程式整體架構

```cpp
#include <iostream>
#include <fstream>
#include <cstdlib>
#include <ctime>
#include <array>
#include<set>
#include<queue>
#include<limits.h>
using namespace std;

int player;//玩家是白棋還是黑棋
const int SIZE = 15;//棋盤大小
array<array<int, SIZE>, SIZE> komoku;//棋盤

// 0: empty  ,1: black  ,2: white
enum SPOT_STATE {···

//從player_ramdom微調的
struct Point {···

bool checkSurrounding(std::array<std::array<int, SIZE>, SIZE> bd, int i, int j);

Point Best;

class State{···

//mini max alpha beta
int Minimax(State state, int depth, int Alpha, int Beta, bool maximizingPlayer);

//確認四面八方有沒有棋子(要不然會超時QQ)
bool checkSurrounding(std::array<std::array<int, SIZE>, SIZE> bd, int i, int j){···

//讀棋盤
void read_board(std::ifstream& fin) {···
```

```cpp
//找出best以及印出有用資訊
Point Next_Point(State &state){···

//決定策略
void write_valid_spot(std::ofstream& fout, State &state) {···

//遞迴
int Minimax(State state, int depth, int Alpha, int Beta, bool maximizingPlayer){···

//從ramdom抓過來用的
int main(int, char** argv) {···
```

Point(跟 ramdom 的 point 差不多)

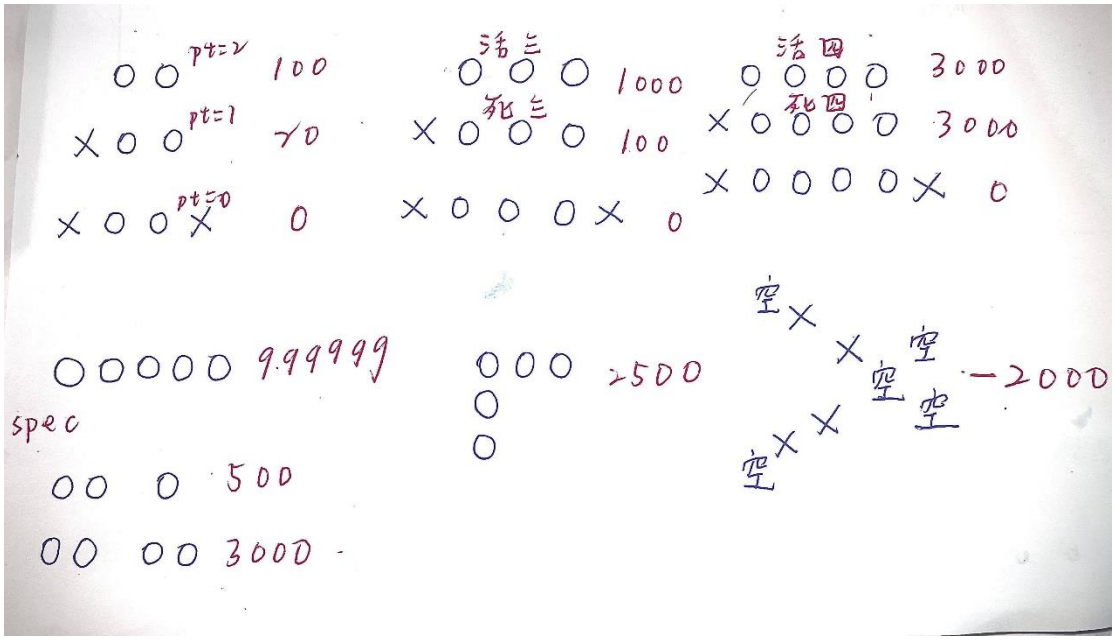
```
//從player_ramdom微調的
struct Point {
    int x, y;
    Point() : Point(0, 0) {}
    Point(float x, float y) : x(x), y(y) {}
    bool operator==(const Point& rhs) const {
        return x == rhs.x && y == rhs.y;
    }
    bool operator!=(const Point& rhs) const {
        return !operator==(rhs);
    }
    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);
    }
    bool operator<(const Point &r)const{
        if(x!=r.x) return x < r.x;
        if(y!=r.y) return y < r.y;
        return 0;
    }
};
```

State 的 class

```
class State{
public:
    array<array<int, SIZE>, SIZE> komoku;
    set<Point> candi;

    State(array<array<int, SIZE>, SIZE>a){
        for(int i=0;i<SIZE;i++){
            for(int j=0;j<SIZE;j++){
                komoku[i][j] = a[i][j];
            }
        }
    }

    State(State& tmp){
        for(int i=0;i<SIZE;i++){
            for(int j=0;j<SIZE;j++){
                komoku[i][j] = tmp.komoku[i][j];
            }
        }
    }

    //可行點
    void next_candi(){
        for(int i=0;i<SIZE;i++){
            for(int j=0;j<SIZE;j++){
                if(komoku[i][j] == 0){
                    if(checkSurrounding(komoku,i,j))
                    {
                      candi.insert(Point(i,j));
                    }
                }
            }
        }
        return;
    }
```

Evaluate_score 整體架構

```
//計算heuristic
int evaluate_score(){
    int h = 0;
    int opponent = 3 - player;
    for(int i = 0; i<SIZE; i++){
        for(int j=0; j<SIZE; j++){
            int flag1=0;
            int flag2=0;
            int flag3=0;
            if(komoku[i][j] == player){
                if(j+1<SIZE && komoku[i][j+1] == player){ …
                if(i+1<SIZE && komoku[i+1][j] == player){ …
                if(i+1<SIZE && j+1<SIZE && komoku[i+1][j+1] == player){…
                if(i+1<SIZE && j-1>=0 && komoku[i+1][j-1] == player){…
            }else if(komoku[i][j] == opponent){
                if(j+1<SIZE && komoku[i][j+1] == opponent){ …
                if(i+1<SIZE && komoku[i+1][j] == opponent){ …
                if(i+1<SIZE && j+1<SIZE && komoku[i+1][j+1] == opponent){…
                if(i+1<SIZE && j-1>=0 && komoku[i+1][j-1] == opponent){…
                //spec
                if(i+3<SIZE&&i-1>=0&&j+3<SIZE&&j-3>=0)…
            }
            else if(komoku[i][j]==0)…
            if(flag1>1)…
            if(flag2>1)…
            if(flag3>1)…

        }
    }
    return h;
}
```



不同情形給盤面評分
敵我的加權不一
在實戰中改善特殊情況
同一個點上的線越多 flag1 越大
同一個點上自己的活 3 越多 flag2 越大
同一個點上敵方的活 3 越多 flag3 越大

main

```
//從ramdom抓過來用的
int main(int, char** argv) {
    std::ifstream fin(argv[1]);
    std::ofstream fout(argv[2]);
    read_board(fin);
    State start(komoku);
    write_valid_spot(fout, start);
    fin.close();
    fout.close();
    return 0;
}
```

跟 ramdom 的差不多

checksurrounding

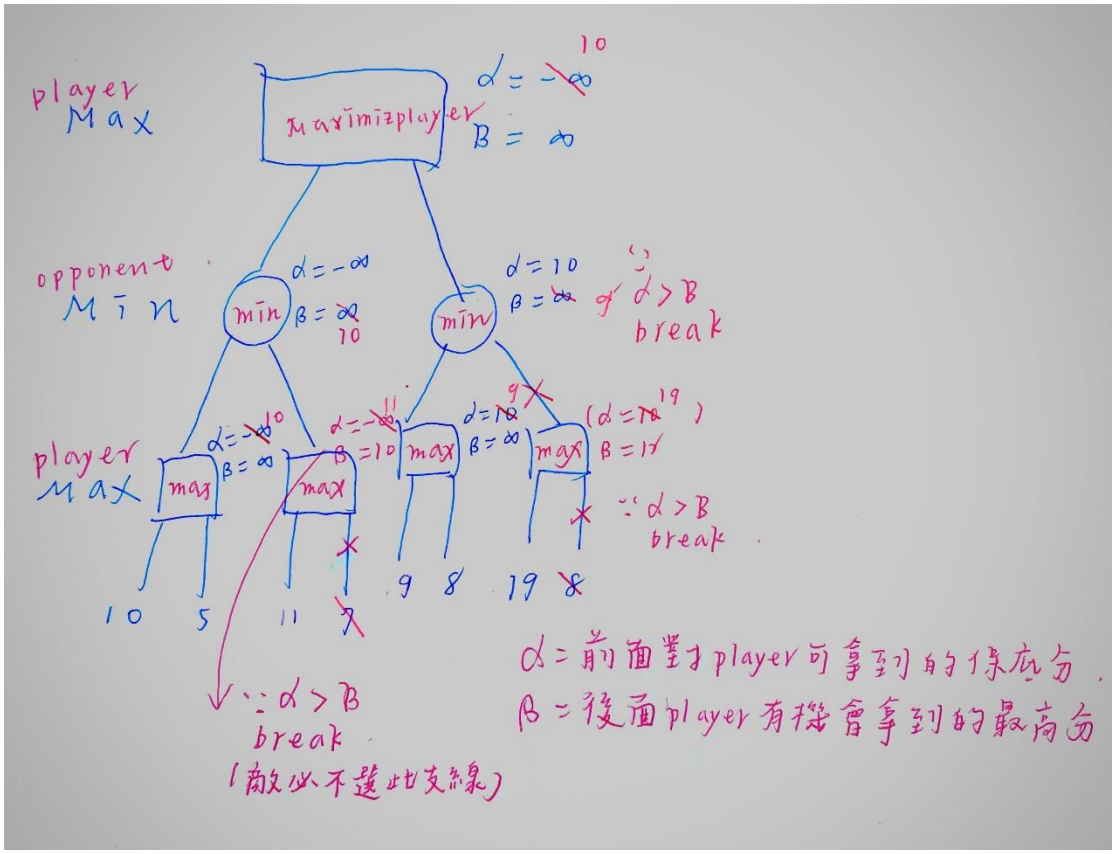
```
//確認四面八方有沒有棋子(要不然會超時QQ)
bool checkSurrounding(std::array<std::array<int, SIZE>, SIZE> bd, int i, int j){
        if(i>0 && i<SIZE-1){
            if(j>0 && j<SIZE-1){
                return bd[i-1][j-1]>0||bd[i-1][j]>0||bd[i-1][j+1]>0||bd[i][j-1]>0||bd[i][j+1]>0 ||bd[i+1][j-1]>0|| bd[i+1][j]>0 ||
            }
            else if(j==0){
                return bd[i-1][j]>0 ||bd[i-1][j+1]>0 ||bd[i][j+1]>0|| bd[i+1][j]>0 ||bd[i+1][j+1]>0;
            }
            else{
                return bd[i-1][j]>0||bd[i-1][j-1]>0||bd[i][j-1]>0 ||bd[i+1][j]>0 ||bd[i+1][j-1]>0;
            }
        }
        else if(i == 0){
            if(j>0 && j<SIZE-1){
                return bd[i][j-1]>0||bd[i][j+1]>0 ||bd[i+1][j-1]>0 ||bd[i+1][j]>0 ||bd[i][j+1]>0;
            }
            else if(j==0){
                return bd[i+1][j]>0 ||bd[i][j+1]>0 ||bd[i+1][j+1]>0;
            }
            else{
                return bd[i+1][j]>0 ||bd[i][j-1]>0 ||bd[i+1][j-1]>0;
            }
        }
        else{
            if(j>0 && j<SIZE-1){
                return bd[i][j-1]>0 ||bd[i-1][j-1]>0 ||bd[i-1][j]>0 ||bd[i-1][j+1]>0 ||bd[i][j+1]>0;
            }
            else if(j==0){
                return bd[i-1][j]>0|| bd[i-1][j+1]>0|| bd[i][j+1]>0;
            }
            else{
                return bd[i][j-1]>0 ||bd[i-1][j-1]>0 || bd[i-1][j]>0;
            }
        }
```

原本是沒有這樣的
但跑三層的 minimax 可能會超時
只好縮減 candidate 來縮短計算時間
四面八方都是空的點不去考慮他

minimax

```
//遞迴
int Minimax(State state, int depth, int Alpha, int Beta, bool maximizingPlayer){
    //alpha:目前為止player至少能拿幾分
    //beta:繼續走下去player最多能拿幾分
    if(depth == 0){
        return state.evaluate_score();
    }
    state.next_candi();
    if(maximizingPlayer){
        int maxEval = -999999;
        for(auto e : state.candi){
            State next = state;
            next.add_Point(e, player);
            int eval = Minimax(next, depth - 1, Alpha, Beta, !maximizingPlayer);
            //player的角度:player越高分越好
            maxEval = max(eval, maxEval);
            if(eval == maxEval){
                if(depth==3)
                {
                   //在頂層紀錄最佳解
                   Best = e;
                }
            }

            Alpha = max(Alpha, maxEval);
            //當beta>alpha就不繼續走下去(後面沒更好的選擇)
            if(Beta < Alpha) break;
        }
        return maxEval;
```

```
    }else{
        int minEval = 999999;
        for(auto e : state.candi){
            State next = state;
            next.add_Point(e, 3-player);
            int eval = Minimax(next, depth - 1, Alpha, Beta, !maximizingPlayer);
            //opponent的角度:player越低分越好
            minEval = min(minEval, eval);
            if(eval == minEval){
                //do nothing
            }
            Beta = min(Beta, minEval);
            //當beta>alpha就不繼續走下去(後面沒更好的選擇)
            if(Beta < Alpha) break;
        }
        return minEval;
    }
    //return 0;
}
```


player
Max
maximizplayer
α = -∞
10
β = ∞
opponent
Min
min
α = -∞
β = ∞
10
min
α = 10
β = ∞
∵ α > β
break
player
Max
max
α = -∞
β = ∞
10
max
α = -∞
β = 10
11
max
α = 10
β = ∞
9
max
(α = 19)
β = 17
∵ α > β
break
10
5
11
9
8
19
8
∵ α > β
break
(敵必不選此支線)
α = 前面對 player 可拿到的保底分
β = 後面 player 有機會拿到的最高分

Next_point 找出 best point

```
//找出best以及印出有用資訊
Point Next_Point(State &state){
    //minimax
    int best_val = Minimax(state, 3, -999999, 999999, true);
    //debug
    cout<<" ";
    cout<<"Best ("<<Best.x<<","<<Best.y<<")\n";
    cout<<player<<"\n";
    cout<<"hi 原:"<<state.evaluate_score()<<"\n";
    State tmp=state;
    tmp.add_Point(Best,player);
    cout<<"hello 後:"<<tmp.evaluate_score()<<"\n";
    //cout<<"hooma"<<best_val<<"\n";
    return Best;
}
```

Read_board

從 file 讀取

```
//讀棋盤
void read_board(std::ifstream& fin) {
    fin >> player;
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            fin >> komoku[i][j];
        }
    }
}
```

github