Com S 472/572 Spring 2022

Project 2: Checkers-Playing Agents (150 pts)

Due at **11:59pm**

Friday, Apr 29

1. The Game

English draughts, also called American checkers, are a popular game played by two opponents on opposite sides of an 8X8 gameboard. It is a game on which AI witnessed one of its earliest successes, as best evidenced from a self-learning checkers program written in 1959 by Arthur Samuel, a pioneer in computer gaming who also popularized the term "machine learning".

In this project, your task is to write a Java program capable of playing checkers against a human player. For rules of the game, please visit Wikipedia (https://en.wikipedia.org/wiki/English_draughts). For a quick tutorial, you may also watch a YouTube video (e.g., https://www.youtube.com/watch?v=ScKIdStgAfU).

2. Your Task

You are required to implement three game playing agents. The first one employs the alpha-beta pruning while the second one employs the Monte Carlo tree search (MCTS). These two agents are respectively implemented by the classes `AlphaBetaSearch` and `MonteCarloTreeSearch`, both of which extend the abstract class `AdversarialSearch`. Below is a list of requirements:

a) Both `AlphaBetaSearch` and `MonteCarloTreeSearch` need to implement a method `makeMove()` that takes the current state as input and returns a move/action to be made by the agent.

b) Implement a move generator function `legalMoves()` within `AdversarialSearch` that takes a state as input and returns a list of legal moves at the state.

In addition, the class `AlphaBetaSearch` needs to implement an evaluation function that takes a state of the game as input and returns a value. The utility function for terminal states has the following values:

- 1 if a win by the agent,
- -1 if a loss by the agent,
- 0 if a draw.

The score on an internal node is generated according to a heuristic defined by you.

The class `MonteCarloTreeSearch` needs to implement the four steps carried out within each iteration: selection, expansion, simulation, and back-propagation, as separate private methods. Generation of a playout follow the rules below:

a) For the selection step that starts at the root of the search tree, use the upper confidence bound formula UCB(n) (i.e., UCB1(n) in the textbook on p. 163) and set the constant C used for balancing exploitation and exploration to its theoretically optimal value $\sqrt{2}$.

b) During simulation, at every state make a uniformly random choice among all legal moves, whether for the agent or for its human opponent.

c) During back propagation, a draw from the playout causes the numerator of every node, whether black or white, along the upward path to the root to increase by 0.5.

To represent the search tree, you are provided a compact implementation of the child-sibling tree in two classes `CSNode` and `CSTree`. However, you are free to consider other tree implementations. In the latter case, simply delete these two classes from the package.

You may make use of the `AlphaBetaSearch` and `MonteCarloTreeSearch` implementations from the online code repository of the AIMA textbook.

Your `main()` method (inside the class `Checkers`) should provide the following three strategies for playing the game:

a) using the alpha-beta pruning to decide on each move throughout the game (this is the first agent);

b) using the MCTS to decide on each move throughout the game (this is the second agent);

c) randomly choosing one between alpha-beta pruning and MCTS to decide on each move (this third agent is referred to as the “hybrid” agent).

The `main()` method should be able to do the following:

a) Take as input a move from the user (i.e., the human player).

b) Update the board with the human player's move (see Section 3 for details).

c) Update the board with the agent's move from the alpha-beta search or MCTS (or output the move if not using the provided graphics).

d) Repeat the steps until the end of the game.

3. Graphical Interface

To help your implementation and to make this computer game appealing, we have provided some GUI code implemented by the class `Checkers` such that the human player just needs to move pieces in the GUI, thus saving the effort of typing. Fig. 1 below shows the board configurations (states) right before and after the agent (Black) makes a move, respectively. In the left board configuration, the black piece at 6e (rows labeled 1 to 8 bottom-up and columns labeled a to h from left to right; see Fig. 2) is about to jump over the red piece at 5f to reach 4g. The outcome is shown in the right board configuration, where the human player (Red) has the only option of letting the red piece at 3f (inside a white box) jump over the black piece at 4g to reach 5f (bounded by a green box). The black piece will be captured. The outcome of this move by Red will show up in the left board configuration on the refreshed screen after a mouse click at 5f.

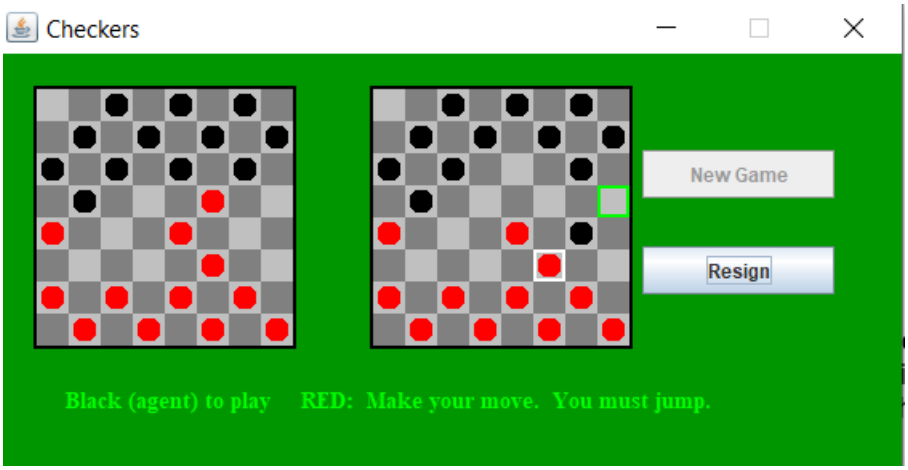

Fig. 1 Board display generated by the GUI.

In short, the left board always shows the state ***just before*** the agent (Black) makes a move, while the right board always shows the state right after this move and ***just before*** the human player makes a move. According to the rule, a move is defined as either a simple move diagonally to an unoccupied dark square or a sequence of jumps.

a) In the case that multiple jumps are made by either the agent or the human player, do not refresh the screen until the last jump is completed.

b) On current display, the right state is generated from the left state by the agent executing a move, which may consist of multiple jumps.

c) The left state on current display was generated, as a result of the human player's last move, from the right state on the ***previous*** display. That move possibly consisted of multiple jumps.

You do have the freedom of going ahead with an implementation that does not make use of the provided GUI. If this is the case, you may print the 8x8 board in the console like shown below:

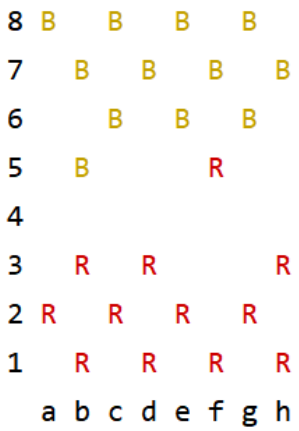

Fig. 2 Console output.

4. Submission

Write your classes in the `edu.iastate.cs472.proj2` package. Turn in a zip file that contains the following:

a) Your source code.

b) A short report in PDF that includes your results from comparisons as described below:

   a. For the alpha-beta agent, compare the effects of increasing search depth and improving the evaluation function (which should be briefly described).

   b. For the MCTS agent, compare the performance using the theoretical optimal value $C = \sqrt{2}$ with one using a different value of C set by yourself.

   c. Compare the performances by the alpha-beta search, MCTS, and hybrid agents.

c) A `README` file (optional) for comments on program execution or other things to pay attention to.

Include the Javadoc tag `@author` in class source files. Your zip file should be named `Firstname_Lastname_proj2.zip`.

Please follow the discussion forums Project 2 Discussion and Project 2 Clarifications on Canvas.