ESIEE Paris

Artificial Intelligence and Cybersecurity

Artificial Intelligence

Lab 3 Report

Bruno Luiz Dias Alves de Castro
Victor Gabriel Mendes Sündermann

April 21, 2023

Contents

1 Introduction

In this report, we will present the results of the third project of the Artificial Intelligence course. The project consists of implementing

2 Tests

To run the program, we use the following command:

```
python3 pacman_AIC.py -p ReflexAgent [-l testClassic]
```

Running the command without the flag *-l testClassic*, renders a version of the maze without any walls.

The first two tests ran as expected, the Pacman tries to eat all the food in the field while avoiding all the ghosts. The mai problem with both tests is that, while avoiding the ghosts, it'll stay in place most of the time, and not go after the food.

This can be improved by implementing an evaluation function, which will be developed in the next section.

3 Reflex Agent

In this part, we were purposed to improve the **ReflexAgent** function in the **multiAgents.py** file in the project by proposing an evaluation function.

3.1 ReflexAgent improvement

In order to improve the **ReflexAgent**, we need a good evalution function to help the AI decide what to do next. A great evaluation function should be able to tell the agent what is the best action to take in a given state, rewarding good actions and punishing bad ones.

For our problem, we decided that a good action should be eating a food pellet, and a bad action should be getting eaten by a ghost. So, our evaluation function calculates the distance of Pacman to the food pallets and the ghosts. It then returns a score like the following:

```
    return score + closestFoodScore - closestGhostScore
```

We also decided on using the Manhattan distance to calculate the distance between Pacman, the ghosts and the food pellets, because it is a less computationally expensive sollution to this problem, and don't really affect the final score.

Finally, the result we got is the following:

```
def evaluationFunction(self, currentGameState, action):

    """ VARIABLES """

    foodList = newFood.asList()

    # manhattan distance
    foodDistances = \
        [manhattanDistance(newPos, food) for food in foodList]
    ghostDistances = \
        [manhattanDistance(newPos, ghost) for ghost in newGhostsPos]

    # closest food and ghost
    closestFood = min(foodDistances) if len(foodDistances) > 0 else 0
    closestGhost = min(ghostDistances) if len(ghostDistances) > 0 else 0

    # score calculation for food and ghost
    closestFoodScore = 0 if closestFood == 0 else 1.0/closestFood
    closestGhostScore = 0 if closestGhost == 0 else 1.0/closestGhost

    score = closestFoodScore - closestGhostScore

    return successorGameState.getScore() + score
```

3.2 Tests

In order to test our new evaluation function, we executed the following tests:

3.2.1 testClassic

Command:

```
    python3 pacman_AIC.py -p ReflexAgent -l testClassic
```

Output:

```
Pacman emerges victorious! Score: 562
Average Score: 562.0
Scores:        562.0
Win Rate:      1/1 (1.00)
Record:        Win
```

3.2.2 mediumClassic

- One ghost test run

Command:

```
python3 pacman_AIC.py -p ReflexAgent -k 1
```

Output:

```
Pacman emerges victorious! Score: 1485
Average Score: 1485.0
Scores:        1485.0
Win Rate:      1/1 (1.00)
Record:        Win
```

- Two ghosts test run

Command:

```
python3 pacman_AIC.py -p ReflexAgent -k 2
```

Output:

```
Pacman emerges victorious! Score: 1407
Average Score: 1407.0
Scores:        1407.0
Win Rate:      1/1 (1.00)
Record:        Win
```

3.2.3 No layout 20 runs average

During this part of the testing, we ran the **ReflexAgent** 20 times on each of the 6 conbinations of the settings listed below, and calculated the average score and the win rate. The possible settings are as listed (indentified with the letters a-e):

- Possible settings:
  (a) One ghost
  (b) Two ghosts
  (c) Random ghosts
  (d) Random ghosts with fixed seed
  (e) Non random ghost

The results obteined are shown in Table 1.

Configuration	(a, c)	(a, d)	(a, e)	(b, c)	(b, d)	(b, e)
Average Score	671.65	785.2	449.9	464.7	228.7	-14.25
Win Rate	14/20 (0.70)	18/20 (0.90)	10/20 (0.50)	10/20 (0.50)	6/20 (0.30)	1/20 (0.05)

Table 1: ReflexAgent test results

3.2.4 Results

The **ReflexAgent** was sucessful in clearing most of the purposed tests. Especially the the ran in sections 3.2.1 and 3.2.2 (**testClassic** and **mediumClassic**), were no problem for the agent.

Things start to fall short when it comes to test in section 3.2.3. The agent obtained decent success in the test with a single ghost, but performenced declined on tests with two ghost. The agent only cleared 6 out of 20 tests with random ghosts, and was onle able to clear a single test with non random ghosts.

4 Minimax algorithm

The Minimax algorithm consists in the following idea: two "players" (or agents) are competing against each other in a game. As both players equally want to win this game, if we assume the point of view of one of them, we can say that player wish to maximize its score, and the opponent is trying to maximize the most it can.

It's possible to simulate such a scenario, by making both players take turns, and attributing a "score" to each play. Every time a "maximizer player" plays, it chooses the maximum score possible, and every time a "minimizer" player player, it chooses the minimum score available.

By simulate all possible plays in a game from a given game state, we can write an algorithm that alternate between maximize and minimize plays, and chooses the optimal path in a tree of possible outcomes for the game. This takes into account that all players always take the best play possible (what may not happen in all scenarios, we explore this further in Section 6.1).

An implementation of such approch is described in the following section.

4.1 Minimax implementation

To make this implementation possible, we started by implementing two functions: one for the maximizer (Pacman), and one for the minimizer (ghosts). We start by calling the maximizer, that looks for all possible plays of the minimizer agent. They recursivly call each other, until a "depth" parameters reachs zero, or the game terminates. This depth parameter is necessary, because of the complexity of this algorithm. As every agent calls the next on recursivly, this results in a tree of calls that expands indefinetly until the end of the game is reach. If we don't interupt the search, this algorithm would take ages to compute a play.

The implementation of both functions can be seen in Tables 2 and 3.

```
def MAX_VALUE(self, gameState, d):
    if d == 0 or gameState.isWin() or gameState.isLose():
        # base case: return evaluation function
        return self.evaluationFunction(gameState), Directions.STOP

    bestScore, bestAction = -math.inf, Directions.STOP

    for action in gameState.getLegalActions(0):
        successors = gameState.generateSuccessor(0, action)

        # call minimaxer agent (ghosts)
        value, _ = self.MIN_VALUE(successors, d, 1)

        # update best score and action
        if value > bestScore:
            bestScore, bestAction = value, action

    return bestScore, bestAction
```

Table 2: Maximizer function

In the case of the minimizer function, when calculating the possible plays for it's children, it first call the remaining ghosts. If all ghosts already played that turn, it then calls the maximizer (Pacman), and decreases the depth parameter.

4.2 Tests

In order to test the Minimax algorithm implemented, we ran it 20 times, using different depths, and the results are shown in the table below:

Analysing the results, we reached the following conclusions: First, is that the algorithm performs better the more depth we add to the search. This is no surprise: the more we search the tree, the closer we approch

```
def MIN_VALUE(self, gameState, d, indexAgent):
    if d == 0 or gameState.isWin() or gameState.isLose():
        # base case: return evaluation function
        return self.evaluationFunction(gameState), Directions.STOP

    bestScore, bestAction = math.inf, Directions.STOP

    for action in gameState.getLegalActions(indexAgent):
        successors = gameState.generateSuccessor(indexAgent, action)
        if indexAgent == gameState.getNumAgents() - 1:
            # call pacman
            value, _ = self.MAX_VALUE(successors, d - 1)
        else:
            # call next ghost
            value, _ = self.MIN_VALUE(successors, d, indexAgent + 1)

        # update best score and action
        if value < bestScore:
            bestScore, bestAction = value, action

    return bestScore, bestAction
```

Table 3: Minimizer function

Depth	2	3	4
Average	-173	546	701
Best	1249	1531	1732
Win Rate	1/20 (0.05)	11/20 (0.55)	11/20 (0.55)

Table 4: Minimax test results

the optimal sollution. Lastly, the more we increment the depth, the more time it takes to conclude the search. Due to the exponential nature of this algorithm, it's not possible to go much further than what we already done. With a depth of 4 it already took close to an hour to terminate. Luckly, there is an improvement that can be done to acelerate this process a bit, and we explore it in the next section.

5 AlphaBeta algorithm

The Alpha Beta implementation for the Minimax algorithm consists of two main changes: the addition of two parameters, alpha and beta, and the pruning of the search tree.

The parameters alpha and beta helps the algorithm identify if it is possible to continue searching the tree, or if a optimal value was already reached. In this case, we proceed with the pruning of the search tree, and return the value of the node.

This pruning is done in both the minimizer and maximizer functions, and help us reduce the number of nodes that are evaluated, and therefore, the time it takes to find the optimal move. This translates into a considerable performance improvement, as we will see in the next section.

5.1 AlphaBeta implementation

In order to implement the AlphaBeta algorithm, we used the same functions as the Minimax algorithm, but added two parameters to the functions, alpha and beta. The alpha and beta parameters are used to prune the search tree, and are initialized as **-math.inf** and **math.inf** respectively. The alpha and beta parameters are updated in the minimizer function, and the maximizer function, as shown in the code below:

```
""" CODE """

if value > beta:
  return value, action

alpha = max(alpha, value)

""" CODE """
```

Table 5: Maximizer function with beta pruning

```
""" CODE """

if value < alpha:
  return value, action

""" CODE """
```

Table 6: Minimizer function with alpha pruning

5.2 Tests

In order to test the performance of the Alpha Beta algorithm, we used the same tests as the Minimax algorithm, and compared the results. The results are shown in the table below:

Depth	2	3	4
Average	-121.5	413.6	635.85
Best	1001	1604	1701
Win Rate	1/20 (0.05)	9/20 (0.45)	11/20 (0.55)

Table 7: AlphaBeta test results

5.2.1 Performance comparison with Minimax

After running the tests, we compared the performance of the Minimax (Table 4) and Alpha Beta (Table 7) algorithms.

As expected, the performance between the two algorithms is pratically the same. That's expected, as the main functionallity of both algorithm is the same, the onlt different between the two being the pruning in the Alpha Beta version.

Where the difference is definitely noticeable is in the runtime of both algorithms. To test this, we ran the same tests as before, but with a depth of 3, and compared the results. The results are shown in Table 8. As we can see, the Alpha Beta algorithm is 15.6% faster than the Minimax algorithm.

Algorithm	Minimax	AlphaBeta
Time	167.78s	145.10s
Speedup	-	15.6%

Table 8: Minimax VS AlphaBeta (Depth=3)

The speedup obtain in this test is not that significant, but it is still a considerable improvement. This shy performance improvement is most likely due to the fact that the main botleneck of the algorithm is not the search. In our tests, we noticed that, due to the depth limit, when the Pacman eventually gets isolated in a area of the maze with no food, the algorithm doesn't really know what to do, and this leads to games that take a long time to finish.

To verify this, we ran the same test with the DirectinalGhost option. This forces the Pacman to move more frequently (as it is now being more effectively chased by the ghosts), but due to the exploration depth limitation, it loses games more oftenly. The results are shown in Table 9.

Now the speedup is much more significant, as the Alpha Beta algorithm is 83.5% faster than the Minimax algorithm. There is still more room to improvement though, but the performance gain is already considerable.

Algorithm	Minimax	AlphaBeta
Time	38.58s	21.03s
Speedup	-	83.5%

Table 9: Minimax VS AlphaBeta (Depth=3, DirectionalGhost)

6 Expectimax algorithm

The Expectimax algorithm is a varation of the Minimax algorithm. In this version, the minimizer function is replaced by an expectation function, which calculates the expected value of the next move. This is done by calculating the average of the values of all the possible moves.

This is extremelly usefull in our Pacman case, especially in cases where we execute the simulations with a random ghost movement. In these scenarios, the minimizer is as likely to take the worst movement possible as it is to do the best. The clasical algorith is great when the movements are always the best ones, but starts to fall apart when the opponent makes unexpected decisions. Knowing this, we alter the minimizer function to calculate an expected value instead of the absolute best one (the minimum in this case).

We describe the implementation in the next section.

6.1 Expectimax implementation

In order to implement the Expectimax algorith, the only change necessary is done to the Minimizer function. We used the same one implementd before, and added the following code to it:

```
def MIN_VALUE(self, gameState, d, indexAgent, alpha, beta):
  """ CODE """

  value = value / len(gameState.getLegalActions(indexAgent))
  bestScore += value

  """ CODE """
```

Table 10: Expectimax implementation

6.2 Expectimax tests

We ran the same tests as before, but with the Expectimax algorithm. The results are shown in Table 11.

Depth	2	3	4
Average	97.85	551.3	873.15
Best	1031	1397	1515
Win Rate	4/20 (0.20)	10/20 (0.50)	15/20 (0.75)

Table 11: Expectimax test results

7 Bonus: Evaluation function improvement

After implementing and running countless test to all algorithm proposed, we identified a serious problem with the evaluation function used in the project: It was the main piece holding our implementations back, and the perormance of our algorithms were suffering a lot because of it.

The default evaluation function implemented in the project is the ***scoreEvaluationFunction***. The evaluation function only takes into account the current score of the games. It gets the work done in some of the cases, but has two serious draw-backs: It doesn't take into account the number of food left in the maze, and it doesn't take into account the distance to the closest food. As a result, when the Pacman gets "isolated" in a part of the map without any food, it hasn't anything to maximize, and needs a ghost to approch in order to proceed the game.

To solve this problem, we sugest a new evaluation function, called ***betterEvaluationFunction***. It's implementation and functionallity is explained in the following section.

7.1 The new evaluation function: betterEvaluationFunction

The proposed evaluation function takes into account this two extra parameters (along with the score): *the number of food pallets left* and *the distance to the closest food*. The number of food pallets left encourages the Pacman to eat more pallets when stuck, and the distance to the closest food will prevent the Pacman of isolating itself in the maze.

the implementation is described in the following section.

7.1.1 betterEvaluationFunction implementation

To implement the new evaluation function, we created a new function in the code, as follows:

```
def betterEvaluationFunction(currentGameState):
  """
  Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
  evaluation function (question 8).

  DESCRIPTION: <write something here so we know what you did>
  """

  def closestFoodDistance(gameState):
    pacmanPosition = gameState.getPacmanPosition()
    foodList = gameState.getFood().asList()

    if len(foodList) == 0:
        return 0

    return min([manhattanDistance(pacmanPosition, food) for food in foodList])

  closestFood = closestFoodDistance(currentGameState)
  return currentGameState.getScore() - (10 * closestFood) \
    - (100 * currentGameState.getNumFood())
```

Table 12: betterEvaluationFunction implementation

For calculation the distance to the closest food, we used the the Manhattan Distance (implemented in the function itself), and the number of food is retrieved from the game state. They are multiplied by 10 and 100, respectively, in order to have a bigger impact in the final score, and subtracted from the current game score.

The results of our implementation is showed in the next section.

7.1.2 betterEvaluationFunction tests

To test the new betterEvaluationFunction, we repeated the same tests executed before, now using the this new evaluation function instead of the old scoreEvaluationFunction. We chose the Expectimax agent to run the tests. Its description and implementation can be found in Section 6.1.

The result of the tests can be found in Table 13.

Depth	2	3	4
Average	1228.6	1340.15	1131.25
Best	1724	1757	1743
Win Rate	19/20 (0.95)	20/20 (1.00)	19/20 (0.95)

Table 13: Expectimax test results with betterEvaluationFunction

Comparing both results of Tables 11 and 13, it's possible to observe a huge gain in performance. By only altering the evaluation function, we were able to obtain way batter results, even reaching a Win Rate of 100% in some test cases, and performing well in the cases with lower depths.

8 Conclusion