Introduction to Programming - Project Part 2

K Ganer, C Sivanandam, H Flindt
Supervisor: Peter Schneider-Kamp
DM536

9. januar 2018

Indhold

1 Introduction to Tic-Tac-Toe

Tic-Tac-Toe is a simple board game traditionally played by 2 players on a 3 x 3 grid. However, our Tic-Tac-Toe game (TTT) must be implemented with towards 6 players and a 7 x 7 grid. The players alternate in placing a mark on one of unmarked fields. A player wins as soon as any of the rows, columns, or diagonals contain 3 of his or her marks in a row. The game is a draw, if no player wins and all 9-49 fields have been marked. The winning condition is the same, i.e., the first player to put 3 marks in a row, column, or diagonal wins.
Connect four is kind of an extension of TTT the few differences between them is that size of grid which is a standard 7 rows and 6 columns, however some variates includes 8 x 7, 9 x 7, 10 x 7 grid. Only two players and that you must have four in a row, column or diagonal. The main difference from TTT is that when a player click on a tile in the game, the mark is made in the lowest free position of that column. The last difference with respect to the user interface is that the board is rectangular.
When people play an ordinary TTT, what often happens is that almost every game will come out a tie. Both players can probably play perfectly without making a mistake that would allow the opponent to win. Probably, most people are not even aware of alternate possibilities, you just instinctively know where to move. However, it is a very different story when it comes to artificial intelligent. And yet the computer must know very explicitly and algorithm or strategy.
The strategy/Algorithm:
There are a lot of ways to implement strategy in TTT and here is one of them. The algorithm needs to know the highest and lowest priority in the rules.
The highest rules:
1. If the player can win on this move, do it.
2. If the other player can win on the next move, block that winning square.
3. If player can make a move that will set up a fork for myself, do it.
The lowest rules:
4. n-2 Take the center square if it's free.
5. n-1. Take a corner square if one is free.
6. n. Take whatever is available.
In these six rules of strategy there is a correlation between offense and defense. For example, the first rule is offense and second rule is defense. It is the same with rule three and four.
The second way is the Minimax algorithm that we implemented in our own artificial intelligence. The way it works is that each position has a value based on how good likely it was that it would lead to a win or at least a draw. If the next position is a winning move the position has a value worth 100. If the other position is a loss position it has a value of 79. The last two are if the positions are marked by AI is worth -10 and 10 if it is your mark.

2 Specification for Tic-Tac-Toe

2.1 Checkboundaries()

This method is supposed to return true when the current position is valid and false if it is invalid.

2.2 Shift()

Shift() should if functional return new x and y values that are shifted by the corresponding dx and dy values.

2.3 isFree()

To avoid laying two moves on the same square this method is to check if the current position on the board is free.

2.4 getPlayer()

A method that is to return the number of the player that has placed a marker on a given position on the board, and return 0 if no player has a marker on that position.

2.5 addMove()

addMove() adds the current players number to the position that was clicked on the board.

2.6 checkFull()

To check if the board is full, I.E. no valid move can be taken, checkFull() needs to return a true if the board is full and otherwise false.

2.7 checkWinning()

Method for checking if a player has won the game.

2.8 checkSequence()

checkSequence takes 3 arguments, a Coordinate, and 2 integers, and uses this to determine if three tiles in a row, column, or diagonal contain the same number aside from zero.

2.9 Artificial intelligence for Tic Tac Toe

A artificial intilligence (AI) opponent that is supposed to be an unbeatable , as it will always win or come to a draw.

2.10 App for Android

A mobil-version that is backwards compatible, to some extent, so to allow on the go gameplay.

2.11 Graphical user interface

Since a command line interface version of TTT might be quick to run for troubleshooting, a graphical user interface (GUI) would be implementet to make the game user friendly.

3 Design of Tic-Tac-Toe

Due to the size and scope of the assigment, each group-member tried to develop the different needed parts. The most interesting solutions are added to this report.

3.1 Checkboundaries()

It was designed to compare the current x and y values with the xSize and ySize values(these will be the same value in TTT as the board is quadratic), and if they are inside the range 0(inclusive)-size(exclusive) it is valid.

3.2 Shift()

The method was altered to also take a Coordinate as argument, this was to make sure that the correct coordinate is shifted.

3.3 isFree()

Check the Board at the given position and if the value is 0 return true, else return false.

3.4 getPlayer()

It returns the value of the board at the position it is passed.

3.5 addMove()

It takes a position and an integer as arguments and places the integer on the position on the board.

3.6 checkFull()

Use two for-loops to run though each position on the board and if it encounters a board position that a value of zero, the method returns false.

3.7 checkFull() by turnCounter()

Calls a function that increments a variable each time called. When the variable becomes larger than the amount of players squared, checkFull() returns a true boolean.

3.8 checkWinning()

Method that uses 2 for loops to generate all the start positions to be used by the checkSequence method to check all rows, columns and diagonals on the board.

3.9 checkWinning() by math

Since each player is represented by a number, it was thought to be mathematically possible to recognize a winning row, column, or diagonal on the board. This would be done directly in the checkWinning() without the need to call other functions. (i.e checkSequence())The class should have an intrinsic ability to check any board size from three and up in all directions possible for three in a row, and be to return an integer if needed to checkResult(), based on a float value without loss of data. (i.e only return the integer one if it is 1.0 and not 1.2 etc.)

3.10 checkSequence()

It was designed to check the board at the starting position and then shifting the position using the shift method, and then checking if this new position on the board contains the same value. it will do this twice to check a sequence of three if it finds three of the same number in a sequence then it returns that value, else it returns zero.

3.11 Artificial intelligence for Tic Tac Toe

The AI was planned to analyze the board and give each position values based on how likely it was that it would lead to a win or at least a draw. It was to do this by iterating through the board and for each position look at the row, column and diagonals, where applicable, and add values together and then choose the position that had the highest value. At first the idea was that recursion should be used to get the AI to look several moves ahead to but initial testing revealed that already while just looking at the next move there was only a few situations where it didn't guarantee at least a draw. It was only written to work for a two player 3x3 board but could be extended to work on both larger boards and more players.

3.12 App for Android

Make an app.

3.13 Graphical user interface

Due to hardware restraints a GUI version of the TTT was also implemented besides the app.

4 Implementation of Tic-Tac-Toe

4.1 Checkboundaries()

A single if statement that checks whether x and y are within the above range and returns the Boolean of the statement.

4.2 Shift()

The method takes the x and y values from the coordinate start and adds the dx and dy respectively to create a new XYCoordinate which it then returns.

4.3 isFree()

Implemented as designed.

4.4 getPlayer()

Implemented as designed.

4.5 addMove()

A try catch block was implemented to catch errors. The method tries to assign the current player at the board position it was passed. If an error occurs it checks whether it was caused by an invalid board position using checkBoundaries or if the playernumber is invalid using the size of the board(as the size of board is created by taking the number of players and adding 1). An extra error log is added in case neither of the above are the reason for the Illegal argument error. This is done in an attempt to fail loudly as to help with debugging.

4.6 checkFull()

Starts out with a boolean variable set equal to true, then using 2 for loops runs through each position on the board and checks if the any value at any position is equal to 0. If so, checkFull() returns false.

4.7 checkFull() by turnCounter()

Each time checkFull() is called it calls turnCounter() and an if-statment checks if the value returned from turnCounter is smaller that the amount of squares in the game. (i,e the amount of turns maximally allowed in the game.) Each time turnCounter() is called the turnCounter variable gets incremented(starting value is zero). If turnCounter() passes the if-statment check checkFull() returns true. Since the full size of the board is known in advanced at the beginning of a game, and each turn uses one square, there is a finite amount of turns available in any size of TTT. Therefore the need to check if all squares in the game are used is not relevant, but only a counter of amount of squares i.e. turns possible is needed. An minor advantage of a turnCounter() based checking system is that even though the size of the TTT-board is of a legible size, a checkFull() that should look at each square in the board. The runtime of checking the entire board if $\log(n^2)$ while using turncounter results in a runtime of log(n).

This is mostly a tour de force in problem finding that for this assignment might not be all that relevant, but non the less a theoretical board for 10.000 players would need checkFull() to go through one hundred million squares and either remember the state of each in a list, or do a full check sequence each time. The turnCounter() would in comparison just need to count to 10.000. A problem would arise with the turnCounter() if players were allowed to change the placement of their piece, but since this fundamentally changes the game no further thought were given to this.

4.8 checkWinning()

The method uses three different nested for-loops to iterate through all possible starting positions used for checkSequence(). First block checks rows and columns, here for the columns the start positions is not the 2 right most columns, as there aren not enough tiles for a full sequence of three squares. The same goes for rows but with the two bottom rows. The other two nested for loops create start positions for checking the diagonals. These are divided into two groups, one for diagonals going up and one for diagonals going down. Starting positions for the first group is a combination of the ones for rows and columns. The starting positions in neither the bottom two rows or the rightmost two columns. For the second group the top two rows and the rightmost two columns are not needed. The method then calls checkSequence() with the generated start positions and dx and dy corresponding to either rows(dx=1 ,dy=0), columns(dx= ,dy=1) etc. If the returned value from checkSequence() differes from zero it means that a player has won and checkWinning() returns the integer.

4.9 checkWinning() by math

The checkWinning() class was implemented as mathematical for-loop which would check each row, column, and diagonally up and down respectively. The main script of each for-loop is quite similar but with variation for the incrementing and direction of the check. Mathematically the checkWinning() consisted of checking if the cubic root of the product of three squares in a row, column, or one of the diagonals were equal to the integer of that root. If so it would return that integer value, which would correspond to the player number which had made the winning combination. If not it would return zero. The overall setup was with a for-loop with a nested for-loop. While the first for-loop selected the starting point the nested for-loop would move the interval being checked by one to a distance of two from the edge of the board. After this the first for loop would increment, or decrement and the nested loop would check the new row, column or diagonal for a winning sequence.

4.10 checkSequence()

checkSequence() takes three arguments, a Coordinate and two integers, dx and dy. The method creates an int variable, called checkValue, equal to the board at the coordinate, and if it is equal to zero, returns it. Then using a for loop to run the shift method twice with the given dx and dy arguments. After each loop it checks whether the board at this new position is equal to checkValue. If the are not equal the method returns zero and if they are equal the second loop occurs

and a new check happens. Again if they are not equal, zero is returned, but if they are equal it means three tiles in a sequence are the same and checkValue is returned, declaring the winner being the player whom is represented by that integer.

4.11 Artificial intelligence for Tic Tac Toe

The values used to analyze the board were to start with 100 for a tile that would lead to a win, 90 for a tile that would lead to a loss, and 10 and -10 for tiles containing the AI's mark and the players mark respectively. Later the lossValue was changed to 79 to avoid edge cases where the AI could win by placing a mark in one spot but choose to instead block the player in another spot. There are 4 methods in the AI.java, 4 checkmethods(), moveEvaluate(), GenerateMove(), and aiAddMove(). The four chec kmethods check each row, column, and if applicable one or both diagonals. These returns a value based on what each tile in that sequence contained, I.E. if it had two AI marks it would return 100 for the winning square. MoveEvaluate() simply adds the four check methods together and returns it. GenerateMove() runs, using two for-loops, through all nine possible positions on the board, and if that position is free(I.E. the board a that position contains a zero) calls moveEvaluate. The int returned here is assigned to the variable currentValue, which is then checked vs the variable bestValue(). If both have the same value a new XYCoordinate() is created using the current x and y values in the loop. This coordinate is then added to bestMoveList(). If currentValue is greater than bestValue bestMoveList() is cleared and then a new coordinate is added. This is done so that if several possible moves have the same value all are saved in the list and a the end a random move from the list is selected. aiAddMove() simply returns the generateMove(), and is called from the addMove() in TTTBoard. There are three exceptions to the standard rules that the AI follow. The first is that when the AI goes first it will always place a marker in the middle, as this is where there most possible paths that lead to a win for the AI. The second is if the human player starts and places a marker in the middle, then all other 8 positions will return a value of -10 and therefore currentValue will never be greater than or equal to bestValue. This results in no move being added to the bestMoveList and a NullPointerException. The solution to this exception was initially chose to set bestValue to -10 and call the generateMove method again, but this resulted in a possible loss if the AI chose to place it's marker in one of the middle tiles (i.e (1;2)(2;1)(2;3)(3,2). To avoide this scenario coordinates for the 4 corners were added to the bestMovelist if the AI was not starting. The last special scenarion was the case where the human player made a move in 1,1, the AI would the mark a corner. Then if the human player then marked the opposite corner the result would be that the AI would force a loss on itself with the standard values (unless the player made a mistake). The solution was to implement a check that would look for this case and then alter bestMoveList to contain 2 different Coordinates.

4.12 Special code added for the AI

An added button for starting game vs ai. It sets numplayers equal to 1 instead of taking value from the numberPicker. One button that makes the ai take it's move as well a different title depending on whether the AI or human starts. Added

a method to game.java and TTTgame.java called playerTurn(). This method is used for controlling when the AI move and the board can be pressed to make sure a button only responds when it should. In TTTGame.java the TTTgame constructor have been changed to randomize starting player when playing vs the ai(could also be extended to the normal gamemode if wished. Will become part of the third DLC for the game.) and addmove() have been altered to handle playing vs the AI. I.E. if player one took his turn it sets currentplayer = seven aka the AI, and vice versa. In TTTBoard.java the TTTBoard contructor was also changed to create a 3x3 board when playing vs AI. addMove() now, if currentplayer is seven, calls aiAddMove() to find the pos at which to add a marker.

4.13 App for Android

Few adjustments as to fit and Android mobilephone was added, but few relevant things.

4.14 Graphical user interface

While a functional version was given as part of the assignment, several cosmetic changes were needed to ensure a more user friendly experience. While an earlier version of the GUI was expanded with background music and sounds based upon actions this was removed again due to compiling errors when sharing the game, cause of the specify needed of the file-path, combined with a minor latency issues when loading each sound file.

5 Testing Tic-Tac-Toe

Generative testing is single tactic that can be actual for tackling testing situations with a huge series of potential inputs (like all the possible game states for a tic tac toe board). Our method to solving this problem was to check all checksequence scenario for a tic tac toe game (starting with an empty board) and confirm that the human player never won. Simulating the computer player's moves is possibly the easiest problem to confront, theoretically, it is the move the computer makes with the given specific board. (For each board, the computer player will make only one move and therefore generate 1 possible game state). When the human player's moves, it happens to be an order to ensure that we are testing all possible game scenarios, we need to make a function that given a board will return a collection of boards, on each of which the human player has played on a different space. (For a board with n empty spaces, the simulated human player will create n possible game possibilities).

5.1 checkFull() and the GUI

A complication arose when implementing the GUI, as the GUI did not recognize when a draw was reached, as should be the case with the checkFull() and turnCounter(). By changing different variables methodically backwards from this first call of checkWinning(), to checkFull() and finally turnCounter() the error was located. It was discovered in the way the GUI generated the board compared to

the CLI. The CLI generates the board and calls checkWinning() after player one turn incrementing turnCounter() till a certain value. If the checkResult() has returned a winner by that point, the else-if-statement comes true thus returning the draw statement. The problem appears to be that the turnCounter() was not incrementing when the GUI was launched. Thus an extra call to turnCounter() was launched when the GUI first generates the user interface. This ensured that a draw would be reached for the GUI.

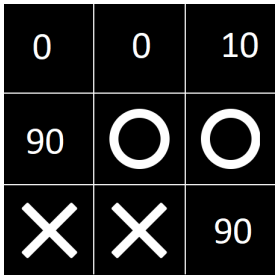
5.2 checkWinning() by math

A special case was discovered that falsely would result in a win for player 2. It arose when three squares were occupied by player one, two and four. These three would result the same number when checkWining were run as if all three squares were occupied by play two. This was resolved by adding two to all terms in the cubicroot equation. No result for 2-6 players could then be equal in any combinations. Subsequent subtraction of two after taking the cubicroot would return the checkWinning() result needed.

5.3 AI

First issue was trying to get a board passed from TTTGame to use for evaluating the best move for the ai, but board in TTTGame isn't a int[][][] so from TTTGame i couldnt pass the board. Instead had the addMove method in TTTBoard call the aiAddMove method and pass the current board along. in the early version we wanted to populate an aiBoard to be used during recursion to check more than 1 move ahead, but this was discovered to be unneeded as just the normal rules with a few exceptions results in a win or draw in all tested cases when only the current board is considered.

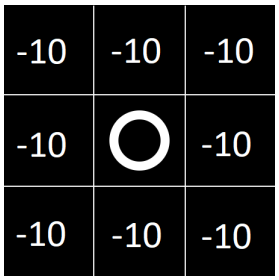
Value bug: having a lossValue of 80 resulted in a few cases where a winning move and a blocking move had the same value(winning move = 100 − 10 for a oppmarker, and blocking move being 80+10 for own marker). Fixed by reducing lossValue to 79



Figur 1:

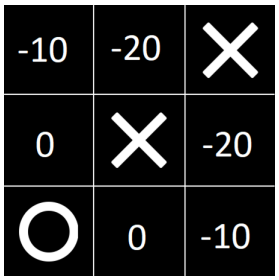
Human start 1,1: if the human player starts and places a token in 1,1 I.E. the middle. All other free positions where evaluated to -10 and as bestValue starts of at 0, none of the free positions returned a higher value than 0 a bestMove was never set. Result= nullpointer on getX() in addMove(); Fix: only in this case

would the bestMoveList be empty so we made an if statement that checked for this and when true would populate the list with positions corresponding to the 4 corners.



Figur 2:

The last exception is in the case in the case in the picture below where the highest value positions leads to a loss for the AI. This is fixed by intercepting the bestMOveList when it contains these 2 positions(there are 4 total cases that lead to the same result just mirrored) and then replacing with two new coordinates(the two free corners). This will at least guarantee a draw.



Figur 3:

6 Conclusion

Since many roads may lead to Rome, so is the way three different programmers may resolve a assignment. Differed implementations of the tic-tac-toe and connect-4 was developed and tested with satisfiable results. Two overall different approaches to checking for a win and a draw was developed, one based on logical checks and one by math. While the one by math proved easier to program and showed a advantage in some areas. Ensuring that the math it self was correct and yielding the desired result, required a bit of mental gymnastics. The logical checks needed more programming, and a theoretical disadvantage for checking a draw was also discovered. (i.e. in relation to an increase in memory usage.) After a few group talks is was thought that a more optimal version of both check winning, for both math and logical check, could be written. While the present programs have the computer running through each permutation of three squares to check for a win, a different approach with the computer only checking relevant squares next to the last square selected. Thus instead of checking the

entire square it would only check a minimum of 3 squares and a maximum of 13 for similarity. (Se appendix) A few other minor bugs were discovered and solution found, as a result of going through the design and values assigned to different parts of the program.

7 Appendix

Designing a quicker checkWinning() with a maximum of 13 checks, could most likely be done in accordance with the table shown belowe.

N/A		N/A		N/A
	7	8	9	
N/A	6	1	2	10
	5	4	3	
13		12		11

In a any board of 4+ size it should register the players input in square nr. 1. After this it would check square nr. 2. If square 1 == square 2 it should check square 10 and square 6. If any three would be the same this would result in a win. If this is not the case, or if square 1 != square 2 the it should check square 3 and so on for all 2 too 9 square in the center. It does not need to check those marked with N/A since if square 1 == square 7 that would be the same as a check from square 1 == 3 and then square 11. Therefore it can be said that it can be transposed and a maximum of 13 checks would be needed.