Mini Project 3 Report

1. Initialize

   使用三維陣列存三種不同狀態的棋盤

```
//用bitset建造遊戲面板
typedef bitset<15> Row; //一行15個
typedef array<Row, 15> Board_mini; //一個board 只記錄012的其中一種
typedef array<Board_mini, 3> Board; //記錄三種點(三維陣列!)
//[哪種點012][x][y]
```

2. State

   State 紀錄遊戲的狀態，包含棋盤、state value 等等

```
//一個class紀錄遊戲的狀態(有沒有結束等等)
class State{
private:
    Board board; //遊戲面板
    int player; //player可能是1或2
    void get_legal_move(); //finish 得到所有可以下的點
    GAME_STATE now_state = UNKNOWN; //一開始的狀態是未知
public:
    std::vector<Point> legal_move; //記錄所有的合法移動
    State(){}; //constructor
    State(Board board, int player); //finish constructor 得到這個player所有可以下的點
    int evaluate(); //finish 計算state value function
    State* next_state(Point move); //finish 得到下一個state
    GAME_STATE check_state(); //finish 檢查現在的state
};
```

```
//constructor
State::State(Board board, int player): board(board), player(player){
    this->get_legal_move();
};
```

```
//得到所有可以走的步驟
void State::get_legal_move(){
    std::vector<Point> moves; //蒐集所有可以下的點
    Board_mini point;
    bool initial = true; //判斷棋盤上有沒有子
    //只要跑子周遭的點就好 不用遍歷整個棋盤
    for(auto pt: all_move){ //棋盤上所有點
        if(board[0][pt.x][pt.y]==0){ //代表不是空的
            initial = false;
            for(auto pt_try: direction){ //跑附近8個方向的點
                int x = pt.x+pt_try[0];
                int y = pt.y+pt_try[1];
                if(x<0 || y<0 || x>=15 || y>=15 || point[x][y] || board[0][x][y]==0)
                    continue; //不合法的或是已經有子就跳過
                moves.push_back(Point(x, y));
                point[x][y] = 1;
            }
        }
    }
    //如果棋盤上沒有子 一開始要下中間!
    if(moves.empty() && initial)
        moves.push_back(Point(15/2, 15/2));
    legal_move = moves; //存所有應該要下的點(周遭8格的合法點or中間)
}
```

```
enum GAME_STATE {
    //未知 輸了 要下子 沒有特定的狀態
    UNKNOWN, LOSE, DRAW, NONE
};
```

3. State value function

   利用自己或對手的連子來計算 state value

```
//計算state value的函式
int State::evaluate(){
    Board_mini empty = board[0];
    Board_mini me = board[this->player];
    Board_mini he = board[3-this->player]; //看我是1或2 對手是2或1
    //傳empty進去讓函式可以判斷周遭有沒有空格
    if(check_5cnt(me) || count_4cnt(me, empty)) //連續五個或是有死||活四就贏了
        return INT_MAX;
    if(count_4cnt(he, empty)>1) //對手會贏的狀況(超過兩個死||活四)
        return INT_MIN;
    //我有的連34數量-他有的連34數量(因為3的分數可能會比較高所以4加權重一點)
    return (count_3cnt(me, empty))-(count_3cnt(he, empty))+(count_4cnt(me, empty))*3-(count_4cnt(he, empty))*3;
}
```

```
//看看有沒有五個連載一起(用bitwise確認!)
int check_5cnt(Board_mini board){
    for(int i=0;i<15-4;i++){
        //因為有5顆所以檢查到15-4
        //橫排的連續5個都沒碰到0就是有連著
        //.any() 測試是不是0或空 至少一個1就回傳1
        if((board[i] & board[i+1] & board[i+2] & board[i+3] & board[i+4]).count())
            return 1;
        //斜的把他們推到同一直排之後比
        else if((board[i] & (board[i+1]>>1) & (board[i+2]>>2) & (board[i+3]>>3) & (board[i+4]>>4)).count())
            return 1;
        else if((board[i] & (board[i+1]<<1) & (board[i+2]<<2) & (board[i+3]<<3) & (board[i+4]<<4)).count())
            return 1;
        //直的直接用bitwise比就好
        for(int j=0;j<15;j++){
            if(((board[j]>>i)&=0b11111) == 0b11111)
                return 1;
        }
    }
    return 0;
}
```

```
//計算四個連載一起
int count_4cnt(Board_mini board, Board_mini empty){
    int count = 0;
    for(int i=0; i<15-4; i+=1){
        //count計算有幾個1
        //橫的(左右一端空白)
        count += (empty[i] & board[i+1] & board[i+2] & board[i+3] & board[i+4]).count(); //左空白
        count += (board[i] & board[i+1] & board[i+2] & board[i+3] & empty[i+4]).count(); //右空白
        //斜的對其之後判斷
        //左上右下
        count += (empty[i] & (board[i+1]>>1) & (board[i+2]>>2) & (board[i+3]>>3) & (board[i+4]>>4)).count();
        count += (board[i] & (board[i+1]>>1) & (board[i+2]>>2) & (board[i+3]>>3) & (empty[i+4]>>4)).count();
        //右上左下
        count += (empty[i] & (board[i+1]<<1) & (board[i+2]<<2) & (board[i+3]<<3) & (board[i+4]<<4)).count();
        count += (board[i] & (board[i+1]<<1) & (board[i+2]<<2) & (board[i+3]<<3) & (empty[i+4]<<4)).count();
        //直的
        for(int j=0;j<15;j++){ //和11110或01111做bitwise可以知道是不是依樣的
            count += (((board[j]>>i)&=0b11110) == 0b11110 && ((empty[j]>>i)&=0b00001) == 0b00001);
            count += (((board[j]>>i)&=0b01111) == 0b01111 && ((empty[j]>>i)&=0b10000) == 0b10000);
        }
        if(count>2) //已經超過了就提早return省時間
            return count;
    }
    return count;
}
```

```
//檢查3個連載一起的情況
int count_3cnt(Board_mini board, Board_mini empty){
    int count = 0;
    for(int i=0;i<15-2;i++){
        //橫的
        for(int j=0;j<15;j++){
            //左邊兩個空 右邊兩個空 雙活 先預設沒有
            bool left_empty = false, right_empty = false, double_empty = false;
            bool target = (((board[j]>>i)&=0b111) == 0b111); //知道target有沒有連起來
            if(i>0 && i<15-3)
                double_empty = empty[j][15-i] && empty[j][15-i-4];
            if(i>1)
                right_empty = empty[j][15-i] && empty[j][15-i+1];
            if(i<15-4)
                left_empty = empty[j][15-i-4] && empty[j][15-i-5];
            //有空格&有三個連起來
            count += (right_empty & target);
            count += (left_empty & target);
            count += (double_empty & target);
        }
```

```
        Row left_empty; //左邊可以連成5顆
        Row right_empty; //右邊可以連成5顆
        Row double_empty; //記錄雙活3
        Row target; //連起來的子

        //直的部分
        right_empty = Row(0); //三個空的row
        left_empty = Row(0);
        double_empty = Row(0);
        target = board[i] & board[i+1] & board[i+2]; //有沒有三連
        if(i>0 && i<15-3) //雙活3
            double_empty = empty[i-1] & empty[i+3];
        if(i>1) //左邊兩顆是不是空的
            left_empty = empty[i-1] & empty[i-2];
        if(i<15-4) //右邊兩顆是不是空的
            right_empty = empty[i+3] & empty[i+4];
        //加進分數裡面
        count += (right_empty & target).count();
        count += (left_empty & target).count();
        count += (double_empty & target).count();
```

```
        //左上右下(邏輯同上)
        right_empty = Row(0);
        left_empty = Row(0);
        double_empty = Row(0);
        target = board[i] & (board[i+1]>>1) & (board[i+2]>>2); //記得對齊
        if(i>0 && i<15-3)
            double_empty = (empty[i-1]<<1) & (empty[i+3]>>3);
        if(i>1)
            left_empty = (empty[i-1]<<1) & (empty[i-2]<<2);
        if(i<15-4)
            right_empty = (empty[i+3]>>3) & (empty[i+4]>>4);
        count += (right_empty & target).count();
        count += (left_empty & target).count();
        count += (double_empty & target).count();

        //右上左下
        right_empty = Row(0);
        left_empty = Row(0);
        double_empty = Row(0);
        target = board[i] & (board[i+1]<<1) & (board[i+2]<<2); //記得對齊
        if(i>0 && i<15-3)
            double_empty = (empty[i-1]>>1) & (empty[i+3]<<3);
        if(i>1)
            left_empty = (empty[i-1]>>1) & (empty[i-2]>>2);
        if(i<15-4)
            right_empty = (empty[i+3]<<3) & (empty[i+4]<<4);
        count += (right_empty & target).count();
        count += (left_empty & target).count();
        count += (double_empty & target).count();
    }
    return count;
}
```

4. Next state & check state

   當下了一顆新的子時就要更新 state，更新棋盤以及可以下的子的位置

```cpp
//根據可以走的步驟得到下一個狀態
State* State::next_state(Point move){
    //建立新的狀態和棋盤 玩家換人當!
    //建立一個新的棋盤來更新新的狀態
    Board new_board = this->board;
    //下棋: player的board(x,y)改成1 empty的board改成沒有空格
    new_board[this->player][move.x][move.y] = 1;
    new_board[0][move.x][move.y] = 0;
    //建立一個新的狀態
    State *next = new State();
    next->board = new_board; //棋盤更新
    next->player = 3-player; //換人下

    //找到所有下一步可以下的(因為更新狀態了要重算!)
    //跟get_legal_move類似
    Board_mini point;
    std::vector<Point> moves;
    for(Point way:legal_move){
        if(way!=move){ //幫剛剛下的子去掉
            moves.push_back(way);
            point[way.x][way.y] = 1; //紀錄在point中
        }
    }
    //跑8個方向
    for(auto p_try: direction){
        //以剛剛下的子為中心 找其他可以下的子
        int x = move.x+p_try[0];
        int y = move.y+p_try[1];
        if(x<0 || y<0 || x>=SIZE || y>=SIZE || point[x][y] || board[0][x][y]==0)
            continue; //不合法的或是空的就跳過
        moves.push_back(Point(x, y));
        point[x][y] = 1;
    }
    next->legal_move = moves;
    return next;
}
```

```cpp
//檢查現在的狀態是甚麼
GAME_STATE State::check_state(){
    if (this->now_state != UNKNOWN)
        return this->now_state;
    Board_mini next = board[3-this->player]; //下一個玩家
    if (check_5cnt(next)) //5個連載一起就輸了q
        this->now_state = LOSE;
    else if (this->legal_move.empty()) //都跑完了就下
        this->now_state = DRAW;
    else
        this->now_state = NONE;
    return this->now_state;
}
```

5. Alpha_beta pruning

Alpha beta prunung 演算法！可以計算出最好的 state value 以決定下一步

```
//用演算法來找最好的移動方法
//evaluate先call第一個玩家 然後以alpha或score當作他的score
Point alpha_beta_get_move(State *state, int depth ,bool player){
    Point best_move = Point(-1, -1); //初始化在-1-1
    int alpha = INT_MIN; //alpha一開始是最小
    auto all_moves = state->legal_move; //把所有可以的步驟丟到all_moves
    for(Point move: all_moves){
        //把計算的分數加負號 alpha直變成最小 beta改成-alpha 並且深度-1
        int score = -alpha_beta_evaluate(state->next_state(move), depth-1, INT_MIN, -alpha ,true);
        if(score > alpha){ //如果算出來的分數比alpha高 那就代表是目前最好的步驟!
            best_move = move;
            alpha = score;
        }
    }
    return best_move;
}
```

```
int alpha_beta_evaluate(State *state, int depth, int alpha, int beta, bool player){
    GAME_STATE now_state = state->check_state();
    //各種遊戲狀態(輸 下子)
    if(now_state == DRAW){
        delete state;
        return 0; //不用算 只是要下子而已
    }
    else if(now_state == LOSE){ //輸了就回傳最小的值
        delete state;
        return INT_MIN;
    }
    //深度=0代表可以return了
    if(depth == 0){
        int score = state->evaluate(); //會呼叫state的evaluate函式 計算state value
        delete state; //計算完之後刪掉這個state
        return score; //回傳他算出來的state value
    }
    //alpha_beta演算法
    if(player){ //player1
        for(auto move: state->legal_move){ //從legal_move中找
            //alpha beta要換人考慮 所以改傳false
            int score = alpha_beta_evaluate(state->next_state(move), depth-1, alpha, beta, false);
            alpha = max(score, alpha); //score和原本的alpha找比較大的
            if(alpha >= beta){ //不用考慮了! 就是最好的步驟
                delete state;
                return alpha; //回傳alpha(一個score)
            }
        }
        delete state;
        return alpha; //跑完之後回傳alpha(一個score)
    }
    else{ //player2
        for(auto move: state->legal_move){ //從legal_move中找
            //alpha beta要換人考慮 所以改傳true
            int score = alpha_beta_evaluate(state->next_state(move), depth-1, alpha, beta, true);
            beta = min(score, beta); //score和原本的alpha找比較大的
            if(alpha >= beta){ //不用考慮了! 就是最好的步驟
                delete state;
                return beta; //回傳beta(一個score)
            }
        }
        delete state;
        return beta; //跑完之後回傳beta(一個score)
    }
}
```

6. Read board & write valid spot

   Read board 的部分改成存進三維陣列 write valid spot 則要結合 alpha_beta 的 get_move 函式來計算最好的位置

```
State root;
void read_board(std::ifstream& fin) {
    Board board; //第一個board
    fin >> player;
    for (int i=0;i<15;i++) {
        //讀進三維陣列中 把空 1 2分開紀錄
        for (int j=0;j<15;j++) {
            board[0][i][j] = 0; //三維陣列初始化
            board[1][i][j] = 0;
            board[2][i][j] = 0;
            int temp; fin >> temp; //輸入012
            board[temp][i][j] = 1; //把他丟進去三維陣列裡
        }
    }
    root = State(board, player); //初始狀態!
}
```

```
void write_valid_spot(std::ofstream& fout) {
    auto moves = root.legal_move; //初始狀態所有可以下的點
    for(auto move:moves){
        //如果下一步會贏就直接下
        if(root.next_state(move)->check_state() == LOSE){
            fout << move.x << " " << move.y << endl;
            fout.flush();
            return;
        }
    }
    //沒地方可以下
    if(moves.empty())
        return;
    //在下棋的過程中持續更新狀態
    int depth = 2; //思考的深度
    while (true){
        auto move = alpha_beta_get_move(&root, depth ,true); //找好的移動策略!
        if(move.x != -1 && move.y != -1){
            //跑完了就輸出
            fout << move.x << " " << move.y << endl;
            fout.flush();
        }
        depth += 1; //上面的get_move會讓depth變少
    }
}
```