1 The Program

This program contains two classes: one to manage the Connect Four board and one to handle the Monte Carlo Tree Search. The Main Function of this program handles gameplay. All gameplay from creation of new gameboard to detecting a final gamestate is contained in a while loop, giving the user an option to play again. There is a section of code that can be commented out to allow the user to select their own choices. This was primarily used for testing purposes to ensure all the member functions of c4Board.

2 The Algorithm

Program Algorithm:

1. Initialize an instance of c4Board for a new game
2. Get move from player (or AI)
3. If move is legal, print new move to console
4. When a player has 4 tokens in a row, print "Win" message to console
   (a) If no player wins, print "Draw" message

Solution algorithm:

1. Based on current gameState, see what plays are possible. (for columns 0-6)
2. For each possible move, determine immediate counter moves
3. For each possible counter move, randomly select moves until the game is over
   (a) Keep track of wins, loses, and draws
   (b) If a node has all subsequent states calculated, write that information to a text file
   (c) If a stored node shows up in a different search some time, don't evaluate it again
4. Backpropogate w/l/d data, as well as the number of nodes tested.
5. Choose the move with the highest number of wins to draws or losses (confidence = number of nodes tried)

3 The Data Structures

This program contains two classes: `c4Board` and `Node`. All the gameplay is handled by `c4Board`. Each board is essentially a node in a doubly linked list, with pointers back to each previous gamestate and a pointer to the next board once the new board is created. Each node has a nested array to keep track of each row and column of the gameboard. By default, the array is propogated with -1s The output stream operator is overloaded to provide formating when printed to the console. Formatting is as follows:

0	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	-1
	0	1	2	3	4	5	6

Each instance of `c4Board` also keeps track of the player who just moved, whether the game is over or not, and which player has won. The class also contains an operator overload for `==`, a member function to generate a new gameboard when a player takes a turn, and a member function to detect when the game is over.

The Monte Carlo Tree Search algorithm is implemented in the class Node. Most of the functionality is provided in the member function `makeMove()`. This function returns the number of the best column to drop a token in. Other member funtions include `getChildren()` and `sampleNodePath()`. These functions handle the Expansion and Simulation steps of the MCTS. The Selection process is made to include every possible move based on the current gamestate. This gung-ho method is possible because Connect Four only has a branching factor of 7, at worst. Once a column is full, the branching factor will decrease.

Other member functions of `Node` include `isPossible()` and `getGameState()`. These fucntions are used to determine if a given column has emtpy spaces left and who, if anyone, has won the game, or if the game is still ongoing.

4 Reflections & Improvements

Upon reflection of this project, the implementation of `sampleNodePath` could use a more efficient method of playout, as opposed to the random selection of lightweight playout. An example of such an improvement might be looking for specific characteristics of a gamestate, such as the opponent having three tokens in a row. By searching for specific gamestates, the AI portion of the program would be able to make a choice quicker by eleminating the need to run `sampleNodePath()` in some instances.

Another improvement would be adding the ability to store certain gamestates and their calculated win/loss ratio by writing them to a text or xml file, to be loaded back into the program each time it is run. This would give it a persistent memory and allow the AI to learn which paths are worth pursing and which are not. The current iteration of this program does not store anything beyond the immediate children of a given move. Adding a persistent memory would give more reason to store nodes 2 or 3 moves beyond the current one, along with the relavent win/loss information.

The current program version only stores the up to 7 moves possible, examines each one, and then makes a choice. Such a simplistic system that immediately makes a choice based on the current data does not ever have an opportunity to reexamine a stored child node, since all stored nodes are more or less irrelavent after a choice is made. As such, there was no need to implment a confidence range and choices are not made based on a confidence threashold.

A future iteration of this program could easily be made to include all of the mentioned improvements by providing an overload for the output stream operator in `Node`, a class constructor for `Node` that can reconstitute a previous instance (or at least the relavent data necessary), and by modifying the function `makeMode()` to compare gameboards from memory with children Nodes generated during either `getChildren` or `sampleNodePath`, depending if the programmer felt that all generation of children nodes should occur in a single function or if the random-generation of nodes 2 or 3 moves out was best left to `sampleNodePath`.