1) Project Description

1-1) Program Flow Chart

判斷 my color

↓

將現在局勢的 board 去跑 4 層的 minimax algorithm

↓

得到可下有效棋最終不同的局面

↓

利用 board evaluator 去加權判斷局勢

↓

回推決定要下什麼棋

1-2) Detailed Description

```
Point* get_valid_orbs(Board* board, char color, int* count)
```

→ 將此局面下的有效棋紀錄於 Point* valid_orbs

```
void create_new_board(Board* newboard, Point p, char color, Board* board)
```

→ 將新的棋子加入後生成 newboard

```
int minimax(Board* board, int depth, int alpha, int beta, bool
isMaximizingPlayer, Point* ret_p)
```

→ 基本上參照 wiki 的 pseudocode

```
function minimax(node, depth, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, minimax(child, depth − 1, FALSE))
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min(value, minimax(child, depth − 1, TRUE))
        return value
```

```
(* Initial call *)
minimax(origin, depth, TRUE)
```

只有為了能回傳 point 所以 Max()跟 min()的部分做了調整。

2) Screen Shots

2-1) Partial Implemented Code

```cpp
#define MAX_INF     1000000000

char my_color;
char opponet_color;
Player red_player(RED);
Player blue_player(BLUE);

You, 2 hours ago | 1 author (You)
struct Point{
    int row, col;
    Point() {}
    Point(int row, int col) : row(row), col(col) {}
};
```

```cpp
int check_weight(Board* board){
    int weight = 0;
    for(int row=0;row<5;row++) {
        for(int col=0;col<6;col++) {
            char c = board->get_cell_color(row,col);
            int cap = board->get_cell_color(row,col);
            int num = board->get_orbs_num(row,col);

            if(c==RED) {
                switch(cap-num) {
                    case 1: weight += 10000; break;
                    case 2: weight += 1000; break;
                    default:
                        weight += 100; break;
                }
            } else if(c==BLUE) {
                switch(cap-num) {
                    case 1: weight -= 10000; break;
                    case 2: weight -= 1000; break;
                    default:
                        weight -= 100; break;
                }
            }
        }
    }
    return weight;
}
```

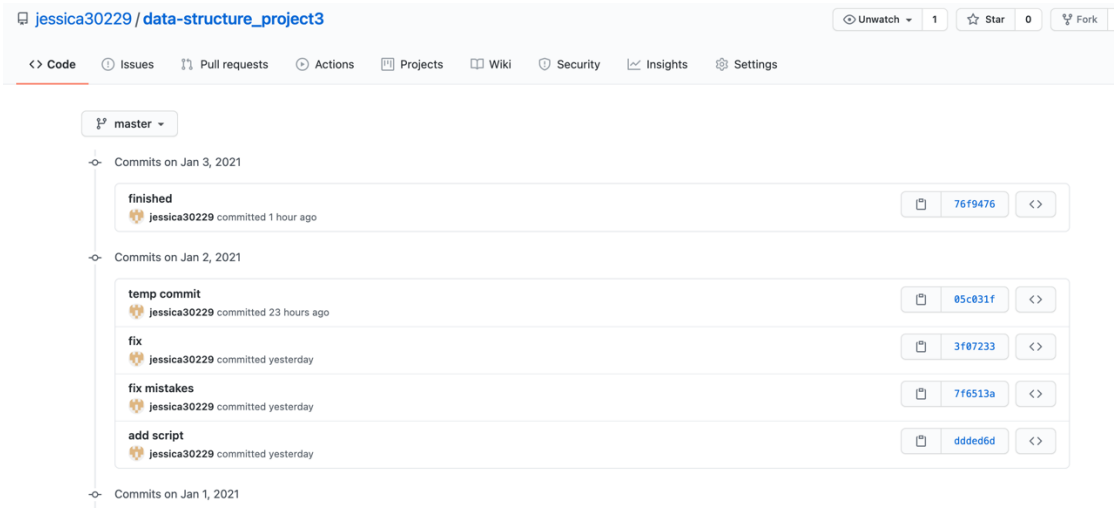
```cpp
Point* get_valid_orbs(Board* board, char color, int* count){
    Point* valid_orbs = new Point[30];
    int idx = 0;
    for(int row = 0; row < 5; row++){
        for(int col = 0; col < 6; col++){
            if(board->get_cell_color(row, col) == color||board->get_cell_color(row,col) == 'w')
                valid_orbs[idx++] = Point(row,col);
        }
    }
    *count = idx;
    return valid_orbs;
}

void create_new_board(Board* newboard, Point p, char color, Board* board){
    *newboard = *board;

    Player player('w');
    if(color == 'r') player = red_player;
    else player = blue_player;
    newboard->place_orb(p.row, p.col, &player);
}
```

```
int minimax(Board* board, int depth, int alpha, int beta, bool isMaximizingPlayer, Point* ret_p){
    int v, new_v, count;

    if(depth==0) {
        ret_p = 0;
        return check_weight(board);
    }

    if(isMaximizingPlayer) {
        v = -MAX_INF;
        Point* valid_orbs = get_valid_orbs(board, my_color, &count);
        Point p, which_p;
        for(int i=0;i<count;i++) {
            Board nboard;
            // place new orb
            create_new_board(&nboard, valid_orbs[i], my_color, board);
            // find next best orb
            new_v = minimax(&nboard, depth-1, alpha, beta, false, &which_p);

            if(new_v>=v) {
                v = new_v;
                p = valid_orbs[i];
            }
            if(alpha>=v) {
                alpha = v;
            }
            if(alpha>=beta)
                break;
        }
        delete valid_orbs;
        *ret_p=p;
        return v;
```

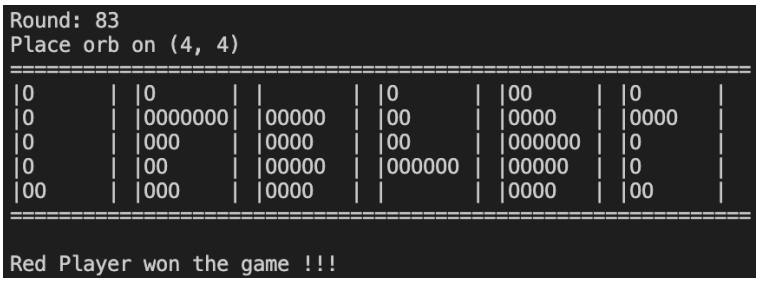
```
    } else {
        v = +MAX_INF;
        Point* valid_orbs = get_valid_orbs(board, opponet_color, &count);
        Point p, which_p;
        for(int i=0;i<count;i++) {
            Board nboard;
            // place new orb
            create_new_board(&nboard, valid_orbs[i], opponet_color, board);
            // find next best orb
            new_v = minimax(&nboard, depth-1, alpha, beta, true, &which_p);
            if(new_v<=v) {
                v = new_v;
                p = valid_orbs[i];
            }
            if(beta<=v) {
                beta = v;
            }
            if(beta<=alpha)
                break;
        }
        delete valid_orbs;
        *ret_p=p;
        return v;
    }
}
```

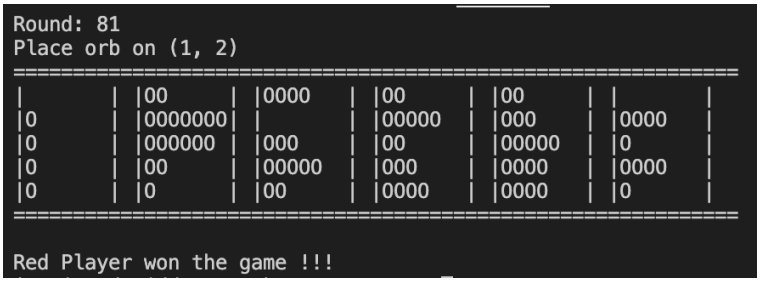
```
void algorithm_A(Board board, Player player, int index[]){
    //////your algorithm design//////////
    if(player.get_color() == 'r') {my_color = 'r'; opponet_color = 'b';}
    else {my_color = 'b'; opponet_color = 'r';}

    Point p;

    minimax(&board, 4, -MAX_INF, +MAX_INF, true, &p);

    index[0] = p.row;
    index[1] = p.col;
}
```

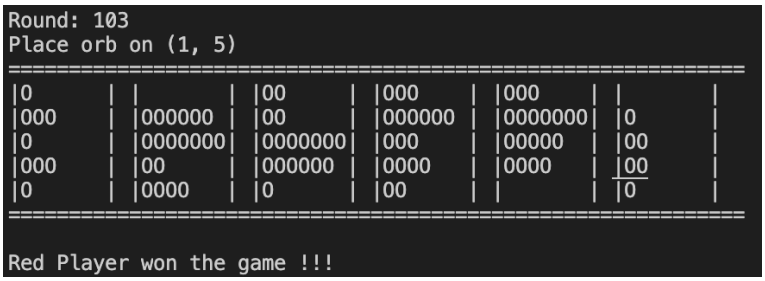
2-2) GitHub Control History

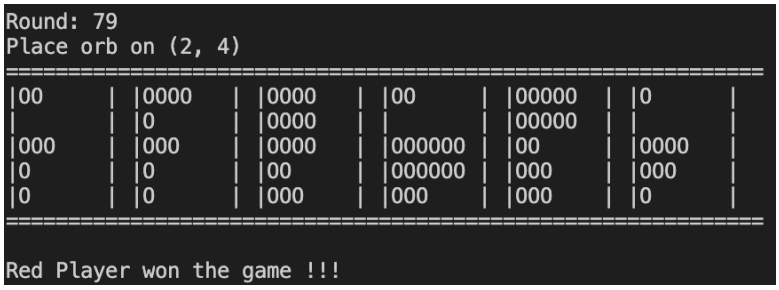


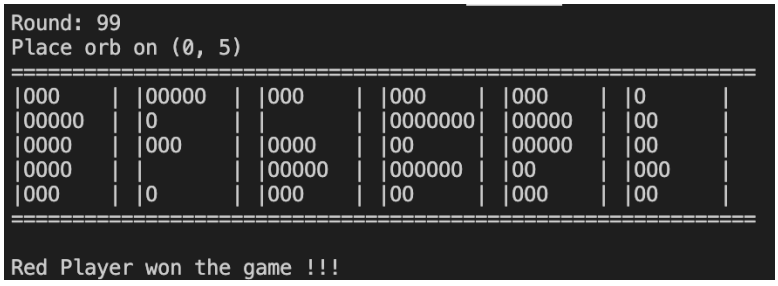
2-3) Compare with TA's AI Code (*random Move*) for 7 results. (7 pictures)

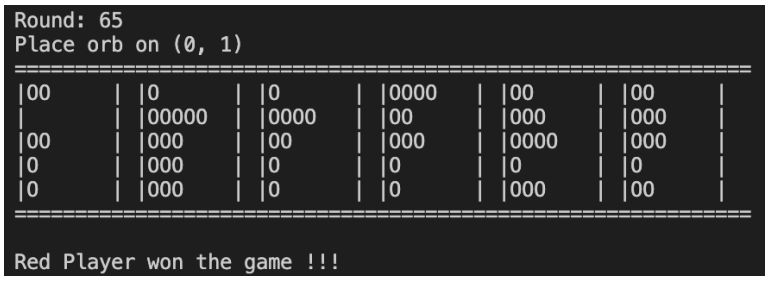


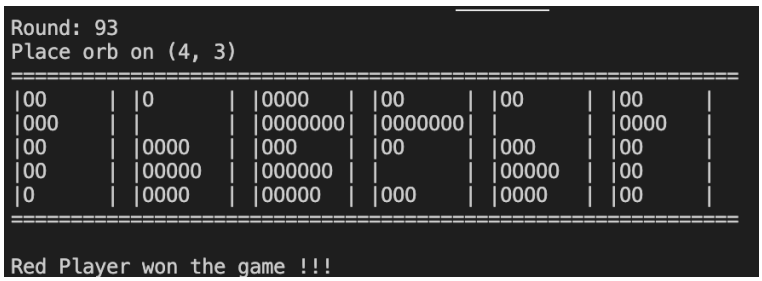












2-4) Describe the reason why you win TA's AI Code or why you can't win TA's AI Code.

基本做法就是利用 minimax 的 game tree 加上 board evaluator。

而 board evaluator 內，是簡單利用是否到的達到 explode 的差異數。

若是對手快到達數量，則減少 weight，反之，則加重 weight。

因此可以明顯知道，此作法能夠包圍隨機的 *random Move*。

以下提供利用 script 測出的勝率：

```
jessica@daviddeMacBook-Pro source % ./match.sh
Compiling
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
red won
30 games
Red wins 30 1
Blue wins 0 0
```