Dakota Sliter

Austin Ackerman

Noah Betz

**AI Project 1**

**Introduction**:

Overall, this project was challenging from a programming aspect, but our group found it surprisingly fun to experiment with our agent behavior. Logistically, we had some issues getting together synchronously to work on the project, so we ended up working asynchronously most of the time and giving one another a summary of what we worked on. This included issues we encountered and solutions we came up with. We also threw around many ideas on how to approach agent behavior and discussed which ones we’d like to attempt implementation of. In the end, we settled on a few key behavioral features for our agents, building off of improvedTeam.py:

* One agent dedicated to defense, one for offense (in the beginning of the match)
* Fix issue with offensive agent cowering in dead end corners when enemy ghost was near
* Add points of interest for defensive agent to stay near (patrol)
* Optimize feature weights

We also established some nice-to-have agent behaviors, although we were not able to successfully implement these, either due to difficulty or time constraints:

* Defensive agent dynamically patrols or waits in areas of high-density food
* Strategic usage of the capsule to kill enemies
* Add feature to prioritize offense agent moving to areas of high-food density over lower densities. For example, if there is 1 food 3 moves away, but 3 food 5 moves away, attempt to prioritize pursuing the 3 food cluster
* Simulate likely enemy paths to either avoid or chase better
* Write genetic algorithm that finds the optimum weights of our agent features

This writeup will be broken into three main phases: *Exploration*, *Experimentation*, and *Optimization*.

**Exploration:**

To get started on the project, we first made a Discord server for all our communication purposes. We then started a GitHub repository with the base project structure and everyone set up their hosts for remote pulling and pushing. After running a few sample games to observe agent behavior, we took to analyzing the starter code line-by-line and commenting what each did. This served the purpose of familiarizing us all with the overall strategy employed by improvedTeam.py and getting an idea of what types of methods and properties were available to us. The commented code may seem a bit redundant in places, but it did help to understand the task better.

Next, we created a shared Google Document to hold some ideas we had about how to approach our agents’ behavior. We decided early on that we wanted one agent to remain on defense to avoid giving up any points in the beginning, so our other agent can focus on building up a lead. One idea was to implement a type of behavioral switch in our offense agent, such that when our point lead is greater than a set amount, the agent returns home and begins patrolling friendly food areas. Most of the ideas we came up with are summarized above in the introduction section of this document.

Observing how the games played out, it was clear that the improvedTeam.py offense agent “cowering” problem was a high priority fix for us. We proposed solutions of adding checks to prevent entering dead ends, even for food, as well as maintaining a state variable of last seen enemy positions. This would allow our agent to choose actions that avoid areas where the enemy is likely to be headed, and to stay away from fatal corridors. Additionally, we decided that the defensive agent could be improved if it prioritized certain areas of the map for patrol and didn’t stray too far from those points of interest, even to pursue an invading enemy. This way, an enemy agent would fail to lure the defense away from key chokepoints or food sources.

We now had a long list of ideas for our agent behavior, but the list was longer than we had time to experiment. As such, we rated the ideas based on priority:

1. Fixing the cowering behavior
2. Add points of interest for defense agent to stay close to
3. Offense agent preference for more food-dense areas
4. Switch offense to defense after a significant point lead
5. Target capsules strategically

Next, we began to experiment with implementation of these features and optimization of weightings.

**Experimentation:**

After sufficient planning and discussion, we decided to start coding up some behaviors. We started with fixing offensive agents from getting stuck in corners. Our first idea to fix this was crude and only somewhat effective: simply prevent the agent from taking the “Stop” action if on offense, and choose a different bestAction. This has the effect of preventing our agent from sitting in corners, but they would still get stuck in narrow corridors with only 2 directions to go. The agent would simply move back and forth, but running in circles is better than cowering. We discussed how we could further improve this behavior, and came up with the idea to add a feature that is based on the agent’s distance to it’s home side. In this case, the feature competes with fleeEnemy, so that the offensive agent’s action prioritizes getting closer to home when enemies are near. In practice, this seems to have fixed pacing and cowering, but an issue exists where the agent will sometimes run right into the enemy, dying, and dropping their food. We decided to minimize the risks posed by this imperfect behavior by limiting maximum food carrying capacity to 3 before the agent begins to path back home. We brainstormed some alternative ideas to avoid the enemy but did not implement them successfully. One such idea was to keep a list on the agent object that tracked every move it made, and when an enemy was within a certain distance, choose actions that take the agent back down the path it arrived from. The thinking here was that if we assume the enemy can come from four different directions at most, and our agent entered its current position from a single direction, then in the best case our agent has 3 “safe” directions to go if its path is reversed, giving a 75% chance of evasion. In the worst case, we have 2 directions to go, giving a 50% chance of evasion (action to action, not overall). This assumes a somewhat random distribution of enemy and player positions, and only accounts for the closest enemy “chasing” agent in each case. Of course, agent distributions are not random, and we decided that this behavior would not be a significant improvement.

Now we began working on the defensive agent. Observing it in action, the base behavior from improvedTeam.py was pretty sufficient for dealing with the base offense agent. However, we are going to be facing off against other teams’ AI, so we needed to make some improvements if we want to win (or at least make it past the first round). After clarifying that the tournament map was the same as the base maze we are running our tests in, we decided that we could analyze the maze for points of interest that offer a strategic advantage for our defense agent. With some trial-and-error experimentation, we grabbed the tuple values of these POIs and added our agent’s distance to them as a feature in getFeatures. We weighted it slightly higher than invaderDistance, to avoid our agent getting pulled away from these chokepoints by a fleeing enemy. In practice runs, this seems to work well, resulting in an agent that seems to patrol the different chokepoints until an enemy gets close, and ensures that improvedTeam.py never scores. This behavior is hard-coded, so it will not work dynamically in different maze layouts. We acknowledge this issue and came up with an idea on how to improve it: dynamically generate points of interest by looking at all the food we need to defend, calculated food densities of each grid position, and then choose the top 2 or 3 positions as our defensive POIs. While we were not able to implement this idea, we believe we could work out the solution given more time. An additional, minor change we made to the defensive agent was to ensure it never went back into the spawn area, since there is no reason those actions would ever be beneficial.

Another major idea we had for our offense agent was to add a state tracker to it that would allows us to switch from offense to defense and back dynamically, based on our score. One way we think we could implement this is to set a simple boolean, self.isOffense, and check it’s value in each getFeatures() call. Based on this value, we could use the same agent class and simply control how features are decided using simple logic branches, adding some checks in chooseAction() as well. Against the improvedTeam.py, this additional behavior doesn’t seem necessary since our defensive agent has no problem holding off their single offense agent. However, against a more dynamic strategy, such as two attackers or “rushers”, such behavior may be useful. The key would be evaluating a “safe” score lead at which to take up defense. Another consideration is that we may need to address the potential issue of agent “clumping”, where each agent will likely choose similar locations to patrol, resulting in a less effective defense against multiple attacking enemies. We could do this by adding a feature based on teammate distance and weighting it to encourage agents’ to maximize their separation, up to some reasonable value.

Since we felt confident that we have decent baseline agents, we moved on to optimization.

**Optimization:**

Tweaking weights and how features are calculated is the main way we decided to optimize our agents’ behavior. However, we quickly discovered the tedium and prohibitive time component of waiting for each game to play out. Since there is an element of randomness in every match, it’s possible a positive change in agent behavior could still lead to a defeat in a test match, incorrectly leading us to believe said change was deleterious. To solve this problem, we discovered how to run the game with no GUI using the -q option when running the game in the command line. We increased the number of games ran using the -n option as well, both found in the capture.py file. We then added some code that saves the results of these matches in a text file that we can look at after all games are done.

Our optimization process was as such: after each significant change, run 30 matches against improvedTeam.py, and examine the output scores for significance. Even without the GUI, these batch runs took anywhere from 15-30 minutes, so we ran multiple tests in parallel. For example, one terminal window would run the test with a weight value of 100, while another would run the same files but with that specific weight set to a different value, like 110. This sped up testing quite a bit, but we found that changing weights was still a bit of a guessing game. Predicting outcomes was difficult, and sometimes weight changes had the opposite effect of what we had intended. We settled on keeping weights close to where we had started after running many tests and seeing unpredictable results.

While we did not implement this, a fun idea we had for optimization was to create a genetic algorithm that would optimize weight values. In that case, we would take weights to be the “genome” of an individual agent, and a generation would be that agents’ performance in a batch of games against improvedTeam.py or a version of myTeam.py. Then, after running many parallel tests with different genomes, we select a few of the best, mutate the weights values, add some minor crossover with other top performers, and then run the whole loop over again with the child generation. We concluded that such a task would take a very large amount of time and computing resources to run significant population numbers, but it was fun to tie in new concepts to the project.

**Final Thoughts:**

As stated in the introduction, our group really enjoyed this project. We all enjoy video games and putting AI concepts into context like this really served to reinforce the ideas we learned in class. It was challenging but engaging to try and imagine the world through the perspective of our agents. While our group had some time conflict issues that prevented us from all talking in the Discord server synchronously, we still made it work and made relatively even contributions to the project. In future, we would get started earlier, since our schedules got filled with prior commitments like other class homework, midterm exams, and work.