TP3

Mariia Drozdova

MAP641 - Reinforcement learning
AI-VIS

March 10, 2020

Contents

1 TP Part A Average and Discounted optimality

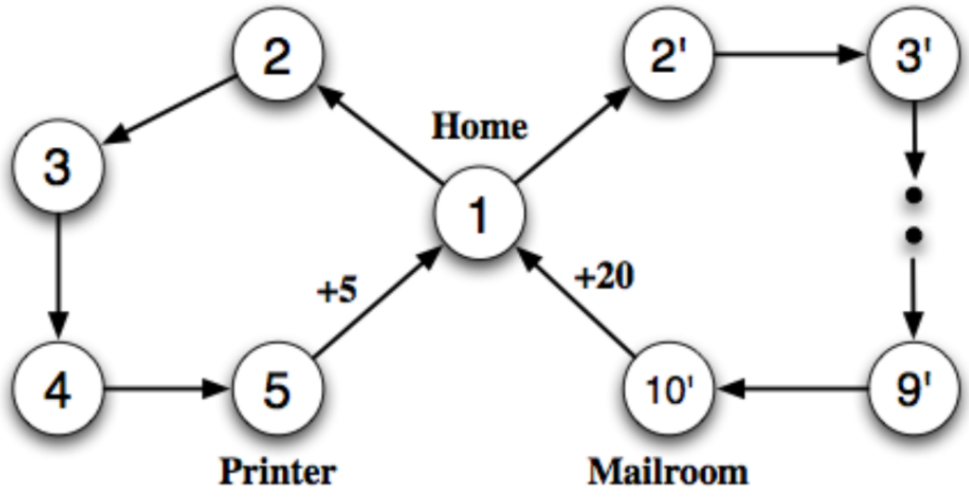


Figure 1: MDP

1.1 What are the stationary deterministic policies for this MDP?

A deterministic stationary policy deterministically selects actions based on the current state. $a_{q\to r}$ stands for an action "going from a state s_q to a state s_r" There are two policies depending on the action taken in that state 1:

$\pi_1(s_1) = a_{1\to 2}$ $(\pi_1(s_2) = a_{2\to 3},\ \pi_1(s_3) = a_{3\to 4},\ \pi_1(s_4) = a_{4\to 5},\ \pi_1(s_5) = a_{5\to 1})$

$\pi_2(s_1) = a_{1\to 2'}$ $(\pi_2(s_{2'}) = a_{2'\to 3'},\ \pi_2(s_{3'}) = a_{3'\to 4'}\ ...\ \pi_2(s_{10'}) = a_{10'\to 1})$

1.2 For each of them, what is the γ-discounted value at state 1 ?

The discount value of state 1:

$R_{\pi_1} = r_{1\to 2} + \gamma r_{2\to 3} + \gamma^2 r_{3\to 4} + \gamma^3 r_{4\to 5} + \gamma^4 r_{5\to 1} + .. = \gamma^4 r_{5\to 1} + \gamma^9 r_{5\to 1} + ... = \frac{5\gamma^4}{1-\gamma^5}$

$R_{\pi_2} = r_{1\to 2'} + \gamma r_{2'\to 3'} + \gamma^2 r_{3'\to 4'} + ... + \gamma^9 r_{10'\to 1} + ... = \gamma^9 r_{10'\to 1} + ... = \frac{20\gamma^9}{1-\gamma^{10}}$

1.3 For which values of $\gamma \in [0,1)$ is it optimal to move to the mail room?

R_{π_2} should be greater than R_{π_1} :

$R_{\pi_2} > R_{\pi_1} \iff \frac{20\gamma^9}{1-\gamma^{10}} > \frac{5\gamma^4}{1-\gamma^5} \iff \frac{4\gamma^5}{1-\gamma^{10}} > \frac{1}{1-\gamma^5} \iff 4\gamma^5 > 1+\gamma^5 \iff 3\gamma^5 > 1 \iff \gamma > (\frac{1}{3})^{\frac{1}{5}}$

For $\gamma > (\frac{1}{3})^{\frac{1}{5}}$ it is optimal to go to the mail room.

1.4 For which values of $\gamma \in [0, 1)$ is it optimal to move to the printer?

R_{π_2} should be smaller than R_{π_1}, thus, for $\gamma < (\frac{1}{3})^{\frac{1}{5}}$ it is optimal to go to the mail room.

1.5 A strategy is Blackwell optimal if there exists γ_0 such that is optimal for all $\gamma \in [\gamma_0, 1)$. does this problem have any Blackwell optimal strategy?

Yes. There can be multiple optimal policies. This is exactly what happens if $\gamma = (\frac{1}{3})^{\frac{1}{5}}$: we have two optimal policies π_1 and π_2. When $\gamma > (\frac{1}{3})^{\frac{1}{5}}$ π_2 is optimal. Thus, π_2 is Blackwell optimal for $\gamma_0 = (\frac{1}{3})^{\frac{1}{5}}$

1.6 For each policy, what is the average value in state 1 ? Which policy is optimal for this criterion ?

The average values are :

$\overline{R_{\pi_1}(s_1)} = \frac{5}{5} = 1$

$\overline{R_{\pi_2}(s_1)} = \frac{20}{10} = 2$

π_2 is optimal in this case.

1.7 For what range of values of does the agent select a policy optimal for the average value?

For $\gamma_0 \geqslant (\frac{1}{3})^{\frac{1}{5}}$.

2 TP Part B Value Iteration, turnpikes

2.1 Implement in package learner a generic method VI to compute an optimal value and policy (you may also implement PI, etc.) using the knowledge of P , R. You may use a discount factor γ. Make sure to store all intermediate policies π_k and values v_k computed at each stage of the algorithm.

We remember values and policies in the member variables of the class learner(which is VI in our case).

```
     def update(self, state, action, reward, observation):
        self.VI(self.epsilon, self.max_iter)

    def VI(self, epsilon=0.01, max_iter=1000):
        u0 = self.u - min(self.u)  # np.zeros(self.nS)
        u1 = np.zeros(self.nS)
        itera = 0
        while True:
            self.t = itera
            sorted_indices = np.argsort(u0)  # sorted in ascending orders
            for s in range(self.nS):
                temp = np.zeros(self.nA)
                print(u0)
                for a in range(self.nA):
                    temp[a] = self.meanrewards[s, a] + self.gamma * sum([u0[ns
    ] * self.transitions[s, a, ns] for ns in range(self.nS)])
                (u1[s], choice) = allmax(temp)
                new_policy= [ 1./len(choice) if x in choice else 0 for x in
    range(self.nA) ]
                self.policy[s]= new_policy
            diff = [abs(x - y) for (x, y) in zip(u1, u0)]
            self.policies.append(self.policy.copy())
            self.values.append(u1.copy())
            if (max(diff) - min(diff)) < epsilon:
                self.u = u1-min(u1)
                break
            elif itera > max_iter:
                self.u = u1-min(u1)
                print("No convergence in VI at time ", self.t, " before ",
    max_iter, " iterations.")
                break
            else:
                u0 = u1- min(u1)
                u1 = np.zeros(self.nS)
                itera += 1
```

Listing 1: Value Iteration algortithm

2.2 When choosing the greedy policy, since the Argmax may not be unique, one needs either a tie-breaking rule or a stochastic policy. Implement a version sample that chooses one action in the Argmax uniformly randomly to specify the policy and another stochastic that outputs a stochastic policy with uniform weights on all actions in the Argmax.

```
                if self.option == 'sample':
                    new_policy= [ 1./len(choice) if x in choice else 0 for x
    in range(self.nA) ]
                if self.option == 'stochastic':
                    new_policy = [0 for x in range(self.nA)]
                    new_policy[random.choice(choice)] = 1
                self.policy[s]= new_policy
```

Listing 2: Adding sample and stochastic choice for VI.

2.3 Choose $\gamma = 1$. Does it always converge? If not, you may simply stop after some large number of iterations for now.

It converged for big enough numbers of iterations for grid games (river-swim, 2-room, 4-room, random grid) but it sometimes diverged for random MDPs with $\gamma = 1$. It might be connected with a way how the transition matrix is determined in a "goal" state. In random MDP a divergence case occurred when there was a state with a gain that could be cycled with an increasing diverging value. (transition probability was one)

2.4 Display for each step and for $\gamma < 1$ the value v_k and policy π_k for both versions sample and stochastic. What do you observe? Do the same with $\gamma = 1$.

Figures 3 - 11. We observe that values become smaller(colors less bright for our visualization) while we are going further from the goal state in case when $\gamma < 1$ comparing to $\gamma = 1$. It is due to the γ factor which is used for computing the future rewards.

The policy values are the same. They depend on the sign of value difference in the neighbouring cells which remain unchanged in this case. ($\gamma = 0.85$)

In case of sample choice if there are two optimal policies, they are both displayed, while in a sample case we should randomly choose one.

Examples of visualization:

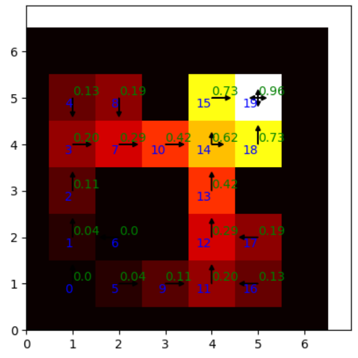


Figure 2: Values are displayed with a color heat map(higher values correspond to brighter colors). Policies are displayed via arrows which show the next movements. Here $\gamma = 0.85$ and the action choice is in a sample mode.

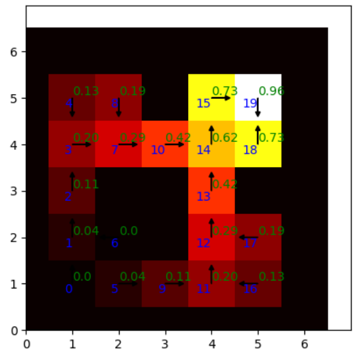


Figure 3: Green number in the right top corner of the cell stands for a corresponding value, blue number - for the corresponding state number of the cell. Here $\gamma = 0.85$ and the action choice is in a stochastic mode.

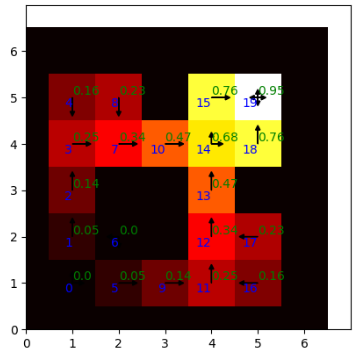


Figure 4: Here $\gamma = 1.0$ and the action choice is in a sample mode.

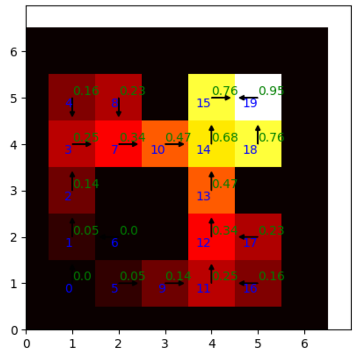


Figure 5: Here $\gamma = 1.0$ and the action choice is in a stochastic mode.

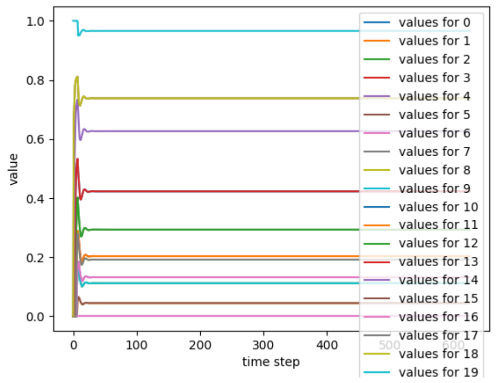


Figure 6: Values. Convergence for $\gamma = 0.95$ for 4room.

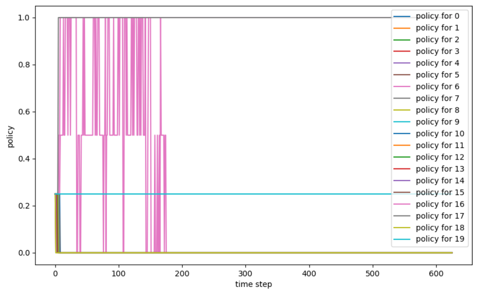


Figure 7: Policies. Convergence for $\gamma = 0.95$ for 4room.

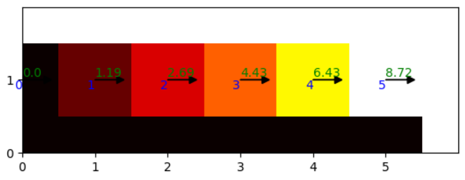


Figure 8: Riverswim6

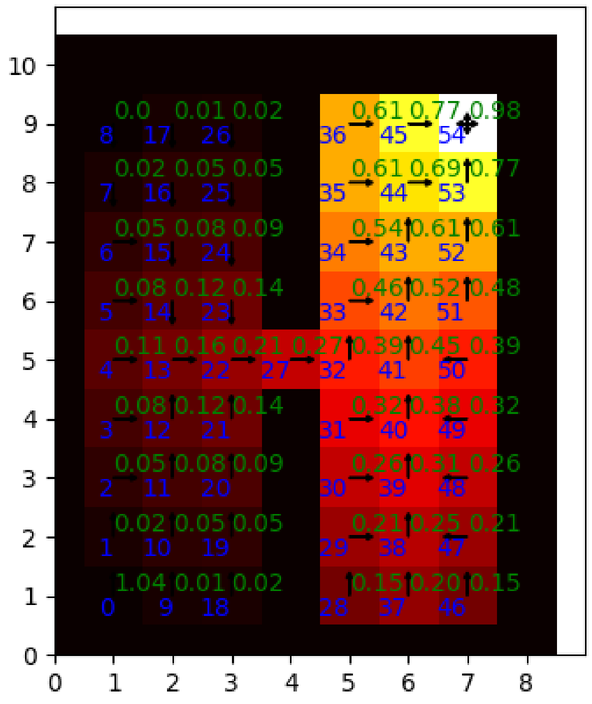


Figure 9: 2-room

2.5 A turnpike N is the minimal N such that $\exists\pi\forall k > N\pi_k = \pi$. The policy is then an optimal policy. For both versions sample and stochastic, and when value iteration converge, find a N such that $\forall k > N\pi_k = \pi$. What do you observe? When Value iteration does not converge, try to identify inside (π_k) a sequence of at least L consecutive steps with same policy, for several vales of L.

To accomplish this task we write the following function:

```python
def find_first_index_of_optimal_policy(p):
    a = np.sum(np.sum(np.abs(p - p[-1]), axis=1), axis=1)
    a = (a == 0)
    s = 0
    ss = []
    indexes = []
    for i in range(len(a)):
        if a[i]:
            s += 1
        else:
            s = 0
            indexes.append(i)
    return indexes[-1]
```

Listing 3: Value Iteration algortithm

Here are the results we got for the given environments ($\epsilon = 0.001$, $\gamma = 0.95$):

	sample	stochastic
RiverSwim6	5	5
RiverSwim25	55	55
2room	281	643
4room	174	624
random10	0	718
3state	0	602

In the stochastic case sometimes we do not have this k until value start converging and the update is not called. It is due to the fact that several policies give exactly the same value and stochastic choice have a randomness inside. Thus we have several policies among optimal ones. Once the value change become negligible with respect to chosen a random optimal policy is chosen.

Another effect that we noticed is due to the precision issue. With values of γ close to one and the case when the path to the victory is long, the algorithm may have such close values (for example, in RiverSwim26) that some not stable regions occur. For us it was second state which sometime would recommend to move right because values are too close on the right and left. In case of sample policy it will simply return both possible movement directions with uniform probability which guarantee to have optimal policy in that sense.

In the figures 14 and 15 we observe the case when values diverge though policies stay the same. It is with an sample choice. In case of stochastic choice we have randomness again, so the policies are changing.

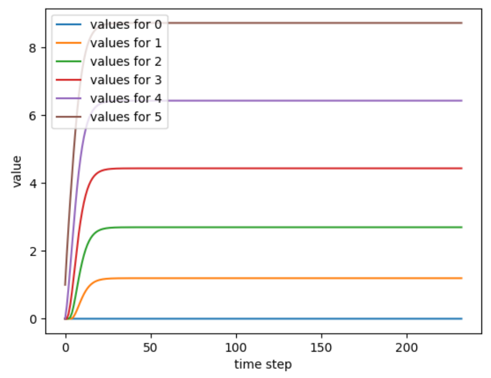


Figure 10: Riverswim6 values

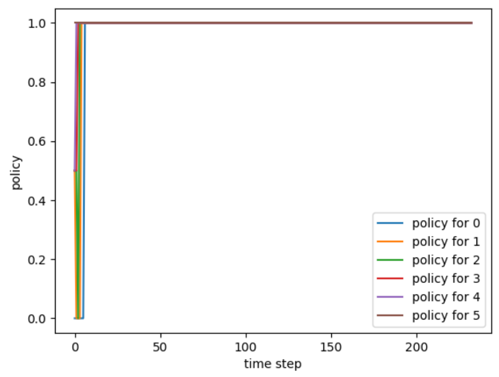


Figure 11: Riverswim6 policies (sample choice for all)

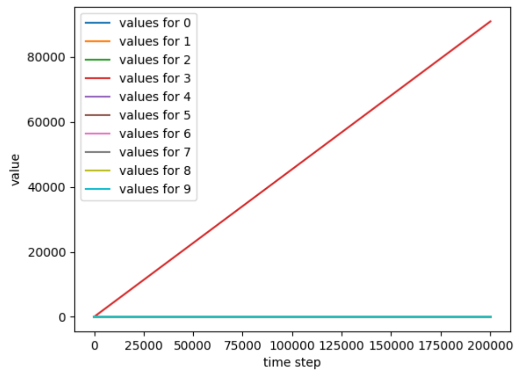


Figure 12: Values. We can see that they diverge. Here $\gamma = 1.0$ for random10.

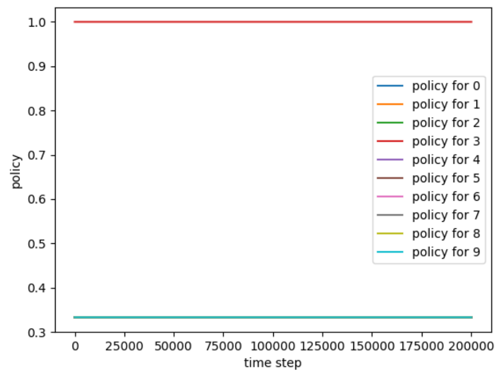


Figure 13: Policies. Despite divergence there is an optimal policy by the given definition. Here $\gamma = 1.0$ for random10.

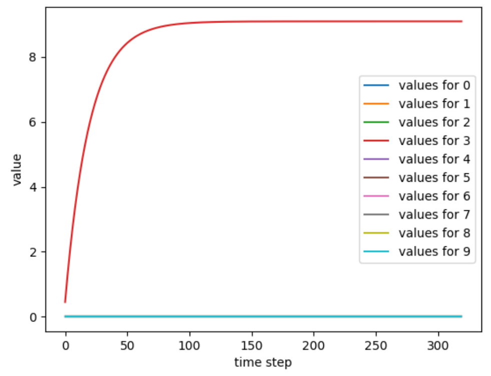


Figure 14: Values. Convergence for $\gamma = 0.95$ for random10.

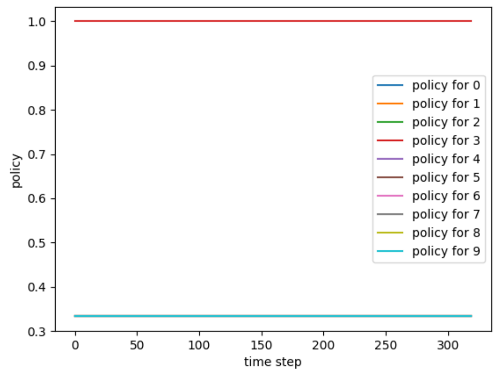


Figure 15: Policies. Convergence with the same optimal policy by the given definition. Here $\gamma = 0.95$ for random10.

2.6 Frequency of visit: For a policy π, with average transition P_π, we let ρ_π be the stationary distribution of P_π, and define the mean reward $\overline{\mu_\pi} = \mathbb{E}_{X\sim\rho_\pi}[\mu_\pi(X)]$. Compute and display $\overline{P_\pi}$, ρ_π, $\overline{\mu_\pi}$ (or an approximation) for being an optimal policy. What do you observe?

To make this experiment we create the following function:

```

def find_stat_distribution(env, learner, timeHorizon=600,
    nb_exp_after_learning=1000, rendermode='text'):
    observation = env.reset()
    learner.reset(observation)
    cumreward = 0.
    P_optimal = np.zeros((learner.nS, learner.nS))
    cumrewards = []
    for t in range(timeHorizon):
        state = observation
        env.render(rendermode)
        action = learner.play(state)  # Get action
        observation, reward, done, info = env.step(action)
        learner.update(state, action, reward, observation)  # Update learners
        cumreward += reward
        cumrewards.append(cumreward)
        if done:
            print("Episode finished after {} timesteps".format(t + 1))
            break
    for i in range(nb_exp_after_learning):
        observation = random.choice(range(learner.nS))
        cumreward = 0.
        cumrewards = []
        for t in range(50):
            state = observation
            env.render(rendermode)
            action = learner.play(state)  # Get action
            observation, reward, done, info = env.step(action)
            cumreward += reward
            cumrewards.append(cumreward)
            P_optimal[state, observation] += 1
            if done:
                break
    a = np.sum(P_optimal,axis=1)
    P = (P_optimal.T/(a+(a==0))).T
    P_new = (P.T - np.diag(np.ones(P.shape[0])))
    A = (np.vstack([P_new, (np.ones(P_new.shape[0]))]))
    b = np.zeros((A.shape[0], 1)), b[-1] = 1
    solution = (np.linalg.solve(A.T @ A, A.T @ b))
    mean_rewards = (np.mean(learner.meanrewards, axis=1))
    mean_p_rewards = (mean_rewards.reshape(-1)*solution.reshape(-1)).reshape
    (-1,1)

    N=10000
    P_k = P.copy()
    P_bar = P_k.copy()
    for k in range(1, N):
        P_k = np.matmul(P_k, P)
        P_bar += P_k
    P_bar = P_bar/N
    return P_bar, solution, mean_p_rewards, P
```

Listing 4: Search for stationary distribution.

First, we run one experiment of 600 iteration to find an optimal policy. Then, we run 1000 experiments to construct a matrix of average transition when the learner is not updated anymore. It uses a learnt policy. Then we construct a linear system that we solve with numpy. The last line in the linear system is the condition that the sum of all components of the solution should be one.

$$\pi = P\pi$$

To visualize the result we create the following function:

```
def display_P_rho_mu(P, rho, mu):
    space = np.zeros((P.shape[0], 2))
    all_P = np.hstack([P, space, rho, space, mu])
    def check(i, j):
        P_copy = np.ones((P.shape))
        rho_copy = np.ones((rho.shape))
        mu_copy = np.ones((mu.shape))
        all_P_copy = np.hstack([P_copy, space, rho_copy, space, mu_copy])
        return all_P_copy[j, i]
    plt.figure(figsize=(20,20))
    for i in range(all_P.shape[1]):
        for j in range(all_P.shape[0]):
            if check(i, j):
                    if all_P[j, i]<0.005:
                        letter = '0.0'
                    else:
                        letter = str(all_P[j, i])[:4]
                    if P.shape[0] <= 36:
                        text = plt.text(i, j, letter,
                            ha="center", va="center", color="g")
    all_P = np.hstack([P/np.max(P), space, rho/np.max(rho), space, mu/np.max(
   mu)])
    plt.imshow(all_P, cmap='plasma')
    plt.show()
```

Listing 5: Visualization for stationary distribution and mean rewards.

On the visualizations (Figures 18 - 23) we notice that the most visited are the states close to the goal one (as the policy got optimal) we always visit them. "Diagonals" are very visible for River Swim game during "test" runs (when the learners are already learnt). This is easy explainable as this close to diagonal elements are the path to get to the goal state and other elements do not have an edee between them.

We can notice the same effect for other games too. Most visited states visualize the transition matrix to achieve the goal state. The stationary distribution shows the most visited states on the way following the optimal policy. For 2 room problem, for example, the state 27 (the door between rooms) have a very big coefficient in the stationary distribution. All the ways from the states from the first room go though it. The states near state 27 has also big coefficients as many ways pass through them too.

An interesting observation is that rows from averaged P matrix are exactly the same as the stationary distribution. There is a theorem that says that in certain conditions there exists a limited distribution that is the same a stationary one. We can notice that in our experiments except for a 2-room game - there are gaps.

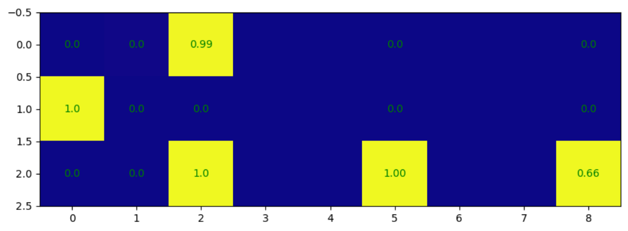


Figure 16: P_π matrix, a stationary distribution and mean rewards for the three state problem. (a stationary distribution is vertical that is at 5 on the x axis, mean rewards is the vertical that is at 8 on the x axis.)

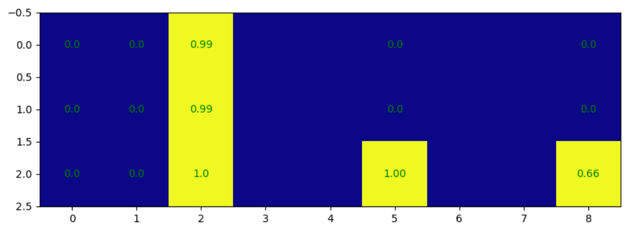


Figure 17: $\overline{P_\pi}$ matrix, a stationary distribution and mean rewards for the three state problem.

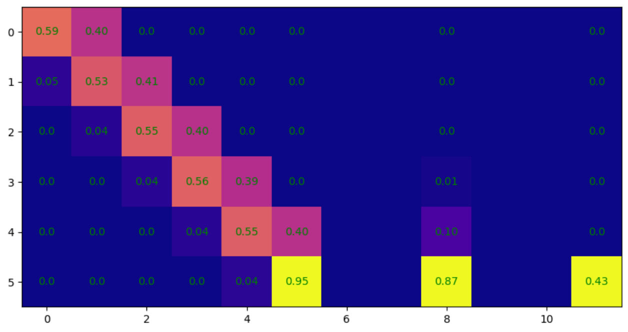


Figure 18: P_π matrix, a stationary distribution and mean rewards for the RiverSwim6. Last row is mean rewards, the row before the last one is a stationary distribution.

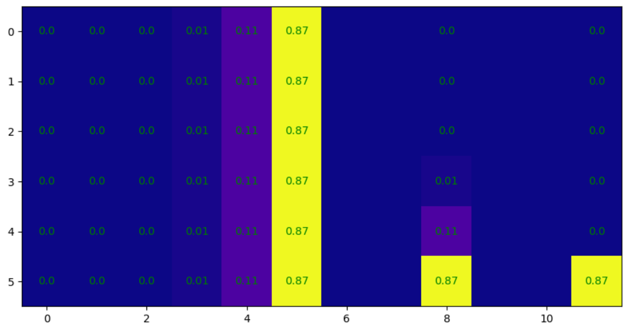


Figure 19: $\overline{P_\pi}$ matrix, a stationary distribution and mean rewards for the RiverSwim6.

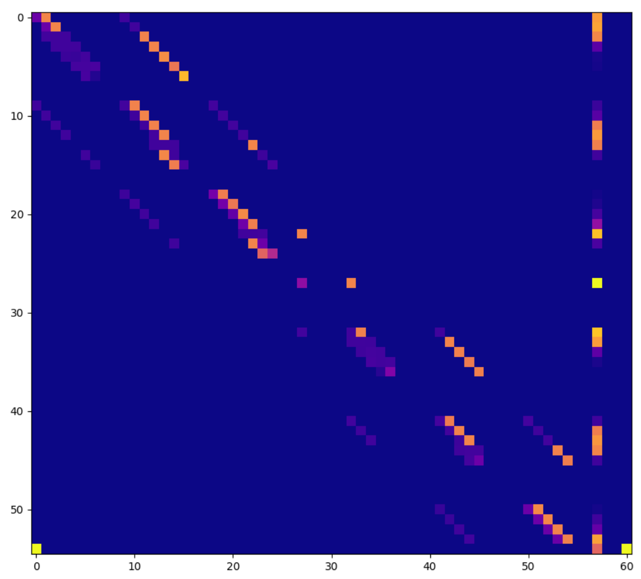


Figure 20: P_π matrix, a stationary distribution and mean rewards for the 2-rooms. Last row is mean rewards, the row before the last one is a stationary distribution.

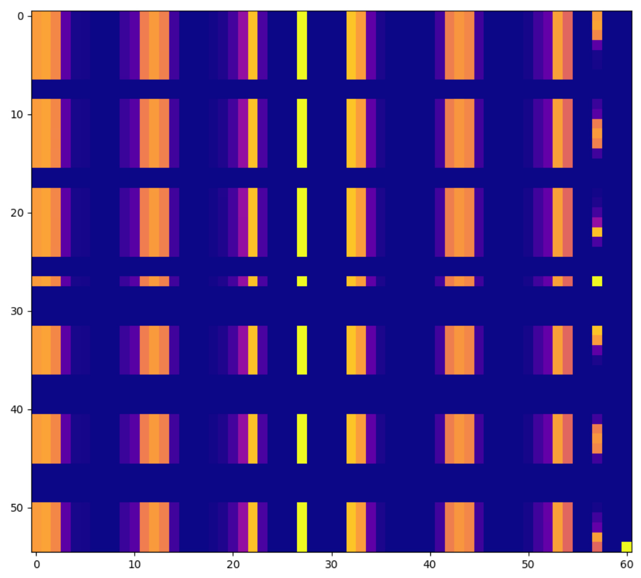


Figure 21: $\overline{P_\pi}$ matrix, a stationary distribution and mean rewards for the 2-rooms.

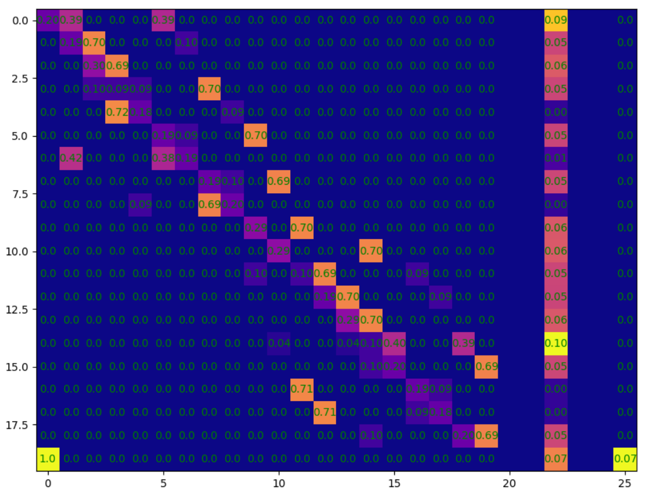


Figure 22: P_π matrix, a stationary distribution and mean rewards for the 4-rooms. Last row is mean rewards, the row before the last one is a stationary distribution.

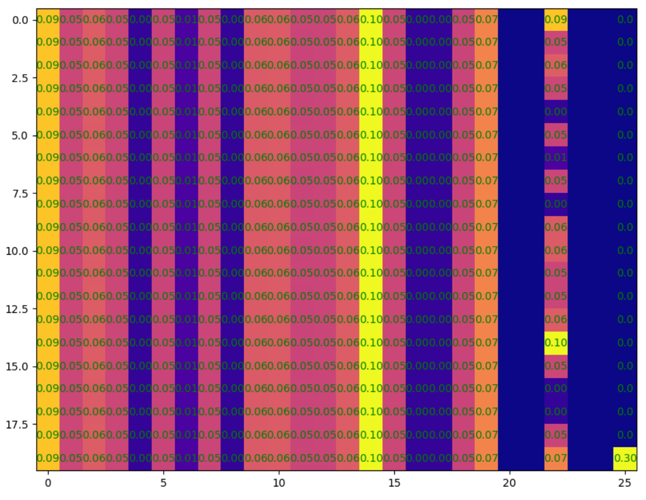


Figure 23: $\overline{P_\pi}$ matrix, a stationary distribution and mean rewards for the 4-rooms.

3 TP Part C Contraction, Reachability, Hitting times

3.1 Contraction

3.1.1 Contraction coefficients for a policy π

For this we create a following function :

```python
def find_contractions(learner ,P, K=1):
    nS = learner.nS
    min_P_sum = np.inf
    if K>1:
        P_k = P.copy()
        sum_P = P_k.copy()
        for k in range(1, K):
            P_k = np.matmul(P_k, P)
    else:
        P_k = P
    for s1 in range(nS):
        for s2 in range(nS):
            sum_P_12 = 0
            for s_prim in range(nS):
                P_1 = P_k[s1, s_prim]
                P_2 = P_k[s2, s_prim]
                P_12 = min(P_1, P_2)
                sum_P_12 += P_12
            min_P_sum = min(min_P_sum, sum_P_12)
    return (1 - min_P_sum)
```

Listing 6: Contraction coefficients for policy π with a transition matrix P.

3.1.2 For both the optimal and a random policy, plot $\gamma_{\pi,k}$ as a function of k. What is the smallest k such $\gamma_k < 1$?

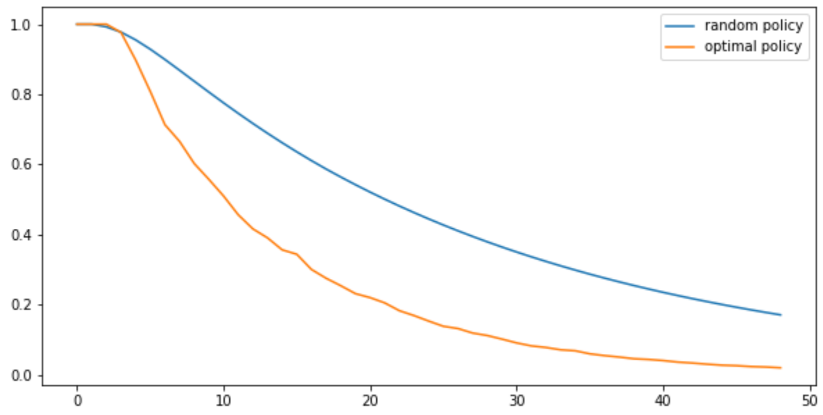


Figure 24: Gamma coefficients for a random and VI policy.

The figure 25 is a plot for 4-rooms problem. Gamma for an optimal policy starts being lower than one with k = 2. For a random policy k = 3.

3.1.3 Compute and plot contraction coefficients of VI seen in class, at each step n of value iteration. What do you observe?

The coefficient is:

$$\overline{\gamma} = 1 - \sum_{s' \in S} \min\left(P(s'|\overline{s}, \pi_{k+2}(\overline{s})), P(s'|\underline{s}, \pi_{k+1}(\underline{s}))\right)$$

$$\overline{s} = argmax_{s \in S}(P_{\pi_{k+2}\Delta_k})(s), \underline{s} = argmin_{s \in S}(P_{\pi_{k+1}\Delta_k})(s)$$

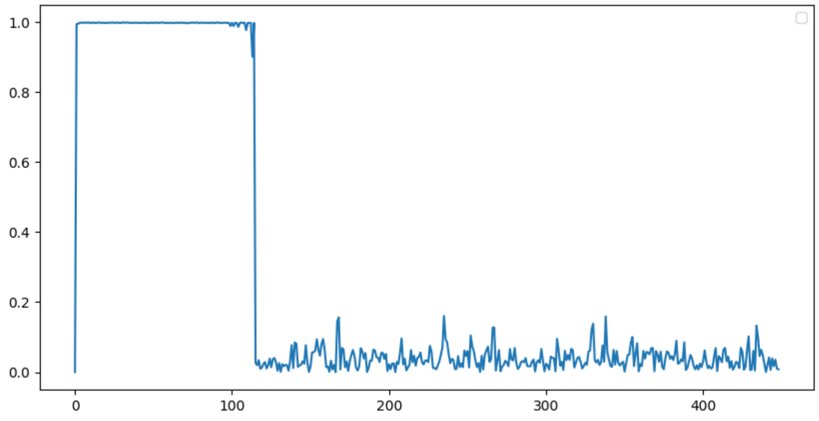


Figure 25: Gamma coefficient for each step of VI.

We observe a significant drop that does not regain at about 107th iteration.

3.2 Hitting, recovering times

3.2.1 Choose a random policy. Starting from some reference state s_0 compute the hitting probabilities $p_{<=k}(\cdot|s_0)$ and the hitting time $\tau(\cdot, s_0; \alpha)$ at level $\alpha = 0.99$ for each state s, for several values of k. (You may display this as a an animation to help visualization).

We make the following functions to answer the questions:

```python
def compute_hitting_proba(learner, s0, P, K):
    P_cur = P.copy()
    nS = learner.nS
    list_P0 = [P.copy() for s in range(nS)]
    list_P = [P.copy() for s in range(nS)]
    #list_P[s0][s0]=0
    for s in range(nS):
        list_P0[s][s] = 0
    #([print(p) for p in list_P])
    probas = [0]*nS
    for k in range(K):
        for s in range(nS):
            probas[s] +=list_P[s][s0, s]
            list_P[s] = np.matmul(list_P[s], list_P0[s])
    #([print(p) for p in list_P])
    return probas

def compute_tau(learner, s0, P, alpha=0.99):
    nS = learner.nS
    cur_proba = [0.0]*nS#learner.nS
    times = [-1]*nS
    K = 0
    while (np.sum(np.array(cur_proba[:s0]+cur_proba[s0+1:])>=alpha) != nS-1 ):
        #print(K)
        cur_proba = compute_hitting_proba(learner, s0, P, K)
        if K % 100 == 0:
            print(cur_proba)
        for s in range(nS):
            if times[s] < 0 and cur_proba[s]>=alpha:
                times[s] = K

        K = K + 1
    return times
```

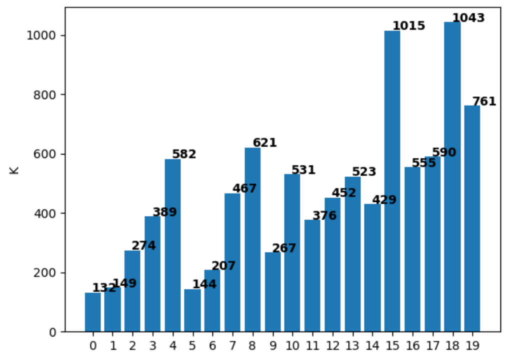
Listing 7: Code for computing $p_{<=k}(\cdot|s_0)$ and $\tau(\cdot, s_0; \alpha)$

3.2.2 What is the smallest k such that all states are α-reachable from s_0 within k steps ?

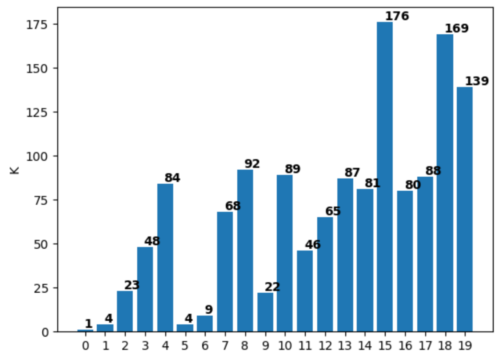
In the Figure 26 we display the k-s such that the corresponding states(x-axis) are α-reachable for 4-rooms. For $\alpha = 0.99$ from the graphics we see that the minimal k such that all states are α-reachable is 1043. For $\alpha = 0.5$ minimal k is equal to 176.

In the same Figure 27 we notice that for $\alpha = 0.99$ minimal k is equal to 2701. For $\alpha = 0.5$ minimal k is equal to 402.

All these results are for random policies. They show that the furthest states from the initial state has the largest K in MDPs.

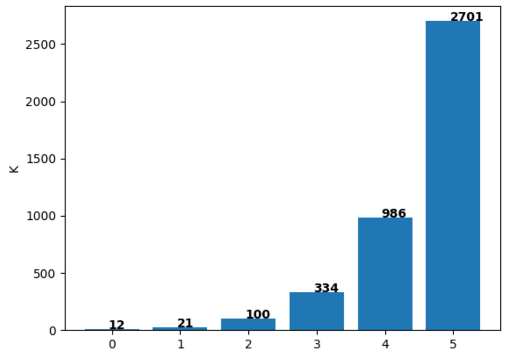


(a) $\alpha = 0.99$

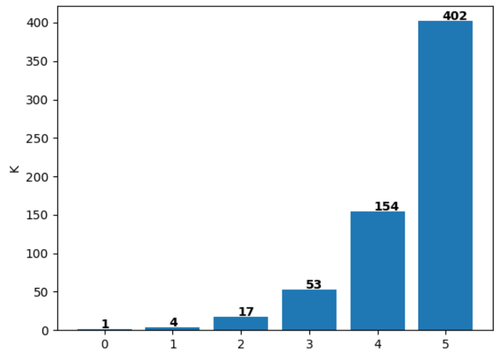


(b) $\alpha = 0.5$

Figure 26: 4-rooms. Minimum K so that the probability to reach each state is at least α. For the first state it is a returning K. (random policy) X-axis - states, y-axis - K.



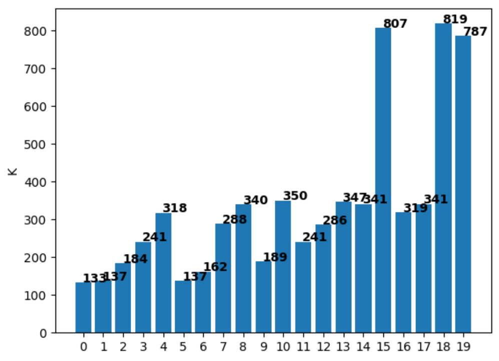
(a) $\alpha = 0.99$



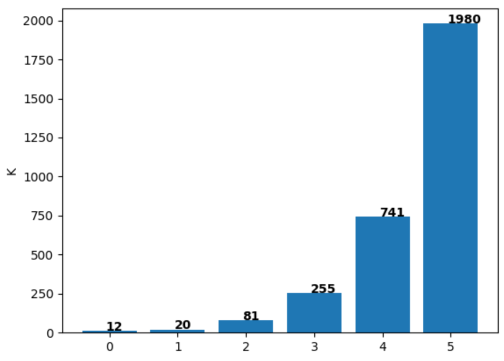
(b) $\alpha = 0.5$

Figure 27: RiverSwim6. Minimum K so that the probability to reach each state is at least α. For the first state it is a returning K. (random policy) X-axis - states, y-axis - K.

3.2.3 Compute and display the returning time $\tau(\cdot, s_0; \alpha)$ for each s. What is the largest returning time in the MDP?



(a) 4rooms

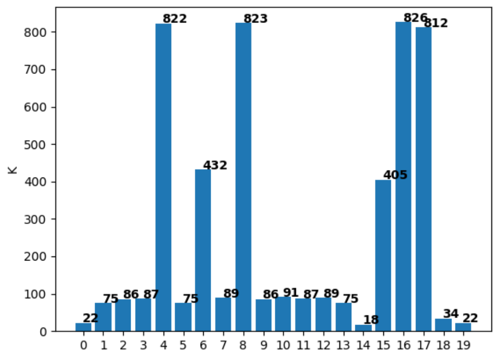


(b) riverSwim

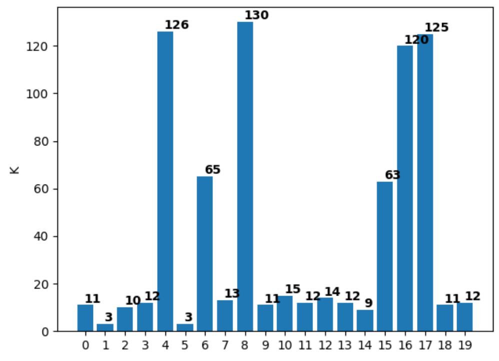
Figure 28: $\tau(\cdot, s_0; \alpha)$ for each state s. (States are on the x axis, τ on the y axis) ($\alpha = 0.99$, random policy). X-axis - states, y-axis - K.

In the Figure 28 we see that the states with the largest returning time are the state that are the furthest from the initial state. (can be seen on the illustration in the figures 2 - 5). This is how the environment works: we start at the initial state and we computer the matrix P based on this. The most frequent states are close to the initial states, thus even with random policy we more probably stay near them.

3.2.4 Answer the previous questions with an optimal policy instead, and discuss the differences.

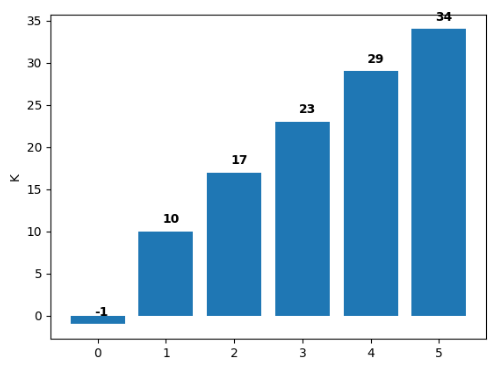


(a) $\alpha = 0.99$

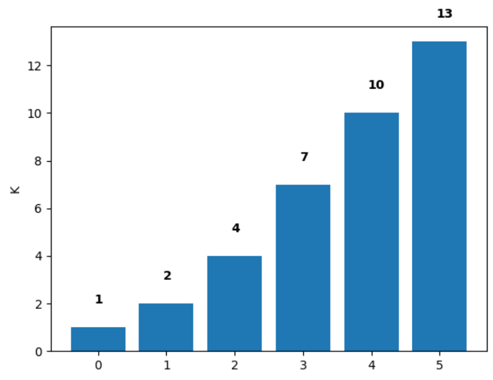


(b) $\alpha = 0.5$

Figure 29: 4-rooms. Minimum K so that the probability to reach each state is at least α. For the first state it is a returning K. (optimal policy) X-axis - states, y-axis - K.

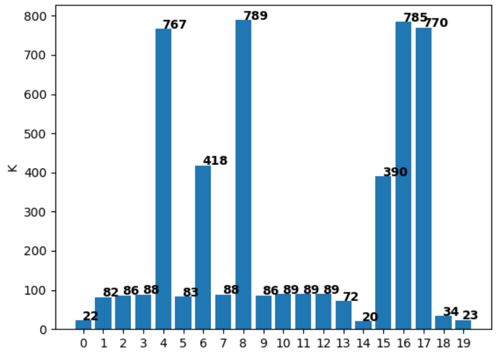


(a) $\alpha = 0.99$

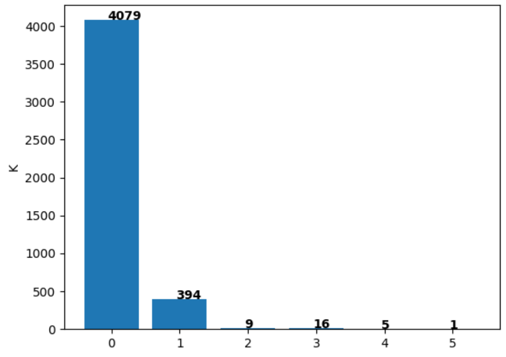


(b) $\alpha = 0.5$

Figure 30: RiverSwim6. Minimum K so that the probability to reach each state is at least α. For the first state the value of returning K with $\alpha = 0.99$ is too big, so we just show -1 instead. (optimal policy). X-axis - states, y-axis - K.



(a) 4rooms



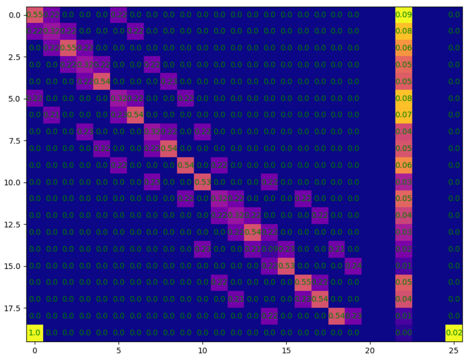
(b) riverSwim6

Figure 31: $\tau(\cdot, s_0; \alpha)$ for each state s. (States are on the x axis, τ on the y axis) (optimal policy) X-axis - states, y-axis - K.

The policies try to put the particle close to the goal state. Thus, the time to reach a goal state is very small (the minimum Ks are small).

The returning time for final states is much smaller than the returning time obtained for them with random policies. The most difficult states to achieve are now the ones that do not lay on the direct way when we go from the initial state to the goal one. (4, 8, 17, 18 for 4-rooms). Returning to the initial state is difficult for riverSwim as the policy takes us away from it.

Example of a random policy transition matrix(symmetrical):


(a) Transition matrix

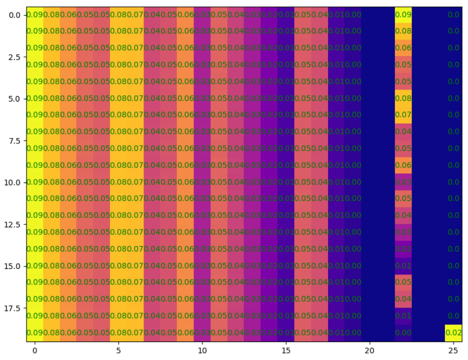

(b) Averaged transition matrix.

Figure 32: TRansition matrices for random policies.

4 TP Part D Learning strategies

4.1 Q-learning

4.1.1 Implement in package learners Q-learning with ϵ-greedy selection.

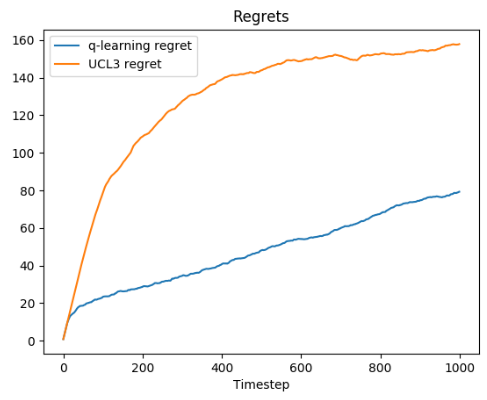
```

    def Q_update(self, state, action, reward, observation):
        old_value = self.q_table[state, action]
        next_max = np.max(self.q_table[observation])
        new_value = (1 - self.alpha) * old_value + self.alpha * (reward + self
.gamma * next_max)
        self.q_table[state, action] = new_value

```

Listing 8: Update rule for implemented Q learning

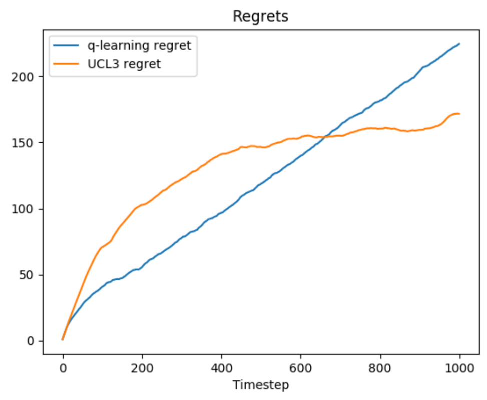
4.1.2 Compare its regret against that of UCRL3 strategy on a small MDP (e.g. RiverSwim with 6 states). Consider various choice for ϵ.

To compute the regret we used the formula:

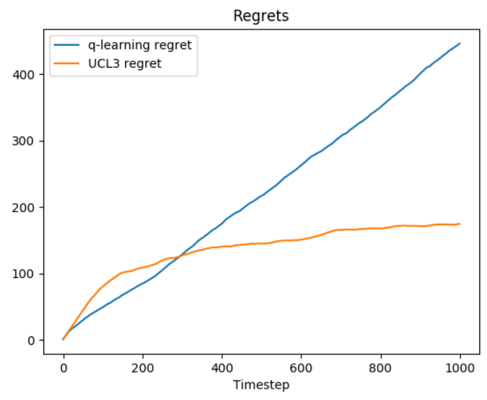
$$R_T = Tg_* - \sum_{t=1}^{T} r(t)$$



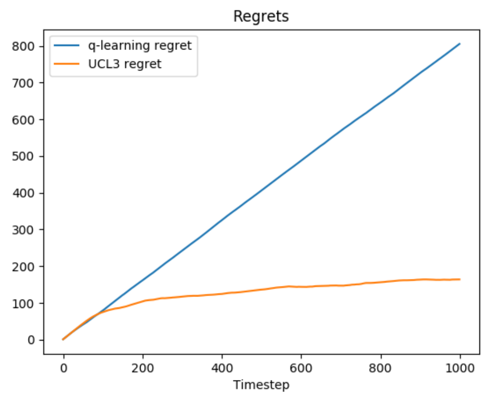
(a) $\epsilon = 0.05$



(b) $\epsilon = 0.1$



(c) $\epsilon = 0.2$



(d) $\epsilon = 0.5$

Figure 33: RiverSwim6. UCL3 curve is the same for all graphics. We run 1000 experiments with 1000 timesteps and than take an average over the experiences.

4.1.3 What is the positive thing about Q-learning ? What is the negative side ?

Qlearning does not need any prior knowledge on the environment. We can control the exploration rate (ϵ) to achieve good results.

4.1.4 What is the positive thing about UCRL ? What is the negative side?

From the experiments we can see that UCRL is more active during the beginning in terms of exploration and then it automatically follows the optimal policy while for Q-learning we have to adjust epsilon ourselves.