人工智能第一次实验报告

AStar求解八数码问题

课程：人工智能原理	年级专业：19级软件工程
姓名：郑有为	学号：19335286

目录

一、N数码问题

1.1 N数码问题简介

N数码问题又称为重拍拼图游戏，以8数码为例：在 3X3 的方格棋盘上，放置8个标有1、2、3、4、5、6、7、8数字的方块和1个空白块，空白块可以上下左右移动，游戏要求通过反复移动空白格，寻找一条从某初始状态到目标状态的移动路径。

令W, H为棋盘的宽高，有$N = W \times H - 1$，维度越高求解难度也就越高。

N数码问题的求解策略有广度优先搜索和启发式搜索等，启发式搜索能够利用N数码信息，加快搜索速度。使用广度优先搜索和A*搜索算法都可以求得最优解，即总开始状态到结束状态的最短路径。

1.2 N数码问题的不可解情况

首先给出逆序数的定义：对于串中相邻的一对数，如果位置较前的数大于位置靠后的数，则成为一对逆序，一个串的逆序总数成为逆序数。例如 21432 的逆序数为3，逆序对包括 21，43，32

在N数码中，并不是所有状态都是可解的。将一个状态表示为一维数组的形式，计算除了空位置（0）之外的所有数字的逆序数之和，如果初始状态和结束状态的逆序奇数的偶性相同，则相互可达，否则相互不可达。

简单证明上述结论的必要性：当空块左右移动是不改变逆序数，上下移动时，相当于一个非空块向前或向后移动2格子，如果跨过的两个数字都大于/小于移动的数字，则逆序数可能增减2，如果两个数字一个大一个小，逆序数不变。

因此在设计测试样例时需要考虑逆序数为偶数，否则程序不会终止。例如：[1,2,3,4,5,6,8,7,0] 就是一个不可解情况。

二、A*搜索算法

2.1 基本思想

A*搜索算法利用启发性信息来引导搜索，动态地确定搜索节点的排序，每次选择最优的节点往下搜索，可以高效地减少搜索范围，减少求解的问题的时间。

A*算法用估价函数来衡量一个节点的优先度，优先度越小的节点越应该被优先访问。估价函数用如下公式表示：$f(n)=h(n)+d(n)$，$h(n)$是对于当前状态的估计值，例如曼哈顿距离，$d(h)$是已经付出的搜索代价，例如节点的深度。

A*算法和A搜索算法的不同之处在于，A*保证对于所有节点都有：$h(n) \leq h^*(n)$，其中$h^*(n)$为当前状态到目的状态的最优路径的代价。

在本次实验中，我们选择曼哈顿距离为$h(n)$，节点深度为$d(n)$，在实际考虑中：所有放错位的数码个数、后续节点不正确的数码个数都可以作为估价方法，也可以选择多个估价方法并给予适当权重来加快搜索速度。

2.2 算法步骤

1. 将起始节点放入 explore_list 中
2. 如果 explore_list 为空，则搜索失败，问题无解，否则循环求解
    1. 取出 explore_list 的估价函数最小的节点，置为当前节点 current_state，若 current_state 为目标节点，则搜索成功，计算解路径，退出
    2. 寻找所有与 current_state 邻接且未曾被发现的节点，插入到 explore_list 中，并加入 visited_list 中，表示已发现

三、原程序说明

原程序在搜索基础上使用了估价函数来进行优化，但无法求得最优解。原程序在每一个状态的下一个可行状态中选择评估函数最小的一个状态作为下一步，搜索的节点为链状结构（可能有环，而一般的A*搜索结果是树状结构）。

程序步骤：

1. 创建 openTable（队列结构） 和 closeTable，并将初始节点加入 openTable
2. 如果 openTable 为空，则结束程序，问题无解，否则循环求解
    1. 从 openTable Pop出一个状态并将其加入到 closeTable 中
    2. 确定下一步的节点：从当前可行状态中寻找下一步可行状态，选择估价值最小的一个加入到 openTable 中
    3. 判断当前状态和终止状态是否相等，若相等则结束程序输出路径

原程序没有充分利用 closeTable，因此可能会出现死循环（未证明）：

- 一个状态在选择它的下一个状态时不会选择父亲状态，但是有可能选择父亲状态的父亲状态（即爷爷状态），导致有可能出现一个长度至少为3的回路。

要修改程序，必须修改2-1、2-2 步骤

- 选取的新状态应该是 openTable 估价值最小的状态，
- 应该将当前所有可行状态加入到 openTable 中，而不是仅加入估价值最小的状态，
- 应充分利用 closeTable，状态不能重复出现

四、程序修改

我们在源程序的基础上修改程序，使其能够实现最优求解。

正确的A*搜索算法步骤：

- 将起始节点放入 explore_list 中
- 如果 explore_list 为空，则搜索失败，问题无解，否则循环求解
    - 取出 explore_list 的估价函数最小的节点，置为当前节点 current_state
    - 若 current_state 为目标节点，则搜索成功，计算解路径，退出
    - 寻找所有与 current_state 邻接且未曾被发现的节点，插入到 explore_list 中，并加入 visited_list 中，表示已发现

4.1 程序修改

- **State类：表示一个状态**
    - 在源代码的基础上，做了以下修改：
        - 删除`def solve(self)`方法：将搜索职责转移到Solution类上，State类只负责管理当前的状态。
        - 增加属性值`self.d`：状态的深度，并在`nextStep()`方法中维护，起始状态的深度为0，终点状态的深度为最右搜索路径长度。
        - 修改`getFunctionValue(self)`方法：返回值为哈夫曼距离和状态深度的和，这是一个满足A*搜索的评估函数。
- **Solution类：负责求解N数码问题**
    - 记录起始状态、结束状态、当前状态和搜索结点总数，并用三个链表记录待检查状态链表、已检查状态链表和最优路径。
    - `getBestState`方法：遍历待检查状态链表 explore_list，返回估价函数最小的状态
    - `isVisited`方法：遍历已检查状态链表 visited_list， 返回 True 如果该状态已经存在于 visited_list 中
    - `getPath`方法：获取最终解路径，保存在 path_list 中
    - `AStarSolve`**方法**：使用A*搜索策略求解问题
    - `BFSSolve`**方法**：使用BFS策略求解问题，用于测试A*算法是否正确

4.2 复杂度分析

由于我们使用普通的链表来记录记录待检查状态链表（`explore_list`）、已检查状态链表（`visited_list`）和最优路径（`path_list`）：

- 每次获取估价函数最小的状态时需要遍历`explore_list`，时间复杂度为$O(n)$
- 每次判断一个状态是否已经被搜索过时需要遍历`visited_list`，时间复杂度为$O(n)$

设最终所搜的节点数目为M，有复杂度为$O(M^2)$

4.3 代码优化

在修改程序的基础上对待检查状态链表（`explore_list`）和已检查状态链表（`visited_list`）进行优化：

- 考虑它们的特性，用一个优先队列来实现 `explore_list`，用一个哈希表（字典）来实现 `visited_list`，可以将每次获取估价函数最小的状态和每次判断一个状态是否已经被搜索过的复杂度都降为$O(1)$
- 而`explore_list`插入节点并维护优先队列结构的复杂度为$O(\log(n))$

从而让整个算法复杂度降低到$O(M\log(M))$，从下面的测试中可以看到优化的运行速度优于优化前的运行速度。

最后，可以通过优化估价函数来缩短搜索次数，例如使用错误码数、曼哈顿距离的加权。

五、测试结果

5.1 使用BFS验证A*算法正确性

- 对于8数码测试样例：

```
begin = State(np.array([[1, 5, 2],[7, 0, 4],[6, 3, 8]]))
end = State(np.array([[1, 2, 3],[4, 5, 6],[7, 8, 0]]))
```

- **原程序测试结果：（非最优路径）**

```
...
Total steps is 28
```

- BFS结果：耗时长，访问节点数多，求得最短路径及其长度 14 。

```
...
Total search node is 5905
Total steps is 14
Totally cost is 105.18757700920105 s
```

- 优化前的结果：正确求得最短路径，耗时0.02秒

```
...
Total search node is 53
Total steps is 14
Totally cost is 0.020945310592651367 s
```

- 优化后的结果：正确求得最短路径，耗时0.01秒

```
...
Total search node is 52
Total steps is 14
Totally cost is 0.010965108871459961 s
```

5.2 优化前后效率对比

- 对于下面八数码测试样例：

```
begin = State(np.array([[1, 3, 2],[4, 5, 6],[8, 7, 0]]))
end = State(np.array([[1, 2, 3],[4, 5, 6],[7, 8, 0]]))
```

- 优化前的结果：需要 9 秒才能求出最优解

```
...
Total search node is 1664
Total steps is 20
Totally cost is 9.302738666534424 s
```

- 优化后的结果：只需要 0.6 秒就能求出最优解

```
...
Total search node is 2730
Total steps is 20
Totally cost is 0.641481876373291 s
```

5.3 24数码问题求解测试

- 对于以下24数码的测试样例：

```
begin = State(np.array([[ 1,  2,  3,  4,  5],
                        [ 6, 12,  8,  9, 10],
                        [11,  7, 13, 14, 15],
                        [16, 19, 18, 17, 20],
                        [21, 22, 23, 24,  0]]))

end = State(np.array([[ 1,  2,  3,  4,  5],
                      [ 6,  7,  8,  9, 10],
                      [11, 12, 13, 14, 15],
                      [16, 17, 18, 19, 20],
                      [21, 22, 23, 24,  0]]))
```

- AStarSearchOptimized 运行输出：求得最短路径及其长度为26，此时非优化A*与BFS无法在可接受时间内完成求解。

```
...
->
26:
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 0 24
->
27:
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15
16 17 18 19 20
```

```
21 22 23 24 0
->
Total search node is 91640
Total steps is 26
Totally cost is 36.743870973587036 s
```

附录

附录 1 - 重写程序 AStarSearch.py

```python
import numpy as np
import time

class Solution:

    # State:   self.begin_state   起始状态
    # State:   self.end_state     结束状态
    # State:   self.current_state 当前状态
    # int:     self.searched_num  搜索结点总数
    # State[]: self.explore_list  待检查状态链表
    # State[]: self.visited_list  已检查状态链表
    # State[]: self.path_list     搜索最优路径

    # 初始化A*搜索
    def __init__(self, begin_state, end_state):
        self.begin_state = begin_state
        self.end_state = end_state
        self.current_state = None
        self.searched_num = 0
        self.explore_list = []
        self.visited_list = []
        self.path_list = []

    # 返回 explore_list 中 估价函数最小的节点
    def getBestState(self):

        return_state = self.explore_list[0]
        for explore_state in self.explore_list:
            if explore_state.f < return_state.f:
                return_state = explore_state

        return return_state


    # isVisited 返回 True 如果该状态已经存在于 visited_state 中
    def isVisited(self, next_state):
        for visited_state in self.visited_list:
            if (visited_state.state == next_state.state).all():
                return True
        return False

    # getPath 获取最终解路径，保存在 path_list 中
    def getPath(self):
        temp_state = self.current_state
        self.path_list.append(self.end_state)

```

```
        while temp_state.parent and temp_state.parent != self.begin_state:
            self.path_list.append(temp_state.parent)
            temp_state = temp_state.parent

        self.path_list.append(self.begin_state)
        self.path_list.reverse()

    # AStarSolve 搜索
    def AStarSolve(self):

        # 将起始节点放入 explore_list 中
        self.explore_list.append(self.begin_state)
        self.visited_list.append(self.begin_state)

        # 如果 explore_list 为空，则搜索失败，问题无解，否则循环求解
        while len(self.explore_list) > 0:
            self.searched_num += 1
            if self.searched_num % 1000 == 0:
                print(self.searched_num)

            # 取出 explore_list 的估价函数最小的节点，置为当前节点 current_state
            # 若 current_state 为目标节点，则搜索成功，计算解路径，退出
            self.current_state = self.getBestState()
            self.explore_list.remove(self.current_state)
            if (self.current_state.state == self.end_state.state).all():
                self.getPath()
                break

            # 寻找所有与 current_state 邻接且未曾被发现的节点，插入到
explore_list 中
            # 并加入 visited_list 中，表示已发现
            next_list = self.current_state.nextStep()
            for next_state in next_list:
                if not self.isVisited(next_state):
                    self.explore_list.append(next_state)
                    self.visited_list.append(next_state)

    # BFS 用于测试A*是否求得了最优解
    def BFSSolve(self):

        # 将起始节点放入 explore_list 中
        self.explore_list.append(self.begin_state)
        self.visited_list.append(self.begin_state)

        # 如果 explore_list 为空，则搜索失败，问题无解，否则循环求解
        while len(self.explore_list) > 0:
            self.searched_num += 1
            if self.searched_num % 1000 == 0:
                print(self.searched_num)

            # 取出 explore_list 的估价函数最小的节点，置为当前节点 current_state
            # 若 current_state 为目标节点，则搜索成功，计算解路径，退出
            self.current_state = self.explore_list[0]
            del(self.explore_list[0])
            if (self.current_state.state == self.end_state.state).all():
                self.getPath()
                break

```

```
            # 寻找所有与 current_state 邻接且未曾被发现的节点，插入到 explore_list 中
            # 并加入 visited_list 中，表示已发现
            next_list = self.current_state.nextStep()
            for next_state in next_list:
                if not self.isVisited(next_state):
                    self.explore_list.append(next_state)
                    self.visited_list.append(next_state)


class State:

    # int[][]:  self.state         当前状态的数码矩阵
    # string[]: self.direction     当前空块的可移动方向
    # State:    self.parent        当前状态的上一个状态
    # int:      self.f             当前状态的估价函数值
    # int:      self.d             当前状态的深度

    dist_state = None

    # 初始化一个状态
    def __init__(self, state, directionFlag=None, parent=None, f=0, d=0):
        self.state = state
        self.direction = ['up', 'down', 'right', 'left']
        if directionFlag:
            self.direction.remove(directionFlag)
        self.parent = parent
        self.f = f
        self.d = d

    # 获得当前状态下空快可移动的方向
    def getDirection(self):
        return self.direction

    # 设置状态的估价函数值
    def setF(self, f):
        self.f = f
        return

    # 打印一个状态的数码矩阵
    def showInfo(self):
        for i in range(len(self.state)):
            for j in range(len(self.state)):
                print(self.state[i, j], end=' ')
            print(" ")
        print('->')
        return

    # 获取0点（即空块在矩阵中的位置）
    def getZeroPos(self):
        postion = np.where(self.state == 0)
        return postion

    # 评估函数计算：目前的评估函数为当前状态与目标状态的哈夫曼距离
    def getFunctionValue(self):
        cur_node = self.state.copy()
        fin_node = self.dist_state.state.copy()
        dist = 0
```

```
        error = 0
        N = len(cur_node)
        for i in range(N):
            for j in range(N):
                if cur_node[i][j] != fin_node[i][j]:
                    index = np.argwhere(fin_node == cur_node[i][j])
                    x = index[0][0]  # 最终x距离
                    y = index[0][1]  # 最终y距离
                    dist += (abs(x - i) + abs(y - j))
                    error += 1

        return dist + self.d

    # 寻找当前状态的下一个可行状态集合
    def nextStep(self):
        if not self.direction:
            return []
        subStates = []
        boarder = len(self.state) - 1
        # 获取0点位置
        x, y = self.getZeroPos()
        # 向左
        if 'left' in self.direction and y > 0:
            s = self.state.copy()
            tmp = s[x, y - 1]
            s[x, y - 1] = s[x, y]
            s[x, y] = tmp
            news = State(s, directionFlag='right', parent=self,d=self.d+1)
            news.setF(news.getFunctionValue())
            subStates.append(news)
        # 向上
        if 'up' in self.direction and x > 0:
            # it can move to upper place
            s = self.state.copy()
            tmp = s[x - 1, y]
            s[x - 1, y] = s[x, y]
            s[x, y] = tmp
            news = State(s, directionFlag='down', parent=self,d=self.d+1)
            news.setF(news.getFunctionValue())
            subStates.append(news)
        # 向下
        if 'down' in self.direction and x < boarder:
            # it can move to down place
            s = self.state.copy()
            tmp = s[x + 1, y]
            s[x + 1, y] = s[x, y]
            s[x, y] = tmp
            news = State(s, directionFlag='up', parent=self,d=self.d+1)
            news.setF(news.getFunctionValue())
            subStates.append(news)
        # 向右
        if self.direction.count('right') and y < boarder:
            # it can move to right place
            s = self.state.copy()
            tmp = s[x, y + 1]
            s[x, y + 1] = s[x, y]
            s[x, y] = tmp
            news = State(s, directionFlag='left', parent=self,d=self.d+1)
```

```
            news.setF(news.getFunctionValue())
            subStates.append(news)
        # 所有状态
        return subStates


if __name__ == '__main__':

    begin = State(np.array([[1, 5, 2],
                            [7, 0, 4],
                            [6, 3, 8]]))
    end = State(np.array([[1, 2, 3],
                          [4, 5, 6],
                          [7, 8, 0]]))

    State.dist_state = end

    time_start = time.time()
    solution = Solution(begin, end)
    solution.AStarSolve()
    # solution.BFSSolve()
    time_end = time.time()

    if solution.path_list:
        i = 0
        for node in solution.path_list:
            i += 1
            print(str(i) + ": ")
            node.showInfo()
        print("Total search node is %d" % solution.searched_num)
        print("Total steps is %d" % (len(solution.path_list) - 1))
    print('Totally cost is', time_end - time_start, "s")
```

附录 2 - 优化代码 AStarSearchOptimized.py

> 注：State类与附录一：AStarSearch.py 中的State类一致，故在这里省略，主函数也省略

```
import numpy as np
from queue import PriorityQueue
import time

class Solution:

    # State:   self.begin_state   起始状态
    # State:   self.end_state     结束状态
    # State:   self.current_state 当前状态
    # int:     self.searched_num  搜索结点总数
    # State[]: self.explore_list  待检查状态链表
    # State[]: self.visited_list  已检查状态链表
    # State[]: self.path_list     搜索最优路径

    # 初始化A*搜索
    def __init__(self, begin_state, end_state):
        self.begin_state = begin_state
        self.end_state = end_state
        self.current_state = None
        self.searched_num = 0
```

```
        self.explore_list = PriorityQueue()
        self.visited_list = {}
        self.path_list = []

    # 返回 explore_list 中 估价函数最小的节点
    def getBestState(self):
        return self.explore_list.get() # 返回并删除

    # isVisited 返回 True 如果该状态已经存在于 visited_state 中
    def isVisited(self, next_state):
        if self.visited_list.get(next_state) == None:
            return False
        else:
            return True

    # getPath 获取最终解路径，保存在 path_list 中
    def getPath(self):
        temp_state = self.current_state
        self.path_list.append(self.end_state)

        while temp_state.parent and temp_state.parent != self.begin_state:
            self.path_list.append(temp_state.parent)
            temp_state = temp_state.parent

        self.path_list.append(self.begin_state)
        self.path_list.reverse()

    # AStarSolve 搜索
    def AStarSolve(self):

        # 将起始节点放入 explore_list 中
        self.explore_list.put(self.begin_state)
        self.visited_list[self.begin_state] = self.begin_state

        # 如果 explore_list 为空，则搜索失败，问题无解，否则循环求解
        while not self.explore_list.empty():
            self.searched_num += 1
            if self.searched_num % 1000 == 0:
                print(self.searched_num)

            # 取出 explore_list 的估价函数最小的节点，置为当前节点 current_state
            # 若 current_state 为目标节点，则搜索成功，计算解路径，退出

            self.current_state = self.getBestState()

            if (self.current_state.state == self.end_state.state).all():
                self.getPath()
                break

            # 寻找所有与 current_state 邻接且未曾被发现的节点，插入到explore_list中
            # 并加入 visited_list 中，表示已发现
            next_list = self.current_state.nextStep()
            for next_state in next_list:
                if not self.isVisited(next_state):
                    self.explore_list.put(next_state)
                    self.visited_list[next_state] = next_state
```