Catholic University Eichstätt-Ingolstadt
Faculty of Business and Economics
Assistant Professorship of Operations Management
Prof. Dr. Pirmin Fontaine, Daniel Müllerklein

Reinforcement Learning - Supply Chain Ordering Management: An application to the beer game

– Project work –

submitted by

Dennis Götz

Vincent Bläske

Paul Posselt

Serkan Akdemir

Table of contents

Table of contents I

List of figures II

1 Introduction and structure of the paper 3

2 Problem description 3

3 Data generation 5

4 Moving to DQN 6

5 Discussion and results 7

Bibliography V

List of figures

Figure 1: Supply chain model 3
Figure 2: RLOM and benchmarks means of main and test sets 1-3 8
Figure 3: Q-value decrease per episode 8

1 Introduction and structure of the paper

In order to illustrate the ideas of dynamic systems, Jay Forrester of MIT first created the iconic Beer Distribution Game in the late 1950s. In this instance, the dynamic system is a supply chain that transports beer from a brewery to the final customer. The supply chain structure and game rules are straightforward, but the ensuing behavior is highly complex, which is what makes the game so exciting.

A way to creating simulation models of real-world systems is agent-based modeling. Agents that are used in agent-based modeling capture entities and exhibit behavior in response to internal or external events. In the beer distribution game, the players will be represented by agents. Instead of teaching the agents a set of predefined behaviors, the agents will be trained to play the beer distribution game using machine learning techniques, particularly a technique known as Reinforcement Learning. Reinforcement learning emphasizes interaction-based learning. The agent must experiment with different actions to determine which ones have the best results rather than being instructed which to do. One approach to reinforcement learning is Q-learning.

In this paper, the goal is to implement a Q-learning algorithm to deal with inventory management challenges such as changing demands and/ or disruptions.

The structure of the essay is as follows: The reinforcement learning problem is briefly described in section 2. The generated data to test and train the algorithm is shown in section 3. In section 4, the Q-learning algorithm will be implemented and adjusted to account for Deep Q-networks. In section 5, the findings on model performance after testing different parameters will be summarized and it will be discussed, on how well reinforcement learning suits the applied problem setting.

2 Problem description

The problem at hand is a minimization problem. The goal is to keep costs to a minimum within a defined supply chain. In our case, the supply chain consists of a manufacturer a distributor and a retailer, from whom the customer orders.

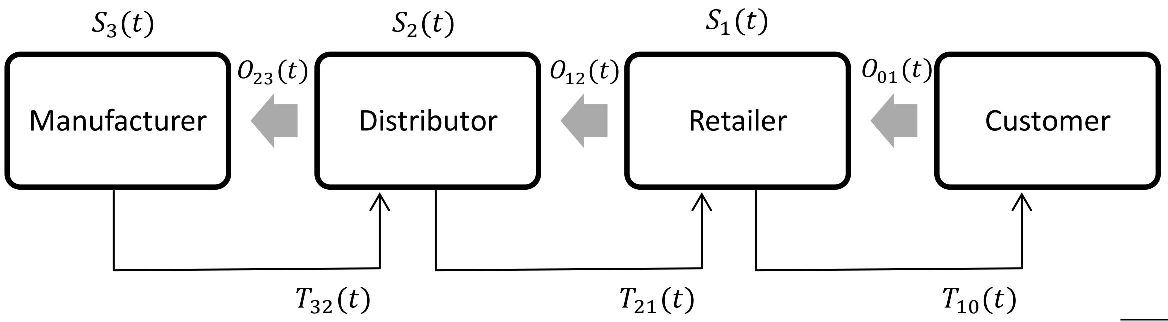


Figure 1: Supply chain model.

Within the supply chain, each actor (manufacturer, distributor, etc.) has an agent that can trigger an order at the upstream actor. Each of these agents tries to minimize not only the costs incurred by itself, but the total sum of all costs incurred by the actors in the supply chain. Costs can arise both from the stockpiling of products and from unsatisfied demand. In every level of the supply chain at each time step, four events happen:

Order size received from the downstream agent.

The received order is fulfilled from on-hand inventory (if possible).

The Agent decides about placing order for stock replenishment.

Previous orders are received from the upstream actor and states are updated.

The supply chain represents the system, which can assume different states. These states result from the inventory position of the individual actors at a specified time and each state is represented by a vector with these values as elements. Since the individual elements of the vector can theoretically assume arbitrarily large or small values, the values which represent the inventory position need to be transformed into a coding first to generate a limited number of possible states. This results in n over k (binomial coefficient) different possible states, where n represents the number of different coded states and k represents the number of different actors within the supply chain. The initial state for the system is determined by a high and low variant where each actor has either an inventory position of 1 or 10 at the beginning, each with a 50% probability.

In this problem, the agent has the possibility to react to the environmental state of the supply chain by placing an order with the upstream actor. The ordered quantity is represented by two different variables. The variable X represents the demand of the downstream actor. If, for example, the customer places an order with the retailer for 10 units, the variable X for the retailer takes the value 10. In turn, the variable Y indicates the deviation of the own order at the upstream actor from the order received from the downstream actor. This means that if the value of Y is -1, the agent will order one less unit from the upstream actor than was requested from it by the downstream actor. With a value of +1, the agent orders one unit more, and with a value of 0, it orders exactly as many units as are ordered from it by its downstream actor. The agents can now set the values for Y as an action. These values for Y can theoretically be between minus X and plus infinity. Simulations can be used to find out to which range the values for Y can be fixed without influencing the result in any essential way.

The transition probabilities are based on the Q-values, whereby, since this is a minimization problem, the action that promises the lowest Q-value is always chosen. At the beginning, the probabilities for each action are the same. By estimating the Q-values, new values are gradually learned for the different actions, resulting in different transition probabilities. In this learning phase, the agent selects with a certain probability the action that promises the lowest Q-value (exploitation) and with a certain probability a randomly chosen action (exploration). The initial probabilities are 0.98 for exploration and 0.02 for exploitation. They always add up to 1. The agent therefore is more likely to explore first because his lack of knowledge about the environment. The relationship between exploitation and exploration shifts during the learning process, with the probability of exploitation increasing linearly and the probability of exploration decreasing linearly. The Q-values are updated with the following formula:

$$Q(s,a) = (1-\alpha) * Q(s,a) + \alpha * [-r(t+1) + max_{a'}Q(s',a')]$$

Alpha indicates the learning rate and influences how much weight is attached to the most recently experienced reward. In our example, alpha has a value of 0.17.

After completing the learning process, the agent applies a greedy search and according to the systems state always chooses the action that grants him the lowest Q-value.

The reward the agent receives from the environment is calculated using a reward function called r(t). In this function, both the inventory position and the backorder are taken into account. These two values are weighted with a cost factor each, alpha and beta, and summed up. It is important here that the costs for all actors involved in the supply chain are calculated and then completely aggregated. This results in an agent wanting to minimize not only the costs incurred within their own operations but also the costs of all parties involved in the supply chain. The two cost factors alpha and beta can be individually chosen here in such a way that they correspond to the costs incurred in the real system.

$$r(t) = \sum_{i=1}^{k} [h_i(t) + 2 * C_i(t)]$$

3 Data generation

For the first main test problem, we use the customer demands from Kimbrough et al.:
[15,10,8,14,9,3,13,2,13,11,3,4,6,11,15,12,15,4,12,3,13,10,15,15,3,11,1,13,10,10,0,0,8,0,14]
For test problems 1-3, we use the data generated by Chaharsooghi et al.:
(1) [5,14,14,13,2,9,5,9,14,14,12,7,5,1,13,3,12,4,0,15,11,10,6,0,6,6,5,11,8,4,4,12,13,8,12]
(2) [15,10,8,14,9,3,13,2,13,11,3,4,6,11,15,12,15,4,12,3,13,10,15,15,3,11,1,13,10,10,0,0,8,0,14]
(3) [13,13,12,10,14,13,13,10,2,12,11,9,11,3,7,6,12,12,3,10,3,9,4,15,12,7,15,5,1,15,11,9,14,0,4]
The parameters were introduced by Kimbrough et al: Customer demand uniformly distributed between [0,15].
For our own test problem, we define a list of random variables for the customer demands. The random values for the customer demands can take the integers 0 to 15. The size of the lists is 35 for both and represents the number of calendar weeks.

4 Moving to DQN

As described in the paper, using an infinite space for the elements in the state vector, it is impossible to determine the near-optimal policy, when using the Q learning approach. This is because it would need infinite search power, as the Q table would grow infinitely. Thus, the paper proposes to discretize the state space to some coded values. However, in the context of the beer game, relying discretized states can be restrictive, as this would mean information loss. It would in consequence be better to represent the state variable, ranging from negative infinity to positive infinity.

Using DQN permits us to handle these scenarios where the number of states aligns more realistically with real-world situations. In our specific case this implies the adoption of a continuous state space for the actors. Instead of the building of a Q table, we would need to find a function to approximate the Q values. This approximation can be achieved through the use of a neural network, that predicts a Q value based on a current state.

To understand better what needs to be changed on the current implementation of the inventory optimization, we will explain how the DQN works.

To begin with, a replay buffer interacts with the environment. In these interactions, it collects data by following an ε-greedy policy for specific state-action pairs. This data collection process is focused on obtaining both the reward and the state that follows a particular action. Once acquired, this data is used as training data, for later use. From training data amassed, a sample is made to create a subset. This subset, becomes the basis for training of a neural network, called the Q neural network. This network is used to predict the Q value of the current state. In parallel, a second neural network, the Target network, predicts the Q value of the next state, that is linked to the action undertaken after the current state. With the neural network architecture in place, the loss is computed in the next step. This metric measures the discrepancy between the predicted Q value from the Q neural network and the target Q value. The target Q value is computed by adding the reward associated with the current action and the next states Q value. This discrepancy could for example be calculated through the Mean Squared Error (MSE). This loss is then used for backpropagation, a process to adjust the neural networks weights with the goal to minimize the loss between the current Q value and the target Q value. Backpropagation works by plotting the gradients of the loss, with respect to the network weights, which in turn indicates how much the weights have to be adjusted to reduce the loss. The extent of the update is controlled by the learning rate α, ensuring a balanced convergence toward more accurate Q value predictions. However, the update of the weights is only carried out for the Q NN and not the target NN, to ensure stability in the learning process. By keeping the target NN's weights fixed for a set number of iterations, we provide a more stable and consistent target for the Q NN to learn from. Although the update of the weights of the target NN happen periodically, by matching the weights to the weights of the Q NN. This whole process is repeated for each time step until a specified maximum number of iterations is reached.

Following this logic, we can now understand what needs to be changed in our implementation to apply the DQN. As already mentioned, the state space doesn’t have to be encoded anymore and we can just assume the infinite state space. Thus, the state can be represented as a vector of continuous values [S1, S2, S3, S4]. This is also recommended as it wouldn’t make sense to use DQN if the state and action space stays the same, as the training of the NN takes more resources, when the state and action space is small.

For the action space, it has to remain discretized. This is because the neural network needs a defined number of output neurons, each corresponding to a specific action. Nevertheless, DQN allows for a broader action space without employing too many resources, where in contrast, the increase of actions would increase the number of operations in a Q table exponentially.

Most importantly, we wouldn’t need a Q table anymore. We could just replace it with the neural network that reliably predicts the Q values. To train that NN, we would have to implement a new algorithm that updates the weights for the NN iteratively, to get more realistic Q values.

We would also need to implement a replay buffer. In this, the state action combination, paired with their reward and next state is stored for each step. This could be saved as a list with multiple “experiences” stored in them. As we have multiple actors in the beer game, we need to consider that each experience stored should capture the combined actions and states of all actors. Thus, each experience should be represented by a vector of states, action, rewards and next states. Additionally, the replay buffer should also include information about if the next state is terminal or not, as if the next state is terminal, there is no future reward to consider, and this would signify the end of an episode (35 weeks). From this list we should be able to sample a batch of those experiences.

On top of the hyperparameters that we have in Q learning, we will get some more parameters to tune, such as the size of the replay buffer, the batch size of the subset for training the NN, the update frequency for the target network and the hyperparameters of the neural networks, including the learning rate for the backpropagation.

Exploration/exploitation stays the same. We still use an ε-greedy strategy with ε decreasing over time, leading to exploitation in contrast to exploration.

In conclusion, the shift to DQN makes it possible to handle realistic state spaces and provides a more robust approach for inventory optimization in complex supply chain systems.

5 Discussion and results

In this section we compare the performance of our Reinforcement Learning Ordering Mechanism to two other models. There are different policies for the benchmarks. Benchmark 1 always orders exactly the quantity of demand x, making y equal to zero. Benchmark 2, on the other hand, adds a random value between 0 and 3 to the demand. As already mentioned, all models are facing stochastic demand during the T=35 weeks. We measured the performance of each model on the three given sets from the paper by Chaharsooghi et al. (2008). The results are displayed in the following table:

	Main Set	Test Set 1	Test Set 2	Test Set 3	Mean
RLOM	1590	1688	1650	1853	1695
Benchmark1	2665	2665	2742	2859	2733
Benchmark2	2356	2354	2103	2313	2282

Figure 2: RLOM and benchmarks means of main and test Sets 1-3.

The outcome shows that the RLOM outperforms both benchmarks in all given problem sets and underlines the usefulness of the calculated policy. This outlines that the proposed model (RLOM) is efficient and can find good policies under complex scenarios where analytical solutions are not available.
As can be seen in the following plot, the percentage reduction per iteration of the Q-values (initially at 1000) decreases very quickly to a value just above zero. Therefore, a similar result can be achieved with a smaller number of iterations. Here, the high variance is worth mentioning, which is due to the learning rate alpha of 0.17. Thus, the more recent observations acquire a higher weighting and the values converge only very slowly.

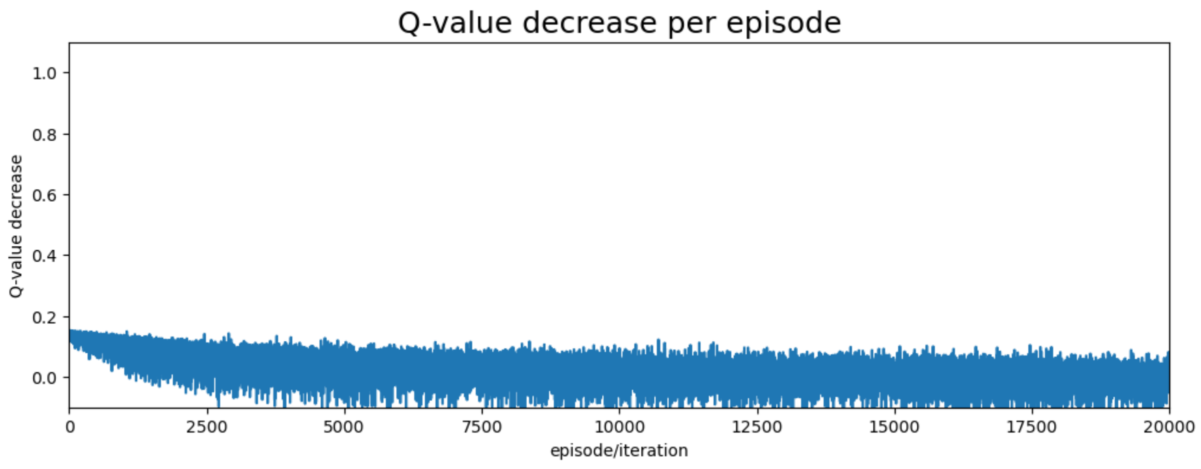


Figure 3: Q-value decrease per episode.

Our model could be further improved by adding lead times as in the original paper to create a more realistic environment. Also, we could think about adding manufacturer and supplier to the supply chain, but that would increase the size of the Q-values table (number of state action pairs) by a lot. This leads to the fact that a significant amount of computational power is required.

Bibliography

S. Kamal Chaharsooghi, J. Heydari, S. Hessameddin Zegordi, A reinforcement learning model for supply chain ordering management: An application to the beer game (2008) 949–959.

S.O. Kimbrough, D.J. Wu, F. Zhong, Computers play the beer game: can artificial agents manage supply chains? Decision Support Systems 33 (2002) 323–333.

Doshi, Ketan (2020). Reinforcement Learning Explained Visually: Part 5 — Deep Q Networks Step by Step. Towards Data Science. https://towardsdatascience.com/reinforcement-learning-explained-visually-part-5-deep-q-networks-step-by-step-5a5317197

inventory_beer_game_implementation

August 19, 2023

1 Reinforcement Learning Project - Inventory beer game

```
[ ]: import numpy as np
     import pandas as pd
     import matplotlib.pyplot as plt
     import random
     import sys
```

1.0.1 Setup

```
[ ]: # Supply Chain and its agents
     supply_chain = {'level 0': 'Customer',
                     'level 1': 'Retailer',
                     'level 2': 'Distributor',
                     'level 3': 'Manufacturer'}

     agents = [supply_chain[i] for i in list(supply_chain.keys())[1:]]


     # All possible coded inventory positions of agents and the respective state␣
      ↪pairs (5 states with 25 state pairs)
     states = np.arange(start=1, stop=6)
     state_pairs = [(i, j, k) for i in states for j in states for k in states]

     # y-value in ordering rule x+y with x equal to the demand from the downstream␣
      ↪agent and the respective action pairs (4 actions with 16 action pairs)
     actions = np.arange(stop=4)
     action_pairs = [(i, j, k) for i in actions for j in actions for k in actions]


     # Initial matrix with Q-values (minimization -> high values)
     Q_values = np.full(shape=(len(state_pairs), len(action_pairs)), fill_value=1000)
     df_Q_values = pd.DataFrame(data=Q_values, index=state_pairs,␣
      ↪columns=action_pairs)
     display(df_Q_values.head())
```

```
           (0, 0, 0)  (0, 0, 1)  (0, 0, 2)  (0, 0, 3)  (0, 1, 0)  (0, 1, 1)  \
(1, 1, 1)       1000       1000       1000       1000       1000       1000
```

```
(1, 1, 2)      1000      1000      1000      1000      1000      1000
(1, 1, 3)      1000      1000      1000      1000      1000      1000
(1, 1, 4)      1000      1000      1000      1000      1000      1000
(1, 1, 5)      1000      1000      1000      1000      1000      1000

          (0, 1, 2)  (0, 1, 3)  (0, 2, 0)  (0, 2, 1)  …  (3, 1, 2)  \
(1, 1, 1)      1000      1000      1000      1000  …       1000
(1, 1, 2)      1000      1000      1000      1000  …       1000
(1, 1, 3)      1000      1000      1000      1000  …       1000
(1, 1, 4)      1000      1000      1000      1000  …       1000
(1, 1, 5)      1000      1000      1000      1000  …       1000

          (3, 1, 3)  (3, 2, 0)  (3, 2, 1)  (3, 2, 2)  (3, 2, 3)  (3, 3, 0)  \
(1, 1, 1)      1000      1000      1000      1000      1000      1000
(1, 1, 2)      1000      1000      1000      1000      1000      1000
(1, 1, 3)      1000      1000      1000      1000      1000      1000
(1, 1, 4)      1000      1000      1000      1000      1000      1000
(1, 1, 5)      1000      1000      1000      1000      1000      1000

          (3, 3, 1)  (3, 3, 2)  (3, 3, 3)
(1, 1, 1)      1000      1000      1000
(1, 1, 2)      1000      1000      1000
(1, 1, 3)      1000      1000      1000
(1, 1, 4)      1000      1000      1000
(1, 1, 5)      1000      1000      1000

[5 rows x 64 columns]
```

[ ]:
```python
# Define parameters
iteration = 1
max_iteration = 20000
T = 10
gamma = 0.9 #low: faster convergence! less variance
alpha = 0.17 #high: faster updating q-values (more weight to new observations)␣
 ↪and big variance in q-value updates
sigma = 0.002
proba_exploitation = 0.02
start_exploitation = proba_exploitation

# Set up Lists to store the results of each iteration/episode
S = [list(np.repeat(0, len(agents))) for i in range(T+4)]
CS = [list(np.repeat(0, len(agents))) for i in range(T+4)]
D = [list(np.repeat(0, len(agents))) for i in range(T+4)]
O = [list(np.repeat(0, len(agents))) for i in range(T+4)]
x = [list(np.repeat(0, len(agents))) for i in range(T+4)]
y = [list(np.repeat(0, len(agents))) for i in range(T+4)]
r = [list(np.repeat(0, len(agents))) for i in range(T+4)]
```

```
R = [0 for i in range(T)]
G = [0 for i in range(T)]
q = [0 for i in range(T)]
Q = [0 for i in range(max_iteration)]
```

Meaning of the variables - S: (start-)inventory positions/states per agent and time step t - CS: coded states per agent and time step t - O: ordering size per agent and time step t - D: distribution amount per agent and time step t - x: demand from downstream level per agent and time step t - y: action per agent and time step t - r: reward per agent and time step t - R: reward of the supply chain per time step t - Return G: total discounted rewards per time step t - q: measure the decrease of the Q-value per time step t - Q: measure the mean decrease of Q-values per episode to indicate convergence

[ ]:
```
# helperfunctions
# function to convert real states to coded states
def coded_state(inventory):
    if inventory < -5:
        return 1
    elif inventory < 0:
        return 2
    elif inventory < 5:
        return 3
    elif inventory < 12:
        return 4
    else:
        return 5


# function to view episodes
def fun_episode(S, CS, D, O, x, y, r, head=True):
    time_steps = ['t='+str(i) for i in np.arange(start=0, stop=T+4)]

    df = pd.DataFrame({'Inventory/States S': [tuple(i) for i in S],
                       'Coded states CS': [tuple(i) for i in CS],
                       'Demand x': [tuple(i) for i in x],
                       'Distribution amount D': [tuple(i) for i in D],
                       'Action y': [tuple(i) for i in y],
                       'Ordering size': [tuple(i) for i in O],
                       'Costs r': [tuple(i) for i in r]},
                       index=time_steps)
    df.index.name = 'Time'
    if head == True:
        return display(df.head())
    else: return display(df)


# function to define the starting states of the supply chain
```

```python
def fun_start_state(how='value', inv=10):
    # select own start values for all agents
    if how == 'value':
        for agent in range(len(agents)):
            S[0][agent] = inv
            CS[0][agent] = coded_state(S[0][agent])

    # each episode has a change of 50% to start with high (12) or with low (0)␣
 ↪inventory for all agents
    elif how == 'high/low':
        starting_state = random.choices(population=['high', 'low'], weights=[0.
 ↪5, 0.5])[0]
        if starting_state == 'high':
            for agent in range(len(agents)):
                S[0][agent] = 10
                CS[0][agent] = coded_state(S[0][agent])
        else:
            for agent in range(len(agents)):
                S[0][agent] = 1
                CS[0][agent] = coded_state(S[0][agent])

    # random starting positions (-10 to 16) of agents for each episode
    elif how == 'random':
        for agent in range(len(agents)):
            S[0][agent] = random.choices(np.arange(start=-10, stop=16))[0]
            CS[0][agent] = coded_state(S[0][agent])


# function to measure the Q-value decrease
def fun_q_decrease(alpha, q_value, G):
    new_q_value = (1-alpha) * q_value + alpha * G
    decrease = (q_value - new_q_value) / q_value
    return new_q_value, decrease


# function to visualize the Q-vlaue decrease
def plot_q_decrease(Q, iteration):
    plt.figure(figsize=(12,4))
    plt.plot(Q)
    plt.xlim(0, iteration)
    plt.ylim(-0.1, 1.1)
    plt.xlabel('episode/iteration')
    plt.ylabel('Q-value decrease')
    plt.title('Q-value decrease per episode', size=18)
    return plt.show()
```

[ ]:
```
# Three possible options to set starting states
fun_start_state(how='value', inv=10)
print('Inventory state S: {}'.format(S))
print('Coded state S: {}\n'.format(CS))

fun_start_state(how='high/low')
print('Inventory state S: {}'.format(S))
print('Coded state S: {}\n'.format(CS))

fun_start_state(how='random')
print('Inventory state S: {}'.format(S))
print('Coded state S: {}\n'.format(CS))
```

```
Inventory state S: [[10, 10, 10], [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]]
Coded state S: [[4, 4, 4], [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]]

Inventory state S: [[10, 10, 10], [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]]
Coded state S: [[4, 4, 4], [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]]

Inventory state S: [[8, 3, 14], [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]]
Coded state S: [[4, 3, 5], [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]]
```

[ ]:
```
#%%capture
# Main function

while iteration <= max_iteration:
    if iteration in [1, 10, 100, 1000, 10000, 25000, 50000, 75000,
                     100000, 250000, 500000, 750000, max_iteration]:
        print('iteration {}'.format(iteration))

    #print('-------------------episode {}------------------'.format(iteration))
    #print('Exploitation probability: {}\n'.format(proba_exploitation))
    t=0

    # random starting position of agents for each episode
    fun_start_state(how='high/low', inv=-2) #option 1: ('value', inv=12);␣
↪option 2: 'high/low'; option 3: 'random'

    while t < T:
        #print('-----------------time step t={}----------------'.format(t))

        # view state and coded state
        state = tuple(S[t])
        c_state = tuple(CS[t])
        #print('State S: {} and coded state CS: {}'.format(state, c_state))

        for agent in range(len(agents)):
            level = supply_chain['level ' + str(agent+1)]
            #print('------------------{}------------------'.format(level))

        # step 1: receive the new demand of the downstream agent
            if level == 'Retailer':
                x[t][agent] = np.random.randint(low=0, high=15)
            else:
                x[t][agent] = O[t-1][agent-1]

            # Add negative inventory (=demand of previous time steps) to the␣
↪new demand
            inventory = S[t][agent]
            if inventory >= 0:
                #print('Demand x from downstream ({}) at t-1: {}'.
↪format(supply_chain['level ' + str(agent)], x[t][agent]))
                pass
            else:
                #print('Demand x from downstream ({}) at t-1 + demand of␣
↪previous time steps:'.format(supply_chain['level ' + str(agent)]))
                #print(    '{} + {} = {}'.format(x[t][agent], np.
↪abs(inventory), x[t][agent] + np.abs(inventory)))
                x[t][agent] += np.abs(inventory)

        # step 2: fulfill order of downstream agent from onhand inventory and␣
↪calculate possible backlog costs
            if (inventory >= 0) & (level != agents[-1]):
                D[t][agent] = min(x[t][agent], max(inventory, D[t-1][agent+1]))␣
↪#distribution quantity is the demand or the maximum of inventory and last␣
↪received order
                #print('Distribution size D: {}'.format(D[t][agent]))
            elif level != agents[-1]:
                D[t][agent] = min(x[t][agent], D[t-1][agent+1]) #negative␣
↪inventory is demand of previous time steps and still needs to be complied
```

```
                #print('Distribution size D (delivery of upstream agent in t-1):
↪ {}'.format(D[t][agent])) #(max of demand and last received order)
            elif (inventory >= 0) & (level == agents[-1]):
                D[t][agent] = min(x[t][agent], max(inventory, O[t-1][agent]))␣
↪#last agent receives his own order out of warehouse
                #print('Distribution size D: {}'.format(D[t][agent]))
            else:
                D[t][agent] = min(x[t][agent], O[t-1][agent])
                #print('Distribution size D (delivery of warehouse): {}'.
↪format(D[t][agent]))

            backlog = x[t][agent] - D[t][agent] #penalty/backlog costs␣
↪(previous backlogs included in demand if inventory is negative)
            #print('Backlog size: {}'.format(backlog))

        # step 3: placing order for stock replenishment
            # define best action for agent and select it with initially very␣
↪small probability (first exploration and mostly random choices)
            best_action = df_Q_values.iloc[df_Q_values.index.
↪get_loc(tuple(CS[t]))].idxmin()[agent]
            #print('Best action y*: {}'.format(best_action))

            y[t][agent] = random.choices([best_action] + list(actions),␣
↪weights=[proba_exploitation] + list(np.repeat((1 - proba_exploitation) /␣
↪len(actions), len(actions))))[0]
            #print('Action y: {}'.format(y[t][agent]))

            if inventory >= 0:
                O[t][agent] = x[t][agent] + y[t][agent]
            else: O[t][agent] = x[t][agent] + inventory + y[t][agent] #subtract␣
↪negative inventory (demand of previous time steps) again - has been ordered␣
↪already
            #print('Ordering size O: {}\n'.format(O[t][agent]))

        # step 4: previous orders are received from the upstream agent (update␣
↪states for t+1)
        #print('UPDATING STATES AND CALCULATING COSTS PER AGENT')
        for agent in range(len(agents)):
            level = supply_chain['level ' + str(agent+1)]
            #print('------------------{}------------------'.format(level))

            # update inventory with demand of t
            if  S[t][agent] >= 0:
                inventory =  S[t][agent] - x[t][agent]
            else: inventory = - x[t][agent] #demand x contains negative␣
↪inventory + new demand = new inventory
```

```
            # update state t+1 with inventory + received order
            if level != agents[-1]:
                S[t+1][agent] = inventory + D[t][agent+1]
                #print('Inventory {} after receiving order +{} of t-1: {}'.
↪format(inventory, D[t][agent+1], S[t+1][agent]))

            # for last agent in supply chain: update state t+1 with inventory +␣
↪order of t-1 (delivers from warehouse)
            else:
                S[t+1][agent] = inventory + O[t-1][agent]
                #print('Inventory {} receiving order +{} of t-1: {}'.
↪format(inventory, O[t-1][agent], S[t+1][agent]))

            # update coded states
            CS[t+1][agent] = coded_state(S[t+1][agent])

            # calculate agent's costs (onhand inventory holding costs + penalty␣
↪costs)
            r[t][agent] = 1 * max(S[t+1][agent], 0) + 2 * (x[t][agent] -␣
↪D[t][agent]) #backlog
            #print('Costs r: {}\n'.format(r[t][agent]))

        # calculate the total supply chain costs in t
        action = tuple(y[t])
        R[t] = np.sum(r[t])
        #print('Supply Chain costs R in state {} with action {} at t={}: {}'.
↪format(state, action, t, R[t]))

        # increase time step t
        t += 1
        #print('\n\n')

    # view last episode
    #fun_episode(S, CS, D, O, x, y, r, head=False)

    # Loop from T to t=0 to calculate immediate rewards and returns
    #print('-----Rewards and Total discounted returns-----')
    for t in np.arange(start=0, stop=T)[::-1]:
        # view immediate reward of each visited state-action-pair
        #print('Immediate reward R in t={}: {}'.format(t, R[t]))

        # calculate the return G: total discounted rewards
        G[t] = R[t] + np.sum(R[t+1:] * np.array([gamma**i for i in np.
↪arange(start=1, stop=T-t)]))
```

```
    # view returns of all state-action-pairs
    #print('\nTotal discounted rewards (Return G): \n{}\n'.format(G))

    # update all visited Q-values
    #print('---------------UPDATING Q-VALUES--------------')
    for t in range(T):
        state = tuple(CS[t])
        action = tuple(y[t])

        # get current Q-value
        q_value = df_Q_values.iloc[df_Q_values.index.get_loc(state),␣
↪df_Q_values.columns.get_loc(action)]
        #print('t={}: state: {}, action: {}, old Q-value: {}, return: {}'.
↪format(t, state, action, q_value, G[t]))

        # update Q-values according to equation 12 in the paper (slide 30␣
↪TD-learning script)
        df_Q_values.iloc[df_Q_values.index.get_loc(state), df_Q_values.columns.
↪get_loc(action)] = (1-alpha) * q_value + alpha * G[t] #equal to: q_value +␣
↪alpha * (G[t] - q_value)
        new_q_value, decrease = fun_q_decrease(alpha, q_value, G[t])
        q[t] = decrease
        #print('New Q-value: {}\n'.format(new_q_value))

    # calculate mean Q-value decrease of current episode
    Q[iteration-1] = np.mean(q)
    #print('Mean Q-value decrease of all visited state-action-pairs in this␣
↪episode: {}'.format(Q[iteration-1]))

    # Check if the convergence criterion has been reached
    if (iteration > 100) & (np.mean(np.abs(Q[iteration-5:iteration])) < sigma):
        #print('Convergence!')
        break

    # increase exploitation probability linearly
    proba_exploitation += (0.95 - start_exploitation) / max_iteration
    #print('\n\n')

    # start with next iteration
    iteration += 1

# round dataframe with Q-values
df_Q_values = np.round(df_Q_values, 2)

# view new Q-value table
display(df_Q_values)
```

```python
# plot decrease of Q-vlaues
plot_q_decrease(Q, iteration)
```

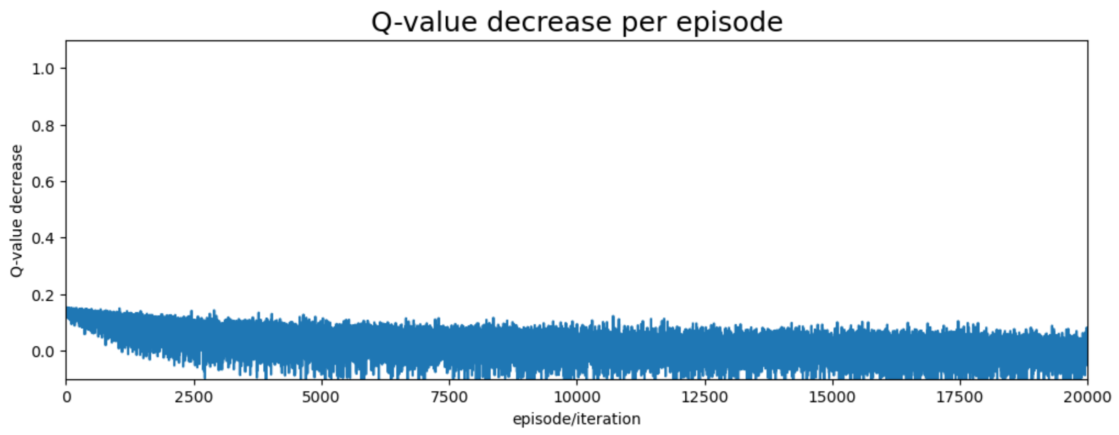
```
iteration 1
iteration 10
```

```
C:\Users\Dennis\AppData\Roaming\Python\Python39\site-
packages\numpy\core\fromnumeric.py:3464: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
C:\Users\Dennis\AppData\Roaming\Python\Python39\site-
packages\numpy\core\_methods.py:192: RuntimeWarning: invalid value encountered
in scalar divide
  ret = ret.dtype.type(ret / rcount)
```

```
iteration 100
iteration 1000
iteration 10000
iteration 20000
```

```
           (0, 0, 0)  (0, 0, 1)  (0, 0, 2)  (0, 0, 3)  (0, 1, 0)  (0, 1, 1)  \
(1, 1, 1)     256.30     259.07     312.41     259.63     268.48     256.72
(1, 1, 2)     290.93     264.66     249.34     284.42     276.68     288.02
(1, 1, 3)     331.09     300.63     304.08     315.28     320.68     302.00
(1, 1, 4)     233.64     176.65     208.48     196.72     167.57     219.97
(1, 1, 5)     843.43    1000.00     747.48     840.71     846.23    1000.00
…               …          …          …          …          …          …
(5, 5, 1)    1000.00    1000.00    1000.00    1000.00    1000.00    1000.00
(5, 5, 2)    1000.00    1000.00     346.03    1000.00    1000.00    1000.00
(5, 5, 3)     854.43     737.61    1000.00    1000.00     735.14    1000.00
(5, 5, 4)     329.28     738.00     616.91     839.18     114.05     613.38
(5, 5, 5)     161.92     591.60     598.97     836.80     341.57     517.67

           (0, 1, 2)  (0, 1, 3)  (0, 2, 0)  (0, 2, 1)  …  (3, 1, 2)  \
(1, 1, 1)     258.15     300.59     260.57     258.18  …     306.08
(1, 1, 2)     249.25     244.76     270.44     256.26  …     250.26
(1, 1, 3)     304.39     306.04     331.11     323.77  …     343.67
(1, 1, 4)     286.30     162.69     176.94     186.42  …     290.78
(1, 1, 5)    1000.00     708.68     506.74     715.54  …     717.00
…               …          …          …          …  …        …
(5, 5, 1)    1000.00    1000.00    1000.00    1000.00  …    1000.00
(5, 5, 2)     844.60    1000.00    1000.00    1000.00  …    1000.00
(5, 5, 3)    1000.00     862.38    1000.00    1000.00  …    1000.00
(5, 5, 4)     229.70     308.80     716.54     715.04  …     186.66
(5, 5, 5)    1000.00     711.40     608.44     454.98  …     318.24

           (3, 1, 3)  (3, 2, 0)  (3, 2, 1)  (3, 2, 2)  (3, 2, 3)  (3, 3, 0)  \
(1, 1, 1)     275.05     247.79     265.87     257.05     268.56     276.58
(1, 1, 2)     244.15     256.08     291.52     250.41     255.78     247.18
```

```
(1, 1, 3)     309.93     300.68     303.90     306.64     319.95     319.32
(1, 1, 4)     165.46     176.20     197.31     172.15     163.28     188.50
(1, 1, 5)     536.56     835.44     857.15     835.78     210.48    1000.00
…              …          …          …          …          …          …
(5, 5, 1)    1000.00    1000.00    1000.00    1000.00    1000.00    1000.00
(5, 5, 2)    1000.00    1000.00    1000.00    1000.00    1000.00    1000.00
(5, 5, 3)    1000.00     860.12    1000.00     727.68    1000.00     115.10
(5, 5, 4)     182.38     118.86     508.67     448.47     391.26     146.40
(5, 5, 5)     306.26     117.54     519.25     710.86     334.94     167.83

           (3, 3, 1)  (3, 3, 2)  (3, 3, 3)
(1, 1, 1)     273.12     264.61     292.37
(1, 1, 2)     263.63     271.39     251.50
(1, 1, 3)     356.12     250.23     321.57
(1, 1, 4)     179.66     166.48     171.03
(1, 1, 5)     626.80    1000.00     607.52
…              …          …          …
(5, 5, 1)    1000.00    1000.00    1000.00
(5, 5, 2)    1000.00    1000.00    1000.00
(5, 5, 3)     715.82     529.90     293.59
(5, 5, 4)     510.41     293.08     538.24
(5, 5, 5)     382.01     838.33     447.18

[125 rows x 64 columns]
```

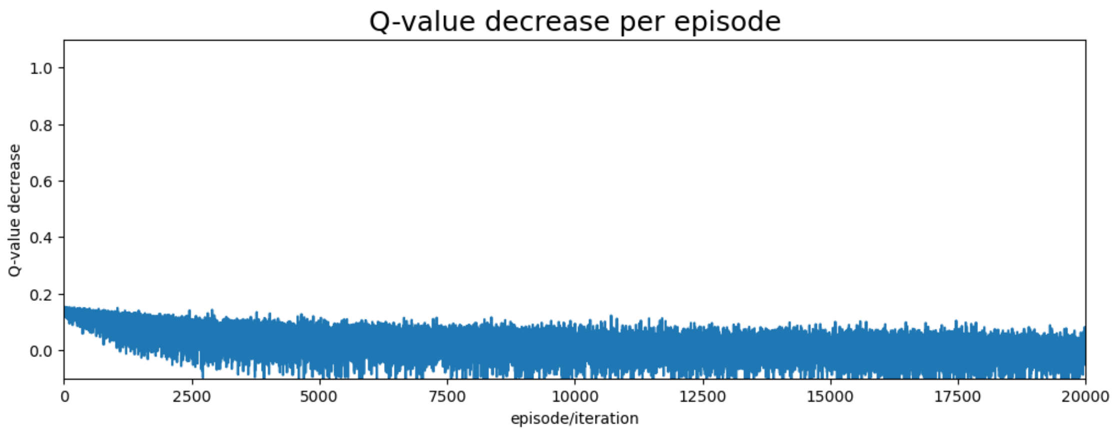


1.0.2 View new Q-values

```
[ ]: # Percentage of updated Q-values
num_pairs = len(df_Q_values.index) * len(df_Q_values.columns)
print('Number of state-action-pairs: {}'.format(num_pairs))
print('Percentage of updated Q-values: {} %\n'.format(np.round(np.sum(np.
 ↪sum(df_Q_values != 10000)) / num_pairs * 100, 4)))
```

```
visits = max_iteration * T
print('Number of visits with max_iteration={} and T={}: {}'.
 ↪format(max_iteration, T, visits))
print('Average amount of visits per state-action-pair: {}\n'.format(np.
 ↪round(visits / num_pairs, 2)))

# plot decrease of Q-vlaues
print('Last five Q-value decreases: {}'.format(np.round(Q[iteration-5:
 ↪iteration], 4)))
print('Mean of last five Q-value decreases: {}'.format(np.round(np.
 ↪mean(Q[iteration-5:iteration]), 4)))

plot_q_decrease(Q, iteration)

df_Q_values
```

```
Number of state-action-pairs: 8000
Percentage of updated Q-values: 100.0 %

Number of visits with max_iteration=20000 and T=10: 200000
Average amount of visits per state-action-pair: 25.0

Last five Q-value decreases: [-0.0083 -0.0157 -0.0016  0.0182]
Mean of last five Q-value decreases: -0.0019
```



```
[ ]:            (0, 0, 0)  (0, 0, 1)  (0, 0, 2)  (0, 0, 3)  (0, 1, 0)  (0, 1, 1)  \
     (1, 1, 1)     256.30     259.07     312.41     259.63     268.48     256.72
     (1, 1, 2)     290.93     264.66     249.34     284.42     276.68     288.02
     (1, 1, 3)     331.09     300.63     304.08     315.28     320.68     302.00
     (1, 1, 4)     233.64     176.65     208.48     196.72     167.57     219.97
     (1, 1, 5)     843.43    1000.00     747.48     840.71     846.23    1000.00
```

```
…              …          …          …          …          …          …
(5, 5, 1)    1000.00    1000.00    1000.00    1000.00    1000.00    1000.00
(5, 5, 2)    1000.00    1000.00     346.03    1000.00    1000.00    1000.00
(5, 5, 3)     854.43     737.61    1000.00    1000.00     735.14    1000.00
(5, 5, 4)     329.28     738.00     616.91     839.18     114.05     613.38
(5, 5, 5)     161.92     591.60     598.97     836.80     341.57     517.67

           (0, 1, 2)  (0, 1, 3)  (0, 2, 0)  (0, 2, 1)  …  (3, 1, 2)  \
(1, 1, 1)     258.15     300.59     260.57     258.18  …     306.08
(1, 1, 2)     249.25     244.76     270.44     256.26  …     250.26
(1, 1, 3)     304.39     306.04     331.11     323.77  …     343.67
(1, 1, 4)     286.30     162.69     176.94     186.42  …     290.78
(1, 1, 5)    1000.00     708.68     506.74     715.54  …     717.00
…              …          …          …          …  …          …
(5, 5, 1)    1000.00    1000.00    1000.00    1000.00  …    1000.00
(5, 5, 2)     844.60    1000.00    1000.00    1000.00  …    1000.00
(5, 5, 3)    1000.00     862.38    1000.00    1000.00  …    1000.00
(5, 5, 4)     229.70     308.80     716.54     715.04  …     186.66
(5, 5, 5)    1000.00     711.40     608.44     454.98  …     318.24

           (3, 1, 3)  (3, 2, 0)  (3, 2, 1)  (3, 2, 2)  (3, 2, 3)  (3, 3, 0)  \
(1, 1, 1)     275.05     247.79     265.87     257.05     268.56     276.58
(1, 1, 2)     244.15     256.08     291.52     250.41     255.78     247.18
(1, 1, 3)     309.93     300.68     303.90     306.64     319.95     319.32
(1, 1, 4)     165.46     176.20     197.31     172.15     163.28     188.50
(1, 1, 5)     536.56     835.44     857.15     835.78     210.48    1000.00
…              …          …          …          …          …          …
(5, 5, 1)    1000.00    1000.00    1000.00    1000.00    1000.00    1000.00
(5, 5, 2)    1000.00    1000.00    1000.00    1000.00    1000.00    1000.00
(5, 5, 3)    1000.00     860.12    1000.00     727.68    1000.00     115.10
(5, 5, 4)     182.38     118.86     508.67     448.47     391.26     146.40
(5, 5, 5)     306.26     117.54     519.25     710.86     334.94     167.83

           (3, 3, 1)  (3, 3, 2)  (3, 3, 3)
(1, 1, 1)     273.12     264.61     292.37
(1, 1, 2)     263.63     271.39     251.50
(1, 1, 3)     356.12     250.23     321.57
(1, 1, 4)     179.66     166.48     171.03
(1, 1, 5)     626.80    1000.00     607.52
…              …          …          …
(5, 5, 1)    1000.00    1000.00    1000.00
(5, 5, 2)    1000.00    1000.00    1000.00
(5, 5, 3)     715.82     529.90     293.59
(5, 5, 4)     510.41     293.08     538.24
(5, 5, 5)     382.01     838.33     447.18

[125 rows x 64 columns]
```

1.0.3 Test performance of the Reinforcement learning ordering mechanism (RLOM) on main and test problems

```
[ ]: # Main test problem
     main_set = {'customer_demand':␣
      ↪[15,10,8,14,9,3,13,2,13,11,3,4,6,11,15,12,15,4,12,3,13,10,15,15,3,11,1,13,10,10,0,0,8,0,14]

     # Test problem 1
     set1 = {'customer_demand':␣
      ↪[5,14,14,13,2,9,5,9,14,14,12,7,5,1,13,3,12,4,0,15,11,10,6,0,6,6,5,11,8,4,4,12,13,8,12]}

     # Test problem 2
     set2 = {'customer_demand':␣
      ↪[13,13,12,10,14,13,13,10,2,12,11,9,11,3,7,6,12,12,3,10,3,9,4,15,12,7,15,5,1,15,11,9,14,0,4]

     # Own test problem
     set3 = {'customer_demand': list(np.random.randint(low=0, high=15, size=35))}
```

```
[ ]: def main_function(problem_set, policy, t=0, T=len(main_set['customer_demand'])):
         # set starting position of agents
         fun_start_state(how='value', inv=10)

         while t < T:
             print('-----------------time step t={}----------------'.format(t))

             # view state and coded state
             state = tuple(S[t])
             c_state = tuple(CS[t])
             print('State S: {} and coded state CS: {}'.format(state, c_state))

             for agent in range(len(agents)):
                 level = supply_chain['level ' + str(agent+1)]
                 print('------------------{}-------------------'.format(level))

             # step 1: receive the new demand of the downstream agent
                 if level == 'Retailer':
                     x[t][agent] = problem_set['customer_demand'][t]
                 else:
                     x[t][agent] = O[t-1][agent-1]

                 # Add negative inventory (=demand of previous time steps) to the␣
      ↪new demand
                 inventory = S[t][agent]
                 if inventory >= 0:
                     print('Demand x from downstream ({}) at t-1: {}'.
      ↪format(supply_chain['level ' + str(agent)], x[t][agent]))
                     pass
```

```
            else:
                print('Demand x from downstream ({}) at t-1 + demand of␣
↪previous time steps:'.format(supply_chain['level ' + str(agent)]))
                print(    '{} + {} = {}'.format(x[t][agent], np.abs(inventory),␣
↪x[t][agent] + np.abs(inventory)))
                x[t][agent] += np.abs(inventory)

        # step 2: fulfill order of downstream agent from onhand inventory and␣
↪calculate possible backlog costs
            if (inventory >= 0) & (level != agents[-1]):
                D[t][agent] = min(x[t][agent], max(inventory, D[t-1][agent+1]))␣
↪#distribution quantity is the demand or the maximum of inventory and last␣
↪received order
                print('Distribution size D: {}'.format(D[t][agent]))
            elif level != agents[-1]:
                D[t][agent] = min(x[t][agent], D[t-1][agent+1]) #negative␣
↪inventory is demand of previous time steps and still needs to be complied
                print('Distribution size D (delivery of upstream agent in t-1):␣
↪{}'.format(D[t][agent])) #(max of demand and last received order)
            elif (inventory >= 0) & (level == agents[-1]):
                D[t][agent] = min(x[t][agent], max(inventory, O[t-1][agent]))␣
↪#last agent receives his own order out of warehouse
                print('Distribution size D: {}'.format(D[t][agent]))
            else:
                D[t][agent] = min(x[t][agent], O[t-1][agent])
                print('Distribution size D (delivery of warehouse): {}'.
↪format(D[t][agent]))

            backlog = x[t][agent] - D[t][agent] #penalty/backlog costs␣
↪(previous backlogs included in demand if inventory is negative)
            print('Backlog size: {}'.format(backlog))


        # step 3: placing order for stock replenishment
            if policy == 'optimum':
                y[t][agent] = opt_policy[tuple(CS[t])][agent]
            elif policy == 'zero':
                y[t][agent] = 0
            else: y[t][agent] = np.random.randint(4)
            print('Action y: {}'.format(y[t][agent]))

            if inventory >= 0:
                O[t][agent] = x[t][agent] + y[t][agent]
            else: O[t][agent] = x[t][agent] + inventory + y[t][agent] #subtract␣
↪negative inventory (demand of previous time steps) again - has been ordered␣
↪already
```

```
            print('Ordering size O: {}\n'.format(O[t][agent]))

        # step 4: previous orders are received from the upstream agent (update␣
↪states for t+1)
        print('UPDATING STATES AND CALCULATING COSTS PER AGENT')
        for agent in range(len(agents)):
            level = supply_chain['level ' + str(agent+1)]
            print('------------------{}------------------'.format(level))

            # update inventory with demand of t
            if  S[t][agent] >= 0:
                inventory =  S[t][agent] - x[t][agent]
            else: inventory = - x[t][agent] #demand x contains negative␣
↪inventory + new demand = new inventory

            # update state t+1 with inventory + received order
            if level != agents[-1]:
                S[t+1][agent] = inventory + D[t][agent+1]
                print('Inventory {} after receiving order +{} of t-1: {}'.
↪format(inventory, D[t][agent+1], S[t+1][agent]))

            # for last agent in supply chain: update state t+1 with inventory +␣
↪order of t-1 (delivers from warehouse)
            else:
                S[t+1][agent] = inventory + O[t-1][agent]
                print('Inventory {} receiving order +{} of t-1: {}'.
↪format(inventory, O[t-1][agent], S[t+1][agent]))

            # update coded states
            CS[t+1][agent] = coded_state(S[t+1][agent])

            # calculate agent's costs (onhand inventory holding costs + penalty␣
↪costs)
            r[t][agent] = 1 * max(S[t+1][agent], 0) + 2 * (x[t][agent] -␣
↪D[t][agent]) #backlog
            print('Costs r: {}\n'.format(r[t][agent]))

        # calculate the total supply chain costs in t
        action = tuple(y[t])
        R[t] = np.sum(r[t])
        print('Supply Chain costs R in state {} with action {} at t={}: {}'.
↪format(state, action, t, R[t]))

        # increase time step t
        t += 1
        print('\n\n')
```

```
    # view last episode
    fun_episode(S, CS, D, O, x, y, r, head=False)


    # View results
    print(R)
    print(np.sum(R))

    return np.sum(R)
```

```
[ ]: T = 35

# Set up Lists to store the results of each iteration/episode
S = [list(np.repeat(0, len(agents))) for i in range(T+4)]
CS = [list(np.repeat(0, len(agents))) for i in range(T+4)]
D = [list(np.repeat(0, len(agents))) for i in range(T+4)]
O = [list(np.repeat(0, len(agents))) for i in range(T+4)]
x = [list(np.repeat(0, len(agents))) for i in range(T+4)]
y = [list(np.repeat(0, len(agents))) for i in range(T+4)]
r = [list(np.repeat(0, len(agents))) for i in range(T+4)]
R = [0 for i in range(T)]

# Extract optimal policy of Q-values
opt_policy = df_Q_values.idxmin(axis=1)
optimum_policy =[]
benchmark1 = []
benchmark2 = []
```

```
[ ]: %%capture

optimum_policy.append(main_function(problem_set=main_set, policy='optimum',␣
 ↪t=0, T=35))
optimum_policy.append(main_function(t=0, T=35, problem_set=set1,␣
 ↪policy='optimum'))
optimum_policy.append(main_function(t=0, T=35, problem_set=set2,␣
 ↪policy='optimum'))
optimum_policy.append(main_function(t=0, T=35, problem_set=set3,␣
 ↪policy='optimum'))
```

1.0.4 Benchmark policy 1: Order = demand (y=0)

```
[ ]: T = 35

# Set up Lists to store the results of each iteration/episode
S = [list(np.repeat(0, len(agents))) for i in range(T+4)]
CS = [list(np.repeat(0, len(agents))) for i in range(T+4)]
```

```
D = [list(np.repeat(0, len(agents))) for i in range(T+4)]
O = [list(np.repeat(0, len(agents))) for i in range(T+4)]
x = [list(np.repeat(0, len(agents))) for i in range(T+4)]
y = [list(np.repeat(0, len(agents))) for i in range(T+4)]
r = [list(np.repeat(0, len(agents))) for i in range(T+4)]
R = [0 for i in range(T)]
```

[ ]:
```
%%capture

benchmark1.append(main_function(problem_set=set1, policy='zero', t=0, T=35))
benchmark1.append(main_function(t=0, T=35, problem_set=set1, policy='zero'))
benchmark1.append(main_function(t=0, T=35, problem_set=set2, policy='zero'))
benchmark1.append(main_function(t=0, T=35, problem_set=set3, policy='zero'))
```

1.0.5 Benchmark policy 2: Order = demand + random y   [0, 3]

[ ]:
```
T = 35

# Set up Lists to store the results of each iteration/episode
S = [list(np.repeat(0, len(agents))) for i in range(T+4)]
CS = [list(np.repeat(0, len(agents))) for i in range(T+4)]
D = [list(np.repeat(0, len(agents))) for i in range(T+4)]
O = [list(np.repeat(0, len(agents))) for i in range(T+4)]
x = [list(np.repeat(0, len(agents))) for i in range(T+4)]
y = [list(np.repeat(0, len(agents))) for i in range(T+4)]
r = [list(np.repeat(0, len(agents))) for i in range(T+4)]
R = [0 for i in range(T)]
```

[ ]:
```
%%capture

benchmark2.append(main_function(problem_set=set1, policy='random', t=0, T=35))
benchmark2.append(main_function(t=0, T=35, problem_set=set1, policy='random'))
benchmark2.append(main_function(t=0, T=35, problem_set=set2, policy='random'))
benchmark2.append(main_function(t=0, T=35, problem_set=set3, policy='random'))
```

1.0.6 Compare costs for all problem sets

[ ]:
```
print('Sum of costs per problem of optimum policy with Q-vales lernt by RLOM:
  ↪\n{}'.format(optimum_policy))
print('Mean costs: {}\n'.format(np.round(np.mean(optimum_policy))))

print('Sum of costs per problem of benchmark1 (y=0):\n{}'.format(benchmark1))
print('Mean costs: {}\n'.format(np.round(np.mean(benchmark1))))

print('Sum of costs per problem of benchmark2 (random):\n{}'.format(benchmark2))
print('Mean costs: {}\n'.format(np.round(np.mean(benchmark2))))
```

```
Sum of costs per problem of optimum policy with Q-vales lernt by RLOM:
[1590, 1688, 1650, 1853]
Mean costs: 1695.0

Sum of costs per problem of benchmark1 (y=0):
[2665, 2665, 2742, 2859]
Mean costs: 2733.0

Sum of costs per problem of benchmark2 (random):
[2356, 2354, 2103, 2313]
Mean costs: 2282.0
```