EECS 391: Introduction to AI (Spring 2016) Programming Assignment 4 (Max Points: 100)

Assigned Friday April 8, due midnight Sunday April 24. Turn in your code using blackboard. Please comment your code extensively so we can understand it, and use sensible variable names. If two people worked on a submission, please include both your names and IDs in a README. You may only turn in an assignment as a pair if both of you have contributed equally to it.

In this assignment you will write a reinforcement learning agent to learn to fight a tactical battle situation in SEPIA. Download the PA4.zip file from the website.

1. Problem Setup

The scenario you will solve is built around the "rl_5fv5f.xml" and the "rl_10fv10f.xml" maps in the zip file and the associated configuration files. In these maps, there are 5 or 10 "Footmen" for each side. Footmen are melee units. Each footman has a fixed amount of "hitpoints." When hit, it loses some hitpoints. If the current hitpoints reach zero, it dies. Your agent will control one side and the provided combatAgent will control the other side. Your reinforcement learning agent's goal is (obviously) to learn a policy to win the battle---i.e., to kill the enemy footmen while losing as few footmen of its own as possible. Note that, unlike in planning, there is no separate "offline" component---the agent will learn by interacting directly with the environment and repeatedly playing the scenario. Also observe that, in this situation, an accurate model is not easy to specify ahead of time, so planning techniques are problematic to implement here.

The *parameterized actions* available to your agent will be of the form "Attack(F,E)," where F is a friendly footman and E is an enemy footman. At each *event point*, you will loop through all living friendly footmen and reallocate targets to each one. An event is a "significant" state change, such as when a friendly unit gets hit or a target is killed. You are free to define your own set of events. Note that if you make the events too fine grained, you will have a lot of decision points with very long times between feedback (rewards), which will make the decision making problem much harder. On the other hand, if they are too coarse, your policy will be suboptimal because you will not react to changes quickly enough during the battle.

When an action Attack(F=f, E=e) has been selected for footman f, it must be executed in SEPIA. Note that such an action is a composite action that involves moving to e and then attacking e. You can use the built-in SEPIA compound attacks to handle this, but be careful of pathfinding issues in close quarters.

The reward function should be set up as follows. Each action costs the agent− 0.1. If a friendly footman hits an enemy for d damage, the agent gets a reward of +d. If a friendly footman gets hit for d damage, the agent gets a penalty of –d. If an enemy footman dies, the agent gets a reward of +100. If a friendly footman dies, the agent gets a penalty of −100.

2. Q-learning with function approximation (100 points)

Implement the Q-learning algorithm with linear function approximation (chapter 21.3.2-21.4 in the book) to solve this problem. The Q(s,a) function will be defined as w*f(s,a)+w0, where w is a vector of learned

weights and f(s,a) is a vector of state-action features derived from the primitive state. For example, a feature might be: “Is e my closest enemy in terms of Chebyshev distance?” (Note that the enemy e is part of the action.) You should write your own set of features to use. Think about features that will help the agent to come up with good policies. Some useful features are “coordination” features such as “How many other footmen are currently attacking e?”. Some others are “reflective” features such as “Is e an enemy that is currently attacking me?” Yet other features could be things like “What is the ratio of the hitpoints of e to me?”

All of the friendly footmen will *share* the same Q-function. Thus this same function will be updated whenever any unit gets feedback, and will be consulted to determine the action of every unit. This is OK because all the units have identical capabilities and prevents combinatorial explosion due to multiple units. Note that each footman still senses the “global” state, and “knows” what the other friendly footmen are doing. This is an example of “central control.” Consult the book and the slides to see the update rules for learning the weights w. Note that in order to do the update properly, you need to “decompose” the rewards on a per-footman basis. Fortunately this is easy to do in this case as this is how I have specified the reward signal. (In general the reward signal would just be a function of the joint state and action.)

One other issue to note is that based on your event definitions, the time between successive decision points may vary. (You could of course also decide to have a “dummy event” always happen every 10 steps, say.) In this case you still need to track the accumulated reward over the intermediate time steps and discount them suitably when performing the Q-update.

Your agent should take the number of *episodes* to play as an option. Each episode is a complete battle up to a victory or defeat. Given this option, the agent should play the number of specified episodes against the opponent. Every 10 episodes, it should freeze its current policy (Q-function) and play another 5 “evaluation” episodes against the opponent (during which the Q function is used to select actions, but not updated). Then it should produce the following output:

Games Played	Average Cumulative Reward
0	xyz
10	xyz
20	xyz
30	xyz
...	

Each “xyz” is the average (undiscounted) cumulative reward obtained by the current policy/Q-function after 0, 10, 20 etc. games played, averaged over the five evaluation games. This gives you an idea of the rate at which the agent is learning and can be plotted as a learning curve.

Important note: You can fix the PRNG seed for this agent right at the start to 12345 to ensure repeatability, however, be careful not to reset the seed at the start of each episode! It is important to let each episode play out differently for the policy to improve.

Output

Generate learning curves for your agent, one for each of the scenarios, with the x-axis going up to at least 10,000 episodes (you can do more if you wish). The y-axis is the average cumulative reward. Average each curve over five runs, that is, run the Q-learner five times so that you get five curves for each scenario, and then average the curves. In order to run these experiments, you should disable VisualAgent. The instructions to do this are in the config files. Finally, save the weights corresponding to your best found policy in a file called "bestweights.data".

We may run your agents against each other and award up to a 10 point bonus to agents that perform well. (You can run competitions like this as well if you can convince someone to share their agent with you! Put the other agent in an appropriate location and modify the config files to load it instead of the provided combat agent.)

Code and Data Structures

The provided RLAgent.java has several functions pre-written for you, as well as the facility to save and load the learned policy weights. It also has comments explaining what to do in each function and what SEPIA functions you may need. Note that this agent takes two parameters; the second is a Boolean value that causes weights to be loaded from a file for the initial policy.

What to turn in

Prepare a ZIP file with RLAgent.java, saved weights from your best policy in "bestweights.data," pdfs showing the averaged learning curves and a README (do NOT include any class files, or any SEPIA code). Include your name(s) and ID(s) in the README along with any other comments you have. Name your file as "yournetworkID(s)_PA4.zip" and use Blackboard to submit it. If you worked as a pair, ensure that both your IDs appear in your submission. Only one submission is required if you worked as a pair.