Lab Assignment 3: Recursive Tic-Tac-Toe

Due: Friday 10/03/2024 @ 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. We will not be using Sepia for this assignment: I have developed a game engine for us to use. In this assignment we will be writing agents to play a Recursive Tic-Tac-Toe (also called Ultimate Tic-Tac-Toe) game.

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/lab3/lib/rttt-X.X.X.jar` to `cs440/lib/rttt-X.X.X.jar`.
  This file is the custom jarfile that I created for you.
- Copy `Downloads/lab3/lib/argparse4j-0.9.0.jar` to `cs440/lib/argparse4j-0.9.0.jar`.
  This is a jarfile that `rttt-X.X.X.jar` depends on. It provides similar functionality to Python's `argparse` module. The documentation for `argparse4j` can be found here.
- Copy `Downloads/lab3/lib/junit-4.12.jar` to `cs440/lib/junit-4.12.jar`.
  This is a jarfile that you can use along with `hamcrest-2.2.jar` to write your own unit testing code. I would **highly recommend** writing your own tests for this (and future) assignments.
- Copy `Downloads/lab3/lib/hamcrest-2.2.jar` to `cs440/lib/hamcrest-2.2.jar`.
  This is a jarfile that you can use along with `junit-4.12.jar` to write your own unit testing code. I would **highly recommend** writing your own tests for this (and future) assignments.
- Copy `Downloads/lab3/src` to `cs440/src`.
  This directory contains our source code `.java` files.
- Copy `Downloads/lab3/rttt.srcs` to `cs440/rttt.srcs`.
  This file contains the paths to the `.java` files we are working with in this assignment. Just like in the past, 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/lab3/doc/labs` to `cs440/doc/labs`. This is the documentation generated from `rttt-X.X.X.jar` and will be extremely useful in this assignment. After copying, if you double-click on `cs440/doc/labs/rttt/index.html`, the documentation should open in your browser.

2. Test run

If your setup is correct, you should be able to compile and execute the following code. A window should appear:

```
# Mac, Linux. Run from the cs440 directory.
javac -cp "./lib/*:." @rttt.srcs
java -cp "./lib/*:." edu.bu.labs.rttt.Main

# Windows. Run from the cs440 directory.
javac -cp "./lib/*;." @rttt.srcs
java -cp "./lib/*;." edu.bu.labs.rttt.Main
```

NOTE: The commands above will **not** run your code. There are several command line arguments you can provide, and two of them specifiy which agent controls which player (e.g. player "X" or player "O"). This argument to control player "O" looks like this:

```
java -cp "./lib/*:." edu.bu.labs.rttt.Main -o <YOUR_AGENT_CLASSPATH>
```

The argument to control player "X" uses a `-x` instead of a `-o`.

You are implementing multiple agents in this assignment. All of them will have the base of their classpath be `src.labs.rttt.agents.`. For instance, your `DepthThresholdedMinimaxAgent` will have classpath `src.labs.rttt.agents.DepthThresholdedMinimaxAgent`. You can alternatively use `--agentO` instead of `-o` if you wish (and also `--agentX` instead of `-x`). Remember that if you want to see the other command line arguments available you need to add a `-h` or `--help` to the end of your command!

Task 3: `DepthThresholdedMinimaxAgent.java` (25 points)

Our first goal is to implement the minimax algorithm (discussed in class). However, we have to limit the size of the tree even for toy(-ish) games like Recursive Tic Tac Toe. For instance, assuming you always know which of the inner boards you have to play on, there are at max 9 possible moves, and 81 potential moves to be made on the board. So the number of nodes in the game tree is at most 9^{81}. We can probably get a tighter bound by accounting for the branching factor: it is very unlikely that you *always* have 9 moves at your disposal, but this looser bound illustrates the point. When I implemented vanilla minimax, I killed it before it could make its first move after waiting 2.5 hours. So, you will be implementing **depth thresholded** minimax.

As discussed in class, depth thresholded minimax is controlled by a `max depth` hyperparameter d_{max}. When expanding the game tree, if we ever reach a depth of d_{max}, we are forced to arbitrarily give up our search and backtrack; even if the current node is not a terminal node. As a result, we need a way of guessing the utility value we **would** have gotten had we the budget to expand the tree all the way out. This heuristic I have already implemented for you in `src/labs/rttt/heuristics/Heuristics.java`. This heuristic counts the number of inner boards that you've won and awards 10 points per win. It also counts the number of inner boards your opponent has won and deducts 10 points per loss. This heuristic is not very good: we would probably want to design a more granular heuristic, but I want you to use it for now. With this heuristic I was able to beat the random agent about 94% of the time with $d_{max} = 3$.

I have provided a starter file `src/labs/rttt/DepthThresholdedMinimaxAgent.java`. In this file, all I want you to do is complete the `minimax` method. This method should return a `edu.bu.labs.rttt.traversal.Node` object, which we are using as a container to store bookkeeping information like the current state of the game, the utility value of this state, the current player, etc. The most important piece of information stored inside a node is the **move** that **got us to this state**. So when you look at a `Node`, you can retrieve the move that led to the state also stored inside that `Node`. Be sure to use the heuristic value to set the utility value when the depth limit is reach and the node is **not** a terminal node. I have implemented the `Node` class to set its utility when its terminal. This utility is defined as:

- +100 if you won the game.
- -100 if you lost the game.
- +0 if you tied.

Note that the heuristic I provided never exceeds the bounds put by the terminal utility nor does it produce values equal to the extrema of the utility (e.g. the heuristic is always bounded between $(-100, +100)$ exclusive).

Task 4: `DepthThresholdedAlphaBetaAgent.java` (25 points)

In the file `src/labs/rttt/DepthThresholdedAlphaBetaAgent.java` I want you to complete the `alphaBeta` method. This method is where you should implement the alpha-beta pruning algorithm discussed in class. This algorithm will look very similar to your `minimax` method from the `DepthThresholdedMinimaxAgent` class and thats ok! Remember, alpha-beta pruning **is** minimax just with some tree-pruning logic added. In the worst case alpha-beta pruning does exactly the same computation as minimax and expands the exact same tree to the exact same granularity as minimax. In order to control the amount of pruning that alpha-beta pruning does, we have to design an ordering scheme for the actions (so we can see better actions before worse ones and then prune). I have provided a default move ordering scheme in `src/labs/rttt/ordering/MoveOrderer.java`. This is a great place for you to try and impose your own ordering scheme: you probably have opinions about what actions are better (in some situations), what actions are worse (in other situations), etc. Try to place actions you think are better earlier in the order. The better you do, the faster your agent should think (and therefore you can increase the depth limit).

Task 5: Extra Credit (25 points)

Now change the heuristics to make them better. You will have to implement the heuristics using `static` methods and they all have to exist within the `Heuristics.java` file. I will award full extra credit depending on how you do against an actual adversary. I will provide a separate gradescope link for you to submit your files to where your agent will face off against mine. We will play 50 games against each other: half of those games you will go first, and the other half I will go first. Let's see what your winrate looks like!

To be clear, your goal is for your `DepthThresholdedAlphaBetaAgent` too expand as much of the game tree as possible. The only way for this to happen is if you control the ordering that your agent uses to iterate through the children. Remember, the closer "better" children are to the front of the ordering, the more the alpha-beta pruning algorithm will prune. The more tree the algorithm prunes, the bigger the tree can be that the algorithm searches through in the same amount of time. If you develop an effective move orderer, you can increase the max depth of your agent and see farther into the future. Remember not to time out the autograder! Also note that seeing farther into the future will help "make up" for ineffective heuristics, but it does not replace them entirely.

Task 6: Submitting Your Assignment

Please drag and drop all of your files `DepthThresholdedMinimaxAgent.java`, `DepthThresholdedAlphaBetaAgent.java`, `Heuristics.java` and `MoveOrderer.java` on gradescope. There will be default versions of `Heuristics.java` and `MoveOrderer.java` provided on the autograder in case you don't submit those. There will be multiple autograders, so be sure to submit to them all!