|  |
| --- |
| **计算机专业类课程** |
| **实验报告** |
| **课程名称：数据结构与算法I**  **学院专业：计算机科学与工程学院**  **学生姓名：王浩沣**  **学　　号：2024080911008**  **指导教师：俸志刚** |
| **日　　期：2024年 12月 23日** |
|  |
| 电子科技大学计算机学院实验中心 |

**电 子 科 技 大 学**

**实 验 报 告**

**实验一**

# 一、实验室名称：

电子科技大学清水河校区品学楼c区222教室

# 二、实验项目名称：

基于cpp与qt 库实现的五子棋AI与人机对战程序

# 三、实验目的：

实现一个拥有足够智能，能够在无禁手规则下战胜大多数人类的五子棋ai程序，并且人类可以通过ui界面与其进行对弈。

# 四、实验主要内容：

基于qt creator开发了一个带有gui的五子棋人机对战程序，并为它设计智能模块，再进行调试与性能优化，改善程序的性能。本项目实际做到了平均每步时长控制在10秒以内的6层常规搜索+5层算杀搜索（内嵌在常规搜索的生成树内）。

# 五、实验器材（设备、元器件）：

硬件平台：处理器：AMD Ryzen 7 5800H with Radeon Graphics 八核

内存：16 GB ( 镁光 DDR4 3200MHz 8GB x 2 )

软件平台：Windows10操作系统, Qt Creator, Clion

# 六、实验步骤：

1. 问题描述

该项目的实现有两个核心模块：实现落子决策的ai模块和实现与用户交互的ui模块。要想ai做出合理的决策，就需要使其了解五子棋的基本规则以及各种棋型在实际对战中的意义，并且通过搜索算法实现智能、通过剪枝算法等加快运算。而要想实现与用户的交互，就需要设计一个基本的游戏界面，它应当可以识别用户的鼠标点击以实现对应的落子、在ai思考的过程中给出“ai正在思考中”的提示、在游戏一方获得胜利时做出判定并且显示胜利的一方。同时，还可以实现落子的音效、悔棋和重新开始的按键以改善用户的游戏体验。最后，对于棋盘背景的调整、可莉语音的插入以及下棋时的背景音乐进一步增强了用户的代入感，优化游戏体验。

2. 算法分析与概要设计

输入：对于交互界面模块来说，是用户鼠标在游戏界面中棋盘对应位置的点击；对于五子棋agent来说，是代表棋盘状态的二维数组。

输出：对于交互界面模块来说是五子棋棋盘以及悔棋、重置、选边按钮、对用户和电脑落子位置的记录；对五子棋agent来说是思考后得出的最佳落子位置

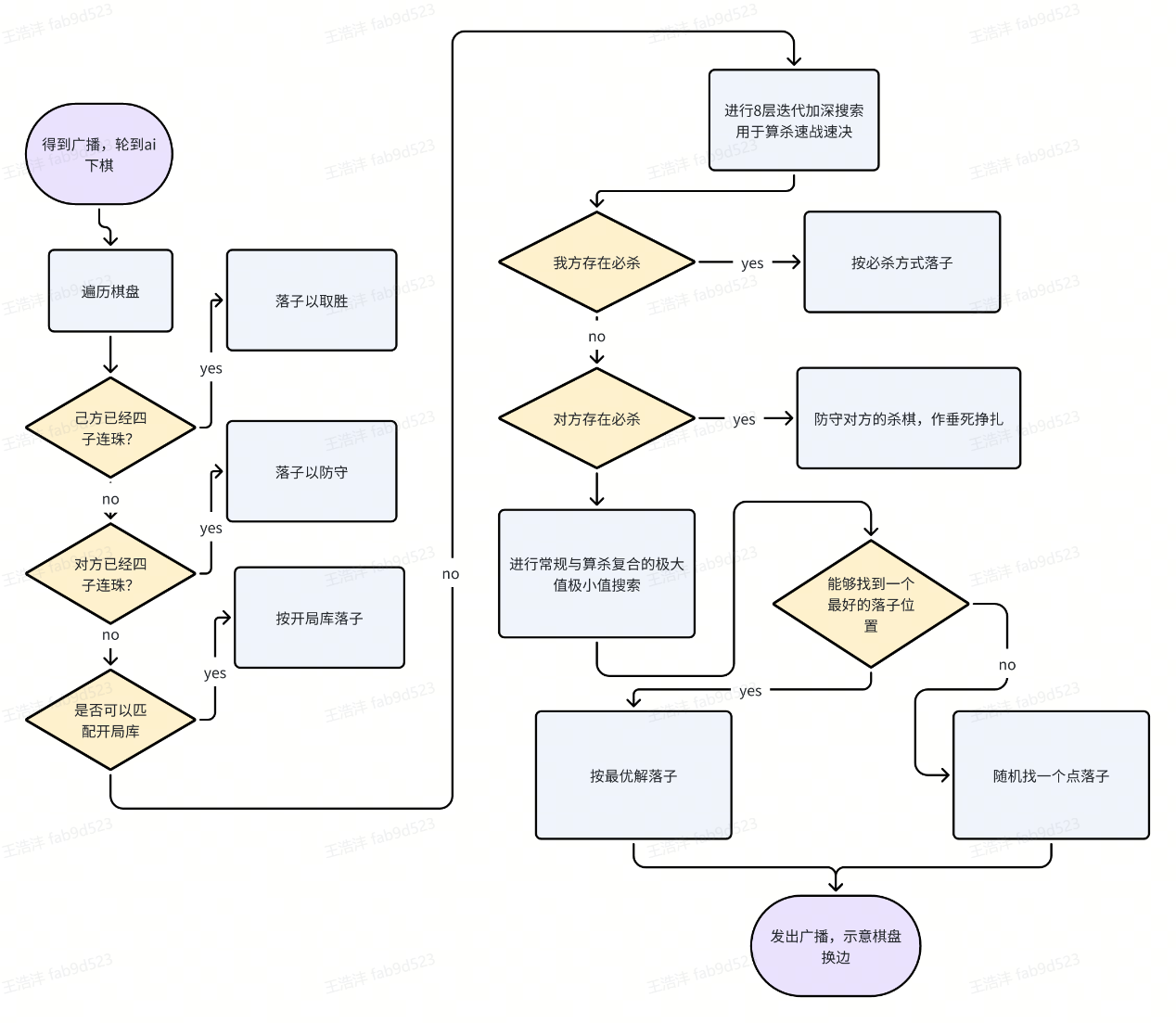
输入转换为输出的算法描述（流程图+文字解释算法设计思想）

本实验中五子棋ai模块主要使用了以下算法：极小值极大值搜索算法、alpha-beta剪枝算法、迭代加深搜索算法。同时，针对取胜方案在搜索过程中内置了算杀模块，使ai最多可以计算到未来的12步棋。单独针对开局决策的准确度和思考速度设计了开局库模块（然而用处不大，为保证棋力只在很小的范围进行应用）针对单线程运算cpu利用率不高且运算速度过慢的问题，采用了QConcurrent库和QThread库进行多线程并行化处理。

本实验中人机对战交互界面主要使用了qt库中的QWidget, QLabel, QThread,QPoint, QSoundEffect, QMouseEvent, QMessageBox, QTimer库，通过将具体的执行函数链接到窗口组件对应的槽函数，实现用户通过窗口组件与程序的交互。

3. 核心算法的详细设计与实现

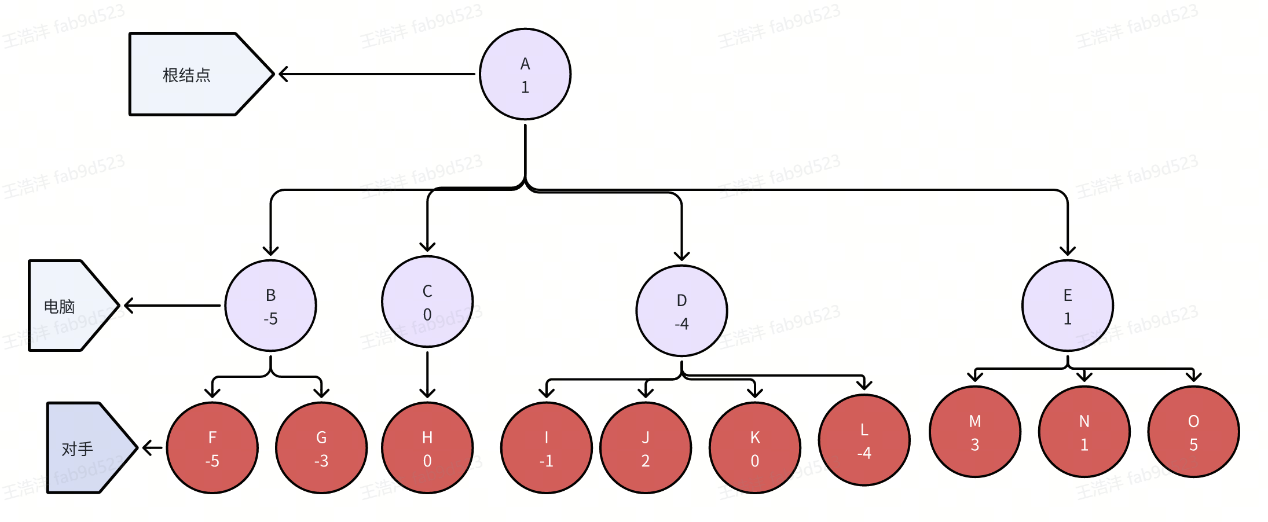
（详细设计一般用流程图+文字解释说清楚算法设计步骤，对于比较复杂的算法，可以粘贴核心代码，但不能有大量代码在这里出现）

ai模块的总体运行逻辑如上图所示（飞书云文档绘制）

对棋盘遍历找四子连珠的算法十分简单，在此不做赘述。迭代加深搜索在代码层面是基于极大值极小值搜索的，因此下面详细介绍极大值极小值搜索算法以及与之配套的启发式评估算法。

由于五子棋中的博弈是零和的，一方收益最大化时，另一方收益就会最小，当一方获胜得到最大的收益时，另一方也会因失败获得最小的收益。在棋局开始之初，两方的收益均为零，故双方想要取胜，就会选择在长远来看最大化自身收益同时最小化他人收益的落子方法。

假设两个玩家（MAX为计算机、MIN为其对手）都发挥最佳效果，那么在计算机移动后，对手将选择最小化(但对于对手来说是最有利的操作)的移动，由此可得计算机应在考虑其移动和对手的最佳移动的情况下选择最佳移动。

如上图所示计算机在根节点有四种选择方式，每种选择之后其对手又有不同种选择方式，在其对手选择完成后就会进入终止状态产生结果，计算机 (MAX) 为确保自己所获利益最大，应当选择B、C、D、E中最大的值，因为对手 (MIN) 总会选择对自己最有利的决策，所以B值的由来是F、G中的最小值、D的由来是I、J、K、L的最小值，由此就可以得到了极小极大值的算法，极小代表的是在计算机 (MAX) 决策后其对手 (MIN) 总会选择对于计算机 (MAX) 来说获利最小的操作，极大值代表的是计算机 (MAX) 需要在所有的决策中(所有的极小值中)，选择最大的值。由此得出的伪代码如下：

函数 Minimax(节点, 深度, 最大化玩家):

如果 节点 是 叶节点 或 深度 == 0:

返回 评估函数(节点)

如果 最大化玩家:

最大值 = -∞

对于 每个 子节点:

值 = Minimax(子节点, 深度 - 1, 假设最小化玩家)

最大值 = max(最大值, 值)

返回 最大值

否则:

最小值 = +∞

对于 每个 子节点:

值 = Minimax(子节点, 深度 - 1, 假设最大化玩家)

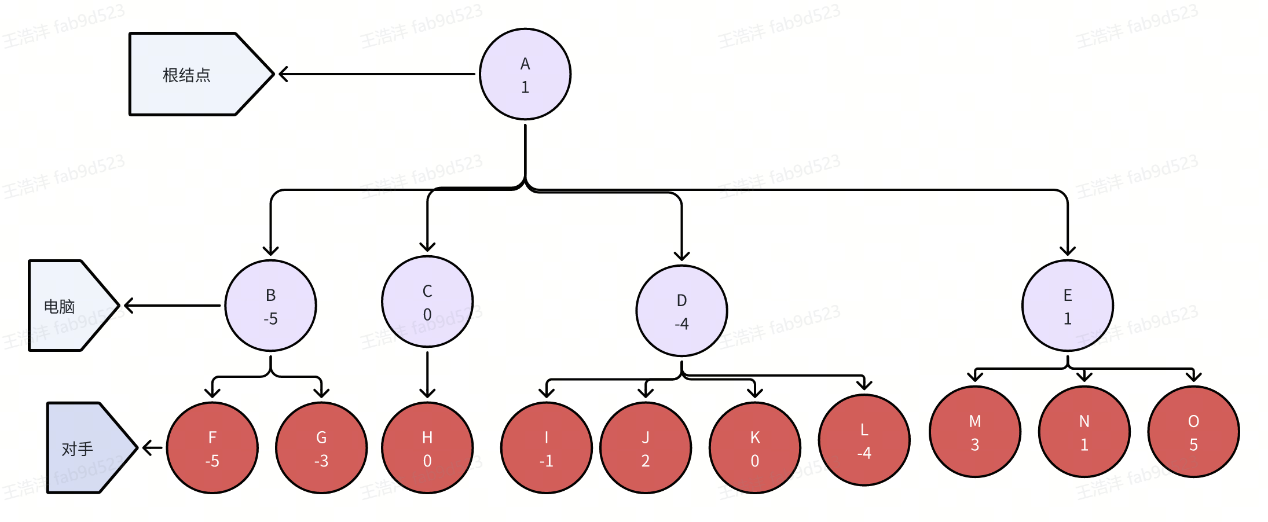
最小值 = min(最小值, 值)

返回 最小值

函数 评估函数(节点):

根据节点的状态返回一个数值，表示当前局面的好坏

未经优化的minmax算法的复杂度为O(k^d),k为生成的分支节点数，d为搜索的深度。而一个熟知五子棋规则的普通棋手的一般能考虑到六步棋左右，即使是对五子棋一知半解的普通人通常也能考虑到之后的四步棋，故欲达成实验目的，搜索深度应当在6以上。本实验中，我为ai设置的分支节点生成逻辑（在正常搜索下）为：在距离当前棋子不超过两格的范围内，所有堵住对方活二、活三、活四的落子（称为防守子）以及创造我方活三、连四、连五的落子（称为进攻子）加上距离棋盘中心最近的十个以内的既不属于防守子又不属于进攻子的落子。故通常情况下，游戏进行到中局时，在一层中生成的节点数量在20左右，在六层搜索的情况下就有64,000,000次评估，而一次评估需要进行大量遍历，大约在0.001秒左右，则总用时就为64000秒，这显然远远超出了我们单次落子所能接受的最长时长。故有必要对该算法进行大幅度的优化。回到之前给出的生成树中，



不难注意到，由于根节点会截取下一层的最大值，假如我们最先搜索的是E节点，那么在搜索过E节点之后，返回根节点的值一定就大于1；此时再对D节点进行搜索，若搜索到K，发现值为0，那么D节点的值就一定小于0，从而一定小于1，则D节点下剩余的其他几个节点都无需进行搜索。基于此，我们得到了alpha剪枝方案，则对于MIN对象的搜索就也有对应的beta剪枝方案。这在程序中对应的代码如下：

int AIWorker::minimax(int state[BOARD\_SIZE][BOARD\_SIZE], int depth, int alpha, int beta, bool maximizingPlayer, int lastMoveX, int lastMoveY)

{

int player = maximizingPlayer ? aiColor : ((aiColor == White) ? Black : White);

int opponent = (player == White) ? Black : White;

if (lastMoveX != -1 && lastMoveY != -1)

{

if (checkWin(state, lastMoveX, lastMoveY, opponent))

{

return maximizingPlayer ? (std::numeric\_limits<int>::min() + depth)

: (std::numeric\_limits<int>::max() - depth);

}

}

// 终止

if (depth <= 0)

{

return evaluate(state, aiColor);

}

// 生成

std::vector<std::pair<int, int>> moves = generateCandidateMoves(state,player,depth);

if (moves.empty())

{

return evaluate(state, aiColor);

}

if (maximizingPlayer)

{

int maxEval = std::numeric\_limits<int>::min();

for (const auto& move : moves)

{

state[move.first][move.second] = player;

int eval = minimax(state, depth - 1, alpha, beta, false, move.first, move.second);

state[move.first][move.second] = Empty;

if (eval > maxEval)

{

maxEval = eval;

}

if (maxEval > alpha)

{

alpha = maxEval;

}

if (beta <= alpha)

{

break; // Beta 剪枝

}

}

return maxEval;

}

else

{

int minEval = std::numeric\_limits<int>::max();

for (const auto& move : moves)

{

state[move.first][move.second] = player;

int eval = minimax(state, depth - 1, alpha, beta, true, move.first, move.second);

state[move.first][move.second] = Empty;

if (eval < minEval)

{

minEval = eval;

}

if (minEval < beta)

{

beta = minEval;

}

if (beta <= alpha)

{

break; // Alpha 剪枝

}

}

return minEval;

}

}

在应用alpha-beta剪枝后，算法的复杂度在理想情况下降低为O(k^(d/2))，对于六层的搜索能够控制在十秒以内，符合我们的预期。

五子棋博弈中的局面常常分为开局、中局和收尾。棋局之初，棋盘上一般不会出现大规模的对攻或无解的威胁，双方的精力主要集中在占据更大的空间和有利的棋型上，对于进攻棋型的考虑并不多，因而在这个阶段常规的深搜能够大致满足我们的需求。五子棋进入收尾阶段也是类似的状况，即棋盘半数以上空间都被双方棋子占据后，双方平局的概率大大增加，此时就需要一些特判来加快行棋，减少垃圾时间。然而，中局阶段，双方对拼进入白热化，有时甚至一连七八步都是对方不得不应对的杀棋，仅仅搜索六层不足以应对大多数连杀的情况，因而考虑在搜索过程中加入更高的深度，但为了降低算力的开销，需要对节点生成规则做出优化。在实验指导给出的基于javascript的五子棋ai项目中，算杀是独立于常规搜索之外的搜索过程，但我认为这么做存在问题：在实际的棋局中，在激烈的拼杀中往往会有几步“闲棋”，即双方的进攻手段用尽，需要几步铺垫来组织下一步的进攻。假如在搜索过程中只考虑杀棋的存在，就会漏掉这几步闲棋从而错失好局，而若把算杀仅仅视作寻找必胜落子的方法，则仅仅六层的常规搜索又会漏掉这几步闲棋之后的杀棋，导致对闲棋的落子评估不当。因此，应当采取在较浅的深度生成完全的节点（其实也没那么完全，可以理解为所有有效的节点），而在较深的的深度，只生成可以阻挡对方活四、活五和生成己方活三、活四的落子对应的节点，即每一层只需要生成个位数的节点，假设为5，则本项目做到的6+5搜索所需要的总评估次数大致为(5^2.5\*20^3)=447200。在对启发式评估函数和利用多线程并行化搜索进行优化后，依然能够控制在可以接受的时间之内。

迭代加深算法在本项目中主要应用于找速胜策略，重要性并不高，对棋力影响也不大，故不作过多赘述，其核心内容为逐渐增加搜索深度，直到找到必胜策略为止。这么做的好处是可以在最小步数内取得胜利，加强终结比赛的能力。为减小迭代加深过程对计算效率的影响，本项目中的迭代加深只进行最高八层算杀搜索，在实际实践中已经足敷使用。

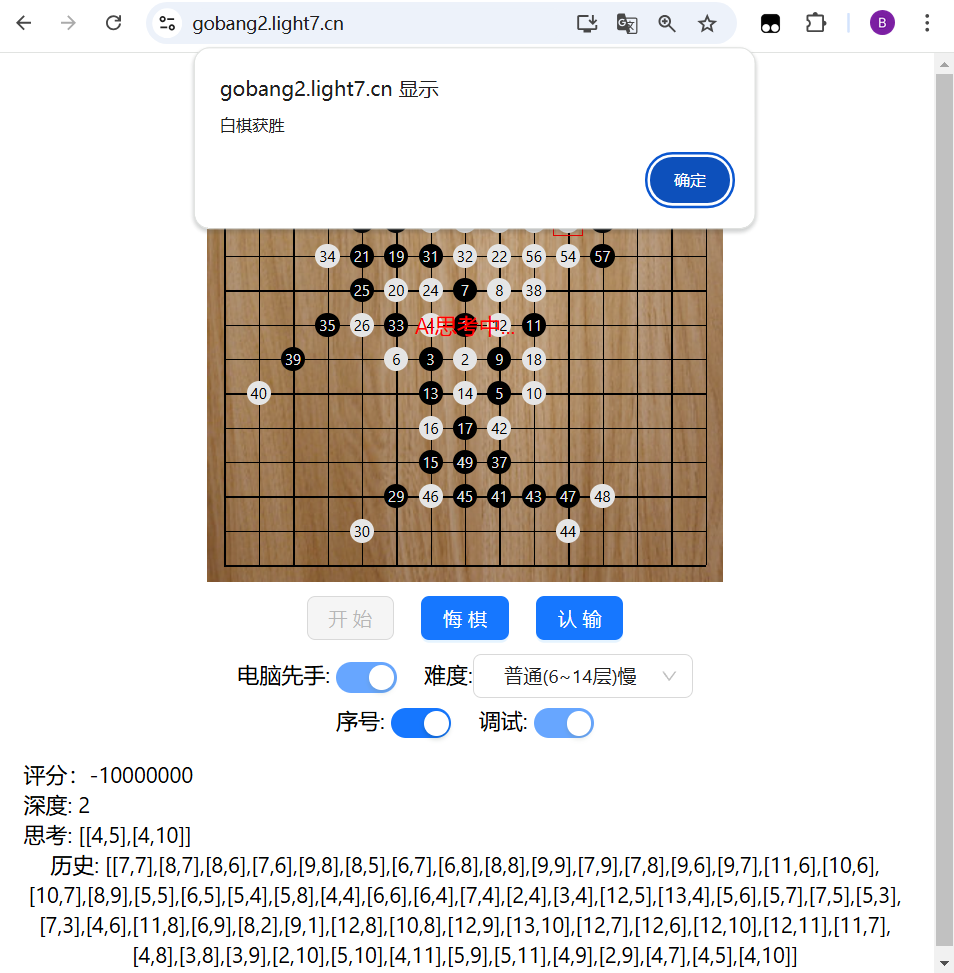
# 七、实验数据及结果分析：

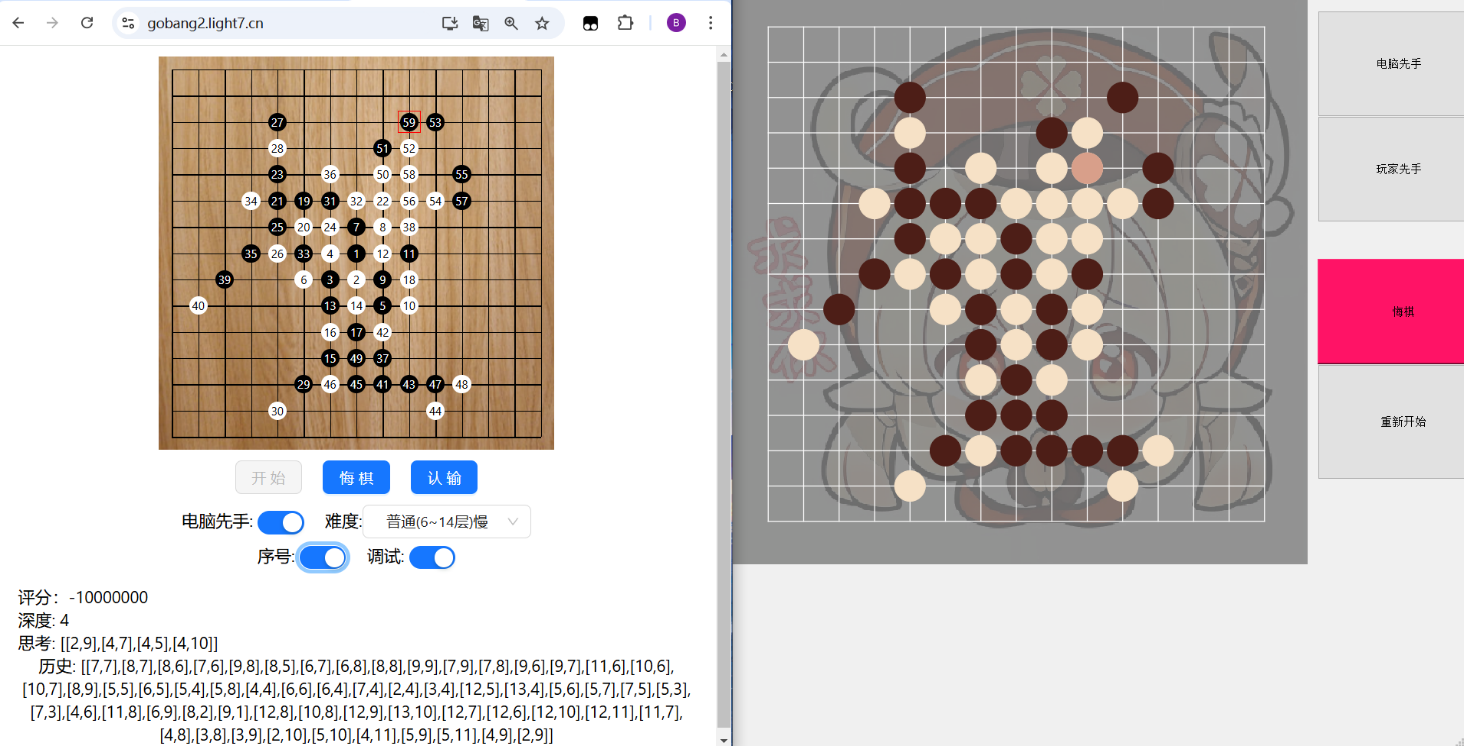
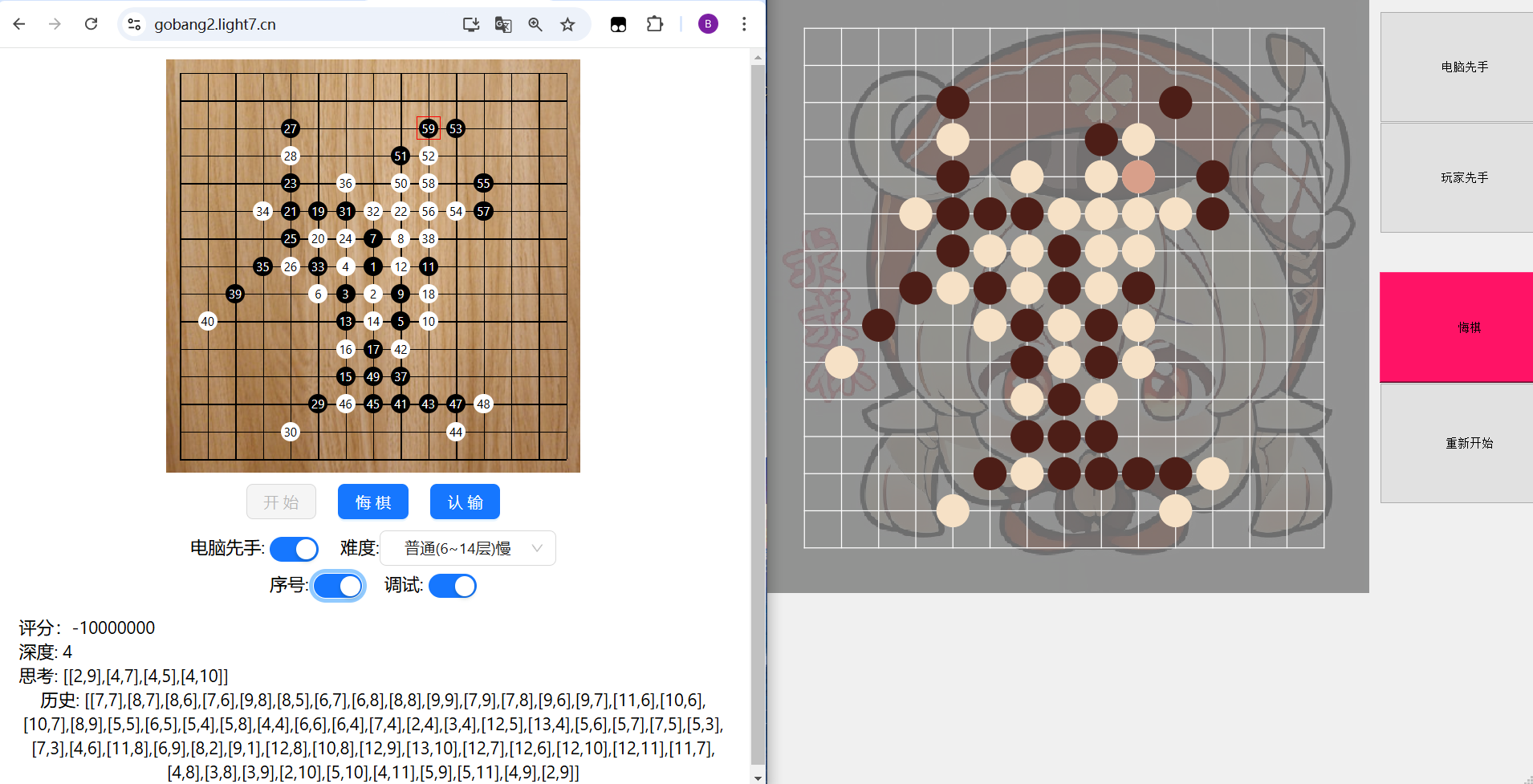
(测试需要给出测试用例（从正常，边界，错误等各方面给出测试用例，建议用3个表格的形式给出3种不同类型的测试用例），贴图（运行结果截屏），分析。做看图说话。每一图都要给出图名。有表则给出表名。请根据实验实际，用大量语言进行描述讨论。)

由于本项目经历的时间线较长，早期的代码大多都被删去，无法复现当时的测试，故对于在项目进展到达6深度常规搜索之前的过程主要以文字形式描述。

本项目的编写历程大致为：

1. 首先基于c语言在Clion上实现了对棋盘的启发式评估以及对生成树的建立与搜索，通过棋盘状态的一些特例验证了该算法的可行性。
2. 在Qt creator中建立了棋盘的结构和游戏进程相关逻辑，并且完成简陋的gui窗口界面。
3. 由于Qt库中存在大量基于类实现的函数库，我在Qt Creator中用cpp重构了算法对应的代码，并且对树生成的节点进行了排序与进一步的剪枝（事后证明当时的做法是错误的，当时将节点根据离中心远近进行排序，再进行剪枝，导致大量有用的节点被剪掉）
4. 在深度为2的条件下对程序进行了初步的调试，发现两个问题：(1)先手在棋盘边缘落子时ai对我方进攻完全没有反应(2)在棋局进行到中局时ai经常忽视我方的进攻意图。
5. 由于怀疑是深度过低和启发式评估考虑情况不到位导致的问题，我将深度提高到了4，并且在启发式评估中加入了对于双活三、冲四活三、双冲四的判定。在调试过程中，首先发现了ai此时的思考时间就已经有些长了，如果用户落子较快，就会跳过ai直接落子，导致棋局出现混乱。为解决此问题我引入了QThread库，为 ai单独建立一个线程，并在该线程运行过程中封锁棋盘，并创建了一个Qlabel来标明ai正在思考，避免用户在ai思考期间落子或失去耐心。而后又发现ai忽视我方进攻的问题依然存在，且大多数时候都是在我方下出跳三或跳四时，故又为评估函数增加了对跳三的判定（跳四直接写了一个在进入搜索之前直接封堵的逻辑）但问题依然存在。
6. 因此又对代码进行了检视，并且与其他同类项目进行了比对，发现在评估函数赋值方面存在差异。修改赋值后依然存在问题。充分排查后发现在生成枝干时会剪去一些有用的枝，故将生成树中节点的逻辑改为在距离当前棋子不超过两格的范围内，所有堵住对方活二、活三、活四的落子（称为防守子）以及创造我方活三、连四、连五的落子（称为进攻子）加上距离棋盘中心最近的十个以内的既不属于防守子又不属于进攻子的落子。之后发现代码运行时间变长，故将深度调整为4，尝试挑战实验指导给出的范例的6~14层模式（地址为<https://gobang2.light7.cn/>），在后手的条件下出乎意料地击败了它（因为它在一个非常奇怪的地方漏看了一个双跳三），但是在己方先手时又很快地输给了它，于是决心模仿其算杀搜索。
7. 由于六层以上运行速度过慢，我之后基于Qconcurrent库对搜索的过程进行了优化，由于时间关系，没能对搜索的每一层都应用多线程，而是采取了进入搜索递归过程之前，先生成一遍走法，然后从这些走法出发分多个线程进行并行化搜索的方法。在搜索完之后，再将这些走法汇总、合并，并得出最终的落子。在应用多线程之前，六层深度搜索下开局的前几步通常需要20秒左右的思考时间，而应用多线程之后时间降到了5秒以内，在性能上获得了巨大的提升。（就是有点烧cpu）
8. 在确保性能足以支持ai实现算杀后，我对generateCandidateMoves函数进行了修改，使其传递当前深度的参数，并且基于不同深度的情况令其给出了不同的生成方法（具体方法见前文）。在完成算杀后，与实验指导中给出的五子棋ai进行对弈，尽管思考时间约为后者的两倍，但无论是先手还是后手都取得了全胜的战绩。其中有一局的对局情况如下：



在第49手棋后，对方对自己的评分来到了+10000000，认为自己稳操胜券，然而，本项目的ai接连发起冲四进攻，并在59手后杀死比赛，使对方对自己的评估发生两极反转。11层>14层，赢！

当然，由于本项目的ai在必胜局面下热衷于冲四的坏习惯，后续还是给它写了一个单独的迭代加深过程来找速胜下法，进一步降低了每步的平均耗时。

而最终综合开局和最后几步的快速收割，本项目的ai在每步平均耗时上取得了平均9.56秒的好成绩，达成了实验目标，真是可喜可贺，可喜可贺啊！

# 八、总结及心得体会：

在该项目中，由于技术原因我并未设计缓存模块，导致基于此的Zobrist 置换表算法和针对评估函数的米字优化未能实现，最终表现为与同样采用极大值极小值搜索算法但同时采用了Zobrist 置换表算法和针对评估函数的米字优化的程序相比，计算时间较长。（但是网上也有说法表示应用置换表对五子棋ai运算速度提升帮助不大）

由于时间关系，一些代码级别的优化尚未得到实现，如： 1.在一些只需要读取棋盘信息的函数中，传递了棋盘本身而非地址，导致内存占用增大 2.由于棋盘信息存储在vector向量中，且项目采用了多线程方法，将其存储为一维数组会更有利于cpu缓存的读取以及内存的分配。 3.对于线程的分配采用了均分的方法，导致在大部分线程结束时，依然有某个线程还在运作，导致算力资源恒定的条件下没有实现运算最大化 。

悔棋功能依然存在bug：假如从中局开始一直回退到开局，标识ai最后一手落子的特殊颜色棋子不会随着最后一次悔棋而消失，而是会存留在棋盘上（尽管我方和ai在该位置的落子不会受到这个标识的影响）。这个bug由一位因为下不过人机破防而疯狂悔棋的室友发现，但是由于时间关系没有办法去修了。

在面对一些针对性下法时，ai有时会掉进圈套。这是基于博弈算法的符号主义人工智能的通病，同时也是没有对下法进行随机化处理埋下的祸端。例如实验指导中给出的基于javascript的五子棋ai示范项目（6常规+14算杀）就会在二十步棋之内输给笔者编写的尚未完成的四层常规搜索ai（但是当换边后，对方又能速胜，说明不是对方搜索不到位导致的）。实际博弈中我们并不能太过依赖对“对方会倾向于下出我们认为的最优解”的假设，在有一些项目中，甚至会优先考虑对方最后一步落子周围八格的情况，而对距离较远的落子进行剪枝，这固然会使其对战评估方式类似且同样采用深度搜索算法的ai时占据优势，但当对上人类棋手时其表现则会下降一个台阶。

# 九、对本实验过程及方法、手段的改进建议及展望：

由于笔者的懒惰加上Qt Creator的profile不太好使，本次实验在性能优化方面无论是过程还是结果都不甚理想。对于耗时最多的evaluatePattern函数的优化也非常有限。要是报告能晚交一周，保证把所有棋盘相关的存储都塞到一维数组里，避免指针跳来跳去或者if判断一堆变量。希望俸爷能高抬贵手，放咱一条生路吧！

**报告评分：**

**指导教师签字：**