((14 if isCorner else 10) + pMin.g) < pCur.g:
            pCur.g = pMin.g + (14 if isCorner else 10)
            pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd - yCur)) #
                更新pCur的代价
            pCur.father = pMin # 将pMin设置为pCur的父节点
    # 不在open list中
    else:
        pCur.g = pMin.g + (14 if isCorner else 10)
        pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd - yCur))
```

```
        pCur.father = pMin # 将pMin设置为pCur的父节点
        self.changeState(pCur, PointState.OPEN.value)
        # 将这个点加入open list
        open_list.add((xCur, yCur))
        heapq.heappush(heap, pCur)
```

曼哈顿距离的计算公式如下：

$$d_{12} = |x_1 - x_2| + |y_1 - y_2|$$

实验中，节点与上下左右四个相邻结点的距离设为 10，左上、左下、右上、右下四个相邻节点的距离设为 20。代码如下：

```
if (xCur, yCur) not in close_list and self.points[xCur][yCur].state !=
    PointState.BARRIER.value:
    # 在open list中
    if (xCur, yCur) in open_list:
        # 起点到pMin再到pCur的代价比起点到pCur的代价小
        if ((20 if i*j != 0 else 10) + pMin.g) < pCur.g:
            pCur.g = pMin.g + (20 if i*j != 0 else 10)
            pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd - yCur)) #
                更新pCur的代价
            pCur.father = pMin # 将pMin设置为pCur的父节点
    # 不在open list中将pMin设置为pCur的父节点
    else:
        pCur.g = pMin.g + (20 if i*j != 0 else 10)
        pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd - yCur))
        pCur.father = pMin # 将pMin设置为pCur的父节点
        self.changeState(pCur, PointState.OPEN.value)
        # 将这个点加入open list
        open_list.add((xCur, yCur))
        heapq.heappush(heap, pCur)
```

5.2 可视化界面的设计

5.2.1 网格

以下代码对网格进行初始化：

```
def generateMesh(self):
    for i in range(self.height + 1):
        self.canvas.create_line((3, i * self.size + 3), (self.width * self.size + 3,
            i * self.size + 3))
    for i in range(self.width + 1):
```

```
        self.canvas.create_line((i * self.size + 3, 3), (i * self.size + 3,
            self.height * self.size + 3))
```

网格的状态有以下几种：障碍物、起点、终点、位于 open list 中、位于 close list 中、路径、未使用。以下代码设置了不同网络状态对应的颜色。

```
class PointState(enum.Enum):
    # 障碍物
    BARRIER = 'black'
    # 未使用
    UNUSED = 'white'
    # 在open list的方格
    OPEN = 'green'
    # 在close list的方格
    CLOSED = 'gray'
    # 路径
    PATH = 'red'
    # 起点
    START = 'yellow'
    # 终点
    TARGET = 'purple'
```

5.2.2 鼠标键盘控制

实验中通过鼠标控制网格的状态：左键设置障碍物、中键设置起点、右键设置终点。并通过 n 键控制开始寻找路径，c 键清空网格。程序中利用 bind 函数进行将鼠标键盘和函数进行关联，代码如下：

```
self.canvas.bind('<Button-1>', self.setBarrier) # 鼠标左键设置障碍物
self.canvas.bind('<Button-2>', self.setStart) # 鼠标中键设置起点
self.canvas.bind('<Button-3>', self.setTarget) # 鼠标右键设置终点

self.canvas.bind('<KeyPress-c>', self.cleanMap) # 键盘c键清空网格
self.canvas.bind('<KeyPress-n>', self.navigation) # 键盘n键开始寻找路径
self.canvas.focus_set()

def setBarrier(self, event):
    x = int((event.x + 3) / self.size)
    y = int((event.y + 3) / self.size)
    if x < self.width and y < self.height:
        if self.points[x][y].state == PointState.BARRIER.value: # 重复点击
            self.changeState(self.points[x][y], PointState.UNUSED.value)
        else:
            self.changeState(self.points[x][y], PointState.BARRIER.value)
```

```

def setStart(self, event):
    x = int((event.x + 3) / self.size)
    y = int((event.y + 3) / self.size)
    self.start = (x, y)
    if x < self.width and y < self.height:
        if self.points[x][y].state == PointState.START.value: # 重复点击
            self.changeState(self.points[x][y], PointState.UNUSED.value)
        else:
            self.changeState(self.points[x][y], PointState.START.value)


def setTarget(self, event):
    x = int((event.x + 3) / self.size)
    y = int((event.y + 3) / self.size)
    self.end = (x, y)
    if x < self.width and y < self.height:
        if self.points[x][y].state == PointState.TARGET.value: # 重复点击
            self.changeState(self.points[x][y], PointState.UNUSED.value)
        else:
            self.changeState(self.points[x][y], PointState.TARGET.value)
```

5.2.3 提示窗口

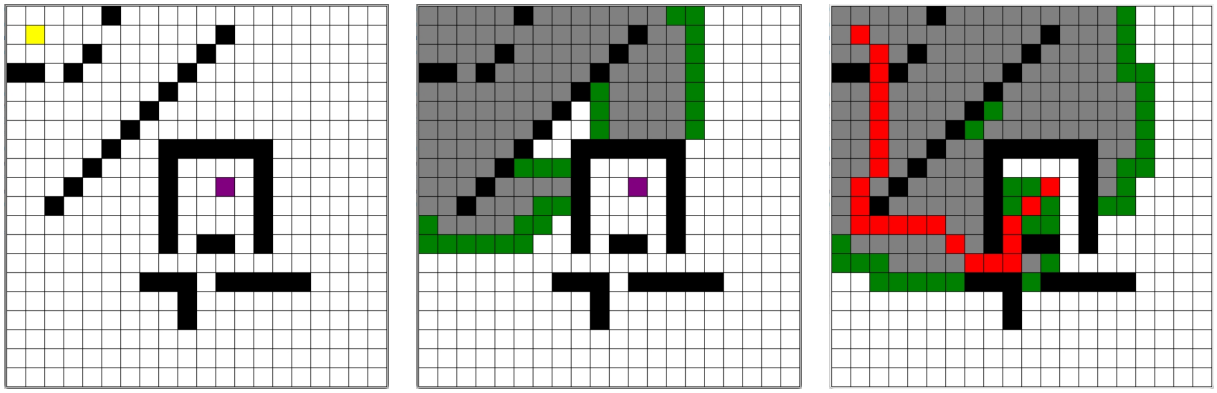
当 open list 的大小为 0 时，代表搜索不到从起点到终点的路径，此时弹出窗口提示没有找到路径。

```
if len(open_list) == 0:
    print('Unreachable!')
    # 弹出一个消息框来提示没有路径
    messagebox.showinfo("提示", "没有找到路径")
    break
```

6 实验结果

6.1 存在可行路径的情况

使用鼠标左键设置障碍物（黑色），鼠标中键设置起点（黄色），鼠标右键设置终点（紫色），点击键盘 n 键开始搜索路径。搜索过程中，灰色的网格代表位于 close list 中，绿色的网格代表位于 open list 中。最后红色的路径为搜索得到的最短路径。图6.3为一次测试结果。

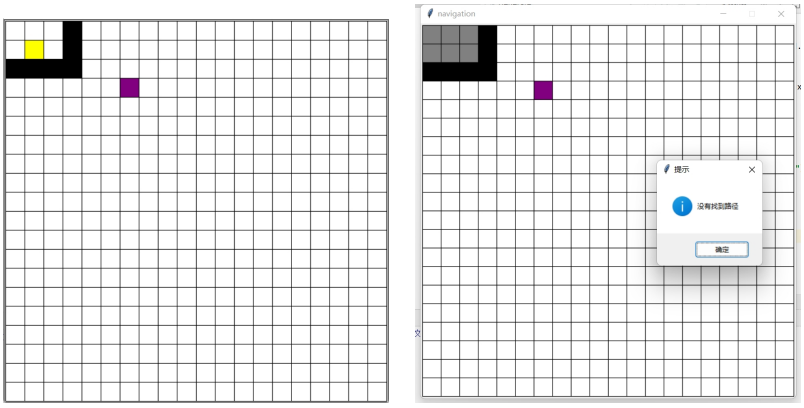


(a) 设置起点、终点和障碍物 (b) 搜索过程 (c) 搜索结果

图 6.3: 测试一

6.2 不存在可行路径的情况

图6.4测试了一种无可行路径的情况，此时搜索完成后，将弹出一个窗口提示“没有找到路径”。

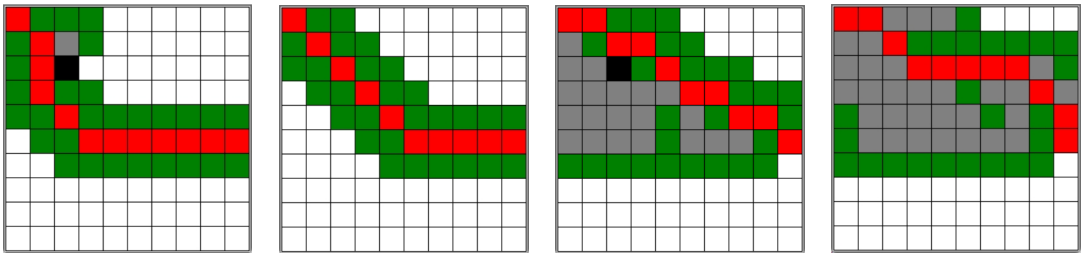


(a) 设置起点、终点和障碍物 (b) 弹出提示

图 6.4: 测试二

6.3 不同距离度量的影响

实验中还测试了欧氏距离和曼哈顿距离两种距离度量对路径搜索的影响。图6.5展示了两种距离度量的区别，可以看出：使用欧氏距离时，路径搜索倾向于**先以对角线的方式接近终点**，而使用曼哈顿距离时**并没有这种倾向**。这是因为欧氏距离中，对角线上的 G 值小于上下左右的 G 值，所以对角线上的代价较小；曼哈顿距离的八个方向上 G 值均相等，代价没有明显差别，不会倾向以对角线的方式接近终点。



(a) 欧氏距离 1 (b) 欧氏距离 2 (c) 曼哈顿距离 1 (d) 曼哈顿距离 2

图 6.5: 不同距离度量的区别

7 分析总结

通过本次实验，我理解无信息搜索与有信息搜索的区别，对 A* 算法的原理和实现有了更加深刻的理解，编写了 A* 算法的 Python 程序，利用 Tkinter 对路径搜索的过程和结果进行了可视化，比较了欧氏距离和曼哈顿距离作为距离度量对于路径搜索结果的影响。

实验一中所用到的回溯算法属于无信息搜索，A* 算法属于有信息搜索。两者的主要区别在于：

- 信息利用：无信息搜索不利用问题的特定信息，只关注搜索的结构，而启发式搜索利用了启发式函数提供的问题特定信息，以更智能地导向搜索过程。
- 搜索效率：由于启发式搜索使用了额外的信息，通常比无信息搜索更快地找到解决方案。启发式搜索在每一步都会尝试朝着估计代价更低的方向前进，因此更有可能在更少的步骤内找到最佳解决方案。

实验的不足之处在于：

- 没有将提供单步搜索的功能，详细展示路径搜索的过程。
- 没有展示 open list 中每个网格当前的 G 值、H 值和 F 值，更清晰地体现启发式算法的原理。

附录 A 代码

```python
from tkinter import *
import enum
import heapq
import time
import _thread
from tkinter import messagebox

class PointState(enum.Enum):
    # 障碍物
    BARRIER = 'black'
    # 未使用
    UNUSED = 'white'
    # 在open list的方格
    OPEN = 'green'
    # 在close list的方格
    CLOSED = 'gray'
    # 路径
    PATH = 'red'
    # 起点
    START = 'yellow'
    # 终点
    TARGET = 'purple'


class MiniMap:
    class Point:
        def __init__(self, x, y, f, g, father, state, rectangle):
            # x坐标
            self.x = x
            # y坐标
            self.y = y
            # f = g + h
            self.f = f
            # 从寻路起点到这个点的代价
            self.g = g
            # 父节点
            self.father = father
            # 当前点状态
            self.state = state
            # 当前点对应画布上的矩形
```

```
            self.rectangle = rectangle

        def __lt__(self, other):
            if self.f < other.f:
                return True
            else:
                return False

    def __init__(self, *args):
        # 高
        self.height = args[0]
        # 宽
        self.width = args[1]
        # 方格尺寸
        self.size = args[2]
        # 起点，初始为 (0,0)
        self.start = (0, 0)
        # 终点，初始为 (19,19)
        self.end = (19, 19)
        # 每次绘制的延迟时间
        self.delay = args[3]

        self.root = Tk()
        self.root.title('A star')
        self.canvas = Canvas(self.root, width=self.width * self.size + 3,
            height=self.height * self.size + 3)
        # 生成方格集合
        self.points = self.generatePoints()
        # 生成网格
        self.generateMesh()

        self.canvas.bind('<Button-1>', self.setBarrier) # 鼠标左键设置障碍物
        self.canvas.bind('<Button-2>', self.setStart) # 鼠标中键设置起点
        self.canvas.bind('<Button-3>', self.setTarget) # 鼠标右键设置终点

        self.canvas.bind('<KeyPress-c>', self.cleanMap) # 键盘c键清空网格
        self.canvas.bind('<KeyPress-n>', self.navigation) # 键盘n键开始寻找路径
        self.canvas.focus_set()

        self.canvas.pack(side=TOP, expand=YES, fill=BOTH)
        self.root.resizable(False, False)
        self.root.mainloop()
```

```
82
83      def generatePoints(self):
84          points = [[self.Point(x, y, 0, 0, None, PointState.UNUSED.value,
85                                self.canvas.create_rectangle((x * self.size + 3, y *
                                  self.size + 3),
86                                                           ((x + 1) * self.size + 3, (y +
                                                             1) * self.size + 3),
87                                                           fill=PointState.UNUSED.value))
                                                             for y in range(self.height)]
88                    for x in range(self.width)]
89          return points
90
91      def generateMesh(self):
92          for i in range(self.height + 1):
93              self.canvas.create_line((3, i * self.size + 3), (self.width * self.size +
                    3, i * self.size + 3))
94          for i in range(self.width + 1):
95              self.canvas.create_line((i * self.size + 3, 3), (i * self.size + 3,
                    self.height * self.size + 3))
96
97      def setBarrier(self, event):
98          x = int((event.x + 3) / self.size)
99          y = int((event.y + 3) / self.size)
100         if x < self.width and y < self.height:
101             if self.points[x][y].state == PointState.BARRIER.value: # 重复点击
102                 self.changeState(self.points[x][y], PointState.UNUSED.value)
103             else:
104                 self.changeState(self.points[x][y], PointState.BARRIER.value)
105
106     def setStart(self, event):
107         x = int((event.x + 3) / self.size)
108         y = int((event.y + 3) / self.size)
109         self.start = (x, y)
110         if x < self.width and y < self.height:
111             if self.points[x][y].state == PointState.START.value: # 重复点击
112                 self.changeState(self.points[x][y], PointState.UNUSED.value)
113             else:
114                 self.changeState(self.points[x][y], PointState.START.value)
115
116
117     def setTarget(self, event):
118         x = int((event.x + 3) / self.size)
```

```
        y = int((event.y + 3) / self.size)
        self.end = (x, y)
        if x < self.width and y < self.height:
            if self.points[x][y].state == PointState.TARGET.value: # 重复点击
                self.changeState(self.points[x][y], PointState.UNUSED.value)
            else:
                self.changeState(self.points[x][y], PointState.TARGET.value)

    def cleanMap(self, event):
        for i in range(self.width):
            for j in range(self.height):
                self.changeState(self.points[i][j], PointState.UNUSED.value)

    def navigation(self, event):
        _thread.start_new_thread(self.generatePath, (self.start, self.end))

    def changeState(self, point, state):
        point.state = state
        self.canvas.itemconfig(point.rectangle, fill=state)

    def generatePath(self, start, end):
        xStart = start[0]
        yStart = start[1]
        xEnd = end[0]
        yEnd = end[1]

        heap = []

        close_list = set()
        open_list = set()

        heapq.heappush(heap, self.points[xStart][yStart])
        open_list.add((xStart, yStart))
        # 寻路循环
        while 1:
            # 从open list中取出代价最小点
            pMin = heapq.heappop(heap)
            open_list.remove((pMin.x, pMin.y))
            # 将这个点放入close list中
            close_list.add((pMin.x, pMin.y))
            self.changeState(self.points[pMin.x][pMin.y], PointState.CLOSED.value)
            # 遍历八个方向
```

```
            for i in range(-1, 2):
                for j in range(-1, 2):
                    if i == 0 and j == 0:
                        continue
                    # 当前要判断的点的坐标
                    xCur = pMin.x + i
                    yCur = pMin.y + j
                    # 如果这个点越界则跳过
                    if xCur >= self.width or xCur < 0 or yCur >= self.height or yCur <
                        0:
                        continue
                    pCur = self.points[xCur][yCur]
                    isCorner = (i != 0 and j != 0)
                    if isCorner:
                        #
                            如果将要判断的斜角方向被阻挡则跳过（如将要判断东南方向，若东方或南方被阻挡，
                        if self.points[xCur][pMin.y].state == PointState.BARRIER.value
                            or \
                                self.points[pMin.x][yCur].state ==
                                    PointState.BARRIER.value:
                            continue

                    # 欧氏距离
                    if (xCur, yCur) not in close_list and
                        self.points[xCur][yCur].state != PointState.BARRIER.value:
                        # 在open list中
                        if (xCur, yCur) in open_list:
                            # 起点到pMin再到pCur的代价比起点到pCur的代价小
                            if ((14 if isCorner else 10) + pMin.g) < pCur.g:
                                pCur.g = pMin.g + (14 if isCorner else 10)
                                pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd -
                                    yCur)) # 更新pCur的代价
                                pCur.father = pMin # 将pMin设置为pCur的父节点
                        # 不在open list中
                        else:
                            pCur.g = pMin.g + (14 if isCorner else 10)
                            pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd - yCur))
                            pCur.father = pMin # 将pMin设置为pCur的父节点
                            self.changeState(pCur, PointState.OPEN.value)
                            # 将这个点加入open list
                            open_list.add((xCur, yCur))
                            heapq.heappush(heap, pCur)
```

```
197
198                 # # 曼哈顿距离
199                 # if (xCur, yCur) not in close_list and
                        self.points[xCur][yCur].state != PointState.BARRIER.value:
200                 #     # 在open list中
201                 #     if (xCur, yCur) in open_list:
202                 #         # 起点到pMin再到pCur的代价比起点到pCur的代价小
203                 #         if ((20 if i*j != 0 else 10) + pMin.g) < pCur.g:
204                 #             pCur.g = pMin.g + (20 if i*j != 0 else 10)
205                 #             pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd -
                        yCur)) # 更新pCur的代价
206                 #             pCur.father = pMin # 将pMin设置为pCur的父节点
207                 #     # 不在open list中
208                 #     else:
209                 #         pCur.g = pMin.g + (20 if i*j != 0 else 10)
210                 #         pCur.f = pCur.g + 10 * (abs(xEnd - xCur) + abs(yEnd -
                        yCur))
211                 #         pCur.father = pMin # 将pMin设置为pCur的父节点
212                 #         self.changeState(pCur, PointState.OPEN.value)
213                 #         # 将这个点加入open list
214                 #         open_list.add((xCur, yCur))
215                 #         heapq.heappush(heap, pCur)
216
217             # 检测是否寻路完成
218             if (xEnd, yEnd) in open_list:
219                 pNext = self.points[xEnd][yEnd]
220                 self.changeState(pNext, PointState.PATH.value)
221                 while pNext.father:
222                     pNext = pNext.father
223                     self.changeState(self.points[pNext.x][pNext.y],
                            PointState.PATH.value)
224                 break
225             # 如果寻路不完成但open list长度为0，则没有可达路径
226                 if len(open_list) == 0:
227                     print('Unreachable!')
228                     # 弹出一个消息框来提示没有路径
229                     messagebox.showinfo("提示", "没有找到路径")
230                     break
231
232             # 等待绘制
233             time.sleep(self.delay)
234
```

```
235
236 # 参数为地图高、宽、方格尺寸、延迟时间
237 demo = MiniMap(20, 20, 30, 0.05)
```