inventory_beer_game_implementation

August 18, 2023

1 Reinforcement Learning Project - Inventory beer game

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

1.0.1 Setup

For better understanding of the code output decrease supply chain from customer to distributor, set max_iteration to one, set lead time lag to False (line 25 in main function), activate print functions (main function) and define state- and action-pairs as follows: - state_pairs = [(i, j) for i in states for j in states] - action_pairs = [(i, j) for i in actions for j in actions]

[ ]:
```python
# 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=7)
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]
```

[ ]:

```
# Define parameters
iteration = 1
max_iteration = 100000 #1000000
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 = 0.001
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)]
```

```python
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

[ ]:
```python
# 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 < 10:
        return 4
    elif inventory < 15:
        return 5
    else:
        return 6


# 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]},
```

```python
                   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
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 switch lead time lags on or off
def fun_lead_time(lag):
    if lag == False:
        lag = 0
    else: lag = random.choices(population=[0, 1, 2, 3, 4], weights=[0.3, 0.25, 0.2, 0.15, 0.1])[0]
    return lag


# function to measure the Q-value decrease
def fun_q_decrease(alpha, q_value, G):
```

```python
    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()
```

[ ]:
```python
# 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: [[5, 5, 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]]

Inventory state S: [[1, 1, 1], [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: [[3, 3, 3], [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: [[4, 0, -8], [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: [[3, 3, 1], [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') #option 1: ('value', inv=10); 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))

        lead_time_lag = fun_lead_time(lag=True)
        #print('Lead time lag of {} time steps\n'.format(lead_time_lag))

        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
            # set lead time lags for all agents except retailer (lead time to␣
↪customer is zero)
            if (level == agents[0]): lag = 0 #no lags in last iteration
            else: lag = lead_time_lag

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

            backlog = max(0, x[t][agent] - D[t+lag][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+lag][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])
```

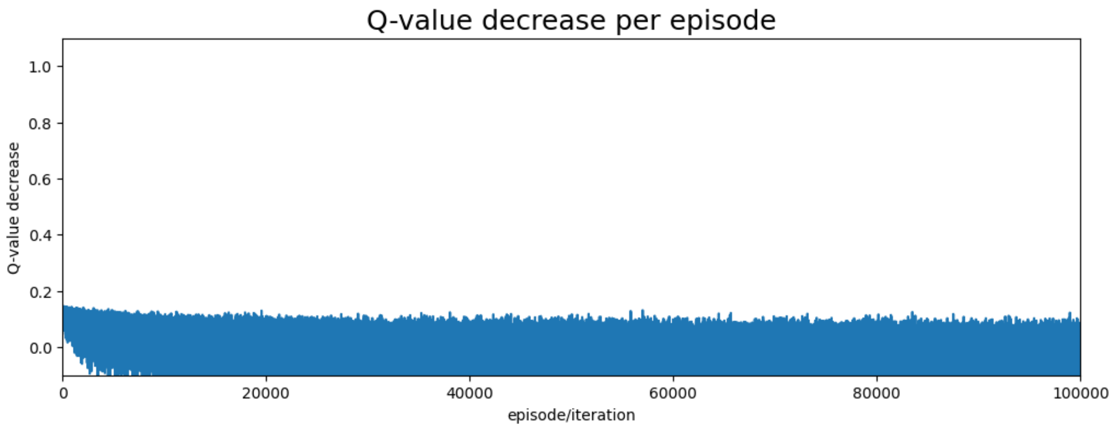
```python
        q[t] = decrease
        #print('New Q-value: {}\n'.format(new_q_value))

    # 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

    # Set list D to zero again
    D = [list(np.repeat(0, len(agents))) for i in range(T+4)]

    # 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)

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

```
iteration 1
iteration 10
iteration 100
iteration 1000
iteration 10000
iteration 25000
iteration 50000
iteration 75000
iteration 100000
```

```
/anaconda/envs/jupyter_env/lib/python3.8/site-
packages/numpy/core/fromnumeric.py:3432: RuntimeWarning: Mean of empty slice.
  return _methods._mean(a, axis=axis, dtype=dtype,
/anaconda/envs/jupyter_env/lib/python3.8/site-
packages/numpy/core/_methods.py:190: RuntimeWarning: invalid value encountered
in double_scalars
```

```
  ret = ret.dtype.type(ret / rcount)
           (0, 0, 0)  (0, 0, 1)  (0, 0, 2)  (0, 0, 3)  (0, 1, 0)  (0, 1, 1)  
(1, 1, 1)     501.23     552.42     518.74     535.64     566.56     536.72  \
(1, 1, 2)     475.17     532.60     493.29     490.40     512.01     546.49   
(1, 1, 3)     560.06     555.98     552.33     551.08     546.45     546.47   
(1, 1, 4)     348.59     339.97     340.74     369.41     360.80     367.38   
(1, 1, 5)     266.57     323.55     300.17     455.90     288.51     398.12   
…               …          …          …          …          …          …     
(6, 6, 2)     223.97     365.03     396.32     242.08     200.17     828.30   
(6, 6, 3)     185.61     198.60     189.97     193.55     200.01     189.43   
(6, 6, 4)     212.60     192.59     203.87     206.33     191.64     215.89   
(6, 6, 5)     200.63     191.76     180.77     178.83     173.97     198.51   
(6, 6, 6)     140.80     140.89     152.81     143.86     149.97     186.05   

           (0, 1, 2)  (0, 1, 3)  (0, 2, 0)  (0, 2, 1)  …  (3, 1, 2)  
(1, 1, 1)     533.26     514.50     518.42     531.77  …     538.83  \
(1, 1, 2)     504.62     496.64     498.18     504.20  …     498.83   
(1, 1, 3)     602.06     573.11     549.24     569.52  …     556.79   
(1, 1, 4)     344.45     368.29     343.38     372.05  …     366.64   
(1, 1, 5)     561.68     568.99     306.52     343.55  …     860.15   
…               …          …          …          …  …        …     
(6, 6, 2)     629.29     263.46     209.75     213.44  …     571.97   
(6, 6, 3)     184.17     189.78     199.21     219.13  …     217.03   
(6, 6, 4)     196.29     206.44     205.57     185.04  …     199.81   
(6, 6, 5)     186.04     177.43     159.42     229.55  …     174.48   
(6, 6, 6)     189.83     140.54     150.28     148.24  …     170.21   

           (3, 1, 3)  (3, 2, 0)  (3, 2, 1)  (3, 2, 2)  (3, 2, 3)  (3, 3, 0)  
(1, 1, 1)     535.68     587.73     626.04     562.00     535.52     542.30  \
(1, 1, 2)     486.54     523.50     516.35     501.66     513.96     508.24   
(1, 1, 3)     574.24     538.40     567.50     573.71     527.94     556.46   
(1, 1, 4)     375.30     342.99     344.38     359.95     376.52     345.27   
(1, 1, 5)     364.00     293.03     697.77     579.99     765.03     269.14   
…               …          …          …          …          …          …     
(6, 6, 2)     425.61     192.17     278.86     313.72     213.58     226.80   
(6, 6, 3)     219.49     209.75     215.54     186.22     211.97     236.76   
(6, 6, 4)     184.25     183.07     185.96     185.52     209.05     199.19   
(6, 6, 5)     163.09     178.95     188.98     172.59     174.38     188.71   
(6, 6, 6)     177.82     174.64     160.59     151.58     173.46     157.79   

           (3, 3, 1)  (3, 3, 2)  (3, 3, 3)  
(1, 1, 1)     544.11     537.64     532.39  
(1, 1, 2)     549.12     523.13     556.65  
(1, 1, 3)     571.69     560.06     551.72  
(1, 1, 4)     375.39     344.34     364.03  
(1, 1, 5)     272.63     367.33     429.78  
…               …          …          …     
```

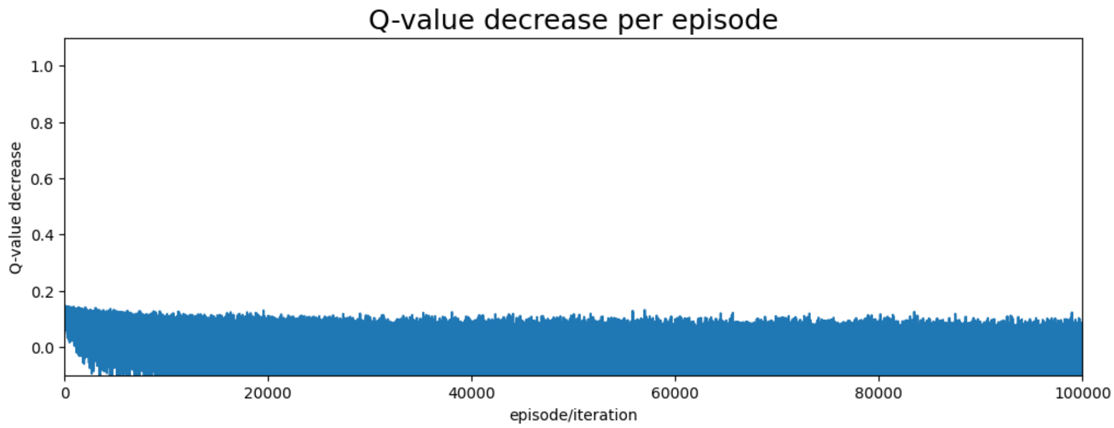
```
(6, 6, 2)     711.26     388.32     463.74
(6, 6, 3)     200.87     193.93     184.56
(6, 6, 4)     192.08     188.18     216.37
(6, 6, 5)     207.22     197.14     159.90
(6, 6, 6)     143.16     147.92     151.58

[216 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: 13824
Percentage of updated Q-values: 100.0 %

Number of visits with max_iteration=100000 and T=10: 1000000
Average amount of visits per state-action-pair: 72.34

Last five Q-value decreases: [ 0.0168  0.0688  0.0176 -0.0177]
Mean of last five Q-value decreases: 0.0214
```



[ ]:

```
           (0, 0, 0)  (0, 0, 1)  (0, 0, 2)  (0, 0, 3)  (0, 1, 0)  (0, 1, 1)  \
(1, 1, 1)     501.23     552.42     518.74     535.64     566.56     536.72
(1, 1, 2)     475.17     532.60     493.29     490.40     512.01     546.49
(1, 1, 3)     560.06     555.98     552.33     551.08     546.45     546.47
(1, 1, 4)     348.59     339.97     340.74     369.41     360.80     367.38
(1, 1, 5)     266.57     323.55     300.17     455.90     288.51     398.12
…              …          …          …          …          …          …
(6, 6, 2)     223.97     365.03     396.32     242.08     200.17     828.30
(6, 6, 3)     185.61     198.60     189.97     193.55     200.01     189.43
(6, 6, 4)     212.60     192.59     203.87     206.33     191.64     215.89
(6, 6, 5)     200.63     191.76     180.77     178.83     173.97     198.51
(6, 6, 6)     140.80     140.89     152.81     143.86     149.97     186.05

           (0, 1, 2)  (0, 1, 3)  (0, 2, 0)  (0, 2, 1)  …  (3, 1, 2)  \
(1, 1, 1)     533.26     514.50     518.42     531.77  …     538.83
(1, 1, 2)     504.62     496.64     498.18     504.20  …     498.83
(1, 1, 3)     602.06     573.11     549.24     569.52  …     556.79
(1, 1, 4)     344.45     368.29     343.38     372.05  …     366.64
(1, 1, 5)     561.68     568.99     306.52     343.55  …     860.15
…              …          …          …          …     …     …
(6, 6, 2)     629.29     263.46     209.75     213.44  …     571.97
(6, 6, 3)     184.17     189.78     199.21     219.13  …     217.03
(6, 6, 4)     196.29     206.44     205.57     185.04  …     199.81
```

```
(6, 6, 5)     186.04     177.43     159.42     229.55  …     174.48
(6, 6, 6)     189.83     140.54     150.28     148.24  …     170.21

           (3, 1, 3)  (3, 2, 0)  (3, 2, 1)  (3, 2, 2)  (3, 2, 3)  (3, 3, 0)  \
(1, 1, 1)     535.68     587.73     626.04     562.00     535.52     542.30
(1, 1, 2)     486.54     523.50     516.35     501.66     513.96     508.24
(1, 1, 3)     574.24     538.40     567.50     573.71     527.94     556.46
(1, 1, 4)     375.30     342.99     344.38     359.95     376.52     345.27
(1, 1, 5)     364.00     293.03     697.77     579.99     765.03     269.14
…              …         …         …         …          …          …
(6, 6, 2)     425.61     192.17     278.86     313.72     213.58     226.80
(6, 6, 3)     219.49     209.75     215.54     186.22     211.97     236.76
(6, 6, 4)     184.25     183.07     185.96     185.52     209.05     199.19
(6, 6, 5)     163.09     178.95     188.98     172.59     174.38     188.71
(6, 6, 6)     177.82     174.64     160.59     151.58     173.46     157.79

           (3, 3, 1)  (3, 3, 2)  (3, 3, 3)
(1, 1, 1)     544.11     537.64     532.39
(1, 1, 2)     549.12     523.13     556.65
(1, 1, 3)     571.69     560.06     551.72
(1, 1, 4)     375.39     344.34     364.03
(1, 1, 5)     272.63     367.33     429.78
…              …         …         …
(6, 6, 2)     711.26     388.32     463.74
(6, 6, 3)     200.87     193.93     184.56
(6, 6, 4)     192.08     188.18     216.37
(6, 6, 5)     207.22     197.14     159.90
(6, 6, 6)     143.16     147.92     151.58

[216 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]
            'lead_times':
 ↪[2,0,2,4,4,4,0,2,4,1,1,0,0,1,1,0,1,1,2,1,1,1,4,2,2,1,4,3,4,1,4,0,3,3,4]}

# 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],
        'lead_times':
 ↪[2,0,2,4,4,4,0,2,4,1,1,0,0,1,1,0,1,1,2,1,1,1,4,2,2,1,4,3,4,1,4,0,3,3,4]}

# Test problem 2
```

```
set2 = {'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],
        'lead_times':␣
↪[4,2,2,0,2,2,1,1,3,0,0,3,3,3,4,1,1,1,3,0,4,2,3,4,1,3,3,3,0,3,4,3,3,0,3]}

# Test problem 3
set3 = {'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],
        'lead_times':␣
↪[4,2,2,0,2,2,1,1,3,0,0,3,3,3,4,1,1,1,3,0,4,2,3,4,1,3,3,3,0,3,4,3,3,0,3]}

# Own test problem
set4 = {'customer_demand': list(np.random.randint(low=0, high=15, size=35)),
        'lead_times': list(np.random.randint(low=0, high=4, 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))

        # define random lead time lag (between 0 and 4)
        lead_time_lag = problem_set['lead_times'][t]
        print('Lead time lag of {} time steps\n'.format(lead_time_lag))

        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
            # set lead time lags for all agents except retailer (lead time to customer is zero)
            if (level == agents[0]): lag = 0 #no lags in last iteration
            else: lag = lead_time_lag

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

            backlog = max(0, x[t][agent] - D[t+lag][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
```

```python
        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+lag][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 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'))
```

```
optimum_policy.append(main_function(t=0, T=35, problem_set=set4,␣
    ↪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'))
benchmark1.append(main_function(t=0, T=35, problem_set=set4, 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'))
benchmark2.append(main_function(t=0, T=35, problem_set=set4, policy='random'))
```

1.0.6 Compare costs for all problem sets

[ ]:
```
print('Sum of costs per problem of optimum policy with Q-vales lernt byRLOM:
 ↪\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 byRLOM:
[5287, 4356, 3569, 3629, 2298]
Mean costs: 3828.0

Sum of costs per problem of benchmark1 (y=0):
[5903, 5903, 4519, 4608, 2978]
Mean costs: 4782.0

Sum of costs per problem of benchmark2 (random):
[3698, 3712, 3907, 3806, 2247]
Mean costs: 3474.0
```