CMPS 140: Pacman Capture the Flag

Team 2Pac
Yona Edell, Stephen Woodbury
March 16th, 2018

1 Introduction

We started understanding the problem by taking several hours to read through the initial provided code. We knew we wanted to have a semi-offensive agent duo for the tournament, so we specifically looked into the OffensiveReflexAgent class provided in *baselineTeam.py*. We saw that it prioritized always eating the closest food (no matter what) by using an attached weight. We first tried making a combination of the defensive & offensive reflex agent by copying them into *myTeam.py* and then began tinkering—that's when the magic happened. We kept the initial idea of weights from the ReflexAgent class; however, heavily modified everything else. The reason we chose a semi-offensive strategy resulted from the fruitless nature of defense in the game model. Since no points are awarded for consuming an enemy pacman, we saw little merit in this strategy. Nevertheless, we did find defense useful in some scenarios to close the gap—particularly against teams with dedicated one defense/one offense agents.

2 Overview

2.1 Food & Score Tallying

We designed a new algorithm that was to be used for both of our agents. It was primarily offensive with a touch of defense. Naturally, it was a utility maximizing algorithm. We calculated and returned a utility value—named *value*—for every square surrounding our agent. To accomplish this, we first calculated the distance to the nearest food, multiplied by a weight, then added that to *value*. In addition, we divided the grid's height by two so we could obtain upper and lower food bounds. With these bounds, we weighted the upper food higher for one agent and the lower food higher for the other agents to encourage them to divide and conquer (so the agent's wouldn't stick together and try to eat the same food pellets). Then we got the current game score, multiplied that by a weight, and added that to *value.* Because our agent looked one turn into the future, we needed to keep track of the future game score (after the food is consumed) to incentivize our agents to eat the food pellets.

2.2 Enemy ghost/capsule accounting (as an invader)

Then we started looking for enemy ghosts/defenders within our sight. Given we found one less than 6 squares away, we calculated its current distance from our agent's next position. If the enemy ghost wasn't scared, we discouraged our agent to move toward it by multiplying their distance by a large weight and subtracting it from *value*. It is at this point that we realized our agents should be smart about the way they chose to eat capsules. We thought about a scenario: if our agent is near an enemy ghost and we have another pacman agent on the same side, the one near the ghost should focus on running away while the one further from the ghost

should focus on eating the nearest capsule. To implement this, we got the index of the other agent on our team (making sure they're a pacman), checked if capsules are available, and then checked if our ally is further from the ghost than our current agent. If this was the case, we discouraged our current agent to travel toward a ghost (even more) by multiplying the current agent's distance by a slightly smaller weight and subtracting it from *value* (similar to previous time). However, if the current agent came by a capsule while it's running away, we encouraged the agent to eat it by multiplying the capsule distance by a small weight and adding it to *value*. If our ally is closer to the ghost than our current agent, we encourage the current agent to move toward the closest capsule. For the case where the ghost was scared and had more than 10 units of time/actions of being scared, we discouraged our current agent to move toward it by multiplying the current agent's distance by weight and subtracting it from *value*. The idea here being that we don't want to eat the ghost and respawn (not scared) and then come after our current agent. However, if the ghost's scared timer was about to expire and it was within sight, we encouraged the current agent to move toward it and eat it by multiplying its distance by a large weight and adding it to *value*.

2.3 Enemy pacman/invader accounting (as a defender) & the Pinning technique

It is here that we wanted to add defensive functionality to our agents. While tinkering with the provided DefensiveReflexAgent code, we discovered a peculiar bug. When an enemy pacman approached our ghost agent, our ghost would chase the enemy to a corner and freeze (instead of eating the invader). For at least half an hour we tried to fix this problem to no avail. It is then that we had a crazy thought—maybe this bug isn't such a bad thing! And thus we named it our championed *Pinning technique*. As it turned out, this "bug" proved to be incredibly effective against teams with one attacker and one defender as well as every team without the smartest attacker agents. To implement this, we looked at the enemy invaders within 5 squares. If our current agent wasn't scared, we wanted it to defend. To do this, we got the distance between the current agent and an invader, found the index of our ally, got the distance between our ally and an invader, and then finally checked if the invader is more than 1 distance away from our ally. At this point we thought about when pinning should used. It was agreed upon that if we were losing by some amount, we should have both agents trying to get the score up by collecting food on the other side. Therefore we added a check: if the successor score is not currently losing by more than 3, we can afford to pin the current invader. To encourage pinning, we multiplied the distance between the invader and current agent by an astronomical value and added it to *value*. If the successor score was losing by 3 or more, we encouraged our agent to eat the current invader. This was accomplished by multiplying the distance between the invader and current agent by a slightly smaller value and adding it to *value*. Aside from pinning, if the current agent were scared, we discouraged it from traveling toward an invader by multiplying its distance to the invader by a small weight and subtracting it from *value*. Finally, *value* is returned.

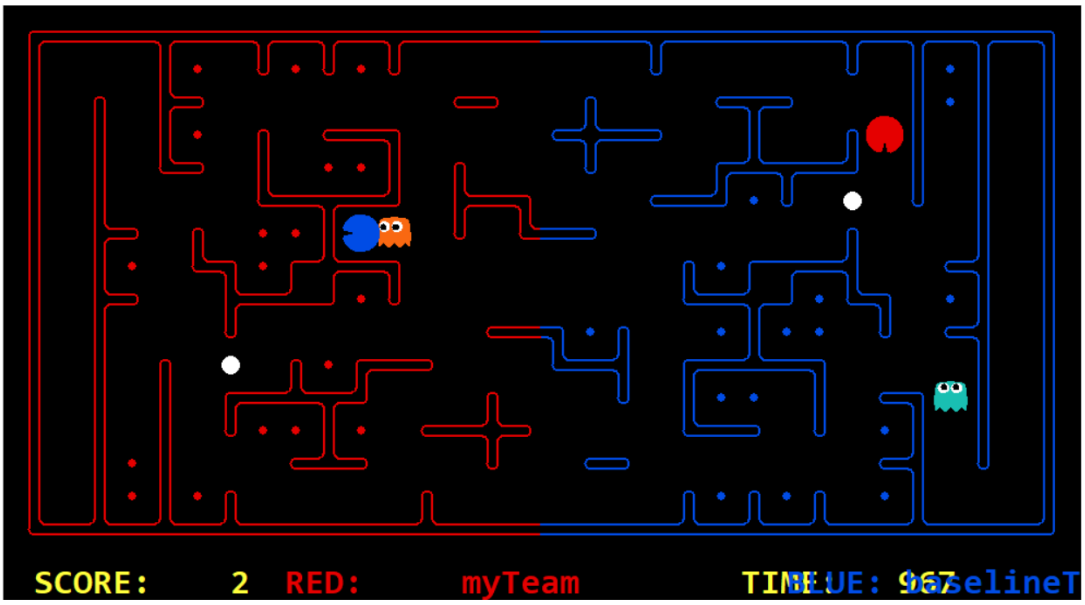

Figure 1: Pinning technique in action

3 Additional Notes

3.1 Work Strategy

We used a GitHub private repository for storing all our code and created separate branches for each of us to work in. Once we submitted something to the tournament, each night we downloaded all of our losing replays. After watching through all of them, we would create a document and include all the teams we lost to with notes about the team color, losing score, reasons we lost, and the strategies we suspected the other teams were using. Unfortunately, we only did this three times and modified our code accordingly from the day we first entered a non-dummy agent—March 6th, 7th, and 12th. I wish we could've worked on this more, but we both couldn't dedicate much time outside of our other classes.

3.2 Results

We achieved our highest ranking—#3—on March 7th with a score of 304 points (11 below the #1 position). Of the games we lost for this run, on average, we lost by 2.77 points. At this point we noticed our team losing games from dumb agent behavior—agents freezing at the border, making the game result in a tie or agents freezing near an enemy ghost. To combat this, we tried adding a strategy to prioritize food more as the game time decreases and smart invader pinning (where our agent could eat a pinned pacman if another invader comes by and then pin the new invader); however, we were ultimately unsuccessful. We tried to do too much at once and these new features added instability and a few crashes among games, so we ultimately reverted to our original code for the final run. We also never had the time to implement enemy invader probability accounting (we only accounted for invaders within sight/5 squares away). On the final run we achieved a ranking of #4 with a score of 308 points (4 points below #3 and 26 points below #1).