Introduction:

Old version:

My first version is the given sample of BotZone. It was coded by C++ and used Json as a measure to transmit the grid information of board and the decision of AI. The program is a process-oriented program. I set the version number of the program as 0 in BotZone.

The decision-making part of this program is a random algorithm. After receiving the checkerboard information from BotZone, the program will restore the board to the current state. Then according to the color of own pieces, the program will check the legitimacy of all 64 points on the current board and store the locations of all legitimate points into the array. The program will select one point from the array randomly, and the selection will be sent to BotZone via Json.

New version:

My new version has been improved on the basis of the old version and the new version number is 1.0. There are two main improvements:

1. The original process-oriented process, into object-oriented procedures

In the new version of the program, the board is encapsulated into a board class. The operations of the board, such as play chess, legality checks, etc., are packaged in the class. The use of object-oriented programming will improve the readability of the program. Meanwhile, taking into account that the decision-making part will use recursive algorithm and use class objects as parameters to transmit, object-oriented programming can simplify the program structure.

1. The decision-making part uses the idea of Monte Carlo Tree to give a choice by calculating the winning possibilities of each legitimate point.

The decision part of the program uses a recursive algorithm. At the beginning of the program, the program will create a new instance of the board class, board1, and initialize it. While the program receives the board information from BotZone, the instance is updated until the board is restored to its current state. After that, the program will check the legitimacies of all 64 points and store the legitimate points into the array of board1. Here the preparation of the decision-making part is completed.

After the program enters the decision part, the board1 is passed into the decision function as a parameter. The function will copy the board1, and the program will be simulated in a copy of board1. In the copy, the program randomly select positions to lit according to the legal point in board1. After that, the program will exchange the color of pieces and update the legal point array after checking the legitimacies of all the 64 points. The program needs to determine whether this game is over. If the board is full or both black and white pieces have no legitimate points, the game is over; otherwise, the game continues. If the game continues, the current status of the board copy will be a parameter into the decision function again. Until the end of the game, the program according to the number of black and white pieces on the board record the result of the situation. Until now, the program complete a simulation of one legal point. The program will run the simulation above of all legal points. After several simulations, according to the number of wins and the number of simulation times, the function will return the winning probability of this point.

The program will select the point with the highest winning probability as a decision and send its coordinate to BotZone.

Details of implementation:

Class Board:

The attributes in the class board include the current board layout, the color of the pieces, the number of pieces on both sides, and an array of legal points.

The chessboard operations, including the lit, board resuming and the decision-making process. The code structure of class board is as follows:

Code structure:

class board

{

public:

int currBotColor;

int gridInfo[8][8];

int blackPieceCount , whitePieceCount;

int possiblePos[64][2], posCount;

board();

inline bool MoveStep(int &x, int &y, int Direction);

bool ProcStep(int xPos, int yPos, int color, bool checkOnly = false);

bool CheckIfHasValidMove(int color);

void chessboard\_resuming(Json::Value input, board &bd);

bool findValidPosition(int currBotColor1);

bool ifGameOver(board board);

void makeAdecision(board &board1,int inputX=-2,int inputY=-2);

}

Function makeAdecision:

Decision-making process using recursive algorithm, through continuous random selection, simulate the chess process. It will record the number of wins of one point and the number of simulations. Finally the ratio of the win times and simulation times will be used as a winning percentage. The decision-making process code structure is as follows:

Code structure:

void makeAdecision(board &board1,int inputX=-2,int inputY=-2)

{

board copyboard = board1;

int resultX, resultY, choice;

if(inputX !=-2 && inputY !=-2){

resultX = inputX;

resultY = inputY;

}

else

{

if (copyboard.posCount > 0)

{

srand(time(0));

choice = rand() % copyboard.posCount;

resultX = copyboard.possiblePos[choice][0];

resultY = copyboard.possiblePos[choice][1];

}

else

{

resultX = -1;

resultY = -1;

}

}

copyboard.ProcStep(resultX,resultY,copyboard.currBotColor,false);

copyboard.currBotColor = -copyboard.currBotColor;

copyboard.findValidPosition(copyboard.currBotColor);

if( ifGameOver(copyboard))

{

simulationTimes ++;

int bkcount = copyboard.blackPieceCount;

int wtcount = copyboard.whitePieceCount;

if(wtcount > bkcount){

whiteWinTimes ++;

}

firstFlag=1;

}

else{

makeAdecision(copyboard);

}

}

Analyzation:

Advantages:

The new version uses object-oriented programming. It will make the program structure more clear and readable.

The decision-making part of the new version uses the idea of Monte Carlo Tree. The program can calculate the winning probability of each point through the simulations and finally select the point with the highest winning probability as result sent to BotZone. Compared to the old version with random algorithm, the result of new version has higher accuracy.

Disadvantages:

The decision-making part of new version uses recursive algorithm and makes class instances as parameters to transfer. Therefore, the program in the running process will cost more resources and the efficiency of implementation will be relatively lower.

At the same time, the accuracy of the result of the new version is proportional to the time of simulations. Only when the time of simulations is large enough, the simulated winning probability will be more close to the real winning probability.

Next generation:

The next generation of the program will improve the efficiency of the implementation and reduce the costs of system resource on the basis of this new version. While the decision-making part of the next-generation program will try to use the idea of Monte Carlo Tress Search.