Programming Assignment 1: Chess

Due: Friday 10/25/2024 @ 11:59pm EST

The purpose of programming assignments is to use the concepts that we learn in class to solve an actual real-world task. To that end you will be writing java code that uses a game engine called Sepia to develop agents that solve specific problems. In this assignment we will be writing agents to play chess.

1. Copy Files

Please, copy the files from the downloaded lab directory to your cs440 directory. You can just drag and drop them in your file explorer.

- Copy `Downloads/chess/lib/chess.jar` to `cs440/lib/chess.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/chess/data/pas` to `cs440/data/pas`.
  This directory contains a game configuration and map files.
- Copy `Downloads/chess/src` to `cs440/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/chess/chess.srcs` to `cs440/chess.srcs`.
  This file contains the paths to the `.java` files we are working with in this lab. Just like last lab, files like these are used to speed up the compilation process by preventing you from listing all source files you want to compile manually.
- Copy `Downloads/chess/doc` to `cs440/doc/`.
  This directory contains the documentation for `chess.jar`. It will be immensely helpful for you when implementing your solution.

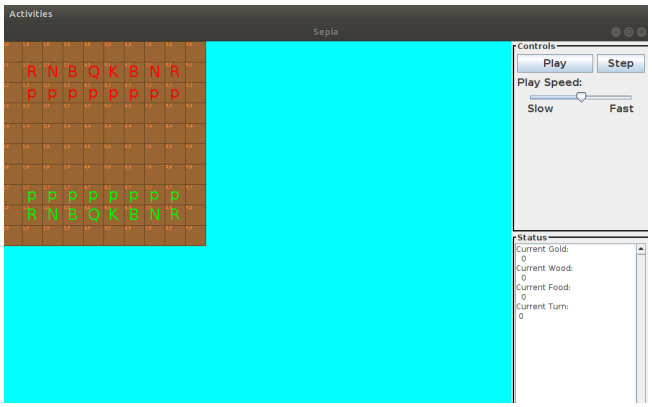
2. Test run

If your setup is correct, you should be able to compile and execute the given template code. You should see the Sepia window appear.

```
# Mac, Linux. Run from the cs440 directory.
javac -cp "./lib/*:." @chess.srcs
java -cp "./lib/*:." edu.cwru.sepia.Main2 data/pas/chess/RandomvsRandom.xml

# Windows. Run from the cs440 directory.
javac -cp .\lib\*;. @chess.srcs
java -cp .\lib\*;. edu.cwru.sepia.Main2 data/pas/chess/RandomvsRandom.xml
```

What you should see is a window that looks like this:



In chess, there are two team colors: white and black. Traditionally (at least what I've seen on most chess apps), the white team is located at the bottom of the chess board, and the black team is located at the top. In Sepia, we can't really choose our team color (at least I haven't figured out how to yet), so we instead have the red (black) and the green (white) teams. If you ever forget which team is which, in chess the white team always makes the first move. In our Sepia chess engine, pawns are represented as "p" units, rooks (also sometimes called castles) are represented as "R" units, knights are represented as "N" (the "K" stands for King), "B" stands for Bishop, "Q" for Queen, and "K" for King.

It may look like the chess board is bigger than a standard 8x8, however the border of the map does not contain playable squares. Instead, the border squares are there to ensure there is always a path to move units from square A to square B (without asking other units to get out of the way). So even though you may see pieces moving along the border, those squares are only used to help get from a valid square to another valid square, and pieces will not stop at squares on the border.

3. Information on the provided files

- Directory `src/pas/chess/agents/` contains two .java files: one called `MinimaxAgent.java`, and the other called `AlphaBetaAgent.java`. `MinimaxAgent.java` already has a correct minimax implementation contained within it, but I want you to write a correct implementation of Alpha-Beta pruning in the `AlphaBetaAgent.java` file.

- Directory `src/pas/chess/heuristics` contains two .java files: one called `DefaultHeuristics.java`, and the other called `CustomHeuristics.java`. `DefaultHeuristics.java` contain an example of some simple heuristics that you can use for inspiration when implementing your own in `CustomHeuristics.java`. Remember, your heuristic function should do two things:

  1. Convert a chess Game (i.e. state of the board, current player, time remaining for each player, etc.) into a number. The larger the number, the better the chess game is for **you** (the MAX player). Remember, the heurstic function should always measure how "good" a chess game is from **your** perspective regardless of whose turn it is to actually play. The quality of your heuristics will directly translate into the quality of chess that your agent plays!
  2. Run as efficiently as possible. Minimax and Alpha-Beta pruning are already expensive algorithms, and we use our heuristics to rank leaf nodes (in our game tree) when those leaf nodes are not terminal (i.e. we have to stop searching for some reason). There can be quite a lot of these non-terminal leaf nodes, so we want our heuristics to be as quick as we can!

  Note: the heuristic value you calculate should always be within the bounds of the "true" utility values. In chess, the utility value for you losing is set to `-Double.MAX_VALUE` and the utility value for you winning is set to `+Double.MAX_VALUE`. A tie has a utility value of `0`. So make sure the heuristic values you calculate are within this range!

- Directory `src/pas/chess/moveorder` contains two .java files: one called `DefaultMoveOrderer.java`, and the other called `CustomMoveOrderer.java`. `DefaultMoveOrderer.java` contains an example of a simple move ordering scheme that you can use for inspiration when implementing your own in `CustomMoverOrderer.java`. Remember, Alpha-Beta pruning is only better than Minimax when it **prunes** subtrees. The only way it can prune subtrees is to see better nodes before worse nodes. Your move orderer is responsible for assigning the order in which Alpha-Beta pruning will examine child nodes. So, you want your algorithm to be fast and also good! How quickly Alpha-Beta pruning runs (with respect to Minimax) is directly proportional to the quality of your move orderer.
- Directory `src/pas/chess/instrumentation` contains one .java file `MinimaxAgent.java`. This file will be used to profile how fast your code runs (as compared to the machine your submissions will be running on for the tournament). You should use this information to tune the depth that your `AlphaBetaAgent.java` uses!
- Directory `src/pas/chess/debug` contains one .java file `MinimaxReflectionAgent.java`. This file will be used to test that your alpha-beta pruning algorithm is correct, by asking your alpha-beta pruning algorithm and my minimax algorithm what the "best" move should be (given the same input) for every turn of the game. The utility values of the two nodes should be identical, even if the actions are not. I would recommend running this agent whenever you want to check that your alpha-beta pruning algorithm is correct.
- `chess.jar`. I have done quite a bit of work for you to abstract away as much of the Sepia-ness as I can. Sepia is not meant to play chess, so there is quite a bit of legwork involved to massage Sepia into becoming a chess engine. Most of the work falls into two categories:
  - Converting chess moves into Sepia actions. In chess, we are interested in things like "move the knight from square A to square B." Unfortunately, moves like that involve a bunch of steps that we cannot skim over in Sepia. In Sepia, we cannot teleport units, so we have to submit actions to move the knight one square at a time. We also cannot "jump" units over each other, so we have to find a path in Sepia that will get the knight from square A to square B. The paths that we come up with may look goofy and un-chess-like, but at the end of the day, the knight moves from square A to square B as intended. This problem becomes more complicated when you perform chess moves like promiting a pawn to another piece type, which is harder to do in Sepia.
  - Getting agents to synchronize. Agents in Sepia can act at the same time, which is expressly forbidden in chess. Worse still is that agents in sepia have no mechanism to communicate, which is sorely needed if one agent must wait for X turns while its opponent moves their knight along some convoluted path. Therefore, I needed to invent some (thread-safe) of de-facto communication between agents to have them synchronize their turns in the game.

  If you are curious about how I actually solved these problems, feel free to ask! I spent many hours engineering around problems like this.

Task 1: `AlphaBetaAgent.java` (40 points)

Your first task is to implement `AlphaBetaAgent.AlphaBetaSearcher.alphaBetaSearch`. This method is where you will write your Alpha-Beta pruning implementation. I recommend doing this **first**, because you will know if your Alpha-Beta pruning algorithm is correct when it behaves identically to Minimax (with the same heuristics). The way you can check is by running the `data/pas/chess/debug/ValidateAlphaBetaPruning.xml` game file. This file plays a game between the `RandomAgent` and the `MinimaxReflectionAgent` located in `src/pas/chess/debug/agents/`. As mentioned earlier, this agent will instantiate your `AlphaBetaAgent`, and then play a complete game,

where everytime your `AlphaBetaAgent` is used to calculate a move, the utility value of the root of your expansion is compared against the minimax expansion. This game will take longer than normal (we have to run minimax and alpha-beta pruning in parallel), but if your alpha-beta pruning algorithm returns the same utility values as minimax, then your alpha-beta pruning is correct. `MinimaxReflectionAgent` uses your heuristic function and your move orderer.

Task 2: `CustomHeuristics.java` (30 points)

Once you have your Alpha-Beta pruning algorithm up and running (and performing identically to Minimax), now you can begin trying to play better chess. Your heuristics are responsible for evaluating the "quality" or "goodness" of a chess game and producing a real value. The output of the heuristics should never be less than the true cost of losing the game (which is set to `-Double.MAX_VALUE`) and also should never be more than the true cost of winning the game (which is set automatically to `+Double.MAX_VALUE`). Remember, true costs are set for you, so you don't have to worry about deciding how much does a terminal state cost.

I recommend looking at the `DefaultHeuristics.java` file. Personally, I like to try and group my heuristics into categories like offense, defense, etc. If you like to think similarly, I encourage you to also group your heuristics. Your heuristics are where you can employ notions of strategy, which surprisingly don't have to be very chess-specific. You don't need to know how to play chess in order to know that putting your pieces in good positions is good for you (like threatening other pieces or controlling lots of squares on the board) and putting pieces in bad positions is bad for you (like putting them in squares where they will get taken by the enemy).

Task 3: `CustomMoveOrderer.java` (30 points)

Now that you have your Alpha-Beta pruning algorithm running and it's playing chess in a manner that suits you, its time to try and make it as fast as you can. While we can always try to make our algorithms faster, one big speed boost here is to encourage Alpha-Beta pruning to prune as much as possible. The only way Alpha-Beta pruning will decide to prune a subtree is if it knows that a subtree will never be picked by the players (i.e. there is a better choice already known for that player). So, we can control the order in which Alpha-Beta pruning examines subtrees: and we should choose an ordering that we believe will cause Alpha-Beta pruning to prune as much as possible.

Your move orderer imposes a scheme on the order of child nodes. This does not have to be a sorting algorithm in a conventional sense, and I would recommend against calculating the heuristic value of each child and sorting based on that (expensive!). For instance, if we believe that capturing enemy pieces is always good, then we should let Alpha-Beta pruning examine all child nodes that have capture moves inside them before anything else.

Notes

Do **NOT** change any of the arguments passed into any of the agents (the `<Argument></Argument>` tags) unless you know what you are doing (or ask beforehand)!

Task 4: Extra Credit (50 points)

Inside the jarfile there are a few agents that are provided for you to play against. The one you probably will play the most against is the `RandomAgent` as you develop your solution. However, there are two others called `ReallyHardAgent.java` and `ImpossibleAgent.java`. As their names suggest, these agents play really good chess. You will earn 40/50 points if you can beat the `ReallyGoodAgent`

and you will earn full extra credit if you can beat the `ImpossibleAgent`. I have included extra files in `data/pas/chess/` for this purpose: to pit your `AlphaBetaAgent` against these two.

Task 5: Submitting your assignment

Please submit `AlphaBetaAgent.java` as well as your `CustomHeuristics.java` and `CustomMoveOrderer.java` files on gradescope (dragging and dropping them is fine)!

Task 6: Tournament Eligibility

In order for your submission to be eligible in the tournament, your submission must satisfy all of the following requirements:

- Your submission must be on time.
- You do not get an extension for this assignment.
- Your agent compiles on the autograder.
- Your agent can beat the `RandomAgent.java` more than 50% of the time (out of 10 trials) in timed (10min) chess games.