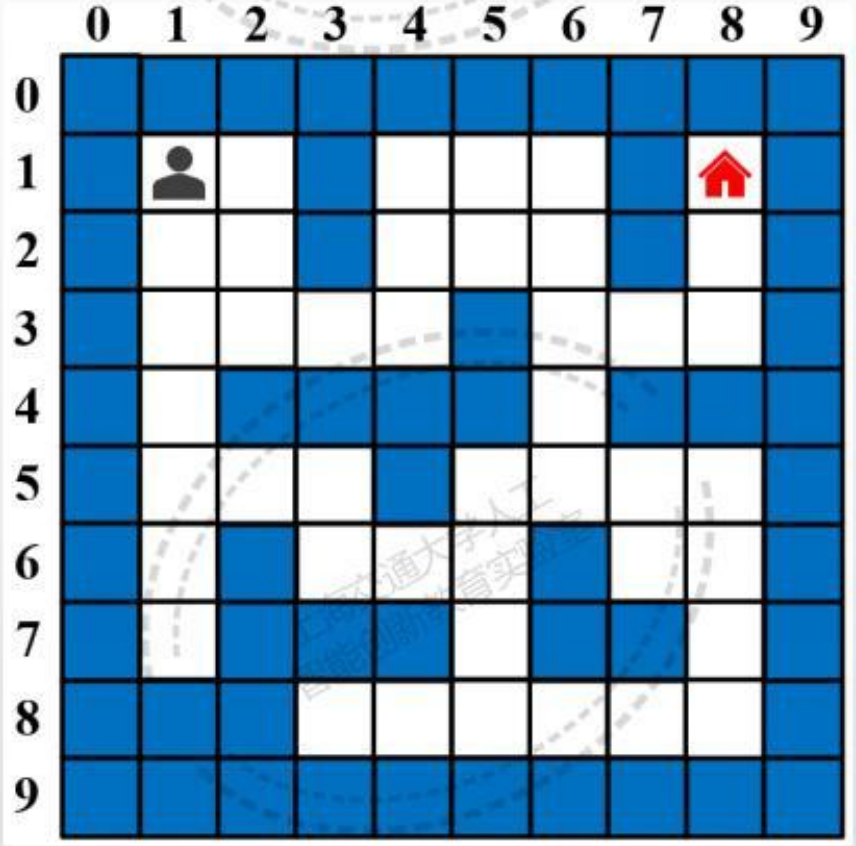
人工智能基础与进阶

搜索算法实践

上海交通大学

目录 content

第一节 定义地图以及搜索方向



定义地图以及搜索方向

```
#迷宫地图,(1,1)起始点,(1,8)终点
maze = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 0, 0, 0, 1, 0, 0, 0, 0, 1],
        [1, 0, 1, 0, 0, 0, 1, 0, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
        [1, 1, 1, 0, 0, 0, 0, 0, 0, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

```
#BFS搜索路经结果
result = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
          [1, 0, 0, 1, 0, 0, 0, 1, 0, 1],
          [1, 0, 0, 1, 0, 0, 0, 1, 0, 1],
          [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
          [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
          [1, 0, 0, 0, 1, 0, 0, 0, 0, 1],
          [1, 0, 1, 0, 0, 0, 1, 0, 0, 1],
          [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
          [1, 1, 1, 0, 0, 0, 0, 0, 0, 1],
          [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
```

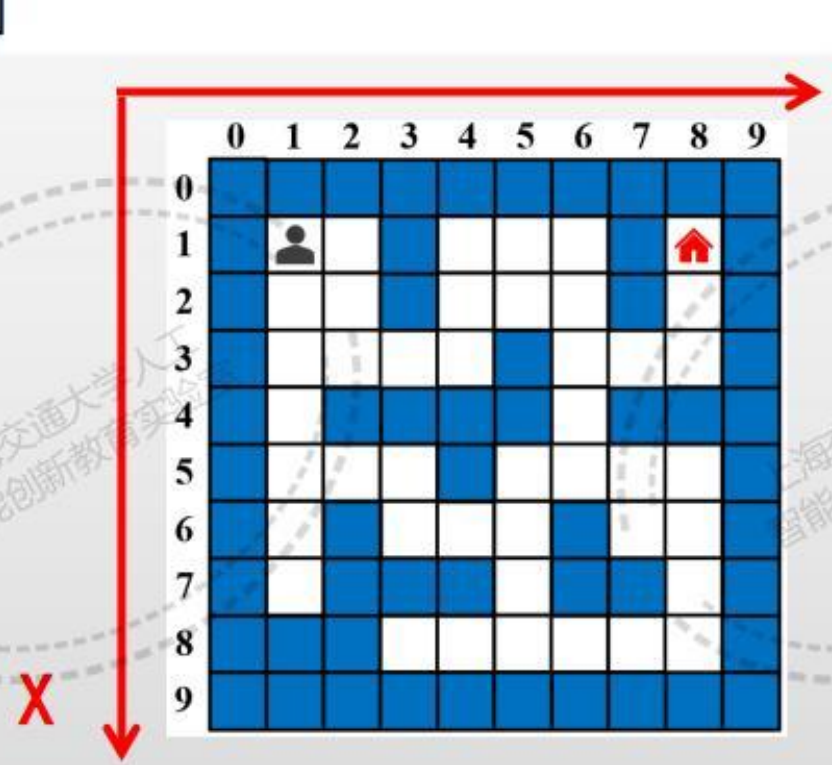
使用**二维数组**存放地图节点

maze表示原始地图

result用于记录搜索路径

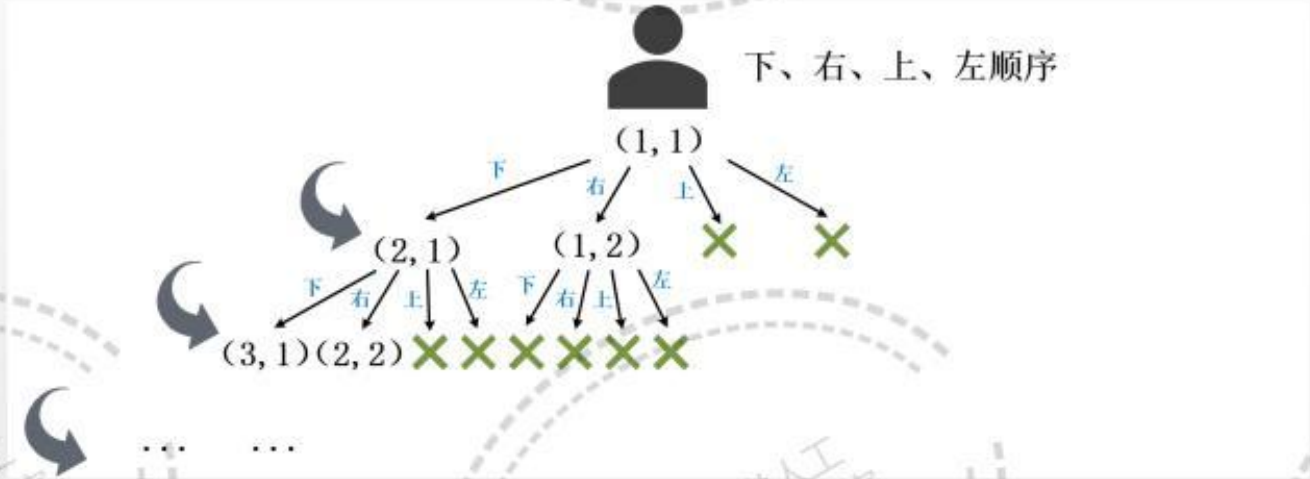
定义地图以及搜索方向

```
#搜索顺序：下右上左
directions = [
    lambda x, y: (x + 1, y),  # 下
    lambda x, y: (x, y + 1),  # 右
    lambda x, y: (x - 1, y),  # 上
    lambda x, y: (x, y - 1),  # 左
]
```

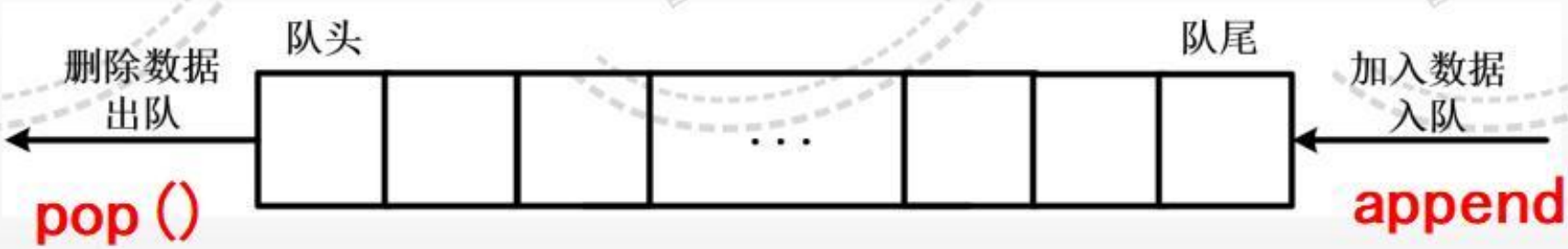


使用匿名函数操作节点的搜索方向，通过列表索引值选择匿名函数。

第二节 广度优先搜索



广度优先搜索-队列结构说明



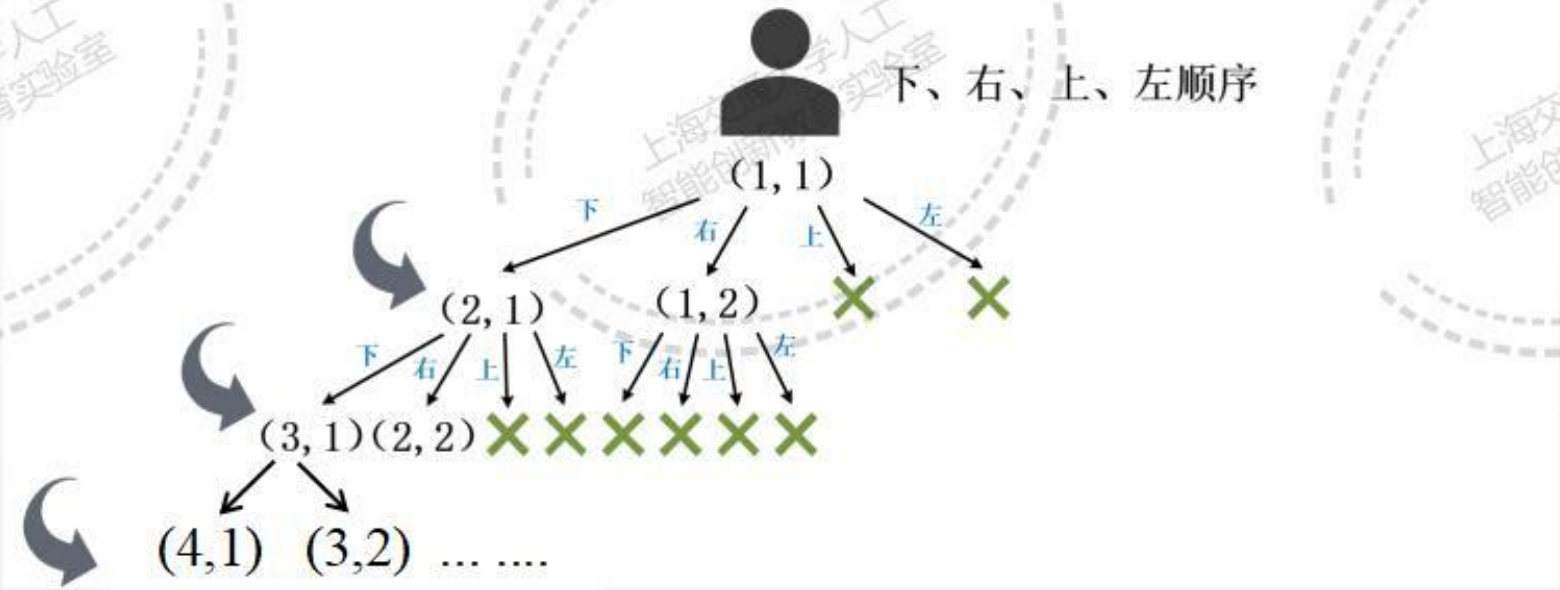
q = Queue( ) 创建一个队列结构。

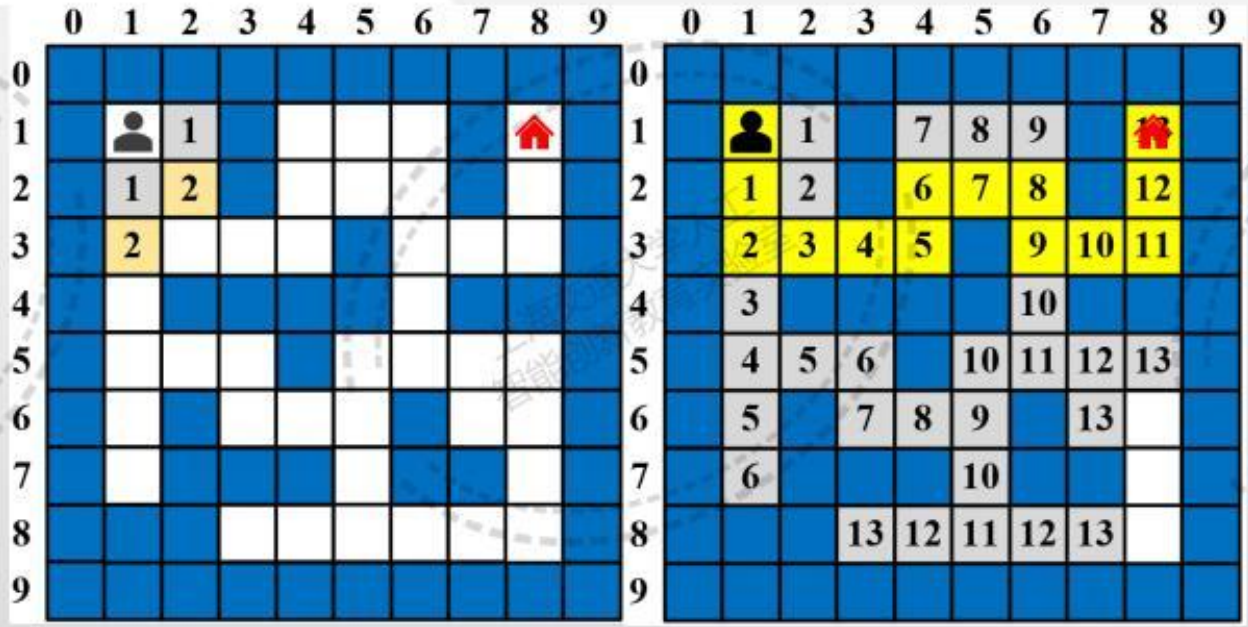
pop ( ) 方法从队列左侧拿出数据。

append( )方法从队列右侧加入数据。

※符合先进先出原则。

广度优先搜索-核心概念分析

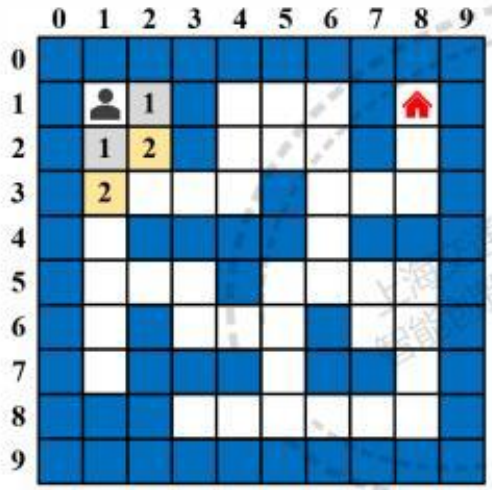




广度优先搜索-代码讲解

```
# 广度优先搜索使用队列结构
def BFS_queue(x1, y1, x2, y2):  # 起始点(x1,y1)，终点(x2,y2)
    q = Queue()  # q为一个队列
    path = []  # path列表存放访问过的节点
    q.enqueue((x1, y1, -1))  # 将起始点放入队列内
    maze[x1][y1] = '*'  # 将起始点设为'*'符号
    while q.size() > 0:  # 当队列结构有节点时开始搜索
        cur_node = q.dequeue()  # 先进先出原则取节点（当前节点）
        path.append(cur_node)  # 将当前节点放入path列表内
        if cur_node[:2] == (x2, y2):  # 如果当前节点为终点则搜索成功
            resultpath = []  # resultpath列表为最终搜索路径的节点
            i = len(path) - 1  # 索引值从0开始，因此需要-1
            while i >= 0:
                resultpath.append(path[i][:2])  # 依序取出节点
                i = path[i][2]  # 将当前节点的索引值更新为其父节点
            resultpath.reverse()  # 颠倒顺序为从起始点到终点的路径
            resultroute = []
            for v in resultpath:  # 将最终搜索路径的节点设为'*'符号
                result[v[0]][v[1]] = '*'
                k = 9 - v[1]
                resultroute.append((k, v[0]))
            # print(path)
            return resultroute
        for d in directions:  # 循环四个方向搜索
            next_x, next_y = d(cur_node[0], cur_node[1])  # 更新为移动过后的节点
            if maze[next_x][next_y] == 0:  # 如果此节点可以前进
                q.enqueue((next_x, next_y, len(path) - 1))  # 将此节点放入队列，且设定当前节点为其父节点
                maze[next_x][next_y] = 2  # 将此节点设为2表示已走过

    print('无路可走')
    return False  # 搜索失败
```



path = [(1, 1, −1), (2, 1, 0), (1, 2, 0), (3, 1, 1), (2, 2, 1), (4, 1, 3), (3, 2, 3) . . .]

```
#搜索顺序：下右上左
directions = [
    lambda x, y: (x + 1, y),  # 下
    lambda x, y: (x, y + 1),  # 右
    lambda x, y: (x - 1, y),  # 上
    lambda x, y: (x, y - 1),  # 左
]
```

广度优先搜索-代码讲解

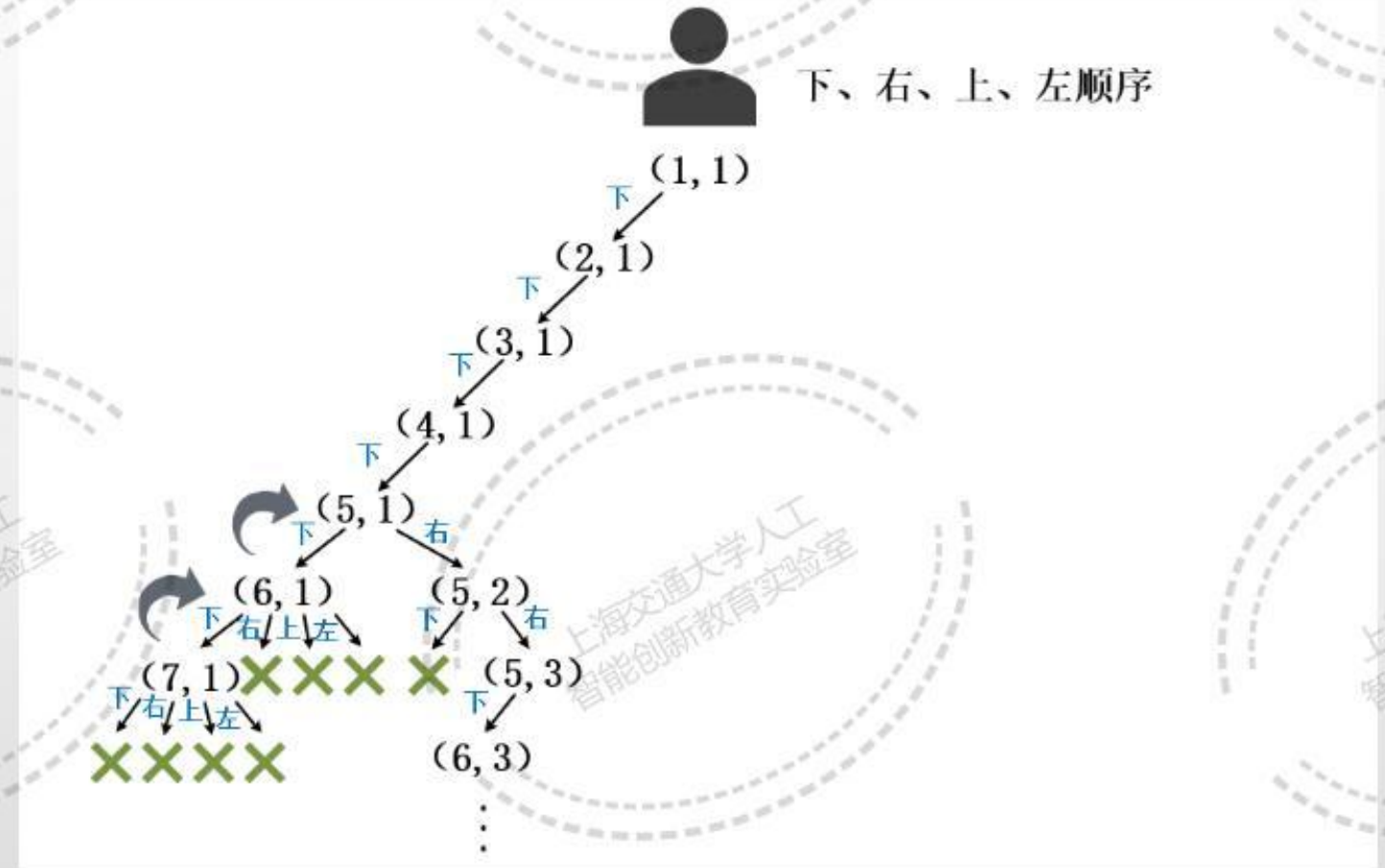
主程序调用广度优先搜索

```
if __name__ == '__main__':
    print('-----------原始地图---------')
    for k in maze:    #输出地图
        for v in k:
            print(v, end="  ")
        print("")
    print("")
    BFS_queue(1, 1, 1, 8)  #调用DFS搜索函数
    print('-------BFS搜索后的结果------')
    for k in result:    #输出最终路径
        for v in k:
            print(v, end="  ")
        print("")
```

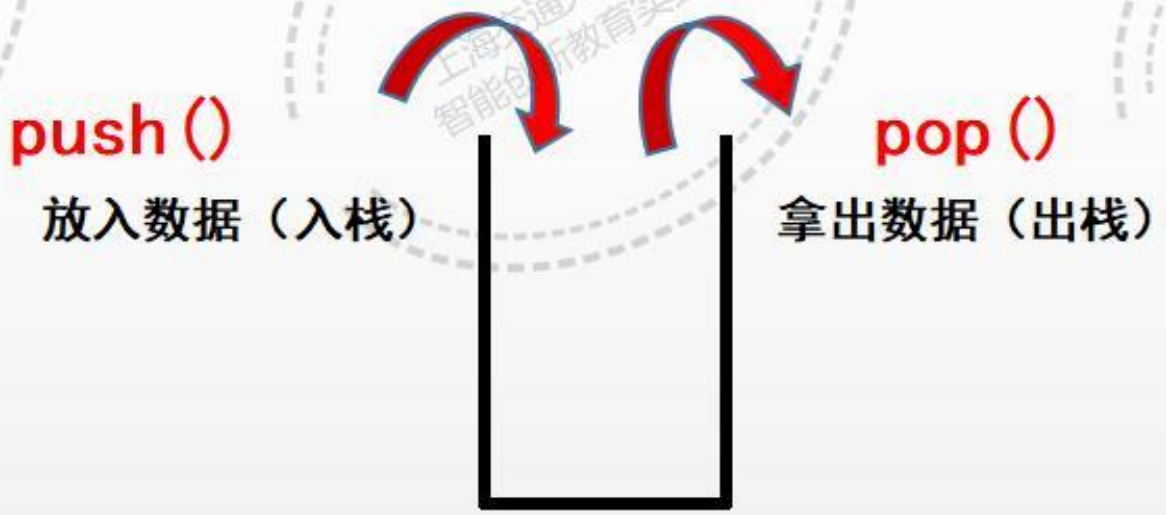
```
-----------原始地图---------
1  1  1  1  1  1  1  1  1  1
1  0  0  1  0  0  0  1  0  1
1  0  0  1  0  0  0  1  0  1
1  0  0  0  0  1  0  0  0  1
1  0  1  1  1  1  0  1  1  1
1  0  0  0  1  0  0  0  0  1
1  0  1  0  0  0  1  0  0  1
1  0  1  1  1  0  1  1  0  1
1  1  1  0  0  0  0  0  0  1
1  1  1  1  1  1  1  1  1  1

-------BFS搜索后的结果------
1  1  1  1  1  1  1  1  1  1
1  *  0  1  0  0  0  1  *  1
1  *  0  1  *  *  *  1  *  1
1  *  *  *  *  1  *  *  *  1
1  0  1  1  1  1  0  1  1  1
1  0  0  0  1  0  0  0  0  1
1  0  1  0  0  0  1  0  0  1
1  0  1  1  1  0  1  1  0  1
1  1  1  0  0  0  0  0  0  1
1  1  1  1  1  1  1  1  1  1
```

第三节 深度优先搜索



深度优先搜索-栈结构说明

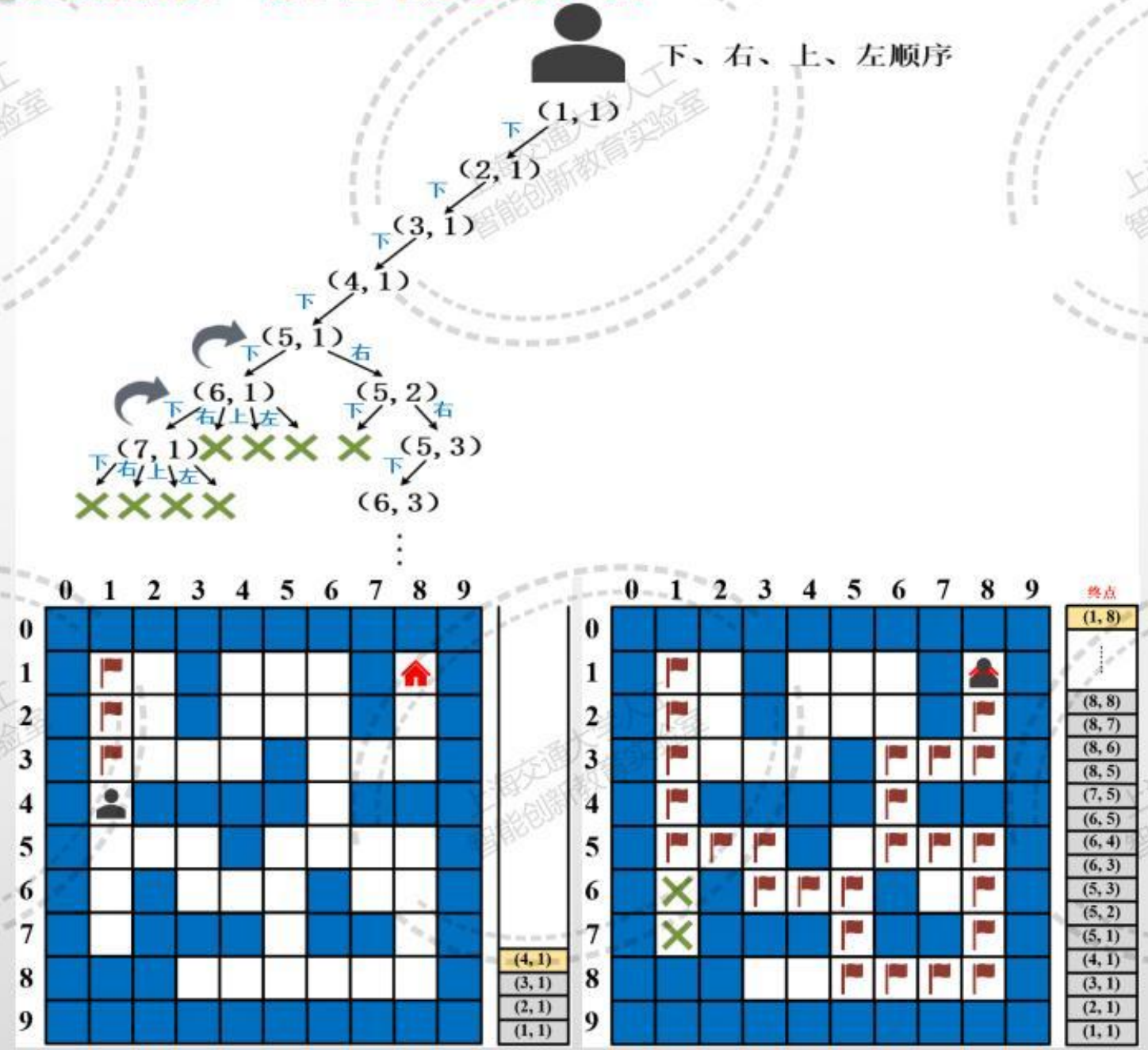


pop( ) 方法出栈数据。

push( ) 方法入栈数据。

※符合先进后出原则。

深度优先搜索-核心概念分析



深度优先搜索-代码讲解

```
# 深度优先搜索使用栈结构
def DFS_stack(x1, y1, x2, y2):                    #起始点(x1,y1)，终点(x2,y2)
    a = Stack()                                   #栈结构存放搜索路径的节点
    a.push((x1, y1))                              #先进后出
    while a.size() > 0:                           #当栈结构有节点时开始搜索
        cur_node = a.stack[-1]                    #当前节点为栈结构最后一个节点
        if cur_node == (x2, y2):                  #如果当前节点为终点则搜索成功
            resultroute = []
            for v in a.stack:                     #循环得到栈结构存放的节点
                result[v[0]][v[1]] = '*'          #将其设定为'*'符号
                k = 9 - v[1]
                resultroute.append((k, v[0]))
            #print(resultroute)
            return resultroute
        for d in directions:                      #循环四个方向搜索
            next_x, next_y = d(*cur_node)         #更新为移动过后的节点
            if maze[next_x][next_y] == 0:         #如果此节点可以前进
                a.push((next_x, next_y))          #将此节点入栈
                maze[next_x][next_y] = 2          #将此节点设为2表示已走过
                break                             #照顺序搜索，某一方向成功就跳出循环，继续搜索下一个节点
        else:
            a.pop()  #如果四个方向搜索失败，则将此节点出栈
    print('无路可走')
    return False  #搜索失败
```

深度优先搜索-代码讲解

主程序调用深度优先搜索

```
if __name__ == '__main__':
    print('-----------原始地图---------')
    for k in maze:    #输出地图
        for v in k:
            print(v, end="  ")
        print("")
    print("")
    DFS_stack(1, 1, 1, 8)  #调用DFS搜索函数
    print('-------DFS搜索后的结果------')
    for k in result:    #输出最终路径
        for v in k:
            print(v, end="  ")
        print("")
```

```
----------原始地图---------
1  1  1  1  1  1  1  1  1  1
1  0  0  1  0  0  0  1  0  1
1  0  0  1  0  0  0  1  0  1
1  0  0  0  0  1  0  0  0  1
1  0  1  1  1  1  0  1  1  1
1  0  0  0  1  0  0  0  0  1
1  0  1  0  0  0  1  0  0  1
1  0  1  1  1  0  1  1  0  1
1  1  1  0  0  0  0  0  0  1
1  1  1  1  1  1  1  1  1  1

-------DFS搜索后的结果------
1  1  1  1  1  1  1  1  1  1
1  *  0  1  0  0  0  1  *  1
1  *  0  1  0  0  0  1  *  1
1  *  0  0  0  1  *  *  *  1
1  *  1  1  1  1  *  1  1  1
1  *  *  *  1  0  *  *  *  1
1  0  1  *  *  *  1  0  *  1
1  0  1  1  1  *  1  1  *  1
1  1  1  0  0  *  *  *  *  1
1  1  1  1  1  1  1  1  1  1
```

第四节 A* 搜索算法

“不再只是采取固定的搜索方向（盲搜索），而是每次都选择代价最小的节点（启发式搜索）作为搜索的下一步。”

A*搜索-核心概念分析

$$f = g + h$$

总代价　实际代价　估计代价

开放列表

从开放列表中选择代价最小的节点作为搜索的下一步。

封闭列表

封闭列表存放已访问过的节点，不需再次检查。

A*搜索-代码讲解

建立坐标点(Point)以及节点(Node)两个类

```
class Point:
    #建立坐标点
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

```
class Node:
    #建立节点
    def __init__(self, point, g = 0, h = 0):
        self.point = point      #节点
        self.father = None      #父节点
        self.g = g              #g值
        self.h = h              #h值
    def manhattan(self, endNode):
        #H值计算：曼哈顿距离
        self.h = (abs(endNode.point.x - self.point.x) + abs(endNode.point.y - self.point.y))*10
    def set_G(self, g):
        #设定G值
        self.g = g
    def set_Father(self, node):
        #设定父节点
        self.father = node
```

A*搜索-代码讲解

建立地图类(maze)的属性

```
class maze:
    def __init__(self):
        #迷宫地图,(1,1)起始点,(1,8)终点
        self.maze = [[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                     [1, 0, 0, 1, 0, 0, 0, 1, 0, 1],
                     [1, 0, 0, 1, 0, 0, 0, 1, 0, 1],
                     [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
                     [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
                     [1, 0, 0, 0, 1, 0, 0, 0, 0, 1],
                     [1, 0, 1, 0, 0, 0, 1, 0, 0, 1],
                     [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
                     [1, 1, 1, 0, 0, 0, 0, 0, 0, 1],
                     [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
        self.w = 10   #地图的宽
        self.h = 10   #地图的高
```

A*搜索-代码讲解

建立地图类(maze)的方法

```
def show_maze(self):
    #显示迷宫地图
    for k in self.maze:
        for v in k:
            print(v, end="  ")
        print("")
    return
def set_maze(self, point):
    #将节点设为'*'符号
    self.maze[point.x][point.y] = '*'
    return
def is_pass(self, point):
    #此节点是否超出界限
    if point.x < 0 or point.x > self.h - 1 or point.y < 0 or point.y > self.w - 1:
        return False
    #此节点是否可以通过
    if self.maze[point.x][point.y] == 0:
        return True
```

A*搜索-代码讲解

建立A_star类及其相关步骤方法和属性。

```
class A_star:
    def __init__(self, maze, startNode, endNode):
        #开放列表
        self.openList = []
        #封闭列表
        self.closeList = []
        #迷宫地图
        self.maze = maze
        #起始点
        self.startNode = startNode
        #终点
        self.endNode = endNode
        #当前节点
        self.currentNode = startNode
        #最终搜索路径
        self.pathlist = []
```

A*搜索-代码讲解

建立A_star类及其相关步骤方法和属性。

```
def Node_In_Openlist(self,node):
    #节点是否在开放列表内
    for nodeTmp in self.openList:
        if nodeTmp.point.x == node.point.x \
        and nodeTmp.point.y == node.point.y:
            return True
    return False

def Node_In_Closelist(self,node):
    #节点是否在封闭列表内
    for nodeTmp in self.closeList:
        if nodeTmp.point.x == node.point.x \
        and nodeTmp.point.y == node.point.y:
            return True
    return False
```

A*搜索-代码讲解

建立A_star类及其相关步骤方法和属性。

```
def Min_F_Node(self):
    #开放列表内最小F值的节点
    nodeTemp = self.openList[0]
    for node in self.openList:
        if node.g + node.h < nodeTemp.g + nodeTemp.h:
            nodeTemp = node
    return nodeTemp
```

```
def Node_From_OpenList(self,node):
    #相邻节点是否已经在开放列表内，是则返回相邻节点
    for nodeTmp in self.openList:
        if nodeTmp.point.x == node.point.x \
        and nodeTmp.point.y == node.point.y:
            return nodeTmp
    return None
```

A*搜索-代码讲解

建立A_star类及其相关步骤方法和属性。

```
def search_node(self,node):
    #搜索相邻节点 #判断此相邻节点是否可以通行
    if self.maze.is_pass(node.point) != True:  return
    #判断此相邻节点是否在封闭列表内
    if self.Node_In_Closelist(node):  return
    #此相邻节点的G值计算
    if abs(node.point.x - self.currentNode.point.x) == 1 and abs(node.point.y - self.currentNode.point.y) == 1:
        gTemp = 14
    else:
        gTemp = 10
    #如果不在开放列表内，就将此相邻节点放入开放列表内
    if self.Node_In_Openlist(node) == False:
        self.openList.append(node)
        node.set_G(self.currentNode.g + gTemp)
        #此相邻节点的H值计算
        node.manhattan(self.endNode)
        node.father = self.currentNode
    #如果已经在开放列表内，判断假设当前节点为父节点时的G值是否会更小，如果更小，就重新计算此相邻节点的G值，并改变父节点
    else:
        nodeTmp = self.Node_From_OpenList(node)
        if self.currentNode.g + gTemp < nodeTmp.g:
            nodeTmp.g = self.currentNode.g + gTemp
            nodeTmp.father = self.currentNode
    return
```

A*搜索-代码讲解

建立A_star类及其相关步骤方法和属性。

```
def search(self):
    #搜索周围八个方向
    self.search_node(Node(Point(self.currentNode.point.x - 1, self.currentNode.point.y - 1)))
    self.search_node(Node(Point(self.currentNode.point.x - 1, self.currentNode.point.y)))
    self.search_node(Node(Point(self.currentNode.point.x - 1, self.currentNode.point.y + 1)))
    self.search_node(Node(Point(self.currentNode.point.x, self.currentNode.point.y - 1)))
    self.search_node(Node(Point(self.currentNode.point.x, self.currentNode.point.y)))
    self.search_node(Node(Point(self.currentNode.point.x, self.currentNode.point.y + 1)))
    self.search_node(Node(Point(self.currentNode.point.x + 1, self.currentNode.point.y - 1)))
    self.search_node(Node(Point(self.currentNode.point.x + 1, self.currentNode.point.y)))
    self.search_node(Node(Point(self.currentNode.point.x + 1, self.currentNode.point.y + 1)))
    return
```

A*搜索-代码讲解

建立A_star类及其相关步骤方法和属性。

```
def start(self):
    #开始搜索
    self.startNode.manhattan(self.endNode)
    self.startNode.set_G(0)
    self.openList.append(self.startNode)
    while True:
        #当前节点放入封闭列表内
        self.closeList.append(self.currentNode)
        #搜索当前节点的八个方向
        self.search()
        #判断开放列表是否为空，如果是则找不到路径
        if len(self.openList) == 0:
            return False
        #判断终点是否在开放列表内，如果是则找到路径
        elif self.endNode_In_OpenList():
            nodeTmp = self.Node_From_OpenList(self.endNode)
            while True:
                #根据终点的父节点依序往回找就是最终的搜索路径
                self.pathlist.append(nodeTmp)
                if nodeTmp.father != None:
                    nodeTmp = nodeTmp.father
                else:
                    return True
        #更新当前节点为开放列表内F值最小的节点
        self.currentNode = self.Min_F_Node()
        #当前节点从开放列表内删除
        self.openList.remove(self.currentNode)
    return True
```

A*搜索-代码讲解

建立A_star类及其相关步骤方法和属性。

```
def set_Map(self):
    #更新迷宫地图，显示最终的搜索路径
    for node in self.pathlist:
        self.maze.set_maze(node.point)
    return
```

A*搜索-代码讲解

主程序调用A*算法

```
import A_star_maze as maze
import A_star
if __name__ == '__main__':
    print('----------原始地图----------')
    #构建地图操作类
    maze_data = maze.maze()
    #显示原始地图
    maze_data.show_maze()
    print("")
    #构建A*算法类
    aStar = A_star.A_star(maze_data, A_star.Node(A_star.Point(1,1)), A_star.Node(A_star.Point(1,8)))
    print('-------A*搜索后的结果-------')
    #开始A*搜索
    if aStar.start():
        #显示最终搜索路径地图
        aStar.set_Map()
        maze_data.show_maze()
    else:
        print("搜索失败")
```

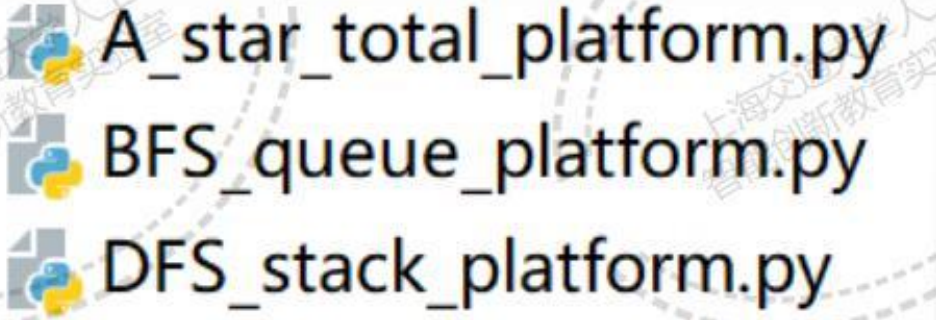
```
----------原始地图----------
1 1 1 1 1 1 1 1 1 1
1 0 0 1 0 0 0 1 0 1
1 0 0 1 0 0 0 1 0 1
1 0 0 0 0 1 0 0 0 1
1 0 1 1 1 1 0 1 1 1
1 0 0 0 1 0 0 0 0 1
1 0 1 0 0 0 1 0 0 1
1 0 1 1 1 0 1 1 0 1
1 1 1 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1
```

```
-------A*搜索后的结果-------
1 1 1 1 1 1 1 1 1 1
1 * 0 1 0 0 0 1 * 1
1 0 * 1 * * * 1 * 1
1 0 0 * 0 1 0 * 0 1
1 0 1 1 1 1 0 1 1 1
1 0 0 0 1 0 0 0 0 1
1 0 1 0 0 0 1 0 0 1
1 0 1 1 1 0 1 1 0 1
1 1 1 0 0 0 0 0 0 1
1 1 1 1 1 1 1 1 1 1
```

第五节 实 验

“依托于仿真平台地图直观感受上述算法。”

三种方法的简单实验



在search文件夹下打开分别打开图示的三个文件，将程序补充完整后运行，观察运行结果。

实验结果分析

1. 三种算法的结果是否一致？

2运行时间有什么区别？

3. 为什么会出现这种结果？

------BFS搜索后结果------

```
1  1  1  1  1  1  1  1  1  1
1  *  0  1  0  0  0  1  0  1
1  *  1  1  0  1  0  0  0  1
1  *  0  0  1  0  0  1  0  1
1  *  *  0  1  1  0  1  0  1
1  1  *  1  1  *  *  *  0  1
1  0  *  *  *  *  1  *  *  1
1  0  1  1  0  0  1  1  *  1
1  1  1  0  0  1  0  0  *  1
1  1  1  1  1  1  1  1  1  1
```

BFS搜索时间:0.1286999999999816毫秒

------DFS搜索后结果------

```
1  1  1  1  1  1  1  1  1  1
1  *  0  1  0  0  0  1  0  1
1  *  1  1  0  1  0  0  0  1
1  *  0  0  1  0  0  1  0  1
1  *  *  0  1  1  0  1  0  1
1  1  *  1  1  *  *  *  0  1
1  0  *  *  *  *  1  *  *  1
1  0  1  1  *  *  1  1  *  1
1  1  1  0  0  1  0  0  *  1
1  1  1  1  1  1  1  1  1  1
```

DFS搜索时间:0.09449999999999736毫秒

---------A*搜索后地图----------

```
1  1  1  1  1  1  1  1  1  1
1  *  0  1  0  0  0  1  0  1
1  *  1  1  0  1  0  0  0  1
1  *  *  0  1  0  0  1  0  1
1  0  *  0  1  1  0  1  0  1
1  1  *  1  1  *  *  *  *  1
1  0  *  *  *  *  1  0  *  1
1  0  1  1  0  0  1  1  *  1
1  1  1  0  0  1  0  0  *  1
1  1  1  1  1  1  1  1  1  1
```

A*搜索时间:0.008899999999997837毫秒

4. 同学们可对地图或起始点位置自行进行修改，观察实验结果。

仿真平台首页

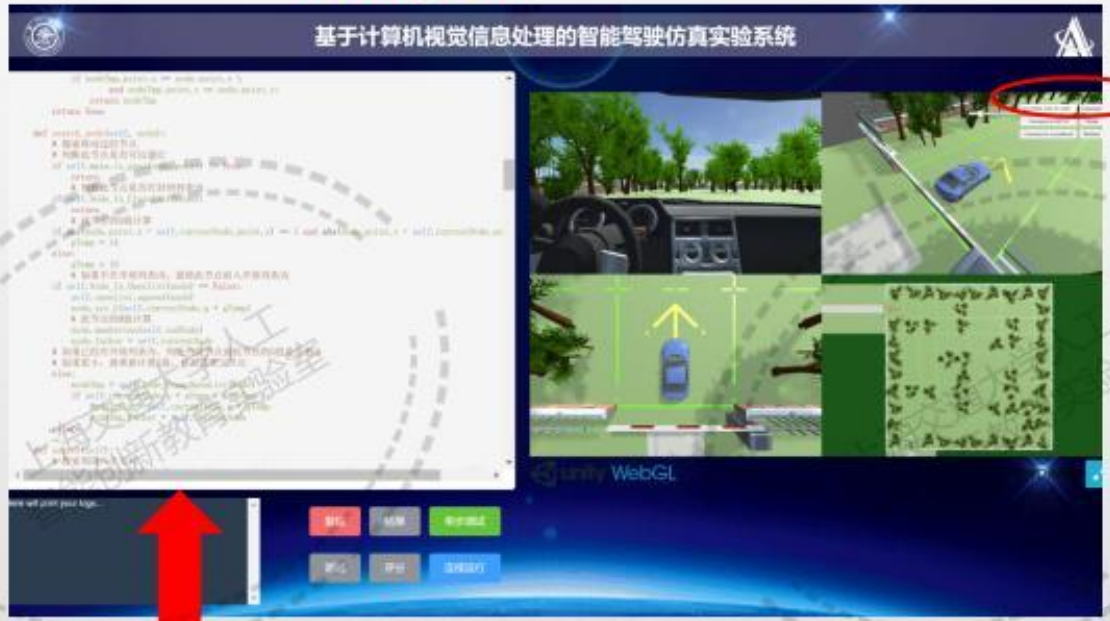
网址：https://ilabs.sjtu.edu.cn/lesson10/#/

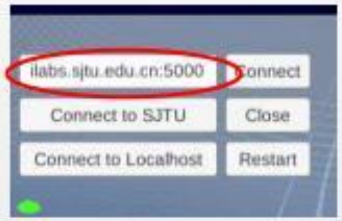
推荐浏览器：Firefox

实验演示

登陆界面

选择相应地图





输入：

wss://ilabs.sjtu.edu.cn:5000/ws

点击Connect

代码区域


谢谢聆听
THANKS FOR YOUR ATTENTION