Extension and Improvement of Packet Scheduling with Q-Learning

Chien Wei, Hsiung

chsiung2@gmu.edu

1 Introduction

In my Milestone 2, I replicated the results in the paper [1]. However, they only experimented with three queues : Queue 1, Queue 2 and Best-effort Queue. They didn't mention anything about systems with more queues. I was curious about how this algorithm performs in four and five queues systems. Therefore, I tried to extend it and examine its performance.

Also, the algorithm in the paper [1] has some problems : First, it can't tell which state is better if two states are $\{0, 1\}$ and $\{1, 0\}$. They are the same good or bad to the original algorithm. This makes parameter C_3 in **Reward** function meaningless in that situation. Second, if a queue exceeds its delay requirement R_i for a long time, it should get more punishment as time goes on. But it gets the same punishment so that the original algorithm can't determine whether it should be solved earlier than other queues in the same situation.

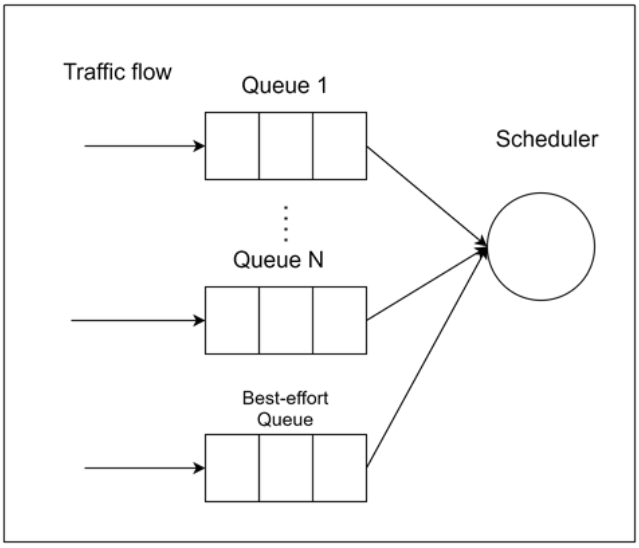


Figure 1: Structure of multiple traffic classes.

Therefore, I tried to improve convergence time of this algorithm by solving these problems.

2 Methodology

In Section 2.1, some of the terms explanations are also shown in the report of Milestone 2. I put them here again to make sure that research questions can be well described in Section 2.2.

2.1 Assumptions and Terms Explanation

The algorithm is to use reinforcement learning to schedule multiple traffic classes showed in Figure 1. Each traffic flow has its own queue and the last queue is a best-effort queue which no performance guarantees are given. Scheduler has to decide that the next processed packet should be picked from which queue while satisfying requirements for each queue and also not starving best-effort queue. The requirement here is mean queuing delay of each queue which can't exceed the limitation on it.

In the system, all packets are fixed length and time is considered to be discrete that using time slot to represent it. In each time slot, one packet is transmitted to each queue and at most one packet can be processed (scheduled by scheduler).

- Arrival rate : Generally, arrival rate is the number of arrivals per unit of time. But here, arrival rate for each queue is represented by the probability of a packet arriving at each queue in any time slot.
- Time slot **t** : time slot is the basic unit of time in this system. I implemented one time slot as one iterations.
- Queuing delay time : For a packet, the difference between the time slot of insertion into the queue and time slot of leaving the queue.
- Mean delay requirement R_i : the maximum acceptable mean queuing delay per packet for the queue.
- Mean queuing delay M_i : measured average queuing delay of packets over the last **P** packets in a queue.
- State **s**: State is represented as $\{s_1...s_{N-1}\}$. s_i is whether traffic flow in Queue q_i meets its R_i requirement. Therefore, there are 2^{N-1} possible state.

$$s_i = \begin{cases} 0, & \text{if } M_i \leq R_i \\ 1, & \text{if } M_i > R_i \end{cases}$$

  For example, $\{1, 0..., 0\}$ means only s_1 doesn't meet the requirement.
- Action : Action $a \in \{a_1...a_N\}$. a_i means that the next processed packet is the first packet in queue q_i.
- Reward : : Reward $r = r_{time} + r_{state}$. $r_{time,i}$

  Function is as follows :

$$r_{time,i} = \begin{cases} \frac{C_1 M_i}{R_i}, & \text{if } M_i < R_i \\ C_1, & \text{if } M_i = R_i \\ -C_2, & \text{if } M_i > R_i \end{cases}$$

  $r_{time,i}$ is positive reward when packets are serviced within their delay time; otherwise, it is negative. Then, get r_{time} as below, w_i is weight for each queue. $w_i = 0.3$ if q_i was the queue serviced by the last action, otherwise $w_i = 1.0$:

$$r_{time} = \sum_{s'}^{N-1} w_i r_{time,i}$$

  r_{state} is that when agent move to a better state, it gets a positive reward. For example, state $\{0, 0..., 0\}$ is better than $\{1, 0..., 0\}$ because fewer queues don't meet the requirement R_i.

$$r_{state} = \begin{cases} C_3, & \text{if } s' \text{ is better than} s \\ 0, & \text{otherwise} \end{cases}$$

- ϵ-Greedy search : Given a state **s**, choose an action **a** such that $Q(s, a)$ has the highest value compared to other actions. But with a probability ϵ, it'll choose randomly from remaining actions.

- Batch search : Given a state **s**, choose an action **a** at random according to a probability distribution : $P(a_i|s) = \frac{k^{Q(s,a_i)}}{\sum_j k^{Q(s,a_j)}}$, where $k > 0$. In algorithm, this distribution will be updated based on newly updated Q-table every T iterations.

2.2 Research Questions

State, Action and Reward are the fundamental parts in Q-learning algorithm. However, State and Action are already simple enough in this algorithm. They're hard to be modified. The only one I can focus on is its **Reward** function. There are two parts I tried to modify :

- In the original formula, if measured mean delay M_i is larger than delay requirement R_i, a queue get a fixed negative value $-C_2$ as punishment. However, if a queue q_i keeps staying in this situation for a long time, it should get more punishment to tell the algorithm that the problem on this queue should be handled as soon as possible. Therefore, I modified the formula as follows :

$$r_{time,i} = \begin{cases} \frac{C_1 M_i}{R_i}, & \text{if } M_i < R_i \\ C_1, & \text{if } M_i = R_i \\ -C_2 + (time \times a), & \text{if } M_i > R_i \end{cases}$$

  a is a small negative value, e.g., -0.005. $time$ is current time slot.

  Now a queue will get more punishment if it doesn't meet its requirement as time goes on.

- While computing r_{state}, algorithm has to distinguish which state is better by counting the number of queues that don't meet their requirements R_i. For example, $\{0, 0\}$ is better than $\{1, 0\}$. But what if $\{1, 0\}$ and $\{0, 1\}$? They are the same good or the same bad and I want to make them different. Therefore, I make a formula for a state as follows :

$$value = \sum_{i}^{N-1} s_i \frac{1}{R_i}$$

  The formula can be viewed as : sum up all reciprocal of delay requirements of queues that don't meet their requirements. N is the number of queues. Only $N - 1$ queues because best-effort queue is not counted in. i is for queue q_i. The definition of s_i is in Section 2.1. I use reciprocal of R_i instead of just R_i because $R_1 > R_2$ means that packets in q_1 can wait longer than the ones in q_2. Packets in q_2 must be processed as soon as possible in order to meet its requirement. And the smaller value is better.

Then, I tried to apply these modifications to see whether they can improve the convergence time of this algorithm.

3 Experiments

I executed the original algorithm and modified algorithm with three queues and four queues respectively with this situation : at time slot 100,000, the system will change condition and requirement for each queues.

3.1 Three Queues

Table 1(a) is the parameters for both original and modified algorithm. In modified algorithm, parameter **a** is set to -0.0005. Table 1(b) is the change of condition and requirement at time slot 100,000.

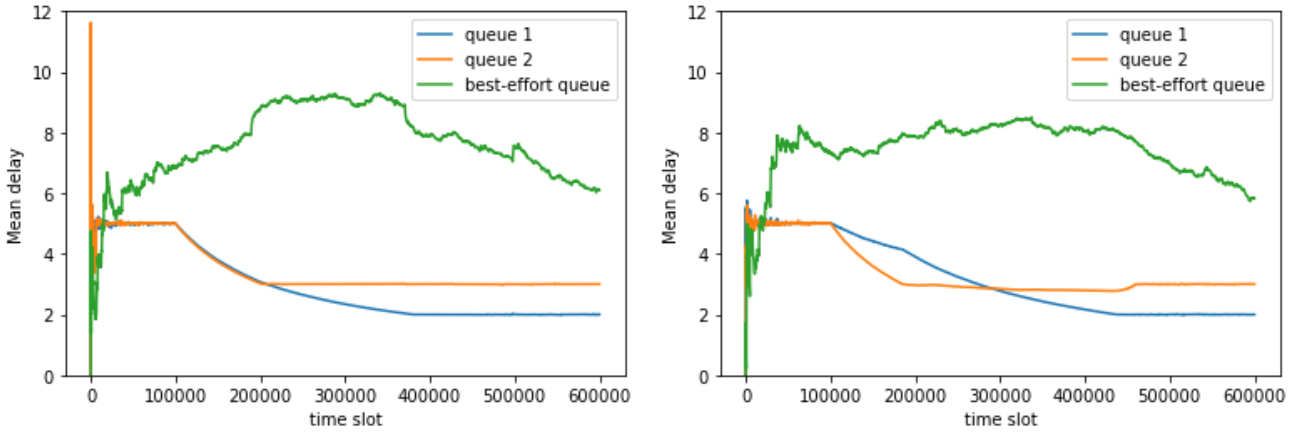
Parameter	Batch	ϵ-greedy
Discount factor γ	0.5	0.5
Reward constant C_1	50	50
Penalty constant C_2	20	20
State reward constant C_3	50	50
Others	$T = 1000$ $k = 3$	$\epsilon = 0.2$

(a) Parameters for algorithm

Queue	Arrival Rate [packets / timeslot]	Mean Delay Requirement [timeslots]
1	0.30 $\rightarrow$ 0.20	5 $\rightarrow$ 2
2	0.25$\rightarrow$ 0.20	5 $\rightarrow$ 3
3	0.40 $\rightarrow$ 0.55	best-effort

(b) Change of condition and requirement

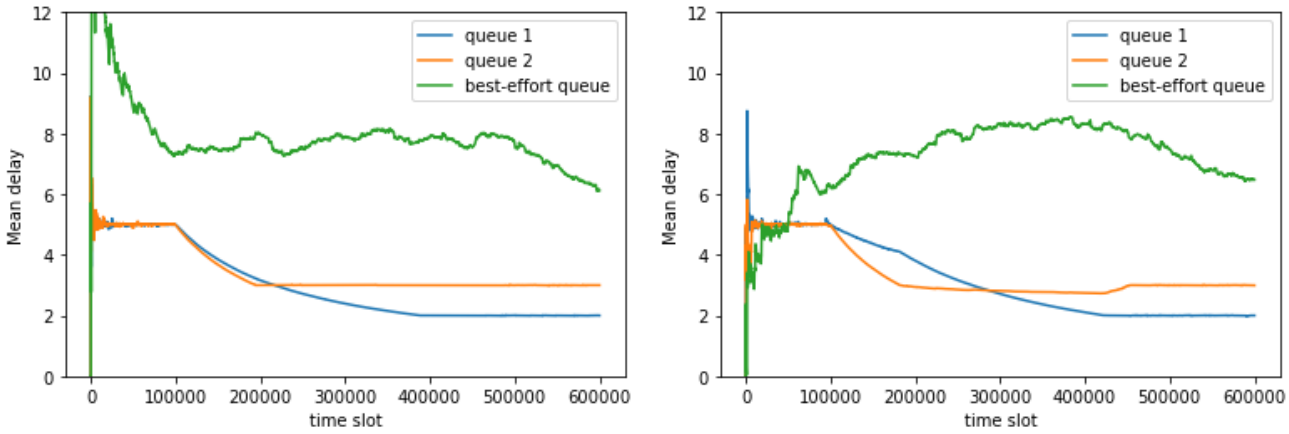
Table 1: Tables for Three Queues



(a) greedy search

(b) batch search

Figure 2: The results of **original** algorithm with 3 queues



(a) greedy search

(b) batch search

Figure 3: The results of **modified** algorithm with 3 queues

Figure 2 is the results of the original algorithm : (a) is greedy search. (b) is batch search.

Figure 3 is the results of the modified algorithm : (a) is greedy search. (b) is batch search.

For greedy search, the convergence time of Queue 2 in modified algorithm is slightly better than the one in original algorithm. But no obvious difference between Queue 1.

For batch search, both queues have no obvious improvements.

The reason of these results is that Greedy search is more sensitive to Q-value than Batch search. Batch search picks an action totally based on a certain distribution. For greedy search, it usually picks the best action. Therefore, only the algorithm with greedy search showed a small improvement.

3.2 Four Queues

Table 2(a) is the parameters for both original and modified algorithm. In modified algorithm, parameter **a** is set to -0.0005. Table 2(b) is the change of condition and requirement at time slot 100,000.

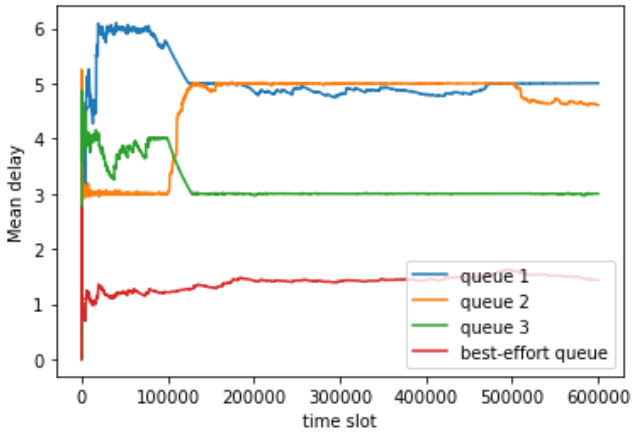
Parameter	Batch	ϵ-greedy
Discount factor γ	0.5	0.5
Reward constant C_1	50	50
Penalty constant C_2	20	20
State reward constant C_3	100	100
Others	$T = 1000$ $k = 3$	$\epsilon = 0.2$

(a) Parameters for algorithm

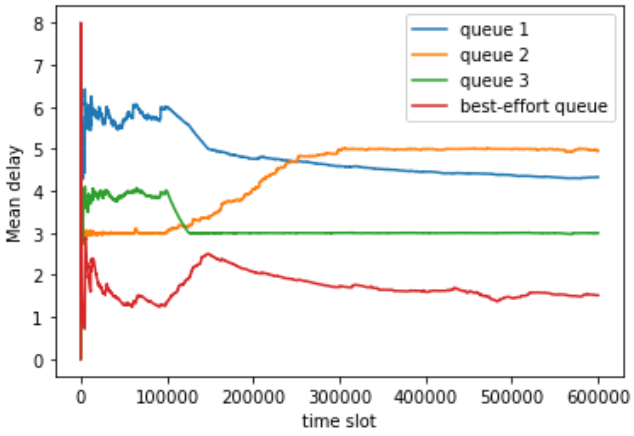
Queue	Arrival Rate [packets / timeslot]	Mean Delay Requirement [timeslots]
1	$0.25 \rightarrow 0.15$	$6 \rightarrow 5$
2	$0.20 \rightarrow 0.15$	$3 \rightarrow 5$
3	$0.15 \rightarrow 0.20$	$4 \rightarrow 3$
4	$0.30 \rightarrow 0.30$	best-effort

(b) Change of condition and requirement

Table 2: Tables for four Queues

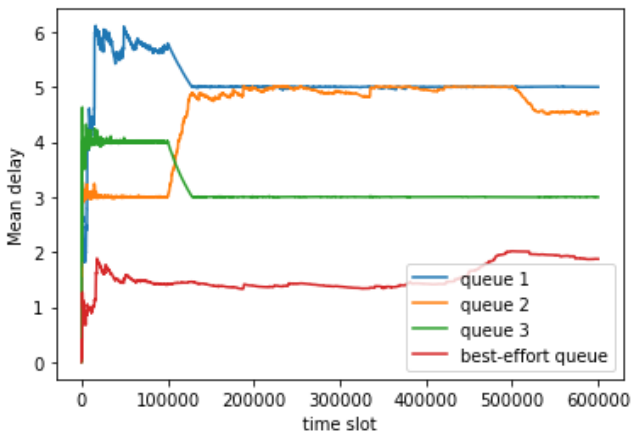


(a) greedy search

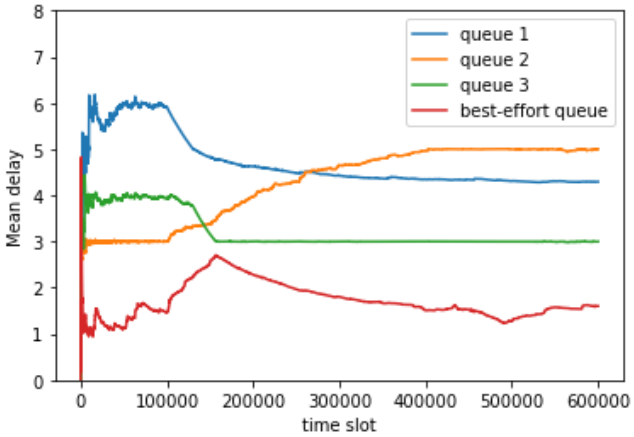


(b) batch search

Figure 4: The results of **original** algorithm with 4 queues



(a) greedy search



(b) batch search

Figure 5: The results of **modified** algorithm with 4 queues

Figure 4 is the results of the original algorithm : (a) is greedy search. (b) is batch search.

Figure 5 is the results of the modified algorithm : (a) is greedy search. (b) is batch search.

For greedy search, the difference of convergence time is too small to tell that there is an improvement.

For batch search, it's also hard to tell whether there is an improvement. For example, Queue 1 in Figure 5(b) converges faster than Queue 1 in Figure 4(b). But Queue 3 in Figure 5(b) is slower than the one in Figure 4(b). Furthermore, I ran both algorithms several times. All of the results show that if one queue converge faster, other queues would probably converge slower.

Also, both of batch search and greedy search use probability while picking an action so that results of different executions are hard to be the same. For three queues, algorithm can generate similar results just because the variation that algorithm can make is small(fewer queues). But for four queues, algorithm generates different result for each execution. There is no standard result for the algorithm with four queues. So it makes comparison difficult.

3.3 Five Queues

In this experiment, I only executed the original algorithm with greedy search. I didn't apply change and didn't examine modified algorithm because finding a set of parameter that doesn't cause starvation is difficult.

Table 3(a) is the parameters for original algorithm with greedy search. Table 3(b) is the initial condition and requirement at time slot 100,000.

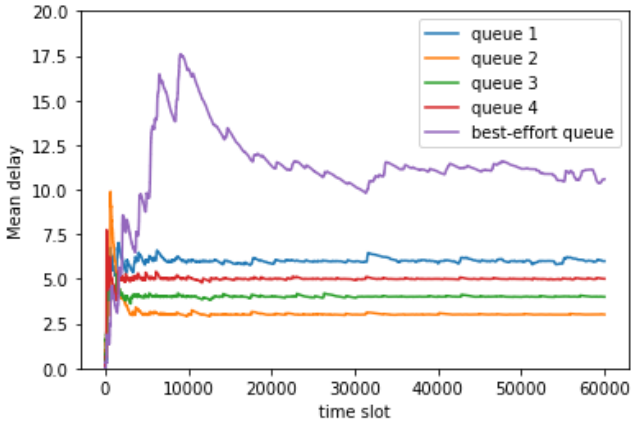
Parameter	ϵ-greedy
Discount factor γ	0.5
Reward constant C_1	50
Penalty constant C_2	20
State reward constant C_3	100
Others	$\epsilon = 0.2$

(a) Parameters for algorithm

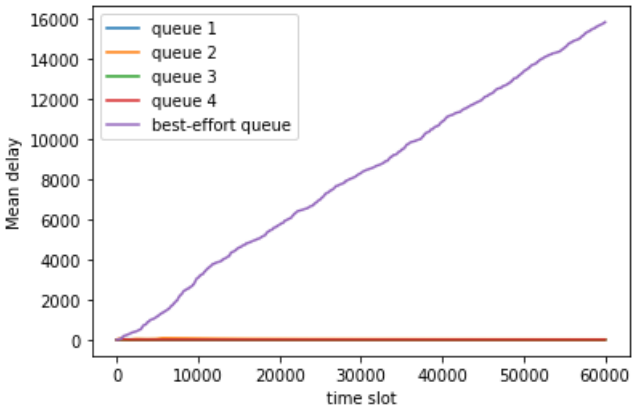
Queue	Arrival Rate [packets / timeslot]	Mean Delay Requirement [timeslots]
1	0.30	6
2	0.10	3
3	0.15	4
4	0.20	5
5	0.20	best-effort

(b) Initial condition and requirement

Table 3: Tables for Five Queues



(a) greedy search



(b) Starvation

Figure 6: The results of original algorithm with 5 queues

Figure 6(a) is the result of original algorithm with greedy search. As you can see, the algorithm executed successfully. However, if I modify the arrival rate of Queue 2 and Queue 3 to 0.2,

starvation takes place and the result is shown as Figure 6(b).

The reason causing this problem is the assumption of this algorithm : in each time slot, **at most one** packet can be processed. While the sum of arrival rate of all queues is close to 1, which means that the total number of packets transmitted into queues can be probably more than one, the processor can not handle this situation and starvation happens.

3.4 Conclusion

For the modification of Reward function, it only shows a small improvement on convergence time with 3 Queues system. However, it seems that the modification doesn't work well on 4 queues system. For greedy search, no obvious improvement. For batch search, algorithm generating different result each time make comparison difficult. It needs a more precise way to evaluate its performance.

For extending the number of queues, this algorithm is not stable with 4 queues and 5 queues while executing. The goal of this algorithm is to adapt change of condition and requirement. But a small change of arrival rate can cause starvation easily. I think 3 queues and 4 queues with low arrival rate is the limitation of this algorithm. However, this can be solved by modifying its assumption. Just let the system process two packets in a time slot by adding one more processor. Now, this algorithm can handle 4 queues, 5 queues and even more queues.

4 Source code

4.1 Language

The code was done in Python 3.8.8.

4.2 Libraries

- Matplotlib : Show the result in the form of graph.

4.3 Code Description

Each Queue has its own arrival rate. But Best-effort Queue don't have mean delay requirements. There are two classes in my code. First class is **Queue** :

- Class **Queue** is the queue for a traffic flow in the system.
- Attributes :
    1. Class **Queue** is embedded with a python build-in queue which stores packets.
    2. Because the system is focused on delay time, I simplified packet to be just a integer that denotes the time slot when it is put into queue.
    3. Attributes arrival rate and delay requirement are the conditions and requirements for this **Queue**.
    4. Another python build-in queue stores each delay time of the last P packets and it is for computing measured mean queuing delay.
- Function **change** : Modify attributes - arrival rate and delay requirement.

- Function **enqueue** : Put a packet into queue with its probability arrival rate. Randomly generate a number between 0 and 1. If less than arrival rate, put a packet into queue; otherwise, get rid of this packet.
- Function **dequeue** : When a packet is extracted from queue, compute and store the delay time of this packet. Then, update mean delay time over the last **P** packets.

Second class is **System** :

- Class **System** is the simulation of the structure as shown in Figure 1.
- Attributes :
  1. A **System** has three **Queues** in it.
  2. $Q(s, a)$: An array called Q-table is for Q-learning algorithm. Q-table is a 2-dimensional array which row is state and column is action.
  3. Parameters for **reward**, **ϵ-greedy search** and **batch search**.
     - For **batch search**, there is an additional 2-dimensional array which is the same size as Q-table. It stores probability $P(a_i|s)$ of each $Q(s, a_i)$ pair.
  4. For statistics, lists store measured mean delay of each Queue respectively at each time slot. At the end of program, lists are used to show the result in the form of graph.
- Function **Qlearning_greedy** and **Qlearning_batch** : Algorithm is implemented in them. ϵ-Greedy and Batch are different ways to choose an action given a state **s**.
  - ϵ-Greedy search : Given a state **s**, choose an action **a** such that $Q(s, a)$ has the highest value compared to other actions. But with a probability ϵ, it'll choose randomly from remaining actions.
  - Batch search : Given a state **s**, choose an action **a** at random according to a probability distribution : $P(a_i|s) = \frac{k^{Q(s,a_i)}}{\sum_j k^{Q(s,a_j)}}$, where $k > 0$. In algorithm, this distribution will be updated based on newly updated Q-table every T iterations.
- Function **all_queue_check** : Check whether all queues are empty.
- Function **action_greedy** : Choose an action based on ϵ-Greedy search. This function is for Function **Qlearning_greedy**.
- Function **check_action_greedy** : After choosing an action, make sure that the queue is not empty. If the queue is empty, choose the action having second large value in Q-table and so on. The queue for new action can't be empty either. This function is for Function **Qlearning_greedy**.
- Function **action_batch** : Choose an action based on Batch search. This function is for Function **Qlearning_batch**.
- Function **check_action_batch** : After choosing an action, make sure that the queue is not empty. If the queue is empty, choose another action at random. The queue for new action can't be empty either. This function is for Function **Qlearning_batch**.
- Function **do_action** : Execute the action. First, extract a packet from the specific queue. Then, compute the delay time of this packet and update measured mean delay.

- Function **next_state** : After executing action, compute the next state.
- Function **reward** : After executing action, compute reward.

Finally, I use **matplotlib** to show statistics (lists in **System**) in the form of graph.

4.4 Algorithm

The algorithm is based on Q-learning which is a kind of reinforcement learning. There are two functions implemented with the algorithm : Function **Qlearning_greedy** is based on **ϵ-Greedy search** and **Qlearning_batch** is based on **Batch search**. The differences between them are step 6 and 7.

The step is as following :

1. Initialize Q-table $Q(s, a)$ to all 0 and time slot **t** to 0.
2. Check whether to change the conditions and requirements for queues.
3. One packet is transmitted to per queue. (Function **enqueue** in class **Queue**).
4. Store measured mean queuing delay of each queue at this time slot. (Stored in lists).
5. Check if all the queues are empty. If so, jump to step 13.
6. Choose an action based on ϵ-Greedy search or Batch search. (Function **action_greedy** and **action_batch** in class **System**).
7. After choosing an action, check whether the queue is empty. If so, choose another action. (Function **check_action_greedy** and **check_action_batch** in class **System**).
8. Execute the action. (Function **do_action** in class **System**).
9. Compute next state s_{t+1}. (Function **next_state** in class **System**).
10. Compute reward. (Function **reward** in class **System**).
11. Update Q-table based on equation :

$$Q(s_t, a_t) = reward + \gamma \max_{a'} Q(s_{t+1}, a')$$

12. Move to next state s_{t+1}.
13. Time slot **t** plus 1.
14. Repeat step 2 to 13 until run out of time (the total number of iterations).

References

[1] Herman L. Ferrá, Ken Lau, C. Leckie, and Anderson Tang. "applying reinforcement learning to packet scheduling in routers". *IAAI*, 2003. vol. 141, pp. 247-254, 2018.