中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	信息与计算科学班	专业（方向）	信息与计算科学
学号	21311359	姓名	何凯迪

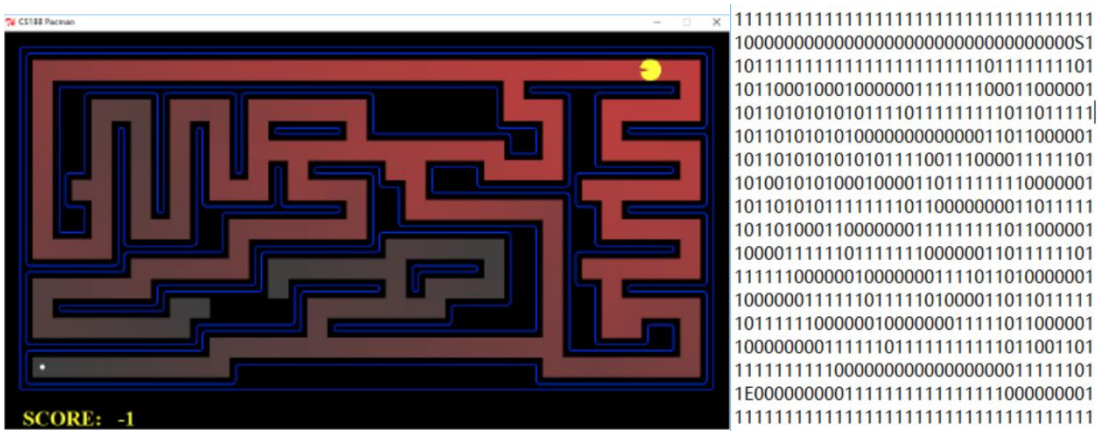
一、 实验题目

从给定类型一和类型二中分别选择一个策略解决迷宫问题。

类型一：BFS、DFS

类型二：一致代价、迭代加深、双向搜索

地图如下：（S 表示起点，E 表示终点，1 表示墙，0 是可通行）



二、 实验内容

1. 算法原理

BFS：广度优先搜索（BFS）是一种用于图或树中的搜索算法，用于遍历或查找从根节点开始的所有节点。它通过逐层扫描节点来实现，即首先遍历根节点的所有直接邻居节点，然后遍历它们的邻居节点，以此类推，直到找到目标节点或完全遍历。

UCS：一致代价搜索（UCS）是一种基于图或树的搜索算法。该算法使用了一个优先级队列来存储节点，优先级由当前路径的代价决定。算法从起始节点开始，在队列中选取代价最小的节点进行扩展，知道找到目标节点或者队列为空。在每一次扩展节点时，算法会将该节点的后继节点加入到队列中，并计算到达每个后继节点的代价。如果队列中已经存在某个后继节点，算法会比较当前路径和已有路径的代价，并选择代价更小的路径。这样算法会保证先处理代价更小的路径，直到找到目标节点为止。

2. 伪代码

BFS：

```
function BFS(start, goal):
    queue = createQueue()   # 初始化一个队列
    visited = createVisited()  # 初始化一个访问记录数组
    parent = createParent()   # 初始化一个记录父节点的字典

    enqueue(queue, start)  # 将起点加入队列
    visited[start] = True  # 标记起点已访问
    parent[start] = None   # 起点没有父节点

    while queue is not empty:
        node = dequeue(queue)  # 取出队列头部的节点
        if node == goal:  # 如果找到目标节点，返回路径
            path = getPath(parent, start, goal)
            return path
        for neighbor in getNeighbors(node):  # 获取当前节点的邻居节点
            if not visited[neighbor]:  # 如果邻居节点没有被访问过
                enqueue(queue, neighbor)  # 将邻居节点加入队列
                visited[neighbor] = True  # 标记邻居节点已访问
                parent[neighbor] = node   # 记录邻居节点的父节点为当前节点

    # 如果遍历完整张图都没有找到目标节点，说明无法到达目标节点，返回空路径
    return []
```

完整伪代码如下：

```
打开文件并读取迷宫数据
将数据存储为二维数组maze_array

定义BFS函数bfs(maze, sx, sy, ex, ey):
    定义队列queue并将起点位置(sx, sy)加入队列
    定义visited数组并初始化为False
    visited[sx][sy] = True
    定义四个方向directions
    定义路径字典path_dict
    将起点位置(sx, sy)的路径加入到path_dict中
    while 队列queue不为空:
        取出队首位置(x, y)
        如果到达终点ex, ey:
            返回path_dict[(ex, ey)]
        搜索四个方向
        for 每个方向(dx, dy):
            计算下一个位置(nx, ny)
            如果下一个位置在迷宫内，且没有被访问过，且不是墙:
                将下一个位置标记为已访问
                将下一个位置加入队列
                将下一个位置的路径加入路径字典path_dict

    如果没有找到路径，返回None

找到起点和终点的位置
使用BFS算法找到从E到S的路径，并将其存储到path列表中

将路径位置标红并输出
```

UCS：

```
function UCS(start, goal, neighbors, cost_function):
    frontier = PriorityQueue() // 优先队列，根据代价排序
    frontier.put(start, 0) // 起点入队
    explored = set() // 已访问的位置
    path_dict = {start: [start]} // 路径字典
    while not frontier.empty():
        current = frontier.get() // 从优先队列中取出代价最小的位置
        if current == goal: // 如果找到终点，返回路径
            return path_dict[current]
        explored.add(current) // 将当前位置标记为已访问
        for neighbor in neighbors(current): // 遍历相邻位置
            if neighbor not in explored: // 如果相邻位置没有被访问过
                new_cost = cost_function(current, neighbor) // 计算到达相邻位置的代价
                if neighbor not in frontier or new_cost < frontier.get_cost(neighbor): // 如果相邻位置没有在队列中，或者新的代价更小
                    frontier.put(neighbor, new_cost) // 将相邻位置加入队列
                    path_dict[neighbor] = path_dict[current] + [neighbor] // 更新路径字典
    return None // 如果没有找到路径，返回None
```

完整伪代码如下：

```
读取迷宫数据并将其转化为二维数组
找到起点和终点的位置

定义UCS算法函数(maze, sx, sy, ex, ey):
    定义优先队列和visited数组
    将起点加入优先队列
    定义四个方向
    定义路径字典
    while 优先队列不为空:
        取出优先队列中代价最小的位置
        如果到达终点，返回路径
        搜索四个方向
        如果下一个位置在迷宫内，且没有被访问过，且不是墙:
            将下一个位置标记为已访问
            计算从起点到下一个位置的代价
            将下一个位置和对应的代价加入优先队列
            将下一个位置的路径加入路径字典
    如果没有找到路径，返回None

使用UCS算法找到从E到S的路径
将路径标红并输出
```

3. 关键代码展示（带注释）

BFS:

```python
# 定义BFS函数
def bfs(maze, sx, sy, ex, ey):
    # 定义队列和visited数组
    queue = [(sx, sy)]
    visited = [[False] * len(maze[0]) for _ in range(len(maze))]
    visited[sx][sy] = True
    # 定义四个方向
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    # 定义路径字典
    path_dict = {(sx, sy): [(sx, sy)]}
    # 开始BFS
    while queue:
        # 取出队首位置
        x, y = queue.pop(0)
        # 如果到达终点，返回路径
        if x == ex and y == ey:
            return path_dict[(x, y)]
        # 搜索四个方向
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            # 如果下一个位置在迷宫内，且没有被访问过，且不是墙
            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and not visited[nx][ny] and maze[nx][ny] != '1':
                # 将下一个位置标记为已访问
                visited[nx][ny] = True
                # 将下一个位置加入队列
                queue.append((nx, ny))
                # 将下一个位置的路径加入路径字典
                path_dict[(nx, ny)] = path_dict[(x, y)] + [(nx, ny)]
    # 如果没有找到路径，返回None
    return None
```

UCS:

```python
# 定义UCS算法函数
def ucs(maze, sx, sy, ex, ey):
    # 定义优先队列和visited数组
    heap = [(0, sx, sy)]
    visited = [[False] * len(maze[0]) for _ in range(len(maze))]
    visited[sx][sy] = True
    # 定义四个方向
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    # 定义路径字典
    path_dict = {(sx, sy): [(sx, sy)]}
    # 开始UCS
    while heap:
        # 取出优先队列中代价最小的位置
        cost, x, y = heapq.heappop(heap)
        # 如果到达终点，返回路径
        if x == ex and y == ey:
            return path_dict[(x, y)]
        # 搜索四个方向
        for dx, dy in directions:
            nx, ny = x + dx, y + dy
            # 如果下一个位置在迷宫内，且没有被访问过，且不是墙
            if 0 <= nx < len(maze) and 0 <= ny < len(maze[0]) and not visited[nx][ny] and maze[nx][ny] != '1':
                # 将下一个位置标记为已访问
                visited[nx][ny] = True
                # 计算从起点到下一个位置的代价
                next_cost = cost + 1
                # 将下一个位置和对应的代价加入优先队列
                heapq.heappush(heap, (next_cost, nx, ny))
                # 将下一个位置的路径加入路径字典
                path_dict[(nx, ny)] = path_dict[(x, y)] + [(nx, ny)]
    # 如果没有找到路径，返回None
    return None
```

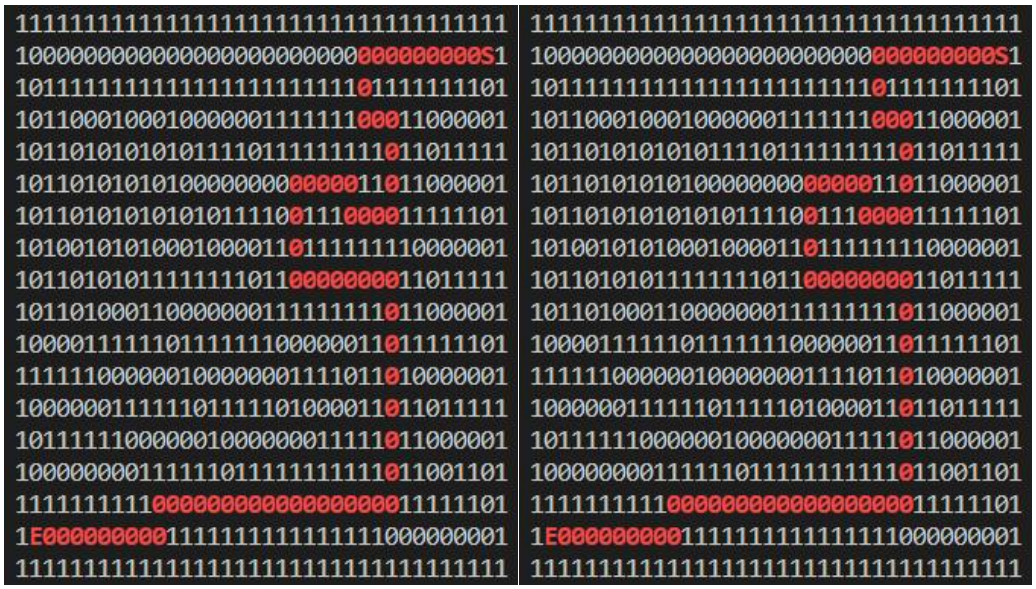
4. 创新点&优化（如果有）

加了 time 和 psutil 库，用于计算进程运行占用的内存和时间。

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

BFS： **UCS：**



在无权图中，BFS 算法从起点开始，它所能够达到的相邻节点都会被遍历到，这意味着第一次到达目标节点时，所经过的路径一定是最短路径；UCS 算法可以说是 BFS 的在有权

图的扩展，以代价最小的节点展开，所以所得到的路径也一定是最短路径。

2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

BFS：

```
Memory used: 86016 bytes
代码运行时间：  0.003000497817993164 秒
```

UCS：

```
Memory used: 94208 bytes
代码运行时间：  0.0030014514923095703 秒
```

可以看到两者进程占用内存相近，之所以 UCS 算法占内存更大，是因为其使用优先队列，在相同的小规模问题中，优先队列比普通队列更占内存。
另一方面，两者运行时间也几乎一样，其时间复杂度都是 O[b^(d+1)]，之所以 UCS 算法运行时间稍长一点，是因为其需要更多计算代价大小。

关于 BFS 算法与 UCS 算法四个方面的性能对比如下：

完备性：两者的完备性通常都可以保证，但是 UCS 算法由于只关心路径总代价，所以如果存在 0 代价行动就看陷入死循环。
最优性：当每个路线的代价都一样时，BFS 算法是最优的。而 UCS 算法的优先队列保证了每一个出队扩展的节点都是最优的，即可得到最优路径。

时间复杂度：BFS 算法的时间复杂度为 $O(b^d)$;

UCS 算法的时间复杂度为 $O(b^{1+\frac{C}{e}})$

空间复杂度：BFS 算法的空间复杂度为 $O(b^d)$;

UCS 算法的空间复杂度为 $O(b^{1+\frac{C}{e}})$

四、 思考题

1. 这些策略的优缺点是什么？它们分别适用于怎样的场景？

（1）广度优先搜索

优点：

①可以找到最短路径。
②搜索效率较高，特别是在搜索目标状态距离起始状态较近的情况下。
③对于无权图或路径代价相等的情况，可以保证找到最优解。

缺点：

①空间复杂度较高，需要存储搜索过程中遇到的所有状态。
②对于搜索目标状态距离起始状态较远的情况，效率较低。

适用场景：

①无权图或路径代价相等的情况。
②搜索目标状态距离起始状态不远，需要找到最短路径的情况。

（2）一致代价搜索

优点：

①可以找到最短路径。

②对于路径代价相等的情况，可以保证找到最优解。

③可以处理一些启发式函数不适用的情况。

缺点：

①空间复杂度较高，需要存储搜索过程中遇到的所有状态。

②对于搜索目标状态距离起始状态较远的情况，效率较低。

适用场景：

①路径代价相等的情况。

②搜索目标状态距离起始状态不远，且需要找到最短路径的情况。

（3）深度优先搜索

优点：

①空间复杂度较低，只需要存储当前搜索路径上的状态。

②对于搜索目标状态距离起始点较远的情况，可以提高效率。

③可以处理一些路径问题。

缺点：

①无法找到最优解。

②可能会陷入无限循环或搜索到无法返回的状态。

适用场景：

①对于需要搜索到任何解的问题。

②无法确定目标状态距离起始状态距离的情况。

（4）深度受限搜索

优点：

①可以限制搜索深度，避免无限递归或搜索到无法返回的状态。

缺点：

①只能在特定深度范围内搜索，可能会错过最优解。

②对于搜索目标状态距离起始状态较远的情况，效率较低。

适用场景：

①需要限制搜索深度的情况。

（5）迭代加深搜索

优点：

①占用内存较少，因为每次只搜索一定深度范围内的节点，不需要把整个搜索树存储在内存中。

②找到的解一定是最浅层的解，因此可以减少搜索时间和空间复杂度。

③搜索结果具有可视化和可扩展性。

缺点：

①在深度限制较浅的情况下，迭代加深搜索的效率并不高。

②对于某些情况，即使增加深度限制，搜索仍然无法找到解，这样会浪费一定的时间和空间资源。

适用场景：

①当搜索树比较深，占用内存较多的情况。

②在需要快速找到解但无需求出所有解的情况。

③对于一些搜索空间非常大的问题，可以使用迭代加深搜索来缩小搜索范围，提高效率。

（6）双向搜索

优点：

①可以减少搜索空间，提高搜索效率。

②对于搜索目标状态距离起始状态较远的情况，可以提高效率。

③可以找到最短路径。

缺点：

①需要额外的存储空间来存储反向搜索的状态。

②对于某些状态，可能不存在反向的扩展操作，故无法双向搜索。

适用场景：

①搜索目标状态距离起始状态较远的情况。

②需要找到最短路径的情况。

③搜索空间较大的情况，可以使用双向搜索减少搜索空间。

五、 参考资料

https://blog.csdn.net/chan1987818/article/details/127968275?csdn_share_tail=%7B%22type%22%3A%22blog%22%2C%22rType%22%3A%22article%22%2C%22rId%22%3A%22127968275%22%2C%22source%22%3A%22kaddyhe%22%7D&fromshare=blogdetail

https://blog.csdn.net/qq_40723803/article/details/105097401?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216804881991680018859898 7%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=168048819916800188598987&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~sobaiduend~default-2-105097401-null-null.142^v80^insert_down1,201^v4^add_ask,239^v2^insert_chatgpt&utm_term=python%E6%9F%A5%E7%9C%8B%E5%8D%A0%E7%94%A8%E5%86%85%E5%AD%98%E5%A4%A7%E5%B0%8F&spm=1018.2226.3001.4187

https://blog.csdn.net/weixin_50853979/article/details/125119368?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216804923301680021721 1234%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=168049233016800217211234&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-2-125119368-null-null.142^v80^insert_down1,201^v4^add_ask,239^v2^insert_chatgpt&utm_term=python%E7%9A%84join%E5%87%BD%E6%95%B0&spm=1018.2226.3001.4187