Pacman Project

Johnson Lin
Yongchen@ucsb.edu
Perm: 8485336

Architecture

For this project, I modified the getAction function in the MultiPacmanAgent class which returns the best move for Pacman in the current game state. This is done by executing a minimax search at the current state and choosing the action that will lead to the state with the highest score (determined by an evaluation function). To accomplish this, I defined a minimax search function that will be called on the current game state. I also defined a maximumSearchAgent function (Pacman) as well as a minimumSearchAgent function (ghosts) which make recursive calls to one another depending on whose turn it is. Finally, my heuristic function (evaluation function) uses a combination of attributes relating to the game state in order to compute a score that reflects how favorable the game is for Pacman (the maximizing agent).

Solution

Evaluation Function

I began designing my evaluation function by thinking of all the attributes that could possibly affect how favorable a game state is, whether this attribute should have a direct or inverse relationship with the score and finally assigning weights to each attribute. I chose to incentivize eating capsules before foods since the more food remaining when capsules are eaten the better. Similarly, I incentivized having a closer distance to capsules over having a closer distance to foods. I also defined different danger levels for when any particular ghost is too closer to Pacman. Defining a minimum threshold for when to worry about ghosts works better than using the nearest ghost distance directly as it allows Pacman to be more greedy in safer situations rather than always being wary of ghosts. I also incorporated the scoreEvaluationFunction into my evaluation function as this incentivized Pacman to eat available foods/capsules as soon as possible. This seemed to be particularly important when there were only a few foods remaining and Pacman would occasionally enter a state where it repeats moves.

Search

Both the minimizing function and the maximizing functions begin by checking whether the current state is a winning or losing state and will return either a very large or a very low score correspondingly. Both functions also check whether the specified depth of the search has been reached, at which point the score of the current state will be returned. Although this check technically should only be required in the maximizing function, this check was needed in the minimizing function in order to avoid timing out. The reason for splitting the search process into three functions was that we needed to return the root move that leads to the highest score and not only the highest score so the minimax function will loop through all possible Pacman actions before checking the highest score that is obtainable by taking that action (by calling the minimizer). The function will then return the action that corresponds to the highest score.

Why Minimax?

My initial thoughts were to use Expectimax since the ghost's behavior is somewhat random and Expectimax would allow Pacman to be greedy in advantageous positions. Expectimax, however, seemed to underestimate the ghost's behavior which would result in frequent losses. Using a more cautious search algorithm like Minimax along with danger thresholds seemed to be the best combination as it allowed Pacman to be greedy while also accounting for the worst-case scenario.

Results

80% win rate on Gradescope with an average score of 3662

Challenges

The biggest challenge of this project was assigning correct weights to state attributes in order to generate the desirable Pacman behavior. I began by intuitively assigning weights and then repeatedly running the agent locally while tweaking weights and adding new attributes until I obtained the behavior I wanted. Another issue was stopping Pacman from entering repetitive states (often at the end of games). I solved this by heavily incentivizing eating foods if there were only a few foods left. Also, instead of subtracting unfavorable attributes from the score directly, I applied an inverse relationship (ie. score += constant/attribute) to them which also helped to avoid repetitions.

Weakness

One of the biggest weaknesses in my implementation is that it fails to account for 50% of the time when the ghosts move randomly. As a result of this, it may miss out on certain opportunities to be greedy and achieve a higher score. This could perhaps be fixed by introducing some element of randomness into the getAction function where 50% of the time it uses Minimax and 50% of the time it uses Expectimax. Another weakness of my model is that it fails to recognize cases where Pacman becomes surrounded by ghosts and is trapped between walls. The win rate may be improved if the evaluation function took this into account.