Midterm Report

Artificial Intelligence, COSE361-02

2014210040

이수호

1. Report screenshot from capture.py

	your_best(red)
	<Average Winning Rate>
your_base1	0.7
your_base2	1
your_base3	0.8
baseline	0.2
Num_Win	4
Avg_Winning_Rate	0.675
	<Average Scores>
your_base1	1.5
your_base2	2.5
your_base3	2
baesline	0.2
Avg_Score	1.55

2. Explanation of 4 different algorithms

a. your_baseline1.py (Enhanced Reflex Agent)

I started making Pacman agents based on given *baseline.py*. Baseline class provides reflex agent with two different characteristics, either offensive or defensive. It provides some basic features and their respective weights to consider and its final score is $S = sum(feature\_score * weight)$. Offensive agent cares about distances between foods and the agent and lower distance yields better score. Defensive agent, while, cares about distances between the agent and invaders which are enemies finding foods on our side. After examining the algorithm, I came up with idea that any agents can be both offensive and defensive, and defensive behavior is more useful if the agent is a ghost while offensive is so if it is a pacman.

I separated weight of a Pacman (which is `getWeightsForPacman()` in a code) from weight of a ghost (`getWeightsForGhost()`) and apply either weight based on agent's

state. Pacman's weight mainly considers distance between food and the agent (`distanceToFood`) while ghost's weight considers distance between enemy Pacman on our side-invader and the agent (`invaderDistance`).

I modified baseline's score calculation method to accept arbitrary function for greater flexibility. Initial value of the method is a reciprocal function, which yields bigger value when input-distance is smaller. Then I added some bias for these functions for extra fine-tuning, which will be mainly used in *your_best.py*.

However, only from these behaviors, it does not yield any score, because carried food of Pacman must be brought to our side. Concerning this game rule, I added new state for agent (a returning Pacman – `getWeightsForReturningPacman()`) and applied it if the agent carries more food than `MIN_NUM_TO_RETURN_HOME`.

After implementing these, I can observe general behaviors in which the agent obtains food when it is a Pacman or chases invader if it is a ghost. Moreover, the agent starts to gain a score and wins for some cases.

b. your_baseline2.py (Minimax Agent with Alpha-beta Pruning)

I made another baseline which uses minimax algorithm for the agent. For game-tree traversal efficiency, I also included alpha-beta pruning in this baseline. Even though the prescence of included pruning, the minimax algorithm clearly takes more time (one step is around 10ms at 2-depth search) to run than reflex agent's, because they basically do a tree traversal.

One thing that I could notice is that minimax agent even does not move properly. The reason I can assume is that game score (`state.getScore()`) does not change while evaluating a shallow-depth game tree, so they does not return optimal action for the state. However, it is impossible to increase depth because the evaluation time increase exponentially.

c. your_baseline3.py (Minimax Agent with Custom Scoring)

To overcome scenario shown in Baseline 2, custom score (which goes into a node value in a game tree) should be calculated. I put a score calculation method from Baseline 1 and replaced `state.getScore()` in Baseline 2. The agent generates far

better moves. However, in this case, the agent generates inefficient action and usually loses against Baseline 1 as a result. Movement of the opponent does not change the score drastically so the decision of the minimax algorithm does not seem to represent optimal moves. Moreover, making a new scoring method for minimax algorithm is harder than I expected. As a result, I thought that minimax algorithm is not optimal in this case – final score of this game is not changing drastically and we cannot utilize deeper traversal.

d. your_best.py (Enhanced Reflex Agent with Fine-tuned Parameters)

Even though Baseline 1 sometimes wins, winning rate of the algorithm is not sufficient. The agent does unwanted behavior to gain scores. For example, the agent is 'trapped' and goes north and south repeatedly in case as shown as [Figure 1]. After some diagnostics, score calculation of the agent seems to be more tuned. Due to the reason that the score of returning Pacman is affected by two or more variables, distance between enemy ghost and distance between home in this case, the Pacman does suboptimal action which is affected by enemy's position. To overcome scenario like this, weight calculation method has to be tuned. I thought that there are two ways to tune parameters. One way is to do it all manually, and the other way is to iterate between parameter assignment, simulation, and evaluation (like training a neural network). Later way seems to be more efficient regarding time consumption, so I started to parameterize all of the variables in the weight calculation method.

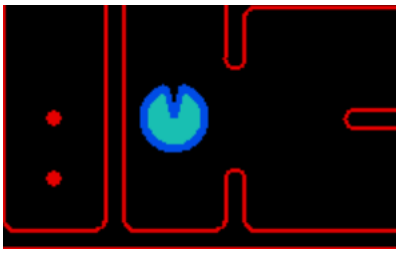

FIGURE 1

The simulation code which I used is included in your_best.py and it starts when the file itself is executed.

3. Further discussions

What happens if the reflex agent is driven with 'wrong' weight – if Pacman uses the weight optimized for ghost's and vice versa?

```
Average Score: -3.8
Scores:        -3, -3, 3, -5, -4, -7, -7, -5, -3, -4
Red Win Rate:  1/10 (0.10)
Blue Win Rate: 9/10 (0.90)
Record:        Blue, Blue, Red, Blue, Blue, Blue, Blue, Blue, Blue, Blue
===
nextX: 1, WR: -0.9, Score: -3.8
We are on a wrong track. Rolling back..
[[1, 1, 4, 1, 1, -6], [1, 1, 4, 1, 1, -4], [1, 1, 6, 1, 1, 0.5]]
```

FIGURE 2 OUTCOME WHEN NON-OPTIMAL WEIGHT IS APPLIED

I've accidentally made a mistake during making your_best.py and doing some simulations. What I did was to apply ghost's desired weight to returning Pacman, and Pacman's to the ghost. The agent which contains a bug (Red team) is losing against Baseline 1 (Blue team) as shown in [Figure 2]. This brings a result that proper weight calculation should be applied for agent's *roles* if we are using a reflex agent algorithm, which might seem to be trivial.