Sinem Ertem (178448)

Model-Free Prediction and Control

1. Model-Free prediction

Een vorm van ervaring op doen noemen we "Sampling" in het veld van Reinforcement Learning. Om een value-function te krijgen zonder gegeven model van de wereld kunnen we dit idee toepassen.

1.1. Monte-Carlo Policy Evaluation

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$

*In **Monte Carlo** we play an episode of the game starting by some random state till the end, record the states, actions and rewards that we encountered then compute the V(s) and Q(s) for each state we passed through. We repeat this process by playing more episodes and after each episode we get the states, actions, and rewards and we average the values of the discovered V(s) and Q(s).*
https://zsalloum.medium.com/monte-carlo-in-reinforcement-learning-the-easy-way-564c53010511

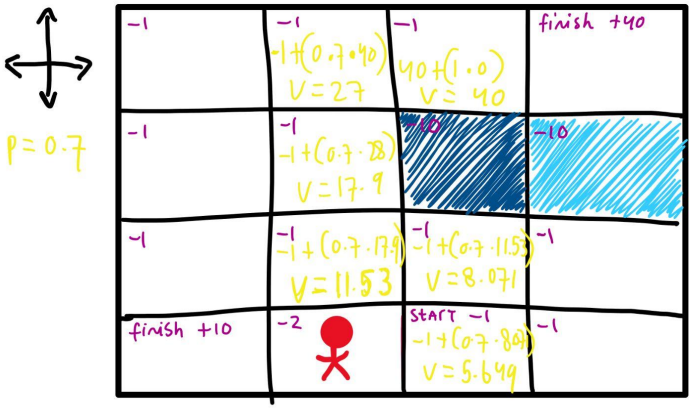
Evaluating the policy is the process of calculating the value of each state when the agent follows the set of actions dictated by that policy.

Monte Carlo Methods are **based on averaging sampled returns**

- **on-policy** (evaluate or improve the policy that is used to make decisions)

- **off-policy** (evaluate or improve a policy different from that used to generate the data)

Voorbeeld met de hand



START = (3,2)
GOAL = (0,3)

RETURN VALUES ARE JUST MADE UP FOR EXPLAINING PURPOSED
Return (sample 1) = UP,LEFT,LEFT,UP,UP,RIGHT,RIGHT = -35
Return (sample 2) = RIGHT,UP,UP,UP = -28
Return (sample 3) = UP,RIGHT,UP,UP = -28

Observed mean return (based on 3 samples) =
((-35)+(-28)+(-28))/3 = **-30.33...**
Thus the state value as per *Monte Carlo Method* Vpi(S(3,2)) is -30.33 on 3 samples following policy pi

Implementeer Monte-Carlo Policy evaluation en voer deze uit op de doolhof. Evalueer daarmee twee policies met γ=0.9 *en* γ=1

Lever een screenshot in waar de ***utilities*** (=value van een state) voor elke state zichtbaar zijn na evaluatie met behulp van Monte-Carlo Policy Evaluation.

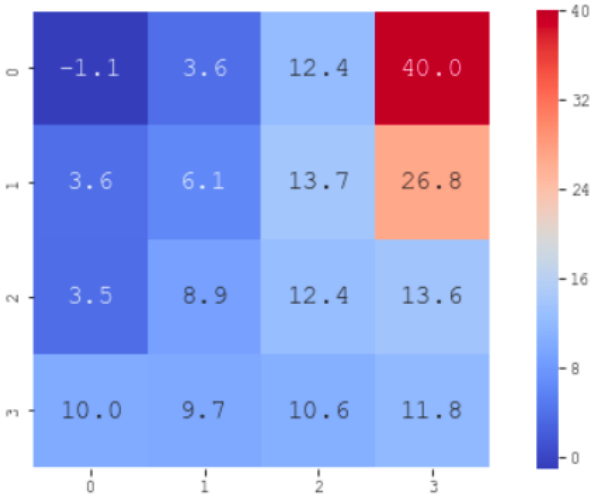
De doolhof is geïnitialiseerd in *maze_class.py*. Het is een 4x4 grid met 16 cellen.

np.array[[0, 1, 2, **3**]
[4, 5, **6**, **7**]
[8, 9, 10, 11]
[**12**,**13**,**14**,15]]

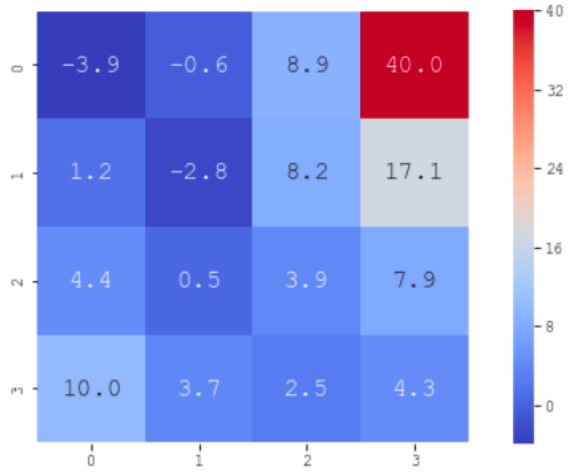
```
# TERMINAL STATES
self.grid[0][3] = 40
self.grid[3][0] = 10

# WATER STATES
self.grid[1][2] = -10
self.grid[1][3] = -10

# ENEMY STATE
self.grid[3][1] = -2
```

De afbeeldingen tonen de *estimated state value function* per discount factor 1 en 0.9

Y = 1



Y = 0.9

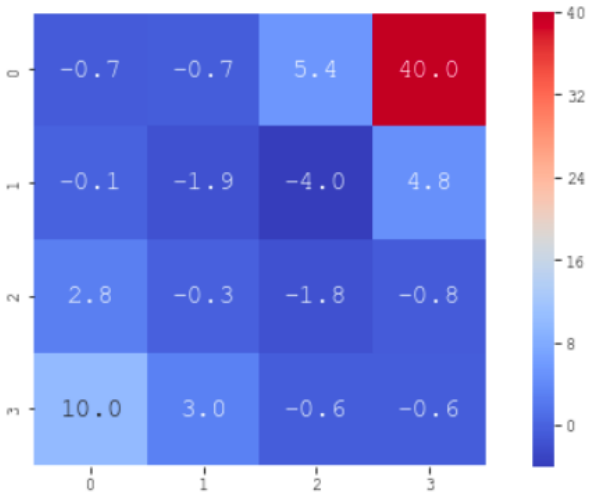


Zie GitHub voor verdere visualisatie: https://github.com/wolfsinem/as/blob/main/evaluation/visualize_evaluation.ipynb

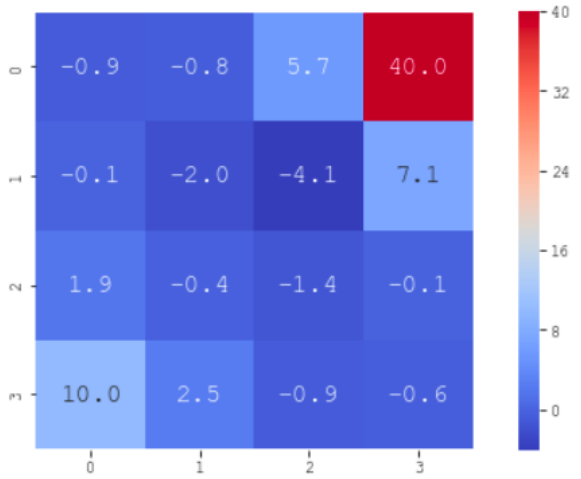
1.2. Temporal Difference Learning

Een nadeel aan MCPE is dat je mogelijk dood gaat voordat je erachter komt dat je bijna dood ging. Een methode die dit soort problemen kan voorkomen is Temporal Difference Learning. Oftewel TD-learning. **Voeg ook hier screenshots bij.**

Y = 1



Y = 0.9



2. Model-free control

Onze policy evaluation algoritmes baseren de value-function met behulp van wetenschap wat de volgende state S' is gegeven dat we actie a kiezen in huidige state S. Maar deze kennis hebben we helemaal niet! In plaats daarvan kunnen we beter met q-waardes gaan werken. **De Q-waarde is een mapping van states en actions naar utility.**

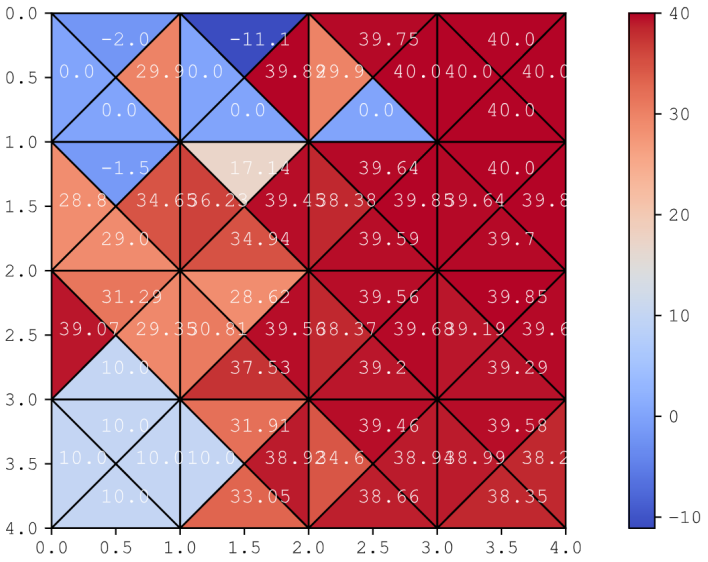
Het grootste verschil is dat we niet hoeven te weten wat de volgende state is. In plaats daarvan slaan we ergens op wat het de resultaten tot dusver waren toen we in state s, actie a namen en wat daarvan de waarde was.

Dan rest ons nog één probleem. Namelijk dat we niet policy iteration of value iteration kunnen toepassen met behulp van een greedy algoritme. Dit leidt namelijk tot **sub-optima**. In plaats daarvan gebruiken we ε-greedy. ε is een kleine waarde die representatief is voor de kans dat we exploreren, in plaats van exploiteren.

2.1. On-policy first-visit Monte-Carlo Control

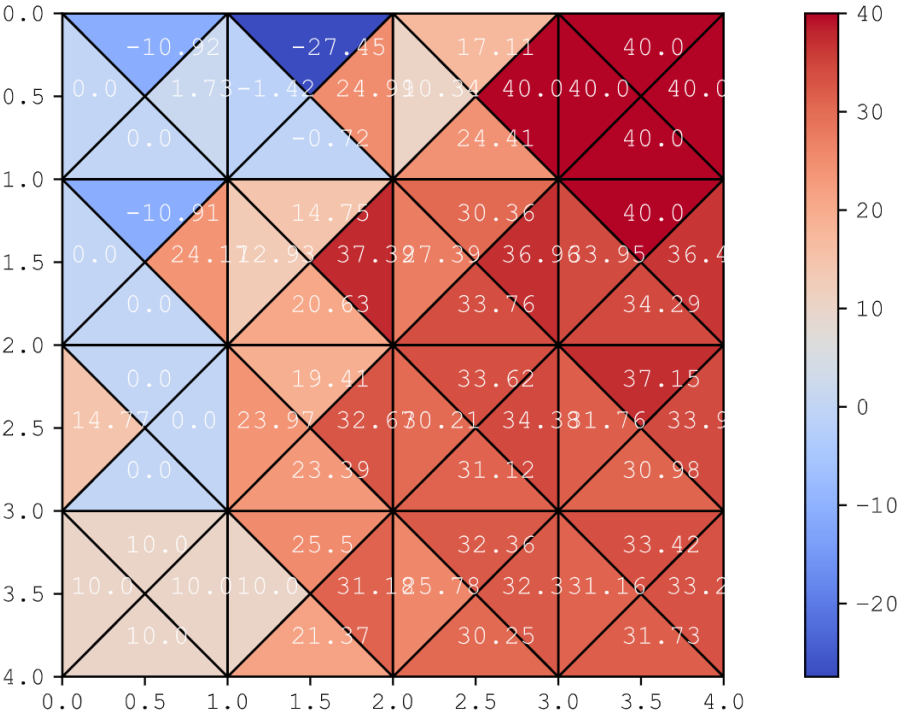
Voer nadat je deze hebt geïmplementeerd je functie weer op de doolhof uit met γ=0.9 en γ=1 en laat visueel zien wat de uiteindelijke policy is. De policy komt terecht in wat we een Q-table noemen. Dus print deze uit.

Y = 1



	0	1	2	3
0	29.9	39.8	40	40
1	34.6	39.4	39.8	40
2	39.0	39.5	39.6	39.8
3	10	38.9	39.4	39.5

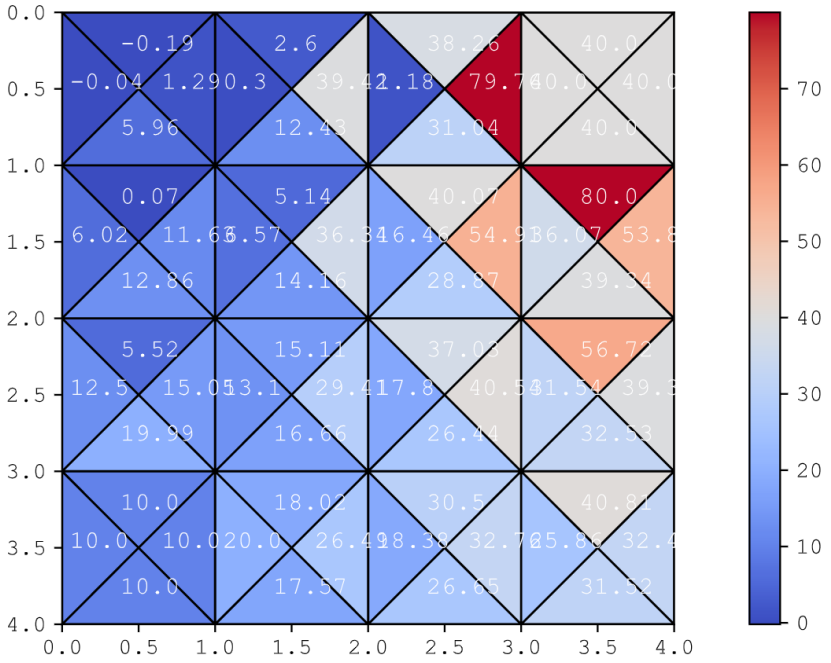
Y = 0.9


0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
-10.92
-27.45
17.11
40.0
0.0
1.73
-1.42
24.99
10.34
40.0
40.0
40.0
0.0
-0.72
24.41
40.0
-10.91
14.75
30.36
40.0
0.0
24.17
12.93
37.32
27.39
36.96
33.95
36.4
0.0
20.63
33.76
34.29
0.0
19.41
33.62
37.15
14.77
0.0
23.97
32.67
30.21
34.38
31.76
33.9
0.0
23.39
31.12
30.98
10.0
25.5
32.36
33.42
10.0
10.0
10.0
31.12
25.78
32.3
31.16
33.2
10.0
21.37
30.25
31.73
40
30
20
10
0
−10
−20

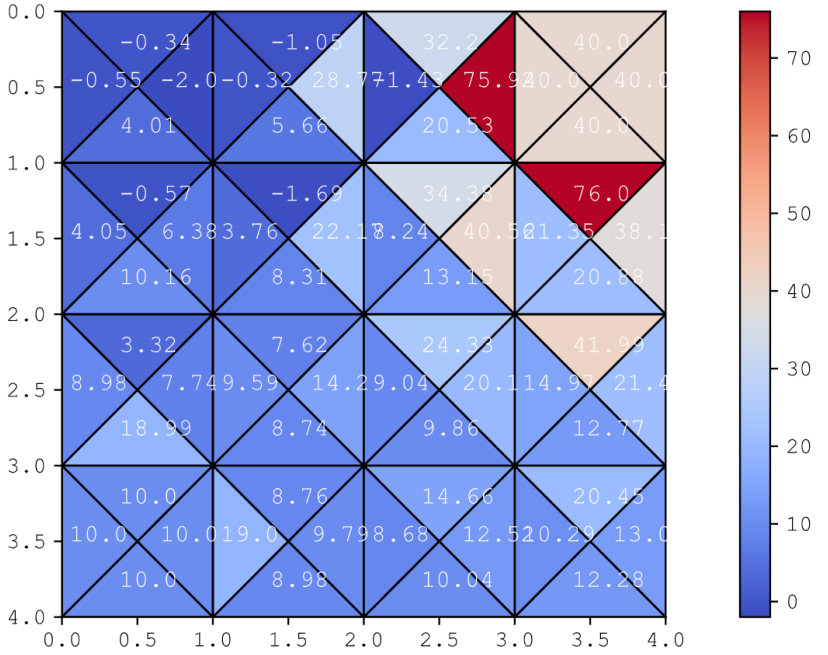
2.2. SARSA

Monte-Carlo Control is nogal inefficiënt, en we zitten dus weer met dat probleem dat we hele episodes moeten draaien voordat we iets van feedback hebben. De volgende stap is weer een Temporal-Difference-learning-techniek. En is het SARSA-algoritme.

Y = 1



Y = 0.9



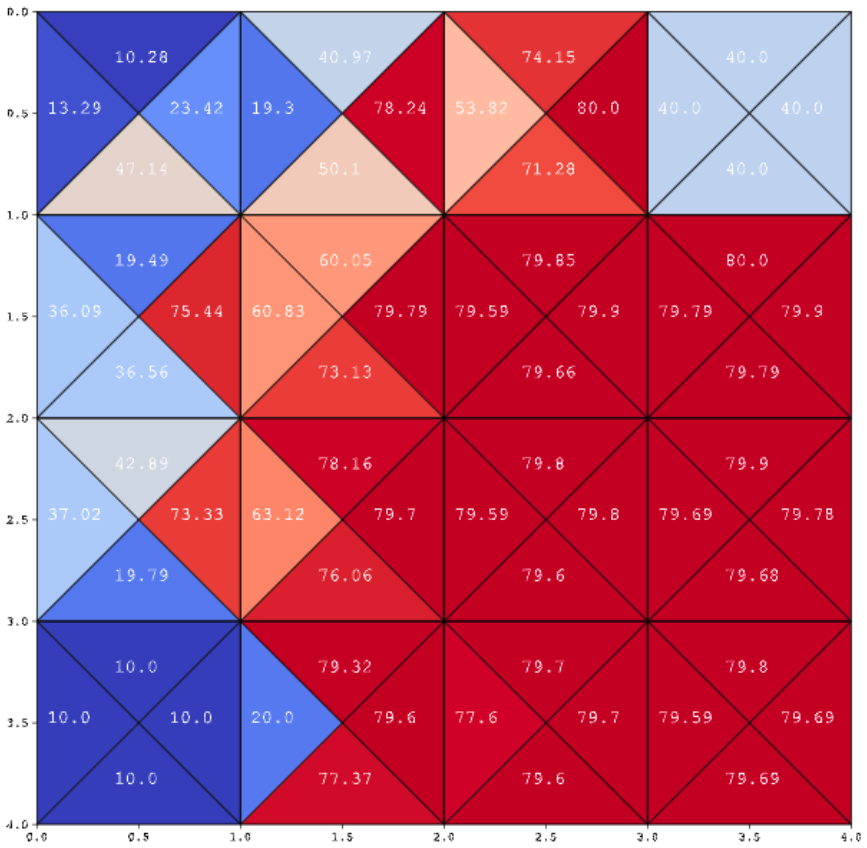
2.3. SARSAMAX a.k.a Q-Learning

Tot slot gaan we het meest bekende algoritme van reinforcement learning implementeren. De gateway naar echt goed werkende systemen voor echte problemen. Q-learning a.k.a. SARSAMAX.

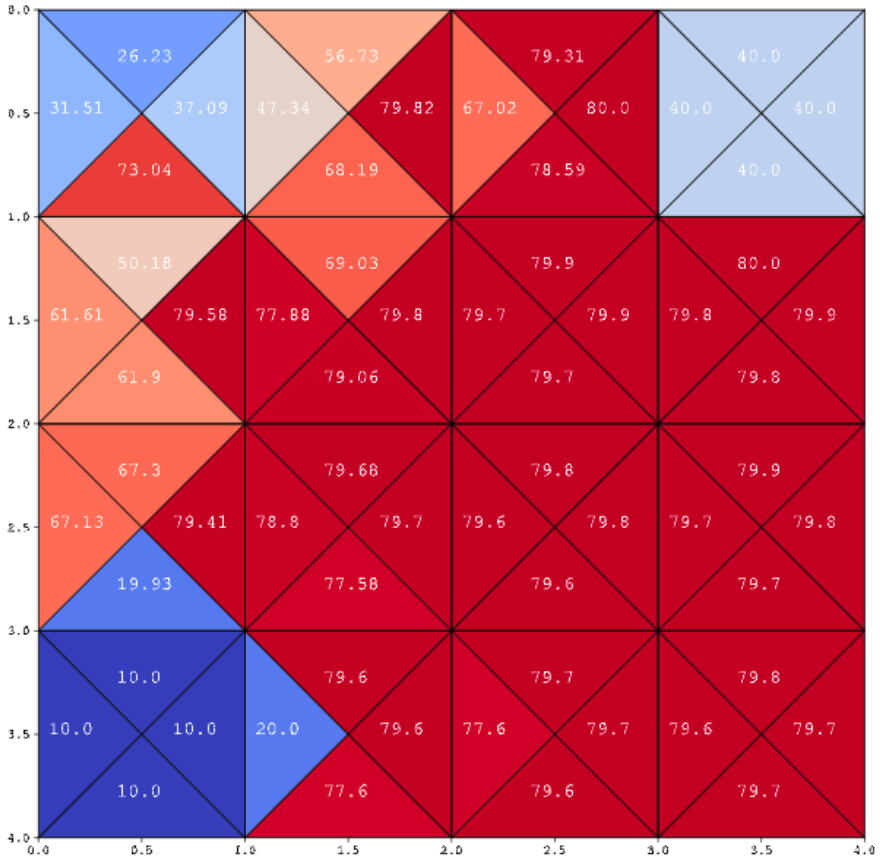
Het verschil met SARSA is klein, maar in de praktijk blijkt dit algoritme wel veel beter te werken. En het was voor meneer Watkins in 1989 relatief makkelijk om te bewijzen dat dit altijd convergeert naar de optimale policy. Helemaal mooi dus

Implementeer Qlearning - SARSAMAX for off-policy control en voer net als bij de vorige opdracht uit en laat visueel zien wat de uiteindelijke policy is.

Y = 1



Y = 0.9


0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
26.23
31.51
37.09
73.04
56.73
47.34
79.82
68.19
79.31
67.02
80.0
78.59
40.0
40.0
40.0
40.0
50.18
61.61
79.58
61.9
69.03
77.88
79.8
79.06
79.9
79.7
79.9
79.7
80.0
79.8
79.9
79.8
67.3
67.13
79.41
19.93
79.68
78.8
79.7
77.58
79.8
79.6
79.8
79.6
79.9
79.7
79.8
79.7
10.0
10.0
10.0
10.0
79.6
20.0
79.6
77.6
79.7
77.6
79.7
79.6
79.8
79.6
79.7
79.7