Srija Konjarla, Nisha Haridoss, Alex Weiner

CMPS 140

June 8, 2019

Pacman Tournament Report

Fundamental problems and solutions

From the beginning of the development process, it was clear that we were aiming to beat the Baseline team without adding too much complexity. We wanted to create an agent that would be runtime-efficient yet smart. To do this we began by understanding and experimenting with the features and weights of the Reflex Capture Agent. We thought it would be smart to tackle one "feature" at a time, allowing us to iterate on the agent strategy. This was the ultimate order of our development:

1. The first strategy we sought to try was "avoid getting eaten" when on offense. This meant that Pac-Man would run from nearby ghosts.
2. Pac-Man should be unafraid of "scared" ghosts after eating a Power Pellet.
3. Our offense and defense agents should switch roles when it can save time. This would save time after Pac-Man respawns.
4. Ghosts should run to the offense when "scared." Ghosts are useless when defending scared.
5. Our defensive agent should not stray too far from the food it's protecting.

Modeling the problems, representations, computational strategy, algorithmic choices

We sought to optimize and upgrade the code that the Baseline Team was using, so we began with the Reflex Capture Agent as a foundation. It seemed to work pretty well without much complexity. It finds the dot product of the feature and weight vectors given each successor state and picks the action that renders the highest evaluation. It uses separate features and weights for offensive and defensive agents. So we used this model and strategically added features and fixed the weights of existing features in order to increase optimality. As we completed each feature, we submitted pull requests on GitHub for us all to read and understand.

1. Avoid getting eaten

This feature is called **distToScaryGhost** which has a positive weight on offense. The value is assigned to be the maze distance from the nearest ghost. We made it so the value only decreased when a ghost came closer than 3 spaces. This way, Pac-Man would not be too distracted by ghosts.

2. Be unafraid of "scared" ghosts

This was an adjustment to **distToScaryGhost** where the value would be unchanged by ghosts whose `scaredTimer` was greater than the time it takes to close the gap between the agents.

3. Switch offense/defense after respawn

Instead of dedicating one specific agent as an attacker from the beginning, we wanted the agents to pick an attacker based on who is closer to the food. This idea was implemented by merging the defensive and offensive agent classes and adding a method determineStrategy() which returned **Offense** or **Defense** before calculating feature values. Whichever agent's distance to an opponent's food was shortest would be picked to attack.

4. Run to offense when "scared"

We made this work simply by returning **Offense** in the determineStrategy() function whenever the agent was scared.

5. Our defensive agent should not stray too far from the food it's protecting.

This one was tricky because the ghosts already had a pretty decent defensive behavior in the base reflex agent. But we did it by calculating the average distance to each of the items we are defending (this feature is called **avgDistToOurStuff**). Weighted negatively, if this value was too large the ghost would return to the general area of our food.

Any obstacles you encountered

One of our early ideas was to implement minimax with alpha-beta pruning. This strategy hit a big roadblock when we discovered a crash when querying for opponent actions. Moreover, we didn't always know the location of the opponent. It was tempting to continue through these limitations and get minimax working, but we found that the feature/weight modification workflow was probably a more efficient use of development time.

Evaluation

The simplest way in which we evaluated our agent was running it ~10 times against Baseline and recording the win rate. Given more time, we certainly would have spent more time considering how it did against real opposing teams.

Lessons learned

Through the whole process of building an agent from scratch, we learned how to implement concepts we learned in class and the ideas we had using the functions given to us. This involved us finding what functions we had to our use, what they did, and how they could help us. We also learned how to strategize and create the best agent we could which involved us closely observing how the agents were behaving during the tests we ran against the Baseline Team. The tests helped us debug, find problems with our agents, and re-strategize. We came out of this project with a solid understanding of building, and manipulating custom agents.