CSC 4240 – Introduction to Artificial Intelligence Fall 2021
Undergraduate Team Project: Robot Defense

DUE: December 6th at NOON (notice the day and time!)

TEAMS: minimum 3/maximum 4 students per team (total limit of 8 teams)

SUBMISSION: ALL materials through the iLearn class-website dropbox

PRESENTATION/COMPETITION: December 7th

Installation, Files and Startup

Begin by downloading the RobotDefense archive from the course iLearn website. (You'll need Java 1.6.2 or above.) You can run the game by first building the sample agent: *javac -cp rd881.jar *.java* , and then running the jar itself*: java -jar rd881.ja*r. (NOTE: If you don't have *javac*, you will need to download the corresponding Java jdk.) Upon start up, you will see a new level with vacuum towers that can be controlled either by a human or agent player. The goal of the game is to capture insects with the vacuums before the insects reach their destination.

Controls

As a human, you can select elements on the map by left clicking. Once a vacuum tower is selected, you can *turn it on* using keys a, s d and f that corresponds to power levels 1-4 respectively. Right clicking on the tower once it is selected will turn it toward the mouse cursor. Hitting the space bar while a vacuum is selected will turn that tower off. Hitting c will begin the flow of insects (essentially it will start the game). Insects will spawn either when 2 have left the map, or after a fixed delay—whichever comes first. You can control the time based spawn methods with a command line flag: -t <delay in seconds>. Hitting r will toggle your agent on and off.

In the upper right corner of the screen, you'll see three displays. At the top you'll see a counter with a decreasing value (beginning at 10000). This is the amount of resources you have left. Resources decrease slowly over time, but decrease more quickly when vacuums are turned on and when their power is higher. Immediately below that you'll see two insect icons and corresponding values, the upper value indicates how many insects you have captured while the lower value indicates how many have escaped. The goal here is to capture as many as possible. The initial resource allocation can also be controlled with a command line flag: -c <initial resources>. When the resource count reaches 0, the game is over.

The Game

Here's how it works. The vacuum towers (once on) will create air currents on adjacent tiles (indicated with colored circles). The insect's path will be affected by these air currents based on the current aero dynamic function. The default setting (and the one you should use for this assignment) is *InsectsMappedToFunctions*. You can view the learning problem here as one of trying to determine how to create the air currents that are most likely to capture each insect.

Setup

Begin by renaming *LearnerOne* and the associated *LearnerOneFactory* as <*YourName*>*Agent* and <*YourName*>*Factory* respectively. Next, modify the *services/jig.misc.rd.ai.AgentFactory* file in META-INF so your agent will be discovered. Now you should be able to run the game with your new agent. Note that the levels folder contains a few different levels. To load one, copy it and change its name to learning.dat. The *simple-4pack.dat* level will be particularly helpful for testing.

Where to Start

The *LearnerOne* agent uses one state-action table to control all the towers on the map. The state is encoded by the *StateVector* and *CellContents* classes and action-value pairs are stored in the inner *QMap* classes. When you reach a point where you want to change the agent's internal representation for state, you will want to take a close look at the *StateVector* and *CellContents* classes. In particular *StateVector.RADIUS* and *CellContents.getContentsCode* will be worth looking at.

Assignment Problems

Each of the questions below will require some coding and some written explanation. **Place the written explanation in a readme file and be sure to turn that in with your code**. Full credit for this assignment is 100 points, but bonus points are possible!

1. Play the game a bit by hand so you can determine what knowledge may be important and what actions are relevant. For the four parts below, don't modify the *StateVector* or *CellContents* classes.

(a) (10 points) Easy: Modify the agent so it can use learning to turn off the vacuums when they're not necessary. You will need to ensure that the agent periodically considers changing its action even if the local state does not change. In your readme, describe your approach.

(b) (10 points) Easy: Modify the agent so it considers a wider variety of actions (at least include power settings 2 and 4 which are critical for success).

(c) (10 points) Easy: Compare your agent's performance against *LearnerOne* when run from the command line with java -jar rd881.jar -c 5000 -t 10 (note the game is stochastic so results will be somewhat varied). Comment on the results. Do you think they would be substantially different if the agents had more time to run? Why or why not?

(d) (10 points) Easy to Medium: Modify the reward function so that the agent performs well on the test maps *simple1.dat* and then on the *simple-4pack.dat* (found in the levels directory). Describe the functions you tested, their effects, and the motivation for testing them.

2. The current state representation indicates how many insects are in which neighboring cells. When an insect changes its grid location, the state changes. Sometimes the *LearnerOne* agent is making progress toward capturing an insect, but then "gives up" before it succeeds because the state has changed.

(a) (10 points) Easy: Change the method of tie breaking for equally valued actions so that the agent is more likely to keep doing the same action if two or more have the same action value.

(b) (20 points) Medium-Hard: Implement a more interesting form of *reinforcement learning* (such as Q-learning) that takes into account both the agent's current state (and its reward) as well as the successor state and its reward when calculating action values. Describe your approach and any perceived impacts on the system.

NOTE: This may require you to read (in more detail than maybe you did already) Chapter 22, "Reinforcement Learning", in the Russell and Norvig textbook. For instance, the Q-learning algorithm can be found in this chapter.

3. In this game as in all other learning problems, efficient use of knowledge is important to maintain a high learning rate. When the state space is large, there is a lot of exploring to do.

(a) (10 points) Medium: Modify your agent to perform directed reasoning by using information from similar, related, states that it has partially explored. Describe your approach in the readme.

(b) (20 points) Easy to Hard: Modify the *CellContents* class (and possibly the *StateVector* class) so that the state representation used by your learning agent is more sophisticated and more useful. Describe your changes, and the motivation behind it in your readme. Show that it improves the agent's performance at least in the long term by comparing it against the *LearnerOne* agent and plotting the data stored in performance.log. Feel free to change the initial resource allocation to illustrate the agent's prowess. Simple enhancements or enhancements that do not improve performance significantly will be given approximately 10 points, while sophisticated and useful enhancements will earn full credit. It may pay to try this in stages so your state representation doesn't get out of hand.

This brings the possible number of points to 100 points.

In addition, you earn a bonus by **implementing the learner that captures the most insects**! In class, each project will be run using the same starting environment. Whichever one captures the most insects, will receive 10 bonus points – bringing a possible total to 110.

Have fun!