CMPS 140 Winter 2018 - Final Project Report:
Team Name: HAL9001 --- *Team Members:* Alex Williamson, David Stewart

What were the fundamental problems we were trying to solve?

There were many fundamental problems that we needed to abstract from the baseline agent given to us in order to form the best agent possible. Some of the first problems we wanted to address included determining the best combination of offensive and defensive agents, how to determine which agents were on our team, how to detect enemy ghosts in the vicinity of our agents, how to find the closest foods to eat for our agents, how to make our agents run away if being chased by enemy ghosts, and how to chase enemy agents who were invading our side to try and collect food. There were other more intricate problems that we worked to find solutions for as our agent progressed and gained feedback from nightly tournament replays, but in order to understand the basics we started with these simple problems and worked up from there.

How did we model/represent the fundamental problems?

To model the fundamental problems, we relied on utilizing a feature vector and its corresponding weights vector. After generalizing our code to record the indices of our agents regardless of which team (red or blue) we were assigned to, our code ran a series of if-statements (like a giant switch) to determine which features needed to be active for the current state and action. These features could be a variety of things related to the fundamental problems, such as determining if an enemy ghost was nearby and sequentially running away. Once certain feature values were selected to be added to the feature vector, we gave them weights for their associated weight vector variables. Finally, in the evaluation function used to determine the utility of taking an action from the current gamestate, we returned the dot product between the weight and feature vectors. This helped our agents determine which action to take by picking from the set of all legal actions from the current gamestate the action with the highest returned utility.

What computational strategies or algorithms did we use to solve each problem?

For determining which types of agents worked best, we figured out that having two offensive agents would be the most effective, as we decided to encode defensive functionalities within the offensive agents themselves. This meant that whenever our agents detected an enemy agent who could be eaten, they would switch roles from eating food to defending their own food and getting rid of the intruder. From here, we assigned one of our ghost agents to one of the enemy pacman agents that was invading our side of the board (based on closest distance to our agents). This allowed each agent to only chase an individual invader it was assigned to, again eliminating any cases where both agents tried chasing the same invader. Having two offensive agents also provided us the opportunity to implement a divide and conquer strategy where we could partition the enemy's side of the map and eat their food in the most optimal way possible. We assigned each offensive agent one portion of the opponent's food, so then each agent would go for separate food items instead of the same.

Each ReflexAgent had its own index attribute, so we utilized this aspect of the Reflex Agent's state to figure out which agents were on our side. We knew that indices (0,2) or (1,3) go together, so we used the getTeam() method to get the indices which correspond to our agents. To detect enemies, we constantly checked to see if their position was “None” and if they were a Pacman with the “isPacman” attribute. If these checks were both not true, this meant that the opposing agent is not within range of one of our agents and is also not on our side since the opposing agent is not a Pacman. Similarly we could detect if an enemy is approaching one of our Pacmen on their side by checking to see if their position is “None” and if they are not a Pacman with the “isPacman” attribute.

What were some obstacles we encountered during development?

One of the first obstacles we encountered during development was figuring out how to access specific information about the gamestate. For example, we wanted to know things like which indices were our team and which indices were our opponents so that we could call things like gamestate.getAgentState() and know for sure that the gamestate we were looking at was actually the one were wanted to be looking at. This extended to other things like knowing how to check if an enemy agent at a gamestate was pacman or not, or checking to see if they were detected to be in range of our agents. Another obstacle we encountered after that was finding a good way to test new iterations of our agents to see if we were improving upon features or not. The most difficult part about implementing new features is that in order to get them to work specific actions, it would have to happen in game when we were testing them. This proved to be very tedious with the baseline team because their AI was very dumb and very random, unlike the opponents we were facing in the tournament. Therefore, in order to get a more accurate representation of how were were doing, we began to save our old code for agents that we had submitted against the autograder to have a better baseline team to test certain actions against. We also would have our old agents duke it out between our new agents on the same map several times and average the results to see if our new agent made any marginal improvement over the old. A third obstacle we encountered was being able to discern what was actually going wrong with our agent after watching replays of our losses so that we could fix certain features to edit or add in for the next tournament day. Since there was no console output from replays, we had to watch them multiple times and compare multiple replays with each other to see if similar patterns for how we were losing occured. Once we were confident about some features that needed fixing, we would try our new agent against mock agents that imitated ones that we lost against to see if any difference was made. Sometimes, this did not prove to be fruitful, as the same mistakes would continue to be made because we didn’t target the correct feature, but overall we think this is what made our agents so dominant throughout the entirety of the 2-week tournament.

What do we think of our final agent?

Our final agent is something that we are very proud of because it did very well throughout the tournament despite, at its core, being very simple in comparison to other teams who placed well. We think that the main reason for this was because our strategy for splitting up our agents and having them collect different food from each other was particularly clever. Rather than having a feature for keeping our agents separated at all times (which many teams seemed to try), we had one agent see different food locations versus the other. We tried a couple variations of which food spots we should give each agent and decided that letting one agent see 1/3rd of the food, while the other agent saw the other 2/3rds, worked best for both the default map and other randomly generated maps. Once one agent collected all of its food, it would work together with the other agent to finish the total remaining food. We added a feature to keep teammates separated here in case they happened to be going for the same food but in most cases they were already fairly spread out and thus our agents were very efficient at food gathering, a crucial problem to get right. Another thing we attribute to our success was our methodical strategy when making changes. If we were deciding to implement a feature for a certain tournament day, we made sure that it yielded positive results for us by testing it out on our past agents, the baseline agent, or imitations of other agents that we went up against during class to confirm that we had improved upon our pitfalls from the previous iteration. This ensured that we were always making progress, rather than trying to improve on iterations of code that didn't work so well, which made adding features a lot easier.

As far as placement goes, we ended up finishing in 1st place for the midway tournament and 2nd place overall after the final tournament. We also secured the stats for longest number of 1st place finishes in a row, most 1st place finishes, and most times placed in the Top 3 (there was only one day we did not make top 3; we were 4th on that day). However, we still believe there are a couple things we could improve one, one main aspect being related to capsule logic. Currently, our agents only collects capsules if they happen to run into them while collecting food. However, we believe that we could have improved upon this logic to make our agents even better by adding some logic to use the capsules only when agents were getting chased. When in the vicinity of an enemy ghost, or a teammate was in the vicinity of an enemy ghost, we could have had our agents value capsules more than food, thereby collecting them first when they were in some close range to them. We could have also used some sort of pathfinding algorithm like A-star to map the agent being chased by a ghost to the nearest food capsule, thereby allowing it to live for longer. We considered adding logic like this initially, but we also took into account that getting killed by an enemy ghost isn't always a bad thing, since you get to respawn and have a chance of eliminating enemy pacmen who may be consuming your food. After doing well in the tournament, we decided to hold off on that feature and keep with what we already had working.

What were some of the lessons learned during the project?

One key lesson that we learned very early on is to keep your code simple and organized if you want to succeed in this tournament. Our first iteration of the agent had fairly messy code with nothing printing to the console telling us when a feature was added or what our values were giving us. This made it very hard to track down errors. The added complexity of what we were doing with the agents also made it hard to add anything new because it felt like we always interfered with other functionality in the project when doing so. It also was really hard to tell if our changes actually worked or did anything novel. After we simplified our solution to things we knew would work, it was very easy to tell if anything we added actually made a difference with the output of our utility values, or if it actually helped us win more games. Another key lesson we learned was that starting early and constantly updating your project solution to stay on top can actually be detrimental to your overall position in the end. Initially, we felt like we had a solid lead because not a lot of people were implementing strategies we had found to work really well. However, eventually, we felt that a majority of the class was using things that we had in our solution, making them harder to beat and also lowering our overall score. It also was harder for us to keep improving everyday if we didn't have someone else in the class better than us to motivate improvement or gain knowledge of their solution. I think if we had strategized our uploads better we may have had better results overall.