A* 算法实验报告

2023200440

2024 年 9 月 26 日

1 摘要

AI 引论课程上学习了深度搜索，宽度搜索和 A* 算法的原理、实现和各自优缺点。本次实验基于已有的 basecode 实现 A* 算法细节，对数字华容道问题求解，从而加深合把握对启发式搜索和评估函数在搜索算法中的应用。本实验通过对比不同 h 函数的计算，探索 h 值计算策略对启发信息强度和搜索效率的影响。最后对实验中的一些探索和结果做了分析和讨论。

2 引言

本实验在 besecode 基础上完成 A* 算法具体实现，并对 A* 算法 h 函数的不同方法进行对比。除此以外，拓展探索更高阶的华容道判断是否可解和求解，以及对多个解的情况返回前 k 个解。

3 实验方法

3.1 实验环境

环境：windows11
解释器：python3.12
编辑器：pycharm

3.2 实验步骤

1. PuzzleBoardState 类的修改；（1）增加 f 和 g 属性
（2）增加 _init_data 方法, 对 data 是否为空和可解进行判断，以及前两个条件均为否时随机初始化一个可解状态
（3）增加 _get_h1 方法：通过不在位数字个数（除 0 外）计算当前状态到目标状态的估计代价
（4）增加 _get_h2 方法：通过不在位数字距离和（除 0 外）计算当前状态到目标状态的估计代价
（5）增加 _get_g 方法：计算初始状态到当前状态的耗散值
（6）修改 _inverse_num 方法: 增加对偶数阶逆序数计算（偶数阶逆序数加上 0 所在行与目标行的差）
（7）修改 _if_solvable: 目标状态给定，逆序数为 0，故只判断初始状态的总逆序数
（8）重载 ___lt___ 方法：open 表使用最小堆结构，便于能够通过 f 值进行排序

2. A* 算法实现；

```python
def astar(puzzle_board_state, k=3):
    # TODO
    # A*算法实现
    open_list = []  # open列表存储待访问节点
    heapq.heapify(open_list)   # 初始化最小堆
    closed_set = set()  # closed列表存储已访问过的节点

    # 将初始节点放入open表
    heapq.heappush(open_list, puzzle_board_state)

    ans = []
    count = 0
    while open_list:  # 防止递归深度过大，栈空间已满
        cur_state = heapq.heappop(open_list)  # 弹出f值最小的节点，计算哈希值并放入clos表
        closed_set.add(cur_state.get_data_hash())

        # 判断是否结束
        if cur_state.is_final():
            while cur_state:   # 开始回溯父节点
                ans.append(cur_state)
                cur_state = cur_state.get_parent()
            return ans[::-1]
        # 获取相邻状态
        next_states = cur_state.next_states()

        for next_state in next_states:
            if next_state.get_data_hash() in closed_set:
                continue
            heappush(open_list, next_state)

    return None
```

图 1: A* 算法实现

3. 可视化；简单的对 3 阶华容道做了可视化，效果如下：

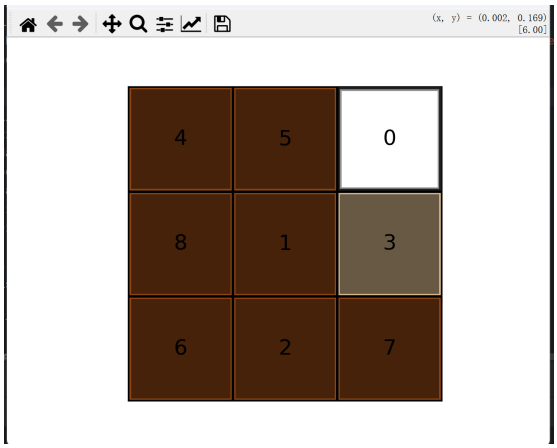

图 2: 可视化

4. 两种 h 函数运行效率比较见讨论与分析

5. 多解问题和高阶问题探索（1）多解问题：用 result_ans 列表存储每个解将，结束循环后返回，但实际中遇到了 bug，暂未解决，具体说明在讨论与分析。

（2）更高阶：修改目标状态生成方式，适用 N 阶

```
1    self.default_dst_data = np.array([j for j in range(1, self.dim ** 2)]
         ↪ + [0]).reshape(self.dim, self.dim) #目标状态
```

运行时间指数级上升。对简单的 4 阶华容道能够成功较快运行，随机华容道则耗时过长

...

4 代码细节

（1）h 函数的实现

```
1    def _get_h1(self):
2        """
3        计算h(n)的值，即当前状态到目标状态的估计代价，
4        h(n)=不在位的数字个数
5        """
6        h = 0
7        flatten_data = self.data.flatten()
```

```
        flatten_dst = self.default_dst_data.flatten()
        for j in range(self.dim ** 2):
            if flatten_data[j] == 0: # 0视作空格
                continue
            if flatten_data[j] != flatten_dst[j]:
                h += 1
        return h

    def _get_h2(self):
        """
                计算h(n)的值，即当前状态到目标状态的估计代价，
                h(n)=不在位的数字到目标位置的距离和
                """
        h = 0
        flatten_data = self.data.flatten()
        flatten_dst = self.default_dst_data.flatten()
        for j in range(self.dim ** 2):
            if flatten_data[j] == 0: # 0视作空格
                continue
            num = flatten_data[j]
            if num != flatten_dst[j]:
                h += abs(j // self.dim - num // self.dim) + abs(j % self.
                    ↪ dim - num % self.dim)
        return h
```

5 实验结果

5.1 结果展示

实验结果如下：

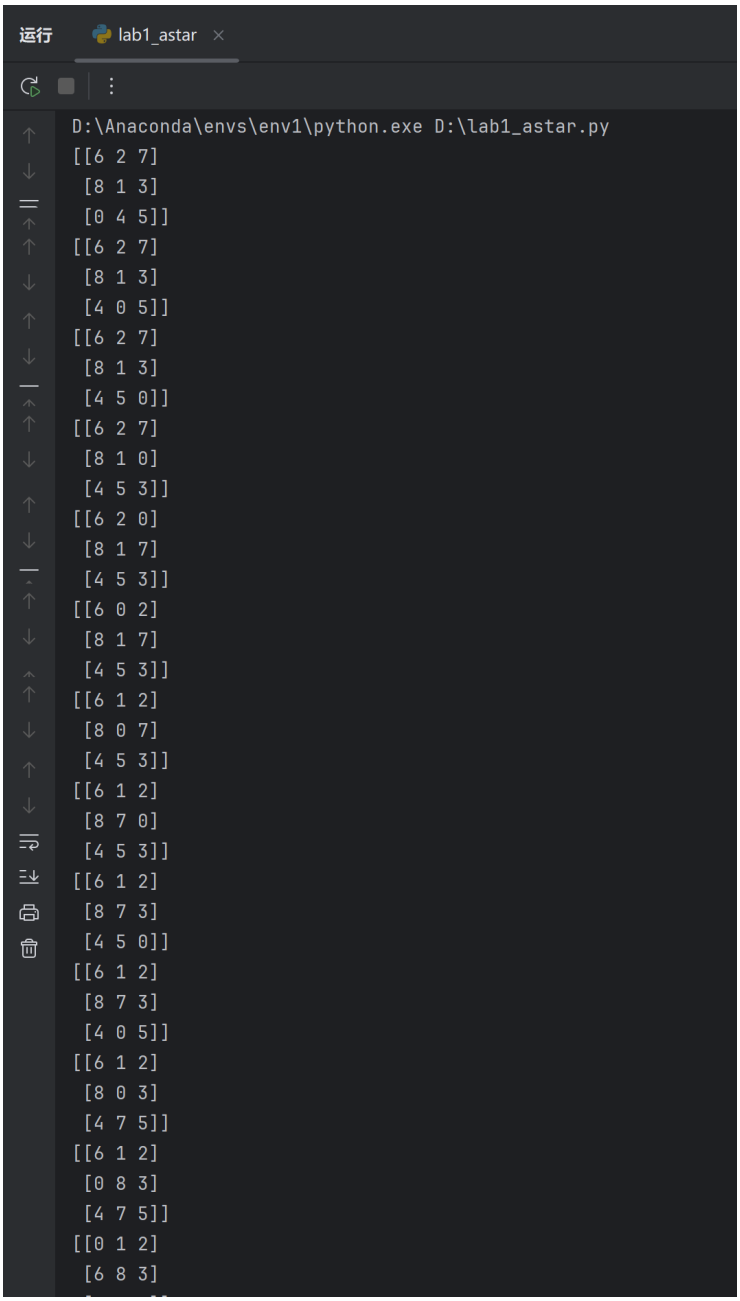
运行
lab1_astar
D:\Anaconda\envs\env1\python.exe D:\lab1_astar.py
[[6 2 7]
[8 1 3]
[0 4 5]]
[[6 2 7]
[8 1 3]
[4 0 5]]
[[6 2 7]
[8 1 3]
[4 5 0]]
[[6 2 7]
[8 1 0]
[4 5 3]]
[[6 2 0]
[8 1 7]
[4 5 3]]
[[6 0 2]
[8 1 7]
[4 5 3]]
[[6 1 2]
[8 0 7]
[4 5 3]]
[[6 1 2]
[8 7 0]
[4 5 3]]
[[6 1 2]
[8 7 3]
[4 5 0]]
[[6 1 2]
[8 7 3]
[4 0 5]]
[[6 1 2]
[8 0 3]
[4 7 5]]
[[6 1 2]
[0 8 3]
[4 7 5]]
[[0 1 2]
[6 8 3]

图 3: 实验结果截图

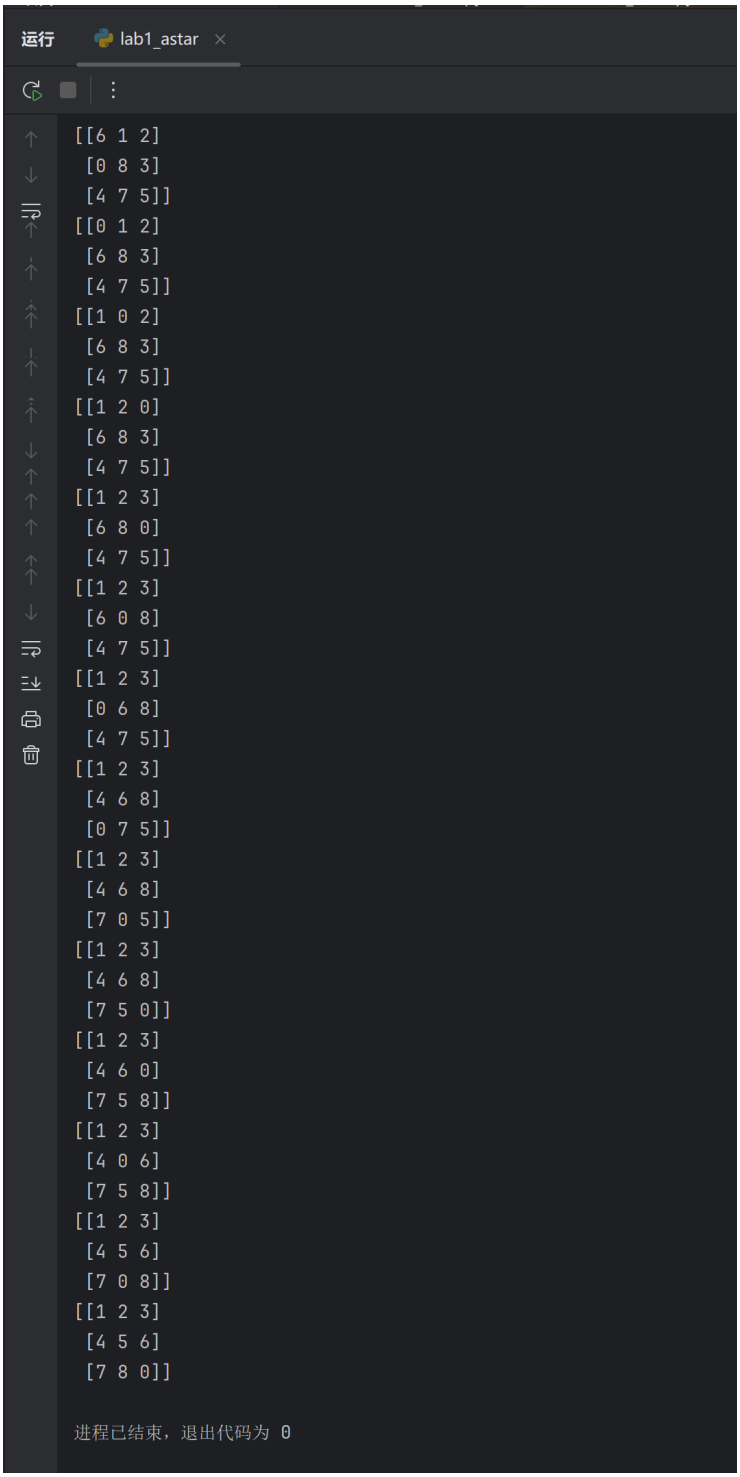
运行
lab1_astar
[[6 1 2]
[0 8 3]
[4 7 5]]
[[0 1 2]
[6 8 3]
[4 7 5]]
[[1 0 2]
[6 8 3]
[4 7 5]]
[[1 2 0]
[6 8 3]
[4 7 5]]
[[1 2 3]
[6 8 0]
[4 7 5]]
[[1 2 3]
[6 0 8]
[4 7 5]]
[[1 2 3]
[0 6 8]
[4 7 5]]
[[1 2 3]
[4 6 8]
[0 7 5]]
[[1 2 3]
[4 6 8]
[7 0 5]]
[[1 2 3]
[4 6 8]
[7 5 0]]
[[1 2 3]
[4 6 0]
[7 5 8]]
[[1 2 3]
[4 0 6]
[7 5 8]]
[[1 2 3]
[4 5 6]
[7 0 8]]
[[1 2 3]
[4 5 6]
[7 8 0]]
进程已结束，退出代码为 0

图 4: 实验结果截图

6 讨论与分析

（1）多解问题实现报错

```
    File basicstyle"basicstyleDbasicstyle:\basicstylelab1_astar
        ↪ basicstyle.basicstylepybasicstyle", line 276, in astar
  next_states = cur_state.next_states()
                ^^^^^^^^^^^^^^^^^^^^^
AttributeError: basicstyle'basicstyleNoneTypebasicstyle' object has no
    ↪ attribute basicstyle'basicstylenext_statesbasicstyle'}
```

调试情况：通过设置断点查看 cur_state 变量和打印 dir(cur_state)，均显示 cur_statePuzzleBoardState 类的方法和属性，但下文报错显示无 next_states 属性，尝试打印其他属性和方法出现同意问题，但可以打印 data 属性。且代码在 3 阶华容道运行没有问题，因此应该可以排除代码问题。无法解决该问题。

（2）修改指向 (父节点) 错误判断 next_state 是否在 open 表中，若在且 g 值更低，将 open 表里相同的 state 的父节点设置为 next_state 的父节点，更新 h 值和 f 值，但未更新 state 子节点的 f 值和 g 值，陷入死循环。

```
def renew_open_list(next_states, open_list, closed_set):
"""
更新open_list表
"""
for next_state in next_states:
    renew = True
    if next_state.get_data_hash() in closed_set: # 节点已搜索过
        continue
    # 只修改当前节点g和f值，没有修改子节点，导致报错
    # for state in open_list:
    # if np.array_equal(state.data, next_state.data):
    # if next_state.g < state.g: # 到当前节点有更优路径
    # state.parent = next_state.parent # 只需将state的父节点改为
        ↪ next_state的父节点
    # state.g = next_state.g
    # state.f = next_state.f
    # renew = False
    # break

```

```
19          if renew:
20              heappush(open_list, next_state)
```

（3）不同 h 函数效率

运行时间如下：

h2：0.3251086999953259

h1 :1.0499829000036698

h2 函数运行时间明显缩短，启发信息强度更高。

7 参考文献

参考以下博客：

https://blog.csdn.net/Zhouzi_heng/article/details/115035298

https://blog.csdn.net/qq_41829060/article/details/98030200

https://www.jianshu.com/p/1c1849d876b2