Worcester Polytechnic Institute
Computer Science Program

ULTIMATE TIC-TAC-TOE AI REPORT

CS4341 - Introduction To Artificial Intelligence

Submitted By Team Walrus
Theo Coppola
Brianna Sahagian
Leona (Nhi) Nguyen

Date Submitted: October11, 2022
Date Completed: October 11, 2022
Course Instructor: Prof. Ruiz
Class Section: CS 4341 - A Term

Project Class Breakdown

The **Gamer** class was created to handle any logistics relating to the way Ultimate Tic-Tac-Toe will be played with the referee program. This class determines when the Walrus AI can play and when the AI should stop playing. This class is also responsible for collecting information about the last move that will be used in our search algorithm minimax. A list of Gamer tasks include:

- Read the first four moves from the *first_four_moves.txt* file and store these moves on an internal gameboard in Walrus AI.
- Wait for a Walrus AI turn before proceeding by checking the existence of a *Walrus.go* file in the referee directory.
- Find a valid board to move in depending on the current state of the game:
  - If Walrus AI must take the first move of the game, set the *board* variable to the *spot* of the last move in *first_four_moves.txt;*
  - If Walrus AI takes a move that is not the first in the game, set the *board* variable to the *spot* of the move present in *move_file.txt* after ensuring that this file exists in the referee directory;
  - If Walrus AI cannot move in the current board because it is full, set the next *board* variable based off of the output from minimax.
- Check the validity of any given move using the legalMove(*board, spot*) function.
- Ensure that Walrus AI only moves when the game is not won, lost, or tied by checking the isGameWon() and bigBoardIsFull() functions before every turn.
  - Ensures the validity of the opponent's last move by using legalMove(*opponentBoard, opponentSpot*) before deciding to continue the game.

The **Node** class was created to store the information of each potential board configuration and any child configurations that resulted at the next depth (*depth* + 1) in minimax. The Node is used as the structure which forms the search tree for minimax. Information stored in a Node includes:

- *player:* The player that the Node (board configuration) corresponds to and the player that the configuration will be evaluated for. This is an int value that is either equal to 1 or 2; 1 represents the Walrus AI (aka Max) and 2 represents the opponent (aka Min).
- *cBoard:* The int value of the board that this Node belongs to; each Node in a single search tree will have the same cBoard number.
- *spot:* The int value of the spot that corresponds to the board configuration in this Node. This Node contains the configuration which results after a move at *cBoard, spot* is placed.
- *boardConfig:* The int[9] array that models the board configuration which results after a move at *cBoard, spot* is placed. Each index in the array may

contain a 0 (spot not taken), a 1 (spot taken by Walrus AI) or a 2 (spot taken by opponent AI).

- *children:* The List<Node> of other board configurations that result from taking each of the remaining moves on the *boardConfig* stored in the current Node.

The **Strategy** class was created to determine the best possible move for Walrus AI from the current board configuration. This class houses three important functions including utility(), evaluate(), and minimax().

- The utility(*player, board*) function takes the int value corresponding to the player that it checks the win for: 1 for Walrus AI (aka Max) or 2 for opponent AI (aka Min) and the int[9] array corresponding to the current board configuration. This function returns 3 distinct cases:
  - If *player* has won the *board*, utility() returns 1;
  - If neither player has won the board, utility returns 0;
  - Else (if the opponent has won the board), utility returns -1.
- The evaluate(*player, board*) function assigns a value score to any given intermediate board state. This function takes the same parameters as utility(). Everytime evaluate() is called, it employs the following calculation for each row, column, and diagonal:
  - If all 3 spots are full but do not contain the marks of solely one player do not adjust the heuristic;
  - If *player* has three marks, adjust heuristic +500;
  - If opponent has three marks, adjust heuristic -500;
  - If *player* has two marks and opponent has no marks, adjust heuristic +50;
  - If opponent has two marks and *player* has no marks, adjust heuristic -50;
  - If *player* has one mark and opponent has zero marks, adjust heuristic +5;
  - If opponent has one mark and *player* has zero marks, adjust heuristic -5.
- The minimax(*n, isMax, alpha, beta, depth*) function employs the capabilities of evaluate() on a recursive level. This function takes in the current Node in the search tree (see Node class above for description), the boolean state to start evaluation in (Max or Min), the int maximum value that can be extracted from the search tree, the int minimum value that can be extracted from the search tree, and the int depth of the tree to explore (where depth is equal to number of new moves simulated on one board). With this information, minimax() executes through these steps:
  1.) If the search tree is at the depth limit (at depth 0), run evaluate(1, n.boardConfig) to evaluate the value of the Node's board configuration for Walrus AI.
  2.) If the search tree is not at the depth limit:

a.) Determine which recursive side of the function to run by checking the value of *isMax*. If *isMax* is true, run the maximum evaluation, otherwise, run the minimum evaluation.
b.) Expand the search tree one level down and populate this level with Nodes that belong to player 1 if *isMax* or player 2 otherwise.
c.) Check to see for a child configuration of *n* whether there is a winning terminal state using utility() function. If so, return the move corresponding to this board configuration and the configuration hValue.
d.) Evaluate the hValue for this child configuration of *n* using evaluate(). If the result of evaluate is > alpha and the minimax() function is on the maximum side, set alpha to the hValue. If the result of evaluate is < beta and the minimax() function is on the minimum side, set beta to the hValue. Regardless of the side of minimax(), if either of these cases occurs, set the *bestSpot* variable to the move from this Node.
e.) Recurse through the function, switch the value of *isMax*, traverse one level deeper in the search tree (*depth* - 1) and repeat from step 1 until all children have been searched (within the depth limit).
f.) Return the int[2] array *bestMove* which contains the {*bestSpot*, *bestHVal*}.

The **Stopwatch** class was created to contain a timer which allows the iterative deepening minimax search to run within the time constraints of Ultimate Tic-Tac-Toe. A Stopwatch object created from this class utilizes *System.currentTimeMillis()* to keep track of the elapsed time from its construction. Once the elapsed time reaches 7 seconds, minimax must return its current output move suggestion and this move will be played by Walrus AI. This 7 second cutoff may place a limit on how deep the minimax function can search, but this timing tactic allows the Walrus AI to remain within the time constraints of the game. Walrus AI uses 7 seconds as a cutoff to avoid getting too close to the in-game limit of 10 seconds.

The **Main** class was created to run a functional version of the Walrus AI at the same time as the referee program. This class runs a while(true){} loop that allows Walrus AI to play Ultimate Tic-Tac-Toe until an invalid opponent move, win, loss, or tie causes the loop to *break*.

Walrus AI uses a combination of the Stopwatch 7 second cutoff and iterative deepening minimax() to expand nodes of the minimax tree without exceeding the game time limit. The *depth* parameter of minimax() makes this function implementable as a depth iterative function, allowing the AI to increment the depth limit on minimax() and run the function again until the cutoff time is reached.

Team Member Contributions

Theo Coppola - Worked primarily on heuristic evaluate() function in Strategy class, Stopwatch implementation, minimax functionality 2.0, and the functions that allow the Walrus AI to work with the referee in Gamer class (see above section).

Leona Nguyen - Worked primarily on minimax functionality 2.0 including revisions and iterative deepening mechanisms, and the functions that allow the Walrus AI to work with the referee in Gamer class (see above section).

Brianna Sahagian - Worked primarily with Minimax framework 1.0, the Node class that populates the potential move tree, and the functions that allow the Walrus AI to work with the referee in Gamer class (see above section).

We all contributed equally to the project but just focused on different areas during the composition.

Results

The program was tested with isolated test functions for the minimax() and evaluate() strategies. These tests are found in the StrategyTests class. Walrus AI also played against its creators in a human vs. AI match of Ultimate Tic-Tac-Toe. Walrus AI was able to win and block opponent wins on the majority of its boards and was even able to set up a double trap in one match. However, the Walrus AI faced a few specific board configurations where it would not pick the move that we believed would be heuristically sound (blocking an opponent's win or taking the win itself). We have updated minimax strategy since and plan to continue improving it so the Walrus AI will find the correct move in these cases.

Justification for Heuristic Function (evaluate(***player***, ***board***))

A heuristic value is an accumulation of the success probabilities of each row, column, and diagonal in a board. As the difference between the amount of placements in a three-spot range for the AI and the opposing player increases, the heuristic value of that board potentially increases or decreases tenfold. This tells the minimax algorithm to pick a position on a board that is more likely to produce a win on that board. Winning moves are more emphasized than good moves, and good offensive and defensive moves are prioritized over irrelevant moves. Board configurations with greater win possibilities, or ones that result in a win, are served to the algorithm.