一、实验目的与要求

实验目的：

1．熟悉状态空间表示法；

2．掌握深度优先、广度优先和等代价无信息搜索算法；

3．掌握启发式函数设计，实现面向实际问题的 A*搜索算法；

实验要求：

1. 实验提交文件为实验报告和相关程序代码，以压缩包的形式提交，命名规则为“学号数字+姓名+Task1”，如 2099154099 张三 Task1；

2. 所有素材和参考材料需列明出处，实验报告中的图片和程序代码建议标注个人水印或标识信息：姓名，班级，学号信息；

二、实验内容与方法

实验内容：

1. 利用无信息搜索算法实现八数码难题求解；

2. 设计启发式信息函数，利用 A*搜索实现八数码难题求解；

三、实验步骤与过程[1]

本次实验报告的内容思维导图如下所示，感谢阅读，审批辛苦了！

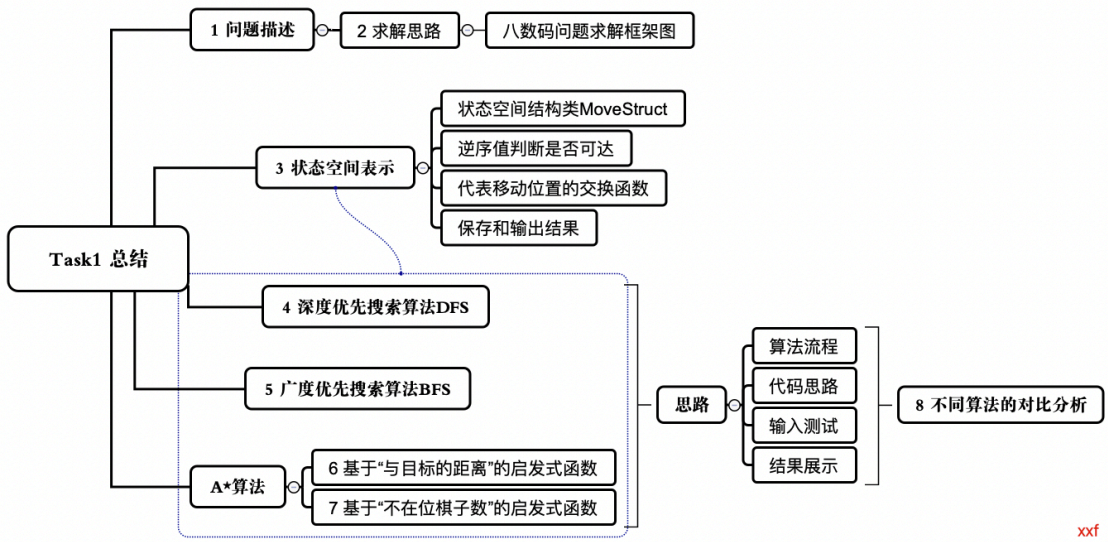

图 1:本次实验报告内容思维导图

1 问题描述

在 3 x 3 的棋盘上，摆放八个棋子，每个棋子上标有 1 至 8 的某一数字。棋盘中留有一个空格，空格用 0 来表示。空格的周围的棋子可以移动到空格中，要求解的问题是：给出一种初始布局和目标布局，找到一种最少移动步骤的移动方法，实现从初始布局到目标布局的

[1] 本实验报告统一字体如下：标题 – 小四宋体加粗，正文 – 五号宋体，标注 – 五号宋体，行距为 1.3

转变。

2 求解思路

本次实验拟通过深度优先搜索和广度优先搜索的无信息搜索算法以及基于“与目标距离”和基于“不在位棋子数”设计启发式函数的启发式信息搜索算法[2]等 4 种算法尝试解决八数码问题，并通过运行速度对比 4 种算法的性能，对不同算法做出评价。（这里说明，无信息搜索算法包括 BFS、DFS 和 UCS，但由于此问题中代价都为 1，DFS 等价于 UCS，因此本实验没有额外展示一致代价搜索算法。）

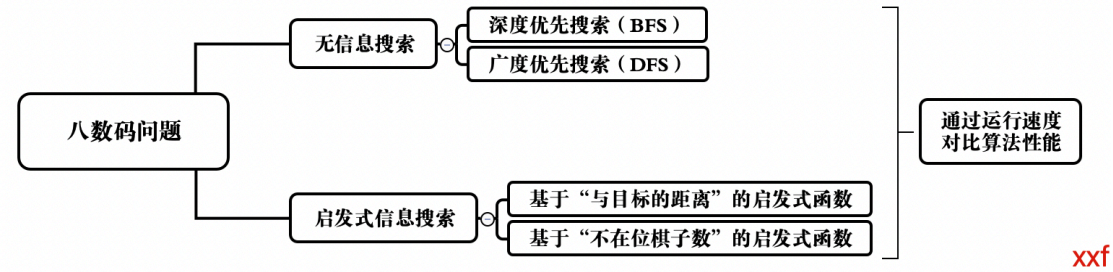

图 2 八数码问题求解框架[3]

3 状态空间表示

在进行不同的搜索算法前，首先需要将问题用状态空间进行表示。八数码问题可表示为空格在九宫格上的移动问题，要求从初始状态改变为目标状态。空格在不同位置有不同的动作集合，这里我使用的 *Python* 基本数据结构是列表和字典，用来模拟栈和队列的形式。

定义一个状态空间结构类 *MoveStruct*。其中，*NextLoc* 存储了空格的动作集合，表示的是数码位于每个位置时下一步可移动的位置，比如[1, 3]表示当空格位于棋盘上的第一个位置时（也就是 0 位时），下一步移动时，空格可能向右移动，出现在 1 位，也可能向下移动，出现在 3 位。*Movement* 存储的是空格在每个位置时，其他位置的数码对应的操作集合，key 代表空格所在的位置，value 代表其他位置可操作动作，如 0: [0, "Left", 0, "Up"]表示，当空格位于棋盘上的第一个位置时（也就是 0 位时），1 位可以移动的对应动作是“Left”，2 位不可以移动，表示为 0，3 位可以移动的对应动作是“Up”。这种做法通过定义九宫格中空格所在的 9 个位置可以移动的方向，并且设定好移动的方向，好处就是可以不考虑数码移动时越界的情况。

```
class MoveStruct:  #定义状态空间  xxf
    def __init__(self):
        self.NextLoc = [];
        self.Movement ={};

Moveable=MoveStruct()
Moveable.NextLoc=[[1,3],[0, 2, 4],[1, 5],[0,4,6],[1,3,5,7],[2,4,8],[3,7],[4,6,8],[5,7]]
```

[2] 参考于课堂讲授 A star 算法优势性时的思路
[3] 本实验报告代码和图片一致使用“xxf”（姓名：谢晓锋的缩写）标注本人信息

```
Movement={0:[0,"Left",0,"Up"], 1:["Right",0, "Left",0,"Up"],2:[0,"Right",0,0,0, "Up"],3:["Down",0,0,0,"Left",0,"Up"],4:[0,"Down",0,"Right",0,"Left",0,"Up"],5:[0,0,"Down",0,"Right",0,0,0,"Up"],6:[0,0,0,"Down",0,0,0,"Left"],7:[0,0,0,0,"Down",0,"Right",0,"Left"],8:[0,0,0,0,"Down",0,"Right"] }
```
[4]

给定一个初始状态，并不是所有的目标状态都存在一个可行的行动序列，在寻找序列解前，需要利用逆序值的性质来判断两个状态是否可达，若两个状态的逆序值同为奇或同为偶，那么就可以求出一组移动序列，若两个状态的逆序值为一个奇数一个偶数，则无解。[5]定义函数 *judge(srcLayout, destLayout)*，*srcLayout* 表示初始状态，*destLayout* 表示目标状态。

```
def judge(srcLayout, destLayout):# 判断初始状态是否能够到达目标状态 xxf
    src = 0
    dest= 0
    for i in range(1, 9):
        fist = 0
        for j in range(0, i):
            if srcLayout[j] > srcLayout[i] and srcLayout[i] != '0':
                fist = fist + 1
        src = src + fist
    for i in range(1, 9):
        fist = 0
        for j in range(0, i):
            if destLayout[j] > destLayout[i] and destLayout[i] != '0':
                fist = fist + 1
        dest = dest + fist
    if (src % 2) != (dest % 2):  # 同奇同偶时才可达
        return -1, None
    else:return 0
```

对于棋盘的状态空间表示，最直接的表示方式就是用一个 3*3 的矩阵表示，但这可能会导致算法性能降低，所以我是用一个符串数组表示棋盘状态，0 代表空格，比如初始状态 *srcLayout="012345678"*表示空格位于棋盘第一个位置的八数码状态，这样表示可以使储存内存降低，为了表示数码的移动，以 BFS 和 DFS 的无信息搜索算法为例，定义函数 *swap_chr(a, i, j)*，其中，a 表示移动前的空间状态，*i* 代表可以移动的数码位置（通过已经定义的 *Moveable.NextLoc(j)* 获取），*j* 表示空格所在的位置（即“0”所在的数组下标，通过 *curLayout.index("0")*的方式获取，*curLayout* 表示当前的空间状态，在后面函数会中定义），最后通过 $i > j$ 的判断交换数组位置实现移动，然后再使用切片的方法表示返回的交换后的空间状态。在定义启发式信息搜索的交换函数时，也是同理，只是需要加上一个 *f(n)*函数来记录 *A**算法中，返回交换后的状态以及交换后距离目标状态的代价。

[4] 使用了代码高亮网站 http://word.wd1x.com/ ，使报告更加美观清晰
[5] 思路参考：https://zhuanlan.zhihu.com/p/127409399

```
def swap_chr(a, i, j):#无信息搜索算法：用于移动位置的交换函数 xxf
    if i > j:
        i, j = j, i
    b = a[:i] + a[j] + a[i+1:j] + a[i] + a[j+1:]#切片方法，返回交换后的状态
    return b
```

接下来需要解决的是：我们应该怎么保存和输出结果呢？这里使用的是 *solvePuzzle(srcLayout, destLayout)*函数，我将结果的所有可能集保存在字典 *Statu_saving= {}* 中，通过 KV 对来标明状态，上一个状态作为 Value，下一个可抵达的状态作为 Key，这样可以得到结果的一系列状态解，结构简单，且便于后面回溯。需要注意的是在每次进行算法搜索的时候，需要通过 *Statu_saving[srcLayout] = -1*（初始状态没有对应的 srcLayout，因此 -1 是终止条件值）初始化字典，防止其他算法结果的干扰。至于输出结果，一是要输出过程状态，通过 *Statu_saving[curLayout]* 将每次移动的状态列表按照最后状态往前逆推回溯，得到之前的路径，并将每一个状态结果 *append* 保存在 *Answer_Output* 中。注意，为了最后能够顺序输出状态，这里使用了 reverse 方法颠倒顺序。二是要输出每次移动的动作合集，同理，这里通过设定好的 *MoveString = {}*字典来存储每个可能发生的动作，可以根据实际情况移动而输出结果，而且是一个完整的动作集，把从初始状态到目标状态的全过程保存下来，再同样通过 *Movement_Output* 输出动作的结果。直到 *Statu_saving* 等于初始状态 *srcLayout* 时，终止回溯，输出结果。

```
def solvePuzzle(srcLayout, destLayout):
    Answer_Output = []#当全部状态完成 xxf
    Answer_Output.append(destLayout)
    curLayout=destLayout
    while Statu_saving[curLayout] != -1:#当等于-1 时，意味着可以退出循环，因为第一个初始状态的字典 value 是-1
        curLayout = Statu_saving[curLayout]#按照最后状态往前逆推回溯
        Answer_Output.append(curLayout)#将每一个状态保存
    Answer_Output.reverse()#用 reverse 得到顺序输出
    for answer in Answer_Output:
        if(answer!=srcLayout):
            Movement_Output.append(MoveString[answer])
    return 0, Answer_Output#如同奇同偶，则可达
```

这样我们就初步定义好了八数码的状态空间表示，整个实验的思路也已经比较成熟了，接下来我将开始对四种算法进行设计、测试和展示。

4 深度优先搜索算法 DFS

4.1 代码思路

深度优先搜索算法解决八数码问题的算法流程和核心代码分别如下：

步骤	*操作*
1	输入八数码初始状态 *srcLayout* 和目标状态 *destLayout*。
2	通过逆序性函数 *judge(srcLayout, destLayout)*判断目标状态是否可达，如果可达，则继续，否则就退出。
3	利用 *Swap_chr(a,i,j)*函数，基于初始状态中空格的位置，得到当前可到达的一系列状态
4	通过 *solvePuzzle_DFS(srcLayout, destLayout)* 函数，将当前可达的一系列状态存入 *Answer_Solving* 列表，并且将当前状态（字典的 value）和下一步状态（字典的 key）存入 *Statu_saving* 字典，将对应的移动动作存入 *MoveString* 字典（DFS 这里使用栈实现，BFS 这里修改为队列实现，其他一致）
5	如果当前状态等于目标状态或者 *Answer_Solving* 列表为空，则继续开始保存和输出结果，否则，回到 3
6	通过 *solvePuzzle(srcLayout, destLayout)*函数，从 *destLayout* 逆推抵达 *scrLayout* 的状态和动作，并存入 *Answer_Output*
7	判断 *Statu_saving* 是否等于初始状态 *srcLayout*，当 *Statu_saving* 回溯到初始状态时，终止回溯，继续输出结果，否则，回到 6
8	将 *Answer_Output* 做 reverse 颠倒回到正常顺序，最后将 *Answer_Output* 切片并循环输出搜索过程的每一步状态和动作，以及输出搜索次数和运行时间

表 1 DFS 算法流程

```
def solvePuzzle_DFS(srcLayout, destLayout):#深度优先搜索算法
    Statu_saving[srcLayout] = -1 #初始化字典
    Answer_Solving = []
    Answer_Solving.append(srcLayout)#当前状态存入列表

    while len(Answer_Solving) > 0: #如果栈中存在状态
        curLayout = Answer_Solving.pop()#出栈（注意：BFS 需要修改的地方)
        if curLayout == destLayout:#判断当前状态是否为目标状态
            break#如果是目标状态，则出栈

        ind_slide = curLayout.index("0")#获得空格位置的下标,
        lst_shifts = Moveable.NextLoc[ind_slide]#当前可进行交换的位置集合
        for nShift in lst_shifts:
            newLayout = swap_chr(curLayout, nShift, ind_slide)#进行交换
            if Statu_saving.get(newLayout) == None:#判断交换后的状态是否没有查询过
                Statu_saving[newLayout] = curLayout#如果状态没有查询过，则将其存到字典
                Answer_Solving.append(newLayout)#存入集合
                MoveString[newLayout]=Movement[ind_slide][nShift]#动作存在 MoveString 中
                if newLayout == destLayout:  # 判断当前状态是否为目标状态
                    break  # 如果是目标状态，则出栈
```

4.2 输入测试

Test	*输入*	*输出*

	初始状态	*目标状态*	*搜索次数（次）*	*运行时间（秒）*
T1	012345678	123045678	60141	1.0897068977355957
T2	123645708	123045678	214	0.003155946731567383
T3	642317805	123045678	31702	0.5701100826263428
T4	136254870	123045678	22817	0.3518791198730469

表 2 DFS 输入测试

4.3 结果展示

图 3 DFS 算法运行结果展示

5 广度优先搜索算法 BFS

5.1 代码思路

DFS 和 BFS 的区别就是访问节点的顺序不同，DFS 利用栈先进后出的特性来实现算法，而 BFS 则利用队列先进先出的特性来实现算法。所以只需将原来的 DFS 函数中存储状态节点的 *Answer_Solving* 修改为队列表示即可，即将 DFS 中的 *Answer_Solving.pop()* 修改为 *Answer_Solving.pop(0)*（如下图 4 所示）每次都是从队列头取节点进行目标状态判断。其他代码都跟 DFS 算法一样，这里就不再重复展示算法流程和具体代码

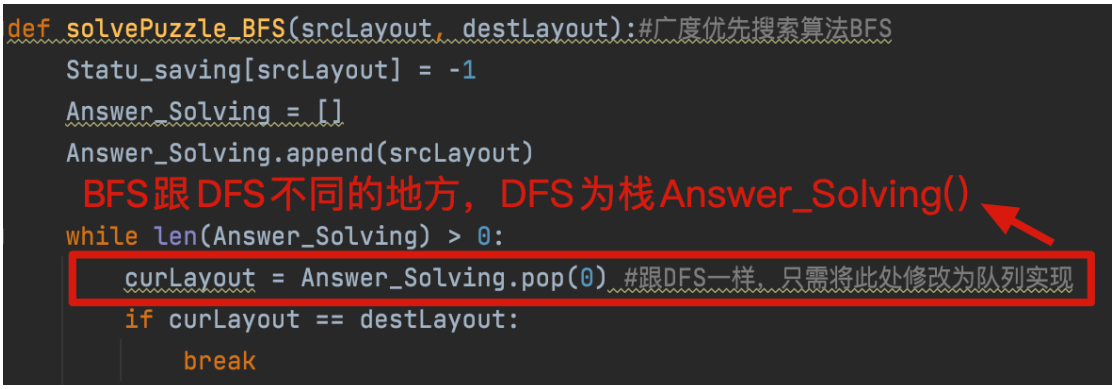

图 4 BFS 算法代码（跟 DFS 不一样的地方在于修改为队列）

5.2 输入测试

Test	*输入*		*输出*	
	初始状态	*目标状态*	*搜索次数（次）*	*运行时间（秒）*

T1	012345678	123045678	13	0.005928993225097656
T2	123645708	123045678	2	0.00011491775512695312
T3	642317805	123045678	24	6.602513074874878
T4	136254870	123045678	23	5.147953271865845

表 3 BFS 输入测试

5.3 结果展示

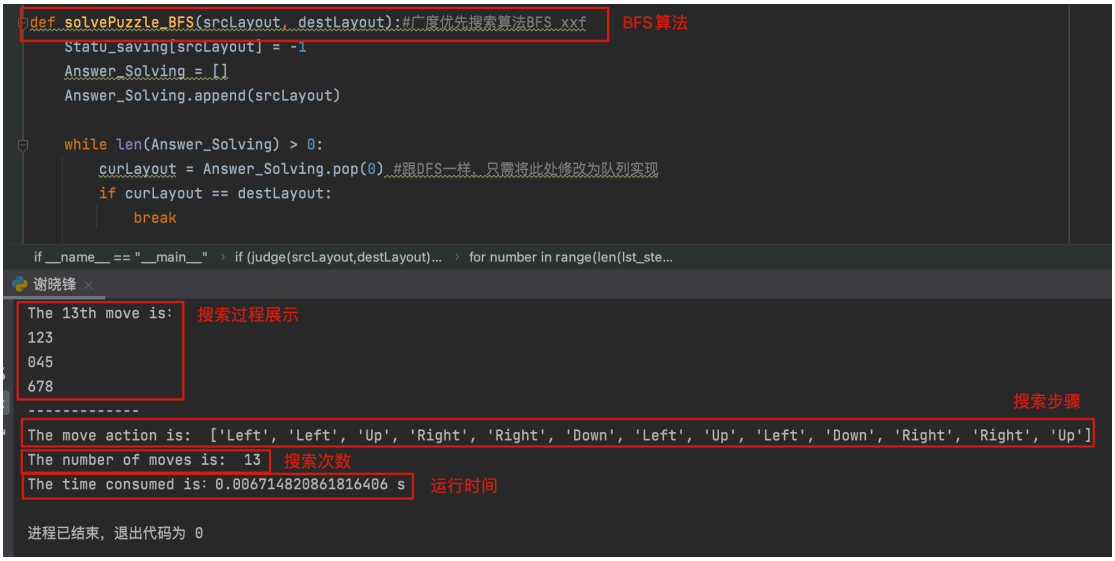

图 5 BFS 算法运行结果展示

6 A*算法 – 基于“与目标的距离”的启发式函数[6][7]

6.1 代码思路

本实验的启发式函数主要用来估计八数码矩阵变换到最终的八数码矩阵所需要的步数，由于每个方格只能上下左右移动，因此基于“与目标的距离”的启发式函数设计的 A*搜索算法指的就是以曼哈顿距离（小于总移动步数，满足可纳性）作为八数码问题的启发式函数，这里利用 *Distance_Compute(srcLayout,destLayout)*函数来计算当前状态到目标状态的代价。

```
def Distance_Compute(srcLayout,destLayout):#基于曼哈顿距离计算代价，返回代价
    sum=0
    a= srcLayout.index("0")#排除空格的下标位置，防止干扰
    for i in range(0,9):#循环其他每个位置
        if i!=a:
            sum=sum+abs(i-destLayout.index(srcLayout[i]))#计算当前状态到目标状态代价
    return sum
```

需要注意的是，在前面状态空间表示也提及到的，在定义用于移动位置的交换函数时，启发式搜索算法比无信息搜索还多了一个计算交换后距离目标状态代价的步骤，这里我将预测接下来要走的代价储存在 *fn* 中。

[6] 思路参考：https://blog.csdn.net/qq_41417335/article/details/86359831
[7] 思路参考：https://www.cnblogs.com/tiangouyzy/p/13837924.html

```python
def swap_chr_A(a, i, j, distance, destLayout):#A*（Distance）xxf
    if i > j:
        i, j = j, i
    b = a[:i] + a[j] + a[i+1:j] + a[i] + a[j+1:]
    fn = distance+Distance_Compute(b, destLayout) #fn,储存预测要走的代价值
    return b, fn
```

同时，我使用 *Statu_distance={}*这个字典来记录当前状态到目标状态的距离，也就是实际代价；使用 *Fn= {}*字典来储存总代价函数，其中，key 代表目前状态，value 代表到目的状态的总代价（实际+预测）。有了这些定义后，基于“与目标的距离”的启发式函数设计的 A*搜索算法解决八数码问题的算法流程和核心代码分别如下。其中跟无信息搜索不一样的地方已经用蓝色字体标注，主要是在 *Distance_Compute*、*swap_chr_A* 和 *solvePuzzle_A* 三个函数上，其他思路都是一致的。

步骤	***操作***
1	输入八数码初始状态 *srcLayout* 和目标状态 *destLayout*。
2	通过逆序性函数 *judge(srcLayout, destLayout)*判断目标状态是否可达，如果可达，则继续，否则就退出。
3	利用 *Swap_chr_A* 函数，基于初始状态中空格的位置，得到当前可到达的一系列状态。并且依据当前状态到目标状态的曼哈顿距离(*Distance)*计算 *h(x)*代价
4	通过 *solvePuzzle_A(srcLayout, destLayout)* 函数，将当前可达的一系列状态存入 *Answer_Solving* 列表，并且将当前状态（字典的 value）和下一步状态（字典的 key）存入 *Statu_saving* 字典，将对应的移动动作存入 *MoveString* 字典（需要找到最小代价后，用 remove 移除变量）
5	如果当前状态等于目标状态或者 *Answer_Solving* 列表为空，则继续开始保存和输出结果，否则，回到 3
6	通过 *solvePuzzle(srcLayout, destLayout)*函数，从 *destLayout* 逆推抵达 *scrLayout* 的状态和动作，并存入 *Answer_Output*
7	判断 *Statu_saving* 是否等于初始状态 *srcLayout*，当 *Statu_saving* 回溯到初始状态时，终止回溯，继续输出结果，否则，回到 6
8	将 *Answer_Output* 做 reverse 颠倒回到正常顺序，最后将 *Answer_Output* 切片并循环输出搜索过程的每一步状态和动作，以及输出搜索次数和运行时间

表 4 基于“与目标的距离”的启发式函数设计的 A*搜索算法流程

```python
def solvePuzzle_A(srcLayout, destLayout):#A* (Distance+Number)  xxf
    Statu_saving[srcLayout] = -1 #初始化字典
    Statu_distance[srcLayout]= 1
    Fn[srcLayout] = 1 + Distance_Compute(srcLayout, destLayout) #Distance的A*算法，计算总代价：实际要走的代价+预测的代价
    #Fn[srcLayout] = 1 + Number_Compute(srcLayout, destLayout) #Number 的A*算法
    Answer_Solving = []
    Answer_Solving.append(srcLayout)#当前状态存入列表，跟无信息搜索一致

    while len(Answer_Solving) > 0:#当存在可移动的状态
        curLayout = min(Fn, key=Fn.get)#比较字典中value，返回Key，找到最小代价作为下一个起点
        del Fn[curLayout]#找到最小代价后删除curLayout中的变量

        Answer_Solving.remove(curLayout)#找到最小fn后还要移除“curlayout”这个对象
        if curLayout == destLayout:#判断当前状态是否为目标状态
            break
        ind_slide = curLayout.index("0")# 寻找0的位置。
        lst_shifts = Moveable.NextLoc[ind_slide]#当前可进行交换的位置集合
        for nShift in lst_shifts:#在可以交换的集合中
            #hn是目前的总代价（实际+预测）
            newLayout, hn = swap_chr_A(curLayout, nShift, ind_slide, Statu_distance[curLayout] + 1, destLayout)#Distance的A*算法
            #newLayout, hn = swap_chr_a(curLayout, nShift, ind_slide, Statu_distance[curLayout] + 1, destLayout)#Number的A*算法
            if Statu_saving.get(newLayout) == None:#判断交换后的状态是否已经查询过
                Statu_distance[newLayout] = Statu_distance[curLayout] + 1 #存入走的路程，走了1步
                Fn[newLayout] = hn#存入fn,当前状态对应的代价
                Statu_saving[newLayout] = curLayout#定义前驱结点
                Answer_Solving.append(newLayout)#存入集合
                MoveString[newLayout] = Movement[ind_slide][nShift]  # 动作存在对应的MoveString
                if newLayout == destLayout:  # 判断当前状态是否为目标状态
                    break  # 如果是目标状态，则出栈
```

图 6 基于“与目标的距离”的启发式函数设计的 A*搜索算法核心代码

6.2 输入测试

Test	*输入*		*输出*	
	初始状态	*目标状态*	*搜索次数（次）*	*运行时间（秒）*
T1	012345678	123045678	13	0.0017738342285156 25
T2	123645708	123045678	2	0.00011491775512695312
T3	642317805	123045678	24	0.015255928039550781
T4	136254870	123045678	25	0.15064692497253418

表 5 基于“与目标的距离”的启发式函数设计的 A*搜索算法输入测试

6.3 结果展示

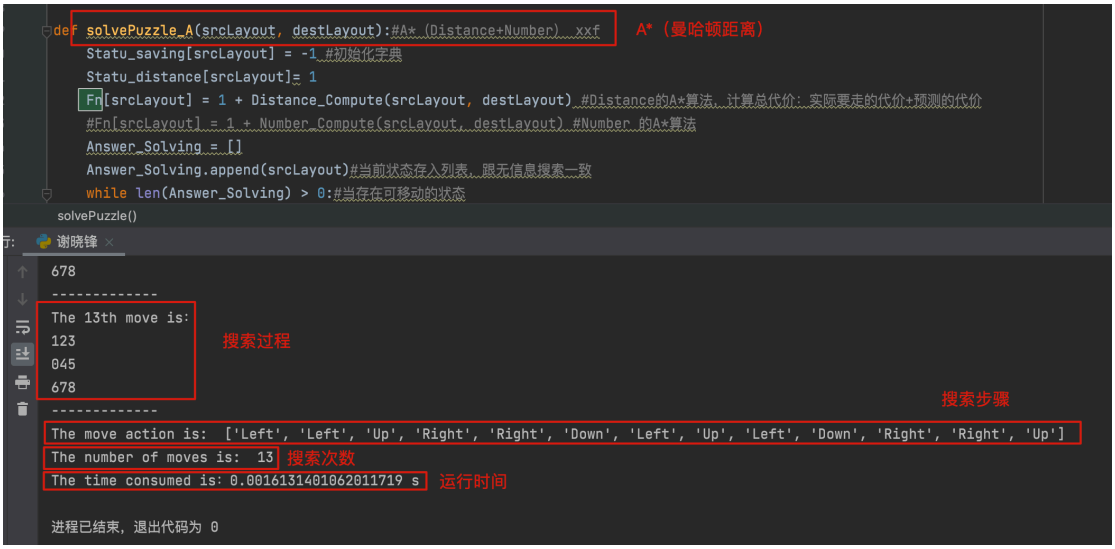

图 7 基于“与目标的距离”的启发式函数设计的 A*搜索算法运行结果展示

7 A*算法 – 基于“不在位棋子数”的启发式函数

7.1 代码思路

基于“不在位棋子数”的启发式函数设计的 A*搜索算法主要在上一个算法的基础上做一些小修改即可。定义 *Number_Compute(srcLayout,destLayout)*计算错和正确码错位个数之和。

```
def Number_Compute(srcLayout,destLayout):#返回错码和正确码错位个数之和 xxf
    sum=0
    a= srcLayout.index("0")
    for i in range(0,9):
        if i!=a:
            if(srcLayout[i]!=destLayout[i]):#如果发生错位
                sum+=1#计算当前状态和目标状态的代价
    return sum
```

后面的算法基于新设计的启发式函数相对应修改即可，本质都是 A*算法，基本一致：

```
def solvePuzzle_A(srcLayout, destLayout):#A* (Distance+Number)  xxf
    Statu_saving[srcLayout] = -1 #初始化字典
    Statu_distance[srcLayout]= 1
    #Fn[srcLayout] = 1 + Distance_Compute(srcLayout, destLayout) #Distance的A*算法，计算总代价：实际要走的代价+预测的代价
    Fn[srcLayout] = 1 + Number_Compute(srcLayout, destLayout) #Number 的A*算法
    Answer_Solving = []
    Answer_Solving.append(srcLayout)#当前状态存入列表，跟无信息搜索一致
    while len(Answer_Solving) > 0:#当存在可移动的状态
        curLayout = min(Fn, key=Fn.get)#比较字典中value，返回Key，找到最小代价作为下一个起点
        del Fn[curLayout]#找到最小代价后删除curLayout中的变量
        Answer_Solving.remove(curLayout)#找到最小fn后还要移除“curLayout”这个对象
        if curLayout == destLayout:#判断当前状态是否为目标状态
            break
        ind_slide = curLayout.index("0")# 寻找0的位置。
        lst_shifts = Moveable.NextLoc[ind_slide]#当前可进行交换的位置集合
        for nShift in lst_shifts:#在可以交换的集合中
            #hn是目前的总代价（实际+预测）
            #newLayout, hn = swap_chr_A(curLayout, nShift, ind_slide, Statu_distance[curLayout] + 1, destLayout)#Distance的A*算法
            newLayout, hn = swap_chr_a(curLayout, nShift, ind_slide, Statu_distance[curLayout] + 1, destLayout)#Number的A*算法
            if Statu_saving.get(newLayout) == None:#判断交换后的状态是否已经查询过
                Statu_distance[newLayout] = Statu_distance[curLayout] + 1 #存入走的路程，走了1步
                Fn[newLayout] = hn#存入fn,当前状态对应的代价
                Statu_saving[newLayout] = curLayout#定义前驱结点
                Answer_Solving.append(newLayout)#存入集合
                MoveString[newLayout] = Movement[ind_slide][nShift]  # 动作存在对应的MoveString
                if newLayout == destLayout:  # 判断当前状态是否为目标状态
                    break  # 如果是目标状态，则出栈
```

基于“不在位“设计启发式函数修改的地方

图 8 基于“不在位棋子数”的启发式函数设计的 A*搜索算法算法代码

7.2 输入测试

Test	*输入*		*输出*	
	初始状态	*目标状态*	*搜索次数（次）*	*运行时间（秒）*
T1	012345678	123045678	13	0.0023970603942871094
T2	123645708	123045678	2	0.00017786026000976562
T3	642317805	123045678	24	7.84287691116333
T4	136254870	123045678	23	2.999298095703125

表 6 基于“不在位棋子数”的启发式函数设计的 A*搜索算法输入测试

7.3 结果展示

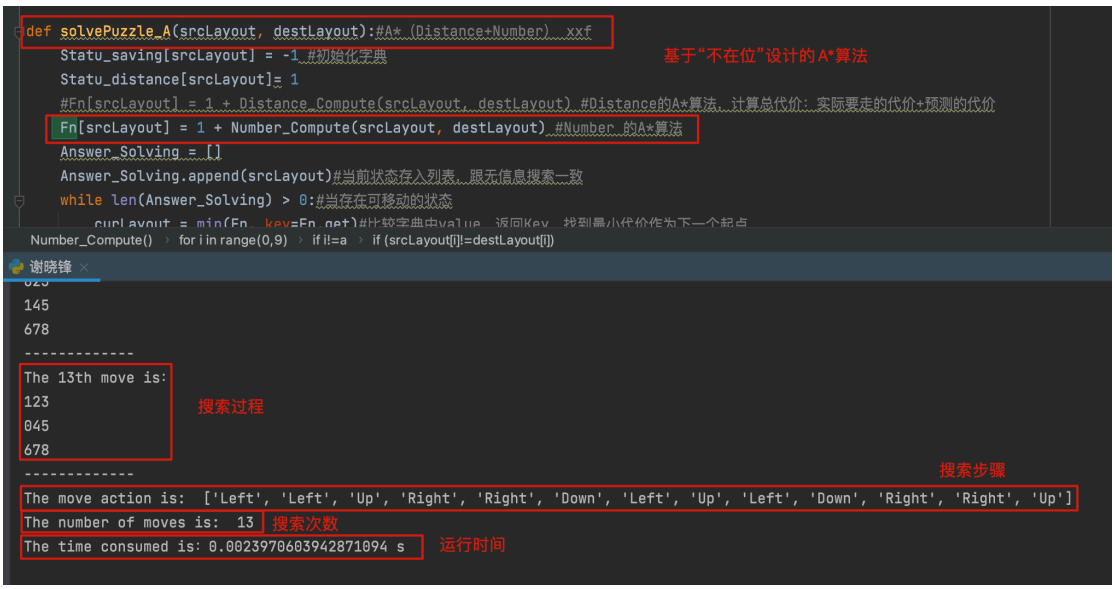

图 9 基于“不在位棋子数”的启发式函数设计的 A*搜索算法运行结果展示

8 不同算法的对比分析

为了让实验结论更加明显并具有说服力，我通过对四种不同算法在 4 个测试实例中求解八数码问题，分别记录了其搜索所用的时间和步数。汇总得到不同算法的运行时间（表 7），并对这些数据进行可视化（图 10）

时间	*DFS*	*BFS*	*A*_Distance*	*A*_Number*
T1	1.0897069	0.00592899	0.00177383	0.00239706
T2	0.00315595	0.00011492	0.00011492	0.00017786
T3	0.57011008	6.60251307	0.01525593	7.84287691
T4	0.35187912	5.14795327	0.15064692	2.9992981

表 7 四种不同算法的运行时间和搜次数

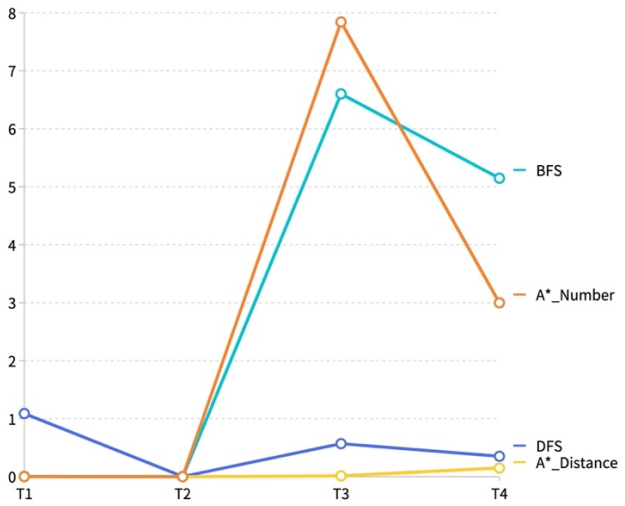

图 10 四种不同算法的运行时间对比趋势图

1、四种搜索算法虽然都用相同的步骤得到了最后的目标状态，但是每个算法的转换路径不同。从整体来看，基于“与目标距离”启发式函数设计 A*算法在完备性、最优性、时

间复杂性度、空间复杂度是最好的。同样是 A*算法，基于“不在位棋子”设计的算法特别是时间复杂度表现得很一般，而且算法性能很不稳定。这是因为 A*算法采用的启发式函数不同导致最后程序的运行时间也不同，根据三角形的三边原理（两边之和大于第三边），在本例中基于曼哈顿距离设计的启发式函数更大，估计值更接近实际值，所以算法运行的速度也就更快，，这在之后其他的八数码问题上都得到了验证。

2、在本实验中深度优先搜索有时搜索次数达到了上万次，这是因为本次算法设计中为了在确保可解的前提下(通过逆序性判断)，能够确保 DFS 的完备性，所以没有设置深度优先搜索的最大深度，所以在获得算法完备性的同时失去了最优性，导致搜索次数达到上万次。

3、至于深度优先算法和宽度优先算法都有比较明显的缺点，从本次测试来看，DFS 正如上面所说的，如果目标节点不在搜索进入的分支上，而该分支又是一个很深度的分支，就可能要搜索上万次才能得到解。而使用 BFS 时，当目标节点距离初始节点较远时会产生许多无用的节点，搜索效率低，只能适用于到达目标结点步数较少的情况，正如 T3、T4 测试中，步数超过 15 步，运行时间长达 5、6 秒，不再起作用。所以这两种算法严重受产生子列表的顺序的影响，低样本量无法得出有效结论，各有缺点。[8]

4、同时还发现每次执行相同的算法，运行时间可能还会不一样，这是因为在不同的 CPU 和电脑内存运行的情况下，受进程的影响导致程序运行时间会有差异，但经过多次测试，还算比较稳定，变化可以忽略不计。

四、实验结论或体会

本次实验在算法设计借鉴了实验课助教和同学的分享，再通过自己的修改完善，使得最终的代码和实验报告更加简洁，可读性也更强。最终通过对四种搜索算法的应用实现了八数码问题，并结合思维导图等工具比较有逻辑地完成了实验报告的撰写。实践出真知，通过这次实验，我对算法的理论也有了更加深刻的理解，也明白了对算法的优缺点以及影响算法优劣的关健因素，这也帮助了我更好掌握理论知识。

整体上，我觉得这次的实验对我还是很有挑战的，第一是第一次真正运用 Python 来解决实际问题，对对 python 这门编程语言还不够熟悉，有些库还不知道怎么去使用，第二是刚开始对八数码问题的算法思想理解不深刻，导致在转化为编程语言的过程中遇到了许多难题，有很多 bug，感觉每走一步都要小心翼翼。

比如在回溯设计中，在我将所有的状态解序列求出后，我本来是将这些状态序列存到一个列表中，导致后面的操作特别难搞，后面在网上看到了一种更巧妙的解法，就是设置一个字典，将节点作为 key，该节点的父节点作为 value，这样就可以简单地根据目标状态回溯到初始状态。还好在实验中比较有耐心，通过一步步尝试和查找资料解决了这些问题，在 debug 的过程中对 Python 语言也有了很大的进步。

本次实验收获颇丰，但在算法设计方面还存在一些缺陷，比如为了对比不同的算法性能，在 DFS 算法上本次没有设置最大深度，在确保完备性的同时也导致了最优性的缺失，后期期待当自己的理论知识和代码技术更加成熟后能够做出更好的优化，设计出性能更好的算法。

[8] 参考：https://www.fx361.com/page/2016/1107/308289.shtml

五、思考题

分析深度优先、广度优化、贪婪、A/A*各算法的优缺点。

	深度优先算法	宽度优先算法	贪婪算法	A/A*算法
完备性	可能不完备（图有环，需进行约束）	完备	可能不完备（图有环，需进行约束）	完备
最优性	否，最左的解	是	否	是
时间复杂性	$O(b^m)$	$O(b^m)$	$O(b^m)$	$O(b^{\varepsilon d})$
空间复杂性	$O(bm)$	$O(b^m)$	$O(b^m)$	$O(b^m)$
优点	有后向代价优点，能找出所有解决方案。	有后向代价优点，结点总是以最短路径访问，适合寻找深度小的情况。	有前向代价优点，扩展“看起来”最近的结点路径。	结合了前向代价估计、后向代价估计优点，利用启发信息优化算法性能
缺点	可能要多次遍历所有可能路径，在深度大的情况下效率低；还可能存在图回环现象；没有利用启发式信息。	当状态空间十分大，内存耗费大，效率低；没有利用启发式信息。	回溯的速度太慢，有时无法得到全局上的最优解数据；最坏的情况退化为 DFS。	启发式函数是可纳/一致的，才能保证最优；启发式函数设计分关键。

表 2. 四种算法的对比

注：b 为每个节点后继节点数，m 为最大深度，d 代表解的深度，$\varepsilon = \dfrac{h^* - h}{h^*}$ 是相对误差（h^* 为真实代价，h 为估计代价）