Foundation of AI Coursework 2
Gemoku

Qi Zhao
Student ID:30571626
Email:qz1y18@soton.ac.uk

ABSTRACT
In this paper, I describe the details of my foundation of AI coursework 2. In this coursework, I use C++ to code a Gomoku game. About the AI section, I use Minimax and Alpha-beta pruning to implement it.

Keywords
Gomoku, Artificial Intelligence, C++, Minimax, Alpha-beta pruning

1. INTRODUCTION
Gomoku is a very simple game. Therefore, there are plenty of people like to play it. I like the game very much. So this coursework I choose to code an AI to play this game. The programming language of the program is C++. And the methods I used are Minimax and Alpha-beta pruning. Firstly I introduce some Gomoku rules and terms which is used in my program. Then I will show my program and have some comments on it. And in the appendix of this report, I put my code in it.

2. GAME RULES AND TERMS
Gomoku, also called Five In A Row, is an abstract strategy board game[1]. It is traditionally played with Go pieces (black and white stones) on a Go board, using 15×15 of the 19×19 grid intersections. Because pieces are not moved or removed from the board, Gomoku may also be played as a paper and pencil game.

2.1 Game Rules
Two players alternatively make moves. The player who chooses black plays first, and the player who chooses white plays secondly. The first player who constructs a line with five pieces wins[2].

Because the player who plays first has advantages to win the game, in order to ensure game fairness, there are disallowed moves to limite the player who plays first. They are double three, double four and overline. These terms will be explained in section 2.2.

2.2 Terms
In fact, there are many terms that are relevant to Gomoku, but in this section, I only list some terms which are relevant to my coursework[3].

Open three: Three pieces are connected and both ends are empty.

Open four: Four pieces are connected and both ends are empty.

Half open four: Four pieces are connected, one end is empty and the other is not.

Double three: Black cannot place a stone that builds two separate lines with three black stones in unbroken rows (i.e. rows not blocked by white stones).

Double four: Black cannot place a stone that builds two separate lines with four black stones in a row.

Overline: six or more black stones in a row.

3. METHOD
3.1 Board
I use a 15 × 15 2-dimensional array to store board state. The different elements in the array mean different symbol. For example, '1' means black chess, while '2' means white chess. Every round I will traverse the board and print it to show the board state.

3.2 Check Chess Type and Disallowed Moves
In order to make evaluation function better, I build some function to check chess type and disallowed moves. For example, Check whether there is a open three, open four or half open four state appear after putting a piece, or whether there is a overline if the player is black.

3.2.1 Check Chess Type
I build two 1 × 8 arrays to help program identify a position's 8 directions. There are dx(1, 1, 0, -1, -1, -1, 0, 1) and dy(0, 1, 1, 1, 0, -1, -1, -1). For example, the right of position [row, col] is [row + dx[0], col + dy[0]]. It can make checking chess type very easy. There are four chess types need to check: open three, open four, overline and half open four. So I build three function to check these three chess types. Their name are ***openThree()***, ***openFour()***, ***overline*** and ***halfOpenFour()***. The input of them is a coordinate. The output is the number of these chess types. They use a loop to implement checking chess types in 8 directions, in every loop, they will check how many pieces are connected into a line, then to check these pieces' chess type.

3.2.2 Check Disallowed Moves
I write this section into function ***disallowedMove()***. The input of it is a coordinate, and the output of it is a Boolean variable. Firstly, the function will check whether the player is black, because only black has disallowed moves. If it is, then check whether the number of open three or open four + half open four is bigger than 1,or check whether there is a overline. If it is the output is true, if not, the output is false.

3.3 Evaluation Function
I write evaluation function in *computeScore()*, There are four conditions of the function. The first one is disallowed move. If this position is disallowed move, the score is zero. The second condition is invalid position. It means that there is no chess piece on the side. The score of this condition is zero as well. The third condition is game over. If this position can make player win, then the score is 10000. The forth condition is that there are pieces surrounds the position. Then we will use the equation below to compute the score.

$$Score = 1000 * N_{open\ 4} + 100 * (N_{open3} + N_{half\ open\ 4})$$

For the equation, $N_{open\ 4}$ means the number of open four, N_{open3} means the number of open 3, and $N_{half\ open\ 4}$ means the number of half open four.

$$\begin{cases} 0 & disallowed\ move \\ 0 & invaild\ position \\ 10000 & end\ move \\ 1000 * N_{open\ 4} + 100 * (N_{open3} + N_{half\ open\ 4}) & normal\ move \end{cases}$$

3.4 AI action

In this section, I choose minimax and Alpha-beta pruning as the strategy of AI. And I use iteration to code min floor and max floor. There are ***Max1()***, ***Min1()***, ***Max2()***.the input of them and output of them are integer.

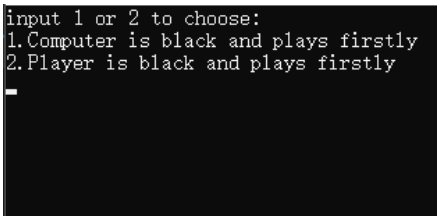
When this turn is AI, firstly it will invoke ***Max1()***. In beginning, ***Max1()*** traverses all of the position on board and use *computeScore()* compute their scores. If it finds some positions' score is zero, because only disable move position or position that there is no chess pieces on the side can get a 0 score, it will skip these position. About the position whose score is not zero, if it is 10000, which means it's a end move, just go it. If not, it will invoke ***Min1()***, and set its **alpha** value as ***Min()*** 's input parameter. And when it receives an output value from ***Min()***, it will compare it with alpha. If the output is bigger than alpha, then change alpha to the output value and records the position. Then repeat it on the next position. If there is no position available, just return the put the chess piece on the position recorded.

About function ***Min***(), when it receives an input value, set it as alpha. Then it traverses all of the position on board. If it finds some positions' score is zero, skip it. And if it finds some positions' score is 10000, which means opponent can win, just end the function and return -10000. If not, it will invoke ***Max2()***, and it well set its beta value as ***Max2()***'s input parameter. When it receives an output from ***Max2()***, compare it with its **beta**. If the output is smaller than **beta**, change beta to the output value. Then repeat it on the next available position. In every circle, check if alpha is bigger than beta, if it is, then return **beta**, which means pruning the following nodes. If there is no position available, just return **beta**.

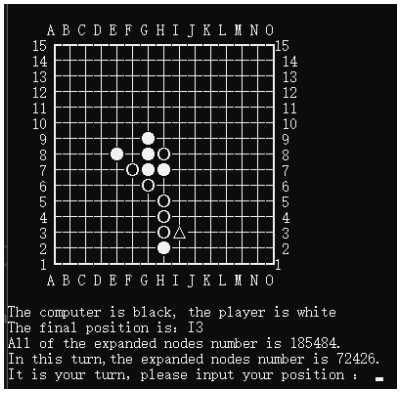
About function ***Max2()***, set its receive input value as its **beta**. Then repeat the similar process as ***Max1()***. Firstly traverses all of the available position on the board, use *computeScore()* compute their scores, if the score is bigger than **alpha**, change **alpha** to this score. If it found alpha is bigger than beta, return **alpha**. If there is no available position to traverse, just return **alpha**.

4. RESULT

When running the program, the first picture is choosing piece color. The picture is showed below.



I use triangle chess piece to represent the pieces in last round, and use circle chess piece to represent normal pieces. The picture of playing the game is shown below.



In addition, because I did not create real nodes in this coursework to do the search, I use function to simulate generating nodes. But I still can record how many times a function used to record how many 'nodes' were generated. And I draw a table below to show how the number of expanded nodes change with the game progressed. In the battle I recorded, AI plays black.

	Expanded nodes number in every turn	The number of all of the expanded nodes
1	1	1
2	20979	20980
3	30712	51692
4	20868	72560
5	24980	97540
6	53039	150579
7	125520	276099
8	92248	368347
9	35334	403681
10	72586	476267

I only record 10 turns of the game, I compare the adjacent numbers of expanded nodes per turn, I found that the value of them are very close. I think the reason is that the adjacent turns have similar board state, so the numbers of expanded nodes in them are close.

In addition, I found with the number of rounds rising, the number of expanded nodes in per turn has an upward trend. I think the reason of this phenomenon is that my algorithm does not search the position who has no neighbor chess piece. With the number of chess pieces increasing, the position need to search will increase, which leads to the number of expanded nodes in every turn rising.

5. LIMITATION & SELF EVALUATION

In my opinion, there are four main limitation. One is that I did not make a good GUI for the program. I just show it on command. And when I want to play it, I need to use keyboard to write the coordinate. It is very inconvenient. I think if it has a good GUI, it will be more interesting.

The second one is that the evaluation function should be improved. In my program, except disallowed moves, I only use three chess types. I think it can consider more kind of chess types, such as sleep three, dead four and open two and so on.

In addition, the score of chess type is not very precise. I just set them through my understanding of these chess types. If it can be changed to a more precise one, I believe the action of AI will be better.

The last one is that I only build two max floor and one min floor. I think it is possible to build more floors to determine which position is the best choice of AI.

6. REFERENCES

[1] *Gomoku*, https://en.wikipedia.org/wiki/Gomoku.

[2] YuHan Lyu. 2011. *Random-Turn Hex and Other Selection Games.* https://www.math.dartmouth.edu/~pw/M100W11/yu-han.pdf

[3] *What is Renju?* http://webcache.googleusercontent.com/search?q=cache:http://www.renju.net/study/rules.php.

APPENDIX:

```
#include "pch.h"
#include<stdio.h>
#include<string>
#include<windows.h>
#include<iostream>

using namespace std;
#define N 15 //size of board
#define sameColor same(row + dx[u] * i, col + dy[u] * i, key)//Check whether its the same color
#define SPD for (i = 1; sameColor; i++)sumNum++; //Search positive direction
#define SND for (i = -1; sameColor; i--)sumNum++;//Search negative direction
#define checkVaildPoint if(!inboard(row + dx[u] * i, col + dy[u] * i) || Board[row + dx[u] * i][col + dy[u] * i] != 0)continue;//Check whether the position is vaild
#define s_Computer 1
int nodeNum = 0;//Record total number of expanded nodes
int TToNN = 0;//Record in this turn how many nodes expanded
int Board[N + 2][N + 2]; //use a 2-dimensional array to store a board
int s = 0;//determine whose turn now
int s0 = 0;//determine who plays first
bool boolEnd = false;//distinguish whether the game is end

//use these two array to represent the positions surrounds a surround.
//For example,as for positon (row,col),the right position of a position is (row+dx[0],col+dy[0]).
//8 means 8 directions of a position
int dx[8] = { 1, 1, 0, -1, -1, -1, 0, 1 };
int dy[8] = { 0, 1, 1, 1, 0, -1, -1, -1 };

int printSymbol(int i, int j)
{
    if (Board[i][j] == 1)return printf("○");
    if (Board[i][j] == 2)return printf("●");
    if (Board[i][j] == -1)return printf("△");
    if (Board[i][j] == -2)return printf("▲");
    if (i == N)
    {
        if (j == 1)return printf("┌ ");
        if (j == N)return printf("┐");
        return printf("┬ ");
    }
    if (i == 1)
    {
        if (j == 1)return printf("└ ");
        if (j == N)return printf("┘");
        return printf("┴ ");
    }
    if (j == 1)return printf("├ ");
    if (j == N)return printf("┤ ");
    return printf("┼ ");
}
// Drawing board
void DrawBoard()
{
    system("cls");
    int row = 0, col = 0, keyr = 0, keyc = 0;
    char alpha = 'A';
    cout<<"\n\n\n     ";
    for (col = 1; col <= N; col++)
        cout<<alpha++<<" ";
```

```
58.     for (row = N; row >= 1; row--)
59.     {
60.         printf("\n   %2d", row);
61.         for (col = 1; col <= N; col++)
62.         {
63.             printSymbol(row, col);
64.             if (Board[row][col] < 0) {
65.                 keyr = row;
66.                 keyc = col;
67.             }
68.         }
69.         cout<<row;
70.     }
71.     alpha = 'A';
72.     printf("\n     ");
73.     for (col = 1; col <= N; col++)
74.         cout<<alpha++<<" ";
75.     printf("\n\n");
76.     if (s0 == s_Computer)
77.         cout<<"The computer is black, the player is white"<<endl;
78.     else
79.         cout<<"The computer is white, the player is black"<<endl;
80.     alpha = 'A';
81.     if (keyr)
82.         cout<<"The final position is: "<<(char)(alpha + keyc - 1)<<keyr<<endl;
83. }
84. //initialization
85. void init()
86. {
87.     //define the background and word color of command
88.     system("color 0f");
89.     cout << "input 1 or 2 to choose:" << endl << "1.Computer is black and plays firstly" << en
    dl << "2.Player is black and plays firstly" << endl;
90.     cin >> s;
91.     if (s != 1 && s != 2) {
92.         cout << "please input a vaild number.";
93.         return init();
94.     }
95.     //ensure who plays firstly
96.     s0 = s;
97.     //use space surrounds the board
98.     for (int i = 0; i <= N + 1; i++)
99.         for (int j = 0; j <= N + 1; j++)
100.                Board[i][j] = 0;
101.        DrawBoard();
102.    }
103.    //check whether the position is included in the board
104.    bool inboard(int row, int col)
105.    {
106.        if (row <1 || row > N)
107.            return false;
108.        return col >= 1 && col <= N;
109.    }
110.    //Check whether the position is vaild to under a pawn.
111.    bool ok(int row, int col)
112.    {
113.        return inboard(row, col) && (Board[row][col] == 0);
114.    }
115.
116.    //distinguish the color of the specific position.
117.    //For the parameter key,1 means computer,2 means player
118.    int same(int row, int col, int key)
```

```
119.        {
120.            //determine whether the position is in board
121.            if (!inboard(row, col))
122.                return false;
123.            return (Board[row][col] == key || Board[row][col] + key == 0);
124.        }
125.        //for position(row,col),computer the sum number of pawn in direction u.
126.        //about direction u,1 means →, 2 means ↗,
127.        //3 means ↑,  4 means ↖,5 means ←,6 means ↙,7 means↓,8 means ↘.
128.        int num(int row, int col, int u)
129.        {
130.            int i = row + dx[u], j = col + dy[u], sum = 0, ref = Board[row][col];
131.            if (ref == 0)
132.                return 0;
133.            while (same(i, j, ref))
134.                sum++, i += dx[u], j += dy[u];
135.            return sum;
136.        }
137.        //Compute the number of open four
138.        int openFour(int row, int col)
139.        {
140.            int key = Board[row][col], sum = 0, i, u;
141.            //Search four directions (left and right, up and down, left up and right down, left
    down and right up)
142.            for (u = 0; u < 4; u++)
143.            {
144.                int sumNum = 1;
145.                for (i = 1; sameColor; i++)
146.                    sumNum++;
147.                if (!inboard(row + dx[u] * i, col + dy[u] * i) || Board[row + dx[u] * i][col +
    dy[u] * i] != 0)
148.                    continue;
149.                for (i = -1; sameColor; i--)
150.                    sumNum++;
151.                if (!inboard(row + dx[u] * i, col + dy[u] * i) || Board[row + dx[u] * i][col +
    dy[u] * i] != 0)
152.                    continue;
153.                if (sumNum == 4)sum++;
154.            }
155.            return sum;
156.        }
157.        //compute the number of half open four
158.        int halfOpenFour(int row, int col)
159.        {
160.            int key = Board[row][col], sum = 0, i, u;
161.            for (u = 0; u < 8; u++)
162.            {
163.                int  sumNum = 0;
164.                bool flag = true;
165.                //Traversing 8 directions
166.                for (i = 1; sameColor || flag; i++)
167.                {
168.                    if (!sameColor)
169.                    {
170.                        if (flag&&Board[row + dx[u] * i][col + dy[u] * i])
171.                            sumNum -= 10;
172.                        flag = false;
173.                    }
174.                    sumNum++;
175.                }
176.                if (!inboard(row + dx[u] * --i, col + dy[u] * i))
177.                    continue;
```

```
178.                SND
179.                if (sumNum == 4)
180.                    sum++;
181.            }
182.            return sum - openFour(row, col) * 2;
183.        }
184.        //compute the number of open three
185.        int openThree(int row, int col)
186.        {
187.            int key = Board[row][col], sum = 0, i, u;
188.            //Search four directions
189.            for (u = 0; u < 4; u++)
190.            {
191.                int sumNum = 1;
192.                SPD checkVaildPoint i++; checkVaildPoint;
193.                SND checkVaildPoint i++; checkVaildPoint;
194.                if (sumNum == 3)
195.                    sum++;
196.            }
197.            //Search 8 directions
198.            for (u = 0; u < 8; u++)
199.            {
200.                int  sumNum = 0;
201.                bool flag = true;
202.                for (i = 1; sameColor || flag; i++)
203.                {
204.                    if (!sameColor)
205.                    {
206.                        if (flag&&Board[row + dx[u] * i][col + dy[u] * i])
207.                            sumNum -= 10;
208.                        flag = false;
209.                    }
210.                    sumNum++;
211.                }checkVaildPoint
212.                    if (Board[row + dx[u] * --i][col + dy[u] * i] == 0)
213.                        continue;
214.                SPD checkVaildPoint
215.                    if (sumNum == 3)sum++;
216.            }
217.            return sum;
218.        }
219.        //Check whether there is overline
220.        bool overline(int row, int col)
221.        {
222.            bool flag = false;
223.            int u;
224.            for (u = 0; u < 4; u++)
225.                if (num(row, col, u) + num(row, col, u + 4) > 4)
226.                    flag = true;
227.            return flag;
228.        }
229.        //Check disallowed moves after undering a pawn
230.        bool disallowedMove(int row, int col)
231.        {
232.            //Check the color of  player, there is no limitation for white.
233.            if (same(row, col, 2))
234.                return false;
235.            bool flag = openThree(row, col) > 1 || overline(row, col) || openFour(row, col) + h
    alfOpenFour(row, col) > 1;
236.            return flag;
237.        }
238.        //Check whether the game is end after undering a pawn on (row,col)
239.        bool gameEnd(int row, int col)
```

```
240.        {
241.            int u;
242.            for (u = 0; u < 4; u++)
243.                if (num(row, col, u) + num(row, col, u + 4) >= 4)
244.                    boolEnd = true;
245.            if (boolEnd)
246.                return true;
247.            boolEnd = disallowedMove(row, col);
248.            return boolEnd;
249.        }
250.        //under a pawn
251.        void go(int row, int col)
252.        {
253.            if (s == s0)
254.                Board[row][col] = -1;
255.            else
256.                Board[row][col] = -2;
257.            //Change the old △ to ○.
258.            for (int i = 0; i <= N; i++)
259.                for (int j = 0; j <= N; j++)
260.                {
261.                    //check whether the pawn is undered in this turn
262.                    if (i == row && j == col)
263.                        continue;
264.                    if (Board[i][j] < 0)
265.                        Board[i][j] *= -1;
266.                }
267.            DrawBoard();
268.            if (disallowedMove(row, col))
269.            {
270.                if (s0 == s_Computer)
271.                    cout<<"Winner is you";
272.                else
273.                    cout<<"Winner is computer";
274.                Sleep(10000);
275.            }
276.            if (gameEnd(row, col))
277.            {
278.                if (s == s_Computer)
279.                    cout<<"Winner is computer";
280.                else
281.                    cout<<"Winner is player";
282.                Sleep(10000);
283.            }
284.        }
285.        //compute the score
286.        int computeScore(int row, int col)
287.        {
288.            //Give disallowed moves zero
289.            if (disallowedMove(row, col))
290.                return 0;
291.            if (gameEnd(row, col))
292.            {
293.                boolEnd = false;
294.                return 10000;
295.            }
296.            int score = openFour(row, col) * 1000 + (halfOpenFour(row, col) + openThree(row, co
    l)) * 100;
297.            //search whether there is a pawn surrounding the position
298.            for (int i = 0; i < 8; i++)
299.                if (Board[row + dx[i]][col + dy[i]])
300.                    score++;
301.            return score;
```

```
302.        }
303.
304.        int Max2(int B)
305.        {
306.            int i, j;
307.            int beta = B;
308.            int alpha = -100000;
309.
310.            for (i = 1; i <= N; i++)
311.                for (j = 1; j <= N; j++)
312.                {
313.                    nodeNum++;
314.                    TToNN++;
315.                    //check whether the position is avaliable
316.                    if (!ok(i, j))
317.                        continue;
318.                    //put piece on the position
319.                    Board[i][j] = s0;
320.                    int pScore = computeScore(i, j);
321.                    //skip invaild position(disallowed moves' position or
322.                    //the position who has no neighbor chess piece)
323.                    if (pScore == 0)
324.                    {
325.                        Board[i][j] = 0;
326.                        continue;
327.                    }
328.                    //detemine if game can be end
329.                    if (pScore == 10000){
330.                        Board[i][j] = 0;
331.                        return 10000;
332.                    }
333.                    Board[i][j] = 0;
334.                    //find the maximum value
335.                    if (pScore> alpha)
336.                        alpha = pScore;
337.                    //Alpha-beta pruning
338.                    if (alpha > beta)
339.                        return alpha;
340.                }
341.            return alpha;
342.        }
343.        int Min(int A)
344.        {
345.            int i, j;
346.            int alpha = A;
347.            int beta = 100000;
348.            for (i = 1; i <= N; i++)
349.                for (j = 1; j <= N; j++){
350.                    nodeNum++;
351.                    TToNN++;
352.                    if (!ok(i, j))
353.                        continue;
354.                    Board[i][j] = 3 - s0;
355.                    int pScore = computeScore(i, j);
356.                    //skip invaild position(disallowed moves' position or
357.                    //the position who has no neighbor chess piece)
358.                    if (pScore == 0)
359.                    {
360.                        Board[i][j] = 0;
361.                        continue;
362.                    }
363.                    //detemine if game can be end
364.                    if (pScore == 10000){
```

```
                            Board[i][j] = 0;
                            return -10000;
                    }
                    pScore = Max2(beta);
                    Board[i][j] = 0;
                    //Find the min value
                    if (pScore < beta)
                        beta = pScore;
                    //Alpha-beta pruning
                    if (alpha > beta)
                        return beta;
                }
            return beta;
        }
        void Max1()
        {
            TToNN = 0;
            nodeNum++;
            TToNN++;
            DrawBoard();
            cout<<"It is computer's turn, please wait:  ";
            //if the center position of the board is availible,under a pawn on there
            if (Board[8][8] == 0)
                return go(8, 8);
            int i, j;
            int alpha = -100000, keyi, keyj;
            for (i = 1; i <= N; i++)
            {
                for (j = 1; j <= N; j++)
                {
                    nodeNum++;
                    TToNN++;
                    if (!ok(i, j))
                        continue;
                    Board[i][j] = s0;
                    int pScore = computeScore(i, j);
                    //skip invaild position(disallowed moves' position or
                    //the position who has no neighbor chess piece)
                    if (pScore == 0)
                    {
                        Board[i][j] = 0;
                        continue;
                    }
                    //detemine if game can be end
                    if (pScore == 10000)
                        return go(i, j);
                    pScore = Min(alpha);
                    Board[i][j] = 0;
                    //Find the max value
                    if (pScore > alpha) {
                        alpha = pScore;
                        keyi = i;
                        keyj = j;
                    }
                }
            }
            return go(keyi, keyj);
        }
        //Actions of player in every turn
        void player()
        {
            DrawBoard();
```

```
427.            cout << "All of the expanded nodes number is " << nodeNum << "." << endl;
428.            cout << "In this turn,the expanded nodes number is " << TToNN << "." << endl;
429.            cout<<"It is your turn, please input your position :  ";
430.            char c = '\n';
431.            int row = 0, col = 0;
432.            while (c < '0') {
433.                cin >> c;
434.                cin >> row;
435.            }
436.            //determine whether the first input value is lowercase
437.            if (c < 'a')
438.                col = c - 'A' + 1;
439.            else col = c - 'a' + 1;
440.            //determine whether the input position is vaild
441.            if (!ok(row, col)||c-'a'<0||c-'z'>0)
442.            {
443.                cout<<"It is not a vaild position";
444.                Sleep(1000);
445.                return player();
446.            }
447.            //under a pawn on the position
448.            go(row, col);
449.        }
450.        int main()
451.        {
452.            init();
453.            while (!boolEnd)
454.            {
455.                //determine whether this turn should be played by computer
456.                if (s == s_Computer)
457.                    Max1();
458.                else
459.                    player();
460.                //change the game turn
461.                s =3-s;
462.            }
463.            return 0;
464.        }
```