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）

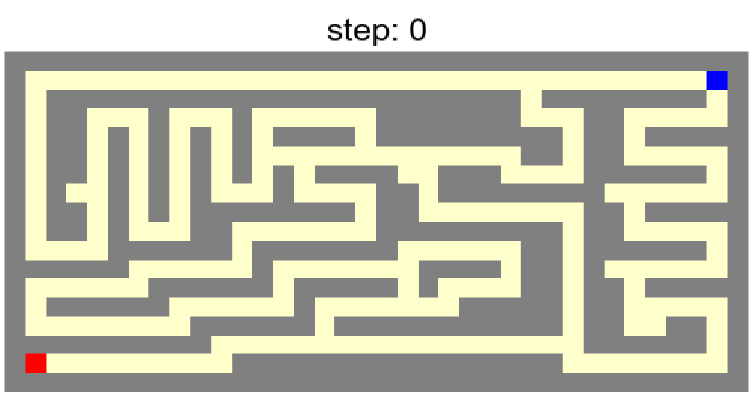
课程名称：Artificial Intelligence

教学班级	计科 1 班	专业（方向）	计算机科学与技术
学号	21307077	姓名	凌国明

一、实验题目

迷宫搜索问题

编写程序，通过传统搜索算法与启发式搜索算法，解决迷宫问题。
动作：四个动作，上下左右移动，成本都是 1
状态：所在位置（如果带路径检测，则是位置+路径）
目标：到达终点（下图红色点为终点，蓝色点为起点）



本次实验，我实现了

1. 宽度优先搜索，双向搜索，$A*$ 搜索
2. 深度优先搜索，迭代加深搜索，$IDA*$ 搜索

注意，迷宫问题中的四个动作（上下左右移动）成本一致。因此一致代价搜索的行为与宽度优先搜索是一致的，可以通过宽搜的表现分析一致代价搜索在迷宫问题中的表现。

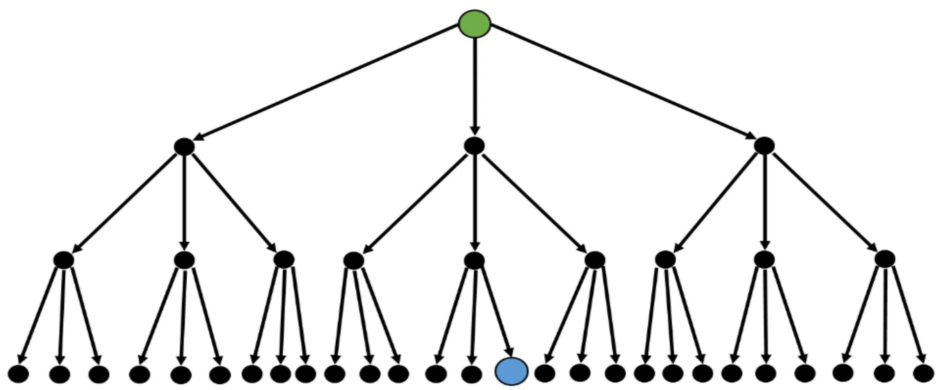
二、实验内容

1、算法原理

双向搜索

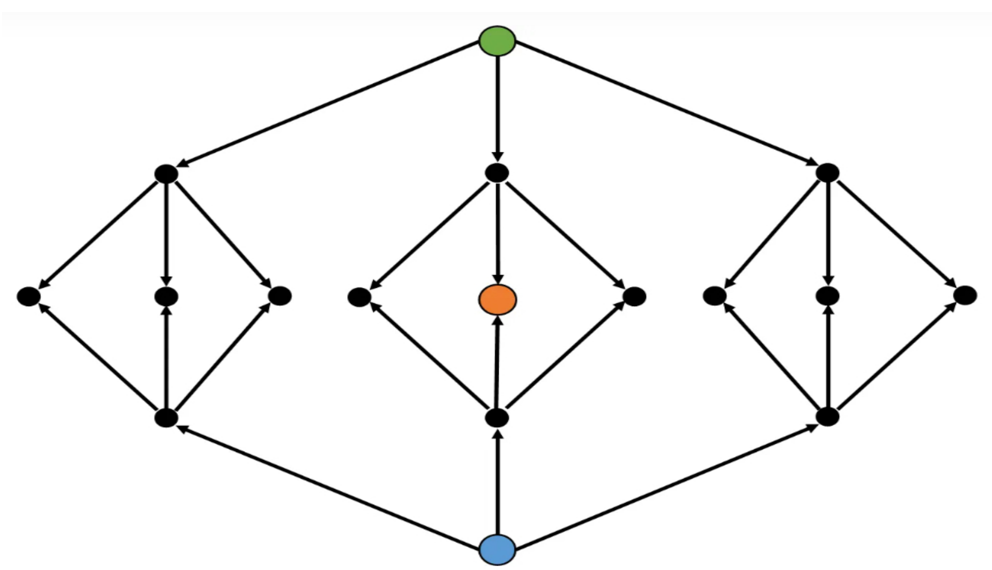
宽度优先搜索的缺点

设搜索树的度为 b，深度为 d，则广度优先搜索的时间复杂度、空间复杂度为 $O(b^d)$，是呈**指数级增长**的，当 d 较大时，时间和空间消耗增长巨大。



双向搜索的引出

既然起点终点都已知，那么我们从起点和终点开始，**分别进行广度优先搜索**，当两个搜索树**相遇时停止**。这样可以把时间复杂度、空间复杂度削减到 $O(b^{d/2})$



迭代加深搜索

深度优先搜索的缺点

传统的深度优先搜索**不能保证最优性**，在状态空间无限、不剪枝的情况下还不具有完备性。深度受限搜索不具有完备性和最优性。

迭代加深搜索的引出

广搜的空间复杂度太大，需要消耗很多空间。深搜的空间复杂度是线性的，但是深搜不能保证最优性。
迭代加深搜索**每次进行一轮深度受限搜索**，如果搜索**失败则提高成本边界**再进行深度受限搜索，直到搜索成功。在保证最优性的同时，尽量削减空间复杂度。

$A*$ 搜索

传统的搜索算法**只考虑了当前的成本**，也就是从起点到某个节点的实际代价；却**没有考虑这个节点的"前程"**，也就是这个节点到终点的代价。
$A*$ 搜索**同时考虑 起点到节点的实际代价 和 节点到终点的估计代价**，使得搜索具有一个特定的"方向"。这个方向在树搜索框架中，体现为对边界中各个状态的排序：维护一个**优先队列**，f(n)+g(n)最低的先出队列，这就完成了对边界的排序。其中 g(n) 表示起点到节点的实际代价，f(n)表示节点到终点的估计代价。

$IDA*$ 搜索

$A*$ 以广度优先为基础，空间复杂度较大。$IDA*$ 以深度优先为基础，与 迭代加深搜索 类似，与 迭代加深搜索 相比，只更改了边界的排序。迭代加深搜索中使用后入先出队列，$IDA*$ 搜索使用优先队列，f(n)+g(n)最低的先出队列。

启发式函数

可采纳性

设每个动作成本非负，且不能无穷小，设 $h(n)$表节点n的启发式函数值，$h^*(n)$表节点n到终点的实际成本。若 $h(n) <= h^*(n)$，则称$h(n)$是可采纳的。
如果一个启发式函数**具有可采纳性，则它具有最优性**（**不能带环检测**），因为最优解一定在 所有成本大于 最优解成本 的路径 之前 被扩展。但如果带了环检测，可能最优解的路径会被阻挡（见课件）

单调性

对于节点 n_1 和节点 n_2，若 $h(n_1) - h(n_2) <= cost(n_1, n_2)$，则启发式函数 $h(n)$ 是单调的。如果一个启发式函数是单调的，则无论用不用环检测，**都能保证最优性。**
易证四动作迷宫问题中，**曼哈顿距离的启发式函数具有单调性**，所以本实验的 $A*$算法可以使用环检测

2、伪代码，流程图

树搜索伪代码

Algorithm 1 树搜索

```
procedure 树搜索(边界, 后继状态函数, 目标)
    if 边界为空 then
        return 失败
    end if
    当前状态 = 从边界中选择一个状态
    if 当前状态 是 目标状态 then
        return 当前状态
    end if
    边界2 = (边界 - 当前状态) 并 当前状态的后继状态
    return 树搜索(边界2， 后继状态函数，目标)
end procedure
```

双向搜索伪代码

Algorithm 2 双向搜索

```
procedure 双向搜索(起点，目标，后继状态函数)
    初始化 先入先出队列
    将(起点, 'start')入队列
    将(终点, 'end')入队列
    while 队列非空 do
        now = 出队列
        if now 已经在另一颗搜索树中 then
            return 成功
        end if
        将now记录在相应的搜索树中
        for now的下个状态 do
            将(状态, 对应的树)入队列
        end for
    end while
    return 失败
end procedure
```

$A*$ 搜索伪代码

Algorithm 3 A*

```
procedure A*(起点)
    初始化 优先队列，f(n)+g(n)最低的先出队列
    计算起点的f(n)+g(n)，将起点入队列
    while 队列非空 do
        now = 出队列
        if now 是目标 then
            return 成功
        end if
        for now的下个状态 do
            计算状态的f(n)+g(n)，将状态入队列
        end for
    end while
    return 失败
end procedure
```

$IDA*$ 搜索伪代码（迭代加深搜索类似）

Algorithm 4 IDA*

```
procedure IDA*(起点)
    初始化 成本边界为 1
    while True do
        IDA*(起点，成本边界)
        if 搜索成功 then
            return 成功
        end if
        成本边界+=1
    end while
end procedure
procedure 深度受限搜索(起点，成本边界)
    初始化 优先队列，f(n)+g(n)最低的先出队列
    计算起点的f(n)+g(n)，将起点入队列
    while 队列非空 do
        now = 出队列
        if now 的g(n) > 成本边界 then
            continue
        end if
        if now 是目标 then
            return 成功
        end if
        for now的下个状态 do
            计算状态的f(n)+g(n)，将状态入队列
        end for
    end while
    return 失败
```

end procedure

3、关键代码展示（带注释）

迷宫类

```
class Maze:
    def __init__(self, data_path):
        # 打开 MazeData 文件
        file = open(data_path, mode='r', encoding='ascii')
        # 颜色，省略了
        # 读取 MazeData 文件
        for line in file:
            tmp = (list(line.strip('\n')))
            self.maze_data.append(tmp)
        # 画图展示 迷宫状态
        self.maze_board = np.zeros((len(self.maze_data), len(self.maze_data[0]), 3), dtype=np.ui
        for i in range(len(self.maze_data)):
            for j in range(len(self.maze_data[0])):
                # 如果是 起点、终点、墙等，设置相应的值和颜色
        self.show_board(1)

    # 移动到这个位置是否合法（可设置环检测）（路径检测代码类似，省略）
    def is_valid(self, pos, circle_detection=True):
        if 0 < pos[0] < len(self.maze_data) and 0 < pos[1] < len(self.maze_data[0]):
            # 不能是墙。如果带环检测，就不能是访问过的。
            if self.maze_data[pos[0]][pos[1]] != 1 and ((not circle_detection) or self.maze_data
                return True
        return False

    # 设置状态（已访问，未访问，当前状态）
    def set_status(self, pos, status):
        if status == 'now':
            self.maze_data[pos[0]][pos[1]] = 3
        elif status == 'visited':
            self.maze_data[pos[0]][pos[1]] = 2
        elif status == 'unvisited':
            self.maze_data[pos[0]][pos[1]] = 0

def next_stage(pos):
    return (pos[0] + 1, pos[1]), (pos[0], pos[1] + 1), (pos[0] - 1, pos[1]), (pos[0], pos[1] - 1
```

双向搜索

```python
# 双向搜索
def double_ended_search(maze: Maze):
    board = [[' ' for j in range(len(maze.maze_data[0]))] for i in range(len(maze.maze_data))]

    que = queue.Queue()
    que.put((maze.start, 's'))
    que.put((maze.end, 'e'))

    step_que = queue.Queue()
    step_que.put(0)
    step_que.put(0)
    # 记录在队列中的元素，不让相同的元素进入队列
    in_queue_element = set()
    in_queue_element.add((maze.start, 's'))
    in_queue_element.add((maze.end, 'e'))

    total_search_step = 0
    while not que.empty():
        total_search_step += 1
        now, now_state = que.get()
        now_step = step_que.get()
        in_queue_element.remove((now, now_state))

        if now_state == 's' and board[now[0]][now[1]] == 'e' or now_state == 'e' and board[now[0
            break
        # 当前位置为 now
        maze.set_status(now, 'now')
        if now_state == 's':
            maze.show_board(0.02, 2*now_step-1)
        else:
            maze.show_board(0.02, 2 * now_step)
        # 将位置为 now 的方格 设置为 已访问
        maze.set_status(now, 'visited')
        board[now[0]][now[1]] = now_state
        # 下一个状态
        for stage in next_stage(now):
            if maze.is_valid(stage) and (stage, now_state) not in in_queue_element:
                in_queue_element.add((stage, now_state))
                que.put((stage, now_state))
                step_que.put(now_step+1)
    if now_state == 's':
        maze.show_board(2, 2 * now_step - 1)
        return 2*now_step-1, total_search_step
    else:
        maze.show_board(2, 2 * now_step)
        return 2 * now_step, total_search_step
```

$A*$ 搜索

```python
def a_star(maze: Maze):
    # 优先队列，按启发式函数值排列，第0个元素的函数值，第1个元素是位置
    # 第2个元素是从初始节点到这个节点的最小代价，第3个元素是路径
    que = queue.PriorityQueue()

    # 启发式函数
    hn_now = h_n(maze.start, maze.end)

    # 排序函数值为 hn_now + 0，位置是 maze.start，从起点到 maze.start 代价是 0， 路径为 {}
    que.put((hn_now + 0, maze.start, 0, set([])))

    # 记录在队列中的元素，不让相同的元素进入队列
    in_queue_element = set()
    in_queue_element.add(maze.start)

    total_search_step = 0
    while not que.empty():
        total_search_step += 1
        now, now_cost, now_path = que.get()[1:4]
        in_queue_element.remove(now)
        # 当前位置为 now
        maze.set_status(now, 'now')
        maze.show_board(0.02, now_cost)
        if maze.is_goal(now):
            maze.show_board(2, now_cost)
            break
        # 将位置为 now 的方格 设置为 已访问
        maze.set_status(now, 'visited')
        # 下一个状态
        for stage in next_stage(now):
            # 带环检测可以得到最优解（h_n单调）
            if maze.is_valid(stage, circle_detection=True) and stage not in in_queue_element:
                # 在路径中添加pos
                hn_stage = h_n(stage, maze.end)
                stage_path = now_path.copy()
                stage_path.add(now)
                que.put((hn_stage+now_cost+1, stage, now_cost+1, stage_path))
                in_queue_element.add(stage)
    return now_cost, total_search_step
```

$IDA*$ 搜索

```python
def id_a_star(maze: Maze, begin=1, factor=1, add_now_cost=True):
    if begin == 1:
        begin = h_n(maze.start, maze.end)
    total_search_step = 0
    res = -1
    counter = begin
    while res == -1:
        res, step = id_a_star_limit(maze, counter, add_now_cost=True)
        total_search_step += step
        maze.reset()
        counter += factor
    return res, total_search_step

def id_a_star_limit(maze: Maze, limit):
    # 排序函数 中 加不加 (从初始节点 到 每个已探索节点 的最小代价)
    # 记录位置
    sta = queue.LifoQueue()
    # 记录代价
    depth_sta = queue.LifoQueue()
    # 记录路径
    path_sta = queue.LifoQueue()

    sta.put(maze.start)
    depth_sta.put(0)
    path_sta.put(set())

    total_search_step = 0
    while not sta.empty():
        total_search_step += 1
        now = sta.get()
        now_depth = depth_sta.get()
        now_path = path_sta.get()

        if now_depth > limit:
            continue

        # 当前位置为 now
        maze.set_status(now, 'now')
        maze.show_board(0.02, now_depth)
        if maze.is_goal(now):
            maze.show_board(2, now_depth)
            break
        # 将位置为 now 的方格 设置为 已访问
        maze.set_status(now, 'visited')

        # 下个状态 的 启发式函数值+已知代价
        hn_plus_gn = [0, 0, 0, 0]
        stages = next_stage(now)
```

```python
        # 下一个状态 优先扩展 启发式函数值+已知代价 最小的 状态
        for i in range(len(stages)):
            # 带环检测得不到最优解
            # 带路径检测
            if maze.is_valid2(stages[i], now_path, path_detection=True):
                # 下个状态 的 启发式函数值+已知代价
                hn_plus_gn[i] = h_n(stages[i], maze.end) + now_depth + 1

        while max(hn_plus_gn) > 0:
            max_pos = hn_plus_gn.index(max(hn_plus_gn))
            # 在路径中添加pos
            hn_stage = h_n(stages[max_pos], maze.end)
            stage_path = now_path.copy()
            stage_path.add(now)
            sta.put(stages[max_pos])
            depth_sta.put(now_depth + 1)
            path_sta.put(stage_path)
            hn_plus_gn[max_pos] = 0

    if maze.is_goal(now):
        return now_depth, total_search_step
    return -1, total_search_step
```

4、创新点&优化

4.1 环检测与路径检测

本实验中，用到环检测的搜索算法有：广度优先搜索，双向搜索，$A*$ 搜索，深度优先搜索
用到路径检测的算法有：深度受限搜索，迭代加深搜索，$IDA*$ 搜索

设 m 和 n 为迷宫的高度和宽度。

环检测

设置一个 $m \cdot n$ 的矩阵记录搜索过程中，某个节点有没有被访问过。空间复杂度为 $O(m \cdot n)$，时间复杂度为 $O(1)$

路径检测

每个状态附带一个 set，记录这个节点路径上的所有祖先节点。set 底层通过 $hash$ 实现，时间复杂度为 $O(1)$

本质是**以空间换时间**，用更高的空间复杂度换取 $O(1)$ 的执行时间

4.2 探索不同的迷宫

为探索各个算法的性能 以及 算法效率随输入规模提升的变化，**进行了不同规模迷宫的探索**，分别是 $18*36$，$40*60$，$80*120$。
通过深度优先搜索生成随机迷宫，迷宫中有多个路径通向终点。（GPT4写的代码，具体见Code部分的maze_generate）探索结果见结果展示部分。

4.3 迭代加深搜索的讨论

在探索不同规模的迷宫时，发现 迭代加深搜索 和 $IDA*$ 搜索的 **执行时间都远大于其他搜索算法。**
经过思考，认为有两种办法可以缩短运行时间

1. **将刚好超出成本边界的点存起来**，等到下一轮深度受限搜索的时候 取出来使用，这样就可以避免重复搜索。但是这就跟 $A*$ 算法的行为一致了，而且**抛弃了空间复杂度低的优点**，背离了 $IDA*$ 算法的初衷。
2. **每次额外扩展成本边界。**在本实验中，成本边界体现为搜索格子的深度限制，成本边界每轮搜索后+=1，这样可以保证最优性。**如果每次将成本边界+2**，**则可以更快地找到解**（但是**不保证最优性**）

基于第二点的想法，在 $80*120$ 的迷宫上进行了实验，采用的是 **不同步长 的迭代加深算法**，$IDA*$ 算法也同理。

步长	结果长度	探索节点数	运行时间/s
1	560	294252413	5449.38
2	560	147674509	2748.96
4	562	75166421	1401.28
8	562	37975360	705.89
16	562	19381641	360.39
32	574	9308508	172.48
64	574	4099417	76.00
128	574	1619190	30.01

由上表可见，**步长翻倍的同时，搜索节点数折半减少，运行时间也折半减少。**因此认为采用“步长”是**加速迭代加深搜索**的有效手段。
易证这种迭代加深搜索算法得出的结果 < 最优结果 + 步长，也就是说带步长的迭代加深搜索得出的结果最坏情况（**上界**） 是 **最优结果 + 步长**。在步长不大的情况下，解的**误差不大**，可以接受。

4.4 可视化

利用 $matplotlib$ 中的 $pyplot$ 进行迷宫的可视化，将算法的全过程和最终结果进行可视化展示，更利于算法过程的透视理解。

三、实验结果与分析

1、实验结果展示示例

表1 迷宫一 18*36

算法	广度优先	双向搜索	深度优先	迭代加深	$A*$	$IDA*$
结果长度	68	68	174	68	68	68
探索节点数	270	194	177	11947	221	11947
运行时间/ms	3.13	2.32	1.99	181.97	2.31	201.11

表2 迷宫二 40*60

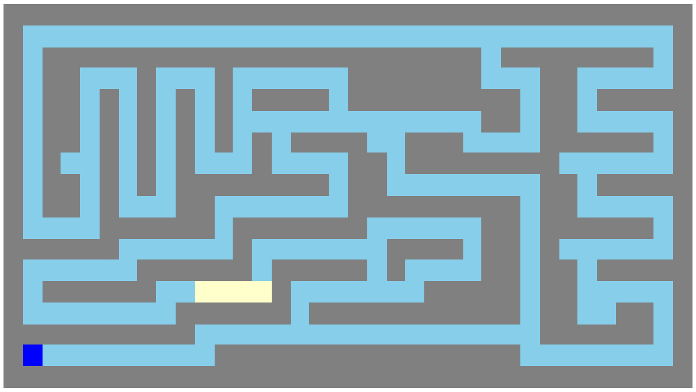
算法	广度优先	双向搜索	深度优先	迭代加深	$A*$	$IDA*$
结果长度	296	296	554	296	296	296
探索节点数	656	1153	705	363924	411	366937
运行时间/ms	7.41	13.34	8.11	6345.05	4.80	7003.05

表3 迷宫三 80*120

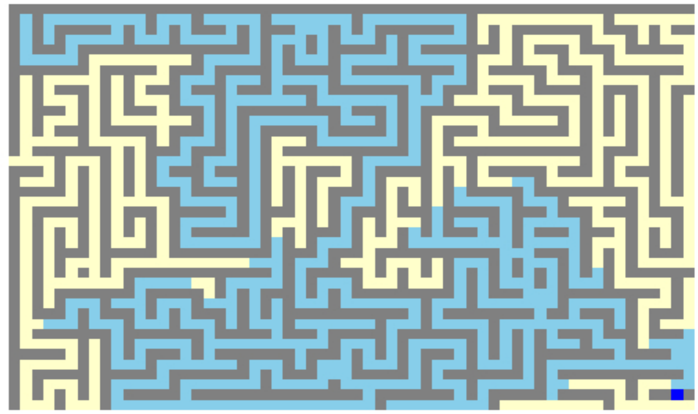
算法	广度优先	双向搜索	深度优先	迭代加深	$A*$	$IDA*$
结果长度	560	560	1362	560	560	560
探索节点数	2975	2965	4144	294252413	2660	292075820
运行时间/ms	33.77	52.57	67.07	5588319.76	60.54	6308396.21

宽度优先搜索在 18*36 和 40*60 的迷宫中的表现

step: 68

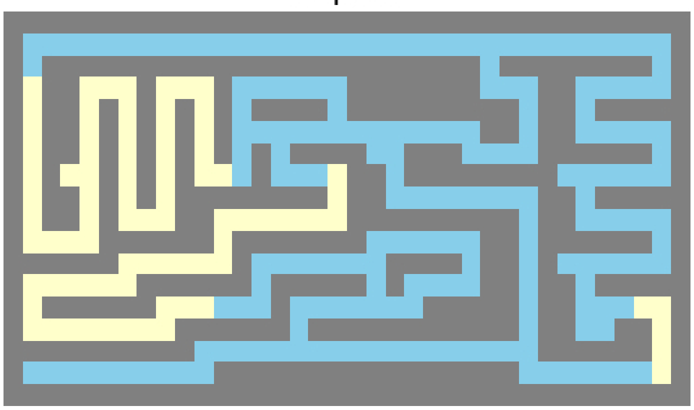


step: 296

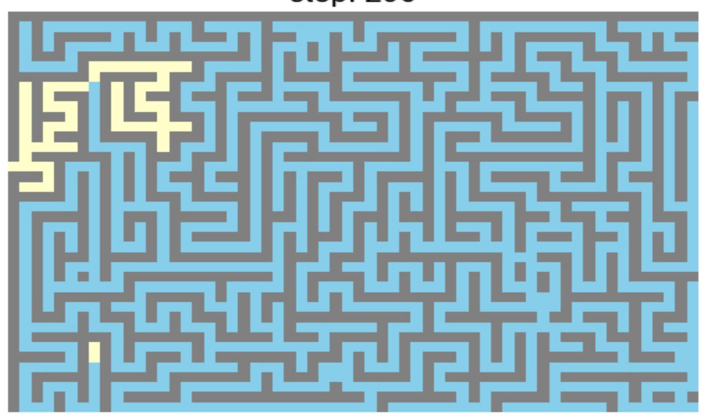


双向搜索搜索在 18*36 和 40*60 的迷宫中的表现

step: 68

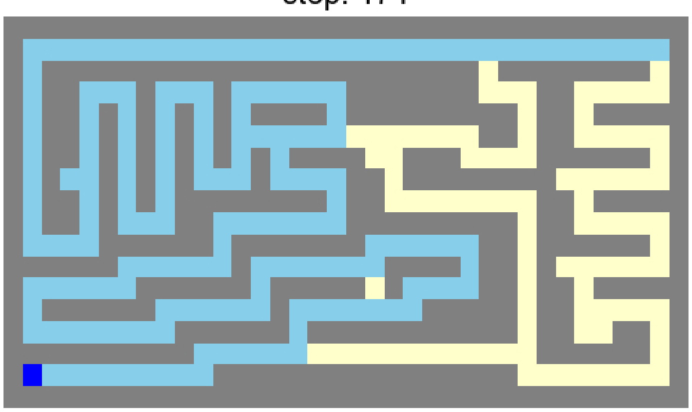


step: 296

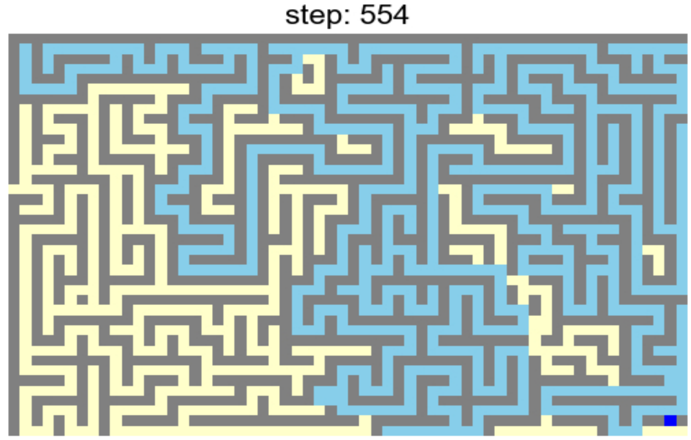


深度优先搜索在 18*36 和 40*60 的迷宫中的表现

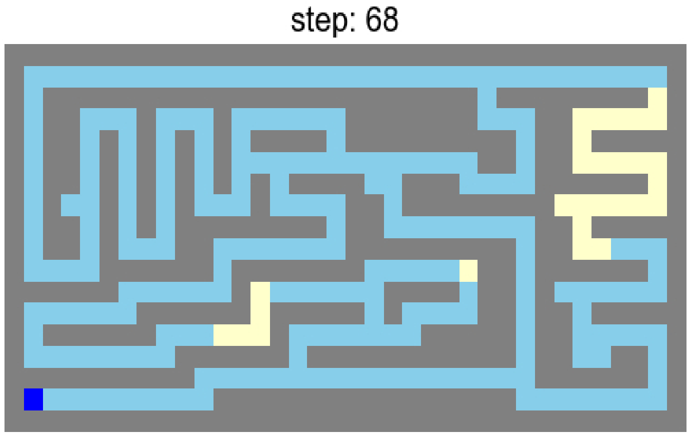
step: 174

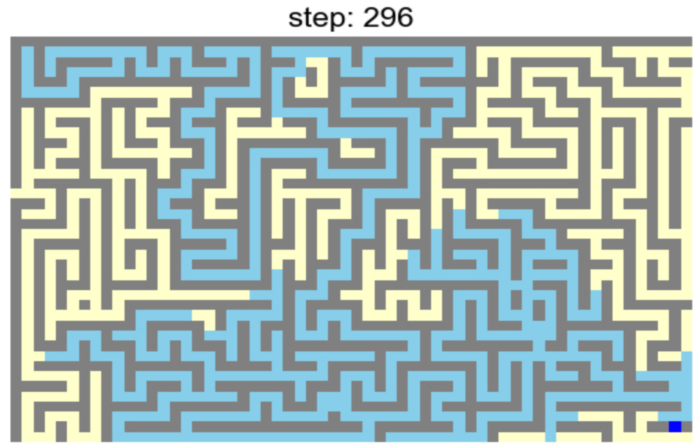


step: 554

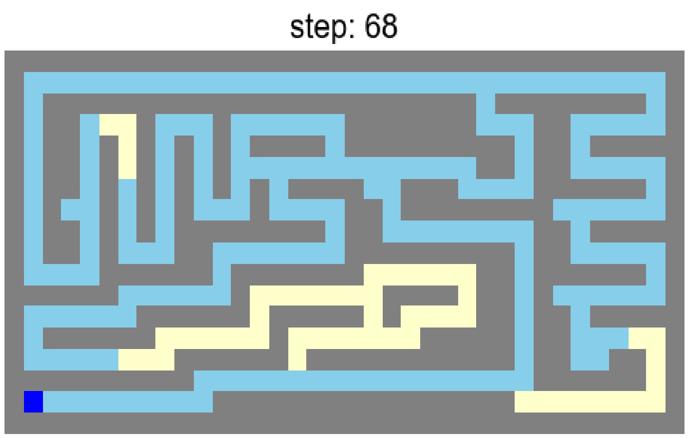


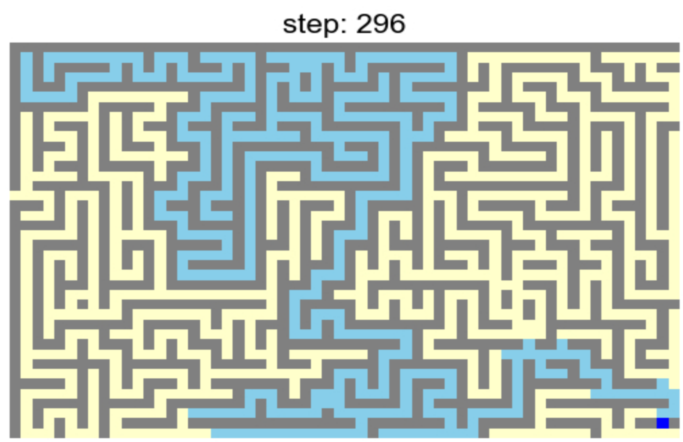
迭代加深搜索在 18*36 和 40*60 的迷宫中的表现



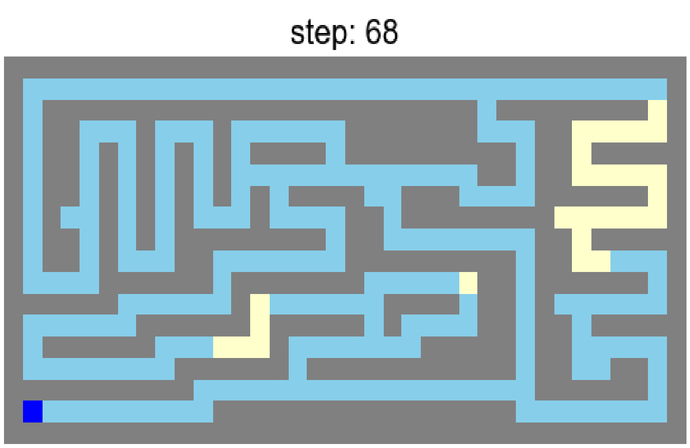


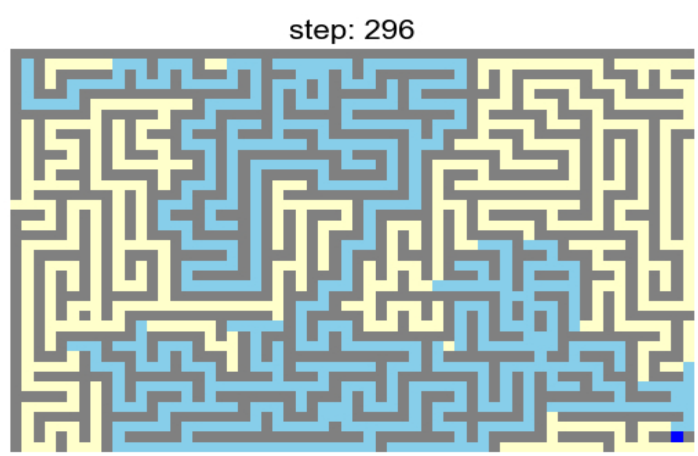
$A*$ 搜索在 18*36 和 40*60 的迷宫中的表现





$IDA*$ 搜索在 18*36 和 40*60 的迷宫中的表现





2、运行时间分析

理论上各个搜索算法的时空复杂度

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

实际上各算法的运行时间

表1 迷宫一 18*36

算法	广度优先	双向搜索	深度优先	迭代加深	$A*$	$IDA*$
探索节点数	270	194	177	11947	221	11947
运行时间/ms	3.13	2.32	1.99	181.97	2.31	201.11

表2 迷宫二 40*60

算法	广度优先	双向搜索	深度优先	迭代加深	$A*$	$IDA*$
探索节点数	656	1153	705	363924	411	366937
运行时间/ms	7.41	13.34	8.11	6345.05	4.80	7003.05

表3 迷宫三 80*120

算法	广度优先	双向搜索	深度优先	迭代加深	$A*$	$IDA*$
探索节点数	2975	2965	4144	294252413	2660	292075820
运行时间/ms	33.77	52.57	67.07	5588319.76	60.54	6308396.21

1. **双向搜索并不总比广度优先搜索好**：当终点展开的搜索树 远大于 起点展开的搜索树时，双向搜索的效率会下降。
2. $A*$ 搜索能**更好地指导搜索的方向**，所以搜索的总节点数相对来说都较少。但是当搜索的**规模较大时**，**维护优先队列的开销较大**，这延长了运行时间。
3. 迭代加深搜索和 $IDA*$ 搜索运行时间都很长，这是因为存在大量的重复搜索。创新点的4.3部分对这点作了讨论。

四、参考资料

https://zhuanlan.zhihu.com/p/119349440 双向搜索
https://chat.openai.com/chat GPT4写的生成迷宫部分代码