基于 Bitboard 的重力四子棋 AI：原理与实现

摘要

“重力四子棋”是一种三维棋类玩法，即在传统 2D 五子棋或四子棋的基础上，对每个格子允许自底向上堆叠多层棋子，从而在行、列、对角线和堆叠层等多个维度上进行博弈。为应对其较高的状态空间，本文通过使用 Bitboard 来存储棋盘状态，并结合 Zobrist Hash、Alpha-Beta + Minimax 搜索、置换表、历史启发等优化手段，成功在 5×5×5 的三维棋盘上完成深度搜索，构建了一个高效的人工智能对弈系统。实验结果表明，基于 Bitboard 和 Zobrist 的技术可显著减少复制和判定成本，大幅提升搜索效率。

关键词： 重力四子棋；Bitboard；Zobrist Hash；Alpha-Beta；Minimax；三维搜索；游戏 AI

1. 引言

传统的四子棋（Connect-Four）常见于二维 7×6 的棋盘，而在实际应用中衍生出了多种变体；“重力四子棋”则将棋盘扩展至三维：既保留 2D 的行列，又增加了垂直堆叠维度，允许在同一 `(r,c)` 位置上最多堆叠若干颗棋子。这样会产生额外的三维连线可能性。

然而，三维棋盘在搜索和复制时所需的运算量大大增加；若仍使用传统的三重 vector 等数据结构来存储棋盘，并用字符串或 MD5 做哈希，会面临大规模的复制、判断开销。为此，本文选用了 Bitboard 技术（将整个 5×5×5=125 个格子用位集表示），并采用 Zobrist Hash 进行快速散列；同时配合 Alpha-Beta 剪枝、置换表和历史启发等经典搜索算法优化，使得在有限的深度内进行“极大极小”搜索依然可行。

下文将详细介绍：

- 重力四子棋的规则；
- 核心数据结构（Bitboard、Zobrist 表）及代码逻辑；
- 搜索算法（Minimax + Alpha-Beta + Null Move + 历史启发）原理及实现。

2. 重力四子棋游戏规则

2.1 棋盘与坐标

使用 5×5 的平面，共 25 个格子；在每个 (r,c) 上最多可堆叠 5 层棋子，故整体相当于 5×5×5=125 个位置。
坐标 (r, c, l) 分别表示行、列、和在此格子上的第几层（自底往上计）。

2.2 玩家与棋子

有黑棋 'B'（AI 执行）与白棋 'W'（玩家），轮流在可用的位置落子。

2.3 落子规则

若（r,c）还没叠满（`board[r][c].size() < 5`），即可落下一颗棋子到“当前堆顶”处，层数 l 对应 `board[r][c].size()`。
不能跳层落子，也不能在已经 5 层满的格子中落子。

2.4 胜负条件

一方若在三维坐标中有 4 个连续的同色棋子（可在行、列、平面对角、或包含堆叠层的斜线方向），则立刻胜利。
棋盘所有位置都被占满且无四连则平局。

3. 代码总体逻辑

在本项目中，主程序大致分为三部分：数据结构与状态表示、搜索算法、主循环人机对战。

3.1 数据结构：Bitboard + Zobrist

Bitboard

将 125 个位置线性化为 [0..124]，用两个 uint64_t（合计 128 位）表示是否被某方占用：

```
struct BitBoard {
    uint64_t lo, hi; // 分别存下标<64 和 >=64 的位
    void setBit(int pos) { ... }
    bool testBit(int pos) const { ... }
};
```

在 Board 结构中用 BitBoard blackBB、BitBoard whiteBB 存黑棋、白棋占位；另有 topIndex[r][c] 记录此格已叠多少层（0~5）。

Zobrist Hash

为 (pos, color) 生成 64 位随机数 `ZOBRIST[pos][color]`；
在落子时，若落的是黑棋，就 `boardHash ^= ZOBRIST[pos][0]`；白棋则 XOR `[pos][1]`。
这样就快速维护了局面哈希，用以置换表中做 key。

3.2 搜索与评估

Minimax + AlphaBeta

递归深度优先搜索，maximizing 层落黑棋，minimizing 层落白棋；
引入 (alpha,beta) 剪枝，若 alpha >= beta 即可提前停止本分支。
配合 Null Move 在 maximizing 层可跳过一手测试，对极端优势情况做剪枝。
评估

若某方已出现 4 连，评估值为极大(99999999) 或极小(-99999999)；
否则对所有 4 连线统计只含同色棋子的潜力评分，如 "BBB-"->5000 等，以黑-白得到最终分。

置换表 + 历史启发

以 boardHash 为 key，在 `unordered_map<uint64_t, TTEntry>` 中存储 `(depth, value, boundType, bestMove)`，下次重复局面可跳过搜索；
对曾导致剪枝的 `(r,c)` 累加 `depth*depth` 到 `historyHeuristic[r][c]`，下次 move generation 时先搜索这些历史分高的走法，往往可更快触发剪枝。

3.3 主循环

初始化空棋盘 Board board，AI 设为黑棋先行。
若轮到玩家，则读入 `(r, c)` 并落子；若轮到 AI，则：
先检查单步必杀或防守；
否则迭代加深搜索 (searchBestMove) 并选出最优走法。
每步后检查 checkWinner 或 isBoardFull，若结束则输出结果并退出。

4. Alpha-Beta 剪枝与 Minimax 详细说明

Minimax 是博弈树搜索的基础，将“黑方 AI 最大化评估”和“白方玩家最小化评估”交替进行；若没有任何剪枝，其时间复杂度可能随深度呈指数级增长。为提升效率，引入 Alpha-Beta 剪枝：

alpha： 当前已知对 maximizing 节点的最优下界；
beta： 当前已知对 minimizing 节点的最优上界。
如果在搜索子节点过程中满足 alpha >= beta，说明对手已经有更优手段或自己已经足以满足最优，不必搜索其余兄弟节点。
这样可在不改变最终结果前提下跳过大量无用分支，通常能将搜索树大小大幅减少（理想情况达到原本的平方根级别）。

在本项目中：

```
pair<double, Move> minimax(Board bd, int depth, double alpha, double beta, bool maximizing) {
    if (depth == 0 || isTerminal(bd))
        return { evaluate(bd), {-1,-1} };
    if (maximizing){
        double bestVal=-∞;
        for(auto move: generateMoves(bd)) {
            val = minimax( apply(bd,move,AI), depth-1, alpha, beta, false );
            bestVal = max(bestVal, val);
            alpha = max(alpha, bestVal);
            if (alpha>=beta) break; // 剪枝
        }
        return { bestVal, bestMove };
    } else {
        double bestVal = +∞;
        for(auto move: generateMoves(bd)) {
            val= minimax(apply(bd,move,PLAYER), depth - 1, alpha, beta, true );
            bestVal = min(bestVal, val);
            beta = min(beta, bestVal);
            if(alpha >= beta) break; // 剪枝
        }
        return {bestVal, bestMove};
    }
}
```

此外还使用了 Null Move（在 maximizing 时跳过一手）和历史启发（对能剪枝的走法加分排序）等进一步降低搜索量。