《人工智能原理》项目一报告
基于A*搜索算法的“砖了个砖”游戏求解器

摘要

本项目旨在为“砖了个砖”益智游戏设计并实现一个智能求解器。该游戏可以被抽象为一个典型的状态空间搜索问题，其目标是在满足特定消除规则的条件下，找到一个最优的移动序列以清空整个棋盘。本文详细阐述了将该问题形式化建模的过程，包括状态与动作空间的定义。在算法选择上，对比了BFS与A*算法的优劣，并最终选择A*作为核心求解算法。报告的核心部分详细记录了A*算法的设计与迭代过程：从一个存在设计缺陷的贪心策略，到修正后将所有可能操作纳入搜索空间的完善版本；此外，报告还涵盖了高效的死锁检测剪枝策略，以及通过分离路径查找与动画数据生成来提升性能的数据结构设计。最终，项目成功实现了一个高效、稳健的求解器，能够针对不同优化目标提供最优解或快速贪心解，并通过可视化的方式演示求解过程。

1 引言

“砖了个砖”是一款消除类游戏，玩家需要在二维棋盘上通过滑动方块，使成对的同种方块在无障碍的路径下相遇并消除。游戏的核心挑战在于，一个看似简单的移动可能会影响后续所有方块的消除可能性，导致棋盘陷入无法解开的死锁状态。

本求解器不仅需要找到一个可行的解，还需要根据不同的优化目标——“最少移动次数”（最短路径）或“最小移动代价”——找到全局最优解。我将此问题建模为状态空间图搜索问题，并采用A*算法作为核心技术路线，通过设计高效的启发函数和剪枝策略来应对其组合爆炸的复杂性。具体代码实现可在 code/src/core/algorithm.py 中找到。

2 问题建模

我将“砖了个砖”游戏抽象为一个状态空间搜索问题。

2.1 状态空间

状态空间是问题所有可能局面（状态）的集合。

- **状态**：棋盘在任意时刻的布局。一个状态可以用一个 $N \times N$ 的二维矩阵表示，其中每个元素可以是某种类型的方块对象，也可以是空（`None`）。为了在搜索中进行高效的查找和存储，我将状态通过特定编码（如元组化）转换为不可变的哈希对象。
- **初始状态**：游戏开始时的棋盘布局。
- **目标状态**：棋盘上所有方块都被消除，即一个全空的矩阵。

2.2 动作空间

动作空间连接不同状态的，定义了从一个状态可以合法地转移到哪些新状态。根据游戏规则，我定义了两种核心动作：

- **直接消除**：如果棋盘上已存在一对同种方块，它们之间在水平或垂直方向上没有其他方块阻挡，则可以直接消除它们。此动作是零代价的。
- **移动并消除**：选择一个方块A，将其沿水平或垂直方向移动到一个空位P，如果从P点可以与另一个同种方块B直接消除，则该移动是合法的。此动作是有代价的。

3 搜索算法设计与演进

3.1 算法选择：从BFS到A*

BFS是一种完备的盲目搜索策略，能够保证找到最短路径（即每步代价为1的情况下）。我最初曾考虑使用BFS作为基线算法。然而，BFS存在两个致命缺陷：

1. 空间复杂度高：BFS需要存储所有待扩展的节点，其空间复杂度为$O(b^d)$，其中b是分支因子，d是解的深度。对于“砖了个砖”这类分支因子巨大的问题，内存消耗会迅速变得无法承受。
2. 无法处理代价问题：BFS只能处理边权重（即动作代价）为1的情况。对于本项目要求的“最小代价”求解，BFS无能为力。

因此，我放弃了BFS，转向了更合适的A*算法。A*算法通过一个评估函数 $f(n) = g(n) + h(n)$ 来指导搜索方向，其中$g(n)$是已知的实际代价，$h(n)$是对未来代价的估计，这使得它在保证最优性的前提下，能极大地削减搜索空间。

3.2 初版贪心策略的陷阱

在设计A*算法的早期版本时，我曾陷入一个逻辑陷阱。我认为，“直接消除”是零代价的，是一种免费的收益，因此应该在每次生成新状态后，贪心地扫描并消除棋盘上所有可以直接消除的方块对，直到棋盘稳定为止。

然而，实践证明这是一个错误的设计。考虑图1所示的局面：

图 1: 贪心策略导致的死锁示例

在这个例子中，棋盘上有两对水桶。如果采用贪心策略，会立刻消除第2竖直列的两个水桶。这样操作后，剩下的两个种子和两个水桶分别处于对角线位置，形成了无法解开的死锁。而正确的解法是先移动某个水桶，为后续的消除创造条件。

这个案例深刻地揭示了：**一个局部的最优选择（零代价消除）可能会导致全局的失败。**

3.3 修正策略：将零代价消除纳入搜索空间

正确的做法是将“直接消除”也视为一个与其他“移动并消除”平等的、只不过**代价为0**的动作。我将这两种动作都作为当前状态的后继节点加入到A*算法的开放列表中，让A*算法的评估函数 $f(n)$ 自行去判断哪一个分支更有希望通向最终解。

这样，即使直接消除的代价$g(n)$更低，但如果它导致了一个启发值$h(n)$急剧增大的坏局面，其综合评价值$f(n)$可能依然会高于另一个需要付出移动代价但局面更有前景的分支。这也体现了A*算法的优势，具备全局视野，能够避免贪心算法的短视。

3.4 启发函数的设计与演进

启发函数$h(n)$的质量直接决定了A*算法的效率。我的设计过程经历了一个从错误到修正，再到完善的重要迭代。

3.4.1 从“剩余对数”到“加权代价”的演进

初期，我为两种求解模式分别设计了直观的启发函数。

1. 最短序列：自然地想到，未来最少需要的移动次数应该与棋盘上还剩下多少对方块有关。因此，我设计了第一个启发函数：

$$h_1(n) = \text{棋盘上剩余方块的总数}/2 = \text{剩余方块对数}$$

2. 最小代价：基于同样的思路，我对$h_1(n)$进行了扩展：未来的总代价不仅与剩余对数有关，还与这些方块本身的移动成本有关。找出最小的滑动代价，便设计了第二个启发函数：

$$h_2(n) = (\text{剩余方块对数}) \times \min_{i \in \text{所有种类}} (\text{种类}i\text{的单位移动代价})$$

一个更精确的估计是，假设未来所有的移动都由当前棋盘上剩余方块中滑动成本最低的方块来完成。

$$h_3(n) = (\text{剩余方块对数}) \times \min_{i \in \text{所有}\textbf{剩余}\text{种类}} (\text{种类}i\text{的单位移动代价})$$

$h_3(n)$在理论上比$h_2(n)$更高效，因为它排除了那些已经被消除的方块种类，滑动代价会增加从而使启发值更大，搜索更具指导性。

3.4.2 不可接受性陷阱

上面两个启发函数的设计看似合理：

- 最短路径模式: $h_{greedy\_path}(n) = \text{剩余方块对数}$
- 最小代价模式: $h_{greedy\_cost}(n) = \text{剩余对数} \times \min(\text{场上方块滑动代价})$

我曾错误地认为它是可接受的。然而，在引入零成本的“直接消除”动作后，这个假设被打破了。

不可接受性证明：一个启发函数是可接受的，当且仅当其估计值 $h(n)$ 永远不大于真实最小成本 $h^*(n)$。考虑一个局面，其中有3对方块，但其中2对可以被直接消除（成本为0），只有1对需要移动（成本为1）。

- 最短路径模式下：
  - 启发函数估计值：$h_{greedy\_path}(n) = 3$。
  - 真实最小成本：最优解是执行2次零成本消除和1次移动消除，总移动次数 $h^*(n) = 1$。
  - 比较发现：$h_{greedy\_path}(n) > h^*(n)$，高估了成本。
- 最小代价模式下（假设最小单位代价为2）：
  - 启发函数估计值：$h_{greedy\_cost}(n) = 3 \times 2 = 6$。
  - 真实最小成本：最优解的总代价 $h^*(n) = 1 \times 2 = 2$。
  - 比较发现：$h_{greedy\_cost}(n) > h^*(n)$，同样高估了成本。

由于启发函数高估了真实成本，它不是可接受的。使用该函数的A*算法不保证能找到最优解。

3.4.3 贪心模式：将缺陷转化为特性

虽然上述函数不能用于寻找最优解，但我发现它们有一个有趣的特性：它们倾向于选择那些能最快减少棋盘上方块数量的路径，因为它直接将“方块更少”等同于“离目标更近”。这使得A*算法的行为接近于**贪心最佳优先搜索**。

因此，我没有完全抛弃这个函数，而是将其包装成一个贪心求解模式。它能以较快的速度找到一个，且通常质量不错的解，但不保证最优。这提供了一个在求解速度和解的最优性之间的选项。

```python
# --- 3b. 贪心版（非可接受的）启发函数 ---
# 以下启发函数会高估实际成本，不保证找到最优解，但通常速度很快
def _heuristic_greedy_remaining_pairs(self, board: Board) -> int:
    """[贪心/非可接受] 启发函数：返回剩余方块对的总数"""
    return board.get_num_of_tiles() // 2

def _heuristic_greedy_weighted_pairs(self, board: Board) -> int:
    """[贪心/非可接受] 启发函数：返回 剩余对数 * 场上最小单位代价"""
    all_tiles = board.get_all_tiles()
    if not all_tiles:
        return 0
    remaining_pairs = len(all_tiles) // 2
    min_cost = min(tile.cost_per_move for tile in all_tiles)
    return remaining_pairs * min_cost
```

图 2: 贪心式启发函数的实现

3.4.4 修正：两个严格可接受的新启发函数

为了实现能保证最优解的求解器，我必须设计严格可接受的启发函数。新的设计基于**被困方块**的概念。

- **定义：**一个方块是“被困的”，如果它无法与任何其他同类方块形成直接的、无障碍的消除路径。

- **新启发函数 (最短路径)：**

$$h_{optimal\_path}(n) = \lfloor \frac{\text{被困方块的总数}}{2} \rfloor = \text{被困方块数}//2$$

- **新启发函数 (最小代价)：**

$$h_{optimal\_cost}(n) = h_{optimal\_path}(n) \times \min(\text{所有被困方块的单位移动代价})$$

可接受性证明：

1. 任何一个被困方块都无法通过零成本的“直接消除”动作被移除。
2. 因此，要消除任何一个被困方块，至少需要一次成本不为0的“移动并消除”操作。
3. 一次“移动并消除”操作最多能解决掉（即消除）两个被困方块。
4. 对于最短序列模式：要消除全部 s 个被困方块，至少需要 $\lfloor (s+1)/2 \rfloor$ 次移动。真实最小移动次数 $h^*(n)$ 必然大于或等于这个值。故 $h_{optimal\_path}(n) \leq h^*(n)$ 恒成立。
5. 对于最小代价模式：消除这 s 个被困方块所需的 $\lfloor (s+1)/2 \rfloor$ 次移动，其总代价必然大于或等于 $\lfloor s/2 \rfloor$ 乘以这些被困方块中单位移动代价最小的那个值。真实最小总代价 $h^*(n)$ 必然大于或等于这个值。故 $h_{optimal\_cost}(n) \leq h^*(n)$ 恒成立。

这两个新的启发函数都是可接受的，它们能保证A*算法找到相应模式下的最优解。

```python
# --- 3a. 最优版 (可接受的) 启发函数 ---
def _heuristic_optimal_shortest_path(self, board: Board) -> int:
    """
    [最优/可接受] 启发函数：返回需要移动才能解决的方块对的最小数量
    计算方法是：统计所有“被困”的方块，然后计算需要多少次移动才能解决它们
    """
    stuck_tiles_count = self._count_stuck_tiles(board)
    # 每次移动最多解决2个被困方块，所以需要 floor(count/2) 次移动
    return stuck_tiles_count // 2

def _heuristic_optimal_min_cost(self, board: Board) -> int:
    """
    [最优/可接受] 启发函数：对于最小代价模式，一个可接受的估计是
    “被困方块”所需的最少移动次数 * 场上最小的单位移动代价
    """
    all_tiles = board.get_all_tiles()
    if not all_tiles:
        return 0

    stuck_tiles_count, min_stuck_cost = self._analyze_stuck_tiles(board)

    if stuck_tiles_count == 0:
        return 0

    min_moves_needed = (stuck_tiles_count + 1) // 2
    return min_moves_needed * int(min_stuck_cost)
```

图 3: 可接受的启发函数实现

4 性能优化：剪枝与数据结构

4.1 死锁检测与剪枝

在搜索过程中，会遇到大量注定无解的死锁状态。如果能提前识别并剪枝，将极大提升算法效率。我发现了一种非常常见且易于检测的死锁模式——“对角死锁”，如图4所示。

图 4: 典型的对角死锁模式

对于两种方块，当它们都**只剩一对**（例如A和B），且它们的坐标恰好构成一个**矩形的四个顶点**时，就形成了死锁。我为此设计了一个高效的检测函数：

1. 筛选出棋盘上所有只剩下一对的方块种类。
2. 如果这类方块少于两种，则不可能形成此死锁。
3. 遍历这些单对种类的所有组合。
4. 对于每一种组合，检查B对的坐标集合是否与A对坐标形成的“对角”坐标集合完全吻合。
5. 如果吻合，则判定为死锁。

在A*循环的开始处加入这个检测，一旦发现死锁，就立即放弃当前分支的探索。这个剪枝策略以极小的计算成本，有效地避免了在无解路径上的资源浪费。

```
# 如果找到了另一条到达此状态的更优路径，则忽略当前路径
if g_score > g_scores.get(current_hash, float("inf")):
    continue

# 剪枝策略：死锁检测。如果当前状态是注定无解的，则放弃此分支
if self._detect_corner_deadlock(current_board):
    continue
```

图 5: 剪枝的实现

4.2 高效的数据结构与实现

最开始的实现中，A*算法的搜索路径与动画演示数据共用一个变量 `path`，单个节点开销大。每次生成新状态时，都需要复制当前路径并附加新的动作，这在状态节点数量激增时，导致了巨大的内存和CPU开销，严重影响了性能。

为了应对海量的状态节点，我采用了两阶段法来分离路径查找和动画数据生成：

- **阶段一：轻量级路径查找：**在A*搜索的核心循环中，我不存储每一步详细的动画数据。取而代之的是，我使用一个`came_from`字典，只记录**子状态哈希：(父状态哈希, 导致转换的轻量级动作)**。这里的轻量级动作只包含重建所需的最少信息（如方块UID和坐标），而不是完整的对象。这使得每个节点的存储开销降到了最低。
- **阶段二：重量级路径重建：**仅在找到最终解之后，我才从目标状态开始，利用`came_from`字典一路回溯到初始状态。在这个回溯过程中，我根据“轻量级动作”和缓存的棋盘状态，构建出供前端GUI使用的、包含完整对象信息的详细动画步骤列表。

这样极大地降低了搜索过程中的内存占用和数据复制成本，是算法能够处理更复杂局面的关键。

5 结果演示与分析

5.1 棋盘生成与代价设定

为了方便测试和使用，本项目设计了灵活的棋盘生成与代价编辑模块，为后续的算法演示提供了丰富的场景。

5.1.1 棋盘生成

我们提供了随机生成和自定义两种棋盘生成方式，如图6所示。

- 随机模式：输入棋盘大小（需在4-12的范围内）和方块的总对数。程序将在棋盘的空余位置上进行随机无放回抽样来放置这些方块对。在此模式下，某一种方块图案可能出现多于一对的情况。
- 自定义模式：首先输入棋盘大小和方块的种类数。随后通过鼠标点击棋盘或输入坐标的方式，为每一种方块指定其所有位置。这种模式可以用于复现特定的难题或进行算法测试。

图 6: 棋盘生成界面：随机模式（上）与自定义模式（下）

5.1.2 滑动代价编辑

本项目提供了三种代价设定方式：

1. 默认代价：所有种类方块的单位移动代价均为1。此时，最小代价模式的结果等同于最短序列模式。
2. 随机代价：程序为每一种方块随机分配一个移动代价值（如1到100之间的整数）。
3. 自定义代价：用户可以为棋盘上的每一种方块手动输入其单位移动代价，如图7所示。

图 7: 自定义代价设定面板

5.2 算法求解

求解器有四种求解模式：最短序列、最小代价、贪心最短序列和贪心最小代价，对应前述的四种启发函数。

求解器最终能够输出一个详细的动画步骤数据包，前端UI可以据此进行可视化演示。演示过程清晰地展示了每一步的决策：是直接消除还是移动后消除，并滑动和高亮相关的方块。

以如下随机生成的棋盘为例。由于演示动画难以通过静态图片展现，以下仅展示最终结果截图。

图 8: 示例棋盘

首先是在“最短序列”模式下的求解过程，总移动序列长16。

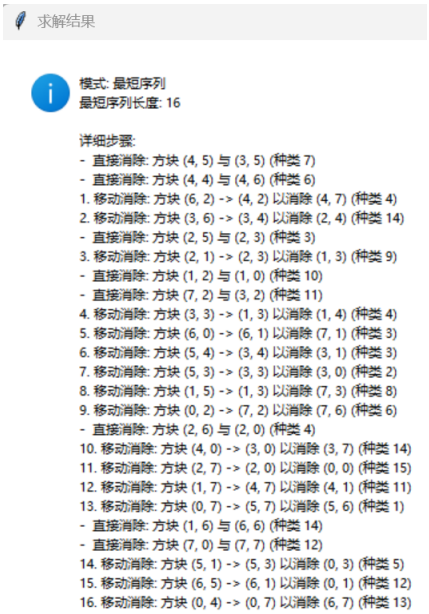

图 9: 最短路径模式求解结果演示

在同一个棋盘上，使用“贪心最短序列”模式求解，得到的移动序列长度为18，略长于最优解，但求解速度更快。

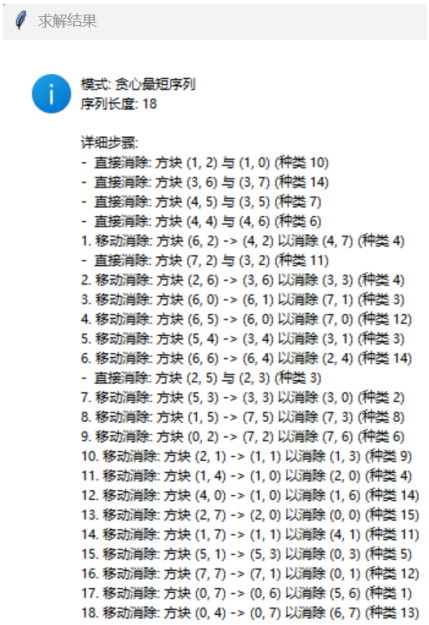

图 10: 贪心最短路径模式求解结果演示

在同一个棋盘上，为不同方块设置了随机滑动代价后，使用“最小代价”模式求解，得到的总代价为692、移动序列长度为16。使用“贪心最小代价”也得到了相同结果。这体现了在不同优化目标下的决策能力。

同时要注意，由于此处搜索难度较高，耗时较久，求解过程中可能会出现界面无响应的现象，请耐心等待。

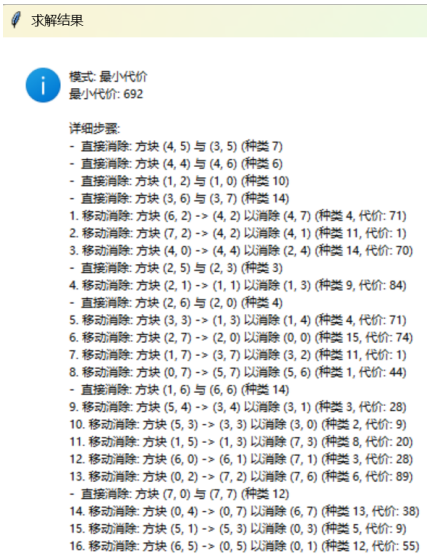

图 11: 最小代价模式求解结果演示

这些可视化结果直观地验证了算法的正确性和在不同优化目标下的灵活性。

6 讨论与展望

6.1 调试过程中的主要挑战

在整个项目开发过程中，我遇到了几个关键挑战，正是解决这些挑战的过程推动了算法的不断完善：

1. **贪心策略的误区**：如前所述，最初对零代价消除的贪心处理是最大的逻辑错误，它让我深刻理解了启发式搜索中全局最优与局部最优的差异。

2. **启发函数的可接受性陷阱**：项目中最深刻的挑战来自于对启发函数的设计。我最初基于“剩余对数”设计的启发函数，在经过严格的理论审视后，被证明是不可接受的。这个发现迫使我从理论层面重新思考并最终设计出基于“被困方块”的、严格可接受的新启发函数。这让我深刻理解了理论正确性在算法设计中的基石作用。

3. **零成本操作与启发函数的一致性**：在解决了可接受性问题后，我进一步探讨了一致性。一个启发函数是一致的，如果满足 $h(n) \leq c(n, n') + h(n')$。我发现，由于零成本的“直接消除”操作存在（即$c(n, n') = 0$），我设计的可接受启发函数也可能不是一致的。例如，消除一个“免费对”可能会解放另一个“被困对”，导致 $h(n) = 1$ 而 $h(n') = 0$，使得 $1 \leq 0 + 0$ 不成立。对这一理论细节的探讨加深了我对A*算法工作原理的理解。

4. **数据结构与性能瓶颈**：随着棋盘变大，早期的实现方案很快就因内存溢出或运行超时而失败。这促使我重新思考数据结构，最终设计出两阶段法，将搜索与数据重建分离，解决了性能问题。

5. **heapq中的对象比较**：在`heapq`中存储包含自定义对象的元组时，如果前序元素（如`f_score`）相同，它会尝试比较后序元素。由于`Tile`对象没有定义`__lt__`方法，导致了`TypeError`。我通

过在元组中加入一个全局唯一的**自增计数器**解决了这个问题。

6.2 未来展望

尽管当前版本的求解器已经非常强大，但仍有进一步优化的空间：

- **更复杂的死锁检测**：除了“对角死锁”，还存在其他更复杂的死锁模式（如三个或更多对形成的依赖环）。未来可以研究更通用的、但计算开销可控的死锁检测算法。
- **更强的启发函数**：可以探索生成更精确的启发值，从而进一步提升搜索效率。
- **探索其他算法**：对于内存极度受限的超大棋盘，可以尝试实现迭代加深A*等内存效率更高的算法。

7 结论

本项目成功将“砖了个砖”形式化为状态空间搜索问题，并基于A*算法设计和实现了一个高效的求解器。通过对算法的不断迭代优化——从修正核心逻辑错误，到设计更优的启发函数，再到引入剪枝策略和高效数据结构——我构建了一个能够在“最短路径”和“最小代价”两种模式下均能找到并可视化最优解的系统。

整个过程不仅锻炼了我将理论知识应用于实际问题的能力，也加深了我对启发式搜索算法的深度、复杂性以及工程实践中各种权衡的理解。