Reinforcement Learning Algorithms for queueing Networks

Tanishq Aggarwal (ta335@cornell.edu)
Trang H. Tran (htt27@cornell.edu)

Spring 2021—ORIE 6591

1 Introduction

The optimal control problem for stochastic queueing networks is one of the most important problems in operations research. These queueing networks have many applications in different areas, such as manufacturing systems, communication networks and transportation.

A queueing system consists of one or more service stations for serving customers. The customers arrive and may wait at some buffer if needed. Then they either are rejected or get the service, follow their route and go home when every service is completed. In this paper, we only consider the case when the inter-arrival times and service times are exponentially distributed. In that case, the queueing systems can be modelled as Markov decision processes (MDP) using uniformization technique.

When the state space is finite and the system is in low dimension, a MDP problem can be solved effectively using value iteration or policy iteration. In order to solve more complicated problems, many other approximate dynamic programming (ADP) methods have been proposed. For example, the reference [7] computes policies via approximate linear programming (ALP), that is approximating the differential cost by a linear form. Other approaches from reinforcement learning are advanced policy gradient (APG) methods, such as Trust Region policy optimization (TRPO) [4] and Proximal policy optimization (PPO) [6]. The papers [4] and [6] propose an optimization algorithm to maximize the expected discounted reward for the reinforcement learning problem. Their practical methods are based on some motivation from theoretical results (Theorem 1 in [4] - Monotonic improvement guarantee). In each policy iteration, their algorithms aim to minimize the surrogate loss function while keeping the changes small (controlling the distance between the old policy and the new update).

In contrast, the authors of [2] investigate the optimization problem of long-run average cost for the queueing systems. This problem is more difficult than the discounted setting mainly because the stability of queueing systems is not always guaranteed, hence the objective function may not converge. The authors of [2] have proposed a drift condition which is sufficient for the existence of a stationary distribution for the Markov chain (the MDP with a specific policy). Based on this condition, they prove a result on the improvement of the policy. Finally, they extend the theoretical framework of PPO methods for the long-run average cost, and specifically for the multiclass queueing networks.

In this paper, we produce a service policy for a set of queueing networks via *reinforcement learning* using PPO and PPO-like methods. We first model a set of queueing networks of interest (which are described in greater detail in the next section) in OpenAI gym environments, then use the vanilla PPO-clip algorithm to get an average reward for each queueing model. The vanilla algorithm (with the generalized advantage estimator) is best suited for infinite-horizon discounted MDP problems, so we use an advantage estimator designed for average-cost MDP problems in its place.

We draw a comparison of our results to a randomized non-idling policy. While our results are generally poorer than that of the randomized policy, we present this work as-is as a test of an implementation of a PPO-based algorithm designed for average-cost problems, and have indeed found that for some queueing networks, the PPO-derived policy is superior to randomization.[1]

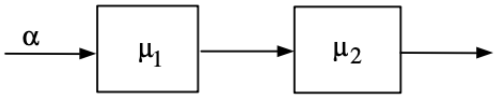
[1] Our implementation is in https://github.coecis.cornell.edu/ta335/queueing-networks.

2 The Queueing Networks

In this section, we formulate the reinforcement learning problem for queueing networks. We start with a simple and straightforward example, the two-series queue. Then we describe the common queueing networks model and our implementation as an OpenAI Gym environment.

2.1 The two-series queue

The two-series queue is a simple model with two stations where customers arrive and get the service at station 1, then move to station 2 and leave the system after getting the services. We assume that the arrival times follow Poisson processes at rate α, and the processing time at the two stations are μ_1 and μ_2 respectively. The process flow is described in the following diagram:



We assume a decision time occurs when a new customer arrives or a service is completed. At each decision time, the system choose an action to proceed for each server. For this model, each server has two actions: Idle or Work. They are allowed to choose idling even if there are customers waiting at each queue.

Now we are ready to formulate a reinforcement learning problem. Our state space describes the number of people in the system: $s(t) = (s_1(t), s_2(t))$ where $s_i(t)$ denote the number of customers (waiting and being served) in station i. We observe that the state space is countably infinite. In our experiments, we truncate the state space and every stations has limited capacity. There are four possible actions in this system:

0. Server 1 idle / Server 2 idle
1. Server 1 idle / Server 2 work
2. Server 1 work / Server 2 idle
3. Server 1 work / Server 2 work

At each time t, a stationary policy maps the current state $s(t)$ to a probability distribution over the action space, and the next action is sampled based on this distribution. Our objective is to find a stationary policy that minimizes the long-run average holding cost in the network:

$$\lim_{T\to\infty} \frac{1}{T}\int_0^T (s(t)^\top c)dt = \lim_{T\to\infty} \frac{1}{T}\int_0^T (c_1 s_1(t) + c_2 s_2(t))dt,$$

where c is the cost vector that describe the holding cost for each person in each queue. Since the inter-arrival times and service times are exponential, we adopt the uniformization technique and change the problem to the discrete-time setting [2]. Now letting $s^{(k)}$ be the system state at k-th time in the new uniformized framework, our objective function becomes:

$$\lim_{T\to\infty} \frac{1}{T}\sum_{k=0}^{T} (s^{(k)\top} c) = \lim_{T\to\infty} \frac{1}{T}\sum_{k=0}^{T} (c_1 s_1^{(k)} + c_2 s_2^{(k)}).$$

Remark: The randomized policy can be not stable, in the sense that under such policy the long-run average cost function may not converge. Therefore in this project, we compare our algorithm with a randomized non-idling policy instead. It is easily seen that the optimal policy for the two-series queue model is simply non-idling (a policy that let two servers work every time), and the problem is straightforward to solve. However we keep this model in this project because it provides a convenient tool to test and to investigate the performance of different algorithms. In contrast, for other complex models where there are more than one non-idle action, the optimization problem is much more difficult and the optimal policy depends on the underlying parameters of the model. This will be an interesting task for the reinforcement learning algorithms.

2.2 The common environment for queueing networks

There are different types of queueing networks, however they share some common properties. In this project, we create a common Gym environment for a generic queueing network. This generic class includes the 2-Series Queue, N Queue, Arrival Routing model, Rybko-Stolyar queue, and many other networks. We formulate these specific queueing models as the subclasses of the common environment. The details of how the models differ are relegated to the appendix. In this section, we describe the common environment for these queueing models.

State space: The queueing networks have state space $\mathbb{Z}_+^n$, where n is the number of queues in the system. The parameter n is further specified in each model. As in the two series queues, we also truncate the state space and the upper bound for each element follows the reference [7].

Action space: The action space is defined individually for each model. For example, the two-series queues has four actions as described in the previous subsection.

Reward: For each model, we have a cost vector $c \in \mathbb{R}^n$ which denotes the holding cost for every queue. Hence the reward is $-c^\top s$, where s is the current state.

Transitions: There are two types of activities in the system: arrivals and service completions. The arrivals rates are encoded using vector α and $\rho_{\max}$, where $\rho_{\max}$ describes the traffic intensity. In the other hand, the service rate are encoded using vector μ. Hence the corresponding transitions happen when one of the activities happen in the system: arrivals of jobs or completions of services. The specific translation is specified in the particular model.

The following table summarizes the parameters for our queueing networks. (Source: [7]). We dropped the implementation of the 4-series queue from the original reference because it is not a particularly interesting problem (the optimal policy is to always service the queues). We have implemented a randomized policy for the 2-series queues and they serve as useful toy problems for testing our algorithms.

Networks	α	μ	c	$\rho_{\max}$
Two-Series Queue ($\mu_1 > \mu_2$)	1	1.5, 1.25	1, 2	0.8
Two-Series Queue ($\mu_1 < \mu_2$)	1	1.15, 1.4	1, 3	0.87
N-Queue	1.2, 0.4	1, 1, 1	1, 1	0.8
Arrival Routing	1	0.65, 0.65	1, 2	0.77
Three Class Reentrant	0.1429	0.6, 0.16, 0.25	1, 1, 1	0.89
Rybko-Stolyar	0.0672, 0.0672	0.12, 0.12, 0.28, 0.28	1,..., 1	0.8
Six-Class Network	6/140, 6/140	1/4, 1, 1/8, 1/6, 1/2, 1/7	1,..., 1	0.6

Table 1: The input parameters for different models: α and $\rho_{\max}$ describe the arrivals rates and traffic intensity. μ describes the service rate and c is the holding cost for each queue.

3 Algorithms

3.1 Standard PPO

In this paper we implemented two versions for PPO algorithm. The original standard PPO-clip algorithm minimizes the discounted cost for infinite-horizon MDP problem. Each policy iteration of PPO consists of (1) a trajectory generation that produces many single episodes, (2) an advantage estimation step, which is used to solve (3) a optimization problem that minimizes the PPO-clip objective with respect to the policy parameter θ. These steps are repeated until the cost converges suitably. The details are in the below algorithm (from [1]).

It is worth noting that in Stable Baseline3, a policy is parameterized by a neural network where θ is typically the weights. Moreover, the PPO algorithms also use a neural network to approximate the value function for the system state. In order to solve the long-run average cost problem, we implement the standard framework from Stable Baselines3 and simply let the (discount) factor γ be 1.

Algorithm 1 PPO-Clip

1: Input: initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, ...$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), \quad g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

3.2 Custom PPO

We modify the algorithm by implementing a different version of the advantage estimation step to be suitable for average cost PPO. (Standard implementations, such as GAE-lambda [5], are written for infinite-horizon discounted cost.) The advantage estimate is given by [3] as:

$$A_\theta(s_t^k, a_t^k) = \sum_{i=0}^{T_0-1} (r(s_{t+i}^k, a_{t+i}^k) - \mu_\theta(r) + \hat{V}_\theta(s_{t+i+1}^k) - \hat{V}_\theta(s_{t+i}^k)) \qquad 0 \le t \le T - T_0$$

$$A_\theta(s_t^k, a_t^k) = \sum_{i=0}^{T-t-1} (r(s_{t+i}^k, a_{t+i}^k) - \mu_\theta(r) + \hat{V}_\theta(s_{t+i+1}^k) - \hat{V}_\theta(s_{t+i}^k)) \qquad T_0 - T \le t \le T$$

where T is the episode length, $T_0 < T$ is the *rollout horizon*, and k, t index an episode and a timestep, respectively. The advantage is a sum of T_0 steps, however for the last indexes where $T_0 - T \le t \le T$ the advantage is truncated to $T - t$ terms because the episode is limited to T.

In this estimation r is the reward function, and $\mu_\theta(r)$ is the *average reward*, computed by averaging the reward across all trajectories and all timesteps. This is motivated by the long-run cost approximation expression from [2]:

$$\widehat{\mu_\theta(r)} = \frac{1}{\sigma(n)} \sum_{k=0}^{\sigma(n)} s^{(k)\top} c,$$

where $\sigma(n)$ is the n-th time when regeneration state $x^* = 0$ is visited. We discard the consideration of regeneration cycles by simply increasing the horizon (so that many cycles are contained). We validate this assumption by the experiment in the next section that computes the regeneration cycle length for the initial state under a randomized non-idling policy.

Note that $\hat{V}_\theta$ is an estimate of the value function given the current policy θ. For each policy iteration, we compute an advantage estimate and hence the rewards-to-go for each state in the simulated trajectories. Then we approximate the value function using a neural network and the fit a new value function to rewards-to-go by regression on mean squared error.

3.3 Reducing Variance

The note in [3] also suggests an additional expression for the advantage estimate. It is similar as the above, except that we replace the value function estimate $\hat{V}_\theta(s^k_{t+i+1})$ with the expectation value of the one-step value change $\mathbb{E}_{s\sim P^\theta(\cdot|s^k_{t+i})}[\hat{V}_\theta(s)]$.

The new advantage estimate is:

$$\begin{aligned} A_\theta(s^k_t, a^k_t) =& r(s^k_t, a^k_t) - \mu(r) + \mathbb{E}_{s\sim P(\cdot|s^k_t,a^k_t)}[\hat{V}_\theta(s)] - \hat{V}_\theta(s^k_{t+i}) \\ &+ \sum_{i=1}^{T_0-1} (r_\theta(s^k_{t+i}) - \mu_\theta(r) + \mathbb{E}_{s\sim P^\theta(\cdot|s^k_{t+i})}[\hat{V}_\theta(s)] - \hat{V}_\theta(s^k_{t+i})) \qquad 0 \le t \le T - T_0, \end{aligned}$$

and similarly for $T_0 - T \le t \le T$.

This change requires knowing the transition probabilities P. Luckily, these are known to us for each model (since the models are implemented explicitly using known Poisson process probabilities). It was technically challenging to use this information, though, so at first we tried to approximate the expectation via a sampling process, e.g. we stepped $s^k_{t+i} \to s^i_{t+k+1}$ and said $\mathbb{E}_{s\sim P^\eta(\cdot|s^k_{t+i})}[\hat{V}^\eta(s)] \approx \hat{V}^\eta(s^k_{t+i+1})$. But even a modest number of samples (e.g. 100) made the algorithm too time-consuming. We then implemented an explicit analytical calculation of $\mathbb{E}_{s\sim P^\eta(\cdot|s^k_{t+i})}[\hat{V}^\eta(s)]$ using the probabilities we implemented in the models. We mention this part of our process as a warning to others who may try the same.

4 Numerical Experiments

4.1 Randomized Non-Idling Policy

We first implement a randomized, non-idling policy for each of these networks. We begin the initial state for all experiments at zero in all queues. In Table 2 we report the approximations of the average cost, obtained by clamping the length of an episode to $T = 20000$ steps and by averaging over 100 episodes. We also estimate the average length of regeneration cycles (i.e the number of time steps the chain needs to go back to initial zero state) for each random non-idling policy.

In estimating value functions for average-cost MDP problems, there can be issues with bias due to truncation of the infinite run time. We have addressed this concern by trying a variety of truncation times for each network ($T = 15000, 16000, \ldots 20000$) and have found that the values reported above are still within the uncertainty reported at those truncation times.

Networks	Average Cost (T = 20k, 100 episodes)	Regeneration Cycles
Two-Series Queue ($\mu_1 > \mu_2$)	4.71 ± 0.06	4.91 ± 18.46
Two-Series Queue ($\mu_1 < \mu_2$)	8.00 ± 0.06	9.68 ± 40.10
N-Queue	3.79 ± 0.06	9.29 ± 39.40
Arrival Routing	4.34 ± 0.04	4.91 ± 17.08
Three Class Reentrant	15.02 ± 0.39	206.81 ± 1250.83
Rybko-Stolyar	16.08 ± 0.83	267.84 ± 1876.97
Six-Class Network	9.09 ± 0.54	136.41 ± 1621.67

Table 2: The estimated long-run average cost and average length of regeneration cycles for different models.

Remark: Since the two-series queue is simple, we know that the average costs displayed in this table are optimal for that model. The other values give us an intuition of how the systems behave in non-idling mode.[2] We also note that the last three models are somehow more complicated and harder to analyze, as we can see from Table 2 that their regeneration cycles are usually longer than the simple models.

[2]Initially we aimed to compare this result with the reference [7]. However that paper calculate the cost in the continuous setting while the cost reported here is for the discrete case. Hence such comparison is not fair and we decided not to mention their results and setting in this paper.

4.2 Tuning Hyperparameters for Standard PPO

The standard PPO algorithm (that was adopted from OpenAI Spinning Up and Stable Baselines[3]) has many hyper-parameters, and we list some of the important factors here:

1. Options to parameterize the policy and value function, which can be a multi layer perceptron (MLP) or a convolutional neural network (CNN). Some parameters allow us to specify the amount and size of the hidden layers and how many of them are shared between the policy network and the value network.

2. Parameters to control the training process: the learning rate for PPO (default value is 3e-4), minibatch size, number of epoch when optimizing the surrogate loss,...

3. Parameters that control the episode generation: the number of steps to run for each environment per update n_steps (i.e. rollout buffer size is n_steps * n_envs where n_envs is number of environment copies running in parallel, default value is 2048), the total number of samples (for all environments) to train on (total_timesteps)

4. Parameters that describe the clipping process and surrogate loss function: clipping parameter (clip_range), clipping parameter for the value function (clip_range_vf, a parameter specific to the OpenAI implementation), entropy coefficient and value function coefficient for the loss calculation (ent_coef and vf_coef), and the maximum value for the gradient clipping (max_grad_norm)

In this paper, we use the MLP policy and implement PPO algorithms using most of the default hyper-parameter settings. We only change the number of steps to run for each environment per update and the total number of samples since these two numbers are much related to the generation of sample episodes. Hence this parameter should be investigated for different queueing models.

The estimated long-run average cost are shown in Table 3. We try $\gamma = 1$ and additionally $\gamma = 0.99999$ to see if the PPO-estimated policy and resultant cost are well-conditioned upon γ.

Networks	Average Cost ($\gamma = 1$)	Average Cost ($\gamma = 0.99999$)
Two-Series Queue ($\mu_1 > \mu_2$)	5.50 ± 0.61	364.64 ± 2.12
Two-Series Queue ($\mu_1 < \mu_2$)	368.13 ± 1.31	367.75 ± 1.33
N-Queue	93.72 ± 0.92	94.66 ± 1.11
Arrival Routing	2.81 ± 0.27	3.72 ± 0.23
Three Class Reentrant	38.32 ± 0.13	38.21 ± 0.08
Rybko-Stolyar	279.04 ± 14.34	292.01 ± 11.29
Six-Class Network	149.14 ± 7.31	149.99 ± 7.92

Table 3: Standard PPO algorithm: the estimated long-run average cost for different models. The number of steps to run for each environment per update is 10000 and the total number of samples (for all environments) to train on is 100000.

Remark: The two versions of Standard PPO produce similar results for most of the models. Surprisingly the choice $\gamma = 1$ gives a significantly better cost than $\gamma = 0.99999$ for the first two-series queue model where $\mu_1 > \mu_2$. However both the algorithms perform poorly for the second two-series queue model where $\mu_1 < \mu_2$. This fact is somehow disappointing because the two-series queue model is the most simple one. For other models, the performance is comparably similar for two choices of gamma.

We also observe that the resulting policies from PPO methods usually perform worse than the randomized non-idling policies for all models except the Arrival Routing. Since this is also a relatively simple model without idling actions, this result is reasonable and the PPO may perform well under that hyper-parameter setting. Hence we conclude that it is maybe hard to choose the good hyper-parameter to tune PPO methods to optimal, even for the simple models.

[3]The code is borrowed from the implementation of OpenAI Spinning Up and Stable Baselines `https://github.com/openai/spinningup/`, `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail` and `https://github.com/hill-a/stable-baselines`

4.3 Execution of Custom PPO

Remark: The performance of our custom PPO algorithms is shown in Table 4. The results from our custom algorithms are slightly worse than the standard PPO algorithm. The variance reduction version perform better for some models and also worse for some others.

Networks	Average Cost (Without Variance Reduction)	Average Cost (With Variance Reduction)
Two-Series Queue ($\mu_1 > \mu_2$)	362.91 ± 1.87	29.15 ± 21.03
Two-Series Queue ($\mu_1 < \mu_2$)	367.36 ± 1.59	394.14 ± 6.96
N-Queue	102.47 ± 9.80	102.69 ± 1.58
Arrival Routing	5.18 ± 0.64	3.19 ± 0.28
Three Class Reentrant	44.60 ± 3.09	39.09 ± 1.50
Rybko-Stolyar	573.02 ± 17.66	564.03 ± 14.42
Six-Class Network	154.08 ± 8.17	151.11 ± 9.46

Table 4: Custom PPO algorithm: the estimated long-run average cost for different models. The number of steps to run for each environment per update is default (2048) and the total number of samples (for all environments) to train on is 100000 and 10000 for each algorithms, respectively.

5 Further Discussion

In this paper, we implement some versions of PPO algorithms for queueing models. When the model is unknown to the algorithm, it is difficult to tune the hyper-parameter and find a good policy. From the experiment we have seen that the RL algorithms can find a better policy easier by taking the advantage of the knowledge about non-idlinng actions. When the transition probabilities is known, we can also implement a variance reduction technique to get a better approximation for each policy iteration.

Since the reinforcement learning task is not easy, there are still many open questions that were not answered in our paper. We discuss these challenges here:

1. Since every model is different and the PPO methods has a lot of input parameters, it is difficult to find a setting that helps the algorithm find the best policy. During this project we were able to test the algorithm using some different settings. We assume the regeneration cycles for a randomized non-idling policy can give us some knowledge on how to choose the time steps needed to train the algorithm. However due to time limit we were not able to test that.

2. It is unknown how to provide a good estimation for the advantage function for PPO methods. There are different ways to estimate the advantage function (with or without variance reduction). While the variance reduction technique may improve each policy step, it also requires a lot more computation. (Note that this computation can be used to sample more data in the setting without variance reduction). Hence there is a trade-off between the precision of estimation and computational resources.

3. It is natural to assume that the algorithm can perform better if more knowledge of the model is available. In this paper we were able to take advantage of the idling actions (by restricting to non-idling policies). However, it remains an open question how we should exploit the special structure of the queueing networks in general, and in particular the transition probabilities when the underlying rates are known.

We believe these are challenging open problems and will be interesting topics for future research.

References

[1] Proximal policy optimization. `https://spinningup.openai.com/en/latest/algorithms/ppo.html`.

[2] J. G. Dai and Mark Gluzman. Queueing network controls via deep reinforcement learning, 2020.

[3] J. G. Dai and Mark Gluzman. Average-cost mdp: Foundation and algorithms for trpo and ppo, 2021.

[4] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. Trust region policy optimization, 2017.

[5] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation, 2018.

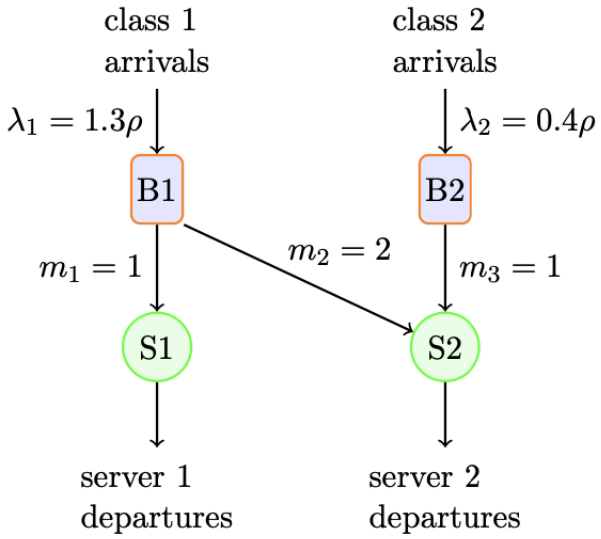
[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms, 2017.

[7] Michael H. Veatch. Approximate linear programming for networks: Average cost bounds. *Computers & Operations Research*, 63:32–45, 2015.

6 Appendix

In the next subsections, we describe the particular settings for each queueing models.

6.1 N Queue

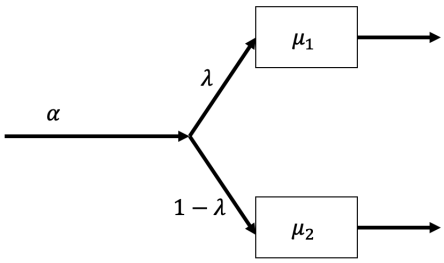


1. State space: $\mathbb{Z}_+^2$ denotes the number of jobs in queue 1 and queue 2. Truncation: $(100, 100)$.
2. Activities and actions:

```
# Activity table:
# 0 - Arrival of class 1 job
# 1 - Arrival of class 2 job
# 2 - Finished processing of class 1 job by server 1
# 3 - Finished processing of class 1 job by server 2
# 4 - Finished processing of class 2 job by server 2

# Action table:
# 0 - Server 1 idle            / Server 2 idle
# 1 - Server 1 idle            / Server 2 work on class 1
# 2 - Server 1 idle            / Server 2 work on class 2
# 3 - Server 1 work on class 1 / Server 2 idle
# 4 - Server 1 work on class 1 / Server 2 work on class 1
# 5 - Server 1 work on class 1 / Server 2 work on class 2
```

6.2 Arrival Routing



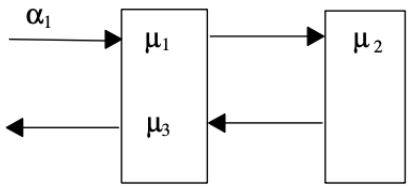
1. State space: $\mathbb{Z}_+^2$ denotes the number of jobs in the queue for processor 1 and processor 2. The state space truncation is not specified in the paper, but we assume it to be $(100, 100)$. In practice, the average cost obtained under a randomized policy does not seem to be affected much by increasing this state space truncation limit.

2. Activities and actions:

```
# Activity table:
# 0 - Arrival at rate lambda
# 1 - Completion of job by processor 1
# 2 - Completion of job by processor 2

# Action: a probability lambda in the range [0, 1], which controls
# the routing of a job to server 1 or server 2.
```

6.3 Three Class Re-entrant Line



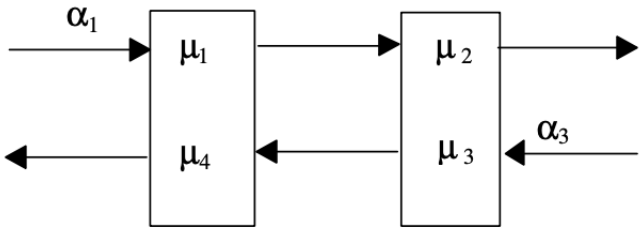
1. State space: $\mathbb{Z}_+^3$ denotes the number of jobs in queue 1 and queue 2. Truncation: $(40, 80, 20)$.

2. Activities and actions:

```
# Activity table:
# 0 - job arrives
# 1 - server 1 finishes incoming
# 2 - server 2 finishes
# 3 - server 1 finishes reentrant

# Action table:
# 0 - Server 1 idle              / Server 2 idle
# 1 - Server 1 idle              / Server 2 work
# 2 - Server 1 work on incoming  / Server 2 idle
# 3 - Server 1 work on incoming  / Server 2 work
# 4 - Server 1 work on reentrant / Server 2 idle
# 5 - Server 1 work on reentrant / Server 2 work
```

6.4 Rybko-Stolyar



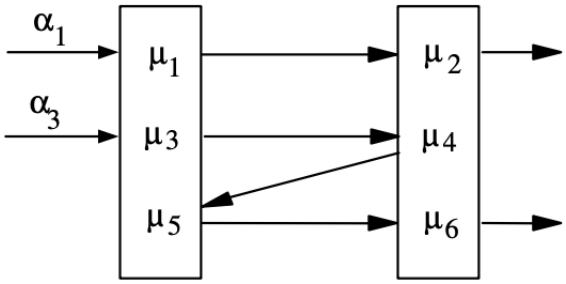
1. State space: $\mathbb{Z}_+^4$ denotes the number of jobs in four queues. Truncation: 500 jobs in each queue.

2. Activities and actions:

```
# Activity table:
# 0 - Arrival of class 1 job
# 1 - Arrival of class 3 job
# 2 - Finished processing of class 1 job by server 1
# 3 - Finished processing of class 2 job by server 2
# 4 - Finished processing of class 3 job by server 2
# 5 - Finished processing of class 4 job by server 1

# Action table:
# 0 - Server 1 idle            / Server 2 idle
# 1 - Server 1 idle            / Server 2 work on class 2
# 2 - Server 1 idle            / Server 2 work on class 3
# 3 - Server 1 work on class 1 / Server 2 idle
# 4 - Server 1 work on class 1 / Server 2 work on class 2
# 5 - Server 1 work on class 1 / Server 2 work on class 3
# 6 - Server 1 work on class 4 / Server 2 idle
# 7 - Server 1 work on class 4 / Server 2 work on class 2
# 8 - Server 1 work on class 4 / Server 2 work on class 3
```

6.5 Six Class Queue



1. State space: $\mathbb{Z}_+^6$ denotes the number of jobs in six queues. Truncation: 500 jobs in each queue.
2. Activities and actions:

```
# Activity table:
# 0 - Arrival of class 1 job
# 1 - Arrival of class 3 job
# 2 - Finished processing of class 1 job by server 1
# 3 - Finished processing of class 2 job by server 2
# 4 - Finished processing of class 3 job by server 1
# 5 - Finished processing of class 4 job by server 2
# 6 - Finished processing of class 5 job by server 1
# 7 - Finished processing of class 6 job by server 2

# Action table:
# 0  - Server 1 idle            / Server 2 idle
# 1  - Server 1 idle            / Server 2 work on class 2
# 2  - Server 1 idle            / Server 2 work on class 4
# 3  - Server 1 idle            / Server 2 work on class 6
# 4  - Server 1 work on class 1 / Server 2 idle
# 5  - Server 1 work on class 1 / Server 2 work on class 2
# 6  - Server 1 work on class 1 / Server 2 work on class 4
# 7  - Server 1 work on class 1 / Server 2 work on class 6
# 8  - Server 1 work on class 3 / Server 2 idle
# 9  - Server 1 work on class 3 / Server 2 work on class 2
# 10 - Server 1 work on class 3 / Server 2 work on class 4
# 11 - Server 1 work on class 3 / Server 2 work on class 6
# 12 - Server 1 work on class 5 / Server 2 idle
# 13 - Server 1 work on class 5 / Server 2 work on class 2
# 14 - Server 1 work on class 5 / Server 2 work on class 4
# 15 - Server 1 work on class 5 / Server 2 work on class 6
```