五子棋AI实验报告

2023080909002　　邓堂瑞

一、实验目的

1. 掌握五子棋游戏的基本规则和实现方法。
2. 学习并实现alpha-beta剪枝搜索算法。
3. 学习并实现启发式搜索进行优化。
4. 学习并使用raylib库进行UI设计。

二、实验环境

- 操作系统：Windows
- 开发工具：CLion
- 使用库：vcpkg+raylib

三、实验内容与步骤

1. 五子棋游戏规则的实现

- 双方在 15 * 15 的棋盘上先后依次下棋

  使用一个结构体表示五子棋棋盘

```
typedef struct Gobang{
    int chessboard[15][15];  // 棋盘 15 x 15, 0表示空, -1表示黑, 1表示白
    PIECE now_player;   // 当前应该下棋的玩家
    Stack *steps;
    int pos_steps[15][15];    // 可能的落子点, 采用一个棋盘表示, 0: 不建议搜索, 1: 建议搜索
    int pos_steps_num;
} Gobang;
```

- 使用枚举类型表示双方棋手和棋子

```
enum PIECE{PLAYER_BLACK = -1, PLAYER_WHITE = 1};    // -1 为黑棋, 1 为白棋
```

- 连五子即可获胜，定义一个判断是否获胜的方法

```
int is_win(Gobang *gobang){
    /*
     * 返回获胜的玩家的棋子, 如果没人获胜, 则返回0
     */
    if(gobang->steps->sp < 9){
        // 低于 9 子不可能获胜
        return 0;
    }
    PIECE piece;

    int *prev = gobang->steps->get_last_element(gobang->steps);
```

```
    piece = -gobang->now_player;    // 检查的棋子颜色
    int count;
    int is_in_bound;
    int x = prev[1], y = prev[2], dir = 1;
    for(int i = 0; i < DIR_NUM; i ++){
        // ...
        // 遍历内容略
        if(count >= 5){
            // 达到获胜条件，返回获胜棋子
            return piece;
        }
    }
    return 0;   // 没有一方棋子获胜，返回0
}
```

- 下棋方法

```
int fall(Gobang *gobang, int x, int y){
    /*
     * 通过坐标落子
     */
    if(!(x < BOARD_SIZE && y < BOARD_SIZE && x >= 0 && y >= 0)){
        return -100;  // 超出边界，落子失败
    } else if(gobang->chessboard[x][y] != 0){
        return -101;  // 已有子，落子失败
    }

    PIECE piece = gobang->now_player;

    // 落子
    gobang->chessboard[x][y] = piece;   // 落子，用 2 来表示最近的操作

    // 添加步骤
    gobang->steps->append(gobang->steps, piece, x, y);

    // 生成可能落子点
     generate_steps(gobang, x, y);

    // 交换玩家
    gobang->now_player *= -1;   // 因为定义方式，* (-1)即可交换棋手

    return piece;   // 返回下的棋
}
```

- 撤回

  撤回最后一步棋

```
int take_back(Gobang *gobang);  // 实现方法略
```

- 使用一个栈来记录双方整局的操作，定义在`struct Gobang`内

```
Stack *steps;
// 栈实现代码略
```

- 初始化五子棋

```
Gobang *create_gobang(){
    Gobang *gobang = (Gobang *)malloc(sizeof(Gobang));
    memset(gobang, 0, sizeof(Gobang));
    gobang->now_player = PLAYER_BLACK;
    gobang->steps = create_stack(BOARD_SIZE * BOARD_SIZE);
    return gobang;
}
```

- 本实验未考虑禁手

2. alpha-beta剪枝搜索算法的基本原理和实现

- alpha-beta剪枝搜索的原理

  Alpha-Beta 剪枝搜索是一种用于优化博弈树搜索的算法，它可以减少搜索的分支数量，从而提高搜索效率。

  在博弈树搜索中，通常使用深度优先搜索（DFS）来遍历所有可能的游戏状态。然而，由于搜索空间的指数增长，完全搜索整个博弈树是不可行的。Alpha-Beta 剪枝算法通过评估游戏状态的值来选择性地剪去一些搜索路径，从而减少搜索的分支。

  Alpha-Beta 剪枝算法使用两个参数：alpha 和 beta。这两个参数分别表示当前搜索路径上最好的已知最大值和最小值。在搜索过程中，如果某个节点的值超出了 alpha 和 beta 之间的范围，那么可以剪去该节点的搜索路径，因为对于当前节点的父节点来说，它已经找到了更好的选择。

  具体来说，Alpha-Beta 剪枝算法遵循以下原则：

  1. 对于最大化玩家（Maximizer）来说，它试图找到最大的值。它维护一个 alpha 值，表示当前已知的最大值。
  2. 对于最小化玩家（Minimizer）来说，它试图找到最小的值。它维护一个 beta 值，表示当前已知的最小值。
  3. 在搜索过程中，如果某个节点的值超过了 beta 值（对于最小化玩家）或低于 alpha 值（对于最大化玩家），那么可以停止对该节点的搜索，因为它不会影响最终结果。
  4. 在每个节点上，根据当前玩家的角色选择合适的子节点进行搜索。如果是最大化玩家，那么选择子节点中的最大值，并更新 alpha 值；如果是最小化玩家，那么选择子节点中的最小值，并更新 beta 值。
  5. 递归地进行搜索，直到达到搜索的终止条件（例如达到最大搜索深度或找到终局状态）。

  通过使用 Alpha-Beta 剪枝算法，可以减少搜索的深度，并且在某些情况下，甚至可以避免搜索整个博弈树。这样可以大大提高搜索效率，使得在有限的时间内找到更好的决策。

  另外Alpha-Beta 剪枝算法的效果取决于搜索树的结构和节点的顺序。如果节点的顺序不合适，可能无法实现最佳的剪枝效果。

- alpha-beta剪枝搜索的实现
  - 计算棋局的分数

    使用评估函数为整个棋局打分，这里采用对棋子分别打分后再把所有棋子的分数相加获得整个棋盘的分

```
int evaluate(Gobang *gobang){
    /*
     * 评估棋盘局势
     */
    int x, y, i;
    int chessboard_score = 0;
    for(i = 0; i < gobang->steps->sp; i ++){
        x = gobang->steps->stack[i][1];
        y = gobang->steps->stack[i][2];
        chessboard_score += piece_score[x][y];
    }
    return chessboard_score;
}
```

其中`piece_score`为二维数组，存储棋盘上每一个棋子的分，`update_score`函数会在每一次下棋后会动态更新`piece_score`

```
void update_score(Gobang *gobang) {
    // 更新棋盘分数可能变化的子的分数
    int i;
    int* prev_pos = gobang->steps->get_last_element(gobang->steps);
    int p_x = prev_pos[1], p_y = prev_pos[2], is_in_bound, dir = 1;

    piece_score[prev_pos[1]][prev_pos[2]] = get_score(gobang,
prev_pos[1], prev_pos[2]);

    for(i = 0; i < DIR_NUM; i ++) {
        // 略
        // 遍历获取附近所有分数可能会更改的子，使用get_score函数重新计算该分数的得
分
    }
}
```

在`get_score`函数中，通过查找连四，活三，冲四，眠三等连子情况来进行打分，该函数会查找位于`(x,y)`点的棋子所参与的连子情况，然后将对应的分数相加，具体情况和分数参照下表

情况	分数
成五	1000000
活四	200000
活三	70000
冲四	50000
死四	5500
跳三	22500
眠三	8000
死三	1000

情况	分数
活二	1000
跳二	720
大跳二	600
眠二	500
死二	300

```
int get_score(Gobang *gobang, const int x, const int y){
    if(gobang->chessboard[x][y] == 0){
        return 0;
    }
    int i, in_is_win;
    char pattern[PATTERN_SIZE] = {0}; // 位于x, y坐标的棋子的字符串模式
    PIECE piece = gobang->chessboard[x][y];
    int score = 0;
    int is_in_bound, space_count;
    int p_x = x, p_y = y, dir = 1;

    // 遍历
    for(i = 0; i < DIR_NUM; i ++){
        // 遍历八个方向
        // 初始化
        int mid = 5, count;    // 起始遍历点
        in_is_win = 1;
        memset(pattern, 0, sizeof(pattern));
        pattern[5] = '#';
        // 开始遍历

        //...
        // 获得单个方向上棋子的组成，以字符串形式表示

        int suc_index; // 成功匹配的下标, -1则表示没有匹配到

        int left = 0;
        // 去除左侧多余的 \0 和 $
        for(; left < PATTERN_SIZE; left ++){
            if(pattern[left] == '$' && pattern[left + 1] == '$'){
                left ++;
                break;
            }
            if(pattern[left] != '\0') {
                break;
            }
        }
        // 匹配
        suc_index = find_patten(pattern + left);
        if(suc_index != -1){
            score += statements[suc_index];
            if((suc_index == 16 || suc_index == 17) && piece == ai){
                score++;
```

```
            }
        }
        if(in_is_win >= 5){
            score += 1000000;
        }
    }
    return score * piece * ai;  // 纠正分数正负，使AI分数恒为正
}
```

- 剪枝搜索

```
int search(Gobang *gobang, int depth, int alpha, int beta, PIECE
cur_player){
    /*
     * 搜索算法，返回一个分数
     */
    if(depth == 0 || gobang->pos_steps_num == 0){
        // 遍历到底层，开始算分
        return evaluate(gobang);
    }
    // 搜索
    int score;
    if(cur_player == ai){
        // max 层
        for(int i = 0; i < BOARD_SIZE; i ++){
            for(int j = 0; j < BOARD_SIZE; j ++){
                if(gobang->pos_steps[i][j] == 0){
                    continue;
                }
                // 具体代码略...
                if(alpha >= beta){
                    return alpha;
                }
            }
        }
        return alpha;
    } else {
        // min 层
        for(int i = 0; i < BOARD_SIZE; i ++){
            for(int j = 0; j < BOARD_SIZE; j ++){
                if(gobang->pos_steps[i][j] == 0){
                    continue;
                }
                // 具体代码略...
                if(alpha >= beta){
                    return beta;
                }
            }
        }
        return beta;
    }
}
```

- 着法生成

显然要下棋的位置应该位于已有子的附近，不可能搜索整个棋盘，所以需要生成着法，去除不可能落子的点

`generate_steps`函数会生成位于`(x,y)`的可能的落子点

```
void generate_steps(Gobang *gobang, int x, int y){
    /*
     * 生成可能的落子点，在落子时生成
     * 生成策略：距离已落子点曼哈顿距离小于等于2的所有点
     */
    int dir = 1;
    int p_x = x, p_y = y;
    int is_in_bound, count;
    for(int i = 0; i < DIR_NUM; i ++){
        // 遍历检查附近2格内所有可以落子的点
        // ...
    }
    // 遍历完成，当前子位置无法落子
    gobang->pos_steps[x][y] = 0;
}
```

同样的，在alpha-beta搜索过程中，需要重新生成着法

3. 启发式搜索的实现

- 启发式搜索的基本原理

  启发式搜索是一种在搜索过程中使用启发信息来引导搜索方向，以提高搜索效率的方法。在五子棋AI中，启发式搜索的主要目标是评估棋盘上的每个位置的价值，以便AI可以决定下一步的最佳位置。

  启发式搜索的基本原理是：对于每个可能的落子位置，我们都会根据一些预定义的规则（启发式函数）来评估其价值。这些规则可能包括：

  1. **连子数**：在五子棋中，连子数是一个重要的评价指标。例如，一个位置如果能形成五子连珠，那么它的价值就非常高。
  2. **威胁和机会**：如果一个位置能阻止对手形成五子连珠，或者能为自己创造形成五子连珠的机会，那么这个位置的价值也会很高。
  3. **位置**：棋盘中心的位置通常比边缘位置更有价值，因为中心位置有更多的形成五子连珠的可能性。

  在实际的搜索过程中，我们会使用启发式函数来评估所有可能的落子位置，然后选择价值最高的位置作为下一步的落子位置。这样，AI就可以在大量的可能性中快速找到最有希望的落子位置，从而大大提高搜索的效率。

  另外，启发式搜索并不保证能找到最优解，但在大多数情况下，它能提供一个相当好的解，而且计算效率高，适合于五子棋这种搜索空间巨大的游戏。

- 启发式搜索的实现
  - 启发式函数

    采用连子数来决定价值

```
void heuristically_search(Gobang *gobang){
    /*
     * 启发式搜索
```

```
     */
    int max_same_piece_num;
    int dir = 1, count, is_in_bound;
    int p_x, p_y, piece;
    int match[4][2] = {
            {1, 1},
            {-1, 1},
            {1, -1},
            {-1, -1}
    };
    int dir_score[2];   // 某条线上的连子数， 0为白棋的连子数，1为黑棋的连子数
    int index[6] = {0, 0, 0, 0, 0, 0};
    for(int x = 0; x < BOARD_SIZE; x ++){
        for(int y = 0; y < BOARD_SIZE; y ++){
            if(gobang->pos_steps_num <= 0 || gobang->pos_steps[x][y] ==
0){
                continue;
            }
            // 遍历所有的待考虑的落子点
            // ... 略
            // 统计连子数来决定搜索顺序
            heu_locations[max_same_piece_num][index[max_same_piece_num]]
[0] = x;
            heu_locations[max_same_piece_num][index[max_same_piece_num]]
[1] = y;
            heu_locations[max_same_piece_num][index[max_same_piece_num]
+ 1][0] = -1;
            index[max_same_piece_num] ++;
            // 更新位置，并标记结束位置
        }
    }
}
```

- alpha-beta剪枝

  这里对第一步的考虑进行启发式搜索

  连子数从高到低依次遍历

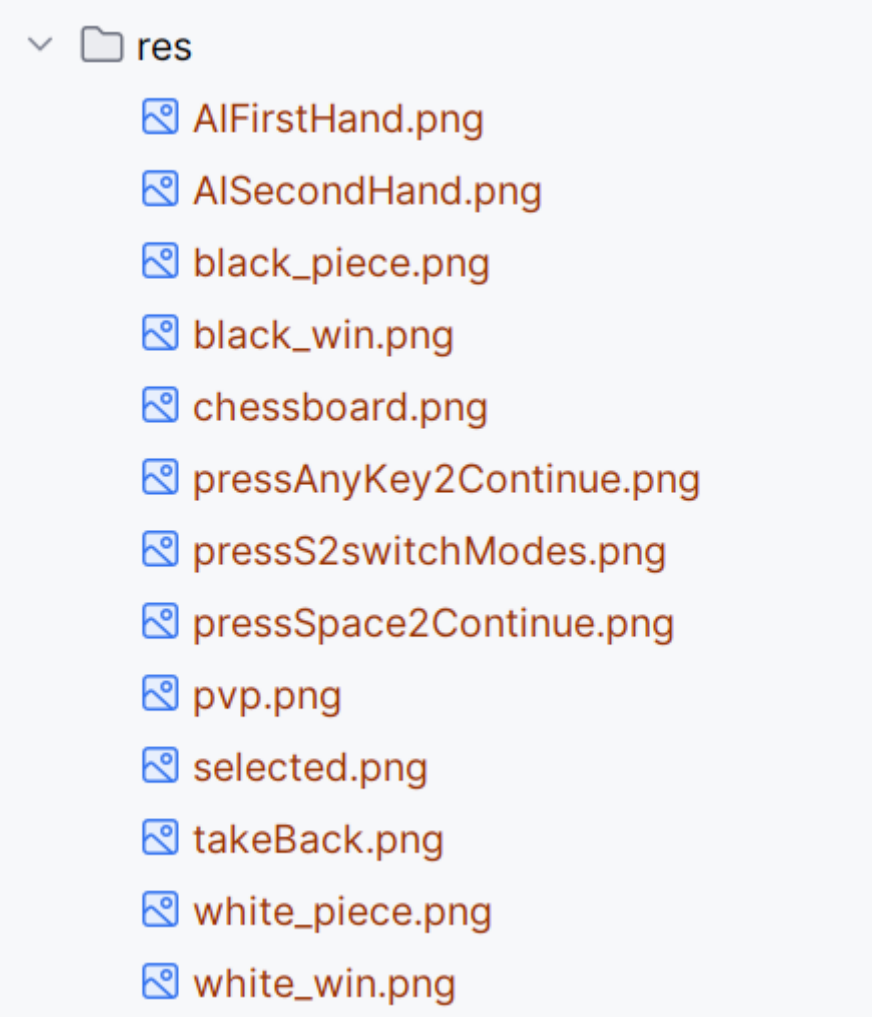
```
int first_search(Gobang *gobang, int depth, int alpha, int beta, PIECE
cur_player){
    /*
     * 第一次搜索
     */

    for(int i = 0; i < 6; i ++){
        heu_locations[i][0][0] = -1;
    }
    heuristically_search(gobang);
    int score;
    int i, j;
    for(int count = 5; count >= 0; count --){
        for(int index = 0; index < BOARD_SIZE * BOARD_SIZE; index ++){
            if (heu_locations[count][index][0] == -1) {
                break;
            }
            i = heu_locations[count][index][0];
```

```
            j = heu_locations[count][index][1];
            fall(gobang, i, j);
            update_score(gobang);
            score = search(gobang, depth - 1, alpha, beta, -cur_player);
            if(score > alpha){
                alpha = score;
                if (depth == MAX_DEPTH){
                    best_pos[1] = j;
                    best_pos[0] = i;
                }
            }
            remove_piece(gobang, i, j);
            update_score(gobang);
            if(alpha >= beta){
                return alpha;
            }
        }
    }
    return alpha;
}
```

4. 使用raylib库进行UI设计

- 使用raylib库进行UI设计
  - 绘制棋盘，棋子图片，文字图片



  - 使用`LoadTexture`函数从本地加载`Texture`

```
// 加载图片资源
Texture t_chess_board =
LoadTexture("..\\..\\gobang\\res\\chessboard.png");
Texture t_white = LoadTexture("..\\..\\gobang\\res\\white_piece.png");
Texture t_black = LoadTexture("..\\..\\gobang\\res\\black_piece.png");
Texture t_selected = LoadTexture("..\\..\\gobang\\res\\selected.png");
Texture t_black_win = LoadTexture("..\\..\\gobang\\res\\black_win.png");
Texture t_white_win = LoadTexture("..\\..\\gobang\\res\\white_win.png");
Texture t_pressSpace2Continue =
LoadTexture("..\\..\\gobang\\res\\pressSpace2Continue.png");
Texture t_take_back = LoadTexture("..\\..\\gobang\\res\\takeBack.png");
Texture t_pressS2SwitchModes =
LoadTexture("..\\..\\gobang\\res\\pressS2switchModes.png");
Texture modes[3] = {
        LoadTexture("..\\..\\gobang\\res\\pvp.png"),   // pvp 0
        LoadTexture("..\\..\\gobang\\res\\AIFirstHand.png"),   // AI
first 1
        LoadTexture("..\\..\\gobang\\res\\AISecondHand.png")   // AI
second 2
};
```

- 初始化窗口并设置背景图片和log

```
InitWindow(WINDOW_WIDTH,WINDOW_HEIGHT,"gobang AI");
SetTraceLogLevel(LOG_WARNING);

// 设置背景色
Color color = GetColor(0xfafafaff);
```

- 遍历棋局并检测鼠标位置和事件绘制ui

```
BeginDrawing();
        ClearBackground(color);
        DrawTexture(t_chess_board, CHESSBOARD_START_X,
CHESSBOARD_START_Y, WHITE);
        DrawTexture(t_take_back, 900, CHESSBOARD_START_Y + 100, WHITE);
        DrawTexture(t_pressS2SwitchModes, 900, CHESSBOARD_START_Y + 200,
WHITE);
        mode %= 3;
        DrawTexture(modes[mode], 900, 900, WHITE);
        // DrawTexture(t_black_win, 1000, 1000, WHITE);
        for(int i = 0; i < 15; i ++){
            for(int j = 0; j < 15; j ++){
                pos_x = BOUNDARY_LENGTH + CHESSBOARD_START_X +
GAP_LENGTH * j;
                pos_y = BOUNDARY_LENGTH + CHESSBOARD_START_Y +
GAP_LENGTH * i;
                if(gobang->chessboard[i][j] < 0){
                    DrawTexture(t_black, pos_x, pos_y, WHITE);
                } else if (gobang->chessboard[i][j] > 0){
                    DrawTexture(t_white, pos_x, pos_y, WHITE);
                }
            }
        }
```

```
        if(selected_x < 15 && selected_y < 15){
            mouse_pos_x = BOUNDARY_LENGTH + CHESSBOARD_START_X +
GAP_LENGTH * selected_x;
            mouse_pos_y = BOUNDARY_LENGTH + CHESSBOARD_START_Y +
GAP_LENGTH * selected_y;
            DrawTexture(t_selected, mouse_pos_x, mouse_pos_y, WHITE);
        }

        if(win == PLAYER_BLACK || win == PLAYER_WHITE){
            is_game_end = 1;
            if(win == PLAYER_WHITE){
                DrawTexture(t_white_win, 900, CHESSBOARD_START_Y,
WHITE);
            } else {
                DrawTexture(t_black_win, 900, CHESSBOARD_START_Y,
WHITE);
            }
            DrawTexture(t_pressSpace2Continue, CHESSBOARD_START_X, 900,
WHITE);
        }
EndDrawing();
```

- 鼠标和键盘事件

```
// 鼠标坐标映射
mouse_x = GetMouseX();
mouse_y = GetMouseY();
selected_x = (mouse_x - piece_start_x) / GAP_LENGTH;
selected_y = (mouse_y - piece_start_y) / GAP_LENGTH;
// selected是图像层的操作
re_x = selected_y;  // 坐标映射
re_y = selected_x;
if(IsMouseButtonPressed(MOUSE_BUTTON_MIDDLE)){
    print_debug_msg(gobang);
}
if(!is_game_end){
    if(gobang->now_player == *ai_piece && (player_move_success == 1
        // 该电脑了
        ai_move(gobang);
        win = is_win(gobang);
    }
    if(IsMouseButtonPressed(MOUSE_BUTTON_LEFT)){
        // 按下鼠标
        // 落子
        player_move_success = abs(player_move(gobang, re_x, re_y));
        // 判断是否获胜
        win = is_win(gobang);
    }
    if(IsKeyPressed(KEY_R)){
        // 撤回
        if(!(mode == 1 && gobang->steps->sp == 1)){
            // AI先手第一步不撤回
            if(mode != 0){
                // 非玩家局面多撤回一次
                take_back(gobang);
            }
```

```
            take_back(gobang);
        }
    } else if (IsKeyPressed(KEY_S)){
        // 在游戏未下子时方可切换
        mode ++;
        mode %= 3;
        if(mode == 0){
            // 玩家对垒
            *ai_piece = -0;
        } else if(mode == 1){
            // AI 执黑棋
            *ai_piece = -1;    // 切换AI棋子， 重新开始
        } else if (mode == 2){
            // AI 执白棋
            *ai_piece = 11;
        }
        while (!gobang->steps->is_empty(gobang->steps)){
            take_back(gobang);
        }
        is_game_end = 0;
        win = -100;
    }
}
if(is_game_end && IsKeyPressed(KEY_SPACE)){
    while (!gobang->steps->is_empty(gobang->steps)){
        take_back(gobang);
    }
    is_game_end = 0;
    win = -100;
}
```

- 五子棋AI和UI结合

```
extern void
gobang_gui(Gobang *gobang, int (*player_move)(Gobang *gobang, int x, int y),
void (*ai_move)(Gobang *gobang), int *ai_piece);
```

接收AI下棋方法和玩家下棋方法

AI下棋方法，调用alpha-beta剪枝搜索函数获得最佳位置

```
void ai_move(Gobang *gobang){
    long start = clock();
    if(gobang->steps->is_empty(gobang->steps)){
        best_pos[0] = 7;
        best_pos[1] = 7;
    } else {
        init_score(gobang);
        first_search(gobang, MAX_DEPTH, -INFINITY, INFINITY - 1, ai);
    }
    long end = clock();
    printf("\ntime: %ld ms\n", end - start);
    if(best_pos[0] == -1){
        printf("[ERROR] FAILED TO SEARCH THE ANSWER!!!\n");
        best_pos[0] = 14;
        best_pos[1] = 14;
```

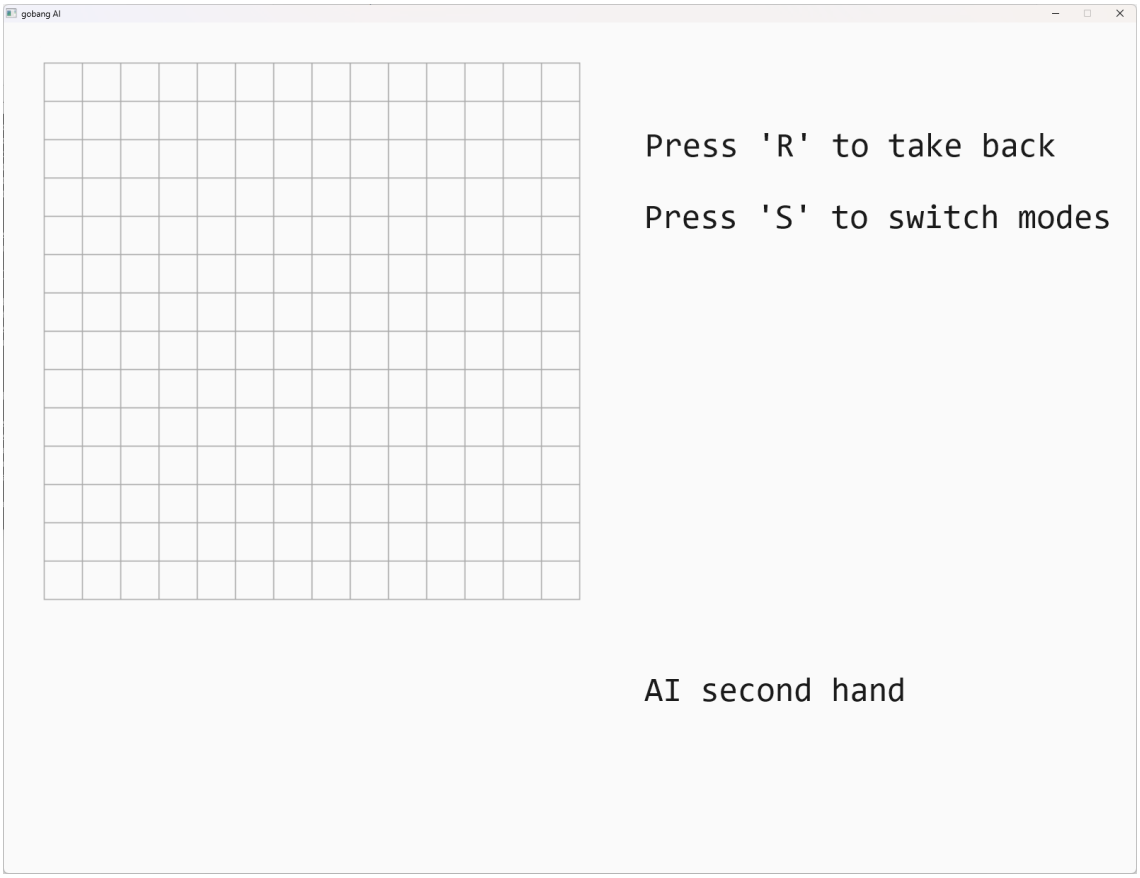
```
    } else {
        fall(gobang, best_pos[0], best_pos[1]);
    }
    best_pos[0] = -1;
    best_pos[1] = -1;
}
```

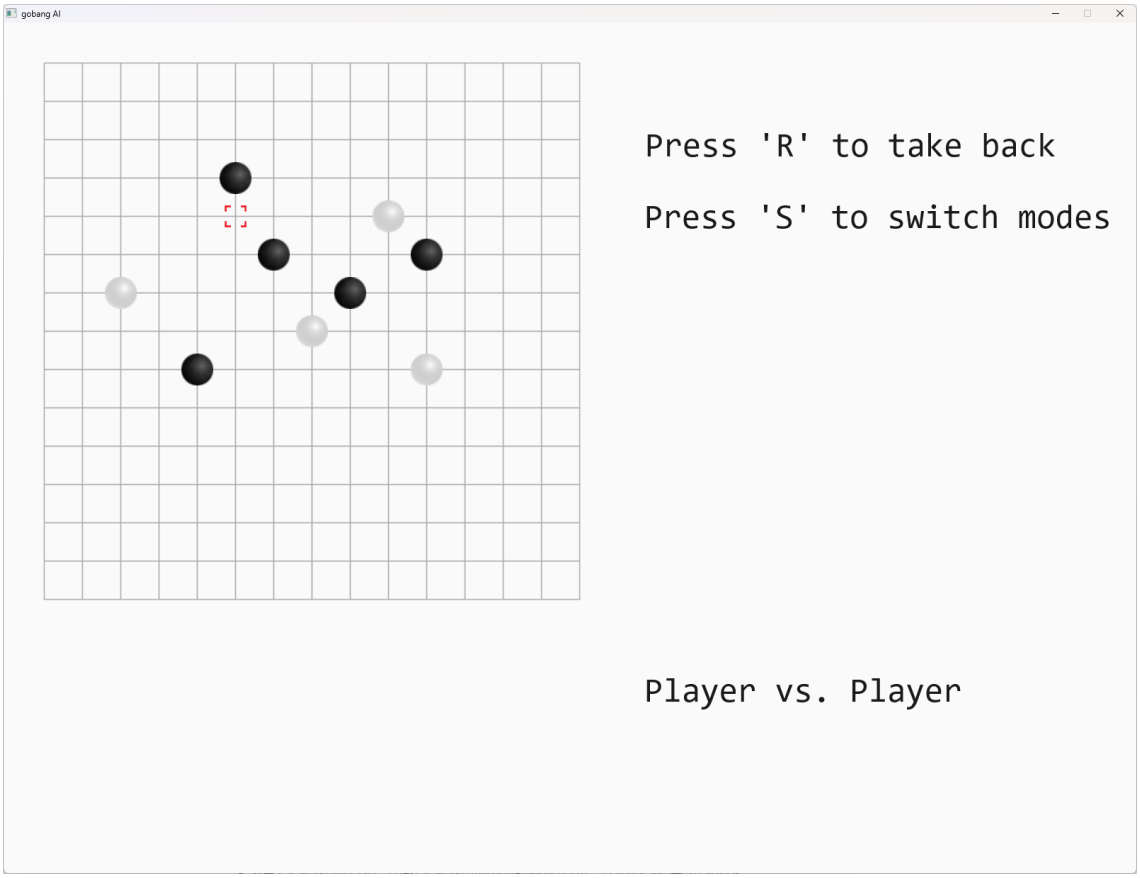
玩家下棋方法

```
int player_move(Gobang *gobang, int x, int y){
    return fall(gobang, x, y);
}
```

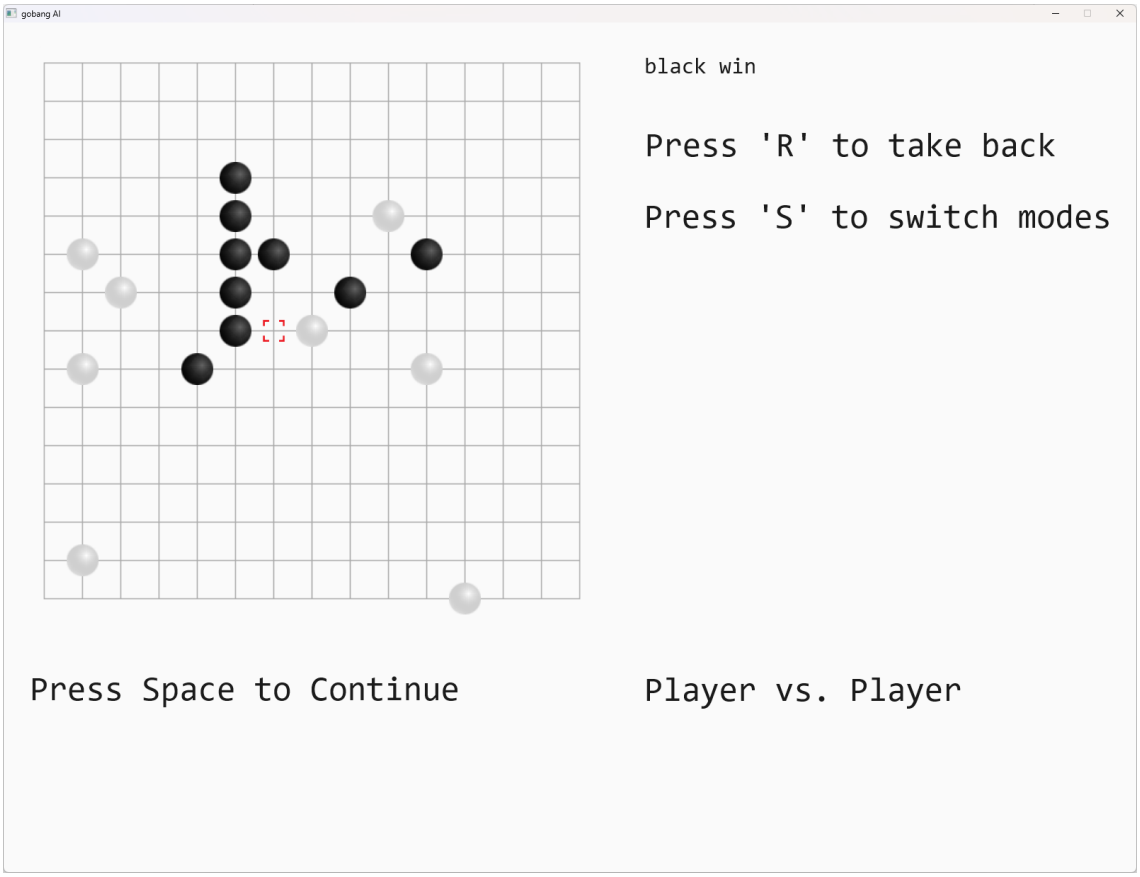
四、实验结果与分析

- UI界面展示





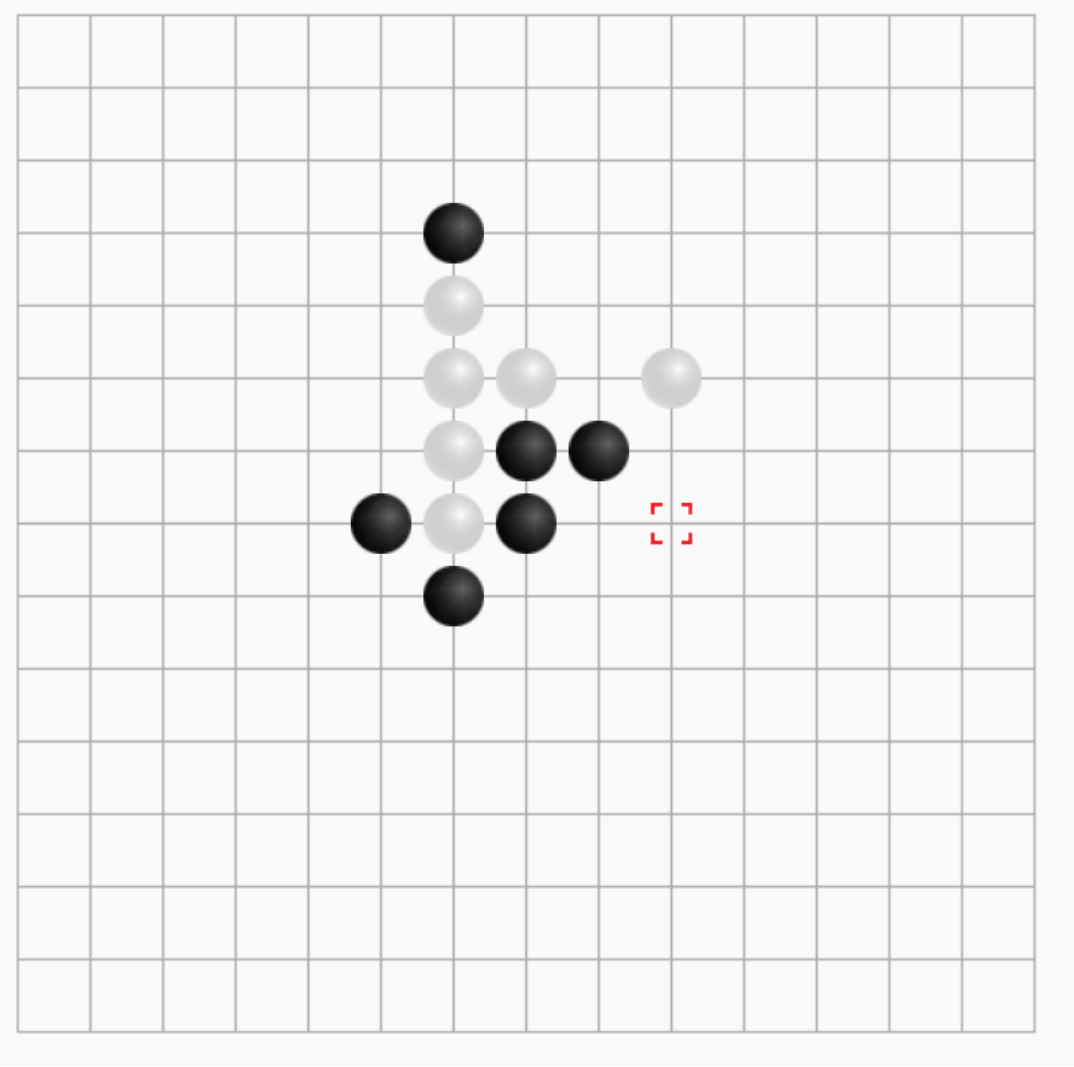
获胜



- AI行为分析
  - AI决策过程

    首先对棋盘进行初始化的打分。

    然后进行启发式搜索



如在该情况下，生成着法后进行启发式搜索，发现能连成四的子和连成三的子等，然后在该处落子，生成新的着法，更新棋盘中棋子的分数，遍历着法继续落子，直到达到最大层，此时结束遍历，使用`evaluate`函数得到此时棋盘的分，随后返回上一步，继续遍历，遍历完所有着法后依据Max/Min层选择最大/最小值，开始剪枝...

- AI性能评估

  在搜索四层的情况下，平均一步耗时3s，性能堪忧，搜索6层一步耗时20s以上，是一个无法接收的时间

- AI策略的有效性

  由于遍历了所有子附近两格的位置，致使几乎所有可能AI都会考虑到，故而策略有效性较高，但由于搜索层数不深，棋力低下

- 错误和异常行为的分析

  - 特殊情况下，能连五时AI不会选择连五

    暂且不明

  - 会下在一些匪夷所思的位置

    分数设置不合理，某处落子的分数影响了距离较远的子的分数

  - 能形成冲四时AI大概率会去形成冲四

    分数设计不合理

五、实验总结

- 学习体会

通过这个实验，深入理解了启发式搜索和alpha-beta剪枝的原理和应用。也学习了如何使用C语言和相关库来开发复杂的项目，提升了编程能力和问题解决能力

- 实验挑战

  实验挑战主要在于如何有效的实现和优化搜索算法，包括定位bug等，以及如何设计一个好的用户界面

- 未来改进

  用户界面和搜索效率上存在很大的改进空间，未来可以完成缓存机制和添加更加完善的算分机制来辅助AI判断

六、参考文献

> Issues · lihongxun945/myblog (github.com)