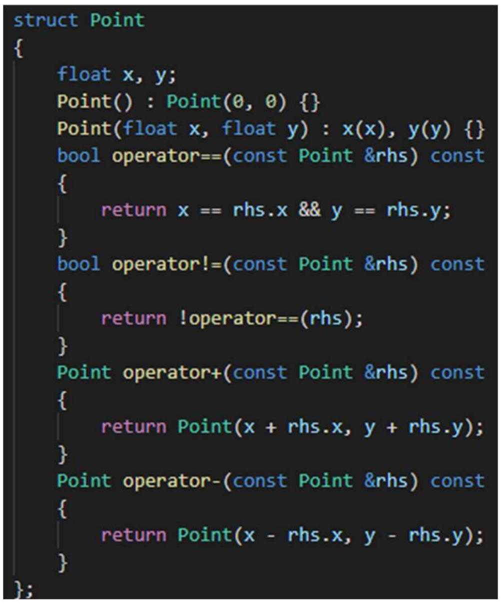
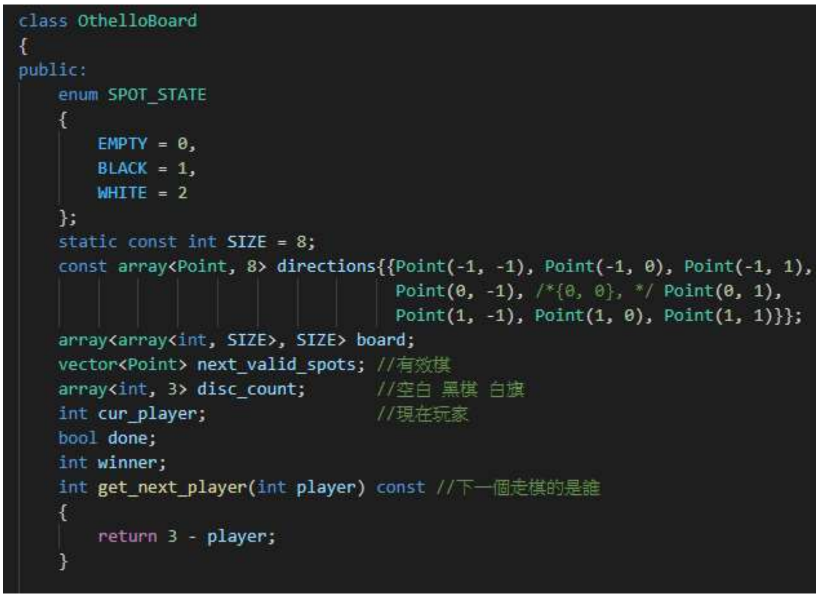
MiniProject3

在設計自己的 AI 之前，首先要先對黑白棋規則有一定的了解，隨後在瀏覽過一次 main code 之後發現裡面寫的 Point class 和 OthelloBoard class 已經為我準備了一個很好的工具供我在設計 AI 時使用。

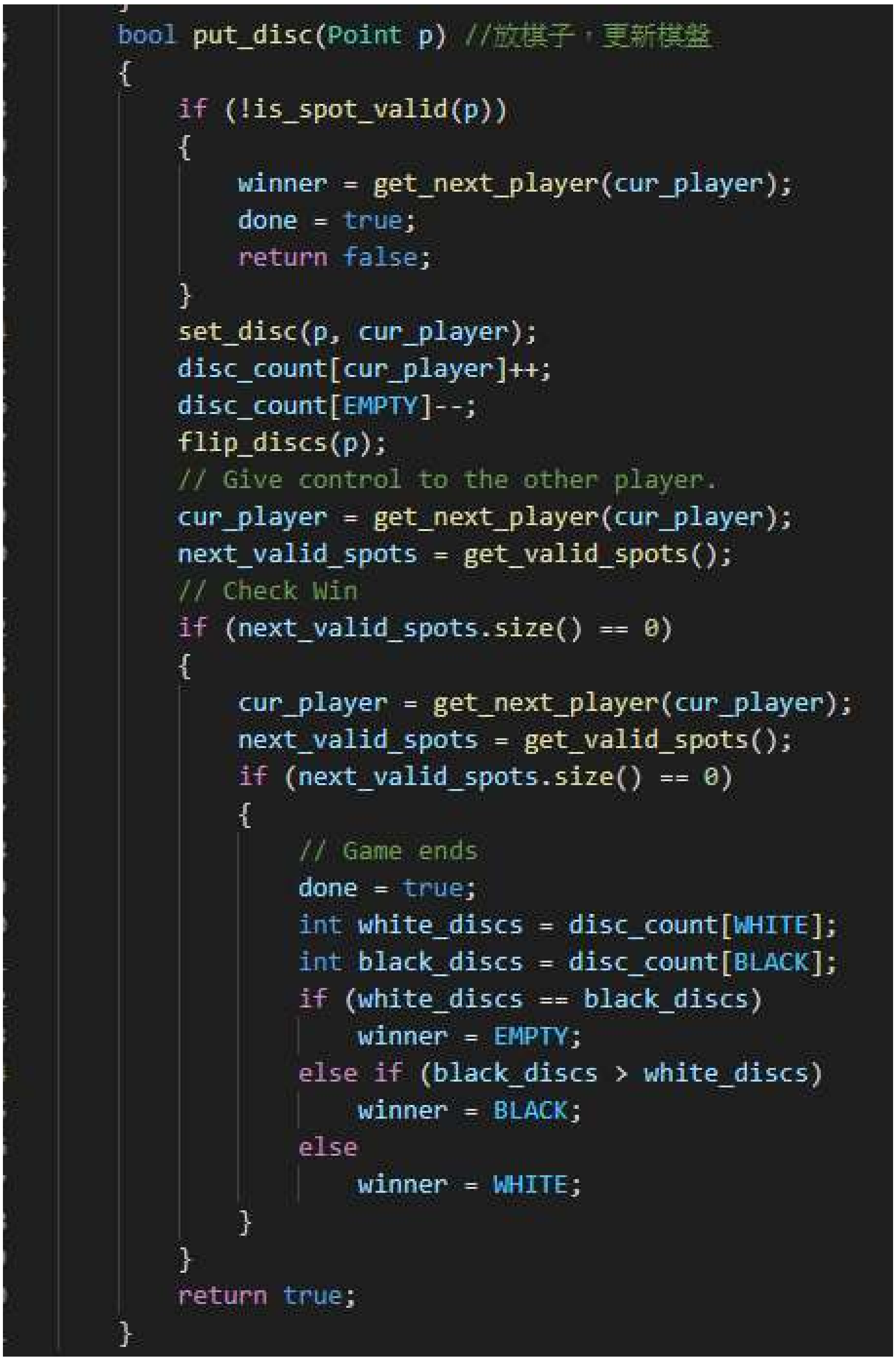


```
struct Point
{
    float x, y;
    Point() : Point(0, 0) {}
    Point(float x, float y) : x(x), y(y) {}
    bool operator==(const Point &rhs) const
    {
        return x == rhs.x && y == rhs.y;
    }
    bool operator!=(const Point &rhs) const
    {
        return !operator==(rhs);
    }
    Point operator+(const Point &rhs) const
    {
        return Point(x + rhs.x, y + rhs.y);
    }
    Point operator-(const Point &rhs) const
    {
        return Point(x - rhs.x, y - rhs.y);
    }
};
```



```
class OthelloBoard
{
public:
    enum SPOT_STATE
    {
        EMPTY = 0,
        BLACK = 1,
        WHITE = 2
    };
    static const int SIZE = 8;
    const array<Point, 8> directions{{Point(-1, -1), Point(-1, 0), Point(-1, 1),
                                      Point(0, -1), /*{0, 0}, */ Point(0, 1),
                                      Point(1, -1), Point(1, 0), Point(1, 1)}};
    array<array<int, SIZE>, SIZE> board;
    vector<Point> next_valid_spots; //有效棋
    array<int, 3> disc_count;       //空白 黑棋 白旗
    int cur_player;                 //現在玩家
    bool done;
    int winner;
    int get_next_player(int player) const //下一個走棋的是誰
    {
        return 3 - player;
    }
```

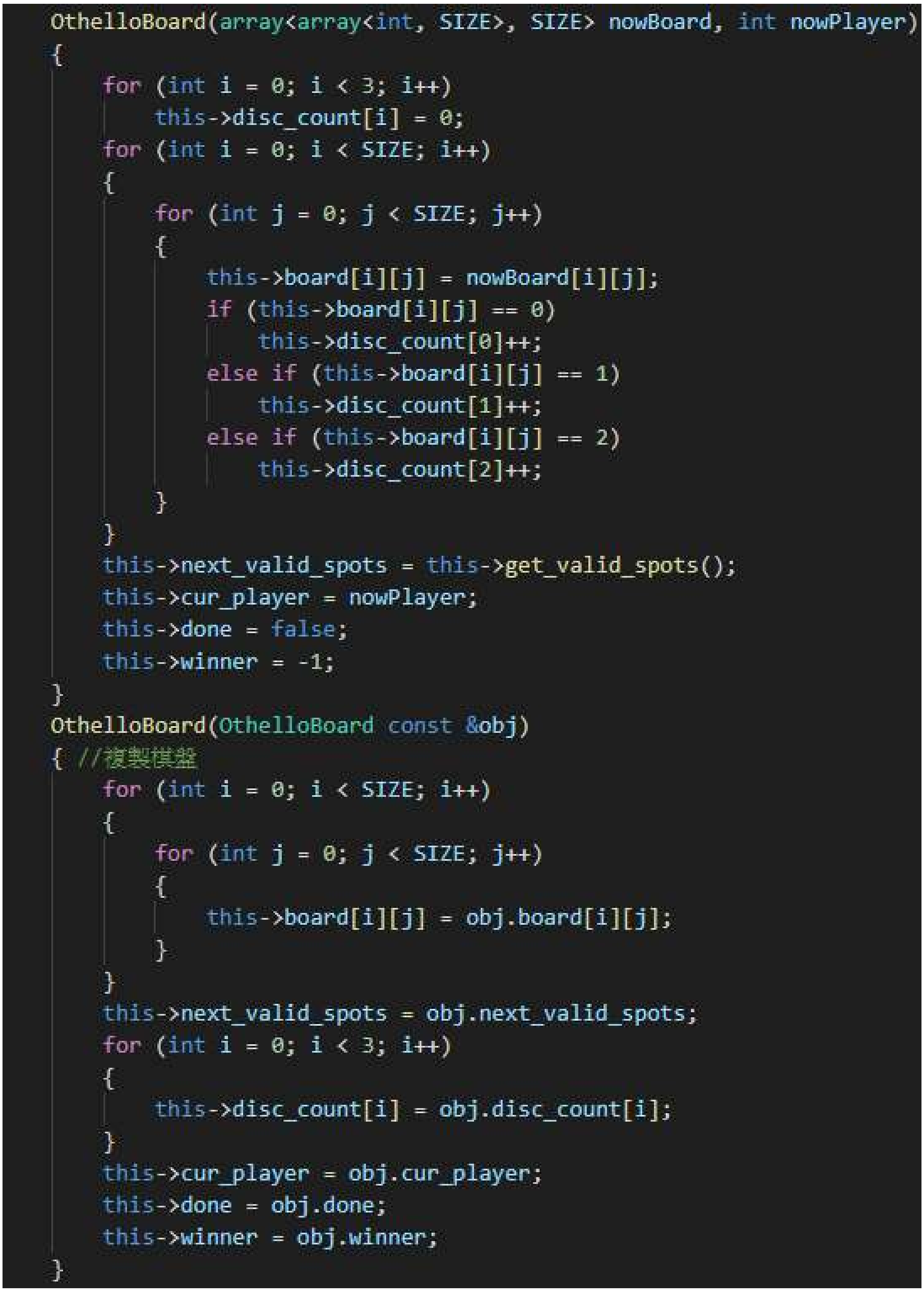
而其中最重要的莫屬於這個 function



```
bool put_disc(Point p) //放棋子，更新棋盤
{
    if (!is_spot_valid(p))
    {
        winner = get_next_player(cur_player);
        done = true;
        return false;
    }
    set_disc(p, cur_player);
    disc_count[cur_player]++;
    disc_count[EMPTY]--;
    flip_discs(p);
    // Give control to the other player.
    cur_player = get_next_player(cur_player);
    next_valid_spots = get_valid_spots();
    // Check Win
    if (next_valid_spots.size() == 0)
    {
        cur_player = get_next_player(cur_player);
        next_valid_spots = get_valid_spots();
        if (next_valid_spots.size() == 0)
        {
            // Game ends
            done = true;
            int white_discs = disc_count[WHITE];
            int black_discs = disc_count[BLACK];
            if (white_discs == black_discs)
                winner = EMPTY;
            else if (black_discs > white_discs)
                winner = BLACK;
            else
                winner = WHITE;
        }
    }
    return true;
}
```

這個 function 可以做翻棋子和更新棋盤，幾乎做完了下棋所需要的全部事情，所以我只需要再寫兩個建構子使得設計 AI 的時候可以實時更新棋盤屬性

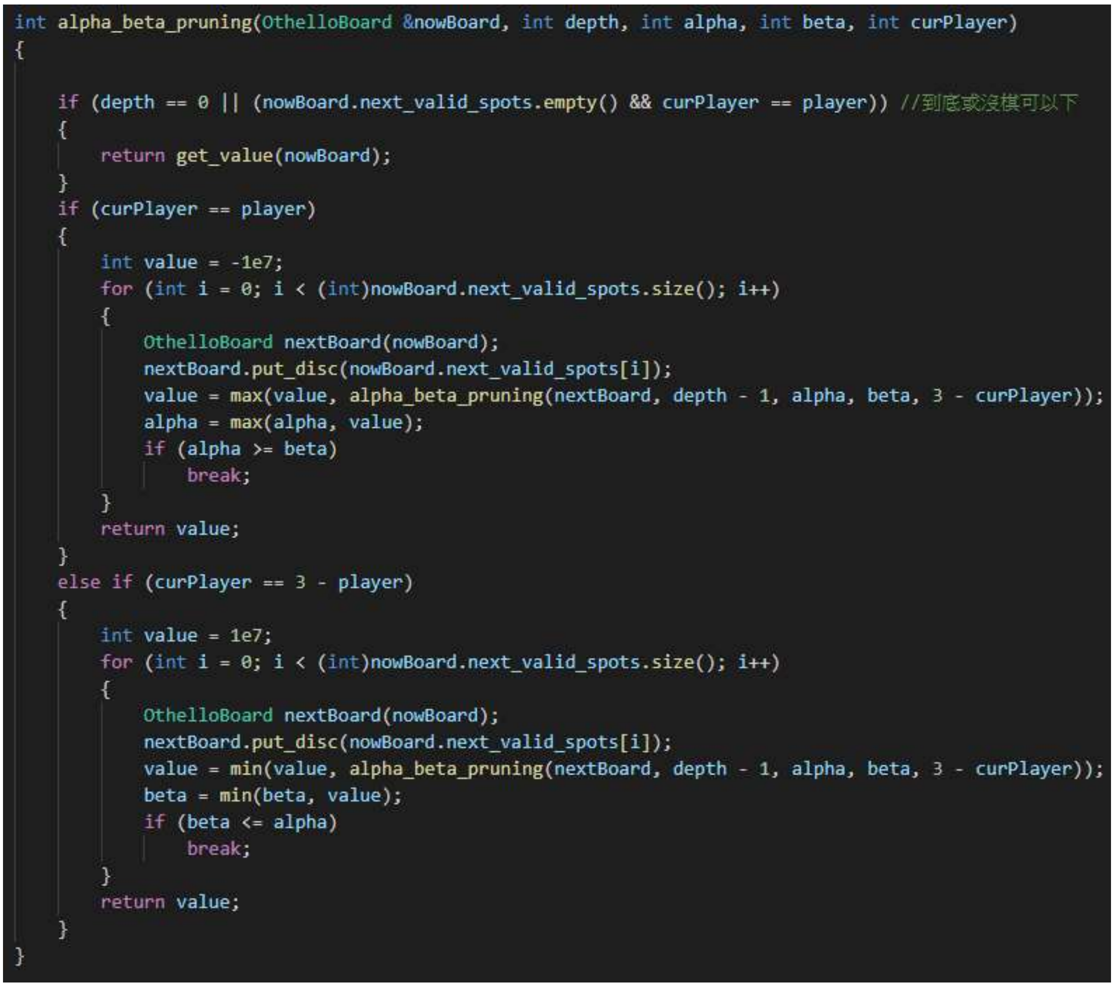
做好，可以直接開始設計AI所需要的function。首先要實作的便是



```cpp
OthelloBoard(array<array<int, SIZE>, SIZE> nowBoard, int nowPlayer)
{
    for (int i = 0; i < 3; i++)
        this->disc_count[i] = 0;
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            this->board[i][j] = nowBoard[i][j];
            if (this->board[i][j] == 0)
                this->disc_count[0]++;
            else if (this->board[i][j] == 1)
                this->disc_count[1]++;
            else if (this->board[i][j] == 2)
                this->disc_count[2]++;
        }
    }
    this->next_valid_spots = this->get_valid_spots();
    this->cur_player = nowPlayer;
    this->done = false;
    this->winner = -1;
}
OthelloBoard(OthelloBoard const &obj)
{ //複製棋盤
    for (int i = 0; i < SIZE; i++)
    {
        for (int j = 0; j < SIZE; j++)
        {
            this->board[i][j] = obj.board[i][j];
        }
    }
    this->next_valid_spots = obj.next_valid_spots;
    for (int i = 0; i < 3; i++)
    {
        this->disc_count[i] = obj.disc_count[i];
    }
    this->cur_player = obj.cur_player;
    this->done = obj.done;
    this->winner = obj.winner;
}
```

隨後以player_random的code做基礎去修改，寫入和寫出文件已經做好，可以直接開始設計AI所需要的function。首先要實作的便是MinMax再加上alpha-beta-pruning根據ppt所給出的psudocode，

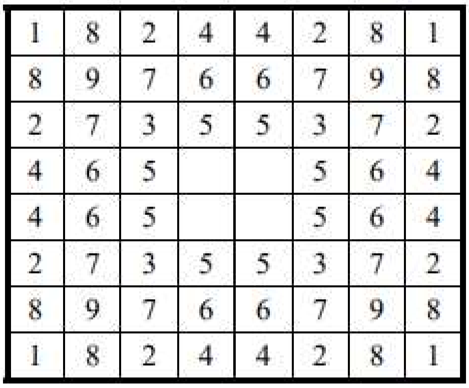
我只需要把內容修改上去並且設定搜尋深度即可



```
int alpha_beta_pruning(OthelloBoard &nowBoard, int depth, int alpha, int beta, int curPlayer)
{

    if (depth == 0 || (nowBoard.next_valid_spots.empty() && curPlayer == player)) //到底或沒棋可以下
    {
        return get_value(nowBoard);
    }
    if (curPlayer == player)
    {
        int value = -1e7;
        for (int i = 0; i < (int)nowBoard.next_valid_spots.size(); i++)
        {
            OthelloBoard nextBoard(nowBoard);
            nextBoard.put_disc(nowBoard.next_valid_spots[i]);
            value = max(value, alpha_beta_pruning(nextBoard, depth - 1, alpha, beta, 3 - curPlayer));
            alpha = max(alpha, value);
            if (alpha >= beta)
                break;
        }
        return value;
    }
    else if (curPlayer == 3 - player)
    {
        int value = 1e7;
        for (int i = 0; i < (int)nowBoard.next_valid_spots.size(); i++)
        {
            OthelloBoard nextBoard(nowBoard);
            nextBoard.put_disc(nowBoard.next_valid_spots[i]);
            value = min(value, alpha_beta_pruning(nextBoard, depth - 1, alpha, beta, 3 - curPlayer));
            beta = min(beta, value);
            if (beta <= alpha)
                break;
        }
        return value;
    }
}
```

在寫完alpha-beta-pruning後便要來實作計算value的function才可以使這個函式可以完整運作。根據網上查找到的資料

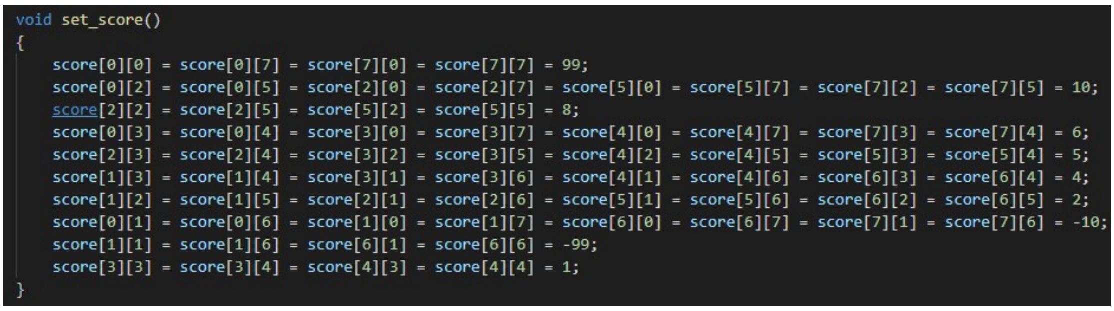


1	8	2	4	4	2	8	1
8	9	7	6	6	7	9	8
2	7	3	5	5	3	7	2
4	6	5			5	6	4
4	6	5			5	6	4
2	7	3	5	5	3	7	2
8	9	7	6	6	7	9	8
1	8	2	4	4	2	8	1

圖 3-13 走子優先順序

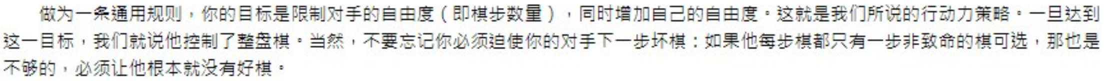
我將棋盤格給定分數好讓 AI 計算每一步的收益，而分數高低則透過

黑白棋的一些通用法則來設定，例如：角落為穩定子很重要，C-square 為角落相鄰的兩格危險性較高不建議先下，X-square 為角落斜相鄰的格危險性極高不能先下，除了這兩個 square 之外的邊緣優先度較高（爬邊）。所以得出以下棋盤分數

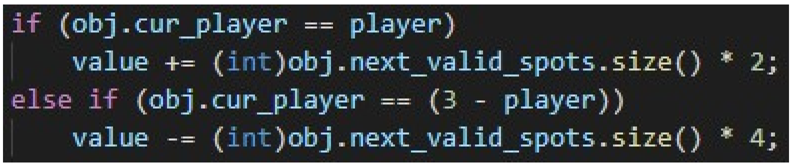


```
void set_score()
{
    score[0][0] = score[0][7] = score[7][0] = score[7][7] = 99;
    score[0][2] = score[0][5] = score[2][0] = score[2][7] = score[5][0] = score[5][7] = score[7][2] = score[7][5] = 10;
    score[2][2] = score[2][5] = score[5][2] = score[5][5] = 8;
    score[0][3] = score[0][4] = score[3][0] = score[3][7] = score[4][0] = score[4][7] = score[7][3] = score[7][4] = 6;
    score[2][3] = score[2][4] = score[3][2] = score[3][5] = score[4][2] = score[4][5] = score[5][3] = score[5][4] = 5;
    score[1][3] = score[1][4] = score[3][1] = score[3][6] = score[4][1] = score[4][6] = score[6][3] = score[6][4] = 4;
    score[1][2] = score[1][5] = score[2][1] = score[2][6] = score[5][1] = score[5][6] = score[6][2] = score[6][5] = 2;
    score[0][1] = score[0][6] = score[1][0] = score[1][7] = score[6][0] = score[6][7] = score[7][1] = score[7][6] = -10;
    score[1][1] = score[1][6] = score[6][1] = score[6][6] = -99;
    score[3][3] = score[3][4] = score[4][3] = score[4][4] = 1;
}
```

但單單給棋盤設置分值不足以應付千變萬化的棋盤，所以在網上查找黑白棋攻略的時候便又學到了幾手下棋方法可以提高獲勝幾率——行動力



做为一条通用规则，你的目标是限制对手的自由度（即棋步数量），同时增加自己的自由度。这就是我们所说的行动力策略。一旦达到这一目标，我们就说他控制了整盘棋。当然，不要忘记你必须迫使你的对手下一步坏棋：如果他每步棋都只有一步非致命的棋可选，那也是不够的，必须让他根本就没有好棋。

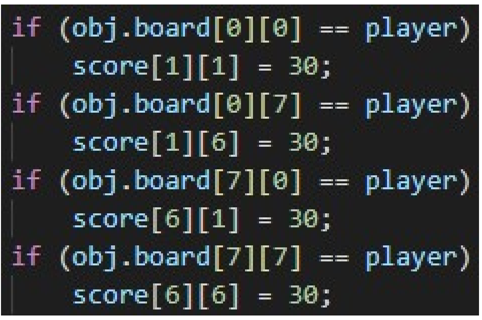


```
if (obj.cur_player == player)
    value += (int)obj.next_valid_spots.size() * 2;
else if (obj.cur_player == (3 - player))
    value -= (int)obj.next_valid_spots.size() * 4;
```

如果能限制對方能下點的數量並且提高我方自由度，那麼棋盤的局面會比較明朗。

而透過這些策略，我的 AI 已經基本可以打敗 baseline 前 4，但最難的 baseline5 還需要一點小技巧。此時我便再添加一個角對角佔領策

略



```
if (obj.board[0][0] == player)
    score[1][1] = 30;
if (obj.board[0][7] == player)
    score[1][6] = 30;
if (obj.board[7][0] == player)
    score[6][1] = 30;
if (obj.board[7][7] == player)
    score[6][6] = 30;
```

如果先佔領了角落，那麼其鄰近的 X-square 點會從危險點變成一個重要的戰略點，可以鞏固邊角使得爬邊戰術可以收益更大。

在做完角對角佔領之後，便可以輕鬆的擊敗 baseline5，此次的 project 也告一段落。