Jake Shapiro
Vincent Wang

Team Jincent Swang

We split up our units into two different classes for offensive and defensive respectively and assigned one bot to each upon the creation of our team.

Defensive unit

Problems

The biggest problem to solve for this bot was more or less giving it the best possibility to catch invaders before they do any damage or if they do happen to succeed in eating food find them as as possible before they score any more points.

Modeling

We realized this problem could be modeled into 3 states -> our pacman sees an enemy, an enemy is in our territory eating our food, or there are no enemies currently visible. We made sure to have a solution for each of those states accordingly in our bot. As such we our bot has three behaviors: move to seen ghost, move to last eaten food, or patrol for ghosts near the center of the map.

Strategies, Algorithms, and Obstacles

There are various conditions that check which state the ghost should be in at a given time. If the enemy is within vision of our ghost we obviously enter our attack state and move towards him. Otherwise, if we notice that one of our foods has been eaten by using a variable that stores the last seen food and comparing it with a call to the current food, we use numpy to find the coordinates of the last point eaten and move to it asap. What we could have done here was use mazeDistance and a timer to determine if points were being attacked by different agents and if so move to just one side first instead of hovering back and forth between them between the two pacman constantly eating food, however since many players were not sending two offensive agents and the ones that did weren't necessarily sending them to opposite sides of the map it wasn't practical to implement this with our given time. If while in path to the point the ghost finds the pacman, we move back into attack mode and set our lastEatenFood variable equal to None. If we reach the point without seeing the ghost we also set the lastEatenFood variable to None.

If there are no attackers and lastEatenPoint is None, we move into our Patrol state. We originally had the idea to use Astar to loop through all food in our pacman's grid in our register initial state to find the least cost path that the pacman needed to take to have seen all the food from within a range of 5 and then continually loop that path. However after closer inspection we realized that although this would be cool, it was a little inefficient. Why waste time guarding only some of the food when you can guard all the food? We realized that on the vast majority of maps there is often a very clear straight line towards the center of the map that would make a very easy patrol path. So we decided to try and find that path. In register initial, we loop through a set range of points near the midpoint and choose a bottom and top point if the point is not in a wall. From there calculate which top, bottom point pair has the best pathcost using mazeDistance and store these two points in a list. Although relatively simple at a surface perspective, this patrol functionality proved to be extremely effective at blocking and detering enemy intruders. Since it was near the midpoint and usually a very low cost traveling between the two points, our agent was effectively able to guard all the pieces at once and make quick, decisive patrols allowing it to easily catch invaders before they could do any damage.

Evaluation

We are able to get our bot to do what we want by using the product of features and weights in our evaluation function. We based it off the baseline team's default weights and features. The weights were {'numInvaders': -1000, 'onDefense': 100, 'invaderDistance': -10, 'stop': -100, 'reverse': -2} where the feature numInvaders corresponds to the number of enemy pacmen that are visible to the bot, onDefense is binary and is 1 if the bot is not pacman, invaderDistance corresponds to the mazeDistance from the agent to his objective, and stop/reverse just prevent the agent from stopping or reversing when it doesn't need to. Our states manipulate

invaderDistance in different ways, and if our state is in patrol we set invaderDistance to the mazeDistance between our unit and the next patrol point. In our move to food state we set our invaderDistance to the mazeDistance between our unit and the last eaten food. And in attack we set invaderDistance to the mazeDistance between our unit and the invading pacman.

More obstacles and solutions

This agent worked surprisingly well. It alone (coupled with the baseline offense bot) gave us a near perfect win rate against the baseline team. However, as I alluded to before it has some problems with defending against multiple agents. Furthermore occasionally it can get juked due to his simple navigation and although it generally catches up this can still result in an extra food or two missing. Our patrol, although effective, wasn't perfect and by passing over the mid line and just looking a bit, our enemies could theoretically check where he is on his patrol, pass back over the line, then attack when he changes coordinates.

What we really wanted to do was to merge the behaviors of the offense and defense bot into one class called superAgent. This would allow us to effectively swap behaviors based on various conditions. For instance, if the offensive bot dies deep in enemy territory and there are no immediate threats in friendly territory, it would be more efficient to promote the patrolling defensive bot to offense and send the dead offense bot to defense. Another good use for this type of agent would be to look at the score of the game and time left and perform various behaviors based on those conditions. For instance, if it is losing and there is not a lot of time left with no immediate enemies in the vicinity, it would promote both bots to offense and try to sack as many points as quick as possible. Likewise if it was winning by a solid amount, it could send both bots to be defensive and essentially blockade the entrance points. This could also allow us the devise one and gun kind of strategies. I.e. if it's the beginning of the game and you just killed an enemy, send a bot into enemy territory to get the first pellet it sees. Quickly send the other bot in and try to rampage as much food as possible while the ghosts are blue. But after getting so many points in this way both bots would become defense agents and the game could stall.

Offensive unit

The problem:

The main problem to solve on the offense that we decided on was survivability. Because with the time it takes to make it back after getting eaten you've given up a significant chunk of time that could have been spent maneuvering to get pellets. So instead of say maximizing pellet groupings we've gone with survivability. Of course the agent will still track down capsules and pellets but by placing survivability at the forefront we hope to also gain inevitability.

The model:

There are a few things that need to be tracked for the model. One is obviously the current position of our agent so that he knows where he is. The next is the location (or noisy location) of our opponents. The other two aspects are the locations of the food pellets and the power capsules. The maze doesn't need to be tracked necessarily cause it doesn't change but the positions of objects within do. Not to mention the fact that walls represent illegal actions so that they'll never be taken anyways.

Model representation:

Since we're building the AI off the pacman game we have a state space representation of the world where each position on the grid is a state containing various bits of information. For our purposes this also means the data containing the positions of relevant materials such as food or the opponents. The agent's position and food can be easily obtained. We can put those into variables or lists for further consideration and update them as the state of the world advances. The real problem lies in getting the state of the opponent. Unless a teammate is within 5 spaces we cannot get an exact location on where the opponents may be. But we are provided with a sort of noisy read on their location. So we have to work with that. So our agent has modeled the locations of the food/capsules, its own location and has an idea of where our enemy may be.

Strategy:

The game plan was to let the defense agent handle the defending work while the offense agent stays on the opponent's side as long as it can. To this end it needs to evaluate the worth of getting food vs getting away from the enemy. This is done via an evaluation function that calculates the worth of say going toward food vs fleeing. Of course the actual weight of each decision falls on the programmer as they are ultimately the ones who set the weights of each evaluation.

The agent on its own side will not fear enemy presence. But once on the opponent's side it has every reason to. It will eat and try to stay away from the opponents. The distance where it is absolutely too close would be the length of a dead end cause from that point on there is no escape. Thusly the evaluation for having only 3 legal moves remaining is quite grim. The only way to turn this around is if a capsule is within a certain distance from pac man given he can reach it at all. In that case the situation can be reversed and pac man can escape. We felt that attacking enemy ghosts would be a waste of time as the respawn location is close and the situation would quickly reset so it simply ignores ghosts for a bit as they run away. Finally, over time as enemy food becomes more scarce the pac man prioritizes food more and more so it can go places it wouldn't have before in an attempt to end the game. This is simply done using an inverse proportion based on food remaining.

Algorithm choice:

Sometimes a simple solution is often the best solution. Thinking about the AI used in video games that I've played I came to the conclusion that they are probably modeling and solving the game states in a way that is easy for it to determine in real time. Simple chess bots and some 2d games come to mind.

The plan was to use an expected maximum algorithm to determine the best direction to go into the enemy field and avoid defending ghosts by looking at possible moves ahead. This requires a basic idea of what the opponent is trying to do in order to derive the expectimax values. So once the pacman is inside enemy territory it would look for the distance to the enemies in the region and pick a direction that maximizes its maze distance from the enemy ghosts and continue to eat pellets. Its ability to look ahead provides at least some kind of survival guideline for our pacman as he stealths his way across on the enemy side. Once a ghost gets close enough though it becomes much easier to model expected enemy behaviours as if they're close enough to see they probably want to kill. That is also where it gets more interesting as now we can simulate the opponents moves now knowing his location and actually run expectimax properly. That's where we do our state look ahead and determine our escape moves.

Obstacles and setbacks:

The biggest set back was dealing with not knowing the exact location of the enemy. That made the creation of the evaluation much more difficult as we have to do a sort of noisy location comparison to get a semblance of movement but even then a noisy distance may lead to an illegal move being passed through when attempting to predict enemy movement which would end the game. So instead of always trying to predict where the enemy is we only do so when they are close and let the regular evaluation function work based on their noisy position and maximize the distance.

Also since the agent works with mostly fractions for its evaluations for thing such as food and enemy distance it's rather difficult to tweak it as small changes to numbers can cause huge behavioural changes. Especially difficult when trying to tweak things to work with capsules as a third set of values make it a lot more difficult to determine what to do if everything is proportioned.

Evaluation:

I think this agent is pretty good but has some glaring flaws that aren't accounted for . Such as 2 enemies closing in from different sides. Or having to go into a dead end to retrieve food but the dead end being deep enough that the enemy can block the entrance. The main problem for this is that modeling these kinds of situations. the dead end example you can't get an exact distance/position until you see the enemy so you can't properly evaluate the move that will allow you to escape. I also think that it's not exactly the most efficient at gobbling down pellets. This is probably cause of the emphasis on survival but often times i think it would be better to aim for clusters of pellets.

Other than that the agent does a good job maintaining its distance from the enemy as well as prioritizing pellets as time goes on.