Pacman Capture The Flag Project Report

Angelina Myles Collin Cleary

I. Introduction

The goal of this project is to produce an AI agent that can play a version of Pacman “Capture The Flag”. The game contains two teams each with two agents. There are two sides to an environment and as agents cross a border from their own side to the opposing side, they transform from defensive ghosts into offense based Pacmen. The objective is to gather food from the enemy team’s side of the environment and return to your side with that food. On the return, you also want to defend your own food from enemy incursions.

II. Methodology

A. Environment

The environment is multi-agent, partially competitive, partially collaborative, partially observable, dynamic, continuous, and deterministic.

III. Agent Design

A. Angelina

AI Algorithm Theory: Please be specific about the theory, as students must understand what algorithms they used (especially if the code is AI generated). Expectimax is a decision-making search method used in environments where outcomes are partially deterministic (controlled by an agent) and partially stochastic (controlled by unpredictable opponents). Expectimax builds a search tree with three types of nodes: Max Nodes, Chance Nodes, and Terminal Nodes. Max Nodes represent the agent’s turn, where it will choose the action that will maximize the agent’s expected utility. Chance Nodes represent randomness or an opponent’s behavior that is not strictly meant to minimize your agent’s utility. Terminal Nodes are located at the end of the search tree and often symbolize the game reaching some type of terminal condition. By recursively selecting the best action at the Max Nodes and averaging the expected outcome of the chance nodes, Expectimax produces a strategy that maximizes its expected utility rather than only guarding against worst case scenarios. Finite State Machine Algorithm moves through a finite set of states, switching between them based on rules. These states and transitions are meant to control behavior in a predictable and organized way. Breadth-First Search explores a search tree level by level, starting from a base node and visiting all neighbors before moving to the next depth. It uses a queue to expand nodes in the exact order they are discovered. Because it always

When I first started out designing an Agent I was thinking of AI Search Models we had talked about in class and ended up choosing to try and implement an Expectimax Search Algorithm because I thought it would be a good fit after understanding the adversarial nature of the game. My thought process was that I would have my agent maximize its expected utility by considering its own actions and random events (i.e. the opponents actions) and find the best average score path between the two options. I had used chatGPT to create the model in python, then would make additions so that logic it created matched what I was thinking.

The code for the Expectimax Agent could be summed up into three steps. First, at every move the agent would call the chooseAction(gameState) function which would take a look at all legal actions my agent could make, then use a recursive Expectimax search to estimate how good a move it was thinking about making. In the recursive search, the Max Nodes would evaluate which of my agents available legal moves had the best expected utility, then the Chance Nodes would take the average of the opponents expected moves. The recursive function would evaluate up to 2 potential moves in the future, then make a decision based on the returned value. At first I was having a hard time getting the Expectimax Model to work, as it felt like the heuristic weights were not affecting the performance of the agent as well as it was it becoming too difficult to encode weights that would allow my agents to take on separate duties that would grant me coverage across the whole environment on my team’s side.

I ended up switching to a Finite State Machine based model where I could set general expectations for specific operating modes I would like my agents to act in as well as further fine tune those modes to best fit the needs of the game. My FSM base class agent had three states to transition between: Offense, Defense, and Retreat. When troubleshooting my first agent, I had looked through the baselineTeam agent and noticed that it had a base class Reflex Agent then two other agents modeled after it that specifically meant to meet offense and defense needs. I took the idea into account and ended up creating an OffenseFSMAgent and a Defense FSM agent. Now with two agents I was able to further specify the transition rules: Offense Agent: If invader is visible = Defense If carrying 4 or more food OR if non-scared ghost is too close = Retreat Else = Offense Defense Agent: If invader is visible = Defense If carrying more than two food OR non-scared ghost is too close = Retreat If no invaders visible AND no ghost visible AND carrying little food = Offense

The FSM gave me a good basis to develop rules and think of what I could do to further optimize either agent, but I didn’t have a clear understanding yet of how the agent should be making these decisions. By this point I talked with my partner, Colin, on how our projects were going and I noticed he had been using A* search to help lock onto targets to focus/prioritize movement. With the help of chatGPT again I added Breadth-First search into the FSM. Now the FSM was deciding what the agent should do and the BFS was telling the agent how to get there. Basically, BFS would search outwardly, tile by tile through the maze. As soon as a target was found BFS would reconstruct the shortest path to that target, then the agent would perform an action based on its operating mode. For example in Offense mode the agent would find the shortest path to food. In Defense mode the agent would find the shortest path to an invader and in Retreat mode the agent would find the shortest path back to its territory.

With the BFS working within the FSM modes, both agents started working against the baseline agent much better and eventually beat it after a few trials. Now that the agent was done, I added a few more helper functions to further improve the agents functionality. In the beginning, for the offensive agent I had the weight for collecting food higher than the need to actually return it. To combat this I added the threshold of food to collect before the retreat transition can be forced. Next, both agents have some offensive and defensive capabilities. When an enemy would get too close to both agents, they would both head after the enemy which would cause the offensive agent to neglect its responsibilities. To fix this I added an invader assignment function that would compare the maze distance for the two agents and whichever agent was closer would defend against the enemy. Finally, when both agents were starting to do better with new improvements made, I noticed how when chasing enemies out, the offensive agent would run straight over the border as if the enemy ghost wasn't on the other side ready to attack. To update this process, I added some border safety logic that would simulate crossing the border after a chase. Basically, the agent would check if the successor state would make us Pacman and if it did AND there was a ghost in the "safeCrossRadius" the agent would refuse to cross and find another safer way around.

These additional functionalities further improved my FSM based agent and allowed for pretty competitive gameplay against the baselineTeam and CollinAgent.

B. Collin

When I started out with the project, I decided to go with a simpler rules based agent. Simpler being a relative term. Originally, I wanted to do 2 separate agents and have one stay on the defending side while the other went to play offense. I figured it would cover my bases and be a good distribution, but ultimately decided against it. Making a single versatile agent class would likely make my agents perform better. I had some behaviors I wanted for the agent and started by laying them out.

I wanted the pacman to deprioritize ghost distance until ghosts were very close. It would be vulnerable to flanking, but that way pacman won't just sit in the corner forever because leaving his side would mean getting closer to a ghost. I wanted my pacman to reduce distance to food, but wanted to outweigh this with the reward for actually picking food up. I remember back in the first pacman assignment, I had trouble with this one but ultimately did get it to work, so I would use some of that logic for inspiration I wanted my ghosts to have a body blocking behavior on the border of the two fields. If the opponent has something similar it might just cause a staring contest, but I still liked the behavior since some Pacmen would just walk straight into it (at least the baseline team would) If the agent is a ghost, it will prioritize chasing intruders before trying to cross over (I just weighted this behavior more heavily). I wanted to ignore the scaring system. If the enemy ghosts were scared, I wanted to keep avoiding them so I don't respawn them at full power. If my ghosts were scared, I wanted them to keep chasing pacman in order to get themselves respawned at full power. I wanted a counter that would make my Pacmen return to their own side with greater urgency depending on how much food they were carrying.

I got chatgpt to give me the initial implementation giving it the baseline team and capture agent files to work from. It threw some compilation errors for a bit, but they weren't too hard to fix. It was a little weird working with the get features method it made, but I was able to add in the eating ¿ proximity logic. From there it was just a matter of futzing with the weights, or so I thought. My agents would always get stuck oscillating and getting stuck in dead ends, so I realized I needed to implement a search. I had ChatGPT help me with that. To little success. Then I ran out of free queries.

then a new issue started showing up. my agents were now really good at defending, but absolutely terrible at offense. In fact, the most they would do is cross the border and go in circles until the enemy got near.The search was only meant to "unstick" the agent if it was stuck in an oscillation and start it on its way home, but I couldn't get the toggling logic to work right so I decided to rebuild it with the A* search having a more ubiquitous place in the logic. calculating every turn. Changes provided by GitHub Copilot which I used for the remainder of the implementation to avoid chat GPT's non-premium query limit.

This had favorable results. for the most part. My agents were now gathering and returning food properly, but brazenly with no regard for the ghosts at all as the ghost positions weren't taken into account whatsoever in the A* search. but it was performing much better, so I put that concern aside for now.But the agents weren't good at winning outright. When they would win, it would be when the timer ran down and they had more points. This was due to their overly offensive nature. Sometimes in one fell swoop, an opposing intruder would gather 18 food pellets and return them. My agents needed to be more defensive again. So I decided that the agent should keep track of how much food has been picked up by the enemy. if it crossed a certain threshold, both of my agents would go on the defensive. returning with whatever food they had and weren't allowed to cross onto enemy lines. this still required refinement. they would stay too close together. then they would still hug the border. But, they were still performing a little better defensively. At least against the baseline team.

This strategy resulted in 9/10 wins against baseline team with 1 loss (on the default map)

still had trouble because the A* search I implemented didn't recognize ghosts. So I added the behavior to treat ghost positions as walls in the A* method. Not a perfect solution, but good enough as I was running out of time. I went with A* because it would perform consistently better or faster than BFS and DFS, given it's complete and optimal (we have Manhattan distance built in which will never overestimate and as a result, always be admissible). The calculation doesn't expand extra unnecessary nodes because it will avoid expanding paths that are already expensive, and that makes it a good all around search performing quickly while still giving optimal paths.

notes: after the implementation of A* search by GitHub copilot, the logic I used to see dead ends and return home was never used due to the A* search implementation could deal with dead ends without falling back to weighted logic. As a matter of fact, getFeatures and getWeights were barely used at all, mostly just relegated to fallback logic. The weight and feature based logic only fires if the invader defense logic fails, or the agents aren't in "defense mode", the agents aren't carrying food themselves, and there is no path to enemy food.

I noticed that the defense mode behavior could get stuck. I had it where if the enemy team stole enough friendly food, my agents would go on the defensive, but if the enemy successfully snuck by and returned the stolen food to their side, my agents would never leave defensive mode since the friendly food amount would never go back up. So I started keeping track of the score when in defensive mode and if the enemy team's score went up, it meant they successfully escaped with our stolen food and we could leave defensive mode.

That's everything. Basically, every turn the chooseActions method runs A* search based on a series of goals. And whichever of those goals were focused on that turn determined the goal of the A* search. Then it would run the A* search to find the shortest path to that goal (treating ghosts as walls)

If an invading pacman is visible, get the position of the closest one if an agent is on defense, it will always chase the invader if an agent is on defense but can see an invader near the border, it will try to intercept (body blocking) does an A* search if the agents decide to defend if the enemies have eaten enough of our food, all agents are put in defensive mode where they seek out the border using A* if agents are carrying more than 3 food, they path home to return it and get the points using A* otherwise default to using weighted features.

Ultimately, the Collin Agent is mostly a reactive agent, getting the board state every turn and choosing actions based on what it can observe each turn. It doesn't have memory. You could argue that it has qualities of a deliberative agent with the A* search, but because it calculates a path every turn, it often won't stick to plans at all. It could make decisions very quickly but made them with no regard for the future or the past.

IV. Results

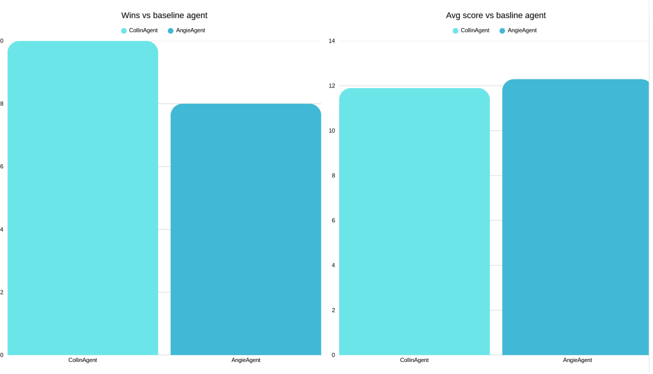
A. VS baseline agent

Final Score vs Game Number

	Game 1	Game 2	Game 3	Game 4	Game 5	Game 6	Game 7	Game 8	Game 9	Game 10
angie Agent	-6	17	18	17	14	18	17	16	-2	14
collin Agent	12	10	12	12	16	8	18	16	7	8

*Data collected from agent against baseline agent on the default environment.

Agent	Min Score	Max Score	Average Score	Win:Loss
angieAgent	-6	18	12.3	8:2
collinAgent	7	18	11.9	10:0



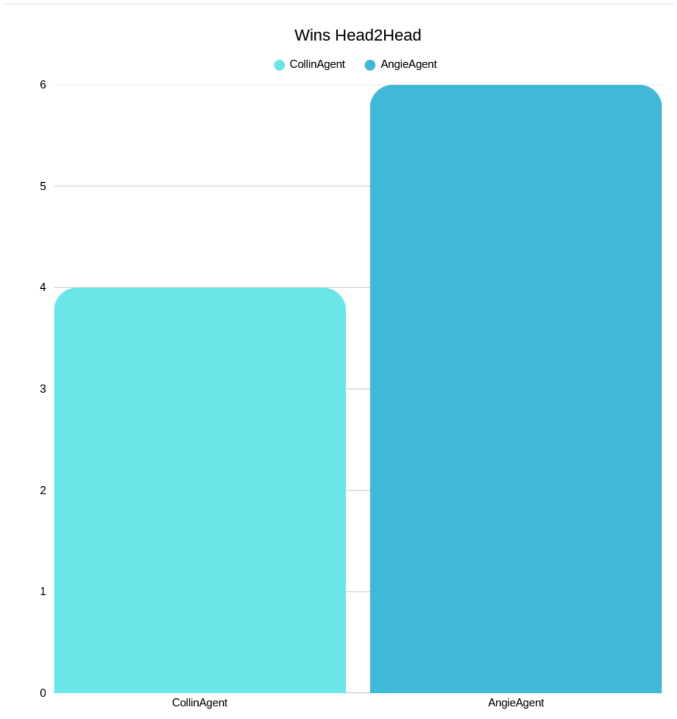
B. Head 2 Head

*Data collected from collinAgent against AngieAgent on the default environment.

Agent	Min Score	Max Score	Average Score	Win:Loss
angieAgent	-7	18	4.3	6:4
collinAgent	-18	7	-4.3	4:6

Final Score vs Game Number

	Game 1	Game 2	Game 3	Game 4	Game 5	Game 6	Game 7	Game 8	Game 9	Game 10
angie Agent	-3	7	9	-7	5	-2	18	-2	5	13
collin Agent	3	-7	-9	7	-5	2	-18	2	-5	-13



V. Some Common Mistakes

A. Observations

Through testing, it became obvious that the AngieAgent would return when threatened or when it reached a certain amount of food carried. The amount of food carried threshold was less than the amount of food stolen threshold that would trigger the Collin agent's forced defense mode, meaning that Collin agent invaders would never return home to defend.

During Head2Head game 4, ¾ of the agents got stuck in an oscillation loop. This stalled out the game timer. This likely resulted from the CollinAgent seeing ghosts as walls and trying to path around them constantly.

Both CollinAgents have the tendency to enter into enemy territory through the center of the map rather than one around. This often allowed for the Offense AngieAgent to get around into the opposite territory where the Defense AngieAgents could take out the CollinAgents before they had the opportunity to block. Though, once at a certain spot on the border CollinAgent's body blocking behavior fed into stalemates where neither team was willing to cross the border.

B. Collin Agent Possible improvements

In its current state, the collinAgent will seek the border when in defense mode instead of searching for the intruders or covering food positions. I could add conditions to the chooseAction method that prevent this.

Rather than having static return and defense thresholds, the threshold should be a fraction of the current amounts of food on either side of the field.

Include actual ghost detection / avoidance in my chooseAction method instead of just treating them as walls in A*

C. Angie Agent Possible Improvements

Currently my offense agent has slightly better rules and conditions for making actions. Because of this my offense agent sometimes outperforms my defense agent when enemies enter my territory. To improve the defense agent, I think I would create some type of probability distribution on the enemy's location (if they're not already visible) anytime food disappears in my territory. They way defense agents can patrol areas where food was recently eaten, most likely putting them in the enemy's path. This fix would allow the defense to better predict where enemies may be instead of blindly patrolling the border.

One improvement for the offensive agent would be to add a threshold to the retreat state, where instead of only retreating when the agent has picked up enough food or feels threatened it can use the time or score as a factor. If time is low and the score is leading the agent should be able to make the decision to retreat more aggressively rather than risk offense but also if time is low and our score is lacking we could force an offense state and push for more food (while also increasing the carrying threshold).

VI. Conclusion

Overall, both agents demonstrated meaningful progress in navigating the strategic complexity of the Pacman Capture the Flag environment. Each agent reflects on different AI design philosophies covered in the duration of the course. CollinTeam relied on two identical versatile reaction agents equipped with A* search pathfinding whose dynamic behavior would theoretically increase their effectiveness, when in reality, this just meant both agents would hit the same pitfalls despite its strong performance against the baseline team. AngieAgent, driven by a structured Finite State Machine, enhanced with Breadth-First Search, proved highly effective due to its clearly defined behavioral modes and transition logic. This agent maintained consistent performance against the CollinAgent as observed in our data, though had a slightly harder time consistently out performing the baseline Agent provided. Together, the agents showcased different yet effective methods for the multi-agent environment and provide valuable insight into how high-level decision models and low-level pathfinding complement (or conflict) with each other in dynamic environments.