CSC 350/500 – Project 2: Connect Four

**Due Date:** Friday, March 7, 2025

# Introduction

In this project, you will implement Minimax and Alpha-Beta Pruning to build a competitive Connect Four-playing agent, as well as an agent capable of playing a 3-player version of Tic-Tac-Toe.

I supply code (in Java and Python) for the games themselves and an agent that makes random moves. In ConnectFour, the only thing you need to do is create another agent class. (If you are not familiar with Connect Four, [read about the game on Wikipedia](https://en.wikipedia.org/wiki/Connect_Four).) For 3-player Tic-Tac-Toe, you will need to expand the AI agent that I’ve already started.

This is a group project. A group can have up to 4 students. A group must be either all undergraduate students or all graduate students, no mixing.

## File(s) to Edit

For Connect Four, you will create a subclass of ConnectFourPlayer, which can be found in c4players.py or ConnectFourPlayer.java. You will probably also want to briefly modify the main function in connect4.py or ConnectFour.java to use your player. Do not modify any other Connect Four code.

For 3-player Tic-Tac-Toe, you will add to the existing TTT3PlayerAIPlayer class.

Please *do not* change the names of any of the provided functions or classes. Modifying code that you were instructed not to modify is grounds for losing credit on this assignment.

More detail about the code provided can be found in the About the Code file.

## What to Submit

Zip all project files for your chosen language, including your answer for Q7, and submit it on Blackboard by the due date. Any submission after the due date is subject to the late penalties described in the syllabus.

## How You Are Graded

Your code will be evaluated based on sample test cases (in the case of the functions you write) and in-game performance (for the overall agent). The grading rubric for this assignment is on the final page of this document.

# Assignment Questions

*For all questions below, function names in camelCase refer to Java. In Python, the functions have similar names, but in snake\_case. Name your functions according to the style expected in your chosen language.*

# Part One – Connect Four

## Question 1 (2 points) – AI Player Class

Implement a subclass of the ConnectFourPlayer class called ConnectFourAIPlayer. For now, follow the guide of ConnectFourRandomPlayer by having the constructor get a copy of the game model. (You may wish to review the public functions of the game model.)

Don’t worry too much about the getMove function right now. (We will implement this function in Question 5.) For now, write something basic in there that ensures a legal move is made. (For example, pick the move closest to the center, or pick the move furthest to the left.)

Note that for this question and all other questions in this assignment, it is the responsibility of the agent to make sure its move is legal, where a legal move is a number between 0 and 6 (inclusive) and the column specified is not already full. The game model will consider an illegal move to be a forfeit.

When you complete this, congratulations! We have a functioning agent, though you should still be able to defeat it easily.

## Question 2 (2 points) – Terminal Test

The minimax and alpha-beta algorithms require a number of helper functions in order to work properly. Implement a function terminalTest that takes in a board state as an argument and returns true if the game would terminate in this state – that is, if a player has won or if the game has reached a full board (a draw).

In Python, the game board is represented as a 2-d list of integers. In Java, the game board is represented as a 2-d array of ints.

The TicTacToeAIPlayer and TTT3PlayerAIPlayer classes provide a sample of this function for 2-player and 3-player Tic-Tac-Toe, respectively.

## Question 3 (3 points) – Actions

For any potential board state, the algorithm must be able to identify what actions are valid. As noted above, an action in Connect Four is a number between 0 and 6 (inclusive) where the number represents a column. Implement a function actions that takes in a board state as an argument and returns a list (Python) or int[] (Java) representing the valid moves from that state.

The TicTacToeAIPlayer and TTT3PlayerAIPlayer classes provide a sample of this function for 2-player and 3-player Tic-Tac-Toe, respectively.

## Question 4 (3 points) – Results

Naturally, if we know the actions, then we need to be able to follow through with creating the game state that follows from those actions. Write a function result that takes two arguments: the board state and an action (an integer between 0 and 6). Return the board state that exists after that action is carried out.

The TicTacToeAIPlayer and TTT3PlayerAIPlayer classes provide a sample of this function for 2-player and 3-player Tic-Tac-Toe, respectively.

*Hint 1: Typically a game state includes whose turn it is, but our game state only includes the game board. How can you deduce whose turn it is? (You’ll need to do this since the grid uses 1s and 2s in the grid to represent whose tokens are whose.)*

*Hint 2: Make sure that you’re making a sufficiently deep copy of the grid when adding the new move. If your copies are too shallow, then your minimax algorithm may not be working from the correct game state as it backtracks.*

## Question 5 (8 Points) – Alpha-Beta

Implement the alpha-beta pruning algorithm described in the textbook (pseudocode on p. 154), including the minimum-value and maximum-value functions as described. The getMove function should start or call the alpha-beta pruning algorithm. You’ll also need a utility function. Since we’re currently not cutting off the depth of our search, define your utility function so that winning states for your player return 1000, losing states for your player return -1000, and draw states return 0.

Make sure that your agent behaves properly regardless of whether the agent is Player 1 or Player 2.

Until we implement a cutoff evaluation function (next question), this algorithm will probably take a *long* time to complete. See below for using 2-player Tic-Tac-Toe to test and debug your algorithm.

### How to Test/Debug For This Question

The 2-player TicTacToeAIPlayer class provides skeleton code for the functions related to getMove and Alpha-Beta Pruning. You should be able to paste your implementation for Connect Four directly into 2-player Tic-Tac-Toe (or vice versa) and see results, since the algorithm is independent of the game being played, as long as you’ve implemented Q2-Q4 correctly. (In Java, you may change the return types on any of these functions for Tic-Tac-Toe, except getMove().) You can run the game from TicTacToe.java or tictactoe.py.

In my tests, the TicTacToeAIPlayer comes up with a move pretty much instantaneously (except on the first move). If your code is correct, the AI player should NEVER lose. If you (the human player) play correctly, the AI player will always force a draw. If you screw up, the AI player may win. If all of these factors check out in Tic-Tac-Toe, including when the AI player is the X or O player, then you should feel confident in your code for this question. If the code works for 2-player Tic-Tac-Toe, but not Connect Four, then there is probably an issue with your solution(s) to earlier questions.

## Question 6 (8 Points) – Difficulty and Mid-Game Utility

If you’ve written your code correctly, you probably have an agent that is incredibly difficult to beat. That’s no fun. It also probably takes quite a long time to decide what move to make because of the game’s branching factor. Now, we are going to modify our code so that the agent cuts off its search after a certain number of plies. The more plies that our agent investigates, the more challenging it will be to play against.

Modify your agent by adding an instance variable that represents the maximum depth that you will search to. Add an optional integer parameter to the constructor (Python) or a second constructor that takes the model and an integer (Java) to represent this cut-off level. (For your default max-depth agent, you could use 42, since that is the maximum number of turns in Connect Four.)

Modify your minimum-value and maximum-value functions so that, in addition to returning a state’s utility if we’ve reached a terminal state, it also does so if we’ve reached the cutoff depth. You should also modify each of these functions so that they have a parameter representing the current depth of search. (Section 5.3.2 of the textbook describes the necessary changes to the Alpha-Beta Search pseudocode.)

Of course, now our utility function is insufficient, since we wrote it to only measure terminal states. Here is where your creativity comes in. Modify your utility function so that it is capable of measuring a state’s utility when given a non-terminal state. What this function looks like is up to you. Give some thought to what a high-quality or low-quality state looks like. Remember that winning and losing states should still return the highest and lowest possible values, respectively.

## Question 7 (3 points) – Explanation of Utility

Write a brief summary of the utility function that you developed for Question 6. Explain the logic that went into measuring a game state’s utility. I should be able to follow the logic of how your agent determines the quality of a state without much effort.

# Part Two – 3-Player Tic-Tac-Toe

3-Player Tic-Tac-Toe uses a 4x4 grid and has three players (X, O, and +). Otherwise, the game is the same as regular Tic-Tac-Toe. The object of the game is to get three-in-a-row. The winner of the game is the first player to achieve that goal.

In the code provided, each square is given a number 1-16 (see grid below). Legal moves must be an unoccupied square in that range.

|  |  |  |  |
| --- | --- | --- | --- |
| 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 |
| 9 | 10 | 11 | 12 |
| 13 | 14 | 15 | 16 |

## Question 8 (6 points) – Multiplayer Minimax

Most of the 3-player Tic-Tac-Toe agent has already been implemented in the TTT3PlayerAIPlayer class. You will find that the actions(), result(), terminalTest(), and utility() functions have already been provided.

The only function that needs to be written is the getMove() method. Adapt the minimax algorithm (see pseudocode on page 150) to handle three players using the modifications discussed in the textbook and in class. Remember that this means that each player is essentially a Max player! Feel free to add any helper methods that you might find useful.

~~For this question, you only need to implement the basic minimax algorithm. That is, you are not required to include alpha-beta pruning or a depth cutoff with an evaluation function for non-terminal states.~~

Correction: For this question, you only need to implement the basic minimax algorithm. That is, you do not need to include alpha-beta pruning. However, because the game tree is larger than in traditional Tic-Tac-Toe, we will need a depth cutoff and an evaluation function for non-terminal states. Realistically, your depth cutoff should be about 5 in order to get a speedy result at the beginning of the game.

As a hint, take a look at the utility() function, which now returns an integer array instead of a single integer.

# Connect Four Tournament! (3 points extra credit)

I will pit all Connect Four agents submitted against each other in a round-robin battle to determine whose agent is the best. The winner(s) receive the extra credit points. The tournament will work as follows:

* Python and Java agents will participate in separate tournaments.
* Your agent may use additional information (e.g. lookup tables) beyond the requirements above. However, the vast majority of the decision making should be based on alpha-beta pruning. Any code that applies only to the tournament should be marked as such with comments.
* The tournament will be held seven days after the project deadline. Agents not submitted by this time will be ineligible to participate in the tournament/receive extra credit.
* An agent will play each other agent twice, once as Player 1 and once as Player 2.
* The winner of each game receives 2 points. In the case of a draw, both players receive 1 point. Losses receive 0 points. Forfeits due to illegal moves (or any other uncaught Exception) are considered losses.
* The tournament will be conducted using a maximum-depth to be determined. It will not be a full-game search. It is likely that the tournament will be held with a depth parameter such that 5 <= depth <= 10.
* The winner of the tournament will be the agent that racks up the highest point total. A winner will be selected to receive extra-credit in both the Java and the Python tournament.
  + Note: A tournament must have at least 4 agents for the winner to receive extra credit.
* Tiebreaking: In the event that two or more agents have the same number of points, tiebreaking will occur as follows:
  + Rank all tied agents according to points earned in games against the other tied agents.
  + If one agent leads this ranking, that agent is the winner. If multiple agents are tied to lead this ranking, but the number of agents is fewer than the number of originally-tied agents, then repeat the process using the remaining agents.
  + If all agents have the same score in the tiebreaker ranking, then all remaining agents receive the extra credit.

# Rubric

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Question** | **3 points** | **2 points** | **1 point** | **0 points** |
| Q1 – AI Player Class  (2 points) | n/a | Class exists with valid constructor and an overridden getMove() method. | Class exists with valid constructor. | No attempt |
| Q2 – Terminal test (2 points) | n/a | No more than minor errors. | Major errors in checking for win/draw or neglects to check for a draw. | Not attempted, skeleton code, or does not compile. |
| Q3 – Actions (3 points) | No more than minor errors. | One significant logic error. | Multiple significant logic errors. | Not attempted, skeleton code, or does not compile. |
| Q4 – Results (3 points) | No more than minor errors. | One significant logic error. | Major error or multiple significant logic errors. | Not attempted, skeleton code, or does not compile. |
| Q5a – Alpha-Beta Algorithm (6 points) | No more than minor errors. | Significant errors exist in algorithm. | Major errors that prevent execution of the algorithm. | Not attempted, skeleton code, or does not compile. |
| Q5b – Utility function  (2 points) | n/a | No more than minor errors. | Significant errors or invalid assumptions about depth or player turn. | Not attempted, skeleton code, or does not compile. |
| Q6a – Depth cut-off  (2 points) | n/a | Correctly implemented in all relevant functions. | Attempted in all relevant functions, or correctly implemented in some. | Not attempted, skeleton code, or does not compile. |
| Q6b – Evaluation function  (3 points) |  |  |  |  |
| Q6c – Agent Performance  (3 points) | Challenging Player | Challenging player under certain circumstances, but still makes some incorrect decisions | Player regularly loses due to questionable move decisions. | Player does not compile or constantly crashes due to illegal moves. |
| Q7 – Explanation  (3 points) | Adequate explanation of strategy used in evaluation function. | Some sense of strategy employed, but details are missing or incorrect. | Describes utility or evaluation at a high level, but not the specifics of the code employed for Q6. | No response submitted |
| Q8 – 3-Player Tic-Tac-Toe (6 points) | No more than minor errors. | Significant errors exist in algorithm. | Major errors that prevent execution of the algorithm. | Not attempted, skeleton code, or does not compile. |