数据结构与算法 I

基于蒙特卡洛树的五子棋AI算法实验报告

学院：计算机科学与工程学院（网络空间安全学院）

专业：计算机大类

班级：计算机三班

姓名：周贻雯

学号：2025080903029

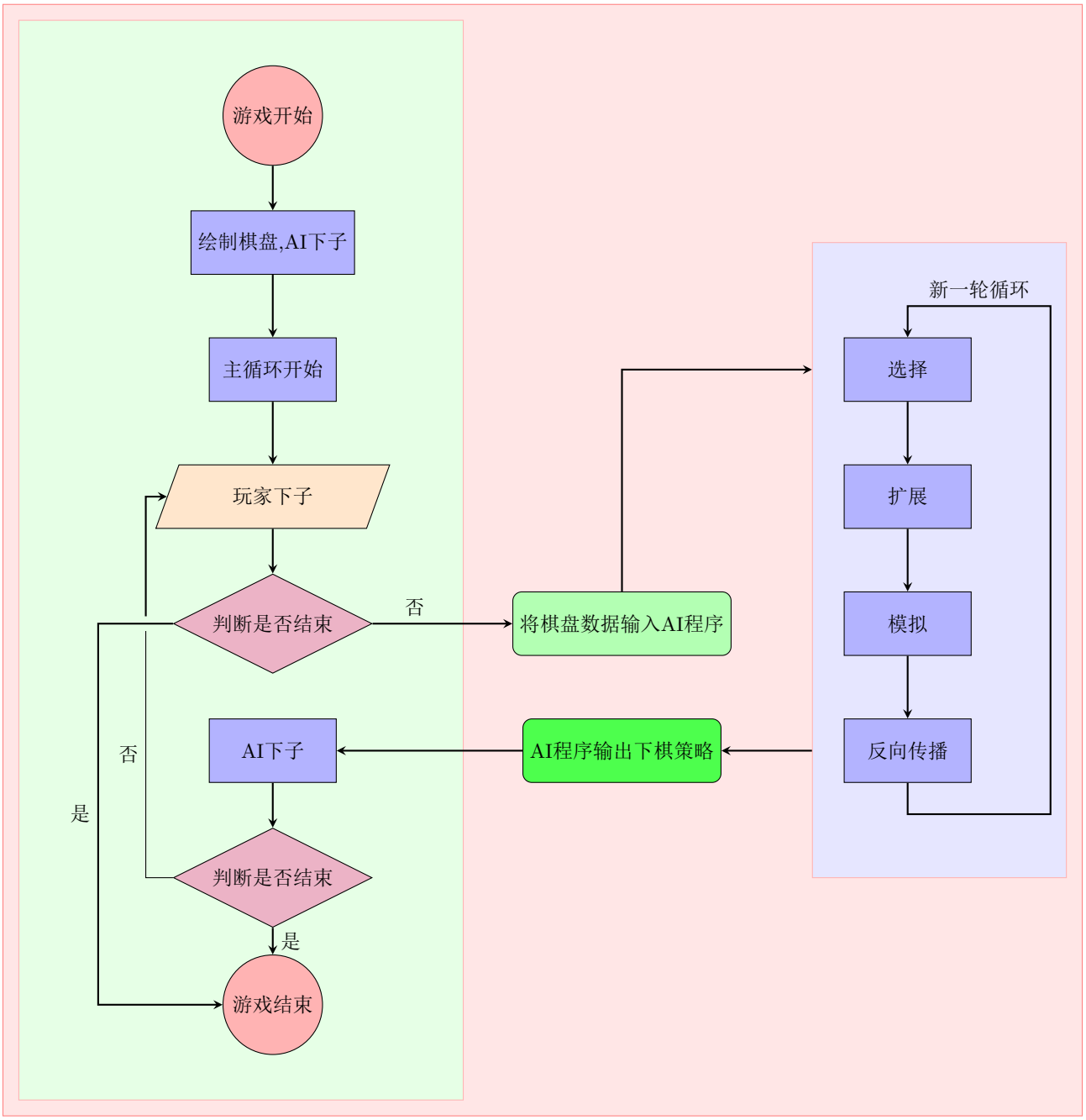
指导老师：俸志刚

实验时间：2025 年 10 月至 2025 年 12 月

摘要

本文主要描述的是对五子棋AI的算法实现。文章将主要从两个方面展开，包括**五子棋游戏的设计思路**以及**对于某些重要细节的实现的具体方法**。

程序实现了五子棋游戏的基本逻辑，包括棋盘管理、落子交互、胜负判定以及核心的AI算法功能。游戏程序的主要流程如下：



在绘制棋盘阶段（相当于预处理阶段），程序将棋盘抽象为15×15的数组，将每个可供下子的位置都将被记录下来。进行运算的棋盘信息包含每一行、列、主对角线、副对角线上的落子情况。游戏限定黑棋（AI）方为先手，初始位置为天元。

在对弈中，AI方的决策是程序的重中之重，每当AI需要进行决策时，当前棋盘的信息将被输入到决策程序当中，经历选择、扩展、模拟以及反向传播的过程，最终得到下棋策略并输出。

- **选择：** AI程序将以输入的棋盘情况作为根节点，进行子节点的选择。对于任何当下被选择的节点，如果它从未被选择过，那么它将被选定，如果它曾被选择过且仍有可拓展的子节点，当前选择将直接从可拓展的子节点中选取。选定子节点后都将进行模拟落子以及结果的反向传播，每完成一次该过程，该节点所在路径上各个节点的状态都将被更新。对于一个父节点，当它的子节点已经全部拓展完成，程序将依据UCB值择优选取子节点。另外，如果选择到一个已经达到终局的节点，程序将立即返回该节点。**选择**是AI决策程序的主要框架。

- **拓展：** 对于一个曾被选择过的节点，需要对其进行子节点的选择时，程序将计算此时棋盘的几何中心，并从距该中心的一定范围内开始选取空白区域落子，将此时的棋盘情况作为该节点的子节点。若此时所有空白区域都曾被选择过，那么该节点将被认为没有可拓展的子节点

- **模拟：** AI程序将以传入的棋盘数据作为起点开始快速随机落子，直到达成游戏终止条件。

- **反向传播：** 程序会将模拟结果沿当前节点的路径依次传递给经过的节点，并更新途径节点的数据，这些数据将参与到UCB值的计算以及最终的决策当中。

对于每个棋盘的数据，AI程序会使用*unordered_map*进行存放，使得在计算UCB时对节点（棋盘状态）的统计信息（访问次数、胜利次数等）的获取变得十分方便，有效地节省了时间。

引入位棋盘以及位运算进行大部分有关棋盘局面的计算、推演，使得程序能在极短的时间里进行大量计算，这显著地提高了程序的运行效率。

对于一些特殊情况，譬如活三这样的显著威胁，单纯的MCTS算法可能无法稳定排除，因此我们引入启发式算法，能够做到对双三，活三等情况的提前防御。很大程度上提升了AI程序的防御水平，也显著提升了一些情况下AI程序的响应速度。

除了对五子棋游戏玩法以及AI程序基础功能的实现，大量努力被应用在对AI算法的优化上，相较于AI程序的雏形，当前展示的AI在运行效率和决策水平等方面都有了长足的进步。例如，程序在早期的响应时间大约在一分钟左右，经过一系列的优化，该数值已经被缩短到了十余秒。在与诸多五子棋爱好者进行对弈的过程中，AI表现出不错的下棋能力。这些实践证明，我们所构建的AI程序已经具有了能够投入实际应用的初步能力。

目录

实验环境

1. 硬件环境

CPU 型号：Inter(R) Core(TM) Ultra 5 125H

内存大小：15.5G

2. 软件环境

操作系统：windows 11

编译工具：MinGW (GCC)

调试工具：GNU gdb

其他辅助软件：Qt Creator

第一部分　五子棋游戏的设计思路

1　预处理阶段

1.1　棋盘状态

首先我们需要建立一个准确描述棋盘的所有落子情况的二维数组（*ChessBoard*），数组中的每一个元素都将对应UI界面棋盘上的一个点位，从而将棋盘抽象为一种便于理解的数据形式。

二维数组的每一个元素都被定义为结构体，其作用如下：

1. 定义清晰的棋盘状态

在五子棋中，一个格子只有三种可能的状态。使用*enum class*可以将这些状态符号化：

- *None* (对应 0)：表示该格点为空。
- *White* (对应 1)：表示该格点落有白子（玩家）。
- *Black* (对应 2)：表示该格点落有黑子（AI）。

```
enum class Player:char{
    None=0,     //空位
    White=1,    //白棋（玩家）
    Black=2     //黑棋（）AI
};
```

2. 性能优化

代码中特意写成*enum class Player* ： *char*，这是一个非常关键的底层优化：

2.1 内存优化

- **缩小体积：**默认的*enum*通常占用 4 字节（*int*）。指定为 *char* 后，每个格点仅占用 1 字节。
- **整体缩减：**对于 15×15 的棋盘，如果用 *int* 存储，一个棋盘需要 900 字节；而用 *char* 只需要 225 字节。

2.2 类型安全

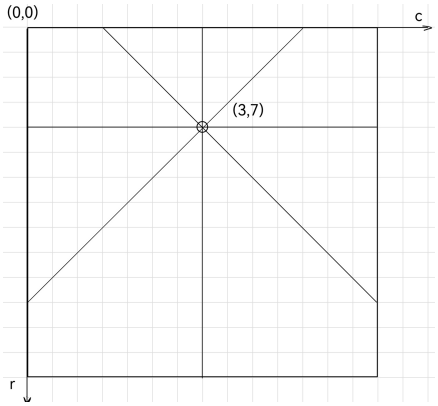
- **强类型检查（防止意外 Bug）** *enum class* 具有强类型属性，编译器不允许将其与其他类型（比如 *int*）进行隐式转换。在五子棋 AI 中，使用 *enum class Player* 可以防止开发者意外地将棋盘状态与普通的整数进行运算或比较，从而增强了程序的鲁棒性。

1.2　位棋盘处理

为了将棋局的相关推演进行简化以达到性能优化的目的，我们需要将*ChessBoard*描述的棋盘局面进一步抽象化。因此我们引入位棋盘，将棋盘上每一行、列、主对角线以及副对角线上的落子情况用16位无符号整数来表示。

采用*BitBoard*结构体来储存棋盘数据，结构体主要包含四个数组，例如，对于一个在第三行第七列落有一子的棋盘（起点为$(0,0)$），如图：

将该棋盘各项数据按照下列计算方法更新：



- **行**：第 3 行的整数 $row[3]$，第 7 列对应二进制的第 7 位偏移（从0开始计数，从右往左数），记录为 0000000010000000
- **列**：第 7 列的整数 $col[7]$，第 3 行对应二进制的第 3 位偏移，记录为 0000000000001000
- **主对角线**：索引为 $i+j$ 即 $3+7=10$，对应 $diag1[10]$，偏移量 Off（以左上角为起点）为 $Off=i=3$，记录为 0000000000001000
- **副对角线**：索引为 $i-j+14$即 $3-7+14=10$，对应 $diag2[10]$，偏移量 $Off=i=3$，记录为 0000000000001000。

每一行、列、主对角线和副对角线的数据都将被如此记录，相关代码如下：

```
if(player==Player::Black){
        black.row[r] &= ~(1u<<c);
        black.col[c] &= ~(1u<<r);
        black.diag1[diag_map[r][c].diag1_id] &= ~(1u<<diag_map[r][c].diag1_off);
        black.diag2[diag_map[r][c].diag2_id] &= ~(1u<<diag_map[r][c].diag2_off);
    }
    else{
        white.row[r] &= ~(1u<<c);
        white.col[c] &= ~(1u<<r);
        white.diag1[diag_map[r][c].diag1_id] &= ~(1u<<diag_map[r][c].diag1_off);
        white.diag2[diag_map[r][c].diag2_id] &= ~(1u<<diag_map[r][c].diag2_off);
    }
```

每个棋盘数据都包含两个*BitBoard*结构体，分别储存黑子和白子的状态。

1.3 *unordered_map*

对于任意一个棋盘局面的数据，程序需要储存它的胜率以及访问次数，在运行过程中，这些数据会被高频次访问，因此建立一个可以快速访问的机制尤为重要，为此我们引入*unordered_map*，以此实现$O(1)$查询。程序定义了两个核心映射表，它们都以棋盘局面（*ChessBoard*）为键：

statemap 值为*StateProperty*,用于存储该局面的胜利次数 win 、访问次数 $visit$以及其子节点，AI 检索当前局面相关数据，从而进行计算 UCB 值等操作。

parantmap 值为*ChessBoard*，用于存储该节点的父节点，在反向传播阶段，AI 需要顺着这条“线索”回到根节点，一路更新沿途所有节点的胜率数据。

*unordered_map*默认的*key*不支持自定义的类和结构体，想要让 *ChessBoard* 作为 *key* 就需要满足两个核心条件：**可哈希**与**可比较**，我们通过哈希函数（*ChessBoardHash*）和重载相等运算符（==）来实现。

1. 哈希函数

为了将 15×15 的棋盘转化为一个唯一的哈希值，程序采用了 $FNV-1a$ 算法：

- 过程：初始化一个偏移量 $offset$，遍历棋盘的每一个格点 $grid[i][j]$，不断进行异或 (XOR) 和大质数相乘。
- 目的：产生“雪崩效应”。只要棋盘上有一个棋子位置不同，生成的哈希值就会产生巨大的差异，从而保证了局面的唯一性。

代码如下：

```
struct ChessBoardHash {
    std::size_t operator()(const ChessBoard& board) const noexcept {
        std::size_t h=0;
        const std::size_t prime=1099511628211ULL;
        const std::size_t offset=14695981039346656037ULL;

        h=offset;
        for(int i=0;i<BOARD_ROWS;i++) {
            for (int j=0;j<BOARD_COLS;j++) {
                h^=static_cast<std::size_t>(board.grid[i][j]);
                h*=prime;
            }
        }
        return h;
    }
};
```

2. 重载相等运算符

当两个不同的局面产生了相同的哈希值（即哈希冲突）时，*unordered_map* 需要一种方法来确定它们是否真的相等。我们选择重载相等运算符来实现。

- 实现：通过重载 ==，程序会逐一对比两个棋盘的 *grid* 数组。只有所有格点状态完全一致，才认为局面相同。
- 对于不会抛出异常的函数（如只涉及查询，计算的函数），代码中使用了 *noexcept* 关键字，提升执行速度。

```
bool operator==(const ChessBoard& a,const ChessBoard& b) noexcept{
    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(a.grid[i][j]!=b.grid[i][j]){
                return false;
            }
        }
    }
    return true;
}
```

1.4 UI界面设置

程序的UI界面实现了传统五子棋的界面设置，符合大众审美，便于玩家接受。

此外，为了便于程序设计，我们将初始化条件进行了限定，规定AI方执黑棋，玩家方执白棋，让AI方先手下于常规的天元位置。

2 游戏过程

此后是常规的对弈环节。玩家方与AI方交替执子，每下完一子之后交换执子权，不允许悔棋，每一步棋无时间限制。玩家方下完一着以后，若游戏未结束，程序会将当前棋盘数据传入AI程序，经由AI分析演算以后传回UI界面，实现AI方着子操作，接着进行后续游戏。

有关AI方分析演算的技术细节，将在第二部分详细介绍。

3 游戏结束判定

程序遵循传统的游戏结束条件，即满足“五连珠”的胜负条件或者棋盘填满的平局条件时，程序即告终止。

3.1 胜负条件

当一方落子后，系统会检查该棋子在四个方向（横向、纵向、主对角线、副对角线）上是否形成了连续的 5 个同色棋子，采用位运算的方法实现判定。

1. 实现原理：位移与按位与

在位棋盘中，棋子的排列被存储在 $uint16\_t$ 类型的整数中。判断是否有“五连珠”，本质上是检查二进制序列中是否存在连续的 5 个 1。

逻辑推导： 假设某个方向（如某一行）的二进制状态为 b：

执行 b & $(b << 1)$：结果中为 1 的位表示该位置和它左边的一位都是 1（连续 2 个 1）。

执行 b & $(b << 1)$ & $(b << 2)$：结果中为 1 的位表示连续 3 个 1。

以此类推，执行 b & $(b << 1)$ & $(b << 2)$ & $(b << 3)$ & $(b << 4)$。

判定：如果最终结果不等于 0，说明该行中至少存在一个位置，向上追溯 5 位全部为 1，即达成五连珠。

```
inline bool has_n_in_a_row(uint16_t mask,int n) noexcept{
    uint16_t x=mask;
    for(int i=1;i<n;i++){
        x &= (mask>>i);
    }
    return x!=0;
}
```

调用此内联函数时只需传入需要检查的整数和 $n = 5$ 即可。

2. 四个维度的全方位检查

为了判定胜负，AI 必须对落子点相关的四个方向进行上述检查。在程序当中，这对应了 $BitBoard$ 结构体中的四个数组。

以横向检查为例，需要进行横行判定时，程序取出对应横行 $row[i]$ 存储的整数，对其进行位移判定，依据结果返回 $bool$ 值。

将检查步骤在四个方向分别进行，只要其中任何一个检查返回 *true* 即判定游戏结束，依据执棋方判定胜者。

完整函数如下：

```
bool GomokuGame::check_win_on_bitboard(const BitBoard& bitboard)const noexcept{
    for(int r=0;r<BOARD_ROWS;r++){
        if(has_n_in_a_row(bitboard.row[r],5)&&!has_n_in_a_row(bitboard.row[r],6)) return true;
    }
    for(int c=0;c<BOARD_COLS;c++){
        if(has_n_in_a_row(bitboard.col[c],5)&&!has_n_in_a_row(bitboard.col[c],6)) return true;
    }
    for(int d1=0;d1<BOARD_ROWS+BOARD_COLS-1;d1++){
        if(has_n_in_a_row(bitboard.diag1[d1],5)&&!has_n_in_a_row(bitboard.diag1[d1],6)) return
    true;
    }
    for(int d2=0;d2<BOARD_ROWS+BOARD_COLS-1;d2++){
        if(has_n_in_a_row(bitboard.diag2[d2],5)&&!has_n_in_a_row(bitboard.diag2[d2],6)) return
    true;
    }
    return false;
}
```

3.2 平局判定

平局的定义非常明确：当经过胜负判定算法检测，得知黑白双方均未达成五连珠，且棋盘上225个位置（15×15）全部被占满时，即判定为平局。

此过程通过双重循环遍历棋盘判定是否有空位来进行。

代码如下：

```
bool GomokuGame::is_terminal(const ChessBoard& board)const noexcept{
    bool flag=true;
    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]==Player::None){
                flag=false;
                return flag;
            }
        }
    }
    return flag;
}
```

第二部分　AI程序的主要算法

从整体来说，我们设计的AI程序主要依托蒙特卡洛树算法，选用这一算法的原因是多样的。

蒙特卡洛树算法依靠大量的重复随机抽样得到数值解，引入 UCB（上限置信区间算法）后 AI能够平衡决策当中的两个关键点：**利用**与**探索**，算法会充分利用那些表现出良好胜率的节点，同时也会兼顾那些虽未被充分探索但是具有“潜力”的节点，确保了AI能够坚持已知的优势路径，也能发掘潜在的最优解。

随机模拟是蒙特卡洛树算法的一个标志性特点，通过大量的随机模拟，每个节点的评估将愈发精准，使得程序最终能够找到当下局面的最优解。

此外，MCT（蒙特卡洛树）的结构允许结合多项技术进行优化，例如运用位棋盘加快运算速度，使用 $unordered\_map$ 实现 $O(1)$ 级别的节点查询。

考虑到如此多的优势，我们最终选择了蒙特卡洛树算法作为AI程序的核心，下文将展示蒙特卡洛树算法在程序中的应用。

4　程序的初始化

4.1　初始化随机数种子

调用 $srand(static_cast < unsigned int > (time(nullptr)))$随机初始化种子。这确保了 AI 在蒙特卡洛模拟阶段的随机落子序列在每次运行程序时都是不同的，避免 AI 表现得过于机械化，保证模拟过程的随机性。

```
GomokuGame::GomokuGame(){
srand(static_cast<unsigned int>(time(nullptr)));
StartGame();
}
```

4.2　内存预分配

由于程序使用的五子棋搜索树十分庞大，对于程序中可能发生的哈希扩容（$Rehash$），如果任由其发生，将有可能导致某次程序运行时间骤增，显然会导致性能抖动和时间损耗。因此，程序在最开始立即对两个核心哈希表进行空间预留。

```
statemap.reserve(500000);
parentmap.reserve(500000);
```

4.3　坐标索引的预计算

在某些操作中（如 $place\_a\_piece$），程序中需要了解某一坐标（r, c）所处的对角线 id 以及偏移量，此时如果每次都进行计算，在如此多的重复操作下，程序运行的耗时将被拉长，鉴于棋盘上某点对应的对角线相关数据是完全固定的，在程序运行的最开始，我们就建立一张索引图（$diag\_map$），后续需要了解相关信息时，直接对索引图进行查询，可以实现 $O(1)$ 的数据更新。

索引图对棋盘上每个位置所在的主对角线及副对角线的 id 进行了计算，同时记录下该位置相对于所在对角线的节点的偏移量 off ，使得程序通过查询索引表可以直接获取相关信息。

```
    std::vector<std::vector<Diaginfo>> init_Diag_map(){
    std::vector<std::vector<Diaginfo>> diag_map(BOARD_ROWS,std::vector<Diaginfo>(BOARD_COLS));
    for(int r=0;r<BOARD_ROWS;r++){
        for(int c=0;c<BOARD_COLS;c++){
            diag_map[r][c].diag1_id=r-c+(BOARD_COLS-1);
            diag_map[r][c].diag1_off=std::min(r,c);

            diag_map[r][c].diag2_id=r+c;
            diag_map[r][c].diag2_off=std::min(r,BOARD_COLS-1-c);
        }
    }
    return diag_map;
}
```

4.4 初始参数与状态设置

棋盘清空：初始化 $ChessBoard$ 结构体，确保 225 个格子全部为 $Player :: None$。

回合管理：设置 $round = 1$ 并在 AI 落子后将操作权切换给玩家（$Player :: White$）

数据清理：清除两个 $unordered\_map$ ，供新游戏使用。

4.5 搜索树树根的建立

开局时，规定AI方落子于天元 $(7, 7)$ 位置。建立棋盘储存该局面，将其作为第一个节点存入 $statemap$ 当中，并将其访问量（$visit$）和胜率（win）初始化为 0。此时它成为了蒙特卡洛树的根节点，所有后续的 UCT 选择和扩展都将以此节点作为树状结构的根部向下延伸。

```
    void GomokuGame:: init_ChessBoard_state(const ChessBoard& board){
    StateProperty p;
    p.visit=0.0;
    p.win=0.0;
    statemap[board]=p;
}
```

整体初始化代码如下：

```
    void GomokuGame::StartGame(){
    statemap.reserve(500000);
    parentmap.reserve(500000);

    current_board=ChessBoard {};
    current_board.grid[7][7]=Player::Black;
    diag_map=init_Diag_map();
    current_player=Player::White;
    round=1;

    statemap.clear();
    parentmap.clear();

    init_ChessBoard_state(current_board);
}
```

5 一些基本操作的实现

在AI程序当中有许多底层的操作，为了增强文章的可读性以及辅助读者了解后文的种种操作的实现逻辑，这一部分将重点介绍包括落子，擦去棋子等基本操作的实现过程。

5.1 放置棋子（*place_a_piece*）

在随机快速落子阶段以及其他的应用场景中，需要快速进行落子操作，我们使用位棋盘加速这一过程。

传入需要更新的黑白位棋盘、需要落子的位置坐标值以及当下落子颜色之后，函数将判断需要更新哪一个位棋盘，并更新它的行、列、对角线数据，并将 is_empty 设为 0（非空）。

数据更新通过位运算实现：

- **行（列）：** 直接对位棋盘对应行（列）与 $1u << r$（$1u << c$）进行按位或赋值。
- **主（副）对角线：**尽管原理一致，在仅输入对应位置的情况下，对对角线的数据更新还需要获取该位置对应的对角线 id 以及偏移量（off）,由于我们在预处理阶段就已经建立起了索引图，此时只需经过简单的查询就可以做到。接着对对角线数据进行同样的按位或赋值更新。

```
    void place_a_piece(const std::vector<std::vector<Diaginfo>>& diag_map,BitBoard& black,
    BitBoard& white,int r,int c,Player player){
    if(player==Player::Black){
        black.row[r] |= (1u<<c);
        black.col[c] |= (1u<<r);
        black.diag1[diag_map[r][c].diag1_id] |= (1u<<diag_map[r][c].diag1_off);
        black.diag2[diag_map[r][c].diag2_id] |= (1u<<diag_map[r][c].diag2_off);
        black.is_empty=0;
    }
    else{
        white.row[r] |= (1u<<c);
        white.col[c] |= (1u<<r);
        white.diag1[diag_map[r][c].diag1_id] |= (1u<<diag_map[r][c].diag1_off);
        white.diag2[diag_map[r][c].diag2_id] |= (1u<<diag_map[r][c].diag2_off);
        white.is_empty=0;
    }
}
```

5.2 擦除棋子（*erase_a_piece*）

AI在模拟落下一子之后，经常需要将其回收，此时要求进行擦除操作，它与落子操作互为逆操作，两者搭配使用可以让深度搜索在相同位棋盘上进行，避免了重复的创建和撤回庞大的棋盘对象。此外，$O(1)$ 级别的擦除使得AI可以以极快的速度撤回数量众多的落子方案。

擦除操作与落子操作在函数结构上基本相同，不同的是，它使用“清除位”操作进行数据更新，且不会对 is_empty 做更新。

```
    void erase_a_piece(const std::vector<std::vector<Diaginfo>>& diag_map,BitBoard& black,
    BitBoard& white,int r,int c,Player player){
    if(player==Player::Black){
        black.row[r] &= ~(1u<<c);
        black.col[c] &= ~(1u<<r);
        black.diag1[diag_map[r][c].diag1_id] &= ~(1u<<diag_map[r][c].diag1_off);
```

```
        black.diag2[diag_map[r][c].diag2_id] &= ~(1u<<diag_map[r][c].diag2_off);
    }
    else{
        white.row[r] &= ~(1u<<c);
        white.col[c] &= ~(1u<<r);
        white.diag1[diag_map[r][c].diag1_id] &= ~(1u<<diag_map[r][c].diag1_off);
        white.diag2[diag_map[r][c].diag2_id] &= ~(1u<<diag_map[r][c].diag2_off);
    }
}
```

5.3 UCB计算

UCB 的计算是选择步骤的重要依据，它能够统合胜率与探索度这两个决策因素，找到当下最值得尝试的子节点。

UCB 值的计算公式为：

$$UCB = \frac{w_i}{n_i} + C\sqrt{\frac{\ln N}{n_i}}$$

参数解释：

符号	含义
i	进行计算的节点
w	节点的胜利次数
n	节点的访问次数
C	探索系数，为常数，决定了探索项的重要程度
N	父节点的访问次数

代码为节点设置了“托底”机制，在函数的最开始将进行判定，如果某一节点从未被探索过（访问量 $visit$ 为 0 ）那么函数将返回一个极大的 UCB 值，确保每一个节点都至少会被选中一次。

按照一般情况，C 被设置为$\sqrt{2}$（采用1.414）；对 N 和 n 都做了加一处理防止 $\ln 0$情况；考虑到节点的视角问题，胜率项按视角做了转换处理。

```
    double GomokuGame::UCB(const ChessBoard& board,Player player) noexcept{
    if(statemap[board].visit==0) return 1e9;
    const double c=1.414;
    double tol_visit=statemap[parentmap[board]].visit;
    double win_rate=statemap[board].win/statemap[board].visit;
    double node_visit=statemap[board].visit;
    double search_weight=(c-1.0/2.0*round/(BOARD_ROWS*BOARD_COLS))
                        *sqrt(log(tol_visit+1.0)/(node_visit+1.0));
    return (player==Player::Black)? win_rate+search_weight:-win_rate+search_weight;
}
```

5.4 棋盘几何中心的计算（cal_center）

在随机落子阶段，盲目落子显然不是我们想要的，虽然不能对每一步下棋做精细控制，但是可以通过将下棋范围限制在当前落子中心的一定范围内，规避一些意义不大的落子，从而节省一定算力。

首先需要对传入的棋盘局面的两个位棋盘验空以保证 $board$ 的两个位棋盘已经装载了棋盘数据，如果两个棋盘确实为空，函数也可以调用 $place\_piece$ 进行一次全量转换。

```
    if(black.is_empty&&white.is_empty) place_piece(board,black,white,diag_map);
```

接下来需要遍历整个棋盘，得到黑白子个数之和并统计所有棋子的行号和以及列号和。我们依靠双重循环来实现。

以 i 作为计数器，循环15次，每次循环计算一次黑白位棋盘在第 i 行棋子总数。这一步我们可以通过 *__builtin_popcount* 来实现， *__builtin_popcount* 可以直接返回一个二进制整数中 1 的个数，对应位棋盘中某行的棋子数。

```
    cnt+=__builtin_popcount(black.row[i])+__builtin_popcount(white.row[i]);
```

循环内嵌两个循环体，分别计算黑白位棋盘的情况，下面仅以黑子位棋盘为例说明计算原理。

当一行中存在棋子时，可以对棋子们逐个进行计数，即：记录一个，删除一个，直到这一行棋子数为零。此时这一行对应的二进制整数中1的个数为零，整数数值同样为零，不再满足循环条件，自此循环结束。

当循环仍在进行时，可以确定这一行仍有棋子，可以对行号直接累加。列号的累加需要计算棋子所在列，通过位运算 *__builtin_ctz* 获得， *__builtin_ctz* 可以返回二进制数末尾 0 的个数，此数值与现存最左端的棋子列号相等，将其叠加到列号之和中。这两步计算的是该行现存最左端棋子的行、列号，计算完毕后需要对其进行清除以计算下一个棋子行、列号。使用减一后按位与清除最后一个 1 ，它对应该行中现存最左端的棋子。

计算完毕后函数返回行号均值与列号均值。

完整代码如下：

```
std::pair<int,int> GomokuGame::cal_center(const ChessBoard& board,BitBoard black,BitBoard white)
    noexcept{
    if(black.is_empty&&white.is_empty) place_piece(board,black,white,diag_map);
    int x=0,y=0,cnt=0;
    for(int i=0;i<BOARD_ROWS;i++){
        cnt+=__builtin_popcount(black.row[i])+__builtin_popcount(white.row[i]);
        while(black.row[i]!=0){
            x+=i;
            y+=__builtin_ctz(black.row[i]);
            black.row[i] &= black.row[i]-1;
        }
        while(white.row[i]!=0){
            x+=i;
            y+=__builtin_ctz(white.row[i]);
            white.row[i] &= white.row[i]-1;
        }
    }
    x=std::round(1.0*x/cnt),y=std::round(1.0*y/cnt);
    return {x,y};
}
```

5.5 剪除多余节点（*reuse*）

AI程序在经历蒙特卡洛树算法的推演后会留下一整棵搜索树，以及两张 *unordered_map* 然而当玩家方下子后，大量数据已经沦为废弃数据。因为程序选定子节点作为下棋策略后，根节点的其他子节点以及它们延伸出来的所有结点都没有意义了，它们事实上没有发生；玩家落下一子以后，被选定的那个子节点的某个符合玩家落子情况的子节点及它延伸出的的所有节点可以保留，因为它正确预言了玩家落子后的棋盘情况，其余子节点应当全部删除。

函数主要分两步进行。

5.5.1 记录需要保留的节点

首先建立一个 *unordered_set* 来储存需要保留的节点，将它的容量设定为 50000，避免哈希扩容。接着定义一个的 *Lambda* 深度搜索表达式，

在表达式中，首先检查是否存在局面 *b* 若有，则返回，避免在复杂的搜索图中陷入死循环。若没有，则将它加入到 *keep* 当中。通过深度搜索，可以将以传入棋盘局面为根节点的子树都添加到 *keep* 中。

```
    std::unordered_set<ChessBoard,ChessBoardHash> keep;

    keep.reserve(50000);

    std::function<void(const ChessBoard&)> dfs= [&](const ChessBoard& b){
        if(keep.count(b)) return;
        keep.insert(b);
        auto it=statemap.find(b);
        if(it!=statemap.end()){
            for(const auto& child : it->second.children){
                dfs(child);
            }
        }
    };

    dfs(board);
```

5.5.2 剪除多余节点

函数通过两个循环分别将未被 *keep* 保留的全部节点从两个 *unordered_map* 中删除。

函数完整代码如下：

```
void GomokuGame::reuse(const ChessBoard& board){
    std::unordered_set<ChessBoard,ChessBoardHash> keep;

    keep.reserve(50000);

    std::function<void(const ChessBoard&)> dfs= [&](const ChessBoard& b){
        if(keep.count(b)) return;
        keep.insert(b);
        auto it=statemap.find(b);
        if(it!=statemap.end()){
            for(const auto& child : it->second.children){
                dfs(child);
            }
        }
    };

    dfs(board);

    for(auto it=statemap.begin();it!=statemap.end();){
        if(!keep.count(it->first)){
            parentmap.erase(it->first);
            it=statemap.erase(it);
        }
        else{
            it++;
        }
```

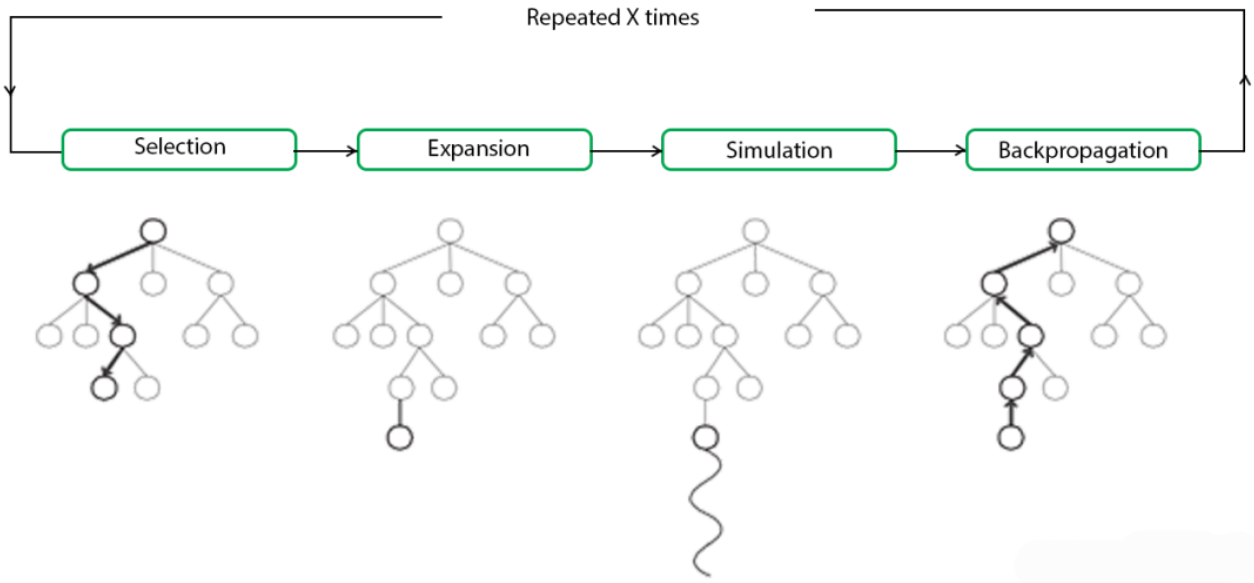
```
27      }
28
29  }
```

6 蒙特卡洛树算法（$MCTS$）简介

蒙特卡洛树算法（$Monte\ Carlo\ Tree\ Search$）是本文所描述的AI决策的核心算法，它的原理在于通过大量的随机模拟评估某一局面的价值，从而做出对于当下局面最优的决策。

蒙特卡洛搜索树当中的节点均为棋盘局面，以传入的当下局面为根节点，任意一个父节点的子节点都是以其父节点为基础再下一子后形成的棋盘局面。正是通过对当下局面可能形成的局面的不断探索与大量模拟，根节点的子节点的优劣可以被逐渐区分，最终AI程序可以基于大量统计数据推导出一个对当下局面最优解的可靠判定。

蒙特卡洛树算法过程如图所示：



蒙特卡洛树算法包含四个不断循环的阶段：

1. 选择（$Selection$）

以初始化的棋盘为根节点，向下选择子节点，选择子节点要兼顾两个标准：

- **1.** 基于算法希望找到当下局面最优解的基本要求，选择进行模拟的节点必然要考虑其胜率。
- **2.** 在搜索树展开的初期，存在大量没有充分探索的节点，它们的潜力不可被忽视，因此选择节点时也要兼顾节点的探索度。

为了综合考虑两个因素在选择节点进行模拟时的作用，我们利用 $UCB(Upper\ Confidence\ Bound)$ 来判断哪个节点最适合被选择。

选择将会沿着搜索树向下不断进行，直到遇到一个叶节点，也即从未被模拟过的节点、无法被拓展的节点或已经达到终局的节点，并将这一节点返回到模拟阶段。

2. 拓展（*Expansion*）

搜索树是逐渐建立的，而拓展就是搜索树建立的途径。它从父节点出发，选择一个空白位置随机落子，得到的新棋盘局面就是该父节点的子节点，将其加入搜索树中。子节点不应有重复，因此落子位置不应当与已有的子节点重复。

可见，在限定拓展范围的前提下，一个父节点能够拓展出的子节点是有限的，当一个节点所有可能的子节点都已被拓展完毕时，我们认为它是**无法被拓展的节点**。

3. 模拟（*Simulation*）

模拟从选定的叶节点出发，开始快速随机落子，直到棋局出现胜者或者已经形成平局局面，模拟过程即告结束。借助位棋盘加速，这一过程极为迅速，可以在短时间内大量进行。

模拟步骤是蒙特卡洛树算法的“评估手段”的核心，它提供了在复杂、未知状态下评估节点价值的方法。在棋盘局面当中，不存在完美的估值函数来评判一个选择的收益，而模拟提供了这样一种基于统计的，近似的价值评估。对于一些未充分探索的分支，它也是“抽样调查”的唯一方式。相对于传统的穷举搜索，它的计算量更小，使得蒙特卡洛树算法在短时间内大量迭代成为可能。

4. 反向传播（*Backpropagation*）

反向传播是对模拟结果的传递，它基于模拟结果，沿叶节点到根节点的路径反溯，更新沿途所有节点的数据（该节点的访问量以及胜率）。

反向传播实现了模拟结果的应用，通过更新节点数据，为下一次选择提供依据。此外节点所记录的胜率将更加贴近实际情况，这为最终决策的合理性提供了基石。

接下来的内容将介绍上述理论是如何付诸实践的。

7 选择

选择过程依靠 *Select* 函数来实现。该函数主要包含一个对搜索范围的预处理过程和一个 *while* 循环。

7.1 搜索范围的确认

为了实现更加贴近当前局面的决策，子节点的选取必然要在以对方落子为中心的一定范围内进行，在一般情况下，这种处理是比较务实的。

搜索半径被设定为 *range* 它是一个与游戏进行轮次相关的量。程序通过人为设定它的初始值，经过通盘考虑，我们将它设置为 2 。考虑到随着游戏进行到后期，棋盘将逐渐变得拥挤，过小的搜索范围很有可能使得搜索树难以展开或是能够考虑到的落点有限，不足以得到一个更优的解。因此我们设置梯度叠加机制，每当 *round* 增加到某个梯度时，就增加 *range* 的大小。

```
1    int range=2;
2    if(round>20) select_range+=2;
3    if(round>34) select_range+=1;
4    if(round>54) select_range+=1;
5    if(round>74) select_range+=1;
```

7.2 循环体

7.2.1 循环条件

按照选择过程的逻辑，*Select*函数需要返回一个可供模拟的局面，因此循环体的任务就是找到这样一个节点，它或许是一个叶节点用于进行模拟或者达到终局进行反向传播（仍然要经过模拟）。所以循环体的进行条件就是棋盘局面还没有终局，如果找到了符合条件的节点，程序将在循环体内部返回，结束循环。

```
    while(check_winner(board)==Player::None&&!is_terminal(board))
```

7.2.2 搜索范围的计算

在循环体之前，我们确认了选择子节点的范围，接下来需要确认搜索空间。随着循环体沿搜索树向下，棋盘的搜索中心在不断改变，因此每次循环都需要重新计算搜索空间，即得到以当下父节点为搜索中心的搜索范围的四角坐标。

使用 $std::pair<int,int>$ $center$ 的两个成员记录搜索中心的坐标，对搜索范围的四角坐标进行计算。使用 $room$ 记录搜索空间里的容量，方便后续对能否继续拓展的判定。

```
    int x1=std::max(0,center.first-range);
    int x2=std::min(BOARD_ROWS-1,center.first+range);
    int y1=std::max(0,center.second-range);
    int y2=std::min(BOARD_COLS-1,center.second+range);
    int room=(x2-x1+1)*(y2-y1+1);
```

7.2.3 分支之一

前文提到，循环体最终要得到两种局面之一，两个分支就是对两种局面的处理。

第一个分支将判断当下棋盘是否仍然可以拓展子节点，通过将搜索范围内的棋子数与当下棋盘局面的子节点之和与搜索空间容量进行比较，如若前者小于后者，说明搜索空间内仍然存在着空白位置可供拓展。反之则说明无法进行拓展，程序将进入另一分支。

```
    if(count_piece(board,x1,x2,y1,y2)+statemap[board].children.size()<room)
```

由于选定节点后将进行模拟，程序需要知道子节点对应落子的棋子颜色，因此做视角判定处理，最后将拓展的棋盘局面和对应视角返回。

```
    if(count_piece(board,x1,x2,y1,y2)+statemap[board].children.size()<room){
    Player p=(player==Player::Black)? Player::White:Player::Black;
    return std::make_pair(expand(board,player,x1,x2,y1,y2),p);
}
```

7.2.4 分支之二

如若当下节点无法再拓展新的节点，就可能需要沿搜索树继续下行。

在分支二的最开始，我们设置了一个判断：如果当下节点没有子节点，那么将跳出循环。不进入分支一确保了当下节点不能再拓展，这包含两种可能的情况：

- **1.** 达到一个叶节点。
- **2.** 达到一个已经拓展完毕的父节点。

对于第一种情况，我们可以确信需要从此开始模拟，因而不应进行接下来的操作，直接退出循环体。

```
    if(statemap[board].children.empty()){
        break;
```

对于第二种情况，则需要继续沿搜索树向下走。此时需要选择一个 UCB 值表现最好的节点向下探索。此处设置一个临时容器 $temp$ 来储存棋盘局面（父节点），将最佳子节点直接赋给 $board$ ，

循环结束后更新父节点，确定传播链，保证反向传播能够按搜索路径进行，随即确定视角。

通过一个简单的遍历比较得到最佳子节点：

```
    ChessBoard temp=board;
    double max_ucb=-1e10;
    for(const auto& child : statemap[board].children){
        double child_ucb=UCB(child,player);
        if(child_ucb>max_ucb){
            max_ucb=child_ucb;
            board=child;
        }
    }
    parentmap[board]=temp;
    player=(player==Player::Black)? Player::White:Player::Black;
```

循环体以更新搜索中心结束。

7.3 尾声

函数以确定当前视角结尾，将其与确定的需要模拟的节点一同返回，函数完整代码如下：

```
std::pair<ChessBoard,Player> GomokuGame::Select(ChessBoard board,Player player,
    std::pair<int,int> center){
    while(check_winner(board)==Player::None&&!is_terminal(board)){

        int x1=std::max(0,center.first-select_range);
        int x2=std::min(BOARD_ROWS-1,center.first+select_range);
        int y1=std::max(0,center.second-select_range);
        int y2=std::min(BOARD_COLS-1,center.second+select_range);
        int room=(x2-x1+1)*(y2-y1+1);
        if(count_piece(board,x1,x2,y1,y2)+statemap[board].children.size()<room){
            Player p=(player==Player::Black)? Player::White:Player::Black;
            return std::make_pair(expand(board,player,center),p);
        }
        else{
            if(statemap[board].children.empty()){
                break;
            }
            ChessBoard temp=board;
            double max_ucb=-1e10;
            for(const auto& child : statemap[board].children){
                double child_ucb=UCB(child,player);
                if(child_ucb>max_ucb){
                    max_ucb=child_ucb;
                    board=child;
                }
            }
            parentmap[board]=temp;
            player=(player==Player::Black)? Player::White:Player::Black;
```

```
        }
        center=cal_center(board);
    }
    Player p=(player==Player::Black)? Player::White:Player::Black;
    return std::make_pair(board,p);
}
```

8 拓展

对于需要进行拓展的节点，使用 *expand* 函数进行拓展操作。

首先需要进行拓展范围的确认，此过程在选择步骤中已经进行，此处通过传入拓展范围的四角坐标得到。

expand 函数的核心同样是循环体，这段循环体将遍历拓展范围内的所有位置，每当发现空位时，它将在此处落子，此时将进行判定，如果能在 *statemap* 当中找到这一局面，说明此局面已被拓展过，不应在此处落子，这一落子将通过同一位置再次赋空值撤销。

如果无法找到这一局面，则说明这一落子是一步成功的拓展，程序将初始化这一节点，并更新两个 *unordered_map* 后直接返回这一节点。

```
    for(int i=x1;i<=x2;i++){
        for(int j=y1;j<=y2;j++){
            if(temp.grid[i][j]==Player::None){
                temp.grid[i][j]=player;
                if(statemap.find(temp)==statemap.end()){
                    init_ChessBoard_state(temp);
                    statemap[board].children.push_back(temp);
                    parentmap[temp]=board;
                    return temp;
                }
                temp.grid[i][j]=Player::None;
            }
        }
    }
```

如果在循环之后函数没有返回，说明父节点没有可供拓展的子节点（虽然在 *Select* 中确保了传入 *expand* 的节点是可拓展的，但是为了程序安全，我们设置了这一防御性措施），此时函数将判定该节点是否有子节点，如果有，就从子节点中随机返回一个；如果没有，就将该节点自身返回。

完整函数如下：

```
ChessBoard GomokuGame::expand(ChessBoard board,Player player,int x1,int x2,int y1,int y2){
    ChessBoard temp=board;
    for(int i=x1;i<=x2;i++){
        for(int j=y1;j<=y2;j++){
            if(temp.grid[i][j]==Player::None){
                temp.grid[i][j]=player;
                if(statemap.find(temp)==statemap.end()){
                    init_ChessBoard_state(temp);
                    statemap[board].children.push_back(temp);
                    parentmap[temp]=board;
                    return temp;
                }
                temp.grid[i][j]=Player::None;
            }
        }
```

```
    }
    if(!statemap[board].children.empty()){
        return statemap[board].children[rand()%statemap[board].children.size()];
    }
    return board;
}
```

9 模拟

虽然在前文“模拟”过程被描述为快速随机落子，但是实际上我们并没有采用完全的随机（那样的话就太蠢了），相反，我们的每一次落子都要经过一定的考量，这主要是考虑落子应当在中心范围还是全局范围进行。这体现了“启发式随机（*Heuristic Randomness*）”的思想：它防止了 AI 在开局时傻乎乎地去下棋盘边缘（纯随机的坏处），同时也防止了 AI 在中后盘死盯着中心不放而错失外围杀机（纯规则的坏处）。

9.1 准备工作

为了实现我们中心策略与全局策略的共同愿景，我们需要知道我们规定的中心区域的范围内有哪些可供落子的位置以及全局范围内有哪些可供落子的位置。

前文描述过如何计算当前局面的中心以及搜索范围的四角坐标，此处不再赘述。总之，我们完成了这样的工作：

- 利用 *place_piece* 将传入节点的黑白位棋盘进行数据初始化。
- 计算了传入节点的中心位置以及中心范围的四角坐标。
- 建立了两个储存中心区域和全局区域可供落子的位置坐标的变长数组。
- 将所有满足条件的位置储存在这两个变长数组中。

简单描述一下存入这些空位的过程。以中心区域为例，通过双层循环遍历中心区域内的所有位置，只要遇到空位，就将空位坐标压入变长数组 *center_round* 当中。

部分代码如下：

```
    std::vector<std::pair<int,int>> center_round;
    std::vector<std::pair<int,int>> whole_board;

    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]==Player::None){
                if(i>=x1&&i<=x2&&j>=y1&&j<=y2){
                    center_round.push_back({i,j});
                }
                whole_board.push_back({i,j});
            }
        }
    }
```

9.2 循环体

整个模拟过程的核心是一个循环体，它的运行条件被设置为 *true* 这意味着循环体只能在内部终止。

在循环体的最开始，函数将检查游戏是否结束，根据结果决定是否结束循环，这也是跳出循环的唯一途径。函数引入迭代器 *it2* ，它专门用于操作全局区域（*whole_board*）内的元素，随机获取变长数组 *whole_broad* 中某元素的地址。随后定义变量 *coord* 用于记录落子位置的坐标。

接着会有一个检查环节，由于循环体可能循环多次，导致在选择在中心区域落子时全局区域内对应位置没有弹出变长数组，因此需要借助循环体检查随机出来的位置是否已有棋子。若是，则弹出该位置并随机选择下一个位置；反之则结束循环。若循环过程中检查到全局区域内以无子可下（变长数组为空），则将立刻跳出循环。

```
std::pair<int,int> coord=*it2;
while(board.grid[coord.first][coord.second]!=Player::None){
    std::swap(*it2,whole_board.back());
    whole_board.pop_back();
    if(whole_board.empty()) break;
    it2=whole_board.begin()+(rand()%whole_board.size());
    coord=*it2;
}
```

这一循环体同样有两条分支，对应中心区域是否为空的两种情况，实际上，分支之二可以看作分支之一的特殊情况，由于分支之一会动态考虑在中心区域或全局区域落子，分支之二相当于分支之一当中只考虑在全局区域落子的极端情况，因此这一部分的代码完全由分支之一移植而来，我们以分支之一作为讲述的重点。

在分支之一当中，函数引入迭代器 *it1* ，它专门用于操作中心区域内的元素，随机获取变长数组 *center_round* 中某元素的地址。

分支之一同样有两个分支，它们将决定落子将在哪个变长数组当中选择。选择的逻辑在于：随机选择 $[1,100]$中的整数与局部区域内现存可落子位置总数的 5 倍比较。若中心区域仍有较大的落子空间，这一结果会倾向于前者小于后者，它要求程序在中心区域内可落子位置较多时应当更多地考虑在中心区域落子。而当中心区域比较拥挤时，比较结果将倾向于前者大于后者，它要求程序不应当局限于机会有限的中心区域，而应当将目光放在更加广阔的全局区域。

在中心区域落子时，首先函数将 *it1* 解引用，并将该位置赋给 *coord* 接着判定随机出来的位置是否为空，若为不为空，将此位置删除，并重新循环；反之直接将此位置从变长数组中删除。

```
    if(rand()%100<center_round.size()*5){
        coord=*it1;
        std::swap(*it1,center_round.back());
        center_round.pop_back();
    }
```

在全局区域里落子时，由于之前已经将 *it2* 的解引用赋给 *coord* 且检查过这一位置为空，是一个合法位置，于是直接将其从表示全局区域的变长数组删去。

```
else{
    std::swap(*it2,whole_board.back());
    whole_board.pop_back();
}
```

选定落子位置之后，函数将更新棋盘与位棋盘，并做转换视角处理。完成这几项工作，一轮循环结束。

9.3 棋局终止以后

循环体结束时，说明一局棋盘就已经终止了，此时只需要返回胜负情况，函数体就结束了。

整个函数的代码如下：

```
double GomokuGame::simulation_method(ChessBoard board,Player player){
    int range=select_range;
    int pieces=count_piece(board,0,BOARD_ROWS-1,0,BOARD_COLS-1);
    if(pieces>20) range+=2;
    if(pieces>34) range+=1;
    if(pieces>54) range+=1;
    if(pieces>74) range+=1;
    BitBoard b_black={},b_white={};
    place_piece(board,b_black,b_white,diag_map);
    std::pair<int,int> center=cal_center(board,b_black,b_white);

    int x1=std::max(0,center.first-range);
    int x2=std::min(BOARD_ROWS-1,center.first+range);
    int y1=std::max(0,center.second-range);
    int y2=std::min(BOARD_COLS-1,center.second+range);

    std::vector<std::pair<int,int>> center_round;
    std::vector<std::pair<int,int>> whole_board;

    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]==Player::None){
                if(i>=x1&&i<=x2&&j>=y1&&j<=y2){
                    center_round.push_back({i,j});
                }
                whole_board.push_back({i,j});
            }
        }
    }

    while(true){
        if(check_winner(board,b_black,b_white)!=Player::None||whole_board.empty()) break;

        auto it2=whole_board.begin()+(rand()%whole_board.size());
        std::pair<int,int> coord=*it2;
        while(board.grid[coord.first][coord.second]!=Player::None){
            std::swap(*it2,whole_board.back());
            whole_board.pop_back();
            if(whole_board.empty()) break;
            it2=whole_board.begin()+(rand()%whole_board.size());
            coord=*it2;
        }
        if(whole_board.empty()) break;

        if(!center_round.empty()){
            auto it1=center_round.begin()+(rand()%center_round.size());
            if(rand()%100<center_round.size()*5){
                coord=*it1;
                if(board.grid[coord.first][coord.second]!=Player::None){
                    std::swap(*it1,center_round.back());
                    center_round.pop_back();
                    continue;
                }
```

```
                std::swap(*it1,center_round.back());
                center_round.pop_back();
            }
            else{
                std::swap(*it2,whole_board.back());
                whole_board.pop_back();
            }
        }
        else{
            std::swap(*it2,whole_board.back());
            whole_board.pop_back();
        }
        board.grid[coord.first][coord.second]=player;
        place_a_piece(b_black,b_white,diag_map,coord.first,coord.second,player);
        player = (player == Player::Black ? Player::White : Player::Black);
    }

    if(check_winner(board,b_black,b_white)==Player::None) return 0.0;
    else if(check_winner(board,b_black,b_white)==Player::Black) return 1.0;
    else return -1.0;
}
```

10 反向传播

通过 *back_up* 函数完成整条传播链的所有节点的数据更新。

此函数的逻辑比较简单，只需要通过一个循环反溯，依据 *parentmap* 当中确定的传播链，沿途更新所有节点的胜率与访问次数即可。

代码如下：

```
void GomokuGame::back_up(ChessBoard current,const ChessBoard& root,double value){//反向传播
    statemap[current].win+=value;
    statemap[current].visit++;
    while(!(current==root)){
        if(parentmap.find(current)==parentmap.end()) break;
        current=parentmap[current];
        statemap[current].win+=value;
        statemap[current].visit++;
    }
}
```

第三部分　AI程序的完整决策过程

11　启发式落子

尽管我们为AI程序设计了一套行之有效的算法，让其具有了对弈的基本能力，但是应该看到的是，面对一些特殊局面，比如冲四这种致命的威胁，AI可能无法一定做出防守反应，而仅仅只是“很可能”防守。因此，我们需要进行干预，使得AI能够对特定局面做出更加合理的反应。并且，当AI方自己达成这些局面时，可以及时落子，以提高获胜概率。

11.1　检查致胜点（$check\_four$）

当局面出现冲四时（无论是AI方还是玩家方），此时都应该立刻做出回应，从而取得胜利或者抵御致命威胁。“冲四”局面在我们的程序当中被阐释为“致胜点”，也即：只需落一子就能使得一方直接获胜的局面。

我们采取遍历整个棋盘的方式来检查是否有这样的位置，使得下在这里可以使得一方立刻获胜。

函数采用双重循环遍历整个棋盘，每次进行这样的操作：

- 在(i,j)处落子。
- 检查是否有胜者。若有，则直接返回 $true$ 以及该位置。反之则继续。
- 擦除该位置的落子。若经过完整循环以后依然没有返回，说明不存在这样的位置，函数将返回 $false$ 。

代码如下：

```
std::pair<bool,std::pair<int,int>> GomokuGame::check_four(const ChessBoard& board,Player player){
    BitBoard b_black,b_white;
    place_piece(board,b_black,b_white,diag_map);
    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]==Player::None){
                place_a_piece(b_black,b_white,diag_map,i,j,player);
                if(check_winner(board,b_black,b_white)==player){
                    return {true,{i,j}};
                }
                erase_a_piece(b_black,b_white,diag_map,i,j,player);
            }
        }
    }
    return {false,{0,0}};
}
```

11.2　检查三子威胁（$check\_three$）

我们将三子威胁定义为“如果不受干预，再下两子就能直接胜利”的局面，例如活三。这种情况同样需要警惕，如果不及时干预，它很有可能立刻演变为绝杀的威胁。

检查方法的逻辑与 $check\_four$ 一脉相承，均要求先下一子，再查证是否满足判定条件，只是这里需要依赖更多层的循环。函数涉及两个视角不同的大循环体，其基本结构一致，接下来仅以检查敌方（$opponent$）是否构成三子威胁局面为例进行说明。

为了节省算力，我们不进行完整的遍历。搜索的逻辑为：先下一子，再调用 $check\_four$ 检查是否在落子附近任何合法位置落子都有冲四威胁。若是，则说明原始棋局存在三子威胁。反之，则说明没有。

循环体首先遍历整个棋盘，在空白（i,j）处落子。随后定义 $bool$ 型变量 $flag$ ，用于记录需要的是否判定

```
    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]!=Player::None) continue;
            board.grid[i][j]=player;
            bool flag=true;
    ......
```

随即计算搜索范围，搜索范围中，行被限定在 $[i-4,i+4]$，如遇棋盘边界，则以边界作为上（下）限。列范围同理。限定搜索范围可以大大减少需要搜索的位置数，并且满足了我们针对位置（i,j）进行搜索的需求。

遍历搜索范围时，每当发现某一位置不会使得局面形成冲四，就要将 $flag$ 设置为 $false$ ，表示此当下局面并非完全无法规避风险。若局部搜索结束以后 $flag$ 仍为 $true$ ，则说明无论怎么下棋，都会使对方达到“冲四”而获胜，进而说明（i,j）是对方的致胜一步，需要提前予以干预。

代码如下：

```
    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]!=Player::None) continue;
            board.grid[i][j]=opponent;
            bool flag=true;
            for(int k1=std::max(0,i-4);k1<=std::min(BOARD_ROWS-1,i+4);k1++){
                for(int k2=std::max(0,j-4);k2<=std::min(BOARD_COLS-1,j+4);k2++){
                    if(board.grid[k1][k2]!=Player::None) continue;
                    board.grid[k1][k2]=player;
                    if(check_four(board,opponent).first==false) flag=false;
                    board.grid[k1][k2]=Player::None;
                }
            }
            if(flag==true){
                return {true,{i,j}};
            }
            board.grid[i][j]=Player::None;
        }
    }
```

最后，如果循环体都没有返回，则说明不存在三子威胁，返回 $false$ 。

完整代码如下：

```


std::pair<bool,std::pair<int,int>> GomokuGame::check_three(ChessBoard board,Player player){
    Player opponent=(player==Player::Black)? Player::White:Player::Black;
    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]!=Player::None) continue;
            board.grid[i][j]=player;
            bool flag=true;
            for(int k1=std::max(0,i-4);k1<=std::min(BOARD_ROWS-1,i+4);k1++){
                for(int k2=std::max(0,j-4);k2<=std::min(BOARD_COLS-1,j+4);k2++){
```

```
                    if(board.grid[k1][k2]!=Player::None) continue;
                    board.grid[k1][k2]=opponent;
                    if(check_four(board,player).first==false) flag=false;
                    board.grid[k1][k2]=Player::None;
                }
            }
            if(flag==true){
                return {true,{i,j}};
            }
            board.grid[i][j]=Player::None;
        }
    }
    return {false,{0,0}};
}
```

11.3 检查双重威胁（*check_double_thread*）

很多时候，五子棋的胜负都来自于一方构造的“四三”、“三三”等杀招，如果只采用上述两种方法，想要不落入对方的陷阱是做不到的。因此，让AI能够识别这些棋型是至关重要的。

11.3.1 威胁情况分析（*threads*）

对于棋盘上的某一位置，我们需要计算它对黑白双方的威胁度，当综合考量后得到的威胁度达到一定阈值，就势必要对这一位点进行干预。

我们利用 *threads*函数计算威胁度。

我们考虑到了六种具有威胁的情景，下面以第一种为例进行说明。

分别引入 *w_threads* 和 *b_threads* 表示这一位置对由白（黑）方下将给对方带来的威胁。

对于局面 OXXOO （“X”表示此处有同色落子，“O”表示此处为空），使用 *threads*函数时，函数传入四个参数（$p1, p2, p3, p4$）。它们以要检查的棋子为中心，沿同一条直线偏移。例如，在示例 OXXOO 中，从左到右第四个位置就表示要计算威胁的空位，$p1, p2, p3$ 分别是空位向某一方向延伸一个、两个、三个格边长（或对角线）的棋子； $p4$是空位沿反方向延伸一个格边长（或对角线）的棋子。OXXOO局面相当于 $p3$ $p2$ $p1$ 空 $p4$ 若满足此条件，程序将给中心位置 $p1$ 对应的 *threads* 增加 1。

```
if((p1==p2)&&(p3==p4)&&(p3==Player::None)){
    if(p1==Player::Black) b_threads++;
    else if(p1==Player::White) w_threads++;
}   //OXXOO
```

函数完整代码如下：

```
void threads(const ChessBoard& board,int i,int j,std::pair<int,int> dr_dc,double& b_threads,double
    & w_threads){
    int dr=dr_dc.first,dc=dr_dc.second;
    Player p1=board.grid[i+dr][j+dc],p2=board.grid[i+dr*2][j+dc*2],
        p3=board.grid[i+dr*3][j+dc*3],p4=board.grid[i-dr][j-dc];

    if((p1==p2)&&(p3==p4)&&(p3==Player::None)){
        if(p1==Player::Black) b_threads++;
        else if(p1==Player::White) w_threads++;
    }   //OXXOO

    if((p1==p2)&&(p2==p3)&&(p4==Player::None)){
        if(p1==Player::Black) b_threads++;
```

```
        else if(p1==Player::White) w_threads++;
    }   //XXXOO

    if((p1==p4)&&(p2==p3)&&(p1==Player::None)){
        if(p2==Player::Black) b_threads++;
        else if(p2==Player::White) w_threads++;
    }   //XXOOO

    if((p1==p2)&&(p2==p4)&&p3==Player::None){
        if(p1==Player::Black) b_threads++;
        else if(p1==Player::White) w_threads++;
    }   //OXXOX
    if((p1==p3)&&(p3==p4)&&(p2==Player::None)){
        if(p1==Player::Black) b_threads++;
        else if(p1==Player::White) w_threads++;
    }   //XOXOX
    if((p1==p4)&&(p2==Player::None)){
        if(p1==Player::Black) b_threads+=0.5;
        else if(p1==Player::White) w_threads+=0.5;
    }   //?OXOX  由于对称，另一侧还会再算一遍，所以 +0.5
}
```

11.3.2 依据威胁度决定是否要干预

*check_double_threads*函数的实现逻辑很简单。考察某一位置时，对于上述任意一种可能带来威胁的棋型，都可能发生在以它为中心的八个方向上。因此只需按八个方向分别考察威胁，最后将双方威胁分别累加，一旦达到某一阈值（这里设置为 2 ，表示面临至少两个威胁），就需要返回这一位置以及时做出干预。

代码如下：

```
std::pair<int,int> GomokuGame::check_double_thread(const ChessBoard& board){
    double b_threads=0,w_threads=0;
    for(int i=0;i<BOARD_ROWS;i++){
        for(int j=0;j<BOARD_COLS;j++){
            if(board.grid[i][j]!=Player::None) continue;

            if(j-3>=0&&j+2<BOARD_COLS){
                threads(board,i,j,{0,-1},b_threads,w_threads);          //向左延伸
                if(i-3>=0&&i+2<BOARD_ROWS){
                    threads(board,i,j,{-1,-1},b_threads,w_threads);     //主对角延伸
                }
                if(i+3<BOARD_ROWS&&i-2>=0){
                    threads(board,i,j,{1,-1},b_threads,w_threads);      //副对角延伸
                }
            }

            if(j+3<BOARD_COLS&&j-2>=0){
                threads(board,i,j,{0,1},b_threads,w_threads);           //向右延伸
                if(i-3>=0&&i+2<BOARD_ROWS){
                    threads(board,i,j,{-1,1},b_threads,w_threads);      //副对角延伸
                }
                if(i+3<BOARD_ROWS&&i-2>=0){
                    threads(board,i,j,{1,1},b_threads,w_threads);       //主对角延伸
                }
            }

```

```
            if(i-3>=0&&i+2<BOARD_ROWS){
                threads(board,i,j,{-1,0},b_threads,w_threads);          //向上延伸
            }
            if(i+3<BOARD_ROWS&&i-2>=0){
                threads(board,i,j,{1,0},b_threads,w_threads);           //向下延伸
            }

            if(std::round(b_threads)>=2||std::round(w_threads)>=2) return {i,j};
            b_threads=0,w_threads=0;
        }
    }
    return {-1,-1};
}
```

12 AI决策的控制函数（*uctSearch*）

这个函数是统筹前文所讲述的所有算法的函数，它通过调用这些函数，得到一个 *bestmove*并返回。这一函数不含有值得一提的算法，因此我们只对它调用函数的过程进行梳理。

完整流程如下：

- 通过启发式落子判断是否有必须处理的位置。
- 若 *statemap* 当中没有当前棋盘，将其传入。
- 指定蒙特卡洛树算法的搜索轮数。
- 经过一定轮数的循环得到充分展开的蒙特卡洛搜索树。
- 依据根节点的子节点的访问次数确定最佳位置。

```
ChessBoard GomokuGame::uctSearch(const ChessBoard& board,Player player,std::pair<int,int> center){

    ChessBoard bestmove=board;

    Player opponent=(player==Player::Black)? Player::White:Player::Black;
    std::pair<int,int> coord={-1,-1};
    if(round>=8){
        std::pair<bool,std::pair<int,int>> temp1=check_four(board,player);
        if(temp1.first){
            coord=temp1.second;
        }
        else{
            std::pair<bool,std::pair<int,int>> temp2=check_four(board,opponent);
            if(temp2.first){
                coord=temp2.second;
            }
        }
        if(coord.first!=-1){
            bestmove.grid[coord.first][coord.second]=player;
            return bestmove;
        }
    }

    if(round>=6){
        std::pair<bool,std::pair<int,int>> temp1=check_three(board,player);
```

```
        if(temp1.first){
            coord=temp1.second;
        }
        else{
            std::pair<bool,std::pair<int,int>> temp2=check_three(board,opponent);
            if(temp2.first){
                coord=temp2.second;
            }
        }

        if(coord.first!=-1){
            bestmove.grid[coord.first][coord.second]=player;
            return bestmove;
        }
    }

    if(round>=8){
        coord=check_double_thread(board);
        if(coord.first!=-1){
            bestmove.grid[coord.first][coord.second]=player;
            return bestmove;
        }
    }


    if(statemap.find(board)==statemap.end()){
        init_ChessBoard_state(board);
    }

    int cnt=SELECT_NUM;

    while(cnt--){
        std::pair<ChessBoard,Player> select_node=Select(board,player,center);
        for(int i=0;i<SIMULATION_NUM;i++){
            double value=simulation_method(select_node.first,select_node.second);
            back_up(select_node.first,board,value);
        }
    }

    if(!statemap[board].children.empty()){
        bestmove=statemap[board].children.front();
        for(const auto& child : statemap[board].children){
            if(statemap[bestmove].visit<=statemap[child].visit){
                bestmove=child;
            }
        }
    }

    return bestmove;
}
```

第四部分　（蒙特卡洛）五子棋AI制作过程当中的回顾与反思

本学期数据与结构课程的大作业——制作五子棋AI让我获益颇丰，也激发了我在AI领域的热情与兴趣，作为第一个接触的工程项目，它为我以后的学习与开发带来了宝贵的经验，在这个过程中，我遵循自顶向下的逻辑思维，从整体框架到工程实现细节再到优化算法以及重构代码，一步步完成了这个极具挑战性的任务，以下是我在工程实践当中的探索：

一、算法细节的不断精准把握

1. 在最初学习蒙特卡洛算法的时候，我错将“最具探索潜力的节点”理解为“下一步的最佳节点”。在算法当中，“最具探索潜力的节点”是根据UCB值的比较而得出的。它是选择阶段的一个判定指标，使得MCT树在向下搜索时可以平衡“价值”和“探索”两项指标。

UCB的应用本应只在探索阶段，我却在选取 $bestmove$ 的时候仍以UCB值作为指标。有些节点只探索了寥寥几次但胜率意外地高，这可能会导致算法选择这样的节点，但由于 $rollout$ 的随机性，这样的“高胜率”并不禁得住考验，所以在最初的对局当中，AI常常会下“坏棋”。经过重新审查算法细节，我将 $bestmove$ 的选取标准改为了节点的访问数，访问数最多的节点在算法眼中就是最具价值的节点，它的结果也往往更具鲁棒性。此处细节更改之后，AI下“坏棋”的现象显著减少。

2. AI的棋力直接取决于选择次数与模拟次数，选取适当的参数可以使AI的搜索能力最大化。在制作初期，我将选择次数单纯地看成一个循环终止的参数，认为模拟次数相比选择次数更为重要，一个节点模拟的次数越多，它得到的结果就越接近真实值。后来通过查询资料了解到，选择次数不单纯是循环终止的参数，每次选择都有可能在叶子节点处向下扩展新的节点，从而产生新的节点，所以选择次数还决定了MCT树的搜索深度。

此外，由于选择和扩展所得到的节点的不确定性，有时存在对“下了一步坏棋”的局面进行模拟，如果将选择次数设置过高，在此种情形下就会浪费过多的算力，加上模拟阶段落子的不确定性，想获得一个接近真实的返回值可能需要上万次的模拟，为了平衡速度与准确性，我将模拟次数调为一个较小值，将选择次数调为一个较大值，使MCT树能够搜索到更多的局面信息。

二、算法优化的不断尝试

1．内存优化：

内存对于MCT树的构建至关重要，树的节点以指数级增长，如果不加管控，最终可能导致内存溢出而使游戏异常终止。事实证明，在早期对局阶段，AI在下至约 50 手棋之后就会出现异常退出的情况，在排查了代码当中的越位风险之后，最终得出结论：是内存溢出导致的异常退出。为了解决这个问题，我最开始是选择每次得到 $bestmove$ 之后就将根节点的 $children$ 全部删除，然而异常退出的情况仍然存在，只不过是得到了些许缓解，我意识到只删除子节点并不能根本性地解决问题，于是我选择实现MCT树的节点复用，每次落子之后都将不再需要的节点剪去，留下需要继续搜索的节点，功能完成以后，异常退出的情况被完美解决。

2．速度优化：

最初的 $check\_four$ 内部两个嵌套 for 循环加上内部调用 $check\_winner$ 来实现的，最初实现的 $check\_winner$ 是遍历整个棋盘， $check\_four$ 的实际复杂度为$O(n^4)$,而 $check\_three$ 内部由四个嵌套 for 循环加调用 $check\_four$ 来实现的，实际复杂度更是来到了惊人的 $O(n^8)$，这导致一次 $check\_four$ 耗时接近 4 秒，一次 $check\_three$ 耗时接近 20 秒，加上搜索过程中多次调用 $check\_winner$ 判断终止

条件，这导致AI落一子的时长达 40 秒以上，这极大地影响了对局体验，也浪费了很多算力。

为此我通过查询资料了解到位棋盘加速棋局类游戏的应用，于是编写了位棋盘及其一系列位运算判断，将 $check\_winner$ 的复杂度降至 $O(1)$。随后我将 $check\_three$ 每个位点的判断范围缩小至一个 $k \times k$ 的矩形之中，成功将 $check\_three$ 优化至 $O(k^2 \times n^4)$，最终一次 $check\_four$ 耗时不足 0.1 秒，一次 $check\_three$ 耗时接近 1 秒，且极大地加速了搜索过程中终局的判定。

此外，最初是使用 map 来实现 $ChessBoard$ 和 $Stateproperty$ 的插入与键值查询，复杂度为 $O(\log n)$，为了加速这一过程，我选择使用 $unordered\_map$，为了实现自定义结构体的可哈希，我参考了 $FNV-1a$ 哈希算法，并最终成功实现了 $O(1)$ 的插入与查询，进一步优化了AI的运行速度。另外在模拟阶段，最初我采用完全的随机落子，通过随机数来产生坐标，在棋子数较多的局面下，往往要进行许多次循环才能恰好撞到那一个未落子点，为了解决这一问题，我在每次模拟前将所有的合法落子点都收集起来，并采用 $lazy\ delete$ 的方法，成功提高了模拟阶段的速度。

3.模拟优化：

$rollout$的质量直接决定了AI的模拟对局是否具有参考性，这也决定了模拟对局是否能给AI的自我强化学习带来正向影响。在MCT的模拟当中，速度快是其首要遵守原则，其次是考虑模拟的质量问题。所以在最初的 $rollout$ 实现过程中，我选择了最简单的随机落子，并采用“ ε 贪心”策略，在中心区域与非中心区域设置了简单的落子权重，让AI倾向于在局面的中心区域落。在AI制作的中后期，我希望可以优化 $rollout$的质量，又不影响AI的速度，所以引入轻量级的启发是最佳选择。

第一次试错是直接在每次模拟落子之前引入 $check\_four$ 进行落子判断，但很不幸的是由于大量的选择与模拟，即使 $check\_four$ 经过位运算优化，最终呈现的结果却是一次落子耗时近两分钟，显然，引入 $check\_four$ 并不是正确的做法。

第二次试错是引入评估函数，在参考了部分 $rollout$ 优化策略以后，我得知可以通过引入评估函数并提前终止对局返回评估分数来平衡模拟质量与速度，于是我编写了一个棋盘的评估函数，每次模拟落子前都会进行一次评估并选择分数最高的落子点，然后返回终止盘面的分数。完成以上工作后，通过与AI进行的不下十盘的对局当中，我发现了两个现象：

- AI的棋力并没有得到跨台阶式的提升。
- AI在面对我相同的落子策略时总会做出和以往同样的反应，即它的落子方式已经被完全定死了。

深入分析之后，我总结出三个原因：**其一**，评估函数对于棋形的打分具有人的主观判断性，很难做到对盘面的精准评价；**其二**，模拟结束返回的分数数值往往较大，这会使得UCB公式当中的利用项被放大许多倍，而探索项便基本被忽略；**其三**，由于评估函数打分的确定性以及模拟阶段每一步都进行评估的“完全贪心”策略，使得AI的落子方式被锁死，MCT的随机性被完全抹杀，成为了披着MCT外表的$\alpha-\beta$剪枝。

为了解决AI落子策略被锁死的问题，我将“完全贪心”改成了“ε 贪心”，让AI倾向于进行评估落子，又不抛弃随机落子，这一改动确实让AI的落子脱离了锁死的状态，然而引入评估函数的结果最终还是不尽人意，所以这一方案也被舍弃了。最终我还是选择了随机落子，只不过这次我将“ε 贪心”改为了“动态 ε 贪心”，ε 值会根据中心可落子数而动态调整，在中心落子数较少的情况下，贪心会自动退化为全局随机落子，这符合五子棋的对局常理，也略微优化了 $rollout$ 的质量。

在五子棋AI的制作过程中，不断地探索、试错、思考极大的锻炼了我的思维能力与工程实践能力，这不仅是一次简单的AI制作，而是一次收获颇丰的编程实践，我从中学习到的方法与思想将在以后的学习中给我带来无尽的效益。