Pac-Man Capture the Flag

Project Report – CS221 Autumn 2011 Stanford University

Team Members: Navin Kolambkar, Harshit Chopra, Tim Chang, Alex Dea Vaccaro

1. Introduction

This document describes the strategies we used to implement a multi-player capture-the-flag variant of Pac-Man, where agents control both Pac-Man and ghosts. Each team tries to eat the food on the far side of the map, while defending the food on the home side. We implemented a few AI techniques based on concepts from the course and chose the algorithms which gave us the best performance based on a matchup with the Test Agent provided to us. We had to consider the tradeoffs imposed by the constraints of the problem in our agent design. Our final team consisted of 2 offensive agents and 1 defensive agent that has a good win/loss ratio against its opponent.

2. Estimation of Opponent's Position

The defensive agent uses Bayesian probabilistic inference about its opponents to choose the best action to take in a particular Game state. Similar to Assignment 2, the following functions implement the inference algorithm to do the tracking.

A) The observe method updates the agent's belief distribution over each opponent positions based on the noisy distance readings and the current belief distributions. The probabilities are calculated using Bayes' rule:

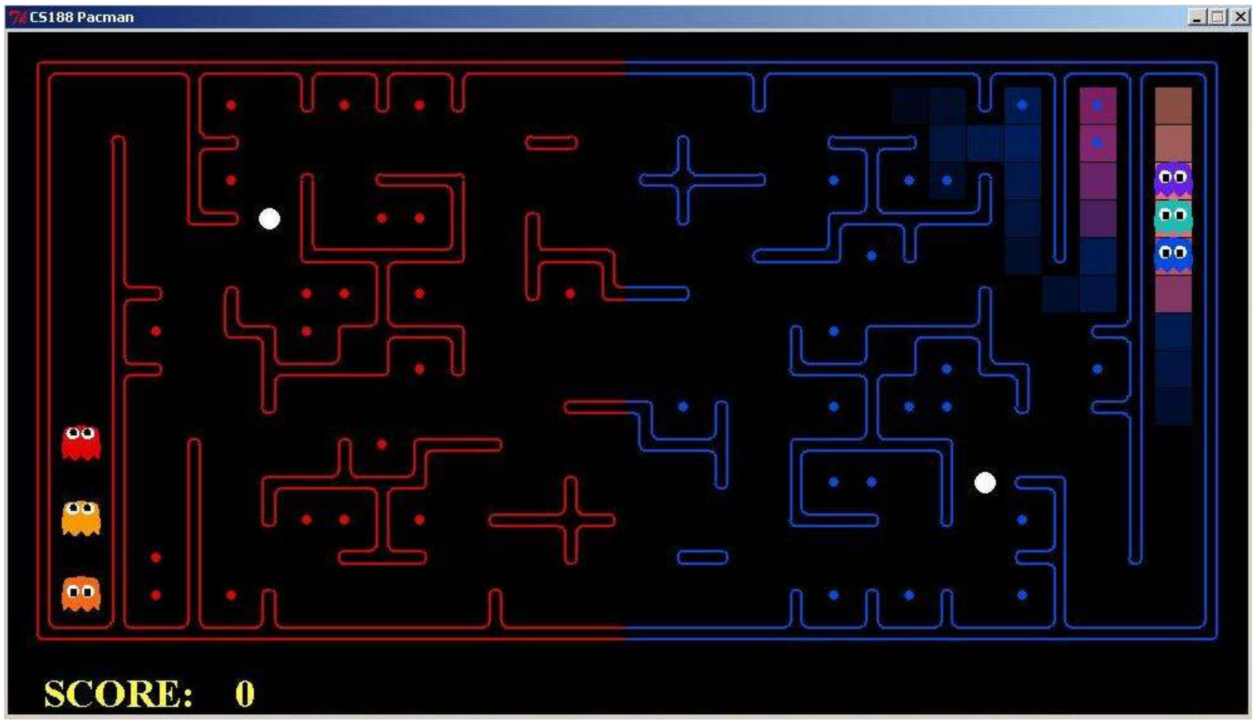
$$P(enemyPos_T | myPos_T, \ldots, myPos_1, n_T, \ldots, n_1)$$
$$= \frac{P(n_T | enemyPos_T, myPos_T, \ldots, myPos_1, n_{T-1}, \ldots, n_1)\, P(enemyPos_T | myPos_T, \ldots, myPos_1, n_{T-1}, \ldots, n_1)}{Z}$$
$$= \frac{P(n_T | enemyPos_T, myPos_T)\, P(enemyPos_T | myPos_T, \ldots, myPos_1, n_{T-1}, \ldots, n_1)}{Z}$$
$$= \frac{P(n_T | distanceToEnemy)\, P(enemyPos_T | myPos_T, \ldots, myPos_1, n_{T-1}, \ldots, n_1)}{Z}$$

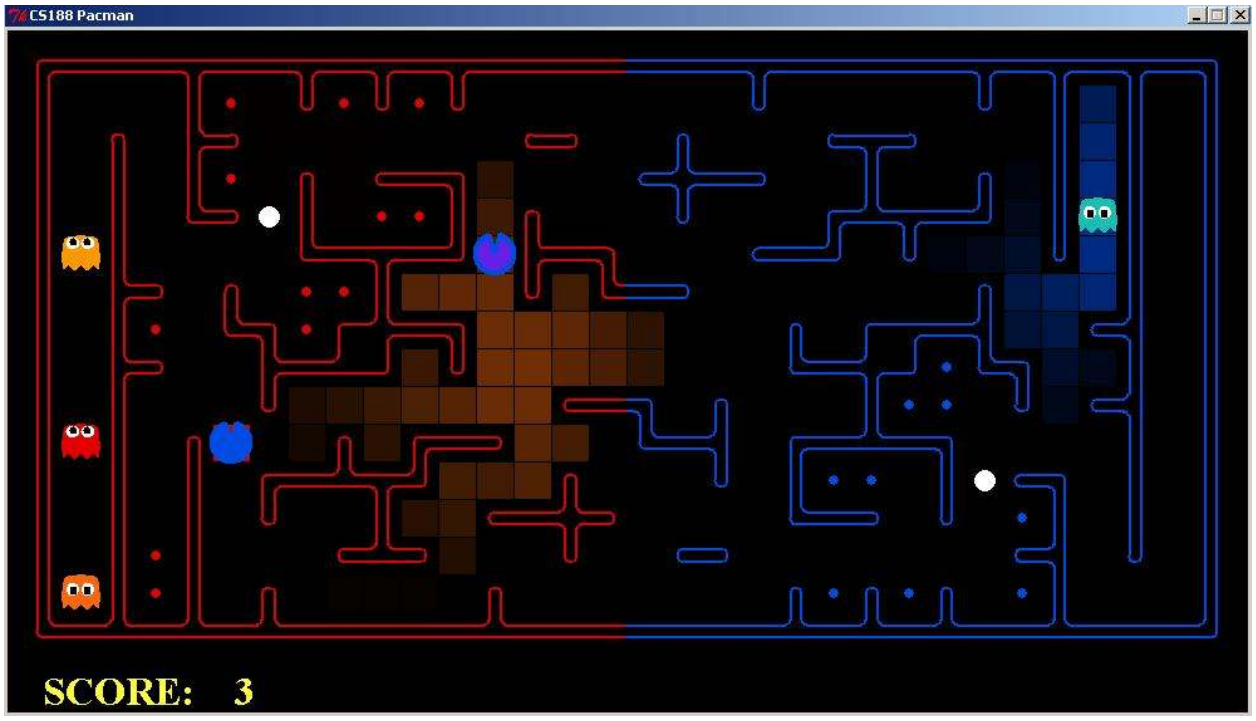
where Z is the normalizing factor and n_t is the noisy distance measurement at time t.

B) The elapseTime method updates the agent's belief distribution over each opponent positions in response to a time step passing from the current state. The formula is based on Bayes' rule:

$$\begin{aligned}
& P(enemyPos_T \mid myPos_T, \ldots, myPos_1, n_{T-1}, \ldots, n_1) \\
&= \sum_{enemyPos_{T-1}} P(enemyPos_T, enemyPos_{T-1} \mid myPos_T, \ldots, myPos_1, n_{T-1}, \ldots, n_1) \\
&= \sum_{enemyPos_{T-1}} P(enemyPos_T \mid enemyPos_{T-1}, newPos_T, \ldots, newPos_1, n_{T-1}, \ldots, n_1) \\
&\qquad P(enemyPos_{T-1} \mid myPos_T, \ldots, myPos_1, n_{T-1}, \ldots, n_1) \\
&= \sum_{enemyPos_{T-1}} P(enemyPos_T \mid enemyPos_{T-1}) P(enemyPos_{T-1} \mid myPos_{T-1}, \ldots, myPos_1, n_{T-1}, \ldots, n_1)
\end{aligned}$$

The accuracy of the belief distributions improve over time until the opponent is within 5 squares (Manhattan distance) when its exact location can be observed exactly.





3. **Pacman Strategy**

Each agent is either a defensive agent or offensive. Defensive agent's main job is to defend its food, while offensive agent's main job is to eat opponent's food. For this, each kind of agent has certain features to reflect its mode (i.e. offensive or defensive).

A) **Defensive Agent Features**

The defensive agent uses the following features:

- *xRelativeToFriends* – This reflects the x-axis distance of this agent with respect to other agents.
- *avgEnemyDistance* – to capture the average distance of opponents from this agent. The agent's belief matrix is used to guess an opponent's most likely position. A defensive agent should be close to the opponent to scare it away
- *percentOurFoodLeft* – to capture percentage of our food left.
- *percentTheirFoodLeft* – to capture percentage of food left at opponent's

- *enemyPacmanNearMe* – to capture if an enemy pacman is near the agent. The agent should try to chase and scare the enemy pacman.
- *numSameFriends* – to capture how many agents are present in the grid as itself.
- *blockableFood* – this feature is active if there is a food pellet which can be blocked for an enemy pacman
- *onDefense* – to capture the fact that this agent is on defense and should never become a pacman
- *numInvaders* – to capture number of invaders that this agent can see
- *invaderDistance* – to capture the closest invader which this agent can see
- *invaderNextDistance* – This feature captures the closest visible invader, the next position of which is guessed using a heuristic. The heuristic guesses the next position of an enemy pacman assuming that all enemy pacman always approach the closest food.
- *likelyEatEnemyPacman* – This feature captures whether to eat the enemy pacman. This feature uses the next position of the enemy pacman, which is guessed using the heuristic (described above).
- *invaderApproxDistance* – This feature calculates the distance to the closest invader when the invaders exact position is not available.
- *chosenEnemyDistance* – This feature calculates the distance to the closest pacman which is closest to all food.
- *stop* – This feature is active when an agent is stopped. A defensive agent shouldn't stop.
- *reverse* – This feature is active for reverse actions. A defensive agent shouldn't reverse its direction in most cases.
- *avgDistancesToFood* – This feature captures the agent's average distance from the food it is defending. It should be close to the food.

B) **Offensive Agent Features**

An offensive agent uses the following features:

- *forbidden* – This feature is used to reflect forbidden states. A forbidden state is when the opponent ghost's position is the same as my position
- *stopped* – to reflect stop actions (which are not desirable)
- *closestVisibleGhost* – Captures the distance to the closest visible ghost
- *stoppedWhenGhostNearby* – to reflect stopped actions when a ghost is nearby (extremely undesirable). The motivation behind this feature is that it is better to move somewhere hoping that the other agent takes a sub-optimal move, rather than just stopping.
- *closestGhostNextDistance* – Captures the distance to closest ghost, after guessing the next position of each visible ghost. The heuristic used to guess a ghost's next position is that the ghost will try to decrease the distance to this agent.
- *#-of-ghosts-1-step-away* – Captures the number of visible ghosts which can reach this agent in 1 step.
- *eats-ghost* – This feature is active if the ghost can be eaten (i.e. it is scared).
- *#-of-scared-ghosts-1-step-away* – Captures the number of scared ghosts 1 step away from this agent.
- *holedInWallsWhenGhostNearby* – This feature is active if the agent is holed (surrounded by walls on 3 sides) when an enemy ghost is 1 step away.
- *surroundedByWallsWhenGhostArround* – This feature is active if the agent is surrounded by walls (i.e. it can't escape if a ghost is at its tail) when a ghost is nearby. This feature is different from the above feature in the sense that the agent need not be immediately surrounded by walls on 3 sides, but will be if it tries to go ahead a few steps (the algorithm limits search to 3 steps). This feature tries to discourage moves where there is no escape for the pacman.
- *canEatCapsule* – This feature is active if the agent pacman can eat capsule, which is when a ghost is at its tail.
- *closestReachableCapsule* – When a ghost is at the agent's tail, it captures the distance to the closest reachable capsule (i.e. no ghost should be able to reach the capsule before this agent).
- *eats-capsules* – This feature is active if a capsule is available at the next position.

- *num-food-left* – Captures the amount of food left to eat. The agent should aim to decrease it.
- *eats-food* – This feature is active if a food pellet is available at the next position.
- *closest-food* – This feature captures the distance to the closest available food, which no other offensive agent is trying to eat. Calculation of this feature is done using Uniform Cost Search.
- *enemyPacmanClose* – This feature is active if an enemy pacman is nearby and this agent is a ghost. This feature (and the features below) infuse defensive capabilities into the agent.
- *distanceToEnemyPacman* – Captures the distance to the closest visible enemy pacman when this agent is a ghost.
- *enemyPacmanNextDistance* – Captures the distance to the closest visible enemy pacman, after the position of each visible pacman is guessed using the heuristic. (the same heuristic used by the defensive agent).
- *eatEnemyPacman* – Captures whether to eat an enemy pacman 1 step away.
- *likelyEatEnemyPacman* – Captures whether to eat an enemy pacman which is 1 step away and whose next position is guessed using the heuristic.

The last 5 features give some defensive capabilities to the offensive agents so that they can eat invaders on the way to enemy side of the board.

C) **Planning**

For each of these features, we initialized the default weights manually after experimenting with them and the ones which gave the best results. We tried to use QLearning to converge the weights for best performance, but learning worsened performance after a few games. We experimented with different rewards for each action, different rates of learning, different values of initial weights and features, but nothing improved the agents' performance. In the end, we decided to just use the hand written default weights, which tend to give the best performance.

The most likely reasons that QLearning didn't help at all here are:

- The complexity of the game. The game is complex and it is not possible to visit each state to have good convergence and good learning.

- The actions which lead to rewards are not obvious to the agent. A seemingly random action can lead to a reward (like an innocuous move to a random position just before other agent ate food leading to a win). This can cause confusion and wrong updates of weights.

- There are certain features which are not related to the action leading to a reward. This can cause those features to be attributed to the reward (and hence to the action), resulting in wrong learning. The only way to mitigate this affect in traditional QLearning is by visiting each possible state, but which requires infinite trainings.

D) **Implementation**

Each agent is assigned an initial mode (i.e. defensive or offensive). We use 2 offensive agents and 1 defensive agent for the game play. Depending on the agent's mode, it uses a different set of features (described before). The function getDefensiveFeatures() returns the set of features for a defensive agent for given gameState and action. Similarly getOffensiveFeatures() return the set of features for an offensive agent.

The function getEnemyLikelyNextPos() returns the next position of a visible opponent using the following heuristic:

- If the enemy is a ghost, its next position will be such that it will try to decrease its distance to this agent.
- If the enemy is a pacman, it will try to eat the closest food.

The class ClosestFoodWithoutOpponent implements the search to find the path to the closest food which doesn't involve any visible opponent. Moreover, no two offensive agents try to approach the same food.

If a defensive agent is scared, it changes its mode to offensive. To have at least one defensive agent, without interrupting any offensive agents in their task, if all agents are offensive, and an agent happen to turn into a ghost (either because it got eaten or it came back to home territory), it is converted to become a defensive agent.

E. **Performance**

The above technique works quite well with the given BaselineAgents. It beats the BaselineAgents almost all the time. If a defensive agent is scared, converting it to become offensive helps, since sometimes the opponent pacman choose not to eat the scared ghost, which renders the ghost useless (since it can't scare the enemy pacmans)

Moreover, assigning each offensive agent to different food pellets help distribute each offensive agent to different part of the enemy's territory, resulting in efficient capture of food. Avoiding a visible ghost to reach the closest food helps the agent make better decisions on where to move while avoiding the ghost.

However, these features don't help the agents all the time. For example, the defensive agent sometimes get confused as to which enemy pacman to chase, if both are equally far away. I tried adding a couple of features to make it incline towards chasing the pacman, which is nearest to most food, but it didn't help much.

An offensive agent sometimes surrounds itself within walls when a ghost is nearby, specially when there is some food to be found within the walls. This is a conscious decision from our part since eating food has higher priority than trying to get away from the ghost. However, it can be improved.

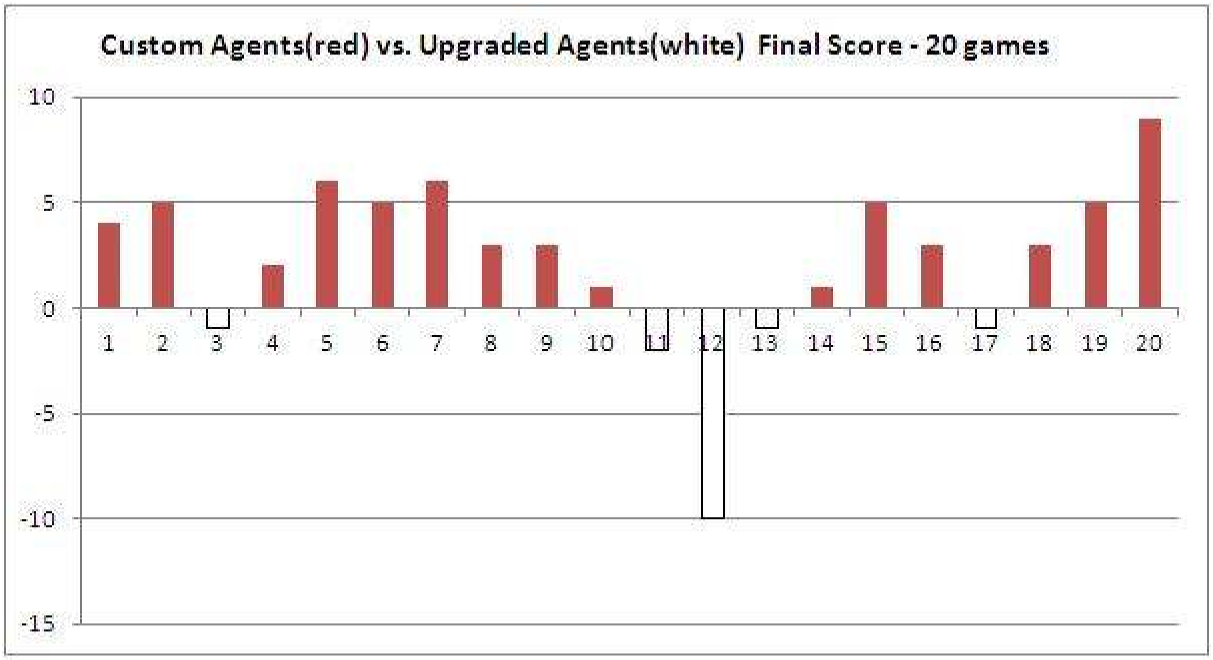
The algorithm to find the closest food employs a greedy approach, which obviously may not be the most efficient. This results in the agent vying for the closest food, rather than choosing the optimal path to eat the closest set of foods.

Sometimes, two agents from opposite sides get involved in a deadlock (repeating the same moves endlessly). This is fine if ours is a defensive agent since it prevents the enemy pacman from eating further. However, it would be much better if it can eat the enemy pacman and focus on other enemy pacmans. This kind of deadlock is not food for the offensive agent though.

4. Game Theory Evaluation

Results

We competed against the UpgradedAgents team in best of 3, and best of 20 matches. For the former, our agent won 2 times out of 3, and for the latter, our agent won 17 times, giving a win percentage between 65-85%. We found that offensive behavior generally gave us better results. A team combination of 2 Offensive and 1 Defensive agents was more successful than a team combination of 1 Offensive and 2 Defensive agents. We also found that adding defense oriented features to the offensive agent also helped in general along with conversion of defensive agent to offensive when it is scared and hence useless as defensive agent.



Future Improvements

The current technique uses a bunch of features with manually assigned weights. While this may be good enough for the BaselineAgents and UpgradedAgents, it may not be good for a better agent. Some improvements that can be employed are:

1. Using a better variant of QLearning or a better learning algorithm.
2. Using better features and heuristics in general. For example, a better feature would choose the closest set of foods which result in more points with minimum action. Better heuristic to guess an enemy agent's next position.
3. Learning enemy agent's behavior to adapt to the current enemy.
4. Using better tactics to avoid an enemy ghost and/or get rid of a chasing ghost.
5. Better defense features/capabilities for an offensive agent.
6. The current defensive agent barely manages to defend. A good defensive agent which uses good features and heuristics to defend will help improve performance.
7. Resolution of a deadlock between an agent and enemy agent.