AI 五子棋项目报告

何子谦

2025-11-19

目录

1 需求分析

本项目实现一个基于 Qt6 与 C++17 的 AI 五子棋游戏。五子棋的基本规则如下：双方轮流在 15x15 棋盘上落子，先将五子连成一线者获胜。经分析，系统需满足以下需求：

- 支持人机对战（可选先手/后手）
- 支持点击棋盘落子与对局重置
- 支持胜负判定与提示
- UI 响应性良好，避免 AI 计算阻塞主线程
- UI 友好，操作直观
- AI 计算应当能在合理时间内给出决策（深度不低于 6）
- 代码库结构清晰，易于维护与扩展
- 跨平台（macOS/Linux/Windows）

注：悔棋、AI 自对弈、禁手规则、通过开局时交换行棋权来平衡先手优势等非关键功能不在需求范围内。

2 总体设计

2.1 架构概述

本项目整体上采用 MVC（Model-View-Controller）架构设计，以此实现 UI 与业务逻辑的解耦，提升代码的可维护性与可扩展性。View 层负责显示界面、处理交互，用 Qt Widgets 实现；Model 层负责棋盘状态管理与 AI 逻辑，Controller 层通过信号/槽机制实现 View 与 Model 层的交互；此外，AI 计算与游戏管理逻辑运行在独立线程中（使用 `QThread` 及 `QtConcurrent` 库），确保 UI 主线程的流畅响应。

2.2 模块划分

2.2.1 文件结构

```
.
├─ Models/
│  ├─ GameManager.{h,cpp}          # Controller，物理上位于 Models/ 目录
│  ├─ BoardManager.{h,cpp}         # 棋盘状态/判断胜负/候选缓存
│  ├─ GomokuAI.{h,cpp}             # AI 落子相关逻辑
```

```
│   └─ Constants.h                    # 常量
├─ UI/
│   ├─ MainWindow.{h,cpp}             # 主窗口
│   ├─ GameWidget.{h,cpp}             # 整合 UI、连接 UI 与 GameManager
│   ├─ BoardWidget.{h,cpp}            # 棋盘绘制/点击事件
│   └─ ColorChooserWidget.{h,cpp}     # 先手/后手选择
├─ Tests/                             # 性能测试
│   ├─ GomokuAIParallelizationTests.cpp
│   └─ GomokuAIOverHeadTests.cpp
├─ docs/                              # 与 AI 的讨论记录及总结
│   ├─ STRUCTURE.md
│   ├─ QT_EVENT_LOOP.md
│   ├─ PARALLELIZATION_ANALYSIS.md
│   └─ PERFORMANCE_OPTIMIZATIONS.md
└─ main.cpp / CMakeLists.txt / Makefile
```

注：`GameManager` 职责上属于 MVC 的 Controller，由于本项目中 Controller 仅有一个，因此当前置于 `Models/` 目录管理；`docs/` 目录下存放与 AI 的讨论记录及总结。

3 详细设计

3.1 搜索算法

3.1.1 minimax 搜索

minimax 搜索是一种适用于双人对弈游戏的算法，其中一方试图最大化其得分，而另一方试图最小化其得分。通过交替调用最大化和最小化函数，算法假设对手会选择最佳走法来有效地搜索自身的最佳走法，并返回得分最高（或最低）的走法。

- 入口：`GomokuAI::getBestMove()`。空棋盘直接走中心；否则复制 `BoardManager` 并调用 `minimaxAlphaBetaRootParallel()`（根节点并行化的 minimax + alpha-beta 剪枝）。
- 主要递归体：`GomokuAI::minimaxAlphaBeta(BoardManager&, depth, isMaximizing, alpha, beta)`。
  - 终止条件：检测胜负（`BoardManager::checkWinner()`）或 `depth == 0`。
    - 胜利/失败返回有符号整数极大/极小值（加入微小偏置，防止双方均将获胜时选择

防守）。
    * 否则用 `evaluate(board, _color)` 始终从 AI 视角打分。
  - 递归展开：
    * 生成候选走子：`candidateMoves(boardManager)`（见候选位置缓存、查找及排序）。
    * 交替 `isMaximizing` 状态，递归调用 `minimaxAlphaBeta()`。
    * 根据 `isMaximizing` 选择最大/最小分并更新 `bestEval` 和 `bestMove`。

3.1.2 Alpha-Beta 剪枝优化

Alpha-Beta 剪枝是一种用于优化 minimax 搜索的技巧，通过在搜索过程中维护两个边界值（α 和 β），有效剪去不可能影响最终决策的分支。举例来说，对于一个最大化节点，如果已经找到一个分数高于 β 的走法（β 记录了父节点能达到的最小值），则无需继续探索该节点的其他子节点，因为最小化的父节点不会选择这个路径。

- 标准 α-β 剪枝：
  - 最大层：`alpha = max(alpha, eval)`；当 `beta <= alpha` 时发生 β cut-off。
  - 最小层：`beta = min(beta, eval)`；当 `beta <= alpha` 时发生 α cut-off。

注：根节点并行时仍可保留“跨块”剪枝：使用 `globalAlpha` 将前一块的最佳值作为下一块的 α 下界，减少后续块的分支探索（见 minimax 搜索根节点并行化）。

3.1.3 评估函数

- 主体：`evaluate(const BoardManager&, const char player)` → `evaluateSequences()`（基础分）+ 若干特殊模式加成。
- 片段扫描：对每个棋盘位置，在四个方向（水平/垂直/两斜）检查累计的连珠（同一颜色连续棋子）长度，得到：
  - `length`（连续棋子数），`openSides`（两端空位数 $\in\{0,1,2\}$）。
  - 基础分由 `sequenceScore(length, openSides)` 给出：
    * 5 连：1000000（胜利）
    * 4 连：双活 50000，单活 10000，其余 300
    * 3 连：双活 2000，单活 400，其余 50
    * 2 连：双活 200，单活 60，其余 10
    * 1 连：双活 20，单活 5，其余 1
- 威胁计数：累计 open/half-open 的“三/四”形成的 `SequenceSummary`（己方与对方各

一份)。

- 特殊模式加成：根据双方的 `SequenceSummary`，对分数进行进行如下调整：
  - 若任一方出现“活四”，直接大额加分/减分。
  - 对“双活三”给予显著奖励/惩罚。
- Center Control：`centerControlBias()` 统计己/对方棋子距中心的曼哈顿距离并加权，用于鼓励中前期时靠近棋盘中心落子。

最终得分 = 我方基础分 - 对方基础分 + 特殊模式得分分差 + Center Control 加权得分分差。

注：在本项目实现中，无论为 AI 落子还是人类落子，均调用 `evaluate(board, _color)`（`_color` 存储 AI 颜色），也即评估函数始终从 AI 视角打分。因此，实际上**本项目中不需要对层数的奇偶性进行区分**。

3.1.4 候选位置缓存、查找及排序

- 候选来源：`BoardManager` 维护 `candidateMovesCache` 与 `candidateMap`。
  - 缓存使用 `(std::unordered_set<BoardPosition>)` 实现快速插入/删除
  - `candidateMap` 使用 `bool[BOARD_SIZE][BOARD_SIZE]` 实现 O(1) 级查询位置可用性。
- 缓存更新策略：
  - 每次落子后，将该点周围 `[row±R, col±R] (R = SEARCH_RADIUS)` 范围内的空位加入候选；撤销时用 `reverseCandidatesCache()` 回滚操作。（具体回滚逻辑见落子与撤销落子）
  - 性能优化：
    * 维护 `candidateMap`（布尔阵列）以实现 O(1) 级别的存在性检查，避免在缓存中进行哈希查找的开销。
    * 为 `BoardPosition` 实现高效率 `hash()` 函数（`(row << 4) | col`），提升哈希表性能。
- `GomokuAI` 中进行候选排序（`GomokuAI::candidateMoves`）。**候选排序使 alpha-beta 剪枝发生率大幅提高**，具体策略如下：
  - 若存在“使任意一方立即取胜”的走法，直接返回单一候选（跳过其余评估）。
  - 其次收集“威胁点”（连三/连四且前/后不被阻断）
  - 最后按中心距离对一般点排序。
- 使用 `candidateMovesCache.reserve(64)` 于初始化时预分配空间，减少动态扩容开销。

3.1.5 落子与撤销落子

本项目区分“真实落子”（由 Controller 执行，影响 UI）与“搜索模拟落子”（由 AI 在内存副本上执行，不影响 UI）。真实落子将会修改棋盘在 UI 侧的副本，并调用后端方法修改后端棋盘状态，而搜索模拟落子则仅在 AI 的内存副本上进行，不影响 UI。二者均使用 `BoardManager` 的 `makeMove(BoardPosition)` 方法实现落子。

`BoardManager` 的关键实现：

- `movesHistory`：记录落子历史的栈（`std::vector<MoveRecord>`），用于支持 `undoMove()`。其中 `MoveRecord` 结构体包含落子位置及候选缓存变更信息 `CandidatesDelta`。
- `makeMove(BoardPosition)`：
  - 内部 `_makeMove()` 将棋子落于 `board[row][col]`，切换 `_blackTurn`
  - 更新 `movesHistory`；
  - 同步调用 `updateCandidatesCache(pos)` 维护候选缓存；
  - 返回 `checkWinner()` 的结果（`EMPTY/BLACK/WHITE`）。
- `undoMove()`：
  - 弹出 `movesHistory` 栈顶元素
  - 将该位置改为 `EMPTY`
  - 调用 `reverseCandidatesCache(delta, position)` 精确回滚候选缓存；
  - 最后切换 `_blackTurn`，回到之前一手的行棋方。
- 候选缓存维护函数：
  - `updateCandidatesCache(pos)` 返回 `CandidatesDelta`，并更新缓存；
  - `reverseCandidatesCache(delta, moveUndone)` 利用 `CandidatesDelta` 回滚缓存状态。

3.2 并行与线程设计

3.2.1 UI 与 Controller/Model 置于不同线程

- `GameManager` 继承于 `QObject`，使其能使用 Qt 信号/槽机制。
- 在 `GameWidget` 初始化时调用 `gameManager→moveToThread(gameThread)` 将 `GameManager` 移至独立 `QThread`。
- `GameManager` 中管理 `GomokuAI` 及 `BoardManager` 实例，后端计算与 UI 线程完全独立。
- `GameManager` 通过信号/槽与 UI 通信：

  - 因 GameManager 位于独立线程，Qt 信号连接自动采用 Qt::QueuedConnection，信号被排队处理，避免阻塞 UI。
  - GameWidget::handleHumanMove(BoardPosition) 信号请求后端处理人类落子。
  - GameManager::handleHumanMove(BoardPosition) 槽处理落子请求，并发出 GameManager::moveApplied(MoveResult) 信号告知 UI 更新；最后启动 AI 计算。
  - AI 计算完成后，发出 GameManager::moveApplied(MoveResult) 通知 UI 更新。
- BoardWidget 持有棋盘“快照”，在收到 moveApplied 信号后增量更新快照并触发 UI 重绘。

3.2.2 协作式 AI 计算任务取消

- 取消入口：Reset Game 按钮中，调用 gameThread→requestInterruption()；随后断开旧连接，摧毁线程并重新初始化 GameManager/gameThread，建立新信号连接。
- AI 协作：在 GomokuAI::getBestMove() 与 GomokuAI::minimaxAlphaBeta() 计算开始前检查 QThread::currentThread()→isInterruptionRequested()，遇到取消及时返回，避免资源泄漏。
- 该模式下，任务取消依赖于 QThread 的协作式中断机制，需要任务管理者与任务本身合作完成任务取消过程。

3.2.3 minimax 搜索根节点并行化

- 考虑到如果对每层的所有分支进行并行化，会引入大量线程调度与同步开销，且使共享的 α/β 变量维护复杂，弊大于利。因此，本项目仅在“根层”实现并行。
- 根层并行颗粒度选择：若对所有候选位置同时并行，线程调度开销过大且无法维护 α/β 变量进行剪枝；因此选择根据可用线程数量对候选进行分块并行处理。
- 实现：GomokuAI::minimaxAlphaBetaRootParallel()
  - 将根层候选按 threadCount 划分为若干块。
  - 使用 QtConcurrent::blockingMapped(&threadPool, chunk, ...) 并行计算每块内各走法的评分。
  - 在 blockingMapped 传入的闭包内部，复制 BoardManager 保证线程安全，再调用 minimaxAlphaBeta() 搜索。
  - 维护 globalAlpha：每处理完一块，以该块的最佳值更新 globalAlpha，作为后续块的 α 下界，实现“跨块剪枝”。
  - 块内无互相通信，免去块内 α/β 变量的同步开销，保证线程安全。

- 效果：根层分块并行将同步成本摊薄到“每块一次”，同时仍保留根节点一定程度的剪枝效率，并且保证线程安全。这一优化在中后期候选数量较多时带来显著加速。

3.3 关于对 `evaluate()` 并行化的尝试

项目早期曾尝试将评估函数 `evaluate()` 的内部实现并行化。然而基于分析与实测，最终选择回退为顺序实现，并选择实现根层并行化的 minimax。

最初设想的 `evaluate()` 并行方案为使用 `QtConcurrent::blockingMappedReduced`，将 225 个格点划为 4–5 个块，并发统计各块的分数，最后合并结果。在实测中发现该方案使性能劣化严重（10x 左右），原因分析如下：

- 主要原因：同步/调度开销远大于计算本身。线程任务派发/线程唤醒所占用时间以及块内线程同步的时间远超单次评估的计算时间。且评估函数调用频率极高，使得该开销被无限放大。
- 缓存局部性：顺序按行扫描具备更好的缓存局部性；细粒度并行导致频繁切换线程上下文，破坏缓存局部性，降低性能。
- 实测对比（深度 5，示例环境）：
  - 并行评估：平均单步计算总用时 $\approx$ 2–3s；
  - 顺序评估：平均单步计算总用时 $\approx$ 0.3s；
  - 经过 Profiler 分析，并行实现中 70–80% 时间耗费在线程调度与同步上（`QThreadPool::start/QWaitCondition` 等）。

因此，最终本项目选择顺序实现评估函数，保留良好局部性与可维护性，转而在根节点实现 minimax 并行化，从而获得更显著的性能提升。可以看出，**对“高频、细粒度”的评估函数并行化弊大于利；并行应当落在“低频、粗粒度”的根分支层，才能获得可观性能优化**。

3.4 性能优化重点总结

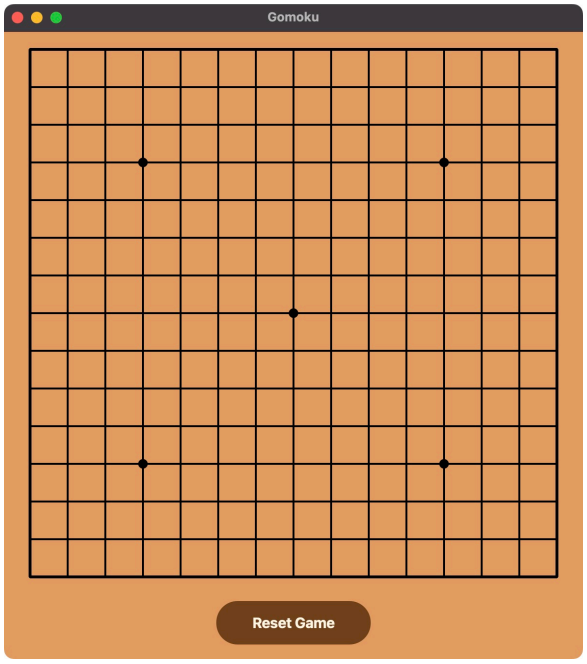
- Alpha-Beta 剪枝
- `candidateMoves` 中进行候选排序，提高剪枝发生率。（2-3x 加速）
- 根节点并行、线程池（`QThreadPool`），线程数上限 `min(idealThreadCount, 12)`。（2x 加速）
- 跨块 Alpha 维护全局 `globalAlpha`，实现跨 chunk 剪枝。
- 缓存候选位置，避免每次生成候选时遍历全盘。

最终，在示例环境下，**AI 计算 7 层时在前期空盘时平均决策时间为 3 秒；中后期局面下，平均决策时间控制在半分钟以内**。

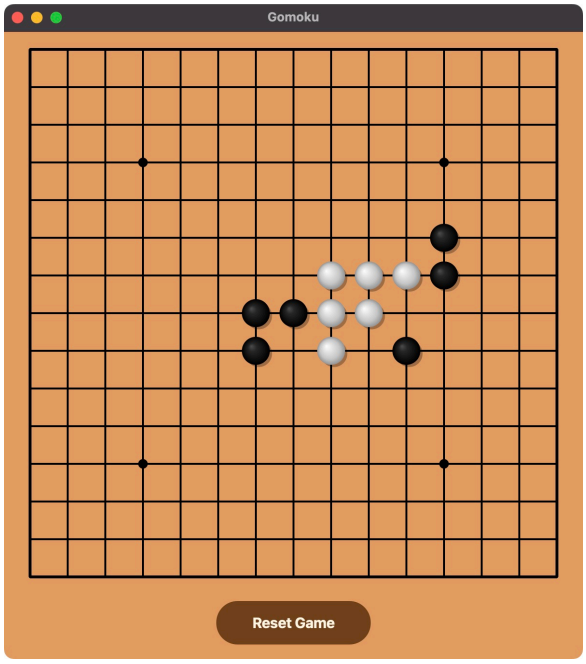
注: 测试环境为 Apple M1 Pro, 8 核 CPU (6P + 2E), 16GB 内存, macOS Sequoia 15.7.2。

4 测试结果

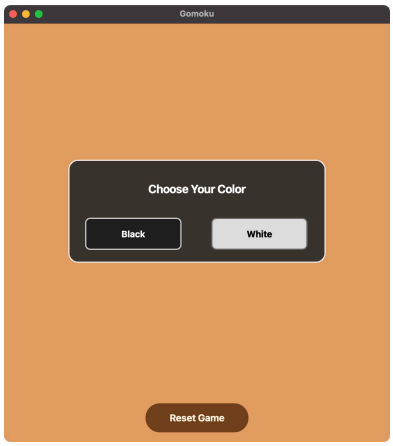
4.1 运行截图



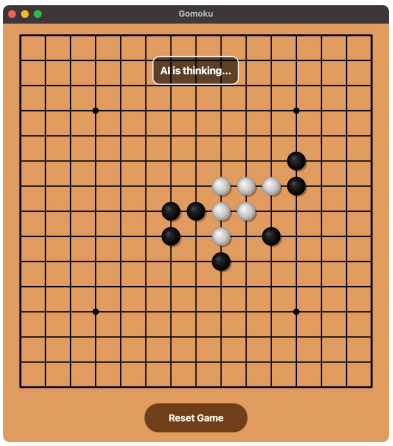
主界面



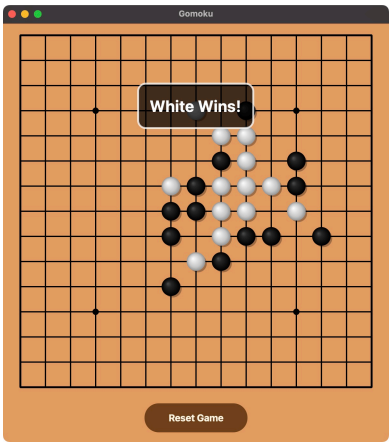
游戏中画面



选择先后手



AI 计算中提示



游戏结束提示

4.2 性能测试

测试环境：Apple M1 Pro，8 核 CPU (6P + 2E)，16GB 内存，macOS Sequoia 15.7.2。

- `Tests/GomokuAIParallelizationTests.cpp`：多场景 best move 与耗时统计、异步开销与单元评估基准。
- `Tests/GomokuAIOverHeadTests.cpp`：研究 `std::async` 开销与对单个位置评估的耗时。

4.2.1 多场景决策时间统计

为验证并行化优化的有效性，本项目在四种典型对局场景下对比了顺序执行（Sequential）与根节点并行化（Parallel）两种实现方式在不同搜索深度下的性能表现。测试场景包括：

1. **First Move after Center**：棋盘仅有一子
2. **Early Opening Pressure**：棋盘有 6 子
3. **Midgame Crossfire**：棋盘有 9 子
4. **Late-Game Threat Net**：棋盘有 12 子

该测试使用 `Tests/GomokuAIParallelizationTests.cpp` 得出时间统计。

4.2.1.1 深度 5 性能对比

场景	Sequential (ms)	Parallel (ms)	加速比
First Move after Center	177.87	66.13	2.69x
Early Opening Pressure	240.01	75.46	3.18x
Midgame Crossfire	14.18	16.45	0.86x
Late-Game Threat Net	344.77	123.69	2.79x
平均	**194.21**	**70.43**	**2.76x**

4.2.1.2 深度 7 性能对比

场景	Sequential (ms)	Parallel (ms)	加速比
First Move after Center	5127.27	2595.73	1.98x
Early Opening Pressure	10615.51	3424.44	3.10x
Midgame Crossfire	532.15	572.93	0.93x
Late-Game Threat Net	17264.11	6882.82	2.51x
平均	**8384.76**	**3368.98**	**2.49x**

在深度 5 和深度 7 下，并行化实现分别实现了平均 2.76x 和 2.49x 的加速比，可以看出**根节点并行化在大多数场景下能带来至少 2 倍的性能提升**。但在某些特定中局场景（如 Midgame Crossfire）下，由于候选走法中包含能使某方立即胜利的走法（见候选位置缓存、查找及排序），线程调度开销反而导致并行化性能略微劣化。

4.2.2 `std::async` 线程开销测试

为了理解为什么在评估函数中使用并行会导致性能劣化（见关于对 `evaluate()` 并行化的尝试），本项目测量了 `std::async` 创建和销毁线程的开销，以及单个位置评估的计算时间。

通过 `Tests/GomokuAIOverHeadTests.cpp` 中的测试代码，得到以下结果：

- **平均 async 任务创建 + 销毁开销**：21.86 µs
- **平均单个位置评估耗时**：0.0014 µs

测试结果清晰地展示了为何细粒度并行化会导致性能劣化：线程创建/销毁的开销（21.86 µs）是单个位置评估耗时（0.0014 µs）的 **15400 倍**。假设将整个棋盘（225 个位置）分为 4 个线程处理，每个线程处理约 60 个位置，则单线程计算总耗时为 `225 × 0.0014 µs = 0.315 µs`，**这仍然远小于单个线程创建与合并的开销**。即使后续引入线程池减少线程优化开销，仍然无法弥补如此巨大的计算与任调度/线程同步开销的差距。

由此可以得出结论：**并行化的粒度选择至关重要**。只有当单个任务的计算时间远大于线程开销时（如本项目中的根节点分支搜索），并行化才能带来净收益。对于高频、细粒度的函数（如评估函数），顺序执行反而更高效。

4.3 棋力评估

本项目 AI 基于 minimax 搜索 + alpha-beta 剪枝实现，在深度 7 的搜索下表现出良好的棋力。AI 能够准确识别对手的连三、连四等威胁模式，并在 `candidateMoves()` 中优先考虑胜负手和威胁点，确保不漏关键防守；同时，评估函数综合考虑了多种棋型模式及 Center Control，能够在中后期复杂局面下做出合理判断。在多次与人类玩家对局中，**AI 展现出较有攻击性的策略**，善于在被威胁时通过制造同等威胁来分散对手注意力，从而寻找突破口。目前没有发现 AI 出现业余人士能看出明显失误的情况。但在较复杂的局面下，由于搜索深度的限制以及评估函数的不准确性，AI 仍有可能出现判断失误导致被绝杀。

本项目 AI 在深度 7 下与三个线上五子棋 AI 进行了对战测试，无论先手还是后手均能取胜。由于使用的三个线上 AI 及本项目 AI 算法实现均具有确定性，因此多次测试结果一致，无法给出胜率统计数据。但从对战结果来看，本项目 AI 在当前实现下已达到较高水平。

在与人类玩家的对局中，本项目 AI 有败绩，但整体表现较佳。很遗憾，未能系统记录人类对局数据，因此无法给出具体胜率统计。

注：使用的线上 AI 包括：

- DKM - Gomoku Online

- Gomoku.com - Challenge AI Opponents

本项目 AI 在深度 7 下与三个线上五子棋 AI 进行了对战测试，无论先手还是后手均能取胜。由于使用的三个线上 AI 及本项目 AI 算法实现均具有确定性，因此多次测试结果一致，无法给出胜率统计数据。但从对战结果来看，本项目 AI 在当前实现下已达到较高水平。

在与人类玩家的对局中，本项目 AI 有败绩，但整体表现较佳。很遗憾，未能系统记录人类对局数据，因此无法给出具体胜率统计。

注：使用的线上 AI 包括：

- DKM - Gomoku Online
- Gomoku.com - Challenge AI Opponents
- YJYao Gomoku

4.4 资源占用

- CPU 占用：在示例测试环境中（6P + 2E），AI 计算时可见多个核心被充分利用，CPU 总使用率最高可达 80%。
- 内存占用：稳定在 40 MB 以内。

5 实现

5.1 搜索算法摘录

```
// 空盘走中心，根层并行搜索
BoardPosition GomokuAI::getBestMove(
  const BoardManager &boardManager
) const {
  if (QThread::currentThread()->isInterruptionRequested()) {
    return {-1, -1};
  }
  if (boardManager.isBoardEmpty()) {
    return {BOARD_SIZE / 2, BOARD_SIZE / 2};
  }

  BoardManager simulatedBoard = boardManager;
```

```
  return minimaxAlphaBetaRootParallel(simulatedBoard, _maxDepth);
}

// 候选排序（立即胜/阻断威胁/中心优先）
std::vector<BoardPosition> GomokuAI::candidateMoves(
  const BoardManager& boardManager
) const {
  std::vector<BoardPosition> threatMoves;
  std::vector<BoardPosition> moves;

  for (const auto& pos : boardManager.getCandidateMoves()) {
    if (wouldWin(boardManager, pos, _color) ||
      wouldWin(boardManager, pos, getOpponent(_color))) {
      return {pos};
    } else if (posesThreat(boardManager, pos, _color) ||
        posesThreat(boardManager, pos, getOpponent(_color))) {
      threatMoves.push_back(pos);
    } else {
      moves.push_back(pos);
    }
  }

  std::sort(
    moves.begin(),
    moves.end(),
    [&](const BoardPosition& a, const BoardPosition& b) {
      return boardManager.centerManhattanDistance[a.row][a.col] <
        boardManager.centerManhattanDistance[b.row][b.col];
    }
  );

  threatMoves.insert(threatMoves.end(), moves.begin(), moves.end());
  return threatMoves;
}
```

```cpp
// 评估函数（始终以 AI 视角评分）
int GomokuAI::evaluate(
  const BoardManager &boardManager,
  const char player
) const {
  const char opponent = getOpponent(player);
  // pSum: playerSummary, oSum: opponentSummary
  const auto [pSum, oSum] = evaluateSequences(boardManager);

  if (pSum.openFours > 0) {
    return 400000 + pSum.openFours * 2000;
  }
  if (oSum.openFours > 0) {
    return -400000 - oSum.openFours * 2000;
  }

  int score = pSum.score - oSum.score;

  const int openThreeBonus = 15000;
  score += openThreeBonus * (pSum.openThrees - oSum.openThrees);

  const int doubleThreeBonus = 60000;
  if (pSum.openThrees >= 2) score += doubleThreeBonus;
  if (oSum.openThrees >= 2) score -= doubleThreeBonus;

  const int semiOpenThreeBonus = 4000;
  score += semiOpenThreeBonus * (pSum.semiOpenThrees - oSum.semiOpenThrees);

  const int semiOpenFourBonus = 20000;
  score += semiOpenFourBonus * (pSum.semiOpenFours - oSum.semiOpenFours);

  const int centerWeight = 2;
  const int centerScore = centerControlBias(boardManager, player)
    - centerControlBias(boardManager, opponent);
```

```
    score += centerWeight * centerScore;
    return score;
}

// minimax + α-β 剪枝（主递归体）
std::pair<int, BoardPosition> GomokuAI::minimaxAlphaBeta(
    BoardManager& boardManager,
    int depth,
    bool isMaximizing,
    int alpha,
    int beta
) const {
    if (QThread::currentThread()->isInterruptionRequested()) {
        return {0, {-1, -1}};
    }
    char winner = boardManager.checkWinner();
    if (depth == 0 || winner != EMPTY) {
        if (winner == _color) {
            return {std::numeric_limits<int>::max()/2 + 10000, {}};
        }
        if (winner == getOpponent(_color)) {
            return {std::numeric_limits<int>::min()/2 - 10000, {}};
        }
        return {evaluate(boardManager, _color), {}};
    }

    BoardPosition bestMove;
    auto moves = candidateMoves(boardManager);
    if (isMaximizing) {
        int maxEval = std::numeric_limits<int>::min();
        for (const auto& pos : moves) {
            boardManager.makeMove(pos);
            auto [eval, _] = minimaxAlphaBeta(boardManager, depth - 1,
                                              false, alpha, beta);
```

```
      boardManager.undoMove();
      if (eval > maxEval) { maxEval = eval; bestMove = pos; }
      alpha = std::max(alpha, eval);
      if (beta <= alpha) break;
    }
    return {maxEval, bestMove};
  } else {
    int minEval = std::numeric_limits<int>::max();
    for (const auto& pos : moves) {
      boardManager.makeMove(pos);
      auto [eval, _] = minimaxAlphaBeta(boardManager, depth - 1,
                                        true, alpha, beta);
      boardManager.undoMove();
      if (eval < minEval) { minEval = eval; bestMove = pos; }
      beta = std::min(beta, eval);
      if (beta <= alpha) break;
    }
    return {minEval, bestMove};
  }
}

// 根节点分块并行（QtConcurrent + globalAlpha）
BoardPosition GomokuAI::minimaxAlphaBetaRootParallel(
  BoardManager& boardManager, int depth
) const {
  if (QThread::currentThread()->isInterruptionRequested()) {
    return {-1, -1};
  }

  BoardPosition bestMove;
  auto moves = candidateMoves(boardManager);
       globalAlpha](const BoardPosition& pos) {
        BoardManager simulatedBoard = boardManager;
        simulatedBoard.makeMove(pos);
```

```
        auto [eval, _] = minimaxAlphaBeta(
          simulatedBoard, depth - 1, false, globalAlpha,
          std::numeric_limits<int>::max()
        );
        return std::make_pair(eval, pos);
      }
    );

    int chunkMaxEval = std::numeric_limits<int>::min();
    BoardPosition chunkBestMove;
    for (const auto& [eval, pos] : results) {
      if (eval > chunkMaxEval) {
        chunkMaxEval = eval;
        chunkBestMove = pos;
      }
    }
    if (chunkMaxEval > globalAlpha) {
      globalAlpha = chunkMaxEval;
      bestMove = chunkBestMove;
    }
  }
  return bestMove;
}
```

5.2 棋盘与候选缓存

```
// 候选缓存更新
BoardManager::CandidatesDelta BoardManager::updateCandidatesCache(
  const BoardPosition pos
) {
  CandidatesDelta lastRecord;
  lastRecord.removedFromCache = candidateMap[pos.row][pos.col];
  candidateMovesCache.erase(pos);
  candidateMap[pos.row][pos.col] = false;
```

```
  const int r = MAX_CANDIDATE_RADIUS;
  const int minRow = std::max(0, pos.row - r);
  const int maxRow = std::min(BOARD_SIZE - 1, pos.row + r);
  const int minCol = std::max(0, pos.col - r);
  const int maxCol = std::min(BOARD_SIZE - 1, pos.col + r);

  for (int newRow = minRow; newRow <= maxRow; ++newRow) {
    for (int newCol = minCol; newCol <= maxCol; ++newCol) {
      if ((newRow == pos.row && newCol == pos.col) ||
        board[newRow][newCol] != EMPTY) continue;
      if (!candidateMap[newRow][newCol]) {
        BoardPosition newPos{newRow, newCol};
        lastRecord.addedCandidates.push_back(newPos);
        candidateMap[newRow][newCol] = true;
      }
    }
  }
  candidateMovesCache.insert(lastRecord.addedCandidates.begin(),
                             lastRecord.addedCandidates.end());
  return lastRecord;
}

// 候选缓存回滚
void BoardManager::reverseCandidatesCache(
  const CandidatesDelta& delta, BoardPosition moveUndone
) {
  for (const auto& candidate : delta.addedCandidates) {
    candidateMovesCache.erase(candidate);
    candidateMap[candidate.row][candidate.col] = false;
  }
  if (delta.removedFromCache) {
    candidateMovesCache.insert(moveUndone);
    candidateMap[moveUndone.row][moveUndone.col] = true;
```

```
    }
}

// 回滚落子
void BoardManager::undoMove() {
    if (movesHistory.empty()) return;
    MoveRecord lastRecord = movesHistory.back();
    BoardPosition position = lastRecord.position;
    board[position.row][position.col] = EMPTY;
    reverseCandidatesCache(lastRecord.candidatesDelta, position);
    movesHistory.pop_back();
    _blackTurn = !_blackTurn;
}
```

5.3 Controller

```
// 创建新对局与人机回合推进
void GameManager::startNewGame(const char humanColor) {
    _humanColor = humanColor;
    _aiColor = (humanColor == BLACK) ? WHITE : BLACK;
    // 重新创建 AI 引擎实例以重置状态
    _aiEngine = new GomokuAI(_aiColor);
    // 重置游戏状态
    initializeNewGameState();
    // 若 AI 先手则立即落子
    if (isAITurn()) { makeAIFirstMove(); }
}

void GameManager::handleHumanMove(const BoardPosition position) {
    MoveResult result = playHumanMove(position);
    if (!result.moveApplied) return;
    // 人类落子后，发出信号更新 UI
    emit moveApplied(result);
    // 若游戏未结束且轮到 AI，则让 AI 落子
    if (isAITurn() && result.winner == EMPTY && !result.boardIsFull) {
```

```
    MoveResult aiResult = playAIMove();
    if (aiResult.moveApplied) emit moveApplied(aiResult);
  }
}
```

5.4 UI

```
// 独立线程与信号槽连接
void GameWidget::setupGameManager() {
  gameManager = new GameManager();
  gameThread = new QThread(this);
  gameManager->moveToThread(gameThread);
  qRegisterMetaType<MoveResult>("MoveResult");
  connectGameManagerSignals();
  connect(
    gameThread, &QThread::finished,
    gameManager, &QObject::deleteLater
  );
  connect(
    gameThread, &QThread::finished,
    gameThread, &QObject::deleteLater
  );
  gameThread->start();
}

void GameWidget::connectGameManagerSignals() {
  connect(
    this, &GameWidget::handleHumanMove,
    gameManager, &GameManager::handleHumanMove
  );

  connect(
    this, &GameWidget::startGame,
    this, [this](bool playerIsBlack) {
```

```
      const char humanColor = playerIsBlack ? BLACK : WHITE;
      board->resetSnapshot();
      QMetaObject::invokeMethod(
        gameManager,
        [gm = gameManager, humanColor]() {
          gm->startNewGame(humanColor);
        },
        Qt::QueuedConnection
      );
    });

  connect(
    gameManager, &GameManager::moveApplied,
    board, &BoardWidget::onMoveApplied
  );
}

// 重置与协作式取消
void GameWidget::connectResetButton() {
  connect(resetButton, &QPushButton::clicked, this, [this]() {
    if (gameThread) {
      gameThread->requestInterruption();
      disconnect(gameManager, nullptr, this, nullptr);
      disconnect(gameManager, nullptr, board, nullptr);
      gameThread->quit();
      gameThread->wait();
    }
    setupGameManager();
    board->resetSnapshot();
    board->setThinking(false);
    boardStack->setCurrentIndex(1);
  });
}

// 点击落子与棋盘绘制
```

```
void BoardWidget::mousePressEvent(QMouseEvent *event) {
  if (currentPlayerSnapshot != humanColor) { event->ignore(); return; }
  if (event->button() != Qt::LeftButton) {
    QWidget::mousePressEvent(event);
    return;
  }
  // 根据当前屏幕大小计算棋盘布局
  // 并缓存在成员变量中 (`startX`, `startY`, `cellSize`)
  calculateBoardLayout();
  const int x = static_cast<int>(event->position().x());
  const int y = static_cast<int>(event->position().y());
  const int col = (x - startX + cellSize / 2) / cellSize;
  const int row = (y - startY + cellSize / 2) / cellSize;
  if (row < 0 || row >= BOARD_SIZE ||
    col < 0 || col >= BOARD_SIZE) { event->ignore(); return; }
  const int targetX = startX + col * cellSize;
  const int targetY = startY + row * cellSize;
  if (abs(x - targetX) > cellSize / 2.5 ||
    abs(y - targetY) > cellSize / 2.5) { event->ignore(); return; }
  emit cellSelected(row, col);
  event->accept();
}

void BoardWidget::paintEvent(QPaintEvent *) {
  QPainter painter(this);
  painter.setRenderHint(QPainter::Antialiasing);
  painter.fillRect(rect(), bgColor);
  calculateBoardLayout();
  drawBorders(painter);
  drawGridLines(painter);
  drawCriticalPoints(painter);
  drawStones(painter);

  // 绘制特殊状态下的覆盖层
```

```
  if (boardFullSnapshot) drawWinnerOverlay(painter, "It's a Draw!");
  if (winnerSnapshot != EMPTY)
    drawWinnerOverlay(
      painter,
      winnerSnapshot == BLACK ? "Black Wins!" : "White Wins!"
    );
  if (!boardFullSnapshot && winnerSnapshot == EMPTY && thinking)
    drawThinkingOverlay(painter);
}
```

5.5 构建与运行

项目使用 CMakeLists.txt 进行构建，并通过 Makefile 简化构建操作。例如：

```
make build  # 构建项目
make launch # 运行项目
make perf   # 性能测试
make help   # 帮助信息
```

6 分析与讨论

6.1 优势与不足

- 优势：良好 UI 响应性与用户体验；清晰模块划分与代码结构；有效的 alpha-beta 剪枝、候选排序、根节点并行化，显著提升 AI 计算性能；跨平台支持。
- 不足：未使用 Iterative Deepening, Zobrist Hashing, Bitboard 等高阶优化技巧；未实现禁手等高级游戏规则；仅实现人机对战模式；无悔棋功能。

6.2 可改进方向

- 引入 Zobrist Hashing 与 Bitboard 等进一步优化性能
- 探索 MCTS 等其他搜索算法，或结合深度学习提升棋力。
- 实现更多游戏规则与模式，如禁手、悔棋、人人对战、自对弈等。
- 增强文档与测试覆盖。

7 备注

- 项目地址：GitHub - Gomoku: Play Gomoku with AI.
- 本文档中出现测试截图与性能数据均在如下环境下获得：
  - 硬件：Apple M1 Pro，8 核 CPU (6P + 2E)，16GB 内存
  - 操作系统：macOS Sequoia 15.7.2
  - 编译配置：Release 模式
- 本文档为总结性设计文档，部分实现细节可能略有不同，具体可参考源码。