FrozenLake 5

UWAGA

Wczytaj do Colab plik **frozen_lake.py** (intrukcja w pliku **COLAB_instrukcja.pdf**).

In [57]:
```
from frozen_lake import FrozenLakeEnv
#from frozen_lake_slippery import FrozenLakeEnv
import numpy as np
import random

env = FrozenLakeEnv()
```

Implementacja algorytmu

Algorytm wygląda następująco (objaśnienia do algorytmu w wykładzie 5):

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Funkcję **Q** inicjujemy **zerami** za pomocą tablicy o wymiarach **16x4**:

In [58]:
```
Q = np.zeros([env.nS,env.nA])
print(Q)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Definicja funkcji, która dla danego stanu **S** zwraca akcję zgodnię z **polityką epsilon-zachłanną**:

In [59]:
```
def epsilon_greedy_action(env,Q,state,epsilon=0.3):
    n = random.uniform(0,1)
    if n<= epsilon:
        return np.random.randint(env.action_space.n)
    else:
        return np.argmax(Q[state])
```

Objaśnienie: losujemy liczbę **n** z przedziału **(0,1)**. Jeżeli **n<epsilon** wówczas funkcja zwraca **losową akcję**. Jeżeli **n>epsilon** wówczas funkcja zwraca **akcję o największym zwrocie**.

Polecenie 1 (do uzupełnienia)

Uzupełnij poniższą funkcję implementującą algorytm **SARSA**:

In [60]:
```
def SARSA_Q(env, episodes=1000, gamma=0.91, alpha=0.1):

    Q = np.zeros([env.nS,env.nA])

    for i in range(episodes):
        finished = False

        env.reset()

        S = env.s
        A = epsilon_greedy_action(env,Q,S)
        while not finished:

            next_S, R, finished, _ = env.step(A)
            next_A = epsilon_greedy_action(env,Q,next_S)
            #Q[S][A] = #DO UZUPEŁNIENIA
            Q[S][A] = Q[S][A] + alpha*(R+ gamma*Q[next_S][next_A] - Q[S
][A]);
```

```
            S = next_S
            A = next_A


    return Q
```

Test:

In [61]:
```
Q = SARSA_Q(env,2000)
print(np.round(Q,2))   #zaokrąglamy do dwóch miejsc po przecinku
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

Polecenie 2 (do uzupełnienia)

Przestestuj działanie algorytmu **SARSA** dla dwóch wartości parametru gamma (**0.1** i **0.99**) i różnych ilości epok. Jak oceniasz działanie algorytmu? Czy wybór w każdym stanie akcji związanej z największym zwrotem gwarantuje dotarcie do celu?

Dzialanie algorytmu jest poprawne, lecz zeby otrzymac jakiekolwiek wartosci musimy ustawic

odpowiednio duza ilosc epok, gdyz "nauczanie" odbywa sie po dlugim czasie.

Przyjmujac parametr gamma rowny 0.1, otrzymane wartosci zwrotow sa bliskie zeru, a przyjmujac wartosc gamma rowny 0.99 nasze wartosci bede o wiele wieksze. Wartosc wykonanych epok nie ma tu wiekszego wplywu.

Duza wartosc epok oraz parametru gamma da nam w kazdym stanie mozliwosc przedostania sie do celu. Jesli trafiamy na akcje ktora prowadzi do wpadniecia do dziury to otrzymujemy znacznie nizszy zwrot niz akcja ktora prowadzi nas sciezka do celu, co daje nam kiedy wybieramy sciezke z najwiekszym zwrotem prosta droge do mety.