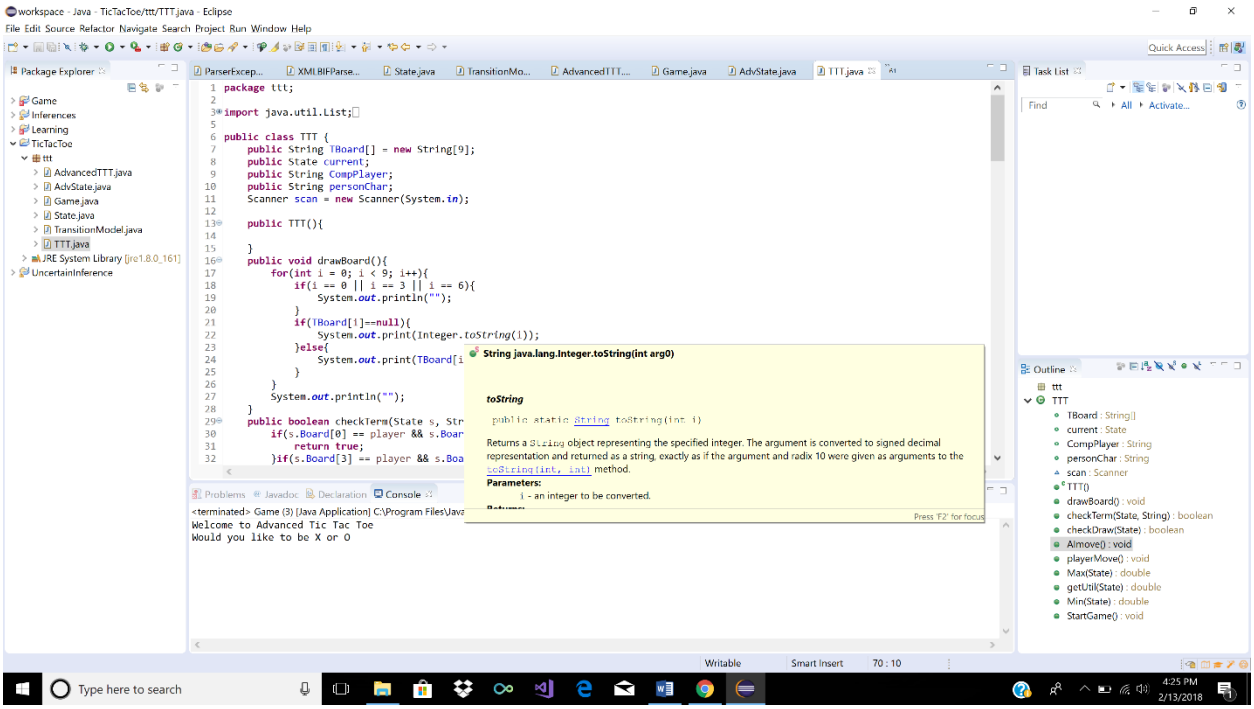
Thomas Burt

CSC 242

February 13, 2018

Writeup

To start off the project, the first thing I had to take into account was how to represent the basic tic tac toe problem. My first thought was that I needed to implement the right data structures for the board, the basic game, states, and actions. First, I started off by making a State class, which the coding is for a page or two below. This just contained the String name of the player and a String array of length 9, which represented a possible board combination. After, I then made a TTT class for Basic Tic-tac-toe. This contains the main board and functions that help with the basic tic tac toe program. The following are my global variables for the class.



I thought about making a search tree at first that way I could try to make it all at once so that it didn't need to use an algorithm to search for the best move. However, I quickly found out that this was not the way to go. Following the MiniMax algorithm from the book and using the basic general idea of Professor Ferguson's state space search algorithm, or best algorithm ever. The idea was to take the initial state expand it using depth first search, meaning to expand the initial state, then its field, and that child's first child, etc. This would then setup a layout of the search tree without creating it thus saving space efficiency. This stopped me from making a search tree, but instead I made a State class in my project. The State class would consist of the board, player

symbol (meaning which player's turn it was) and the successors of the state. I thought about making a transition model class, which was mentioned in the project instructions, but I felt that it was overdoing it and that, if needed, I could easily implement it later. I saw no need for the transition model. Following this I made my MiniMax algorithm functions in the TTT class called Min and Max. These functions had helper functions like checkTerm(state s, String player) which checked if the state was a terminal state with the specified player. This helped a lot with the getUtil functions as well.

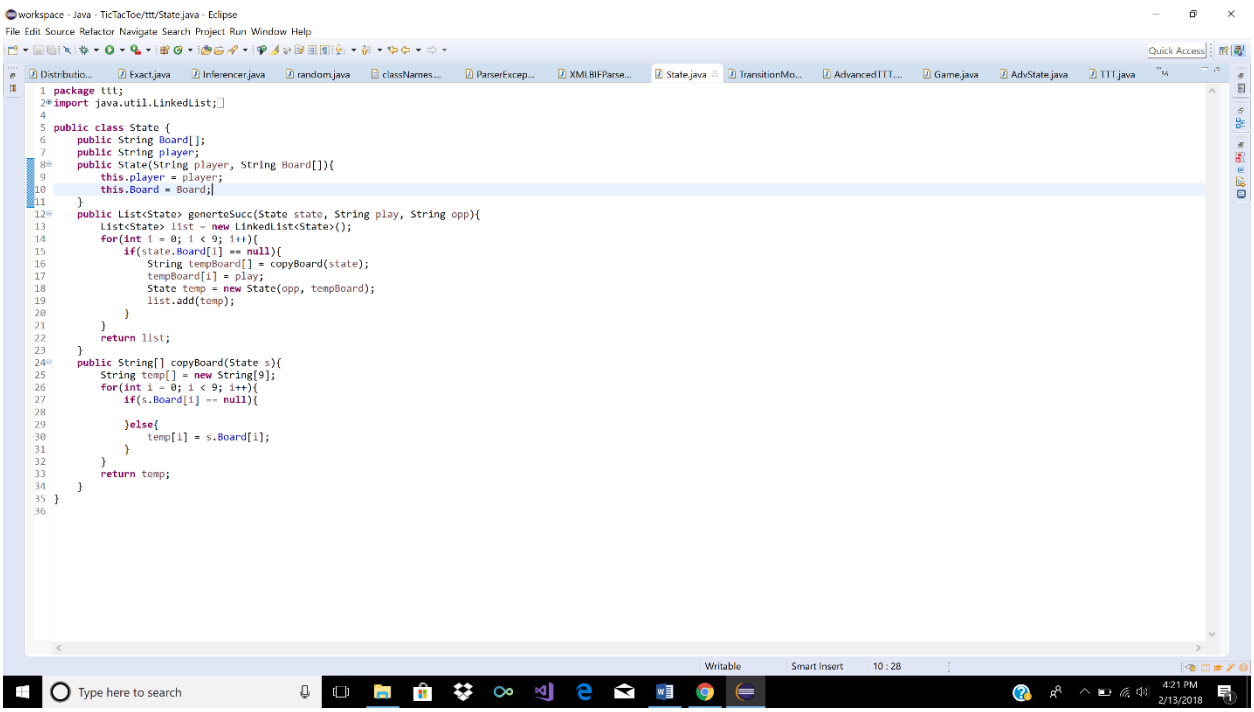
Once I was done with the MiniMax algorithm I started testing it. At first it was being completely crazy. It would output in the first move:

OOOOOO

OOOO

XOO

This was very funky, but I found out it was my generteSucc function in my State class which was causing it. I was referencing the state whose successors I was trying to generate; thus, its table was changed when exploring the options. So, I made a new function called copyBoard(), which made a brand new board all copied from the root State's board. This solved the problem with the crazy board. The changes and present State class coding is in the picture below.



The next problem I ran into was that my MiniMax was not stopping easy wins on my part. This was an easy fix because I realized I forgot to implement function that checks if there is a draw. So, I made a checkDraw function, added it to my Min, Max and, most importantly, getUtil function so that the computer would go for a draw rather than seeking out a win when I am clearly about to win. This solve the problem and the program was almost perfect, however, I was still able to use

strategies that people players don't see, but the program should. An example of a board where I played this strategy is:

From

O2X

O56

78X

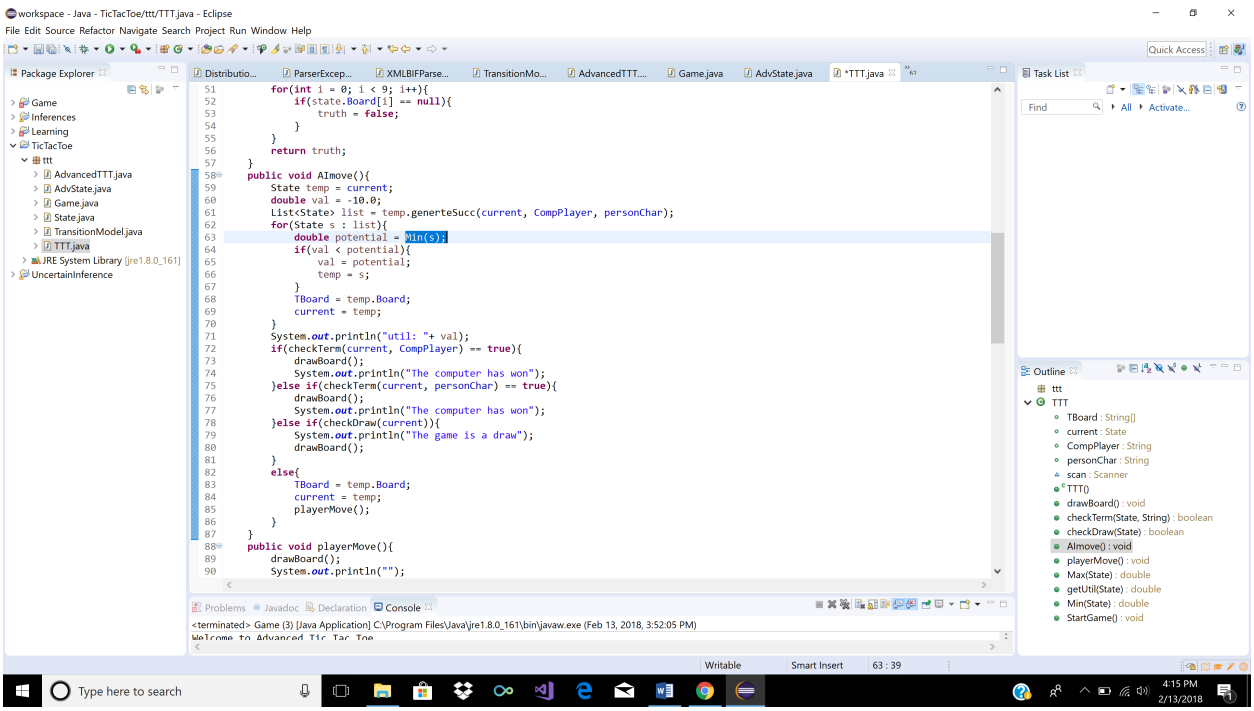
to

O2X

O56

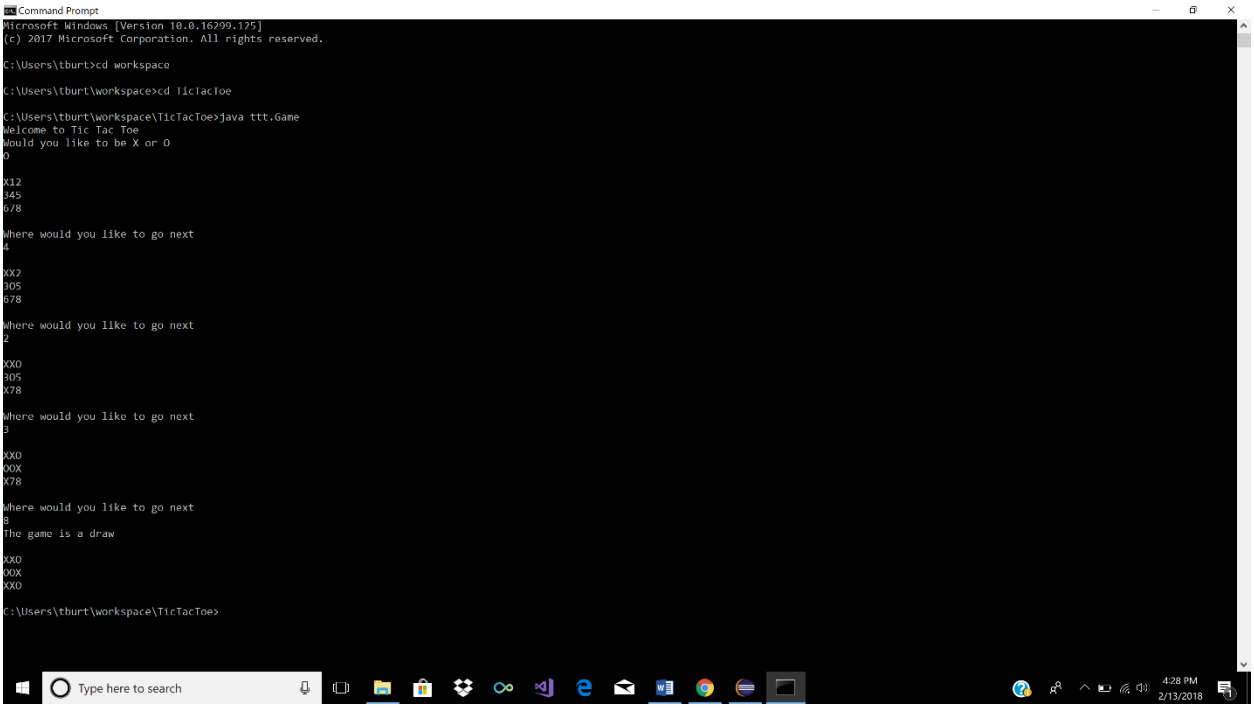
X8X

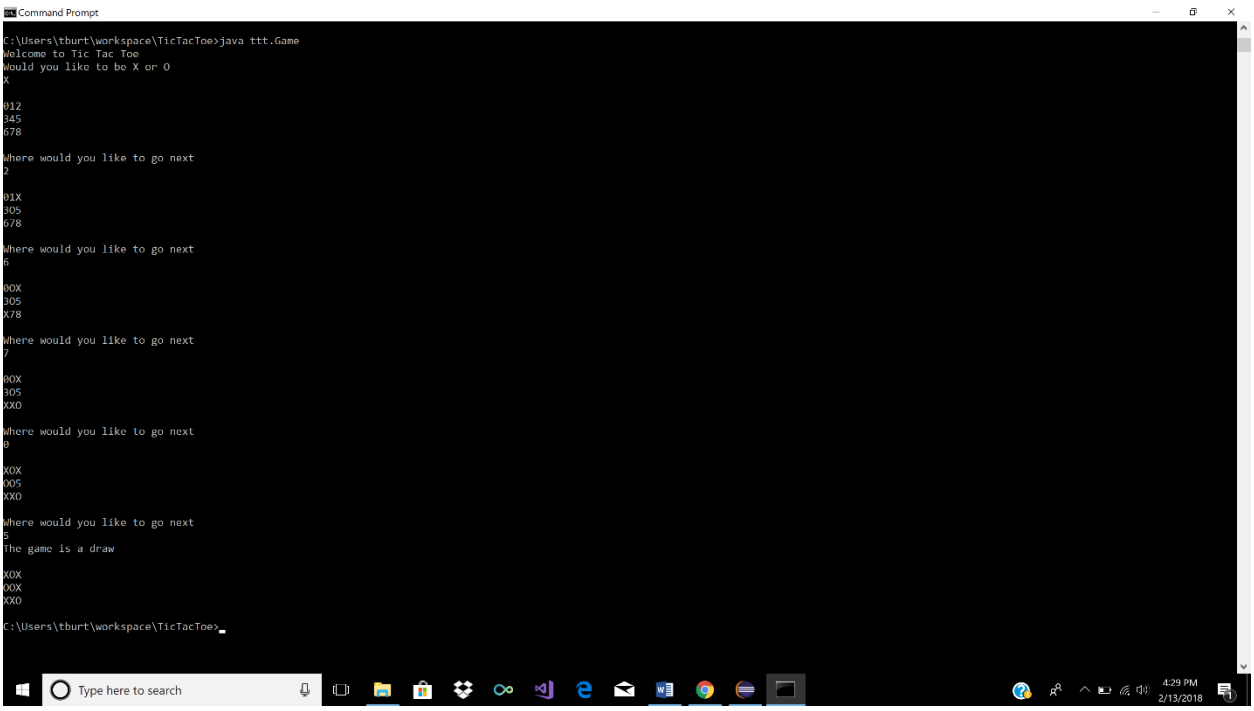
I am the X in this instance and this got the computer stuck because there is no conceivable way the computer could stop my win; thus, the program was not ideal. I took a break for a couple days and then looked back into it and found something that I did wrong. In the function AImove(), the computer would take the max value of the max function called on all successors. This is incorrect because the algorithm takes the max value of the min value function called on all successors. I fixed this error by changing the max call to the min call which is highlighted in the picture below.



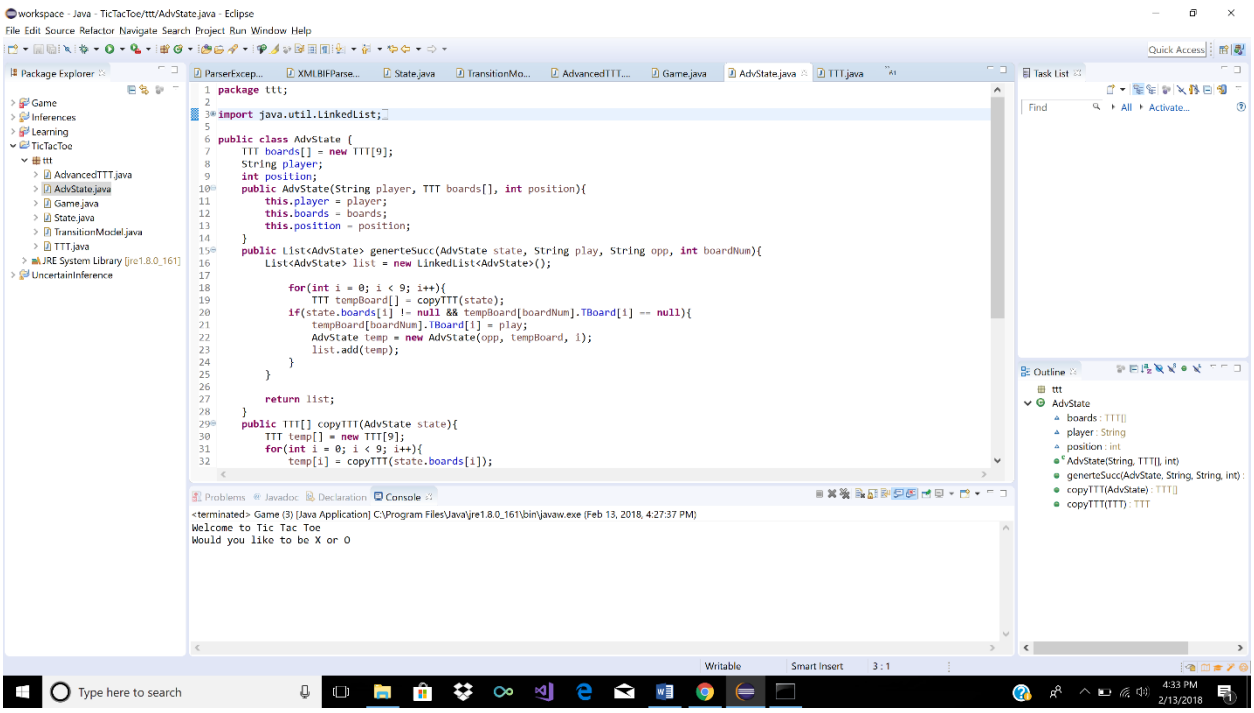
After this error was solved my Basic Tic Tac Toe program was complete here the first picture below is the program never letting me win. The second picture is the program stopping me from

doing my strategy that was beating it before. Please note that this is not the final way I printed out the project but shows that my program is at its peak in terms of MiniMax.





For Advanced Tic-tac-toe, the process of representing the 9 tic tac toe boards was easy. I reused the TTT class that was used for Basic Tic-tac-toe and made a length 9 array of TTT[]. This gave me my 9 boards, which was stored as a global variable. I also created an AdvState class which had the following global variables:



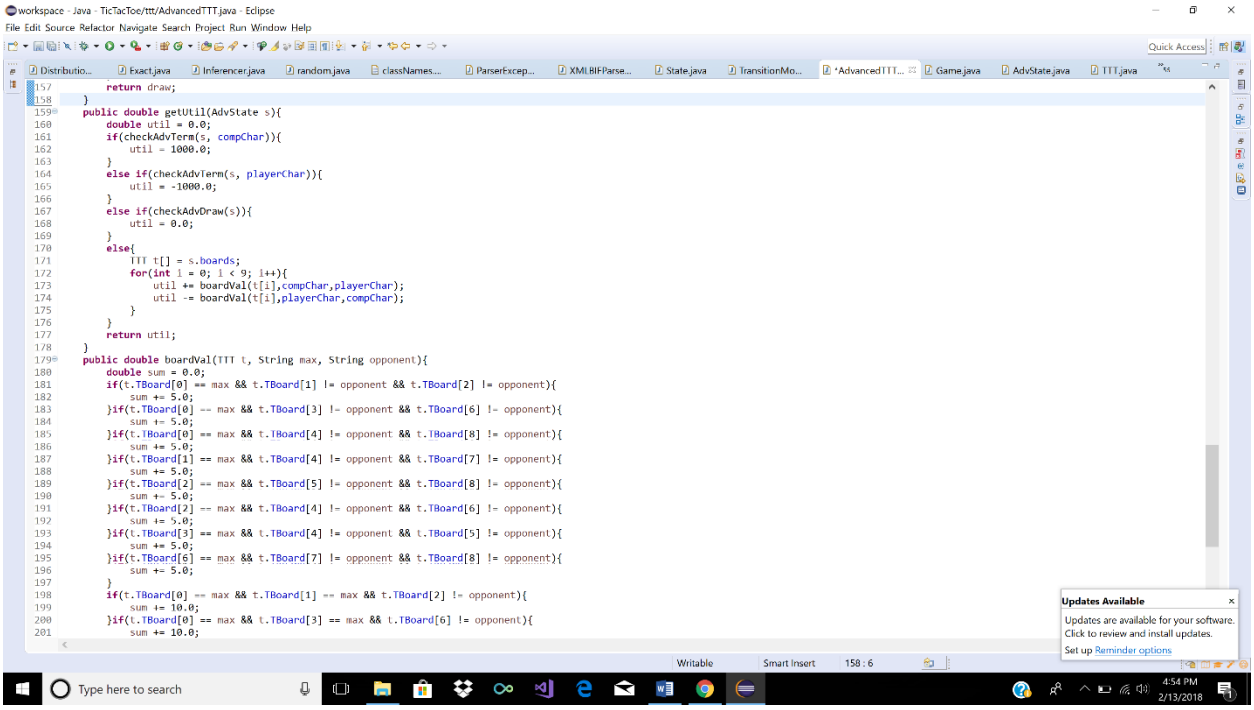
On top of the length 9 TTT[] I also had String player and int position. At first, I did not have int position in the code at first, but will explain a page later why I needed to add that in.

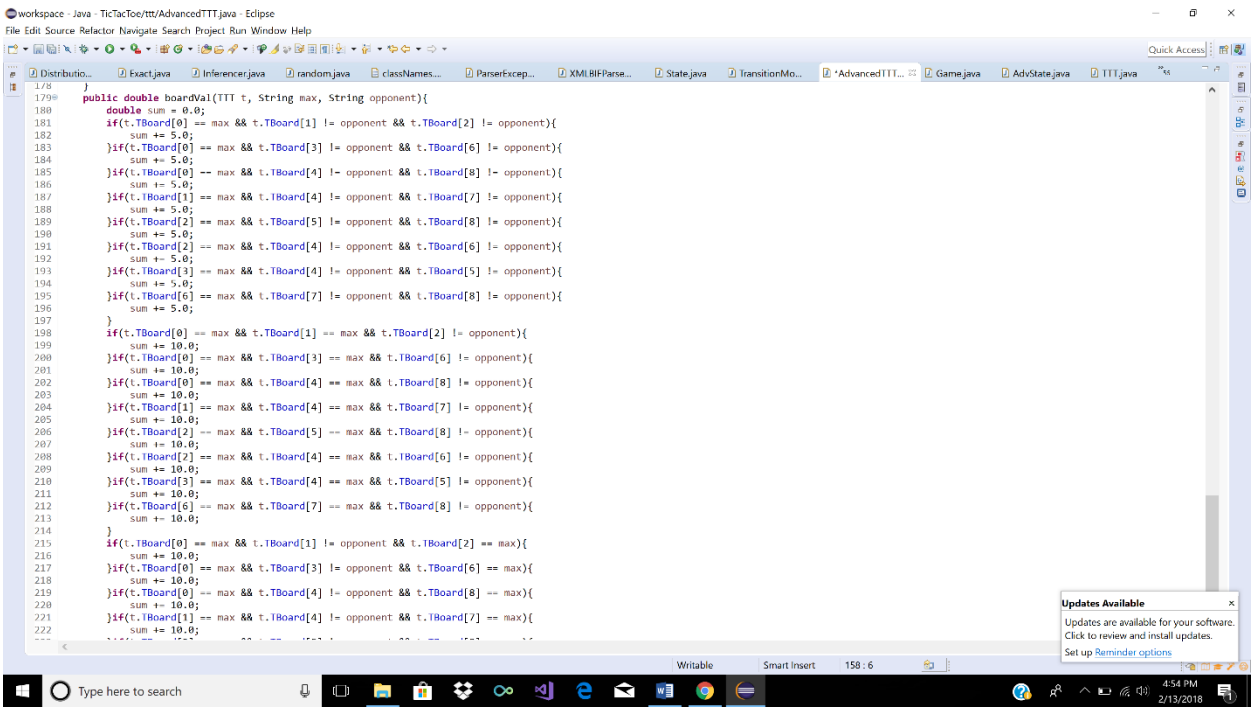
Once the representation was done, I needed to then implement a player move function, AI move function, and the Heuristic MiniMax functions. Following what I did in the basic Tic-tac-toe class, I essentially had the same function just stopping once the search hit a depth of four or hit a terminal state. This was fairly easy and after looking at the book, easily implemented alpha-beta pruning into the code as well. This is where my first problem came into play however. After the AI made its first move, it never made a move after when I made my second, third, fourth moves, etc. It was only doing one move the entire game at Board 0, position 0. I found the error inside my generteSucc function. If you see the parameter or the if function, I added in tempBoard[boardNum].Tboard[i] == null. Once I added that in, the problem was solved. The thing was, the generteSucc function was counting the current state itself as a successor thus continually choosing it. Keep in mind, at this point in the project I did not have my heuristic function down, so I was mainly just looking to see if the Computer was making moves at least. It was after solving this error.

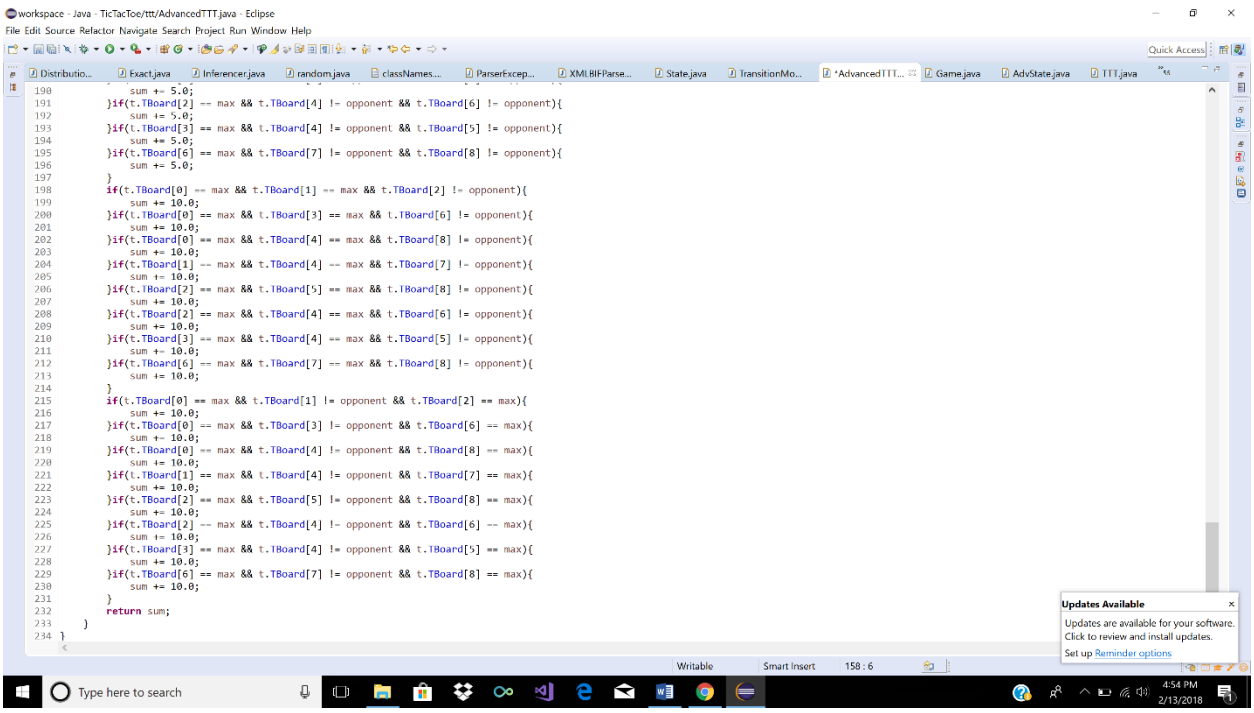
Afterwards, I made my heuristic function which would add up all the heuristics of every board. Each board heuristic would be as follows:

- Adding up all possible ways to win for the computer
    - Every way with two symbols already positioned there would add 10 to the sum of the boar heuristic, and every way with 1 symbol in that way thus far would add 5 to the sum. So, if there are 3 ways to win with two symbols in each way, the heuristic value of the board would be 30

- Once the value of a table for the computer was summed up, the program would then call the same function but instead value of the table for the player and subtract it from the value that was just summed up for the computer.
- If g(n) was the function and f(n) was the util of the board then the f(n) would be
    - g(CompChar) – g(playerChar) = f(n)
- a screen shot of both the function that takes each board's individual utility for a player and the function that computes the total utility of the state are shown below


workspace - Java - TicTacToe/ttt/AdvancedTTT.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Distributio... Exact.java Inferencer.java random.java classNames... ParserExcep... XMLBIFParse... State.java TransitionMo... *AdvancedTTT... Game.java AdvState.java TTT.java
return draw;
}
public double getUtil(AdvState s){
double util = 0.0;
if(checkAdvTerm(s, compChar)){
util = 1000.0;
}
else if(checkAdvTerm(s, playerChar)){
util = -1000.0;
}
else if(checkAdvDraw(s)){
util = 0.0;
}
else{
TTT t[] = s.boards;
for(int i = 0; i < 9; i++){
util += boardVal(t[i],compChar,playerChar);
util -= boardVal(t[i],playerChar,compChar);
}
}
return util;
}
public double boardVal(TTT t, String max, String opponent){
double sum = 0.0;
if(t.TBoard[0] == max && t.TBoard[1] != opponent && t.TBoard[2] != opponent){
sum += 5.0;
}if(t.TBoard[0] == max && t.TBoard[3] != opponent && t.TBoard[6] != opponent){
sum += 5.0;
}if(t.TBoard[0] == max && t.TBoard[4] != opponent && t.TBoard[8] != opponent){
sum += 5.0;
}if(t.TBoard[1] == max && t.TBoard[4] != opponent && t.TBoard[7] != opponent){
sum += 5.0;
}if(t.TBoard[2] == max && t.TBoard[5] != opponent && t.TBoard[8] != opponent){
sum += 5.0;
}if(t.TBoard[2] == max && t.TBoard[4] != opponent && t.TBoard[6] != opponent){
sum += 5.0;
}if(t.TBoard[3] == max && t.TBoard[4] != opponent && t.TBoard[5] != opponent){
sum += 5.0;
}if(t.TBoard[6] == max && t.TBoard[7] != opponent && t.TBoard[8] != opponent){
sum += 5.0;
}
if(t.TBoard[0] == max && t.TBoard[1] == max && t.TBoard[2] != opponent){
sum += 10.0;
}if(t.TBoard[0] == max && t.TBoard[3] == max && t.TBoard[6] != opponent){
sum += 10.0;
Updates Available
Updates are available for your software.
Click to review and install updates.
Set up Reminder options
Writable
Smart Insert
158 : 6
Type here to search
4:54 PM
2/13/2018


workspace - Java - TicTacToe/ttt/AdvancedTTT.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Distributio... Exact.java Inferencer.java random.java classNames... ParserExcep... XMLBIFParse... State.java TransitionMo... *AdvancedTTT... Game.java AdvState.java TTT.java
}
public double boardVal(TTT t, String max, String opponent){
double sum = 0.0;
if(t.TBoard[0] == max && t.TBoard[1] != opponent && t.TBoard[2] != opponent){
sum += 5.0;
}if(t.TBoard[0] == max && t.TBoard[3] != opponent && t.TBoard[6] != opponent){
sum += 5.0;
}if(t.TBoard[0] == max && t.TBoard[4] != opponent && t.TBoard[8] != opponent){
sum += 5.0;
}if(t.TBoard[1] == max && t.TBoard[4] != opponent && t.TBoard[7] != opponent){
sum += 5.0;
}if(t.TBoard[2] == max && t.TBoard[5] != opponent && t.TBoard[8] != opponent){
sum += 5.0;
}if(t.TBoard[2] == max && t.TBoard[4] != opponent && t.TBoard[6] != opponent){
sum += 5.0;
}if(t.TBoard[3] == max && t.TBoard[4] != opponent && t.TBoard[5] != opponent){
sum += 5.0;
}if(t.TBoard[6] == max && t.TBoard[7] != opponent && t.TBoard[8] != opponent){
sum += 5.0;
}
if(t.TBoard[0] == max && t.TBoard[1] == max && t.TBoard[2] != opponent){
sum += 10.0;
}if(t.TBoard[0] == max && t.TBoard[3] == max && t.TBoard[6] != opponent){
sum += 10.0;
}if(t.TBoard[0] == max && t.TBoard[4] == max && t.TBoard[8] != opponent){
sum += 10.0;
}if(t.TBoard[1] == max && t.TBoard[4] == max && t.TBoard[7] != opponent){
sum += 10.0;
}if(t.TBoard[2] == max && t.TBoard[5] == max && t.TBoard[8] != opponent){
sum += 10.0;
}if(t.TBoard[2] == max && t.TBoard[4] == max && t.TBoard[6] != opponent){
sum += 10.0;
}if(t.TBoard[3] == max && t.TBoard[4] == max && t.TBoard[5] != opponent){
sum += 10.0;
}if(t.TBoard[6] == max && t.TBoard[7] == max && t.TBoard[8] != opponent){
sum += 10.0;
}
if(t.TBoard[0] == max && t.TBoard[1] != opponent && t.TBoard[2] == max){
sum += 10.0;
}if(t.TBoard[0] == max && t.TBoard[3] != opponent && t.TBoard[6] == max){
sum += 10.0;
}if(t.TBoard[0] == max && t.TBoard[4] != opponent && t.TBoard[8] == max){
sum += 10.0;
}if(t.TBoard[1] == max && t.TBoard[4] != opponent && t.TBoard[7] == max){
sum += 10.0;
Updates Available
Updates are available for your software.
Click to review and install updates.
Set up Reminder options
Writable
Smart Insert
158 : 6
Type here to search
4:54 PM
2/13/2018



These heuristics were quite good. The computer never let me go to a board where I would be able to win, it always won when the possibility was given, and played above how I thought it would. Because of this I never tried any other heuristics just because this one was a pretty good one. I still have never beaten the computer after implementing this heuristic in.

After this was done I ran into a problem with the speed. The speed was terrible when I was running this at first. It took 5-10 minutes to complete the first move and I knew there was something wrong with the efficiency. I tried to make the depth limit at to 3, but this just dropped the time to 2-3 minutes. This problem was found when I realized you could only choose what board you wish to play on if you are the first to go. Thus, I just needed to save the board number in the state, which is why I created int position in AdvState class. This significantly boosted my efficiency because I was generating successors for all possible positions on all possible boards. That means it takes polynomial time or O(n^2) to generate successors, which takes quite a while. Once I changed it to only generate successors for the board provided the time went from polynomial to linear making a ton faster and the heuristic still worked just as well. I could then increase the depth limit to 4 and it honestly go even deeper. After this was done, I had completed both projects and fit in any specifics like validifying moves, checking if a board is full in Advanced TTT and filling in an system.err statements as needed.