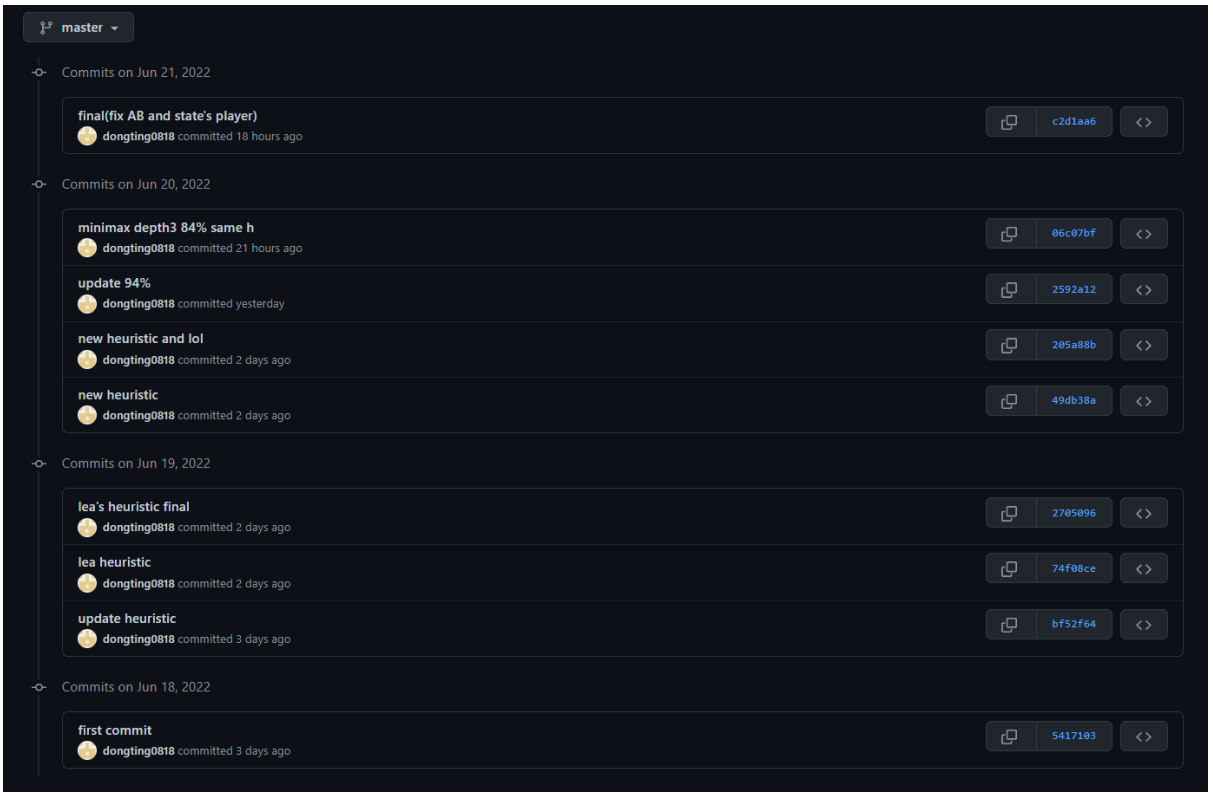
110062128 Mini Project 3 Report

Outline

- Version Control (GitHub)
- class “State”
- class “Point”
- Main Function
- write_valid_move
- Next_Point
- Algorithm: Minimax with Alpha-Beta Pruning
- Heuristic Function

Version Control (GitHub):



class State:

```
std::array<std::array<int, SIZE>, SIZE> Board;
std::set<Point> enum_move_point;
```

constructors:

```
State(std::array<std::array<int, SIZE>, SIZE>b){
    //this->player = ply;
    //this->score = NEG_INF;
    for(int i=0;i<SIZE;i++){
        for(int j=0;j<SIZE;j++){
            this->Board[i][j] = b[i][j];
        }
    }
}
State(State& copy){
    for(int i=0;i<SIZE;i++){
        for(int j=0;j<SIZE;j++){
            this->Board[i][j] = copy.Board[i][j];
        }
    }
    //this->player = copy.player;
}
```

enumerate next move:

```
void next_move_enum(){
    //scan the whole board
    for(int i=0;i<SIZE;i++){
        for(int j=0;j<SIZE;j++){
            if(Board[i][j] == EMPTY){
                //check Jiugongge

                if(exam_jugongge(i,j,board) == 1){
                    this->enum_move_point.insert(Point(i,j));
                }
            }
        }
    }
    return;
}
```

boost:

```
int exam_jugongge(int i, int j,std::array<std::array<int, SIZE>, SIZE> gomoku){
    if(i>0 && i<SIZE-1){
        if(j>0 && j<SIZE-1){
            if(gomoku[i-1][j-1]>0 || gomoku[i-1][j]>0 || gomoku[i-1][j+1]>0 || gomoku[i][j-1]>0 || gomoku[i][j+1]>0 || gomoku[i+1][j-1]>0 || gomoku[i+1][j]>0 || gomoku[i+1][j+1]>0)
                return true;
        }
        else if(j==0){
            if(gomoku[i-1][j]>0 || gomoku[i-1][j+1]>0 || gomoku[i][j+1]>0 || gomoku[i+1][j]>0 || gomoku[i+1][j+1]>0) return true;
        }
        else{
            if(gomoku[i-1][j]>0 || gomoku[i-1][j-1]>0 || gomoku[i][j-1]>0 || gomoku[i+1][j]>0 || gomoku[i+1][j-1]>0) return true;
        }
    }
    else if(i == 0){
        if(j>0 && j<SIZE-1){
            if(gomoku[i][j-1]>0 || gomoku[i][j+1]>0 || gomoku[i+1][j-1]>0 || gomoku[i+1][j]>0 || gomoku[i][j+1]>0)return true;
        }
        else if(j==0){
            if(gomoku[i+1][j]>0 || gomoku[i][j+1]>0 || gomoku[i+1][j+1]>0) return true;
        }
        else{
            if(gomoku[i+1][j]>0 || gomoku[i][j-1]>0 || gomoku[i+1][j-1]>0) return true;
        }
    }
    else{
        if(j>0 && j<SIZE-1){
          if(gomoku[i][j-1]>0 || gomoku[i-1][j-1]>0 || gomoku[i-1][j]>0 || gomoku[i-1][j+1]>0 || gomoku[i][j+1]>0) return true;
        }
        else if(j==0){
            if(gomoku[i-1][j]>0 || gomoku[i-1][j+1]>0 || gomoku[i][j+1]>0)return true;
        }
        else{
            if(gomoku[i][j-1]>0 || gomoku[i-1][j-1]>0 || gomoku[i-1][j]>0)return true;
        }
    }
    return false;
}
```

add_Point, remove_Point:

```
void add_Point(Point point, int disc){
    Board[point.x][point.y] = disc;
}
void remove_Point(Point point){
    Board[point.x][point.y] = EMPTY;
}
```

class Point:
I refer to main.cpp, and I add an opearator<() just so I can use set.

```
struct Point {
    int x, y, score;
    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;
    }
};
```

Main Function

```
451  int main(int, char** argv) {
452      std::ifstream fin(argv[1]);
453      std::ofstream fout(argv[2]);
454      read_board(fin);
455
456      State init(board);
457
458      write_valid_spot(fout, init);
459      fin.close();
460      fout.close();
461      return 0;
462  }
```

Refer to “Player_Random”. Read the board from the input file.

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

initialize a State with the input board.

Use the function “write_valid_spot” to get and output the best move.

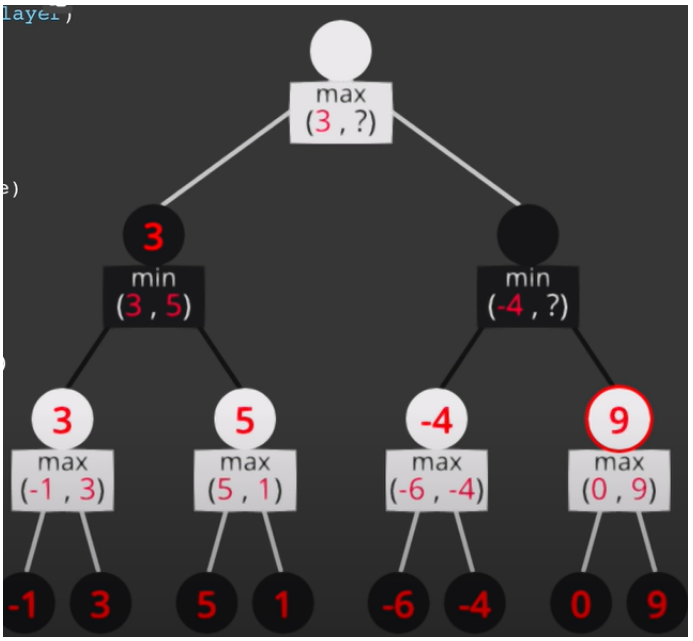
```
void write_valid_spot(std::ofstream& fout, State &state) {
    srand(time(NULL));
    //int x, y;
    bool flag = false;
    for(int i=0;i<SIZE;i++){
      for(int j=0;j<SIZE;j++){
            if(state.Board[i][j] == player){
                flag = true;
                break;
            }
        }
    }
    if(flag == true){
        Point next = Next_Point(state);
        fout<<next.x<<" "<<next.y<<"\n";
    }else{
        if(board[SIZE/2][SIZE/2] == 0) fout<<SIZE/2<<" "<<SIZE/2;
        else fout<<(SIZE/2)+1<<" "<<SIZE/2+1;
    }
    fout.flush();
}
```

Fix the first move. if you’re playing firsthand, place it at (7,7), else place it at (8,8)
If it’s not the first move (flag == true means there is at least one player’s disc on the board), **use the function “Next_Point” to find which Point is the best possible move**.

```
Point Next_Point(State &state){

    int tmp_score = Minimax(state, 3, NEG_INF, INF, true, true);

    state.next_move_enum();

    for(Point child : state.enum_move_point){
        State next = state;
        next.add_Point(child, player);
        int score = next.evaluate_score(player);
        int opp_score = next.evaluate_score(3-player);
        if(score>550000 || opp_score>550000){
            Best = child;
        }
        state.remove_Point(child);
    }

    return Best ;
}
```

First, the program enumerates the possible Points you can put the disc using “next move enum” in the State class.

Minimax:




```
int Minimax(State state, int depth, int Alpha, int Beta, bool maximizingPlayer, bool flag)
```

```
int tmp_score = Minimax(state, 3, NEG_INF, INF, true, true);
```

The player plays as the maximizing player, which means the player wants the board value to be greater. The opponent plays as the minimizing player, which means the opponent wants the player's board value to be smaller.

state: the current board state.

depth:
The total depths of my program are 3.
Depth 3: player
Depth 2: opponent
Depth 1: player
Depth 0 (終止條件): return the current value of the board

```
if(depth == 0){
    //return the score base on the board (no recursion)
    return state.evaluate_score(player) - state.evaluate_score(3-player);
}
```

maximizingPlayer: We use a boolean value to know whether it is the player's turn.

flag: We only decide the best move in the depth 3. Because we didn't really make the move in the previous depth, we only predicted it. The use of **flag** is to check if it's depth 3.

Structure:

```
if(maximizingPlayer){
    state.next_move_enum();
    int maxEval = NEG_INF;
    for(auto child : state.enum_move_point){
        State next = state;
        next.add_Point(child, player);
        int eval = Minimax(next, depth - 1, Alpha, Beta, false, false);
        next.remove_Point(child);
        if(eval > maxEval && flag == true){
            Best = child;
            Best.score = eval;
        }
        maxEval = max(maxEval, eval);
        Alpha = max(Alpha, maxEval);
        if(Beta <= Alpha) break;
    }
    return maxEval;
```

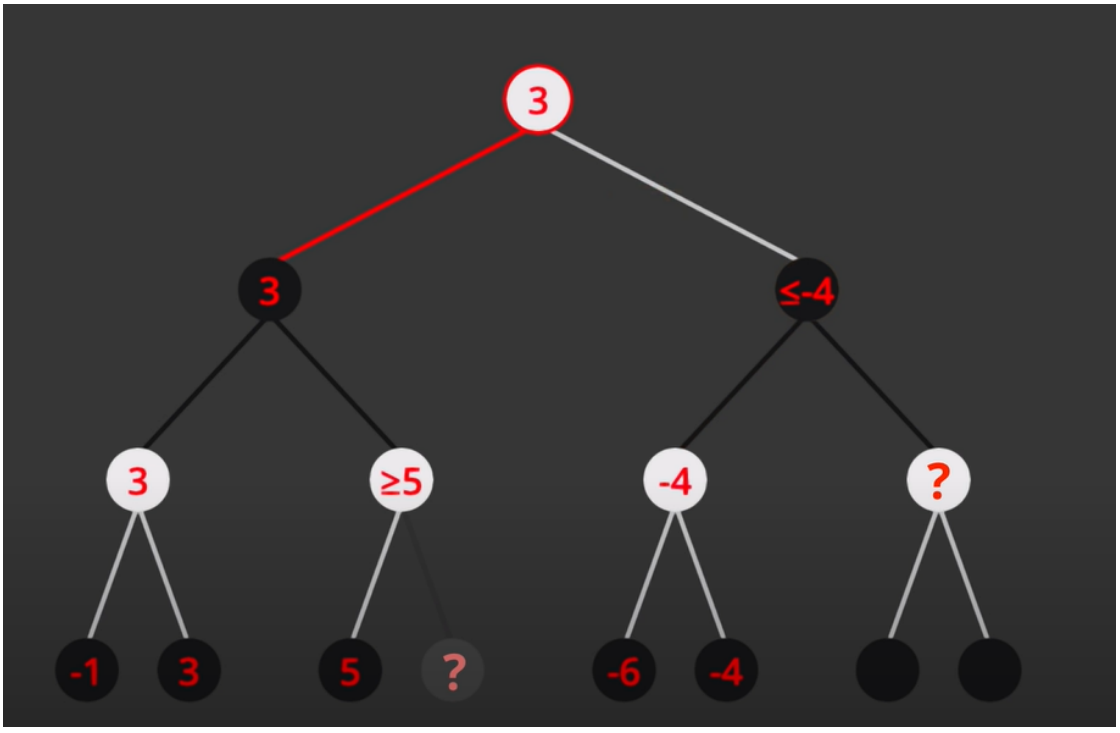
First, the state enumerates the valid Points the program can put on the board using "next_move_enum" and store them in the set "enum_move_point."

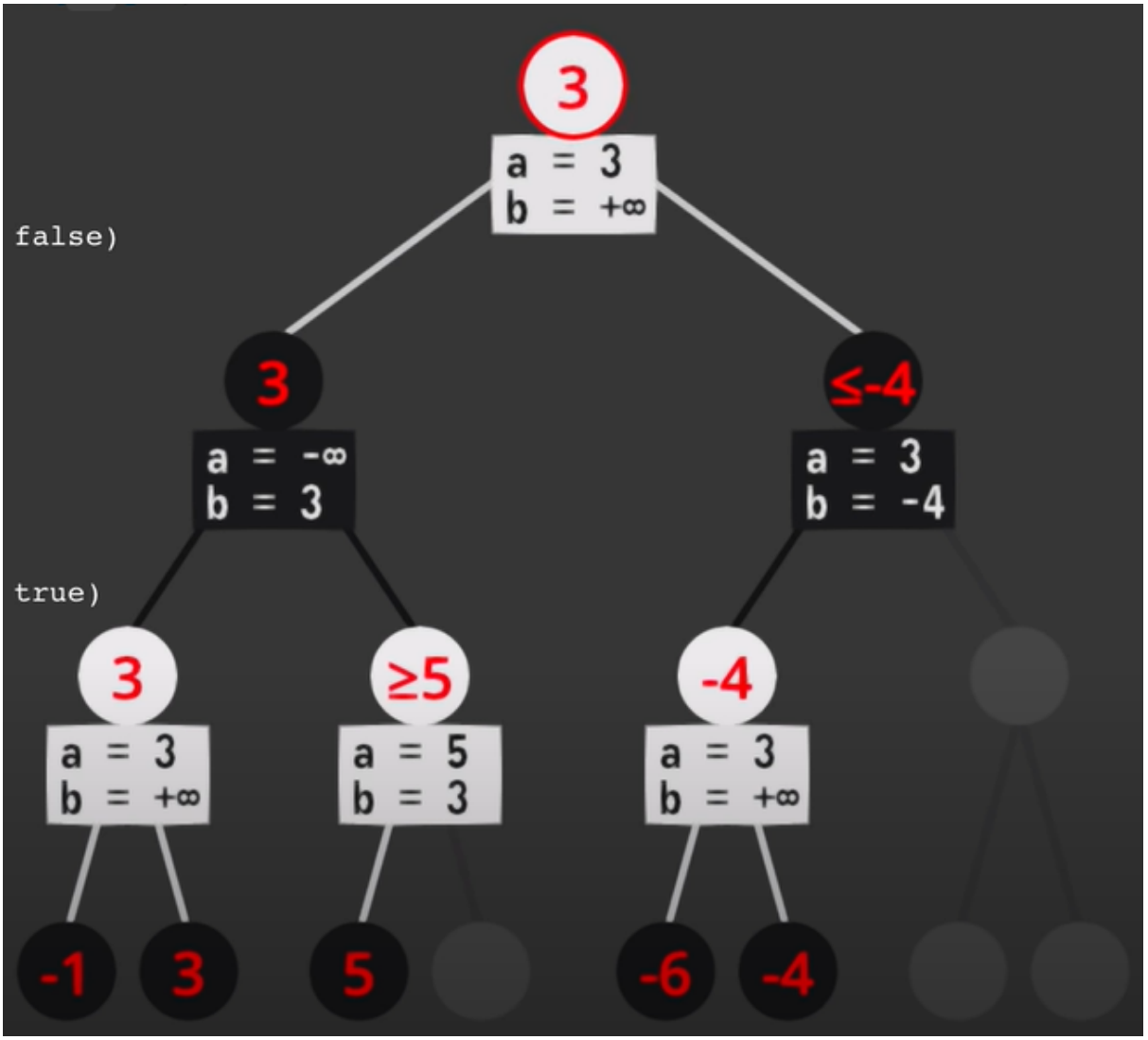
Then it iterates the whole possible moves. The program will add the point to the current state and see the outcome of this move using minimax, and store the value in "**eval.**" Then it will compare **eval** to **maxEval** because we want the higher value. If this is depth 3 and eval > maxEval, the best move will be updated in Point "**Best.**"

For minimizing players, the structure is basically the same, but the smaller the better.

```
}else{
    state.next_move_enum();
    int minEval = INF;
    for(Point child : state.enum_move_point){
        State next = state;
        next.add_Point(child, 3-player);
        int eval = Minimax(next, depth - 1, Alpha, Beta, true, false);
        next.remove_Point(child);
        if(eval < minEval){
        }
        minEval = min(minEval, eval);
        Beta = min(Beta, minEval);
        if(Beta <= Alpha) break;
    }
    return minEval;
```

Alpha-Beta Pruning:





Alpha: 這一層你可以選到的最高分
Beta: 這一層對手允許你選到多少分
在maximizing player只有alpha會變，在minimizing player 只有beta會變
若alpha>=beta，代表在同一層的另一個child對parent來說已經是一個更好得選擇(也就是說我走這步一定會招來parent node不想要的結果的話， 所以現在這個node不需要繼續在拓展下去，可直接break。

```
Alpha = max(Alpha, maxEval);
if(Beta <= Alpha) break;
```

```
Beta = min(Beta, minEval);
if(Beta <= Alpha) break;
```

How I evaluate score:

```
int evaluate_score(int who)
```

```
int the_other = 3-who;
```

```
for(int i=0;i<SIZE;i++){
   for(int j=0;j<SIZE;j++){
       if(Board[i][j] == who)
```

```
FIVE_IN_ROW ooooo
LIVE_FOUR _oooo
LIVE_THREE _ooo_
DEADFOUR _oooox || xoooo_
DEAD_THREE _ooox || xooo_
/DEAD_TWO _oox || xoo_
LIVE_TWO _oo_
_oo_o_ || _o_oo_
```

Ex:

```
//DEAD_TWO _oox || xoo_
if(Board[i][j-1] == EMPTY && Board[i][j+1] == who && Board[i][j+2] == the_other){
    half2++;
}
if(Board[i-1][j] == EMPTY && Board[i+1][j] == who && Board[i+2][j] == the_other){
    half2++;
}
if(Board[i-1][j-1] == EMPTY && Board[i+1][j+1] == who && Board[i+2][j+2] == the_other){
    half2++;
}
if(Board[i-1][j+1] == EMPTY && Board[i+1][j-1] == who && Board[i+2][j-2] == the_other){
    half2++;
}
```

加權:

```
int g;
if(who == player)
  g = 50000*N5 + 4800*open4 + 2500*half4 + 800*special1 + 2000*open3 + 200*half3 + 50*open2 + 10*half2;
else
  g = 55000*N5 + 5500*open4 + 3000*half4 + 1000*special1 + 2000*open3 + 200*half3 + 50*open2 + 10*half2;
return g;
```