AI 五子棋项目报告

何子谦

2025-11-17

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 ≈ 2–3s；
  - 顺序评估：平均单步计算总用时 ≈ 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 实现

4.1 关键类代码摘录

- `BoardManager`：棋盘存储、落子合法性、胜负检测
- `GameManager`（Controller)：状态机、异步调用 AI、取消机制
- `GomokuAI`：候选生成、评估函数、搜索入口

4.2 线程与任务提交

- `GameWidget::setupGameManager` 将控制器 `GameManager` 移至子线程，注册跨线程 `MoveResult`，连接信号后启动线程。
- 重置流程：请求中断 → 断开连接 → 退出等待 → 重建 `GameManager/QThread`。

4.3 UI 结构

- 主窗口布局：控制区 + 棋盘绘制区 + 状态信息
- `BoardWidget` 绘制逻辑：坐标换算、鼠标事件件

4.4 信号/槽示例

- `GameManager::moveApplied(MoveResult)` → `BoardWidget::onMoveApplied(MoveResult)` 刷新棋盘与胜负状态。

- `BoardWidget::cellSelected(int,int)` → `GameWidget::handleHumanMove(BoardPositio`
  → `GameManager::handleHumanMove(BoardPosition)`。
- `ColorChooserWidget::colorChosen(bool)` → `GameWidget::startGame(bool)`
  → `GameManager::startNewGame(char)/makeAIFirstMove()`。

4.5 配置与常量

- `BOARD_SIZE=15`, `EMPTY/BLACK/WHITE`, `MAX_DEPTH=7`, `MAX_CANDIDATE_RADIUS=2`。

4.6 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
```

5 测试结果

5.1 运行截图

5.2 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`：`std::async` 开销与单点评估耗时基准。
- 备注：后续可在此记录不同硬件上的实际耗时与线程规模对比。

5.3 棋力评估

- 当前未集成自对弈模式；建议通过脚本驱动 `GameManager/GomokuAI` 进行离线对弈评测。

5.4 资源占用

- 待补：长局对战的内存曲线与 CPU 峰值；重置频繁下的线程回收情况。

6 分析与讨论

6.1 优势与不足

- 优势：UI/逻辑解耦清晰；根节点并行带来可观加速；评估函数含威胁/中心控制。
- 不足：无迭代加深/TT/哈希；禁手未实现；仅人机单模式；无悔棋。

6.2 可改进方向

- 引入 Zobrist 哈希与置换表；迭代加深 + 期望剪枝；更细粒度并行；自对弈训练参数。
- 规则扩展：禁手/开局规则；支持人人与 AI 自对弈。