Vrije Universiteit Amsterdam

Multi-Agent Systems
XM_0052

Homework Assignment 6 - Personal

Authors:
Simone Montali (2741135) - s.montali@student.vu.nl
December 26, 2021

1 Monte Carlo Sampling

We are now faced with a problem that we could actually define a *sequential decision problem*, even though no game theory will be applied in the solution. Our *agent* is looking for a rental apartment in Amsterdam, and has to decide, knowing the score of the apartment, whether it will be a viable choice or not. In other words, it is a matter of *exploration vs. exploitation*: at the end of every visit, we wonder whether the visits in the future will yield better, or worse, apartments. To formalize this problem, we could make use of some probability theory notation: we define the scores for apartments 1 to i as i.i.d. random variables from 0 (lowest possible score, terrible house) to 1 (highest possible score, really nice house).

1.1 A possible solution to the problem

A good interpretation of this problem would therefore be: *given that the houses until now have had an average score of $\overline{X}$, how possible is it that the actual average is way higher than this?* In this way, we would be able to know when a house is above-average and deserves our attention: we could just set a threshold t for which, when $X_i - \overline{X} > t$, we pick the house. **Hoeffding's inequality** [4] allows us to set an upper bound to the probability that our sample average $\overline{X}$ will be highly different from the real average μ. Formally, we know that

$$P(\mu \geq \bar{X} + u) \leq e^{-2nu^2/L^2}$$

This quantity basically tells us how uncertain we are of our sample average. A good u value for our house search would probably be around 0.05: we want to pick a house only when it is above average (by a given threshold t), and the average we have is at most u far from the real mean with $p\%$ probability. These parameters are arbitrary and vary basing on how interested we are in finding a house fast: setting a high threshold and high probability will make the search faster, yet less accurate. Note that L, in this case, is just $1 - 0 = 1$. We now have a final result for this: let's say we wanted to be 90% sure of having picked an above-average (by 0.1) house: at every visited house, we compute the bound for the probability $P(\mu \geq \bar{X} + u)$ and therefore stop when a house is above-average, and this result is 10%.

1.2 Testing this method

In order to test this idea, we can perform Monte Carlo sampling: sampling data using Python's `random.random`, it is possible to simulate a *real-estate tour* to see whether our agent would pick a good house. Note that, using the random library, we're implying that the distribution is uniform. This, though, is probably not the case: we expect the distribution to have an unknown mean, around which most of the houses are. This would basically look like a normal distribution, but bounded between 0 and 1. Something that is similar to that is a **Beta distribution**, a continuous distribution defined in that range, having two parameters α, β that shape it. Having therefore a beta distribution with parameters sampled from `random.random()` would model our problem better.

```
alpha = np.random.rand()
beta = np.random.rand()
print(f"Simulation␣started,␣parameters␣are␣{alpha},␣{beta}")
sample_mean = None
picked = None
for i in range(N):
    this_house = np.random.beta(alpha, beta)
    sample_mean = this_house if not sample_mean else ((sample_mean * (i - 1)) + this_house)
        ↪ / i
    hoef_bound = math.exp(-2 * (i ** 2) * u)
    if this_house > sample_mean + over_average and hoef_bound < probability and picked ==
        ↪ None:
        picked = i
        picked_score = this_house
        print(f"Our␣agent␣decided␣to␣pick␣house␣{i}␣with␣score␣{this_house}")
```

The results are outstanding: over 1000 simulations with the parameters set to $u = 0.05, t = 0.3, p = 0.05, n_{houses} = 100$, our agent managed to pick a house in an average of ≈ 15 visits, with a score that's on average 1% better than the real mean. This is an *impatient* setup: tweaking the parameters by lowering the probability and raising the threshold will raise both the average visits and the score delta. A single simulation round would look as the one seen in figure 1.

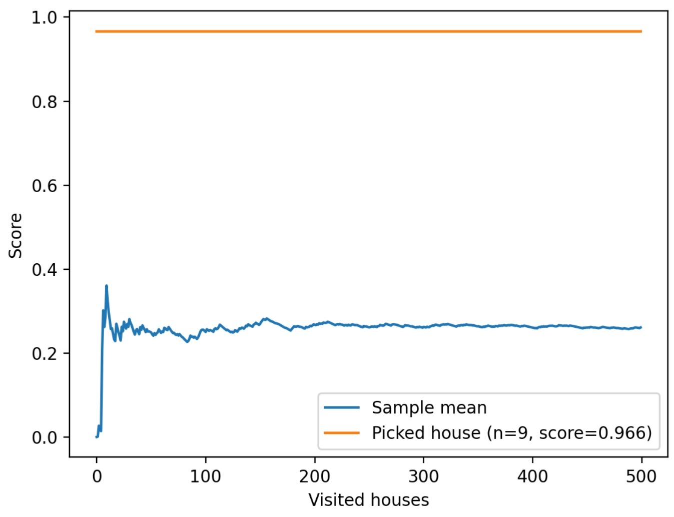


Figure 1: Monte Carlo simulation of our method using a Beta distribution

1.3 Improvements

Modeling the score distribution as a *Beta* is nice, but probably not enough. Fortunately, a large amount of **real** data is available nowadays. The InsideAirBnb project provides review data for the city of Amsterdam in a publicly available dataset. Sampling the house scores from AirBnb reviews would therefore be a more reliable simulation for the environment. Using *Pandas*, we can just import the dataset and sample the scores from there, rescaling them from the 5-stars system to a $[0, 1]$ range.

2 Monte Carlo Tree Search

This problem could be considered a benchmark for Monte Carlo Tree Search methods. We are going to build a binary tree (i.e. a tree in which every node only has 2 children) in which every node contains the path (a sequence of `L` and `R`) to reach it. This structure is actually known as a *Trie*[1], and it is often used for the creation of lists of strings with read complexity $O(1)$. After building the tree, a random leaf node is picked as a target, and values are computed for the other nodes with respect to the distance from the target. The objective function is defined as $x_i = Be^{-d_i/\tau}$, with d_i being the **edit-distance** from the target. For this experiment, the parameters have been set as follows:

$$B = 1 \quad \tau = 2$$

2.1 Building the tree

The tree can be easily built by defining a `Node` class that represents an element of the tree, having the following parameters:

- `self.info: str` contains the string for the node. *e.g. LRRRLRRLLLR*
- `self.children: List[Node]` contains the subtrees starting from this branch
- `self.parent: Node` contains the parent of the node
- `self.visits: int` contains the total visits we performed during the search to this specific node
- `self.depth: int` contains the depth in the tree, which is equal to `len(self.info)`
- `self.distance: float` contains the edit-distance from the target
- `self.value: float` contains the objective function's value

The `MCTS.expansion(node)` will then build the actual strings.

2.2 Search algorithm

As aforementioned, the technique we will use to search the tree is **Monte Carlo Tree Search**: this method approximates the value of a leaf node by computing random trajectories starting from the node itself. 4 steps are needed at every iteration of the search:

2.2.1 Selection

After instantiating the root of the tree with an empty string, we can proceed to select the best leaf. The method traverses the tree until a *final* subtree is reached (meaning a node having leaves as children and not subtrees), then chooses between these children. To perform the selection, this method computes the **UCB value** $UCB_n = \frac{value_n}{visits_n} + C\sqrt{\frac{log(visits_p)}{visits_n}}$ of each child as follows:

```
for child in node.children:
          if child.visits == 0:
              score = np.inf
          else:
              exploit = child.value / child.visits
              explore = np.sqrt(np.log(node.visits) / child.visits)
              score = exploit + c * explore
          if score == best_score:
              best_children.append(child)
          if score > best_score:
              best_children = [child]
              best_score = score
      return np.random.choice(best_children)
```

2.2.2 Expansion

As the chosen node has no children, generating them is necessary. These children will just represent the strings we can build by appending L and R to the current node:

```
if not node.children and node.depth < self.target_depth:
    for addition in ["L", "R"]:
        child = Node(node.info + addition)
        node.add_child(child)
```

2.2.3 Simulation

This is the crucial part for MCTS: instead of traversing the whole tree, only randomly sampled trajectories are considered.

```
while node.depth < self.target_depth:
    addition = np.random.choice(["L", "R"])
    child = Node(
        node.info + addition,
    )
    if node != self.root:
        node.add_child(child)
    node = child
node.set_distance(self.compute_distance(node.info))
return node
```

Note that the tree is explored until the target depth is reached, meaning we'll only compute the values of strings that could be the target. What is now needed is a propagation up the tree of the value we just computed.

2.2.4 Backpropagation

Having simulated a trajectory and obtained a value for its final state, it is necessary to propagate the value up the tree. This is straightforward: we just save the values/visits in a node, than iterate with recursion on its parent, until we reach the root. Note that the *Q-value* for a node will be the average value over all its visits: $Q_i = \frac{\sum v_i}{n_i}$.

```
node = leaf.parent
while node is not None:
    node.visits += 1
    node.value += leaf.value
    node = node.parent
```

2.2.5 Wrapping up

All the components for the MCTS are now available: it is just needed to set a time limit for the iterations, and try to guess the string using the search.

```
while time.time() - start_time < time_limit:
    node = self.selection(self.root)
    self.expansion(node)
    node = self.simulation(node)
    self.backpropagation(node)
```

Now, to use this method, we can try to build the target string one letter at a time. We first instantiate the target as a random choice of *depth* letters: `target = "".join(np.random.choice(["L", "R"], size=` ↪ `target_depth))` and iteratively get letters with MCTS until we reach the desired length.

```
while len(result) < target_depth:
    root = Node(result)
    mcts = MCTS(root, target, verbose=True)
    picked_letter = mcts.search(time_limit=2).info
    result = picked_letter
    print(f"Searched␣string␣now␣{result},␣target={target}")
```

2.3 Results

The results are as good as expected: the target string was found 100% of the times over 1000 iterations with $c = 3$. To better understand the convergence of the algorithm, it is interesting to see how the *Q-values* change over the iterations in figure 2.

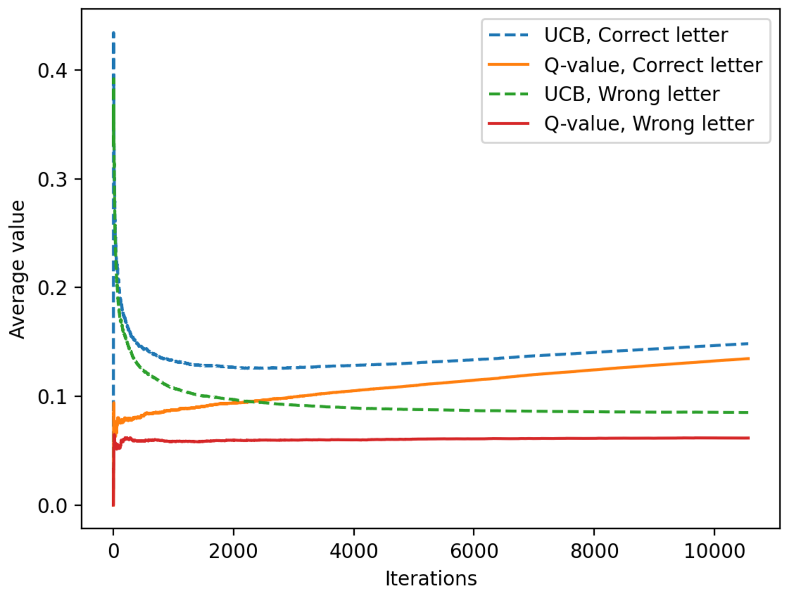


Figure 2: Q-values and UCT convergence using MCTS

Furthermore, several tests with different values of c ranging from 1 to 5 led to the same results: this tree is a really simple structure (as it is binary), and it suffices to have at least a couple of non-greedy explorations to converge to the right results. Tweaking the c parameter to obtain a wrong guess by the MCTS is though possible: with really small exploration values, for example 0.5, the algorithm is not able to find the target node (but a really similar one). The correct target starts to be found with a $c \geq 0.7$.

3 Reinforcement Learning: SARSA and Q-learning

In a Reinforcement Learning setting, we can distinguish between two, major, classes of solutions: the ones having a complete model of the MDP (S, A, R, P, γ), which allow for a *thorough exploration of the space*, and the ability to propagate values using the Bellman equations, and the ones that will have to rely on samples, or better, *trajectories*. We call the first **model-based**, and the latter **model-free**. Of course, having a model of the environment at our disposal is often better than having to rely on trajectories. Some problems, though, have a continuous or large state space, or simply are not fully known to the experimenter. In these cases, model-free approaches represent a viable and interesting solution. Taking the road of *model-free approaches*, two main classes of algorithms exist: *on-policy* algorithms, which sample trajectories using the policy, and *off-policy* algorithms, which don't have to follow the policy while sampling trajectories. The latter are usually more efficient: changing the policy in on-policy methods means that the data collected is not completely reliable anymore. We are now interested in a comparison of two methods that represent the two classes: SARSA [5] and Q-learning [6]. The first updates its state-action value function $q(s, a)$ using on-policy trajectories:

$$q_\pi(s,a) \leftarrow q_\pi(s,a) + \alpha \left(r\left(s,a,s'\right) + q_\pi\left(s',a'\right) - q_\pi(s,a)\right)$$

This is, consequently, an *on-policy* method. Q-learning is a Temporal Difference method as SARSA, but it is based on *off-policy* updates: instead of sampling the trajectory, Q-learning picks the action a' (i.e. the action that is taken after the transition, used to compute the sum of discounted rewards starting from s') to be the one with optimal Q value.

$$q\left(s_t,a_t\right) \leftarrow q\left(s_t,a_t\right) + \alpha\left[r_{t+1} + \gamma \max_{a'} q\left(s_{t+1},a'\right) - q\left(s_t,a_t\right)\right]$$

3.1 The problem

To benchmark these two methods, we can introduce a 9×9 *grid world* containing a treasure, walls and pitfalls. Our agent has to reach the target (returning reward $r_T = 50$) without falling into the pitfall (reward $r_P = -50$), in the shortest time possible (every step has a cost/reward $r_S = -1$). In each cell, the agent can either go up, down, left or right. When the agent hits a wall, its position does not change. On the other hand, falling into a pitfall ends the game.

3.2 Encoding the problem into a Python script

The first, straightforward way to encode the problem is creating a 9×9 NumPy [3] array, containing a symbol that represents whether a cell is empty, a wall, a pitfall, the player or the treasure. A more elegant solution (yet, computationally suboptimal), though, would consist in saving lists of walls, pitfalls, and the ending and starting positions. To make the code as general as possible, it is best to create a separate GridWorld class, having methods similar to the ones seen in *OpenAI Gym Environments* [2]:

```
class GridWorld:
"""
Responsible for the game.
Walls represent cells of the grid we cannot enter. A transition to these can't happen.
Pitfalls represent cells of the grid that will end the game if reached, with a negative
    ↪ reward.
The goal is the only cell returning a reward and ending the game.
"""
def __init__(self, walls: np.array, pitfalls: np.array, size: tuple = (9, 9), start:
    ↪ tuple = (0, 0), goal: tuple = (8, 8)):
    """
    Inits a new game.
    :param walls: the walls of the game
```

```
        :param pitfalls: the pitfalls of the game
        :param size: the size of the grid
        :param start: the starting position
        :param goal: the goal position
        """
```

The crucial component in this class is the GridWorld.step() method, which steps the environment to a new state and returns a reward:

```
def step(self, move: str):
    """
    Steps the agent in a direction.
    :param move: the direction to move: ["up", "down", "left", "right"]
    """
    assert move in ["up", "down", "left", "right"], "Invalid␣move"
    target = self.player_position[:]
    if move == "up" and self.player_position[0] > 0:
        target[0] -= 1
    elif move == "down" and self.player_position[0] < self.size[0] - 1:
        target[0] += 1
    elif move == "left" and self.player_position[1] > 0:
        target[1] -= 1
    elif move == "right" and self.player_position[1] < self.size[1] - 1:
        target[1] += 1
    else:
        return tuple(self.player_position), 0, False
    if tuple(target) == self.goal:
        self.done = True
        return tuple(self.player_position), 50, True
    elif tuple(target) in self.pitfalls:
        self.done = True
        return tuple(self.player_position), -50, True
    elif tuple(target) in self.walls:
        return tuple(self.player_position), 0, False
    else:
        self.player_position = target
        return tuple(self.player_position), 0, False
```

3.3 Monte Carlo policy evaluation

The first task we're interested to perform is an analysis of the **equiprobable policy**, i.e. a policy that picks a random action with equivalent probability. To do so, we will perform a complete sweep of the state space, sampling a trajectory N times and averaging the $v_\pi(s)$ values, equal to the sum of rewards for the trajectory.

```
for i in range(9):
    for j in range(9):
        state_value_sum = 0
        if (i, j) not in walls: # Walls are not traversable
            for _ in range(TRIALS_PER_STATE):
                env.reset()
                env.player_position = [i, j]
                value_sum = 0
                done = False
                while not done:
                    if not ((i, j) == (8, 8) or (i, j) in pitfalls):
                        position, reward, done = env.step(np.random.choice(["up", "down", "
                            ↪ left", "right"]))
                        value_sum += reward
                    elif (i, j) == (8, 8):
```

```
                    value_sum += 50
                    done = True
                else:
                    value_sum -= 50
                    done = True
            state_value_sum += value_sum
        values[i, j] = state_value_sum / TRIALS_PER_STATE
```

The results we obtain are shown in figure 3. We can notice how, as the agent isn't rational, the value just depends on how far we are from the target (and how close we are to walls/pitfalls).

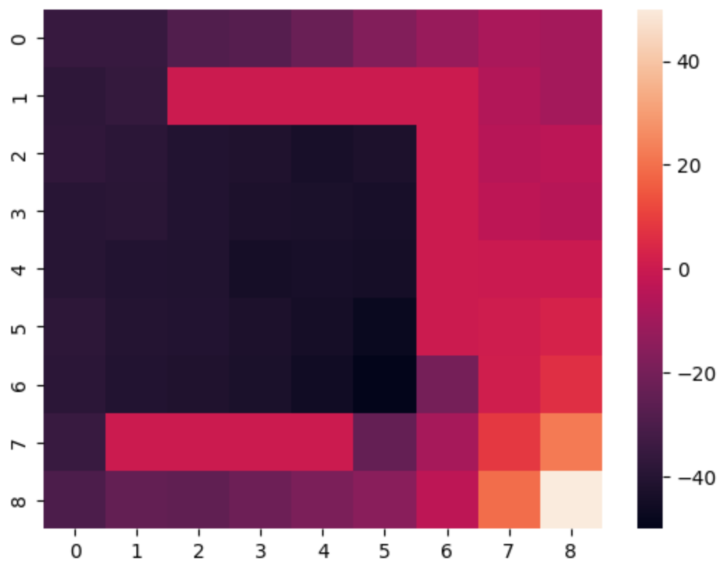


Figure 3: Heatmap of the $v_\pi(s)$ values for the equiprobable policy

3.4 SARSA

As aforementioned, SARSA is an on-policy model-free method that allows us to approximate the $q(s, a)$ function given a dataset of transitions (s, a, s', r). We will then integrate SARSA with an ϵ-greedy approach to ensure a sufficient exploration of the action space. This means that, at each step, our algorithm will pick an optimal (basing on $\max_a q(s, a)$) action with probability $1 - \epsilon$ or a random action with probability ϵ. Given the foundations of SARSA, we can perform an arbitrary number of updates $N = 100000$ consisting of the following steps:

1. The algorithm picks an action a basing on an ϵ-greedy policy
2. The action is played, collecting the next state s', a reward r, and a boolean variable `done` indicating whether the game has terminated
3. The algorithm picks an action a' for state s' and uses it to get a q-value for the next state
4. The Q values are shifted by a quantity $\alpha\,(r + \gamma q(s', a') - q(s, a))$

3.4.1 Encoding

The Python encoding of the algorithm is the following:

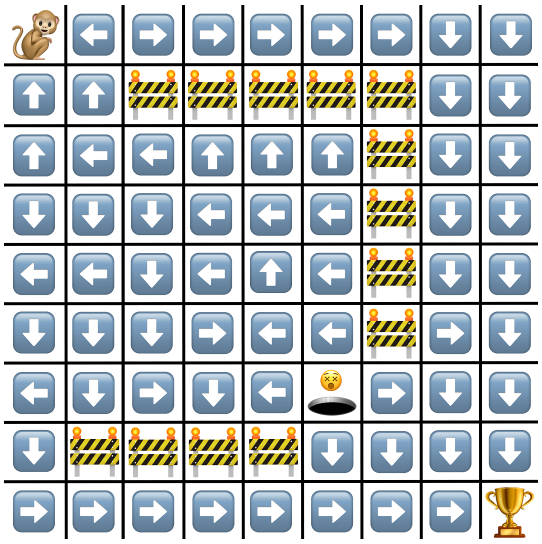
```
for _ in range(1000000):
    if random.random() < epsilon:
        action = np.random.randint(0, 4)
    else:
        action = np.argmax(q_values[tuple(env.player_position)])
    old_position = tuple(env.player_position[:])
    next_state, reward, done = env.step(action_space[action])
    if random.random() < epsilon:
        next_action = np.random.randint(0, 4)
    else:
        next_action = np.argmax(q_values[tuple(next_state)])
    update = alpha * (reward + gamma * q_values[next_state][next_action] - q_values[
        ↪ old_position][action])
    q_values[old_position][action] += update
```

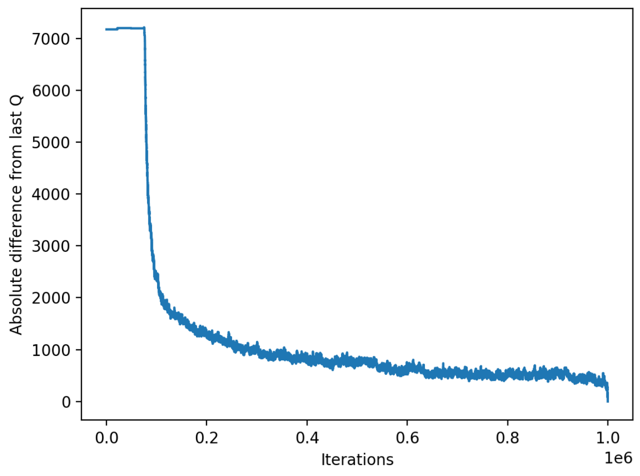
3.4.2 Hyperparameter tuning

Two new variables are now present in the problem: the α and ϵ **hyperparameters**. Setting these correctly is crucial to obtain optimal results and performance. As the problem itself is not too complex, we can perform a parallel grid search to test the different combinations of parameters and how well they converge.

3.4.3 Results using SARSA

Having computed the q-values, we can *greedify* and obtain the optimal policy, i.e. the policy which, for each state, picks the action a: $\max_a q(s, a)$. The resulting policy is shown in figure 4a. We can notice that this policy is, in some points of the grid, not doing the optimal thing: it tends to go towards walls instead of avoiding them, which in the long-term results in wasted resources.


(a) Optimal policy obtained with SARSA



(b) Convergence of the Q-values for SARSA

Figure 4: SARSA results using $\alpha = 0.1$ and $\epsilon = 0.6$

3.5 Q-learning

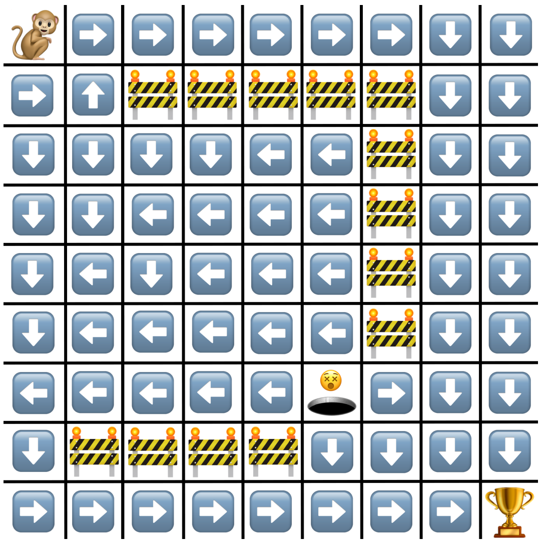
Q-learning is a method which is similar to SARSA, as they both are part of the Temporal Difference algorithm family. The difference between the two lies in the fact that Q-learning is, in fact, an *off-policy* method, meaning that the trajectories can be sampled without looking at the policy. The actual difference in the algorithms is therefore just the way we pick action a': it will now be picked to be $\max_{a'} q(s', a')$. This means that a small modification is needed to the code seen in section 3.4.1: we substitute the selection of `next_action` with a deterministic, greedy one:

```
next_state, reward, done = env.step(action_space[action])
next_action = np.argmax(q_values[tuple(next_state)])
update = alpha * (reward+ gamma * q_values[next_state][next_action]- q_values[old_position][
    ↪ action])
```

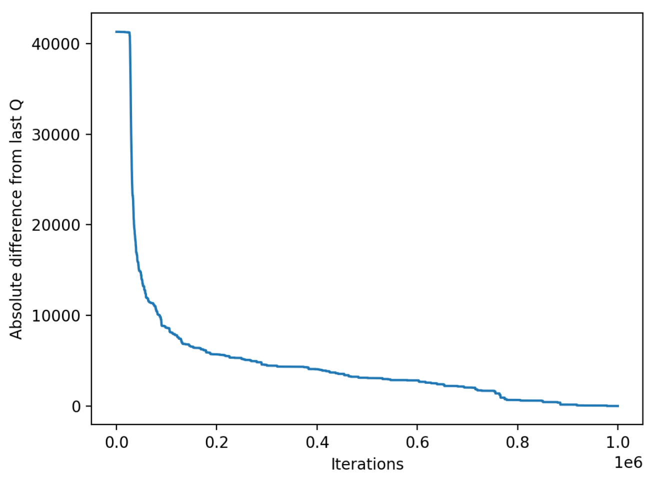
A second round of hyperparameter tuning is clearly needed for this new algorithm, resulting in the optimal parameters $\epsilon = 0.8, \alpha = 0.1$.

3.5.1 Results using Q-learning

This policy looks significantly better than the one seen in figure 4a: walls and pitfalls are avoided, and the goal is reached in the minimal distance from every state. This is accountable to the higher efficiency in Q-learning: this method has actually been proved to converge to the optimal values. Taking a look at the convergence in figure 5b, though, we notice that Q-learning starts with a severely higher difference from the last Q matrix, but recovers quickly, while SARSA is closer to the end result since the beginning.



(a) Optimal policy obtained with Q-Learning



(b) Convergence of the Q-values for Q-Learning

Figure 5: Q-learning results using $\epsilon = 0.8$ and $\alpha = 0.1$

References

[1] Briandais1959FileSUBriandais, RDL. 1959. File searching using variable length keys File searching using variable length keys. IRE-AIEE-ACM Computer Conference. Ire-aiee-acm computer conference.

[2] brockman2016openaiBrockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J. Zaremba, W. 2016. Openai gym Openai gym. arXiv preprint arXiv:1606.01540.

[3] harris2020arrayHarris, CR., Millman, KJ., van der Walt, SJ., Gommers, R., Virtanen, P., Cournapeau, D.Oliphant, TE. 202009. Array programming with NumPy Array programming with NumPy. Nature5857825357–362. https://doi.org/10.1038/s41586-020-2649-2 10.1038/s41586-020-2649-2

[4] 10.2307/2282952Hoeffding, W. 1963. Probability Inequalities for Sums of Bounded Random Variables Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association5830113–30. http://www.jstor.org/stable/2282952

[5] Rummery94on-lineq-learningRummery, GA. Niranjan, M. 1994. On-Line Q-Learning Using Connectionist Systems On-line q-learning using connectionist systems .

[6] watkins1989learningWatkins, CJCH. 1989. Learning from delayed rewards Learning from delayed rewards.