CS 6613 Project Instruction

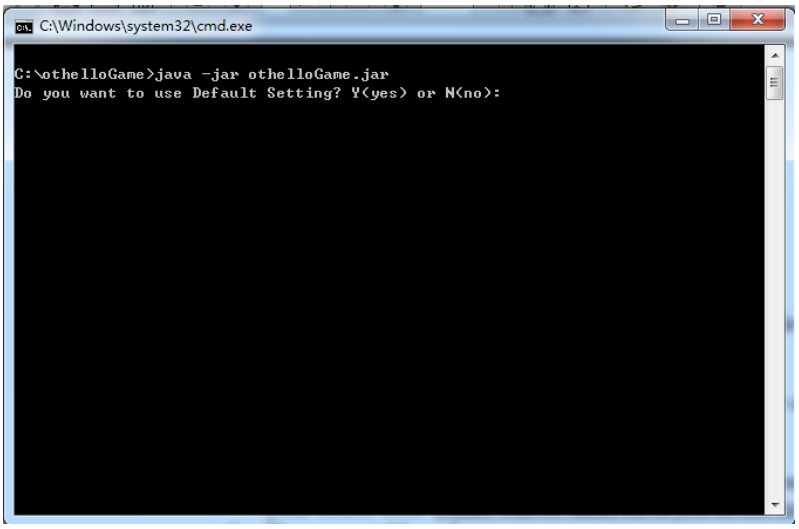
Name: Yang Xu Poly ID: 0447755

1. How to compile and run your program:

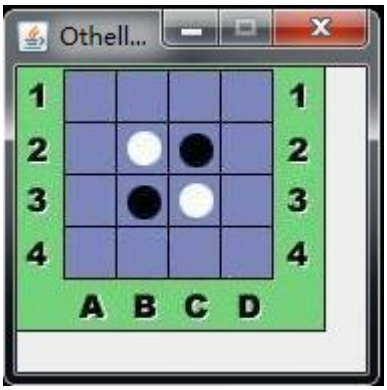
My code is written by using Java. And I use Eclipse to develop it. The detailed information could be found in the development document. I hand in both the source code and the compiled program.

(1) Run compiled program:

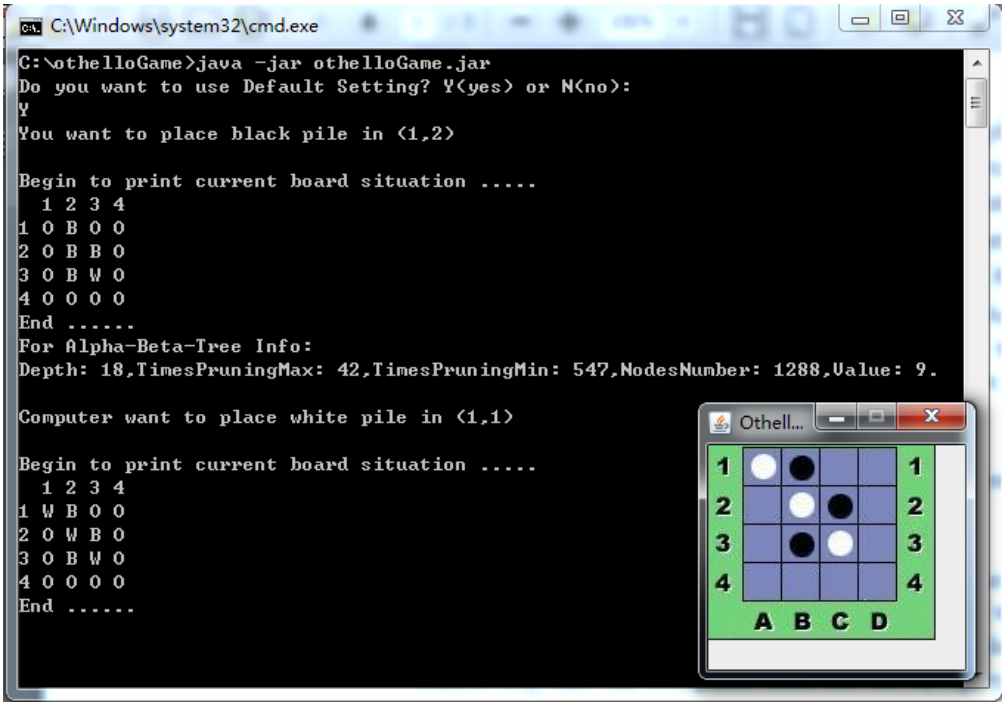
In order to run the compiled program, you just need to have the JRE environment. For Windows system, you can just double click "main.bat". For Linux system, you need to run "java -jar othelloGame.jar" in your terminal (I haven't tested it in Linux). After you run, then a command view would show like below:



You can select the game setting here. After you select the setting, then a graphical GUI would show like below:

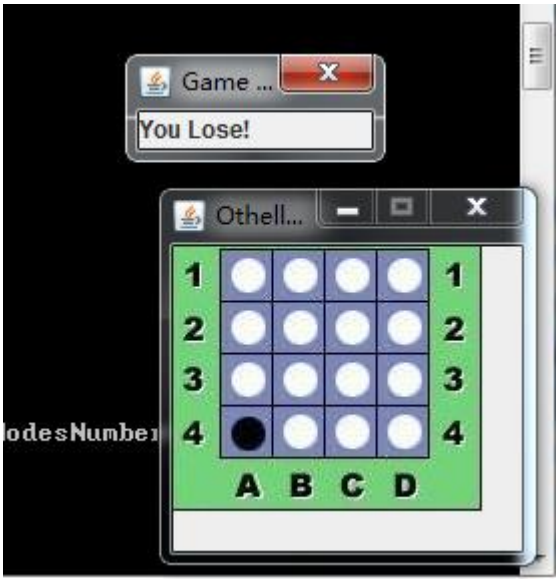


you can use the mouse to pick the position to place the disc. At the same time, the terminal would display some information about the program, like below:



In that information, I display who place a disc and the situation of the board after every move. For the computer move, I display maximum depth of tree (Depth), total number of nodes generated (NodesNumber), number of times pruning occurred within the MAX-VALUE function (TimesPruningMin) and number of times pruning occurred within the MIN-VALUE function (TimesPruningMax).

Finally, a small window would emerge to display the final result like below:



(2) Run source code:

In order to run the source code, you need Eclipse + JDK. My code is written by using Eclipse IDE. You can import my project into the eclipse. Then you can run two java file, which corresponds to two different game display modes.

“CommandLineMode.java” corresponds to the command line game mode. If you run that

java file, you can play the game in the pure command line form. I haven't add that function in the compiled program.
"GUIMode.java" corresponds to the GUI mode, which is the same as the compiled program.

2. High Level Description:

In the implementation of the game, I realize all the functions mentioned in the project file, two extra parts also included.

The most important algorithm in this implementation is the Alpha-Beta Search Algorithm, which is like below:

function ALPHA-BETA-SEARCH(*state*) **returns** an action
  v ← MAX-VALUE(*state*, -∞, + ∞)
  return the *action* in ACTIONS(*state*) with value *v*

function MAX-VALUE(state, *α, β*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  v ← -∞
  for each *a* **in** ACTIONS(*state*) **do**
    v ← MAX(*v*, MIN-VALUE(RESULT(*state*, *a*), *α*, *β*))
    if *v* ≥ *β* **then return** *v*
    α ← MAX(*α, v*)
  return *v*

function MIN-VALUE(*state*, *α, β*) **returns** *a utility value*
  if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
  v ← +∞
  for each *a* **in** ACTIONS(*state*) **do**
    v ← MIN(*v*, MAX-VALUE(RESULT(*state*, *a*), *α*, *β*))
    if *v* ≤ *α* **then return** *v*
    β ← MIN(*β, v*)
  return *v*

When computer player turns to place a disc, he would use Alpha-Beta Algorithms to search the current board condition. In Alpha-Beta Algorithms, the computer would choose to play the best strategy, which is the default setting in my program. Under such condition, people could hardly win.

In order to make the program has several difficulty levels. I just use cutting off search strategy. And we just need to replace Terminal-Test function by CUTOFF-TEST(state,depth) function and replace Utility function by Eval function.
CUTOFF-TEST(state,depth) function would return true when the search tree reach the final state nodes or the search tree branch reaches a certain depth. Thus here, different difficulty

levels just correspond to different search depth. The higher the search depth is, the more difficulty the game would be. Under such philosophy, I design different difficulty levels.
At this time, the search tree may not reach the final state nodes. Thus we need a Eval function to replace the Utility function. My Eval function place weight 16 for the 4 vertex points, weight 4 for the 8 points on the board side and weight 1 for the 4 point on the middle. My heuristics come from the observation that: when you win the vertex position, it would belong to you forever. For the middle position, it would change it owner very frequently. The position on the side would change less frequently. From these observations, I place different weights for different point positions. In this way, I realize the program having different difficulty level function.

Before the move of player, I would check whether the program ends. If the program ends, I would display the result. I would also check whether this user has valid move. If not, the opponent would continue with the next move. For this part, the command line would react immediately when game ends or user having no valid move. In the GUI mode, I add listener to the user mouse movement and also add these judgment in that listener. Thus, if the user has no valid move, the program would show that he has no valid move only when he clicks the board, not immediately. However, such behavior doesn't affect the use of the program. Thus, I didn't fix it.

To check whether the user has valid move, I just search very possible position for the user. And starting from this position, I check whether the move in that position would outflank the other color discs. That is the basic idea of check valid movement. To check whether game ends, I just check whether one of two users have valid move or not.

For GUI part, I first use command line to let user set the game enviable like game difficulty, whether go first, etc. Then the GUI would emerge. I add the listener on the mouse click. When you click on the board, the corresponding result would show: your move fail (Your move doesn't result in outflanking of discs), your move succeeds and the computer responds accordingly, game ends, etc.

This is high level description of my design, more detailed parts could be found in my development document and program codes.