THU AI Assignment II: Berkeley Pacman Project 2

Multi-Agent Search

Rui Wang*
2015010445

1 Q1: Reflex Agent

1.1 Determine the Evaluation Function

The class *ReflexAgent* mainly applies an evaluation function to estimate the best strategy every step. Basically, we consider the component of a game state. We need to take into consideration several factors:

- The possibility of being eaten by ghosts, denoted by the distance to ghosts.
- The chance of eating a dot, denoted by the distance to the nearest dot.

One thing worth noting is that in a state, pacman will have already eaten the dot if it is at a dot. Therefore, when we want to evaluate a new state lead to by a certain action, we need to calculate the distance between NEW pacman position and the closest dot in the CURRENT state.

The evaluation function I chose is as follows. Let d_i be the distance between pacman and ghost number i, d_{food} be the shortest manhattan distance between dot and pacman, P be the penalty of taking a specific step while B be the bonus of taking it. We define:

$$P = \sum_i \frac{k_1}{(d_i - k_2)^{k_3}}$$

where k_j are constants(parameters) to be determined through test and practice. For reference, in my version of implementation, $k_1 = 3$, $k_2 = 0.5$, $k_3 = 2$.

$$B = \frac{m_1}{d_{food} + m_2}$$

Also in my code, $m_1 = 2$, $m_2 = 0.2$.

1.2 Test Result

The test result of these parameters proves promising, with an average score of 1236.5. This result shows the validity of such evaluation.

*wangrui15@mails.tsinghua.edu.cn

2 Q2: Minimax

2.1 Summary

Since a stack-simulated recursive search would prove too complicated for this algorithm, a recursive function that calls itself is defined in the implementation of this Minimax algorithm.

2.2 Special Discussion

Notice: Huge pitfall: a very important thing to notice is that the assigned searching depth for pacman might result in no choice for it, i.e. the function *getLegalActions* may end up returning no action at all. In this case, we need to break the exploration in this branch at once or we will end up returning the *maxScore* as -99999 or *minScore* as 99999, as they are initialized.

This bug in particular cost me more than 3 hours to find out. Actually the *autograder.py* will return a searching tree with its depth and assigned depth not matching. This is, however, legal case since depth is '**ARBITRARY**'.

2.3 Test Results

Run the following command, and we will find:

```
python pacman.py -p MinimaxAgent -a depth=3 -l smallClassic
```

In the case of trapped classic maze, pacman will bump into the closest ghost since it is calculating its grade several steps later. This is not always true especially in a case where ghost behavior is unpredictable(and often enough suboptimal). In the case of a small classic maze, pacman takes around 0.5 seconds to make a move(on a macOS system, with 8GB memory).

3 Q3: $\alpha - \beta$ Pruning

3.1 Summary

The most important part of $\alpha - \beta$ Pruning is the algorithm itself. It is not as trivial as it seems at first sight. In a typical search tree, in order to keep trace of nodes expanded earlier than the direct parent, we need to pass two parameters, a and b to a node. The specific method is described in detail in lecture note.

3.2 Test Result and Discussion

Running $\alpha - \beta$ Pruning on a classic small maze with the following command:

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

Pacman will be running a bit faster than with a traditional minimax agent. However, improvement is not satisfactory. This implies that $\alpha - \beta$ Pruning cannot bring substantial improvement to a searching algorithm.

4 Q4: Expectimax

ExpectimaxAgent simply calculates all possible moves of a ghost. There's not much to be altered in the code of *MinimaxAgent*. Test result shows that pacman can survive half the time in a *trappedClassic* maze. It also wins the **minimaxClassic** maze which *minimaxAgent* fails to win.

5 Q5: Better Evaluation Function

This evaluation function here is called by the *autograder* to make evaluation for different states. There is, however, some difference between a state-estimation function and an action-estimation function. For example, when estimating whether a state is good or not, we should consider the total number of food remaining instead of the distance to the closest food, because the state even later should not affect pacman's evaluation of the state for the time being, otherwise pacman would often keep static even if a food is close to it-since eating the food will only result in a worse state estimation.

My implementation of state estimation consists of the following factors:

- If the state is win/lose, will go to/avoid it.
- The (unscared) ghosts should be kept as far away as possible.
- The food remaining should be as little as possible.

Denote p as penalty and b as bonus, and then define

$$p = -\sum_i \frac{5}{(d_i - 0.5)^2}$$

where d_i id the distance to the i^{th} ghost.

$$b = \frac{1}{(d_i - 0.5)^2} - 5n - 100m$$

where n is the number of food remaining and m is the number of unscared ghosts. Finally return the following result as the evaluation of this state:

$$p + b + currentGameState.getScore()$$

The last item $currentGameState.getScore()$ is extremely tricky. Adding it will result in an eligible agent, while lacking it will result in strange behavior that is against common sense. However the root of this problem is still unknown since there is no clue of how this function is called. Thus students should be reminded of this detail which otherwise would be impossible to find out.