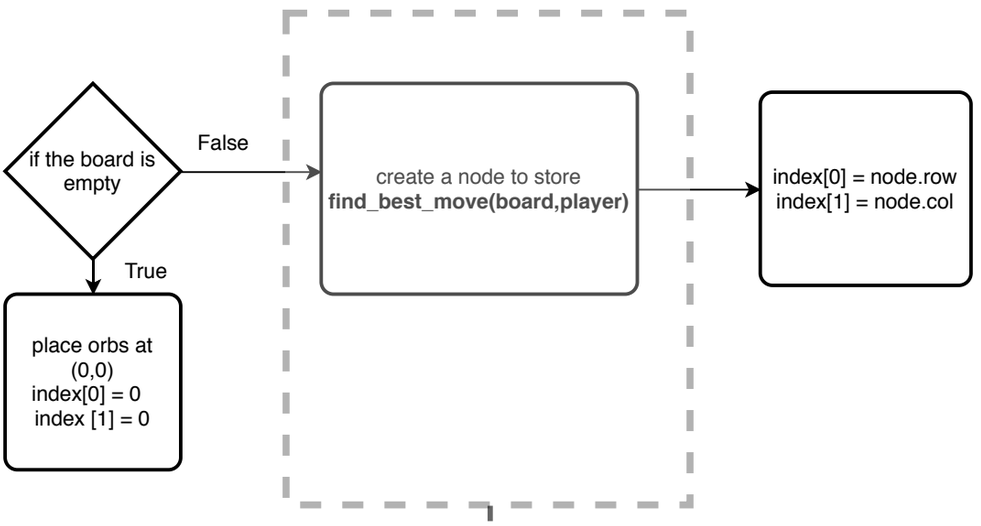
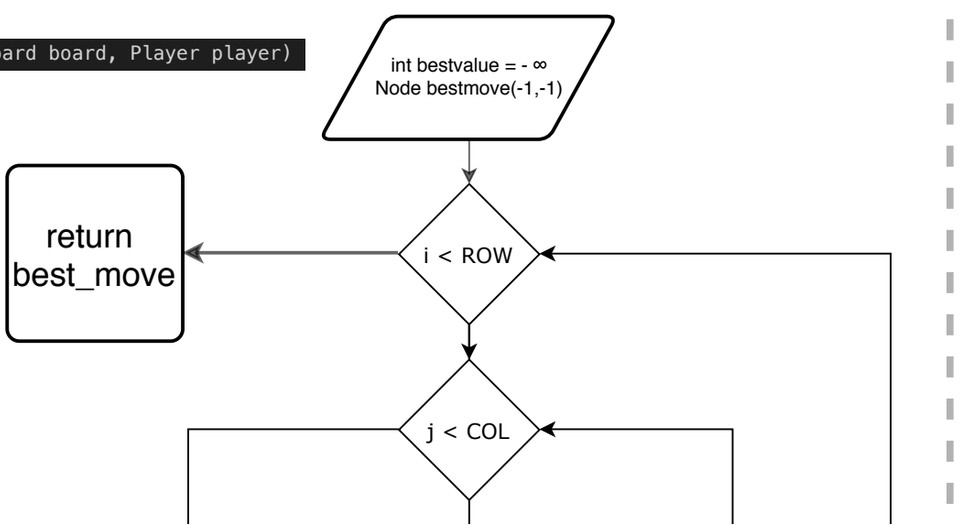
Project_3 Description
1-1)Flow Chart

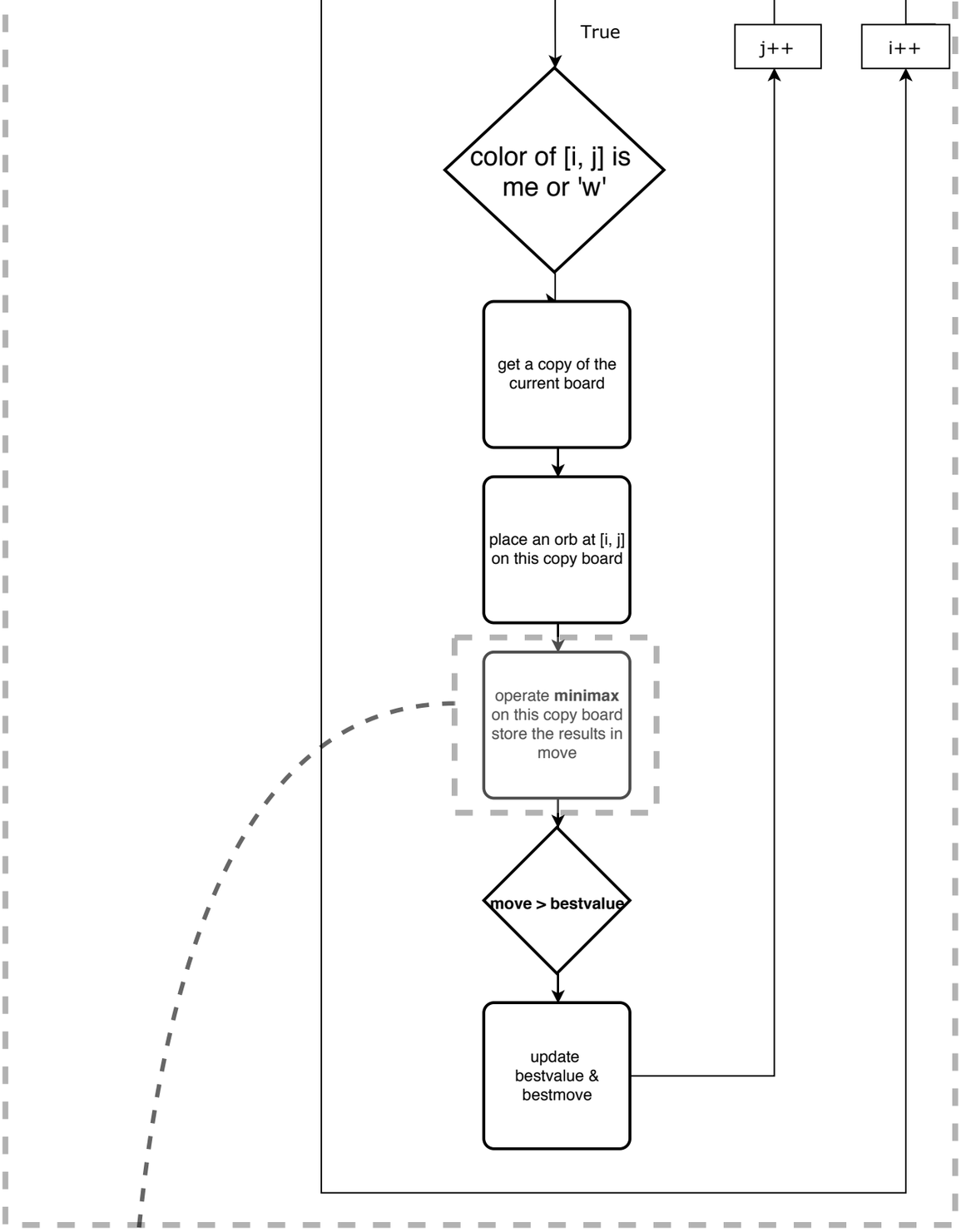
學號 1073007S
姓名 葉致廷

```
void algorithm_A(Board board, Player player, int index[])
```

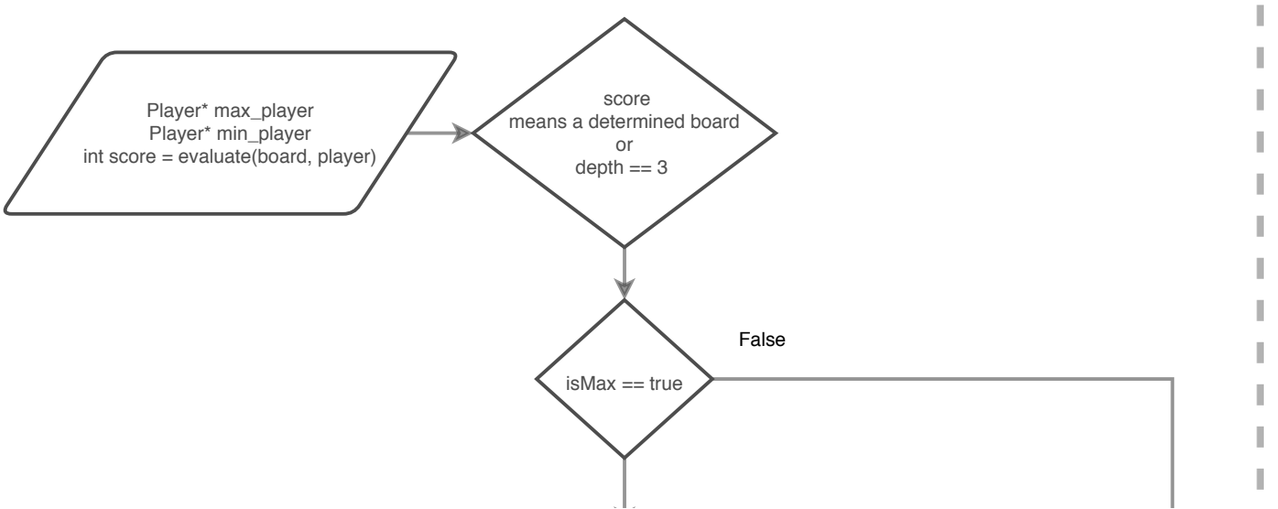


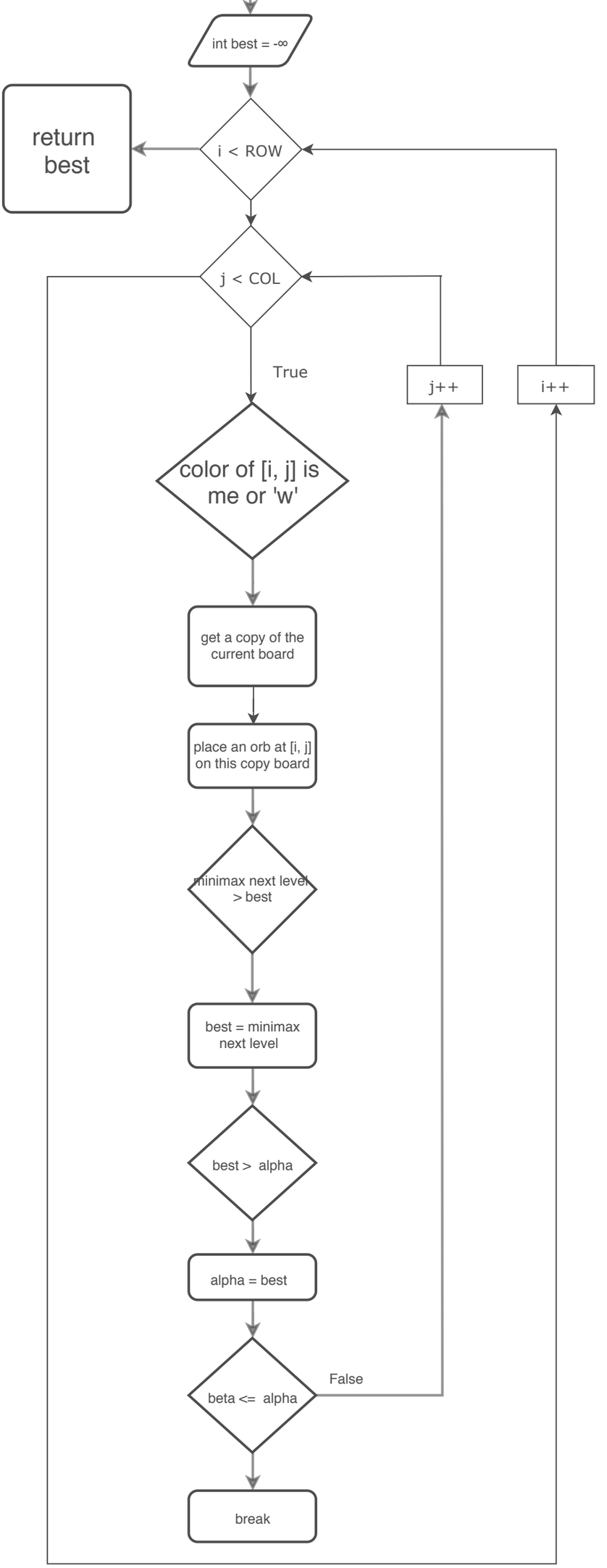

```
Node find_best_move(Board board, Player player)
```

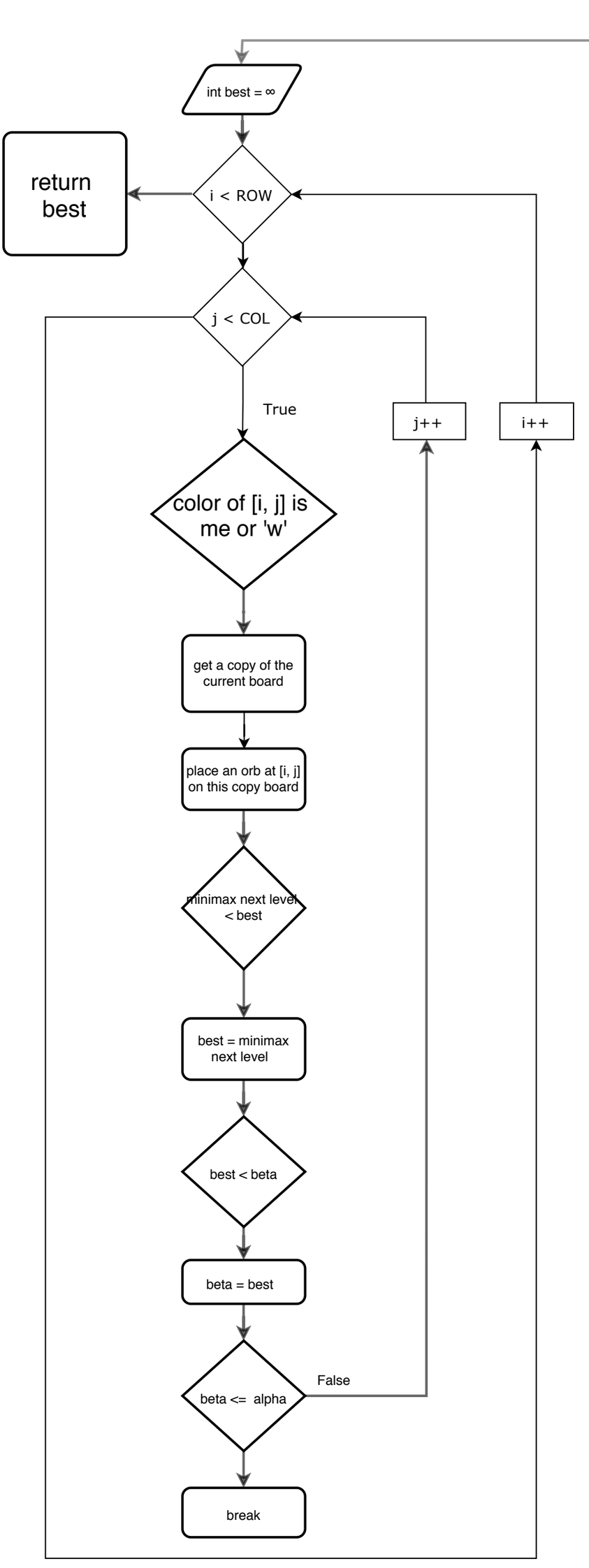





```
int minimax(Board board, int depth, Player player, Player opponent, int alpha, int beta, bool isMax)
```




int best = -∞
return best
i < ROW
j < COL
True
j++
i++
color of [i, j] is me or 'w'
get a copy of the current board
place an orb at [i, j] on this copy board
minimax next level > best
best = minimax next level
best > alpha
alpha = best
beta <= alpha
False
break


int best = ∞
return best
i < ROW
j < COL
True
j++
i++
color of [i, j] is me or 'w'
get a copy of the current board
place an orb at [i, j] on this copy board
minimax next level < best
best = minimax next level
best < beta
beta = best
beta <= alpha
False
break

1-2)

Detailed Description:

struct Node:

- stores the indices of a certain position.

Board copy(Board board):

- copy an identical board as the given parameter. Used in minimax function to implement “do and undo”.

bool is_enemy(Board board, int i, int j, Player player):

- check whether position [i, j] is an enemy.

bool is_critical(Board board, int i, int j, Player player):

- check whether position [i, j] is about to explode.

int* find_contiguous(Board board, Player player):

- evaluate how each critical position of ours is connected to other critical cells of ours. The larger the contiguous chain is formed the higher the score this position gets. As for how to find a contiguous chain, use BFS to link the contiguous critical position and add one to the score simultaneously.

int neighbor_evaluate(Board board, int i, int j, Player player):

- this function measures how strong the adjacent enemies are. Since the order of importance of the positions is corner > line > middle, 1. we subtract 5 minus the critical mass of that cell from the value, which means the stronger(more likely to blow) a cell is, the less likely it will be chosen. 2. Also, if the cell has no critical adjacent enemies, we add the critical mass minus the number of orbs of a cell to make sure the one that is further from an explosion will be considered first. We use a variable to record the result and return it.

i**nt position_aug(Board board, int i, int j, Player player)**:

- this function determines how strong a cell is with the following criteria: corner > line = about to explode > middle and returns the value accordingly.

int evaluate(Board board, Player player):

- this function determines how strong a certain board is for us. It consists of three major calculations: 1. the summation of the value of neighbor_evaluate + position_aug of all our cells 2. the differentiating factor: since it is an important factor, it shall be multiplied with 2 3.the chaining factor: add twice the number of cells in the block to the score. The final score is the summation of these 3 factors.
- In addition, when we win, return 100000, on the other hand, when the enemy win, return -100000. The higher the score, the better the chance a certain board is likely to lead to a win.

bool cutoff(int evaluate):

- In case the parameter is either 100000 or -100000, which implies that a certain path of the game tree has a result and we should stop going to the next level of the tree. as placing orbs on a result will lead to an infinite explosion, we should definitely be aware of it and cut it off right away.

int minimax(Board board, int depth, Player player, Player opponent, int alpha, int beta, bool isMax):

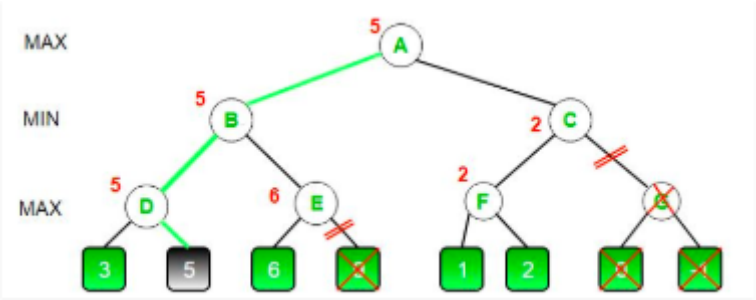
- how this function works is shown in the flowchart. To check whether or not the current move is better than the best move we take the help of **minimax()** function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally. We will focus on the idea of minimax and alpha-beta pruning.

Idea:

Pseudocode :

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

- It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.



Node find_best_move(Board board, Player player):

- This function evaluates all the available moves using **minimax()** and then returns the best move the maximizer can make.

2)**Screenshots**

2-1)

algorithm_A.cpp

```
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include "../include/algorithm.h"
using namespace std;
#define INFINITY 1000000
#define WIN 100000
#define LOSE -WIN

/*****************************************************
* In your algorithm, you can just use the the funcitons
* listed by TA to get the board information.(functions
* 1. ~ 4. are listed in next block)
*
* The STL library functions is not allowed to use.
*****************************************************/

/*************************************************************************
* 1. int board.get_orbs_num(int row_index, int col_index)
* 2. int board.get_capacity(int row_index, int col_index)
* 3. char board.get_cell_color(int row_index, int col_index)
* 4. void board.print_current_board(int row_index, int col_index, int round)
*
* 1. The function that return the number of orbs in cell(row, col)
* 2. The function that return the orb capacity of the cell(row, col)
* 3. The function that return the color fo the cell(row, col)
* 4. The function that print out the current board statement
```

```
**********************************************************************/


//Data Sturcture <stack>
template <class T>
class stack{
   private:
       struct Node{
           T data;
           Node* next_node;
           Node(T data = 0):data(data){}
           Node(T data, Node* next):data(data),next_node(next){}
       };
       Node* topNode;
       int Size;
   public:
       stack(): Size(0),topNode(0){}
       void push(T element);
       int size();
       T top();
       bool empty();
       void pop();
};
template <class T>
int stack<T>::size(){
   return this->Size;
}
template <class T>
bool stack<T>::empty(){
   if(this->size() == 0){
       return true;
   }
   return false;
}
template <class T>
void stack<T>::push(T n){
   if(empty()){
       Node* newnode = new Node(n,0);
       this->topNode = newnode;
       this->Size++;
       return;
   }
   Node* newnode = new Node(n,topNode);//point to the current top address
```

```
    topNode = newnode; //Update the top address to newnode
    this->Size++;
}
template <class T>
void stack<T>::pop() {
    if(empty()) return;
    Node* deletenode = topNode;
    topNode = topNode->next_node;
    delete deletenode;
    deletenode = 0;
    Size--;
}
template <class T>
T stack<T>::top(){
    return topNode->data;
}
//endof <stack>

struct Node {
    int row;
    int col;
    Node(int i, int j):row(i),col(j){}
};

Board copy(Board board) {
    int record[ROW][COL];
    for(int i = 0; i < ROW; i++) {
        for(int j = 0; j < COL; j++) {
            record[i][j] = board.get_orbs_num(i,j);
        }
    }
    char player[ROW][COL];
    for(int i = 0; i < ROW; i++) {
        for(int j = 0; j < COL; j++) {
            player[i][j] = board.get_cell_color(i,j);
        }
    }
    Board copy;
    for(int i = 0; i < ROW; i++) {
        for(int j = 0; j < COL; j++) {
            for(int k = 0; k < record[i][j]; k++) {
                Player* play = new Player(player[i][j]);
```

```
                //cout<<"problem in copy"<<endl;
                copy.place_orb(i, j, play);
                delete play;
            }
        }
    }

    return copy;
}

bool is_enemy(Board board,int i, int j, Player player) {
    bool enemy = false;
    // is_enemy:= not player, 'w' excluded
    if(board.get_cell_color(i,j) != player.get_color() && board.get_cell_color(i,j)
!= 'w')
        enemy = true;
    return enemy;
}

bool is_critical(Board board,int i, int j) {
    return board.get_orbs_num(i,j) == (board.get_capacity(i,j)-1);
}

int* find_contiguous(Board board, Player player) {
    bool table[ROW][COL];
    for(int i = 0; i < ROW; i++) {
        for(int j = 0; j < COL; j++) {
            table[i][j] = false;
            if(board.get_cell_color(i,j) == player.get_color()  &&
is_critical(board,i,j)) {
                table[i][j] = true;
            }
        }
    }
    int* chain = new int[ROW*COL];
    for(int i = 0; i < ROW*COL; i++) {
        chain[i] = 0;
    }
    int num = 0;
    for(int i = 0; i < ROW; i++) {
        for(int j = 0; j < COL; j++) {
            if(table[i][j]) {
```

```
                int l = 0;
                Node node(i,j);
                stack<Node> s;
                // bfs search
                while(!s.empty()) {
                    Node pos = s.top();
                    int x = pos.row;
                    int y = pos.col;
                    table[x][y] = false;
                    l++;
                    if(x-1 >= 0 && table[x-1][y]) {
                        Node temp(x-1,y);
                        s.push(temp);
                    }
                    if(x+1 <= 4 && table[x+1][y]) {
                        Node temp(x+1,y);
                        s.push(temp);
                    }
                    if(y-1 >= 0 && table[x][y-1]) {
                        Node temp(x,y-1);
                        s.push(temp);
                    }
                    if(y+1 <= 5 && table[x][y+1]) {
                        Node temp(x,y+1);
                        s.push(temp);
                    }
                }
                chain[num] = l;
                num++;
            }
        }
    }
    return chain;
}

int neighbor_evaluate(Board board, int i, int j, Player player) {
    int adj_val = 0;
    if( i-1 >= 0 && is_enemy(board, i-1, j, player) && is_critical(board,i-1, j)) {
        adj_val = adj_val - (5 -board.get_capacity(i-1,j)) + (
board.get_capacity(i-1,j) - board.get_orbs_num(i-1,j) );
    }
    if(i+1 <= 4 && is_enemy(board, i+1, j, player) && is_critical(board, i+1, j)) {
```

```
        adj_val = adj_val - (5 -board.get_capacity(i+1,j)) + (
board.get_capacity(i+1,j) - board.get_orbs_num(i+1,j) );
    }
    if( j-1 >= 0 && is_enemy(board, i, j-1, player) && is_critical(board, i, j-1)) {
        adj_val = adj_val - (5 -board.get_capacity(i,j-1)) + (
board.get_capacity(i,j-1) - board.get_orbs_num(i,j-1) );
    }
    if(j+1 <= 5 && is_enemy(board, i, j+1, player) && is_critical(board, i, j+1)) {
        adj_val = adj_val - (5 -board.get_capacity(i,j+1)) + (
board.get_capacity(i,j+1) - board.get_orbs_num(i,j+1) );
    }
    return adj_val;
}

int position_aug(Board board, int i, int j, Player player) {
    int aug = 0;
    //corner
    if(board.get_capacity(i,j) == 2) {
        aug = 4;
    }
    //line
    else if(board.get_capacity(i,j) == 3) {
        aug = 3;
    }
    //middle
    else if(board.get_capacity(i,j) == 4) {
        aug = 0;
    }
    //about to explode
    if(is_critical(board, i, j)) {
        aug += 3;
    }
    return aug;
}

// Need to be accurate :((
int evaluate(Board board, Player player) {
    int score =0;
    int my_count = 0;
    int enemy_count = 0;
    // Parameter 1
    for(int i = 0; i < ROW; i++) {
```

```
        for(int j = 0; j < COL; j++) {
            // if the cell is me
            if(board.get_cell_color(i,j) == player.get_color()) {
                my_count += board.get_orbs_num(i,j);
                int neighbor = neighbor_evaluate(board, i, j, player);
                score += neighbor;
                // corner or not, line or not
                if(neighbor == 0) score += position_aug(board, i, j, player);
            }
            //enemy
            else {
                enemy_count += board.get_orbs_num(i,j);
            }
        }
    }
    // check if someone wins
    if(enemy_count == 0 && my_count != 0){
        return 100000;
    }
    if(enemy_count != 0 && my_count == 0){
        return -100000;
    }
    // if no one wins
    // Parameter 2
    int count_differ = my_count - enemy_count;
    score += 2*count_differ;
    // Parameter 3
    int* chain = find_contiguous(board, player);
    for(int i = 0; i < ROW*COL; i++) {
        if(chain[i] > 1) {
            score += 2*chain[i];
        }
    }
    return score;
}

bool cutoff(int evaluate) {
    if(evaluate == WIN || evaluate == LOSE) {
       return true;
    }
    return false;
}
```

```
int minimax(Board board, int depth, Player player, Player opponent, int alpha, int
beta, bool isMax) {
   // seperate our and oppponent's color
   char max_round = player.get_color();
   char min_round = opponent.get_color();

   Player* max_player= new Player(max_round);
   Player* min_player= new Player(min_round);

   int score = evaluate(board, player);
   // terminal state
   if(cutoff(score) || depth == 3) return score;

   //isMax : this_round is us while opponent_round is enemy
   if(isMax == true) {
       int best = -INFINITY;
       for(int i = 0; i < ROW; i++) {
           for(int j = 0; j < COL; j++) {
               // if cell can be placed by the player
               if(board.get_cell_color(i,j) == max_round ||
board.get_cell_color(i,j) == 'w') {
                   Board board_copy = copy(board);
                   board_copy.place_orb(i,j,max_player);
                   if(minimax(board_copy, depth+1, player, opponent, alpha, beta,
false) > best ) {
                       best = minimax(board_copy, depth+1, player, opponent, alpha,
beta, false);
                   }
                   if(best > alpha) {
                       alpha = best;
                   }
                   if (beta <= alpha)
                       break;
               }
           }
       }
       return best;
   }
   // isMin
   else {
       int best = INFINITY;
```

```
       for(int i = 0; i < ROW; i++) {
           for(int j = 0; j < COL; j++) {
               // if cell can be placed by the player
               if(board.get_cell_color(i,j) == min_round ||
board.get_cell_color(i,j) == 'w') {
                   Board board_copy = copy(board);
                   board_copy.place_orb(i,j,min_player);

                   if(minimax(board_copy, depth+1, player, opponent, alpha, beta,
true) < best ) {
                       best = minimax(board_copy, depth+1, player, opponent, alpha,
beta, true);
                   }
                   if(best < beta) {
                       beta = best;
                   }
                   if (beta <= alpha)
                       break;
               }
           }
       }
       return best;
   }
   // Error occurs
   return -1;
}
Node find_best_move(Board board, Player player) {
   // player_me is the Maximum player
   char me = player.get_color();
   Player* player_me = new Player(me);
   // create enemy for minimax
   char enemy;
   if(me == 'r') { enemy = 'b'; }
   else { enemy = 'r';}
   Player opponent(enemy);
   // might cause violation
   int bestvalue = -INFINITY;
   Node best_move(-1,-1);
   // we operate minimax on each of the possible placement and return the best one
(score wised) among them.
   for(int i = 0; i < ROW; i++) {
       for(int j = 0; j < COL; j++) {
```

```
            if(board.get_cell_color(i,j) == me || board.get_cell_color(i,j) == 'w'){
                // use board_copy to avoid corrupting the current board
                Board board_copy = copy(board);
                board_copy.place_orb(i,j,player_me);
                // operate minimax
                int alpha = -INFINITY;
                int beta =  INFINITY;
                int depth = 0;
                bool isMax = true;
                int move = minimax(board_copy, depth, player ,opponent, alpha, beta,
false);
                if(move > bestvalue) {
                    bestvalue = move;
                    best_move.row = i;
                    best_move.col = j;
                }
            }
        }
    }
    //cout<<"best move: [ "<<best_move.row<<", "<<best_move.col<<" ] and the best
value is: "<<bestvalue<<endl;
    return best_move;
}
void algorithm_A(Board board, Player player, int index[]){
    // we are the first one; first step get corner
    int count = 0;
    for(int i = 0; i < ROW; i++) {
        for(int j = 0; j < COL; j++) {
            count += board.get_orbs_num(i,j);
        }
    }
    if(count == 0) {
        index[0] = 0;
        index[1] = 0;
    }
    // other than the first step
    else {
        Node best = find_best_move(board, player);
        index[0] = best.row;
        index[1] = best.col;
    }
}
```

2-2)
git log --all-match

```
(base) macdeMacBook-Air-5:source mac$ git log --all-match
commit fa9fffa2db0d1626e78cbef414fa3dfd6f167ccb (HEAD -> master, origin/master, origin/HEAD)
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Tue Jan 14 16:51:41 2020 +0800

     clean code

commit 4ab48ed837814b505da622bb957846b6e6e776b8
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Tue Jan 14 16:42:26 2020 +0800

    evaluate() bug fixed aug = 0

commit 8d408273c829d690e3c926d2a24feea452b0be20
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Tue Jan 14 01:38:07 2020 +0800

    evaluate() bug fixed

commit 906f1c2c921b74dc88a1cbae46680795895111e0
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Sun Jan 12 18:01:08 2020 +0800

    the clean version

commit 1a02a341abf5fee397b45cd38636cd6152cc6346
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Sat Jan 11 20:24:57 2020 +0800

    neighbor_evaluate upgrade
```

```
commit bea7278bd18ac786582ffd45fd9efc3993158bc5
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Tue Jan 7 22:26:08 2020 +0800

    if critical: aug*2

commit 6a3637c3e927f8a35c7d3a66d71c3f2b791ea3ca
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Tue Jan 7 15:36:31 2020 +0800

    evaluation function update

commit 44e39fd2e8210dfb7c16364c89c89ef44b9333f8
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Mon Jan 6 16:30:47 2020 +0800

    -fix stl-library func {min()&max()} to if statement

commit a9664e85c84ecba22ed2e6e6c7ad6ae094e2b685
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Sat Dec 21 20:22:33 2019 +0800

    make it more efficient

commit f484505834df84a98f98b7b88ad4f4c775bc1692
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Sat Dec 21 20:09:56 2019 +0800

    bug fixed
```

```
commit eef8ab14eb940ba237c1571067088908aa9fd297
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Sat Dec 21 19:14:49 2019 +0800

    bug found: illegal placement

commit b33f0a8d1a0f63c0aa6efd2c7bafb98e12dde8c7
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Sat Dec 21 12:34:28 2019 +0800

    beating all the other versions currently

commit 80972acf3565c769aa7cd2bbd21d066b4b421595
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Fri Dec 20 14:13:48 2019 +0800


    update cut off condition in alpha-beta pruning

commit 0ba08a1bc5d56bf26ca4e8705611be122d3c0bcb
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Thu Dec 19 21:02:44 2019 +0800

    Cut-off condition in find_best_move

commit fb935e7447961bbde6431a081638d81917705f06
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Thu Dec 19 17:41:00 2019 +0800

     four levels of look-ahead
```

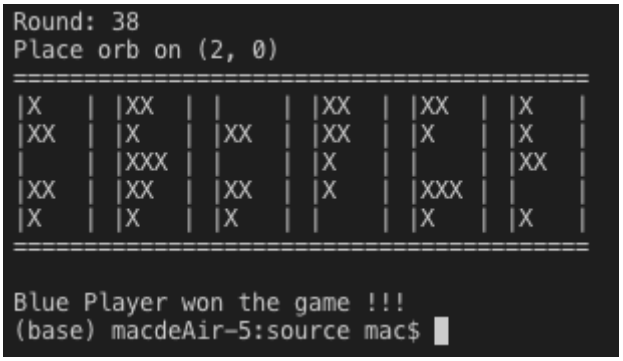
```
commit 45e13bc6b39d839bb8f616c3e3f6c4e2e373e72d
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Sat Dec 14 02:37:03 2019 +0800

    first version: able to beat random bot

commit 79d331a86e474202b1a34101fd485f22f5985c0c
Author: justinyeh <justinyeh1995@gmail.com>
Date:   Fri Dec 13 21:24:09 2019 +0800

    first version: able to beat random bot

commit 9787ea05d5ba040a7120c896331fd8b3806b73b1
Author: JustinYeh <42970023+justinyeh1995@users.noreply.github.com>
Date:   Fri Dec 13 21:20:14 2019 +0800

    Initial commit
(END)
```

git log --online

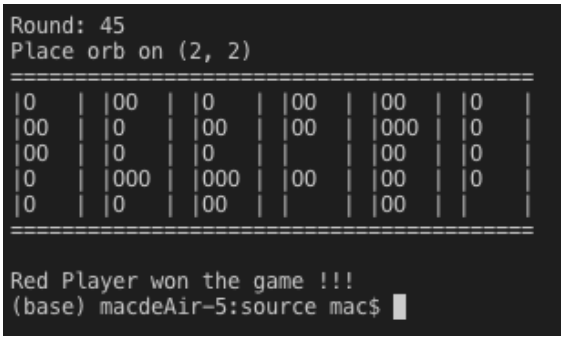
```
(base) macdeMacBook-Air-5:source mac$ git log --oneline
fa9fffa (HEAD -> master, origin/master, origin/HEAD)  clean code
4ab48ed evaluate() bug fixed aug = 0
8d40827 evaluate() bug fixed
906f1c2 the clean version
1a02a34 neighbor_evaluate upgrade
bea7278 if critical: aug*2
6a3637c evaluation function update
44e39fd -fix stl-library func {min()&max()} to if statement
a9664e8 make it more efficient
f484505 bug fixed
eef8ab1 bug found: illegal placement
b33f0a8 beating all the other versions currently
80972ac update cut off condition in alpha-beta pruning
0ba08a1 Cut-off condition in find_best_move
fb935e7  four levels of look-ahead
45e13bc first version: able to beat random bot
79d331a first version: able to beat random bot
9787ea0 Initial commit
```

2-3)
#1 **algotithm_B vs algorithm_A:**

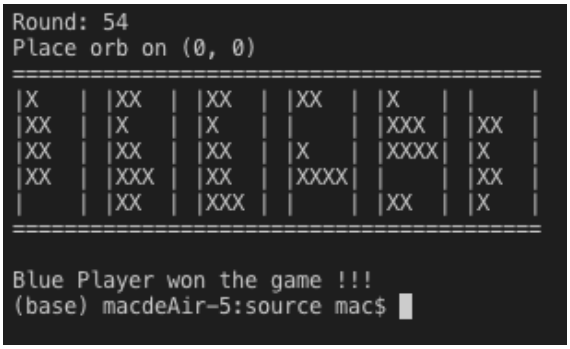
Round **1**: B goes first
results



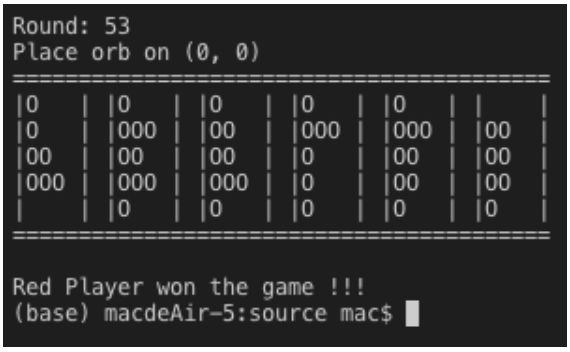
Round **2**: A goes first
results



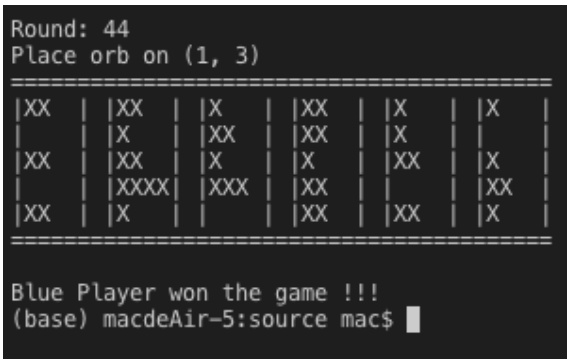
Round **3**: B goes first
results



Round **4**: A goes first
results



Round **5**: B goes first
results



Analysis:

1. **our game tree goes down to the 4th level with a very strong evaluation function,(B has no evaluation function) so it can prevent itself from any dumb move. Thus, it can surely beat algorithm_B every time.**

A: 5W0L

#2 **algotithm_C vs algorithm_A**:

Round **1**: C goes first
results

Round **2**: A goes first
results

```
Round: 5
Place orb on (0, 5)
=========================================
|O   | |    | |    | |    | |O   | |O   |
|    | |    | |    | |    | |O   | |    |
|    | |    | |    | |    | |    | |O   |
|    | |    | |    | |    | |    | |    |
|    | |    | |    | |    | |    | |    |
=========================================

Red Player won the game !!!
PS C:\Users\User\Desktop\ChainReactionFramework\source>
```

Round **3**: C goes first
results

```
Round: 42
Place orb on (0, 1)
=========================================
|X   | |X   | |XX  | |XX  | |XX  | |    |
|XX  | |X   | |X   | |XXX | |    | |XX  |
|XXX | |    | |XXX | |XX  | |XX  | |XX  |
|    | |X   | |XXX | |    | |XXX | |XX  |
|X   | |    | |    | |X   | |XX  | |    |
=========================================

Blue Player won the game !!!
PS C:\Users\User\Desktop\ChainReactionFramework\source>
```

Round **4**: A goes first
results

```
Round: 5
Place orb on (0, 5)
=========================================
|O   | |    | |    | |    | |O   | |O   |
|    | |    | |    | |    | |O   | |    |
|    | |    | |    | |    | |    | |O   |
|    | |    | |    | |    | |    | |    |
|    | |    | |    | |    | |    | |    |
=========================================

Red Player won the game !!!
PS C:\Users\User\Desktop\ChainReactionFramework\source>
```

Round **5**: C goes first
results

```
Round: 42
Place orb on (0, 1)
=========================================
|X   | |X   | |XX  | |XX  | |XX  | |    |
|XX  | |X   | |X   | |XXX | |    | |XX  |
|XXX | |    | |XXX | |XX  | |XX  | |XX  |
|    | |X   | |XXX | |    | |XXX | |XX  |
|X   | |    | |    | |X   | |XX  | |    |
=========================================

Blue Player won the game !!!
PS C:\Users\User\Desktop\ChainReactionFramework\source>
```

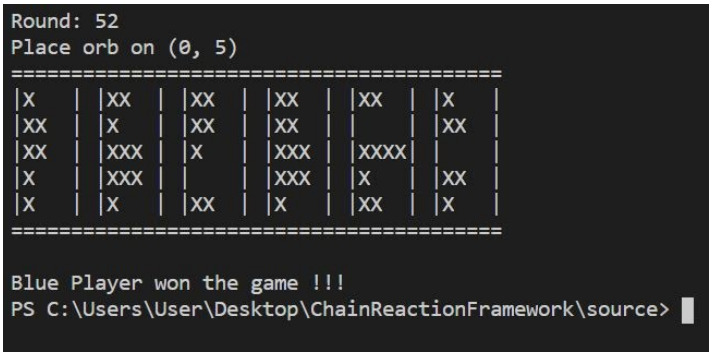
Analysis:

A: 5W0L

1. **algorithm_C has an evaluation algorithm, however, it does not consider other important factors such as “contiguous blocks” and “adjacent cells surround it”, algorithm_C might make a dumb move using this evaluation function. On the other hand, our game tree goes down to the 4th level with a very strong evaluation function, so it can prevent itself from any dumb move. Thus, it can surely beat algorithm_C every time.**

#3 **algotithm_D vs algorithm_A:**

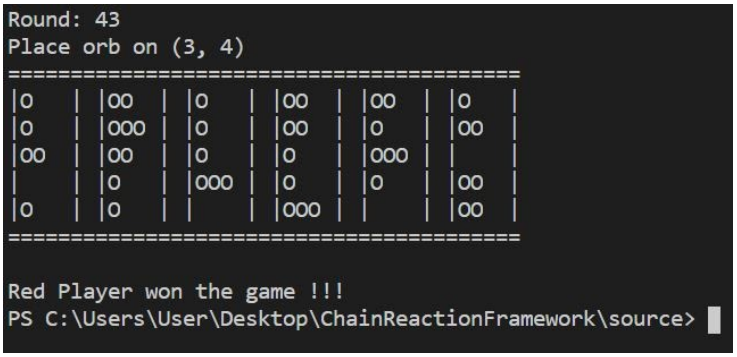
Round **1**: D goes first
results



Round **2**: A goes first
results

Round **3**: D goes first
results

Round **4**: A goes first
results



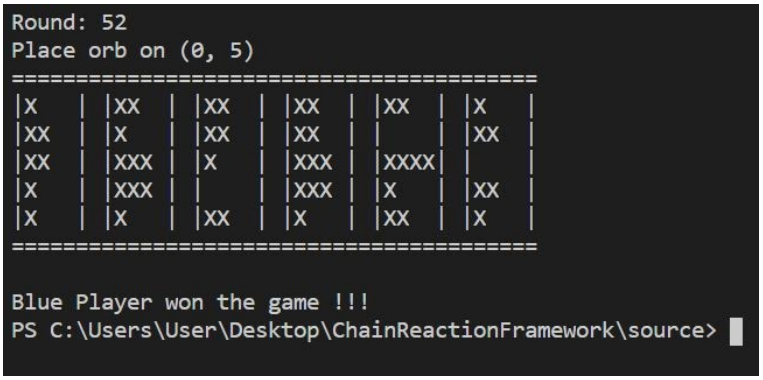
Round **5**: D goes first
results

Analysis:
A: 5W0L

1. **Though algorithm_D does go down 4 levels of the game tree, it's evaluation function simply just counts the difference of the orbs, and thus does not consider other important factors such as "contiguous blocks" and "adjacent cells surround its possible moves", it might lead to total destruction in the ending state of game tree. In contrast, our evaluation function measures each move in a more balanced manner by considering more factors.**

#4 **algotithm_E vs algorithm_A:**

Round **1**: E goes first
results



Round **2**: A goes first
results

Round **3**: E goes first
results

Round **4**: A goes first
results

Round **5**: E goes first
results

Analysis:
A: 5W0L

1. **algorithm_E has a better evaluation strategy and a 4 level go-ahead as well, however, our evaluation function considers more factors than algorithm_E. Both E and A consider the difference between orbs and their relation with opponents' orbs, yet, algorithm_A not only considers how vulnerable the surrounding enemies are but also considers the importance of these neighbors, to be more specific, following the rule of corner > line > middle, we would like to avoid critical corner cell if possible, and that's why if the neighbor enemy is a corner cell, the evaluation score would be lower etc.. Also, the function find_contiguous() helps us search the best explosion sequence, which makes the evaluation function even more credible.**

***Strangely, even with a better evaluation strategy, algorithm_E does not perform better than algorithm_D…**

To sum up, more factors considered yields a better evaluation function, and since the game tree has its limit in speed, we can't go any much deeper and therefore, the dominant difference in the game lies in the robustness of the evaluation function. In my opinion, "the contiguous chain detection" and "the difference of orbs" are the primary factors so I give them higher weights by multiple them by 2. The next important factor is related to the character of each cell and its neighbors. Can't emphasize more about the relation: corner > line > middle, we do addition and subtraction according to this rule. Consider it as a secondary factor, I did not operate multiplication on it. There should be a stronger algorithm existing, however, I believe by doing tree search with this evaluation strategy, algorithm_A shall perform better than most of the bots.