CS 4365 Artificial Intelligence
Spring 2015
Course Project: Capture The Flag

Preliminary Evaluation due by Friday, April 24
Final Evaluation due by Friday, May 8

0. Download

Fixes/Updates:

- 3/24: we fixed a bug in TestPlaySurface.java (courtesy of Max Hall): all occurrences of ChenRandomAgent were replaced with RandomAgent.
- 3/16: class files are now included in the CTF package.

Get the Java CTF package:
project.zip.

1. Introduction

This project asks you to develop an agent that acts intelligently in an unfamiliar external world. Though the inner workings of agents are simple and easy to understand, complex and intelligent behavior can emerge. If programmed correctly, such an agent can determine an optimal action given its limited information about the state of the external world. Furthermore, a simple, deterministic approach can be combined with other approaches to yield behavioral results that can be astounding. In this project, you will explore these issues.

2. Background

Capture The Flag (CTF) is a popular outdoor education game in which teams of players attempt to capture the flag of the other team. Each team has a base location, and the team's flag is usually located on or near the base. A player captures the flag of the enemy team by stealing the flag from the enemy base and making it back to his/her own home base without being tagged by the enemy. When a player is tagged, the player must return to his/her home base. CTF for humans is best played on interesting wilderness terrain that contains forests, fields, and creeks. CTF for agents is best played in a two-dimensional grid world.

3. The CTF Framework

To facilitate the development of intelligent agents for CTF games, we have developed a Java CTF framework. All code for processing the board state, handling agent moves, and board visualization (GUI) has already been written. Your assignment is to write agent code that can be plugged in to this framework. Because you only need to implement one part of the system, required knowledge of Java is minimal for this project. Anyone who is familiar with Java or C-style syntax and can mimic an example implementation should have no problem. We provide you with a graphical environment which allows you to configure game start states and select agent implementations to use for each team. Once you implement your own Agent subclass, you can easily load your agent into the graphical environment to test its performance against other types of agents, or even against itself.

4. CTF Rules

As mentioned above, agents in this framework play CTF in a two-dimensional grid world. The framework supports only two-team games. The environment is made up of an NxN grid of board spaces, and each space in the grid is either open or blocked by an obstacle. Each board has a home base location for each of the two teams, which are situated in the center of the far east and west edges of the grid. Each board also has a flag for each team which starts out at the team's home base location. Agents start the game aligned vertically with their own base/flag, and as far north and south as possible (if teams have more than two agents, the agents are spread so that at most one agent occupies each grid space). During each round of play, all agents on the board are queried for a move. Possible moves include the following:

- Move North
- Move South
- Move East
- Move West
- Plant Hyperdeadly Proximity Mine
- Do Nothing

If the move specified by an agent is not blocked by an obstacle (or by the edge of the board), the agent is moved to the new position. Play proceeds according to the following rules:

- No obstacles can exist in the far east and west columns of the board.
- Two agents from the same team may not occupy the same board space. If an agent attempts to move into a board space occupied by a teammate, the move will be blocked.
- Two agents from opposing teams CAN move into the same board space. However, if an agent moves into a space occupied by an enemy, it counts as a tag, and both agents return immediately to their starting positions.
- An agent may not "babysit" it's own home base. Thus, if the agent's flag is located on the agent's base, any move onto the base by the agent will be blocked. (Thus, an agent can't sit on the base and automatically tag any enemy that tries to get the flag.) However, agents ARE permitted to move onto their own base if their flag isn't there.
- An agent that moves onto a board space occupied by the enemy team's flag picks up the flag. The enemy flag then follows the agent for subsequent moves.
- An agent may move onto its own home base if it is carrying the enemy flag (seemingly a violation of the babysitting rule above). Such a move garners one point for the agent's team. After such a flag capture, the board resets to its initial state.
- An agent may move onto its own home base if an enemy is sitting on the base with the agent's flag (to tag the enemy). This will prevent certain stall tactics.
- If an agent is tagged by an enemy agent while carrying the enemy flag, the enemy flag is returned immediately to the enemy base, and the agents both return to their starting positions, as above.
- A proximity mine becomes active exactly one time step after it is planted by an agent.
- If an agent moves onto a proximity mine (or remains on a mine one time step after planting it), it returns immediately to its starting position, and the mine is removed from the grid.

To make the agent move process deterministic and fair, during each time step every agent makes a move, but agents' moves are processed in alternating team order. Thus, if team A={ A1, A2 } and team B={ B1, B2 }, one possible ordering of agent move processing would be A1, B1, A2, B2. Between time steps, teams alternate being the first to have a move processed.

5. Writing Agent Code

We have written an abstract Agent class. For those unfamiliar with Java or object-oriented programming, an abstract class is one that only defines interfaces for certain methods (and doesn't define bodies for these methods). An abstract class is not meant to be instantiated directly (since calling an un-implemented method doesn't make sense), but is meant to be subclassed. Thus, to write code for your agent, you should create a class that extends Agent. For example:

```
public class ChenSimpleAgent extends Agent {
```

The Agent class contains only one abstract method, which means that to implement your agent, you only need to implement one method. The abstract method as declared in Agent is as follows:

```
public abstract int getMove( AgentEnvironment inEnvironment );
```

Thus, in our ChenSimpleAgent class, we can implement this method as follows:

```
public int getMove( AgentEnvironment inEnvironment ) {
        ... our code here ...
        }
```

The getMove method takes an AgentEnvironment object and must return an integer that represents the move the agent is trying to make (one of the six moves listed above, see integer constants defined in AgentAction in the project javadoc). An AgentEnvironment object contains all the information an agent can access about the environment state. The information is agent-centric, in that the point of reference is the agent's location. All information about the environment is obtained from methods in AgentEnvironment that return boolean values describing whether a particular property is true or false. For example, the "isFlagNorth" method returns true if and only if the flag is located to the north of the agent. For each time step, your agent's getMove method will be called, and your agent will be passed a new AgentEnvironment describing the new environment state. Note that the information provided by AgentEnvironment is quite limited in that it only provides cardinal direction (N, S, E, W) information about various board features. Also, no methods in AgentEnvironment deal with proximity mines, which effectively means the mines are invisible for all agents. To browse a complete list of methods available in AgentEnvironment, see the project javadoc.

You should create only one .java file for this project. You should entitle the file "NetIDAgent.java", where "NetID" is the net-id of one of your team members. Your Java file should contain a java class called "NetIDAgent" that extends Agent as mentioned above. Your class should be in the package "ctf.agent". Classes that your agent can access are in "ctf.common". Thus, if your net-id was cxc107920, you would create a file called "cxc107920Agent.java", and the top few lines of your file could look like:

```
package ctf.agent;

import ctf.common.*;
```

```
public class cxc107920Agent extends Agent {
```

Note that your Agent subclass must also have a parameterless constructor. An easy way to meet this requirement would be to write no constructor at all.

For an example class that does all of this correctly, see "ChenSimpleAgent.java", which is in "ctf/agent". You should save your Java file in the "ctf/agent" directory also. Please DO NOT turn in more than one .java file. Note that Java forbids defining more than one class in each .java file. If you feel that you need to implement other classes besides one that extends Agent (e.g., if you want to write advanced data structures), then you should do this with inner classes (basically a class defined inside the body of your Agent subclass). See the project javadoc for more information about subclassing Agent.

For fairness, please DO NOT perform any IO operations in your agent code. In other words, reading from files is prohibited. To make sure your agent class adheres to this rule, we will be grep'ing all incoming .java files for "java.io" and "java.net" import statements.

6. Loading Agents in the Test Environment

To test your agent code, put your "NetIDAgent.java" file in the ctf/agent directory. To compile the file (because it is in package ctf.agent), change to the directory containing "ctf" and type:

```
javac -cp . ctf/agent/NetIDAgent.java
```

This tells the java compiler to set the user class path to the current directory (the directory containing "ctf") and to compile your .java file.

After compiling, your "NetIDAgent" class will be loaded each time the test environment is launched. However, you need to tell the environment that your class should be listed as one of the available agents. To do this, edit the file "AgentClasses.txt", and add an entry for your class (follow the pattern from the classes that are already listed in the file). Note that you must increment the number at the top of the file so that it reflects the number of classes listed in the file (the file that we have included has three classes listed in it, so the number "3" is on the first line of the file).

7. Running the Test Environment

To launch the environment, change to the directory containing "ctf" and type:

```
java -cp . ctf.environment.TestPlaySurface
```

The board is initialized using default settings, which include a randomized obstacle map. The game starts out paused. The control frame is filled with various options: most of these should be self-explanatory. Note that the board is reset to the game start state every time an option is changed that could affect board state integrity (e.g., switching the obstacle map). When using the "Random" board set, reselecting "Random" from the list will generate a new randomized board state. Playing with the various options should give you a feel for how they work, so we won't explain them any further here.

8. Testing Your Agents

Note that though teams larger than two agents each are supported by the test environment, you will be graded only on how your agents perform in teams of two (situations get very complicated when three or more agents are on each team).

Three Agent subclasses are provided for testing purposes. ChenSimpleAgent, one of your dear TA's agents, is a simplistic agent that naively chases after the enemy flag until it gets it, then naively chases its own home base until it gets there. ChenSimpleAgent doesn't pay attention to any other agents (enemies or teammates), but it does do some rudimentary obstacle navigation (though it gets permanently stuck in certain maps). ChenSimpleAgent doesn't plant any mines. RandomAgent and BomberAgent are two other test agents that don't need any explanation.

Note that you will NOT be graded on your performance against RandomAgent and BomberAgent: they are provided merely for your own amusement (or to convince you why planting lots of mines is a bad idea). You will only be graded on your performance against the TA's simple agent (ChenSimpleAgent), his secret agent (SecretAgent), and your classmates' agents in the tournament.

Five fixed-obstacle maps (10x10) are provided. You will be graded based on how your agent perform on these maps, and also on how they perform on one additional map of undisclosed size and obstacle distribution. The "Random" map setting is provided only to give you an idea of the possibilities, but you will not be graded on how your agents perform on random maps (random maps tend not to be fair to both teams).

Maps are stored in the "sets" directory. If you want to create your own maps, add text files to this directory that mimic the format of the files already there (the first number in the file is the dimension of the board). This directory is scanned automatically whenever the test environment is launched, so your custom map will appear in the drop-down list of board sets.

9. A Few Useful Things to Know

- The number of agents per side running simultaneously in the tournament is fixed. For the tournament, we will be running games with two agents per team.
- When an agent is tagged and returns to starting position, it will NOT be reinstantiated as new object. The position variable associated with the agent is changed, but that's all. However, agents ARE reinstantiated after every flag capture: the entire board is reinstantiated along with all of the agents. Thus, if you're saving local state in your agent class, it will be lost after each flag capture. Static local state won't be lost, of course, but managing static local state can be tricky. Anyway, it's not clear how static local state could be useful.
- Whenever two agents from opposing teams end up on the same board space, both agents are tagged and return to their starting locations. Any other specification would involve assumptions about which agent was "trying" to tag the other: we really shouldn't be ascribing these sorts of internal meta-states to our agents. No agents ever get "credit" for a tag. Agents score points for their team only by capturing the enemy flag.
- There is NO way for agents to get information about the environment/board/board state that isn't provided by the AgentEnvironment class. The point of this project is to get you to design agents that can operate intelligently in limited-information environments. All information that's available to your agent is available in the AgentEnvironment interface. If you don't see particular piece information available in AgentEnvironment, then you should assume that there's no way to get that information directly. To clarify, here is a brief list of information that your agents CANNOT get directly:
    - board size
    - score
    - their own board position
    - whether they have just been tagged or not
    - whether another agent is immediately next to them diagonally

  Note that most of these pieces of information can be obtained indirectly through clever, deductive use of local agent states.

10. Due Dates

You may work individually or in a group of two. There are two due dates (note that late days are not applicable to the course project; in other words, late submissions will not be accepted):

April 24, 2015 (Preliminary Evaluation): Each team should submit to eLearning (1) a single "NetIDAgent.java" file (as described above), and (2) a README listing the names of all group members. Make sure your java file compiles before you turn it in, and make sure that your class is in package ctf.agent (see above). We will be running your agents against ChenSimpleAgent and SecretAgent (the TA's secret agent) on the five provided maps and on one extra, undisclosed map. To prevent these games from going on forever, a step limit will be imposed of 2 * boardSize^2 steps. Thus, on the 10x10 boards, the round will be declared a draw after 200 steps. Draws count as **wins for the TA's agents** (and thus your agent gets no points).

May 8, 2015 (Final Evaluation): The goal of the final evaluation is to give you another chance to improve your agent if you are not happy with how it performs in the preliminary evaluation. Each team should submit to eLearning (1) a single "NetIDAgent.java" file (as described above); (2) a README listing the names of all group members; and (3) a 1-2 page report describing how your agent works (i.e., its strategy). [Note: if you are happy with the agent you submit during the preliminary evaluation, you can simply resubmit the same java file in this second evaluation.] Make sure your java file compiles before you turn it in, and make sure that your class is in package ctf.agent (see above). We will be running your agent against every other agent from the class in a round-robin tournament. Each agent from the class will play each other agent for one round on each of the six maps. A round consists of play up to the first flag capture, with the team capturing the flag winning the round. As before, to prevent these games from going on forever, a step limit will be imposed of 2 * boardSize^2 steps. Thus, on the 10x10 boards, the round will be declared a draw after 200 steps. Draws count as **no points for either team**. We will record how many games your agent wins in total.

11. Grading

The course project is worth 12% of your course grade. The 12 points are distributed as follows.

Preliminary Evaluation: The preliminary evaluation is worth 6 points. As mentioned before, we will perform 12 runs on your agent: 6 runs on the six maps against ChenSimpleAgent and 6 runs on the six maps against SecretAgent. You will earn 0.5 points for each of the 12 games your agent wins.

Final Evaluation: The final evaluation is worth 5 points, awarded based on class-wide tournament ranking. The top team gets all 5 points, the bottom team gets 0 points, and points are distributed linearly for the groups in between.

Report: Your report is worth 1 point.

12. Extras

You can use whatever approach you want to implement your agent. You are free to implement any kind of data structure inside your agent, and you can use any method you can think of for determining the correct move to make in a give situation. If you have knowledge of machine learning, we should mention that learning algorithms might even be possible if you modify the framework to suit your training needs. Of course, this would require some additional work, and for the finished product you would need to hard-code the learned parameters into your agent code (since you can't turn in a modified framework or read anything from file).

If you're feeling artisticly inclined, you can implement a custom icon for your agent. You can do this by overriding the drawIcon method.