Project 2: Learning Robots
Computer Science 4013: Artificial Intelligence
Due 12:01AM March 30, 2010
Note: NOT the beginning of class

Introduction

Now that your bots are moving around intelligently, it's time for learning! For this project, your bots have many more action choices available - specifically through the option cards. Option cards are additions to your robot that you can acquire by ending a TURN (not a register cycle) on a 2-wrench square. If your bot does this, it gets a choice of healing 2 damage or drawing an option card. Option cards can give the robot special powers or special weapons. The specific list of option cards that have currently been implemented is below but please note that you can implement additional option cards and turn the code in to be placed in the repository for extra credit. If a robot is damaged, option cards can be exchanged instead of obtaining one point of damage.

- **Ablative Coat**: This option card enables the robot to absorb a specified amount of damage without actually taking damage. When the amount has been reached, the card is disabled.
- **Extra Memory**: This option card enables the robot to receive extra cards each deal. This can enable a robot to make a better program! This option lasts until the robot discards it.
- **Modified Movements:** This set of option cards enables the robot to modify specific movement cards. One option enables the robot to move 4 instead of 3, one enables the robot to execute a no-op (move 0), and one enables the robot to backup 2 instead of 1.
- **Rear Laser:** This set of option cards enables the robot to shoot a laser from its back as well as its front. The laser can shoot with 1 or 2 points of damage (depending on the card).
- **Double Front Laser:** This option card enables the robot to shoot a laser with 2 beams (causing 2 damage) from its front laser.

The goal for this project is to use learning to make your robot even smarter. Although the goal of touching the most flags in as few steps as possible remains, your score now incorporates kills and deaths (so it behooves you to both stay alive and to kill the other bots):

$$0.5 * \left(\frac{\text{total number of kills}}{\text{total number of deaths}} \right) + 0.5 * \left(\frac{\text{total number of flags}}{\text{total number of games}} \right)$$

Because there are so many ways that you can incorporate learning into your bot's behavior, this project is more open-ended than project 1. If you do want to do something other than the decision-tree approach discussed below, **You must talk to Dr McGovern first!**. Learning methods can be very labor intensive and Dr McGovern wants to ensure that you don't choose a project that can't be completed in time.

For this project, you should learn at least one probability estimation tree based on the bot's behavior. For example, you could train a tree to predict whether a bot will be successful at hitting an opponent robot. Or you could train a tree to see if your moves will take you to where you expected (e.g. did another robot interfere). The possibilities for prediction using trees are quite large! To do this, you will need to write code to save out all possible attributes and data to a data file and then run your agent a number of times to collect data. I suggest using the laddertest to do this as it enables you to run your agent without the GUI repeatedly and quickly. Once you have the data saved, you should write a decision tree learner that grows a tree based on your data. Once you have the tree trained, you should use the results of the tree back in your bot in an intelligent manner. For example, if you trained a tree to predict probability of hitting another robot, you might choose to be aggressive only if you have a high probability of success.

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: bugs or exploits in the simulator

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

- **3 points:** If you find an exploit and report it to us you can receive 3 points extra credit upon verification of the report. Note, we know of two exploits for which you cannot get extra credit as it is already discovered. Since both are both extremely difficult to fix and extremely difficult to implement, I'm not listing them here.
- **6 points:** If you find an exploit and give us a fix for it, you can receive 6 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 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 run every night from the day the project is handed out. Extra credit opportunities start 2 weeks before the project is due. For this project, each game will either be played against each other (in a round robin fashion) or against hard-coded heuristic agents. Players will be ranked by score given above.

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 Flag Collector 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 different learning techniques (e.g. incorporating clustering or using GAs), creating a forest instead of a single tree, incorporating pruning into your tree, etc. If you choose to implement anything that you consider creative, please do the following:

- Document it in your writeup! I can't give extra credit unless I know you did something extra.
- Your agent must still be learning! Come see Dr McGovern for approval of any ideas. I'm not going to stop you from pursuing something you really want to try but I just want to make sure your project is doable.

- 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 your group except for the sample players that we provide.

Option cards

I am sure the class can come up with some great new option card ideas and implementations. The list of all option cards provided in the board game is on D2L and we have only implemented a fraction of these so far. You can either choose to implement some of the remaining option cards or you can make up your own ideas! I know you are creative and will come up with some fun ideas! Any functioning option cards (you MUST include a unit test!) can be emailed to Dr McGovern for incorporation into the repository. Extra credit will be awarded for each such card.

Implementation details

All of your source code must reside in your src/4x4 directory and be in your 4x4 package. You may name your files within this package anything that makes sense to you (remember that we are grading on coding style as well).

Your Project 1 agents will not work in this project due to an API change! However, the changes are relatively minimal and your agents should work very quickly. The biggest changes are:

1. You are now allowed to chose your initial heading at the start of the game and whenever you come back from being dead. This method is called getInitialHeading() and should be implemented in your agent.

2. You can now choose to power down/shutdown your agent. This leaves your agent on the board but clears all its damage. It can be shot or otherwise damaged while shutdown so be careful to shutdown in a safe place! To choose this, you should look at Program.setPowerDownNextTurn and call this from your program while inside endAction. Do not call it from startAction or it will be ignored.

3. You can now receive and use option cards. To make use of this, you will need to implement a RobotListener (this is an interface) that takes the state of the world and tell the simulator whether your bot wants to accept an option card. This is only called when you end a turn on a 2-wrench square. There are also two methods in the RobotListener that tell the simulator whether you want to use an option card given

the state of the world and whether you want to exchange the option card in favor of not being damaged.

As before, the agent also contains a startAction(), endAction(), and initialize() method by default. startAction() is called each time an agent is about to begin a turn and it must return a valid action for the agent to execute. endAction() is called after all agents have ended their actions but before the simulator goes to the next turn. This may be left empty if you have no need for cleaning up after a turn. initialize() is called when an agent is created (but not when it comes back to life from being killed).

Using methods from the Java SDK is acceptable (and encouraged as there are some nice built-in graph classes and a PriorityQueue class) but downloading or using code from any other sources is not allowed. See the syllabus for more details on what is considered academic misconduct. As discussed below, any additional files you create should be turned in along with your main agent class.

Competing heuristics

For this project, you may play against the following heuristics:

- **Random** makes completely random moves. Not very bright and hardly ever reaches a flag! Dies frequently but can get in your way easily.
- **FlagCollector** is our naive flag collecting agent (it just aims straight for them). It does a decent job of aiming right for the flag but it gets stuck in many situations.
- **Amy's A* agent** I will put my A* agent in the competition for a better heuristic. Note that this agent does not look at kills or hits and only goes for the flag (but it does so better than Flag Collector!). This code will also be available to you (see below for details).
- **Not So Random:** Dan and Sam have kindly agreed to release their A* agent which is MUCH faster than mine! It will be on D2L (see below).

Those pesky details

1. Update your Roborally code from project 1. If you got subclipse working, you can update your code from within eclipse using Team → Update. If not, use the command line or tortoiseSvn to do "svn update". If you did not get the code checked out for project 0, follow the instructions to check out the code in that writeup.

2. Download my A* agent from D2L. Dan and Sam are also releasing their A* agent and it will appear on D2L as well. For both agents, put them into src and unzip the file. This will give you two GOOD heuristics to compare to and you are welcome to use either or both of our code for improving your navigation.

3. Change the worldconfig.xml file in examples.robotest to point to your agent in src/4x4. The detailed instructions for this are in project 0. Make sure to copy over a clientinit.xml in the src/4x4 directory so your agent knows how to start. Because you will most likely want to run the ladder this time, also change the ladderconfig.xml and roboconfig.xml in examples.laddertest to point to your agent.

4. Create a robot that uses learning as described above. If you are doing trees, don't collect your data and learn online but rather collect it offline, learn in separate code (outside the agent), and then have your agent use the final tree. Build and test your code using the ant compilation system within eclipse or using ant on the command line if you are not using eclipse (we highly recommend eclipse or another IDE!).

5. Test your player on a sample ladder. This will ensure that the agent runs on the CSN linux machines and that you have the agent correctly configured. To do this, we have created a submission script that will take your agent as input and run it against the heuristic agents several times and output the information to your terminal window. WARNING: It will output a LOT of text so either be ready to scroll or cat the output to a file. To submit to this agent, do the following:

   ```
   /opt/ai4013/bin/submit CS4013 Project2TestLadder *.java clientinit.xml
   ```

   where you can simply list the java files instead of using *.java if you choose. Note that the xml file MUST be named clientinit.xml!

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. Remember to change your password!

   (b) Make sure your working directory contains all the files you want to turn in. All files should live in the package 4x4. For example, if all of your code lives in `MyRoborallyAgent.java`, you would submit your code using the following command. The clientinit file is required to run your client!

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

If you have extra code in `AStar.java` and `Graph.java`, you would submit using the following command:

```
/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 decision trees. A correct decision tree will learn outside of the simulator and will grow a tree using information gain. Each leaf node will have probabilities (instead of only labels).
  - 45 points if there is only one minor mistake. An example of a minor mistake would be a minor error in information gain, off-by-one errors, not correctly calculating probabilities at the leaves, etc.
  - 40 points if there are several minor mistakes.
  - 35 points if you have one major mistake. An example of a major mistake would be not using information gain correctly (e.g. implementing only the binary version when you have more than binary choices), incorrectly choosing attributes that do not yield the highest gain, etc.
  - 30 if there are several major mistakes.
  - 25 points if you implement some form of learning that turns out a tree but contains enough mistakes that you are unsure exactly how it makes a tree
  - 15 points for an agent that at least does something other than random movements.
- 25 points for correctly taking the trained trees and using them to intelligently control your robot
  - 20 points for one minor mistake. An example would be correctly bringing the decision tree over (using if/then rules) but then failing to use it to intelligently control your agent (e.g. you never adjust the agent's actions based on the tree or you never adjust the probability thresholds for the agent)

  - 15 points for several minor mistakes or one major mistake. An example of a major mistake would be incorrectly bringing the decision tree into the agent.

- 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 (not sure you can do that with A* anyway!)

- 15 points for your writeup. A full-credit writeup will describe your implementation of learning, include a learning curve showing how your trees grow, and a discussion of how you used the information from the trees to improve your agent.

- As with the previous project, we will deduct 5 points from your total score if your password has not changed from the default (cs#4x4) password.