CSCI 446, A.I.
Jon Koenes, Adam Bartz, Stephen Bush
Fall 2014

Polar Tic-Tac-Toe:

Design Document

Index

R1: A basic Polar Tic-Tac-Toe game with a legal move checker

The Polar Tic-Tac-Toe game is implemented with a simple GUI interface and a status frame that updates the player on the current status of the parts defined below (Heuristic, Classifier, etc…). The class relationships are depicted in the UML diagram below on page 6. The legal move checker is implicitly implemented by passing the AI or Human player a list of playable nodes. The AI can only see these playable nodes as valid moves and therefore only performs its heuristic, classifier, or TDNN functions on these nodes. The Human player is only allowed to select the cells represented by the playable nodes.

At the beginning of the game every cell is considered a legal move. After the first move, the playable nodes list only contains neighbors of played nodes, in order to help reduce the state space. Once a cell is filled, the playable nodes list is updated to represent this newly filled cell (removed from playable nodes) and the new non-filled neighbors of that cell (added to the playable nodes). Cells are marked as filled by a 'x' or 'o' character with the character 'n' indicating an empty cell. The Human player can display legal moves by checking the appropriate box on the GUI. A move is made by double-clicking the appropriate cell. A single-click will only select the cell as once a move is made it can not be undone.

<u>R2:</u> Win Checker using resolution with unification

The win checking class is called Resolution. It uses the Back-Chaining and Unification algorithms to process queries, as well as add knowledge when facts are made. It contains three lists, one for known Facts (e.g. "at(X,2,2)") represented as Strings. New facts are added after every move, and once a fact is added, it is also unified with the second list, the list of Rules, to then create sentences that are added to the Knowledge list. The rules are first order logic sentences, also represented as Strings, and there are only four of them:

at(player,ring,radial) & at(player,ring-1,radial) & at(player,ring-2,radial) & at(player,ring-3,radial) | win(player)

at(player,ring,radial) & at(player,ring,radial-1) & at(player,ring,radial-2) & at(player,ring,radial-3) | win(player)

at(player,ring,radial) & at(player,ring-1,radial-1) & at(player,ring-2,radial-2) & at(player,ring-3,radial-3) | win(player)

at(player,ring,radial) & at(player,ring-1,radial+1) & at(player,ring-2,radial+2) & at(player,ring-3,radial+3) | win(player)

Each rule represents one of four directions that could be a win for a player at a certain location (i.e. across rings, across radials, and the two different angle of spirals). The "&"'s are in place of " $\wedge$ " 's and the "|"'s are in place of " $\Rightarrow$ ", just for clarification. Originally, there were 16 rules to figure in all the ways a single node could be part of a win, but that was found to be terribly redundant and inefficient. Therefore, the rules were reduced to the four given above. When a fact is added to the knowledge base, such as at(X,4,8), then four clauses would be added to the Knowledge list:

at(X,3,8) & at(X,2,8) & at(X,1,8) | win(X)

at(X,4,7) & at(X,4,6) & at(X,4,5) | win(X)

at(X,3,7) & at(X,2,6) & at(X,1,5) | win(X)

at(X,3,9) & at(X,2,10) & at(X,1,11) | win(X)

The first at (X,4,8) disappears from each of the rules due to resolution at (X,4,8), each of the rules, and the necessary math is also completed, which takes into account when the radial becomes less than 0 or greater than 11. Once knowledge and facts have been added, then queries can be made to the Resolution class checking for a win for either X or O (for example, using a string formatted as "win(X)"). The Back-Chaining algorithm then goes through checking. First, it will see if the goal is disjunctive. If it is it will separate the goal into parts and then recursively call itself on each part of the goal. If the goal is not disjunct it will then check the list of Facts, to see if the goal is a fact, and therefore true trivially. If the goal is not a fact, it then checks the list of Knowledge, resolving itself if it finds the goal as a result of an implication. If the goal is a fact, the algorithm will then get a unification string, resolve itself with the knowledge String, and then add the antecedent clauses as the next goal to be met (recursively calling itself on the new goal). If at anytime all the goals are met, the algorithm then returns True for that query.

The Unification algorithm follows exactly as any psuedo-code that can be found. It simply checks two clauses represented as Strings and returns a 2d String array that contains each variable and its substitution. If no substitution exists, then the String "Failure" appears in the array to signify that they cannot be unified. The list of variables available are simply "player", "ring", and "radial". Functions include "at(" and "win(", lists are known to contain a comma ",", and arguments are surrounded with a pair of parentheses "()". The occurs check is trivial, as no variable is ever given a function as a substitution in this particular situation.

<u>R3:</u> Heuristic function for AI

Our design incorporates a Heuristic Interface, which requires an implementation of one or more evaluateState() functions. These functions allow a single prospective move or even an entire board state to be passed into evaluation functions and a value is returned which represents the comparative strength of each move.

There are two primary heuristics implemented for the AI:

Heuristic #1:

The first is an evaluation of a single board space. This heuristic is used by a greedy AI to efficiently do a shallow examination of all of the available moves and select the best-looking one at the time. It works by examining the spaces surrounding the target and calculating a comparable value for that space using factors like adjacent spaces of the same symbol, availability of future moves, and spaces blocked by an opposing symbol. Each direction (vertical, horizontal both ways, and both diagonals) is examined up to 3-spaces out counting the number of symbols found which match the specified symbol. If the opposite symbol is found, it looks no further in that direction. Points are assigned based on the number of symbols found, 1 symbol is 1 point, 2 symbols is 3 points and 3+ is 9 points. Special attention is paid to directly adjacent symbols, such that if taking that move would complete 4 in a row then a value of 36 (maximum score) is immediately returned indicating that it is a sure-win.

Heuristic #2:

The second Heuristic incorporates the first, performing a similar evaluation on each board space, generating a value for the entire board state which can be compared to other states and enables a series of moves to be compared. This heuristic is used by the MinMax tree to search farther ahead and look for better, smarter moves.

<u>R4:</u> Classifier

The classifier used for Polar Tic-Tac-Toe is a k-nearest neighbor classifier where the default value of k is set to 1. This value can be changed through a parameter. However, due to the small number of test cases the best results are rendered from a k value of 1. There are nine hard coded test cases that each have three values. The first and second values are based on the surrounding symbol count (my symbol and enemy symbol respectively) of the one or two cell(s) in the state with the greatest of those values. These values are based on cells that could lead to a potential 4-in-a-row, discounting blocked paths. The third value is one of four classes defined as; 'win', 'possible win', 'possible loss', and 'loss'.

The classifier returns one of the four classes mentioned above for the current state. This class value is defined by integer values associated with each class. In this way a loss is < 0 (loss = -2, possible loss = -1) and a win is > 0 (win = 2, possible win = 1). The class loss and win refer to an immediate 'win'/'loss' given a next move on the current state, whereas the 'possible win'/'possible loss' indicates the next move could move the current class up one step. I.e. the current state classified as 'possible loss' could become classified as 'loss' with the next move.

Three public methods are used to achieve these results. The first method changes the k-neighbor value. The second method returns the class (one of four) for the given state. The third method sets the priority to aggressive or defensive. The priority is a boolean value that gives the classifier either an aggressive (optimistic) or defensive (pessimistic) perspective. Since a current state can potentially lead to a win or loss depending on if you prioritize your

3-in-a-row above your opponent's 3-in-a-row, this priority value gives the classifier precedence.

Euclidean distance is used to measure the distance of the values of the current state with the test cases. This distance measure works very well as it does not count diagonals which resolves possible ties. In the case of a tie (equal distance between two neighbors) a coin flip is used. This tie should only happen in the case of a state resting in the middle of 'possible win' and 'possible loss' where 'my symbol' and 'enemy symbol' both equal a count of 1 with 3 other potential empty cells (that could contribute to a win/loss).

<u>R5:</u> Temporal Difference Neural Network

The NeuralNet class represents the Temporal Difference Learning Neural Net requirement. It simply takes in a board state and outputs a value between [0,1], indicating the chances for X to have a win in that state. It is designed to use arrays as placeholders for nodes with corresponding values and outputs. It contains a list of "nodes" as input, which, although set to 96 currently, can be set to any number if need be. There is also a layer of hidden nodes which can have any number of nodes. Lastly, although it is represented as an array for ease of programming coherency, the result node only contains a single value as the neural net is designed to be a simple perceptron. The network is calculated like any other. The input layer simply outputs whatever value it has, which is then multiplied by the weight of the edge between the input node and the corresponding hidden node. Once the input nodes are set and weights exist, the hidden node sums all the values (output of input nodes multiplied by weight of edge) going into that particular hidden node. It then uses a sigmoid function as an activation function to give an output, which is stored in an hidden-node-output array. This process is then repeated for the final results array, which also uses the sigmoid function as an activation function.

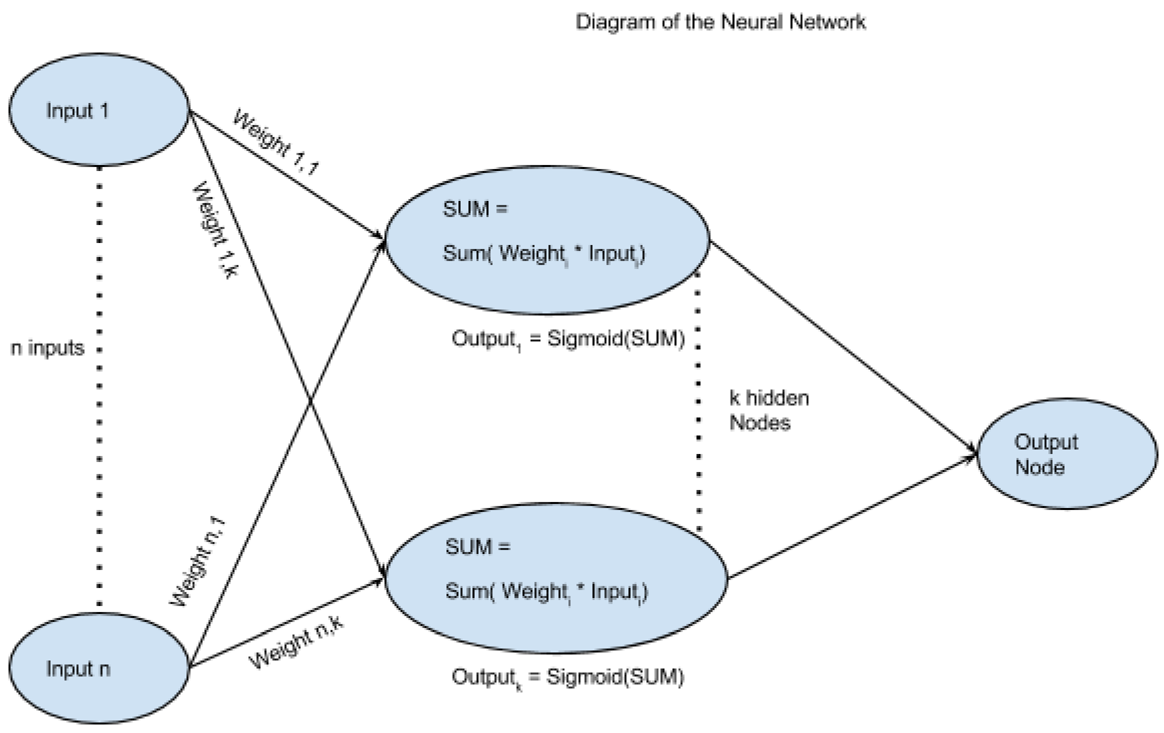


Diagram of the Neural Network
Input 1
Weight 1,1
Weight 1,k
n inputs
Weight n,1
Weight n,k
Input n
SUM =
Sum(Weight$_i$ * Input$_i$)
Output$_1$ = Sigmoid(SUM)
k hidden Nodes
SUM =
Sum(Weight$_i$ * Input$_i$)
Output$_k$ = Sigmoid(SUM)
Output Node

The class has two options for initialization. It can either create a new neural net with random weights and train from there, or it can load old weights from a previous training to use for evaluation or to continue training. Training is done using a temporal difference algorithm, which will be explained in detail in the final report. For the design, since the knowledge of how much correction a state, s_i ,needs isn't known until the correction for state s_{i+1} has been made, there is a stack that contains a list of the states the game has gone through. Only once the game has ended does the actual correction and training of the neural net take place. The training is done using typical back-propagation using the derivative of the sigmoid function , with the final state being the first to be corrected. It is corrected to 1 given what ended up being a win for X, a 0 for a loss for X, and 0.5 in any case that ended as a cat's game.

The learning rate is hardcoded and has been changed several times for testing purposes, from 0.001 up to 0.3. The class also has a bias that is hardcoded in as well, as "nothing can be learned without bias". The bias is set to 0.1 and can be simply thought of as an extra input and an extra hidden node, where every weight it has is always 1. Other than that, the actual performance and final parameters have not been decided yet, as training with differing numbers of hidden nodes and differing learning rates is taking place.

R6: Examine Neighboring States for each ply, applying the heuristic function to those states

We have created one concrete IPlayer AI class who uses a simple heuristic evaluation (see Heuristic #1) to evaluate each state from a list of available moves given to the player by the Game Engine. Each move is assigned a value according to it's estimated impact on the game if selected , and the move with the highest value is returned to the Game Engine to be played.

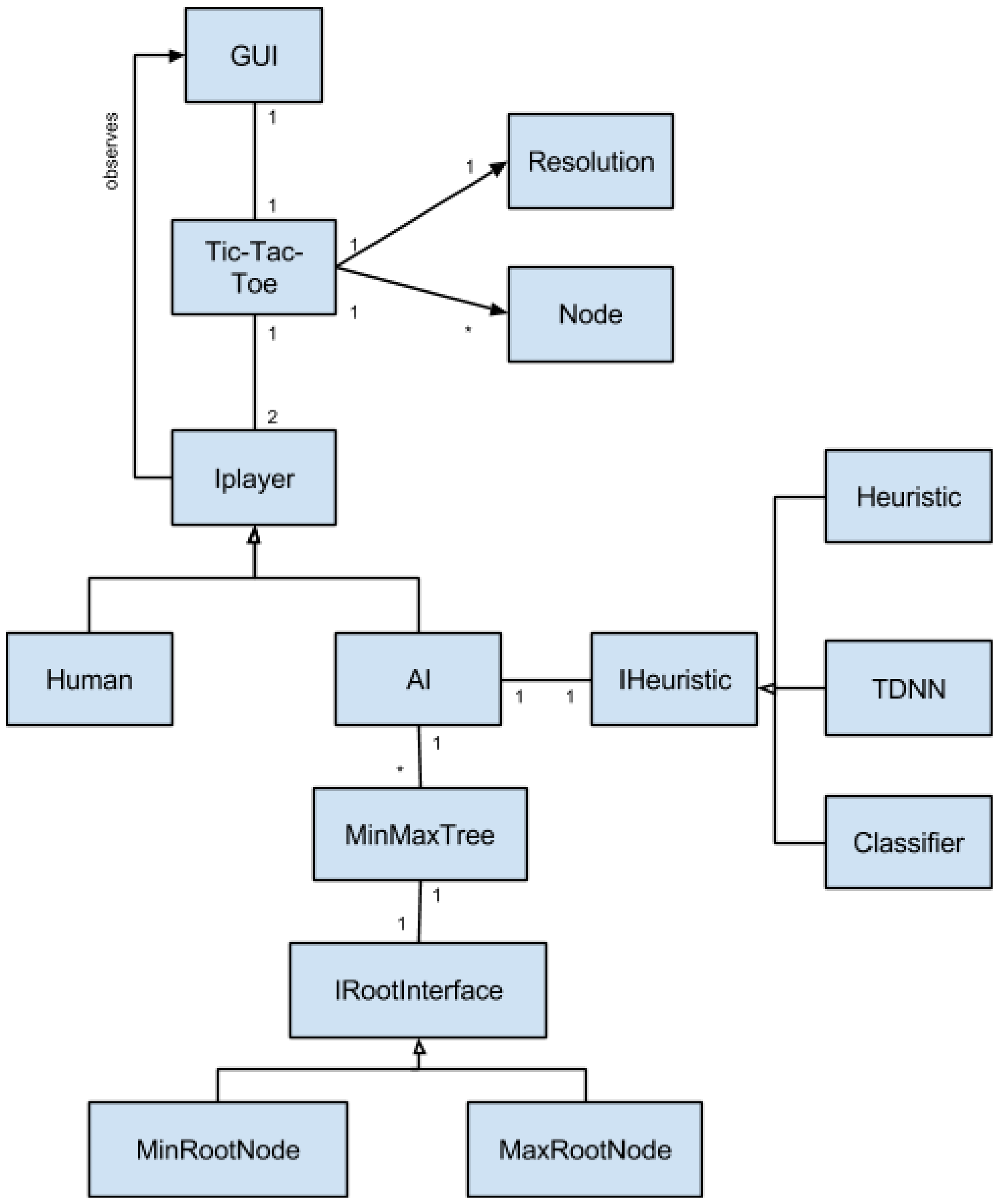
R7: Minimax Search with and without Alpha-beta pruning from either X or O perspective

The MinMax tree consists of three component classes: a MinMaxTree class which serves as the interface between the AI Player and the MinMax tree evaluator, a MaxRootNode class and a MinRootNode class which both implement a IRootNode interface. We chose to use two root classes instead of one. While they both behave similarly, their differences are significant enough that should one class be used instead, very little code would be shared directly by them and it would be an entire class of If-one-else-the-other functions. Splitting the classes up removes these cases, making the code much more streamlined and allowing it to run more efficiently by removing the need for these checks.

The MinMaxTree class has two operational modes which affect the depth and and thus the value of the data it collects about the game. In mode #1, the tree will be expanded up to a specified depth, at which the tree is resolved and the path that appears best is followed. In mode #2, the tree is expanded continuously (and evenly across all branches) for a specified amount of time, but the potential depth it might explore is unbounded. This allows the tree to expand deeper if time allows (a well pruned tree, for example), and also prevents it from expanding too deeply if it's starting to take too long (a less-prunable tree with many potential neighbors). A combination of both modes may be used, and both modes can be run with or without alpha-beta-pruning.

One additional feature, which may or may not be implemented if time allows and/or performance requires it, is tree re-usability. Once a tree is expanded and moves are made, the next starting-state to be evaluated by the min-max tree will likely already have been expanded in the previous tree, and it could be expanded further from that point rather than the entire tree being re-created. However, this would add significantly more complexity and would require design changes that aren't necessarily warranted at this point, as well as possibly yielding negligible benefit.

UML Diagram:



Design Implementation:

The Polar Tic-Tac-Toe game uses an Observer pattern, a Model-View-Controller pattern, and several interfaces to implement the game. The GUI functions as the View in the MVC pattern and handles the PTTT game's graphical "listening" for updates in the Model. The Tic-Tac-Toe class functions as the Controller, or game engine, and drives the logic behind the game itself (calling the necessary functions such as the Win-Checker and play methods of the player classes). The model part of the MVC constitutes the various classes that contain the methods and functions used in the game (E.g. Resolution, MinMaxTree, Heuristic, etc…)

The Human and AI player classes implement the play and subscribe methods from the Iplayer interface. The Human class's play() returns a node that represents the human player's next move as interpreted from click updates in the GUI. The AI player's play() returns a node based off of any Heuristic functions used and/or the MinMaxTree depending on the initial game parameters (where the human player can decide what kind of AI player to use). The Node class contains information about the Node or cell. This information is symbolic of the cell's location, whether or not an X or O is in that cell, as well as the cell's neighbors.