Project 1
Computer Science 4013: Artificial Intelligence
Due 12:01AM Feb 19, 2009
Note: NOT the beginning of class

Introduction

This project will get your spaceships started by focusing on the sub-task of navigation with an initial heuristic agent using this navigation to play intelligently. Projects 2 and 3 will work more on shooting and moving around in an intelligent manner. The code that you generate for this project can be reused for the later projects. At the completion of this project, we will also hand out agents that can intelligently move around which means that problems with this project will not keep you from completing projects 2 or 3.

The goal for this project is to collect as many energy beacons as possible and to kill as many of the opponents as possible in your five minutes of game time. Remember that the energy beacons refill your energy levels. You will respawn when you die but that will take time away from your game time. The ship will know its own location and the location of the beacon. Beacons stay put unless they are run into by an obstacle or the ship reaches the beacon. In both cases, the beacon will randomly reappear somewhere else in the environment. Any ship can be killed by running out of energy. The energy torpedos and mines deplete a small amount of energy from the shooting ship and a larger amount from the ship hit.

This project will use A* search for navigation. However, because the environment is continuous and the obstacles and ships are moving about the environment, you will not be able to use straightforward A* search. To deal with the continuous environment, you should implement ONE of the following changes to A*. We will deal with the dynamics of the environment on the next page.

- **Gridded A*:** Break the environment up into n grid squares (where you determine the grid spacing). Create a graph based on these grid squares where adjacent grid squares are connected unless one of them contains an obstacle or a ship. Implement A* search to traverse the graph and find a path from the starting state to the goal beacon. You must design your own admissible heuristic (more detail on this below).

- **Hierarchical gridded A*:** The idea behind this approach is similar to the gridded A* except that the grid spacing is not fixed. Instead, the agent searches at a very coarse

granularity (large grid squares) and finds a path in this easier space. The agent then refines its path by making finer grid squares along the previous path. This process is refined until the agent has a reasonable path from the start state to the goal state. As with the previous approach, this approach will require you to design your own admissible heuristic.

- **Roadmap A*:** This approach is based on roadmap navigation. Instead of making grid squares of your entire environment, sample n points from the space and draw a straight line between each set of points. If the line connecting the two points does NOT intersect an obstacle or ship, connect the two points in your search graph. Implement A* and search on this graph. As with the previous approaches, this approach will require you to design your own admissible heuristic.

Any one of these approaches will allow you to create a navigation plan in a continuous environment using A*. However, none of these approaches will allow you to deal with the dynamics of the environment. This is inherently a tricky problem and the most straightforward way to deal with the dynamics is to replan frequently. For this project, you should keep in mind that replanning at every time step will be too slow to be feasible. Instead, your agent should replan dynamically with at least every 20 simulations steps in between plans. When the agent is not planning, it should execute its plan from the previous planning period. You may choose other approaches to replanning but you can not replan at each step.

Once you have intelligent navigation, your next task is have the agents choose among the actions intelligently using local search. In this case, you should make your navigation action available to the agent as a higher level action such as "navigate to beacon." Then the agent should then use local search to determine the next best action to take, such as moving to the beacon, shooting an enemy ship, laying a mine, raising the shield, or moving somewhere else.

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 Spacewar

We are reasonably certain that you cannot exploit the simulator (say by directly affecting your opponents energy 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 *using the google code bug reporting system*, you can receive 5 points extra credit upon verification of the report. Note, we know of one exploit (discovered last year) for which you cannot get extra credit as it is already discovered.
- **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 Feb 5th (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 beacons collected and the number of enemy ships killed. Ties will be broken by the minimum number of times an agent dies (e.g. it is better to die less).

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 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 A* search that deal with continuous and dynamic environments. 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.
- Your high level behavior heuristic must still have local search 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).

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 SampleRandomClient provided in examples.random shows you how to access the internal state of the agent and of the environment. It also is an example of how to do high-level actions since the sample random client chooses a location at random and navigates to it directly using pd control. This pd control is highly encouraged for your A* agent!

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 MySpacewarAgent.java.

Competing heuristics

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

- **Random** makes completely random moves. This generally results in random going through the space at high velocity and shooting. Random does not usually live very long nor does it collect many beacons.
- **DoNothing** simply sits and does nothing. If it gets hit by an obstacle, it moves around with the velocity imparted on it by the obstacle. Surprisingly, this is a good strategy for surviving although it clearly won't get collect many beacons!
- **BeaconCollector** is our naive beacon collecting agent (it just aims straight for them). It does well at collecting beacons but it dies a lot and it never shoots or lays mines.

Those pesky details

1. Update your Spacewar code from project 0. We have added an examples.project1 directory and a project1 target to build.xml. You can update your code from within eclipse using Team → Update or you can do it at the command line with "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.project1 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 ship 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. Create a ship that uses the A* navigation system and uses local search to move around the world intelligently.
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 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.

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 `MySpacewarAgent.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 MySpacewarAgent.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 target beacons efficiently. The ship will rarely run into obstacles, other ships, or bullets. It should collect a fair number of beacons by the end of the game. Sample projects and point distributions are given below:

  - 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 mark a grid square as occupied (which causes you to run into obstacles, ships, or bullets) or failing to correctly connect your graph.
  - 20 if there are several major mistakes.

  - 10 points if you implement a search other than A* 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.

- 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)

- 20 points for correctly implementing local search for the high level behavior. For a minor error, you will get 15 points. If you move around intelligently but not using local search, you will get 10 points.

- 10 points for correctly dealing with the dynamics of the environment. If you replan as often as needed (at least every 20 steps as specified above), you will get full credit here. If you never replan, you will get 0 points here. If you replan only once you achieve your target (and no other times), you will get 5 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 heuristic, why it is admissible, what form of A* you implemented, and how you dealt with the dynamics of the environment. 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 continuous 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.