RL Assignment 1 – AI6101 Introduction ot AI & AI Ethics

Reinelle Jan C. Bugnot
G2304329L
October 9, 2023

1. Introduction

Reinforcement learning is a branch of machine learning that focuses on training intelligent agents to make sequential decisions in an environment. It relies on the concept of reward maximization, where agents learn to take actions that maximize their cumulative rewards through a process of trial and error, making it a fundamental paradigm in building autonomous systems and solving complex decision-making tasks [1]. In this assignment, I implemented a *Q-learning* algorithm to train an agent to push a box towards a goal position as efficiently as possible, while avoiding obstacles along the way. With the use of a simple epsilon decay exploration strategy on top of the standard Q-Learning algorithm, I was able to train the agent to find the optimal policy within 9,000 episodes.

2. Background Theory

Q-learning is a popular reinforcement learning algorithm that provides a framework for agents to learn optimal policies in an environment through trial and error. At its core, Q-learning is a model-free and value-based method that uses a Q-table to estimate the expected cumulative rewards an agent can obtain by taking specific actions in particular states. This algorithm is especially useful in scenarios where the agent has no prior knowledge about the environment and must learn from interactions [2].

The key idea behind Q-learning is to iteratively update the Q-values, denoted as *Q(s, a)*, where *s* represents a state, and *a* represents an action. The Q-values represent the expected return an agent will receive when starting in state *s*, taking action *a*, and then following the optimal policy thereafter. The algorithm aims to find the optimal Q-values for all state-action pairs, which ultimately leads to an optimal policy [2].

We can represent the Q-learning update step as follows [1]:

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha)Q(s_t, a_t) + \alpha(r_t + \gamma \max_a(s_{t+1}, a)) \tag{1}$$

where:

$Q(s_t, a_t)$ is the current estimate of the Q-value for state-action pair (s_t, a_t)

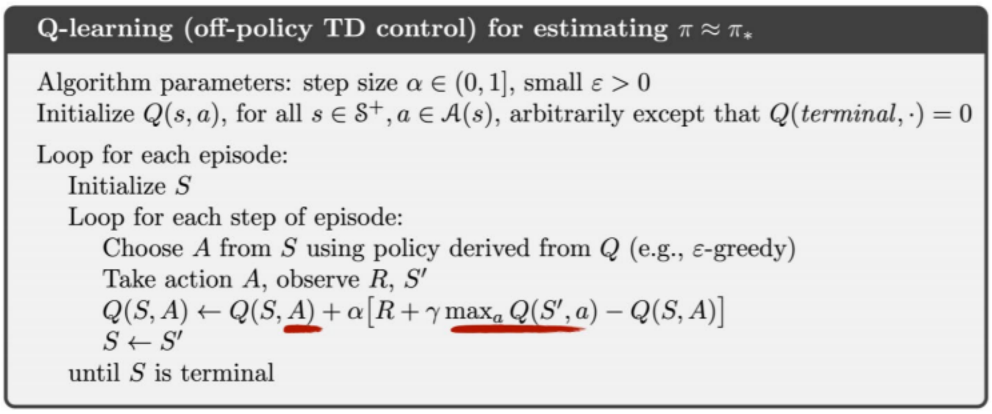
α is the learning rate which controls the step size of Q-value updates

r_t is the immediate reward obtained by taking action a_t in state s_t

γ is the discount factor that specifies the weight of future potential rewards

$\max_a Q(s_{t+1}, a)$ is the maximum Q-value among all possible actions in the next state s_{t+1}

To learn an optimal policy, the agent follows an exploration strategy, typically using an *epsilon-greedy strategy*. This means that with probability 1 – ε, where 0 ≤ ε ≤ 1, the agent selects the action with the highest Q-value (exploitation), and with probability ε, the agent explores a random action to discover new information about the current environment [1]. The Q-learning algorithm continues to interact with the environment, updating Q-values and refining its policy iteratively. Over time, the Q-values converge to their optimal values, representing the maximum expected return for each state-action pair. Once convergence is reached, the agent can deterministically find the optimal policy simply by selecting the action with the highest Q-value for each state. Using equation (1), this process can be implemented using the following algorithm, shown in Fig. 1.



Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

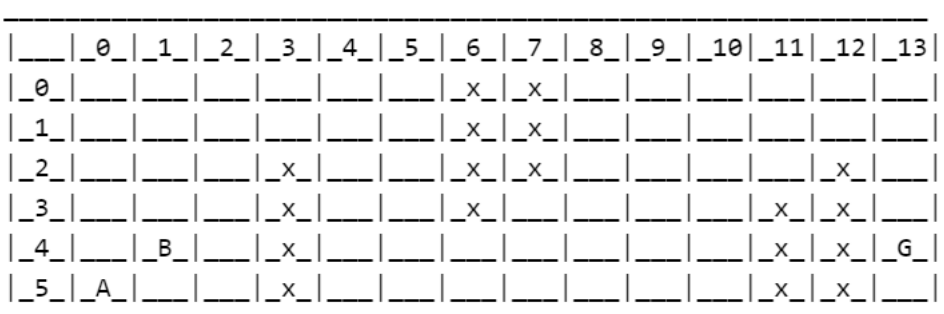
```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
        Take action $A$, observe $R$, $S'$
        $Q(S,A) \leftarrow Q(S,\underline{A}) + \alpha[R + \gamma \underline{\max_a Q(S',a)} - Q(S,A)]$
        $S \leftarrow S'$
    until $S$ is terminal
```

Fig. 1. Q-Learning Algorithm [1].

3. CliffBoxPushing grid-world environment

The environment used for this assignment is shown in Fig. 2. The size of the environment is 6x14. In the original text-based implementation, '*A*' represents the agent, '*B*' represents the box, '*G*' the goal, and '*X*' for the cliff. For each timestep in a given episode, the agent takes a step in one of the four directions (up, down, left, right). The game ends based under three conditions [3]:

1. The agent or the box steps into the dangerous region (cliff).
2. The current time step attains the maximum time step of the game.
3. The box arrives at the goal.



```
___________________________________________________________
|___|_0_|_1_|_2_|_3_|_4_|_5_|_6_|_7_|_8_|_9_|_10|_11|_12|_13|
|_0_|___|___|___|___|___|___|_x_|_x_|___|___|___|___|___|___|
|_1_|___|___|___|___|___|___|_x_|_x_|___|___|___|___|___|___|
|_2_|___|___|___|_x_|___|___|_x_|_x_|___|___|___|___|_x_|___|
|_3_|___|___|___|_x_|___|___|_x_|___|___|___|___|_x_|_x_|___|
|_4_|___|_B_|___|_x_|___|___|___|___|___|___|___|_x_|_x_|_G_|
|_5_|_A_|___|___|_x_|___|___|___|___|___|___|___|_x_|_x_|___|
```

Fig. 2. The original text-based implementation of CliffBoxGridWorld.

In order to enhance visual clarity (and also just for fun), I wrote an *AnimatedWorld()* class layer on top of the original base environment class using the python Pygame package, as shown in Fig. 3. This allows me to map the original elements from the base environment to colored tiles (or, optionally, custom sprites), and then animate subsequent frame renders on a separate window, instead of printing each frame consecutively on the cell's output block.

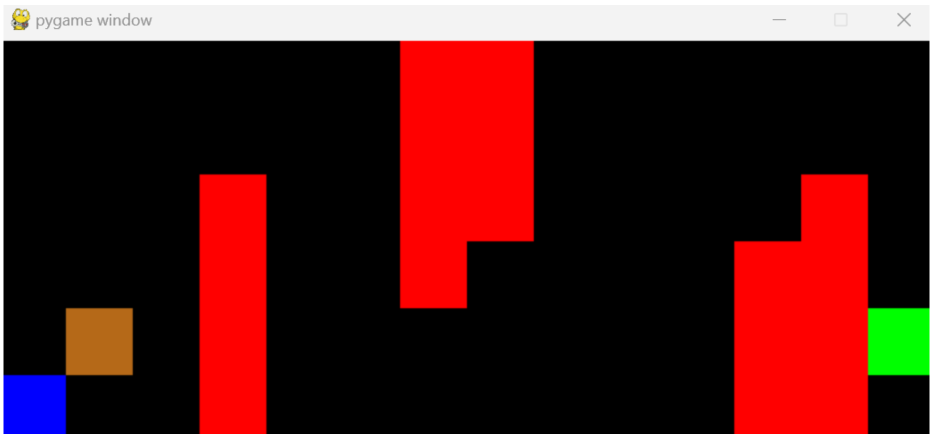


Fig. 3. My Pygame implementation of the RL environment.

4. Training the Agent

The Q-learning algorithm shown in Fig. 1 is implemented by adding the corresponding code to the provided *learn* method of the *RLAgent* class. By calling the *learn()* method of our instantiated *RLAgent()* class, we are effectively able to train our agent to interact and navigate the environment, while guided by the default reward function, shown in Table 1, and the epsilon-greedy strategy. To track the learning progress, I recorded the reward accumulated by the agent per episode. Due to the inherent randomness in RL training strategies and the execution of 10,000 episodes, the reward vs. episode plot is expected to exhibit significant noise, making it challenging to interpret without any post-processing. To address this issue, it is crucial to apply a smoothing function to the time-series data. In this assignment, I've opted for a quadratic Savitzky-Golay (Savgol) filter due to its superior signal-to-noise ratio compared to other filters.

TABLE 1. Default Reward Function

Event (for each timestep)	**Reward Value**
constant reward	-1
current distance between box and goal	$-d_{b,g}$
current distance between agent and box	$-d_{a,b}$
if box or agent enters a cliff tile	-1000
if box reaches the goal position	+1000

However, simply using the default reward parameters and/or epsilon-greedy strategy does not yield promising results, even with a bit of hyperparameter-tuning, as shown in Fig. 4. Although we are able to observe an increase in the overall reward received by the agent for each episode towards the latter half of the training process, the agent was never able to push the box towards the goal position, which should give a reward of +1000, as evident in the learned policy in Fig. 5.

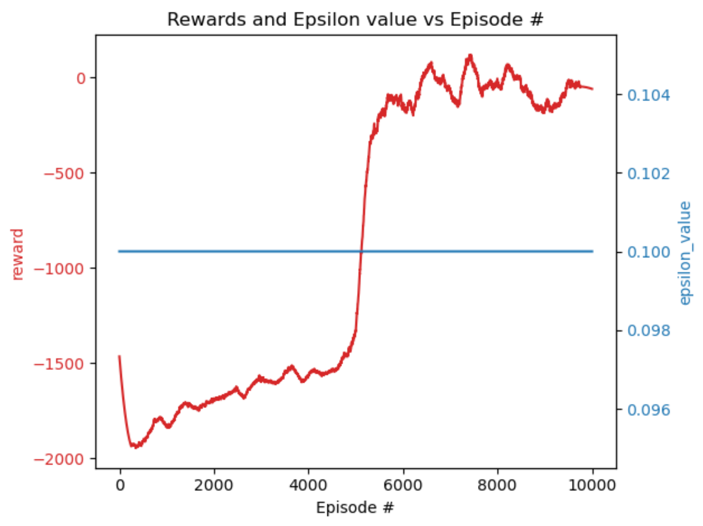


Fig. 4. Learning Progress for Q-learning with default epsilon-greedy strategy.

5. The Epsilon-Decay Strategy

There are several options to tune the Q-learning model for better performance. Before deciding on my approach, I made slight modifications to the code to ensure that key quality parameters for each time step, such as the chosen action, current state, and Q-values, are explicitly recorded during testing. This allows me to monitor and gain insights into the underlying calculations, ensuring that the agent is executing the anticipated actions and that the Q-value updates for specific time steps align with expectations. This decision proved especially useful when observing the learned policy in detail, such as the one shown in Appendix A.

I then chose to implement an *epsilon-decay* exploration strategy. Simply put, instead of having a constant epsilon value across the entire training process, as shown in Fig. 4, the epsilon decay strategy starts with a relatively high value of epsilon, and as the training progresses, the epsilon value linearly decays by a specified rate down to a specified minimum value. The logic behind this strategy is to encourage the agent to initially focus on exploring and gaining an understanding of the environment through random actions, and then progressively shift towards exploiting the learned knowledge as training progresses [4]. The corresponding code for this is integrated within the *learn()* method of the *RLAgent()* class.

Q-Learning with Epsilon Decay Pseudocode

```
Initialize Q-table
Set epsilon to a starting value (e.g., 1.0)
Set epsilon_decay_end_factor as a value within (0,1] (e.g., 0.80)
Set min_epsilon to a small value (e.g., 0)
Calculate epsilon_decay_end as the episode # where epsilon stops decaying
Calculate epsilon_decay_rate as the ratio of (epsilon – min_epsilon) and epsilon_decay_end

Loop for each episode:
  Set the initial state
    Loop for each step of episode:
      Choose an action using epsilon-greedy policy based on the Q-table
```

```
        Take the chosen action and observe the reward and next state
        Update the Q-table using the Q-learning update equation
        Set the current state to the next state
        If current episode # is less than epsilon_decay_end,
            then, subtract epsilon_decay_rate from epsilon
    until a terminal state is reached
```

6. Results

Fig. 5 shows the learning progress of the model when the epsilon decay strategy is implemented. In contrast to the standard epsilon-greedy strategy with a uniform epsilon value in Fig. 4, the cumulative rewards obtained by the agent in a Q-learning model using an epsilon-decay strategy tend to *initially* exhibit irregular fluctuations of accumulated rewards per episode—often without an apparent trend towards convergence. This occurs because, with a high initial epsilon value, the agent engages in extensive exploration through random actions, rather than adhering to a predefined policy. These exploratory actions will usually not align with the optimal path to the agent's goal. Regardless, each step taken in this manner provides the agent with an opportunity to update its Q-table with the best possible action for each randomly explored state, facilitating the learning of the environment.

Hence, around the latter part of the training process, when the *epsilon* value has substantially decayed, agent starts to more consistently exploit its learned knowledge by choosing actions with higher Q-values. Due to the extensive knowledge acquired from earlier explorations, each update to the Q-values during the exploitation stage carries more significant impact to the goal via the reward function. In this assignment, I configured my epsilon to decay to 0 by around episode 8,000. Thus, from thereon, the agent will always greedily select the learned optimal action for each state. This strategy, as shown in the figure, is enough to eventually (or rather, suddenly) lead towards convergence.

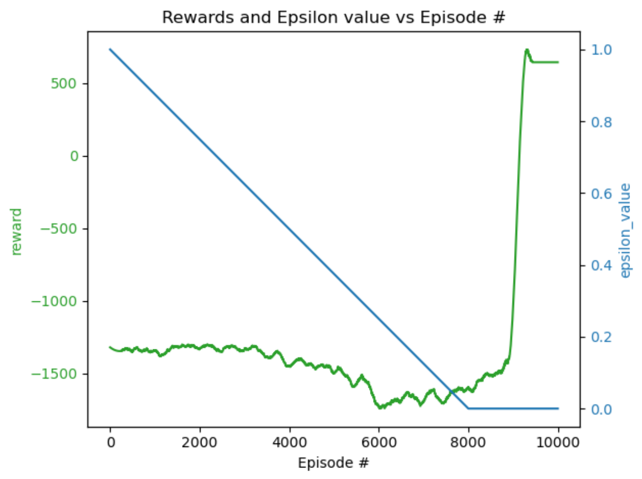


Fig. 5. Learning Progress. Epsilon Decay mechanism was implemented that linearly reduces the epsilon value as training progresses.

After training a Q-learning model, the resulting Q-table captures the learned action-value estimates for an agent in all states in the environment. The learned policy can then be derived by selecting and performing actions with the highest Q-values at each subsequent state starting from the initial state. This policy deterministically defines the agent's optimal strategy for decision-making in the environment; i.e. to maximize the reward earned. The learned policy for my trained agent can be seen in Fig. 6, represented by the sequence of yellow arrows. The complete output parameters of the learned policy (including the sequence of actions and the learned Q-values for the state-action pairs) can be viewed in Appendix A.

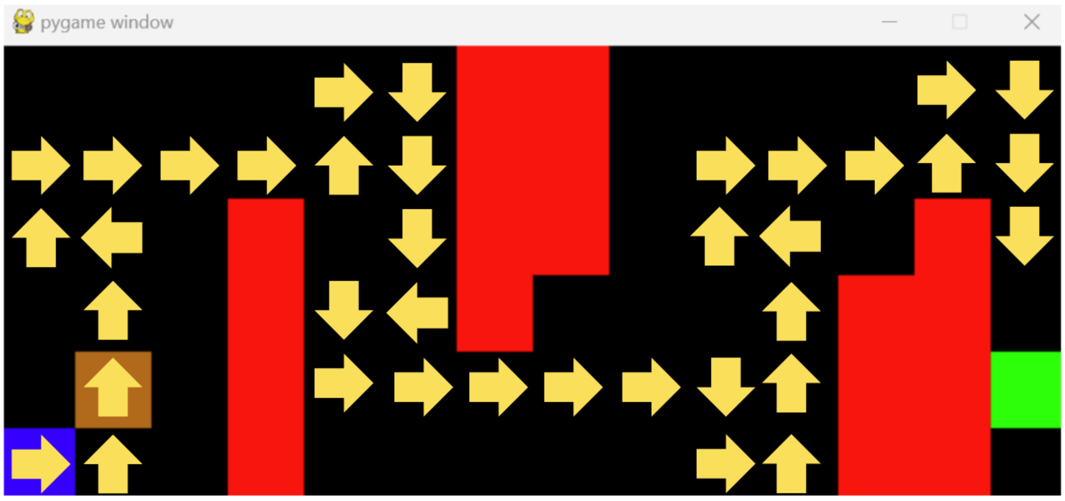
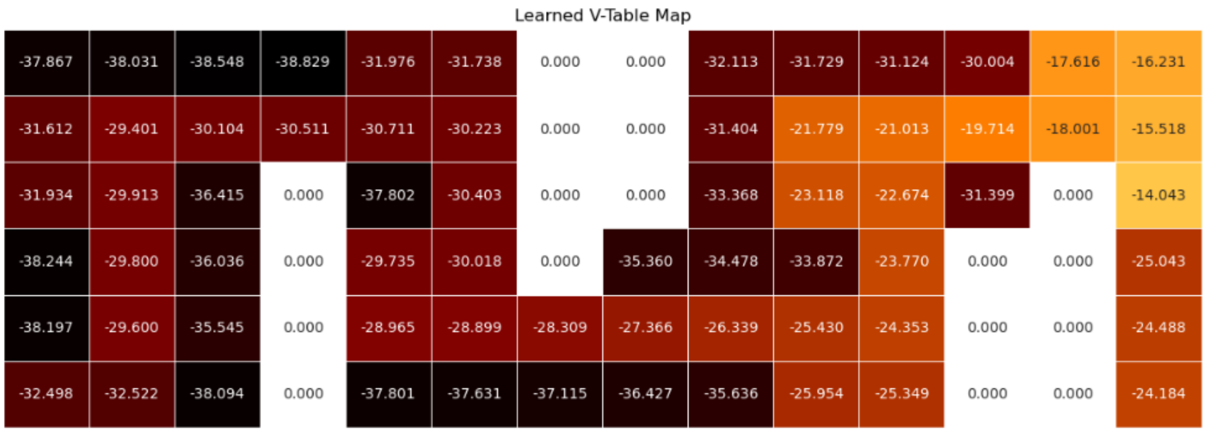


Fig. 6. Learned Policy

Typically, in standard Q-learning algorithms, V-tables are not necessary to calculate since Q-learning primarily deals with Q-tables that represent the quality values assigned for each action-value pair. However, if we want to consider a V-table in the context of an agent pushing a box to a goal position, then the table would represent the value associated with each possible agent position. This 'value' is the aggregate Q-values learned by the agent during training. In my case, I considered the average Q-value corresponding to the best action (i.e., the max Q-value for state *s*) that the agent can take for any given agent position, for all possible box positions. This yields the V-table shown in Fig. 7.



-37.867	-38.031	-38.548	-38.829	-31.976	-31.738	0.000	0.000	-32.113	-31.729	-31.124	-30.004	-17.616	-16.231
-31.612	-29.401	-30.104	-30.511	-30.711	-30.223	0.000	0.000	-31.404	-21.779	-21.013	-19.714	-18.001	-15.518
-31.934	-29.913	-36.415	0.000	-37.802	-30.403	0.000	0.000	-33.368	-23.118	-22.674	-31.399	0.000	-14.043
-38.244	-29.800	-36.036	0.000	-29.735	-30.018	0.000	-35.360	-34.478	-33.872	-23.770	0.000	0.000	-25.043
-38.197	-29.600	-35.545	0.000	-28.965	-28.899	-28.309	-27.366	-26.339	-25.430	-24.353	0.000	0.000	-24.488
-32.498	-32.522	-38.094	0.000	-37.801	-37.631	-37.115	-36.427	-35.636	-25.954	-25.349	0.000	0.000	-24.184

Fig. 7. Learned V-Table

From the figure above, we can derive the following observations:

1. Goal Position: The highest non-zero value in the V-table corresponds to the agent position that contains the action that will take the box to the goal position (in this case, precisely two steps above the goal position). This signifies that the best action from this specific state yields, on average, the highest expected cumulative reward.

2. Cliff Positions: From the V-table, we can see that the calculations generated 0 values for all the tiles corresponding to cliff positions. This is primarily because the game terminates upon entering this area, and therefore inhibits any Q-values from being calculated while on the cliff positions during training. Consequently, these states remain at their zero-initialized Q-values and are never updated.

3. Non-Cliff / Non-Goal Positions: For the rest of the positions, we can observe that the V-table values that align with the learned policy (sequence of arrows) in Fig. 6 have a higher calculated value that those that don't—the value of which increases the closer the position is to the goal position, indicating that they are desirable positions that lead to positive rewards.

7. Epsilon-Decay vs the Default Epsilon-Greedy strategy

In this assignment, I demonstrated the use of a simple epsilon-decay mechanism to train an agent to push the box to the goal position and compared its results (Fig. 5) with the standard non-exploration strategy (Fig. 4). The logic behind this technique is simple: instead of assigning a fixed epsilon value, we begin training with a relatively high epsilon value and decay it over time [4]. This dynamic adjustment of the epsilon value allows us to influence the behavior of the agent over time. That is, at the start of training, we encourage the agent to focus on exploring the environment and discover optimal policies through random actions. Only then when our agent has learned enough context do we slowly shift towards encourage the agent to exploit its learned knowledge to try to fully maximize the reward that it earns.

The main advantage of this strategy as compared to the default (epsilon-greedy) non-exploration strategy is that, depending on the epsilon parameter you select at the beginning of training, the agent might prematurely favor exploitation at the start, which limits the agent's ability to fully explore the state space, potentially missing out on critical learning opportunities (e.g. knowing that there is a state-action pair just up ahead that gives +1000 reward). This is usually fine for relatively simple tasks where the low probability of exploration can still provide decent coverage of the available state space. But in cases where there is considerable complexity in the task (i.e., agent must push the box and therefore be near it) and in the state space (i.e. nearby cliffs that punish the agent with a steep negative reward), a standard non-exploration strategy might struggle more in converging compared to an epsilon-decay strategy. For instance, in this assignment, since the agent receives a negative reward value corresponding to the distance between the agent and the box for each timestep (see Table I), then a non-exploration strategy can sometimes yield a policy of constantly alternating between known *safe* states beside the box.

~ *** ~

References

[1] B. An, “Reinforcement Learning”, Lecture Slides, AI6101 Introduction to AI and AI Ethics, Nanyang Technological University, Singapore, Sem 1, 2023-2024.

[2] C. Shyalika, “A beginners guide to Q-learning,” Medium, https://towardsdatascience.com/a-beginners-guide-to-q-learning-c3e2a30a653c (accessed Oct. 20, 2023).

[3] B. An, “Reinforcement Learning Assignment”, Assignment Instructions file, AI6101 Introduction to AI and AI Ethics, Nanyang Technological University, Singapore, Sem 1, 2023-2024.

[4] V. Kumar and M. Webster, “Importance Sampling based Exploration in Q Learning”, arXiv:2107.00602v1 [math.OC], Jul 2021

Appendix A.
Learned Policy (detailed)

```
LEARNED POLICY:
Time Step: 0, Action: 4 - right
               ((5, 0, 4, 1)) : {1: -267.9266635034692, 2: -269.6732926351668, 3: -2
68.7109040783736, 4: 205.2094572766309}
Time Step: 1, Action: 1 - up
               ((5, 1, 4, 1)) : {1: 223.68311967003152, 2: -259.7048930377623, 3: -2
62.4168924127608, 4: -259.34483016167025}
Time Step: 2, Action: 1 - up
               ((4, 1, 3, 1)) : {1: 243.55420374493008, 2: -250.23752724797959, 3: -
249.18164797819145, 4: -249.81478120038003}
Time Step: 3, Action: 1 - up
               ((3, 1, 2, 1)) : {1: 264.85122831115314, 2: -239.4154515365476, 3: -2
38.9078728539012, 4: -238.87546452793526}
Time Step: 4, Action: 3 - left
               ((2, 1, 1, 1)) : {1: -226.5709912831057, 2: -225.90476056406047, 3: 2
87.6032941950543, 4: -225.87183516995833}
Time Step: 5, Action: 1 - up
               ((2, 0, 1, 1)) : {1: 311.84009611740237, 2: -213.7441326730475, 3: -2
12.86451519048654, 4: -214.9647152967059}
Time Step: 6, Action: 4 - right
               ((1, 0, 1, 1)) : {1: -200.22714344600524, 2: -199.29162778742074, 3:
-199.07791422022473, 4: 335.5511184871454}
Time Step: 7, Action: 4 - right
               ((1, 1, 1, 2)) : {1: -187.13879780746217, 2: -190.51191688501578, 3:
-188.73202191490586, 4: 358.7256311093321}
Time Step: 8, Action: 4 - right
               ((1, 2, 1, 3)) : {1: -175.94253654892637, 2: -175.21998964041055, 3:
-175.45505088495943, 4: 381.3526848054411}
Time Step: 9, Action: 4 - right
               ((1, 3, 1, 4)) : {1: -164.51363309472674, 2: -1014.9999999999659, 3:
-164.7252694400314, 4: 403.4211069443276}
Time Step: 10, Action: 1 - up
               ((1, 4, 1, 5)) : {1: 424.91949688196684, 2: -155.31514373677294, 3: -
154.7559282462204, 4: -1011.9999454330681}
Time Step: 11, Action: 4 - right
               ((0, 4, 1, 5)) : {1: -144.54223625370523, 2: -144.6509253477101, 3: -
144.81490807719752, 4: 447.87703763466}
Time Step: 12, Action: 2 - down
               ((0, 5, 1, 5)) : {1: -133.63525904479206, 2: 470.2826914639388, 3: -1
35.96888951286547, 4: -1013.9990054716941}
Time Step: 13, Action: 2 - down
               ((1, 5, 2, 5)) : {1: -124.67120958336903, 2: 492.12519537136615, 3: -
123.66387735157682, 4: -1012.1974187137308}
Time Step: 14, Action: 2 - down
               ((2, 5, 3, 5)) : {1: -115.73799329135196, 2: 513.3930565013941, 3: -1
15.59712887978927, 4: -1010.4340058503042}
Time Step: 15, Action: 3 - left
               ((3, 5, 4, 5)) : {1: -107.87384954607302, 2: -108.34483120578366, 3:
534.0745474504024, 4: -1008.5555519927397}
Time Step: 16, Action: 2 - down
               ((3, 4, 4, 5)) : {1: -98.7370663142527, 2: 556.1985178065333, 3: -100
2.6659474987031, 4: -99.71414670541397}
Time Step: 17, Action: 4 - right
               ((4, 4, 4, 5)) : {1: -91.67881259399354, 2: -90.17828085574615, 3: -9
88.2343038836426, 4: 577.7535895985037}
Time Step: 18, Action: 4 - right
               ((4, 5, 4, 6)) : {1: -83.42319912413485, 2: -83.20854022767057, 3: -8
3.79872957624669, 4: 598.7281526515344}
Time Step: 19, Action: 4 - right
               ((4, 6, 4, 7)) : {1: -839.7178905600001, 2: -76.71217033830472, 3: -7
6.62422084265566, 4: 619.1103598485046}
Time Step: 20, Action: 4 - right
```

```
              ((4, 7, 4, 8)) : {1: -71.50921804537407, 2: -70.9179680145481, 3: -71
.42431755636774, 4: 638.8881222943926}
Time Step: 21, Action: 4 - right
              ((4, 8, 4, 9)) : {1: -66.40321550050663, 2: -66.42408595466127, 3: -6
6.88692829506986, 4: 658.0491043820335}
Time Step: 22, Action: 2 - down
              ((4, 9, 4, 10)) : {1: -62.99956521964689, 2: 676.5807187571771, 3: -6
3.00474014470285, 4: -361.44000000000005}
Time Step: 23, Action: 4 - right
              ((5, 9, 4, 10)) : {1: -59.5849167946506, 2: -58.270706705979954, 3: -
59.14489885944323, 4: 696.5109375073239}
Time Step: 24, Action: 1 - up
              ((5, 10, 4, 10)) : {1: 715.8274872523714, 2: -55.0070135069076, 3: -5
5.73667568824484, 4: -201.20000000000002}
Time Step: 25, Action: 1 - up
              ((4, 10, 3, 10)) : {1: 736.5586604616037, 2: -50.073702592030386, 3:
-50.015069432774354, 4: -491.41600000000005}
Time Step: 26, Action: 1 - up
              ((3, 10, 2, 10)) : {1: 758.7333270016363, 2: -44.53515678377569, 3: -
44.50531344010467, 4: -201.60000000000002}
Time Step: 27, Action: 3 - left
              ((2, 10, 1, 10)) : {1: -38.137606115009554, 2: -38.088883292449744, 3
: 782.3809459200373, 4: -37.44491603417001}
Time Step: 28, Action: 1 - up
              ((2, 9, 1, 10)) : {1: 807.5315774694258, 2: -31.128198865809757, 3: -
30.238061877495902, 4: -31.26051262592683}
Time Step: 29, Action: 4 - right
              ((1, 9, 1, 10)) : {1: -24.49957258737314, 2: -23.321651660649845, 3:
-24.286542177536443, 4: 832.1750790504343}
Time Step: 30, Action: 4 - right
              ((1, 10, 1, 11)) : {1: -17.548151694431297, 2: -17.75061862292354, 3:
-17.821753997414266, 4: 856.3011010718719}
Time Step: 31, Action: 4 - right
              ((1, 11, 1, 12)) : {1: -12.393010554873186, 2: -13.042190961791388, 3
: -12.696823291221138, 4: 879.8990827263998}
Time Step: 32, Action: 1 - up
              ((1, 12, 1, 13)) : {1: 902.9582476799999, 2: -201.20000000000002, 3:
-9.372593453839444, 4: -9.821906624297451}
Time Step: 33, Action: 4 - right
              ((0, 12, 1, 13)) : {1: -5.952191616307201, 2: -6.665705619762013, 3:
-6.279519142400002, 4: 927.5084160000001}
Time Step: 34, Action: 2 - down
              ((0, 13, 1, 13)) : {1: -2.988016, 2: 951.5392000000003, 3: -4.2583062
37747201, 4: -2.988016}
Time Step: 35, Action: 2 - down
              ((1, 13, 2, 13)) : {1: -1.8, 2: 975.0400000000002, 3: -1.8, 4: -1.596
8}
Time Step: 36, Action: 2 - down
              ((2, 13, 3, 13)) : {1: -0.8, 2: 998.0000000000001, 3: -200.8, 4: -0.6
000000000000001}

rewards: [-14, -15, -16, -17, -18, -17, -16, -15, -14, -13, -14, -13, -12, -11, -10
, -11, -10, -9, -8, -7, -6, -5, -6, -5, -6, -7, -8, -9, -8, -7, -6, -5, -6, -5, -4,
-3, 998]
cumulative rewards: 642
action history: [4, 1, 1, 1, 3, 1, 4, 4, 4, 4, 1, 4, 2, 2, 2, 3, 2, 4, 4, 4, 4, 4,
2, 4, 1, 1, 1, 3, 1, 4, 4, 4, 1, 4, 2, 2, 2]
```