Reinforcement learning - Lab 3

Dimitri Diomaiuta - 30598109

University of Southampton

1 Answer 1

In a 2-player capture the flag game with 2 flags, 5 sites and 100 rounds we have a frequency distribution, which describes how many times a site has been selected. The hider claims that the distribution shown by equation 1 is correct.

$$H_{dist} = (47, 53, 31, 36, 34) \tag{1}$$

The seeker claims, correctly, that this distribution is incorrect. Since the hider selects 2 sites at each timestep t the sum of the frequency distribution should be consistent with equation 2, where T is the number of rounds and $dist$ a frequency distribution.

$$\sum_{i=1}^{|dist|} x_i = flags * T \tag{2}$$

Equation 2 does not hold for the hider distribution, in fact, we have a total sum of 201 ($\sum_{x_i \in H_{dist}}^{sites} x_i$) which is bigger than 200 ($flags * T$). The frequency distribution described by equation 3 respects the constraint described by equation 2 and, hence, can be a correct frequency distribution for the outlined game setting.

$$H_{newdist} = (102, 26, 24, 40, 8) \tag{3}$$

2 Answer 2

Given the hider's frequency distribution described by equation 4 we want to derive the best fixed guess strategy for the seeker.

$$H_{dist} = (36, 24, 40, 63, 37) \tag{4}$$

A fixed guessed strategy is a type of strategy that does not change over time and that involves selecting the same action at each timestep (the same two sites in a capture the flag game setting). In a capture the flag game setting with 2 flags a fixed strategy is any binary combination of the sites where the flags can be hidden. Given distribution described by equation 4, the best fixed strategy for the seeker is to take the action described by equation 5.

$$a_{best} =< x_3, x_4 > where\, x_3, x_4 \in H_{dist} \tag{5}$$

In equation 5, x_3 and x_4 describe the sites where the hider puts the flags more often, respectively 40 and 63 times. The best fixed strategy for the seeker, hence, is to select the two sites where the flags are most frequently hidden.

3 Answer 3

In this section we describe the implementation of different online learning algorithms to play the capture the flag game. Capture the flag is a game with a nonassociative setting, meaning that an agent has to learn which action to take in just one situation that repeats for T times [1]. Online learning algorithms are particularly suitable for this nonassociative and sequential setting. We implemented the seeker's strategy following the Exp3 algorithm since it has efficient theoretical guarantees about the lower bound of the expected regret. We then tested the seeker's strategy against a hider implementing three different strategies: uniform random, epsilon-greedy and FPL. The average reward, the average regret and the total regret of the seeker's are the metrics we used to analyze the algorithms performances. Figures 1, 2 and 3 describe the results of the analyzed metrics.

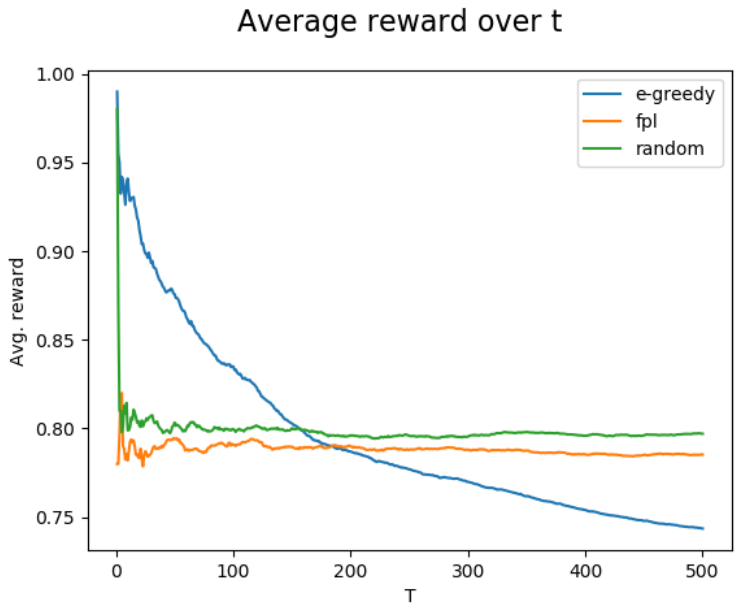

Fig. 1. The seeker's Exp3 strategy average reward

As we can observe from figures 1 and 2 the hider's strategies converge all to a uniform random strategy. In this type of game, the uniform random strategy is the optimal strategy since it cannot be learned and forces the opponent to play a uniform random strategy as well. In this case, the epsilon-greedy and FPL strategies, via a reward and frequency vector, keep track of their actions consequences and update their values in order to minimize the opponent's total reward. The seeker's Exp3 algorithm does something similar by updating a

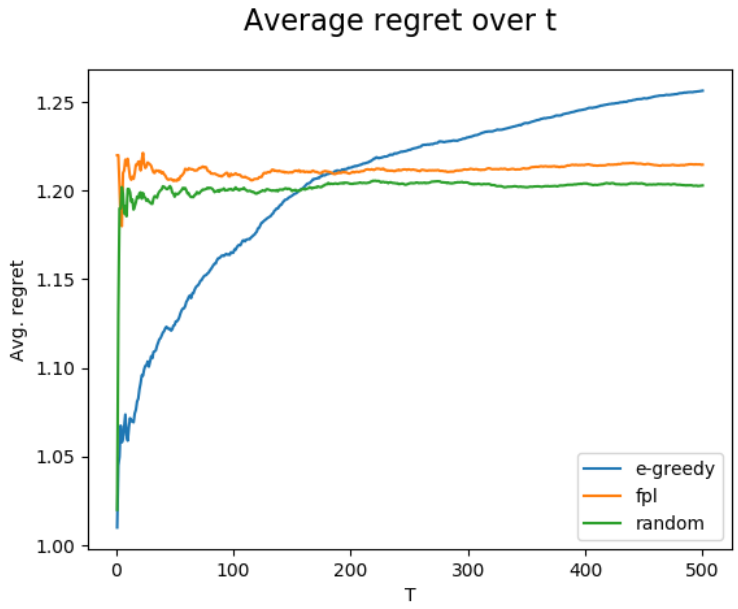

Fig. 2. The seeker's Exp3 strategy average regret

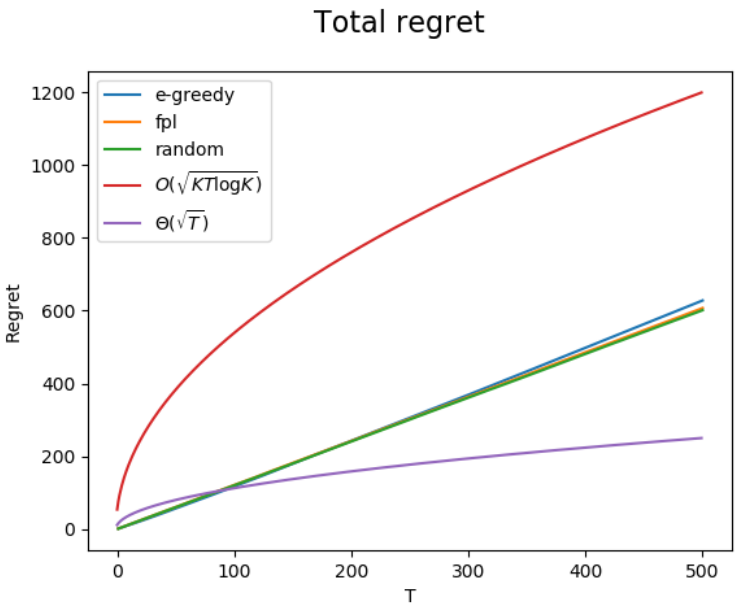

Fig. 3. The seeker's Exp3 strategy total regret

weight vector, which describes the probability of taking an action. In a 2 player capture the flag game setting with 2 flags and 5 sites the number of actions is described by all the possible unique binary combinations of the sites. The resulting Exp3 strategy weights against uniform, epsilon-greedy and FPL strategies result in a uniform (equiprobable) distribution. We can observe from figure 3 that Exp3 total expected regret is better than $\sqrt{T}$, where T is the number of timesteps, even when the adversaries play optimal strategies (uniform random distribution in this case).

All the implemented strategies, except the uniform random one, face the exploration versus exploitation dilemma. At each timestep the algorithms explore according to a small probability degree. The algorithms, on the other hand, exploit by keeping track of the previous interactions with the environment by using weight, frequency or reward history vectors. We have to note that FPL adaptivity strongly depends on the exponential distribution η parameter. When using a fixed η the exponential distribution at the core of the algorithm does not affect the non-normalized action weights anymore. An adaptive η is needed in order to not fully separate the exploration versus exploitation phases. The code implementing the algorithms and plots of the results can be found in appendix 4.

References

1. Sutton, R.S. and Barto, A.G., 2011. Reinforcement learning: An introduction.

4 Appendix A: source code

This appendix section contains the source code of the implemented program.

4.1 capture_the_flag.py

```
import numpy as np
import random
import math
import matplotlib.pyplot as plt


class Flags:
    EMPTY_PLACE = 0
    FLAG_PLACE = 1

    def __init__(self, places_no, flags_no):
        self.places_no = places_no
        self.flags_no = flags_no
        self.flags_array = np.full(places_no, self.EMPTY_PLACE)
```

```
    # Inserts flags on given positions
    def put_flags(self, flags_position):
        self.flags_array.fill(self.EMPTY_PLACE)
        self.flags_array.put(flags_position, self.FLAG_PLACE)

    # Given a set of positions returns the number of matches (1
        for
    # every position rewards correctly)
    def check_flags(self, flags_position):
        result = 0
        for pos in flags_position:
            if self.flags_array[pos] == self.FLAG_PLACE:
                result += 1
        return result

    def __str__(self):
        return "{}".format(self.flags_array)


# Given a flag object computes all the possible actions
def compute_actions(flags):
    x = np.arange(flags.places_no)
    mesh = np.array(np.meshgrid(x, x)).T.reshape(-1, 2)
    mesh.sort(axis=1)
    mesh = np.unique(mesh, axis=0)
    duplicates = []
    for i in range(len(mesh)):
        if mesh[i][0] == mesh[i][1]:
            duplicates.append(i)
    return np.delete(mesh, duplicates, 0)


class Hider:

    def __init__(self, flags, hider_type):
        self.valid_types = ['random', 'e-greedy', 'fpl', 'fixed']
        if hider_type not in self.valid_types:
            raise ValueError("results: status must be one of {}".
                format(self.valid_types))
        self.hider_type = hider_type
        self.flags = flags
        self.actions = compute_actions(flags)
        self.frequencies = np.zeros(len(self.actions))
        self.epsilon = 0.1
```

```
        self.rewards = np.zeros(len(self.actions))
        self.action_index = 0
        self.fpl_lr = 35

    def select_action(self):
        if random.random() > self.epsilon:
            return np.argmax(self.rewards)
        return np.random.choice(self.actions.shape[0])

    # normalize and reverse the reward obtain by seeker
    def compute_hider_reward(self, reward):
        return abs(reward - self.flags.flags_no) / float(self.
            flags.flags_no)

    def hide_flags_epsilon_greedy(self):
        self.action_index = self.select_action()
        positions = self.actions[self.action_index]
        self.frequencies[self.action_index] += 1
        self.flags.put_flags(positions)

    def update_reward_epsilon_greedy(self, reward):
        reward = self.compute_hider_reward(reward)
        n = float(self.frequencies[self.action_index])
        prev_reward = self.rewards[self.action_index]
        self.rewards[self.action_index] = ((n - 1) / n) *
            prev_reward + (1 / n) * reward

    def hide_flags_fpl(self):
        exp_dist = np.random.exponential(self.fpl_lr, len(self.
            actions))
        # Add exponentially drawn noise
        new_rewards = self.rewards + exp_dist
        self.action_index = np.argmax(new_rewards)
        self.frequencies[self.action_index] += 1
        self.flags.put_flags(self.actions[self.action_index])

    def update_reward_fpl(self, reward):
        reward = self.compute_hider_reward(reward)
        self.rewards[self.action_index] += reward

    def hideFlagsRandomly(self):
        random_index = np.random.choice(self.actions.shape[0])
        positions = self.actions[random_index]
        # positions = [0, 1]
        self.frequencies[random_index] += 1
```

```
        self.flags.put_flags(positions)

    def hide_flags_fixed(self):
        self.flags.put_flags(self.actions[0])

    def hide_flags(self):
        if self.hider_type == self.valid_types[0]:
            return self.hideFlagsRandomly()
        if self.hider_type == self.valid_types[1]:
            return self.hide_flags_epsilon_greedy()
        if self.hider_type == self.valid_types[2]:
            return self.hide_flags_fpl()
        return self.hide_flags_fixed()

    def update_reward(self, reward):
        if self.hider_type == self.valid_types[1]:
            return self.update_reward_epsilon_greedy(reward)
        if self.hider_type == self.valid_types[2]:
            return self.update_reward_fpl(reward)
        return


class Seeker:

    def __init__(self, flags):
        self.flags = flags
        self.rewards = 0
        self.gamma = 0.2
        self.actions = compute_actions(flags)
        self.weights = np.ones(len(self.actions))

    def categorical_distribution(self, probabilities):
        rand, cumulative = random.random(), 0
        for i, p in enumerate(probabilities):
            cumulative += p
            if cumulative > rand:
                return i, p
        return i, p

    # exp3 place selection
    def select_place(self):
        probabilities = (1 - self.gamma) * (self.weights / sum(
            self.weights)) + (self.gamma * 1.0 / self.flags.
            places_no)
        return self.categorical_distribution(probabilities)
```

```
    # exp3 weight update
    def update_weight(self, action_index, prob, reward):
        estimated_reward = (reward / self.flags.flags_no) / prob
        self.weights[action_index] *= math.exp(estimated_reward *
            self.gamma / len(self.actions))

    def seekFlags(self):
        # positions = random.sample(range(self.flags.places_no),
            self.flags.flags_no)
        # self.rewards += self.flags.check_flags(positions)
        action_index, prob = self.select_place()
        reward = self.flags.check_flags(self.actions[action_index
            ])
        self.rewards += reward # / self.flags.flags_no
        self.update_weight(action_index, prob, reward)
        return reward


strategies = ['e-greedy', 'fpl', 'random'] # add 'fixed' to play
    against fixed strat opponent
rounds_no = 500
flags_places = 5
flags_no = 2
flags = Flags(flags_places, flags_no)
results = dict()
for s in strategies:
    cumulative_rewards = np.zeros(rounds_no)
    reward_history = np.zeros(rounds_no)
    for i in range(100):
        hider = Hider(flags, s)
        seeker = Seeker(flags)
        pred = 0
        for i in range(rounds_no):
            hider.hide_flags()
            reward = seeker.seekFlags()
            reward_history[i] += reward
            cumulative_rewards[i] += reward + pred
            pred += reward
            hider.update_reward(reward)
    cumulative_rewards /= 100
    reward_history /= 100
    print("Results␣for␣hider␣strategy:␣{}".format(s))
    print(seeker.rewards)
    print(hider.frequencies)
```

```
    results[s] = (cumulative_rewards, reward_history)

# Plotting seeker's average reward
for s in strategies:
    reward = results[s][0]
    x_axis = np.linspace(1, rounds_no, rounds_no)
    # total_regret = np.cumsum(flags_no - reward)
    plt.plot(x_axis, reward / x_axis, label=s)
plt.xlabel('T')
plt.ylabel('Avg. reward')
plt.legend()
plt.suptitle("Average reward over t", fontsize=16)
plt.show()
# Plotting seeker's average regret ==> cumulative reward /
    rounds_no
for s in strategies:
    regret = 2 - results[s][1]
    plt.plot(x_axis, np.cumsum(regret)/x_axis, label=s)
plt.xlabel('T')
plt.ylabel('Avg. regret')
plt.legend()
plt.suptitle("Average regret over t", fontsize=16)
plt.show()
# Plotting total regret against different strategies
K = len(hider.actions)
T = rounds_no
upper_bound = np.sqrt(x_axis * K * T * np.log(K)) / 2
lower_bound = np.sqrt(T * x_axis) / 2
for s in strategies:
    total_regret = np.cumsum(flags.flags_no - results[s][1])
    plt.plot(x_axis, total_regret, label=s)
plt.plot(upper_bound, label=r'$O(\sqrt{KT\log{K}})$')
plt.plot(lower_bound, label=r'$\Theta(\sqrt{T})$')
plt.xlabel('T')
plt.ylabel('Regret')
plt.legend()
plt.suptitle("Total regret", fontsize=16)
plt.show()
```