Reinforcement Learning Coursework 1

Vitiello Pietro

CID: 01348614

October 2020

Question 1

a.

My CID is 01348614 which, by following the indications proposed in the outline of the coursework, produces the following trace.

$$\tau = s_3\ 1\ s_1\ 3\ s_2\ 0\ s_2\ 0\ s_0\ 2\ s_3\ 1\ s_2\ 0$$

Since the reward is collected when leaving a state, it is accepted to omit considering the last reward, however this does not imply that the last state is a terminal one.

b.

The process under consideration could be treated as a Markov Reward Process, where the future state is only dependent on the current one, not also on the ones that have been visited in the past. Moreover there is only one possible action, hence the dynamics of the system only depend on the current state and the successor one. To draw the simple diagram in Figure 1 I have considered the states visited in my trace and connected each state with the next, reporting the reward of the transition.

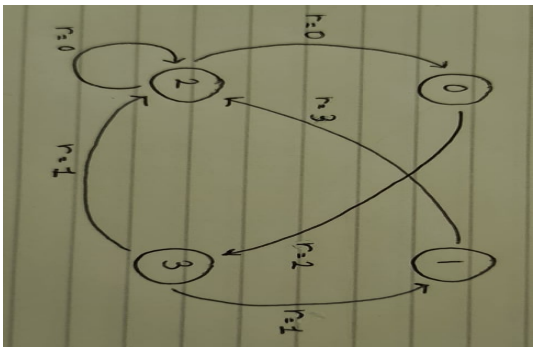


Figure 1: minimal process' graph

c.

By looking at the graph shown in the above Figure we can immediately notice which states can be reached from each state. Instead by looking at the actual trace we can understand how many times a certain transition occurs, which can be used to infer the probability of going to a particular successor state from each state. For instance, if there were three transitions from state 1 and 2 of those were going to state 2, we could infer that when you are in state 1 you have a $\frac{2}{3}$ chance of going to state 2. By following this reasoning we can infer the full transition matrix, which is found below.

$$P_{ss'} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0.5 & 0 & 0.5 & 0 \\ 0 & 0.5 & 0.5 & 0 \end{pmatrix} \tag{1}$$

Other than the transition matrix we can also infer the reward matrix, which is shown below. From the latter it can be noticed that each transition from the same state will give rise to the same reward. This can be explained by the way in which the reward has been calculated, which only depended on the CID and hence on the leaving state, not the successor. Because of this peculiarity of the reward function, which only depends on the state s, we could condense the reward matrix simply to a reward vector, where each element corresponds to the reward collected when departing from the corresponding state. This characteristic of R further proves the fact that this could be considered to be a reward process and not a decision process.

$$R_{ss'} = \begin{pmatrix} 2 & 2 & 2 & 2 \\ 3 & 3 & 3 & 3 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{pmatrix} \rightarrow R_s = (2, 3, 0, 1) \tag{2}$$

Since $\gamma = 1$ it is as if the states that will be visited in the future will have the same influence on the return as the current state. For this reason the return of an infinite trace will be infinite as well, which prevents us from using the Bellman's equation, as this means that the term xxxxx is not invertible. Nonetheless we can use the obtained trace to calculate the sample return for the state $s(t-0)$.
We can calculate the estimated value of state $s(t=0)$ by summing the discounted instantaneous rewards of all the transitions. However, the question tells us that $\gamma = 1$, hence the value of $s(t=0)$ is simply the sum of all the instantaneous rewards, without considering the last one.

$$\begin{aligned} V(s) &= r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + \gamma^4 r_{t+5} + \gamma^5 r_{t+6} \\ V(3) &= 1 + (1)3 + (1)^2 0 + (1)^3 0 + (1)^4 2 + (1)^5 1 = 7 \end{aligned} \tag{3}$$

sample return for each state

Question 2

a. Reward state, p and γ

Following the equations proposed in the coursework outline and using the last three digits of my CID (614), the reward state of my environment is the third, the probability of the desired direction is $p = 0.6$ and my discount factor is $\gamma = 0.25$. Moreover in Figure 2 are shown the maps of the overall environment. The first map on the left shows in yellow the obstacles. The second map shows the two absorbing states in purple, while in the third map it is possible to distinguish between them, as the reward state is the top yellow cell and the bottom one is the penalty state.

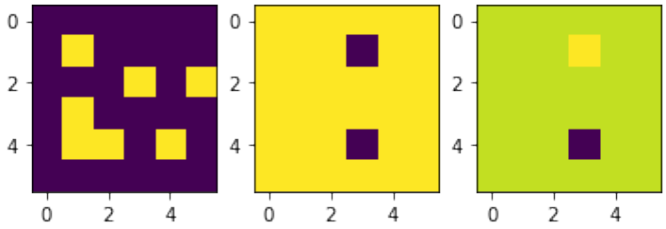


Figure 2: Environment maps

b. Dynamic Programming

Using the Dynamic Programming (DP) algorithm known as Value Iteration (Appendix A) it has been possible to find the optimal value function and optimal policy, which have been reported in Figure 3a and Figure 3b respectively. The value iteration algorithm takes advantage of the fact that following Bellman's optimality equation, any change in policy evaluation will be an improvement. Therefore this program instead of finding the optimal value function for each policy, will improve the policy after each step of policy evaluation, ensuring a faster convergence. Moreover the algorithm has been used with state 12 as initial state and a threshold value of 0.001. The latter has been chosen as it is small enough to allow for a value function accurate up to two decimal points, but large enough to allow for

a fast convergence. The accuracy is more than enough considering that the two absorbing states have very high rewards of 10 and -100 and that there are no two states which have the same value up to the second decimal place.

(a) Optimal value function

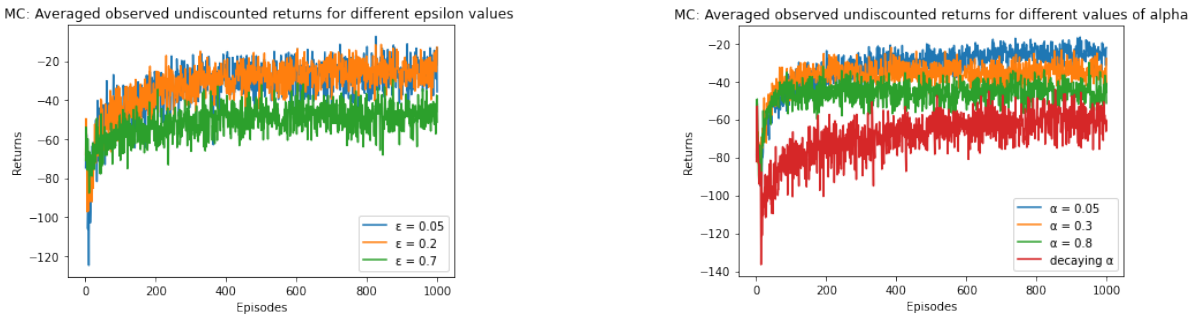
(b) Optimal policy function

Figure 3: Optimal policy and value functions computed using policy iteration (DP)

The values of γ and p have a big impact on the optimisation task. The probability p represents the probability of actually performing the action you have chosen. Being there four possible actions, a p of value 0.25 means that no matter what action has been chosen, you will perform any action with the same probability, therefore the choice of action will have no impact on the outcome. This is reflected in a resulting optimal policy where the chosen action is always North, which is the first action that gets processed. Using this as a point of reference, having a $p < 0.25$ results in the outcome of the chosen action being the least possible. This means that if the objective of the agent is to go right, the best choice is actually not to go right, hence the resulting optimal policy will be mostly wrong. On the other hand a $p > 0.25$ allows the outcome of the chosen action to be the most probable. This ensures that in fact choosing the right action is the best thing to do in order to go in the right direction. For this reason the optimal policy computed will converge to the correct one.

Another parameter that influences the operation of the algorithm is the discount factor γ. A low γ corresponds to a more short-sighted vision of the agent, as the states that will be visited later have less of an impact on the value of the current state. This means that only the closer states to the absorbing ones will feel an influence, while the states which are further from the terminal ones will be less impacted. On the other hand a high value of the discount factor indicates that the states that will be visited later in time have a greater impact on the current state. Another conclusion that could be drawn is that a lower γ will care less about the length of the possible trace compared to a higher γ. Meaning that a higher discount factor will bring the agent to choose to go to states which will bring it closer to the reward state even if they will also be closer to the penalty one. Instead a low discount factor will be more careful around the penalty state and this could lead to the choice of a longer path towards the goal. Moreover a lower discount factor leads to faster convergence, this because the current state will be less affected by the successor ones and for this reason it will be also less affected by changes in the them. Hence the value function will delta will reach the threshold faster, as changes will be smaller sooner. To conclude, 0 and 1 could be viewed as the extremes of the discount factor, hence $\gamma = 0.5$ could be considered the turning point between short and long-sighted agents.

c. Monte Carlo RL



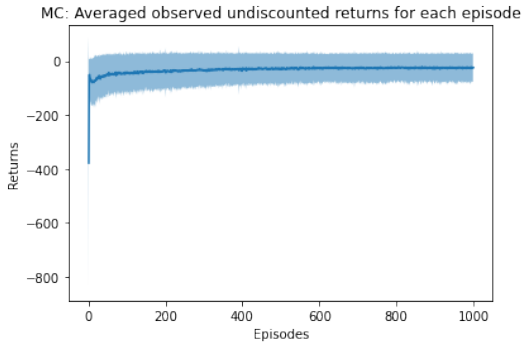
(a) Comparison between different ϵ values

(b) Comparison between different α values

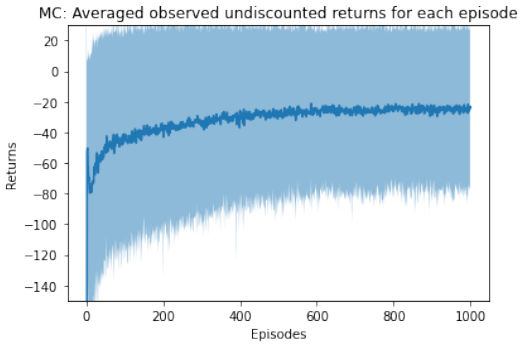
Figure 4: Sum of the rewards of each episode averaged over 200 repetitions of Monte Carlo

Dynamic programming is an optimisation algorithm which has knowledge about the environment. However usually this knowledge is not provided and that is where the learning process comes in. Monte Carlo (MC) is an algorithm which uses sample back-ups to infer the dynamics of the environment. This means that the latter method is model-free and tries to break the curse of dimensionality (since it has a linear cost of back-ups). The algorithm that has been used for this coursework can be found in Appendix B. It is an online every-visit Monte Carlo algorithm which learns from complete episodes, meaning that the program computes several traces from exploring starts and uses empirical mean returns to update the Q value of each state-action pair visited. The incremental Monte Carlo updates use a learning rate α which could be treated as the rate at which the program is forgetting about old visits. Moreover the policy is then improved using an ϵ-greedy condition, where each action can be chosen with a probability of $\frac{\epsilon}{number\ of\ actions}$ and the optimal action can be chosen by an additional probability of $1-\epsilon$, this ensures that the agent continues to explore all options without immediately settling for a policy. This being said these two parameters, α and ϵ, can be varied to improve the functioning of the algorithm, as it has been shown in Figure 4. The ϵ governs the exploration of the agent. The higher its value is the more likely it is to randomly choose an action and hence explore the various states. If you set ϵ to a very low value, you will be greedy from the beginning, instead if you choose an ϵ close to one you will continue to explore even at later episodes, when the agent should concentrate more on the optimal policy. However by looking at Figure 4a it is possible to see that the best returns have been achieve with $\epsilon = 0.05$ and 0.2. Even though the latter value is low, the reason why the algorithm still learns fast could be the fact that the environment is quite noisy ($p = 0.6$), therefore some degree of exploration is achieved anyways by the fact that the agent will not necessarily perform the desired action. Different is instead the α which for low values will lead to a slower convergence but a smaller variance as well, on the other hand a higher value of the learning rate will allow a faster convergence, but also a higher variance since the last visits will have a greater influence than the past ones, therefore if, as it happens, a very good or bad trace gets computed, that will considerably shift the Q value. However, by testing it can be seen that in my case a lower value of alpha produces the best results, as in Figure 4b $\alpha = 0.05$ leads to a higher return.

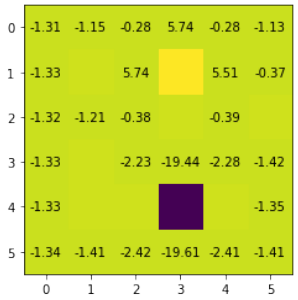
Following the above reasoning, to find the optimal value function and policy I have used $\epsilon = 0.2$ and $\alpha = 0.05$. I have assumed that the process under consideration is an episodic MDP and hence each trace will eventually terminate in an absorbing state. I have also chosen the online Monte Carlo instead of the batch method as the problem in question is simple and with a small state-space. The algorithm has been run 1000 times for 1000 episodes as this has been found empirically as the best number of repetitions in terms of results obtained and time spent training. The mean undiscounted ($\gamma = 0$) return has been plot in blue together with the standard deviation, represented in light blue in Figures 4c and 4d. The resulting optimal value function and policy have been drawn in Figures 4e and 4f.



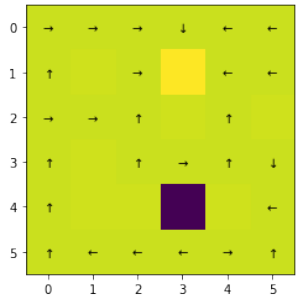
(c) Sum of the rewards of each episode



(d) Zoomed Figure 4c



(e) Optimal value function



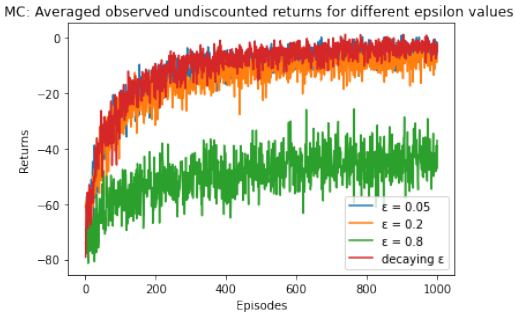
(f) Comparison between different ϵ values

Figure 4: Monte Carlo algorithm run 1000 times for 1000 episodes

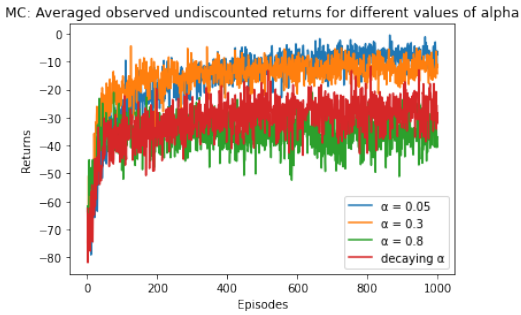
d. Temporal Difference RL

What could be interpreted as the combination of DP and MC is the Temporal Difference (TD) algorithm. The latter could be considered to take the best properties from both as it is model-free like the Monte Carlo, but uses bootstrapping as dynamic programming. In fact TD uses the estimate of the values of the next state-action pair to improve the estimate of the value of the current one. This allows the algorithm to update the Q value after each iteration, without having to wait for the end of the episode.

As it has been done with MC, also for Temporal Difference the algorithm has been tested for several values of ϵ and α, as it is shown in Figure 5. From the latter it can be seen how decaying ϵ yields the best results when looking at the ϵ parameter. This because as the agent trains, and hence accumulates experience, it becomes more confident on the chosen policy and can therefore allow a more greedy policy improvement. On the other hand, a small value of α such as 0.05 still produces the best result as it has happened for MC.



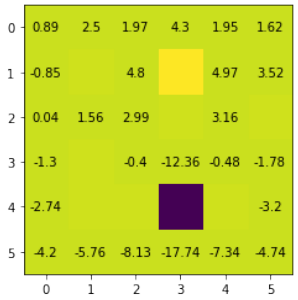
(a) Comparison between different ϵ values



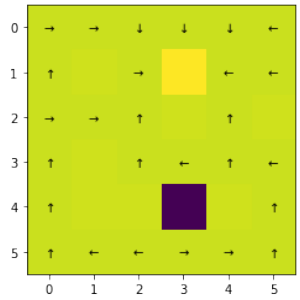
(b) Comparison between different α values

Figure 5: Sum of the rewards of each episode averaged over 100 repetitions of Temporal Difference

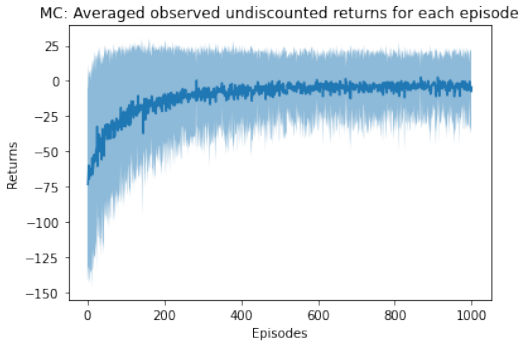
Following the conclusions found above I have used the SARSA algorithm (Appendix D) with decaying ϵ which yielded the results shown in Figure 6. In the latter can be found the sum of the rewards for each episode averaged over 100 repetitions of the algorithm and the resulting optimal value function and policy.



(a) Comparison between different ϵ values



(b) Comparison between different ϵ values



(c) Comparison between different α values

Figure 6: Undiscounted return of each episode averaged over 100 repetitions of Monte Carlo

e. Comparison of learners

There are important differences between TD and MC. Temporal Difference has bias but a lower variance, instead MC has no bias but a higher variance. There is therefore a Bias-Variance trade off. This is reflected in the plot of the RMSE against episodes. The TD algorithm produces an RMSE that decreases faster, but due to the bias there will later be a slight increase in RMSE, especially for higher alpha values, which however will allow faster convergence. On the other hand MC Converges more slowly, but thanks to its unbiased estimate, the error continues to decrease. For MC it could be seen how a higher alpha will yield a much faster convergence but also a higher variance as it has been discussed previously.

Another way of looking at the RMSE is against the reward of the episode. Figure 7 shows the plot I have obtained for MC (Blue) and TD (Orange) with the RMSE as y-axis and episode returns as x-axis. From the plot we are not able to draw many conclusions. However, the ideal plot should be negative correlation between the RMSE and the reward. Hence, there should be a higher RMSE for more negative rewards, as the lower rewards will correspond to longer traces or traces ending in the penalty state, which should have happened at the beginning, when the agent hadn't learnt yet. From the figure it is not possible to see this for my TD algorithm, but it can be seen for MC as the lower returns have more RMSE values that are higher, while higher returns have RMSE values which are more concentrated towards low values.

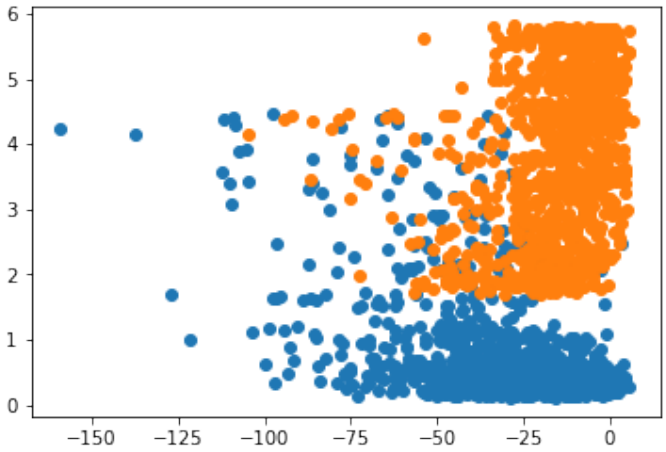


Figure 7: Value function estimation error of the episode against the reward for that episode

Overall I believe that TD yields better results as the latter uses bootstrapping, which could be seen as having more data to work with. Additionally from my results I have seen that TD converges slightly faster, but most importantly after 1000 episodes it converges to a higher reward which is reflected in the optimal policy that has been drawn previously. The latter in fact shows optimal arrows, while the Monte Carlo, even though it produces values which are closer to those from Dynamic Programming, has some of the action choices which would by common sense not be considered the optimal ones.

Appendices

A Value Iteration

```
def value_iteration(self, threshold):
    #Get the transition and reward matrices that define the environment
    T = self.get_transition_matrix()
    R = self.get_reward_matrix()

    #Initialisation of the parameters
    delta = 2*threshold #Setting delta to a value that allows to start the while loop
    V = np.zeros(self.state_size) # Value function initialised at zeros for every state

    while delta >= threshold:
      delta = 0 #Reinitialise delta

      #Go through all the current states
      for state_idx in range(self.state_size):
        #Skip the computation when considering absorbing states
        if not(self.absorbing[0, state_idx]):
          #Store the previous value for that state
          v = V[state_idx]

          #Compute Q value
          Q = np.zeros(4) # Initialise with value 0
          for state_idx_prime in range(self.state_size): #Go through all successor states
            Q += T[state_idx_prime,state_idx,:] * (R[state_idx_prime,state_idx,:] + self.discount
    * V[state_idx_prime])

          #Set the new value to the maximum of Q
          V[state_idx]= np.max(Q)
          #Compute the new delta
          delta = max(delta, np.abs(v - V[state_idx]))

    #Find the corresponding optimal policy
    optimal_policy = np.zeros((self.state_size, self.action_size)) #Initialisation

    #Go through all the states
    for state_idx in range(self.state_size):
      #Compute Q value
      Q = np.zeros(4)
      for state_idx_prime in range(self.state_size):
        Q += T[state_idx_prime,state_idx,:] * (R[state_idx_prime,state_idx,:] + self.discount * V[
    state_idx_prime])

      #The action that maximises the Q value gets probability 1
      optimal_policy[state_idx, np.argmax(Q)] = 1

    return V, optimal_policy
```

B Monte Carlo with fixed ϵ and α

```
def monte_carlo_iterative_learning(self, epsilon, alpha, iterations, discount=None):
    if discount == None:
      discount = self.discount #Use the discount factor calculated from the CID
    Q = np.zeros((self.state_size, self.action_size)) #Initialise the Q function

    #Initialise a policy
    probs = [[0.25, 0.25, 0.25, 0.25]]
    policy = np.repeat(probs, self.state_size, axis=0)

    #Lists used for the plots
    iterations_list = [] #Trace length of each epsiode
    episode_returns = [] #Undiscounted return of each epsiode
    episode_discounted_returns = [] #Discounted return of each episode

    for i in range(iterations):
      #Find a random initial state that isn't terminal
      s = [next((i for i in range(self.state_size) if self.absorbing[0,i] == 1), None),]
      while self.absorbing[0,s[-1]]:
        s = [random.randint(self.state_size),] #Vector s that will contain the states visited in
    the trace
      a = [] #Vector a that will contain the actions performed in the trace
      r = [] #Vector r that will contain the rewards collected in the trace
      R = 0 #Discounted return of the trace
      R_not_disc = 0 #Undiscounted return of the trace

      #Create trace
      while i >= 0:
        a.append(self.event_chooser(policy[s[-1]])) #Find an action from the last state
        next_state = self.event_chooser(self.T[:,s[-1],a[-1]]) #Find where the action will bring
    you, considering the noise

        r.append(self.R[next_state, s[-1], a[-1]]) #Find reward corresponding to successor state,
    state and action
        if self.absorbing[0,next_state]: #If you land on an absorbing state the trace is done
          break
        else : #otherwise continue the trace
          s.append(next_state)

      #Once the trace is complete
      for j in range(len(s)-1, -1, -1): #Go through the trace starting from the last state
    visisted
        R = discount * R + r[j] #Calculate the return of the trace
        R_not_disc += r[j] #Calculate the return without applying the discount (plot purposes)
        Q[s[j], a[j]] = Q[s[j], a[j]] + alpha * (R - Q[s[j], a[j]]) #Update the Q values of the
    state action pair

      #Append the information used for the plots
      iterations_list.append(len(s)+1)
      episode_returns.append(R_not_disc)
      episode_discounted_returns.append(R)

      #Update the epsilon-greedy policy using the new Q values
      policy = epsilon/4 * np.ones((self.state_size, self.action_size))
      for state in range(policy.shape[0]):
        policy[state, np.argmax(Q[state,:])] += 1-epsilon

    return policy, Q, iterations_list, episode_discounted_returns, episode_returns
```

C Monte Carlo with fixed ϵ and decaying α

```
def MC_decaying_alpha(self, epsilon, starting_alpha, alpha_rate, iterations, noisy=True, discount=
    None):
    if discount == None:
      discount = self.discount
    Q = np.zeros((self.state_size, self.action_size))
    probs = [[0.25, 0.25, 0.25, 0.25]]
    policy = np.repeat(probs, self.state_size, axis=0)

    iterations_list = []
    episode_returns = []
    episode_discounted_returns = []
    visits = np.zeros(self.state_size)
    alpha = np.ones(self.state_size) * starting_alpha

    for i in range(iterations):
      s = [next((i for i in range(self.state_size) if self.absorbing[0,i] == 1), None),]
      while self.absorbing[0,s[-1]]:
        s = [random.randint(self.state_size),]
      a = []
      r = []
      R = 0
      R_not_disc = 0
      #create trace
      while i >= 0:
        a.append(self.event_chooser(policy[s[-1]]))
        if noisy:
          next_state = self.event_chooser(self.T[:,s[-1],a[-1]])
        else:
          next_state = int(self.neighbours[s[-1],a[-1]])
        r.append(self.R[next_state, s[-1], a[-1]])
        if self.absorbing[0,next_state]:
          break
        else :
          s.append(next_state)

      for j in range(len(s)-1, -1, -1):
        R = discount * R + r[j]
        R_not_disc += r[j]
        Q[s[j], a[j]] = Q[s[j], a[j]] + alpha[s[j]] * (R - Q[s[j], a[j]])
        visits[s[j]] += 1
        if visits[s[j]] % alpha_rate == 0:
          alpha[s[j]] = starting_alpha/(visits[s[j]]/alpha_rate+1)

      iterations_list.append(len(s)+1)
      episode_returns.append(R_not_disc)
      episode_discounted_returns.append(R)

      policy = epsilon/4 * np.ones((self.state_size, self.action_size))
      for state in range(policy.shape[0]):
        policy[state, np.argmax(Q[state,:])] += 1-epsilon

    return policy, Q, iterations_list, episode_discounted_returns, episode_returns
```

D SARSA with decaying ϵ

```
def sarsa_decay(self, starting_epsilon, starting_alpha, iterations, epsilon_decay=True,
    alpha_decay=True, noisy=True, discount=None):
    if discount == None:
      discount = self.discount
    Q = np.zeros((self.state_size, self.action_size))
    probs = [[0.25, 0.25, 0.25, 0.25]]
    policy = np.repeat(probs, self.state_size, axis=0)

    visits = np.zeros(self.state_size)
    alpha = starting_alpha * np.ones(self.state_size)

    iterations_list = []
    episode_returns = []
    for i in range(iterations):
      epsilon = starting_epsilon/(i+1) if epsilon_decay else starting_epsilon

      #create trace
      state = next((i for i in range(self.state_size) if self.absorbing[0,i] == 1), None)
      while self.absorbing[0,state]:
        state = random.randint(self.state_size)
      trace_length = 1
      action = self.event_chooser(policy[state])

      episode_return = 0
      iteration = 0
      while i >= 0:
        if noisy:
          next_state = self.event_chooser(self.T[:,state,action])
        else:
          next_state = int(self.neighbours[state,action])
        next_action = self.event_chooser(policy[next_state])
        R = self.R[next_state, state, action]

        Q[state, action] = Q[state, action] + alpha[state] * (R + Q[next_state, next_action] - Q[
    state, action])
        visits[state] += 1 if alpha_decay else 0
        alpha[state] = starting_alpha/(visits[state]+1)

        policy[state, :] = epsilon/4
        policy[state, np.argmax(Q[state,:])] += 1-epsilon
        trace_length += 1

        episode_return += R
        iteration += 1

        if self.absorbing[0,next_state]:
          break
        else :
          state = next_state
          action = next_action

      iterations_list.append(iteration+1)
      episode_returns.append(episode_return)

    return policy, Q, iterations_list, episode_returns
```