Pacman Capture the Flag Competition: The Pacboys

Robert Chatterton
Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts, USA
chatterton.r@northeastern.edu

Zoheb Aziz
Khoury College of Computer Sciences
Northeastern University
Boston, Massachusetts, USA
aziz.zo@northeastern.edu

I. Introduction

A. Motivation

The goal with this project was to make a team of Pacman agents to compete in a game of two-versus-two Capture the Flag. We created two agents, one for both offense and defense while taking into account enemy AI agents trying to accomplish the same goal. This is an interesting challenge because there are many more things for our agent to worry about than your standard game of Pacman, such as: defending against enemy agents, making decisions about grabbing food or returning it for points, power pellets, different board sizes/layouts and not knowing beforehand the behavior of enemy invader and defender agents.

In more detail, the game has two sides, with two teams, red and blue. While an agent is on its own side, it acts like one of the ghosts in Pacman, and when on the enemy side, it acts as a normal Pacman. For these agents, the rules are the same as they would be in normal Pacman, except for the fact that the Pacman agent has to eat the food pellets and drop them off back on its own side to get the points. The ghost has the power to eat a Pacman and make it respawn. When a Pacman is eaten, it drops all the pellets it is carrying around where it dies and respawns on its own side. A Pacman eating a power pellet is the same as in the regular game, where the ghosts can be eaten. They are not worth any points, but they do respawn, giving the attackers an opening. There are two stopping conditions for each game: one, the game runs out of turns (each agent has 300 moves total), or two, a team returns all but two or fewer pellets from the enemy team. Our team only trained on the default map layout, however the final contest was played on the much larger 'Jumbo' map layout.

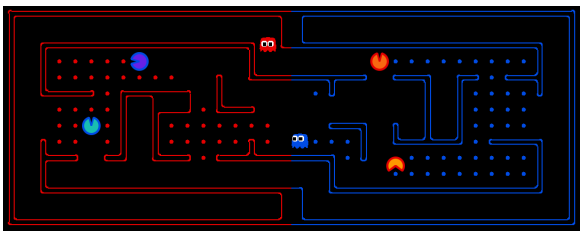


Fig. 1. The Capture the Flag default game layout

B. Challenges

We touched on this a bit up above in the Motivation, but this type of game has numerous special properties that make it more difficult than the normal game of Pacman. Firstly, in the normal game of Pacman, you know exactly what to expect. Four enemy ghosts, plenty of dots (that give you points right away when you eat them), that familiar map, and four power pellets in the corners. In the Capture the Flag gamemode though, the locations of the pellets change when attackers get eaten, and the agent needs to optimize between going for more food and dropping it off. The one we found most jarring in our implementation is dealing with the unknown behaviors of our competition. We found that after training, our agents would have very unique behaviors depending on the agents it was training on. For example, against the provided baseline agents, our defensive agent would play really far forward, preventing the enemy in, while against a much more aggressive offensive agent, it would play farther back and cut them off.

We ran into many challenges originally just implementing the genetic algorithm mainly because it was something we hadn't tampered with before this project. Our main way of conquering this was referencing online resources about genetic algorithms. Another problem we ran into was trying to find the best way to generalize the actions of our agents to work against a variety of different opponents. We could train our agents to perform well against a certain opponent, but training was computationally heavy and would not be viable in the actual competition. Our solution here was to train against a Q-Learning agent since we believed that was very general to the types of opponents we would face in a competition setting and we tampered with the number of generations to train on to try to ensure we didn't overfit our parameters to a specific opponent. Some other challenges we ran into were agents finding a location that optimized the fitness score, but didn't necessarily sought out to improve the score and games would very often end in a deadlock scenario where either we would tie or win/lose by a very small amount and nothing would happen until time runs out.

II. Related Work

We searched for a while for other implementations of the Pacman Capture the Flag project that utilized genetic programming, but were unable to find any. However, we did find a paper [1] that theorized strategies for "higher level" genetic programming with a game like Pacman. The paper discussed having goals like "get the closest power capsule" or "run from this ghost" rather than simple "move up", "move down", etc. This idea was what gave us the spark of inspiration to proceed in a similar way to how the BaseAgent worked, rather than following some other path of implementation.

Genetic Algorithms in general have a variety of use cases within the field of Artificial Intelligence. A big one that is somewhat similar to our use case being in the field of robotics. A specific robotics problem [2] we found that a Genetic Algorithm has solved was that of a robot finding its way through a maze. This is actually somewhat similar to our case with Pacman who is essentially navigating through a maze with the major difference being in Pacman Capture the Flag there are adversaries to deal with. The main difference between the approach we ended up on and the one outlined in this piece of related work, is that the robot approach created a very elaborate fitness function and did not rely on the use of weightings to influence actions. The robot created a "chromosome" encoding that would be constructed based on the surroundings of a given state and would determine the next action which could then be evaluated and scored by the fitness functions. The chromosome mappings were updated based on scores achieved throughout the course of a generation when crossing over to the next generation. While this approach is very successful in solving the simple maze, we would not have been able to emulate this exact approach due to adversaries positions changing and different throughout each game. The robot only acknowledged the information that was given in its current state and had no reason to suspect there was a moving target working against it while our agents had to take this into account and that is where the weightings came into play.

Another Genetic Algorithm use case we looked at was in an AI used in a self driving car [3]. While this was very similar to the Robotics project in the sense that there were no adversaries, this one was much more complex because it also has to deal with many external factors such as getting off course and crashing. The fitness function was derived from how close it gets to the optimal path and was run for hundreds of generations. This Genetic Algorithm, much like ours, used weightings to determine likeness of making a certain action. The actions however were determined from the weights being plugged into a neural network to try and predict the optimal action at each state.

III. Approach

Genetic programming, or the genetic algorithm, was the basis of our implementation for this project. Genetic programming is, in essence, a replication of how natural selection chooses the "fittest" of a population for reproduction. Boiled down, the algorithm has a few key components: some agent state, with values or features that are tweaked for every new agent, a fitness function to determine the efficacy and performance of the agent after each simulation, and a crossover function to analyze the current generation's best agents and continue those successful qualities into the next generation. The process of the algorithm is as follows: first, in generation 0, an agent is initialized with random weight values. It runs through the simulation, gets scored, and those parameters are saved for analysis at the end of the generation. Once the current generation has ended, the most fit agents are used as starting points for the next generation of agents. After some number of generations, the agents being made (hopefully) perform well at whatever simulation they are running on.

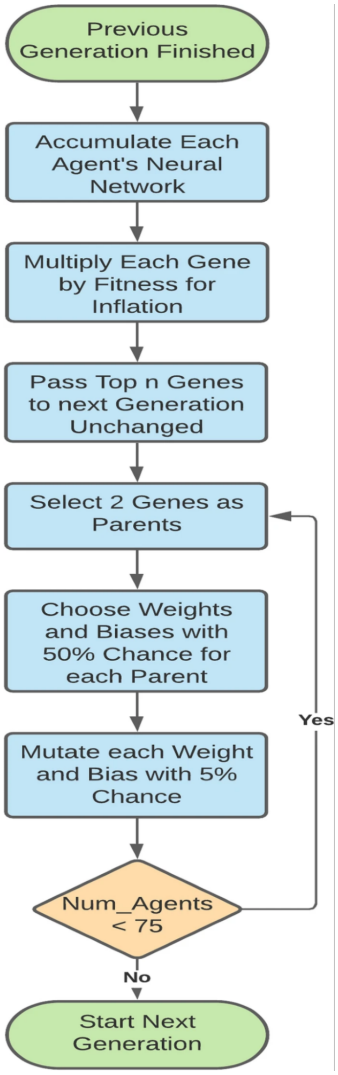


Fig. 2. The structure of the self-driving car algorithm

For our implementation of the Pacman Capture the Flag competition, we first were going to mutate the hyperparameters of a Q-Learning agent. However, when looking into it deeper, that came with a bunch of different challenges. For one, the time required to train Q-Learning agents was really long, and with training multiple generations while testing our

implementation, it just was not going to be feasible.

Our implementation was an improvement to the project's BaseAgent. This agent worked by assigning each potential move a score, based on a set of features, each of which had an assigned weight (initialized on creation of the agent, randomly based on the parents of the previous generation). These scores were calculated for each potential move, every turn, for each of our agents. After the game, the fitness of each agent was calculated as of turns played + score of the game. The best agent from each generation's values were used as the "center point" for each of the next generation's agents. When calculating a new agent's features, a random point between a set range of acceptable values for each feature was chosen, divided by the number of generations so far (in an attempt to converge over time), and added to the chosen parent's weights. The goal of our Offensive Agent was to eat enemy food and return it without being captured. The Offensive agent was initialized to spend time on the enemy board and capture food. We implemented a second behavior for our offensive agent for when it had captured food to then prioritize bringing it back to the defensive side as fast as possible to score points. After the training rounds the Offensive Agent generally learned to grab a single food and return it and then repeat. The biggest problem we had with this agent is that it would sometimes place too much emphasis on food and not enough on agents trying to capture it and would end up in a loop of dying before it could accomplish its goal. Another problem is because of its priorities, it would only grab one or two food before trying to deposit the points back to the defensive side when there were clear opportunities to get more points. Some improvements we could make is better awareness of opposing agents as well as the ability to make better decisions of when to eat more food or go back and deposit when carrying food. This agent also did not take into account the power pellet and could be improved to use that to its advantage.

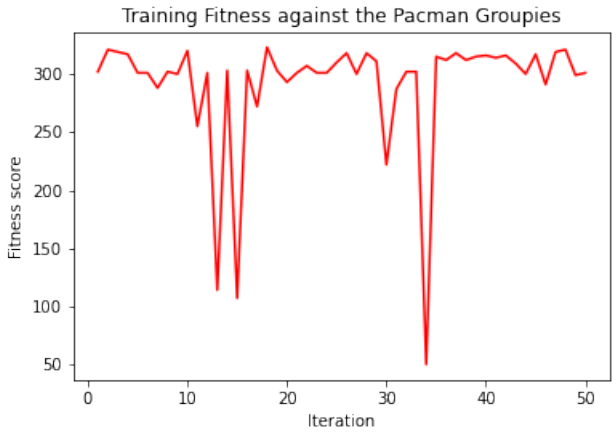
The goal of our Defensive Agent was to keep out invading agents and protect our food from being eaten and returned to the enemy base. The Defensive agent was initialized to prioritize these goals while disregarding things such as going on the offensive or capturing enemy food. The defensive agent in many scenarios after being trained learned to "lock down" a specific zone where the enemy would generally enter from and oftentimes created a deadlock zone causing both it and enemy agents to repeat actions indefinitely. The main scenario where the defensive agent failed was when an invader managed to capture a power pellet and the defensive agent did not know how to react to this and would end up running into the enemy agent and dying. The biggest improvement we could make here is power pellet awareness. Other than that the agent performed fairly well and putting an opponent or two into a deadlock was more often than not a favorable scenario for our team.

IV. Testing and Evaluation

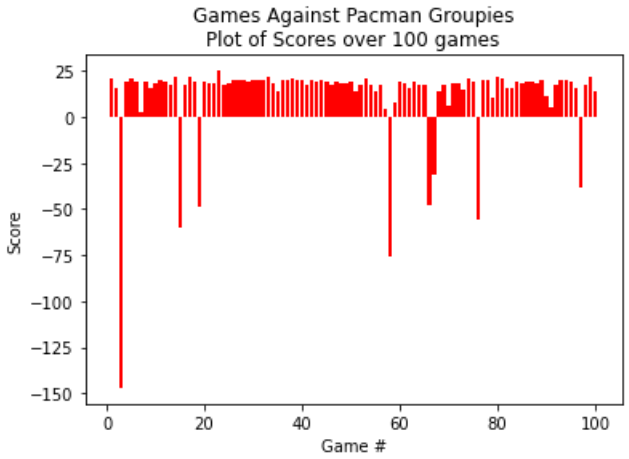
Once we had decided the goals and objectives of our team, we had to implement it. Starting off with the BaseAgent, we assumed that all we really would need to do was add some mutation/random logic to the weights, and we'd be all good to go. Shockingly (not really), that was not how our testing process went. We had a lot of problems with unexpected behaviors and our agents not meshing well together, which was when we split it up into two different, separate offense and defense agents. One aspect of our program that proved to be very useful was when we added a logger. This logger saved the mutated feature weights for each iteration, for each agent, as well as providing some other useful information regarding our generation status.

The best way we thought of evaluating our agent was to put it to the competition test against other Pacman Capture the Flag agents. Our first method of evaluation was the Baseline agent provided in the codebase. Since this was a trivial agent we expected to be better than it and our agent was able to a majority of times even when untrained. Next we searched the internet for agents to try our agent out against. We handpicked two agents we found on Github, one of which was picked at random, the other which we picked because it utilized a Q-Learning model (the style we anticipated other teams would follow and wanted to be ready). The Q-Learning team we found online was the team that we trained against for the competition.

For the in-class competition, it felt like all teams who had agents ready came fairly close in performance. Sometimes we would beat an agent, we would lose to the other team, and then the team we beat would win against the team we lost to. Sometimes games would be even, and sometimes games would be blowouts. It was interesting to really take a look at some of the strategies the enemy teams used and see them applied in the game, live. The first team (who we'll call Team Groupies) used a very similar strategy to how our agents worked, split into offense and defense, using a feature extraction/weight calculation to score each action-state pair. The second team (who we'll call Team Brandon) had a very different strategy. Their offense and defense agents utilized entirely different methods. Their defensive agent was a standard approximate Q-Learning agent, like we had worked with in the homeworks. However, their offensive agent utilized a Monte Carlo Tree Search when in close proximity to an enemy agent. This strategy really worked well against our defensive agent's strategy, making it so that our defense agent would end up chasing after the offensive one rather than catching it out.



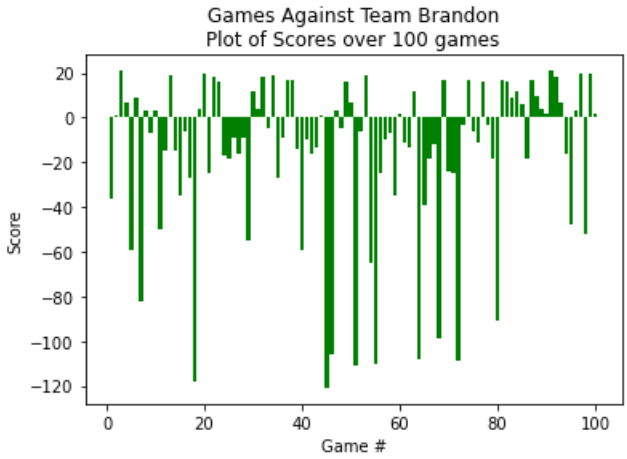
Trained against the Pacman Groupies, we won most of our games. We think that because our strategy was very similar to theirs, it was easier for our own agents to exploit while training against them. In the test of 100 games, our agents won 92 times. Looking at the graph below, one interesting feature is that our agents usually win by a much smaller margin than the margin that happened the few times that the enemy agents won. While we won with an average score of 17.696, we would lose against the Pacman Groupies by an average score of 63.125. Overall, the average score of a game played was 11.23.



Trained against Team Brandon, we lost a lot more. In the test of 100 games, our agents won 43 times and lost 57 times. Interestingly, our wins against Team Brandon were not as high as against the Pacman Groupies, and we also would not lose by as much as we did against the Pacman Groupies when we lost. When we won we had an average score of 11.674, we would lose against Team Brandon by an average score of 36.439. Overall, the average score of a game played was -15.75 (15.75 points towards Team Brandon's favor).

V. Future Work

Something we want to implement is multiagent crossover, where at the end of a generation the "starting point" for the next generation is a weighted combination of two or more of the best parents from the previous generation rather than just the single best parent. Adding more features to our agents is also a goal of ours since there are still many things the agent doesn't take into account, for example behavior does not change at all when any power pellets are eaten. Another improvement we could make is finding a better generic agent to train on for the purposes of competition, this would mostly be done through a trial and error approach of scouring agents on the internet to train on and testing to see how they perform against other agents.



VI. Appendix

Acknowledgments

We worked as a team of two for this project and all the programming was done using a pair program VSCode extension (called LiveShare) where we programmed all of the code simultaneously while together in person. The codebase we worked with was all written by UC Berkeley's Computer Science program. The agents we trained with are linked in the references, here [4] and here [5].

REFERENCES

[1] Teaching introductory artificial intelligence through java-based games. (n.d.). Retrieved May 2, 2022, from https://www.researchgate.net/profile/Amy-Mcgovern/publication/267206588_Teaching_Introductory_Artificial_Intelligence_through_Java-based_Games/links/557eeafe08aeea18b7795953/Teaching-Introductory-Artificial-Intelligence-through-Java-based-Games.pdf

[2] Dang, Anh T. "Genetic Algorithm Based Approach for Robotic Controllers." Medium, Towards Data Science, 30 July 2020, https://towardsdatascience.com/genetic-algorithm-based-approach-for-robotic-controllers-3966a9b874fb.

[3] Luthra, Jatin, et al. "Implementation of Genetic Algorithm for Path Estimation in Self Driving Car - SN Computer Science." SpringerLink, Springer Singapore, 31 Jan. 2022, https://link.springer.com/article/10.1007/s42979-022-01030-2.

[4] Q-Learning Agent we trained against, https://github.com/mrin17/PacmanCTF/blob/master/Code/leeroyTeam.py.

[5] Random team we found and trained against, https://github.com/pcgotan/PacmanCTF_Agent/blob/master/myTeam3.py.

CODE REPOSITORY

Link to GitHub Repository