



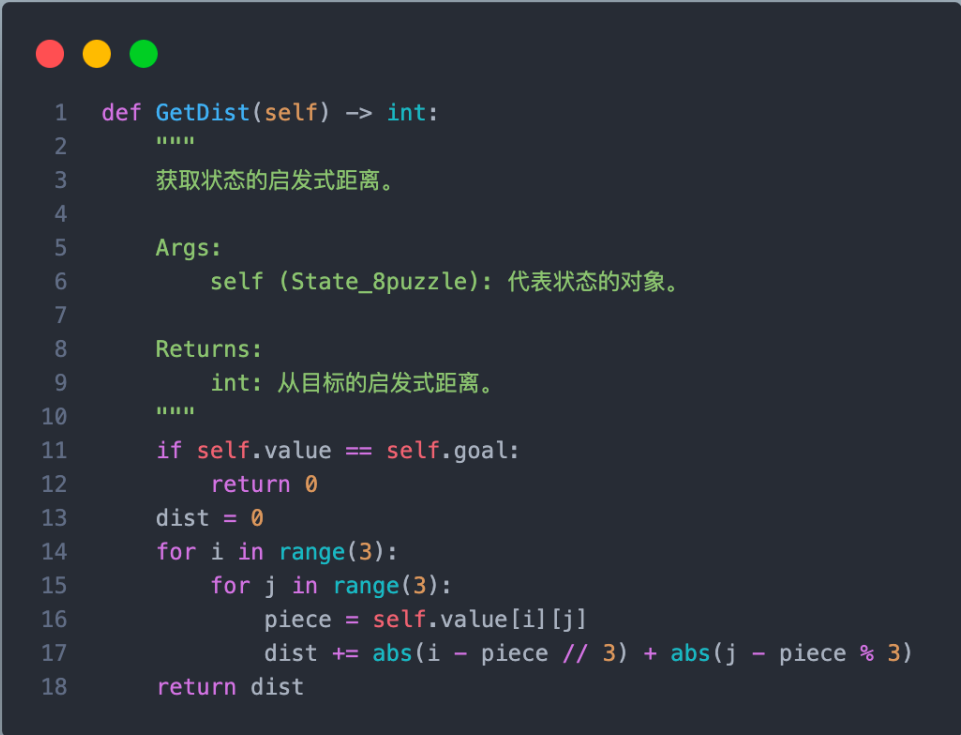
## 步骤二：扩展状态类

为 8 数码问题添加特定的启发式函数和子状态生成功能。

在 `State_8puzzle` 类中实现 `GetDist` 方法，用于计算当前状态与目标状态之间的启发式距离。在 8 数码问题中，这通常是通过计算每个数字到其目标位置的曼哈顿距离来完成的

类中的接口：

用于获取启发式距离的接口 `GetDist()`



```
1  def GetDist(self) -> int:
2      """
3      获取状态的启发式距离。
4
5      Args:
6          self (State_8puzzle): 代表状态的对象。
7
8      Returns:
9          int: 从目标的启发式距离。
10     """
11     if self.value == self.goal:
12         return 0
13     dist = 0
14     for i in range(3):
15         for j in range(3):
16             piece = self.value[i][j]
17             dist += abs(i - piece // 3) + abs(j - piece % 3)
18     return dist
```

生成子状态:

在 `State_8puzzle` 类中实现 `CreateChildren` 方法，用于生成当前状态的所有可能子状态。它将查找空白位置（数字 0），然后尝试将空白位置上、下、左、右移动，以生成新的状态

```

1  def CreateChildren(self) -> None:
2      """
3      生成子状态。
4
5      Args:
6          self (State_8puzzle): 代表状态的对象。
7      """
8      if not self.children:
9          i, j = [(i, j) for i in range(3) for j in range(3) if self.value[i][j] == 0][0]
10         dirs = [(1, 0), (0, 1), (-1, 0), (0, -1)]
11         for dir in dirs:
12             ni, nj = i + dir[0], j + dir[1]
13             if 0 <= ni < 3 and 0 <= nj < 3:
14                 val = [row[:] for row in self.value]
15                 val[i][j], val[ni][nj] = val[ni][nj], val[i][j]
16                 child = State_8puzzle(val, self)
17                 self.children.append(child)

```

结果：实现了 State\_8puzzle 类，可以计算与目标状态的启发式距离并生成子状态。

步骤三：实现 A\*算法的主类

使用优先队列来管理待探索的状态。

创建一个 PriorityQueue 类，它使用 heapq 模块来管理状态的优先队列。优先队列保证了每次弹出的总是启发式距离最小的状态

优先队列依靠 List[Tuple[int, State]]实现

```

1  class PriorityQueue:
2      """
3      支持A*算法的优先队列。
4
5      使用Python的heapq模块按其启发式值维护状态。
6
7      Attributes
8          queue (list): 按启发式排序的状态列表。
9      """
10     queue: List[Tuple[int, State]]

```

步骤四:A\*算法

在 AStar 类中实现 Solve 方法，这个方法会执行 A\*搜索算法。它从起始状态开始，不断地生成子状态，并将它们添加到优先队列中。同时，它会跟踪访问过的状态以避免重复工作。一旦找到目

标状态，它会构建并返回解决方案路径

```
1 def Solve(self) -> List[List[List[int]]]:
2     """
3     使用A*算法解决8数码问题。
4     Returns:
5         list: 从开始到目标的解决方案路径。
6     """
7     startState = State_8puzzle(self.start, None, self.start, self.goal)
8     count = 0
9     self.priorityQueue.push(startState)
10    while not self.path and self.priorityQueue:
11        closestChild = self.priorityQueue.pop()
12        closestChild.CreateChildren()
13        self.visitedQueue.append(closestChild.value)
14        for child in closestChild.children:
15            if child.value not in self.visitedQueue:
16                count += 1
17                if not child.dist:
18                    self.path = child.path
19                    break
20                self.priorityQueue.push(child)
21    if not self.path:
22        print("Goal not possible!")
23    return self.path
```

这段代码中

首先初始化起始状态：将初始状态作为一个 `State_8puzzle` 对象创建，并命名为 `startState`。这里还没有创建任何子状态。

然后将起始状态推入优先队列：将 `startState` 推入一个名为 `priorityQueue` 的优先队列中。这个队列会根据状态的 `dist` 值（即状态的启发式评估）来决定顺序。

循环搜索：进入一个循环，不断进行搜索直到找到路径或优先队列为空（即没有更多的状态可以搜索）。

其次弹出最有可能的状态：从优先队列中弹出启发式评估值最小的状态，称为 `closestChild`。这个状态被认为是离目标最近的状态。

弹出状态后生成子状态：对 `closestChild` 调用 `CreateChildren` 方法，生成所有可能的子状态。

将当前状态标记为已访问：将 `closestChild` 的值添加到一个名为 `visitedQueue` 的列表中，以防止重复访问相同的状态。

遍历子状态：对于 `closestChild` 的每一个子状态，执行以下操作：

- 1.检查子状态是否已在 `visitedQueue` 中，若在，则不处理该子状态，继续下一个。
- 2.如果子状态不在 `visitedQueue` 中，将子状态的生成次数增加 1（用 `count` 变量记录）。
- 3.检查子状态是否已经是目标状态（`dist` 值为 0），如果是，则将这个子状态的路径赋值给 `self.path` 并跳出循环。
- 4.如果子状态不是目标状态，则将其推入 `priorityQueue` 中，等待后续的处理。

最后检查是否找到解决方案：

- 1.如果 `self.path` 不为空，则表示找到了解决方案，方法将返回这个路径。
- 2.如果循环结束后 `self.path` 仍然为空，说明没有找到解决方案，打印 “Goal not possible!” 并返回一个空的路径

结果：成功实现了 A\* 算法，并通过 PriorityQueue 类有效地管理了待探索的状态。

步骤五:检查可解性 (isSolvable):

实现一个函数 isSolvable，它检查 8 数码问题是否有解。它通过计算谜题状态中的逆序数来实现。只有当逆序数是偶数时，问题才是可解的。

```
1 def isSolvable(grid: List[List[int]]) -> bool:
2     """
3     检查给定的8数码问题是否有解。
4
5     Args:
6         grid (list): 表示8数码状态的3x3列表。
7
8     Returns:
9         bool: 如果问题有解，则为True，否则为False。
10    """
11    flat_list = [item for sublist in grid for item in sublist]
12    inversions = 0
13
14    for i in range(len(flat_list)):
15        for j in range(i + 1, len(flat_list)):
16            # Don't count zero as it represents the empty space
17            if flat_list[i] != 0 and flat_list[j] != 0 and flat_list[i] > flat_list[j]:
18                inversions += 1
19
20    return inversions % 2 == 0
```

这段代码的具体实现方式:

1.将 3x3 网格转换为一维数组 flat\_list。

2.初始化逆序数计数变量 inversions。

3.使用两层嵌套循环计算逆序数:

外循环变量 i 从 0 开始到列表的倒数第二个元素。

内循环变量 j 从 i+1 开始到列表的最后一个元素。

如果 flat\_list[i] 和 flat\_list[j] 都不是 0 (即不考虑空格)，并且 flat\_list[i] 大于 flat\_list[j]，那么逆序数 inversions 增加 1。

4.返回逆序数的奇偶性:

5.如果 inversions 是偶数，返回 True，表示 8 数码问题有解。

6.如果是奇数，返回 False，表示问题无解。

步骤六：解决八数码问题

实现一个函数 solve，它首先检查起始状态是否可解。如果可解，它将创建一个 AStar 实例，调用 Solve 方法，并输出到达目标状态的步骤。如果不可解，它将打印出不可解的消息。

```

1  def solve(
2      start: List[List[int]],
3      goal: List[List[int]]
4  ) -> List[List[List[int]]]:
5      """
6      使用A*算法解决8数码问题。
7
8      Args:
9          start (list): 谜题的起始状态。
10         goal (list): 谜题的目标状态。
11
12     Returns:
13         list: 从开始到目标的解决方案路径。
14     """
15     if isSolvable(start):
16         print("Solving...")
17         a = AStar(start, goal)
18         a.Solve()
19         for i in a.path:
20             print(f"step {str(a.path.index(i))}")
21             for row in i:
22                 print(row)
23             print("")
24         print(f"Solved in {str(len(a.path))} moves.")
25         return a.path
26     else:
27         print("The provided start state is not solvable.")
28         return []

```

该函数实现方式:

如果 `isSolvable` 返回 `False`, 则打印出起始状态无解的信息, 并返回空列表。

如果返回 `True`, 则继续执行。

初始化: 打印 "Solving..." 表明开始解决问题, 并创建 `AStar` 类的实例 `a`, 将起始状态和目标状态传递给它。

求解: 调用 `AStar` 类的 `Solve` 方法来进行实际的求解过程。

`Solve` 方法会创建一个初始状态 `startState` 并将其推入优先队列 `priorityQueue`。

然后, 它会进入一个循环, 不断从优先队列中取出启发式距离最小的状态 `closestChild`, 生成这个状态的所有可能的子状态, 并将其推入优先队列。

每产生一个子状态, 就检查它是否已在 `visitedQueue` 中。如果不在, 且其启发式距离为 0 (即已经是目标状态), 则找到了解决方案, 将该子状态的路径赋值给 `self.path` 并退出循环。

输出路径: 如果找到了解决方案, 则 `self.path` 会包含从起始状态到目标状态的路径。

函数会遍历这个路径, 对于路径中的每个状态, 按步骤顺序打印出来。

最后, 打印解决问题所需的步数。

返回结果：返回路径，它是一个由状态组成的列表，每个状态是一个 3x3 的数字列表。

#### 四. 实验总结

通过这次实验，深入了解了 A\*搜索算法的工作原理和其在 8 数码问题中的应用。

8 数码问题是一个经典的人工智能问题，通过解决它，我们学习到了启发式搜索的重要性和优点。

实验中，我们遇到了一些挑战，如如何有效地管理待探索的状态和如何设计一个有效的启发式函数，但通过不断的调试和优化，我们成功地克服了这些挑战。

总的来说，这次实验不仅提高了我们的编码能力，还加深了我们对人工智能搜索算法的理解。