Data Book and Version Change Log

2015-01-06 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 1.0 (Gameplay Framework)**

* Created implementation class (GameBoard) and built framework for game inside main method
* Includes calling methods for both AI and human player input compatibility, checking for terminal states, and switching player turn
* Initialized instance variables (rows at 6, columns at 7, whether or not player is human, AI iteration count)

**Figure 1: Version 1.0 Code Sample**

**boolean** gameLoop = **true**;

**char**[][] gameBoard = **new** **char** [*rows*][*columns*];

**while**(gameLoop)

{

*buildBoard*(gameBoard, *turn*);

System.***out***.println("\n" + "Player " + *turn* + "'s Turn");

**if** (*turn* == 'O' && *humanPlayer1*) //Human player 1

{

*playTurn*(*turn*, gameBoard);

}

**else** **if** (*turn* == 'X' && *humanPlayer2*) //Human player 2

{

*playTurn*(*turn*, gameBoard);

}

**else** **if** (*turn* == 'X' && !*humanPlayer2*) //MCTS AI initialization for player 1

{

*playAI*(gameBoard);

}

**else** **if** (*turn* == 'O' && !*humanPlayer1*) //MCTS AI initialization for player 2

{

*playAI*(gameBoard);

}

**if** (*checkWin*(gameBoard, *turn*))

{

*buildBoard*(gameBoard, *turn*);

System.***out***.println("\nPlayer "+ *turn* +" wins!");

**break**;

}

**else** **if** (*checkTie*(gameBoard))

{

*buildBoard*(gameBoard, *turn*);

System.***out***.println("It's a tie!");

**break**;

}

**if** (*turn* == 'O'){*turn* = 'X';} //changes turn

**else**{*turn*='O';}

}

2015-01-10 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 1.1**

* Built game board mechanisms
  + buildBoard method 🡪 prints out 2D array in viewable board
  + playTurn method 🡪 simulates a dropped piece (natural game gravity)

**Figure 2: Version 1.1 Code Samples**

(Sample from playTurn method)

piece--; //adjust for index starting at 0

//drop piece in first available row within column

**for**(**int** i = (*rows*-1); i >= 0; i--)

{

**if** (boardState[i][piece]=='-')

{

boardState[i][piece]= color;

**break**;

}

}

**public** **static** **void** buildBoard(**char**[][] boardState, **char** player){

System.***out***.println("");

**for**(**int** i = 0; i < *rows*; i++){

**for**(**int** j = 0; j < *columns*; j++){

**if** (j == 0){System.***out***.print("| ");}

**if** (boardState[i][j] == '\0'){boardState[i][j] = '-';}

//fills empty spaces with '-'

System.***out***.print(boardState[i][j] + " | ");

}

System.***out***.println("\n-----------------------------");

}

}

**Figure 3: Sample Board Build**

| - | - | - | - | O | - | - |

-----------------------------

| - | - | - | O | O | - | - |

-----------------------------

| - | - | O | O | X | - | - |

-----------------------------

| - | O | X | X | X | - | O |

-----------------------------

| X | X | O | O | O | - | X |

-----------------------------

| O | X | X | O | X | - | X |

-----------------------------

2015-01-15 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 1.2**

* Added out of bounds and invalid input catches for user input in playTurn method
* Debugging
  + Fixed if statement with incorrect inequality in playTurn
* Added terminal board state checking accessor methods (checkWin, checkTie)
  + checkWin 🡪 uses loops to update counter for 4 in a rows
  + checkTie 🡪 checks if checkWin is false and board is full

**Figure 4: checkWin Code Sample**

**public** **static** **boolean** checkWin (**char**[][] boardState, **char** winner){

//check horizontal win

**for** (**int** i = 0; i < *rows*; i++){

**int** horizontalCounter = 0;

**for** (**int** j = 0; j < *columns*; j++){

**if** (boardState[i][j] == winner){

horizontalCounter++; //add 1 for every time the color in question comes up

}

**else**{horizontalCounter = 0;} //resets counter if same two pieces are not in a row

**if** (horizontalCounter == 4){

**return** **true**;

}

}

}

//check vertical win

**for** (**int** i = 0; i < *columns*; i++){

**int** vertCounter = 0;

**for** (**int** j = 0; j < *rows*; j++){

**if** (boardState[j][i] == winner){

vertCounter++; //add 1 for every time the color in question comes up

}

**else**{vertCounter = 0;} //resets counter if same two pieces are not in a row

**if** (vertCounter == 4){

**return** **true**;

}

}

}

**for** (**int** i = 0; i < *columns*; i++){

**for** (**int** j = 0; j < *rows*; j++){

//check diagonals down and to the right (negative slope)

**if** (j <= *rows* - 4 && i <= *columns* - 4) //starts 4 up from the bottom and 4 away from the right to prevent OutOfIndex

{

**if** (boardState[j][i]==winner

&& boardState[j+1][i+1]==winner

&& boardState[j+2][i+2]==winner

&& boardState[j+3][i+3]==winner){

**return** **true**;

}

}

//check diagonals down and to the left (positive slope)

**if** (j >= 3 && i <= 3) //starts 4 down from the top and 4 away from the left to prevent OutOfIndex

{

**if** (boardState[j][i]==winner

&& boardState[j-1][i+1]==winner

&& boardState[j-2][i+2]==winner

&& boardState[j-3][i+3]==winner){

**return** **true**;

}

}

}

}

**return** **false**;

}

**Figure 5: playTurn catch loop**

**while**(**true**)

{

**try** {

piece = *play*.nextInt(); //asks user to input a column to drop it in

//if input is within range of columns and there is an open space in that column, continue to drop piece

**if** ((piece > 0 && piece <= *columns*) && (boardState[0][piece-1]=='-')){**break**;}

//check for full columns

**else** **if** ((piece > 0 && piece <= *columns*) && (boardState[0][piece-1]=='X' || boardState[0][piece-1]=='O')){

System.***out***.println();

System.***out***.println("Sorry, this column is full!");

System.***out***.println();

System.***out***.print("In which column would you like to drop your piece?");

}

//make sure input is within valid range of columns

**else** **if**(piece < 1 || piece > *columns*){

System.***out***.println();

System.***out***.println("Please input a number from 1 to " + *columns* + "!");

System.***out***.println();

System.***out***.print("In which column would you like to drop your piece?");

}

}

//makes sure input is a valid int

**catch** (InputMismatchException e){

System.***out***.println();

System.***out***.println("Please input a valid integer!");

System.***out***.println();

System.***out***.print("In which column would you like to drop your piece?");

*play*.next();

}

}

2015-01-22 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 1.3**

* Begin AI development (keeping playAI method empty for now)
* Created node class
  + Instance fields 🡪 node gameState (gameBoard array) and node stats (visits, wins, etc)
  + Created (2) node constructors 🡪 both take gameState, one takes boolean for opponent moves
* Created MCTS method for framework of running AI
  + Selection, Expansion, Simulation, Backpropagation fields
  + For backpropagation, added each node to list of visited and chose to use default Win (1), Tie (.5), Loss (0) values 🡪 WILL TEST LATER
  + Put in firstIteration boolean to preserve root node / checked for terminal gameStates in while loops

**Figure 6: MCTS Method Framework**

**public** **void** monteCarloTreeSearch()

{

List<Node> visitedNodes = **new** ArrayList<Node>();

Node currentNode = **this**;

visitedNodes.add(**this**);

**boolean** firstIteration = **true**;

//Selection: find next optimal node to explore, given that the current node has children to select from

//first iteration has no children, thus this loop will be skipped for the first iteration each time

**while** (currentNode.hasChildren()

&& !GameBoard.*checkWin*(currentNode.boardState, 'O')

&& !GameBoard.*checkWin*(currentNode.boardState, 'X')

&& !GameBoard.*checkTie*(currentNode.boardState))

{

firstIteration = **false**; //first iteration cannot have children (as it has not reached expansion yet) thus this must become false in this loop

currentNode = currentNode.select(); //selects optimal node based on UCB and follows it

**if** (!currentNode.oppMove) {visitedNodes.add(currentNode);}

}

//Simulation/Expansion: simulate (random) game until there is a winner or there is a tie.

**while** (!GameBoard.*checkWin*(currentNode.boardState, 'O')

&& !GameBoard.*checkWin*(currentNode.boardState, 'X')

&& !GameBoard.*checkTie*(currentNode.boardState))

{

currentNode.expansion(currentNode.boardState, currentNode.children, firstIteration); //Expands a given node to create child nodes for the current players turn

firstIteration = **false**; //after any node has expanded, it is impossible for it to be the root node

currentNode.oppExpand(currentNode.boardState, currentNode.children); //Simulates all possible random opponent moves as children of players possible moves

currentNode = currentNode.select(); //selects a player child node to continue simulation (randomly since none of these nodes have been visited yet as they are newly made)

**if** (!currentNode.oppMove) {visitedNodes.add(currentNode);}

**if** (!GameBoard.*checkWin*(currentNode.boardState, 'O')

&& !GameBoard.*checkWin*(currentNode.boardState, 'X')

&& !GameBoard.*checkTie*(currentNode.boardState))

{

currentNode = currentNode.select(); //selects opponent child of player children randomly

}

}

//set opponents character based on current player character

**if** (GameBoard.*turn* == 'X'){*opp* = 'O';}

**else** **if** (GameBoard.*turn* == 'O'){*opp* = 'X';}

**if** (GameBoard.*checkWin*(currentNode.boardState, GameBoard.*turn*))

{

//backpropogate and update stats for positive result (win game)

**for** (Node visitedNode : visitedNodes)

{

//GameBoard.buildBoard(visitedNode.boardState, GameBoard.turn);

visitedNode.nodeVisits++;

visitedNode.nodeWins+=1; //adds 1 for a win

}

}

**else** **if** (GameBoard.*checkWin*(currentNode.boardState, *opp*))

{

//backpropogate and update stats for negative result (lose game)

**for** (Node visitedNode : visitedNodes)

{

visitedNode.nodeVisits++;

//visitedNode.nodeWins-=0; //adds nothing for a loss

}

}

**else**

{

**for** (Node visitedNode : visitedNodes)

{

//backpropogate and update stats for a neutral result (tie game)

visitedNode.nodeVisits++;

visitedNode.nodeWins+=0.5; //adds 0.5 for a tie

}

}

}

2015-02-02 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 1.4**

* Created expansion method to find available node children and create them
  + Create a similar oppExand method to do this alternatively with expansion to simulate opponent moves (randomly) as well (thus implemented *simulation*)
* Fixed ordering of expansion and oppExpand methods in MCTS method framework (minor change)

**Figure 7: Expansion Method Sample**

**public** **void** expansion(**char**[][] boardState, List<Node> children, **boolean** firstIteration)

{

**for** (**int** i = 0; i < GameBoard.*columns*; i++)

{

**for** (**int** j = GameBoard.*rows* - 1; j >= 0; j--)

{

**if** (boardState[j][i] == '-') //finds lower-most (if any) blanks spaces in each column to simulate opponent move

{

**char**[][] tempBoard = **new** **char** [GameBoard.*rows*][GameBoard.*columns*];

boardClone(tempBoard, boardState);

tempBoard[j][i] = GameBoard.*turn*;

children.add(**new** Node(tempBoard));

//if (counter < 100) {GameBoard.buildBoard(tempBoard, GameBoard.turn);}

*counter*++;

**break**;

}

**else** **if** (boardState[j][i] != '-' && j == 0 && firstIteration)

{

children.add(**null**); //preserves indexes of children (in case one column fills up and is skipped)

//only necessary for firstIteration because the index is used to play move

}

}

}

}

2015-02-12 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 1.5**

* Created select method to implement bandit policy
  + Iterates through all the children of an implicit node and returns node with highest UCT value (selected)
* **𝑋j=𝑎𝑣𝑒𝑟𝑎𝑔𝑒 𝑟𝑒𝑤𝑎𝑟𝑑 𝑓𝑟𝑜𝑚 𝑎𝑟𝑚 𝑗 (𝑣𝑎𝑙𝑢𝑒÷𝑡𝑜𝑡𝑎𝑙 𝑣𝑖𝑠𝑖𝑡𝑠)**
* **𝐶𝑝=𝑡𝑟𝑒𝑒 𝑒𝑥𝑝𝑙𝑜𝑟𝑎𝑡𝑖𝑜𝑛 𝑐𝑜𝑛𝑠𝑡𝑎𝑛𝑡 (𝑔𝑒𝑛𝑒𝑟𝑎𝑙𝑙𝑦 1/√2)**
* **𝑛=number of visits to the parent node**
* **𝑛𝑗=number of visits to child node**

**Figure 8: Select Method**

**private** Node select() //Upper Confidence Bound (Bandit Policy) implemented to select most optimal node based on exploration-exploitation balance

{

Node selected = **null**;

**double** bestUCB = 0;

**for** (Node childNode : children)

{

**if** (childNode == **null**){**continue**;} //for the special case of firstIteration children

// UCB = estimated value(average reward) + C \* sqrt(ln(number of visits to parent) / (number of times (child) node has been visited))

// Where C is some constant -- sqrt(2) in this case

// buffer prevents divide by 0 errors in the case of unvisited nodes

**double** ucbValue = childNode.nodeWins / (childNode.nodeVisits + *buffer*) //estimated value

+ (2/Math.*sqrt*(2)) \* Math.*sqrt*(2 \* Math.*log*(nodeVisits+1) / (childNode.nodeVisits + *buffer*)) +

*randomNumb*.nextDouble() \* *buffer*; //generates a very small random number to break ties between similar/unvisited nodes

**if** (ucbValue > bestUCB)

{

selected = childNode;

bestUCB = ucbValue;

}

}

**return** selected;

}

2015-02-12 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 2.0 (Begin Testing)**

* Tested for backpropagation values
  + Edited program to have two AIs play against one another at 1000 iterations
  + Tested different loss, tie, win values for balanced, offensive, and defensive playstyles
  + Balanced (0/.5/1)
  + Offensive (0/0/1)
  + Defensive (0/1/1) – believe to be best due to high equal success against all other types and great success against offensive

**Table 1: Testing Backpropagation Playstyles**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | B vs B | B vs D | B vs O | D vs D | D vs O | O vs O |
| Player 1 Wins | 413 | 421 | 450 | 220 | 567 | 501 |
| Ties | 165 | 201 | 92 | 549 | 124 | 22 |
| Player 2 Wins | 422 | 378 | 458 | 231 | 309 | 477 |
| Totals | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |

**Graph 1: Testing Backpropagation Playstyles Graph**

2015-02-15 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 2.1**

* Cleaned up some unneeded/unused methods
* Added more comments for extended readability
* Adjusted while GameBoard while loop for continuous games (for future testing)

2015-02-20 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 2.2**

* Tested MCTS AI vs AI playouts for different playout iterations (see below)
  + Found that increased playout iterations only increased success marginally (hypothesized arctan or logarithmic model)
* Automated results with BufferedReader Output

**Table 2: MCTS AI vs AI Turnouts (1000 Trials for Each Matchup)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 5000 vs 5000 | 5000 vs 1000 | 5000 vs 500 | 1000 vs 1000 | 1000 vs 500 | 500 vs 500 |
| Player 1 Wins | 355 | 573 | 798 | 353 | 567 | 411 |
| Ties | 220 | 51 | 54 | 280 | 124 | 222 |
| Player 2 Wins | 425 | 376 | 148 | 367 | 309 | 367 |
| Totals | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |

**Graph 2: MCTS AI vs AI Turnouts (1000 Trials for Each Matchup) [Graph]**

2015-02-22 Somil Govani [somilgovani@gmail.com](mailto:somilgovani@gmail.com)

**Version 2.3**

* Separated testing features into another program (kept it as 2.2)
* Added additional comments
* Tested time elapsed for initial move and noticed a linear relationship with playout iterations (see below)
  + Approx y = x/1000, where y is # of seconds elapsed for first move and x is number of playout iterations
  + Used this LSR to extrapolate Minimax values (see paper)
* Tested UCT against E-greedy policy – No contest (see below)

**Table 3: UCT Selection Policy vs Egreedy Selection Policy (at 5000 iterations)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1000 vs n = 0.1 | 1000 vs n = 0.2 | 500 vs n = 0.1 | 500 vs n = 0.2 | 5000 vs n = 0.1 | 5000 vs n = 0.2 |
| MCTS Wins | 1000 | 997 | 978 | 953 | 1000 | 1000 |
| Ties | 0 | 3 | 22 | 45 | 0 | 0 |
| E-Greedy Wins | 0 | 0 | 0 | 2 | 0 | 0 |
| Totals | 1000 | 1000 | 1000 | 1000 | 1000 | 1000 |

**Table 4: Time Elapsed for Initial Move with Varying Playout Iterations (1000 trials)**

|  |  |
| --- | --- |
| Playout Iterations | Time Elapsed for Initial Move (Average of 1000) |
| 500 | 0.306 |
| 1000 | 1.014 |
| 2000 | 1.974 |
| 3000 | 2.591 |
| 4000 | 4.1 |
| 5000 | 5.24 |

**Graph 3: Time Elapsed for Initial Move with Varying Playout Iterations (1000 trials) [Table 4] (w/ LSR Model)**