总结报告

目　录

1 需求分析

1.1 基本需求

本次大作业选择的是第一个题目，要求做出将打乱的顺序的 n 数码块（或图片）恢复成给定的状态。

1. （完成）允许随机生成一个九宫排列，给出移动偶成，到达最终的一个目标。
2. （完成）允许使用者自己手动定义一个九宫排列，给出移动过程，到达最终的一个目标。
3. （完成）做出 M*N 的排列，给出移动过程，到达最终的一个排列目标。

2 编程设计

为了尽可能遵循模块化设计的设计理念，此次大作业设计了 3 个大类，分别包含在 State.py、Board.py、MainWindow.py 文件中，程序的启动入口设在 main.py 文件中。类 State 用于表示 N 数码的每个状态。类 Board 用于生成 N 数码的界面，并含有操作界面的接口。类 Window 是主窗口，包含了各种 Qwidget 控件，Board 将会显示在 Window 中。

2.1 操作界面设计

主界面如下图所示：

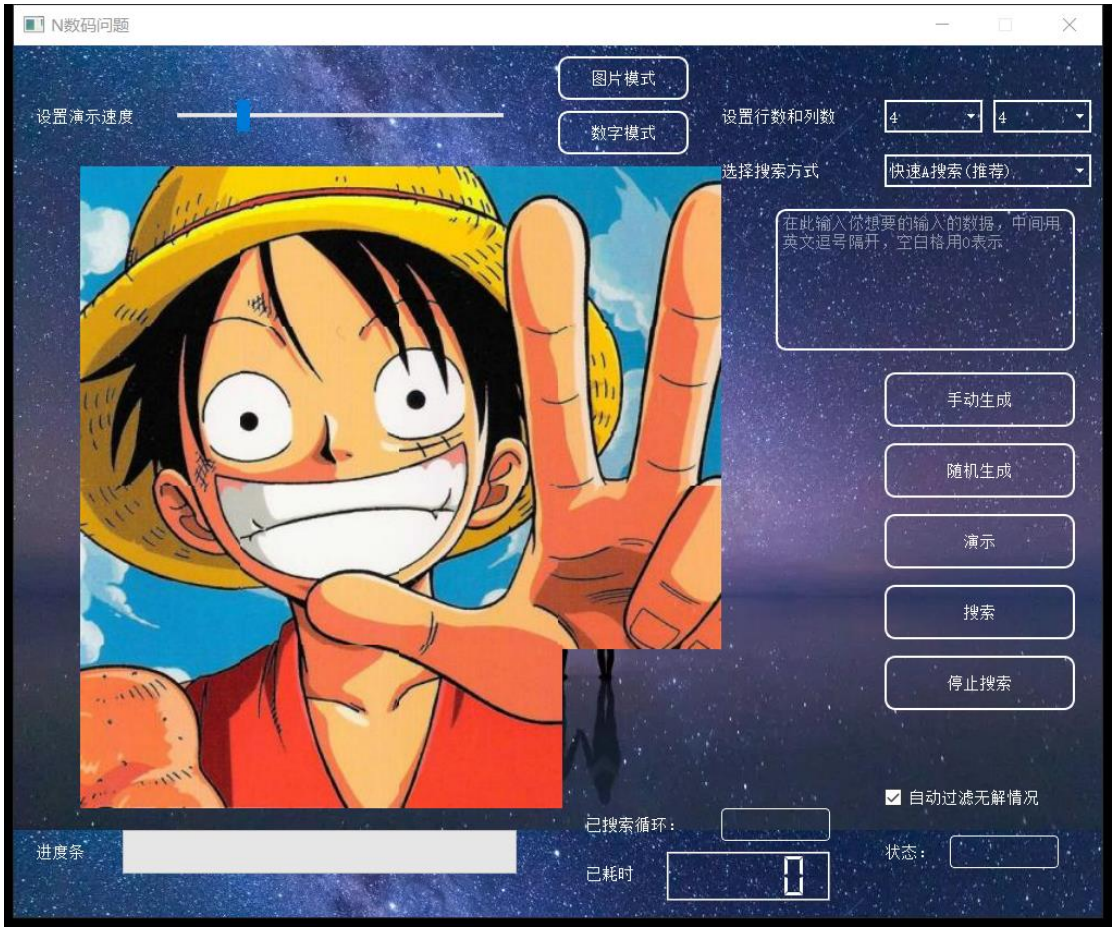

图 1 主操作界面

初始状态：

程序刚运行默认是显示图片的 4×4 状态，数码排列为目标排列，通过“设置行数和列数”可以改变生成的 N 数码界面的行数和列数，建议生成 6×6 及以下的序列。点击“图片模式”可以选择一张图片进行显示，点击“数字模式”可以将显示图片改为显示数字。

生成序列：

点击“随机生成”按钮可以生成随机的数码排列，如果将“自动过滤无解情况”勾选，将会自动过滤随机生成的无解排列，直到找到

有解排列为止；如果想要生成自定义的数码排列，可以在输入框输入符合规范的排列，然后点击“手动生成”按钮即可以想要的排列。

开始搜索：

在任意排列情况（包含目标状态的排列）下点击“搜索”按钮可以进行路径的搜索，在“选择搜索方式”的下拉框下可以选择搜索方式。点击“搜索”按钮后，如果当前排列无解，“状态”标签后会显示“无解”，如果有解，会进入搜索状态，此时进度条显示忙碌的状态、“已搜索循环”实时更新当前已经搜索的次数，“已耗时”会显示当前已经消耗的时间。如果想要中止当前搜索，可以点击“停止搜索”按钮来中止。

开始演示：

搜索完毕后会弹出一个对话框显示搜索循环的总次数、路径的长度以及消耗时间。此时点击“演示”，程序会自动开始演示，下方的进度条会显示当前演示完成的进度，“已耗时”标签后会显示已经消耗的时间。在程序运行的任何时候，都可以通过“设置演示速度”来控制演示的速度，设置的演示速度从 0.01s 到 1s 之间。

其他说明：

在以下情况下可能会弹出弹窗：

在搜索状态下点击“随机生成”、“手动生成”、“图片模式”、“数字模式”、“演示”按钮会弹出弹窗提示正在搜索，可以点击“停止搜索”来中止搜索。

在演示状态下点击“随机生成”、“手动生成”、“图片模式”、

“数字模式”、“搜索”按钮会弹出弹窗提示正在搜索，可以通过滑动“设置演示速度”滑动条来调节演示速度来快速结束演示。

2.2 数据结构设计

本次大作业使用的搜索算法是 A 算法、A*算法及它们的衍生搜索算法。由于涉及到 Open 表和 Closed 表的查找和排序，因此选择合适的数据结构对算法的效能大有影响。

设计描述：

对于 Open 表的设计，采用字典和集合存两份，字典的键为某个节点的代价函数值，值的数据结构为集合，集合中包含了目前已经访问过的节点中代价函数值和键相同的所有节点；集合中存储所有 Open 节点。对于 Closed 表，用一个集合存储。在一次循环中，在字典最小的键对应的集合中任取一个节点，将其作为用于生成孩子节点的父节点。生成的孩子节点先判断在不在 Open 表的集合结构和 Closed 表中，如果不在的话再求取它的代价函数值，并加入 Open 表的字典结构和 Closed 结构中，否则舍弃该孩子节点。

设计原理：

在 Python 中字典和集合都是以元素的 Hash 值为唯一表示进行存储，查找的复杂度为 O（1），因此查找的效率较高。当搜索次数较大时，Open 表的容量将会非常大，直接查找代价函数值最小的节点十分耗时。注意到这些节点中具有相同代价函数值的节点非常多，相应的代价函数值的个数比较少（大约为几百个）。因此采用字典存储，

保证了能够快速找出 Open 表中的最优节点。

2.3 其他设计

由于搜索时占用内存较大，有时耗时较长，搜索过程中可能会造成当前窗口无响应。因此采用多线程的设计方式，将搜索的过程放到线程中，可以防止主界面窗口无响应情况的出现。

在优化过程中还发现， Python 有些内置函数例如 copy.deepcopy 运行速度较慢，因此采用了自己写的函数来替代此类函数。

3 搜索算法

3.1 针对一般性搜索算法的改造

为了简化搜索流程、调高搜索速度，考虑到 N 数码问题的实际情况，本次大作业采用的算法对原算法有所精简。

一般性的 A 或 A*搜索，当找到一个孩子节点时，如果这个孩子节点在 Open 表和 Closed 表中出现过，会先根据老节点和新节点的代价函数值大小来判断是否加入新节点或更新老节点。由于本题的在搜索过程中以搜索深度作为每个节点的 G 值，如果一个节点被二次访问，它的 G 值一般会比老节点的 G 值大，两者的 H 值相同，所以新节点的代价函数会大于老节点。因此搜索时一旦在 Open 表或 Closed 表中发现了老节点，就直接将新节点舍弃。

搜索算法的流程图如下：

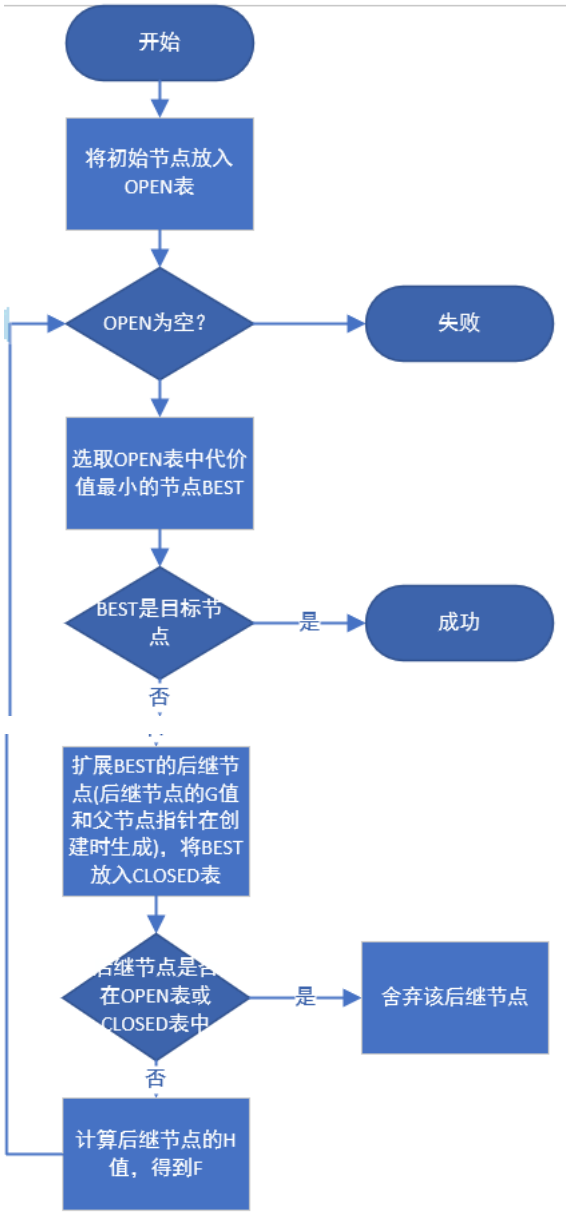

图 2 A 搜索流程图

3.2 A 搜索和 A*搜索

A 搜索是本程序的默认搜索方式。A 搜索有两类搜索方法，第一种是启发函数采用如下形式：

$$H(n) = \max\{column, row\} * ManhattanDistance$$

式中 row 和 $column$ 分别表示生成的 N 数码的行数和列数。$ManhattanDistance$表示每个非空方格距离目标位置的曼哈顿距离

之和。经测试，该启发函数能够较好地满足速度要求。

第二种 A 搜索方法的启发函数是

$$H(n) = \max\{column, row\} * ManhattanDistance + ReverseValue * 2 + In$$

$ReverseValue$是当前序列按照从左到右从上到下的顺序的逆序数对的数量，In是空白格是否归位（即是否回到最后一行最后一列）的信号，空白格归位则取 1 否则取 0。该搜索算法的启发函数更加贴合实际，不过由于存在计算逆序数对的过程，该算法循环一次耗时比快速 A 搜索算法耗时长。

A*搜索和 A 搜索的不同之处在于启发函数的评估应该小于实际值，因此对于 A*搜索，$H(n)$采用一倍的曼哈顿距离，以保证能求取最优解。

$$H(n) = ManhattanDistance$$

3.3 双向 A 搜索和双向 A*搜索

一般的 A 搜索或 A*搜索都是生成初始节点的孩子节点，当孩子节点中出现了目标节点时搜索完毕。这种搜索方法发散的节点较多，可能会有较多无效的节点。因此考虑在以初始节点为根节点生成孩子节点的同时，同时以目标节点为根节点生成孩子节点，启发函数采用

$$H(n) = \max\{column, row\} * ManhattanDistance$$

当以目标节点为根节点的孩子节点出现在初始节点的 Open 表或 Closed 表中，或者以初始节点为根节点的孩子节点出现在目标节点的 Open 或 Closed 表中时，表明已经找到一条路径，结束搜索。

同理，双向 A*搜索也是相同原理。

3.4 各种搜索算法的运行效能

下表是各种搜索算法对于不同大小的 N 数码的搜索效能对比。表格中每一列测试时使用相同的排列，电脑 RAM 为 8G，CPU 为 Inter（R）Core（TM） i7-8550U CPU @1.80GHz 1.99GHz。

搜索算法	项目	3×3	4×4	5×5	6×6
快速 A 搜索	用时/s	0.033	0.545	74.304	46.992
	搜索循环	152	6779	715824	316400
	步数	30	93	222	405
双向 A 搜索	用时/s	0.016	0.238	4.122	99.100
	搜索循环	121	1177	18170	299754
	步数	34	91	196	443
优化 A 搜索	用时/s	0.064	1.068	23.432	326.551
	搜索循环	248	9783	133298	1031048
	步数	32	127	272	441
A*搜索（曼哈顿距离）	用时/s	1308	>180	>420	>400
	搜索循环	0.112	>2357000	>1500000	>1000000
	步数	24	无	无	无
双向 A*搜索	用时/s	0.102	74.443	>420	>400
	搜索循环	360	446536	>1500000	>1000000
	步数	24	63	无	无

表 1 不同搜索算法对于不同大小的 N 数码的搜索效能

从上表可以看出，在 N 数码问题的规模是 3×3 时，各种搜索算法的时间消耗差异不大，A*搜索和双向 A*搜索都能找到最优解。当规模上升到 4×4 时，A*搜索无法在 3min 内找到路径，双向 A*搜索能够满足效用的要求，而 A 搜索都能在 1s 左右找到一个可行解。规模是 5×5 时，A 搜索都能在较短时间内找到一条路径 A*搜索都不能在较短时间内找到解。规模上升到 6×6，A 搜索都能保证在较短时间找到解，A*搜索效能较低。

对于 6×6 的问题，有关快速 A 搜索和双向 A 搜索更加详细的对比如下表所示：

搜索次数 / 搜索算法		1	2	3	4	5
快速 A 搜索	用时/s	297.317	60.833	2.748	29.082	18.890
	搜索循环	1648866	379089	20058	202367	141377
	步数	415	381	389	393	429
双向 A 搜索	用时/s	341.412	135.822	6.416	67.382	46.060
	搜索循环	889826	379085	20055	202365	141374
	步数	433	381	389	393	429

表 2 快速 A 搜索和双向 A 搜索对于 6×6 问题的搜索效能对比

可见，双向 A 搜索和 A 搜索比较，双向 A 搜索的循环次数每次都小于 A 搜索，但某些情况下双向 A 搜索的搜索循环次数仅仅比 A 搜索的次数少 3~4 步，相比之下双向 A 搜索由于每次循环的时间相当于 A 搜索的两倍，因此双向 A 搜索显得耗时更长。总体而言，根据当前的代价函数，大部分 6×6 问题都能较快速求解。

搜索 3×3 规模以上的 N 数码问题时建议使用 A 算法。其中快速 A 算法更加详细的运行效能如下表所示：

搜索次数 / N 数码规格	1	2	3	4	5	平均
3×3	0.01s	0.0	0.02s	0.01s	0.01s	0.01s
4×4	0.10s	0.07s	0.54s	0.17s	0.87s	0.34s
5×5	0.70s	17.14s	5.27s	6.98s	0.837s	6.185s
6×6	31.222s	22.808s	>600s	>600s	9.171s	>172.64s

表 3 快速 A 搜索算法对于不同规模问题的搜索效能对比

可见快速 A 算法能够满足 6×6 以内的 N 数码问题的快速求解，对于 6×6 规模的 N 数码问题，耗时从几秒到十分钟左右不等。

4 心得体会

在本次做大作业的过程中，一开始我设计的搜索算法采用的数

据结构只有列表，运行最基本的 3×3 的都很慢，后来发现是因为列表的查找复杂度是 O（n），和同学交流思路之后，把列表换为了查找复杂度是 O（1）的字典和集合，使得搜索速度有了质的提升。由此更加深刻地认识到数据结构的选择对于算法的效率是多么重要。

另外，由于搜索的效率也与启发函数的设计有关。尽管数据结构的优化能够减少每一次搜索的时间，但是总的搜索的次数和启发函数密切相关，这两者都会影响最终的耗时。目前为止尚未找到一个能够在所有情况下都能较快搜索出路径的启发函数，即便是最终采用的启发函数，面对某些高维 N 数码问题可能搜索效率很低。