Project 2
Computer Science 4013: Artificial Intelligence
Due 12:01AM March 28, 2008
Note: NOT the beginning of class

Introduction

This project will use evolutionary computation to intelligently choose from a set of high-level behaviors. Your A* agent will be a useful piece of this project. You may also use the provided A* code if you do not want to use your project 1 code.

Since the last project focused entirely on collecting beacons, you may have wondered why you would want to collect those silly beacons! The answer is: now that people can shoot at you, beacons will be useful in maintaining high energy levels and not dying. Each bullet that hits you costs you energy so so collecting beacons will be very useful for living longer! The project 2 ladder will be ranked by rounded kill average, then deaths, then beacons.

Your learning agent will learn to intelligently choose among the following high-level behaviors:

- Move to nearest beacon (AStar code!)
- Move near to another ship's location (presumably to shoot, also uses AStar code)
- Shoot at ship
- Shoot at asteroid (by popular request!)
- Lay a mine (by popular request!)
- Raise shields (by popular request!)
- Move near to nearest asteroid (presumably to follow it, not to ram it, also uses AStar code)
- Flee from asteroid(s) - you tell it how many asteroids to flee from
- Flee from ship(s) - you tell it how many ships to flee from

Your task is to 1) design an appropriate fitness function to aid the agent in choosing among these behaviors and 2) design an encoding for each individual that will enable the agent to choose the best actions under the right circumstances and 3) implement an evolutionary computation approach that will enable your agent to learn to choose intelligently among the high-level behaviors.

For part 1, you should design a fitness function that rewards the agent for performing well and penalizes it for performing badly. You will need to decide what the criteria for performing well and badly are but I suggest that you do not make your fitness function overly complicated. For the fitness function, you should focus on your goals for your agent and not on how the agent will accomplish those goals. For example, remember the example of swinging your feet above a bar that we discussed in class. A good fitness function here tells the agent how successful it is but it does not tell the agent how to accomplish the goal.

Part 2 is likely the most difficult part! You must choose an encoding of an individual such that two individuals can cross-over and create a new individual and the new individual is still viable. For example, as we discussed in class for tic-tac-toe players, you may create a high-level state space and encode a policy of action responses to each state in your individual. Other encodings are quite possible and may work even better! Even if you choose this approach, you will need to design your state space carefully. We have given you a few ideas in the writeup and in our sample agent but you will need to extend these ideas for your own agent.

Once you have completed part 1 and 2, the remaining pieces of an evolutionary computation solution are selection, crossover and mutation. You need to pick approaches to each of these using what we discussed in class. The following article on the web may be helpful:

`http://www.tjhsst.edu/~ai/AI2001/GA.HTM`

The following is a list of ideas for state space features. Note that this representation lacks information about your ship's current beacon count or your opponent's health or ... (the list goes on. You can't represent it all in main memory so the problem must become partially observable!).

- distance to nearest 3 asteroids: near, middle, far
- angle to nearest 3 asteroids: 4 values for each
- distance to nearest beacon: near, middle, far
- angle to nearest beacon: 4 values
- health: low, middle, high

- number of ships in game: 1, 2, 3 or more
- distance to nearest ship (other than self): near, middle, far
- angle to nearest ship (other than self): 4 values

For this assignment, you **SHOULD** implement evolutionary computation. However, if you are interested in reinforcement learning, you could also choose to use this approach. **BEWARE, we will NOT be covering RL in class.** If you choose to implement RL, you should only implement Q-learning or Sarsa-learning. The pseudo-code for SARSA is given in table 6.9 of the RL book and the pseudo-code for Q-learning is given in table 6.12. I will be happy to talk to you about RL outside of class if you choose this route. The RL book can be read online at:
`http://www.cs.ualberta.ca/~sutton/book/the-book.html`.

Extra-credit opportunities

To keep grades within a fair range, all extra credit will be capped at 10 points. This means that no project can receive a grade higher than 110, no matter if they win the ladder, find great exploits, and are very creative.

Wanted dead or alive: exploits in the simulator

We are reasonably certain that you cannot exploit the simulator (say by directly affecting your opponents health or by directly moving yourself to the goal location). However, any project has bugs and we want to know about them! If you find an exploit to the simulator, you can receive extra credit according to the following scale:

- **5 points:** If you find an exploit and report it to us *using the google code bug reporting system*, you can receive 5 points extra credit upon verification of the report.
- **10 points:** If you find an exploit and give us a fix for it, you can receive 10 points extra credit upon verification of both the exploit and the fix.
- **1 or 2 points:** General simulator bugs are much more likely than exploits. Finding a bug and reporting it using the google code bug reporting system can get you 1 point. Fixing the bug and giving us the fix (you can't check it in directly but you can give it to us in the bug report) can get you two points. Both bug and fix must be verified for any extra credit to be awarded.

Competition ladder

The class-wide competition ladder will start on March 13th (2 weeks before the project is due). For this project, each game will be played with several heuristics, your agent, and two of your fellow classmate's agents (looping through all of them in a round robin fashion).

This ladder will be ranked differently than project 1. Players will be ranked first by the average number of kills (rounded to the nearest tenth). Ties will be broken by an agent's number of deaths with fewer deaths being preferred. Further ties will be broken by the number of beacons collected.

The first place player will receive one extra point for each night that that player wins the ladder up to a maximum of 5 points. To win, you must be ranked above the random player. The second place player will receive 1/2 extra point for each night that the player is in second place. No player can receive more than 5 extra credit points from the ladder and points will be distributed down the ladder accordingly should the maximum be reached.

Wanted (alive please): creative individuals

Creativity is highly encouraged! To make this real, there are up to 10 points of extra-credit available for creative solutions. Some ideas here include: function approximation (this environment is perfect for function approximation), reinforcement learning, and hierarchical learning (e.g. learning to choose among the high-level behaviors and learning better low-level behaviors such as improved steering, shooting, or obstacle avoidance).

- Document it in your writeup! I can't very well give extra credit unless I know you did something extra.
- You must still be doing either evolutionary computation or reinforcement learning
- Remember that by being creative I am referring to the algorithm and not to the ability to creatively download code. All project code must be written exclusively by you except for the sample code that we provide.

Implementation details

We have implemented a few of the high-level behaviors using the provided A* search algorithm but you are VERY welcome to use your own code! The agent chooses among the high-level behaviors randomly - you need to add the intelligence!

A learning agent needs to store the learned values somewhere and we have given you a sample agent that uses xml to save the values across games. The agent has this ability already using xml (Xstream). You can choose to use xstream or you can implement your own input/output capability but you should load your knowledge file in initialize() and save it out in shutdown(). The example agent shows how to do this. Also note that we have added an isAlive() call to the ImmutableShip so that your ship will know when it has died.

Those pesky details

1. Update your Spacewar code from project 1. There have been a few API changes since project 1 and your code will not work directly. In addition, you may need to do a clean or else eclipse (or netbeans or whatever you are using to compile/edit) will be confused.

2. Download the *unpublished* code from D2L. Please do not distribute this code as it is intended for this year's class only. It contains solutions to project 1 (AStar code). Install this code in your src directory. When unzipped, it should create a directory called 'nonpublish'. You should use the example RandomGeneticAgent in src/nonpublish/agents to start your project.

3. Run the ant build target 'project 2' to see the sample project 2 agent working. You will need to copy the RandomGeneticAgent to your 4x4 directory and the clientinit.xml in examples/project2. You will need to modify TWO lines in clientinit.xml now. First,

   ```
   <clientClass>nonpublish.agents.RandomGeneticAgent</clientClass>
   ```

   The previous line should point to your 4x4 and your agent's name. If you name your agent, 4x4.MySpacewarAgent then change this package line to that.

   ```
   <data>../../src/nonpublish/agents/knowledge.txt.gz</data>
   ```

   The previous line should be modified ONLY to point at your 4x4. Leave the ../../ part alone as it will be necessary for the ladder. For example, I would make mine read:

   ```
   <data>../../src/my4x4/knowledge.txt.gz</data>
   ```

4. Create a ship that acts intelligently using learning as described above.

5. Ensure that your player runs on linux on the CSN class machines. You can ssh into these machines or go into the lab personally to verify this. Java should be in your path by default. If it is not, java lives in:

   ```
   /opt/java/bin/java
   ```

6. Submit your project on codd.cs.ou.edu using the submit script as described below.

   (a) Log into codd.cs.ou.edu using the account that was created for you for this class. Your username is your 4x4 and your default password is cs#4x4.

   (b) Make sure your working directory contains all the files you want to turn in. Using a similar submit command to the previous projects, you would use the following command to turn in your code.

   ```
   /opt/ai4013/bin/submit CS4013 Project2 *.java clientinit.xml
   ```

   (c) After the project deadline, the above command will not accept submissions. If you want to turn in your project late, use:

   ```
   /opt/ai4013/bin/submit CS4013 Project2Late *.java clientinit.xml
   ```

Point distribution

- 50 points for correctly implementing evolutionary computation. A correct player will choose among the high-level actions intelligently and generally maximize the fitness function. The ship will rarely run into obstacles, other ships, or bullets. It should try to shoot a number of ships and have a long lifetime. It may also collect a fair number of beacons by the end of the game. Sample projects and point distributions are given below:

  - 45 points if there is only one minor mistake. An example of a minor mistake would be incorrectly implementing selection, off-by-one errors in state calculations, incorrectly configuring your configuration files, and other minor coding errors.
  - 40 points if there are several minor mistakes.
  - 35 points if you have one major mistake. An example of a major mistake would be accidentally starting your population over from scratch each generation (rather than using selection), failing to crossover individuals, and other major errors.

  - 25 if there are several major mistakes.
  - 10 points if you implement a learning algorithm other than evolutionary computation or RL that at least moves the agent around the environment in an intelligent manner.
  - 5 points for an agent that at least does something other than random movements.

- 20 points for designing and correctly implementing a useful fitness (or reward if you choose RL) function. This will be graded as follows:

  - 15 points for one minor mistake. An example would be correctly designing a useful fitness function but failing to implement it correctly (such as putting a minus sign in the wrong place or having an off-by-one error).
  - 10 points for several minor mistakes or one major mistake. An example of a major mistake would be designing a fitness function that is not useful (e.g. a constant fitness value is NOT useful).

- 10 points: We will randomly choose from one of the following good coding practices to grade for these 10 points. Note that this will be included on every project. Are your files well commented? Are your variable names descriptive (or are they all i, j, and k)? Do you make good use of classes and methods or is the entire project in one big flat file? This will be graded as follows:

  - 10 points for well commented code, descriptive variables names or making good use of classes and methods
  - 5 points if you have partially commented code, semi-descriptive variable names, or partial use of classes and methods
  - 0 points if you have no comments in your code, variables are obscurely named, or all your code is in a single flat method.

- 20 points for your writeup. A full-credit writeup will describe your choice of fitness function and why you chose that function, your chose of knowledge representation (the individual, the state representation, any actions, etc) and why you chose that representation, your choice of learning parameters (selection, crossover, mutation, and any required parameters to your chosen function), any creativity that you implemented and why it worked or did not work. 10 points of this write up will be given for a learning curve with time (games) on the x-axis and performance on the y-axis. To get all 10

points, this curve should demonstrate actual learning (meaning performance improves over time). 5 points if you can at least give the curve and explain why you think it is not learning. Additional learning curves measuring different forms of performance are welcome.