EECS 391: Introduction to AI (Spring 2020) Programming Exercise 3

General Instructions:

Please comment your code extensively so we can understand it, write efficient code using good data structures and sensible variable names. We will use the git logs to ensure that each person is contributing equally to all exercises. Therefore, any code written by you must be clearly logged under your own name/network ID by git. You will not receive credit for code committed by anyone other than yourself even if you wrote it, or if we cannot make out who committed the code. **There will be NO exceptions to this rule.**

The redmine system used by csevcs has issues with some Unicode characters. Please ensure your git names use only standard ASCII characters.

Your commits are due on csevcs.case.edu by 11:59pm on the due date specified after the question. You will receive a 10% bonus for any solution turned in a week or more in advance of the due date. You must notify us of such a commit by email. You can use one late day each week (up to Saturday 11:59pm) with a penalty of 20%. Submissions after Saturday 11:59pm for any week will not be graded. Other bonus points may be awarded for exceptionally well-written and commented, easy to read, clean and efficient code.

These programming exercises will use SEPIA ("Strategy Engine for Programming Intelligent Agents"), a game environment written by other CWRU students tailored to writing AI players. SEPIA is a real-time strategy (RTS) game (though we will not use the real-time aspects in these exercises). RTS games generally have "economic" and "battle" components. In the economic game, the goal is to collect different types of resources in the map. Typical resources are "Gold" and "Wood." Resources are collected using units called "Peasants." Having resources allows the player to build other buildings (Peasants can be used to build things) and produce more units. Some of these units are battle units that can be used to fight the opponent. Games generally end when one player has no more units left; however, in SEPIA, a variety of victory conditions can be declared through XML configuration files. For example we can declare a victory condition to be when a certain amount of Gold and Wood have been collected, some number of units of a certain type built, etc.

You need Java 1.8 to run SEPIA. Note that, although the components of SEPIA you will use have been fairly well tested by now, there is still a possibility of bugs remaining. If you encounter behavior that seems strange, please let me or the TAs know.

Programming 3: Playing against an Opponent (First Commit 2/28 (50 points), Final Commit 3/6 (50 points))

The data for this exercise is in the ProgrammingExercise3.zip file on Canvas.

In this exercise, you will implement the alpha-beta algorithm for playing two player games to solve some SEPIA scenarios.

Provided are three maps and config files in data/, an opponent agent in archer_agent/ and skeleton files for your agent in src/. The archer agent is not part of any package, so place it at the root of your class hierarchy. The Minimax agent is part of the edu.cwru.sepia.agent.minimax package. The maps have two Footmen belonging to player 0 and one or two Archers belonging to player 1. Footmen are melee units and have to be adjacent to another unit to attack it, and they have lots of health. Archers are ranged units that attack at a distance. They do lots of damage but have little health. In these scenarios, your agent will control the Footmen while the provided agent, ArcherAgent, will control the Archers. The scenario will end when all the units belonging to one player are killed. So your goal is to write an agent that will quickly use the Footmen to destroy the Archers. However, these Archers will react to the Footmen and try to outmaneuver them and kill them if they can. You will use game trees to figure out what your Footmen should do.

Your agent should take one parameter as input. This is an integer that specifies the depth of the game tree in plys to look ahead. This is specified in the configuration XML file as the "Argument" parameter under the Minimax agent. At each level, the possible moves are the joint moves of both Footmen, and the joint moves of the Archers (if more than one). For this assignment, assume that the only possible actions of each Footman are to move up, down, left, right and attack if next to the Archer(s). The Archers have the same set of actions: move up, down, left right and attack (which means they stay where they are). Thus when your agent is playing, there are 16 joint actions for the two Footmen you control (if you are next to an Archer, you also have the Attack action). When ArcherAgent is playing, it has either 5 or 25 (joint) actions depending on whether there are one or two Archers. You can see that even for this simple setting, the game tree is very large!

Implement alpha-beta search to search the game tree up to the specified depth. Use linear evaluation functions to estimate the utilities of the states at the leaves of the game tree. To get these, you will need state features; use whatever state features you can think of that you think correlate with the goodness of a state. A state should have high utility if it is likely you will shortly trap and kill the Archer(s) from that state.

As part of your implementation, you will need to track how the state changes as you take actions. You should write your own simple state tracker for this. *Don't* use SEPIA's state cloning functions; they will be inefficient and may introduce additional problematic issues.

Since the game tree is very large, the order of node expansion is critical. Use heuristics to determine good node orderings. For example, at a Footman level in the game tree, actions that move the Footmen away from the Archers are almost always guaranteed to have low utility and so should be expanded last. If adjacent to an Archer, a Footman should always attack. Similarly, if the Archer(s) is (are) very far away from your Footmen, they will not run but shoot your Footmen, so expand that action first, and so forth.

We will award up to 10 bonus points for well written code that is able to quickly search a large number of plies relative to the rest of the class (e.g. if you are in the top three runtimes to finish a scenario with a fixed large number of plies). Note that we will test your code with other maps than the ones provided with this assignment.

In the skeleton agent files, MinimaxAlphaBeta is the main class. It includes the main alphaBetaSearch method and a node reordering method (orderChildrenWithHeuristics). These are the methods you will fill in. A helper class, GameState, is provided to track SEPIA's state, which can then be used to compute the utility (getUtility) and to find the possible results after taking an action from a state (getChildren). You will fill this in as well. You should not modify the GameStateChild class. It just pairs an action map with a GameState.

The code is structured in this way so that your alphaBetaSearch method can be implemented abstractly (i.e. it will not need to contain any SEPIA specific code). The SEPIA related code should reside in GameState. You can add fields and functions to GameState as needed (add comments to explain what they are doing).

Your git repository is located at https://csevcs.case.edu/git/2020_spring_391_N, where N is your Canvas group number. **Because this is a long assignment, you are expected to make intermediate weekly commits to the repository.** Each week, we expect to see substantial progress made in the implementation by each person. Progress must be *proportional to the points allocated that week*. Create a separate subdirectory "P3agents" in your git repository, place your agent in it and push to csevcs. Include a short README file containing (i) a summary of the work done for the week and (ii) your experience with the API and documentation, and anything you found confusing. Do **NOT** commit any files other than the java files containing your agent and the README.