中山大學
SUN YAT-SEN UNIVERSITY

《人工智能实验》

实验报告

（实验九）

学 院 名 称 ：	数据科学与计算机学院
专业（班级）：	17 计算机科学与技术
学 生 姓 名 ：	仲逊
学　　号	16327143
时　　间 ：	2019　年　11　月　21　日

实 验 ９ ：无信息搜索与启发式搜索

一、 算法原理：

1. 一致代价搜索

当每一步的行动代价都相等时，宽度优先搜索是最优的，因为它总是先扩展深度最浅的未扩展结点。基于这种思想，我们可以找到一个能够有效用于加权图的搜索算法。

一致代价搜索就是基于宽度优先搜索思想的一种算法。它的主要目标是找到能够到达目标节点的具有最低累积成本的路径。一致代价搜索会维护一个优先级队列，访问一个节点的代价越低，队列中给该节点赋予的优先级就越高。我们每次都当前代价最小的节点进行扩展（即对优先级队列做出队操作），因此可以保证无论每一步代价是否一致，都能够找到最优解。如果所有边的代价相同，此时一致代价搜索等价于宽度优先搜索。

2. A*搜索

A*搜索是最常见的启发式搜索算法，它同时使用了启发式函数$h(n)$和从起始状态到当前状态（也就是n状态）的实际代价$g(n)$来预估到目标节点的代价$f(n)$，估值函数$f(n)$的公式如下：

$$f(n) = g(n) + h(n)$$

A*算法与一致代价搜索是非常相似的，不同之处就在于一致代价搜索仅仅使用$g(n)$，而它使用的是$g(n) + h(n)$，他的描述也和一致代价搜索大致相同，只需从起点开始，检查所有相邻点，对每个点计算$g + h$得到f，在所有可能的扩展点中，选择f最小的那个点进行扩展，并将这些新的扩展点添加到扩展点列表。同样其中也要维护一个优先级队列，不过此处的优先级队列中是f值越小，其优先级越高。

二、 伪代码

1. 一致代价搜索：

---

function $UNIFORM\_COST\_SEARCH(problem)$ **returns** $a\ solution, or\ failure$:

$node \leftarrow$ 带状态的$node$ ＝初始条件，路径$cost$;

$frontier \leftarrow$ 根据路径$cost$维护优先队列，初始化为起始节点;

$explored \leftarrow$ 已经探索过的节点集合，初始为空集;

loop do:

if ($frontier$为空) ***then return*** $failure$;

$node \leftarrow frontier.pop()$; /* 选择路径 cost 最低的下一个节点*/

if 已经到达终点 ***then return*** $SOLUTION(node)$;

将该节点添加到$explored$集合中;

for each $action$ ***in*** $problem.ACTIONS(node.STATE)$ ***do***:

$child \leftarrow CHILD\_NODE(problem, node, action)$;

if $child.STATE$没有被探索过且不在优先队列中 ***then***:

$frontier.INSERT(child)$;

else if $child.STATE$在优先队列中且路径$cost$更高 ***then***:

$frontier.REPLACE(node\ with\ child)$;

2. A*搜索：

function $A\_STAR\_SEARCH(problem)$ ***returns*** $a\ solution, or\ failure$:

$node \leftarrow$ 带状态的$node$;

$open\_list \leftarrow$ 根据路径评估函数f值维护优先队列，初始化为起始节点;

$close\_list \leftarrow$ 已经探索过的节点集合，初始为空集;

loop do:

if ($open_list$为空) ***then return*** $failure$;

$node \leftarrow open\_list.pop()$; /* 选择评估函数 f 值最小的下一个节点*/

$close\_list.INSERT(node)$;

if 已经到达终点 ***then return*** $SOLUTION(node)$;

获取$node$的相邻节点$neighbors$;

for each $neighbor$ ***in*** $neighbors$ ***do***:

if $neighbor$可达 ***and*** 不在$close_list$中 ***then***:

if $neighbor$在$open_list$中 ***and*** $g(neighbor) > g(node)$***then***:

$update(neighbor$的g值, $neighbor.father \leftarrow node)$;

else:

$update(neighbor$的g值, $neighbor.father \leftarrow node)$;

$open\_list.INSERT(neighbor)$;

三、 代码截图

1. 一致代价搜索：

■ 根据文件建立迷宫：

```
def get_maze(file_path):
    """
    用给定文件建立迷宫
    Args:
        file_path: 字符串 代表文件路径
    Returns:
        maze: 二维列表 代表迷宫
              '1'墙 '0'通路 'S'起点 'E'终点
    """
    fp = open(file_path)
    maze = [line.strip() for line in fp.readlines()]
    fp.close()
    maze = [list(line) for line in maze]
    return maze
```

■ 一致代价搜索：

```
def ucs(maze, start, end):
    """
    给定 迷宫 起点 终点 进行一致代价搜索
    Args:
        maze: 二维列表 代表迷宫
        start: (int, int) 起点坐标
        end: (int, int) 终点坐标
    Returns:
        fathers: 字典 键为子节点坐标 值为父节点坐标
                 可以从终点回溯路径直到起点
    """
    # 迷宫高和宽
    h, w = len(maze), len(maze[0])
    fathers = {}
    # 访问过的节点
    visited = set()
    # 优先队列，按照总路径cost自小到大
    pqueue = PriorityQueue()
    # 插入起始节点
    pqueue.put((0, start))
    fathers[start] = (-1, -1)
```

```
    # 直到队列为空
    while pqueue:
        # 取出最小的cost和对应节点
        cost, node = pqueue.get()
        if node not in visited:
            visited.add(node)
            # 到达终点则可以返回
            if node == end:
                return fathers
            # 遍历当前节点的邻居节点(即上下左右四个节点)
            for dx,dy in [(-1,0),(1,0),(0,-1),(0,+1)]:
                next_x, next_y = node[0]+dx, node[1]+dy
                # 保证坐标要合法
                if next_x < h and next_x >= 0 and next_y < w and next_y >=
0:
                    # 保证改坐标不是墙或者已经访问过
                    if maze[next_x][next_y] != '1' and
                       (next_x, next_y) not in visited:
                        # 将邻居节点的父节点设为当前节点
                        fathers[(next_x, next_y)] = (node[0], node[1])
                        # 总路径cost需要加上新的路径cost，此处迷宫全部为1
                        total_cost = cost + 1
                        # 将新的总cost和节点加入优先队列
                        pqueue.put((total_cost, (next_x, next_y)))
```

2. A*搜索：

由于 A*算法这一类启发式搜索需要记录的信息比无信息搜索要多，所以分别建立了节点类和 AStar 类来实现。

■ 节点类：

```
class Node(object):
    """节点类
    Attributes:
        x, y: int 横纵坐标
        g: float 到当前节点的路径成本
        h: float 当前节点到终点的启发式估计值
        f: float 起点到终点的成本估计值
        parent: Node 节点类 父节点 用于寻路
        accessible: bool 是否可达
    """
    def __init__(self, x, y, accessible):
```

```
        """初始化"""
        self.x = x
        self.y = y
        self.g = 0
        self.h = 0
        self.f = 0
        self.parent = None
        self.accessible = accessible

    def __lt__(self, other):
        """重载函数< 用于优先队列中的比较排序
        Args:
            other: <的右值 Node 节点类
        """
        return self.f < other.f
```

■ AStar 类：

```
class AStar(object):
    """A*算法
    Attributes:
        maze: 二维 list 元素为 Node 迷宫
        height: int 迷宫高度
        width: int 迷宫宽度
        open_list: list A*算法维护的 open 列表，需要堆化为优先队列
        close_list: set A*算法维护的 close 列表
        start: Node 起点
        end: Node 终点
    """
    def __init__(self, file_path):
        """初始化"""
        self.maze = self.init_maze(file_path)
        self.height = len(self.maze)
        self.width = len(self.maze[0])
        self.open_list = []
        self.close_list = set()
        self.start = self.maze[1][34]
        self.end = self.maze[16][1]

        heapq.heapify(self.open_list)

    def init_maze(self, file_path):
        """根据给定文件初始化迷宫
        Args:
```

```
            maze: 二维 list 元素为 Node 迷宫
        """
        fp = open(file_path)
        maze = [line.strip() for line in fp.readlines()]
        fp.close()
        maze = [list(line) for line in maze]
        h, w = len(maze), len(maze[0])
        res = [[None for j in range(w)] for i in range(h)]

        # '1'为墙不可达，其余可达
        for x in range(h):
            for y in range(w):
                accessible = False if maze[x][y] == '1' else True
                res[x][y] = Node(x, y, accessible)
        return res

    def heuristic(self, node):
        """计算给定节点到终点的启发式估计值 h
        Args:
            node 节点
        """
        return abs(node.x - self.end.x) + abs(node.y - self.end.y)

    def get_neighbour(self, node):
        """得到给定节点的相邻节点
        Args:
            node 节点
        Returns:
            neighbours list node 的相邻节点列表
        """
        neighbours = []
        # 判断上下左右四个坐标是否合法，是则加入 neighbours
        if node.x + 1 < self.height:
            neighbours.append(self.maze[node.x + 1][node.y])
        if node.x > 0:
            neighbours.append(self.maze[node.x - 1][node.y])
        if node.y + 1 < self.width:
            neighbours.append(self.maze[node.x][node.y + 1])
        if node.y > 0:
            neighbours.append(self.maze[node.x][node.y - 1])
        return neighbours

    def get_path(self):
        """从终点回溯得到路径上的节点坐标列表
```

```
        Returns:
            neighbours list 路径上的节点坐标
        """
        res = []
        node = self.end
        while node.parent is not self.start:
            node = node.parent
            res.append((node.x, node.y))
        return  res

    def update_node(self, neig, node):
        """更新节点
            将 node 设为邻居节点 neig 的 father
            更新 neig 的路径成本 启发式估计值 总估计值
        """
        neig.parent = node
        neig.g = node.g + 1
        neig.h = self.heuristic(neig)
        neig.f = neig.h + neig.g

    def find_path(self):
        """寻找最短路径
            使用 A*算法寻找 maze 的最短路径
        """
        # 由于此处需要 open 列表为可迭代对象故没有直接使用 PriorityQueue
        # 堆化，即建立优先队列，将起点和起点估计值加入 open_list 中
        heapq.heappush(self.open_list, (self.start.f, self.start))

        # 直至 open_list 为空
        while len(self.open_list) > 0:
            # 从 open_list 取出总估计值 f 最小的节点加入 close_list 中
            f, node = heapq.heappop(self.open_list)
            self.close_list.add(node)
            # 到达终点则可以返回
            if (node.x, node.y) == (self.end.x, self.end.y):
                break

            neighbours = self.get_neighbour(node)
            # 遍历其所有邻居节点
            for neig in neighbours:
                # 保证其可达性且不在 close_list 中
                if neig.accessible and neig not in self.close_list:
                    # 如果已经在 open_list 中且 new_g > old_g + cost 则需要更新
节点
```

```
                    if (neig.f, neig) in self.open_list:
                        if neig.g > node.g + 1:
                            self.update_node(neig, node)
                    # 如果不在 open_list 中则更新节点并加入 open_list 中
                    else:
                        self.update_node(neig, node)
                        heapq.heappush(self.open_list, (neig.f, neig))
```

■ 路径可视化：

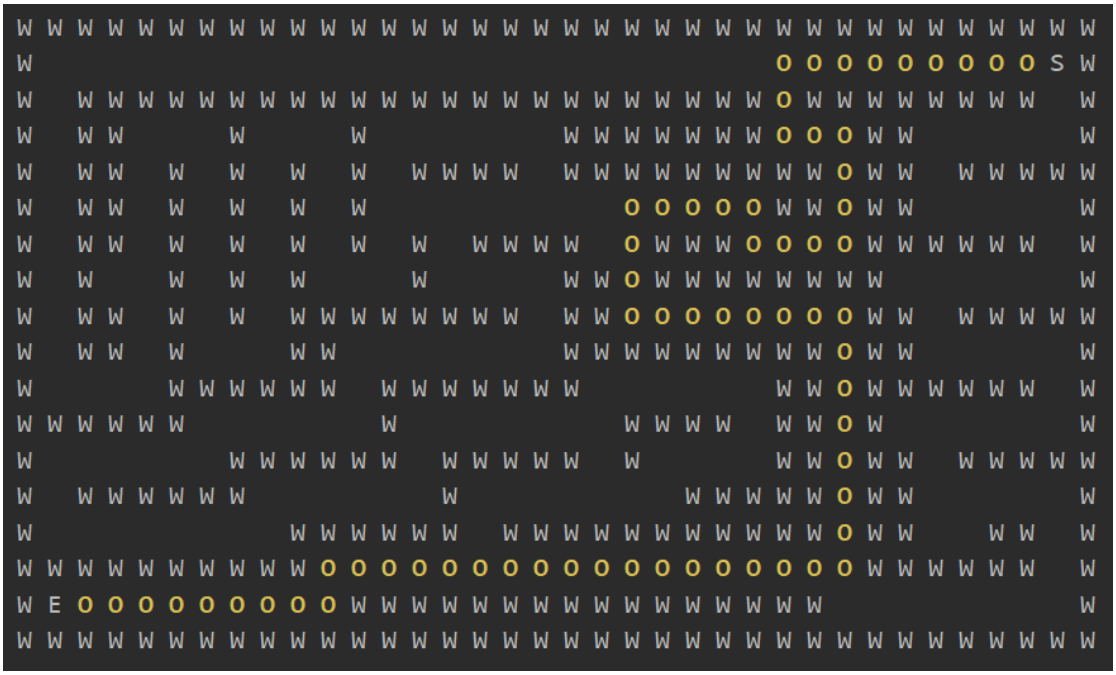
```
def display_path(path):
    """
    给定 path 列表 可视化迷宫路径
    Args:
        path: list 路径上的坐标
    """
    fp = open('D:/AI Lab/lab9/MazeData.txt')
    maze = [line.strip() for line in fp.readlines()]
    fp.close()
    maze = [list(line) for line in maze]
    h, w = len(maze), len(maze[0])

    # 路径节点置为黄色的'O'
    for (x,y) in path:
        maze[x][y] = '\033[1;33mO\033[0m'
    # 将迷宫有墙的地方置为'W', 通路置为' '
    for x in range(h):
        for y in range(w):
            if(maze[x][y] == '0'):
                maze[x][y] = ' '
            elif(maze[x][y] == '1'):
                maze[x][y] = 'W'

    maze = [' '.join(line) for line in maze]
    for line in maze:
        print(line)
```

四、 实验结果及分析

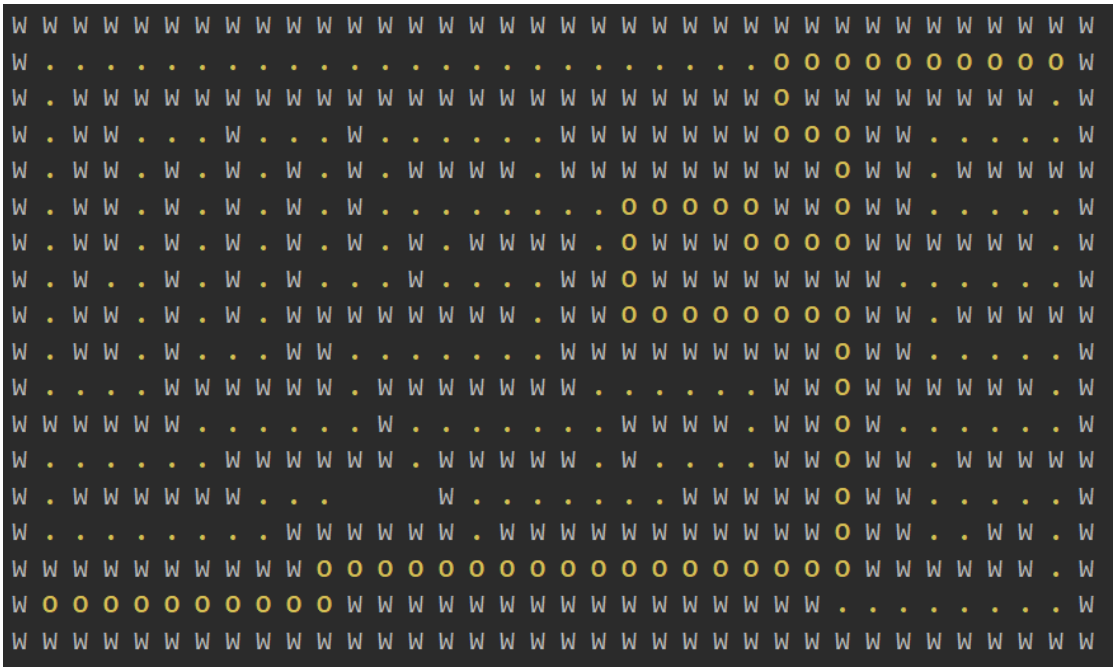
1. 一致代价搜索：

运行时间：0.003178119659423828

最终得到的路径：



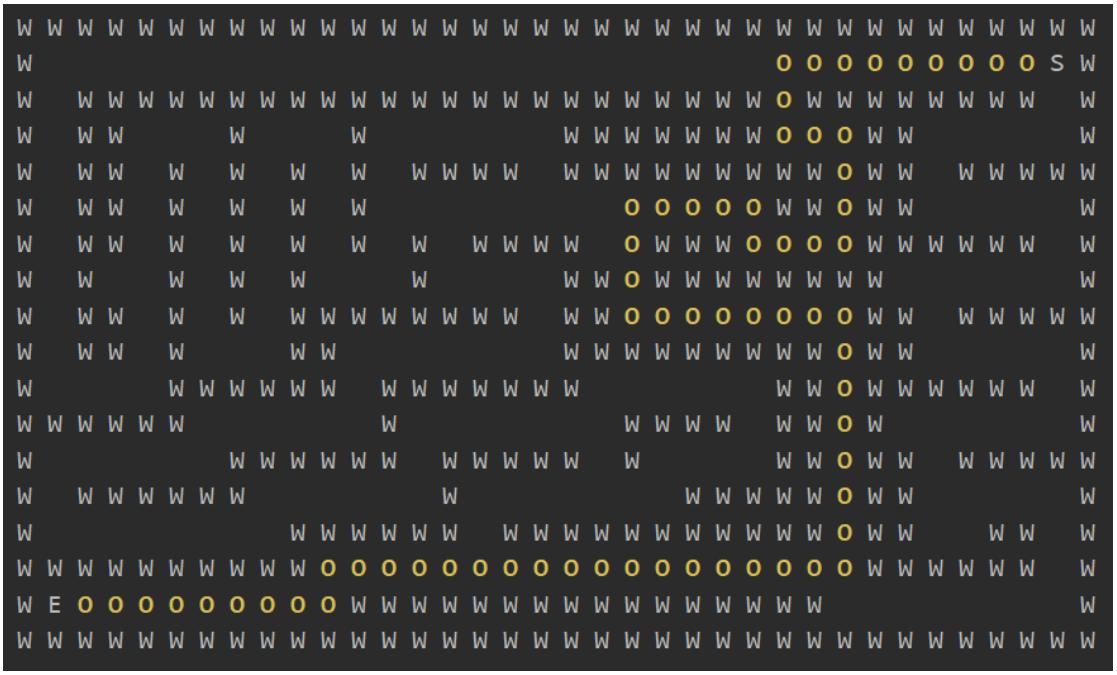
访问过的所有节点：



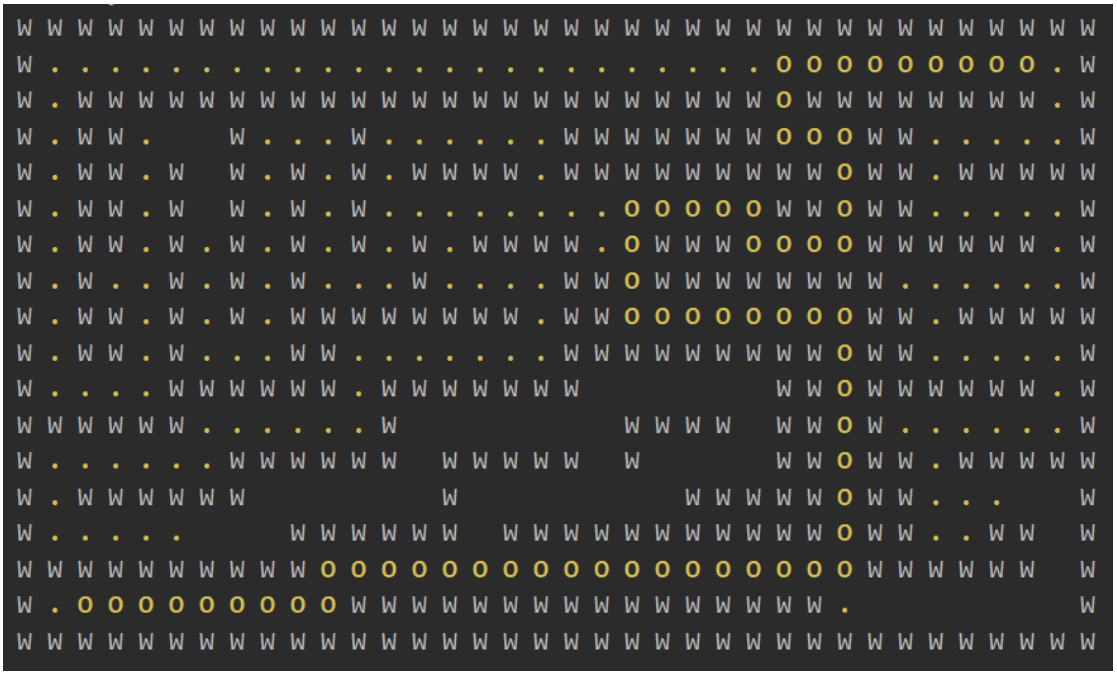
2. A*算法

运行时间 0.0009944438934326172

最终得到的路径：



访问过的所有节点：



经过验证，一致代价搜索和A*搜索得到的都是最短路径，体现了两者解的最优性。

根据探索过的节点来看，由于迷宫的路径cost相同，一致代价搜索退化为了BFS，探索了几乎所有节点（仅有几个没有探索过是因为找到终点提前退出，没有从优先队列中取出，相当于被剪枝了），而A*算法由于具有启发式函数的额外信息，探索的节点明显减少。

从运行时间也来看，一致代价搜索的运行时间是0.0032，且多次运行时间都在这个时间左右，而A*算法运行时间为0.00095，多次运行也有多次出现运行时间为0.0的情况，明显比一致代价搜索快了很多。

五、 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？

策略	优点	缺点	适用场景
深度优先搜索	空间复杂度为线性，状态空间有限且剪枝条件下具有完备性，一定条件下能快速得到解。	不具有最优性，遍历路径过长时时间效率很低	在只需寻找通路无需最优情况下表现优秀。
宽度优先搜索	具有完备性和最优性，多解情况下一定可以找到最优解。	空间复杂度为指数，内存消耗巨大。	对于稀疏无权图效果良好。
一致代价搜索	具有完备性和最优性，使用宽度优先搜索的思想，但空间复杂度比宽度优先搜索低。	空间复杂度依然为指数，且时间复杂度比BFS高	对于稀疏带权图效果良好。
迭代加深搜索	具有完备性和最优性，空间复杂度为线性，比BFS更高效，不用扩展深度限制上的节点。	时间复杂度高，运行耗时长。	适用于不确定状态空间是否有限的条件。
A*搜索	具有最优性和完备性，启发式函数含有额外信息，比上述搜索都要高效	需要维护开启列表和关闭列表，空间复杂度为指数。	适用于启发式函数设计较好且图较为稠密时。
IDA*搜索	使用回溯方法，不用保存中间状态，大大节省了空间	重复搜索：每次depth变大都要再次从头搜索	在稀疏有向图中表现优秀，性能比A*有提高。