PactureFlag

PROJECT REPORT – WINTER 2011 – CS221

Emma Pierson, Nandu Jayakumar, Nicholas Hippenmeyer

Stanford University

1 Introduction

This report details the methods used and approach taken to program agents to play a variant of Pacman that is structured as a multi-player game. The goal of the team of agents is to eat as much food as possible on the opponent's portion of the map while defending food on their portion of the map. Our approach to the problem involved experimenting with various AI techniques/algorithms and weighing the benefits and shortcomings of each. We broke up the problem into developing defensive and offensive agents separately, and then combining elements of each as a final step. Through an iterative design process and rigorous testing, we arrived at a solution that performed well the majority of the time.

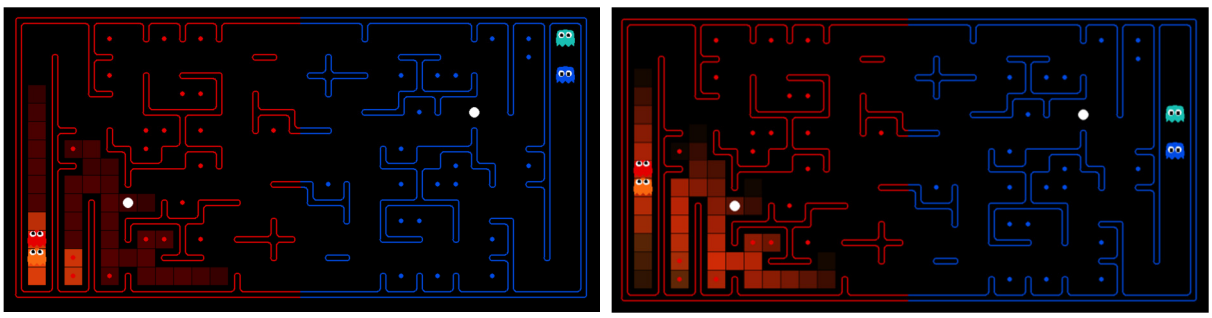
2 Approach

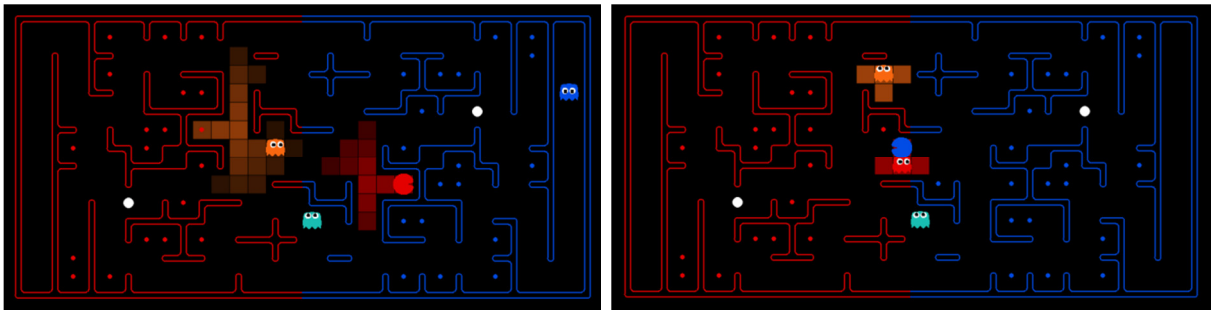
2.1 Development of the Custom Defensive Agent

The first step in developing a defensive agent was to implement probabilistic inference for the opponents when they were out of sight range. Two inference functions were implemented:

- *observe*(self, observedState):
  Inference observation function: Combines the existing belief distributions with the noisy distances measured to each enemy agent and updates the distributions accordingly
- *elapseTime*(self, observedState):
  Inference time elapse function: Updates the belief distributions for all enemy agents based on their possible moves and the likelihood of each move

The above functions are called every time an action is chosen for the defensive agent. Belief distributions are stored and updated for every opponent on the map. As the opponents move around, the accuracy of these beliefs improves, and as they move within sight range the uncertainty disappears. Below are a few screenshots of progressive agent belief distributions:





The above inference modelling was used to determine the most likely position for each opponent so that that the defensive agent could move move towards the closest enemy attacker. If no opposing attackers were located on our side of the map, the defensive agents would move towards the center without crossing into enemy territory and anticipate where the closest opponent would cross into our territory. This approach worked well and several tests were run to determine under which circumstances the strategy performed poorly.

The next iteration in this approach involved assigning specific opponents to each defensive agent so that not all agents would chase the same attacker. This modification drastically improved results where multiple defensive agents were used on a team. Finally, the decision for a defensive agent to move towards the closest attacker was changed to moving away when the agent was scared (i.e when the opposing team had consumed a capsule).

2.2 Development of the Caesar (Offensive) Agent

After learning that our agent for Assignment 4 had been one of the best in the class, we decided to use it as our starting offensive agent. The design was simple and intuitive: we used a reflex, feature-based agent that would approach the closest dot (measured by maze distance), avoid ghosts, eat capsules, and pursue scared ghosts. We considered including "nearest-ghost-distance" as a feature, but decided against it both because it had not proved effective in Assignment 4 and because the distances were noisy.

To determine the weights given to each of these goals, we implemented feature-based Q-learning. We started with approximate initial weights—repugnant, perhaps, to lovers of pure Q-learning, but essential if the agent was to learn within a reasonable timeframe. Without an initial hint, it would take forever to even make it to the other side of the board.

We found Q-learning ineffective because it was impossible to overcome a fundamental difficulty: the rewards the agents received were too rare to allow for effective feedback. (The original Pacman problem was ideal because of the frequent feedback.) We tried a number of methods to overcome this difficulty. Running very long training periods proved ineffective both because it took too long and because the agents just got worse and worse. Increasing the learning rate so the agent would learn more rapidly from the few rewards it *did* gain often made the weights diverge. It also seemed unlikely that the agent would be able to learn cooperative behavior. For these reasons we concluded that human intuition, and patient tinkering, produced more effective weights than Q-learning. This is supported by the fact that it's not difficult to rank an offensive agent's priorities: first, avoiding ghosts, second, eating adjacent ghosts, third, eating adjacent food, and fourth, going after nearby food.

We set the weights to reflect these priorities, and then tackled the problem of getting the agents to cooperate—to spread out and not pursue each other's dots. This proved surprisingly difficult to accomplish. We tried assigning a "distance-to-nearest-teammate" feature with a positive weight, but this on its own proved insufficient to spread the agents out. If we set the weights high enough to make the agents spread, we also got suboptimal behavior (eg, one agent staying at the starting position to maximize distance from its teammates). We tried combating this by weighting the feature with the distance from the starting position—eg, the farther the agents traveled, the more

spread out they would try to be—but this suffered from the same problems as the original formulation[1]. We tried weighting the feature by agent index—so the lower indexed agents would follow their original paths, and the higher ones would get out of the way—which was more effective. The most effective approach, however, was to create a "zone" offense—to assign each of the offensive agents a half or third of the board, and have them only pursue dots in that sector. (Once they had consumed all the dots in their sector, they went after other sectors.) We tried generalizing this method, by assigning a small weight to food in other sectors rather than completely ignoring it, but this was beaten every time by the naïve zone offense[2]. We also left in the "distance-to-nearest-teammate" feature, which spread out the agents when they happened to be in the same zone.

Having optimized our offense, we tackled the question of how to combine it with defense. We explored the idea of combining our offensive and defensive agents into a single "super-agent", but ultimately rejected this idea for several reasons. First, the offensive and defensive agents were structured differently—the offensive agent was feature based, while the defensive one was not—and restructuring them and setting appropriate weights would've been difficult. More importantly, role differentiation is often optimal in both natural and artificial multi-agent systems. Workers in ant or bee colonies, for example, have differentiated roles assigned from birth; similarly, the NERO multi-agent system, developed at the University of Texas at Austin, uses agents with differentiated roles. Within Pacman, it seemed better to have agents that prioritized offense or defense rather than vacillating between the two. As a soldier on the front lines is less effective if he's worrying about his family at home, an offensive Pacman is less effective if he's worrying about his dots at home.

At the same time, however, there are certain circumstances where it is clearly optimal for the offensive agent to behave defensively. We incorporated them into our offensive agent to produce a "super-agent" that prioritized offense. When the offensive agent is a ghost, it is clearly optimal to eat enemy Pacmen who are immediately adjacent to it. It is less clear whether the offensive agent ought to *pursue* the enemy Pacmen if they aren't in immediate range. To determine which was optimal, we created two versions of our superagent, one of which pursued (Caesar1) and one of which did not (Caesar), and had them compete under a variety of circumstances. The results, summarized below, were ambiguous. We initially believed the pursuing agent was better, because it outperformed the other agent when teamed with defensive agents. We discovered, however, that the combination of three non-pursuing Caesars destroyed any team with defensive agents. We therefore tested various combinations of pursuing and non-pursuing Caesars, and found that the team with two pursuing Caesars marginally outperformed the other two. This became our final team.

The defensive agents were outperformed because they had a harder job in two respects: it's easier to escape than to catch up, and it's easier to hide than to seek. The large amount of noise made the second aspect particularly challenging, although in future iterations of the project the defensive agents would benefit from triangulation: combining the individual agent inferences to reduce noise. Our final team performs well because it emphasizes the easier task of offense while still retaining rudimentary defensive capabilities. In addition, because the Caesar agents ignore the noisy distance readings, it is not affected by noise.

2.3 Reflex Agent vs Adversarial Search

Once again, based on our success with adversarial search in the fourth assignment and the fact that enemy agents would react to the environment as well as our agent's actions, we decided to implement minimax/expectimax agents in order to match well against opponents. We found that even with a depth-limited search there were times when the expectimax search would take a long time to compute causing the multi-player game to stall. This

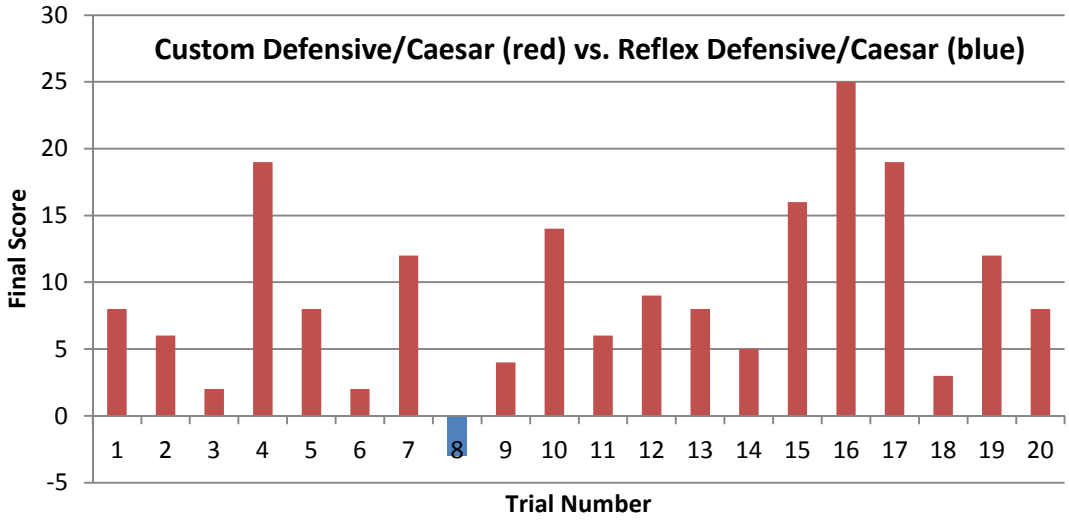
[1] All of this analysis is qualitative as opposed to quantitative because any numbers provided would be trivial—it was immediately obvious from watching a game that the agents were suboptimal.

[2] This is intuitively odd, since the small weight method is a generalization of the naïve method (which just sets the weight to zero). On the other hand, adding more features randomly is also a "generalization" which may undermine the agent's efficacy.

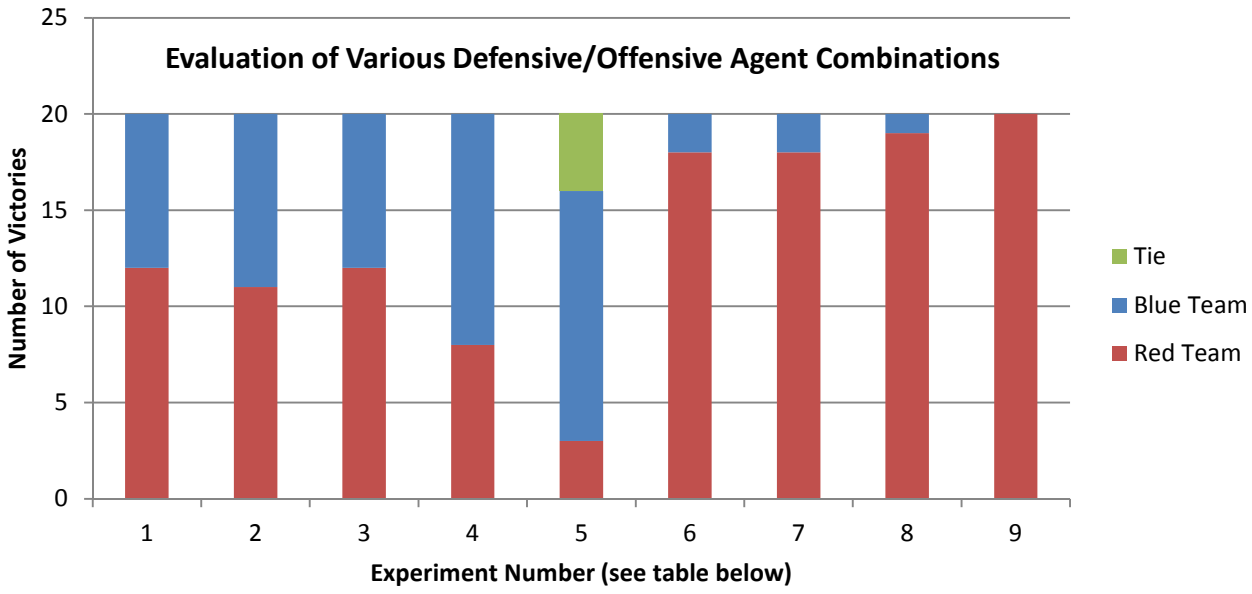
seemed to happen even with a depth-limited search restricted to two levels. Additionally the position of enemy agents in the map is not fully observable to the team of agents especially when far from enemy agents. We considered a conditional implementation that would only go into effect when enemy agents were closer, but ultimately opted to not use the adversarial search implementation due to the computation time issue.

3 Results

The graph below shows the results of 20 trial runs where a Caesar agent was paired with a first-iteration Custom Defensive agent on one side and a Reflex Defensive agent on the other. The victory rate for the Custom Defensive/Caesar agent team was 95%, a very promising result. With the implemented modifications outlined in section 2.1, the success rate improved to 100%.



This next graph illustrates which combination of agents performed optimally:



Experiment Number	Red Team	Blue Team
1	3 x Caesar	3 x Caesar1
2	3 x Caesar	1 x Caesar, 2 x Caesar1
3	3 x Caesar	2 x Caesar, 1 x Caesar1
4	1 x CustomDefense, 2 x Caesar	1 x CustomDefense, 2 x Caesar1
5	2 x CustomDefense, 1 x Caesar	2 x CustomDefense, 1 x Caesar1
6	3 x Caesar	1 x CustomDefense, 2 x Caesar1
7	3 x Caesar	1 x CustomDefense, 2 x Caesar
8	3 x Caesar	2 x CustomDefense, 1 x Caesar
9	3 x Caesar	2 x CustomDefense, 1 x Caesar1

Despite the custom defensive agent significantly outperforming the reflex defensive agent, from the series of tests run we were able to conclude that the optimum team consisted of 2 Caesar1 and 1 Caesar agents. The reasoning for this is outlined in section 2.2. Even 1 defensive agent combined with 2 offensive agents was significantly outperformed by 3 offensive agents. Both the Caesar and Caesar1 agents were designed to include defensive tendencies while acting as ghosts, and these characteristics definitely contributed to their superiority over the strictly defensive agents.

4 Discussion - Open Problems/Future Work

4.1 Early/Mid/End-game strategy

- We considered the example of chess and also the factor that the multi-player games would limit the number of moves per agent to 500. We wondered if agents should change roles or their entire strategy based on time. We could have chosen a strategy that switched between offense and defense based on how far the game had progressed. This could be coded in as a feature with appropriate weights and be used as an overall governing strategy that switches roles for the agents. This is definitely one of the strategies we would have experimented with further if we had the time to.

4.2 Conclusion

The iterative design process we followed to come up with our 3 final agents proved to be quite successful in achieving well-performing agents. Ultimately we settled on eliminating the defensive agent and integrating defensive elements into our offensive agents as a compromise. This yielded the best results overall. Unfortunately, various techniques we experimented with turned out to be unsuccessful and we had to focus on the ones that led to better performance. Other considerations such as complexity and computational time presented challenges in which we had to examine trade-offs and make compromises in various areas.