Tic Tac Toe

Two Players Games with Artificial Intelligence

Po-Teng Tseng
Thomas Lynn

Table of Contents

1. Introduction

Two players game is a classic problem in Artificial Intellegence field. We want to let computers solve the problems by themselves during the game. Search algorithms such as Alpha-Beta Search and MinMaxAB Search are common algorithms to try to get the best desicion in these two players games. With appropriate evaluation funcitons, the program can possibly find near perfect solutions, and sometimes they are better than human.

2.Contribution

2.1. Team Memebers

2.1.1. Po-Teng Tseng

Alpha-Beta Search
Po-Teng's Evaluation

2.1.2. Thomas Lynn

MinMaxAB Search
Thomas's Evaluation

3. Analysis of Problem

Tic-Tac-Toe is a well-known two players game. An big advantage of choosing this game is because its simplicity. Its rules are easy to understand, so it will also be easy to implement them in the program. The limited game path (9 moves maximum) also reduce the complexity of whole project. However, it has enough different possibilities in each game, so it's a good game to analysis. With these conditions, we don't have to worry too much about the problems of time and space, while we still can have enough data to examine our algorithms and evaluation functions.

4. Methodology

By building a program taking resiponsibility of holding game, we can simulate Tic-Tac-Toe games on it with two computer players. By implementing two algorithm "Alpha-Beta Search" and "MinMaxAB Search", we have tools to help make desicions in game. Three evaluation functions will be created to be the 'strategy' for each computer players.
We then make all combinations for different algorithm and evaluation functions to let two computer players fight each other.
By analysising the data from those games, we can evaluate each algorithm and each evaluation function.

5.Evaluation Function

5.1. Evaluation Function from Text Book

The more possible ways to win, the higher the score is.

5.2. Evaluation Function by Po-Teng Tseng

If the current player can win in the next few moves, those moves get +10 values.
If the current player can win in the next few moves, those moves get -20 values.
If the current player can make two marks on a line and there is no enemy mark, those moves get +1 value for each of these conditions on the board.
If the current player can make a mark on the center of the board, those moves get +5 values.

5.3. Evaluation Function by Thomas Lynn

This evaluation function simply gives higher score to the center and lower score to edges.

6. Improvement of Programs

6.1. Pruning

A big improvement of Alpha-Beta Search is its pruning feature. This feature greatly saves the time to go through nodes because it will abandon nodes and their sub-tree if they are impossible to be choosen.

6.2. Structure of tree

The first level (depth 1) of out tree has 9 nodes in total, as the first move has 9 possibilities.
The second level (depth 2) has 9*8 = 72 nodes in total, as each node in depth 1 has 8 children nodes which express 8 possibilities of next move. With the same idea, a node in depth 3 has 7 children nodes, and a node in depth 4 has 6 children nodes.
There will be 9 + 9*8 + 9*8*7 + 9*8*7*6 = 3609 nodes in total.

6.3. The reason of 9-8-7-6 nodes for each level.

It's the minimum number of required nodes to complete the algorithm. The first move will always have the most possibilities to concern, so we need to build a tree with decent size for the first move if we want to re-use the tree.
If we simply use 9 nodes for every depth, the total nodes are 9 + 9*9 + 9*9*9 + 9*9*9*9 = 7380. There is almost 200% difference by doing this.

7.Sample Run of Application

7.1. Alpha-Beta Search v.s. MinMaxAB

7.1.1.depth 2

7.1.2.depth 4

7.2. Evaluation Function (Text Book) v.s. (Po-Teng's)

7.2.1.Alpha-Beta Search

7.2.1.1. depth 2

Tested nodes (Po-Teng's) = 23 + 20 + 8 = 51
Tested nodes (Text Book) = 15 + 13 = 28

```
[]    []    []

[]    []    []

[]    []    []
```
The best move is 4, value = 8 . Make the move to 4

```
[]    []    []

[]    x     []

[]    []    []
```
The best move is 0, value = -20 . Make the move to 0

```
o     []    []

[]    x     []

[]    []    []
```
The best move is 0, value = 8 . Make the move to 0

```
o     x     []

[]    x     []

[]    []    []
```
The best move is 0, value = -20 . Make the move to 0

```
o     x     o

[]    x     []
```

[]	[]	[]
The best move is 3, value = 48 . Make the move to 3

o	x	o

[]	x	[]

[]	x	[]

In this run, Po-Teng's evalution beats the evalution in text book. We can see X player tried to occupied the center first, and to grab the chance of winning.

7.2.1.2. depth 4

Tested nodes (Po-Teng's) = 103 + 79 + 13 + 3 = 198
Tested nodes (Text Book) = 42 + 17 + 6 = 65

[]	[]	[]

[]	[]	[]

[]	[]	[]

The best move is 4, value = 8 . Make the move to 4
[]	[]	[]

[]	x	[]

[]	[]	[]

The best move is 0, value = -20 . Make the move to 0

o	[]	[]

[]	x	[]

[]	[]	[]

The best move is 2, value = 8 . Make the move to 2

o	[]	[]

x	x	[]

[]	[]	[]

The best move is 0, value = -20 . Make the move to 0

o o []

x x []

[] [] []

The best move is 0, value = 48 . Make the move to 0

o o x

x x []

[] [] []

The best move is 0, value = -2 . Make the move to 0

o o x

x x o

[] [] []

The best move is 0, value = 48 . Make the move to 0

o o x

x x o

x [] []

In this run, Po-Teng’s evalution beats the evalution in text book, but lost a chance to win once. It means this evaluation function is not perfect yet. We can still see X player tried to occupied the center first.

7.2.2.MinMaxAB

7.2.2.1.depth 2

7.2.2.2.depth 4

7.3. Evaluation Function (Text Book) v.s. Evaluation Function (Thomas's)

7.3.1.Alpha-Beta Search

7.3.1.1. depth 2

Tested nodes (Thomas's) = 23 + 17 + 8 = 48
Tested nodes (Text Book) = 15 + 15 + = 30

[] [] []

[] [] []

[] [] []
The best move is 4, value = 3 . Make the move to 4

[] [] []

[] x []

[] [] []

The best move is 0, value = -20 . Make the move to 0

o [] []

[] x []

[] [] []
The best move is 1, value = 5 . Make the move to 1

o [] x

[] x []

[] [] []
The best move is 1, value = -4 . Make the move to 1

o [] x

o x []

[] [] []

The best move is 2, value = 7 . Make the move to 2

o [] x

o x []

x [] []

Thomas’s evaluation function simply gives higher score to the center, and give lower score to the edges. As we can see, Thomas’s won the game by simply occupied corners and the center.

7.3.1.2. depth 4

Tested nodes (Thomas’s) = 106 + 58 + 13 + 3 + 1 = 181
Tested nodes (Text Book) = 42 + 22 + 10 + 2 = 76

[] [] []

[] [] []

[] [] []

The best move is 4, value = 4 . Make the move to 4

[] [] []

[] x []

[] [] []

The best move is 0, value = -20 . Make the move to 0

o [] []

[] x []

[] [] []

The best move is 1, value = 6 . Make the move to 1

o [] x

[]   x   []

[]   []   []

The best move is 0, value = -20 . Make the move to 0

o   o   x

[]   x   []

[]   []   []

The best move is 4, value = 9 . Make the move to 4

o   o   x

[]   x   []

[]   []   x

The best move is 0, value = -20 . Make the move to 0

o   o   x

o   x   []

[]   []   x

The best move is 2, value = 10 . Make the move to 2

o   o   x

o   x   []

[]   x   x

The best move is 0, value = -2 . Make the move to 0

o   o   x

o   x   o

[]   x   x

The best move is 0, value = 10 . Make the move to 0

o   o   x

o   x   o

x x x

As described in the depth 2 version, Thomas's simply grab the center and corners, so we can see it actually lost many chances to win. The result is still good as occuping the center is a big advantage among the games between two computers with non-perfect evaluation functions.

7.3.2. MinMaxAB

7.3.2.1. depth 2

7.3.2.2. depth 4

7.4. Evaluation Function (Thomas's) v.s. Evaluation Function (Po-Teng's)

7.4.1. Alpha-Beta Search

7.4.1.1. depth 2

Tested nodes (Thomas's) = 23 + 17 + 11 = 51
Tested nodes (Po-Teng's) = 14 + 18 + 6 = 38

[] [] []

[] [] []

[] [] []

The best move is 4, value = 3 . Make the move to 4

[] [] []

[] x []

[] [] []
The best move is 0, value = 5 . Make the move to 0

o [] []

[] x []

[] [] []
The best move is 1, value = 5 . Make the move to 1

o [] x

[] x []

[] [] []

The best move is 3, value = 5 . Make the move to 3

o [] x

[] x []

o [] []

The best move is 4, value = 7 . Make the move to 4

o [] x

[] x []

o [] x

The best move is 1, value = 45 . Make the move to 1

o [] x

o x []

o [] x

As we can see, although Thomas's tried to occupy the center, Po-Teng's has the ability to defend and attack. Po-Teng's tried to block enemy when enemy is about to win, and grab the chance when itself is about to win.

7.4.1.2. depth 4

Tested nodes (Thomas's) = 106 + 75 + 13 + 3 + 1 = 198
Tested nodes (Po-Teng's) = 58 + 21 + 10 + 2 = 91

[] [] []

[] [] []

[] [] []

The best move is 4, value = 4 . Make the move to 4

[] [] []

[] x []

[]	[]	[]

The best move is 0, value = -15 . Make the move to 0

o	[]	[]

[]	x	[]

[]	[]	[]

The best move is 1, value = 6 . Make the move to 1

o	[]	x

[]	x	[]

[]	[]	[]

The best move is 0, value = -15 . Make the move to 0

o	o	x

[]	x	[]

[]	[]	[]

The best move is 4, value = 9 . Make the move to 4

o	o	x

[]	x	[]

[]	[]	x

The best move is 2, value = -14 . Make the move to 2

o	o	x

[]	x	[]

o	[]	x

The best move is 0, value = 9 . Make the move to 0

o	o	x

x	x	[]

o	[]	x

The best move is 0, value = 5 . Make the move to 0 o o x

x x o

o [] x

The best move is 0, value = 9 . Make the move to 0

o o x

x x o

o x x

Draw. Po-Teng's doesn't work well in 4 depth as it does in 2 depth. The result is draw.

7.4.2. MinMaxAB

7.4.2.1. depth 2

7.4.2.2. depth 4

8. Table

8.1. Game Path Length

	text book	Po-Teng's	Thomas'
alpha-beta depth 2	30, 28	51, 38	51, 48
alpha-beta depth 4	76, 65	198, 91	198, 181

The table has two values because each evaluation function was used twice in the sample run.

8.2. Nodes Generated

The maximum number of nodes used in the first move is always 3610 for max depth 4, and 82 for max depth 2. Next move will need less nodes, and so on.

	Max Depth 2	Max Depth 4
Move #1	82	3610
Move #2	65	2081
Move #3	...	...
Move #4	...	...
Move #5	...	...
Move #6	...	...
Move #7	...	...
Move #8	...	...
Move #9	...	...

8.3. Win/Lose Result

alpha-beta	text book	Po-Teng's	Thomas's
text book		lose	lose
Po-Teng's	win		draw
Thomas's	win	draw	

minmaxAB	text book	Po-Teng's	Thomas's
text book		lose	lose
Po-Teng's	win		draw
Thomas's	win	draw	

9. Conclusion

With good evaluation functions, a computer can easily beat other computers. However, the evaluation functions we use here are not perfect, so we can easily find out sometimes they don't make the best move.
It seems like a more complicated evaluation funciton will get longer path, but the improvement of the algorithm can greatly reduce the game path and the cost of time and space.

10. Appendix

10.1. Source Code

```
#include <iostream>
#include <string>
#include <sstream>


using namespace std;


struct nodeMove
{
	int number;	//the number of this node in same depth
	int infoBoard[9];	//9 positions on the board
	nodeMove* nodeChild[9]; //maximum number of children is 9
};

struct result
{
	int value;
	string path;
};



int depthMax = 4; //the maximum depth for this algorithm
int widthMax = 9; //the maximum possible moves in each move

int sumTest = 0;
int alpha = -9999;
int beta = 9999;
```

```
int chooseEvaluationP1 = 0;
int chooseEvaluationP2 = 0;

int totalCheckEveryTurn = 0;


void test()
{
	sumTest++;
	cout <<"this #"<<sumTest<<endl;
}




int char_to_int(char a)
{
	int b = (int)a - 48;
	return b;
}

string int_to_string(int a)
{
	stringstream buffer;
	buffer << a;
	string b = buffer.str();
	return b;
}


//display the board
void display_board(int* listBoard)
{
	for (int i = 0; i < widthMax; i++)
	{
	if (listBoard[i] == 3)
	{
	cout << "x\t";
	}
	else if (listBoard[i] == 5)
	{
	cout << "o\t";
	}
	else if (listBoard[i] == 2)
	{
```

```
        cout << "[]\t";
        }
        if (i % 3 == 2)
        {
        cout << endl << endl;
        }
        }
}


//build the tree. create and link children to parents "recursively"
//it gets the reference of pointer to the parent node and the current depth,
//and returns the total number of nodes of it and its sub-tree
int create_tree(nodeMove*& parent, int* currentBoard, int depth)
{
        int sum = 0;    //the total number of all parent's children

        if (depth <= 0) { return 1; }   //reach the depth limit

        if (parent == NULL) //create root
        {
        nodeMove* nodeNew = new nodeMove;
        parent = nodeNew;
        parent->number = 0;
        for (int i = 0; i < widthMax; i++)
        {
        parent->infoBoard[i] = 0;
        parent->nodeChild[i] = NULL;
        }
        }

        //append children to parents
        //(widthMax - depthMax + depth) will be 9, 8, 7, 6 ... etc
        for (int i = 0; i < (widthMax - depthMax + depth); i++)
        {
        nodeMove* nodeNew = new nodeMove;
        nodeNew->number = i;
        parent->nodeChild[i] = nodeNew;
        sum += create_tree(nodeNew, currentBoard, depth - 1);
        }
        if (depth == 4) {cout <<"sum = "<<sum<<endl;}   //test message

        //return current node + all its children
        return 1 + sum;
}
```

```
void delete_tree(){}


string move_gen(nodeMove* parent, bool player)
{
	string stringSucc = "";
	int countNode = 0;  //which child to work with

	//read parent board
	for (int i = 0; i < widthMax; i++)
	{
	//if any position out of 9 can be put next move
	if (parent->infoBoard[i] == 2)
	{
	//put generated board info into children (generate children)
	for (int j = 0; j < widthMax; j++)
	{
		//empty position
		if (j == i)
		{
		//put next move depends on player
		//we don't have 9 children in each level so it's not 9 times
		if (player == true)
		{
		parent->nodeChild[countNode]->infoBoard[j] = 3;
		}
		else	//opponent's move
		{
		parent->nodeChild[countNode]->infoBoard[j] = 5;
		}
		}
		else	//not empty position
		{
		//copy rest of 8 positions
		parent->nodeChild[countNode]->infoBoard[j] = parent->infoBoard[j];
		}
	}

	//record which nodes are generated
	stringSucc += int_to_string(countNode);
	countNode++;
	}
	}

	//return a string describing which nodes are generated.
```

```
    return stringSucc;
}




int evaluationText(nodeMove* position, bool player)
{

    int playblock,
    oppblock;

    int playcount = 0,
    oppcount = 0;

    int score;

    int* grid;

    playblock = 5;
    oppblock = 3;

    grid = position->infoBoard;

    if (grid[0] != playblock && grid[1] != playblock && grid[2] != playblock)
    playcount++;

    if (grid[3] != playblock && grid[4] != playblock && grid[5] != playblock)
    playcount++;

    if (grid[6] != playblock && grid[7] != playblock && grid[8] != playblock)
    playcount++;

    if (grid[0] != playblock && grid[3] != playblock && grid[6] != playblock)
    playcount++;

    if (grid[1] != playblock && grid[4] != playblock && grid[7] != playblock)
    playcount++;

    if (grid[2] != playblock && grid[5] != playblock && grid[8] != playblock)
    playcount++;

    if (grid[0] != playblock && grid[4] != playblock && grid[8] != playblock)
    playcount++;
```

```
if (grid[2] != playblock && grid[4] != playblock && grid[6] != playblock)
playcount++;


if (grid[0] != oppblock && grid[1] != oppblock && grid[2] != oppblock)
oppcount++;

if (grid[3] != oppblock && grid[4] != oppblock && grid[5] != oppblock)
oppcount++;

if (grid[6] != oppblock && grid[7] != oppblock && grid[8] != oppblock)
oppcount++;

if (grid[0] != oppblock && grid[3] != oppblock && grid[6] != oppblock)
oppcount++;

if (grid[1] != oppblock && grid[4] != oppblock && grid[7] != oppblock)
oppcount++;

if (grid[2] != oppblock && grid[5] != oppblock && grid[8] != oppblock)
oppcount++;

if (grid[0] != oppblock && grid[4] != oppblock && grid[8] != oppblock)
oppcount++;

if (grid[2] != oppblock && grid[4] != oppblock && grid[6] != oppblock)
oppcount++;


score = playcount - oppcount;

if (grid[0] * grid[1] * grid[2] == 18)
score = 20;
else if (grid[3] * grid[4] * grid[5] == 18)
score = 20;
else if (grid[6] * grid[7] * grid[8] == 18)
score = 20;
else if (grid[0] * grid[3] * grid[6] == 18)
score = 20;
else if (grid[1] * grid[4] * grid[7] == 18)
score = 20;
else if (grid[2] * grid[5] * grid[8] == 18)
score = 20;
else if (grid[0] * grid[4] * grid[8] == 18)
score = 20;
else if (grid[2] * grid[4] * grid[6] == 18)
```

```
        score = 20;

        if (grid[0] * grid[1] * grid[2] == 50)
        score = -20;
        else if (grid[3] * grid[4] * grid[5] == 50)
        score = -20;
        else if (grid[6] * grid[7] * grid[8] == 50)
        score = -20;
        else if (grid[0] * grid[3] * grid[6] == 50)
        score = -20;
        else if (grid[1] * grid[4] * grid[7] == 50)
        score = -20;
        else if (grid[2] * grid[5] * grid[8] == 50)
        score = -20;
        else if (grid[0] * grid[4] * grid[8] == 50)
        score = -20;
        else if (grid[2] * grid[4] * grid[6] == 50)
        score = -20;

        if (player == false)
        score = -1 * score;

        return score;


}


int evaluationPodeng(nodeMove* nodeCurrent, bool player)
{
        int* grip = nodeCurrent->infoBoard;
        int valueCheck;
        int countCheck = 0;
        int center = 0;
        int playerMark;
        int win = 0;
        int valueWin;
        int lose = 0;
        int valueLose;

//display_board(grip);
for (int i = 0; i<9; i++)
{
        //cout << grip[i]<<" ";
}
//cout<<endl;
```

```
if (player)
{
playerMark = 3;
valueCheck = 18;
valueWin = 27;
valueLose = 125;
}
else
{
playerMark = 5;
valueCheck = 50;
valueWin = 125;
valueLose = 27;
}

int row[3] = {1, 1, 1};
int col[3] = {1, 1, 1};
int dia[3] = {1, 1, 1};

if (grip[4] == playerMark)
{
center += 3;
}

for (int i = 0; i < 9; i += 3)
{
col[0] *= grip[0 + i];
col[1] *= grip[1 + i];
col[2] *= grip[2 + i];
}
for (int i = 0; i < 3; i ++)
{
row[0] *= grip[0 + i];
row[1] *= grip[3 + i];
row[2] *= grip[6 + i];
}
dia[0] = grip[0] * grip[4] * grip[8];
dia[1] = grip[2] * grip[4] * grip[6];

for (int i = 0; i < 3; i++)
{
if (row[i] == valueCheck) { countCheck++; }
if (col[i] == valueCheck) { countCheck++; }
if (dia[i] == valueCheck) { countCheck++; }
```

```
        if (row[i] == valueWin) { win += 40; }
        if (col[i] == valueWin) { win += 40; }
        if (dia[i] == valueWin) { win += 40; }

        if (row[i] == valueLose) { lose -= 20; }
        if (col[i] == valueLose) { lose -= 20; }
        if (dia[i] == valueLose) { lose -= 20; }
        }

if((5 + win + center + countCheck)>7)
{
        //display_board(grip);
}

//cout<<"playerMark = "<< playerMark<<" return "<< (5 + win + lose + center +
countCheck)<<endl;

        if (depthMax%2 == 1)
        {
        return 5 + win + lose + center + countCheck;
        }
        else
        {
        return (5 + win + lose + center + countCheck);
        }

}


int evaluationTommy(nodeMove* position, bool player)
{
        int mark = 0;

        int score = 0;

        int* grid;

        if (player == true)
        mark = 3;
        else
        mark = 5;

        grid = position->infoBoard;

        if (grid[4] == mark)
```

```
        score = score + 3;

        if (grid[0] == mark)
        score = score + 2;

        if (grid[2] == mark)
        score = score + 2;

        if (grid[6] == mark)
        score = score + 2;

        if (grid[8] == mark)
        score = score + 2;

        if (grid[1] == mark)
        score = score + 1;

        if (grid[3] == mark)
        score = score + 1;

        if (grid[5] == mark)
        score = score + 1;

        if (grid[7] == mark)
        score = score + 1;

        cout << "Eval = " << score << endl;

        if (player == false)
        score = -1 * score;

        return score;
}

int evaluation(nodeMove* nodeCurrent, bool player)
{
        totalCheckEveryTurn++;
        if (player == true)        //player1
        {
        if (chooseEvaluationP1 == 1)
        {
        return evaluationText(nodeCurrent, player);
        }
        else if (chooseEvaluationP1 == 2)
        {
        return evaluationPodeng(nodeCurrent, player);
```

```
	}
	else if (chooseEvaluationP1 == 3)
	{
	return evaluationTommy(nodeCurrent, player);
	}
	}
	else if (player == false)		//player2
	{
	if (chooseEvaluationP2 == 1)
	{
	return evaluationText(nodeCurrent, player);
	}
	else if (chooseEvaluationP2 == 2)
	{
	return evaluationPodeng(nodeCurrent, player);
	}
	else if (chooseEvaluationP2 == 3)
	{
	return evaluationTommy(nodeCurrent, player);
	}
	}
}


result alpha_beta_search(nodeMove* , int , bool );
result max_value(nodeMove* , int , bool );
result min_value(nodeMove* , int , bool );
bool CheckWin(nodeMove*, bool);


result alpha_beta_search(nodeMove* nodeCurrent, int depth, bool player)
{
	result resultFinal;

	resultFinal = max_value(nodeCurrent, depth, player);

	//return action and value
	return resultFinal;
}

result max_value(nodeMove* nodeCurrent, int depth, bool player)
{

	result resultCurrent;
	result resultSucc;
```

```
		if (depth >= depthMax || CheckWin(nodeCurrent, player || CheckWin(nodeCurrent,
!player))) //leaves
		{

		resultCurrent.value = evaluation(nodeCurrent, player);
		resultCurrent.path = int_to_string( nodeCurrent->number );
		//cout << "Leave #"<<resultCurrent.path<<" value = "<<resultCurrent.value<<endl;
		return resultCurrent;
		}
		else if (depth < depthMax)
		{
		string successors ="";

		int scoreBest = 0;
		int valueNew = 0;

		//expand the node
		//move_gen gives a list of nodes

		successors = move_gen(nodeCurrent, player);

		//if successors is empty, return current values
		if (successors == "")
		{
		resultCurrent.value = evaluation(nodeCurrent, player);
		resultCurrent.path = int_to_string( nodeCurrent->number );
		return resultCurrent;
		} //no more move
		else //if it's not empty, examine each element in successors
		{
		scoreBest = -999999; //best score for any element in successors

		for (int i = 0; i < successors.length(); i++)
		{
			resultSucc = min_value(nodeCurrent->nodeChild[ char_to_int( successors[i] )
], depth + 1, player);
			valueNew = resultSucc.value;

			//
			if (valueNew > scoreBest)
			{
			//cout << "max choose " << valueNew<<" than " << scoreBest<<" depth =
"<<depth<<endl;
			scoreBest = valueNew;
			resultCurrent.path =  int_to_string(i);
			//cout << "i = "<<i<<endl;
```

```
			}

			if (scoreBest >= beta)
			{
			resultCurrent.value = scoreBest;
			cout << "\nPruning in max_value.  ";
//cout << "resultCurrent.value = " << resultCurrent.value << ", #" << i << " node. depth = 
"<<depth<<endl;

			if (depth != 0)
			{
			return resultCurrent;
			}
			}
			if (scoreBest > alpha)
			{
			alpha = scoreBest;
			}
		}

		resultCurrent.value = scoreBest;
		//resultCurrent.path = int_to_string( resultSucc->number );
		return resultCurrent;
		}
		}

}

result min_value(nodeMove* nodeCurrent, int depth, bool player)
{

		result resultCurrent;
		result resultSucc;
		if (depth >= depthMax || CheckWin(nodeCurrent, player) || CheckWin(nodeCurrent, 
!player)) //leaves
		{

		resultCurrent.value = evaluation(nodeCurrent, player);
		resultCurrent.path = int_to_string( nodeCurrent->number );
			//cout << "Leave #"<<resultCurrent.path<<" value = 
"<<resultCurrent.value<<endl;
		return resultCurrent;
		}
		else if (depth < depthMax)
		{
		string successors ="";
```

```
	int scoreBest = 0;
	int valueNew = 0;

	//expand the node
	//move_gen gives a list of nodes

	successors = move_gen(nodeCurrent, !player);

	//if successors is empty, return current values
	if (successors == "")
	{
	resultCurrent.value = evaluation(nodeCurrent, player);
	resultCurrent.path = int_to_string( nodeCurrent->number );
	return resultCurrent;
	} //no more move
	else //if it's not empty, examine each element in successors
	{
	scoreBest = 999999; //best score for any element in successors

	for (int i = 0; i < successors.length(); i++)
	{
		resultSucc = max_value(nodeCurrent->nodeChild[ char_to_int( successors[i] )
], depth + 1, player);
		valueNew = resultSucc.value;
		//
		if (valueNew < scoreBest)
		{
		//cout << "min choose " << valueNew<<endl;
		scoreBest = valueNew;
		resultCurrent.path = int_to_string(i);
		}

		if (scoreBest <= alpha)
		{
		resultCurrent.value = scoreBest;
		cout << "\nPruning in min_value.  ";
		//cout << "Depth = " << depth << ", #" << i << " node.\n";
		return resultCurrent;
		}
		if (scoreBest < beta)
		{
		beta = scoreBest;
		}
	}
```

```
            resultCurrent.value = scoreBest;
            return resultCurrent;
            }
            }


}
```

```
result alpha_beta_search_old(nodeMove* nodeCurrent, int depth, bool player)
{
            result resultCurrent;
            result resultSucc;

            if (depth < depthMax)
            {
            string successors ="";

            int scoreBest = 0;
            int valueNew = 0;
            string pathBest = "";

            //expand the node
            //move_gen gives a list of nodes

            successors = move_gen(nodeCurrent, player);

            //if successors is empty, return current values
            if (successors == "")
```

```
	{
		//cout << "successors == \"\", end." <<endl;
	resultCurrent.value = evaluation(nodeCurrent, player);
	resultCurrent.path = "";
	return resultCurrent;
	} //no more move
	else //if it's not empty, examine each element in successors
	{
	scoreBest = -999999; //best score for any element in successors

		//cout << "depth = "<<depth<<endl;

	//assume the best succ is the first child
	pathBest = "0";

	for (int i = 0; i < successors.length(); i++)
	{
		//convert the char in successors[i] to int
			//cout << "succ #"<<successors[i]<<endl;

		resultSucc = alpha_beta_search(nodeCurrent->nodeChild[ char_to_int(
successors[i] ) ], depth + 1, !player);
			//cout << "resultSucc.value = "<<resultSucc.value<<endl;
		valueNew = -(resultSucc.value);
			//cout << "Is "<<valueNew<<" > "<<scoreBest<<"?\n";


		//As the root also gets the reverse value, we choose the smaller one.
		if (valueNew > scoreBest)
		{
		scoreBest = valueNew;
		pathBest = int_to_string(nodeCurrent->nodeChild[i]->number);
		}
	}

	//append current node to the best path returned from children
	//convert int to string so it can be appended to another string

	cout << "depth = "<<depth << " pathBest = " <<pathBest<<" + "<<resultSucc.path<<"
= "<< pathBest<<endl;
	pathBest = pathBest + resultSucc.path;

	resultCurrent.value = scoreBest;
	resultCurrent.path = pathBest;

	return resultCurrent;
```

```
        }
        }
        else if (depth >= depthMax) //leaves
        {
        resultCurrent.value = evaluation(nodeCurrent, player);
        cout << "eval for #"<<nodeCurrent->number<<" node is
"<<resultCurrent.value<<endl<<endl;
        resultCurrent.path = "";
        if (nodeCurrent->number == 5)
        {
        //display_board(nodeCurrent->infoBoard);
        }

        return resultCurrent;
        }
}

bool CheckWin(nodeMove *position, bool player)
{
        int playmark;

        if (player == true)
        playmark = 3;
        else
        playmark = 5;

        bool win = false;

        int *grid;
        grid = position->infoBoard;

        if (grid[0] == playmark && grid[1] == playmark && grid[2] == playmark)
        win = true;
        else if (grid[3] == playmark && grid[4] == playmark && grid[5] == playmark)
        win = true;
        else if (grid[6] == playmark && grid[7] == playmark && grid[8] == playmark)
        win = true;
        else if (grid[0] == playmark && grid[3] == playmark && grid[6] == playmark)
        win = true;
        else if (grid[1] == playmark && grid[4] == playmark && grid[7] == playmark)
        win = true;
        else if (grid[2] == playmark && grid[5] == playmark && grid[8] == playmark)
        win = true;
        else if (grid[2] == playmark && grid[4] == playmark && grid[6] == playmark)
        win = true;
        else if (grid[0] == playmark && grid[4] == playmark && grid[8] == playmark)
```

```
	win = true;


	return win;
}




void play(nodeMove *&root)
{
	result resultFinal;
	bool player = true;

	//set board
	for (int i = 0; i < widthMax; i++)
	{
	root->infoBoard[i] = 2;
	}


	//start the game
	display_board(root->infoBoard);
	int countMove = 0;
	while (!CheckWin(root, player) && !CheckWin(root, !player) && countMove <= 8)
	{

	alpha = -99;
	beta = 99;
	//decide which move to make
	resultFinal = alpha_beta_search(root, 0, player);
	cout << "\nThe best move is " << resultFinal.path << ", value = "<< resultFinal.value
<<" . Make the move to " << resultFinal.path[0] << endl<<endl;

	//make the move and update the board
	for (int i = 0; i < widthMax; i++)
	{
	root->infoBoard[i] = root->nodeChild[ char_to_int( resultFinal.path[0] ) ]->infoBoard[i];
	}

	display_board(root->infoBoard);
	player = !player;
	countMove++;


	cout << totalCheckEveryTurn << " possibilities were checked.\n";
```

```
        totalCheckEveryTurn = 0;
        }

        if (CheckWin(root, player) || CheckWin(root, !player))
        {
        cout << "\n\nGame Ends.\n\n";
        }




        //decide which move to make


        //make the move and update the board

        return;

}



bool Opposite(bool player)
{
        if (player == true)
        return false;
        else
        return true;
}



int main()
{

        int board[9] = {5, 5, 2, 5, 3, 5, 5, 5, 5}; //for testing
        nodeMove* root = NULL;
        int sumNode = 0;

        //display_board(board);

        sumNode = create_tree(root, board, depthMax);
        cout << sumNode << " nodes created." << endl;

        cout << "Enter maximum depth:\n";
        cin >> depthMax;
```

```
	cout << "Choose evaluation function for Player1:\n1. Text Book  2.Po-Teng\'s 
3.Thomas\'s\n";
	cin >> chooseEvaluationP1;
	cout << "Choose evaluation function for Player2:\n1. Text Book  2.Po-Teng\'s 
3.Thomas\'s\n";
	cin >> chooseEvaluationP2;


	play(root);


	delete_tree();
	return 0;
}
```