Contest: Pacman Capture the Flag

Introduction

The contest involves a multi-player capture-the-flag variant of Pacman, where agents control both Pacman and ghosts in coordinated team-based strategies. Your team will try to eat the food on the far side of the map, while defending the food on your home side. The contest code is available as a zip archive in the post.

Key Files to Read:

`capture.py`	The main file that runs games locally. This file also describes the new capture the flag GameState type and rules.
`captureAgents.py`	Specification and helper methods for capture agents.
`baselineTeam.py`	Example code that defines two very basic reflex agents, to help you get started.
`myTeam.py`	This is where you define your own agents for inclusion in the competition. (This is the only file that you submit.)

Supporting Files (Do not Modify):

`game.py`	The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
`util.py`	Useful data structures for implementing search algorithms.
`distanceCalculator.py`	Computes shortest paths between all maze positions.
`graphicsDisplay.py`	Graphics for Pacman
`graphicsUtils.py`	Support for Pacman graphics
`textDisplay.py`	ASCII graphics for Pacman
`keyboardAgents.py`	Keyboard interfaces to control Pacman
`layout.py`	Code for reading layout files and storing their contents

Rules of Pacman Capture the Flag

Layout

The Pacman map is now divided into two halves: blue (right) and red (left). Red agents (which all have even indices) must defend the red food while trying to eat the blue food. When on the red side, a red agent is a ghost. When crossing into enemy territory, the agent becomes a Pacman.

Scoring

As a Pacman eats food dots, those food dots are stored up inside of that Pacman and removed from the board. When a Pacman returns to his side of the board, he "deposits" the food dots he is carrying, earning one point per food pellet delivered. Red team scores are positive, while Blue team scores are negative.

If Pacman is eaten by a ghost before reaching his own side of the board, he will explode into a cloud of food dots that will be deposited back onto the board.

Eating Pacman

When a Pacman is eaten by an opposing ghost, the Pacman returns to its starting position (as a ghost). No points are awarded for eating an opponent.

Power Capsules

If Pacman eats a power capsule, agents on the opposing team become "scared" for the next 40 moves, or until they are eaten and respawn, whichever comes sooner. Agents that are "scared" are susceptible while in the form of ghosts (i.e. while on their own team's side) to being eaten by Pacman. Specifically, if Pacman collides with a "scared" ghost, Pacman is unaffected and the ghost respawns at its starting position (no longer in the "scared" state).

Observations

Agents can only observe an opponent's configuration (position and direction) if they or their teammate is within 5 squares (Manhattan distance). In addition, an agent always gets a noisy distance reading for each agent on the board, which can be used to approximately locate unobserved opponents.

Winning

A game ends when one team returns all but two of the opponents' dots. Games are also limited to 1200 agent moves (300 moves per each of the four agents). If this move limit is reached, whichever team has returned the most food wins. If the score is zero (i.e., tied) this is recorded as a tie game.

Computation Time

We will run your submissions on our laptop. Each agent has 1 second to return each action. Each move which does not return within one second will incur a warning. After three warnings, or any single move taking more than 3 seconds, the game is forfeit. There will be an initial start-up allowance of 15 seconds (use the `registerInitialState` function).

Designing Agents

Unlike homework 2, an agent now has the more complex job of trading off offense versus defense and effectively functioning as both a ghost and a Pacman in a team setting. Furthermore, the limited information provided to your agent will likely necessitate some probabilistic tracking (like homework 3). Finally, the added time limit of computation introduces new challenges.

Baseline Team

To kickstart your agent design, we have provided you with a team of two baseline agents, defined in `baselineTeam.py`. They are quite bad. The `OffensiveReflexAgent` simply moves toward the closest food on the opposing side. The `DefensiveReflexAgent` wanders around on its own side and tries to chase down invaders it happens to see.

File Format

You should include your agents in a file of the same format as `myTeam.py`. Your agents must be completely contained in this one file.

Interface

The `GameState` in capture.py should look familiar, but contains new methods like `getRedFood`, which gets a grid of food on the red side (note that the grid is the size of the board, but is only true for cells on the red side with food). Also, note that you can list a team's indices with `getRedTeamIndices`, or test membership with `isOnRedTeam`.

Finally, you can access the list of noisy distance observations via `getAgentDistances`. These distances are within 6 of the truth, and the noise is chosen uniformly at random from the range [-6, 6] (e.g., if the true distance is 6, then each of {0, 1, ..., 12} is chosen with probability 1/13). You can get the likelihood of a noisy reading using `getDistanceProb`.

Distance Calculation

To facilitate agent development, we provide code in distanceCalculator.py to supply shortest path maze distances.

CaptureAgent Methods

To get started designing your own agent, we recommend subclassing the `CaptureAgent` class. This provides access to several convenience methods. Some useful methods are:

`def getFood(self, gameState):`

Returns the food you're meant to eat. This is in the form of a matrix where m[x][y]=True if there is food you can eat (based on your team) in that square.

`def getFoodYouAreDefending(self, gameState):`

Returns the food you're meant to protect (i.e., that your opponent is supposed to eat). This is in the form of a matrix where m[x][y]=True if there is food at (x,y) that your opponent can eat.

`def getOpponents(self, gameState):`

Returns agent indices of your opponents. This is the list of the numbers of the agents (e.g., red might be [1,3]).

`def getTeam(self, gameState):`

Returns agent indices of your team. This is the list of the numbers of the agents (e.g., blue might be [1,3]).

`def getScore(self, gameState):`

Returns how much you are beating the other team by in the form of a number that is the difference between your score and the opponents score. This number is negative if you're losing.

`def getMazeDistance(self, pos1, pos2):`

Returns the distance between two points; These are calculated using the provided distancer object. If `distancer.getMazeDistances()` has been called, then maze distances are available. Otherwise, this just returns Manhattan distance.

`def getPreviousObservation(self):`

Returns the GameState object corresponding to the last state this agent saw (the observed state of the game last time this agent moved - this may not include all of your opponent's agent locations exactly).

```
def getCurrentObservation(self):
```

Returns the GameState object corresponding this agent's current observation (the observed state of the game - this may not include all of your opponent's agent locations exactly).

```
def debugDraw(self, cells, color, clear=False):
```

Draws a colored box on each of the cells you specify. If clear is True, will clear all old drawings before drawing on the specified cells. This is useful for debugging the locations that your code works with. color: list of RGB values between 0 and 1 (i.e. [1,0,0] for red) cells: list of game positions to draw on (i.e. [(20,5), (3,22)])

Restrictions

You are free to design any agent you want. However, you will need to respect the provided APIs if you want to participate in the competition. Agents which compute during the opponent's turn will be disqualified. In particular, any form of multi-threading is disallowed, because we have found it very hard to ensure that no computation takes place on the opponent's turn.

Warning About Output

If one of your agents produces any stderr output during its games in an online match, that output will be included in the contest results posted on the website. Additionally, in some cases a stack trace may be shown among this output in the event that one of your agents throws an exception. You should design your code in such a way that this does not expose any information that you wish to keep confidential.

Getting Started

By default, you can run a game with the simple `baselineTeam` that the staff has provided:

    python capture.py

A wealth of options are available to you:

    python capture.py --help

There are four slots for agents, where agents 0 and 2 are always on the red team, and 1 and 3 are on the blue team. Agents are created by agent factories (one for Red, one for Blue). See the section on designing agents for a description of the agents invoked above. The only team that we provide is the baselineTeam. It is chosen by default as both the red and blue team, but as an example of how to choose teams:

```
python capture.py -r baselineTeam -b baselineTeam
```

which specifies that the red team -r and the blue team -b are both created from baselineTeam.py. To control one of the four agents with the keyboard, pass the appropriate option:

```
python capture.py --keys0
```

The arrow keys control your character, which will change from ghost to Pacman when crossing the center line.

Layouts

By default, all games are run on the `defaultcapture` layout. To test your agent on other layouts, use the -l option. In particular, you can generate random layouts by specifying RANDOM[seed]. For example, -l RANDOM13 will use a map randomly generated with seed 13.

Recordings

You can record local games using the --record option, which will write the game history to a file named by the time the game was played. You can replay these histories using the --replay option and specifying the file to replay. All online matches are automatically recorded and the most recent ones can be viewed on the contest site. You are also able to download the history associated with each replay.

Acknowledgements

Thanks to Barak Michener and Ed Karuna for providing improved graphics and debugging help.