# Stochastic dynamic job scheduling with interruptible setup and processing times: An approach based on queueing control

Dongnuan Tian
Department of Management Science, Lancaster University Management School, Lancaster University, Lancaster LA1 4YW, United Kingdom. Email: d.tian2@lancaster.ac.uk.

Rob Shone*
Department of Management Science, Lancaster University Management School, Lancaster University, Lancaster LA1 4YW, United Kingdom. Email: r.shone@lancaster.ac.uk.

## Abstract

We consider a stochastic, dynamic job scheduling problem, formulated as a queueing control problem, in which a single server processes jobs of different types that arrive according to independent Poisson processes. The problem is defined on a network, with jobs arriving at designated demand points and waiting in queues to be processed by the server, which travels around the network dynamically and is able to change its course at any time. In the context of machine scheduling, this enables us to consider sequence-dependent, interruptible setup and processing times, with the network structure encoding the amounts of effort needed to switch between different tasks. We formulate the problem as a Markov decision process in which the objective is to minimize long-run average holding costs and prove the existence of a stationary policy under which the system is stable, subject to a condition on the workload of the system. We then propose a class of index-based heuristic policies, show that these possess intuitively appealing structural properties and suggest how to modify these heuristics to ensure scalability to larger problem sizes. Results from extensive numerical experiments are presented in order to show that our heuristic policies perform well against suitable benchmarks.

*Keywords:* Job scheduling; dynamic programming; index heuristics

## 1 Introduction

Job scheduling and server scheduling problems have been very widely studied in operations research. Stochastic, dynamic versions of such problems typically draw upon both scheduling and queueing theory (Leung (2004), Pinedo (2016), Blazewicz et al. (2019)). A classical problem formulation involves a system of $N$ parallel queues, each with its own arrival process, and a single server with the ability to switch dynamically between queues. Jobs at queue $i$ arrive at a

*Corresponding author

rate of $\lambda_i$ and can be processed at a rate of $\mu_i$, $i = 1, ..., N$. A holding cost $c_i$ is incurred for each unit of time that a type $i$ job spends in the system. The server is able to observe queue lengths continuously and can provide service to one queue at a time. This type of formulation induces a well-known $c\mu$-rule, whereby the queues are ranked in descending order of the product $c_i\mu_i$, and the server always selects a job from the queue with the largest $c_i\mu_i$ value among the non-empty queues. The optimality of the $c\mu$-rule in various queueing settings has been well-documented (Smith et al. (1956), Baras et al. (1985), Buyukkoc et al. (1985), Van Mieghem (1995)).

In recent decades, researchers have sought to extend the applicability of the $c\mu$-rule to address more complex scheduling problems. Mandelbaum and Stolyar (2004) introduced a generalized version of the rule for systems with convex delay cost structures, reflecting more realistic scenarios where penalties for delays increase at an accelerating rate. Atar et al. (2010) modified the classical $c\mu$-rule to incorporate abandonment effects, resulting in the $c\mu/\theta$-rule. This rule considers both service rates and abandonment rates, prioritizing queues based on the ratio of the cost-weighted service rate to the abandonment rate. Saghafian and Veatch (2015) proposed a $c\mu$-rule for a parallel flexible server system with a two-tier structure. In this model, the first tier consists of job classes that are assigned to specific servers, while the second tier includes a class of jobs that can be served by any available server. In more recent years, applications of the $c\mu$-rule have continued to attract a lot of attention. Lee and Vojnovic (2021) established a learning-based variant of the rule, where the algorithm learns the job parameters over time and adapts its scheduling decisions to minimize cumulative holding costs, even when the statistical parameters of the jobs are initially unknown. Cohen and Saha (2022) also considered uncertain model parameters, and showed that the $c\mu$-rule attains a type of asymptotic optimality. Ozkan (2022) investigated the performance of the $c\mu$-rule in systems with non-parallel configurations, such as tandem queues.

Job scheduling and server scheduling problems have broad applications in several real-world domains including cloud computing, communications networks and manufacturing (Shaw and Singh (2014), Zhang et al. (2019), Xu et al. (2021)). In this paper we consider a stochastic, dynamic job scheduling problem formulated on a network of nodes and edges, in which certain nodes are designated as 'demand points' and act as entry points for jobs of specific types. There is a single server, which may represent a machine (for example) that needs to process jobs one at a time. In this context, the time needed for the server to move from one demand point to another represents the setup time required for the machine to switch from processing one type of job to another. The design of the network may be seen as a way to encode the amounts of effort needed for the server (or machine) to switch between jobs of different types.

As an example of our network-based approach, consider the situation shown in Figure 1. The white-colored nodes are demand points, at which new jobs arrive according to independent Poisson processes, and the gray-colored nodes are 'intermediate stages', which must be traversed in order to move from either of the left-hand demand points to the right-hand demand point or vice versa. A single server occupies one node in the network at any given time, and can either remain at its current node or attempt to move to an adjacent node. In order to process jobs of type $i$ (for $i \in \{1, 2, 3\}$), it must be located at node $i$. The times required to process jobs and switch between adjacent nodes are random, and jobs waiting to be processed incur linear

holding costs. We provide a detailed problem formulation in Section 2.

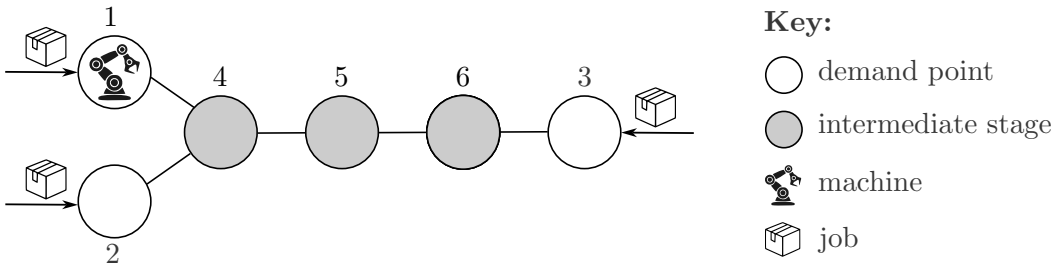


Figure 1: A network with 3 demand points, 3 intermediate stages and a single server. White-colored nodes represent demand points at which new jobs arrive, and gray-colored nodes represent intermediate stages.

An important feature of our network formulation is the fact that the server is always free to choose a new action (i.e. a new direction to move), regardless of actions chosen at previous time points. This implies that it can interrupt the processing of a particular job, or the transition between different job types, in order to follow a different course. For example, suppose the server is located at node 1 in Figure 1, but wishes to move to node 3 so that it can process jobs of type 3. To accomplish this, it must pass through the intermediate nodes 4, 5 and 6. We assume that the amount of time needed to switch between any two adjacent nodes is randomly distributed (further details are provided in Section 2). However, suppose that after arriving at node 5, it decides to change course and move towards node 2 instead. (This may happen, for example, if new jobs have recently arrived at node 2.) In the job scheduling context, we would say that the setup time needed to prepare to process jobs of type 3 is interrupted so that the server can prepare to process jobs of type 2 instead. Moreover, our network formulation not only allows setup times to be interrupted, but also allows them to depend on any partial progress made on interrupted setup attempts. For example, in Figure 1, job types 1 and 2 may be regarded as having similar setup requirements, as they are located close to each other in the network. If the server begins at node 3 and initially moves towards node 1, but then decides to move to node 2 instead, then any progress made in setting up jobs of type 1 (i.e. moving towards node 1) is helpful in reducing the amount of time needed for setting up jobs of type 2. On the other hand, if the server begins at node 1 and initially moves towards node 3, but then decides to move to node 2 instead, then any progress made in setting up jobs of type 3 actually delays the setting up of type 2 jobs. Accurate modeling of complex setup time requirements (including the effects of interrupted setups) is important in settings where new jobs arrive frequently and randomly, requiring the server to allocate its time efficiently.

While our network formulation is intuitively quite simple, it allows consideration of features that are not particularly common in job scheduling problem formulations, such as sequence-dependent, interruptible setup times that are affected by partial setup progress. These kinds of generalizations can be particularly relevant in production and manufacturing applications, where the reallocation of a server from one task to another can involve complicated reconfigurations or the assembly of specialized tools and equipment (Allahverdi et al. (1999), Gholami et al. (2009)). We also note that although job scheduling applications are a primary motivation for our study, our formulation is sufficiently general to allow potential applications to other common operations

research problems. For example, one could consider a dynamic vehicle routing problem in which a vehicle provides service to customers in geographically distinct locations (Ulmer et al. (2020)), or a security problem in which a defensive agent responds to threats that appear in a computer network (Hunt and Zhuang (2024)).

The stochastic, dynamic nature of our problem implies that a Markov decision process (MDP) formulation is appropriate, although (as in many other problems formulated as MDPs) it is not possible to compute optimal solutions unless the scale of the problem is very small; thus, we must consider heuristic approaches. Several previous studies have also used MDP formulations in job scheduling contexts, although their assumptions are usually quite different from ours. Zhang et al. (2017) considered a problem in which jobs with different processing requirements arrive randomly and wait in queues to be processed by machines, and at each decision epoch a job must be chosen to be processed next. The machines are relatively 'static' in this setting, rather than being assigned dynamically to different tasks. Luo (2020) (see also Lei et al. (2023), Zhao et al. (2023)) formulated a dynamic flexible job shop scheduling problem with new job insertions and used a Q-learning approach, but their objective is quite different from ours as it is based on minimizing the total tardiness of a given set of jobs with fixed processing times. Elsayed et al. (2022) also considered the processing of a given set of jobs on multiple machines and used a novel graph-based MDP formulation, but their study does not consider the stochastic factors present in our model. Fan (2012) considered a single machine scheduling problem (SMSP) with transportation costs and used an MDP algorithm to minimize the overall cost of processing jobs in a sequence and transporting the finished products to a single customer; however, their problem (unlike ours) is based on a finite set of jobs. Yang et al. (2022) formulated an SMSP in which the machine's state is subject to uncertainty, resulting in job time parameters being expressed in probabilistic terms. Their problem is also of a finite-time nature, with an objective based on minimizing the makespan. In summary, our paper addresses an important research gap by making a connection between job scheduling problems with random arrivals and infinite-horizon queueing control problems.

A particularly relevant paper to ours is Duenyas and Van Oyen (1996), which considers the problem of dynamically allocating a single server to process jobs of different types that arrive according to independent Poisson processes and wait in parallel queues. As in our problem, the authors consider random processing times and switching times (between different queues), and the time horizon of interest is infinite. However, they do not allow processing times or switching times to be interrupted. Furthermore, they do not use a network formulation, and the time required to switch from one queue to another depends only on the destination queue; thus, there are no sequence-dependent setup times. Despite these differences, their approach of developing index heuristics is similar to the approach that we pursue in this study, and their algorithms use certain steps and conditions that we aim to adapt and generalize for use in our problem (full details can be found in Section 3). We also note that the study of Duenyas and Van Oyen (1996) makes use of certain results from the classical literature on 'polling systems', in which a single server visits a set of queues according to a predefined sequence (Boxma and Groenendijk (1987), Browne and Yechiali (1989), Altman et al. (1992), Yechiali (1993)). In Section 2, we also draw upon this body of research to show that our system possesses an important stability

property.

As an alternative to index heuristics, one can also consider reinforcement learning (RL) approaches, and several recent studies have applied RL to machine scheduling problems (Li et al. (2020), Wang et al. (2021), Kayhan and Yildiz (2023), Li et al. (2024)). Although RL methods are undoubtedly powerful, they also have some drawbacks in comparison to alternative approaches (Dulac-Arnold et al. (2019)). Solutions given by RL algorithms tend to be less interpretable than those given by simpler heuristics, and therefore less appealing to system operators. Additionally, there is the issue of online replanning, as discussed in Bertsekas (2019). If the parameters of the problem (e.g. job arrival rates or setup time distributions) change suddenly and unexpectedly, RL methods may struggle to adapt 'on-the-fly' as they require expensive offline training in order to be able to discover strong-performing policies, whereas index heuristics can quickly adapt to the new parameters of the problem. Thus, whilst we acknowledge the potential of RL methods, these are not within the scope of our current study.

The main contributions of this paper are as follows:

- We provide a novel formulation of a stochastic, dynamic job scheduling problem in which the setup time distributions for different types of jobs are encoded by a network structure, allowing them to be both sequence-dependent and interruptible.
- We prove that there always exists a decision-making policy under which the system is stable, subject to a condition on the total workload of the system.
- We propose a class of index-heuristics, referred to as $K$-stop heuristics, which are suitable for our network-based problem as they make decisions by taking the topological structure of the network into account.
- We prove that our heuristics possess a pathwise consistency property which ensures that the server always proceeds to a demand point in finite time. Moreover, we show that these heuristics attain both system stability and optimality under certain conditions.
- We propose a further class of heuristics, known as ($K$ from $L$)-stop heuristics, that scale much more readily to large network sizes than the $K$-stop heuristics.
- We present extensive results from numerical experiments in order to compare the relative performances of our heuristics, comparing them (where possible) to optimal values given by dynamic programming and also to the performance of the unmodified heuristic policy in Duenyas and Van Oyen (1996).

The rest of the paper is organized as follows. In Section 2 we formulate our stochastic, dynamic job scheduling problem as an MDP and prove an important stability property. In Section 3 we derive index heuristics and prove some useful properties of these heuristics, such as pathwise consistency. In Section 4 we present the results of our numerical study. Finally, our concluding remarks can be found in Section 5.

## 2 Problem Formulation

The problem is defined on a connected graph, referred to as a *network*. Let $V$ and $E$ denote the sets of nodes and edges respectively, and let $D \subseteq V$ be a subset of nodes referred to as *demand points* (the white nodes in Figure 1). Let $N := V \backslash D$ denote the other nodes (the gray nodes in Figure 1), referred to as *intermediate stages*. We use $d = |D|$ and $n = |N|$ to denote the numbers of demand points and intermediate stages, respectively. We will also assume that the demand points are numbered $1, 2, ..., d$ and the intermediate stages are numbered $d+1, d+2, ..., d+n$. Jobs arrive at demand point $i \in D$ according to a Poisson process with intensity rate $\lambda_i > 0$, referred to as an *arrival rate*, and wait in a first-come-first-served queue until they are processed. Arrivals at different demand points are assumed to occur independently of each other. Additionally, a linear holding cost $c_i > 0$ per unit time is incurred for each job waiting to be processed at node $i \in D$.

Jobs are processed by a single *server* (or *machine*, *resource* etc.) which can move around the network and, at any given time, is located at a single node in $V$. At any point in time, the server can either remain at its current node or make an attempt to move to an adjacent node. In the latter case, the server must also decide which node to move to. Thus, if the server's current node is adjacent to $k$ other nodes then there are $k+1$ possible decisions for the server. If the server decides to remain at a node $i \in D$ (i.e. at a demand point) and this node has a number of jobs $x_i > 0$ waiting to be processed, then the jobs are processed at an instantaneous rate $\mu_i > 0$, where $\mu_i$ is the *processing rate* for demand point $i$. If the server remains at a demand point $i$ with no jobs present (i.e. $x_i = 0$), then the server is said to be *idle*. Likewise, idleness can also occur when the server chooses to remain at some intermediate stage $j \in N$. On the other hand, if the server chooses to move (or 'switch') to an adjacent node, then the switch occurs at an instantaneous rate $\tau > 0$, referred to as a *switching rate*. Switching and processing times are assumed to be independent of the arrival processes for demand points in $D$.

The assumption of instantaneous processing and switching rates implies that processing and switching times are exponentially distributed, but they are also *interruptible* because the server can change its decision at any point in time. For example, it can choose to remain at a non-empty demand point but then choose to switch before any further jobs have been processed. Alternatively, it could choose to switch from one node $i$ to another node $j$, but then change direction and attempt to switch to a different node $k$ before the switch to $j$ is complete. We note that, although the switching time between two adjacent nodes has the memoryless property in our model, the time to switch from one demand point to another (which, in general, requires passing through a sequence of intermediate stages) is instead distributed as a sum of i.i.d. exponential switching times, implying that it has an Erlang distribution. Throughout this paper we assume exponential processing times and switching times (between adjacent nodes), with the latter assumption implying Erlang-distributed setup times (between demand points). However, later in this section we also comment on how the formulation could be extended to the more general case of phase-type processing and switching times.

Under the above assumptions, the system can be formulated as a continuous-time Markov

decision process (MDP). The state space can be written as

$$S := \{(v, (x_1, ..., x_d)) \mid v \in V,\ x_i \geq 0 \text{ for } i \in D\},$$

where $v$ is the node currently occupied by the server and $x_i$ is the number of jobs waiting to be processed (including any job currently being processed) at demand point $i \in D$. We will use vectors such as $\mathbf{x}$ and $\mathbf{y}$ to represent generic states in $S$ and use $v(\mathbf{x})$ to denote the server's location under state $\mathbf{x} \in S$. In order to simplify notation we will sometimes write $v$ instead of $v(\mathbf{x})$ if the state associated with $v$ is clear. Considering that the server cannot be processing jobs whilst also switching and also cannot serve more than one demand point at a time, the total of the transition rates under any state is at most $\sum_{i \in D} \lambda_i + \max\{\mu_1, ..., \mu_d, \tau\}$ and we can therefore use the technique of uniformization (Serfozo (1979)) to transform the system into a discrete-time counterpart that evolves in time steps of size

$$\Delta := \left( \sum_{i \in D} \lambda_i + \max\{\mu_1, ..., \mu_d, \tau\} \right)^{-1}. \tag{1}$$

Let $R(\mathbf{x})$ be the set of nodes adjacent to node $v(\mathbf{x})$ (not including $v(\mathbf{x})$ itself) and $A_{\mathbf{x}} = \{v(\mathbf{x})\} \cup R(\mathbf{x})$ be the action set available under state $\mathbf{x} \in S$ at a particular time step. We can interpret action $a \in A_{\mathbf{x}}$ as the node that the server tries to move to next, with $a = v(\mathbf{x})$ indicating that the server remains where it is. Then, for any pair of states $\mathbf{x}, \mathbf{y} \in S$ with $\mathbf{x} \neq \mathbf{y}$, the transition probability of moving from $\mathbf{x} := (v(\mathbf{x}), (x_1, ..., x_d))$ to $\mathbf{y} := (v(\mathbf{y}), (y_1, ..., y_d))$ following the choice of action $a \in A_{\mathbf{x}}$ can be expressed as

$$p_{\mathbf{x},\mathbf{y}}(a) := \begin{cases} \lambda_i \Delta, & \text{if } y_i = x_i + 1 \text{ and } y_j = x_j \text{ for } j \neq i, \\ \mu_i \Delta, & \text{if } y_i = x_i - 1,\ y_j = x_j \text{ for } j \neq i \text{ and } a = v(\mathbf{x}) = i, \\ \tau \Delta, & \text{if } y_j = x_j \text{ for all } j \in D,\ v(\mathbf{y}) \in R(\mathbf{x}) \text{ and } a = v(\mathbf{y}), \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

with $p_{\mathbf{x},\mathbf{x}}(a) = 1 - \sum_{\mathbf{y} \neq \mathbf{x}} p_{\mathbf{x},\mathbf{y}}(a)$ being the probability of remaining at the same state. Since the units of time are arbitrary, we will assume $\Delta = 1$ without loss of generality. This implies that parameters such as $\lambda_i$, $\mu_i$ and $\tau$ can be interpreted as probabilities in our uniformized MDP, rather than transition rates.

At each time step, the single-step cost $f(\mathbf{x})$ is calculated by summing the holding costs of jobs that are still waiting to be processed or currently being processed, so that

$$f(\mathbf{x}) := \sum_{i \in D} c_i x_i. \tag{3}$$

Let $\theta$ denote a decision-making policy for the MDP. In a general sense, $\theta$ can take actions that depend on the history of the process thus far and might also be randomized (see Puterman (1994)). The policy is said to be *stationary* and *deterministic* if action $\theta(\mathbf{x}) \in A_{\mathbf{x}}$ is always chosen under state $\mathbf{x} \in S$. The expected long-run average cost per unit time (or *average cost*

for short) under policy $\theta$, given that the initial state of the system is $\mathbf{x} \in S$, can be expressed as

$$g_\theta(\mathbf{x}) = \liminf_{T\to\infty} T^{-1}\mathbb{E}_\theta\left[\sum_{t=0}^{T-1} f(\mathbf{x}(t)) \;\middle|\; \mathbf{x}(0) = \mathbf{x}\right] \tag{4}$$

where $\mathbf{x}(t)$ denotes the state of the system at time step $t \in \mathbb{N}_0$. The theory of uniformization implies that under a stationary policy $\theta$, the discretized system has the same average cost $g_\theta(\mathbf{x})$ as that incurred by its continuous-time counterpart, in which we allow actions to be chosen every time the system transitions from one state to another. In the continuous-time system, after choosing an action $a \in A_\mathbf{x}$ in state $\mathbf{x} \in S$, the amount of time until the next transition is exponentially distributed with mean $1/q(\mathbf{x}, a)$, where $q(\mathbf{x}, a)$ is the sum of the transition rates (for example, if the server is processing a job at demand point $i$ then $q(\mathbf{x}, a) = \sum_{j\in D} \lambda_j + \mu_i$). On the other hand, in the uniformized system (with $\Delta = 1$), there is a fixed time step of unit length until the next transition, and we also allow 'self-transitions' into the same state. It can be seen from (2) that the probability of a self-transition is $1 - q(\mathbf{x}, a)$ and hence the total number of time steps until the next change of state (assuming that the decision-maker continues to choose action $a$) is geometrically distributed with mean $1/q(\mathbf{x}, a)$. Thus, the expected amount of time (and also the cost incurred) until the next change of state is the same in both systems. This provides an intuitive justification for the uniformization method, and we defer to Lippman (1975) and Serfozo (1979) for full technical details.

The objective of the problem is to minimize the long-run average cost $g_\theta(\mathbf{x})$; that is, to find a policy $\theta^*$ such that

$$g_{\theta^*}(\mathbf{x}) \le g_\theta(\mathbf{x}) \quad \forall \theta \in \Theta,\ \mathbf{x} \in S,$$

where $\Theta$ is the set of all admissible policies.

**Remark 2.1.** *The MDP model that we describe in this section can be extended without great difficulty to the case of phase-type processing and switching times. In order to consider phase-type processing times, suppose the processing time of a job at node $i$ is distributed as a sum of $k_i \ge 2$ exponentially-distributed service phases, with $\mu_i^{(r)}$ being the service rate for the $r^{th}$ phase, $r = 1, ..., k_i$. The state variable $x_i$ should then be the number of service phases remaining to be processed, rather than the number of jobs. Thus, if a new job of type $i$ arrives at a particular time step (with probability $\lambda_i$), this causes $x_i$ to increase by $k_i$. One can define $y_i = \lceil x_i/k_i \rceil$ as the number of jobs remaining (including any that are partially complete). Since jobs are processed in first-come-first-served order, if $x_i = 2k_i + 1$ (for example) and the server remains at node $i$ at a particular time step then a service phase completion occurs with probability $\mu_i^{(k_i)}$, since there is a partially complete job with only one phase remaining. The cost function should still be based on job counts rather than phase counts, so the single-step cost function is $\sum_{i\in D} c_i y_i$. The uniformization parameter in (1) can be adjusted in an obvious way. To incorporate phase-type setup times (between demand points), one can allow the switching rates between adjacent nodes to be edge-dependent and also (if desired) direction-dependent, so that the setup times are distributed as sums of independent exponential random variables with non-identical rates. This is a powerful generalization because phase-type distributions can approximate any continuous distribution to an arbitrary degree of accuracy (Asmussen (2003)).*

Given that our MDP has an infinite state space, the average cost $g_\theta(\mathbf{x})$ can only be finite if the system is stable under $\theta$, in the sense that $\theta$ induces an ergodic Markov chain on $S$. Let $\rho$ denote the *traffic intensity* of the system, defined as

$$\rho = \sum_{i \in D} \lambda_i / \mu_i.$$

Our first result establishes that the condition $\rho < 1$ is sufficient to ensure the existence of a deterministic stationary policy under which the system is stable.

**Theorem 2.2.** ***(Stability.)*** *Suppose $\rho < 1$. Then there exists a deterministic stationary policy $\theta$ such that $g_\theta(\mathbf{x}) =: g_\theta < \infty$ for all $\mathbf{x} \in S$.*

Details of the proof can be found in Appendix A. It relies upon results from the literature on polling systems, but these results cannot be applied directly to our system because it is not possible to construct a stationary policy $\theta$ that visits the demand points $i \in D$ in a fixed cyclic pattern and serves each point exhaustively on each visit. To illustrate this point, consider a network with only two demand points ($D = \{1, 2\}$) separated by a single intermediate stage ($N = \{3\}$) as shown in Figure 2. Suppose we wish to implement a simple 'polling system' type of policy under which the server moves between the two demand points in an alternating pattern $(1, 2, 1, 2, ...)$ and, each time it arrives at point $i \in D$, stays there until all jobs have been processed ($x_i = 0$) before moving directly to the other point. Unfortunately, in order to know which action to choose under the state $(3, (0, 0))$ (or any other state $\mathbf{x}$ with $v(\mathbf{x}) = 3$), the server must know which demand point was the last to be visited. Since this information is not included in the system state, the server is forced to follow a nonstationary policy in order to achieve the required alternating pattern.

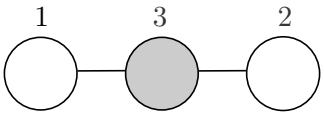


Figure 2: A network with 2 demand points and 1 intermediate stage.

Our proof circumvents this difficulty by considering a new MDP with a modified state space, in which the system state includes an extra variable indicating the most recent demand point $i$ to have had no jobs present while the server was located there. This enables 'polling system' type policies, like the one described above, to be represented as stationary policies. The condition $\rho < 1$ is known to ensure that, under an exhaustive 'polling'-type policy, the system is stable (Altman et al. (1992)). We can then use the 'approximating sequence' approach developed by Sennott (Sennott (1997, 1999)) to compute an optimal policy for the modified MDP using value iteration. Finally, we use an inductive argument to show that the decisions taken under the optimal policy for the modified MDP are independent of the extra information that we included in the state space, and therefore the optimal policy $\theta^*$ for the modified MDP (under which the system is stable) also qualifies as an deterministic stationary (and optimal) policy for the original MDP.

It is interesting to note that, by Theorem 2.2 the condition $\rho < 1$ is always sufficient to

ensure that the system is stable under an exhaustive polling regime, regardless of how long it takes for the server to switch between different demand points. In our model, switching times generally become longer if the switching rate $\tau$ is reduced or extra intermediate stages are inserted in the network. Intuitively, if switching times become longer, then the number of jobs present when the server arrives at any particular demand point tends to increase. However, under the exhaustive regime, the server is committed to processing all jobs at a demand point before moving to the next one. Suppose the expected switching times between demand points are large but finite. If the system is unstable, then there must be at least one demand point at which the number of jobs present tends to infinity over time, but this implies that the overall proportion of time that the server spends processing jobs (as opposed to switching between nodes) tends towards 1. This is not possible if $\sum_i \lambda_i/\mu_i < 1$, since the proportion of time spent processing jobs at any node $i \in D$ cannot be greater than $\lambda_i/\mu_i$. Thus, regardless of how large the expected switching times are, the system is indeed stable under exhaustive polling when $\rho < 1$.

In theory, the 'approximating sequences' approach discussed in the proof of Theorem 2.2 can be combined with value iteration in order to compute an optimal policy for our MDP, assuming that $\rho < 1$. In practice, however, the well-known 'curse of dimensionality' prevents us from being able to compute optimal policies in systems with more than (roughly) three or four demand points. Therefore we need to develop heuristic methods to obtain easily implementable policies that can achieve strong performances across a range of different possible system configurations. Our proposed heuristic methods are introduced in Section 3.

## 3 Index heuristics

As mentioned in Section 1, Duenyas and Van Oyen (1996) considered a job scheduling problem that has some similarities to ours, although their problem is not defined on a network and does not allow switching or processing times to be interrupted. The approach used in Duenyas and Van Oyen (1996) (see also Bell (1971), Harrison (1975)) is based on a well-known equivalence between the minimization of congestion-based costs and the maximization of a reward criterion under which the server obtains rewards at a constant rate while it is processing jobs at a particular demand point. We will show that a similar equivalence holds in our problem, despite the inclusion of novel features such as interruptible setup and processing times. In order to motivate the approach, it is necessary to begin by discussing how the problem formulated in Section 2 would be modified if the objective was to minimize expected total *discounted* cost, rather than long-run average cost. Let $\alpha$ be a discount factor satisfying $0 < \alpha < 1$ (values close to 1 are typically assumed). We will consider a deterministic, stationary policy $\theta$ and use $\theta(\mathbf{x})$ to denote the action $a \in A_{\mathbf{x}}$ selected by $\theta$ under state $\mathbf{x} \in S$. Then the expected total

discounted cost under an admissible policy $\theta$ can be expressed as

$$V(\mathbf{x}) = \mathbb{E}_\theta \left[ \sum_{t=0}^{\infty} \alpha^t f(\mathbf{x}(t)) \;\middle|\; \mathbf{x}(0) = \mathbf{x} \right]$$

$$= \mathbb{E}_\theta \left[ \sum_{t=0}^{\infty} \alpha^t \sum_{i \in D} c_i [x_i(0) + A_i(t) - B_i(t)] \;\middle|\; \mathbf{x}(0) = \mathbf{x} \right], \tag{5}$$

where $x_i(0)$ is the initial number of jobs at demand point $i \in D$, $A_i(t)$ is the total number of new jobs of type $i$ that arrive by time step $t$ and $B_i(t)$ is the total number of jobs of type $i$ that are completely processed by time step $t$. Since $x_i(0)$ and $A_i(t)$ (for all $i \in D$, $t \geq 0$) are independent of the server's actions, it is clear from (5) that minimizing $V(\mathbf{x})$ is equivalent to maximizing a weighted sum of expected departure counts, $B_i(t)$. Furthermore, if the server is processing a job of type $i$ at time step $t$, then with probability $\mu_i$ the number of jobs at $i$ is reduced by one and this implies an expected cost saving of $\mu_i \sum_{u=t+1}^{\infty} \alpha^u c_i = c_i \mu_i \alpha^{t+1}/(1-\alpha)$ compared to a passive policy which allows this job to remain in the system forever. It follows that minimizing $V(\mathbf{x})$ is equivalent to maximizing

$$W(\mathbf{x}) := \mathbb{E}_\theta \left[ \sum_{t=0}^{\infty} \alpha^t w(\mathbf{x}(t), \theta(\mathbf{x}(t))) \;\middle|\; \mathbf{x}(0) = \mathbf{x} \right], \tag{6}$$

where the single-step reward function $w(\mathbf{x}, a)$ is defined for $\mathbf{x} = (v, (x_1, ..., x_d)) \in S$, $a \in A_{\mathbf{x}}$ by

$$w(\mathbf{x}, a) = \begin{cases} \dfrac{\alpha c_i \mu_i}{1-\alpha}, & \text{if } v = i \text{ for some } i \in D,\ x_i \geq 1 \text{ and } a = v, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

It is also important to note that this equivalence holds under quite general conditions. The fact that setup and processing times are interruptible in our formulation does not cause any complications, as the function $B_i(t)$ in (5) is a step function that increases only when jobs are finally completed and exit from the system.

It is clear from (7) that the single-step reward associated with processing a job of type $i$ tends to infinity as $\alpha \to 1$, because the extra cost associated with holding a job in the system forever becomes infinite. Therefore, following Duenyas and Van Oyen (1996), we derive a bounded 'reward rate' by dividing the expected total discounted cost saving from choosing action $a$ under state $\mathbf{x}$ (namely, $w(\mathbf{x}, a)$) by the discounted length of the time horizon, $\sum_{t=0}^{\infty} \alpha^t = (1-\alpha)^{-1}$, to obtain the following reward function for undiscounted problems:

$$r(\mathbf{x}, a) = \lim_{\alpha \to 1} (1-\alpha) w(\mathbf{x}, a) = \begin{cases} c_i \mu_i, & \text{if } v = i \text{ for some } i \in D,\ x_i \geq 1 \text{ and } a = v, \\ 0, & \text{otherwise.} \end{cases} \tag{8}$$

**Remark 3.1.** *If an extension to phase-type processing times is to be considered (as discussed in Remark 2.1) one can derive an expression for the reward rate $r(\mathbf{x}, a)$ similar to the one in (8), except that the condition $x_i \geq 1$ should be replaced by $x_i \equiv 1 \pmod{k_i}$, since a service*

*completion is possible only if the job at the front of the queue has one phase remaining.*

The reward rates $r(\mathbf{x}, a)$ are useful in the development of index heuristics because they depend only on the server's current location and choice of action. On the other hand, the cost function (3), despite its simple appearance, has a disadvantage in that the aggregate cost incurred by the system at any given time step must be calculated according to the number of jobs present at every demand point in the system. Thus, it is more convenient to use reward rates when seeking to develop *index heuristics*, which operate by associating easily computable scores (or *indices*) with the decision options under any given state. Throughout this section, the index heuristics that we develop are based on the reward rates defined in (8). However, in our numerical experiments in Section 4, we evaluate and compare the performances of different policies according to the original cost formulation presented in Section 2.

### 3.1 The DVO heuristic

We refer to the heuristic policy proposed in Duenyas and Van Oyen (1996) as the 'DVO heuristic' for short. The DVO heuristic does not allow processing or switching times to be interrupted, and therefore decision epochs occur only if (1) the server finishes processing a job, (2) the server arrives at a demand point, or (3) the server is idle and a new job arrives in the system. As such, we cannot represent the DVO heuristic as a stationary policy in our system, since the action chosen at a particular time step constrains the actions chosen at future time steps. For example, if the server begins switching from one demand point to another, then it must continue to do so until it arrives at the new demand point. Similarly, if the server remains at a non-empty demand point then it must continue to do so until a job is processed. However, despite its lack of compatibility with our MDP formulation, we can still consider the DVO heuristic as a nonstationary policy in our system and estimate its performance using simulation experiments. We use this approach to provide a useful benchmark for our heuristic policies in Section 4.

The complete steps of the DVO heuristic are described in Duenyas and Van Oyen (1996). For the reader's convenience, we provide a summary of how the heuristic makes decisions at the three different types of decision epoch (1)-(3) mentioned above in Appendix B.

### 3.2 $K$-stop heuristics

Although the DVO heuristic can be applied to our problem and its performance in a particular system can be simulated, it is unlikely to achieve near-optimal performances in general. There are two main reasons for this: (i) it over-constrains the action sets by not allowing switching or processing times to be interrupted; (ii) it chooses the next demand point to switch to without considering the proximity of that demand point to other demand points in the network. The latter of these two properties implies that the heuristic works in a short-sighted (myopic) way, and fails to recognize the potential benefits of moving to demand points that are located near other demand points. This is not a weakness in the problem studied by Duenyas and Van Oyen (1996), since in their model, the switching (or setup) time to move from node $i$ to node $j$ only depends on the destination node $j$, so the currently-occupied node does not have

any effect on future switching times. In our network-based formulation, however, it clearly makes sense to consider heuristic policies that can take the network topology into account when making decisions.

In this subsection we introduce a class of heuristic policies designed to exploit the novel features of our problem. We refer to these policies as $K$-stop heuristics, since the decision at any particular time step is made by assuming that the server will visit a sequence of up to $K$ demand points (where $K$ is a pre-determined integer) and serve these demand points until exhaustion. The name '$K$-stop' derives from the fact that the server is assumed to visit a sequence of demand points in the same way that a public transport service visits different 'stops' along its route; however, it is important to emphasize that the optimal sequence to be followed by the server is calculated only for the purposes of making a single decision at a particular time step, and the server is not actually committed to following this route in future time steps. Thus, given that each time step is also a decision epoch in our problem formulation, routes are continuously re-optimized and therefore switching and processing times can be interrupted.

Let $K \geq 1$ be a pre-determined integer. At any given time step we consider all non-empty sequences of demand points with length not exceeding $K$; that is, we consider sequences of the form $s = (s_1, s_2, ..., s_m)$, where $1 \leq m \leq K$ and $s_j \in D$ for each $j = 1, 2, ..., m$. We also require that all elements of the sequence are distinct (that is, $s_j \neq s_k$ for $j, k \in \{1, ..., m\}$ with $j \neq k$), and furthermore the first element $s_1$ must be different from the server's current location $v$. For each sequence $s$ we calculate a 'reward rate', also referred to as an *index*, and this index is calculated by assuming that the server visits demand points in the order $s_1, ..., s_m$ and serves each demand point exhaustively before moving to the next one. Some further conditions related to stability and idling (described below) are also used to decide which sequences should be considered 'eligible'. After identifying the eligible sequence with the highest index, we choose an action at the current time step by assuming that this sequence should be followed.

Before providing full details of the $K$-stop heuristic, it will be useful to introduce some extra notation. Given a state $\mathbf{x} = (v, (x_1, ..., x_d))$ and a sequence $s$, we define $s_0 = v$ for notational convenience; that is, $s_0$ is the server's current node (which may not be a demand point). We also use $\delta(i, j)$ to denote the length of the shortest path (in terms of the number of nodes that must be traversed) from node $i \in V$ to $j \in V$, with $\delta_{ii} := 0$ for $i \in V$. For a given state $\mathbf{x}$, sequence $s$ and each $j = 1, ..., |s|$, we define two quantities $T_j(\mathbf{x}, s, t)$ and $R_j(\mathbf{x}, s, t)$ as follows:

$$T_j(\mathbf{x}, s, t) = \frac{x_{s_j} + \lambda_{s_j}\left[t + \sum_{k=1}^{j-1}\left(\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, t)\right) + \delta(s_{j-1}, s_j)/\tau\right]}{\mu_{s_j} - \lambda_{s_j}}, \quad t \geq 0, \tag{9}$$

$$R_j(\mathbf{x}, s, t) = c_{s_j}\mu_{s_j}T_j(\mathbf{x}, s, t), \quad t \geq 0. \tag{10}$$

In words, $T_j(\mathbf{x}, s, t)$ is an approximation for the expected amount of time required for the server to process all jobs at node $s_j$ after arriving there, assuming that it remains idle for $t$ time units at the current node $v$ before beginning to follow the sequence $s$. This approximation is obtained by assuming a fluid-type model of the system dynamics in which jobs arrive at node $i \in D$ at a continuous rate $\lambda_i$ and are processed at a continuous rate $\mu_i$, and the server requires $1/\tau$ time units to move between any adjacent pair of nodes in the network. On the other hand, $R_j(\mathbf{x}, s, t)$

is the total reward earned while the server is processing jobs at node $s_j$ in this fluid model. To make sense of equation (9), note that the number of jobs present at node $s_j$ when the server arrives there is given by adding the original number of jobs under state $\mathbf{x}$ (that is, $x_{s_j}$) to the number of new jobs that arrive while the server is either idling, traveling to one of the earlier demand points in the sequence, processing jobs at one of the earlier demand points, or traveling towards $s_j$. Furthermore, once the server arrives at node $s_j$, the number of jobs at $s_j$ decreases at a net rate of $\mu_{s_j} - \lambda_{s_j}$. We note that the equations 9 and (10) can also be adapted quite easily to more general models with edge-dependent switching rates, implying phase-type setup times, as discussed in Remark 2.1. In this case, the expression $\delta(s_{k-1}, s_k)/\tau$ should be replaced by the minimum possible expected amount of time to travel from node $s_{k-1}$ to $s_k$, taking into account the rates of travel on individual edges.

It is important to clarify that $t$ is interpreted as a certain amount of 'idle time' before the server begins to follow the sequence $s$. During this idle time, the server remains idle at its current node $v$ and (in the event that $v$ is a demand point) does not process any jobs there. This is somewhat contrary to our MDP formulation in Section 2, which assumes that if the server remains at a non-empty demand point then it must be processing jobs there. However, we make this 'idle time' assumption only for the purpose of deriving index quantities for use in our heuristics. Next, for a given state $\mathbf{x}$ and sequence $s$, we define

$$\psi(\mathbf{x}, s, t) = \frac{\sum_{k=1}^{|s|} R_k(\mathbf{x}, s, t)}{t + \sum_{k=1}^{|s|} [\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, t)]}, \quad t \geq 0, \tag{11}$$

$$\phi_j(\mathbf{x}, s, t) = \frac{\sum_{k=1}^{j} R_k(\mathbf{x}, s, t)}{t + \sum_{k=1}^{j} [\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, t)] + \delta(s_j, v)/\tau}, \quad t \geq 0,\ j \in \{1, ..., |s|\}. \tag{12}$$

We can interpret $\psi(\mathbf{x}, s, t)$ as the average reward per unit time earned while the server follows sequence $s$. On the other hand, $\phi_j(\mathbf{x}, s, t)$ is the average reward earned during a truncated sequence that visits the demand points $s_1, s_2, ..., s_j$ and then returns to the starting node $v$. In this case, the average reward calculation includes the time taken to switch back to node $v$. The quantities $\psi(\mathbf{x}, s, t)$ and $\phi_j(\mathbf{x}, s, t)$ are used for slightly different purposes in our heuristics. Recall that we only consider sequences in which the first node $s_1$ differs from the server's current location $s_0$, and therefore the denominators in (11) and (12) are always non-zero.

Before presenting the algorithmic steps for the $K$-stop heuristic, we prove a useful property of the average reward $\psi(\mathbf{x}, s, t)$.

**Lemma 3.2.** *For any given state $\mathbf{x} \in S$ and sequence $s$, the average reward $\psi(\mathbf{x}, s, t)$ is a monotonic function of $t$.*

*Proof.* We prove the statement by showing that the derivative $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)$ has the same sign (either positive, negative or zero) for all $t \geq 0$. The state variables $x_1, ..., x_d$ can be regarded as constants. The system parameters (including arrival rates, processing rates, switching rate, cost rates and the distances between nodes in the network) can also be regarded as constants. Therefore, using a trivial inductive argument, we can show that the quantities $T_j(\mathbf{x}, s, t)$ defined in (9) are linear functions of $t$, for $j = 1, ..., |s|$. Hence, the average rewards $R_j(\mathbf{x}, s, t)$ in (10) are also linear in $t$. It follows that the quantity $\psi(\mathbf{x}, s, t)$ in (11) is a ratio of two linear

functions, with the general form $(a_1 + b_1 t)/(a_2 + b_2 t)$, where $a_1, a_2, b_1, b_2$ are positive constants. Any such function can be represented graphically as a hyperbola, with a derivative of the form $(a_2 b_1 - a_1 b_2)/(a_2 + b_2 t)^2$. Hence, the sign of the derivative is the same for any $t \geq 0$. □

Next, we provide details of the steps used in the $K$-stop heuristic algorithm. Recall that $K \geq 1$ is a pre-determined, fixed integer and we let $\mathbf{x} = (v, (x_1, ..., x_d))$ denote the current state. At each time step, there are three possible cases: (1) the server is at a non-empty demand point, (2) the server is at an empty demand point, (3) the server is at an intermediate stage. The details below explain how actions are chosen in each of these cases.

**$K$-stop heuristic algorithm**

1. If the server is at a non-empty demand point ($v \in D$ and $x_v > 0$) then we perform the following steps:

   (a) Let $\mathcal{S}$ be the set of all sequences of the form $s = (s_1, s_2, ..., s_m)$, where $1 \leq m \leq K$, $s_j \in D$ for each $j \in \{1, 2, ..., m\}$, $s_1 \neq v$ and $s_i \neq s_j$ for any pair of elements $s_i, s_j \in s$ with $i \neq j$. Initialize $\sigma = \emptyset$ as a set of 'eligible' sequences from which we will later select a reward-maximizing sequence.

   (b) For each sequence $s \in \mathcal{S}$, define $\beta_j(\mathbf{x}, s, t)$ for $j = 1, ..., |s|$ as follows:

$$\beta_j(\mathbf{x}, s, t) := \begin{cases} \dfrac{\sum_{k=1}^{j} R_k(\mathbf{x}, s, t)}{\sum_{k=1}^{j} T_k(\mathbf{x}, s, t)} \rho + c_v \mu_v (1 - \rho), & \text{if } v \notin \{s_1, s_2, ..., s_j\}, \\ 0, & \text{otherwise.} \end{cases} \tag{13}$$

   If $\frac{\partial}{\partial t} \psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$ and $\phi_j(\mathbf{x}, s, 0) \geq \beta_j(\mathbf{x}, s, 0)$ for all $j \in \{1, ..., |s|\}$, then add $s$ to the set $\sigma$. Otherwise, do not make any change to the set $\sigma$.

   (c) If $\sigma = \emptyset$, then the action chosen under state $\mathbf{x}$ should be to remain at node $v$. Otherwise, let $s^*$ denote the sequence in $\sigma$ with the largest value of $\psi(\mathbf{x}, s, 0)$, with ties broken according to some fixed priority ordering of the demand points in $D$, applied lexicographically to the sequences in $\sigma$. The action chosen under $\mathbf{x}$ should be to switch to the first node on a shortest path from $v$ to $s_1^*$.

2. If the server is at either an empty demand point ($v \in D$ and $x_v = 0$) or an intermediate stage ($v \in N$) then we perform the following steps:

   (a) Let $\mathcal{S}$ be defined in the same way as in step 1(a). Initialize $\sigma_1 = \emptyset$ as a set of 'high-priority' eligible sequences and $\sigma_2 = \emptyset$ as a set of 'low-priority' eligible sequences.

   (b) For each sequence $s \in \mathcal{S}$, add $s$ to $\sigma_2$ if and only if $\frac{\partial}{\partial t} \psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$.

   (c) For each sequence $s \in \sigma_2$, define $\gamma(\mathbf{x}, s, t)$ as follows

$$\gamma(\mathbf{x}, s, t) := \frac{\sum_{k=1}^{|s|} R_k(\mathbf{x}, s, t)}{\sum_{k=1}^{|s|} T_k(\mathbf{x}, s, t)} \rho, \quad t \geq 0. \tag{14}$$

Let $\mathbf{y}$ denote a state identical to $\mathbf{x}$ except that the server is located at $s_1$ (the first node in sequence $s$) instead of $v$. If $\psi(\mathbf{x}, s, 0) \geq \gamma(\mathbf{x}, s, 0)$ and either (i) $|s| = 1$ or (ii) $|s| \geq 2$ and $\psi(\mathbf{y}, s, 0) \geq \gamma(\mathbf{y}, s, 0)$, then remove $s$ from $\sigma_2$ and add it to $\sigma_1$. Otherwise, do not make any changes.

(d) If $\sigma_1$ is non-empty, set $\sigma = \sigma_1$. Otherwise, set $\sigma = \sigma_2$.

(e) Carry out step 1(c).

The steps presented above require some explanation. Consider the first case, where the server is at a non-empty demand point. In this case, $\sigma$ is a set of 'eligible' sequences and we choose a sequence from this set that yields the highest average reward, $\psi(\mathbf{x}, s, 0)$, in the fluid model. To determine whether some sequence $s \in \mathcal{S}$ is eligible, we need to check two conditions. If the condition $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$ is satisfied, this indicates that we should begin following sequence $s$ immediately (i.e. we should take an immediate step towards the first node in $s$), as the average reward will not increase if we wait for some 'idle time' before following $s$. Note that, due to Lemma 3.2, we know that if $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$ then $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t) \leq 0$ for all $t \geq 0$, so there is no advantage to be gained by waiting for any amount of idle time. The second condition, $\phi_j(\mathbf{x}, s, 0) \geq \beta_j(\mathbf{x}, s, 0)$, is automatically satisfied if the server returns to its current location $v$ at some point in the first $j$ stops of the sequence (since we define $\beta_j(\mathbf{x}, s, t) = 0$ in this case). In other cases, the condition is intended to provide a balance between two important considerations. To elaborate on this, note that if $\rho$ is close to 1 then the condition $\phi_j(\mathbf{x}, s, 0) \geq \beta_j(\mathbf{x}, s, 0)$ is almost equivalent to

$$\frac{\sum_{k=1}^{j} T_k(\mathbf{x}, s, 0)}{\sum_{k=1}^{j} [\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, 0)] + \delta(s_j, v)/\tau} \geq \rho,$$

which states that if the server serves nodes $s_1, ..., s_j$ exhaustively and then returns to node $v$, then the proportion of time spent processing jobs (as opposed to switching between nodes) during this time should be at least $\rho$. This is an important condition for system stability. On the other hand, if $\rho$ is close to zero, then $\phi_j(\mathbf{x}, s, 0) \geq \beta_j(\mathbf{x}, s, 0)$ is almost equivalent to

$$\frac{\sum_{k=1}^{j} R_k(\mathbf{x}, s, 0)}{\sum_{k=1}^{j} [\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, 0)] + \delta(s_j, v)/\tau} \geq c_v \mu_v,$$

which states that the average reward earned while serving nodes $s_1, ..., s_j$ and then returning to $v$ should be at least as great as the average reward that would be earned by continuing to process jobs at the current node; that is, $c_v \mu_v$. Effectively, this states that the server should only move away from node $v$ to process jobs at other demand points if there exists a sequence $s$ for which the average reward is greater than the average reward for remaining at $v$. The condition $\phi_j(\mathbf{x}, s, 0) \geq \beta_j(\mathbf{x}, s, 0)$ is a generalization of a rule used by the DVO heuristic in Duenyas and Van Oyen (1996), which effectively considers sequences of length 1 only.

From a computational standpoint, we note that it is not necessary to derive expressions for the derivatives $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)$ in terms of the system parameters. Indeed, these expressions become very complicated as the sequence length $|s|$ increases. Instead, we can simply compare the values of $\psi(\mathbf{x}, s, 0)$ and $\psi(\mathbf{x}, s, \varepsilon)$, where $\varepsilon$ is some small positive number. This approach is justified by

Lemma 3.2, which implies that $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$ if and only if $\psi(\mathbf{x}, s, \varepsilon) \leq \psi(\mathbf{x}, s, 0)$.

Next, consider case 2, where the server is at either an empty demand point or an intermediate stage. In this case, by using two different sets $\sigma_1$ and $\sigma_2$, we effectively separate the eligible sequences into two different subsets, with sequences in $\sigma_1$ being given a higher priority for selection than those in $\sigma_2$. The DVO heuristic in Duenyas and Van Oyen (1996) also separates the decision options into two sets when the server is at an empty demand point, but the conditions that we use in our sequence-based algorithm are quite different. Firstly, as in case 1, we introduce a derivative-based condition and require sequence $s$ to satisfy $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$ in order to be included in $\sigma_2$. This condition implies that there is no benefit to be gained by waiting for some 'idle time' before following $s$. In order to be included in the higher-priority set $\sigma_1$, sequences must also satisfy the condition $\psi(\mathbf{x}, s, 0) \geq \gamma(\mathbf{x}, s, 0)$ and (for sequences of length greater than one only) $\psi(\mathbf{y}, s, 0) \geq \gamma(\mathbf{y}, s, 0)$, where $\mathbf{y}$ is a state identical to $\mathbf{x}$ except that the server is located at node $s_1$ instead of $v$. We defer discussion of these conditions to the proof of Theorem 3.3, where it is shown that they are sufficient to ensure that the sequence $s$ remains included in the set $\sigma_1$ at all stages while the server travels from $v$ to $s_1$, provided that no further jobs arrive in the meantime. This is useful in order to ensure that the server follows a consistent path through the network, rather than changing direction without an obvious reason. We also note that the condition $\psi(\mathbf{x}, s, 0) \geq \gamma(\mathbf{x}, s, 0)$ is somewhat similar to the condition $\phi_j(\mathbf{x}, s, 0) \geq \beta_j(\mathbf{x}, s, 0)$ used in case 1, but there are some differences. Firstly, the condition applies to the entire sequence $s$, rather than the subsequences $(s_1, ..., s_j)$ for $1 \leq j \leq |s|$, so in this sense it is more relaxed than the condition used in case 1. Secondly, the expression for $\gamma(\mathbf{x}, s, t)$ does not include the extra term $c_v\mu_v(1-\rho)$ that can be seen in the expression for $\beta_j(\mathbf{x}, s, t)$, implying that we do not wish to ensure that the average reward obtained by following sequence $s$ is greater than $c_v\mu_v$. Indeed, this is logical since there are no jobs present at node $v$, and therefore it is not possible to earn any immediate reward by remaining there.

It is worthwhile to emphasize that even if a sequence $s$ fails to satisfy the conditions for inclusion in $\sigma_1$, it may still be included in $\sigma_2$, in which case it may be selected in step 2(e) if $\sigma_1$ is empty. The conditions for inclusion in $\sigma_1$ (that is, $\psi(\mathbf{x}, s, 0) \geq \gamma(\mathbf{x}, s, 0)$ and $\psi(\mathbf{y}, s, 0) \geq \gamma(\mathbf{y}, s, 0)$) are primarily intended to promote system stability. However, even if sequence $s$ fails to satisfy these conditions, it may still be better to follow $s$ rather than remaining idle at node $v$. This is intuitive, since idling at an empty demand point or at an intermediate stage is not necessarily helpful for maintaining stability, so following sequence $s$ should not be seen as a worse option in this regard.

Our next result states that the $K$-stop heuristic has the property of *pathwise consistency*, which implies that the server follows a consistent path through the intermediate stages of the network as long as the number of jobs in the system remains unchanged. This is an intuitively appealing property, as it means that the server avoids wasting time (by going back and forth between intermediate stages, for example) when moving between demand points.

**Theorem 3.3.** ***(Pathwise consistency.)*** *Suppose the server is located at an intermediate stage $v \in N$ and the system operates under the $K$-stop heuristic policy. Then there exists a demand point $j^* \in D$ such that the server moves directly along a shortest path to node $j^*$ until either (i) it arrives at node $j^*$, or (ii) a new job arrives in the system.*

Proof of Theorem 3.3 can be found in Appendix C. It is important to clarify that if a new job arrives in the system while the server is moving towards the demand point $j^*$ referred to in the theorem, then the server may change direction and move towards a different demand point instead. This does not contradict the theorem; indeed, the theorem does not make any claim about what happens after the next job arrival. The theorem essentially states that the server follows a consistent path until the next time a new job arrives, and we are able to use this in order to prove that the expected amount of time until it arrives at a demand point must be finite. We state this as a corollary below and provide a proof in Appendix D.

**Corollary 3.4.** *Suppose the conditions of Theorem 3.3 apply. Then the expected amount of time until the server arrives at a demand point is finite. More specifically, if* $T_{\text{switch}}$ *denotes the amount of time until the server arrives at a demand point, then*

$$\mathbb{E}[T_{\text{switch}}] \leq \frac{M}{\tau}\left(\frac{\Lambda+\tau}{\tau}\right)^{2(M-1)},$$

*where* $\Lambda := \sum_{i=1}^{d} \lambda_i$ *and* $M := \max_{\{i\in N,\, j\in D\}} \delta(i,j)$ *denotes the maximum distance between an intermediate stage and a demand point.*

The next result concerns a special case of the problem in which job types are homogeneous, which means that the arrival rates $\lambda_i$, service rates $\mu_i$ and holding costs $c_i$ are identical for all $i \in D$. In this scenario, we are able to prove that the system is stable under the policy given by the $K$-stop heuristic. Furthermore, if $V$ is a complete graph (so that all demand points are directly connected to each other) then the policy given by the $K$-stop heuristic is optimal. We emphasize that these statements hold for any number of demand points and any $K \geq 1$, and the stability part of the result also holds for any network layout.

**Theorem 3.5.** *Suppose we have a homogeneous system in which* $\lambda_1 = ... = \lambda_d$, $\mu_1 = ... = \mu_d$ *and* $c_1 = ... = c_d$. *Then, for any* $K \geq 1$, *the system is stable under the* $K$*-stop heuristic policy when* $\rho < 1$. *Furthermore, if* $V$ *is a complete graph then the* $K$*-stop heuristic policy is optimal.*

Proof of the theorem can be found in Appendix E. Subject to the conditions of the theorem, it can be shown that the server visits all demand points infinitely often under the $K$-stop policy and also serves demand points exhaustively on each visit. This ensures that the system is stable. In the case where $V$ is a complete graph, it can also be shown that the $K$-stop policy directs the server to switch to the demand point with the largest number of jobs if it is currently at an empty demand point, and the same rule is used by an optimal policy. In the latter scenario, the optimal policy is a type of 'Serve the Longest Queue' (SLQ) policy, and we note that SLQ policies have previously been studied in certain kinds of polling systems. In particular, Liu et al. (1992) showed that a SLQ policy is optimal for a 'symmetric' polling system, similar to the one described in Theorem 3.5, in which arrival and service rates are the same at all demand points. However, to the best of our knowledge, no similar result has previously been proved for a system with interruptible switching and processing times.

It should be noted that if $K \geq 2$ then the number of sequences in $\mathcal{S}$ increases rapidly with the number of demand points $d$, implying that the computational requirements of the $K$-stop heuristic become unmanageable in large-scale problems. In the next subsection we propose

an alternative heuristic under which the number of indices to be calculated at any time step increases only linearly with $d$, enabling greater scalability.

### 3.3 ($K$ from $L$)-stop heuristic

As explained in Section 3.2, the $K$-stop heuristic considers all possible sequences of demand points $(s_1, s_2, ..., s_m)$ at each time step, for each $1 \leq m \leq K$. The only restrictions are that all demand points in the sequence are distinct and the first demand point must be different from the server's current location. This implies that the number of sequences to be considered at each time step is of order $d^K$, which grows polynomially with the number of demand points $d$ (for fixed $K$) and grows exponentially with $K$. Hence, in order for the $K$-stop heuristic to be computationally feasible, $d$ and $K$ must be relatively small. In our numerical experiments in Section 4 we restrict attention to systems with $d \leq 8$ and consider $K \in \{1, 2, 3, 4\}$.

In this subsection we propose a modified version of the $K$-stop heuristic in which the number of sequences considered at each time step increases only linearly with $d$. Note that if $K = 1$ then the $K$-stop heuristic already has this property, since it considers only sequences of the form $(j)$, for $j \in D$. Our modified heuristic works as follows: suppose the system is in state $\mathbf{x} \in S$ at an arbitrary time step. First, we carry out the same steps as if we are using the $K$-stop heuristic with $K = 1$, and calculate the indices $\psi(\mathbf{x}, (j), 0)$ for each $j \in D$. In this step we also allow the demand point $j$ to be equal to the server's current location $v$ (unlike in the standard $K$-stop heuristic) and set $\psi(\mathbf{x}, (v), 0)$ equal to $c_v\mu_v$ if $v$ is non-empty, and zero otherwise. After computing $\psi(\mathbf{x}, (j), 0)$ for each $j \in D$, we then form a set $\mathcal{L}$ of size $L$ (where $L \leq d$ is a pre-determined integer) consisting of a limited number of demand points. We consider a couple of different ways of selecting the demand points to be included in $\mathcal{L}$:

- **Impartial method:** In this method, we simply choose the $L$ demand points with the highest indices $\psi(\mathbf{x}, (j), 0)$, regardless of their positions in the network. (Ties are broken arbitrarily.)
- **Stratified method:** In some systems, there may be a natural way of dividing the network into 'clusters' of demand points, with any pair of demand points in the same cluster being relatively close to each other. In this case, we can select a pre-determined number of demand points from each cluster (taking the ones with the highest indices), in such a way that the total number of demand points selected is $L$.

In the impartial method we also enforce the following rule: if the server is at an empty demand point or an intermediate stage then we divide the demand points into two sets. The first set consists of sequences $(j)$ such that $\psi(\mathbf{x}, (j), 0) \geq \gamma(\mathbf{x}, (j), 0)$ and the second set consists of sequences $(j)$ such that $\psi(\mathbf{x}, (j), 0) < \gamma(\mathbf{x}, (j), 0)$, where $\gamma(\cdot)$ is defined in (14). If the first set includes at least $L$ sequences then we select the $L$ sequences with the highest values of $\psi(\mathbf{x}, (j), 0)$ to be included in $\mathcal{L}$. Otherwise, all of the sequences in the first set are included in $\mathcal{L}$ and we obtain the remaining sequences by choosing the sequences with the highest indices in the second set. This is consistent with the prioritization rule described in step 2(c) of the

$K$-stop heuristic. In the stratified method a similar rule is used, except the two sets are formed for each cluster separately, and in each cluster we choose a pre-determined number of demand points from the two sets, with the first set being given priority over the second set as in the impartial method.

After forming the set $\mathcal{L}$, we then consider all possible sequences $(s_1, ..., s_m)$ for $1 \leq m \leq K$, where $s_j \in \mathcal{L}$ for each $j = 1, ..., m$, and carry out the rest of the steps in the $K$-stop heuristic as described in Section 3.2. The sequences are required to satisfy the same eligibility conditions described in Section 3.2 in order to be selected.

We note that the impartial method is simpler and might often perform better than the stratified method, but the stratified method offers a potential advantage in that it forces the server to consider moving to other clusters in the network. Under the impartial method, there is a risk that if all of the $L$ demand points selected are in close proximity to the server's current location then the server acts in a short-sighted way, as (given that demand points are selected based on indices for sequences of length one only) it fails to detect the potential benefits of moving to another cluster and serving multiple demand points within that cluster.

As an example, consider the system shown in Figure 3, with 8 demand points and 4 intermediate stages. Suppose we use the ($K$ from $L$)-stop heuristic with $K = 2$ and $L = 4$. For each demand point $j = 1, ..., 8$ we calculate the index $\psi(\mathbf{x}, (j), 0)$. Under the impartial method, assuming that the server is at a non-empty demand point, we choose the 4 demand points $j$ with the highest indices and then consider all possible sequences of length 1 or 2 involving these 4 demand points only. The total number of sequences to consider is $4 + (4!/2!) = 16$. If the server is at an empty demand point or an intermediate stage then the process is similar except we prioritize sequences $(j)$ that satisfy the condition $\psi(\mathbf{x}, (j), 0) \geq \gamma(\mathbf{x}, (j), 0)$. In this case we still obtain 16 sequences. Under the stratified method, a logical approach (given the layout of the network) is to define $\{1, 2, 3, 4\}$ as one cluster and $\{5, 6, 7, 8\}$ as another cluster, and select two demand points from each according to their index values. Suppose, for example, we choose 2 and 3 from the first cluster and 6 and 7 from the second cluster. Then, in the next step, we consider all possible sequences of length 1 or 2 consisting of demand points from the set $\{2, 3, 6, 7\}$. Thus, we again consider 16 sequences in total.

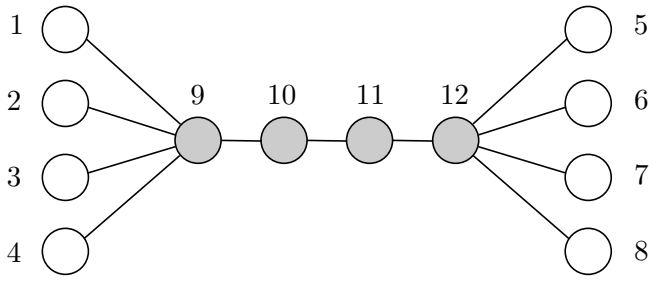


Figure 3: A network with 4 demand points on the left, 4 demand points on the right and 4 intermediate stages.

In general, in large systems (with a lot of demand points), the bulk of the computational effort required is in the calculation of indices $\psi(\mathbf{x}, (j), 0)$ for each $j \in D$. Following this, the

number of sequences to be considered is $\sum_{m=1}^{K}(L!/(L-m)!)$, which is independent of $d$ and can be kept relatively small. Thus, if $K$ and $L$ are fixed then the computational effort required by the ($K$ from $L$)-stop heuristic increases only linearly with $d$. If $L = d$ then the ($K$ from $L$)-stop heuristic becomes equivalent to the $K$-stop heuristic. Thus, it is obvious that the ($K$ from $L$)-stop heuristic should perform worse than the $K$-stop heuristic in general, since it considers a smaller number of possible sequences. However, in the next section, we show that it may be able to achieve a similar performance at smaller computational expense.

# 4 Numerical results

In this section we report the results of numerical experiments in order to compare the performances of the heuristics described in Section 3. In Section 4.1 we focus on a specific network layout, with two 'clusters' of demand points separated by a series of intermediate stages. In Section 4.2 we present results from problem instances with randomly-generated network layouts. Finally, in Section 4.3 we provide additional details of the computational requirements of our heuristics.

## 4.1 Two clusters of demand points

In this subsection we consider a relatively simple network layout, shown in Figure 4, in which two distinct 'clusters' of demand points are separated by a chain of $n$ intermediate stages. The demand points on the left-hand side belong to a cluster denoted by $D_1$ of size $d_1$, and similarly the demand points on the right-hand side form a cluster $D_2$ of size $d_2$. In order to move from $D_1$ to $D_2$ or vice versa, the server must pass through all of the intermediate stages in succession. Also, as the figure indicates, in order to move from one demand point to another point in the same cluster, it must pass through one intermediate stage (it is not possible to move directly from one demand point to another). By adjusting the value of $n$ we can vary the distance between the two clusters. In the case $n = 1$, we obtain a special case where all demand points are equidistant from each other. In these experiments we consider $1 \leq n \leq 6$, $1 \leq d_1 \leq 4$ and $1 \leq d_2 \leq 4$. Thus, the number of demand points $d$ satisfies $2 \leq d \leq 8$.

Our numerical study is based on $11,250$ randomly-generated problem instances. In each instance we uniformly sample the values of $d_1$, $d_2$ and $n$ from the ranges specified above. We also randomly generate the values of $\rho$, $\tau$, $\lambda_i$, $\mu_i$ and $c_i$ (for each $i \in D$) using a method that ensures consideration of a wide range of different scenarios for the system parameters. For full details of our parameter generation methods, please refer to Appendix F.

In each problem instance, we test the performances of the heuristic policies in Section 3 using simulation experiments. The DVO heuristic described in Section 3.1 can be regarded as a nonstationary policy for our MDP and its performance can be simulated. This provides a useful benchmark for our other heuristics. We test the $K$-stop heuristic for each $K \in \{1, 2, 3, 4\}$. We note that in instances where $d < K$ (i.e. the number of demand points is smaller than $K$), the $K$-stop heuristic becomes equivalent to the $(K-1)$-stop heuristic, as it is not possible to define a sequence of length $K$ such that all demand points in the sequence are distinct (but the

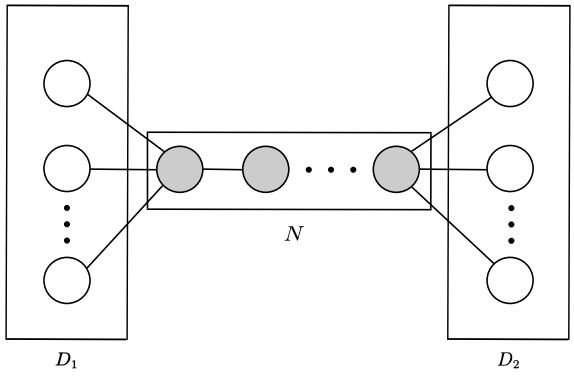


Figure 4: A diagrammatic representation of the network with $d_1$ demand points on the left, $d_2$ demand points on the right and $n$ intermediate stages.

definition of the heuristic allows it to consider sequences of length smaller than $K$). For example, if $d = 2$ then the 2-stop, 3-stop and 4-stop heuristics are equivalent. We also implement the ($K$ from $L$)-stop heuristic with $K = 2$ and $L = 4$ in each instance, using both the impartial and stratified methods. In the stratified case, we adopt the obvious strategy of defining $D_1$ and $D_2$ as separate clusters.

In each problem instance, the simulation process for each heuristic consists of two phases: a warm-up period and a main simulation run. Since the traffic intensity $\rho$ is randomly sampled from a continuous distribution on $(0.1, 1)$ (see Appendix F), it may approach 1, potentially leading to very large average queue sizes and possible system instability under some of our heuristics. For this reason, in systems with $\rho \geq 0.9$, we allow the warm-up period to continue until either the system appears to stabilize or a termination condition is reached. To determine the point at which the system stabilizes, we sample the average costs after $r, 2r, 3r, ...$ time steps (with $r = 1000$) and, at each of these stages, apply the Mann–Kendall test with the Hamed–Rao correction (see Hamed and Rao (1998), Gocic and Trajkovic (2013)), a nonparametric method for detecting monotonic trends in time series while accounting for the presence of autocorrelation. This test determines whether or not the sequence of average costs shows a monotonically increasing trend, and the warm-up period ends at the point where such a trend is no longer detected. For instances with $\rho < 0.9$, a fixed warm-up length of $10,000$ steps is applied, which is generally sufficient for steady state conditions to be reached. For implementation details of our simulation experiments, please refer to Appendix G.

In instances where $d$ is small, it may be possible to compute the optimal long-run average cost $g^*$ using dynamic programming (specifically, relative value iteration). Although DP algorithms require a finite state space, the 'approximating sequence' method of Sennott (1997) can be used to obtain the infinite-state optimal value as a limit of finite-state optimal values (for further explanation, see the proof of Theorem 2.2). In each problem instance we have used an iterative method, described in Appendix H, to test whether or not it is computationally feasible to obtain $g^*$ using DP. We have found that it is usually only feasible to compute $g^*$ if $d \leq 3$ and $\rho$ is not too large. In total, we have been able to compute $g^*$ in 1229 of the 11,250 instances.

Table 1 shows, for each heuristic policy, a 95% confidence interval for the mean percentage

suboptimality of the heuristic in comparison to the optimal value $g^*$ computed using relative value iteration, based on the results from 1229 'small' problem instances. For each heuristic policy $\theta \in \{\text{DVO, 1-stop, 2-stop, 3-stop}\}$, the percentage suboptimality is calculated as $100 \times (g_\theta - g^*)/g^*$. The 10th, 25th, 50th, 75th and 90th percentiles of the distribution of percentage suboptimality for each heuristic are also reported. Given that $d \leq 3$ in all of these small instances, the 4-stop policy is equivalent to the 3-stop policy, so we do not include it in the table. Similarly, the (2 from 4)-stop heuristics are equivalent to the 2-stop heuristic because they always consider all demand points in the network as potential sequence elements, so we do not include them either. The table shows that the DVO heuristic (which does not allow switching or processing times to be interrupted) is about 22% suboptimal on average in these instances. The $K$-stop heuristics are able to improve upon the DVO heuristic very significantly. The $K = 2$ and $K = 3$ policies are within 4% of optimality on average, and in more than 50% of instances they are within 3%. As expected, the performance tends to improve as $K$ increases (at the expense of greater computation time), although increasing $K$ from 2 to 3 gives only a small additional improvement.

Table 1: Percentage suboptimalities of heuristic policies in 1229 'small' problem instances.

| Heuristic | Mean | 10th pct. | 25th pct. | 50th pct. | 75th pct. | 90th pct. |
|---|---|---|---|---|---|---|
| DVO | $22.05 \pm 0.83$ | 6.61 | 11.68 | 19.50 | 28.88 | 40.03 |
| 1-stop | $5.97 \pm 0.45$ | 0.73 | 1.92 | 3.93 | 7.20 | 13.08 |
| 2-stop | $3.96 \pm 0.24$ | 0.42 | 1.49 | 2.92 | 5.01 | 8.44 |
| 3-stop | $3.73 \pm 0.24$ | 0.35 | 1.39 | 2.75 | 4.64 | 7.80 |

Table 2 shows a comparison between the heuristic policies across all $11,250$ instances, with the DVO heuristic used as a benchmark. In each instance we report 95% confidence intervals for the mean percentage improvements of the $K$-stop and ($K$ from $L$)-stop heuristics over the DVO heuristic, and also report the 10th, 25th, 50th, 75th and 90th percentiles of the distributions for the percentage improvement. We find that all of the $K$-stop and ($K$ from $L$)-stop heuristics that we consider are able to improve significantly over the DVO heuristic, with the mean improvements ranging from about 8% to 10%. Clearly, the ability to interrupt switching and processing times offers major advantages, as the decision-maker is able to respond more rapidly to the arrivals of new jobs. We also observe again that the performance of the $K$-stop heuristic tends to improve as $K$ increases, although for $K \geq 3$ the marginal extra improvements become quite small. While larger values of $K$ enable more long-sighted choices of actions, it is not necessarily clear that these should result in dramatic improvements over smaller $K$ values, as the server is always able to change its direction at each time step and is never committed to following a sequence through to its end. Indeed, the larger $K$ is, the less likely it becomes that the server follows a sequence through to completion. Nevertheless, the results do seem to indicate a trend for larger $K$ values to yield stronger policies. It is also encouraging to note that both versions of the (2 from 4)-stop heuristic yield performances very close to that of the 2-stop heuristic. Recall that, in general, the ($K$ from $L$)-stop heuristic is supposed to be a more computationally scalable version of the $K$-stop heuristic. We accept a small loss of performance in exchange for greater scalability. The results indicate that the simpler impartial version of the (2 from 4)-stop heuristic tends to perform slightly better than the stratified version, indicating

that there is no obvious benefit in forcing the server to consider moving to a different cluster in the network.

Table 2: Percentage improvements of heuristic policies with respect to the DVO policy in 11,250 problem instances.

| Heuristic | Mean | 10th pct. | 25th pct. | 50th pct. | 75th pct. | 90th pct. |
|---|---|---|---|---|---|---|
| 1-stop | $8.57 \pm 0.18$ | -1.54 | 3.89 | 8.94 | 14.28 | 19.22 |
| 2-stop | $9.66 \pm 0.17$ | 0.23 | 4.98 | 9.78 | 14.99 | 19.97 |
| (2 from 4)-stop [imp.] | $9.64 \pm 0.17$ | 0.11 | 4.93 | 9.77 | 14.97 | 19.94 |
| (2 from 4)-stop [str.] | $9.40 \pm 0.18$ | -0.15 | 4.70 | 9.60 | 14.91 | 19.84 |
| 3-stop | $10.09 \pm 0.17$ | 0.76 | 5.38 | 10.21 | 15.51 | 20.13 |
| 4-stop | $10.30 \pm 0.16$ | 1.28 | 5.70 | 10.43 | 15.53 | 20.20 |

Next, we investigate the effects of the system parameters on these results by categorizing the $11,250$ problem instances according to the values of specific parameters. Three parameters of particular interest to us are the number of intermediate stages $n$, the traffic intensity $\rho = \sum_i \lambda_i/\mu_i$ and the relative switching rate, which we define as $\eta := \tau/(\sum_i \lambda_i)$ (so that it indicates the relative speed of switching compared to the frequency of new job arrivals). Table 3 summarizes the relative performances of the heuristics for each $n \in \{1, 2, 3, 4, 5, 6\}$. Note that the first row of the table shows the improvements of the 1-stop policy over the DVO heuristic, and the remaining rows show the additional improvements of the other heuristics over the 1-stop heuristic. We have chosen to present the results in this way in order to neatly summarize the benefits of allowing switching and processing times to be interrupted (shown in the first row of the table) and also the additional benefits of allowing the heuristic policies to make longer-sighted decisions (shown in the remaining rows). Tables 4 and 5 show similar results for $\rho$, and Table 6 shows similar results for $\eta$. Note that our method for randomly generating the system parameter values (detailed in Appendix F) implies that, in any problem instance, $\rho$ is equally likely to fall within any of the 9 intervals shown as columns in Tables 4 and 5, and similarly $\eta$ is equally like to be fall within any of the 6 intervals shown as columns in Table 6.

Table 3: Percentage improvements of the 1-stop policy (vs. the DVO heuristic) and the 2-stop, (2 from 4)-stop, 3-stop and 4-stop policies (vs. the 1-stop policy) for different values of $n$.

| Heuristic | $n = 1$ [1902 instances] | $n = 2$ [1854] | $n = 3$ [1855] | $n = 4$ [1917] | $n = 5$ [1870] | $n = 6$ [1852] |
|---|---|---|---|---|---|---|
| 1-stop (vs. DVO) | $9.49 \pm 0.51$ | $9.80 \pm 0.41$ | $9.32 \pm 0.41$ | $8.32 \pm 0.42$ | $7.49 \pm 0.45$ | $6.99 \pm 0.47$ |
| 2-stop (vs. 1-stop) | $1.67 \pm 0.16$ | $0.99 \pm 0.13$ | $0.79 \pm 0.18$ | $0.79 \pm 0.17$ | $0.99 \pm 0.18$ | $1.15 \pm 0.21$ |
| (2 from 4)-stop [imp.] (vs. 1-stop) | $1.53 \pm 0.16$ | $0.94 \pm 0.14$ | $0.91 \pm 0.16$ | $0.85 \pm 0.17$ | $1.00 \pm 0.17$ | $1.04 \pm 0.20$ |
| (2 from 4)-stop [str.] (vs. 1-stop) | $1.42 \pm 0.15$ | $0.81 \pm 0.13$ | $0.66 \pm 0.16$ | $0.54 \pm 0.20$ | $0.60 \pm 0.22$ | $0.73 \pm 0.23$ |
| 3-stop (vs. 1-stop) | $2.26 \pm 0.18$ | $1.40 \pm 0.14$ | $1.34 \pm 0.16$ | $1.28 \pm 0.17$ | $1.27 \pm 0.20$ | $1.49 \pm 0.22$ |
| 4-stop (vs. 1-stop) | $2.39 \pm 0.20$ | $1.58 \pm 0.14$ | $1.51 \pm 0.17$ | $1.38 \pm 0.20$ | $1.61 \pm 0.20$ | $1.84 \pm 0.22$ |

The first row of Table 3 appears to suggest a trend for the improvements given by the 1-stop policy (compared to the DVO heuristic) to decrease as $n$ increases. To make sense of this, it is useful to bear in mind that under any policy and any network design, the simulated proportion of time that the server spends processing jobs should be approximately equal to $\rho$, assuming that the system is stable. Thus, regardless of how small or large $n$ is, the server should spend approximately the same proportion of time visiting the intermediate stages of the network. In the $n = 1$ case, the server derives the maximum possible advantage from being able

to switch direction, as it can switch from the intermediate stage to any of the demand points in a single transition. Under the DVO heuristic, on the other hand, the server has no ability to change direction. It is also worthwhile to note that the 'impartial' version of the (2 from 4)-stop heuristic appears to outperform the 'stratified' version across all values of $n$. Intuition might suggest that the stratified version should become stronger as $n$ increases, as the clusters in the network become more distinct in this situation and the stratified version is designed to ensure that demand points in both clusters are always considered as potential destinations for switching. However, this intuition is not borne out by the results. It appears that allowing the demand points in the set $\mathcal{L}$ to be selected solely according to the indices given by the 1-stop policy (regardless of their locations in the network) is consistently the most effective approach.

Table 4: Percentage improvements of the 1-stop policy (vs. the DVO heuristic) and the 2-stop, (2 from 4)-stop, 3-stop and 4-stop policies (vs. the 1-stop policy) for $\rho \in [0.1, 0.5)$.

| Heuristic | $0.1 \leq \rho < 0.2$ [1257 instances] | $0.2 \leq \rho < 0.3$ [1191] | $0.3 \leq \rho < 0.4$ [1219] | $0.4 \leq \rho < 0.5$ [1295] |
|---|---|---|---|---|
| 1-stop (vs. DVO) | $18.23 \pm 0.34$ | $14.97 \pm 0.33$ | $12.43 \pm 0.34$ | $10.16 \pm 0.35$ |
| 2-stop (vs. 1-stop) | $0.50 \pm 0.12$ | $0.60 \pm 0.13$ | $0.74 \pm 0.16$ | $0.84 \pm 0.16$ |
| (2 from 4)-stop [imp.] (vs. 1-stop) | $0.51 \pm 0.12$ | $0.62 \pm 0.13$ | $0.78 \pm 0.15$ | $0.87 \pm 0.15$ |
| (2 from 4)-stop [str.] (vs. 1-stop) | $0.48 \pm 0.12$ | $0.57 \pm 0.12$ | $0.68 \pm 0.15$ | $0.73 \pm 0.15$ |
| 3-stop (vs. 1-stop) | $0.81 \pm 0.11$ | $0.94 \pm 0.14$ | $1.22 \pm 0.17$ | $1.30 \pm 0.16$ |
| 4-stop (vs. 1-stop) | $0.96 \pm 0.12$ | $1.15 \pm 0.15$ | $1.40 \pm 0.18$ | $1.50 \pm 0.18$ |

Table 5: Percentage improvements of the 1-stop policy (vs. the DVO heuristic) and the 2-stop, (2 from 4)-stop, 3-stop and 4-stop policies (vs. the 1-stop policy) for $\rho \in [0.5, 1)$.

| Heuristic | $0.5 \leq \rho < 0.6$ [1247 instances] | $0.6 \leq \rho < 0.7$ [1251] | $0.7 \leq \rho < 0.8$ [1260] | $0.8 \leq \rho < 0.9$ [1280] | $0.9 \leq \rho < 1.0$ [1250] |
|---|---|---|---|---|---|
| 1-stop (vs. DVO) | $8.21 \pm 0.32$ | $5.80 \pm 0.44$ | $4.35 \pm 0.35$ | $2.20 \pm 0.38$ | $1.25 \pm 0.94$ |
| 2-stop (vs. 1-stop) | $1.28 \pm 0.18$ | $1.47 \pm 0.20$ | $1.42 \pm 0.18$ | $1.37 \pm 0.20$ | $1.32 \pm 0.43$ |
| (2 from 4)-stop [imp.] (vs. 1-stop) | $1.27 \pm 0.17$ | $1.40 \pm 0.19$ | $1.37 \pm 0.17$ | $1.28 \pm 0.20$ | $1.27 \pm 0.41$ |
| (2 from 4)-stop [str.] (vs. 1-stop) | $1.11 \pm 0.17$ | $1.22 \pm 0.19$ | $1.00 \pm 0.21$ | $0.80 \pm 0.26$ | $0.56 \pm 0.46$ |
| 3-stop (vs. 1-stop) | $1.70 \pm 0.20$ | $1.98 \pm 0.22$ | $1.91 \pm 0.20$ | $1.84 \pm 0.22$ | $1.83 \pm 0.42$ |
| 4-stop (vs. 1-stop) | $1.94 \pm 0.20$ | $2.24 \pm 0.23$ | $2.17 \pm 0.20$ | $2.12 \pm 0.22$ | $1.97 \pm 0.44$ |

Next, we discuss the effect of the traffic intensity, $\rho$. By examining the first rows in Tables 4 and 5, we observe that the improvements given by the 1-stop policy (vs. the DVO heuristic) are much greater when $\rho$ is small than when it is large. Indeed, if $\rho$ is small, then the server spends a relatively large proportion of its time traversing the intermediate stages of the network. In such situations, the ability to react immediately to the arrival of a new job (by changing the direction of travel) clearly offers major advantages. On the other hand, when $\rho$ is large, queue lengths tend to become longer at the demand points and the server spends more of its time processing jobs at the demand points. It is useful to bear in mind that the DVO heuristic retains the ability to make new decisions every time the server finishes processing a job (which happens often when $\rho$ is large), whereas it does not have the ability to make new decisions while the server is switching, so in this sense larger $\rho$ values work in its favor. Therefore, it is more difficult for the 1-stop policy to gain an advantage when $\rho$ is large. The improvements of the $K$-stop heuristics over the DVO heuristic are more clearly evident when $\rho$ is of small or moderate size, since the novel features of our problem (in particular, the network formulation and the ability to interrupt switches) become more prominent under such circumstances. The

remaining rows in Tables 4 and 5 show that, in most cases, the improvements of the $K$-stop and ($K$ from $L$)-stop policies (for $K \geq 2$) over the 1-stop policy tend to be greatest when $\rho$ is between 0.6 and 0.7. This suggests that in order for the longer-sighted heuristics to show a clear improvement over the 1-stop heuristic, $\rho$ should be reasonably large (in order to ensure that there is enough congestion in the system to penalize short-sighted policies that always move towards the most congested demand point), but not so large that the server spends only a small proportion of its time switching between nodes.

Table 6: Percentage improvements of the 1-stop policy (vs. the DVO heuristic) and the 2-stop, (2 from 4)-stop, 3-stop and 4-stop policies (vs. the 1-stop policy) for different values of $\eta$.

| Heuristic | $0.1 \leq \eta < 0.4$ [1806 instances] | $0.4 \leq \eta < 0.7$ [1884] | $0.7 \leq \eta < 1$ [1818] | $1 \leq \eta < 4$ [1963] | $4 \leq \eta < 7$ [1895] | $7 \leq \eta < 10$ [1884] |
|---|---|---|---|---|---|---|
| 1-stop (vs. DVO) | $6.81 \pm 0.47$ | $8.35 \pm 0.43$ | $8.88 \pm 0.41$ | $10.31 \pm 0.43$ | $8.61 \pm 0.47$ | $8.31 \pm 0.48$ |
| 2-stop (vs. 1-stop) | $1.64 \pm 0.20$ | $1.47 \pm 0.18$ | $1.33 \pm 0.19$ | $1.03 \pm 0.15$ | $0.62 \pm 0.16$ | $0.33 \pm 0.17$ |
| (2 from 4)-stop [imp.] (vs. 1-stop) | $1.53 \pm 0.19$ | $1.43 \pm 0.17$ | $1.40 \pm 0.17$ | $1.01 \pm 0.16$ | $0.58 \pm 0.15$ | $0.36 \pm 0.16$ |
| (2 from 4)-stop [str.] (vs. 1-stop) | $1.20 \pm 0.23$ | $1.09 \pm 0.19$ | $1.04 \pm 0.19$ | $0.79 \pm 0.18$ | $0.40 \pm 0.17$ | $0.30 \pm 0.17$ |
| 3-stop (vs. 1-stop) | $2.42 \pm 0.21$ | $2.06 \pm 0.19$ | $1.96 \pm 0.18$ | $1.33 \pm 0.17$ | $0.81 \pm 0.17$ | $0.55 \pm 0.16$ |
| 4-stop (vs. 1-stop) | $2.87 \pm 0.22$ | $2.40 \pm 0.19$ | $2.25 \pm 0.18$ | $1.48 \pm 0.15$ | $0.85 \pm 0.18$ | $0.55 \pm 0.18$ |

Finally, we discuss the effect of the switching rate parameter, $\eta$. From Table 6 we observe that the mean percentage improvement of the 1-stop policy (vs. the DVO heuristic) tends to decrease as $\eta$ becomes small, but also as $\eta$ becomes large. Indeed, when $\eta$ is small, switching between nodes is relatively slow and the 1-stop policy is more likely to be deterred from changing direction, as this will result in too much wasted time traversing the intermediate stages of the network. On the other hand, when $\eta$ is large, switching between nodes is relatively fast and in this situation it becomes less likely that any arrivals occur while the server is traversing the intermediate stages, so the server has no reason to change its course. In general, the improvement over the DVO policy should be greater when direction changes are frequent. From the remaining rows in Table 6, we observe that the $K$-stop and ($K$ from $L$)-stop heuristics (for $K \geq 2$) are able to achieve greater improvements over the 1-stop policy when $\eta$ is small. Indeed, a primary motivation for using the longer-sighted heuristics is that they can recognize the effects of distances between the demand points and plan a sequence of visits accordingly. When $\eta$ is large, distances become less important as the server is always able to switch quickly between any two demand points and react quickly to new job arrivals, so it becomes more difficult for the longer-sighted heuristics to achieve improvements over the 1-stop policy.

## 4.2 Randomly-generated network layouts

The results that we reported in Section 4.1 were based on a specific network layout, depicted in Figure 4. In order to investigate how well our heuristics perform on other network topologies, we have also carried out $3,375$ additional experiments based on randomly-generated network layouts. In these experiments we used the same method for generating the values of arrival rates, processing rates and switching rates as in Section 4.1 (described in Appendix F) but changed the method for generating the network layout. To generate a random network layout, we begin by creating a $5 \times 5$ integer lattice in which 25 nodes are connected via horizontal and vertical edges. We then randomly select $d$ of these nodes, where $2 \leq d \leq 8$, and designate these as

demand points. Hence, the position of any demand point $i \in D$ can be represented by a pair of coordinates $(a_i, b_i)$ with $a_i, b_i \in \{1, 2, 3, 4, 5\}$, and the shortest path between two demand points $i, j \in D$ is the Manhattan distance $|a_i - a_j| + |b_i - b_j|$. The remaining nodes in the lattice are designated as intermediate stages, although some of these stages may be redundant (in the sense that they will never be visited under a sensible policy), and therefore can be eliminated from the network. Figure 5 shows 4 of the random network layouts generated in our experiments.

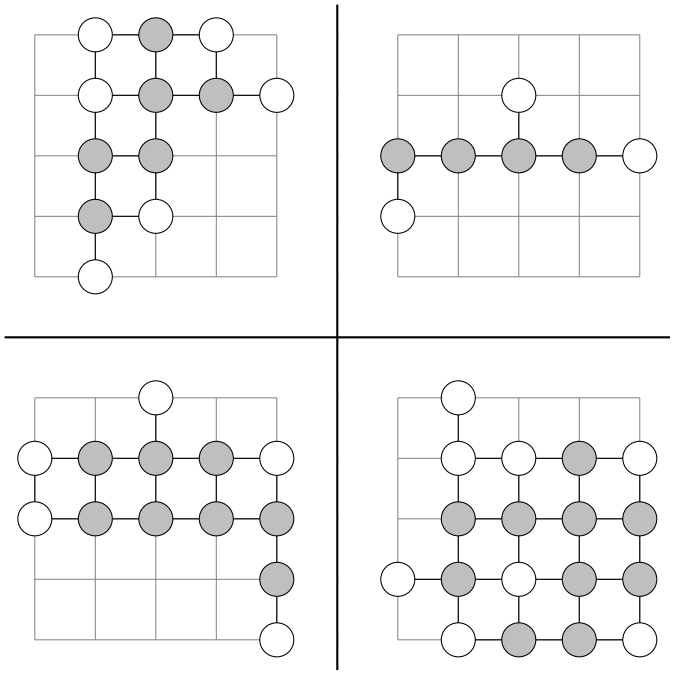

Figure 5: 4 randomly-generated network layouts with demand points shown in white and intermediate stages shown in gray, after removal of redundant intermediate stages.

As in Section 4.1, we have used relative value iteration to compute exact suboptimalities of our heuristics in 'small' instances where this is feasible. In total, 635 of the 3,375 instances satisfied the feasibility criteria described in Appendix H, and these suboptimality percentages are shown in Table 7. By comparing this table with Table 1 from earlier, we may observe that all of the heuristics have slightly higher mean suboptimality percentages than in the previously-considered network design. From a qualitative point of view, however, the results are similar, with the $K$-stop heuristic tending to perform better as $K$ increases. It is also interesting to note that the $50^{\text{th}}$ percentiles for the heuristics are similar to those in Table 1, and even lower in some cases. This suggests that the increases in the mean suboptimalities are due to worse tail performances; in other words, there may be some randomly-generated networks on which our heuristics perform particularly poorly. One possible explanation for this is that, in the randomly-generated networks, there may sometimes be multiple shortest paths between pairs of demand points, and currently our heuristics select a shortest path based on an arbitrary priority ordering of the nodes, without taking into account of which path(s) might leave the server in a better position if a switch between demand points is interrupted. Selection of the 'best' shortest path (which might depend on the current system state) would significantly complicate our heuristics, and could be a direction for future work.

Table 8 shows the percentage improvements achieved by the $K$-stop and ($K$ from $L$)-stop

Table 7: Percentage suboptimalities of heuristic policies in 635 'small' problem instances on randomly-generated networks.

| Heuristic | Mean | 10th pct. | 25th pct. | 50th pct. | 75th pct. | 90th pct. |
|---|---|---|---|---|---|---|
| DVO | $26.52 \pm 1.95$ | 5.82 | 10.62 | 21.07 | 33.83 | 50.87 |
| 1-stop | $7.45 \pm 1.47$ | 0.81 | 1.76 | 3.45 | 7.35 | 15.01 |
| 2-stop | $4.72 \pm 0.53$ | 0.62 | 1.43 | 2.85 | 5.00 | 9.48 |
| 3-stop | $4.52 \pm 0.51$ | 0.53 | 1.36 | 2.75 | 4.95 | 9.18 |

heuristics against the DVO policy in all of the 3,375 instances with randomly-generated networks. The table can be interpreted in the same way as Table 2 from Section 4.1, except that we have excluded the 'stratified' version of the (2 from 4)-stop heuristic, because it was already shown to perform worse than the impartial version of the heuristic in the previous results, and additionally the division of demand points into 'clusters' does not work in such an obvious way in the random networks. By comparing Table 8 with Table 2 from earlier, we may observe that the improvements of our heuristics compared to the DVO policy are slightly greater than in the previously-considered network design. As discussed earlier, the main weaknesses of the DVO policy are that it works in a short-sighted way and does not allow interruptions of setup or processing times. Thus, the results suggest that the benefits of interruptions are greater in the randomly-generated networks. Indeed, it can be seen from the examples in Figure 5 that, while moving from one demand point to another, the server might pass near (or through) another demand point, in which case it might be advantageous to switch to that point.

Table 8: Percentage improvements of heuristic policies with respect to the DVO policy in 3,375 problem instances on randomly-generated networks.

| Heuristic | Mean | 10th pct. | 25th pct. | 50th pct. | 75th pct. | 90th pct. |
|---|---|---|---|---|---|---|
| 1-stop | $11.62 \pm 0.39$ | 1.66 | 6.72 | 12.05 | 17.61 | 22.75 |
| 2-stop | $12.89 \pm 0.39$ | 2.82 | 7.86 | 13.48 | 19.14 | 23.98 |
| (2 from 4)-stop [imp.] | $12.89 \pm 0.40$ | 2.76 | 7.74 | 13.42 | 19.14 | 24.04 |
| 3-stop | $13.25 \pm 0.40$ | 3.17 | 8.20 | 13.84 | 19.40 | 24.37 |
| 4-stop | $13.35 \pm 0.40$ | 3.22 | 8.29 | 14.05 | 19.47 | 24.45 |

We have also investigated the effects of individual parameters (such as $\rho$ and $\eta$) on the results for the randomly-generated networks and found that, from a qualitative point of view, the effects are similar to those discussed in Section 4.1, so we have omitted these for brevity.

### 4.3 Computational requirements of our heuristics

An advantage of using index-based heuristics, as proposed in our paper, is that they have much lighter computational requirements than (for example) a reinforcement learning algorithm, or any method that relies on extensive training in order to learn a strong decision-making policy. However, as discussed previously, the computational requirements of the $K$-stop heuristic increase rapidly as $K$ increases (or as the number of demand points increases), due to the growth in the size of $\mathcal{S}$, the set of candidate sequences. This provides motivation for using the ($K$ from $L$)-stop heuristics that we propose in Section 3.3.

In Table 9 we present a summary of the average computation times (in seconds) needed by the various heuristics to select an action at a single time step. Our experiments were performed

on 108 problem instances with randomly-generated networks and 8 demand points ($d = 8$). The software used was Python 3.7.13, with the PyPy just-in-time compiler (`http://pypy.org`) used to speed up computations, and all experiments were carried out on an Apple M1 Pro (8-core CPU) with 16 GB unified memory, running macOS Sequoia. The times shown in the table are average times (in seconds) needed to carry out all decision-making steps at a single discrete time step. For example, in the case of the $K$-stop heuristic, this includes the time needed to construct the set of sequences $\mathcal{S}$ and then select an action according to the steps described in the algorithm as presented in Section 3.2. The running times are averaged over all time steps within each instance and then averaged over all instances. The results show that all of the heuristics are able to make decisions within 0.001 seconds (and much less in some cases), which implies that they are suitable for use in fast-changing systems with hundreds or even thousands of state changes per second. As expected, the increase in running time as $K$ increases is significant, with the average running time tending to increase by a factor of between 4 and 10 each time $K$ increases by one. Notably, the (2 from 4)-stop heuristic has a very short running time and is even faster than the 2-stop heuristic because, once the subset of 4 demand points has been selected (which requires only a short amount of time), the number of sequences of length 2 to be evaluated is much smaller than in the 2-stop heuristic.

Table 9: Average running times (in seconds per time step) for action selection by the DVO policy, 1-stop, 2-stop, (2 from 4)-stop [imp.], 3-stop, and 4-stop heuristics, over 108 problem instances with 8 demand points on randomly-generated networks.

| Heuristic | DVO | 1-stop | 2-stop | (2 from 4)-stop [imp.] | 3-stop | 4-stop |
|---|---|---|---|---|---|---|
| Av.run.times | $7.19 \times 10^{-5}$ | $1.72 \times 10^{-6}$ | $7.21 \times 10^{-6}$ | $3.17 \times 10^{-6}$ | $6.36 \times 10^{-5}$ | $4.61 \times 10^{-4}$ |

# 5 Conclusions

The main novelty of the job scheduling problem studied in this paper lies in its network-based formulation, which allows setup and processing times to be interruptible and also enables the modeling of complex dependence structures between the setup requirements of different tasks. We consider a highly stochastic, infinite-horizon problem in which jobs of different types arrive at random points in time and the server's setup and processing times are also random. The dynamic nature of our problem implies that decision epochs occur very frequently, and this creates some challenges. For example, as discussed in Section 2, we are unable to directly leverage results from the literature on polling systems in order to prove the existence of a deterministic, stationary policy under which the system is stable, because polling-type policies are nonstationary under our MDP formulation. However, the stability result can be established (provided that $\rho < 1$) by proving the equivalence of a similar MDP in which the system state includes extra information.

The index policies that we develop in Section 3 are influenced by the heuristic approaches used in Duenyas and Van Oyen (1996), but these approaches must be adapted in order to exploit the novel features of our problem. In particular, our network-based formulation motivates the use of a long-sighted, sequence-based algorithm that takes the topology of the network into

account when making decisions. In addition, we ensure that the algorithm makes new decisions at each time step, so that setup and processing times can always be interrupted. We also introduce derivative-based conditions in the steps of the $K$-stop algorithm and use these to show that the resulting policies possess the property of 'pathwise consistency', which ensures that the server always proceeds to a demand point in finite time. Furthermore, we propose a modified version of the algorithm (known as the ($K$ from $L$)-stop algorithm) that scales much more readily to systems with many demand points. In special cases of the problem we can prove system stability and optimality of our heuristics (Theorem 3.5), but in more general cases we must study their performance empirically.

The numerical results in Section 4 demonstrate the advantages of using heuristics that are well-tailored to the novel features of our problem. In small problem instances, we are able to show that the $K$-stop and ($K$ from $L$)-stop heuristics are much closer to optimality than the unmodified DVO heuristic. In larger systems, we are also able to observe the benefits of allowing the server to make long-sighted decisions and to change its course of action when necessary. From a practical perspective, it is encouraging to see that the impartial version of the (2 from 4)-stop heuristic performs almost as well as the more computationally intensive 2-stop heuristic, which in turn offers considerable improvements over the more myopic 1-stop heuristic. Thus, in larger problem instances, the (2 from 4)-stop heuristic may be seen as a strong candidate to achieve significant cost savings over simpler alternatives, without incurring excessive computational costs.

**Acknowledgements**

We are grateful for the constructive feedback and suggestions provided by the anonymous reviewers, which have helped to improve this paper. The first author was funded by a PhD studentship from the Engineering and Physical Sciences Research Council (EPSRC), under grant EP/W523811/1.

## References

Allahverdi, A., Gupta, J., and Aldowaisan, T. (1999). A review of scheduling research involving setup considerations. *Omega*, 27(2):219–239.

Altman, E., Konstantopoulos, P., and Liu, Z. (1992). Stability, monotonicity and invariant quantities in general polling systems. *Queueing Systems*, 11:35–57.

Asmussen, S. (2003). *Applied Probability and Queues*. Springer.

Atar, R., Giat, C., and Shimkin, N. (2010). The c$\mu/\theta$ rule for many-server queues with abandonment. *Operations Research*, 58(5):1427–1439.

Baras, J. S., Dorsey, A., and Makowski, A. M. (1985). Two competing queues with linear costs and geometric service requirements: The $\mu$c-rule is often optimal. *Advances in Applied Probability*, 17(1):186–209.

Bell, C. (1971). Characterization and computation of optimal policies for operating an M/G/1 queuing system with removable server. *Operations Research*, 19(1):208–218.

Bertsekas, D. (2019). *Reinforcement Learning and Optimal Control*. Athena Scientific.

Blazewicz, J., Ecker, K., Pesch, E., Schmidt, G., Sterna, M., and Weglarz, J. (2019). *Handbook on Scheduling: From theory to practice (2nd ed)*. Springer.

Boxma, O. J. and Groenendijk, W. P. (1987). Pseudo-conservation laws in cyclic-service systems. *Journal of Applied Probability*, 24(4):949–964.

Browne, S. and Yechiali, U. (1989). Dynamic priority rules for cyclic-type queues. *Advances in Applied Probability*, 21(2):432–450.

Buyukkoc, C., Varaiya, P., and Walrand, J. (1985). The c$\mu$ rule revisited. *Advances in Applied Probability*, 17(1):237–238.

Cohen, A. and Saha, S. (2022). Asymptotic optimality of the generalized $c\mu$ rule under model uncertainty. *Stochastic Processes and their Applications*, 136:206–236.

Duenyas, I. and Van Oyen, M. P. (1996). Heuristic scheduling of parallel heterogeneous queues with set-ups. *Management Science*, 42(6):814–829.

Dulac-Arnold, G., Mankowitz, D., and Hester, T. (2019). Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*.

Elsayed, E. K., Elsayed, A. K., and Eldahshan, K. A. (2022). Deep reinforcement learning-based job shop scheduling of smart manufacturing. *Computers, Materials & Continua*, 73(3).

Fan, J. (2012). Supply chain scheduling with transportation cost on a single machine. *Advanced Materials Research*, 382:106–109.

Gholami, M., Zandieh, M., and Alem-Tabriz, A. (2009). Scheduling hybrid flow shop with sequence-dependent setup times and machines with random breakdowns. *The International Journal of Advanced Manufacturing Technology*, 42:189–201.

Gocic, M. and Trajkovic, S. (2013). Analysis of changes in meteorological variables using Mann-Kendall and Sen's slope estimator statistical tests in Serbia. *Global and Planetary Change*, 100:172–182.

Hamed, K. H. and Rao, A. R. (1998). A modified Mann-Kendall trend test for autocorrelated data. *Journal of Hydrology*, 204(1-4):182–196.

Harrison, J. (1975). A priority queue with discounted linear costs. *Operations Research*, 23(2):260–269.

Hunt, K. and Zhuang, J. (2024). A review of attacker-defender games: Current state and paths forward. *European Journal of Operational Research*, 313(2):401–417.

Kayhan, B. M. and Yildiz, G. (2023). Reinforcement learning applications to machine scheduling problems: a comprehensive literature review. *Journal of Intelligent Manufacturing*, 34:905–929.

Lee, D. and Vojnovic, M. (2021). Scheduling jobs with stochastic holding costs. *Advances in Neural Information Processing Systems*, 34:19375–19384.

Lei, K., Guo, P., Wang, Y., Zhang, J., Meng, X., and Qian, L. (2023). Large-Scale Dynamic Scheduling for Flexible Job-Shop With Random Arrivals of New Jobs by Hierarchical Reinforcement Learning. *IEEE Transactions on Industrial Informatics*, 20(1):1007–1018.

Leung, J. Y. (2004). *Handbook of Scheduling: Algorithms, Models and Performance Analysis*. CRC Press.

Li, F., Lang, S., Hong, B., and Reggelin, T. (2024). A two-stage RNN-based deep reinforcement learning approach for solving the parallel machine scheduling problem with due dates and family setups. *Journal of Intelligent Manufacturing*, 35:1107–1140.

Li, Y., Fadda, E., Manerba, D., Tadei, R., and Terzo, O. (2020). Reinforcement learning algorithms for online single-machine scheduling. In *15th Conference on Computer Science and Information Systems (FedCSIS)*, pages 277–283. IEEE.

Lippman, S. A. (1975). Applying a new device in the optimization of exponential queuing systems. *Operations Research*, 23(4):687–710.

Liu, Z., Nain, P., and Towsley, D. (1992). On optimal polling policies. *Queueing Systems*, 11:59–83.

Luo, S. (2020). Dynamic scheduling for flexible job shop with new job insertions by deep reinforcement learning. *Applied Soft Computing*, 91:106208.

Mandelbaum, A. and Stolyar, A. L. (2004). Scheduling flexible servers with convex delay costs: Heavy-traffic optimality of the generalized c$\mu$-rule. *Operations Research*, 52(6):836–855.

Ozkan, E. (2022). On the asymptotic optimality of the $c\mu$ rule in queueing networks. *Operations Research Letters*, 50(3):254–259.

Pinedo, M. L. (2016). *Scheduling: Theory, Algorithms and Systems (5th ed)*. Springer.

Puterman, M. (1994). *Markov Decision Processes - Discrete Stochastic Dynamic Programming*. Wiley & Sons, New York.

Saghafian, S. and Veatch, M. (2015). A c-mu rule for two-tiered parallel servers. 61(4):1046–1050.

Sennott, L. I. (1997). The computation of average optimal policies in denumerable state Markov decision chains. *Advances in Applied Probability*, 29(1):114–137.

Sennott, L. I. (1999). *Stochastic dynamic programming and the control of queueing systems*. John Wiley & Sons.

Serfozo, R. F. (1979). An equivalence between continuous and discrete time Markov decision processes. *Operations Research*, 27(3):616–620.

Shaw, S. B. and Singh, A. (2014). A survey on scheduling and load balancing techniques in cloud computing environment. In *5th International Conference on Computer and Communication Technology (ICCCT)*, pages 87–95. IEEE.

Smith, W. E. et al. (1956). Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66.

Ulmer, M., Goodson, J., Mattfeld, J., and Thomas, B. (2020). On modeling stochastic dynamic vehicle routing problems. *EURO Journal on Transportation and Logistics*, 9(2):100008.

Van Mieghem, J. (1995). Dynamic scheduling with convex delay costs: The generalized c-mu rule. *The Annals of Applied Probability*, 5(3):809–833.

Wang, L., Pan, Z., and Wang, J. (2021). A review of reinforcement learning based intelligent optimization for manufacturing scheduling. *Complex System Modeling and Simulation*, 1(4):257–270.

Xu, D., Li, G., and Zhang, F. (2021). Scheduling an automatic IoT manufacturing system with multiple servers. *Computers & Industrial Engineering*, 157:107343.

Yang, W., Chen, L., and Dauzère-Pèrés, S. (2022). A dynamic optimisation approach for a single machine scheduling problem with machine conditions and maintenance decisions. *International Journal of Production Research*, 60(10):3047–3062.

Yechiali, U. (1993). Analysis and control of polling systems. In *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation*, pages 630–650. Springer.

Zhang, T., Xie, S., and Rose, O. (2017). Real-time job shop scheduling based on simulation and Markov decision processes. In *2017 Winter Simulation Conference (WSC)*, pages 3899–3907.

Zhang, Y., Du, P., Wang, J., Ba, T., Ding, R., and Xin, N. (2019). Resource scheduling for delay minimization in multi-server cellular edge computing systems. *IEEE Access*, 7:86265–86273.

Zhao, L., Fan, J., Zhang, C., Shen, W., and Zhang, J. (2023). A DRL-Based Reactive Scheduling Policy for Flexible Job Shops With Random Job Arrivals. *IEEE Transactions on Automation Science and Engineering*, 21(3):2912–2923.