人工智能实验——A*算法 实验报告

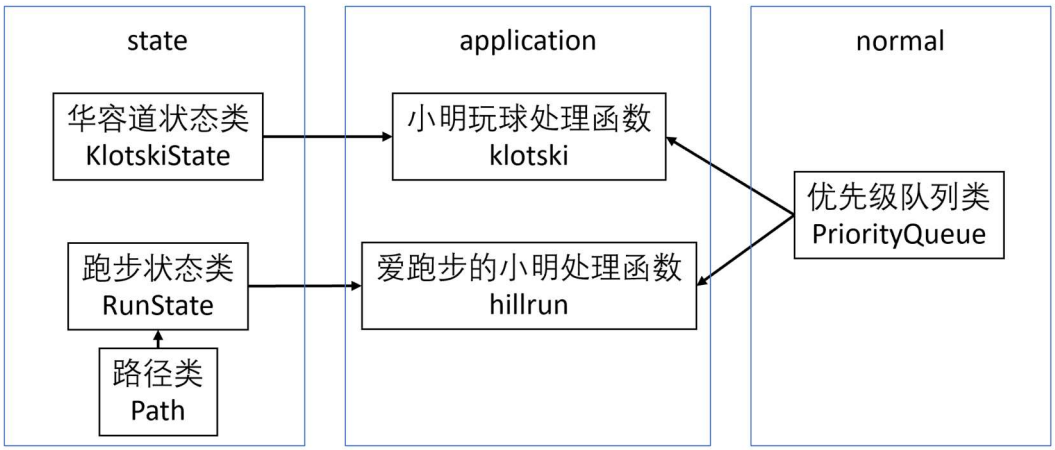
姓名：	郑逸潇	学号：	10192100406

实验目标：

使用 A*算法解决两个问题：小明玩球（数字华容道）、爱跑步的小明（路径规划）。

简介：

本次实验我将其主要分解成三个结构：通用层优先级队列类、状态层 State 类、应用层函数，应用层调用通用层和状态层接口实现解决问题的函数，以下为示意图：



结构图 1 整体架构设计

实验环境：

VSCode、Python

实验过程：

主体结构设计：

主体结构设计如结构图 1 所示，按此设计是因为优先级队列可以构建后重

复使用，两个状态类分开则是为了更清晰的条理，没必要硬整合成一个类，应用函数与主函数分开是为了多次调用，可以一次性测试所有案例。测试方法共有两种：修改 input.txt 使用命令行解析 input.txt；直接在命令行输入测试案例，具体测试方法见 README.md。

优先级队列设计：

以小顶堆数据结构为基础实现优先级队列，由于本实验的队列一开始为空，因此不需要《算法导论》中的构建堆函数，只需要两个堆调整函数即可：siftup 和 siftdown，由于本实验并非数据结构实验，因此不过多赘述，在代码中有相应注释。优先级队列类提供四个接口：push（入队，需传入 f，state）、pop（出队，返回 f，state）、length（队列元素个数）、empty（队列是否为空）。

状态类设计：

KlotskiState（小明玩球问题）状态类：

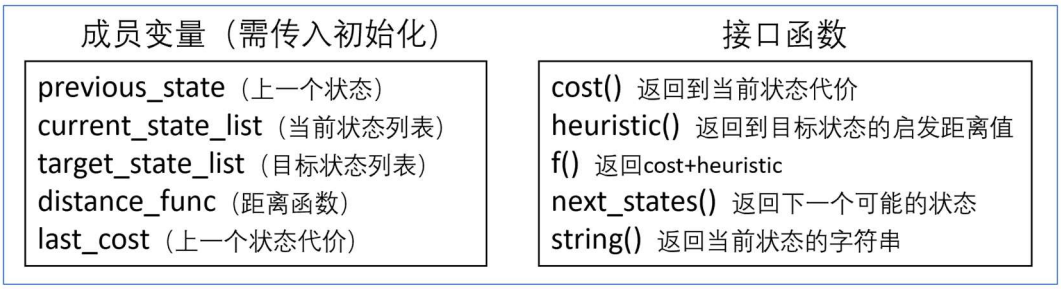
由于小明玩球问题的状态转移时每一步的代价均相同，因此主要就是**设计启发函数（Heuristic Function）**，在本次实验中，我将启发函数的距离计算方法定义在应用层，状态层仅使用应用层传入的距离计算方法来实现启发函数，因此可以在应用层切换曼哈顿距离、欧几里得距离、抑或为正确位置的个数等，可以在应用层切换不同距离函数定义来更改启发函数，具体实现参考应用函数设计。

由于我将数据以一维列表形式存储，所以在距离函数实现时需要将列表状态进行转换，转为 row、col 坐标再计算距离。

为了使得应用层方便调用接口，在状态层中我实现了 next_states 接口，返回能从这个状态转到下一个状态的状态列表，因此需要传入目标状态。而该接口的实现则是基于空格子（0 格子）的状态来决定的：遍历每一个格子，找到与空格子距离为 1 的格子，将其交换即可产生新的状态。

为了使得状态转移可溯源，在状态层中我将上一个状态保存在此状态类的 previous_state 成员变量中，如果是第一个状态，则其为 None，如此便可以在队列弹出目标时，回溯整个状态变化过程。

因此这个状态类的结构如下图所示：



结构图 2 小明玩球状态类设计

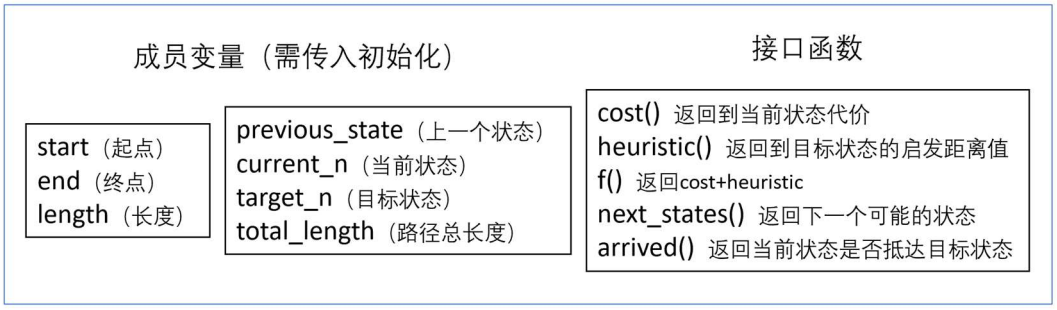
RunState（爱跑步的小明）状态类：

由于爱跑步的小明这一问题**代价函数即为状态转移时的路径长度**，因此我将 Path 类独立出来单独存储路径，使得逻辑更加清晰。而启发函数的设计在本题中则比较难以琢磨，因为我们无法得知每一条路径到底是多长（因为也可能存在每个坐标之间距离均为 1 的情况），而 A*算法要求启发函数必须在 0 到真实值之间，不能大于真实值，因此在本题中，启发函数我即设计为 0，无论任何状态，均将其到目标状态的启发函数值设计为 0，当然这其实也就等同于使用 UCS 算法了，当然我们也可以将启发函数设计为 0 到 1 之间的小数。

由于我将所有启发函数值设为 0，那么启发函数值为 0 当作判定状态达到目标状态的方法也就失灵了，因此在这个状态类中添加了 arrived()接口来实现判断状态是否达到目标状态。

同上一个状态类设计，为了使结果可溯源，引入了 previous_state 成员变量和 next_states()接口，而这个状态类的 next_states()接口则是根据路径对象列表中的所有路径来找出下一个可能状态，较为简单，由于路径为单项路径，因此 Path 类设计时也直接采用 start end length 设计，遍历找到 start 为当前状态的路径即可知道下一个状态以及 cost 的增量。

因此这个状态类的结构如下图所示：



结构图 3 爱跑步的小明状态类设计

应用函数设计：

小明玩球应用函数：

首先定义距离函数（实验中使用曼哈顿距离）。这里由于我保存状态使用的是列表，而实际为二维图形，因此需要进行转换。传入参数 current_idx 表示某一个数当前所在的索引值，target_idx 表示其在目标状态中所在的索引值，我们只需使用整除和取余的方法即可将当前索引和目标索引转化为一个二维坐标，然后根据曼哈顿距离的定义，即可返回两个索引之间的曼哈顿距离。

```
# Compute path with two index 曼哈顿距离
def manhattan_distance(current_idx, target_idx):
    current_x, current_y = current_idx // col, current_idx % row
    target_x, target_y = target_idx // col, target_idx % row
    dx = math.fabs(target_x - current_x)
    dy = math.fabs(target_y - current_y)
    return int(dx + dy)
```

初始化优先级队列，初始状态入队。

```
priorityq = PriorityQueue()
current_state = State(origin_state_list, target_state_list, manhattan_distance, -1)
priorityq.push(current_state.f(), current_state)
```

然后类似 DFS 那样的方法循环直到当前状态的启发函数为 0，否则弹出队首作为当前状态，并调用状态类的 next_states 接口，将下一步可能的状态列表全都加入队伍中，另外，由于此题需要使用图 A*搜索，因此需要维护一个已经展开过的状态集合（实验中定义为 expanded_state_set），将所有展开过的状态放入此集合。

```
# 类似DFS，只不过队列换成优先队列，即可实现A*算法
while(current_state.heuristic() != 0):
    f, current_state = priorityq.pop()
    for state in current_state.next_states():
        if state.string() not in expanded_state_set:
            priorityq.push(state.f(), state)
            expanded_state_set.add(state.string())
```

最后返回当前 cost 和状态转移列表。

爱跑步的小明应用函数：

初始化 Paths 列表，使用主函数传入的 paths 列表。

```
Paths = []
for path in paths:
    Paths.append(Path(path[0], path[1], path[2]))
```

初始化优先级队列，初始状态入队。

```
current_state = RunState(origin_n, target_n)
priorityq = PriorityQueue()
priorityq.push(current_state.f(), current_state)
```

循环 K 次，找到 K 条路径：如果优先级队列为空，则直接 continue，并在 lengths 列表中添加-1（题目要求）；然后弹出当前队首作为当前状态，循环找到一条路径，并将 next_states()接口返回的状态列表加入队列，如果队列为空直接 break，否则弹出队首作为当前状态；如果当前状态达到终点，则将其添加入输出列表中。

```
# 循环K次找到K条路径
for _ in range(K):
    if priorityq.empty():
        lengths.append(-1)
        continue

    # 循环找到一个路径
    f, current_state = priorityq.pop()
    while(not current_state.arrived()):
        for state in current_state.next_states(Paths):
            priorityq.push(state.f(), state)

        # 如果队列中没有元素了，直接break
        if priorityq.empty():
            break
        f, current_state = priorityq.pop()

    # 如果状态是达到终点，则将其添加到输出的列表中
    if current_state.arrived():
        lengths.append(current_state.cost())
        runpath = []
        state = current_state
        while(state.previous_state):
            runpath.append(state.current_n)
            state = state.previous_state
        runpath.append(state.current_n)
        runpath.reverse()
        runpaths.append(runpath)
```

最后返回 K 条路径总长度以及对应的 K 条路线。

主函数设计：

两题我均设计了两种输入方法：文件输入和命令行输入，在应用函数的 py 文件的主函数中编写了两个判断条件。由于本次实验不注重文件读入等操作，因此在这不多赘述，只需按照 README 文件操作即可。

实验结果：

实验结果均在 output 文件夹中，点开各自文件名的 output.txt 即可查看，在文件目录中 output_readonly 为我本机跑出的结果，而 output 文件夹在未运行代码时为空。

小明玩球测试案例结果：

案例编号	输入	输出	转移过程
1	024657318	22	024657318 624057318 624357018 624357108 624357180 624350187 624305187 624035187 024635187 204635187 234605187 234065187 234165087 234165807 234165870 234160875 230164875 203164875 023164875 123064875 123864075 123864705 123804765
2	587346120	26	587346120 587340126 587304126 507384126 057384126 357084126 357184026 357184206 357104286 307154286 370154286 374150286 374105286 304175286 034175286 134075286 134705286 134785206 134785026 134085726 134805726 134825706 134825760 134820765 130824765 103824765 123804765
3	375148206	21	375148206 375108246 305178246 035178246 135078246 135708246 135780246 135786240 135786204 135786024 135086724 135806724 135826704 135826740 135820746 130825746 103825746 123805746 123845706 123845760 123840765 123804765
4	512768340	26	512768340 512760348 512706348 512746308 512746038 512046738 012546738 102546738 142506738 142056738 142756038 142756308 142756380 142750386 142705386 142785306 142785036 142085736 142805736 142835706 142835760 142830765 142803765 102843765 120843765 123840765 123804765

5	123804765	0	123804765

爱跑步的小明测试案例结果：

案例编号	输入	输出	路径
1	5 8 3 5 4 1 5 3 1 5 2 1 5 1 1 4 3 4 3 1 1 3 2 1 2 1 1	1 2 2	路径: 5 1 路径: 5 2 1 路径: 5 3 1
2	6 10 4 6 3 2 6 5 1 5 4 1 5 3 1 5 2 1 5 1 1 4 3 4 3 1 1 3 2 1 2 1 1	2 3 3 3	路径: 6 5 1 路径: 6 5 2 1 路径: 6 3 1 路径: 6 5 3 1
3	6 10 12 6 3 2 6 5 1 5 4 1 5 3 1 5 2 1 5 1 1 4 3 4 3 1 1 3 2 1 2 1 1	2 3 3 3 4 4 7 8 -1 -1 -1 -1	路径: 6 5 1 路径: 6 5 2 1 路径: 6 3 1 路径: 6 5 3 1 路径: 6 3 2 1 路径: 6 5 3 2 1 路径: 6 5 4 3 1 路径: 6 5 4 3 2 1 路径: 路径: 路径: 路径:
4	8 16 8 8 7 2 8 5 2 8 4 3 8 2 1 7 6 2 7 4 3 6 3 2 6 5 1 5 4 1 5 3 1 5 2 1 5 1 1 4 3 4 3 1 1 3 2 1 2 1 1	2 3 4 4 5 6 7 7	路径: 8 2 1 路径: 8 5 1 路径: 8 5 2 1 路径: 8 5 3 1 路径: 8 5 3 2 1 路径: 8 7 6 5 1 路径: 8 7 6 5 2 1 路径: 8 7 6 3 1

5	8 16 16 8 7 2 8 5 2 8 4 3 8 2 1 7 6 2 7 4 3 6 3 2 6 5 1 5 4 1 5 3 1 5 2 1 5 1 1 4 3 4 3 1 1 3 2 1 2 1 1	2 3 4 4 5 6 7 7 7 8 8 8 8 9 9 10	路径: 8 2 1 路径: 8 5 1 路径: 8 5 2 1 路径: 8 5 3 1 路径: 8 5 3 2 1 路径: 8 7 6 5 1 路径: 8 7 6 5 2 1 路径: 8 7 6 3 1 路径: 8 7 6 5 3 1 路径: 8 7 6 5 3 2 1 路径: 8 4 3 1 路径: 8 7 6 3 2 1 路径: 8 5 4 3 1 路径: 8 4 3 2 1 路径: 8 5 4 3 2 1 路径: 8 7 4 3 1

总结：

本次实验回顾了数据结构优先级队列的构建，应用了 A*算法解决两个实际问题，提高了代码解耦分离架构的能力。体会到了 A*算法在实际问题中的妙用，其效率要比普通 BFS，DFS 等快，且满足 admissible 的启发式函数能保证最终结果一定是正确的。

当然在解题过程中也有一点疑惑，其实第一题是偏向贪心的搜索方法，只要用贪心法即可达到最优解；第二题是偏向 UCS 的搜索方法，即使现在使用的 A*本质上也是 UCS，为什么不分别使用各自更简单的方法呢？理解后发现题目看似是需要用 A*算法解题，但是好像是让我们更深刻地了解 A*算法是贪心算法和 UCS 算法的结合。当我们掌握 A*算法后基本就可以套用大部分的解题场景了。