Applying Reinforcement Learning to Basic Routing Problem

Sigurður Gauti Samúelsson and Esa Hyytiä

Department of Computer Science,
University of Iceland

Abstract. Routing jobs to parallel servers is a common and important task in today's computer and communication systems. As each routing decision affects the jobs arriving later, determining the (near) optimal decisions is non-trivial. In this paper, we apply reinforcement learning techniques to the job routing problem with heterogeneous servers and a general cost structure. We study the convergence of the reinforcement learning to a near-optimal policy (that we can determine by other means), and compare its performance against heuristic policies such as Join-the-Shortest-Queue (JSQ) and Shortest-Expected-Delay (SED).

Keywords: Job dispatching; Task assignment; Machine learning; Reinforcement learning; Value function; Parallel servers

1 Introduction

Routing jobs to parallel servers has been a long standing problem class for queueing theory. The problem was first studied by Haight already in 1958 [1]. Today, the same problem arises in many new contexts. For example, when routing data traffic in Internet, alternative routes can be modelled as parallel servers. Similarly, in cloud computing, each task needs to be assigned to one of the available servers. In supercomputing, the time scales are longer but the same fundamental question appears. Moreover, the heterogeneity of computing hardware is increasing both in large-scale systems comprising several (thousands of) physical computers, as well as within a single physical device (cf. GPUs vs. CPUs, and new heterogeneous multi-core architectures for mobile devices)

In this paper, we study an elementary routing (or dispatching) problem to heterogeneous parallel servers subject to a large class of cost structures. Both job inter-arrival times and service times are assumed to be exponentially distributed. The state information is the number of jobs in each server. One of the most popular routing policies is Join-the-Shortest-Queue (JSQ), which chooses the server with the fewest jobs. JSQ has been shown to be optimal in some specific cases, but, especially when the service rates are unequal, the exact analysis of the system becomes surprisingly tedious.

The optimization problem for the optimal routing falls in the category of Markov decision processes (MDPs). However, our state space is countably infinite and optimal routing decisions are difficult to determine. We apply reinforcement

learning techniques to this problem [2]. The infinite state space remains as a problem as it is impossible to visit every state (preferably multiple times) in any finite time, and therefore learning the optimal action for every state is impossible.

We work around this by focusing on a finite subset of states where decisions presumably matter the most, and rely on an appropriately chosen heuristic routing elsewhere. Effectively, similarly as in [3], we aggregate states so that the resulting optimization problem has a finite set of states, and then apply the reinforcement learning in this state space. If a good heuristic policy is sufficient, then the first policy iteration step (FPI) can be considered. In this case, it is often possible to determine the corresponding value function analytically given the basic policy is static and the system decomposes [4, 5]. The value function can also be estimated by a set of short Monte Carlo simulations at each decision point [6]. In this case, the basic policy can be dynamic.

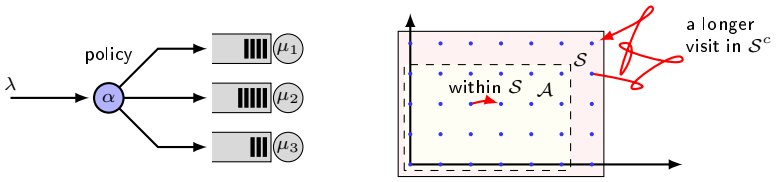
The main contributions of this paper are as follows: First, we show that the heuristic partitioning of the original infinite state space into two classes yields a computationally efficient optimization problem for which machine learning techniques can be applied. Second, we experiment with different learning parameters to gain insight on how fast a near-optimal policy can be learned. This is important especially when the system parameters evolve in time.

The rest of the paper is organized as follows. The routing problem is formally defined in Section 2, to which the reinforcement learning technique is in Section 3. Section 4 gives some numerical examples, and Section 5 concludes the paper.

2 Model

The model for a parallel server system, illustrated in Figure 1(a), is as follows:

1. Jobs arrive according to a Poisson process with rate λ.
2. Jobs are routed immediately upon arrival to one of the K servers, where the service time in server i is exponentially distributed with parameter μ_i.
3. We consider the so-called *number-aware* setting, where state $\mathbf{n} = (n_1, \ldots, n_K)$ means that server i has n_i jobs. The state space is thus $\mathcal{X} = \mathbb{N}^K$, where $\mathbb{N}$ denotes the set of natural numbers, $\mathbb{N} = \{0, 1, 2, \ldots\}$.
4. Each state $\mathbf{n}$ has an associated *cost rate* $r_{\mathbf{n}}$ at which the system incurs costs when in state $\mathbf{n}$.
   (a) For the mean response time metric, the cost rate is the number of jobs,
   $$r_{\mathbf{n}} = n_1 + \ldots + n_K.$$
   (b) The costs can also deter the use of some servers by using server-specific weights w_i for response time,
   $$r_{\mathbf{n}} = \sum_i w_i n_i.$$
   (c) If servers incur costs when busy, we have running cost rates,
   $$r^{(r)}_{\mathbf{n}} = \sum_i w^{(r)}_i \, \mathbf{1}(n_i > 0).$$



(a) Three server system (b) Sample transitions

Fig. 1. Partitioning the infinite state space by the finite subset $\mathcal{S}$. Visits outside $\mathcal{S}$ may involve several jobs arriving and departing before the state of the system returns to $\mathcal{S}$.

Thus, serving a job in server i incurs an average cost of $w_i^{(r)}/\mu_i$. Note that this serves also as an elementary model for energy consumption.

(d) Similarly, with very minor modifications, we can also introduce *admission costs* $c_{i,n}$ incurred when a job enters server i in state n (cf. PASTA).

3 Learning The Optimal Routing Policy

Our aim is to devise a *machine learning* procedure that determines the optimal policy. As mentioned, the state space of the system is infinite, which tends to be a problem as it is not possible to visit every state in finite time. However, often important routing decisions need to be made only in some relatively small subset and elsewhere an appropriate heuristic rule such as Join-the-Shortest-Queue (JSQ) does an adequate job. In particular, decisions near the origin (empty system) are typically critical for the performance.

Therefore, similarly as in [3], we limit our focus on a finite set of states $\mathcal{S} \subset \mathcal{X}$. For example, with two server systems we can consider $n \times n$ boxes,

$$\mathcal{S}_n = \{(i,j) \mid i < n,\, j < n\}.$$

However, unlike in [3], we are not limited to some specific shapes but $\mathcal{S}$ can be arbitrary finite subset of $\mathcal{X}$. The idea is that we will determine the so-called value function only for states in $\mathcal{S}$. Consequently, $\mathcal{S}$ induces the action set $\mathcal{A}$ that includes those states $\mathbf{n}$ for which all routing decisions lead to a state in $\mathcal{S}$,

$$\mathcal{A} = \{\mathbf{n} : \mathbf{n} + \mathbf{e}_i \in \mathcal{S}\ \forall\, i\},$$

where $\mathbf{e}_i$ is a vector with the i^{th} component one and all other zero. Hence, given the value function in $\mathcal{S}$ is known, it defines the corresponding policy in $\mathcal{A}$. Elsewhere, in $\mathcal{A}^c$, we assume a fixed heuristic rule such as RND (random split), JSQ or SED (shortest expected delay).[1].

[1] RND (random) chooses the server independently in random using some probabilities p_k, JSQ chooses the queue with the least number of jobs, and SED the queue with the shortest expected response time, i.e., the admission cost to queue i is $(n_i+1)/\mu_i$.

The division of the state space, induced by $\mathcal{S}$, is illustrated in Figure 1(b). Note that there are (i) direct transitions within $\mathcal{S}$, as well as, (ii) longer visits outside $\mathcal{S}$. For example, long busy periods with many jobs correspond to long visits outside $\mathcal{S}$. Eventually, after a random time T, a stable system still returns to $\mathcal{S}$. The costs C incurred during time T can obviously be high. Nonetheless, both $\mathbb{E}[T]$ and $\mathbb{E}[C]$ can be estimated by straightforward simulations. Effectively, we view states in $\mathcal{S}^c$ as one or more *aggregated super state(s)*: the system "escapes" from $\mathcal{S}$ to somewhere in $\mathcal{S}^c$, and then returns after a random time.

3.1 Learning the Value Function

Let $v(\mathbf{n})$ denote the value function with a fixed routing,

$$v(\mathbf{n}) \triangleq \lim_{t\to\infty} \mathbb{E}[V(\mathbf{n},t) - rt],$$

where $V(\mathbf{n},t)$ denotes the cost incurred during time $(0,t)$ when initially in state $\mathbf{n}$ and r is the long-run mean cost rate (assumed to be finite). The value function for any state $\mathbf{n} \in \mathcal{S}$ satisfies (cf. Howard's and Bellman's equations [7, 8]),

$$v(\mathbf{n}) = c(\mathbf{n},\mathcal{S}) - t(\mathbf{n},\mathcal{S}) \cdot r + \sum_{\mathbf{m}\in\mathcal{S}} p_{\mathcal{S}}(\mathbf{n},\mathbf{m}) \cdot v(\mathbf{m}), \tag{1}$$

where $c(\mathbf{n},\mathcal{S})$ denotes the average costs incurred since arriving to state $\mathbf{n}$ until the system moves to a (new) state in $\mathcal{S}$ (that can be the same state $\mathbf{n}$ if the system first moves to a state in $\mathcal{S}^c$), $t(\mathbf{n},\mathcal{S})$ denotes the corresponding mean time interval, and $p_{\mathcal{S}}(\mathbf{n},\mathbf{m})$ is the probability that the next state (in $\mathcal{S}$) is $\mathbf{m}$. Equation (1) is the basis for the Reinforcement learning algorithm aiming to find the *optimal control in* $\mathcal{A}$.

Suppose first that the routing is fixed $\omega_0(\mathbf{n})$, and the aim is to determine (estimate) the value function in $\mathcal{S}$ corresponding to $\omega_0(\mathbf{n})$. Let $\mathbf{n}_j \in \mathcal{S}$ denote the j^{th} state visited in $\mathcal{S}$, i.e., $\mathbf{n}_j$ is a sequence of states the system visits from which the states outside $\mathcal{S}$ have been omitted. Then the learning equations for the value function are

$$\begin{aligned} C &\leftarrow C + c_j, \\ T &\leftarrow T + t_j, \\ r &\leftarrow C/T, \\ v(\mathbf{n}_j) &\leftarrow (1-\alpha_j) v(\mathbf{n}_j) + \alpha_j \left[c_j - t_j \cdot r + v(\mathbf{n}_{j+1})\right], \end{aligned} \tag{2}$$

where c_j is the costs incurred since entering state $\mathbf{n}_j$ until reaching state $\mathbf{n}_{j+1}$, t_j is the corresponding time interval, and α_j is the learning rate at step j. The first three equations provide an estimate for the mean cost rate r, and the last equation updates the estimate for the value function. Initially, the learning rate can be set a high value, close to one, and then, as time goes by, it is decreased gradually to zero (or a value close to zero). For example, one can use

$$\alpha_j = e^{-\beta j},$$

where $\beta > 0$ is an appropriately chosen constant. If the system parameters keep on changing, as often is the case in practice, then one can use some fixed small value, e.g., $\alpha = 0.1$.

As the constant offset in the value function is irrelevant (for routing decisions), we can fix it, e.g., so that $v(0) = 0$. In this case, whenever empty state $\mathbf{n} = (0, \ldots, 0)$ is updated, we immediately subtract its new value from all states,

$$v(\mathbf{n}) \leftarrow v(\mathbf{n}) - v(0), \qquad \forall\, \mathbf{n}. \tag{3}$$

Equation (2), combined with (3), learns the value function for states $\mathcal{S}$ for a given routing policy.

3.2 Policy Improvement

Given the value function, one policy iteration round can be carried out, yielding a new routing policy that is better than $\omega_0(\mathbf{n})$ (unless $\omega_0(\mathbf{n})$ was already optimal). This is known as the *first policy iteration* (FPI). In our case, when a job arrives in state $\mathbf{n} \in \mathcal{A}$, the improved policy routes the job to server j such that

$$v(\mathbf{n} + \mathbf{e}_j) \leq v(\mathbf{n} + \mathbf{e}_i) \quad \forall i.$$

Possible ties can be resolved, e.g., in random. Letting $v_0(\mathbf{n})$ denote the value function corresponding to $\omega_0(\mathbf{n})$, the improved routing policy is

$$\omega_1(\mathbf{n}) \triangleq \operatorname*{argmin}_j v_0(\mathbf{n} + \mathbf{e}_j).$$

Example 1. Suppose we have $K = 2$ identical servers, $\mu_1 = \mu_2 = \mu$, and arrival rate $\lambda < 2\mu$. The (basic) routing policy is uniform random split routing a job to server 1 with probability of 0.5, and otherwise to server 2. As the routing decision does not depend on the state of the system, the routing policy is static and the value function decomposes,

$$v(\mathbf{n}) = v_1(n_1) + v_2(n2).$$

Suppose further that the cost structure is the response time metric. Then

$$v_i(n) = \frac{n(n+1)}{2(\mu_i - \lambda_i)} - \frac{\lambda\mu}{(\mu - \lambda)^3},$$

where now $\mu_i = 1$ and $\lambda_i = 0.5 \cdot \lambda$. As the constant in the value functions is irrelevant, we can as well choose $v(0, 0) = 0$, yielding

$$v(\mathbf{n}) = \frac{n_1(n_1+1)}{2(\mu - \lambda/2)} + \frac{n_2(n_2+1)}{2(\mu - \lambda/2)} = \frac{n_1(n_1+1) + n_2(n_2+1)}{2\mu - \lambda}. \tag{4}$$

For example, with $\mu = 1$ and $\lambda = 1$, the mean cost rate is $r = 2$ and (4) gives

$$v(\mathbf{n}) = \begin{bmatrix} 0 & 2 & 6 & 12 & \\ 2 & 4 & 8 & 14 & \ldots \\ 6 & 8 & 12 & 18 & \\ 12 & 14 & 18 & 24 & \\ \vdots & & & & \ddots \end{bmatrix} \tag{5}$$

In policy iteration, one next determines the value function $v_1(\mathbf{n})$ corresponding to $\omega_1(\mathbf{n})$, yielding a new policy $\omega_2(\mathbf{n})$. This is repeated until the mean cost rate no longer improves and an optimal routing policy has been found. In contrast, with reinforcement learning, one updates the routing policy at the same time as the estimates for the (optimal) value function. This leads to the algorithm described in the next section.

3.3 Reinforcement Learning

Several reinforcement learning techniques have been proposed in the literature. For example, in Q-learning the aim is to learn the utility function $Q(s, a)$ for each state s and corresponding action a. With the optimal policy, one always chooses such action a that maximizes the utility. Q-learning is typically applied to models with a finite horizon or a discounting factor. In this case, the dynamic programming equations (1) defining the value function also look different. In particular, there is no need to subtract the mean cost rate.

However, our problem formulation has the infinite time-horizon and the mean cost rate r is an integral part of the dynamic programming equations (1), leading to update rules (2). Table 1 describes the complete reinforcement learning algorithm based on (2).

Note also that the reinforcement learning involves two basic modes of operation: *exploration* and *exploitation*. Exploration refers to making random decisions which provide information on the value of actions that currently may seem non-optimal. Exploitation, on the other hand, refers to decisions that utilize the available information and choose (typically) the action that appears to be the optimal. Choosing the ratio between exploration and exploitation is an important optimization problem in reinforcement learning. In our algorithm, we have an implicit function exploit(j) that as a function of time decides (in random) whatever to choose the action that appears optimal (exploit) or to choose a random server (explore). Typically, it is important to explore more at start, but as the time goes by, exploitation should become the default action.

Example 2. Let us continue with the previous example. As a basic policy, we now utilize JSQ outside $\mathcal{A}$. Within $\mathcal{A}$, the routing policy is according to the Reinforcement learning rule. As the system has two identical exponential servers, the optimal routing policy is JSQ also within $\mathcal{A}$. That is, once Reinforcement learning algorithm converges, the resulting value function should be such that

$$v(\mathbf{n} + \mathbf{e}_1) < v(\mathbf{n} + \mathbf{e}_2) \qquad \forall\, \mathbf{n} \in \mathcal{A},$$

whenever $n_1 < n_2$, and vice versa, which means that

$$\omega(\mathbf{n}) = \operatorname*{argmin}_j n_j \qquad \forall\, \mathbf{n} \in \mathcal{A},$$

with ties resolved in an arbitrary fashion.

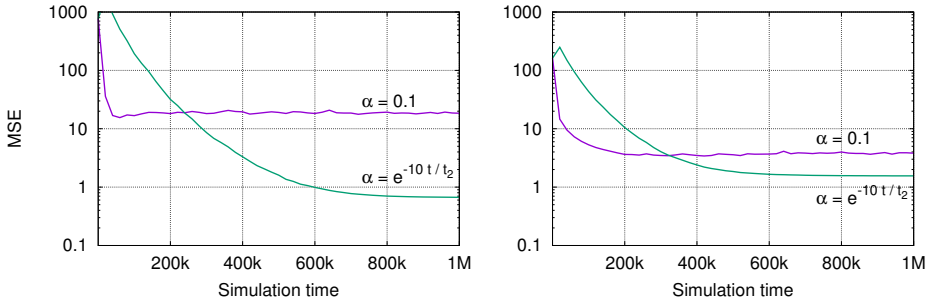
Initialization:

$v(\mathbf{x}) \leftarrow 0 \quad \forall\, \mathbf{x} \in \mathcal{S}$
$j \leftarrow 0$ {Step counter}
$\mathbf{x} \leftarrow (0, \ldots, 0)$ {Initial state, $\mathbf{x} \in \mathcal{S}$ }
$\mathbf{n} \leftarrow (0, \ldots, 0)$ {Initial previous state}
$t \leftarrow 0$ {Time between visits in the observed states $\mathcal{S}$}
$c \leftarrow 0$ {Incurred costs between the visits}
$T \leftarrow 0$ {Total (discounted) elapsed time}
$C \leftarrow 0$ {Total (discounted) costs}

After every time step Δt:

$t \leftarrow t + \Delta t$ {Time since the last departure or arrival}
$c \leftarrow c + \Delta c$ {Costs incurred during Δt}
if New job **then**
  if $\mathbf{x} \notin \mathcal{A}$ **then**
    $k \leftarrow \underset{i}{\operatorname{argmin}}\, n_i$ {outside $\mathcal{A}$ use (e.g.) JSQ}
  else if exploit(j) **then**
    $k \leftarrow \underset{i}{\operatorname{argmin}}\, v(\mathbf{n} + \mathbf{e}_i)$ {Ties in random}
  else
    $k \leftarrow \text{random}(1, \ldots, K)$ {Explore, in random}
  end if
  $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{e}_k$ {Send the new job to server k}
else if Departure from server k **then**
  $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{e}_k$ {Remove a job from server k}
end if
if $\mathbf{x} \in \mathcal{S}$ **then**
  $j \leftarrow j + 1$
  $\alpha \leftarrow e^{-\beta j}$ {or α is a small constant}
  $T \leftarrow \gamma T + t$ {e.g., $\gamma = 0.99$}
  $C \leftarrow \gamma C + c$
  $r \leftarrow C/T$ {Mean cost rate}
  $v(\mathbf{n}) \leftarrow (1 - \alpha) v(\mathbf{n}) + \alpha\,[c - t \cdot r + v(\mathbf{x})]$
  if $\mathbf{n} = 0$ **then**
    $\Delta \leftarrow v(0)$ {Adjust offsets}
    for all $\mathbf{n} \in \mathcal{S}$ **do**
      $v(\mathbf{n}) \leftarrow v(\mathbf{n}) - \Delta$
    end for
  end if
  $t \leftarrow 0$ {New epoch starts}
  $c \leftarrow 0$
  $\mathbf{n} \leftarrow \mathbf{x}$
end if

Table 1. Reinforcement learning for optimal routing in sub-space $\mathcal{A}$.



(a) Two servers, 6×6 box (b) Three servers, $4 \times 4 \times 4$ box

Fig. 2. Learning of the value function with a fixed learning rate $\alpha = 0.1$ and when $\alpha = e^{-10t/t_2}$ for two and three server example scenarios. On the x-axis is the time and the y-axis corresponds to the mean squared error (MSE) in log-scale.

4 Numerical Examples

In this section, we discuss some numerical experiments with the reinforcement learning. First we assume identical servers so that the correct results are known in advance (see examples 1 and 2). Then we consider two heterogeneous systems and compare reinforcement learning to some well-known heuristic policies.

4.1 Learning the Value Function

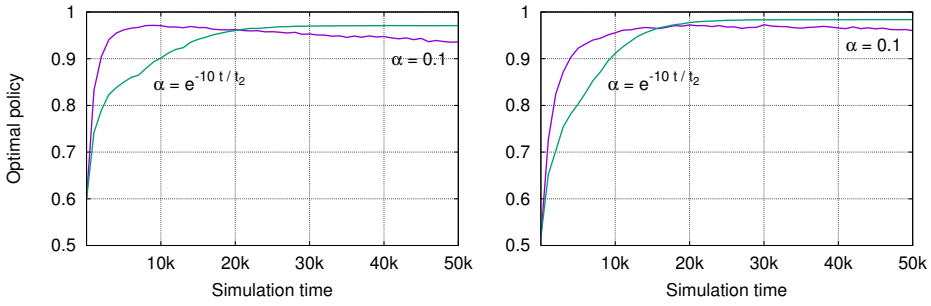
In the first numerical experiment, we study how fast the value function can be learned. To this end, we assume two or three identical servers with $\mu = 1$, unit arrival rate $\lambda = 1$, and the RND basic policy. The boxes for the substate spaces have 6×6 and $4 \times 4 \times 4$ states, respectively, which (relative) values are to be learned. Moreover, we use either a fixed $\alpha = 0.1$, or let α decay exponentially, $\alpha(t) = e^{-\beta t/t_2}$, where $\beta = 10$ and t_2 is the length of the simulation. The simulation algorithm is otherwise the same as in Table 1, but the server for the new jobs is always chosen using the basic policy, i.e.,

$$k = \omega_0(\mathbf{x}),$$

where ω_0 is RND in our case. That is, we update $v(\mathbf{x})$ but do not use it to make (better) routing decisions. Note that we could learn the value function of any given policy, but we have chosen RND because its value function is known exactly, and we see how fast the system learns it.

Figure 2 depicts the convergence of the learned value function to the known exact solutions, given in (5) for two servers. On the x-axis is the simulation time, and the y-axis corresponds to the mean squared error (MSE),

$$\text{MSE} = \frac{1}{N} \sum_i (\hat{v}_i - v_i)^2,$$



(a) Two servers, 6×6 box (b) Three servers, $4 \times 4 \times 4$ box

Fig. 3. Learning the optimal policy happens much faster.

where $N = 6^2$ and $N = 4^3$ in our case. Note that the y-axis is in logarithmic scale. We can see that at with a fixed $\alpha = 0.1$, the learning converges fast to a certain level. When α decreases exponentially (with $\beta = 10$), the learning rate is slower, but the final result is more accurate, as expected. However, the estimates for the value function are useful long before that, as we will see next.

4.2 Learning the Optimal Policy

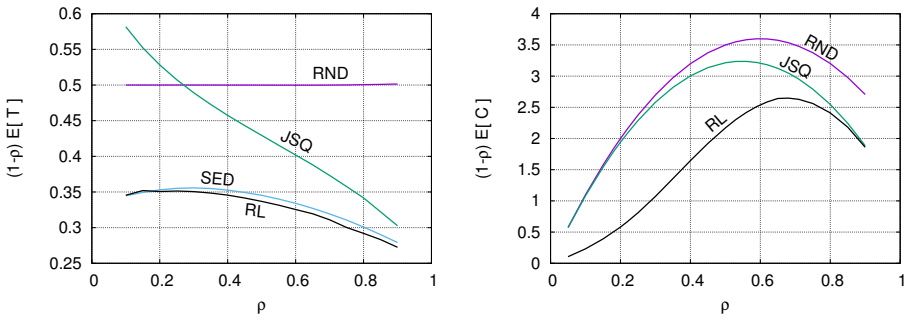
Next we study how the reinforcement learning algorithm converges to the optimal policy in this elementary case. That is, with the identical servers and the response time cost metric, the optimal policy is JSQ. At start, the value function is initialized to zero, and thus random server would be chosen at every state. However, as different states have been visited and the corresponding updates for the value function recorded, we soon start to make correct decisions.

Figure 3 depicts the fraction of states where the correct routing decision is made as a function of the simulation time. We can observe that the optimal behavior is learned much faster than the "correct" values for the value function. This suggests that a routing policy based on the reinforcement learning can quickly adapt to changes in its operating environment. In such cases, a fixed learning rate such as $\alpha = 0.1$ is naturally preferred.

4.3 Heterogeneous Service Rates

Let us next consider a heterogeneous system with service rates $(\mu_1, \mu_2) = (3, 1)$, i.e., server 1 is now three times faster than server 2. We note that optimal routing policy for heterogeneous systems is not available in closed-form for the mean response time metric even when the service times are exponentially distributed. The near-optimal policy can be determined numerically [3], and here we apply the reinforcement learning algorithm to the same end.

The simulation results are depicted in Figure 4(a). On the x-axis is the offered load ρ, and the y-axis corresponds to the scaled mean response time, $(1-\rho)\,\mathbb{E}[T]$.



(a) Heterogeneous service rates (b) Heterogeneous running costs

Fig. 4. Left figure (a) depicts the simulation results with heterogeneous service rates $(\mu_1, \mu_2) = (3, 1)$. Right figure (b) shows the simulation results with unequal running cost rates, $(r_1^{(r)}, r_2^{(r)}) = (0, 10)$.

With the load balancing random split, the system reduces into K independent M/M/1 queues, and the mean response time is

$$\mathbb{E}[T] = \sum_i \frac{\mu_i}{\sum_j \mu_j} \cdot \mathbb{E}[T_i] = \sum_i \frac{1}{\sum_j \mu_j} \frac{1}{1-\rho} = \frac{K}{(1-\rho)\sum_j \mu_j},$$

and thus with the two servers the scaled mean response time with RND is $1/2$. Other reference policies are JSQ and SED. The reinforcement learning (RL) uses SED outside the 6×6 box, where the optimal value function is learned and utilized to make near-optimal routing decisions. We can observe that the performance with RL indeed is better than with JSQ or SED.

4.4 Unequal Running Costs

Finally, suppose we have two equally fast servers, $(\mu_1, \mu_2) = (1, 1)$, but server 2 is owned by a third party and they charge us according to used CPU cycles so that the corresponding running costs are $(r_1^{(r)}, r_2^{(r)}) = (0, 10)$. In other words, processing a job at server 2 costs on average $r_2^{(r)}/\mu_2 = 10$, whereas at server 1 it is free. In addition to the running costs, the mean response time is also minimized (i.e., the mean response time is the quality of service component) and the total cost rate at state (n_1, n_2) is given by

$$r_{\mathbf{n}} = n_1 + n_2 + 10 \cdot \mathbf{1}(n_2 > 0).$$

The simulation results are depicted in Figure 4(b). As $\mu_1 = \mu_2$, SED reduces to JSQ and it has been omitted. On the x-axis is the offered load ρ, and the y-axis corresponds to the scaled mean cost rate, $(1-\rho)\,\mathbb{E}[C]$. We can see that all policies, RND, JSQ and RL, have the same shape. Moreover, the dynamic policies, JSQ and RL, seem to converge to the same mean cost rate as $\rho \to 1$. At

this limit, both servers must be busy all the time and the unequal running cost rates no longer matter. However, when ρ is small or moderate, the reinforcement learning based policy RL reduces costs significantly. It routes jobs to server 2 only to the extend it is meaningful!

5 Conclusions

A straightforward reinforcement learning approach is studied in this paper. The approach is more general than our numerical examples suggest. First, as mentioned, the cost structure can be rather general and could, e.g., penalize the system when a queue length exceeds given thresholds. Second, without any modifications, the number of servers can be more than two or three. The finite substate space unavoidably becomes larger, which eventually limits the applicability to small systems in terms of number of servers. However, if some servers are identical, the corresponding symmetries can be taken into account to mitigate the scaling problem. Third, it is also straightforward to include *batch arrivals* to the model and the learning algorithm. By adjusting the batch size distribution, more bursty arrival processes can be modelled, which makes the approach more applicable. Fourth, in our case, the jobs were identical. It is possible to introduce job classes, having, e.g., different size distributions or holding cost rates. However, each job class increases the dimensionality of the state space, and therefore we are again limited to a small number of job classes.

In our future work, we plan to investigate on how well the reinforcement learning based dispatching policy adapts to changing environment. In particular, we will compare it to other adaptive and (load) insensitive routing policies.

Acknowledgements

This work was supported by the Academy of Finland in the FQ4BD project (grant no. 296206) and by the University of Iceland Research Fund in the RL-STAR project.

References

1. F. A. Haight, "Two queues in parallel," *Biometrika*, vol. 45, no. 3-4, 1958.
2. C. Watkins, "Learning from delayed rewards," Ph.D. dissertation, Cambridge University, 1989.
3. E. Hyytiä, R. Righter, and S. G. Samúelsson, "Beyond the shortest queue routing with heterogeneous servers and general cost function," in *ValueTools*, Dec. 2017.
4. P. Whittle, *Optimal Control: Basics and Beyond.* Wiley, 1996.
5. E. Hyytiä, "Lookahead actions in dispatching to parallel queues," *Performance Evaluation*, vol. 70, no. 10, pp. 859–872, 2013, (IFIP Performance'13).
6. E. Hyytiä and J. Virtamo, "Dynamic Routing and Wavelength Assignment Using First Policy Iteration," in *the Fifth IEEE ISCC'2000*, Jul. 2000, pp. 146–151.
7. R. A. Howard, *Dynamic Probabilistic Systems, Volume II: Semi-Markov and Decision Processes.* Wiley Interscience, 1971.
8. R. Bellman, *Dynamic programming.* Princeton University Press, 1957.