Project 1　　basic ttt

1. Preparation for design

Problem goal:　as most research have done that there is no strategy to keep win in basic ttt, so the problem goal is that the intelligent computer can win people, at least make a draw.

State space:　the whole board state which will change as keep moving.

Initial state:　the empty board

Move:　fill any blank in current board.

Transform Model: after moving, some blank will be filled and board changes.

Goal test: judge in the current board and find who win the game by the game rule- 3 same mark in lines, rows, or diagonals. (or no one win and keep move)

Player: who should move now

Utility: who win the game will get their value. For my design, if 'X' win, get 1, else 'O' win, get -1. If get draw state, get 0.

2. Tictactoe Class

At first I decide to use a game entrance- can help me to start recursively and will let user choose what kind of move they wants to do(X or O). I'm ready to use io library to input char but I meet some bug like the picture below, I don't know what happens, so I changed my input by using scan function. Because scanner can not read char, so I use **'flag'** as mark to choose the move. At meantime I put flag into next class to function better. The project will call the **'run'** function to run the game. After one game is over, it will recover the result and mark to restart next game automatically.

3. Game Class

After that, I need a game class to define the whole game process. In this class, all the basic function happened in game will be define. The main function is 'run' function which control the whole process.

At beginning, I define all the global variable like: **board length is 3, the state board is a 2-D array with length of 3, and firstmove is 'X' and so on**.

Then I init the game. The **displayboard** variable is to display the current board in terminal. The **state** space is empty as initial state. **Mark** variable is to decide whose turn and is zero at beginning. The **result** variable is a switch to control the game and is true at first. If goal test find the terminal state, result will be false.

After that with **flag** variable from the game entrance, I will define people and computer represent which move(flag 1:first and 0: second).Then use **mark** to decide who need to move now. If it is people turn, terminal will ask people to make a choice about place they want to move then check legality of that choice, if not, will recursively ask people to input. If the move is legal, algorithm will use **place** array to record the choice and the state will change which function like transform model. After move, process will generate the state **node** with current state.(the node class will be declared detailed in next part)Then execute **isover** method to check node flag if it is terminal state. If it is, will change the result into false and return run method end, then this time game is over. If it is not terminal state, return back to **run** method and change mark variable to change turn, then computer method will execute.

If it is computer turn, computer at first will check if it is the first step of the game by counterstep used to count step of one game. If it is, at this moment, the state is empty and it is easy to think about the center move is the best move, so I just make computer do that move without think process. If it is not first step, then computer will create state node, get current's node **successors**, then evaluate their **scores**. And from all the successors of the node it chooses the most beneficial move to do. Then record the place, change state and check it is terminal state just as people method do. If not terminal state, the next move will be people.

4. GameTree Class

At final the class is GameTree class. This class is the most important part of the whole game, including many methods. The core method is **successor()** and **eval()**, separately representing the way to create search tree and heuristic function.

When call GameTree class, it will first initialize the parameter of the node state, like **flag**-used to judge if it is terminal state, **first** and **second**-the value of 'X' and 'O', **ppchoice** -user's choice about 'X' and 'O', **nextplayer**-who is the next turn, **score**-use the heuristic function to get the value of state which help computer to make choice, **successors**-a list containing all the possible next move as its successors got by method **successor()**, **position**-the move place which make the current state and board(and copy)-record current state.

Then is a method called **GameTree()**, including 4 input variable , the int **player**-will be the **nextplayer** value, int **[][]board**-represent the state and will be the board and value's value, int **position**-record about current move and be **this.position** value, int **flag**-represent people choice about X and O, and be value of **ppchoice. This.flag** is got by calling the next method **judgeend()** and judge it is terminal state.

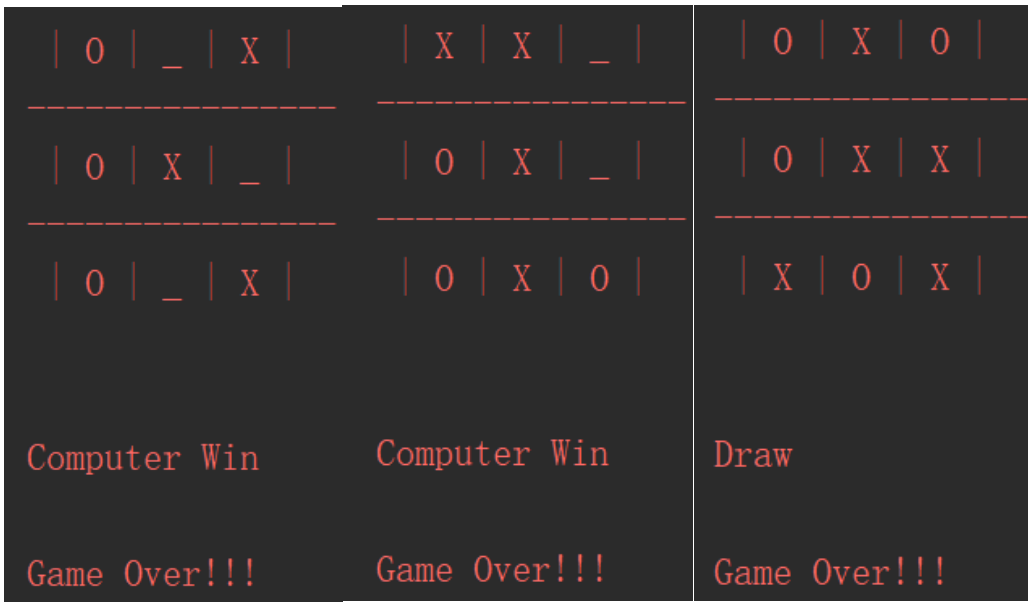
The next method is successor() used to create successors of a state, actually meaning all the possible next move. The successor() will not only create the next move which is deeper by just one step, but also create more deeply until the terminal state. SO it is a search tree. In this process, it adapts deep-first algorithm to generate all state. The successors' input is like that: One node's successor's nextplayer will be reversed with node's nextplayer, the state of successor will be parents' board filled with some blank which represent the place successor will move, the ppchoice is never changed. After next level of successors are created, the node board have to recover, which won't interface other successors' process. So by the way all the successors will be generated recursively.

Then it is judgeend() which judge the state space from lines, rows and diagonals, which is all the measures to be terminal state. If some rule is satisfied and it will check who win the game and return corresponding value to this.flag.(X win is 1, O win is -1, draw is 0).

The final one is eval(), which is aimed to return every node's value and help to find best move. Firstly if input node is terminal state, it will directly return the value by rules. If not, it also adapt deep-first idea to get the deepest level successors and get their score, then use the minmax algorithm--if the upper level successors' nextplayer is X, it will get its own score by choose the largest score of its successors, because for X, it will as large as possible to get better benefit. If nextplayer is 0, it will choose the smallest one as score.

5. Results

It is all the process of part-1 ttt. The pictures below are the results of test:



The thinking time for computer is really fast and within 0.5s, so the strategy is fine for basic ttt.

Project 2　super ttt

1. Preparation for design

Problem goal:　same

State space:　**a larger state space than basic ttt state space, which has 9 single board in a big board.**

Initial state:　same

Move:　fill any blank in current board with special rule that the firstmove choose which board to play then the next player must play on the board in the corresponding position in the grid. And the place on that board is not required. If the required board is full, the nextplayer can choose another board.

Transform Model: same

Goal test: same as part 1.

Player: same

Utility: who win the game will get their value. For my design, if 'X' win, get 1, else 'O' win, get -1. If get draw state, get 0. **Otherwise due to the larger search space, so sometimes algorithm can not get the end node. So at some search level, algorithm will judge the node there as terminal node, then evaluate them.**

2. Tictactoe Class

Almost the same as part 1

3. Game Class

Most part is the same as part 1. The major differences are those:

1. The **state** space become a 3-D array so can contain 9 boards' states.
2. The **place** will not only record the place in single board, it also record the board place in the bigger board. So place become array with size 2.
3. Original 2-D array **board** is used to represent the current single board, which is more convenient than handle the whole state space.
4. use int **order** to represent the single board place in big board.

The differences between methods majorly concentrate in details, but their core ideas are not change.

1. **Isover() :** in part 1, there is only 4 different value of flag (-2,-1,0,1), but in part-2, there is a special condition when the required board is full, the nextplayer can choose another board and it can not be judged as draw state.

Only when all 9 boards are full, it can be draw terminal state. It defines that state's value be 3. So flag equals zero are not judge to terminal state again.

2. **Give():** this is a new method as transform model which is used to put current single **board** states into the big board **state** so the state space will change.

3. **Back():** due to the game rule, if player put some place in single board, the next one need to move in corresponding board. So after moving, there will be a required single board in that case call **back()** to input state data into board array from the whole state space.

4. **Displayboard():** it just spend more time to print every single board, other thing is same as part 1.

5. **People():** the same as part one. There is a small change. When array place changes, the original place[1] will be new place[0] if required board is not full.

6. **Computer():** almost same, the differences are if the required board is full, it will call **inputerorder()** to ask people to choose another board because after computer move if it is not terminal state, it must be people move, so I can call **inputerorder()** in **computer()** rather than **people()**. And if the computer is first step, it will still choose the center, but in which single board center is decided by random number between 1-9. I think it is same in initial state and don't need special choice. By the way, the computer don't need to think if the required board is full because if it is full, the node successors will be more and different, but the election process is same.

7. **Inputoder():** it is used to get the choice from people about which single board they want to play with when the game start and people is first, or the required board is full.

8. **Run():** almost same except adding a order variable representing the single board place in big board.

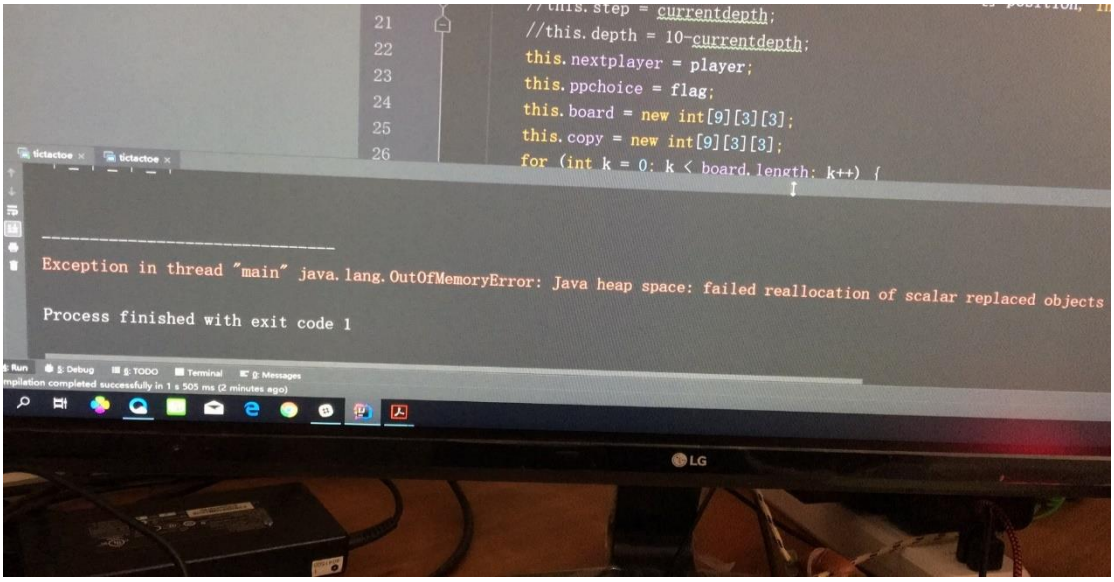
4. GameTree Class

Most part is same, there are differences:

1. Add 2 new parameters **maxdepth** and **depth.** The former represent the deepest level the algorithm can search which is not like part 1 because the whole state space is huge and RAM can not support that. It use the idea that even the node isn't terminal state, but I can let algorithm to decide it is terminal. The latter variable depth records current node's depth with initialization as 1. So in recursive search the method can stop when depth is more than maxdepth. Another 2 variable is max and min which will be larger or less than the best score and worst score. And they will help to execute a-b pruning function.

2. **Successor() and Createsuc():** in fact it does not change in nature, but for simplicity of code I make the redundant code as a method **creatsuc().** Then I add some judgements to decide how to generate successors, like if flag is zero which means required board is full, so the successor will enlarge to the whole state space, if the node is terminal node, it will not have successors. Furthermore, I need to use a-b algorithm to prune, so I put heuristic function **eval() and calUtility()** into the process, instead of functioning separately. In that case, after reach the terminal node heuristic function will get a score for terminal node, then if the parent node's nextplayer is first, when successors' score it will change parent's min to the score. After that when the next successor's score is smaller than parents' min, the successor will be abandoned. It is similar idea when parents' nextplayer is second. Through this strategy, the number of successors node will largely decreased, so the maxdepth can be deeper.

Performance：when there is no pruning measure, if maxdepth is more than 4, the program will create memory error like below:

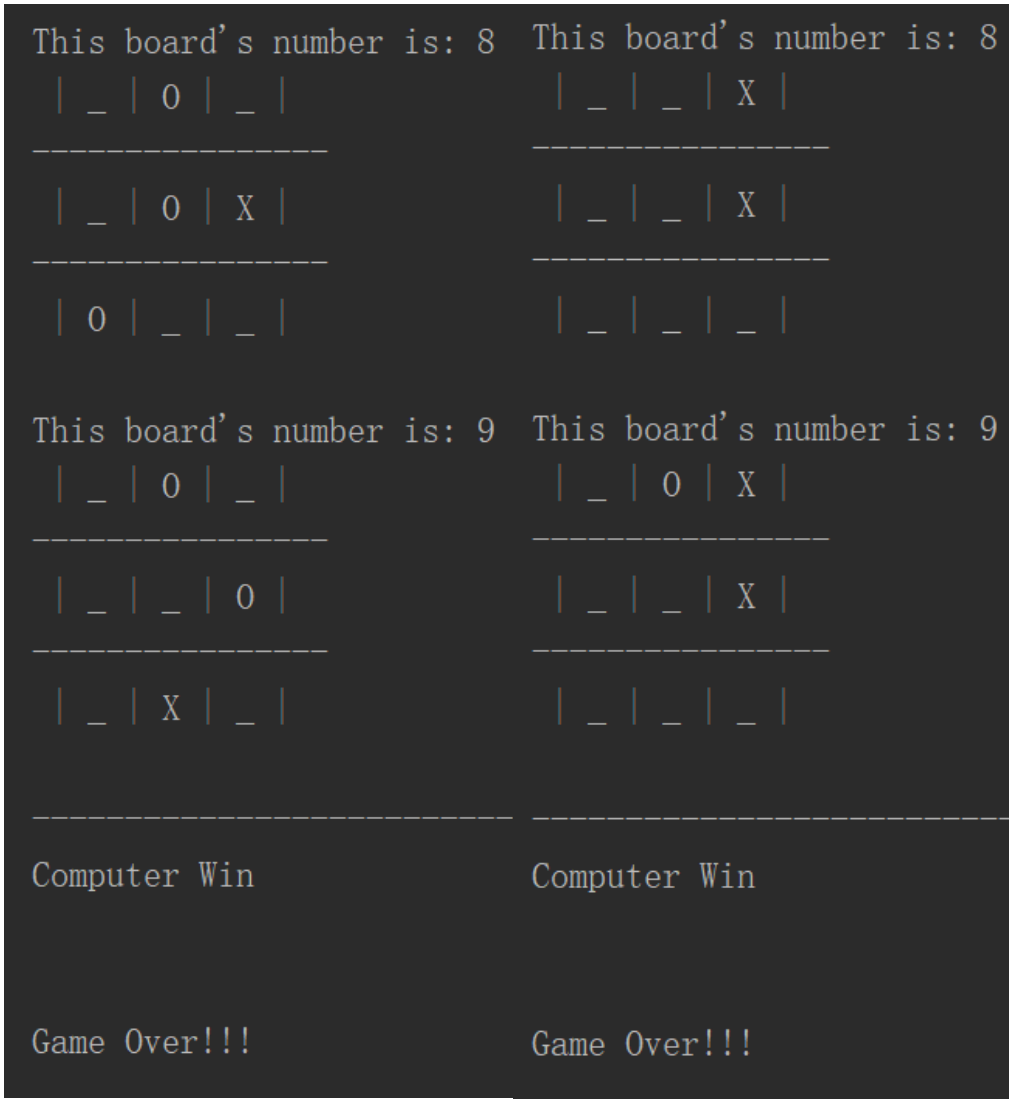


After using a-b pruning, the algorithm can deep into at least 9 levels, even though it will take so much time to deal data, but it can work. In the debug process, without the a-b, every level of successor node will be more than 7 at least, usually the 8 and 9 in the beginning. It needs large RAM to deal with. But with a-b pruning, basically the number of successors have been down to 5, so the space will largely saved and can search much deeper level.

3. **Judgeend():** there is no change basically, because even though the state space are larger, but every time the algorithm only process one single board, so what I need is to point to that board by position[0], then other process is the same. Moreover, there is one more parameter called **int[] status** to save every single board state in big board . If the single board is full, the corresponding value in status will be 1. Then for flag the draw state's value is 3 when all single boards are full.

4. **End(), min(),max(),calUtility() and eval():** they are heuristic function. The end(), min(),max() and eval() are same with eval() in part 1 in nature. Just simplify the code. The calUtility() is used to calculate score for the node whose depth is more than maxdepth but they are not seriously terminal node. In that case, I decide to use math strategy that for every single board firstly refill their blanks with X mark, then sum all the scores belonging to the ways can win. Then use the same procedures to refill the blanks with O marks then sum scores. In the end I add two sums together and get a final score for every single board. Then I choose the most benefit score from them as their parents' score. In this case, what I do is to calculate the chance to win and lose for every single board, then get together, which is a good way for evaluation. If the board existed mark is really great for X or O to win, they will definitely get a score while represent such trend.

5. Result



The time of thinking is a little slow and when the maxdepth is 8, computer needs to spend about 4 mins. As the maxdepth decreases, time for thinking largely goes down. In this case, when applying into reality, the maxdepth needs to changes as limitation about time.

Project 3 ultimate ttt

1. Preparation for design

Problem goal: same as part 1.

State space: **a larger state space than basic ttt state space which is 3-D cube of boards. Every board's length is 4 and totally have 64 blanks.**

Initial state: same

Move: same as the project 1 rule, can move any blank with 2 numbers to place the blank.

Transform Model: same

Goal test: judge in the current board and find who win the game by the game rule- 3 same marks in lines, rows, or diagonals in any kind 4*4 board, no matter how exist it does(or no one win and keep move)(same as project 1)

Player: same

Utility: same as part 2.

2. Tictactoe Class

Almost the same as part 1

3. Game Class

Most part is the same as part 1 and 2. The major differences are those:

1. The **state** space length is 4.
2. The **place** record the height (1-4) and place in single board(1-16)
3. **order** represent the height.

The differences between methods majorly concentrate in details, but their core ideas are not change.

1. **Isover() :** same as part 1.
2. **Displayboard():** same as part 2.
3. **People():** same as part 1, just ask another place about height.
4. **Computer():** almost same as part 1, only with one more parameter about height.
5. **Inputoder():** now it is used to ask about height choice for people.
6. **Run():** same as part 1.

4. GameTree Class

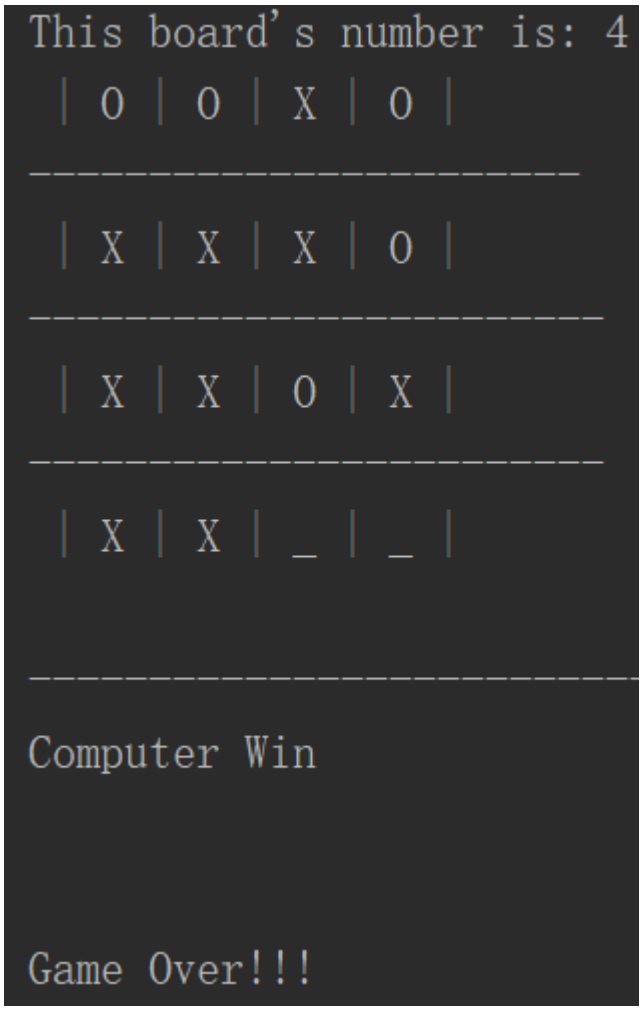
Almost all the methods and ideas are same as part 2, but the core difference is that the measures to win is enlarged much, so the heuristic function correspondingly becomes more complicated.

As the rules of Qubic says, as long as there is one 4 same marks can linked as lines, the game is over. In that case, after serious calculate, there are 76 ways to end the game. The text below is cited from Wikipedia:

‘On the 4x4x4 board there are 76 winning lines. On each of the four 4x4 boards, or horizontal planes, there are four columns, four rows, and two diagonals, accounting for 40 lines. There are 16 vertical lines, each ascending from a cell on the bottom board through the corresponding cells on the other boards. There are eight vertically-oriented planes parallel to the sides of the boards, each of these adding two more diagonals (the horizontal and vertical lines of these planes have already been counted). Finally, there are two vertically-oriented planes that include the diagonal lines of the 4x4 boards, and each of these contributes two more diagonal lines—each of these including two corners and two internal cells.’

In this case, the **judgeend()** will need to contain all the way to end. Firstly, I check the single board from up to down by call **check_up_to_down()** which is not hard to understand and just need be careful. Then I use **check_vertic()** to check 16 poles in the cube. Finally I check diagonals through **check_diagonal()**, which check every line formed by every pair of vertex except normal rows and columns which have been checked before. After these checks, it can make sure if the state is terminal. The **max(),min(),end(),eval()** is same as part 2. The **calUtility()** is same in idea that by refill blanks to calculate the chance of win and lose, then sum them. The difference is that there will be 76 ways to win so it needs to calculate count carefully.

5. Result



For this part ,the successor for every parents is too much, so even though I adapt the a-b pruning the time for running is unbearable. And the game is so complicated, debug for adjusting for heuristic is too much cost, so maybe there are many drawbacks but I don't have time to analyze them.