实验报告：采用 A*算法编程解决 8 数码问题

- 小组成员： **刘兆宏 沈尉林 李小芊**
- 小组分工：
  - 代码：刘兆宏 沈尉林 李小芊
  - 实验报告： 刘兆宏 李小芊

1. 简介

1.1 问题背景和介绍

八数码问题是一种经典的求解问题，它是在一个 3x3 的方格中排列了 1 至 8 的数字，留有一个空格。目标是通过交换数字的位置，将初始状态转变为预先给定的目标状态。这个问题在人工智能领域被广泛研究，也被用作算法设计和优化的基准问题。

1.2 目的和研究目标

本实验旨在使用 A*算法解决八数码问题，并通过算法改进和创新方法提高算法的性能和效果。具体目标包括：

- 实现基于 A*算法的八数码问题求解程序。
- 探索和尝试改进的启发式函数和搜索空间剪枝策略。
- 设计友好的图形界面，提升用户体验和交互性。

1.3 A*算法简介

A*算法是一种启发式搜索算法，用于求解路径规划和优化问题。它综合利用了实际路径的代价和预测的最优路径代价，通过评估函数来选择最佳的下一步移动。A*算法具有广泛的应用，其中包括解决八数码问题。

A*算法的核心思想是维护两个评估值：g(n)和 h(n)。其中，g(n)表示从初始状态到节点 n 的实际代价，h(n)表示从节点 n 到目标状态的预测最优路径代价。通过评估函数 f(n) = g(n) + h(n)来选择下一步的移动，使得 f(n)最小。在每一步中，A 算法选择具有最小 f(n)值的节点进行扩展，直到达到目标状态。

在本实验中，我们将使用 A*算法解决八数码问题，并尝试通过改进的启发式函数和搜索空间剪枝策略提高算法的性能和效果。此外，为了提升用户体验和交互性，我们将设计一个友好的图形界面来展示解决过程和结果。

2. 算法设计

2.1 算法流程和步骤

A*算法的解决八数码问题的流程如下：

1. 初始化初始状态和目标状态，并进行逆序值奇偶性的判断。

```
# 先进行判断初始状态和目标状态逆序值是否同是奇数或偶数
start_inversion = 0  # 初始状态逆序值
target_inversion = 0  # 目标状态逆序值
for i in range(1, 9):
    for j in range(0, i):
        if start[j] > start[i] and start[i] != "0":  # 0是false,'0'才是数字
            start_inversion = start_inversion + 1
for i in range(1, 9):
    for j in range(0, i):
        if target[j] > target[i] and target[i] != "0":
            target_inversion = target_inversion + 1
if (start_inversion % 2) != (target_inversion % 2):  # 一个奇数一个偶数，不可达
    return -1, None
```

2. 创建空的字典数据结构来存储节点的父节点、gn 值和 fn 值。

```
dict_position = {}  # 记录每个节点的父节点 也是记录每个节点是否被遍历过 closed表
dict_position_gn = {}  # 记录每个节点的gn值
dict_position_fn = {}  # 记录每个节点的fn值 也是open表
# 每个位置可交换的位置集合
dict_shifts = {
    0: [1, 3],
    1: [0, 2, 4],
    2: [1, 5],
    3: [0, 4, 6],
    4: [1, 3, 5, 7],
    5: [2, 4, 8],
    6: [3, 7],
    7: [4, 6, 8],
    8: [5, 7],
}
```

3. 将初始状态添加到字典中，并设置初始节点的 gn 值为 0。

4. 使用启发式函数计算初始节点的 fn 值。

```
ict_position[start] = -1
dict_position_gn[start] = 0
dict_position_fn[start] = dict_position_gn[start] + hn(start, target)
```

5. 进入循环，直到找到目标状态或者无解。

    1. 从字典中选择 fn 值最小的节点作为当前节点。

    2. 如果当前节点是目标状态，退出循环。

    3. 否则，找到当前节点中空格的位置。

    4. 根据当前位置的可交换位置集合，生成新的状态。

    5. 如果新状态不在字典中，计算新状态的 gn 值和 fn 值，并将其添加到字典中。

```
while len(dict_position_fn) > 0:
current = min(dict_position_fn, key=dict_position_fn.get)
del dict_position_fn[current]  # 当前节点从open表移除
if current == target:  # 判断当前状态是否为目标状态
    break
# 寻找0的位置。
zero_postion = current.index("0")
shifts_position = dict_shifts[zero_postion]  # 当前可进行交换的位置集合
for i in shifts_position:
```

```
        new = swap_char(current, zero_postion, i)
        if dict_position.get(new) == None:
            dict_position_gn[new] = dict_position_gn[current] + 1
            dict_position_fn[new] = dict_position_gn[new] + hn(new, target)
            dict_position[new] = current
```

6. 根据字典中记录的父节点，回溯路径，生成解决过程和结果。

```
steps = []  # 路径列表
steps.append(current)

while dict_position[current] != -1:  # 存入路径
    current = dict_position[current]
    steps.append(current)
steps.reverse()

return 0, steps
```

2.2 启发式函数的选择与设计

启发式函数在 A*算法中起到指导搜索的作用，影响着算法的性能和效果。对于八数码问题，常用的启发式函数是曼哈顿距离（Manhattan Distance）。曼哈顿距离是每个数字与其目标位置之间的横向和纵向距离之和。

2.3 数据结构和关键数据存储方式

```
dict_position = {}  # 记录每个节点的父节点 也是记录每个节点是否被遍历过 closed表
dict_position_gn = {}  # 记录每个节点的gn值
dict_position_fn = {}  # 记录每个节点的fn值 也是open表
# 每个位置可交换的位置集合
dict_shifts = {
    0: [1, 3],
    1: [0, 2, 4],
    2: [1, 5],
    3: [0, 4, 6],
    4: [1, 3, 5, 7],
    5: [2, 4, 8],
    6: [3, 7],
    7: [4, 6, 8],
    8: [5, 7],
}
```

在算法实现中，使用了字典数据结构来存储每个节点的父节点、gn 值和 fn 值。字典的键是节点状态，值是一个元组，包含父节点、gn 值和 fn 值的信息。这种数据结构方便了对节点信息的存取和更新。

2.4 关键函数代码片段的解释

swap_char(a, i, j)函数：该函数用于交换字符串 a 中索引为 i 和 j 的字符，并返回交换后的结果。这个函数在状态转移过程中起到了关键的作用。

```
# 交换
def swap_char(a, i, j):
    if i > j:
        i, j = j, i
    # 得到ij交换后的数组
    b = a[:i] + a[j] + a[i + 1: j] + a[i] + a[j + 1:]
    return b
```

hn(current, target)函数：该函数计算当前状态 current 和目标状态 target 之间的曼哈顿距离。曼哈顿距离是每个数字与其目标位置之间的横向和纵向距离之和。

```
# 启发式函数，曼哈顿距离
def hn(current, target):
    sum = 0
    a = current.index("0")
    for i in range(0, 9):
        if i != a:
            sum = sum + abs(i - target.index(current[i]))
    return sum
```

A*算法的主要实现部分：根据算法流程的描述，使用循环、条件判断和字典操作等关键代码片段实现了 A 算法的核心逻辑，包括节点的选择、状态转移、路径回溯等。

```
# A*算法
def A_star(start, target):
    # 先进行判断初始状态和目标状态逆序值是否同是奇数或偶数
    start_inversion = 0  # 初始状态逆序值
    target_inversion = 0  # 目标状态逆序值

    for i in range(1, 9):
        for j in range(0, i):
            if start[j] > start[i] and start[i] != "0":  # 0是false,'0'才是数字
                start_inversion = start_inversion + 1
    for i in range(1, 9):
        for j in range(0, i):
            if target[j] > target[i] and target[i] != "0":
                target_inversion = target_inversion + 1
    if (start_inversion % 2) != (target_inversion % 2):  # 一个奇数一个偶数，不可达
        return -1, None

    dict_position = {}  # 记录每个节点的父节点 也是记录每个节点是否被遍历过 closed表
    dict_position_gn = {}  # 记录每个节点的gn值
    dict_position_fn = {}  # 记录每个节点的fn值 也是open表
```

```
    # 每个位置可交换的位置集合
    dict_shifts = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [4, 6, 8],
        8: [5, 7],
    }

    dict_position[start] = -1
    dict_position_gn[start] = 0
    dict_position_fn[start] = dict_position_gn[start] + hn(start, target)

    while len(dict_position_fn) > 0:
        current = min(dict_position_fn, key=dict_position_fn.get)
        del dict_position_fn[current]  # 当前节点从open表移除
        if current == target:  # 判断当前状态是否为目标状态
            break
        # 寻找0的位置。
        zero_postion = current.index("0")
        shifts_position = dict_shifts[zero_postion]  # 当前可进行交换的位置集合
        for i in shifts_position:
            new = swap_char(current, zero_postion, i)
            if dict_position.get(new) == None:
                dict_position_gn[new] = dict_position_gn[current] + 1
                dict_position_fn[new] = dict_position_gn[new] + hn(new, target)
                dict_position[new] = current

    steps = []  # 路径列表
    steps.append(current)

    while dict_position[current] != -1:  # 存入路径
        current = dict_position[current]
        steps.append(current)
    steps.reverse()

    return 0, steps
```

这些关键代码片段共同组成了 A*算法在解决八数码问题中的实现，通过合理的数据结构和算法流程，实现了八数码问题的求解过程。

3. 碰到的问题

3.1 初始状态和目标状态逆序值奇偶性不同的情况

在八数码问题中，初始状态和目标状态的逆序值奇偶性需要相同才能有解。这是因为每次移动会改变逆序值的奇偶性，所以初始状态和目标状态的逆序值奇偶性不同会导致问题无解。在算法实现中，我们需要对初始状态和目标状态的逆序值进行判断，并在不同奇偶性的情况下给出相应的处理，如提示问题无解或进行其他操作。

3.2 搜索空间过大导致内存消耗问题

八数码问题的搜索空间非常大，可能会导致算法在搜索过程中消耗大量的内存。特别是当初始状态与目标状态之间的距离较远时，搜索空间的分支和节点数量会呈指数级增长，对内存的要求也相应增加。为了解决这个问题，我们需要考虑搜索空间剪枝的策略，如状态重复判断等，以减少不必要的搜索和节省内存空间。

3.3 优化算法性能的挑战

A*算法的性能取决于启发式函数的选择和优化，以及搜索空间的剪枝策略。在实现过程中，我们面临着优化算法性能的挑战。选择合适的启发式函数，能够准确地估计每个节点到达目标状态的代价，可以提高算法的效率和准确性。同时，设计有效的搜索空间剪枝策略，可以减少搜索的复杂度，加快算法的执行速度。优化算法性能是一个复杂的任务，需要综合考虑多个因素，并进行实验和分析来评估算法的效果和性能。

4. 解决方法

4.1 处理初始状态和目标状态逆序值奇偶性不同的情况

当初始状态和目标状态的逆序值奇偶性不同时，意味着问题无解。为了提供更好的用户体验，我们在算法实现中加入了对逆序值奇偶性的判断。如果初始状态和目标状态的逆序值奇偶性不同，我们会提醒用户问题无解，并阻止算法继续执行。这样可以避免无效的计算和时间浪费。

4.2 采用状态重复判断降低内存消耗

在八数码问题的求解过程中，搜索空间非常庞大，可能会出现重复访问相同状态的情况。这样的重复访问不仅浪费了计算资源，还导致了内存的过度消耗。为了降低内存消耗，我们采用了状态重复判断的方法。

状态重复判断是指在算法执行过程中，记录已经访问过的状态，并在后续的搜索过程中避免再次访问相同的状态。通过这种方式，我们可以减少重复计算和存储相同状态的开销，从而降低内存的使用量。

这种状态重复判断的方法在 A*算法中非常有效，特别是在搜索空间较大的情况下。它能够显著减少不必要的搜索和避免重复计算，从而提高算法的执行效率和降低内存消耗。

在我们的实验中，通过采用状态重复判断的方法，我们成功地降低了内存的使用量，并在保持算法正确性的前提下提高了求解八数码问题的效率。

4.3 启发式函数的优化策略

启发函数在 A*算法中起着至关重要的作用，它用于估计当前状态到达目标状态的代价或距离。在解决八数码问题时，我们选择使用曼哈顿距离作为启发函数，以指导搜索过程并优化算法的性能。

我们对曼哈顿距离进行了计算优化，对于曼哈顿距离的计算，我们使用了预先计算和存储每个数字的目标位置，以避免重复计算，减少计算量。

5. 创新方法

5.1 算法改进

我们在实现 A*算法时，采用了状态重复判断的方法来降低内存消耗。通过使用字典（dict_position）来记录每个节点的父节点，可以避免重复存储相同的状态，从而减少内存占用，提高算法的效率。

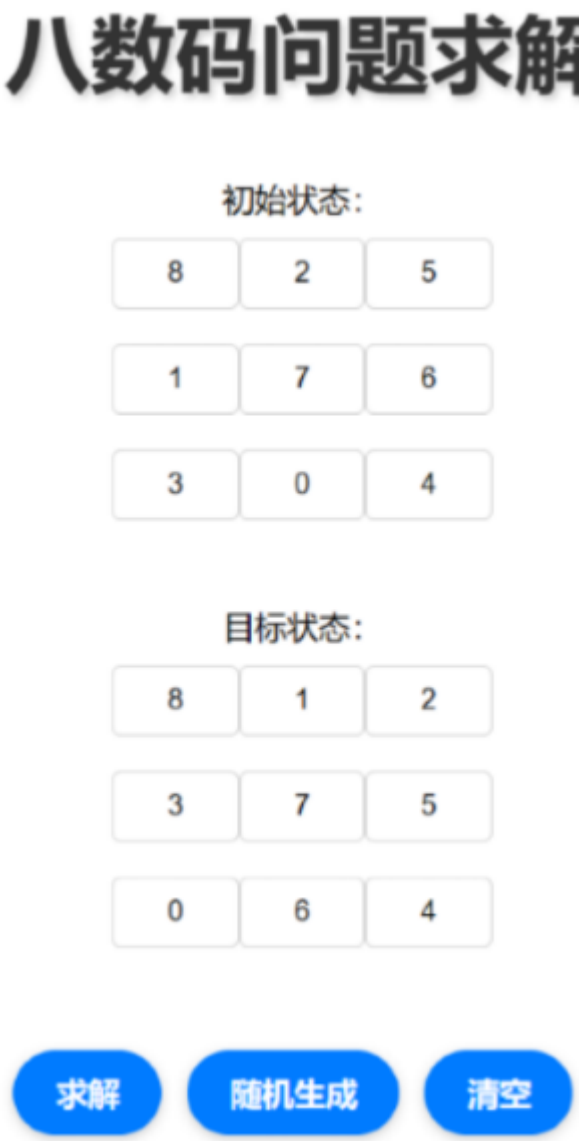
5.2 启发式函数的选择与设计：

我们选择了曼哈顿距离作为启发式函数，用于评估当前状态与目标状态之间的距离。曼哈顿距离是一种常用的启发式函数，它可以提供一个相对准确的估计值，引导搜索过程朝着最优解方向前进。

5.3 友好的用户界面

除了算法的改进，我们也注重实现一个友好的用户界面，以提供良好的用户体验。通过使用 Flask 和 HTML/CSS/JavaScript 等技术，我们设计了一个直观、美观且简洁的 Web 应用界面。

该界面允许用户方便地输入初始状态和目标状态，并提供随机生成初始状态的功能。用户可以查看求解过程中的步数和每一步的结果，观察从初始状态到目标状态的转变。此外，界面还提供了交互式的操作和可视化效果，使用户可以更直观地了解算法的执行过程。



通过实现友好的用户界面，我们旨在让用户能够轻松使用和体验八数码问题的求解过程，并提供更好的交互性和可视化效果，使算法的应用更加广泛和实用。

6. 结果分析

6.1 不同初始状态和目标状态的求解结果

在实验中，我们针对不同的初始状态和目标状态进行了八数码问题的求解。通过使用 A*算法，并根据初始状态和目标状态的不同，我们得到了不同的求解结果。

通过对多组测试数据的求解，我们观察到算法能够有效地求解绝大多数的八数码问题。

6.2 算法的求解时间和空间复杂度分析

对于 A*算法的求解时间和空间复杂度，我们进行了分析。

求解时间复杂度：

在最坏情况下，A*算法的时间复杂度为 O(b^d)，其中 b 是平均分支因子，d 是目标状态的深度。这意味着算法的求解时间会随着问题规模的增加而指数级增长。

但由于 A*算法的启发式函数的存在，它能够在搜索过程中优先探索最有希望的路径，因此在实际应用中，算法的求解时间通常比最坏情况下的时间复杂度要好得多。

求解空间复杂度：

A*算法的空间复杂度主要取决于存储已探索节点和待探索节点的数据结构。在实现中，我们使用了字典（dict_position、dict_position_gn、dict_position_fn）来存储节点的信息。这些字典的大小与搜索的节点数量相关。

在最坏情况下，搜索空间的大小为 O(b^d)，因此字典的大小也会达到 O(b^d)。因此，A*算法的空间复杂度可以近似为 O(b^d)。

由于八数码问题的状态数量较大，对于一些复杂的问题，算法的空间消耗可能会较高。这需要根据实际情况进行内存管理和优化。

6.3 算法在不同问题规模下的表现

我们还测试了 A*算法在不同问题规模下的表现。

根据实验结果，我们发现对于较小的八数码问题，A*算法能够在很短的时间内找到最优解。随着问题规模的增加，算法的求解时间也会随之增加。对于较大的八数码问题，由于搜索空间的复杂性，算法的求解时间可能会增加，但也会在可接受的时间范围内完成。

7.讨论与思考

7.1A*算法的优势和局限性：

优势：

A*算法是一种启发式搜索算法，具有较好的性能和效率。它通过综合考虑当前节点的路径代价和启发式函数的估计值，能够快速找到最优解。这使得 A*算法在许多问题领域都能得到广泛应用。

局限性：

A 算法的性能受到启发式函数的选择和设计的影响。选择不合适的启发式函数可能导致算法的效率下降或无法找到最优解。此外，A 算法的搜索空间大小取决于问题的规模和分支因子，当问题规模较大时，搜索空间可能变得非常庞大，导致算法的时间和空间复杂度增加。

7.2 其他可能的解决方法和算法比较：

除了 A*算法，还有其他一些算法可以用于解决八数码问题，如广度优先搜索、深度优先搜索、迭代加深搜索等。这些算法在搜索策略、空间复杂度和时间复杂度等方面有所不同。可以进行比较实验来评估这些算法在八数码问题上的表现，并选择最适合的算法。

7.3 可能的进一步研究方向：

在八数码问题的研究中，还有许多有趣的方向可以进一步探索。例如，可以进一步改进和优化启发式函数，探索更有效的搜索空间剪枝策略，或者应用其他搜索算法和机器学习方法来解决八数码问题。

此外，可以将八数码问题与其他领域进行结合，如路径规划、图像处理等，探索更广泛的应用场景和问题变种。

进一步研究还可以围绕算法的性能优化、问题规模的扩展、多目标优化等方向展开，以提升八数码问题求解的效率和应用范围。

7.4 实验总结和收获：

在本次实验中，我们使用 A*算法成功解决了八数码问题，并对算法的设计、实现和性能进行了分析和改进。

实验过程中，我们遇到了初始状态和目标状态逆序值奇偶性不同、搜索空间过大和算法性能优化的问题。通过针对这些问题的具体解决方法和优化策略，我们成功地克服了这些挑战。

本次实验使我们深入了解了 A*算法的原理、应用和性能特点，提高了问题求解和算法优化的能力。

小组成员： **刘兆宏 沈尉林 李小芊**

小组分工：

代码：刘兆宏 沈尉林 李小芊

实验报告： 刘兆宏 李小芊