Seth Raley
Prof. Ng
CS 4365.001
10 May 2020

Project Report

In the Pacman CTF game, the goal is to capture pellets from the enemy's territory and safely return to you own territory to earn points. To program this, I implemented the minimax algorithm using Alpha-Beta pruning. This was not difficult to do, but created many problems that were difficult to handle.

At each leaf node of the adversarial search tree, I made sure it was my team's turn so that I could accurately evaluate the state of the game that would result from a valid action. This was done by simply making the depth of the tree an odd number, so that my agent would be the root and the leaf in the search tree. In the evaluation function, I determined if the leaf node was a defender or an attacker. On my team, I limited the number of defending agents to one, and the rest would become attacking agents. If the rules of the game called for one agent per team, I would most certainly lose (because the agent with the lowest index is the defender), but if the number of agents per team was greater than one, my team would stand a chance. Having only defenders definitely would have lost me the game, since I would score no points. If I had only attackers, however, I would be able to score more points but would be at risk of losing the pellets in my base. I tried making an agent that would play attack/defense depending on the context of the game state, but this was too time consuming to construct. In the evaluation function, if the agent was determined to be a defender, it would return a linear combination of the features and weights I extracted using two separate functions. The features function of the defender determined if the agent was in its home base, if the action it chose was stop or reverse, and if the agent was currently scared. It also counted how many agents were invading, the distance to the nearest enemy, and the distance to the border. The weights function of the defender gave punishments if the defender left its home base, if it chose to stop or reverse, if it was scared and near an enemy, and if it was far from the border when no invaders were present. Because of this, it was rewarded for minimizing the distance to the closest enemy and killing it. The greatest weakness of the defender is that if no enemies were around, it would return to the middle of the board next to the border, regardless of the enemies' positions. If the evaluation function determined the agent was an attacker, it would return a linear combination of the features and weights extracted from the corresponding offense functions. The features function of an attacker determined if the current action was stop, the score of the resulting state, the amount of food remaining, the distance to the nearest food pellet, the distance to the nearest enemy, the distance to the agent's home, if the agent would get trapped in a dead end, if the agent would get killed by taking an action, and if the agent has visited the position on the board before. The weights function rewarded the agent for picking up food and avoiding enemies, avoiding tight spaces that would get it trapped, and avoiding repeating the same set of moves too many times. This made it strong while no enemies were near and very nimble when collecting food, and allowed it to retreat whenever an enemy approached. The moves were tracked using a dictionary data structure that was incremented each time the agent visited a position on the board. This prevented it from getting stuck going back and forth in the same spot. After the agents were evaluated, the best action was returned and once more evaluated in the choose action function. If the agent was carrying three or more food pellets, it used the A* algorithm to find a safe path to

its home territory. Because of this, the greatest weakness of the agent is that it never collects more than three pellets.

I learned a lot in this project, especially while tweaking the weights returned in the weights function. I wish I had more time to implement a machine learning strategy to optimize a model for the weight values, but I believe it would've taken me too much time to learn how to do this. I could not find perfect weights for the attacking agent that would allow it to escape an enemy's range of sight and continue collecting pellets. In one example, I used the keyboard to put an enemy right on the border (while still being a ghost). When the attacking agent was on the far side of the keyboard agent (further from the border than the ghost), it simply stayed within the enemy's range, since the weight function rewarded it for escaping the enemy's range. Whenever it would get far enough away, it would return to the enemy's range and escape once again to get this higher reward. If I increased the reward for picking up food to counteract this, the agent would stop avoiding enemies altogether.

This was a great project, in my opinion. I liked how it was very open-ended in terms of implementation and left a lot of room for creativity. Given the opportunity to improve my team's agents, I would take the time to create a way for them to switch between attacking and defending based on the context of the game state, find a way to optimize the weights of the weight function using a machine learning technique, and find a quicker way to implement the A* algorithm so it could be used in the leaf nodes of my adversarial search tree. Overall, I am very grateful for this experience and everything this class has taught me.