Erjie Zhang
Trevor Sartor
Martin Bojinov
Leilani H. Gilpin
CSE 140

P4 - Pacman Capture the Flag

Strategy:

- Have both take the offensive strategy., with one prioritizing the north half of the screen and the other prioritizing the south, so they don't both go the same way
- Try two different implementations (one minmax, one search)

Evaluation Things to consider:

- If enemy is nearby (~2 tiles?) and ghost and ghost isn't scared, move away
- If GhostScared and the time is high enough that we can get to them, move towa
    - getMazeDistance helpful here
- If enemy is nearby and not ghost, move towards?
- Moving Closer to Food = Good

Eval Function Ideas:

- isBraveGhost(them) and isPacman(us)—> run away
- is ScaredGhost(them) ignore?
- isPacman(them) and isBraveGhost(us) → move towards them, if we're a defensive agent, otherwise very small weight to move towards

Progress and issues:

1. We started with a minimax agent and we have to modify it because now the game is in capture mode.
    a. A lot of useful methods have been given, so it didn't take too much time.
    b. The reason we used minmax is that minimax agents made decent and functioning predictions for the game. And it is not that difficult to code.
2. After modifying the minimax agent, we worked on evaluation functions
    a. We found our game kept freezing or crashing if the minimax tree depth >= 3
        i. And sometimes it's even lagging for tree depth = 2
        ii. But a depth of 1 might not predict that much.
    b. We set it to 1 because deeper tree depth makes the evaluation function less important.
3. Our strategy is to have both super offensive agents. So we started with a linear combination of max/min food distances. However, it caused the agent to starve instead of eating food. Therefore, we added some reward scores for reducing the number of food (in other words, eating a food)

4. We have split our two agents: one eats food in the upper maze and the other eats food in the lower maze. The small issues like starving in some circumstances still existed
5. We shouldn’t use qualified input, and our agent didn’t run in the first tournament. So we have to put everything into myTeam.py
6. We found the problem of thrashing happened in some particular mazes during the tournament against other teams and given agents. The reason could be after eating a dot, the next dot is too far away, so the agent would just stay in place. To fix that, we tried to run that maze against some maze, adding some attributes like max food distance. It did help with some maps, but it didn’t solve the problem.
    a. Modifying weights for some attribute might fix thrashing in a map, but it eventually would end up in a situation where it kept thrashing for some map. Therefore, we had to make it consistent for all mazes (or maybe for most mazes at least)
7. Instead, we didn’t understand the getObservation() method in capture.py, we thought it might help to fix thrashing problems.
8. By printing out positions of observation, we decided to add an “anti-thrashing” attribute depending on the observation and previous observation.
    a. If the current state is the same as the previous state (keep thrashing around the same position), more penalties on the final score
    b. And it actually helped a lot to fix thrashing for most maps.
9. One thing we really don't like is that there seems to be no way to test many random seed games using one line of cmd.
10. We also found that other teams’ agents were also thrashing in some matches.
11. However, even though we won 1st place in one RR, we still lost to the baseline.
    a. Baseline Agent seems to be very consistent. Sometimes simple does work better than other more sophisticated agents.
12. In a recent RR after new_staff agent was added, we lost to the new agent

The fundamental problem we are trying to solve in this project is to let our agents eat food efficiently. To model this problem, we use the Minimax agent with an evaluation function.

The idea is similar to our previous assignments on implementing minimax agents. In the search space of the capture mode, our agents will take the best action (max) corresponding to our opponents’ decision (min). It is not very different from our previous assignments because the target is still eating pellets.

Minimax agents can do basic predictions, and they behave very well assuming our opponents also behave near-optimally. Furthermore, alpha-beta pruning can be applied to increase efficiency dramatically.

Our strategy is to have both two agents take an offensive strategy. The game has a time limit, and the more food we eat, the better score we have. Therefore, double offensive agents can be very effective. One agent eats food in the northern maze and the other takes care of the southern maze.

However, minimax agents do have some disadvantages. One of them is our first challenge at the beginning of this project. Minimax could be very expensive in both time and space complexity. When we set our tree depth to 4, the game is frozen after a few ticks; it still behaves very laggy and sometimes crashes after reducing the depth to 3. The game can run with depth = 2, but the performance is not very smooth even though we already have applied alpha-beta pruning. As a result, we turn in our code with depth = 1 in the minimax tree for consistent performance.

Because of a relatively small depth value, we have to focus on our evaluation function. As we know, the smaller the depth is, the more important the evaluation function is. Our evaluation function is a linear combination of the following attributes:

1. Number of food to eat
2. Distance to the closest food
3. Split agents to bot/top side
4. Avoiding enemies
5. A special anti-thrashing feature

The number of food to eat is to make agents eat a pellet to reduce the number of enemy food, where numFood = 1000 * (1 / len(enemyFood). Agents will get a large award if they eat a pellet. The award weight, which we set to 1000, has to be large enough to force agents to eat instead of starving.

Distance to the closest food is also considered. As mentioned above, agents will only consider food in their target fields (upper or lower maze). However, if there are not many pellets on one side of the maze, the agent will look for the closest food on the other side as well. The weight of this attribute is -4.5, which means the- 4.5 * minDistance. Agents will be encouraged to get closer to food to avoid penalties on evaluation. We also considered adding maxDistance, but it eventually did not work well.

Avoiding enemies is not prioritized that much because our first objective is to eat as many pellets as possible. However, sometimes eating by enemies can be a huge loss in some situations. To be effective, agents should only avoid enemies if and only if there are brave ghosts nearby. If there are any brave ghosts in maze distance < 3, avoidGhost = -6, which will let agents choose to avoid enemies or eat a pellet to maximize the evaluation score.

One thing we did not add to our evaluation function is the capsule. We believe in most cases chasing a scared enemy will be a waste of time since we need to prioritize food eating. Therefore, eating capsules neither increases or reduces the evaluation scores. A scared enemy should be ignored in our strategy.

During the implementation period, many groups including us have the problem of thrashing. In other words, it means Pacman only thrashes around the food instead of actually eating it, also called “starving”. In the beginning, we assume that eating a pellet will make the next closest food very far away, so the minDistance penalty will be large. Then, we reduce the penalty and increase the eating reward, and it helps a lot in some maps.
However, the thrashing issue still exists in many random maps. To make our agents more robust against other teams, we add an anti-thrashing attribute. The algorithm is: by calling getPreviousObservation() and getCurrentObservation(), we check if our agent's current position is the same as the previous one, antiThrash = -5. In other words, there will be penalties if an agent is stuck around the same position. This dramatically improves the behavior of our agents against other teams in random mazes.

A lesson we learned during our project is to fix an issue, we have to locate where the problem is before taking action. We spent a lot of time figuring out the exact reason that caused the thrashing. However, it was not that hard to code after we understood how to fix the problem. Hence, locating the bug is the first step to fixing a bug. Also, communications between teams help us figure out the solution to our main problem of thrashing.

Additionally, one thing we realize is that the baseline agents are very consistent against some of our teams. Baseline staff are just reflex agents with simple evaluation functions. Unlike our strategy, the baseline has one offensive and one defensive. We have won one of RRs but lost most games against the baseline. We have tried many strategies and modifications on our agents to increase the win rate against the baseline, and it helps reinforce our agents and perform better in the contest.

Overall, we have fun working with each other on P4. We all have learned a lot while implementing the capture agents and fixing bugs. Now we have more experience in the representation of AI problems and applying the knowledge we learned from this quarter.