电子科技大学

计算机科学与工程学院

标 准 实 验 报 告

实验名称 五子棋

姓名 林凯滔

学号 2023080903013

电子科技大学教务处制表

电 子 科 技 大 学

1. 引言

五子棋是一种古老而具有深厚文化底蕴的策略性棋类游戏，其简单的规则却蕴含了丰富的战术和战略。随着计算机科学的不断发展，人工智能在博弈领域的应用成为一个备受关注的研究方向。本实验旨在通过实现五子棋游戏，并运用 Alpha-Beta 剪枝算法实现智能决策，从而提高计算机对手在游戏中的水平。

Alpha-Beta 剪枝算法作为一种优化的搜索算法，通过剪除对游戏结果无影响的搜索路径，显著提高了搜索效率。其工作原理基于极小极大搜索，通过递归遍历博弈树来寻找最优解。相较于传统的搜索算法，Alpha-Beta 剪枝在搜索空间中引入了上下界，从而避免了冗余的搜索，使得在有限的时间内更准确地找到最优解。

在本实验中，我们将结合游戏框架的设计和 Alpha-Beta 剪枝算法的实现，构建一个智能的五子棋对手。通过对实验结果的深入分析，我们旨在探讨 Alpha-Beta 剪枝算法在五子棋博弈中的应用，深化对其工作原理和效果的理解。通过本次实验，我们希望为进一步探索人工智能在棋类游戏中的应用奠定基础，同时为未来在更广泛领域中的应用提供有益的经验和启示。

2. 理论基础

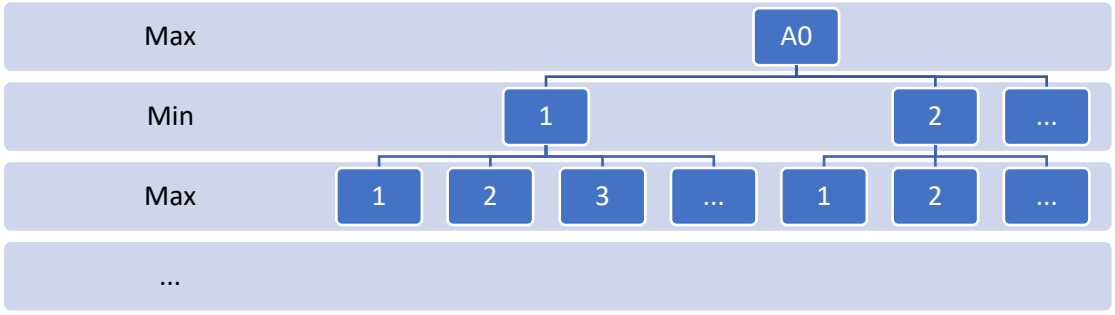
2.1 五子棋规则

五子棋一在 15*15 的棋盘上进行。游戏中有两名玩家，一方使用黑子，另一方使用白子。游戏的目标是在棋盘上形成一个连续的、由五颗自己棋子构成的直线，横向、纵向或斜向，即五子成线，即可获胜。
初始时棋盘为空，先由黑方下一棋，而后白方下一棋，如此循环直到一方胜利或平手。每一步，玩家选择一个空格落下自己的棋子，棋子落下后不能移动或更改。
五子棋的判胜规则为一色棋子先连成五子一线者为胜，连线可成横，纵或斜线，若棋盘下满且双方皆无法连成五子，则游戏宣告平局。

2.2 极大极小搜索算法及 alpha-beta 剪枝算法

2.2.1 极大极小搜索算法

假设下棋双方皆为完全理性，则双方下棋者皆会选择对己方价值最高的位置进行下子。同时，下棋者应考虑下在该位置后敌方所有可能的位置及相应局势的价值，因此对某一位置的价值判定应包含下在该点位后所有可能性，若以博弈树对此展开，则为



其中每一个节点代表一个可下的位置，一层为一次黑白换方。对某一方而言，其价值分析为：根节点层及双数层为己方层，应选择价值最高的点位，而在单数层则为敌方层，敌方将会选择对敌方价值最高，对己方价值最低的点位，如此构成一颗双数层搜索最大值，单数层搜索最小值的博弈树。
若希望程序能判定此树所代表的最佳下棋位置，我们可以先设置一个评估函数判定（对己方而言）某一局面的价值，同时设置一递归函数对所有可能的下棋位置进行穷举遍历，对对应层级进行最大和最小值搜索，即可判定最佳点位，是为极大极小搜索算法。

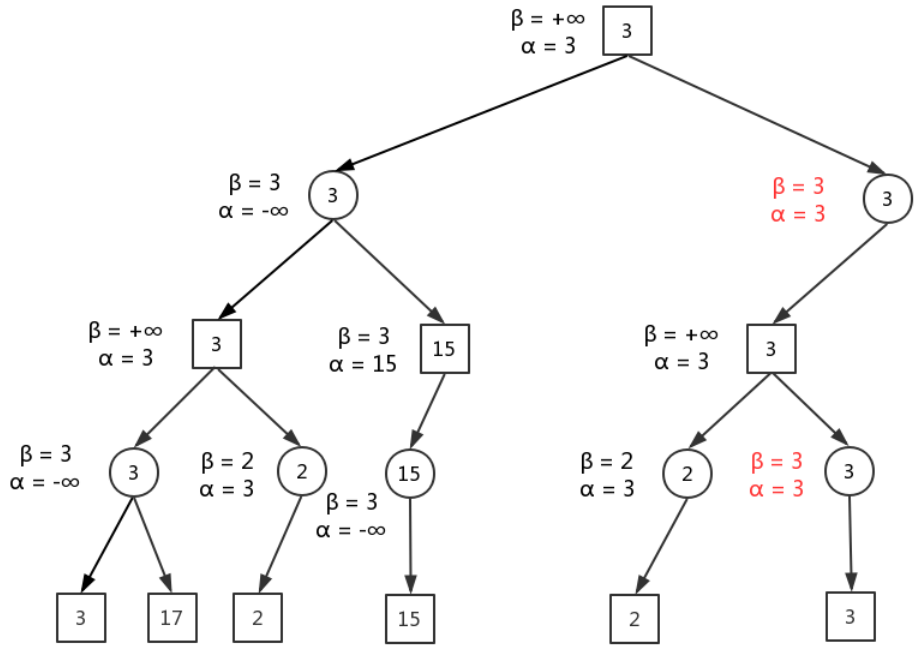
2.2.2 问题与 alpha-beta 剪枝算法

极大极小搜索算法虽然能顾全所有可能局面，但由于棋盘大小，若增加遍历层

数，其所需要遍历的节点数量将以指数级上涨，同时节点中必然包含大量无价值的棋位，对此类位置而言可以透过某种方法筛去而不消耗算力处理。基于此思想，我们可以使用 alpha-beta 剪枝算法。

2.2.3 核心思想

alpha-beta 剪枝算法的核心为在递归中传递的变量 alpha 和 beta，alpha 表示极大值层的最佳值，beta 表示极小值层的最佳值。当某一节点的值超出 alpha-beta 范围时可以剪枝，不再搜索其子节点，因为这部分子树不会影响最终结果。对应现实为：假设思考四步棋，若某一位置在第二步已经判定为对己方不利，则其后续步也大概率都同样时对己方不利，也就不再需要思考该位置第三，四步的价值，取其第二步时的判定即可。此行为即为对某节点进行剪枝。在实际运行途中，某一节点的价值由其子节点的极大或极小值决定。对极小值层而言，当发现一个节点的值比当前的 beta 值更小，即该节点对极小值层来说过于优越（对敌方更具优势），可以剪枝。同理，当发现一个节点的值比当前的 alpha 值更大，即该节点对极大值层来说过于优越，同样可以剪枝。

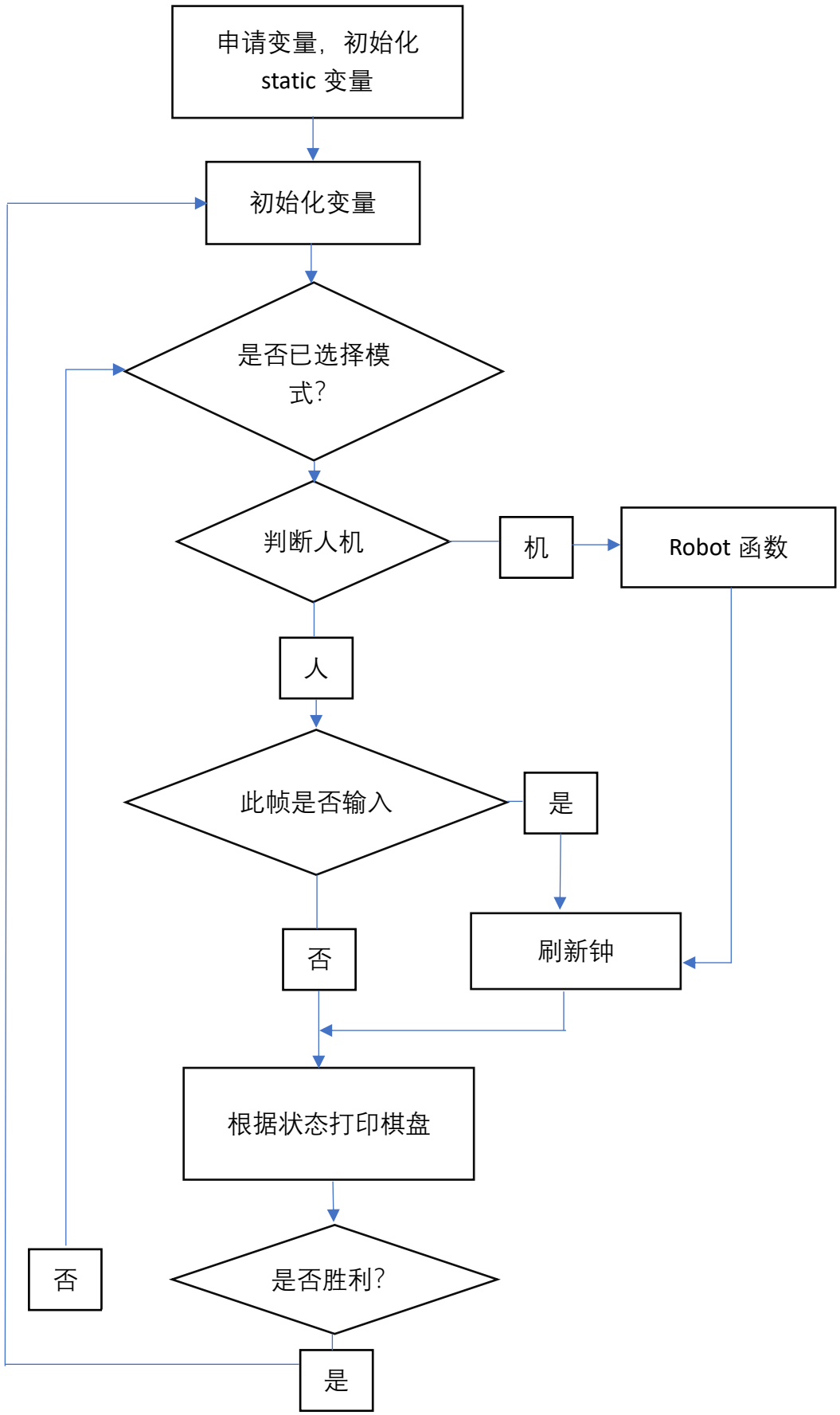


总体而言，Alpha-Beta 剪枝算法在相同搜索深度下，相比传统的极小极大搜索算法，具有更高的效率，能够更快地找到最优解。

3. 编程实现

3.1 五子棋平台

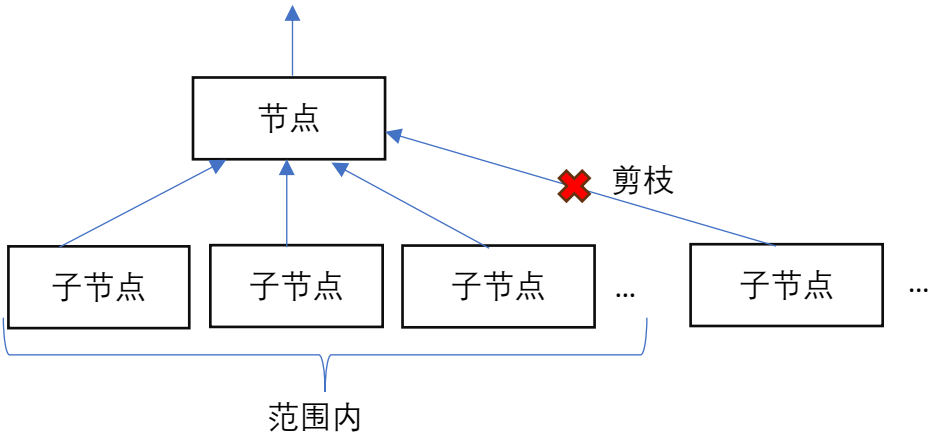
程序以 C 语言为基础，显示使用 raylib 库。游戏流程图如下：



其中棋盘以枚举表示黑，空及白，分别使用-1，0，1 表示，以便处理时可使用*-1 表示敌方棋子。处理人机关系的核心为一变量 relation，其值为 0（未选择），1 和-1。当需要判断时，记录现在出棋方的钟会透过此变量映射到枚举变量 Human_Robot_Turn，并分别执行各自的代码，以此实现选择人或机先手。代码将运算与渲染分开，代码运算时保留上次刷帧结果，将结果以枚举输出后再根据枚举状态渲染至棋盘，以此避免打印时因运算耗时而出现闪屏和打印不全的问题。

3.2 robot 函数 alpha-beta 剪枝算法

robot 函数为五子棋算法的入口函数，若出现空棋盘（上一个下棋位置为无效的100,100）则下在正中间，否则进入递归算法。递归算法如下：
为了提升计算速度，递归算法里下棋位置被限定在正负而中间。当算法运行至叶节点时执行 value 判定并返回至上一层，中间层节点根据下一层返回值进行极大-极小搜索，若为双数层则将目前已返回数值中的最大值赋予 alpha，若为单数层则将最小值赋予 beta。若出现超出界限外的值则打断遍历（剪枝），再根据层数返回 alpha 或 beta 值。若为根节点则在改变 alpha 值的同时刷新最佳下落点位（一个全局变量）。遍历完成后 robot 函数输出输出点位，函数终止。
单个递归函数逻辑如下：



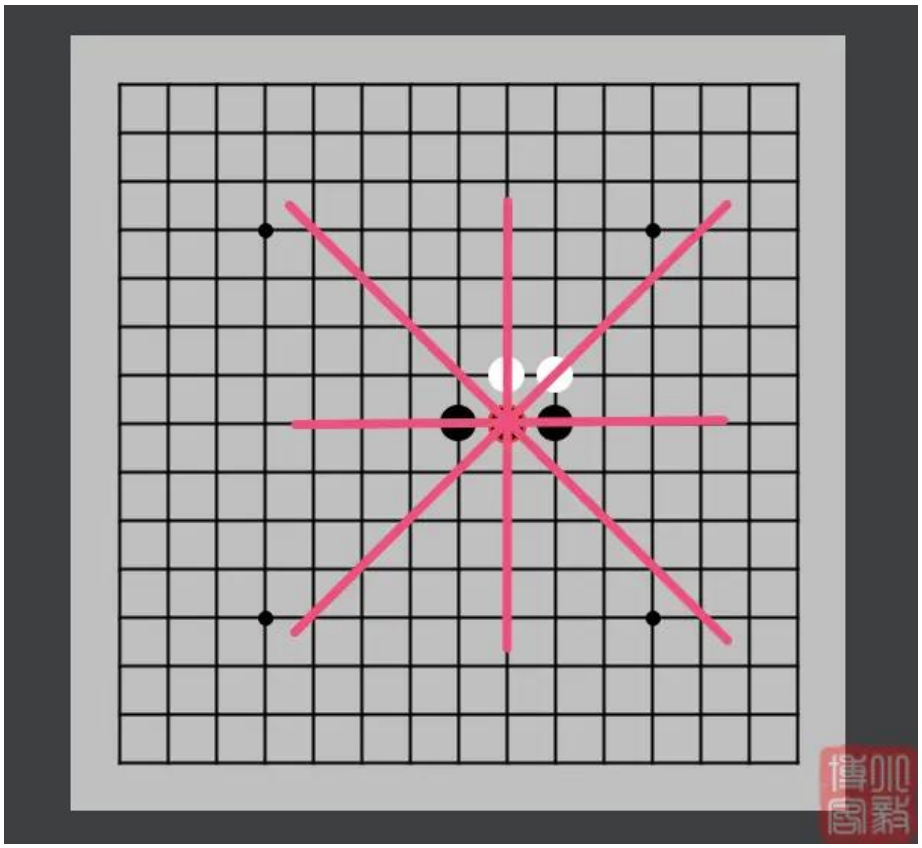
代码为：

```
int alpha_beta_search(enum chess Chessboard[15][15], enum chess
Friend_color, int depth, int alpha, int beta,Vector2 put_chess) {

    if (depth == 0) {
        return evaluate(Chessboard, Friend_color, put_chess);
    }
    bool root = (depth == MAX_DEPTH);
```

```
    for (int X = max(put_chess.x - BRANCH_SEARCH_RANGE, 0); X <=
put_chess.y + BRANCH_SEARCH_RANGE && in_boundary(X); X++) {
        for (int Y = max(put_chess.y - BRANCH_SEARCH_RANGE, 0); Y <=
put_chess.y + BRANCH_SEARCH_RANGE && in_boundary(Y); Y++) {
            if (Chessboard[X][Y] == empty) {
                Chessboard[X][Y] = Friend_color;
                int value = alpha_beta_search(Chessboard,
Friend_color, depth - 1, alpha, beta, (Vector2) {X, Y});

                Chessboard[X][Y] = empty;

                if (!(depth % 2)) {
                    if (value >= alpha) {
                        alpha = value;
                        if (root) {
                            printf("%d, %d, %d\n", X,Y, value);
                            global.put_chess.x = X;
                            global.put_chess.y = Y;
                        }
                    }
                } else {
                    if (value <= beta) {
                        beta = value;
                    }
                    if (alpha >= beta) {
                        break;
                    }
                }
            }

        }
    }
    return (!(depth % 2)) ? alpha : beta;
}
```

3.3 value 判定函数

对函数周边正负 2 的范围内，每一个棋位进行判定己方及敌方两次判定并累加和累减，以此得出最终得分。对每一个棋位，算法会从该位置 X,Y 往某一方向探索，同时进行计数，直到单个方向延申五格，碰到边界或遇到敌方棋子。在往反方向探索后，算法会根据该方向已延伸的步数依次录入 chain，以此构成一不大于 11 子长的棋链，其中首尾可能为己，敌或空，中间则为己或空。若棋链长度小于 6，此时连无法连成五子，可判定为无价值并跳过。若为大于等于 6 的棋链则会从第 0 位开始每六个子为一段判定棋形并累加。对每一棋链段，棋子的可能为首位为(空，己，敌)，其余为(己，空)，此链可以二进制表示为：
敌方 2，己方 1，空白 0，并以此得出一介于 0~(2^6 + 2^5-1)数值，可以此为哈希值直接输出对应的值进行累加，以此循环 4 次覆盖横，纵，左斜及右斜四个方向，共 8 条棋链，以此得出该棋位的价值。



代码如下：

```
struct chess_type hashTable[97]={
        {"empty", 0},
        {"21", 0},
        {"010", 1},
        {"211", 150},
        {"010", 1},
        {"1012", 250},
        {"0110", 650},
        {"2111", 500},
        {"010", 1},
        {"1001", 200},
        {"101", 400},
        {"10112", 800},
        {"0110",  650},
        {"11012", 800},
        {"01110", 1500},
        {"11112", 2500},
        {"010",1},
        {"10001", 100},
        {"1001", 200},
        {"100112", 600},
        {"101", 550},
        …
        {"11111", 10000000},
        {"empty", 0}
};
```

```
struct chess_chain get_chain(enum chess Chessboard[15][15], enum
chess Friend_color,Vector2 put_chess, Vector2 direction) {
    struct chess_chain chain;
    int p=0,q=0;


    for(int i = 1;i <= 5;i++){
        int X = (int)put_chess.x + i*(int)direction.x;
        int Y = (int)put_chess.y + i*(int)direction.y;
        if(!(in_boundary(X) || in_boundary(Y))||
```

```
            Chessboard[X][Y] == Friend_color*-1){break;}
        p=i;
    }
    for(int i = 1;i <= 5;i++){
        int X = (int)put_chess.x - i*(int)direction.x;
        int Y = (int)put_chess.y - i*(int)direction.y;
        if(!(in_boundary(X) && in_boundary(Y))||
           Chessboard[X][Y] == Friend_color*-1){break;}
                q=i;
    }
    chain.c[0] = Friend_color*-1;
    int index = 1;
    chain.count = p+q+1;
    for(int i = 1-q;i<p;i++){
        int X = max((int)put_chess.x + i*(int)direction.x, 0);
        int Y = max((int)put_chess.y + i*(int)direction.y, 0);
        chain.c[index] = Chessboard[X][Y];
        index++;
    }
    chain.c[chain.count+1] = Friend_color*-1;

    return chain;

}

struct chess_chain put_temp(struct chess_chain input, int index){
    struct chess_chain temp;
    for(int i = 0;i<6;i++){
        temp.c[i]=input.c[i+index];
    }
    return temp;
}

int to_int(enum chess input, enum chess Friend_color){
    if(input == Friend_color) return 1;
    else if(input == Friend_color*-1) return 2;
    else return 0;
}
```

```
int get_key(struct chess_chain input, enum chess Friend_color){
   if(input.c[5] == Friend_color*-1) return
(c(5)*32+c(4)*16+c(3)*8+c(2)*4+c(1)*2+c(0)*1);
    return (c(0)*32+c(1)*16+c(2)*8+c(3)*4+c(4)*2+c(5)*1);
}


int evaluate(enum chess *Chessboard[15][15], enum chess Friend_Color,
Vector2 put_chess) {
    int count = 0;
    for(int X = max(put_chess.x - LEAF_SEARCH_RANGE, 0); X <=
put_chess.x + LEAF_SEARCH_RANGE && in_boundary(X); X++){
        for(int Y = max(put_chess.y-LEAF_SEARCH_RANGE,0); Y <=
put_chess.y + LEAF_SEARCH_RANGE && in_boundary(Y); Y++){

            for(int i = 0;i<4;i++){
                struct chess_chain friend_chain =
get_chain(Chessboard, Friend_Color, (Vector2){X,Y}, direction[i]);
                if(friend_chain.count < 6){
                    count += 0;
                    continue;
                }
                for(int j = 0;j<=friend_chain.count-5;j++){
                    struct chess_chain temp = put_temp(friend_chain,
j);
                    int key = get_key(temp, Friend_Color);
                    count += hashTable[key].value;
                }




                struct chess_chain enemy_chain =
get_chain(Chessboard, Friend_Color*-1, (Vector2){X,Y}, direction[i]);
                if(enemy_chain.count < 6){
                    count -= 0;
                    continue;
```

```
                }
                for(int k = 0;k<=enemy_chain.count-5;k++){
                    struct chess_chain temp = put_temp(enemy_chain,
k);
                    int key = get_key(temp ,Friend_Color*-1);
                    count -= hashTable[key].value;
                }
            }
        }
    }
```

4 实验结果

由于过晚进行系统重构，该算法的性能未能进行测试

5 改进方向

鉴于 alpha-beta 剪枝算法仍基于 min-max 搜索算法遍历，我们可以有以下改进方向

5.1 优先级判定

在节点运算时可以先行对空白棋位进行判定下此位置是否会满足某些特殊棋形乃至绝杀，并根据优先级进行遍历与剪枝。此方法理论上可以提供更优的 alpha 和 beta 值，同时去除大部分无价值但仍位于旧剪枝范围内的子节点。

5.2 深度调整

在棋局开头时，大部分皆为无价值或价值极低的节点，但由于分值平均，alpha 和 beta 值无法剪去此类无价值点位，此时搜索深度的变化不会造成影响。我们可以设定搜索深度随棋局进程加深，以此减少开头几回的搜索时间。

5.3 价值网络深度学习

哈希表里特殊棋形的分值直接影响了算法决定下棋的位置，而人类下棋时往往不会只看高棋子数的棋链。我们可以透过深度学习改变算法对特殊棋形的分数，以此优化下子策略。

5.4 连续下子策略

人类下棋时往往会考虑连续数个回合内的策略，我们可以设置一个列表保存递归途中发现的高价值策略，若下一回合发现棋盘状况与策略吻合，则直接输出该策略或优先对该策略进行搜索。

6 结论

Alpha-Beta 剪枝算法在五子棋等博弈问题的解决中具有重要的实际意义，为提高计算机决策效率和性能提供了可靠的解决方案。未来的研究方向可以考虑进一步优化算法，结合深度学习等技术，以应对更复杂的博弈场景。

7. 附录

7.1 完整代码

```
//
// Created by Admin on 20/11/2023.
//

#ifndef C2023_CHALLENGE_CHESS_ROBOT_H
#define C2023_CHALLENGE_CHESS_ROBOT_H

#include "Includes.h"



#define ATTACK_INDEX 0.7
#define DEFEND_INDEX 1.3
#define MAX_DEPTH 2
#define ROOT_SEARCH_RANGE 3
#define BRANCH_SEARCH_RANGE 2
#define LEAF_SEARCH_RANGE 2

#define c(n) to_int(input.c[n], Friend_color)

struct chess_type{
    char name[15];
    int value;
};
struct chess_chain{
    int count;
    enum chess c[11];
};

struct Global {
    Vector2 put_chess;

};
```

```
Vector2 Get_best_location(enum chess Chessboard[15][15], enum chess
Friend_color, int depth, Vector2 Last_location);
int alpha_beta_search(enum chess Chessboard[15][15], enum chess
Friend_color, int depth, int alpha, int beta,Vector2 put_chess);


int max(int A, int B) {
    if (A > B)return A;
    else return B;
}

int min(int A, int B) {
    if (A < B)return A;
    else return B;
}


bool in_boundary(int i) {
    if (i >= 0 && i <= 14) return true;
    else return false;

}


struct chess_type hashTable[97]={
        {"empty", 0},
        {"21", 0},
        {"010", 1},
        {"211", 150},
        {"010", 1},
        {"1012", 250},
        {"0110", 650},
        {"2111", 500},
        {"010", 1},
        {"1001", 200},
        {"101", 400},
```

```
        {"10112", 800},
        {"0110",  650},
        {"11012", 800},
        {"01110", 1500},
        {"11112", 2500},
        {"010",1},
        {"10001", 100},
        {"1001", 200},
        {"100112", 600},
        {"101", 550},
        {"10101", 550},
        {"1011",3100},
        {"101112", 3000},
        {"0110", 650},
        {"110012", 600},
        {"1101", 2100},
        {"11011", 3000},
        {"01110", 1500},
        {"211101", 3000},
        {"011110", 3000000},
        {"11111", 10000000},
        {"21", 0},
        {"21", 0},
        {"10001", 100},
        {"100011", 175},
        {"1001", 200},
        {"100101", 300},
        {"210011", 600},
        {"100111", 725},
        {"1012", 250},
        {"101001", 300},
        {"10101",550},
        {"101011", 937},
        {"11012", 800},
        {"101101", 1000},
        {"111012", 3000},
        {"101111", 3625},
        {"211", 150},
```

```
        {"100011", 175},
        {"100112", 600},
        {"110011", 750},
        {"10112", 800},
        {"101011", 937},
        {"110112", 2600},
        {"110111", 3650},
        {"1112", 500},
        {"100111", 725},
        {"101112", 3000},
        {"110111", 3650},
        {"11112", 2500},
        {"101111", 3625},
        {"11111", 10000000},
        {"111111", 10000000},
        {"empty", 0},
        {"010", 1},
        {"010",1},
        {"211", 150},
        {"010", 1},
        {"1012", 250},
        {"0110", 650},
        {"2111", 500},
        {"010", 1},
        {"1001", 200},
        {"101", 550},
        {"10112", 800},
        {"0110", 650},
        {"11012", 800},
        {"01110", 1500},
        {"11112", 2500},
        {"21", 0},
        {"10001", 100},
        {"1001", 200},
        {"100112",  600},
        {"1012", 250},
        {"10101", 550},
        {"11012", 800},
```

```
        {"111012", 3000},
        {"211",  150},
        {"210011", 600},
        {"10112", 800},
        {"110112", 2600},
        {"1112", 500},
        {"211101", 1500},
        {"11112", 2500},
        {"11111", 10000000},
        {"empty", 0}
};


struct Global global = {(Vector2){7,7}};
static Vector2 direction[4] = {{0,  1},
                               {1,  0},
                               {1,  1},
                               {-1, 1}};


Vector2 robot(enum chess Chessboard[15][15], enum chess my_color,
Vector2 last_location) {

    if(last_location.x == 100 || last_location.y == 100){
        return (Vector2){7,7};
    }

     alpha_beta_search(Chessboard, my_color, MAX_DEPTH, -
INFINITY,INFINITY-1, last_location);
    return global.put_chess;

}



struct chess_chain get_chain(enum chess Chessboard[15][15], enum
chess Friend_color,Vector2 put_chess, Vector2 direction) {
    struct chess_chain chain;
```

```
    int p=0,q=0;


    for(int i = 1;i <= 5;i++){
        int X = (int)put_chess.x + i*(int)direction.x;
        int Y = (int)put_chess.y + i*(int)direction.y;
        if(!(in_boundary(X) || in_boundary(Y))||
             Chessboard[X][Y] == Friend_color*-1){break;}
        p=i;
    }
    for(int i = 1;i <= 5;i++){
        int X = (int)put_chess.x - i*(int)direction.x;
        int Y = (int)put_chess.y - i*(int)direction.y;
        if(!(in_boundary(X) && in_boundary(Y))||
           Chessboard[X][Y] == Friend_color*-1){break;}
                q=i;
    }
    chain.c[0] = Friend_color*-1;
    int index = 1;
    chain.count = p+q+1;
    for(int i = 1-q;i<p;i++){
        int X = max((int)put_chess.x + i*(int)direction.x, 0);
        int Y = max((int)put_chess.y + i*(int)direction.y, 0);
        chain.c[index] = Chessboard[X][Y];
        index++;
    }
    chain.c[chain.count+1] = Friend_color*-1;


    return chain;


}


struct chess_chain put_temp(struct chess_chain input, int index){
    struct chess_chain temp;
    for(int i = 0;i<6;i++){
        temp.c[i]=input.c[i+index];
    }
    return temp;
```

```
}

int to_int(enum chess input, enum chess Friend_color){
    if(input == Friend_color) return 1;
    else if(input == Friend_color*-1) return 2;
    else return 0;
}

int get_key(struct chess_chain input, enum chess Friend_color){
   if(input.c[5] == Friend_color*-1) return
(c(5)*32+c(4)*16+c(3)*8+c(2)*4+c(1)*2+c(0)*1);
    return (c(0)*32+c(1)*16+c(2)*8+c(3)*4+c(4)*2+c(5)*1);
}


int evaluate(enum chess *Chessboard[15][15], enum chess Friend_Color,
Vector2 put_chess) {
    int count = 0;
    for(int X = max(put_chess.x - LEAF_SEARCH_RANGE, 0); X <=
put_chess.x + LEAF_SEARCH_RANGE && in_boundary(X); X++){
        for(int Y = max(put_chess.y-LEAF_SEARCH_RANGE,0); Y <=
put_chess.y + LEAF_SEARCH_RANGE && in_boundary(Y); Y++){

            for(int i = 0;i<4;i++){
                struct chess_chain friend_chain =
get_chain(Chessboard, Friend_Color, (Vector2){X,Y}, direction[i]);
                if(friend_chain.count < 6){
                    count += 0;
                    continue;
                }
                for(int j = 0;j<=friend_chain.count-5;j++){
                    struct chess_chain temp = put_temp(friend_chain,
j);
                    int key = get_key(temp, Friend_Color);
                    count += hashTable[key].value;
                }
```

```
                struct chess_chain enemy_chain =
get_chain(Chessboard, Friend_Color*-1, (Vector2){X,Y}, direction[i]);
                if(enemy_chain.count < 6){
                    count -= 0;
                    continue;
                }
                for(int k = 0;k<=enemy_chain.count-5;k++){
                    struct chess_chain temp = put_temp(enemy_chain,
k);
                    int key = get_key(temp ,Friend_Color*-1);
                    count -= hashTable[key].value;
                }
            }
        }
    }


//W????0???W
//count = 10
//i = 1~3

    return count;
}

int alpha_beta_search(enum chess Chessboard[15][15], enum chess
Friend_color, int depth, int alpha, int beta,Vector2 put_chess) {

    if (depth == 0) {
        return evaluate(Chessboard, Friend_color, put_chess);
    }
    bool root = (depth == MAX_DEPTH);
    for (int X = max(put_chess.x - BRANCH_SEARCH_RANGE, 0); X <=
put_chess.y + BRANCH_SEARCH_RANGE && in_boundary(X); X++) {
        for (int Y = max(put_chess.y - BRANCH_SEARCH_RANGE, 0); Y <=
put_chess.y + BRANCH_SEARCH_RANGE && in_boundary(Y); Y++) {
            if (Chessboard[X][Y] == empty) {
```

```
                Chessboard[X][Y] = Friend_color;
                int value = alpha_beta_search(Chessboard,
Friend_color, depth - 1, alpha, beta, (Vector2) {X, Y});

                Chessboard[X][Y] = empty;

                if (!(depth % 2)) {
                    if (value >= alpha) {
                        alpha = value;
                        if (root) {
                            printf("%d, %d, %d\n", X,Y, value);
                            global.put_chess.x = X;
                            global.put_chess.y = Y;
                        }
                    }
                } else {
                    if (value <= beta) {
                        beta = value;
                    }
                    if (alpha >= beta) {
                        break;
                    }
                }
            }

        }
    }
    return (!(depth % 2)) ? alpha : beta;
}

Vector2 Get_best_location(enum chess Chessboard[15][15], enum chess
Friend_color, int depth, Vector2 Last_location){


    for(int X = max(Last_location.x - ROOT_SEARCH_RANGE,
0);X<=Last_location.x+ROOT_SEARCH_RANGE && in_boundary(X);X++){
        for(int Y = max(Last_location.y - ROOT_SEARCH_RANGE,
0);Y<=Last_location.y+ROOT_SEARCH_RANGE && in_boundary(Y);Y++){
```

```
            if(Chessboard[X][Y] == empty){
                alpha_beta_search(Chessboard, Friend_color,depth,1-
(INFINITY), INFINITY-1, Last_location);
                return global.put_chess;
            }
        }
    }


}



#endif //C2023_CHALLENGE_CHESS_ROBOT_H
```

```
//
// Created by Admin on 6/11/2023.
//

#ifndef C2023_CHALLENGE_CHESSBOARD_H
#define C2023_CHALLENGE_CHESSBOARD_H

#endif //C2023_CHALLENGE_CHESSBOARD_H
#include "Includes.h"
enum chess{
    black = 1, white = -1, empty = 0
};
enum chess chessboard[15][15] = {empty};
void zero_matrix(){
    for (int i = 0;i<255;i++){
     chessboard[i%15][i/15] = empty;
    }
}
typedef struct node{
    int x;
    int y;
    enum chess chess;
}node_t;
```

```
void push_node(node_t *timeline, int steps, int x, int y, enum chess
chess){
    timeline[steps-1].x = x;
    timeline[steps-1].y = y;
    timeline[steps-1].chess = chess;
}
```

```
//
// Created by Admin on 13/11/2023.
//

#ifndef C2023_CHALLENGE_TEXTS_H
#define C2023_CHALLENGE_TEXTS_H

#endif //C2023_CHALLENGE_TEXTS_H
enum status {
    NORMAL = 0,
    PLACE_ON_PREVIOUS_CHESS = 101,
};
enum game{
    GoingOn = 0,
    BlackWin = 1,
    WhiteWin = -1

};


void Print_win(enum game game_status){
    switch (game_status) {
        case BlackWin:
            DrawText("Black Win", 142, 230, 100, DARKGREEN);
            return ;

        case WhiteWin:
            DrawText("White Win", 142, 230, 100, DARKGREEN);
            return ;

        case GoingOn:
```

```
            return ;
    }
}

void draw_winning_line(int X, int Y, enum chess chess_color){
    Vector2 start_chess;
    Vector2 end_chess;
    Vector2 direction[4] = {{0, 1},{1, 0},{1,1},{-1, 1}};


    for(int i = 0; i < 4; i++){
        int dir = 1;
        int steps = 1;
        int count_on_dir = 1;
        while((chessboard[(int)(X + dir*steps*(int)(direction[i].x))]
               [(int)(Y + dir*steps*(int)(direction[i].y))] ==
chess_color)
              && (X + dir*steps*(int)(direction[i].x) >= 0) &&  (X +
dir*steps*(int)(direction[i].x) <= 14)
              &&  (Y + dir*steps*(int)(direction[i].y) >= 0) &&  (Y +
dir*steps*(int)(direction[i].y) <= 14))
        {
            steps++;count_on_dir++;
        }
        start_chess.x = (float)(X + dir*(steps-
1)*(int)(direction[i].x));
        start_chess.y = (float)(Y + dir*(steps-
1)*(int)(direction[i].y));
        dir *= -1;
        steps = 1;
        while((chessboard[(int)(X + dir*steps*(int)(direction[i].x))]
               [(int)(Y + dir*steps*(int)(direction[i].y))] ==
chess_color)
              && (X + dir*steps*(int)(direction[i].x) >= 0) &&  (X +
dir*steps*(int)(direction[i].x) <= 14)
              &&  (Y + dir*steps*(int)(direction[i].y) >= 0) &&  (Y +
dir*steps*(int)(direction[i].y) <= 14))
        {
```

```
            steps++;count_on_dir++;
        }
        end_chess.x = (float)(X + dir*(steps-
1)*(int)(direction[i].x));
        end_chess.y = (float)(Y + dir*(steps-
1)*(int)(direction[i].y));

        if(count_on_dir >= 5){
            DrawLineEx(
                    //direction[i].x * 10
                    //direction[i].y
                    (Vector2){25 + (start_chess.x) * 50+
direction[i].x * 21,25 + (start_chess.y) * 50 + direction[i].y * 21},
                    (Vector2){25 + (end_chess.x) * 50 -
direction[i].x * 21,25 + (end_chess.y) * 50 - direction[i].y * 21},
                    5,GREEN );
            return;
        }

    }


}
```

```
//
// Created by Admin on 13/11/2023.
//

#ifndef C2023_CHALLENGE_WINCONDITION_H
#define C2023_CHALLENGE_WINCONDITION_H

#endif //C2023_CHALLENGE_WINCONDITION_H
#include "Includes.h"

enum game determinine_win(int X, int Y, enum chess chess_color){
    Vector2 direction[4] = {{0, 1},{1, 0},{1,1},{-1, 1}};

    for (int i = 0; i < 4; i++){
```

```
        int link_on_dir = 1;
        int dir = 1;
        int reverse = 0;
        int steps = 1;
        while(reverse < 2){
            if
                    ((chessboard[(int)(X +
dir*steps*(int)(direction[i].x))]
                     [(int)(Y + dir*steps*(int)(direction[i].y))] ==
chess_color)
                     && (X + dir*steps*(int)(direction[i].x) >= 0) &&
(X + dir*steps*(int)(direction[i].x) <= 14)
                     &&  (Y + dir*steps*(int)(direction[i].y) >= 0)
&&  (Y + dir*steps*(int)(direction[i].y) <= 14))
            {
                steps++;link_on_dir++;
            }
            else{
                reverse++;
                steps = 1;
                dir*=-1;
            }
        }
        if(link_on_dir >= 5){
            return (enum game)chess_color;
        }
    }
     return GoingOn;

}
```

```
#include "Includes.h"

#define Humans_turn if(Gesture == GESTURE_TAP){X =
(int)(touchPosition.x / 50);Y = (int)(touchPosition.y /
50);if(chessboard[X][Y] == 0){chessboard[X][Y] = (enum
chess)turn;push_node(chess_timeline, steps, X, Y, (enum
```

```
chess)turn);turn*=-1;steps++;}else{error = PLACE_ON_PREVIOUS_CHESS;}}
#define Robots_turn robot_location = robot(chessboard, (enum
chess)turn);X = (int)robot_location.x;Y =
(int)robot_location.y;chessboard[X][Y] = (enum
chess)turn;push_node(chess_timeline, steps, X, Y, (enum
chess)turn);turn*=-1;steps++;


enum turn_t {
    black_turn = 1, white_turn = -1
};
enum robot_human {
    ROBOT = 1, HUMAN = -1, null = 0
};

int pos(int input) {
    int temp;
    temp = 25 + (input) * 50;
    return temp;
}

void DrawCross(int X, int Y, int radius, int width, Color color) {
    Vector2 Startpos1 = {X - radius, Y - radius};
    Vector2 Endpos1 = {X + radius, Y + radius};
    Vector2 Startpos2 = {X + radius, Y - radius};
    Vector2 Endpos2 = {X - radius, Y + radius};

    DrawLineEx(Startpos1, Endpos1, width, color);
    DrawLineEx(Startpos2, Endpos2, width, color);
}


int relation;

void main() {

    const int screenWidth = 752, screenHeight = 752;
    bool Exit,CHOSE_HUMAN,CHOSE_ROBOT;
```

```
int X, Y,steps,Gesture;
enum game Current_Status = GoingOn;
enum turn_t turn = black_turn;//black first
enum robot_human RobotHumanTurn;
Vector2 cursor_position,robot_location;
InitWindow(screenWidth, screenHeight, "Gomoku");
Vector2 touchPosition;
enum status error;
static Rectangle
touchArea = {1, 1, screenWidth - 2, screenHeight - 2},
Robot_first = {51, 320, 300, 75},
Human_first = {401, 320, 300, 75},
Continue_Button = {225, 400, 300, 75},
End_Button = {225, 500, 300, 75};
SetTargetFPS(60);
//Define variables


Start:
/*************************
*   Initialise variables  *
*************************/
zero_matrix();
error = NORMAL;
touchPosition = (Vector2){0, 0};
node_t *chess_timeline = (node_t*) malloc(sizeof(node_t) * 255);
chess_timeline[0].x = 100;
chess_timeline[0].y = 100;
chess_timeline[0].chess = empty;
Gesture = GESTURE_NONE;
steps = 1;
Current_Status = GoingOn;
turn = black_turn;//black first
CHOSE_ROBOT = false;CHOSE_HUMAN = false;Exit = false;
RobotHumanTurn = null;
```

```
    //Main loop
    while (!Exit) {
        /***********************************
        *   Numeral calculations and logics  *
        ***********************************/
        error = NORMAL;
        Gesture = GetGestureDetected();
        touchPosition = GetTouchPosition(0);
        if (RobotHumanTurn == null) {
            if (Gesture == GESTURE_TAP) {
                if (CheckCollisionPointRec(touchPosition,
Robot_first)) {
                    RobotHumanTurn = ROBOT;
                    relation = 1;
                    CHOSE_ROBOT = true;
                } else if (CheckCollisionPointRec(touchPosition,
Human_first)) {
                    RobotHumanTurn = HUMAN;
                    relation = -1;
                    CHOSE_HUMAN = true;
                }
                goto Draw_directly;
            } else goto Draw_directly;

        }//choose sides
        if(Current_Status){
            if (Gesture == GESTURE_TAP) {
                if (CheckCollisionPointRec(touchPosition,
Continue_Button)) {
                    Exit = false;
                    free(chess_timeline);
                    goto Start;
                } else if (CheckCollisionPointRec(touchPosition,
End_Button)) {
                    Exit = true;
                }
            }goto Draw_directly;
```

```
        } //Ending logic




        if ((enum robot_human) turn * relation == ROBOT) {
            robot_location = robot(chessboard, (enum chess)turn,
(Vector2){chess_timeline[steps-2].x, chess_timeline[steps-2].y});
            X = (int)robot_location.x;
            Y = (int)robot_location.y;
            chessboard[X][Y] = (enum chess)turn;
            push_node(chess_timeline, steps, X, Y, (enum
chess)turn);turn*=-1;steps++;
        } else if ((enum robot_human) turn * relation == HUMAN) {
            Humans_turn;
        }
        //Human's turn or robot's turn

        Current_Status = determinine_win(X, Y, (enum chess) (turn * -
1));



        /************************************
        *  Printing chessboards and statuses  *
         ************************************/

        Draw_directly:
        BeginDrawing();
        DrawRectangleRec(touchArea, WHITE);
        for (int i = 0; i < 15; i++) {
            DrawLine(25 + i * 50, 25, 25 + i * 50, 725, BLACK);
        }
        for (int i = 0; i < 15; i++) {
            DrawLine(25, 25 + i * 50, 725, 25 + i * 50, BLACK);
        }
        ClearBackground(WHITE);
        for (int i = 0; i < steps - 1; ++i) {
            cursor_position.x = pos(chess_timeline[i].x);
```

```
            cursor_position.y = pos(chess_timeline[i].y);
            switch
(chessboard[chess_timeline[i].x][chess_timeline[i].y]) {
                case black:
                    DrawCircleV(cursor_position, 21, BLACK);
                    break;
                case white:
                    DrawCircleV(cursor_position, 21, BLACK);
                    DrawCircleV(cursor_position, 20, WHITE);
                    break;
                case empty:
                    break;
            }
        }

        if (RobotHumanTurn == null) {
            DrawText("Choose your side", 110, 230, 60, DARKGRAY);
            DrawRectangleRec(Robot_first, GRAY);
            DrawText("Robot First", 53, 340, 50, BLACK);
            DrawRectangleRec(Human_first, GRAY);
            DrawText("Human First", 401, 340, 50, BLACK);
        }
        if (CHOSE_ROBOT) {
            DrawRectangleRec(Robot_first, WHITE);
            DrawText("robot turn", 53, 340, 50, WHITE);
            CHOSE_ROBOT = false;
        }
        if (CHOSE_HUMAN) {
            DrawRectangleRec(Human_first, WHITE);
            DrawText("Human First", 401, 340, 50, WHITE);
            CHOSE_HUMAN = false;
        }

        switch (error) {
            case NORMAL:
                break;
            case PLACE_ON_PREVIOUS_CHESS:
                DrawCross(pos(X), pos(Y), 15, 6, RED);
```

```
                break;
        }


        draw_winning_line(X,Y,(enum chess)(turn*-1));
        Print_win(Current_Status);

        if(abs(Current_Status)){
            DrawRectangleRec(Continue_Button, GRAY);
            DrawText("Continue", 267, 415, 50, BLACK);
            DrawRectangleRec(End_Button, GRAY);
            DrawText("End", 330, 515, 50, BLACK);
        }



        EndDrawing();
    }
    //system("pause");

}
```