Artificial Intelligence

Lab 3

Groupe: William TRAVERS The Tai VUONG

Sommaire

I-Reflex Agent

In this exercise, the Reflex Agent goal is to make the Pac Man eat the dots and avoid the ghost if he is close to him.The primary method, evaluationFunction, considers both the proximity of food and the distance from ghosts to keep Pacman alive while maximizing score.

This correspond to the position of the Pac Man:

```
newPos = successorGameState.getPacmanPosition()
```

This corresponds to the the food:

```
newFood = successorGameState.getFood()
```

Thanks to that we can now define the minimal distance between the food and Pac Man thus giving the action to Pac Man to eat the closest food. In order to do it, the function asList() will convert the newFood into a list. `newFoodList = newFood.asList()`

With this, we can dig into the foods list when we compute the loop for:

```
for food in newFoodList:
        distMin = min(distMin, manhattanDistance(newPos, food))
        print("Pos de Pac Man", newPos)
```

We now want to code the movement of PacMan according to the ghost position. If he is close Pac Man needs to run away. We compute the Manhattan distance between them and whenever it is smaller than 2, the score will be low and equal to “-inf” when actions lead to this case.

```
for ghost in successorGameState.getGhostPositions():
    if (manhattanDistance(newPos, ghost) < 2):
        return -float('inf')
```

Therefore the other action will provide a bigger score allowing Pac Mac to move further from the ghost and make it avoid the ghost.
Finally, we want to prioritize eating food so the move variable will have more weight than the Manhattan distance and a greater score will encourage the Pac Man to move to the dots.

```
return successorGameState.getScore() + 1.0/distMin
```

Testing the code on 10 runs:
(a,c): One ghost and random ghosts

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ReflexAgent -l smallClassic -k 1 -n 10 -q
Pacman emerges victorious! Score: 976
Pacman emerges victorious! Score: 965
Pacman emerges victorious! Score: 959
Pacman emerges victorious! Score: 967
Pacman emerges victorious! Score: 977
Pacman emerges victorious! Score: 973
Pacman emerges victorious! Score: 982
Pacman died! Score: -249
Pacman emerges victorious! Score: 914
Pacman emerges victorious! Score: 961
Average Score: 842.5
Scores:        976.0, 965.0, 959.0, 967.0, 977.0, 973.0, 982.0, -249.0, 914.0, 961.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win
```

(a,d): One ghost and random ghosts with fixed seed

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ReflexAgent -l smallClassic -k 1 -n 10 -q -f
Pacman emerges victorious! Score: 926
Pacman emerges victorious! Score: 977
Pacman emerges victorious! Score: 948
Pacman emerges victorious! Score: 983
Pacman emerges victorious! Score: 919
Pacman emerges victorious! Score: 755
Pacman emerges victorious! Score: 975
Pacman emerges victorious! Score: 725
Pacman emerges victorious! Score: 955
Pacman emerges victorious! Score: 872
Average Score: 903.5
Scores:        926.0, 977.0, 948.0, 983.0, 919.0, 755.0, 975.0, 725.0, 955.0, 872.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

(a,e): One ghost and non random ghost

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ReflexAgent -l smallClassic -k 1 -n 10 -q -g DirectionalGhost
Pacman emerges victorious! Score: 971
Pacman emerges victorious! Score: 977
Pacman emerges victorious! Score: 972
Pacman emerges victorious! Score: 1151
Pacman emerges victorious! Score: 959
Pacman emerges victorious! Score: 951
Pacman emerges victorious! Score: 964
Pacman emerges victorious! Score: 970
Pacman emerges victorious! Score: 901
Pacman emerges victorious! Score: 983
Average Score: 979.9
Scores:        971.0, 977.0, 972.0, 1151.0, 959.0, 951.0, 964.0, 970.0, 901.0, 983.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

(b,c): Two ghosts and random ghost

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ReflexAgent -l smallClassic -k 2 -n 10 -q
Pacman emerges victorious! Score: 977
Pacman emerges victorious! Score: 952
Pacman died! Score: -51
Pacman emerges victorious! Score: 927
Pacman emerges victorious! Score: 825
Pacman emerges victorious! Score: 974
Pacman emerges victorious! Score: 984
Pacman emerges victorious! Score: 944
Pacman emerges victorious! Score: 961
Pacman died! Score: -165
Average Score: 732.8
Scores:        977.0, 952.0, -51.0, 927.0, 825.0, 974.0, 984.0, 944.0, 961.0, -165.0
Win Rate:      8/10 (0.80)
Record:        Win, Win, Loss, Win, Win, Win, Win, Win, Win, Loss
```

(b,d): Two ghosts and random ghost with fixed seed

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ReflexAgent -l smallClassic -k 2 -n 10 -q -f
Pacman died! Score: -198
Pacman emerges victorious! Score: 981
Pacman died! Score: -241
Pacman emerges victorious! Score: 958
Pacman emerges victorious! Score: 949
Pacman emerges victorious! Score: 770
Pacman emerges victorious! Score: 1134
Pacman emerges victorious! Score: 941
Pacman emerges victorious! Score: 954
Pacman died! Score: -152
Average Score: 609.6
Scores:        -198.0, 981.0, -241.0, 958.0, 949.0, 770.0, 1134.0, 941.0, 954.0, -152.0
Win Rate:      7/10 (0.70)
Record:        Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Loss
```

(b,e): Two ghosts and non random ghost

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ReflexAgent -l smallClassic -k 2 -n 10 -q -g DirectionalGhost
Pacman emerges victorious! Score: 978
Pacman died! Score: -295
Pacman died! Score: -76
Pacman died! Score: -119
Pacman died! Score: 114
Pacman emerges victorious! Score: 962
Pacman emerges victorious! Score: 963
Pacman died! Score: -66
Pacman emerges victorious! Score: 937
Pacman died! Score: -322
Average Score: 307.6
Scores:        978.0, -295.0, -76.0, -119.0, 114.0, 962.0, 963.0, -66.0, 937.0, -322.0
Win Rate:      4/10 (0.40)
Record:        Win, Loss, Loss, Loss, Loss, Win, Win, Loss, Win, Loss
```

The more ghosts there are the more the difficulty is for Pac Man thus we have more losses. We can see that with a non random movement Pac Man has more difficulty to get maximum scores.

II-Minimax

Minimax is a search algorithm used in two-player games, such as Pacman vs. ghosts. Each player alternates turns, aiming to minimize the possible loss for a worst-case scenario. In the context of Pacman, the MinimaxAgent class extends MultiAgentSearchAgent, implementing a recursive depth-first search that evaluates game states at varying depths.

We will compute the minimax function first:

```python
def minimax(self, gameState, agentIndex, depth):
    if depth is self.depth * gameState.getNumAgents() or gameState.isLose() or gameState.isWin():
        return self.evaluationFunction(gameState)
    if agentIndex == 0:
        return self.maxval(gameState, agentIndex, depth)[1]
    else:
        return self.minval(gameState, agentIndex, depth)[1]
```

The minimax function takes three parameters: the current gameState, the agentIndex (0 for Pacman and 1+ for ghosts), and the depth of recursion.
If depth equals the maximum depth (calculated as self.depth * gameState.getNumAgents()) or if the gameState is a losing or winning state, the recursion terminates. It returns the evaluation of the gameState, calculated by self.evaluationFunction.
For Pac Man (Max Player), If agentIndex is 0 (Pacman), it calls the maxval function to determine the best move for Pacman and returns the action part (index 1 of the returned tuple). And for the Ghosts (Min Players), if agentIndex is greater than 0, it calls the minval function to find the optimal move from the ghosts' perspective and returns the action.

We will now implement the max value and the min value functions:

```
def maxval(self, gameState, agentIndex, depth):
    bestAction = ("max",-float("inf"))
    for action in gameState.getLegalActions(agentIndex):
        succAction = (action,self.minimax(gameState.generateSuccessor(agentIndex,action),(depth + 1)%gameState.getNumAgents(),depth+1))
        bestAction = max(bestAction,succAction,key=lambda x:x[1])
    return bestAction
```

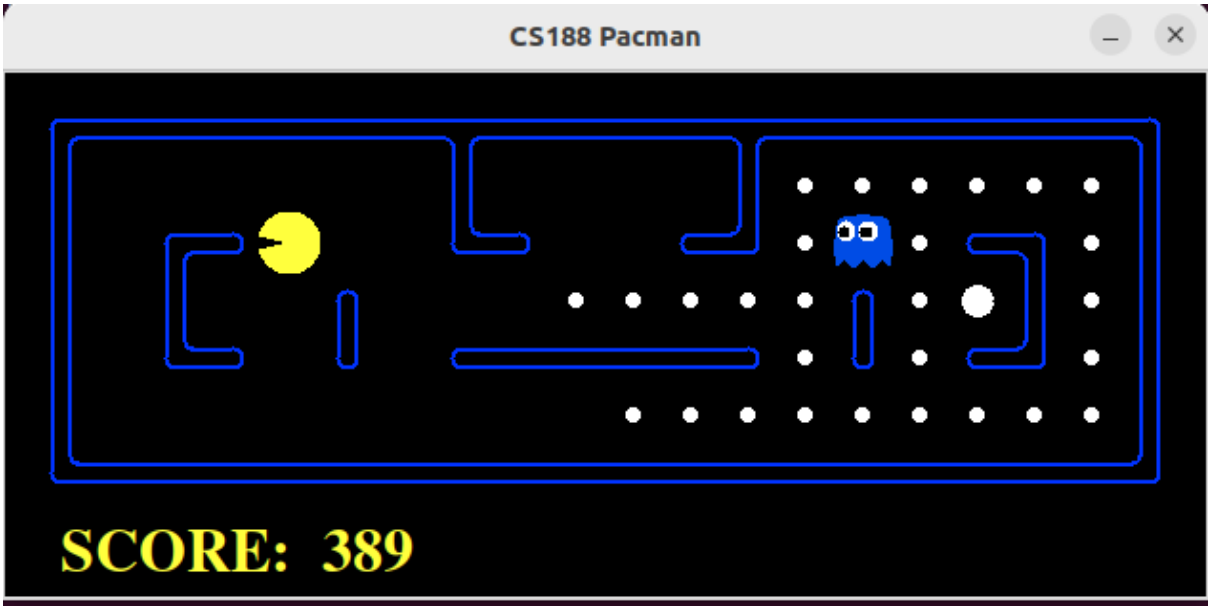
bestAction starts as a tuple with "max" and negative infinity, effectively setting a very low initial maximum score. The function iterates over all legal actions available to the agent. For each action, it computes the successor state using gameState.generateSuccessor. It recursively calls minimax on the successor state, adjusting the agentIndex and incrementing depth. This accounts for the turn sequence in the game (from Pacman to each ghost and back).It updates bestAction with the maximum value returned, comparing the current bestAction with the new action's result (succAction). It uses the second element of the tuple for comparison (which is the minimax value). Then it returns the action that leads to the highest value.

```
def minval(self, gameState, agentIndex, depth):
    bestAction = ("min",float("inf"))
    for action in gameState.getLegalActions(agentIndex):
        succAction = (action,self.minimax(gameState.generateSuccessor(agentIndex,action),(depth + 1)%gameState.getNumAgents(),depth+1))
        bestAction = min(bestAction,succAction,key=lambda x:x[1])
    return bestAction
```

Similar to maxval, but it initializes bestAction with positive infinity, setting a high initial minimum. The loop for is similar to maxval, but for ghosts. Each action's successor state is evaluated by calling minimax. It seeks the minimal value (opposite of maxval), updating bestAction based on which action leads to the lowest score, using a lambda to compare values. Finally it returns the action that results in the minimum value.
The maxval and minval are very similar. The initialization and the loop are the same. It is on the maximization and the minimization which has some differences. In fact we set the best value to “-inf” as it is the worst case for the ghosts. On the one hand maxval seeks the highest possible outcome from available moves. On the other hand minval searches for the lowest possible outcome to constrain Pacman's options.

Testing the code:
One ghost:

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p MinimaxAgent -a depth=3 -l smallClassic -k 1
```



Sometimes Pac Man seems to not move and stay at a place. He waits for the ghost to approach before moving. This problem is caused by the depth. In fact our algorithm does not see faraway, resulting in Pac Man preferring not moving and staying far away from the ghost. Rather than taking the risk to eat the dot and be close to the ghost.

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p MinimaxAgent -a depth=3 -l smallClassic -k 1
Pacman emerges victorious! Score: 1043
Average Score: 1043.0
Scores:        1043.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Two ghosts:

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p MinimaxAgent -a depth=3 -l smallClassic -k 2
```


```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p MinimaxAgent -a depth=3 -l smallClassic -k 2
Pacman emerges victorious! Score: 1048
Average Score: 1048.0
Scores:        1048.0
Win Rate:      1/1 (1.00)
Record:        Win
```

We can see that in both cases here. Pac Man seems to win. But on multiple runs, Pac Man ends up losing precisely when there are 2 ghosts which can cornered him more often.

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p MinimaxAgent -a depth=3 -l smallClassic -k 2 -q -n 5
Pacman emerges victorious! Score: 783
Pacman died! Score: -125
Pacman died! Score: -244
Pacman died! Score: 154
Pacman died! Score: -165
Average Score: 80.6
Scores:        783.0, -125.0, -244.0, 154.0, -165.0
Win Rate:      1/5 (0.20)
Record:        Win, Loss, Loss, Loss, Loss
```

III-Alpha Beta

The AlphaBetaAgent improves on the Minimax algorithm by incorporating an alpha-beta pruning, which reduces the number of nodes that need to be evaluated during the decision process. This enhancement helps the computation without deleting the quality of the decisions, making it a more efficient choice compared to the standard Minimax approach.

At the start of the search process the getAction function is called with the current game state. This function sets initial values for alpha (-infinity) and beta (infinity), representing the

best already explored options for the maximizer (Pacman) and minimizer (ghosts), respectively.

```
def max_value(gameState, depth, alpha, beta):
    actionList = gameState.getLegalActions(0) # Get actions of pacman
    if len(actionList) == 0 or gameState.isWin() or gameState.isLose() or depth == self.depth:
        return (self.evaluationFunction(gameState), None)

    v = -(float("inf"))
    goAction = None

def min_value(gameState, agentID, depth, alpha, beta):
    actionList = gameState.getLegalActions(agentID) # Get the actions of the ghost
    if len(actionList) == 0:
      return (self.evaluationFunction(gameState), None)

    v = float("inf")
    goAction = None
```

As the agent explores potential moves for Pacman, it updates alpha if it finds a better option. If the agent encounters a situation where it realizes that pursuing a particular branch won't lead to a better outcome than what has already been established (since the outcome would be worse than beta), it prunes this branch to avoid wasting computational resources.

```
for thisAction in actionList:
    successorValue = min_value(gameState.generateSuccessor(0, thisAction), 1, depth, alpha, beta)[0]
    if (v < successorValue):
        v, goAction = successorValue, thisAction

    if (v > beta):
        return (v, goAction)

    alpha = max(alpha, v)
```

Similarly, when evaluating moves for the ghosts, the agent updates beta if a less favorable outcome for Pacman is identified. If it determines that continuing down a certain path would lead to an outcome better than current possibilities for Pacman (meaning it's less than alpha), this path is also pruned, as Pacman would avoid it in an optimal play scenario.

```
for thisAction in actionList:
    if (agentID == gameState.getNumAgents() - 1):
        successorValue = max_value(gameState.generateSuccessor(agentID, thisAction), depth + 1, alpha, beta)[0]
    else:
        successorValue = min_value(gameState.generateSuccessor(agentID, thisAction), agentID + 1, depth, alpha, beta)[0]
    if (successorValue < v):
        v, goAction = successorValue, thisAction

    if (v < alpha):
        return (v, goAction)

    beta = min(beta, v)
```

When comparing the AlphaBetaAgent to the MinimaxAgent, it is much more efficient due to its ability to delete branches that won't affect the final decision. Both algorithms strive to find the best possible move by simulating future moves assuming perfect play, but the AlphaBetaAgent's pruning ability allows it to evaluate fewer nodes. This efficiency makes it better suited for complex scenarios or games that require quick decision-making.

Testing the code:
One ghost:

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p AlphaBetaAgent -a depth=3 -l smallClassic -k 1
Pacman emerges victorious! Score: 998
Average Score: 998.0
Scores:        998.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Two ghosts:

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p AlphaBetaAgent -a depth=3 -l smallClassic -k 2
Pacman emerges victorious! Score: 1112
Average Score: 1112.0
Scores:        1112.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Similar to the Minimax algorithms, Pac Mac sometimes prefers to not move in order to avoid the ghost and not dying. However we can see that with the Alpha Beta Algorithm, when there are 2 ghosts it seems to perform better.

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p AlphaBetaAgent -a depth=3 -l smallClassic -k 2 -q -n 5
Pacman emerges victorious! Score: 1292
Pacman died! Score: -411
Pacman emerges victorious! Score: 918
Pacman emerges victorious! Score: 1481
Pacman died! Score: -731
Average Score: 509.8
Scores:        1292.0, -411.0, 918.0, 1481.0, -731.0
Win Rate:      3/5 (0.60)
Record:        Win, Loss, Win, Win, Loss
```

IV-Expectimax

The Expectimax algorithm is an alternative to the Minimax algorithm, particularly useful in scenarios where opponents may not always make the best possible moves. Instead of strictly considering the worst-case scenario like Minimax, Expectimax uses a probabilistic approach to evaluate the outcomes of decisions, making it better suited for games involving elements of chance or unpredictable behavior.

In the Expectimax approach, we still begin by examining the possible actions for the maximizing player, which in this case is Pacman.

```python
def max_value(gameState, depth):
    " Cases checking "
    actionList = gameState.getLegalActions(0) # Get actions of pacman
    if len(actionList) == 0 or gameState.isWin() or gameState.isLose() or depth == self.depth:
        return (self.evaluationFunction(gameState), None)

    " Initializing the value of v and action to be returned "
    v = -(float("inf"))
    goAction = None

    for thisAction in actionList:
        successorValue = exp_value(gameState.generateSuccessor(0, thisAction), 1, depth)[0]

        " max(v, successorValue) "
        if (v < successorValue):
            v, goAction = successorValue, thisAction

    return (v, goAction)
```

This function looks at all possible moves Pacman can make from the current game state. For each action, it generates a successor state which represents the game state after Pacman takes that action. It then calls the exp_value function (to handle the ghosts' turns) for each of these successor states and finds out the expected utility of these actions. The action that leads to the highest utility is selected. This utility is determined by taking the maximum value returned by the exp_value function for the ghosts. This reflects a more realistic scenario where the ghosts might make suboptimal moves.

For each ghost, the algorithm calculates the average outcome of all possible moves from that state, weighting each by its probability.

```
def exp_value(gameState, agentID, depth):
    " Cases checking "
    actionList = gameState.getLegalActions(agentID) # Get the actions of the ghost
    if len(actionList) == 0:
      return (self.evaluationFunction(gameState), None)

    " Initializing the value of v and action to be returned "
    v = 0
    goAction = None

    for thisAction in actionList:
        if (agentID == gameState.getNumAgents() - 1):
            successorValue = max_value(gameState.generateSuccessor(agentID, thisAction), depth + 1)[0]
        else:
            successorValue = exp_value(gameState.generateSuccessor(agentID, thisAction), agentID + 1, depth)[0]

        probability = successorValue/len(actionList)
        v += probability

    return (v, goAction)
```

This function is for ghost agents. It starts by examining all possible actions a ghost can take. Like the max_value function, for each action, it generates a successor state. If the current ghost is the last ghost (checking agentID == gameState.getNumAgents() - 1), it calls max_value to return control to Pacman, moving to the next level of depth in the game tree. If not the last ghost, it recursively calls exp_value for the next ghost. For each successor state, the expected value is calculated by considering the utility value returned divided by the number of possible actions (assuming equal probability of each action). This averages the potential outcomes. The cumulative expected value of all actions is calculated and this value is what the exp_value returns. This allows the algorithm to understand what might happen next, rather than preparing for the worst-case scenario. The final decision for Pacman is then based on which move leads to the highest score, considering both optimal and suboptimal responses from the ghosts.
Thus while Minimax provides a strong strategy under the assumption of optimal play from all players, Expectimax offers a more flexible and often more realistic approach for scenarios where opponents might not always make the best moves.

Testing the code:
One ghost:

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ExpectimaxAgent -a depth=3 -l smallClassic -k 1
Pacman emerges victorious! Score: 935
Average Score: 935.0
Scores:        935.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Two ghosts:

```
tai22@Taitaii:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ExpectimaxAgent -a depth=3 -l smallClassic -k 2
Pacman emerges victorious! Score: 1694
Average Score: 1694.0
Scores:        1694.0
Win Rate:      1/1 (1.00)
Record:        Win
```

We can see that this algorithm performs better than the previous ones and has a higher score than previous.

V-Comparisons

The situation is a two ghosts case in a smallClassic with 10 runs for each algorithm.

❖ Minimax:

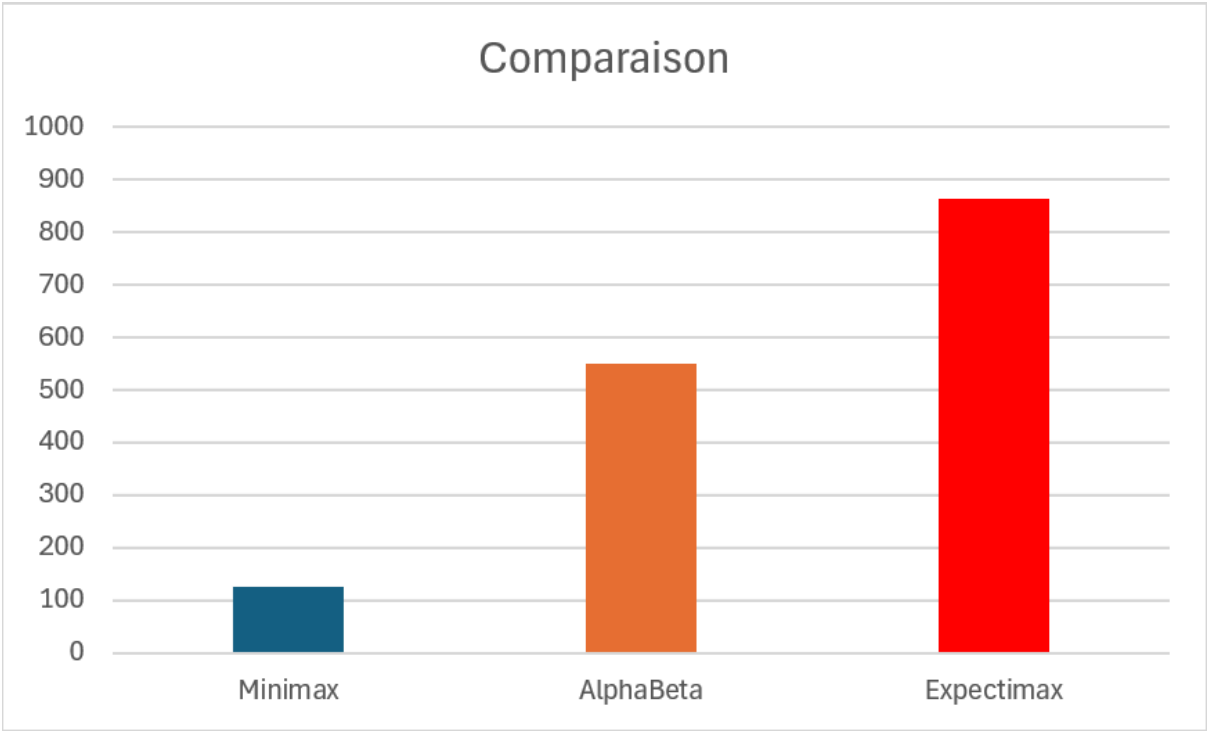
```
tai22@Taitail:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p MinimaxAgent -a depth=3 -l smallClassic -k 2 -q -n 10 --frameTime 0
Pacman died! Score: -67
Pacman died! Score: -95
Pacman died! Score: -397
Pacman died! Score: 54
Pacman emerges victorious! Score: 999
Pacman died! Score: -891
Pacman emerges victorious! Score: 1025
Pacman died! Score: -136
Pacman emerges victorious! Score: 795
Pacman died! Score: -25
Average Score: 126.2
Scores:        -67.0, -95.0, -397.0, 54.0, 999.0, -891.0, 1025.0, -136.0, 795.0, -25.0
Win Rate:      3/10 (0.30)
Record:        Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Win, Loss
```

❖ Alpha Beta:

```
tai22@Taitail:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p AlphaBetaAgent -a depth=3 -l smallClassic -k 2 -q -n 10 --frameTime 0
Pacman died! Score: -165
Pacman emerges victorious! Score: 1276
Pacman died! Score: -29
Pacman died! Score: 34
Pacman emerges victorious! Score: 1359
Pacman died! Score: 364
Pacman emerges victorious! Score: 1109
Pacman died! Score: -34
Pacman died! Score: -32
Pacman emerges victorious! Score: 1628
Average Score: 551.0
Scores:        -165.0, 1276.0, -29.0, 34.0, 1359.0, 364.0, 1109.0, -34.0, -32.0, 1628.0
Win Rate:      4/10 (0.40)
Record:        Loss, Win, Loss, Loss, Win, Loss, Win, Loss, Loss, Win
```

❖ Expectimax:

```
tai22@Taitail:~/Documents/IA/multiagent_AIC$ python3 pacman_AIC.py -p ExpectimaxAgent -a depth=3 -l smallClassic -k 2 -q -n 10 --frameTime 0
Pacman died! Score: -468
Pacman emerges victorious! Score: 1079
Pacman died! Score: -652
Pacman emerges victorious! Score: 947
Pacman emerges victorious! Score: 1454
Pacman emerges victorious! Score: 1322
Pacman emerges victorious! Score: 1037
Pacman emerges victorious! Score: 1472
Pacman emerges victorious! Score: 1301
Pacman emerges victorious! Score: 1135
Average Score: 862.7
Scores:        -468.0, 1079.0, -652.0, 947.0, 1454.0, 1322.0, 1037.0, 1472.0, 1301.0, 1135.0
Win Rate:      8/10 (0.80)
Record:        Loss, Win, Loss, Win, Win, Win, Win, Win, Win, Win
```



This graph is the average scores of each algorithm. The Expectimax algorithm seems to be a better algorithm. In fact, it tries to predict the errors of the ghosts and maximize the score. Moreover with this algorithm, Pac Mac is less scared of dying and will try to eat the dots.

❖ Differents situations

We will try to see the score with the situations seen in the ReflexAgent on 10 runs:

Ghost	Random	Fixed Seed	Non Random				
1				WR%	Random	Fixed Seed	Non Random
Reflex Agent	842,5	903,5	979,9	Reflex Agent	90%	100%	100%
Minimax	128,4	609,4	1219,4	Minimax	100%	100%	100%
Alpha Beta	736,3	609,4	1192,9	Alpha Beta	100%	100%	100%
Expectimax	557,1	319,5	1292,1	Expectimax	100%	100%	100%
2				WR%	Random	Fixed Seed	Non Random
Reflex Agent	732,8	609,6	307,6	Reflex Agent	80%	70%	40%
Minimax	202,6	709,6	-132,4	Minimax	20%	60%	0%
Alpha Beta	444,2	709,6	370,7	Alpha Beta	40%	60%	30%
Expectimax	327,6	912,3	266	Expectimax	30%	70%	20%

After the 10 runs, we can see that if there is only one ghost in the small maze, Reflex Agent and Alpha Beta are the best options because they are working for smaller environments. However, having more ghosts or a bigger maze, Expetimax seems to be the best algorithm for this situation.

VI-Conclusion

To conclude, we saw multiple reinforcement learning algorithms to enhance the Pac Man behavior. To sum up the algorithms:

- Minimax is a search algorithm used in two-player games, such as Pacman vs. ghosts. Each player alternates turns, aiming to minimize the possible loss for a worst-case scenario.
- Alpha Beta improves on the Minimax algorithm by incorporating an alpha-beta pruning, which reduces the number of nodes that need to be evaluated during the decision process. This enhancement helps the computation without deleting the quality of the decisions and make it faster
- Expectimax algorithm is an alternative to the Minimax algorithm, particularly useful in scenarios where opponents may not always make the best possible moves. Instead of strictly considering the worst-case scenario like Minimax making it better suited for games involving elements of chance or unpredictable behavior.