Artificial Intelligence

Lab 4

Groupe: William TRAVERS The Tai VUONG

Sommaire:

I-Value Iteration

Value Iteration is a method used in reinforcement learning to compute an optimal policy by iteratively improving the estimates of state values. The process consists of the following steps:
Initialization: All state values are initialized to zero.
Iteration: For a specified number of iterations, the value of each state is updated based on the expected utility of taking each possible action from that state, considering the rewards and transitions.
The core of the Value Iteration algorithm involves updating the state value using the formula:

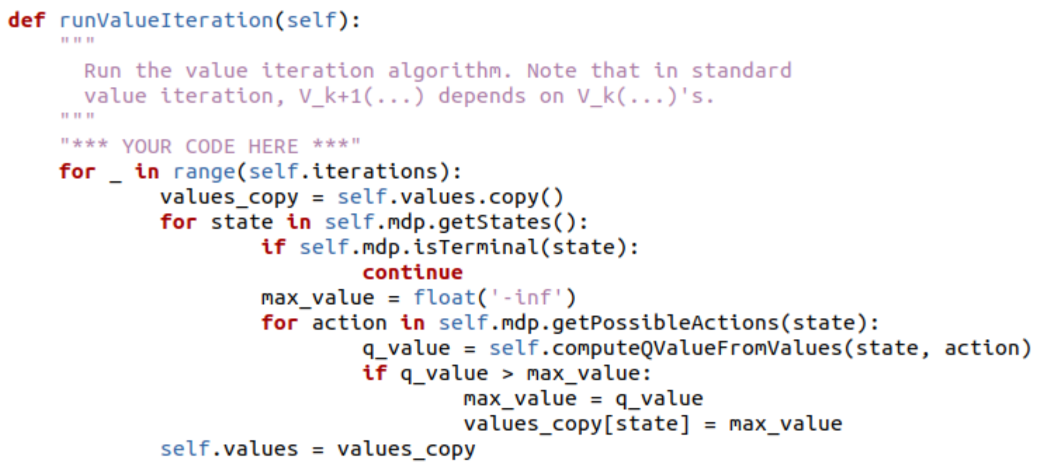
$$\forall s \in S, V_{k+1}(s) = max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V_k(s')]$$

The methods to implement are:
- runValueIteration(): runs the value iteration algorithm self.iterations times
- computeActionFromValues(state): computes the best action according to the state value function (self.values[i] is the current state value of state i).

- computeQ-valueFromValues(state, action): returns the q-value of the (state, action) pair given by the state value function given by self.values.

❖ RunValueIteration

It estimates the values of each state by calculating and using the best expected outcomes from each possible action.



```
def runValueIteration(self):
    """
      Run the value iteration algorithm. Note that in standard
      value iteration, V_k+1(...) depends on V_k(...)'s.
    """
    "*** YOUR CODE HERE ***"
    for _ in range(self.iterations):
            values_copy = self.values.copy()
            for state in self.mdp.getStates():
                    if self.mdp.isTerminal(state):
                            continue
                    max_value = float('-inf')
                    for action in self.mdp.getPossibleActions(state):
                            q_value = self.computeQValueFromValues(state, action)
                            if q_value > max_value:
                                    max_value = q_value
                                    values_copy[state] = max_value
            self.values = values_copy
```

for _ in range(self.iterations): This loop runs the value iteration algorithm for a specified number of iterations. Each iteration corresponds to an update of the value function across all states based on the future rewards.

values_copy = self.values.copy(): Before updating the values, the current state values are copied. In order to ensure that updates within a single iteration don't affect the calculations of other state values during the same iteration.

for state in self.mdp.getStates(): It iterates over every state in the MDP. For each state, it performs the following steps unless the state is a terminal state:

> If self.mdp.isTerminal(state): continue
> If the state is terminal, it skips to the next state since no actions are possible from a terminal state.
>
> max_value = float('-inf'): Initializes the maximum value for this state as negative infinity. This variable will store the highest value found for actions available in the current state.

for action in self.mdp.getPossibleActions(state): It iterates through all possible actions available from the current state.

> q_value = self.computeQValueFromValues(state, action)
> For each action, calculate the Q-value. This value represents the expected utility of an action in a given state, based on the estimated future rewards.

if q_value > max_value: If the Q-value for the current action is greater than the maximum value for this state, update the maximum value.

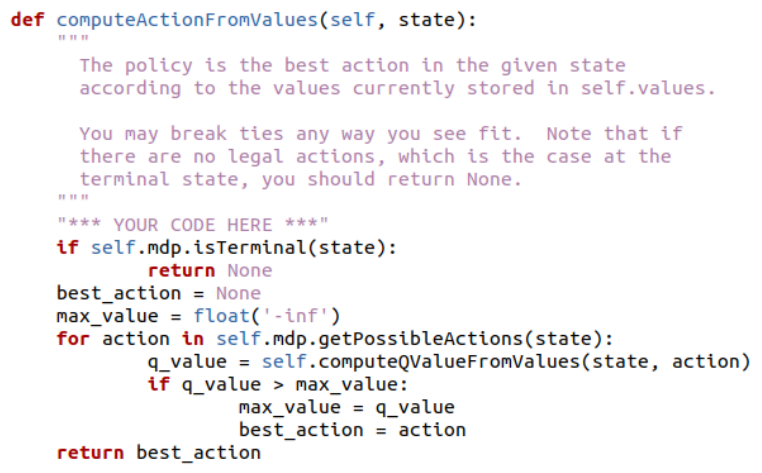
> max_value = q_value Sets the maximum value to this higher Q-value.

values_copy[state] = max_value: Once all actions for a state have been considered, the maximum value found is assigned to this state in the copied state. It updates the value of the state based on the best possible action as determined by the Q-values.

self.values = values_copy: After completing an iteration for all states, the original state values are updated with the values from the copy. It ensures that all state values are updated, maintaining a good update requirement of value iteration.

❖ ComputeActionFromValues

ComputeActionFromValues determines the best action to take from a given state in a Markov Decision Process based on the value iteration results. It uses the values that have been previously computed for each state to make this decision.



```
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    if self.mdp.isTerminal(state):
            return None
    best_action = None
    max_value = float('-inf')
    for action in self.mdp.getPossibleActions(state):
            q_value = self.computeQValueFromValues(state, action)
            if q_value > max_value:
                    max_value = q_value
                    best_action = action
    return best_action
```

if self.mdp.isTerminal(state): return None: It checks if the current state is terminal state. If it is terminal, the function returns None because no action is needed or possible.

best_action = None: This variable will hold the best action found.
max_value = float('-inf'): This sets up max_value as negative infinity to ensure that any actual Q-value calculated will be larger. max_value will store the highest Q-value found through all possible actions.

for action in self.mdp.getPossibleActions(state): It iterates over all actions that can be taken from the current state.

> q_value = self.computeQValueFromValues(state, action) - For each action, the Q-value is calculated using the computeQValueFromValues function. This function computes the expected action in the current state, based on the values from previous iterations.

if q_value > max_value: It checks if the Q-value for the current action is greater than the current maximum value.

> max_value = q_value - If it is, update max_value to this new value.
> best_action = action - It updates best_action to the current action, as the best one found.

return best_action: After evaluating all possible actions, the function returns the action that has the highest Q-value leading to the highest expected utility.

❖ ComputeQvalueFromValues

ComputeQValueFromValues calculates the Q-value for an action taken in a given state.

```
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    q_value = 0
    for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
            reward = self.mdp.getReward(state, action, next_state)
            q_value += prob * (reward + self.discount * self.values[next_state])
    return q_value
```

q_value = 0: It initializes the Q-value at zero. It is important because the Q-value will add results from different potential outcomes.

for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action): It goes through each possible next state given the current state and action. For each next state, the MDP provides a probability of moving to that state from the current state when the given action is taken.

reward = self.mdp.getReward(state, action, next_state): It retrieves the reward received when moving from the current state to the next state via the specified action. This reward is a critical component of the Q-value calculation because it provides immediate feedback about the desired action.

q_value += prob * (reward + self.discount * self.values[next_state]): This updates the running total of the Q-value. This equation uses:

prob: Transition to the next state.

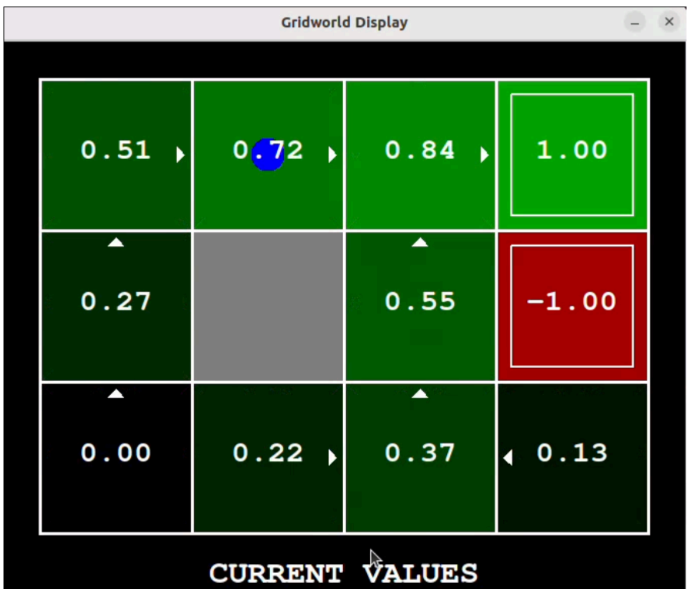
reward: Immediate reward received for the transition.

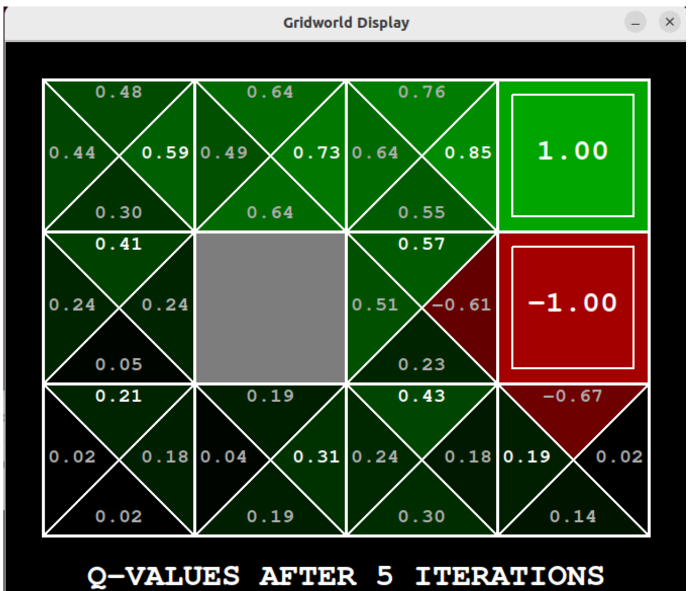
self.discount: Discount factor that reduces the value of future rewards, It expects immediate rewards over distant rewards.

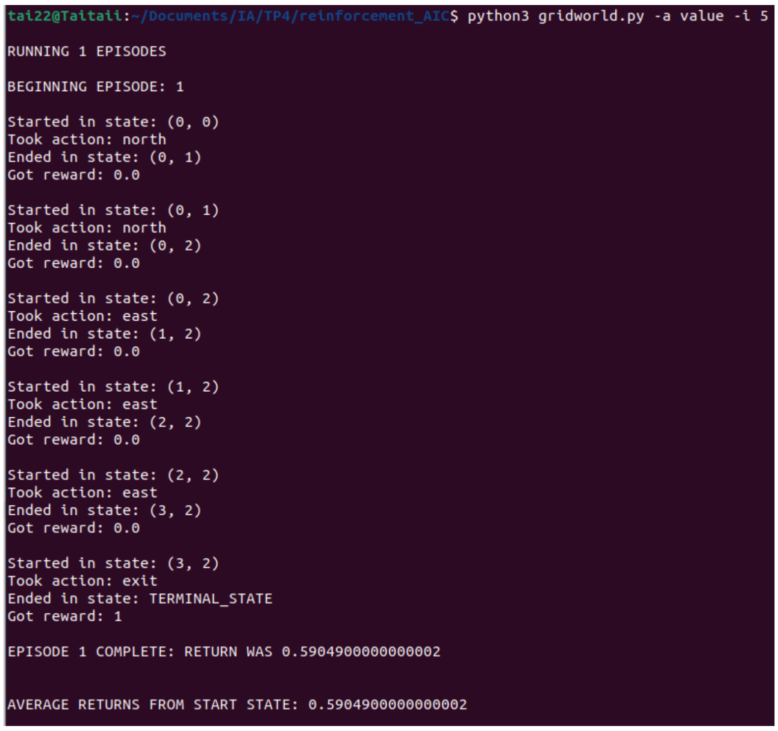
self.values[next_state]: Estimated value of the next state coming from previous computed values, It is the expected future benefits from that state.

return q_value: It outputs the calculated Q-value, which counts the expected utility of choosing a particular action in the given state. This value is important for the agent to decide the best action to take in each state during the value iteration process.

Testing the code:
5 iterations:



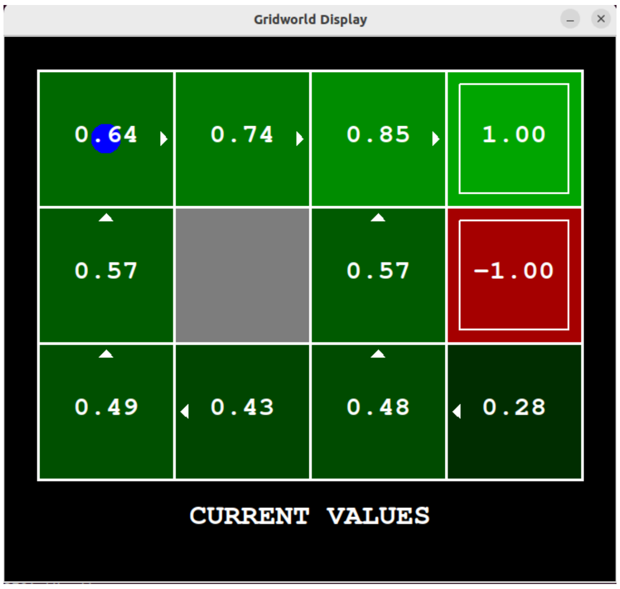


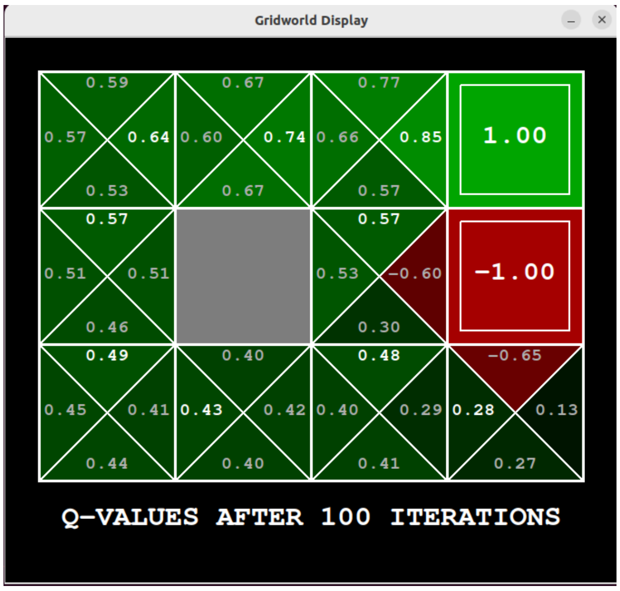


```
tai22@Taitai:~/Documents/IA/TP4/reinforcement_AI$ python3 gridworld.py -a value -i 5

RUNNING 1 EPISODES

BEGINNING EPISODE: 1

Started in state: (0, 0)
Took action: north
Ended in state: (0, 1)
Got reward: 0.0

Started in state: (0, 1)
Took action: north
Ended in state: (0, 2)
Got reward: 0.0

Started in state: (0, 2)
Took action: east
Ended in state: (1, 2)
Got reward: 0.0

Started in state: (1, 2)
Took action: east
Ended in state: (2, 2)
Got reward: 0.0

Started in state: (2, 2)
Took action: east
Ended in state: (3, 2)
Got reward: 0.0

Started in state: (3, 2)
Took action: exit
Ended in state: TERMINAL_STATE
Got reward: 1

EPISODE 1 COMPLETE: RETURN WAS 0.5904900000000002


AVERAGE RETURNS FROM START STATE: 0.5904900000000002
```

The first image shows the estimated values for each state, it is the best move for Pac Man in order to maximize the rewards. Green squares represent higher values indicating much more favorable positions. Whereas a red square indicates negative outcomes such as ghosts. And the arrows represent the optimal directions to move.
The second image is the calculation of Q-value after 5 iterations for each possible action for each state split in 4 directions showing the probability for moving to a good position. After the 5 iterations, the Q-values for rewards are more than the Q-values after 1 iteration.
Thus, Q-values will help and change the behavior of Pac Man to a seeking reward and avoiding ghosts attitude, improving the decision-making at each state.

100 iterations:





We can see after 100 iterations, the Q-values for rewards are more defined and it will incite the Pac Man to follow the correct path to follow in order to get the reward while avoiding the ghosts.

II-Q-learning

Q-Learning is an off-policy learner that aims to find the best action to take given the current state. Unlike policy-based strategies, it directly approximates the optimal action-value function independent of the policy being followed.

❖ __init__

```python
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)

    "*** YOUR CODE HERE ***"
    self.qValues = util.Counter()
```

In the init method create a dictionary to store the q-values of (state,action) already explored.

❖ Update

```python
def update(self, state, action, nextState, reward: float):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here
      NOTE: You should never call this function,
      it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"
    sample = reward + self.discount * self.computeValueFromQValues(nextState)
    self.qValues[(state,action)] = (1 - self.alpha) * self.getQValue(state, action) + self.alpha * sample
```

It computes the new q-value using the Update rule of the Q-learning algorithm.

2. Update $Q(s,a)$: $Q(s,a) = (1-\alpha)Q(s,a) + \alpha\ sample$

❖ ComputeValueFromQValues

```python
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    "*** YOUR CODE HERE ***"
    legalActions = self.getLegalActions(state)
    if not legalActions:  # Check for terminal state
            return 0.0
    return max([self.getQValue(state, action) for action in legalActions])
```

It returns maximum q-value Q(state,action) where the max is over legal actions. Note that if there are no legal actions, which is the case at the terminal state, you should return a value of 0.0.

❖ GetQValue

```python
def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    "*** YOUR CODE HERE ***"
    return self.qValues[(state,action)]
```

It returns the q-value Q(state,action). Returns 0.0 if we have never explored a state or the q-value otherwise.

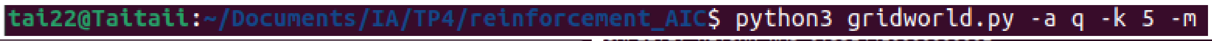
❖ ComputeActionFromQValue

```python
def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    "*** YOUR CODE HERE ***"

    legalActions = self.getLegalActions(state)
    if not legalActions:
            return None
    bestValue = float('-inf')
    bestActions = []
    for action in legalActions:
            qValue = self.getQValue(state, action)
            if qValue > bestValue:
                    bestValue = qValue
                    bestActions = [action]
            elif qValue == bestValue:
                    bestActions.append(action)
    return random.choice(bestActions)
```

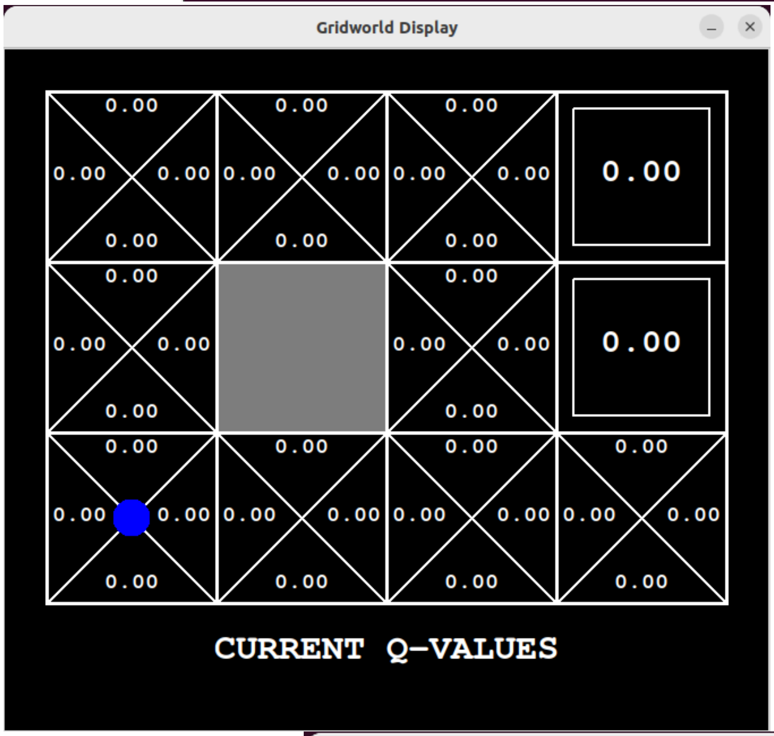
It computes the best action to take in a state. Break ties randomly for better behavior using the random.choice() function. If there are no legal actions, which is the case at the terminal state, it returns None.

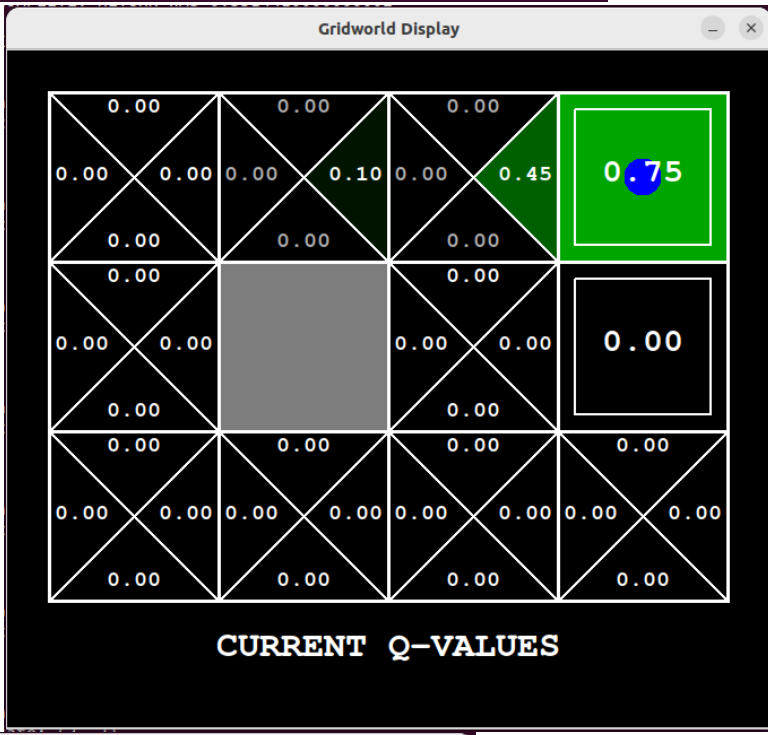
Testing the code

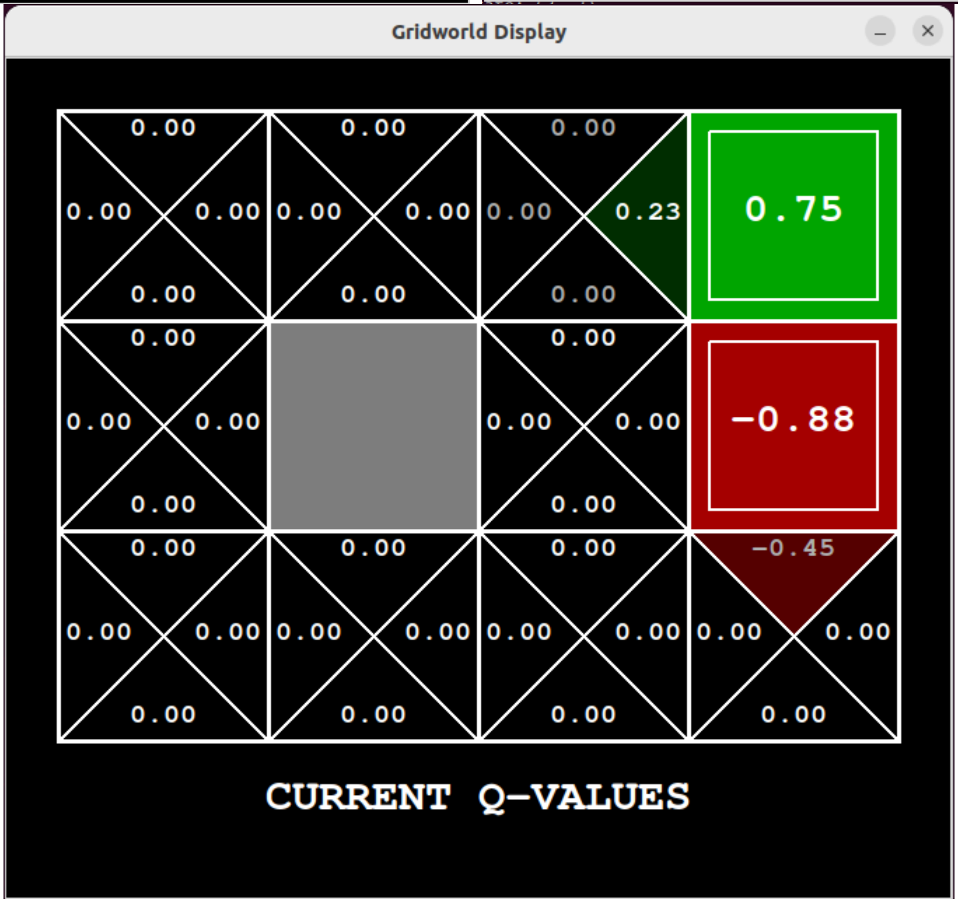
5 iteration and manual mode



```
tai22@Taitaii:~/Documents/IA/TP4/reinforcement_AIC$ python3 gridworld.py -a q -k 5 -m
```





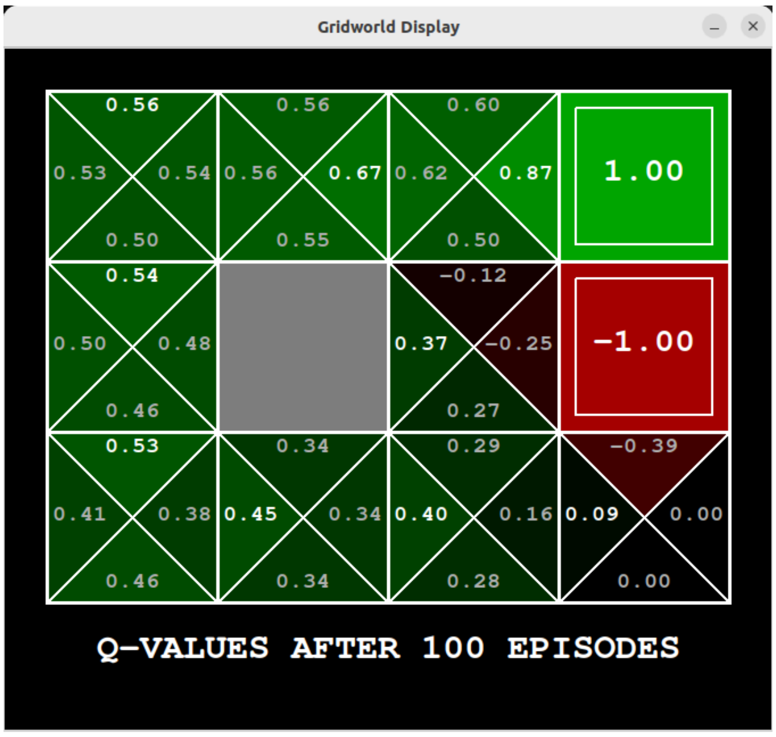


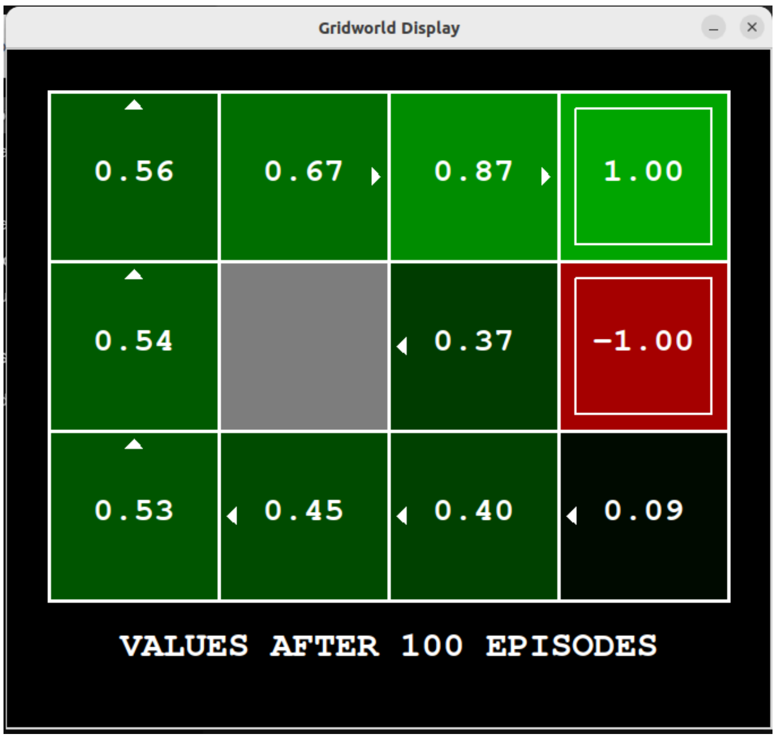
After the 5 iterations, we can see each state split in 4 directions corresponding to the probability of moving toward a reward or a danger.

❖ GetAction

```
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.
      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    if util.flipCoin(self.epsilon):
            action = random.choice(legalActions)
    else:
            action = self.computeActionFromQValues(state)
    return action
```

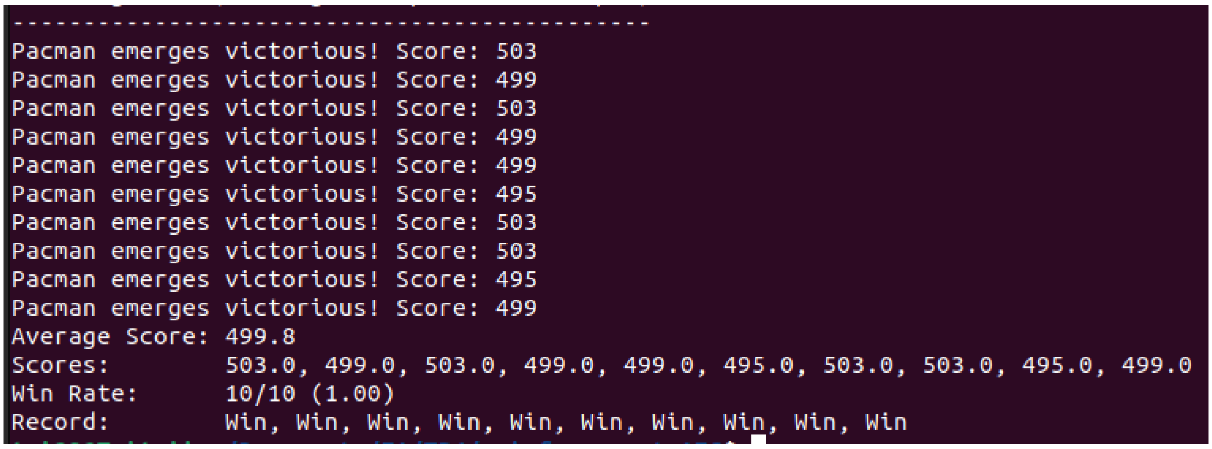
It computes the action to take in the current state. With probability self.epsilon, it should take a random action and take the best policy action otherwise. Note that if there are no legal actions, which is the case at the terminal state, you should choose None as the action.





After the 100 iterations, the Q-values are close to the Value Iterations Algorithm but the values of the Q-learning are quite better for this algorithm when Pac Man move closer to the reward and the Q-values toward the ghosts are smaller making Pac Man moving more for the rewards and optimize its actions.

2000 iterations:
While a total of 2010 games will be played, the first 2000 games will not be displayed because of the option -x 2000, which designates the first 2000 games for training. Thus, we will only see Pacman play the last 10 of these games.



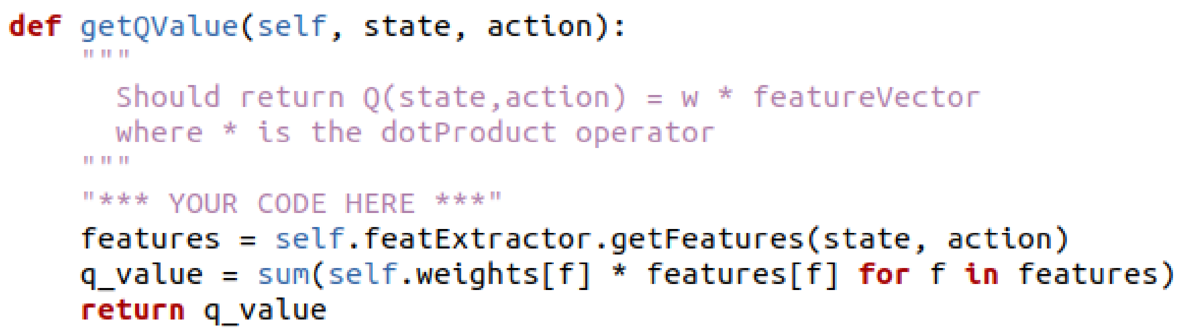
```
--------------------------------------------
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Average Score: 499.8
Scores:        503.0, 499.0, 503.0, 499.0, 499.0, 495.0, 503.0, 503.0, 495.0, 499.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

The Q-learning algorithm is working well on a smaller maze. We count up to a 100% win rate. However when we are in a different situation such as a bigger environment, it takes a lot of time to train and makes Pac Man lose a lot.

III-Approximate Q-learning

Approximate Q-Learning agent will be able to learn from its environment and adjust its strategy based on the rewards it receives and the features of the state-action pairs it encounters. Agent utilizes a set of weights linked to specific features, which are stored in self.weights. The process begins with training the agent to familiarize it with the environment, followed by some tests to evaluate its performance. This method is effective in dealing with large state spaces in traditional Q-learning, where direct calculations are often impossible. In our setup, the Q-value is calculated as a linear combination of the weights assigned to the features of a given state. This approach is to identify the feature values for states and to adjust the weights appropriately based on the agent's real-time interactions and observations.

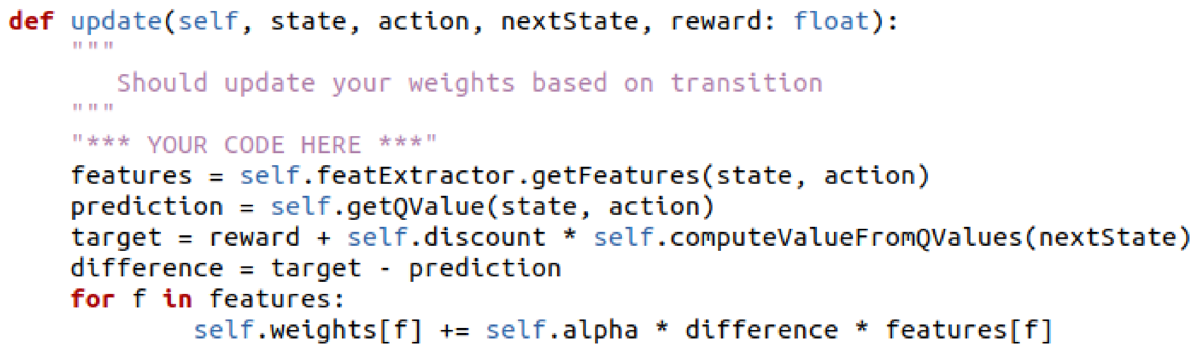
❖ GetQValue



```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    features = self.featExtractor.getFeatures(state, action)
    q_value = sum(self.weights[f] * features[f] for f in features)
    return q_value
```

self.featExtractor.getFeatures(state, action) extracts the features for the given state-action pair.It returns Q(state, action) = w_1 × f_1(state, action) + w_2 × f_2(state, action) + . . . + w_n × f_n(state, action) with n is the size of features = self.featExtractor.getFeatures(state, action)

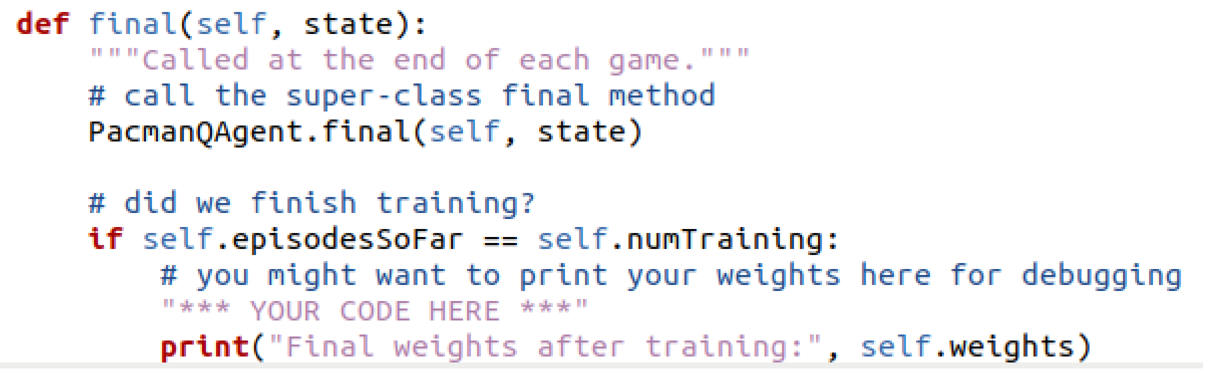
❖ Update



```
def update(self, state, action, nextState, reward: float):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    features = self.featExtractor.getFeatures(state, action)
    prediction = self.getQValue(state, action)
    target = reward + self.discount * self.computeValueFromQValues(nextState)
    difference = target - prediction
    for f in features:
            self.weights[f] += self.alpha * difference * features[f]
```

ComputeValueFromQValues computes the maximum Q-value for the next state. It update your weights based on Approximate Q-Learning algorithm

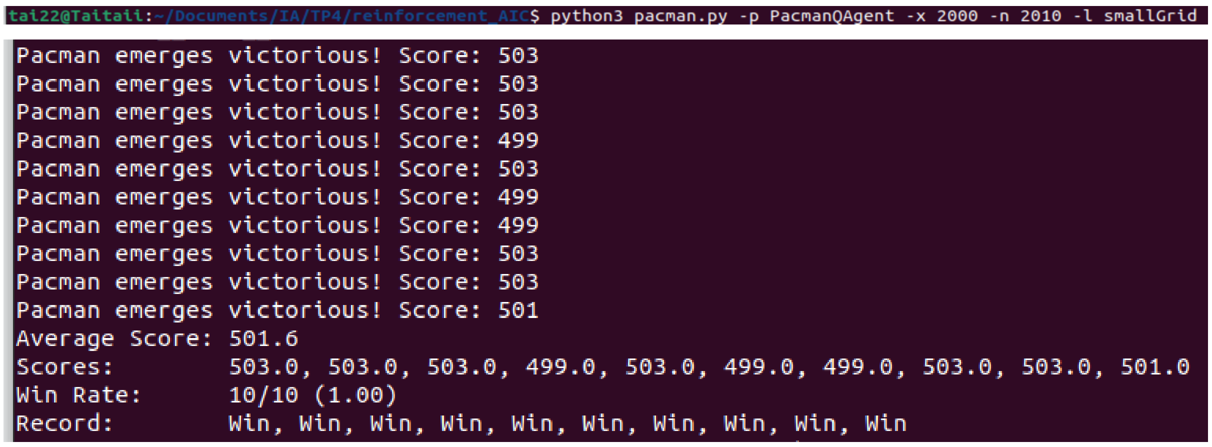
❖ Final



```
def final(self, state):
    """Called at the end of each game."""
    # call the super-class final method
    PacmanQAgent.final(self, state)

    # did we finish training?
    if self.episodesSoFar == self.numTraining:
        # you might want to print your weights here for debugging
        "*** YOUR CODE HERE ***"
        print("Final weights after training:", self.weights)
```

This print the weights here for debugging

Testing the code

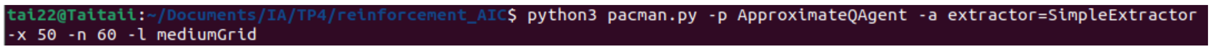


```
tai22@Taitaii:~/Documents/IA/TP4/reinforcement_AIC$ python3 pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

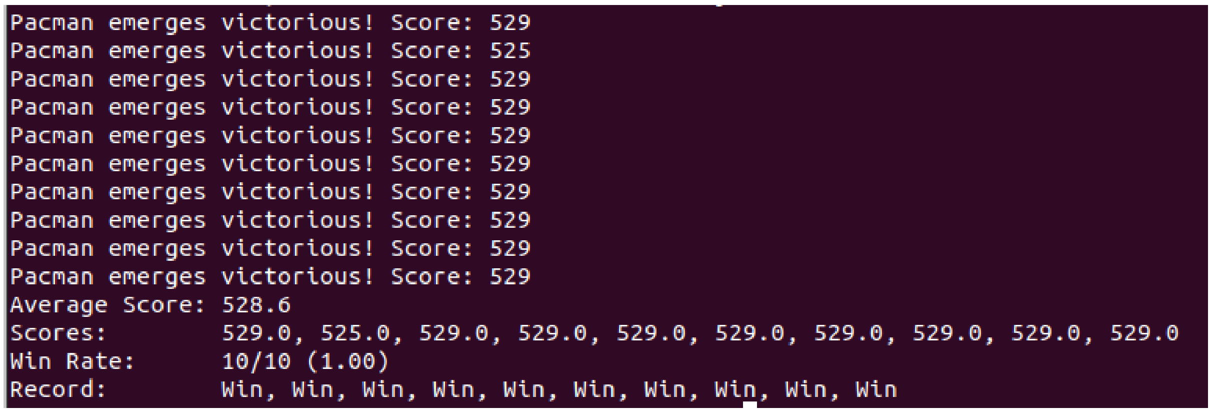
```
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 501
Average Score: 501.6
Scores:        503.0, 503.0, 503.0, 499.0, 503.0, 499.0, 499.0, 503.0, 503.0, 501.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

By default, ApproximateQAgent uses the IdentityExtractor, which assigns a single feature to every (state,action) pair. With this feature extractor, your approximate Q-learning agent should work identically to PacmanQAgent.

50 training games and Medium Grid:

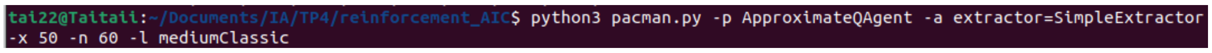


```
tai22@Taitaii:~/Documents/IA/TP4/reinforcement_AIC$ python3 pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumGrid
```

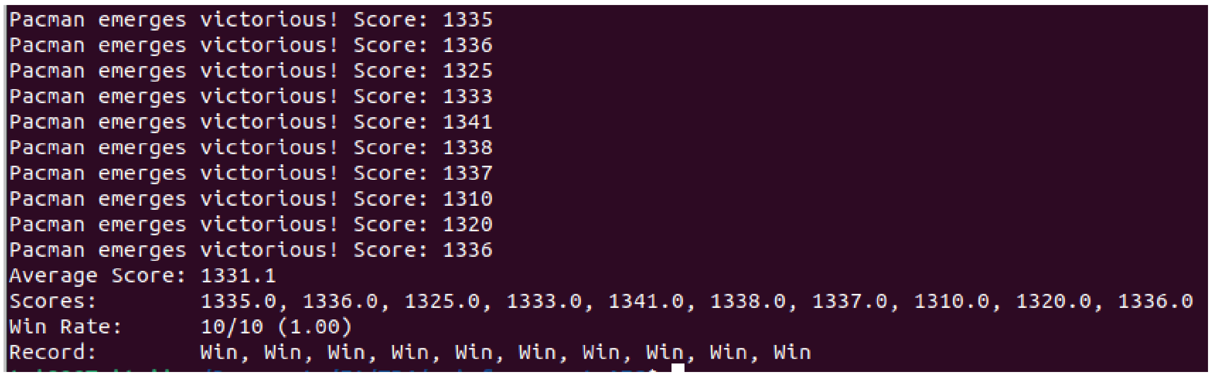


```
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 525
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Pacman emerges victorious! Score: 529
Average Score: 528.6
Scores:        529.0, 525.0, 529.0, 529.0, 529.0, 529.0, 529.0, 529.0, 529.0, 529.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

50 training games and Medium Classic:



```
tai22@Taitaii:~/Documents/IA/TP4/reinforcement_AIC$ python3 pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor
-x 50 -n 60 -l mediumClassic
```



```
Pacman emerges victorious! Score: 1335
Pacman emerges victorious! Score: 1336
Pacman emerges victorious! Score: 1325
Pacman emerges victorious! Score: 1333
Pacman emerges victorious! Score: 1341
Pacman emerges victorious! Score: 1338
Pacman emerges victorious! Score: 1337
Pacman emerges victorious! Score: 1310
Pacman emerges victorious! Score: 1320
Pacman emerges victorious! Score: 1336
Average Score: 1331.1
Scores:        1335.0, 1336.0, 1325.0, 1333.0, 1341.0, 1338.0, 1337.0, 1310.0, 1320.0, 1336.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

After 50 training sessions, Pac Man consistently wins in bigger and complex environments. This success is due from generalization abilities it demonstrates in large state spaces. By using this function to calculate Q-values, the agent significantly cuts down on the memory and computational resources making it adapt at much larger-scale challenges.

Approximate Q-Learning is rapid on adapting and efficiently learning, by generalizing from known to unknown states. This is especially valuable in dynamic real settings where environmental variables are continually changing. Whereas, Expectimax is theoretically oriented and better suited for structured environments where outcomes are more predictable due to probabilistic actions. This method calculates all potential future scenarios to identify the optimal move.

While Expectimax has a higher average score by embracing riskier strategies, it does not match the consistent win rate and the score of Approximate Q-Learning. This difference highlights the approach of the two algorithms: while Expectimax seeks to maximize potential scores by taking calculated risks, Approximate Q-Learning focuses on enhancing the probability of winning.

IV-Conclusion

To conclude, in this lab we have seen a different approach from the previous one. In fact here Pac Man has a better behavior and is more inclined to go to the reward.
To sum up the algorithms seen:

- Value Iteration: is a method used in reinforcement learning to compute an optimal policy by iteratively improving the estimates of state values.
- Q-learning: is an off-policy learner that aims to find the best action to take given the current state. Unlike policy-based strategies, it directly approximates the optimal action-value function independent of the policy being followed.
- Approximate Q-learning: Agent will be able to learn from its environment and adjust its strategy based on the rewards it receives and the features of the state-action pairs it encounters.