Xinyi Ma [xim002@ucsd.edu](mailto:xim002@ucsd.edu)

Yijun Zhang [yiz160@ucsd.edu](mailto:yiz160@ucsd.edu)

Yuanchi Ha [yuha@ucsd.edu](mailto:yuha@ucsd.edu)

**HW4 Writeup**

**Algorithm:**

**P1: evaluationFunction**

In our function, if there is still capsule left, we encourage Pacman to eat the ones near them by giving them 5-capsule\_dist when the distance is smaller than 5. This is to help them realize that it’s not worth chasing the capsule too far away from it. Moreover, to set the evaluation within a reasonable domain, we set the enemy\_dist bigger than 10 as 11.

After getting the distance from the nearest food, the distance from the nearest enemy(ghosts), we subtract the distance from food from the distance from enemy. Then add award value to evaluate the position.

**P2: MinimaxAgent**

In the minimax agent class, we defined function min\_value and max\_value separately. The recursion call is realized by a function called get\_value.

The get\_value function consider 2 cases. If the current index of the agent is 0, which means the current agent is Pacman, then we call the max\_value function. Otherwise, we call the min\_value function.

In min\_value function, we first define the value as infinite to update it with smaller value later. Then we consider two distinct conditions. If the game is won or lost already, we return the evaluationFunction(gameState) directly. Otherwise, we consider another two conditions. If the index of current agent is the same as the number of ghosts, which mean that we already evaluated all ghosts, then we call max\_value function with pacman’s index inside the loop; otherwise, we recursively call min\_value with next ghost’s index.

In max\_value function, the logic is smiliar, except that we define value as negative infinite to start with, so that we can update it with bigger value later. Moreover, after considering whether the game is win or lost, we don’t need to consider another two conditions. Instead, we call min\_value with the first ghost’s index to update the value inside the loop.

Outside these three functions, we have the main function to start the game. First, we get all the legal actions to loop with. We also define the result action as direction.stop and score as negative infinity to be updated later. Then we start the game by looping through all legal actions and update the scores each time by calling min\_value with the index of the first ghost inside inside a max. If the score is better than the previous one, then we update the result action with the new action too.

**P3: AlphaBetaAgent**

In the AlphaBeta agent class, like what we did in the minimax agent class, we defined function min\_value and max\_value separately and function get\_value to do the recursion. Moreover, we update alpha and beta and return the value of based on its relation with alpha and beta inside the min\_value and max\_value function after each recursion call.

The get\_value function consider 2 cases. If the current index of the agent is 0, which means the current agent is Pacman, then we call the max\_value function. Otherwise, we call the min\_value function.

Similar to min\_value function in minimax agent class, we first define the value as infinite to update it with smaller value later. Then we consider two distinct conditions. If the game is won or lost already, we return the evaluationFunction(gameState) directly. Otherwise, we consider another two conditions. If the index of current agent is the same as the number of ghosts, which mean that we already evaluated all ghosts, then we call max\_value function with pacman’s index inside the loop, then we check if the value is already smaller than the best of max(alpha), if so, we return the value; otherwise, we update beta. If the index of current agent is not the same as the number of ghosts, we recursively call min\_value with next ghost’s index, then check value and/or update beta.

The max\_value function is exactly the same as the one in minimax function.

Outside these three functions, we have the main function to start the game. First, we get all the legal actions to loop with. We also define the result action as direction.stop, score as negative infinity, alpha as negative infinity, and beta as infinity to be updated later. Then we start the game by looping through all legal actions and update the scores each time by calling min\_value with the index of the first ghost inside inside a max. If the score is better than the previous one, then we update the result action with the new action too. After that, if score is already bigger than min’s best option, we return the result already, otherwise, we update alpha.

**P4: ExpectimaxAgent**

Similar to the minimax agent class, we we defined function min\_value and max\_value separately and function get\_value to do the recursion. Besides that, we calculate the expected value inside the min\_value function.

The get\_value function consider 2 cases. If the current index of the agent is 0, which means the current agent is Pacman, then we call the max\_value function. Otherwise, we call the min\_value function.

In the min\_value function, we first define the value as 0 to update it with expected value later. We also calculate the probability based on the length of legal actions. If it is zero, the the probability is 0, otherwise, it is the reciprocal of the length of the legal actions. Then we consider two distinct conditions. If the game is won or lost already, we return the evaluationFunction(gameState) directly. Otherwise, we consider another two conditions. If the index of current agent is the same as the number of ghosts, which mean that we already evaluated all ghosts, then we call max\_value function with pacman’s index inside the loop, then update value by adding the product of the probability and the value to it. If the index of current agent is not the same as the number of ghosts, we recursively call min\_value with next ghost’s index, then update value in a similar way.

The max\_value function is exactly the same as the one in minimax function.

Just like the minimax function, outside these three functions, we have the main function to start the game. First, we get all the legal actions to loop with. We also define the result action as direction.stop and score as negative infinity to be updated later. Then we start the game by looping through all legal actions and update the scores each time by calling min\_value with the index of the first ghost inside inside a max. If the score is better than the previous one, then we update the result action with the new action as well.

**P5: betterEvaluationFunction**

Similar as the evaluation from p1, in this function, if there is still capsule left, we encourage Pacman to eat the ones near them by giving them 5-capsule\_dist when the distance is smaller than 5. This is to help them realize that it’s not worth chasing the capsule too far away from it. Moreover, to set the evaluation within a reasonable domain, we set the enemy\_dist bigger than 10 as 11.

After getting the distance from the nearest food, the distance from the nearest enemy(ghosts), we subtract food distance and 50/enemy distance from the current game state’s score, and we add 5 times reward and 100/(food\_count+1) and 200/(capsule\_count+1) as a linear combination of the value

**Performance on test data:**

**P1: evaluationFunction**

Average Score: 1215.6

Scores: 1377.0, 1060.0, 1193.0, 1115.0, 1232.0, 1186.0, 1206.0, 1380.0, 1224.0, 1183.0

Win Rate: 10/10 (1.00)

Time: 4mins,42secs

**P2: MinimaxAgent**

Average Score: 84.0

Scores: 84.0

Win Rate: 0/1 (0.00)

Time: 55secs

**P3: AlphaBetaAgent**

Average Score: 84.0

Scores: 84.0

Win Rate: 0/1 (0.00)

Time: 57secs

**P4: ExpectimaxAgent**

Average Score: 84.0

Scores: 84.0

Win Rate: 0/1 (0.00)

Time: 56secs

**P5: betterEvaluationFunction**

Average Score: 1309.3

Scores: 1365.0, 1374.0, 1319.0, 1333.0, 1379.0, 1101.0, 1364.0, 1292.0, 1383.0, 1183.0

Win Rate: 10/10 (1.00)

Time: 2mins,24secs

**Performance on test data and Analysis of result:**

**P2: MinimaxAgent bigO(b^m)**

**On small board the pacman is not bad at winning:**

python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=1 -n 10

Average Score: 211.8

Scores: -496.0, 516.0, 516.0, 516.0, 516.0, -493.0, 516.0, -499.0, 516.0, 510.0

Win Rate: 7/10 (0.70)

python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=2 -n 10

Average Score: 8.4

Scores: -492.0, -500.0, -497.0, 513.0, 510.0, -492.0, 516.0, 516.0, -500.0, 510.0

Win Rate: 5/10 (0.50)

python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=3 -n 10

Average Score: -92.5

Scores: -497.0, -495.0, 514.0, 509.0, 513.0, -492.0, -496.0, 510.0, -495.0, -496.0

Win Rate: 4/10 (0.40)

python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4 -n 10

Average Score: 213.0

Scores: -492.0, 516.0, 516.0, 516.0, -495.0, -495.0, 516.0, 516.0, 516.0, 516.0

Win Rate: 7/10 (0.70)

**On larger board, pacman takes forever to run, which shows that it’s good at living, but not winning**

**P3: AlphaBetaAgent Ideally:bigO(b^m/2)**

**Theoratically, it should be faster than p2, but the result will not change much, since pruning does not affect final result: we didn’t change the logic of the algorithm, except for setting limit for max and min. And the test data confirm that, again, it’s Not bad at winning in small games, but not good in larger game:**

python pacman.py -p AlphaBetaAgent -l smallClassic -a depth=2 -n 5 -q

Average Score: -206.2

Scores: -169.0, -171.0, -392.0, -98.0, -201.0

Win Rate: 0/5 (0.00)

Record: Loss, Loss, Loss, Loss, Loss

Time: 13.02 secs

python pacman.py -p AlphaBetaAgent -l smallClassic -a depth=3 -n 5 -q

Average Score: 737.6

Scores: 1115.0, -52.0, 271.0, 1283.0, 1071.0

Win Rate: 3/5 (0.60)

Record: Win, Loss, Loss, Win, Win

Time: 129.04 secs

python pacman.py -p MinimaxAgent -l smallClassic -a depth=2 -n 5 -q

Pacman emerges victorious! Score: 1087

Average Score: -1.0

Scores: -243.0, -122.0, -761.0, 34.0, 1087.0

Win Rate: 1/5 (0.20)

Record: Loss, Loss, Loss, Loss, Win

Time: 34.33

python pacman.py -p MinimaxAgent -l smallClassic -a depth=3 -n 5 -q

Average Score: 585.2

Scores: -223.0, -11.0, 1377.0, 998.0, 785.0

Win Rate: 3/5 (0.60)

Record: Loss, Loss, Win, Win, Win

Time: 145.24 secs

|  |  |  |
| --- | --- | --- |
| Agent | Depth | Time(secs) |
| MiniMax | 2 | 34.33 |
| 3 | 145.24 |
| AlphaBeta Pruning | 2 | 13.02 |
| 3 | 129.24 |

* The order is not the best, so our big O is not O(b^m/2)

**P4: ExpectimaxAgent**

**In this case, we should expect our Pacman to win more, since we calculate chance node based on their possibilities instead of min or max among all options.**

python pacman.py -p ExpectimaxAgent -l smallClassic -a depth=2 -n 10 -q

Average Score: 462.3

Scores: 918.0, 997.0, 1067.0, 1606.0, -179.0, 751.0, 320.0, -233.0, -360.0, -264.0

Win Rate: 5/10 (0.50)

Record: Win, Win, Win, Win, Loss, Win, Loss, Loss, Loss, Loss

Time: 50.34

python pacman.py -p MinimaxAgent -l smallClassic -a depth=2 -n 10 -q

Average Score: -23.2

Scores: -1.0, -326.0, -403.0, -225.0, -203.0, 968.0, -381.0, -91.0, 618.0, -188.0

Win Rate: 2/10 (0.20)

Record: Loss, Loss, Loss, Loss, Loss, Win, Loss, Loss, Win, Loss

Time: 38.07

python pacman.py -p ExpectimaxAgent -l smallClassic -a depth=1 -n 10 -q

Average Score: -121.5

Scores: -66.0, -207.0, 540.0, -491.0, 340.0, -399.0, 96.0, -378.0, -253.0, -397.0

Win Rate: 2/10 (0.20)

Record: Loss, Loss, Win, Loss, Win, Loss, Loss, Loss, Loss, Loss

real 11.66

python pacman.py -p MinimaxAgent -l smallClassic -a depth=1 -n 10 -q

Average Score: -319.6

Scores: -356.0, -409.0, -150.0, -375.0, -514.0, -227.0, -195.0, -542.0, -179.0, -249.0

Win Rate: 0/10 (0.00)

Record: Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss

real 6.76

|  |  |  |
| --- | --- | --- |
| Agent | Depth | Win Rate |
| Minimax | 1 | 0/10 |
| 2 | 2/10 |
| Expectimax | 1 | 2/10 |
| 2 | 5/10 |

**P1 & P5 Evaluation & their heuristic & reaction to ghosts:**

The heuristic in P1 compares its score to the previous state. Therefore, it is able to know whether a food or capsule has been eaten and give award for eating them. Since the only thing that matters is the relative score of each action, there are less to care about (for example, we can safely ignore the enemy when it is far away). Our heuristic did not distinguish between a normal ghost and a scared ghost to react differently for each other. The heuristic of a ghost is simply the manhattan distance between us plus the ghost’s scared time.

This is not the case in P5. It is much harder to know whether the pacman has eaten anything. And we have to seek for better evaluation function to compute the food and capsules left on the board. Therefore, we use reciprocal function to “regularize” the award for eating food and capsules. Our pacman also becomes more “aggressive” and tends to eat a scared ghost because of the way we construct the linear function.