# FrozenLake 4

## UWAGA

Wczytaj do Colab plik **frozen_lake_slippery.py** lub **frozen_lake.py** (intrukcja w pliku **COLAB_instruk**

Wczytaj też plik **plot_utils.py**.

```
from frozen_lake import FrozenLakeEnv
#from frozen_lake_slippery import FrozenLakeEnv
import numpy as np
from plot_utils import plot_values

env = FrozenLakeEnv()
```

## Objaśnienie algorytmu

Algorytm wygląda następująco:

**Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$**

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
      $\Delta \leftarrow 0$
      Loop for each $s \in \mathcal{S}$:
         $v \leftarrow V(s)$
         $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
         $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      *old-action* $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

W algorymie można wyróżnić dwa główne bloki.

**Blok 1**

Algorytm wylicza **wartość zwrotów *V(s)*** dla wszystkich stanów ***s*** przy zadanej **polityce determinis** się w **punkcie 2** algorytmu).

**Blok 2**

Algorytm znajduje **politykę deteministyczną *pi*** na podstawie wyliczonych wcześniej wartości ***V(s)*** zarówno ***V(s)*** jak i polityka ***pi*** są początkowo dowolne (***punkt 1*** algorytmu).

To co ciekawe odbywa się w nastepującej **pętli**:

Algorytm wylicza ***V(s)*** dla zadanej **polityki deterministycznej *pi***, a następnie wyliczone wartości ***V(*** Zmodyfikowana polityka służy do ponownego wyliczenia ***V(s)*** itd.

W efekcie takich wzajemnych modyfikacji ***V*** i ***pi*** znaleziona zostaje **polityka optymalna** (najlepsza problemu.... ***MAGIA*** :)

Warto zwrócić uwagę na **punkt 3** w algorytmie:

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      *old-action* $\leftarrow \pi(s)$
      $\boxed{\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]}$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go t

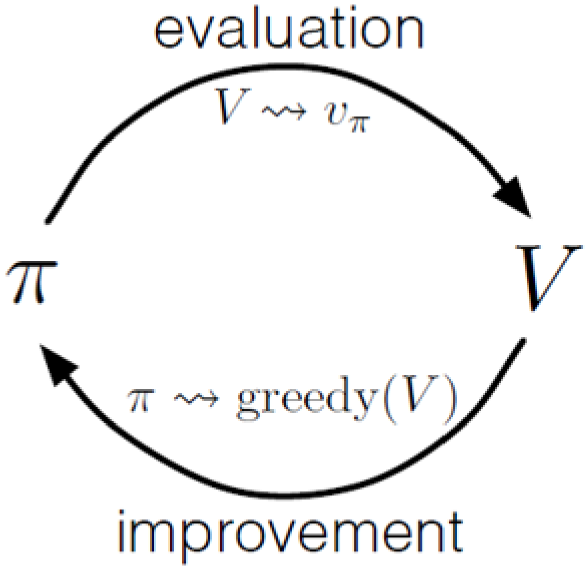
W formule w ramce użyta jest funkcja **argmax**. Jej definicja jest następująca:

$$\arg\max_{x \in S \subseteq X} f(x) := \{x \mid x \in S \wedge \forall y \in S : f(y) \leq f(x)\}.$$

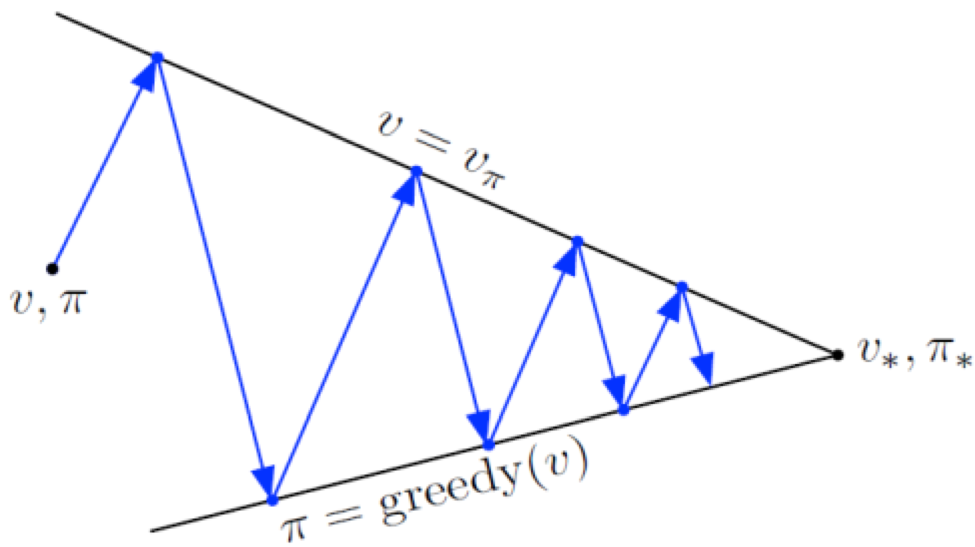
***argmax f(x)*** zwraca wartość argumentu ***x*** dla którego funkcja ***f*** osiąga **wartość maksymalną**.

Linika oznaczona powyżej **czerwoną** ramką oznacza, że **nowa polityka *pi*** przypisuje stanowi ***s*** akc **największy zwrot**! Czyli stosowana jest tutaj **strategia zachłanna (greedy)**.

Petlę w algotymie możemy zobrazować następująco:




Wynikiem działania algorytmu jest **optymalna polityka** (deterministyczna) **pi*** i **V*** dla tej polityki.




## Polityka deterministyczna

W implemntacji algorytmu będziemy stosowali **politykę deteministyczną**. Jest to polityka, która ka
być wykonana w tym stanie. Możemy ją zdefiniować następująco (***env.nS*** to liczba stanów w środ

```
pi = np.random.randint(0, env.nA, size=env.nS)

print(pi)
```

```
[3 1 1 2 3 2 0 2 0 3 2 0 3 0 1 1]
```

Polityka ta dla każdego stanu **s** określa akcję.

Przykład: akcja wykonana w stanie 4 (stany numerowane są od 0 do 15):

```
pi[4]
```

```
3
```

# Wyliczenie V dla zadanej polityki

Zajmijmy się **punktem 2** algorytmu czyli wyliczeniem ***V(s)*** dla zadanej **polityki deterministycznej *p*** dotyczyło **zadanie 3** z **RL_lab_4.pdf**.

# Polecenie 1 (do uzupełnienia)

**Uzupełnij poniższą definicję** funkcji zwracającej ***V*** dla zadanej **polityki deterministycznej *policy***.

```
def det_policy_evaluation(env, policy, gamma=0.9, theta=1e-8):
    V = np.zeros(env.nS)

    while True:
        delta = .0
        for state in range(env.nS):
            Vs = 0
            for next_state in range(len(env.P[state][policy[state]])):

                prob, next_state, reward, done = env.P[state][policy[state]][next_state]

                #DO UZUPEŁNIENIA
                Vs+=prob*(reward+gamma*V[next_state])

            delta = max(delta, np.abs(V[state] - Vs))
            V[state] = Vs
        if delta < theta:
            break

    return V
```

UWAGA: warto zwrócić uwagę, gdzie w powyższym kodzie użyta jest polityka deterministyczna **po**

```
for next_state in range(len(env.P[state][policy[state]])):
```

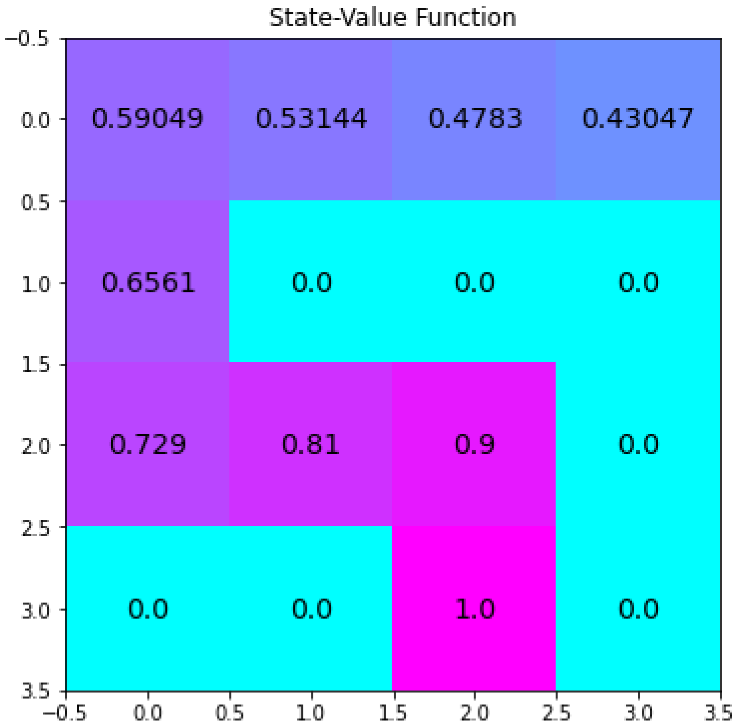
Testujemy działanie funkcji dla **polityki deterministycznej *pi***:

```
pi = np.array([1,0,0,0,1,0,0,0,2,2,1,0,0,0,2,0])
V = det_policy_evaluation(env,pi)
print(V)
```

```
[0.59049    0.531441   0.4782969  0.43046721 0.6561     0.
 0.         0.         0.729      0.81       0.9        0.
 0.         0.         1.         0.        ]
```

**Wartości *V(s)*** można przedstawić graficznie korzystając z funkcji **plot_values()** z **biblioteki plot_ut**

```
plot_values(V)
```




# Znalezienie polityki dla zadanego V metodą zachłanną

Zajmijmy się teraz **punktem 3** algorytmu czyli znalezieniem **polityki deterministycznej *pi*** dla dane
kodu:

3. Policy Improvement
   *policy-stable* $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
      *old-action* $\leftarrow \pi(s)$
      $\pi(s) \leftarrow \operatorname{argmax}_a \boxed{\sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]}$
      If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go t

Warto zauważyć, że w powyższym kodzie nie tylko jest znajdowana nowa polityka, ale **także jest c** obie są takie same, wówczas algorytm kończy działanie. W przeciwnym razie następuje powrót dc

## Polecenie 2 (do uzupełnienia)

Znalezie polityki dla danego ***V*** można "zamknąć" w funkcji **det_policy_iteration**, która jako argume nam wyliczoną politykę. Warto zwrócić uwagę na to, że **formuła oznaczona powyżej czerwoną rar części 5**. Czyli możemy wykorzystać funkcję już zdefiniowaną (w **FrozeLake_3.ipynb**). Uzupełnij je

```
def Q_from_V(env, V, s, gamma=0.9):
    Q = np.zeros(env.nA)
    for action in range(env.nA):
      for next_state in range(len(env.P[s][action])):
        prob, next_state, reward, done = env.P[s][action][next_state]
        Q[action]+=prob * (reward + gamma * V[next_state])
        #DO UZUPEŁNIENIA

    return Q
```

Czas na funkcję **det_policy_iteration**:

```
def det_policy_iteration(env, V, gamma=0.9):
    policy = np.zeros([env.nS])

    for state in range(env.nS):
      policy[state]=np.argmax(Q_from_V(env,V,state,gamma))
        #DO UZUPEŁNIENIA

    return policy
```

Przetestuj działanie funkcji **det_policy_iteration** dla poniższego ***V***:

```
V = np.array([0.1,0.5,0.8,0.2,0.2,0.,0.4,0.,0.3,0.5,0.8,0.,0.,0.1,1.,0.])
pi = det_policy_iteration(env,V)
print(pi)
```

```
[2. 2. 3. 0. 1. 0. 1. 0. 2. 2. 1. 0. 0. 2. 2. 0.]
```

# Implementacja algorytmu

W implementowanym algorytmie po znalezieniu nowej polityki następuje **porównanie** jej ze starą p

$$\text{If } \textit{old-action} \neq \pi(s), \text{ then } \textit{policy-stable} \leftarrow \textit{false}$$

**Polityki** są u nas zapisane **w tablicach**. Musimy mieć zatem **metodę porównującą dwie tablice** i zw **są/nie są** identyczne.

# Polecenie 3 (do uzupełnienia)

Uzupełnij poniższą funkcję pozwalającą porównywać dwie polityki (tablice) będące jej argumentar

```
def compPolicy(p1,p2):
  if((p1==p2).all()):
    return True
  return False
    #DO UZUPEŁNIENIA
```

Przetestuj działanie funkcji dla dwóch przykładowych polityk:

```
pi_1 = np.array([3,2,0,3,0,3,3,0,1,3,3,0,0,3,0,0])
pi_2 = np.array([3,2,0,3,0,0,1,0,1,3,3,0,0,3,0,0])

print(compPolicy(pi_2,pi_1))
```

```
False
```

Mamy już wszystkie elementy wymagane do implementacji całego **algorytmu**.

# Polecenie 4 (do uzupełnienia)

Uzupełnij poniższą petlę tak, aby otrzymać **pełną implementację algorytmu**:

```
pi = np.array([1,0,0,0,1,0,0,0,2,2,1,0,0,0,2,0])
V = np.array([0.1,0.5,0.8,0.2,0.2,0.,0.4,0.,0.3,0.5,0.8,0.,0.,0.1,1.,0.])
```

```
while True:
  V = det_policy_evaluation(env,pi)
  pi2 = det_policy_iteration(env,V)
  if(compPolicy(pi,pi2)):
    break
  pi=pi2



      #DO UZUPEŁNIENIA
```

Po wykonaniu powyższego kodu w zmiennej ***pi*** będzie zapisana **optymalna polityka**. Wypiszmy ją:

```
print(pi)
```

```
[1. 2. 1. 0. 1. 0. 1. 0. 2. 1. 1. 0. 0. 2. 2. 0.]
```

# Polecenie 5 (do uzupełnienia)

Sprawdź czy **znaleziona polityka** jest optymalna. Odpowiedź uzasadnij.

TUTAJ WPISZ ODPOWIEDŹ:

Znaleziona polityka jest optymalna, ponieważ pokazuje najkrótsza możliwą drogę do celu.

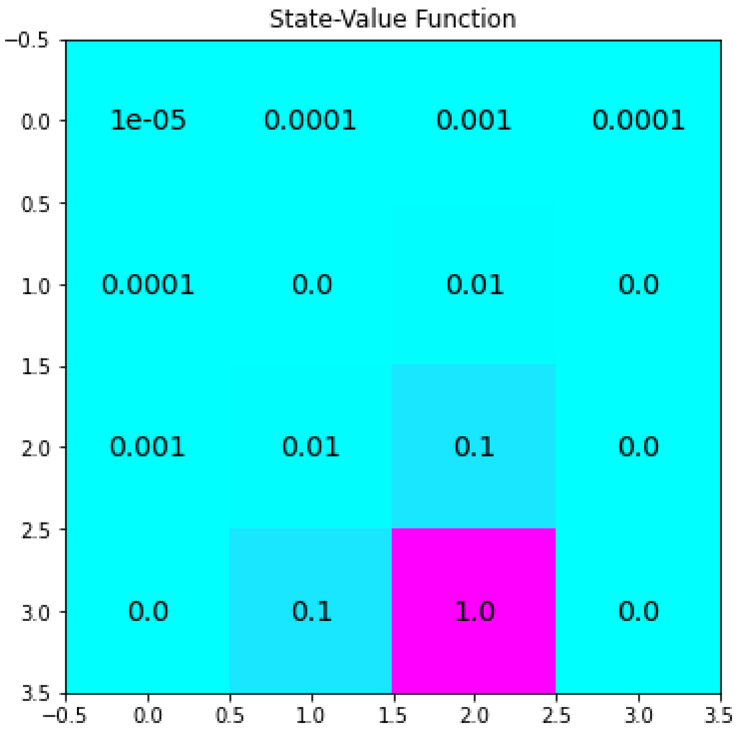
Zobaczmy jak wyglądają **wartości oczekiwane zwrotów V** dla znalezionej polityki:

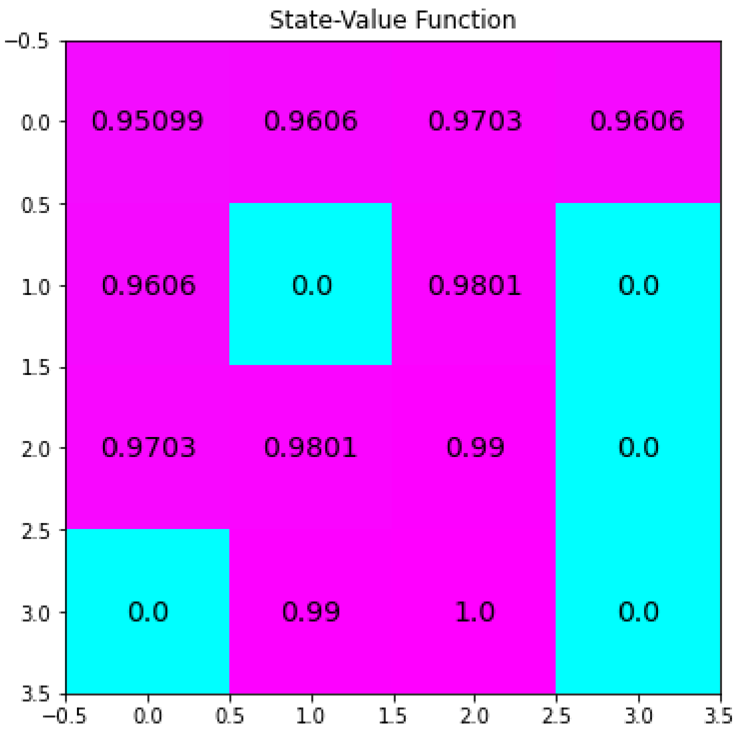
```
plot_values(V)
```

# Polecenie 6 (do uzupełnienia)

Sprawdź czy znaleziona powyżej **polityka optymalna** zmienia się jeżeli przyjmiemy dwie różne war

```
pi = np.array([1,0,0,0,1,0,0,0,2,2,1,0,0,0,2,0])
V = np.array([0.1,0.5,0.8,0.2,0.2,0.,0.4,0.,0.3,0.5,0.8,0.,0.,0.1,1.,0.])

while True:
  V = det_policy_evaluation(env,pi,0.1)
  pi2 = det_policy_iteration(env,V,0.1)
  if(compPolicy(pi,pi2)):
    break
  pi=pi2

plot_values(V)

pi = np.array([1,0,0,0,1,0,0,0,2,2,1,0,0,0,2,0])
V = np.array([0.1,0.5,0.8,0.2,0.2,0.,0.4,0.,0.3,0.5,0.8,0.,0.,0.1,1.,0.])

while True:
  V = det_policy_evaluation(env,pi,0.99)
  pi2 = det_policy_iteration(env,V,0.99)
  if(compPolicy(pi,pi2)):
    break
  pi=pi2

plot_values(V)
```




```
TUTAJ WPISZ ODPOWIEDŹ:

Po zmiane wartości gamma obydwie polityki nadal są optymalne
```