

数据结构与算法 I

综合大实验 实验报告

学院：计算机科学与工程学院（网络空间安全学院）

专业：计算机类

班级：2025080911

姓名：董庆锋

学号：2025080911006

指导教师：俸志刚

实验日期：2025 年 12 月 15 日

实验地点：校信息中心御林安全工作室

五子棋 AI 实验报告

摘要

本实验实现了一个具备完整交互界面与 AI 对弈能力的五子棋程序。项目采用 **Electron + C++** 的分层架构：前端（Electron）负责 UI 渲染、交互与状态展示；后端（C++）负责棋局规则与 AI 搜索计算，并通过 **标准输入输出 (stdio)** 与前端进行实时通信。

在工程实现上，我经历了从 **Qt UI 到 Electron UI** 的重构过程：将原有桌面界面迁移到 Web 技术栈，完成了页面布局、动效、主题切换、数据持久化等功能，并梳理了“主进程/渲染进程/子进程（C++）”之间的通信链路，形成一个可维护的前后端协作模式。这一过程让我系统性地理解了全栈开发中的模块划分、接口设计与数据流动。

在算法实现上，AI 以 **Minimax + Alpha-Beta 剪枝** 为核心，并结合 **迭代加深、启发式估值（模型打分/预设定式）** 与 **Zobrist 哈希置换表** 等优化策略，使其能够在限时条件下稳定给出较高质量的落子决策。实验最终实现了一个可跨平台运行、界面美观且具备一定强度的五子棋 AI 对弈系统。

最终，棋力达到了中等水平（使用互联网大模型驱动的五子棋ai进行测试），能够在标准时间限制下击败大部分初学者玩家。通过本次实验，我不仅巩固了 C++ 算法实现能力，还提升了前端开发与全栈架构设计的综合素养。

1. 实验环境

硬件环境 (Hardware)

- CPU:** Intel Core Ultra 9 275HX
- RAM:** 32GB
- Storage:** 1TB UMIS SSD

软件环境

本项目具有良好的跨平台兼容性，在以下环境中均经过测试并可正常运行：

- Windows:** Windows 11 25H2 26220.7523 (运行于本地机器)
- Linux:** Ubuntu 22.04 (运行于 VMware 虚拟机)
- 开发框架:** Electron (前端), C++ (后端算法)

2. 核心实现与全栈架构

本项目采用 **Electron + C++** 的全栈架构：前端负责界面渲染与交互，后端负责高性能 AI 计算；两者通过标准输入输出 (Standard I/O) 进行实时通讯。

(项目早期曾使用 Qt 编写界面，但为了获得更灵活的 UI 表达与更高的可扩展性，后续重构为 Electron 前端。)

2.1 全栈架构与前后端通讯

Electron 前端

前端使用 Electron 框架，通过 Node.js 的 `child_process` 模块启动 C++ 核心进程。在 `electron_ui/main.js` 中，我们管理着后端进程的生命周期：

```
JS
// electron_ui/main.js
const { spawn } = require('child_process')
// ...
// 启动 C++ 核心
const gameProcess = spawn(execPath, [], {
  cwd: path.dirname(execPath),
  stdio: ['pipe', 'pipe', 'pipe']
})

// 监听 AI 输出
gameProcess.stdout.on('data', (data) => {
  const str = data.toString()
  // 发送给渲染进程 (UI)
  if (mainWindow) mainWindow.webContents.send('game-output', str)
})
```

在 `electron_ui/renderer.js` 中，UI 事件被转化为指令发送给主进程，进而转发给 C++：

```
JS
// electron_ui/renderer.js
// 玩家落子
function handleUserMove(x, y) {
  // ... 更新 UI ...
  // 发送指令给 C++: "MOVE x,y"
```

```
    ipcRenderer.send('game-input', `MOVE ${x},${y}\n`)
}
```

C++ 后端

后端是一个控制台应用程序，核心逻辑位于 `src/widget/main_console.cpp`。它维护一个死循环，不断读取标准输入指令，调用 AI 计算，并将结果打印到标准输出。

C++

```
// src/widget/main_console.cpp
int main() {
    // ... 初始化 ...
    GomokuLogic game;
    AlphaBeta ai(1000, 10000, 1.414, true, 2); // 1秒限时

    while (true) {
        string line;
        if (!getline(cin, line)) break; // 读取指令

        vector<string> parts = split(line, ' ');
        string command = parts[0];

        if (command == "MOVE") {
            // ... 解析坐标 ...
            game.placePiece(x, y); // 玩家落子

            // AI 思考
            cout << "AI_THINKING" << endl;
            pair<int, int> bestMove =
ai.getBestMove(game.getBoard());
            game.placePiece(bestMove.first, bestMove.second);

            // 输出结果供前端解析
            cout << "MOVED " << bestMove.first << "," <<
bestMove.second << ",1" << endl;
        }
        // ... 其他指令处理 ...
    }
}
```

2.2 AI 核心算法

AI 模块 (`src/widget/aibrain.cpp`) 实现了基于 **Alpha-Beta 剪枝** 的 Minimax 算法，并结合了多种优化策略。

1. 迭代加深

为了在有限时间 (`timeLimitMs_`) 内给出最佳走法，我们不固定搜索深度，而是从浅到深逐步搜索。这确保了 AI 随时都能返回当前最优解，避免超时。

C++

```
// src/widget/aibrain.cpp :: getBestMove
auto start = std::chrono::steady_clock::now();
auto deadline = start + (std::chrono::milliseconds)(long long)timeLimitMs_;

for(int depth=1; depth <= MAX_DEPTH; ++depth){
    if(std::chrono::steady_clock::now() > deadline) break; // 超时检测

    // 执行 Alpha-Beta 搜索
    // ...

    // 如果找到必胜走法，提前结束
    if(bestScore >= SCORE_OPEN_FOUR) break;
}
```

2. 启发式评估与预设定式

为了弥补纯搜索的不足，我们在 `evaluate` 函数中引入了基于模型的评分机制。这实际上是一种“软编码”的定式库。

我们定义了不同模型的权重：

C++

```
// src/widget/aibrain.cpp
static constexpr int SCORE_FIVE = 1'000'000'0; // 五连（必胜）
static constexpr int SCORE_OPEN_FOUR = 1'000'000; // 活四（必胜）
static constexpr int SCORE_BLOCKED_FOUR = 300'000; // 冲四
```

```
static constexpr int SCORE_OPEN_THREE = 15'000; // 活三
// ...
```

通过 `countPatternsLine` 函数扫描全盘，统计各种模型的数量并计算总分。这使得 AI 能够识别“活三”、“冲四”等关键形状。

3. Zobrist 哈希 & 置换表

使用 Zobrist Hashing 将棋盘状态映射为唯一的 64 位哈希值，并利用置换表 (`TRANS_TABLE`) 缓存搜索结果，极大减少了重复计算。

C++

```
// src/widget/aibrain.cpp
// 计算哈希
uint64_t computeHash(const int b[BOARD_SIZE][BOARD_SIZE]){
}

// 在 Alpha-Beta 中查表
int ttIndex = currentHash & TT_MASK;
if(TRANS_TABLE[ttIndex].hash == currentHash &&
TRANS_TABLE[ttIndex].depth >= depth){
    return TRANS_TABLE[ttIndex].value;
}
```

4. 必胜开局定式

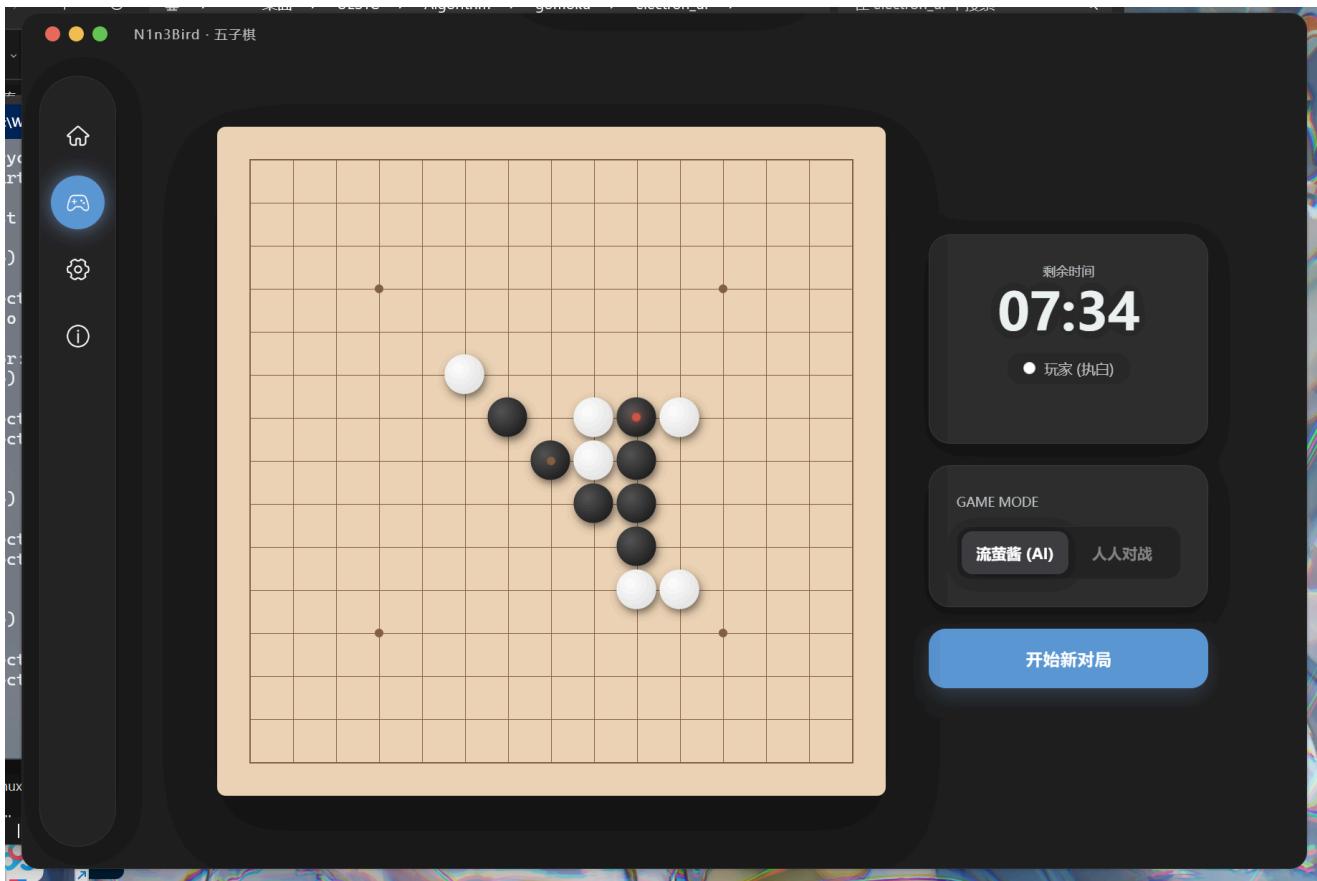
在 PvE 模式下，为了保证先手优势，我们在逻辑层强制 AI (黑棋) 第一手走天元 (7,7)。

C++

```
// src/widget/main_console.cpp
if (isPvE) {
    game.placePiece(7, 7); // 必走天元
    cout << "MOVED 7,7,1" << endl;
}
```

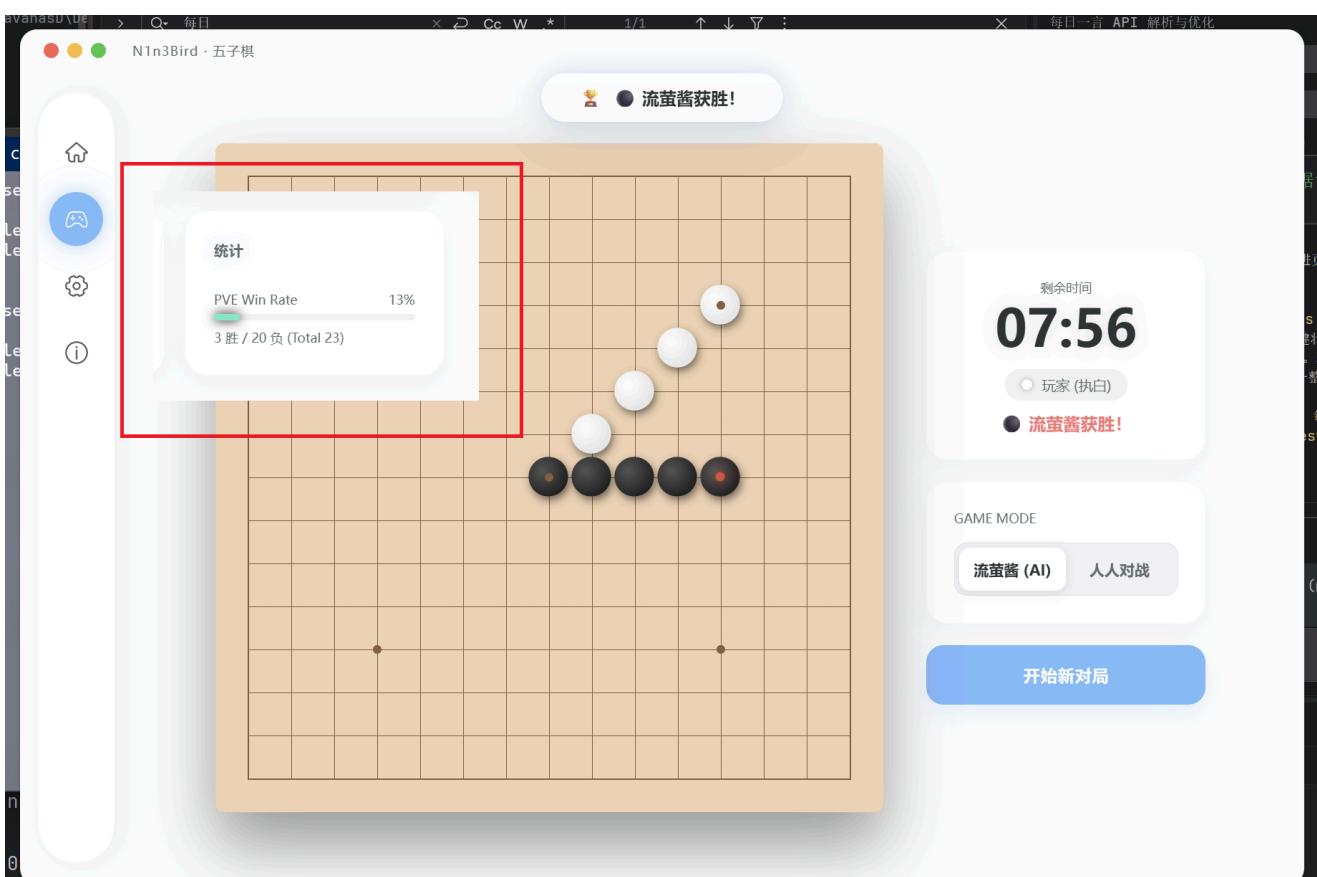
4. UI 展示

主界面 ![游戏主界面](./electron_ui/images/主页.png) 对局

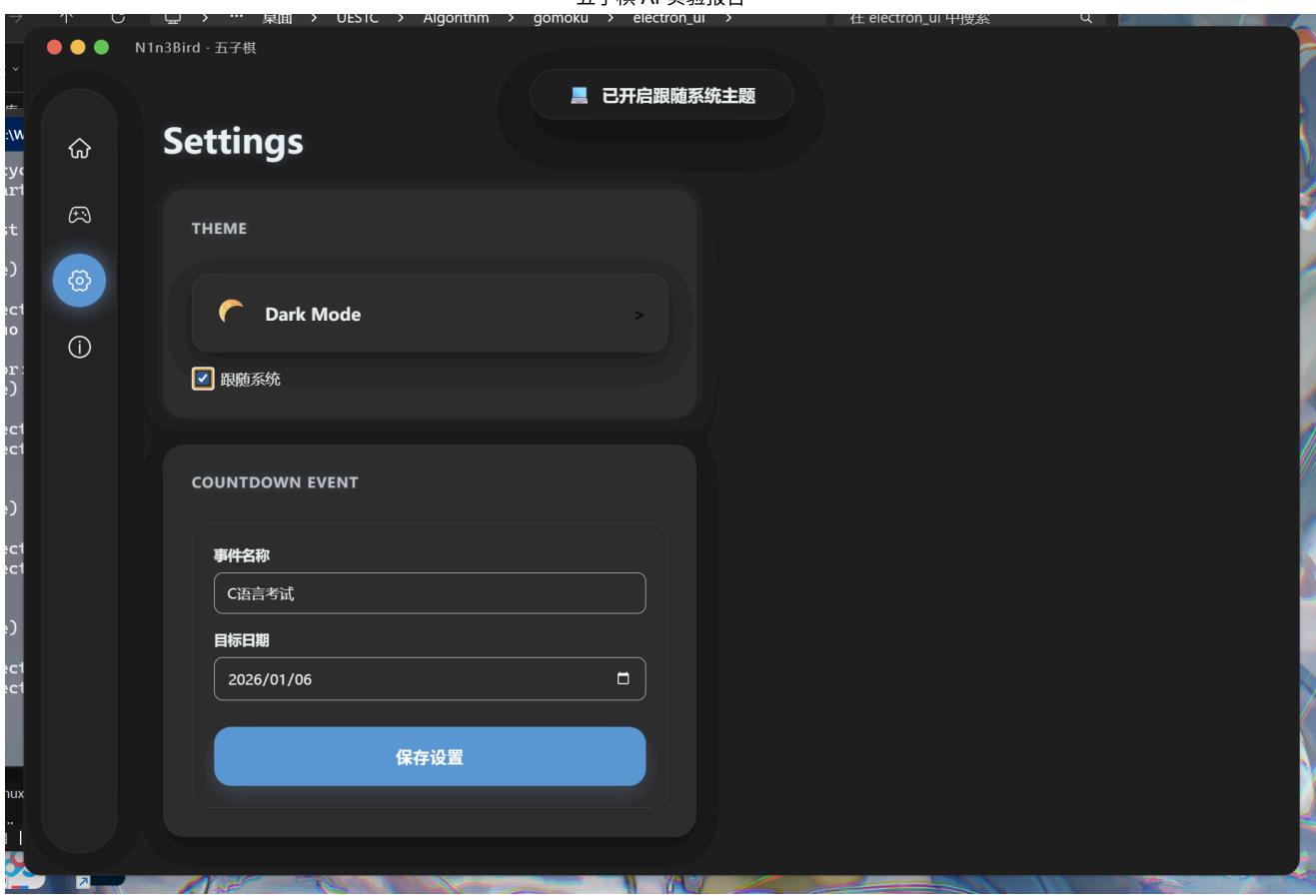


思考时加入了高斯模糊遮罩 ![对局画面](./electron_ui/images/下棋_思考.png)

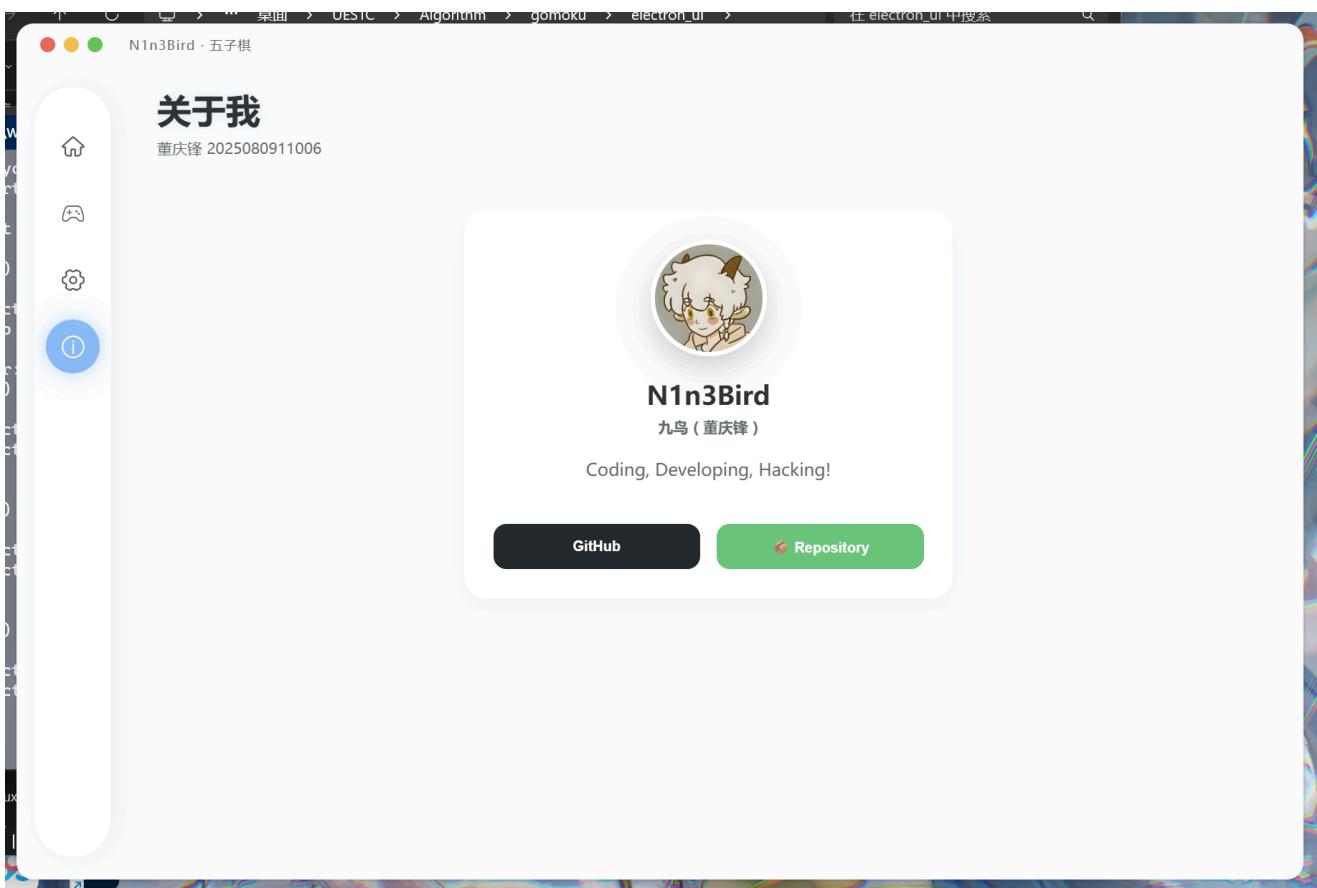
胜利统计功能



支持深色浅色切换 ![设置界面](./electron_ui/images/浅色模式.png) ! [设置界面](./electron_ui/images/浅色主页.png) 支持跟随系统 (Node.js库)



主页倒计时支持自定义 ![设置界面](./electron_ui/images/自定义事件.png) 个
人主页



5. 改进方向

结合当前代码结构，未来可以在以下方面进行优化：

1. 算力优化：

目前的 `alphabeta` 搜索是单线程的。也许可以尝试CUDA并行化，利用GPU强大的并行计算能力来加速搜索过程，从而提升AI的思考深度和速度。

2. 更完善的开局库：

目前仅硬编码了天元开局。可以引入外部的 `.lib` 或数据库文件，存储常见的高胜率开局（如花月、浦月等），在游戏前几乎直接查表，瞬间落子。

3. 蒙特卡洛树搜索 (MCTS)：

虽然 Alpha-Beta 在五子棋中很强，但结合 MCTS 可以更好地处理复杂的中盘局面，提供更具“大局观”的走法。

4. 神经网络评估：

目前的 `evaluate` 函数是基于人工定义的模式权重（Feature Engineering）。可以训练一个轻量级的 CNN（卷积神经网络）来替代手工估值函数，让 AI 学习更微妙的模型组合。