Project 1
Computer Science 4013: Artificial Intelligence
Due 12:01AM Feb 23, 2010
Note: NOT the beginning of class

Introduction

This project will focus on moving around the Robrally board intelligently. In future projects, you will deal intelligently with the opponents and with teams. For this project, your agent can pretend that there are no other agents on the board, although if you want to deal with them, that is fine. The code that you generate for this project will be reused for the future projects. At the completion of this project, we will also hand out our solution to the project which means that any problems you have with this project will not stop you from completing the remaining projects.

The goal for this project is to touch the flags in order in as few steps as possible and with as few deaths as possible. Your agent will use A* search for navigation. However, because the environment is partially observable, stochastic, and multi-agent, you will not be able to use straightforward A* search. To handle the partial observability and the stochasticity, implement the following variant of A*. For this project, you can choose to ignore the multi-agent aspect or you may address it using heuristics.

At each turn, the agent will be dealt up to 9 cards. In turn, the agent chooses the best 5 cards to execute in order. The number of cards an agent is dealt is reduced by one for each damage point that the agent has. If the agent has 5 or more damage points, some of the cards will be locked in place and the agent can only make choices about the remaining cards.

An agent does not have any information about what cards will be given to it in future turns and this limits the applicability of of traditional A* search. For this project, we want you to implement A* with *rollouts*. Rollouts are described briefly in the book and we will cover them in class but they are also described below specifically for this environment. What you will do is search to either a depth of the next flag or 10 hands deep, whichever is shorter. Recall that returning failure is a valid option for A* but you will not want to return a random hand if you get failure! Be sure you have a backup plan.

Rollouts, also called Monte Carlo search, are a technique for simulating the possibilities of unknown elements of the environment such as cards or dice. In this case, you will pretend that you can see at least 10 hands into the future for your search. You'll do his by dealing yourself that many hands (search for the depth of your hand (5 moves out of as many cards as the agent is dealt) and then simulate at least 500 different random deals to the agent. For each "deal", make a new Deck() and then deal yourself cards. Then search using A* for 5 moves out of the available cards until you reach the goal, the maximum search depth, or failure.

To implement A*, you will need to design an admissible heuristic. Because A* may not reach the goal in the available depth, you should also design an evaluation function (you can use your heuristic if you want) to evaluate a potential search path even if the goal is not reached.

The following pseudocode should make this more clear.

1. For r = 1 to number of rollouts (at least 500)
   (a) Deal myself 10 new hands
   (b) Search from current hand through 5 new hands until agent reaches the goal, reaches the maximum depth, or fails
   (c) Evaluate the quality of the search
2. Return the first 5 cards with the best AVERAGE quality

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:

- **5 points:** If you find an exploit and report it to us you can receive 5 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.

- **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 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 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 the number of steps they took to get to the final flag (with smaller numbers being better). Ties will be broken by the minimum deaths of an agent.

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 good heuristics for dealing with opponents, real-time planning where you plan in the background while executing your current best plan (see the discussion of real-time A^* in the book), or other forms of intelligent search. To get ideas on this, you may find the website `www.gamasutra.com` useful. 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 navigation search must still have A^* at the heart of it. If you are unsure if your search qualifies, come talk to me.
- 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.

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 0 agents will not work in this project due to an API change! You will need to start from either the FlagCollector or the Random agent to get the proper new client interface.

The agent class contains a startAction(), endAction(), and initialize() method by default. startAction() is called each time an agent is about to begin an action 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 timestep. This may be left empty if you have no need for cleaning up after an action. initialize() is called when an agent is created (but not when it comes back to life from being killed). The example heuristic agent shows you how to access the internal state of the agent and of the environment.

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.

Those pesky details

1. Update your Roborally code from project 0. 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. 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.

3. Create a robot that makes moves using A* as described above. 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!).

4. 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 or either be ready to scroll or put the output to a file. To submit to this agent, do the following:

   ```
   /opt/ai4013/bin/submit CS4013 Project1TestLadder *.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!

5. 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 Project1 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 Project1 *.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 Project1Late *.java clientinit.xml
   ```

Point distribution

- 40 points for correctly implementing A*. A correct player will have an admissible heuristic and will move to the flags efficiently. The robot will rarely run into trouble on obstacles.
  - 35 points if there is only one minor mistake. An example of a minor mistake would be having off-by one errors (where you miss a search node).
  - 30 points if there are several minor mistakes.
  - 25 points if you have one major mistake. An example of a major mistake would be failing to correctly mark nodes as visited so the search might infinite loop, failing to have an admissible heuristic, or failing to use both your heuristic and the path costs.
  - 20 if there are several major mistakes.
  - 15 points if you implement a search other than A* that at least moves the agent around the environment in an intelligent manner.
  - 10 points for an agent that at least does something other than random movements.
- 10 points for designing and correctly implementing an admissible and consistent heuristic. This will be graded as follows:
  - 8 points for one minor mistake. An example would be correctly designing an admissible heuristic but failing to implement it in an admissible way or making a minor coding error.
  - 5 points for several minor mistakes or one major mistake. An example of a major mistake would be designing a heuristic that is not admissible (but correctly implementing it)
- 30 points for correctly implementing rollouts. If you have a minor error, you'll get 25 points. Major errors will be 20 points
- 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!)

- 10 points for your writeup. A full-credit writeup will describe your implementation of A*, your heuristic, and why it is admissible. 5 of the 10 points will be given out for correctly describing the environment using the terms from chapter 2. We already said it is several of them but list the other environmental characteristics.
- 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.