中山大学计算机学院

人工智能

本科生实验报告

（2023学年春季学期）

课程名称：Artificial Intelligence

教学班级	计科1班	专业（方向）	计算机科学与技术
学号	21307035	姓名	邓栩瀛

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。

- 类型一：一致代价、迭代加深、双向搜索
- 类型二：A*和IDA*

S表示起点，
E表示终点。
1表示墙，
0是可通行。

```
11111111111111111111111111111111111
10000000000000000000000000000000S1
10111111111111111111111101111111101
10110001000100000011111100011000001
10110101010101111011111111011011111
10110101010100000000000001101100001
10110101010101011110011100011111101
10100101010001000011011111111000001
10110101011111111011000000011011111
10110100011000000011111111011000001
10000111111011111110000011011111101
11111100000010000000111101101000001
10000001111110111111010001101101111
10111111000000100000001111101100001
10000000111111011111111111011001101
11111111110000000000000000011111101
1E000000000111111111111111000000001
11111111111111111111111111111111111
```

二、实验内容

1.算法原理

一致代价搜索

一致代价搜索是一种基于代价的搜索算法，它寻找从起点到终点的最短路径。UCS算法通过对节点的代价进行排序，每次选择当前代价最小的节点进行扩展，直到找到终点或者扩展完所有可达节点。

算法步骤：

1. 初始化：将`start`加入待扩展列表，路径代价为0。
2. 扩展节点：从待扩展列表中选择代价最小的节点进行扩展，计算该节点的相邻节点的路径代价，并将它们

加入到待扩展列表中。如果相邻节点已经在待扩展列表中，更新其路径代价为更小的值。

3. 判断终点：如果待扩展列表为空，则搜索失败；如果待扩展列表中的节点是终点，则搜索成功，返回`path`。

UCS算法的时间复杂度取决于图的大小和复杂度，并且需要扩展所有可达节点，其时间复杂度为$O(b^{1+\lfloor C*/\varepsilon \rfloor})$，空间复杂度为$O(b^{1+\lfloor C*/\varepsilon \rfloor})$

(b: 最大分支因子，$C*$: 最短路径的代价，ε: 代价最小的两条路径之间的比率)

A*算法

A*算法是一种启发式搜索算法，在USC算法的基础上增加了启发式函数来引导搜索方向，从而提高搜索效率。

启发式函数用于预测从当前节点到目标节点的最短路径长度，以便选择最有可能导向目标的节点进行扩展。A*算法在保证最优解的情况下，相对于UCS算法，可以更快地找到起点到终点的最短路径。

算法步骤

1. 初始化：将起点加入待扩展列表`open list`，路径代价为0。
2. 扩展节点：从待扩展列表中选择f值最小的节点进行扩展

$$f(current) = g(current\text{到起点的路径代价}) + h(current\text{到终点的启发式函数值}) \quad (1)$$

对于`current`的相邻节点，计算从起点到该节点的路径代价，并更新其启发式函数值和前驱节点。

3. 判断终点：如果待扩展列表`open list`为空，则搜索失败；如果待扩展列表中的节点是终点，则搜索成功，返回路径。

启发式函数的选择：

1. 可采纳：$h(n) <= h^*(n)$
2. 单调：$h(n) <= cost(n, n') + h(n')$
3. 常用的启发式函数：曼哈顿距离、欧式距离、切比雪夫距离等

2.伪代码

一致代价搜索

`Graph`表示一个包含节点和边的图，`start`表示起点，`goal`表示终点。`frontier`表示待扩展的节点列表，`came_from`表示每个节点的前驱节点，`cost`表示从起点到每个节点的路径代价。`neighbors(current)`表示获取当前节点`current`的相邻节点，`cost(current, next)`表示获取从节点`current`到节点`next`的边的代价。

算法流程:

- 通过一个优先队列（`PriorityQueue`）来存储待扩展的节点列表，按照路径代价从小到大排序。
- 每次扩展将路径代价最小的节点从队列中取出，然后对其相邻节点进行扩展。若从起点到相邻节点的新路径代价更小，则更新路径代价和前驱节点，并将相邻节点加入待扩展列表。
- 如果搜索成功，返回起点到终点的最短路径，否则返回空值。

```
function UniformCostSearch(Graph, start, goal)
    frontier = PriorityQueue()     // 存储待扩展的节点，按照路径代价从小到大排序
```

```
    frontier.put(start, 0)          // 将起点加入待扩展列表
    came_from = {}                  // 存储每个节点的前驱节点
    cost = {}                 // 存储起点到每个节点的路径代价
    came_from[start] = None
    cost[start] = 0

    while not frontier.empty()
        current = frontier.get()  // 获取路径代价最小的节点
        if current == goal        // 若当前节点为终点，则返回路径
            return reconstruct_path(came_from, start, goal)

        for next in Graph.neighbors(current)  // 对当前节点的相邻节点进行扩展
            new_cost = cost[current] + Graph.cost(current, next)
            // 计算从起点到next的路径代价
            if next not in cost or new_cost < cost[next]
            // 若next未被扩展过或新路径比原路径更优
                cost[next] = new_cost    // 更新路径代价
                priority = new_cost                  // 以路径代价作为优先级
                frontier.put(next, priority)    // 将next加入待扩展列表
                came_from[next] = current       // 更新next的前驱节点为current

    return None  // 若搜索失败，返回空值

function reconstruct_path(came_from, start, goal)  // 通过前驱节点重构路径
    path = [goal]
    while path[-1] != start
        path.append(came_from[path[-1]])
    path.append(came_from[start])
    path.reverse()
    return path
```

A*算法

在A*算法中，使用的`open_list`、`pre`（前驱节点）和`cost`（存储起点到每个节点的g值）和一个启发式函数`heuristic`，用于预测从当前节点到目标节点的最短路径长度，以便选择最有可能导向目标的节点进行扩展。在本案例中，使用Manhattan距离作为启发式函数。

算法流程：

- 从`open_list`中获取`f`最小的节点`current`，然后对其相邻节点进行扩展。
- 对于每个相邻节点`next`，计算从起点到该节点的路径代价`new_cost`，如果该节点未被扩展过或新路径比原路径更优，则更新其`cost`值和前驱节点`pre`，并计算该节点的`f`。
- 将该节点加入到`open_list`中，按照其`f`大小排序。
- 路径重构函数`reconstruct_path`通过前驱节点回溯路径。

```
function AStarSearch(Graph, start, goal)
    open_list = PriorityQueue()     // 存储待扩展的节点，按照f值（g值+估价函数值）从小到大排序
    open_list.put(start, 0)         // 将起点加入待扩展列表，f值为0
    pre = {}                  // 存储每个节点的前驱节点
    cost = {}                 // 存储起点到每个节点的g值（路径代价）
    pre[start] = None
    cost[start] = 0
    while not open_list.empty()
```

```
        current = open_list.get()  // 获取f值最小的节点
        if current == goal        // 若当前节点为终点，则返回路径
            return reconstruct_path(pre, start, goal)

        for next in Graph.neighbors(current)   // 对当前节点的相邻节点进行扩展
            new_cost = cost[current] + Graph.cost(current, next)
              // 计算从起点到next的路径代价
            if next not in cost or new_cost < cost[next]:
            //若next未被扩展过或新路径比原路径更优
                cost[next] = new_cost    // 更新g值
                f = new_cost + heuristic(next, goal)  // 计算next的f值
                open_list.put(next, f)    // 将next加入待扩展列表，f值为f
                pre[next] = current      // 更新next的前驱节点为current
    return None  // 搜索失败，返回空值

function heuristic(a, b) //启发式函数，采用Manhattan距离
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

function reconstruct_path(pre, start, goal)  // 通过前驱节点重构路径
    path = [goal]
    while path[-1] != start:
        path.append(closed_list[path[-1]])
    path.append(closed_list[start])
    path.reverse()
    return path
```

3.关键代码展示（带注释）

文件读取与迷宫数据的存储

- 打开文件`MazeData.txt`，用`read()`读取文件内容，并关闭文件。
- 按行遍历`maze_str`字符串，用字符串切片的方式将每个字符提取出来，并根据元素的类型将其转化为数字或保留为字符串。
- 将每行的列表组成二维列表`maze`。

```
maze_file = open("MazeData.txt", "r")
maze_str = maze_file.read()
maze_file.close()
# 用列表存储maze
maze = []
for line in maze_str.splitlines():#按行读取
    row = [int(line[i]) if line[i].isdigit() else line[i] for i in range(len(line))]
    # 数字0、1直接存储，字母（S、E）用字符串类型存储
    maze.append(row)
```

一致代价搜索的代码实现

- 初始化`closed`列表、前驱节点`pre`列表、起点代价值`g`和`open`表，将起点加入`open`表。
- 当`open`表非空时，从中取出代价最小的节点进行扩展。
- 对当前节点的四个相邻节点进行扩展，更新其前驱节点、代价值和加入`open`表。

- 如果相邻节点已经在`closed`表中，并且代价高于之前扩展到该节点的代价，则不更新。
- 如果相邻节点未被扩展过或者新的代价值更小，则更`pre`前驱节点和代价值，并加入`open`表。
- 如果扩展到终点，则返回路径；如果`open`表为空，说明无法找到路径，返回False。

```python
def ucs(maze, start, end):
    # 定义四个方向的相邻节点，分别为右、左、下、上
    neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    close_set = set() # closed列表
    pre = {}  # 前驱节点列表
    g = {start: 0}  # g值初始化
    open_heap = []  # open表初始化
    heapq.heappush(open_heap, (g[start], start))
    # 当open表不为空时，从中取出代价最小的节点，并进行扩展
    while open_heap:
        current = heapq.heappop(open_heap)[1]
        if current == end:
            # 找到终点，返回路径
            data = []
            while current in pre:
                data.append(current)
                current = pre[current]
            return data
        close_set.add(current)
        # 对当前节点的四个相邻节点进行扩展
        for i, j in neighbors:
            neighbor = current[0] + i, current[1] + j
            tentative_g = g[current] + 1
            # 判断相邻节点的有效性：在迷宫范围内且不是墙
            if 0 <= neighbor[0] < len(maze) and 0 <= neighbor[1] < len(maze[0]):
                if maze[neighbor[0]][neighbor[1]] == 1:
                    continue
            else:
                continue
            # 如果相邻节点已经在closed表中，并且代价高于之前扩展到该节点的代价，则不更新
            if neighbor in close_set and tentative_g >= g.get(neighbor, 0):
                continue
            # 如果相邻节点未被扩展过或者新的代价值更小，则更新pre前驱节点和代价值，并加入open表
            if tentative_g < g.get(neighbor, 0) or neighbor not in [i[1] for i in open_heap]:
                pre[neighbor] = current
                g[neighbor] = tentative_g
                heapq.heappush(open_heap, (g[neighbor], neighbor))
    # 如果open表为空，则表示无法到达终点，返回False
    return False
```

启发式函数设计

在迷宫问题中，只允许向四邻域的移动，因此使用Manhattan距离

```python
# 定义A*算法中的启发式函数：Manhattan距离
def heuristic(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

A*搜索算法代码实现

- 初始化`closed`表、前驱节点`pre`列表、起点代价值`g`和启发式函数值`h`，将起点加入`open`表。
- 当`open`表非空，从中取出启发式函数值最小的节点进行扩展。
- 对当前节点的四个相邻节点进行扩展，更新其前驱节点、代价和启发式函数值，加入`open`表。
- 如果相邻节点已经在`closed`表中并且代价高于之前扩展到该节点的代价，则不更新。
- 如果相邻节点未被扩展过或者当前的代价值更小，则更新前驱节点、代价值和启发式函数值，并加入`open`表。
- 如果扩展到终点，则返回路径；如果`open`表为空，说明无法找到路径，返回False。

```python
def a_star(maze, start, end):
    # 定义四个方向的相邻节点，分别为右、左、下、上
    neighbors = [(0, 1), (0, -1), (1, 0), (-1, 0)]
    close_set = set()  # closed表的初始化
    pre = {}  # 前驱节点列表的初始化
    g = {start: 0}  # 起始点的代价值为0
    f = {start: heuristic(start, end)}  # 起始点的f值为其对应的启发式函数的值
    open_heap = []  # open表的初始化
    heapq.heappush(open_heap, (f[start], start))
    # 当open表不为空时，从中取出代价值最小的节点进行扩展
    while open_heap:
        current = heapq.heappop(open_heap)[1]
        if current == end:
            # 找到终点，返回路径
            data = []
            while current in pre:
                data.append(current)
                current = pre[current]
            return data
        close_set.add(current)
        # 对当前节点的四个相邻节点进行扩展
        for i, j in neighbors:
            neighbor = current[0] + i, current[1] + j
            tentative_g = g[current] + 1  # 相邻节点的g值为当前节点的g值+1
            # 判断相邻节点的有效性：在迷宫范围内且不是墙
            if 0 <= neighbor[0] < len(maze) and 0 <= neighbor[1] < len(maze[0]):
                if maze[neighbor[0]][neighbor[1]] == 1:
                    continue
                else:
                    continue
            # 如果相邻节点已经在closed表中并且代价高于之前扩展到该节点的代价，则不更新
            if neighbor in close_set and tentative_g >= g.get(neighbor, 0):
                continue
            # 如果相邻节点未被扩展过或者当前的代价值更小
            if tentative_g < g.get(neighbor, 0) or neighbor not in [i[1] for i in
open_heap]:
                pre[neighbor] = current  # 更新前驱节点
                g[neighbor] = tentative_g  # 更新代价值
                f[neighbor] = tentative_g + heuristic(neighbor, end)  # 更新启发式函数
值
                heapq.heappush(open_heap, (f[neighbor], neighbor))
    # 如果open表为空，则表示无法到达终点，返回False
```

```
41      return False
```

获取起点与终点的坐标

使用`enumerate()`函数获取`maze`中每个元素的索引和对应的值

```
# 获取迷宫中的起点和终点
def find_start_end(maze):
    start = None
    end = None
    for i, row in enumerate(maze):  #i为行索引
        for j, val in enumerate(row): #j为列索引，val为元素值
            if val == "S":
                start = (i, j)
            elif val == "E":
                end = (i, j)
    return start, end
```

绘制含最短路径的迷宫

路径用“■”表示，并保留起点S和终点E

```
# 将结果绘制在迷宫上
def draw_path(maze, path):
    # 路径用"■"表示
    for step in path:
        if maze[step[0]][step[1]] == "S" or maze[step[0]][step[1]] == "E":
            continue
        maze[step[0]][step[1]] = "■"
    # 墙用"□"表示，可通行用” “表示
    for i, row in enumerate(maze):
        for j, val in enumerate(row):
            if val == 1:
                maze[i][j] = "□"
            elif val == 0:
                maze[i][j] = " "
    # 将行中的每个元素转换为字符串类型，并组合成新列表
    # 使用join()将列表中的元素用空格连接，形成新的字符串
    for row in maze:
        print(' '.join([str(elem) for elem in row]))
```

三、实验结果及分析

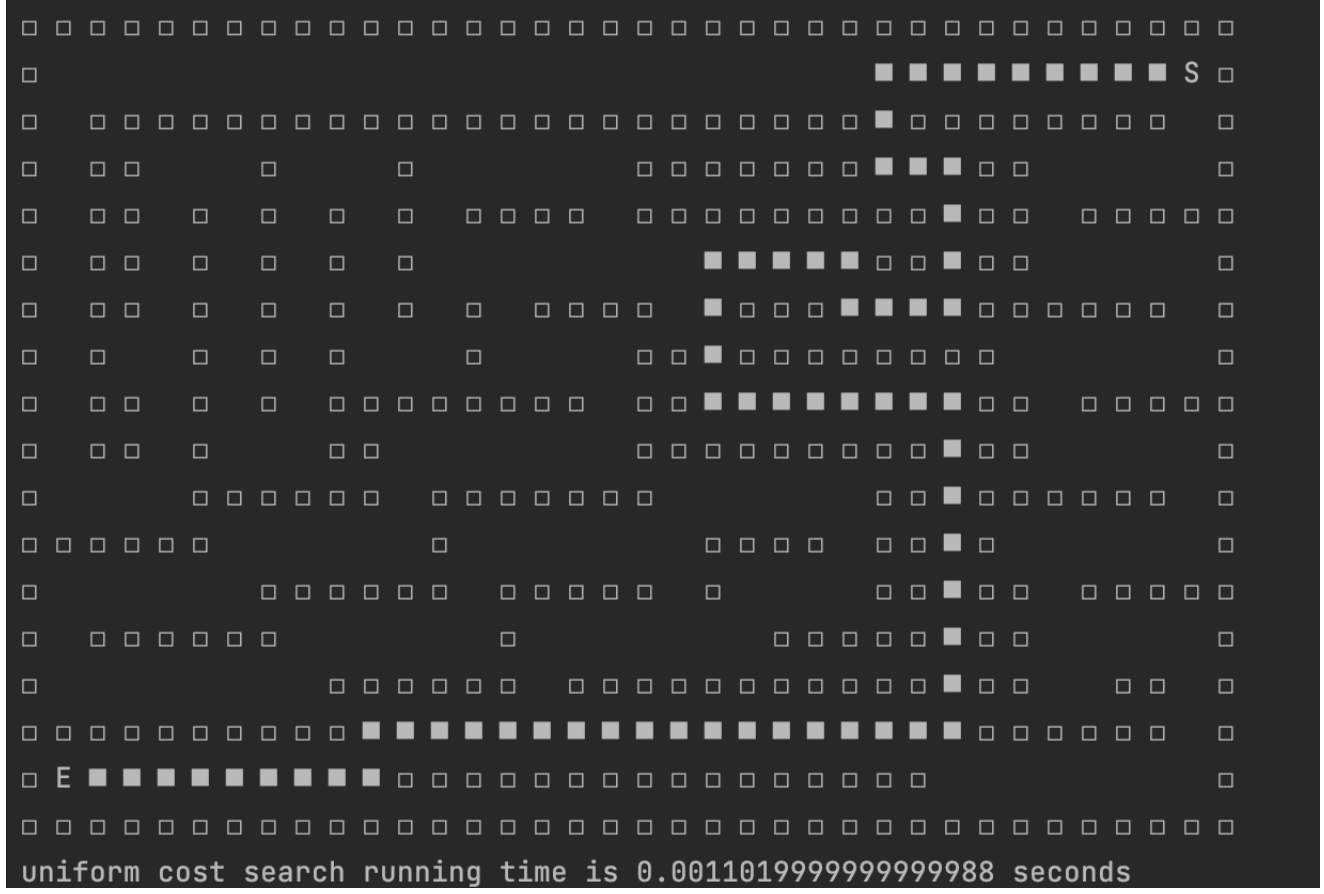
1.实验结果展示示例

路径用“■”表示，墙用”□“表示，可通行用” “表示

一致代价搜索

```
The path is
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1,

The maze is
```



```
uniform cost search running time is 0.0011019999999999988 seconds
```

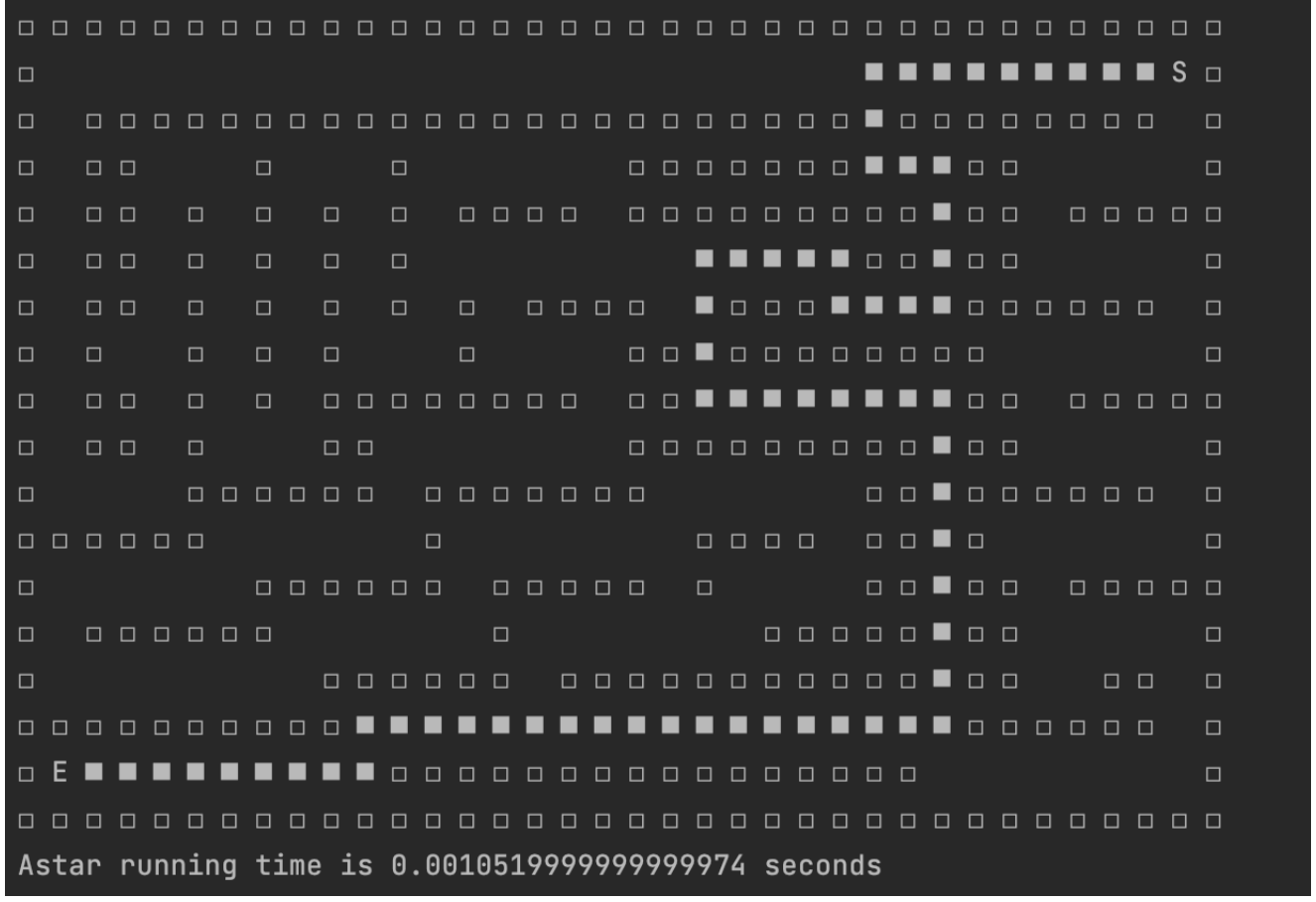
`path`为：（用坐标表示）

```
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1, 26),
(1, 25), (2, 25), (3, 25), (3, 26), (3, 27), (4, 27), (5, 27), (6, 27), (6, 26), (6,
25), (6, 24), (5, 24), (5, 23), (5, 22), (5, 21), (5, 20), (6, 20), (7, 20), (8,
20), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26), (8, 27), (9, 27), (10,
27), (11, 27), (12, 27), (13, 27), (14, 27), (15, 27), (15, 26), (15, 25), (15, 24),
(15, 23), (15, 22), (15, 21), (15, 20), (15, 19), (15, 18), (15, 17), (15, 16), (15,
15), (15, 14), (15, 13), (15, 12), (15, 11), (15, 10), (16, 10), (16, 9), (16, 8),
(16, 7), (16, 6), (16, 5), (16, 4), (16, 3), (16, 2), (16, 1)]
```

A*搜索

```
The path is
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1,

The maze is
```



```
Astar running time is 0.0010519999999999974 seconds
```

`path`为：（用坐标表示）

```
[(1, 34), (1, 33), (1, 32), (1, 31), (1, 30), (1, 29), (1, 28), (1, 27), (1, 26),
(1, 25), (2, 25), (3, 25), (3, 26), (3, 27), (4, 27), (5, 27), (6, 27), (6, 26), (6,
25), (6, 24), (5, 24), (5, 23), (5, 22), (5, 21), (5, 20), (6, 20), (7, 20), (8,
20), (8, 21), (8, 22), (8, 23), (8, 24), (8, 25), (8, 26), (8, 27), (9, 27), (10,
27), (11, 27), (12, 27), (13, 27), (14, 27), (15, 27), (15, 26), (15, 25), (15, 24),
(15, 23), (15, 22), (15, 21), (15, 20), (15, 19), (15, 18), (15, 17), (15, 16), (15,
15), (15, 14), (15, 13), (15, 12), (15, 11), (15, 10), (16, 10), (16, 9), (16, 8),
(16, 7), (16, 6), (16, 5), (16, 4), (16, 3), (16, 2), (16, 1)]
```

2.评测指标展示及分析

	完备性	最优性	时间复杂度	空间复杂度
一致代价搜索 UCS	Yes(树的分支因子有限&代价值为正)	Yes	$O(b^{1+\lfloor C*/\varepsilon \rfloor})$	$O(b^{1+\lfloor C*/\varepsilon \rfloor})$
A*搜索	Yes	$h(n)$单调且可采纳	$O(b^d)$	$O(b^d)$

```
uniform cost search running time is 0.0011019999999999988 seconds
Astar running time is 0.0010519999999999974 seconds
```

- 根据上述运行时间结果，可以看出一致代价搜索算法和A*搜索算法的运行时间非常接近，但A*搜索算法的运行时间略短一些，因为A*搜索算法利用了启发式函数来引导搜索过程，从而能够更快地找到最优解。
- 一致代价搜索算法尽可能地扩展到代价最小的节点，从而找到最优路径。但一致代价搜索没有使用启发式函数，因此它在找到最优路径时可能需要扩展更多的节点，导致运行时间略长。
- A*搜索算法使用了启发式函数来引导搜索过程，从而更快地找到最短路径。启发式函数用于估计从当前节点到目标节点的最短距离，A*搜索尽可能地扩展启发式函数值最小的节点，从而找到最优解。因此，A*搜索在找到最优路径时通常比一致代价搜索更快。

- A*搜索相对于一致代价搜索而言，具有更好的性能，能够更快地找到最优解。但是，在某些情况下，一致代价搜索可能比A*搜索更适合，例如：启发式函数不可用或不准确时，一致代价搜索可能比A*搜索更稳定和更可靠。