COMS 4720/5720 Fall 2024

Project 2: Checkers-Playing Agent (100 pts)

Due at **11:59pm**

Sunday, Nov 17

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 in which AI witnessed one of its earliest successes, as best evidenced from a self-learning checkers program written in 1959 by Arthur Samuel (1901-1990), 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 a game playing agent that employs the Monte Carlo tree search (MCTS). This agent is implemented by the class `MonteCarloTreeSearch`, which extends the abstract class `AdversarialSearch`. (After the project is turned in, just for fun you might try adding another class `AlphaBetaSearch` to extend `AdversarialSearch` and play the same game using alpha-beta pruning for a performance comparison with `MonteCarloTreeSearch`.) Below is a list of requirements:

a) `MonteCarloTreeSearch` needs to implement a method `makeMove()` that 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. (The load of implementation can be shifted to the method `getLegalMoves()` within `CheckersData`,)

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, every state makes 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.

The classes `MCTree` and `MCNode` represent the Monte Carlo search tree and its nodes. You may simply store the children of a node in an `ArrayList` or a linked list.

You may refer to the [MonteCarloTreeSearch] implementations from the online code repository of the AIMA textbook. Your implementation nevertheless is expected to largely follow the given skeleton code.

Your `main()` method (inside the class `Checkers`) should provide the strategy for playing the game that uses the MCTS to decide on each move throughout the game. The `main()` method repeats the following three steps until the end of the game:

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 MCTS (or output the move if not using the provided graphics).

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.

For notational convenience, rows on the checkerboard are labeled 1 to 8 bottom-up and columns are labeled a to h left to right. See Fig. 1(a). According to the rule, a *move* is defined as either a simple move diagonally to an unoccupied square or a sequence of jumps.

- Making a move first, the human player (Red) can choose one of four pieces, which are all boxed in green as shown in Fig. 1(b). The human player clicks on the piece at b3 as the choice. The display changes to (c) showing a4 and c4, all boxed in green, as destinations of two simple moves.

- The human player then clicks on a4. The board configuration (state) changes to the one on the left in (d), which is right before the agent (Black) makes a move.

- The agent moves the piece at c6 to b5 (both boxed). The resulting state is shown on the right in (d), where the only Red piece allowed to move, located at a4, and its destination square at c6 are both boxed.

- Once the human player clicks on c6, the piece will jump over the Black piece at b5, capturing it and reaching c6. The updated display will be showing the outcome on the left, and that following the agent's next move (which can be inferred) on the right, and so on.

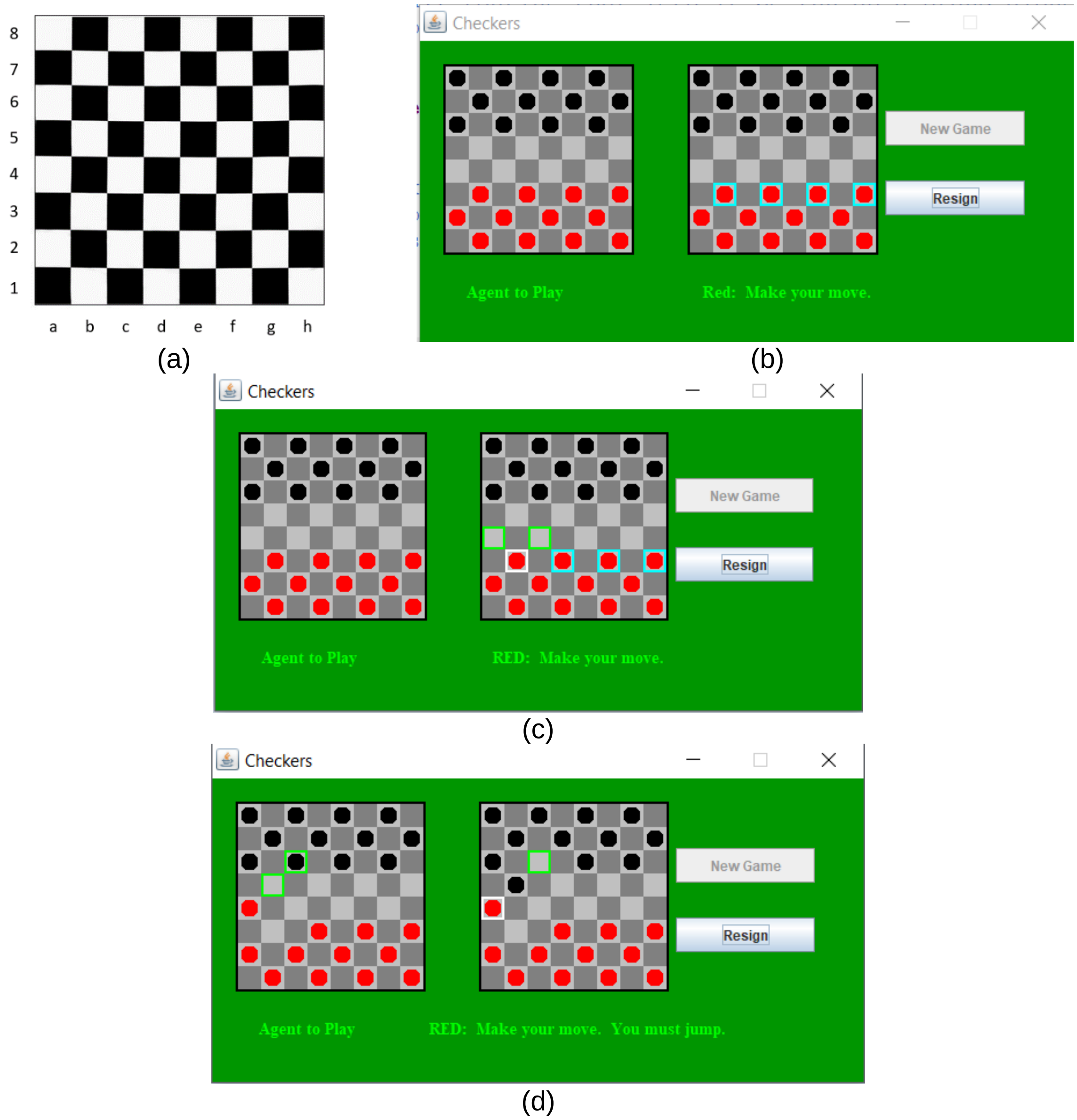
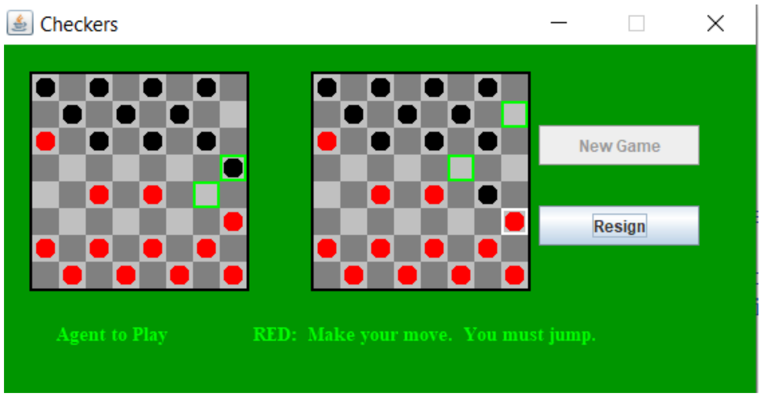


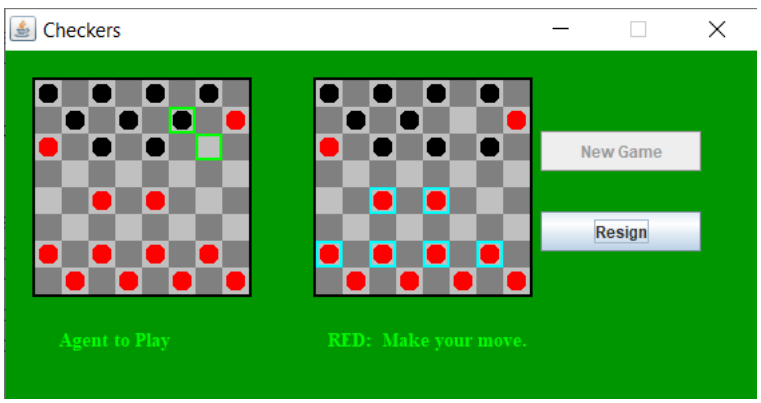
Fig. 1. The first two moves by the human player and MCTS agent, respectively.

Jumping is *mandatory* in that a player who has the option to jump must take it, whether or not doing so will cause a disadvantage. It is also compulsory to keep jumping until all the jumps are completed, according to rule 1.19 in the official rulebook of American Checkers Federation.

If a move, whether by the human player or by the agent, consists of multiple jumps, all the squares to be sequentially jumped onto should be boxed. Shown on the left and right in Fig. 2(a) are two states S_L and S_R, respectively, such that S_R results from the agent moving the Black piece at h5 to g4. In S_R, the human player can only move the Red piece at h3 (boxed in white) because it can jump over at least one Black piece. There are two jumps, first to f5 and then to h7. Both squares are boxed in green. The human player clicks on the last square at h7 on the jump path to finish the two jumps. The screen gets refreshed to show the resulting state S_L' on the left in Fig. 2(b). Immediately, the agent makes a move (from f7 to g6, both boxed) to change the state to S_R' on the right in Fig. 2(b).



(a) States S_L (left) and S_R (right).



(b) New states S_L' (left) and S_R' (right).

Fig. 2. Four states in two screen displays.

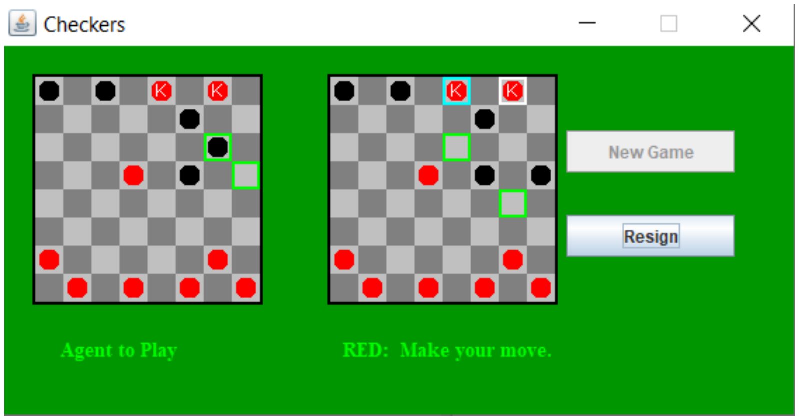
In the above example, while the state transitions is $S_l \rightarrow S_R \rightarrow S_L' \rightarrow S_R' \rightarrow \cdots$, the display sequence is $(S_L, S_R) \rightarrow (S_L', S_R') \rightarrow \cdots$. In the case that there are multiple Red pieces for the human player to choose, the right board gets updated one more time, as shown in Fig. 1(c).

a) If there exist multiple jump paths for a piece, then all the squares on each path should be boxed. One of the paths must be chosen for execution, according to rule 1.20 in official rulebook of American Checkers Federation.

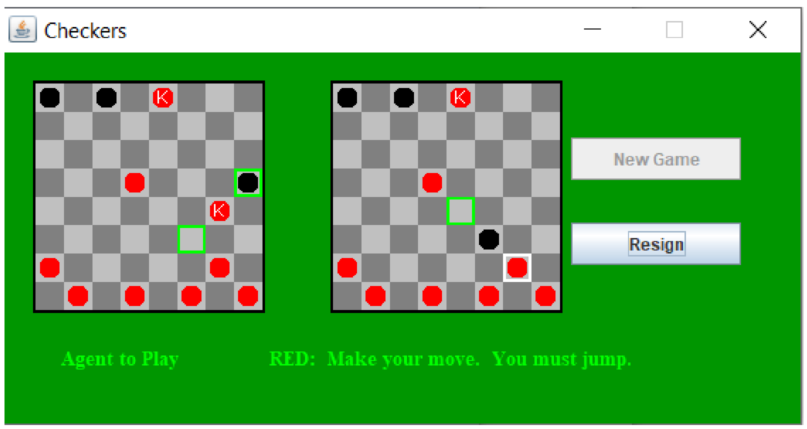
b) Some paths for the same piece may consist of a simple jump while others may consist of multiple jumps.

c) Two paths for the same piece, each consisting of multiple jumps, may share the same first few jumps.

Fig. 3 below shows how a Red *king* at g8, a piece crowned after it reached the top row, makes two jumps in a row as shown in (a) before it is captured by the black piece at h5 in (b).



(a)



(b)

Fig. 3. (a) In the right state, the Red king at g8 is about to jump to e6 and then to g4 to capture two Black pieces at f7 and f5. (b) The same Red king, now at g4 in the left state, is then captured by the Black piece at h5 in the right state.

To summarize, except for the start of the game, 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.

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 the 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 the 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 also.

4. Text Display Option

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:

```
8 B   B   B   B
7   B   B   B   B
6     B   B   B
5   B       R
4
3   R   R       R
2 R   R   R   R
1   R   R   R   R
  a b c d e f g h
```

Fig. 4 Console output.

Text display is nonetheless not encouraged because it would impair code debugging.

5. 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 to include your results from comparing the performance using the theoretical optimal value $C = \sqrt{2}$ with one using a different value of C set by yourself.

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.