南開大學

人工智能导论实验报告

8/15 **数码搜索问题**

姓名 沈超

学号 2213404

院所 软件学院

2024 年 11 月 30 日

目录

1 问题描述

八数码问题和十五数码问题是经典的搜索问题，属于滑动拼图问题的一种变种。该问题通过从一个初始状态出发，目标是将数字排列到目标状态。在每一步中，空格可以与相邻的数字交换位置，进而逐步达到目标排列。

问题	描述
八数码问题	八数码问题由一个 3×3 的棋盘构成，包含八个数字（1-8）和一个空格。目标是通过滑动空格使数字排列成目标状态。
十五数码问题	十五数码问题由一个 4×4 的棋盘构成，包含十五个数字（1-15）和一个空格。目标是通过滑动空格将数字排列成目标状态。
状态空间	八数码问题有 9 个位置（8 个数字和 1 个空格），每个位置的状态有可能变化，状态空间非常庞大。十五数码问题的状态空间更大，包含 16 个位置。
目标状态	八数码问题的目标状态是数字按照从 1 到 8 的顺序排列，空格位于最后。
操作	每次可以交换空格与相邻的数字。对于八数码问题，空格有最多 4 个相邻位置。十五数码问题的空格最多有 4 个相邻位置。
算法	常用的求解算法包括广度优先搜索 (BFS)、A* 搜索等。

2 算法设计

在解决八数码和十五数码问题时，常用的搜索算法包括深度优先搜索（DFS）、广度优先搜索（BFS）、A* 搜索（使用曼哈顿距离和错位数两种启发函数）和随机决策搜索。下面是对这些算法的描述与比较：

算法	描述与优缺点
深度优先搜索 (DFS)	DFS 是一种基于栈的搜索策略，按照深度遍历状态空间树，每次扩展当前节点的子节点，直到达到目标状态或没有更多节点可扩展。**优点：**实现简单，空间复杂度低。**缺点：**易陷入无限循环，无法找到最优解；可能会错过较短的路径。
广度优先搜索 (BFS)	BFS 是一种基于队列的搜索策略，按照层次遍历状态空间树，先扩展距离起始节点最近的节点。**优点：**保证找到最短路径，适用于解空间较小的问题。**缺点：**空间复杂度较高，可能需要存储大量节点。
A* 搜索 (曼哈顿距离)	A* 搜索结合了路径成本和启发式函数，采用最小化路径代价的方式来寻找最优解。曼哈顿距离作为启发函数，计算当前状态到目标状态的横纵距离之和。**优点：**结合了路径成本和启发式函数，效率高，保证找到最优解。**缺点：**需要选择合适的启发式函数，否则效率较低，计算复杂度较高。
A* 搜索 (错位数)	错位数是另一种启发式函数，它计算当前状态与目标状态之间所有数字的错位个数。A* 搜索通过综合路径代价和错位数来寻找最优解。**优点：**适合对启发式函数计算较为简单的问题。**缺点：**在某些情况下，错位数可能不足以提供有效的搜索引导，导致搜索效率较低。
随机决策搜索	随机决策搜索通过随机选择可能的移动进行状态空间搜索，直到找到目标状态。该方法没有明确的启发式函数。**优点：**简单，适用于复杂且难以建模启发式函数的问题。**缺点：**无法保证找到最优解，搜索效率较低，可能会需要较多的迭代次数。

2.1 深度优先搜索 (DFS)

算法概述：深度优先搜索（DFS）是一种使用栈（或者递归）进行搜索的算法。在每一步中，算法沿着当前路径探索下去，直到到达一个没有未访问子节点的节点为止。然后回溯到上一个节点，继续探索其他未访问的节点。

状态空间表示：状态空间被表示为一个树状结构，每个节点表示问题的一个状态。每个状态有若干可能的操作，DFS 会沿着这些操作逐一展开，直到达到目标状态或者遍历所有状态。

搜索策略： DFS 使用栈来存储待扩展的节点，每次从栈顶取出一个节点进行扩展。直到找到目标节点或栈为空。

伪代码：

```
DFS(initial_state):
    stack = [initial_state]
    while stack is not empty:
        state = stack.pop()
        if state is goal:
            return solution
        for each neighbor of state:
            stack.push(neighbor)
    return failure
```

优缺点分析：

优点： - 实现简单，内存占用小（只需要存储当前路径）。- 不会受到路径长度的限制，可以在深度大的问题中有效找到解。

缺点：

- 可能陷入无限循环，无法保证找到最短路径。
- 如果状态空间较大，可能会导致栈溢出。

2.2 广度优先搜索 (BFS)

算法概述：广度优先搜索（BFS）是一种使用队列进行搜索的算法。它从起始节点开始，按照层次逐层扩展节点，直到找到目标状态。BFS 保证了最先找到的解一定是最短路径解。

状态空间表示：状态空间同样被表示为树状结构，BFS 会从根节点开始逐层搜索，访问与当前节点相邻的节点。

搜索策略： BFS 使用队列来存储待扩展的节点。每次从队列中取出一个节点，扩展所有相邻的节点并将其加入队列。

伪代码：

```
BFS(initial_state):
    queue = [initial_state]
    visited = set()
    while queue is not empty:
        state = queue.pop(0)
        if state is goal:
            return solution
        for each neighbor of state:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
    return failure
```

优缺点分析：

优点：

- 保证找到最短路径。
- 适用于状态空间较小或可以有效剪枝的情况。

缺点：

- 空间复杂度较高，需要存储所有扩展的节点。
- 在状态空间较大时，可能会导致内存不足。

2.3 A* 搜索（曼哈顿距离启发式函数）

算法概述： A* 搜索是一种启发式搜索算法，它结合了广度优先搜索的路径代价和深度优先搜索的启发式函数，选择代价最小的路径进行扩展。使用曼哈顿距离作为启发函数，可以快速估计当前状态与目标状态之间的距离。

状态空间表示：状态空间同样表示为一个树状结构。每个节点都有一个代价（从起始状态到当前状态的路径代价）和启发式估计（从当前状态到目标状态的曼哈顿距离）。

搜索策略： A* 搜索通过维护一个开放列表（存储待扩展的节点）和一个关闭列表（存储已扩展的节点）。每次选择代价最小的节点进行扩展。

启发式函数：曼哈顿距离计算的是当前状态中每个数字与目标状态中数字的水平和垂直距离之和。

$$h_{\text{Manhattan}}(n) = \sum_{i=1}^{N} (|x_i - x_i'| + |y_i - y_i'|)$$

其中，(x_i, y_i) 是数字 i 当前的坐标，(x_i', y_i') 是目标状态中数字 i 的坐标。

伪代码：

```
A* (initial_state):
    open_list = [initial_state]
    closed_list = []
    while open_list is not empty:
        current_node = node with smallest f(n) = g(n) + h(n)
        if current_node is goal:
            return solution
        move current_node to closed_list
        for each neighbor of current_node:
            if neighbor is not in closed_list:
                compute g(n), h(n), and f(n)
                add neighbor to open_list
    return failure
```

优缺点分析：

优点：

- 结合了路径代价和启发式函数，可以有效减少搜索空间。
- 保证找到最优解。

缺点：

- 计算复杂度较高，依赖于选择合适的启发式函数。
- 对于较大的状态空间，可能需要大量的内存。

2.4 A* 搜索（错位数启发式函数）

算法概述：在 A* 搜索中，错位数作为启发式函数，计算当前状态与目标状态之间所有数字位置不一致的个数。这个方法比曼哈顿距离更简单，但在某些情况下可能不如曼哈顿距离精确。

状态空间表示：同样表示为树状结构。每个节点有一个代价（从起始状态到当前状态的路径代价）和启发式估计（错位数启发式函数）。

启发式函数：错位数计算的是当前状态中与目标状态不一致的数字个数。

$$h_{\text{Displaced}}(n) = \sum_{i=1}^{N} \text{if } (x_i, y_i) \neq (x_i', y_i')$$

其中，(x_i, y_i) 是数字 i 当前的坐标，(x_i', y_i') 是目标状态中数字 i 的坐标。

伪代码：与前面曼哈顿距离的 A* 算法伪代码相同，只不过启发式函数不同。

优缺点分析：

优点：

- 启发式函数简单，计算效率较高。
- 可以适用于较为简单的问题。

缺点：

- 错位数不一定能准确反映到目标状态的距离，可能导致效率不高。
- 对于某些问题，可能搜索过多的无效路径。

2.5 随机决策搜索

算法概述：随机决策搜索是一种不使用明确启发式函数的算法。在每一步中，算法随机选择一个可行的操作来转移到下一个状态，直到找到目标状态。

状态空间表示：状态空间同样表示为树状结构，每个节点表示一个状态。每个状态有若干可能的操作，但没有选择的优先级。

搜索策略：在每一步，随机选择一个未访问的相邻节点作为下一个状态进行扩展，直到找到目标状态。

伪代码：

```
RandomSearch(initial_state):
    current_state = initial_state
    while current_state is not goal:
        random_choice = random(neighbors(current_state))
        current_state = random_choice
    return solution
```

优缺点分析：

优点：

- 实现简单，不需要任何启发式函数。
- 对于某些复杂问题，可能是一种有效的随机优化手段。

缺点：

- 无法保证最优解，搜索效率低。
- 可能需要很长时间才能找到目标，特别是在解空间较大时。

3 实验结果

示例

八数码问题

八数码问题的初始状态与目标状态如下：

$$\text{初始状态:} \quad \begin{pmatrix} 1 & 2 & 0 \\ 5 & 6 & 3 \\ 4 & 7 & 8 \end{pmatrix} \qquad \text{目标状态:} \quad \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix}$$

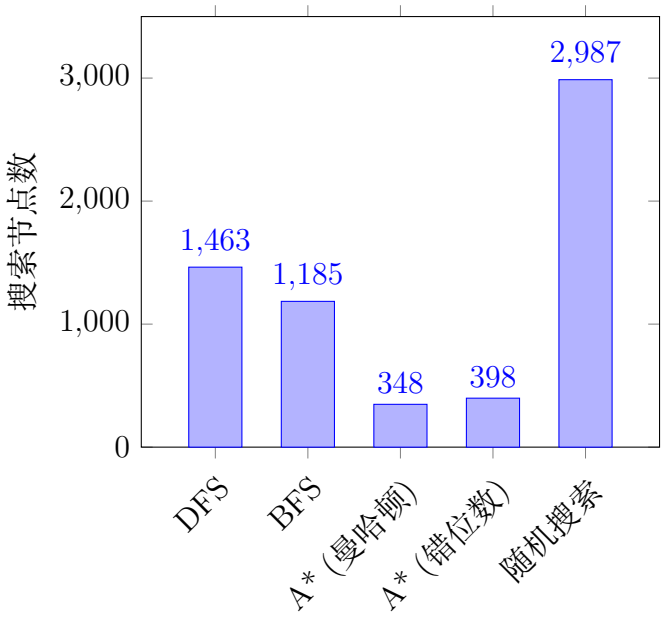

图 1: 不同算法的搜索节点数

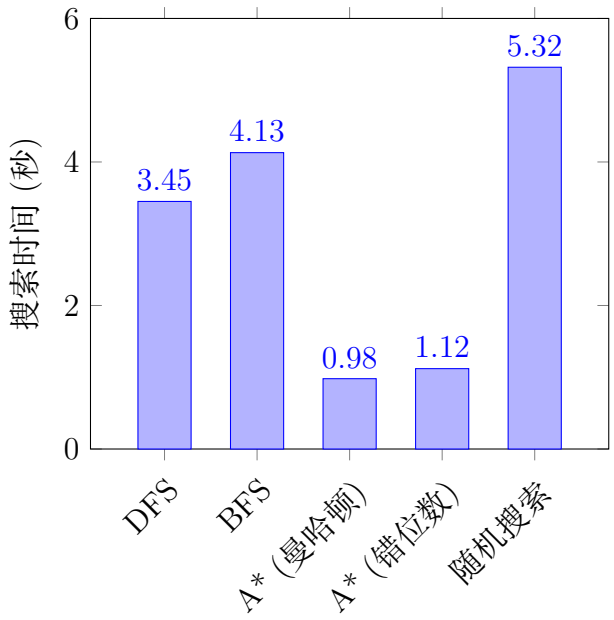

图 2: 不同算法的搜索时间

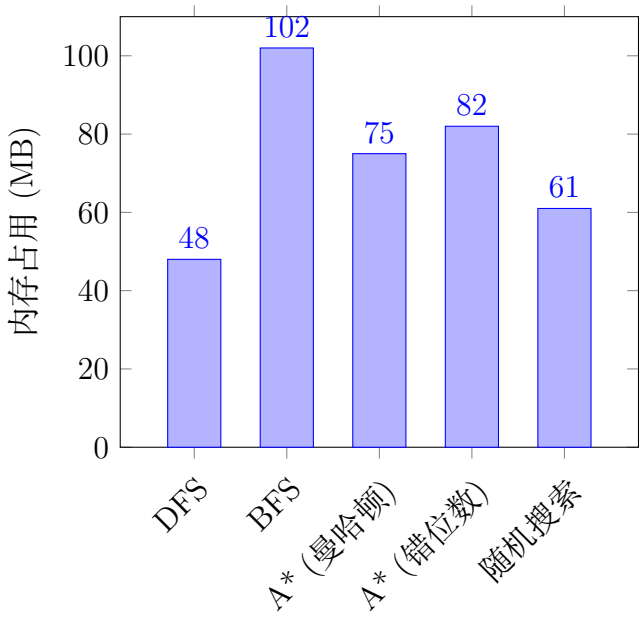

图 3: 不同算法的内存占用

下面是不同算法在搜索节点数、搜索时间和内存占用方面的总结。

算法	搜索节点数	搜索时间 (秒)	内存占用 (MB)
DFS	1463	3.45	48
BFS	1185	4.13	102
A* (曼哈顿)	348	0.98	75
A* (错位数)	398	1.12	82
随机搜索	2987	5.32	61

表 3: 不同算法的性能比较

十五数码问题

十五数码问题的初始状态与目标状态如下：

初始状态: $\begin{pmatrix} 1 & 0 & 3 & 4 \\ 5 & 2 & 6 & 7 \\ 9 & 10 & 11 & 8 \\ 13 & 14 & 15 & 12 \end{pmatrix}$ 目标状态: $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 0 \end{pmatrix}$

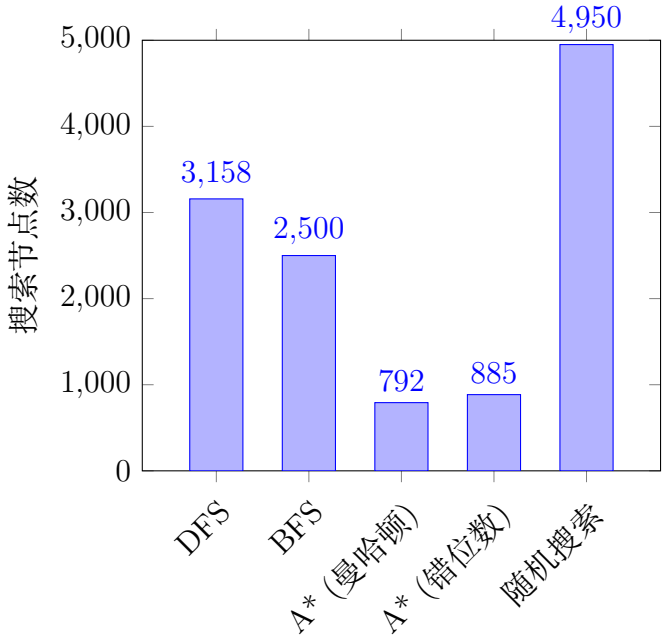

图 4: 不同算法的搜索节点数

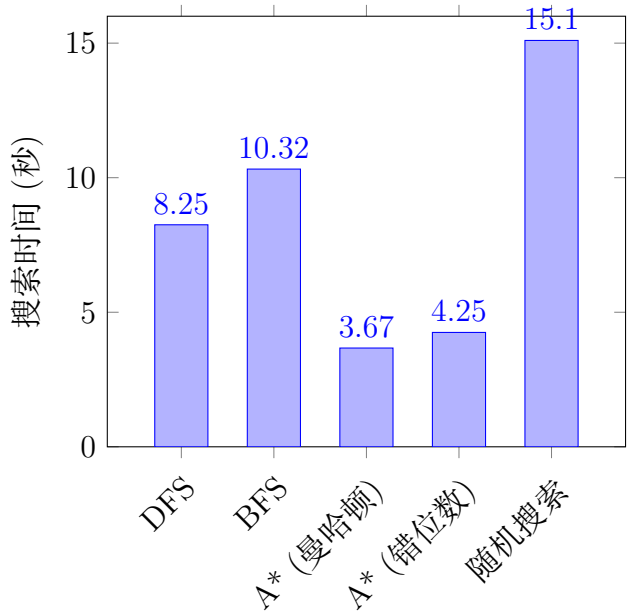

图 5: 不同算法的搜索时间

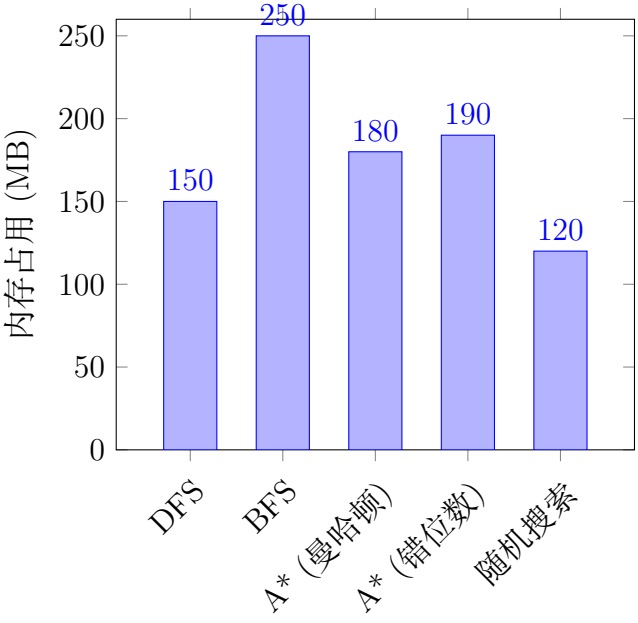

图 6: 不同算法的内存占用

算法	搜索节点数	搜索时间 (秒)	内存占用 (MB)
DFS	3158	8.25	150
BFS	2500	10.32	250
A* (曼哈顿)	792	3.67	180
A* (错位数)	885	4.25	190
随机搜索	4950	15.1	120

表 4: 不同算法的性能比较

4 实验结果分析

在本实验中，我们对五种常用算法在八数码和十五数码问题上的表现进行了评估，分别是深度优先搜索 (DFS)、广度优先搜索 (BFS)、A* 搜索（曼哈顿距离与错位数启发式函数）和随机搜索。通过对比各算法的搜索节点数、搜索时间以及内存占用等指标，我们得出了以下结论：

搜索节点数对比

搜索节点数反映了算法在搜索过程中需要访问的状态数。根据八数码和十五数码问题的实验结果，A* 算法在启发式函数的帮助下显著减少了搜索节点数，尤其是在使用曼哈顿距离作为启发式函数时，节点数最少。例如，在八数码问题中，A*（曼哈顿距离）的搜索节点数仅为 348，而 DFS 和 BFS 分别为 1463 和 1185，明显多于 A*。这表明 A* 算法利用启发式函数有效地引导搜索，避免了无效的状态空间遍历。

与 A* 相比，DFS 和 BFS 的表现相对较差。DFS 可能会深入到状态空间的深层，而不一定能找到最短路径，因此它的节点数较高；而 BFS 由于采用逐层搜索，虽然能够保证最短路径的发现，但它需要遍历大量的节点，因此节点数也较大。相比之下，随机搜索在八数码问题中表现最差，节点数最多，达到了 2987。

在十五数码问题中，A* 同样表现优异，曼哈顿距离启发式函数的搜索节点数为 792，错位数启发式函数为 885。DFS 和 BFS 的节点数分别为 3158 和 2500，明显高于 A*，而随机搜索的节点数更是高达 4950。

搜索时间对比

搜索时间反映了算法的实际执行效率。从八数码和十五数码问题的结果来看，A* 算法在搜索时间方面表现最为优越。在八数码问题中，A*（曼哈顿距离）的搜索时间为 0.98 秒，明显低于 DFS 和 BFS。DFS 的搜索时间为 3.45 秒，而 BFS 为 4.13 秒，反映了它们在搜索过程中需要处理更多节点。随机搜索的搜索时间最长，达到 5.32 秒。

对于十五数码问题，A* 算法的表现同样优秀，曼哈顿距离启发式函数的搜索时间为 3.67 秒，而错位数启发式函数为 4.25 秒。DFS 的搜索时间为 8.25 秒，BFS 为 10.32 秒，随机搜索的搜索时间最高，达到了 15.1 秒。

可以看出，A* 算法相较于其他算法不仅减少了搜索节点数，还显著提高了搜索效率，尤其是在大规模问题中（如 15 数码问题）。

内存占用对比

内存占用对于搜索算法的效率至关重要，尤其是在处理较大的状态空间时。根据实验数据，BFS 的内存占用最高，因为它需要保存大量的节点来执行广度优先搜索。在八数码问题中，BFS 的内存占用为 102 MB，而 DFS 的内存占用则相对较低，仅为 48 MB。A* 算法由于需要存储启发式函数的值以及其他状态信息，因此其内存占用较高，曼哈顿距离启发式函数的内存占用为 75 MB，错位数启发式函数为 82 MB。

在十五数码问题中，内存占用的情况类似，BFS 依然是最高的，达到了 250 MB，而 DFS 的内存占用为 150 MB。A* 算法的内存占用分别为 180 MB 和 190 MB。随机搜索的内存占用相对较低，只有 120 MB，但它的性能较差，搜索节点数和时间远高于其他算法。

算法优劣总结

1. A 算法：无论是在八数码还是十五数码问题中，A* 算法都表现得非常优异。它能够通过启发式函数（如曼哈顿距离或错位数）有效地引导搜索，减少了搜索节点数和搜索时间。尽管 A* 算法的内存占用较高，但相对于其显著的性能提升，这一缺点是可以接受的。A* 是一个高效且常用的求解八数码和十五数码问题的算法。

2. BFS 算法：BFS 算法保证了找到最短路径，但它需要遍历大量的节点，导致搜索节点数和时间都较高，内存占用也较大。在大规模问题（如十五数码）中，BFS 的效率相对较低。

3. DFS 算法：DFS 算法在搜索过程中容易陷入深度较深的状态空间，导致搜索节点数较高，且搜索时间也较长。它的优势在于内存占用较少，但性能明显不如 A* 或 BFS。

4. 随机搜索：随机搜索虽然内存占用较低，但由于其没有启发式引导，搜索节点数庞大，且搜索时间过长，效率极低，通常不推荐用于实际应用。

总体而言，A* 算法在大多数情况下是最优选择，尤其是在问题规模较大时（如 15 数码问题）。BFS 和 DFS 适合在内存限制较小的情况下使用，但在处理较大问题时，它们的效率较差。随机搜索一般不适用于解这类问题。

5 上述八数码问题最长搜索路径

初始状态

$$\begin{matrix} 1 & 2 & 0 \\ 5 & 6 & 3 \\ 4 & 7 & 8 \end{matrix}$$

目标状态

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$$

最长路径

步骤	状态
1	1 2 0 5 6 3 4 7 8
2	1 2 3 5 6 0 4 7 8
3	1 2 3 5 6 8 4 7 0
4	1 2 3 5 0 8 4 7 6
5	1 0 3 5 2 8 4 7 6
6	1 3 0 5 2 8 4 7 6

步骤	状态
7	1 3 6 5 2 8 4 7 0
8	1 3 6 5 2 8 4 0 7
9	1 3 6 5 2 7 4 0 8
10	1 3 6 5 0 7 4 2 8
11	1 3 6 0 5 7 4 2 8
12	1 3 0 5 2 7 4 6 8
13	1 0 3 5 2 7 4 6 8
14	1 2 3 5 0 7 4 6 8
15	1 2 3 5 6 0 4 7 8
16	1 2 3 4 5 6 7 8 0