Contest : Pacman Capture the Flag

Detailed document

Based on UC Berkeley CS188(http://ai.berkeley.edu/contest.html)

Introduction

The course contest involves a multi-player capture-the-flag variant of Pacman, where agents control both Pacman and ghosts in coordinated team-based strategies.

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.

In the final contest, the map is set at random. We will use -l RANDOM option of capture.py. Make sure to make algorithms that can be applied to any layout.

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.

Make sure to remember that each Pacman can carry at the most 5 pellets.

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.

Designing Agents

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. This function will provide you a (True distance) + (noise) measurement. Noise will follow the rounding of the Gaussian distribution with (mean) = 0 and (variance) = 9.

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:

---

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.

---

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.

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]).

---

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]).

---

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.

---

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.

---

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).

---

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).

---

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)])
