Lancaster University

Dynamic Allocation of Mobile Servers in a Network

Dongnuan Tian, BSc, MSc

A thesis submitted in fulfilment of the requirements for the degree of Doctor of Philosophy

in the

Lancaster University Management School
Department of Management Science

May 2026

Abstract

Many operations research problems focus on optimizing network flows and routing in deterministic settings, yet real-world systems are inherently stochastic and dynamic. Allocating resources efficiently in such environments requires decision-making that responds to both spatial and temporal variations in demand. This thesis studies such problems in stochastic networks, where customer demand arrives randomly over time. Demand points generate jobs according to independent Poisson processes, and homogeneous servers travel across the network to provide service, with exponentially distributed service and switching times. Service and switching times are interruptible, allowing servers to adjust tasks in response to new changes. The objective is to minimize long-run average holding costs by balancing immediate responsiveness with long-term efficiency.

The system is modeled as a Markov Decision Process, but the large state space renders exact optimization infeasible. To address this, the thesis develops scalable approximation methods that combine structural insights from index heuristics with computational approaches from approximate dynamic programming and reinforcement learning. The first part of the thesis focuses on single-server systems. Chapter 2 analyzes an infinite-state model and develops index-based policies with desirable structural properties. Chapter 3 considers a finite-state model and introduces reinforcement learning techniques to refine index heuristics through approximate policy improvement. The second part, introduced in Chapter 4, extends the analysis to multi-server systems, where additional challenges of coordination and workload balancing arise. In this setting, multiple servers may occupy the same node and provide service simultaneously. To address these, new heuristics are proposed, involving proportional assignment of demand points to servers to form server-specific local regions, allowing a modified version of the index-based heuristic from Chapter 2 to be applied. Numerical experiments across a variety of network configurations demonstrate that the proposed policies deliver strong performance while remaining computationally tractable.

Keywords: Dynamic Resource Allocation, Mobile Server Allocation, Stochastic Networks, Markov Decision Process, Index Heuristics, Reinforcement Learning

Acknowledgements

I would like to express my deepest gratitude to my supervisors, Dr. Robert Shone (Rob) and Prof. Kevin Glazebrook, for their invaluable guidance and support throughout my PhD journey. Although Kevin retired before the completion of my doctorate, he made a significant contribution by shaping the direction of my research through his constructive insights and encouragement. Rob has provided exceptional support in every aspect of my development as a researcher. His commitment was evident in our regular weekly meetings, his patient guidance in strengthening my understanding of mathematics, statistics, and operations research, and his encouragement in honing my research skills. He consistently offered timely and thoughtful feedback on my work, helped me refine my academic writing and communication across listening, speaking, reading, and writing, and gave me invaluable opportunities to practice presenting, including sustained mock viva preparations. Rob's care extended beyond academics to my physical and mental wellbeing, and his guidance has shaped not only my scholarly abilities but also my way of thinking, instilling in me a habit of critical reflection that will continue to benefit me throughout my life.

I would also like to extend my thanks to Lancaster University, Lancaster University Management School, and in particular the Department of Management Science, for providing a stimulating and supportive academic environment. I am grateful to the staff and colleagues, especially Ms. Gay Bentinck, the PhD Programme Coordinator, for her continuous assistance. She not only kept me informed of important events and milestones relevant to my doctoral progress but also provided invaluable practical help, such as liaising persistently with the university and the bank to resolve a missing scholarship payment, for which I remain deeply appreciative.

I am also grateful to Lancaster University and the Engineering and Physical Sciences Research Council (EPSRC) for awarding me a four-year full scholarship. This support ensured that my PhD journey was financially secure and allowed me to dedicate myself fully to my research without distraction.

My sincere thanks also go to my PhD colleagues and friends, Jing Lyu, Ke Fang, and Yuhan Jiang, for their companionship, encouragement, and constant care. Their support has been a vital source of strength for both my academic progress and my personal wellbeing.

Finally, I would like to thank my family, my close friends (Yokimi and Gabe) at home, and several professors at Tongji University for their continuous encouragement and guidance. Their regular contact, thoughtful advice, and genuine care have been an enduring source of strength, helping me to balance academic commitments with personal wellbeing. Their unwavering support has been a powerful driving force throughout my PhD journey, for which I am deeply grateful.

Statement of Authorship

This thesis by publication was developed in collaboration with my supervisor, Dr. Robert Shone, with my contributions and his support outlined below.

Chapter 1

Material organisation: I identified and organised the material to be included, while my supervisor advised on refining the structure and suggested additional elements to strengthen the introduction.

Content drafting and refinement: I drafted the content and iteratively refined it over multiple rounds of revision, incorporating wording, formatting, and structural refinements based on regular feedback from my supervisor.

Chapters 2, 3 and 4

Formulation of models: I investigated and reviewed the relevant literature, formulated the program models, and designed the experimental framework, with my supervisor providing critical feedback to improve the models.

Code development and experimentation: I developed the code, conducted the experiments, and compiled the results. My supervisor offered constructive suggestions on the technical parts and the analysis of the results.

Visualisation and presentation of results: I designed and created the figures and tables, which were refined based on feedback from my supervisor.

Investigation of theoretical results: My supervisor suggested directions for additional results to be investigated and I verified these through programming before trying to develop proofs.

Proof establishment of theoretical results: I made initial attempts at proving the theoretical results and refined them based on feedback from my supervisor, with some of the longer proofs further refined and completed by my supervisor, as presented in the appendices.

Manuscript drafting: I drafted the original manuscript, adapted the templates, and carried out multiple rounds of revision based on regular feedback from my supervisor. My supervisor further improved the standard of English in preparation for journal submission.

Chapter 5

Content drafting: I drafted the chapter, summarising the key findings, presenting the overall conclusions, and outlining directions for future work.

Content refinement: I continuously refined the chapter over multiple rounds of revision, incorporating regular feedback from my supervisor to improve clarity, accuracy of wording, structure, and presentation of results.

Dongnuan Tian

May 2026

Declaration

I hereby declare that this thesis entitled:

"Dynamic Allocation of Mobile Servers in a Network"

is the result of my own original work, carried out under the supervision of Dr. Robert Shone (primary) and Prof. Kevin Glazebrook at Lancaster University. It has not been submitted, in whole or in part, for any other degree at Lancaster University or elsewhere. All sources of information and assistance have been acknowledged, and where the work of others has been used, it has been duly cited and referenced.

I also declare the following:

This thesis is by publication. Chapters 2, 3 and 4 are independent research papers, each intended to be read as a standalone piece of work.

Chapter 2 of this thesis has been published in *European Journal of Operational Research*, with the title of "Stochastic dynamic job scheduling with interruptible setup and processing times: An approach based on queueing control", by Dongnuan Tian and Rob Shone (Tian and Shone (2026b)). The version that we provide in this thesis is the original version of the paper submitted to EJOR, subsequently updated to include revisions requested by the thesis examiners.

Chapter 3 of this thesis has been published in *Computers & Operations Research* with the title of "Dynamic repair and maintenance of heterogeneous machines dispersed on a network: A rollout method for online reinforcement learning", by Dongnuan Tian and Rob Shone (Tian and Shone (2026a)). The version that we provide in this thesis is the original version of the paper submitted to COR, subsequently updated to include revisions requested by the thesis examiners.

Chapter 4 of this thesis represents one paper that has not been submitted yet. We plan to submit it to a reputed journal in due course.

Dongnuan Tian

May 2026

Contents

Abstract i

Acknowledgements ii

Statement of Authorship iv

Declaration vi

List of Figures x

List of Tables xi

1 Introduction 1

1.1 Background and motivation 1
1.2 Problem description 3
1.3 Practical applications 5
1.3.1 Logistics and transportation 6
1.3.2 Cloud computing and telecommunications 8
1.3.3 Manufacturing and robotics 9
1.3.4 Maintenance and infrastructure inspection 11
1.3.5 Healthcare service delivery 13
1.4 Theoretical framework 15
1.4.1 Markov Decision Processes 15
1.4.2 Dynamic Programming 18
1.4.3 Approximate Dynamic Programming 19
1.4.4 Reinforcement Learning 22
1.4.5 Index heuristics 24
1.5 Research contributions 26
1.6 Thesis outline 28

2 Stochastic Dynamic Job Scheduling with Interruptible Setup and Processing Times **30**

2.1 Introduction 31
2.2 Problem formulation 36
2.3 Index heuristics 41
2.3.1 The DVO heuristic 42
2.3.2 K-stop heuristics 42
2.3.3 (K from L)-stop heuristic 50
2.4 Numerical results 53
2.4.1 General performance of the heuristics 55
2.4.2 Performance under specific parameters 56
2.4.3 Computational requirements of our heuristics 62
2.5 Conclusions 63

3 Dynamic Repair and Maintenance of Heterogeneous Machines Distributed on a Network: A Reinforcement Learning Approach 65
3.1 Introduction 66
3.2 Problem formulation 70
3.3 Index heuristic 75
3.4 Online policy improvement 84
3.4.1 Offline part 87
3.4.2 Online part 88
3.5 Numerical results 91
3.5.1 Polling system heuristic 93
3.5.2 Performances of heuristic policies in systems with $2 \leq m \leq 8$ . . . 94
3.5.3 Effects of varying the system parameters 96
3.6 Conclusions 99

4 Stochastic Dynamic Job Scheduling with Interruptible Setup and Processing Times for Multiple Servers 101
4.1 Introduction 102
4.2 Problem formulation 108
4.3 Index heuristics 113
4.3.1 Polling heuristic 120
4.3.2 Modified 1-stop heuristic 121
4.4 Numerical results 124
4.4.1 Flexible heuristic 126
4.4.2 Exclusive heuristic 127
4.4.3 Results of experiments 128
4.5 Conclusions 130

5 Conclusions and Future Work 132
5.1 Summary 132
5.2 Future work 135

A Appendices for Chapter 2 **153**
A.1 Proof of Theorem 2.1. 153
A.2 DVO Heuristic 158
A.3 Proof of Theorem 2.3. 160
A.4 Proof of Corollary 2.4. 165
A.5 Methods for generating the parameters for the numerical experiments in Section 2.4 167
A.6 Simulation methods for the numerical experiments in Section 2.4 168
A.7 Method for testing the feasibility of dynamic programming (DP) for the numerical experiments in Section 2.4 170

B Appendices for Chapter 3 **172**
B.1 Proof of Proposition 3.1. 172
B.2 Proof of Proposition 3.2. 174
B.3 Proof of Proposition 3.6 175
B.4 Proof of Proposition 3.8 180
B.5 Details of Approximate Policy Improvement 187
B.5.1 SampleTrajectory module 188
B.5.2 Offline part - preparatory phase 189
B.5.3 Offline part - main phase 190
B.5.4 Online part 190
B.5.5 Pseudocode of API 192
B.6 Methods for generating the parameters for the numerical experiments in Section 3.5 197
B.7 Simulation methods for the numerical experiments in Section 3.5 198

C Appendices for Chapter 4 **200**
C.1 Methods for generating the parameters for the numerical experiments in Section 4.4 200
C.2 Simulation methods for the numerical experiments in Section 4.4 202

List of Figures

1.1 A network with 14 nodes, 4 demand points, 10 intermediate stages, and 3 servers. White-colored nodes represent demand points at which new customer demand arrives, and gray-colored nodes represent intermediate stages. . . . 3
1.2 A structural overview of this thesis, outlining how the research is organised across chapters. . . . 29

2.1 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. . . . 33
2.2 A network with 2 demand points and 1 intermediate stage. . . . 40
2.3 A network with 4 demand points on the left, 4 demand points on the right and 4 intermediate stages. . . . 53
2.4 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. . . . 54

3.1 A network with 4 machines, 3 intermediate stages and a single repairer. Gray-colored nodes represent machines, and white-colored nodes represent intermediate stages. . . . 68
3.2 A star network with 6 machines and a radius $r = 3$. . . . 84
3.3 A network with machines at $(1,3)$, $(2,5)$ and $(3,1)$ and intermediate stages at $(2,1)$, $(2,2)$, $(2,3)$ and $(2,4)$. . . . 91

4.1 A network with 4 demand points, 10 intermediate stages and 3 servers. White-colored nodes represent demand points at which new jobs arrive, and gray-colored nodes represent intermediate stages. . . . 105
4.2 The number of waiting jobs at demand point i_k evolves with time. . . . 118
4.3 4 randomly-generated network layouts with demand points shown in white and intermediate stages shown in gray, after removal of redundant intermediate stages. . . . 125

B.1 A star network with 4 machines and a radius $r = 2$, where the repairer is at an intermediate stage, machine 1 has failed with $x_1 = 1$ and other machines are working with $x_2 = x_3 = x_4 = 0$. . . . 182
B.2 Division of time into periods $T_1, S_1, T_2, S_2, ...$ under the index heuristic. . . 186

List of Tables

2.1 The percentage suboptimalities of the DVO, 1-stop, 2-stop and 3-stop heuristics with respect to the optimal policy given by relative value iteration, computed across 1229 systems. 55
2.2 The percentage improvements of the 1-stop, 2-stop, (2 from 4)-stop, 3-stop and 4-stop heuristics with respect to the DVO heuristic, computed across 10,000 systems. 56
2.3 The mean percentage improvements (with 95% confidence intervals) 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) shown for different values of n over all 10,000 instances. 57
2.4 The mean 'direction change' percentages (with 95% confidence intervals) for each of the K-stop and (K from L)-stop heuristics shown for different values of n over all 10,000 instances. 57
2.5 The mean percentage improvements (with 95% confidence intervals) 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) shown for different values of η over all 10,000 instances. 59
2.6 The mean 'direction change' percentages (with 95% confidence intervals) for each of the K-stop and (K from L)-stop heuristics shown for different values of η over all 10,000 instances. 60
2.7 The mean percentage improvements (with 95% confidence intervals) 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) shown for different values of ρ over all 10,000 instances. 61
2.8 The mean 'direction change' percentages (with 95% confidence intervals) for each of the K-stop and (K from L)-stop heuristics shown for different values of ρ over all 10,000 instances. 61
2.9 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. 63

3.1 Decisions made under an optimal policy for Example 3.1 75
3.2 Percentage suboptimalities of heuristic policies in 412 'small' problem instances with $2 \leq m \leq 4$, evaluated under the cost-based formulation. . . 95

3.3 Percentage suboptimalities of heuristic policies in 412 'small' problem instances with $2 \leq m \leq 4$, evaluated under the reward-based formulation. 95
3.4 Percentage improvements of API over the index policy for different values of m, evaluated under the cost-based formulation. 96
3.5 Percentage improvements of API over the index policy for different values of m, evaluated under the reward-based formulation. 96
3.6 Percentage of 'safe' action choices under θ^{API} for different values of m. . . 96
3.7 Summary of numerical results categorized according to the value of $\rho = \sum_i \lambda_i/\mu_i$, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font) 97
3.8 Summary of numerical results categorized according to the value of $\eta = \tau/(\sum_{i \in M} \lambda_i)$, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font) 97
3.9 Summary of numerical results categorized according to the type of cost function $f_i(x_i)$ used, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font) . . 98
3.10 Summary of numerical results categorized according to the maximum degradation state K, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font) 98

4.1 The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for $\alpha \in \{0, 0.1, 0.2, 0.3, 0.4\}$, with 100 instances evaluated for each α across a total of 1000 instances. 128
4.2 The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for $\alpha \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$, with 100 instances evaluated for each α across a total of 1000 instances. 128
4.3 The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for different values of d over all 1000 instances. . . . 129
4.4 The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for different values of m over all 1000 instances. . . . 130
4.5 The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for different values of ρ over all 1000 instances. . . . 130

Chapter 1

Introduction

1.1 Background and motivation

Many operations research (OR) problems aim to optimize network flows and routing systems, with objectives such as minimizing costs and improving efficiency. These challenges can be found across diverse domains, including transportation, logistics, maintenance, telecommunications and healthcare. While traditional optimization models often assume deterministic environments with known parameters such as demand rates, service times and travel costs, real-world systems are subject to considerable uncertainty. Such uncertainty may be caused by random changes in demand, variability in service times, unexpected disruptions, or dynamic environmental factors like weather and congestion. To address these challenges we need to develop stochastic models and adaptive decision-making frameworks capable of responding to evolving conditions while maintaining operational efficiency.

A fundamental challenge in such systems is the allocation of mobile resources to meet demand that varies spatially and temporally. Effective resource management is essential in industries like logistics, healthcare, and infrastructure maintenance. For instance, in emergency response operations, ambulances must be dynamically dispatched based on real-time demand fluctuations to minimize response times while ensuring adequate coverage across a city. In logistics, inefficient routing can lead to increased travel distances, longer delivery times and higher operational costs. In infrastructure maintenance, such as utility repairs or equipment servicing, failure to schedule technicians dynamically based on real-time fault reports can result in extended downtimes, growing service backlogs, and underutilized crews. These examples illustrate the importance of

making timely and adaptive decisions in response to fluctuating demand and evolving network conditions.

Dynamic server allocation specifically addresses the coordination of mobile resources in uncertain, time-varying environments. This involves making routing and scheduling decisions that consider travel constraints, changing workloads, and shifting service priorities. Unlike static allocation, where assignments are determined once under fixed assumptions, dynamic allocation requires continual monitoring of system states and timely adjustments to resource movements. As the number of servers or service locations increases, the decision space grows rapidly, making exact optimization computationally intractable. In multi-server systems, effective coordination is also needed to prevent congestion, inefficient service allocation, and imbalanced workload distribution. Practical approaches must therefore achieve a balance between short-term responsiveness and long-term operational efficiency.

To model and solve these complex dynamic allocation problems, mathematical frameworks such as Markov Decision Processes (MDPs), Dynamic Programming (DP), Approximate Dynamic Programming (ADP), and Reinforcement Learning (RL) are commonly employed. MDPs provide a structured way to model decision-making under uncertainty, where system states evolve probabilistically, and decisions must be made to minimize long-term expected costs. However, as the number of states and decisions increases with multiple servers and service locations, solving MDPs using exact DP techniques becomes computationally infeasible due to the curse of dimensionality (Powell (2007)). To address this, ADP techniques aim to approximate optimal solutions by using heuristics and value function approximations. RL methods, by contrast, offer a data-driven approach, learning optimal policies through iterative interactions with the environment, making them particularly suitable for problems where system dynamics are complex and not fully known a priori. Hybrid approaches, such as ADP-RL combinations, have also been explored to balance computational efficiency and solution accuracy (Powell (2022)). The choice of method depends on the specific problem characteristics, including the availability of system state information, computational constraints, and the desired level of adaptability.

This research contributes to the development of scalable heuristics for dynamic resource allocation that are both computationally tractable and effective in practice by exploring ideas from operations research, stochastic optimization, and machine learning.

1.2 Problem description

This thesis addresses the problem of dynamically allocating mobile servers within a network, where customer demand occurs randomly over time. The network is composed of nodes connected by edges, with demand appearing at specific nodes based on independent Poisson processes. Mobile servers, which are homogeneous in nature, are responsible for traveling between these nodes to service the demand. Their movements must be controlled in order to minimize long-run average cost, which is incurred as a result of delays in processing the demand. Figure 1.1 provides a visual example of the system, depicting a network with 14 nodes, including 4 demand points, 10 intermediate stages and 3 mobile servers. For each pair of demand points, there exists a sequence of intermediate stages between them, and a server can traverse the corresponding stages when moving from one demand point to another. For example, suppose a server is located at demand point 3 in Figure 1.1 and wishes to move to demand point 4 in order to serve demand at that node. It can then pass through the intermediate stages 13, 11, 12 and 14, corresponding to the path $(3, 13, 11, 12, 14, 4)$. In the context of job scheduling, these intermediate stages could represent different stages in the setup time needed to reallocate effort from one job to another.

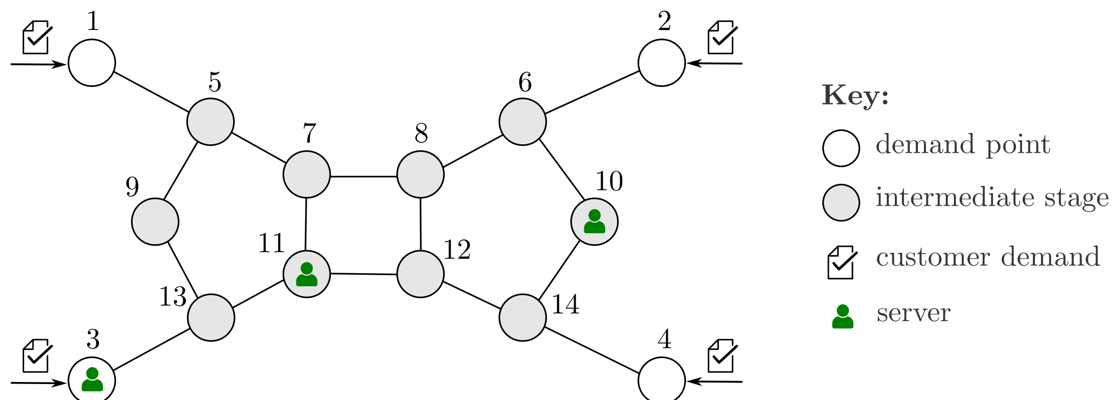

FIGURE 1.1: A network with 14 nodes, 4 demand points, 10 intermediate stages, and 3 servers. White-colored nodes represent demand points at which new customer demand arrives, and gray-colored nodes represent intermediate stages.

In this network setup, one server is actively serving customers at the demand point located in the bottom left of the figure, while the other two servers are not currently positioned at demand points. They could be in the process of moving between two demand points, or they could be stationary at their current locations. The goal is to develop strategies that enable these servers to respond most efficiently to incoming demand. Each of the next three chapters in this thesis concerns a slightly different

version of the general problem described above, and we present detailed mathematical formulations of the problem versions (including costs, number of servers etc.) within these individual chapters. Each chapter also includes a review of relevant literature, keeping the discussion focused on the specific problem setting.

A key feature of the general problem is that both the service and switching processes are interruptible. This allows servers to abandon their current tasks or alter their planned movements in response to newly arriving demand, as demand arrival rates and delay-related holding costs may vary across demand points. For instance, a server may choose to interrupt service at one demand point and move to another if a more urgent task (i.e., one whose delay would incur a higher expected system cost) arises elsewhere in the network. In the current model, we assume uniform switching rates between nodes, meaning that the time a server takes to travel from one node to an adjacent one follows an exponential distribution with a uniform rate. However, to further enhance the model's realism and flexibility, the methods developed in this thesis can be extended to account for edge-dependent and direction-dependent switching rates. In such cases, the switching time between two nodes (not necessarily adjacent) would no longer follow a simple distribution but would instead follow a more complex phase-type distribution, which can capture variations in travel times or switching costs depending on the specific characteristics of the edges (e.g., terrain, distance, traffic conditions, or node connectivity). This generalization allows for a more accurate model of server movements across diverse and varied network configurations.

The challenge of dynamically managing the movement and service of mobile servers lies in balancing immediate service needs with long-term operational efficiency. While processing the demand at a particular node may be effective for reducing costs in the short-term, this may require a server to have to locate itself far from other demand points, which may lead to larger long-run costs. Thus, a long-sighted view of the future is needed.

To formally represent this dynamic control problem, we model the system as a Markov Decision Process (MDP), where the state of the system includes the locations of the servers and the distribution of demand across the nodes. However, the dimensionality of the state space can become prohibitively large, making the exact solution of the MDP computationally intractable for large-scale systems. To overcome this challenge, the thesis investigates approximation techniques such as index heuristics, ADP and RL to develop scalable decision-making policies which avoid overwhelming computational complexity.

Chapters 2 and 3 of the thesis focus on a single-server system, where one server moves across the network, adjusting its path based on observed demand. This analysis is conducted for both finite-state and infinite-state systems. In both cases we develop index heuristics, which make decisions based on the system state and the network's topological structure. For the infinite-state systems, we extend these index heuristics so that they can make long-sighted decisions. For the finite-state systems, we use reinforcement learning techniques to approximate the value function associated with a base index policy and then use a one-step policy improvement approach during online implementation.

Chapter 4 then extends the analysis to a multi-server scenario within the infinite-state space framework, where multiple servers operate concurrently to service demand points across the network. This setup introduces extra challenges such as workload balancing between the servers. To address these, novel heuristics and solution techniques are developed, designed to scale with increasing system complexity. Initially, demand points are allocated to servers proportionally, and each server is assigned to a specific region within the network. Although alternative global heuristics permit unrestricted server movement across the network, we show that the region-based pre-assignment strategy achieves superior performance.

1.3 Practical applications

The dynamic allocation of mobile servers under uncertainty is a challenge in many real-world service systems, where demands are distributed across space, arrive stochastically, and often require a timely response. These systems rely on a limited number of mobile resources, which could be physical agents such as vehicles, robots, or drones, or virtual entities like computational tasks and software agents. A key feature considered in this thesis is the ability to interrupt both the movement of servers and ongoing service processes in response to evolving information. In dynamic settings, where conditions shift rapidly, such flexibility is often essential. The modeling framework developed in this thesis is broadly applicable across multiple domains. To provide context for our work, we present five application areas that demonstrate how interruptibility and dynamic service assignment function in practice.

1.3.1 Logistics and transportation

The modeling framework developed in this thesis is well suited to logistics and transportation systems, where a fleet of mobile servers, such as delivery vehicles, autonomous agents, must visit fixed locations like warehouses, depots, or customer addresses to carry out tasks. An important challenge in these systems is to dynamically adapt routes and assignments in response to uncertain, time-varying demand, while operating under practical constraints such as vehicle capacity, travel times, and resource availability.

Urban logistics provides a typical example of dynamic fleet management, where fleets must continuously respond to disruptions such as last-minute orders, variable traffic conditions, and unanticipated service delays (Berbeglia et al. (2010); Yang and Strauss (2017); Pillac et al. (2013); Cai et al. (2023)). There is a growing body of research focused on developing real-time decision-making methods for vehicle routing, task assignment, and fleet repositioning under such dynamic and uncertain conditions. For example, Ulmer (2020) proposes a joint pricing and routing framework for same-day delivery services, demonstrating that anticipatory policies can improve both customer coverage and revenue by preserving future flexibility. Liu and Luo (2023) develop a dynamic dispatching and routing approach for on-demand delivery systems that incorporates stochastic demand arrivals and achieves significant improvements over static policies through a structured approximation and exact solution algorithms. Pan and Liu (2023) introduce a deep reinforcement learning framework for the dynamic and uncertain vehicle routing problem, modeling real-time changes in customer demands via a partially observable Markov decision process and leveraging advanced neural network techniques to adapt to uncertainty and dynamics in urban logistics.

Freight logistics offers another prominent setting for dynamic fleet coordination, where vehicles such as trucks and drones must operate efficiently across scheduled deliveries and on-demand requests. Several surveys provide comprehensive overviews of freight transportation models and logistics decisions under various multimodal and supply chain settings (Bravo and Vidal (2013); Sun et al. (2015); Tavasszy et al. (2020)). Recent research has begun to incorporate dynamic and uncertain elements into freight routing and resource allocation. For example, Letnik et al. (2018) develop a dynamic assignment model for loading bays in last-mile deliveries, using fuzzy clustering and routing algorithms to reduce travel distance, fuel consumption, and emissions. Baubaid et al. (2023) propose a dynamic freight routing framework for less-than-truckload carriers, formulating the problem as a Markov decision process and applying ADP techniques

to integrate value function approximations within integer programming subproblems. Gu et al. (2023) consider a hybrid truck–drone system for handling scheduled deliveries alongside real-time pickup requests, introducing an MDP-based approach that reallocates routes dynamically and demonstrates substantial improvements in service efficiency and request acceptance through drone deployment.

Emergency logistics is another important setting for dynamic fleet coordination. In this context, the rapid deployment of vehicles, services, and relief resources is essential to mitigate the impact of natural or man-made disasters. Optimization-based models have played a central role in structuring such operations, as reviewed in several comprehensive surveys (Caunhye et al. (2012); Firoozabadi et al. (2017); Lu and Wang (2019); Kundu et al. (2022)). These reviews categorize key problems into pre-disaster tasks such as evacuation planning, facility location, and stockpiling, and post-disaster tasks such as relief distribution, casualty transportation, and emergency vehicle dispatch. Models typically address the allocation, routing, and scheduling of limited resources under uncertainty and severe time constraints, balancing competing objectives such as response time, coverage and logistics efficiency. For example, analytical models support dispatch and routing decisions for emergency vehicles (Lu and Wang (2019)), while mathematical formulations guide real-time coordination of rescue operations based on disaster severity and logistics constraints (Firoozabadi et al. (2017)). In practice, ADP methods have shown particular promise for managing real-time dispatching and relocation, enabling faster responses and more effective resource utilization under stochastic demand and rapidly evolving environments (Schmid (2012); Nasrollahzadeh et al. (2018)). These methods enhance responsiveness by allowing systems to adapt decisions as new information becomes available, supporting more effective emergency logistics operations.

While the above studies represent the state of the art in dynamic fleet management across urban logistics, freight transportation, and emergency response, they differ fundamentally from the problem studied in this thesis in that they generally do not explicitly model service interruptibility during task execution, and in some cases cannot be naturally formulated within a Markov decision process framework. By addressing this limitation, the present study contributes a novel and promising modeling direction for dynamic fleet management under uncertainty, enabling mid-task reassignment and flexible decision-making within a unified MDP-based structure.

1.3.2 Cloud computing and telecommunications

Cloud computing infrastructure offers a clear example of the mobile server model, where virtualization enables flexible allocation of resources across geographically distributed data centers. Major providers such as AWS, Microsoft Azure, and Google Cloud rely on orchestration systems like Kubernetes (see, for example, Paintsil (2025); Ifrah (2024); Sabharwal and Pandey (2020)) to manage virtual machines and containers, adjusting placement in response to user demand, hardware availability, or cost considerations. This area has been widely studied; surveys by Manzoor et al. (2020) and Aldossary (2021) provide overviews of key strategies and challenges, including issues of scale, efficiency, and energy use. More specific approaches include the work of Xiao et al. (2012), who propose methods for real-time adjustment of virtual machine resources based on workload, and Wang et al. (2015), who introduce an economic model that links pricing and user satisfaction to guide allocation , which helps inform the choice and interpretation of practical system parameters in this thesis. Belgacem (2022) offer a recent taxonomy of resource management techniques, classifying them by goals such as load balancing, fault tolerance, and energy efficiency. These studies reflect ongoing efforts to match resource allocation more closely to varying and uncertain workloads.

Edge computing, and in particular mobile edge computing (MEC), presents similar challenges in a more localized setting. Here, computational tasks must be assigned to edge nodes located near users, with decisions adapting to changes in demand, mobility patterns, and network status. Compared to cloud data centers, MEC environments typically face stricter latency requirements and have more limited capacity, making static allocation policies less effective. Surveys by Luo et al. (2021) and Malazi et al. (2022) summarize the main difficulties in MEC resource management, including hardware heterogeneity and the need for fast migration. Several practical methods have been proposed to address these issues: for instance, Wang et al. (2018) describe resource scheduling integrated with radio access networks; Feng et al. (2020) explore network slicing techniques for prioritizing traffic under quality-of-service constraints; and Liu et al. (2024) focus on task-level scheduling for IoT applications. These examples highlight the growing reliance of edge systems on real-time, context-aware scheduling mechanisms, which aligns closely with the central focus of this thesis.

Telecommunication networks form another domain where dynamic server scheduling is becoming more important. The development of 5G, and the anticipated features of 6G, has increased the range of service types supported, from low-latency communication

to large-scale sensor networks. To accommodate this diversity, operators use network slicing to allocate virtualized resources dynamically across the network, which further motivates the abstraction of the scheduling problem in a network context, a novel aspect set up in this thesis. Unlike traditional server systems, telecom networks must coordinate functions such as packet inspection, session management, and load balancing while meeting tight latency and reliability requirements. Surveys by Su et al. (2019) and Mamane et al. (2022) discuss the underlying scheduling models and formal classifications used in this setting. These systems often rely on SDN/NFV integration for control, as reviewed by Bonfim et al. (2019). Empirical studies by Cerrato et al. (2015) and Martin et al. (2018) illustrate how virtual network functions are instantiated and moved to match load conditions. These developments suggest that the mobile server model can be adapted to telecom settings, where services are actively positioned across both time and space.

Content delivery networks (CDNs) offer a related application, where dynamic server scheduling supports the distribution of digital content such as video, software, and web data. CDN operators route user requests to edge servers based on proximity, server availability, and current network load. In high-traffic situations, such as during live broadcasts or viral content surges, CDNs must adjust by replicating content and rerouting traffic to avoid congestion, highlighting the importance of timely update decisions and the potential need to interrupt ongoing processes in response to newly available information, a concept central to the modeling framework in this thesis. Surveys by Zolfaghari et al. (2020) and Salahuddin et al. (2017) review techniques used for content placement and delivery, including caching, prefetching, and load balancing. At the system level, dynamic scheduling policies have been proposed for virtual CDNs (Um et al. (2014)), and recent studies have applied learning-based methods to forecast demand and improve allocation (Lei et al. (2019)). Additional improvements have been reported in wireless networks through joint scheduling and power control (Choi et al. (2020)). These works show how CDN architectures rely on dynamic coordination to maintain service quality, in line with the broader principles of mobile server systems.

1.3.3 Manufacturing and robotics

Manufacturing systems increasingly rely on mobile robots and autonomous vehicles to perform tasks such as machine tending, material transport, and multi-step assembly.

These systems, such as automated guided vehicles (AGVs), autonomous mobile robots (AMRs), and mobile manipulators, can be seen as mobile servers that move within a factory to provide service where needed. This development is part of a broader shift toward Industry 4.0 and smart factories, where digital infrastructure supports more flexible and responsive control. Lee et al. (2015) describe the core features of cyber-physical manufacturing systems, including real-time feedback and decentralized decision-making. Wang et al. (2016) illustrate how such systems use connectivity and sensor data to support adaptive scheduling across the factory floor.

A common task in this setting is machine tending, where robots are responsible for loading and unloading machines in response to varying workloads. Instead of assigning a robot to a fixed station, manufacturers often deploy mobile manipulators that move between machines, depending on task priorities and current load. This introduces dynamic scheduling problems analogous to classical server routing and allocation models: the system must determine which robot should visit which station, in what order, and how frequently, while minimizing machine idle time and ensuring timely job completion. Jia et al. (2024) review vision-based machine-tending systems, focusing on robot perception and real-time control under changing workloads. Human-in-the-loop systems have also been proposed to handle unstructured environments; for example, Annem et al. (2019) and Al-Hussaini et al. (2020) describe hybrid approaches that combine autonomous scheduling with human supervision to improve flexibility and safety. Power management is another important concern in mobile settings. Jia et al. (2023) propose an integrated system that combines visual navigation and automated charging, allowing robots to operate continuously over long shifts without human intervention.

Material transport, another key task, typically involves AGVs or AMRs that carry workpieces, parts, or tools between stations, storage areas, and work cells. In flexible manufacturing systems, transport tasks must be scheduled in real time, often in the presence of deadlines, battery constraints, and workspace congestion. The underlying models are related to dynamic pickup-and-delivery problems with mobile servers and limited capacity. Jun et al. (2021) incorporate charging constraints directly into their scheduling framework, enabling the system to balance delivery tasks with battery management. More advanced task assignment algorithms have been proposed using reinforcement learning; Park et al. (2025), for instance, develop a hierarchical framework that coordinates multiple AMRs with heterogeneous load capacities. Martin et al. (2023) focus on environments with frequent disruptions or task reprioritization, proposing a dispatching scheme that re-evaluates assignments continuously. While this approach implicitly

captures the need to interrupt or revise ongoing tasks, it considers a fixed set of jobs and aims to minimize total processing time. In contrast, the model developed in this thesis allows service interruption at any point during task execution and permits dynamic switching between tasks, with performance evaluated under a long-run average holding-cost criterion. At the systems level, Vlachos et al. (2024) describe an IoT-based control platform that uses real-time sensor data to coordinate AGV movements and avoid deadlocks in shared paths.

As the number of robots in a system increases, coordination becomes more complex. Multi-robot systems must share workspaces, avoid collisions, and complete tasks within strict timing constraints. In some settings, robots collaborate directly with each other or with human workers on joint tasks, such as assembling components or transferring parts. This requires real-time coordination policies that account for both spatial proximity and task dependencies. In this thesis, these interactions are modeled using a network-based representation, which provides a unified abstraction for robot coordination and constitutes a novel modeling perspective in the manufacturing and robotics context. By contrast, existing studies primarily focus on specific aspects of multi-robot operation rather than adopting such a unified abstraction. Villani et al. (2018) survey collaborative robotics in industrial settings, emphasizing the challenges of safety, communication, and mixed autonomy. Fault detection and recovery are also essential to maintaining productivity. Sabry and Amirulddin (2024) review recent methods for diagnosing faults in robotic systems, including sensor-based monitoring and data-driven anomaly detection. For systems that involve shared human–robot workspaces, safety constraints are often enforced through adaptive motion planning. Byner et al. (2019), for example, propose a method that adjusts robot trajectories in real time based on proximity to human workers, ensuring compliance with safety zones and reducing the risk of accidents.

1.3.4 Maintenance and infrastructure inspection

The mobile server framework also offers an effective modeling structure for coordinating mobile maintenance agents, including technicians, drones, and autonomous robots, across dispersed infrastructure systems such as power grids, oil and gas pipelines, railway networks, bridges, and roadways. These systems typically involve dispersed spatial

configurations, are subject to stochastic deterioration and failures, and require a balance between preventive maintenance and timely response to emergent disruptions.

In the context of power grids, reinforcement learning has been applied to derive optimal policies for maintenance and operational decision-making under uncertainty. Rocchetta et al. (2019) propose an MDP framework that accounts for component aging and stochastic failure mechanisms, enabling agents to learn cost-effective maintenance strategies over time. In parallel with these developments, integrated smart grid platforms have been developed to support real-time system visualization, monitoring, and control. For instance, the Smart Power Management System by Tang (2011) provides a decision-support environment for managing energy flow and system contingencies in dynamic grid environments. Autonomous aerial systems are also increasingly being deployed for inspection of transmission infrastructure. Calvo et al. (2022) present a high-level mission planning and execution framework for heterogeneous teams of drones conducting power line inspections. Their approach emphasizes cooperative routing and scheduling under spatial, temporal, and energy constraints, demonstrating its feasibility in complex field scenarios. Furthering this line of work, Silano et al. (2021) propose a multi-layer software architecture for drone fleets, which integrates cognitive capabilities such as onboard perception, adaptive flight control and decentralized coordination. This modular design enhances the robustness and scalability of aerial inspection systems for large-scale power grid monitoring.

For oil and gas pipelines, the inspection and maintenance challenge can be even greater due to their extended geographical coverage, exposure to environmental conditions and limited accessibility. Dey (2004) introduces a decision support system that incorporates economic, reliability, and risk-based factors to prioritize inspection schedules under budgetary constraints. At the policy level, Iqbal et al. (2017) conduct a comprehensive review of inspection practices, noting a fragmented regulatory landscape and emphasizing the need for more consistent strategies that address technical, environmental, and organizational factors. Technological solutions are also advancing: the SPAMMS system described by Kim et al. (2010) integrates mobile sensor units and embedded monitoring technologies to autonomously detect pipeline anomalies such as leaks or corrosion. Building on this approach, Botteghi et al. (2021) propose a hierarchical reinforcement learning framework for pipeline inspection robots. Their architecture separates high-level mission planning from low-level control, enabling agents to adaptively navigate and inspect complex pipeline environments while responding to sensor feedback.

In railway systems, regular and well-coordinated maintenance is critical for ensuring safety and sustaining high service capacity. One of the earliest integrated models in this area is developed by Higgins (1998), who constructs a scheduling framework for allocating maintenance crews to track repair tasks while accounting for operational disruptions. More recent approaches emphasize predictive maintenance using data analytics. Li et al. (2014) use historical sensor and maintenance records to train predictive models that anticipate component failures, allowing for proactive maintenance interventions and improved network efficiency. In post-disruption scenarios, such as after storms or natural disasters, rapid restoration of service often relies on mobile repair teams. Morshedlou et al. (2018) formulate a work crew routing problem that captures access constraints, task urgency and equity among crews. Their optimization model supports centralized planning of workforce deployment to expedite infrastructure recovery across affected regions.

In Chapter 3, this thesis studies a finite-state Markov decision process model for maintenance and repair operations with a single mobile agent. The index-based policy and reinforcement learning approach developed therein provide an efficient and scalable decision-support framework for infrastructure maintenance under uncertainty. While the analysis focuses on a single-server setting, the modeling framework can be extended to systems with multiple mobile maintenance agents, as explored in Chapter 4, and thus represents a promising direction for future research.

1.3.5 Healthcare service delivery

Mobile server systems are applicable to various healthcare operations where timely and geographically responsive service is essential. Relevant applications include emergency medical response, home health care delivery, and mobile clinic deployment. Each setting presents specific challenges related to resource coordination, demand uncertainty, and spatial coverage, which have been addressed using various operations research methods.

In emergency medical services (EMS), dynamic vehicle dispatching and redeployment strategies are key to improving patient outcomes. Bandara et al. (2012) develop an optimization model for dispatching ambulances with the aim of maximizing patient survivability, considering the spatial and temporal dynamics of incident arrivals. Building on this, ADP has been used to generate near-optimal redeployment policies that adapt to real-time demand fluctuations. Maxwell et al. (2010) apply such techniques to

ambulance positioning, enabling systems to proactively rebalance coverage as calls are completed or new ones arrive. In earlier work, Gendreau et al. (2001) propose a dynamic relocation model using a parallel tabu search heuristic, enabling efficient reallocation of ambulances under evolving demand conditions. A key feature introduced in this thesis, “interruptibility”, allows ambulances to switch dynamically between tasks in response to evolving stochastic information, enabling timely adjustments to allocation plans and more efficient use of EMS resources in practice.

In home health care, mobile caregivers must be routed and scheduled to meet patient needs while considering time windows, service dependencies, and travel constraints. Fikar and Hirsch (2017) provide a review of existing models for home health care routing and scheduling, noting the importance of accounting for care continuity and individual preferences. Mankowska et al. (2014) introduce a model for coordinating multiple caregivers when interdependent services are required, such as joint or sequential visits. Hybrid metaheuristics are commonly used to address the resulting computational complexity. For instance, Bertels and Fahle (2006) combine different heuristic components to improve performance on realistically sized problem instances.

Beyond traditional care settings, mobile medical units (MMUs) and autonomous clinics have been explored as solutions to expand healthcare access in underserved or disaster-affected regions. Büsing et al. (2021) present a robust optimization framework for strategic planning of MMUs, accounting for both predictable (steerable) and uncertain (unsteerable) demand scenarios. New technologies are also reshaping delivery mechanisms. Liu et al. (2022) discuss the role of autonomous mobile clinics that operate with minimal human intervention to deliver cost-effective, on-demand services. Advances in mobile health applications contribute to this trend. For example, Butt et al. (2024) provide a taxonomy of mobile health monitoring frameworks that facilitate remote patient supervision and early detection of health anomalies. From a diagnostic perspective, Baron and Haick (2024) explore sensor-integrated mobile clinics capable of performing sophisticated medical evaluations, including noninvasive screening and real-time biomarker analysis.

1.4 Theoretical framework

1.4.1 Markov Decision Processes

Markov Decision Processes (MDPs) provide a mathematical framework for sequential decision-making in stochastic environments, where each action taken influences both the immediate *costs* and the probabilistic evolution of the system's state (Puterman (1994)). Formally, an MDP is defined by a 4-tuple (S, A, P, c), where:

- S is the state space, which may be finite or countably infinite;
- A is the action space, and $A_{\mathbf{x}} \subseteq A$ denotes the set of admissible actions in state $\mathbf{x} \in S$;
- $P_{\mathbf{x},\mathbf{y}}(a)$ is the transition probability of moving from state $\mathbf{x}$ to state $\mathbf{y}$ when action $a \in A_{\mathbf{x}}$ is taken;
- $c(\mathbf{x}, a)$ is the immediate *cost* incurred when action a is taken in state $\mathbf{x}$.

A policy $\theta = \{\theta_t\}_{t \geq 0}$ is a sequence of decision rules indexed by time t, where each θ_t specifies a probability distribution over actions given the current state at time t. A policy is called *stationary* if $\theta_t = \theta$ for all $t \geq 0$, and *deterministic* if it selects a single action with probability one in each state.

The performance of a policy in an MDP is typically measured using one of two *cost* criteria: (i) the *discounted cost*, which emphasizes near-term costs by downweighting future ones through a discount factor $\gamma \in [0, 1)$; (ii) the *average cost*, which evaluates a policy based on the long-run average cost per time step.

MDPs can be formulated in either continuous or discrete time, depending on the nature of the application.

1. **Continuous-time MDPs (CTMDPs):** In a CTMDP, the system transitions between states according to a continuous-time Markov process.

   (a) Discounted-cost criterion: Given a policy θ and initial state $\mathbf{x}$, the total expected discounted cost is defined as:

$$V_\theta(\mathbf{x}) = \mathbb{E}_{\mathbf{x}}^{\theta} \left[\int_0^\infty e^{-\gamma t} c(\mathbf{x}_t, a_t)\, dt \right], \tag{1.1}$$

where $\mathbf{x}_t$ denotes the state of the system at time t, and a_t is the action selected according to the policy θ given $\mathbf{x}_t$.

(b) Average-cost criterion: The long-run expected average cost under policy θ is given by:

$$g_\theta(\mathbf{x}) = \limsup_{T\to\infty} \frac{1}{T}\mathbb{E}_{\mathbf{x}}^{\theta}\left[\int_0^T c(\mathbf{x}_t, a_t)\, dt\right]. \tag{1.2}$$

A classical technique for analyzing CTMDPs is through *uniformization* (Serfozo (1979)), which reduces a CTMDP to an equivalent discrete-time MDP with properly rescaled transition probabilities, allowing application of discrete-time methods.

2. **Discrete-time MDPs (DTMDPs):** In a DTMDP, the system evolves at discrete time steps $t = 0, 1, 2, \ldots$, and the decision-maker selects an action a_t at each step based on the current state $\mathbf{x}_t$.

(a) Discounted-cost criterion: The total expected discounted cost under policy θ is

$$V_\theta(\mathbf{x}) = \mathbb{E}_{\mathbf{x}}^{\theta}\left[\sum_{t=0}^{\infty}\gamma^t c(\mathbf{x}_t, a_t)\right]. \tag{1.3}$$

To find the optimal value function $V^*(\mathbf{x}) = \inf_\theta V_\theta(\mathbf{x})$ and identify a policy θ^* such that $V_{\theta^*}(\mathbf{x}) = V^*(\mathbf{x})$ for all $\mathbf{x}$, the V^* satisfies the *Bellman optimality equation* (Puterman (1994)):

$$V^*(\mathbf{x}) = \min_{a\in A_{\mathbf{x}}}\left\{c(\mathbf{x}, a) + \gamma\sum_{\mathbf{y}\in S} P_{\mathbf{x},\mathbf{y}}(a)V^*(\mathbf{y})\right\}, \quad \forall\mathbf{x}\in S. \tag{1.4}$$

(b) Average-cost criterion: The long-run expected average cost under policy θ is defined as:

$$g_\theta(\mathbf{x}) = \limsup_{T\to\infty} \frac{1}{T}\mathbb{E}_{\mathbf{x}}^{\theta}\left[\sum_{t=0}^{T-1} c(\mathbf{x}_t, a_t)\right]. \tag{1.5}$$

If the limit exists and is independent of $\mathbf{x}$, denote it by g_θ. The optimal long-run expected average cost is

$$g^* := \inf_\theta g_\theta. \tag{1.6}$$

Under appropriate conditions (e.g., unichain, bounded costs), there exists a stationary deterministic policy θ^* and a relative value function $h^* : S \to \mathbb{R}$ such that (g^*, h^*) solves the *average-cost Bellman equation* (Puterman

(1994)):

$$g^* + h^*(\mathbf{x}) = \min_{a \in A_{\mathbf{x}}} \left\{ c(\mathbf{x}, a) + \sum_{\mathbf{y} \in S} P_{\mathbf{x},\mathbf{y}}(a) h^*(\mathbf{y}) \right\}, \quad \forall \mathbf{x} \in S. \tag{1.7}$$

The function h^* captures the differential cost relative to a reference and characterizes the optimal policy structure.

The problems studied in this thesis are formulated as average-cost DTMDPs, enabling the analysis to focus on long-run performance and steady-state policy. The theory and solution methods for DTMDPs differ depending on whether the state space is finite or infinite. Finite-state models are generally more tractable and allow stronger guarantees and computational methods, whereas infinite-state models require more sophisticated methods and additional assumptions such as stability or ergodicity to ensure they are well-posed.

Solution techniques for DTMDPs under the average-cost criterion include:

- Dynamic Programming (DP): Iterative methods such as *value iteration* and *policy iteration* solve the Bellman equations directly. These methods are effective for finite-state models and converge under suitable conditions in infinite-state settings. See Section 1.4.2.
- Linear Programming (LP): The optimal policy can be obtained by solving a linear program that represents how frequently each action is taken in each state over the long run. This approach focuses on the steady-state behavior of the system and helps identify the best action distribution to minimize the average cost. For finite-state problems, this reduces to a finite linear program that can be solved using standard optimization methods. However, LP methods are not used in this thesis.
- Approximate Dynamic Programming (ADP): ADP approximates value functions or policies using basis functions, aggregation or simulation. This is necessary for large-scale or infinite-state MDPs. See Section 1.4.3.
- Reinforcement Learning (RL): RL algorithms estimate optimal policies through interaction with the environment, without explicit transition or cost models. This is suitable for data-driven or online problems. See Section 1.4.4.

1.4.2 Dynamic Programming

Dynamic Programming (DP), introduced by Bellman (1957), provides a recursive optimization framework for solving sequential decision-making problems. It is based on the *principle of optimality*, which states that any tail segment of an optimal policy must itself be optimal, irrespective of the preceding decisions. This principle enables the decomposition of complex decision processes into a sequence of simpler subproblems whose solutions can be composed to form an optimal policy.

DP methods are particularly effective for finite-state DTMDPs under the average-cost criterion, and can be extended to infinite-state systems under suitable regularity conditions. The goal is to find a policy that minimizes the long-run expected average cost per time step. The key quantities are the long-run average cost g_θ under a policy θ, and the associated relative value function $h_\theta : S \to \mathbb{R}$, which measures the differential cost starting from each state relative to a reference.

Two fundamental DP algorithms for average-cost DTMDPs are:

1. **Value Iteration:** This iterative method approximates the relative value function h through repeated updates. Given an initial guess h_0, the iteration for $k \geq 0$ updates h_{k+1} by

$$h_{k+1}(\mathbf{x}) = \min_{a \in A_\mathbf{x}} \left\{ c(\mathbf{x}, a) + \sum_{\mathbf{y} \in S} P_{\mathbf{x},\mathbf{y}}(a) h_k(\mathbf{y}) \right\} - h_k(\mathbf{x}_0), \quad \forall \mathbf{x} \in S, \tag{1.8}$$

   where $\mathbf{x}_0$ is a fixed reference state used for normalization to ensure uniqueness and stability. Under suitable conditions, h_k converges (up to an additive constant) to the relative value function h^*. The optimal average cost g^* can be estimated from the stabilized increments of $h_k(\mathbf{x}_0)$.

2. **Policy Iteration:** This method alternates between policy evaluation and policy improvement. Given a policy θ_k, the evaluation step solves for the average cost g_{θ_k} and relative value function h_{θ_k} satisfying

$$g_{\theta_k} + h_{\theta_k}(\mathbf{x}) = c(\mathbf{x}, \theta_k(\mathbf{x})) + \sum_{\mathbf{y} \in S} P_{\mathbf{x},\mathbf{y}}(\theta_k(\mathbf{x})) h_{\theta_k}(\mathbf{y}), \quad \forall \mathbf{x} \in S. \tag{1.9}$$

The improvement step constructs a new policy θ_{k+1} by choosing

$$\theta_{k+1}(\mathbf{x}) \in \arg\min_{a \in A_{\mathbf{x}}} \left\{ c(\mathbf{x}, a) + \sum_{\mathbf{y} \in S} P_{\mathbf{x},\mathbf{y}}(a) h_{\theta_k}(\mathbf{y}) \right\}. \tag{1.10}$$

Under standard assumptions (e.g., finite state, action spaces and unichain structure), policy iteration converges in finitely many steps to an optimal stationary deterministic policy.

While DP methods provide exact solutions to DTMDPs, their applicability is limited by the *curse of dimensionality*: computational requirements grow exponentially with the sizes of the state and action spaces. This motivates the use of approximate solution methods discussed in Sections 1.4.3 and 1.4.4, which employ function approximation, sampling, and simulation to handle large-scale or infinite-state problems efficiently.

1.4.3 Approximate Dynamic Programming

In large or continuous state spaces, classical dynamic programming techniques (such as value iteration or policy iteration mentioned in Section 1.4.2) become computationally infeasible. This difficulty is particularly pronounced under the average-cost criterion, which focuses on long-run, steady-state performance rather than transient behavior. Approximate Dynamic Programming (ADP) addresses this challenge by leveraging simulation-based learning, function approximation, and iterative updates to approximate value functions or policies without requiring exact transition dynamics (Bertsekas and Tsitsiklis (1996); Powell (2007)).

Under the average-cost criterion in DTMDPs, the performance of a stationary policy θ is characterized by a scalar average cost g_θ and a relative value function $h_\theta(\mathbf{x})$, which together satisfy the Poisson equation:

$$g_\theta + h_\theta(\mathbf{x}) = \mathbb{E}_{a \sim \theta(\mathbf{x})} \left[c(\mathbf{x}, a) + h_\theta(\mathbf{y}) \right], \quad \forall \mathbf{x} \in S, \tag{1.11}$$

where $\mathbf{y}$ is the next state following action a taken in state $\mathbf{x}$, and $\theta(\mathbf{x})$ represents the probability distribution over actions prescribed by the policy at state $\mathbf{x}$. Solving this

equation exactly requires full knowledge of the transition dynamics and is generally intractable when S is large or infinite.

ADP overcomes this barrier by using data-driven methods to approximate value functions and policies. These methods can be broadly grouped into two perspectives: (i) *policy evaluation*, which assumes a fixed policy θ and seeks to estimate its performance; and (ii) *policy learning and improvement*, which aims to directly learn improved policies through iterative interaction with the environment.

1. **Policy Evaluation:** Given a stationary policy θ, the goal is to approximate the pair $(g_\theta, h_\theta(\mathbf{x}))$ using the following methods:

   (a) Function approximation: The relative value function $h_\theta(\mathbf{x})$ is often approximated using a parameterized model, reducing the dimensionality of the state space. A common approach is linear function approximation:

   $$h_\theta(\mathbf{x}) \approx w^\top \phi(\mathbf{x}), \tag{1.12}$$

   where $\phi(\mathbf{x})$ is a feature vector and $w \in \mathbb{R}^d$ is a learnable parameter vector. For more complex problems, nonlinear architectures, such as neural networks, are used in deep ADP (see, for example, Xu et al. (2020)).

   (b) Simulation-based estimation: When the transition kernel is unknown or computationally intractable, simulation-based learning can be employed. In this approach, the agent interacts with the environment or a simulator, and the action taken at each state is determined by the policy θ. The average cost over a sample path of length t is estimated as:

   $$\hat{g}_t = \frac{1}{t} \sum_{k=0}^{t-1} c(\mathbf{x}_k, a_k), \tag{1.13}$$

   where $(\mathbf{x}_k, a_k)$ are the state-action pairs visited during the simulation. These empirical samples can then be used to estimate the relative value function $h_\theta(\mathbf{x})$ by applying regression or incremental updates (Powell (2007)).

   (c) Temporal-difference (TD) learning: TD learning incrementally updates value estimates from one-step transitions under policy θ. In the average-cost setting, a TD(0) update for the relative value function is:

   $$h(\mathbf{x}_t) \leftarrow h(\mathbf{x}_t) + \alpha_t \left[c(\mathbf{x}_t, a_t) - \hat{g}_t + h(\mathbf{x}_{t+1}) - h(\mathbf{x}_t) \right], \tag{1.14}$$

where α_t is the learning rate and $\hat{g}_t$ is an estimate of the average cost. The action a_t is sampled from $\theta(\mathbf{x}_t)$, making the data on-policy. The TD error is defined as:

$$\delta_t = c(\mathbf{x}_t, a_t) - \hat{g}_t + h(\mathbf{x}_{t+1}) - h(\mathbf{x}_t). \tag{1.15}$$

As $h_\theta(\mathbf{x})$ is defined up to an additive constant, normalization (e.g., fixing $h(\mathbf{x}_0) = 0$) is commonly applied. Multi-step methods, such as TD(λ), can also be adapted to the average-cost case using eligibility traces and corrections (Bertsekas and Tsitsiklis (1996); Powell (2007)).

(d) Structural approximations for infinite state spaces: For problems with infinite or unbounded state spaces, a powerful method for evaluating a fixed policy θ is to construct a sequence of finite-state MDPs that approximate the original system. A classical approach introduced by Sennott (1989, 1997) develops such structural approximations under the average-cost criterion. These approximating models preserve the essential dynamics of the system under policy θ, and under certain regularity conditions, the value functions and average costs of the finite models converge to those of the original problem. While these approximations do not directly optimize the policy, they enable tractable evaluation of the policy θ in otherwise intractable settings.

2. **Policy Learning and Improvement:** Beyond evaluating a fixed policy θ, many ADP methods focus on directly learning or improving a policy, often without assuming prior knowledge of an optimal policy. This involves estimating g_θ and h_θ for multiple candidate policies and using them to guide policy updates.

(a) Policy gradient methods: If the policy is parameterized by λ, denoted as θ_λ, a gradient-based approach can be used to minimize the average cost:

$$\nabla_\lambda g_{\theta_\lambda} \approx \mathbb{E}\left[\nabla_\lambda \log \theta_\lambda(a \mid \mathbf{x}) \cdot Q_{\theta_\lambda}(\mathbf{x}, a)\right], \tag{1.16}$$

where $\theta_\lambda(a \mid \mathbf{x})$ is the probability of taking action a in state $\mathbf{x}$ under the parameterized policy, and $Q_{\theta_\lambda}(\mathbf{x}, a)$ is the corresponding action-value function. In practice, $Q_{\theta_\lambda}(\mathbf{x}, a)$ is typically approximated using temporal-difference methods or Monte Carlo estimates. The policy parameters are optimized based on simulated trajectory evaluations (Spall (2005); Meyn (2008)).

(b) Actor–critic methods: These methods simultaneously learn a policy (the actor) and a value function (the critic). The critic estimates g_θ and h_θ (or Q_θ), which are used to guide the actor's policy updates. This allows on-policy

learning with incremental updates, suitable for steady-state environments (Marbach and Tsitsiklis (2002); Kushner and Yin (2003)).

(c) Approximate policy iteration: An extension of classical policy iteration, this method alternates between approximate policy evaluation and policy improvement. After evaluating a policy approximately using ADP techniques, the policy is greedily improved based on the learned value estimates (Puterman (1994)).

(d) Exploration and off-policy learning: When the policy θ is not fixed, exploration becomes essential to ensure that the learning algorithm gathers sufficient data across the state–action space. Off-policy variants of ADP, such as Q-learning, adapt by learning value functions under a target policy θ while following a different behavior policy (Watkins and Dayan (1992)).

ADP provides a unifying framework for both evaluating given policies θ and learning improved ones in average-cost settings. The choice between these perspectives depends on the application context: whether a baseline policy θ is known and trusted, or the goal is to discover a near-optimal policy from scratch using data and simulation. The flexibility of ADP allows it to scale to large or continuous state spaces, making it a powerful tool in solving DTMDPs and other dynamic optimization problems.

1.4.4 Reinforcement Learning

Reinforcement Learning (RL) extends the MDP framework to settings where transition dynamics and cost structures are either unknown or difficult to model explicitly. Unlike DP (Section 1.4.2), which assumes full knowledge of transition probabilities and costs, RL algorithms learn optimal policies through direct interaction with the environment. By observing transitions and incurred costs, the agent incrementally improves its decision-making over time (Sutton and Barto (2018)). RL thus facilitates near-optimal control in complex, data-driven environments where explicit modeling is infeasible.

Part of the work in this thesis focuses on RL for average-cost DTMDPs, where the goal is to minimize the long-run expected average cost, as defined in (1.5) and (1.6). To accommodate this objective, RL algorithms either jointly estimate the optimal long-run

expected average cost g and the action-value function Q, or use estimates of Q alone to infer g, according to the average-cost Bellman equation for DTMDPs.

We now present three representative simulation-based approaches: relative Q-learning, R-learning, and Q-P-learning, formulated under the cost-minimization criterion, following the frameworks introduced in Chapter 9 of Gosavi (2003). At each time step t, the agent observes the current state $\mathbf{x}_t \in S$, selects an action $a_t \in A_{\mathbf{x}_t}$, incurs an immediate cost $c(\mathbf{x}_t, a_t)$, and transitions to a new state $\mathbf{x}_{t+1}$ according to unknown dynamics.

1. **Relative Q-Learning:** This algorithm modifies the classical Q-learning update to estimate the relative action-value function without explicitly tracking the long-run average cost. A fixed reference state-action pair $(\mathbf{x}^*, a^*)$ is used as a normalization baseline for value updates. The Q-value for the current state-action pair is updated by:

$$Q(\mathbf{x}_t, a_t) \leftarrow (1 - \alpha_t)Q(\mathbf{x}_t, a_t) + \alpha_t \Big[c(\mathbf{x}_t, a_t) + \min_{a' \in A_{\mathbf{x}_{t+1}}} Q(\mathbf{x}_{t+1}, a') - Q(\mathbf{x}^*, a^*)\Big],$$

   where $\alpha_t \in (0, 1)$ is a learning rate satisfying standard diminishing step-size conditions. The term $Q(\mathbf{x}^*, a^*)$ centers the updates relative to the fixed baseline pair.

2. **R-Learning:** This method extends Q-learning by maintaining an explicit estimate of the long-run average cost g_t alongside the Q-values. The updates are:

$$Q(\mathbf{x}_t, a_t) \leftarrow (1 - \alpha_t)Q(\mathbf{x}_t, a_t) + \alpha_t \left[c(\mathbf{x}_t, a_t) + \min_{a' \in A_{\mathbf{x}_{t+1}}} Q(\mathbf{x}_{t+1}, a') - g_t\right],$$

$$g_{t+1} \leftarrow (1 - \beta_t)g_t + \beta_t \left[c(\mathbf{x}_t, a_t) + \min_{a' \in A_{\mathbf{x}_{t+1}}} Q(\mathbf{x}_{t+1}, a') - \min_{a' \in A_{\mathbf{x}_t}} Q(\mathbf{x}_t, a')\right],$$

   where α_t and β_t are learning rates for the Q-values and g_t is the average cost estimate, respectively. Unlike relative Q-learning, R-learning separately tracks g and Q.

3. **Q-P-Learning:** This algorithm maintains two sets of action-value estimates, Q and P, to improve learning stability in average-cost problems. Actions are selected via exploration (e.g., ϵ-greedy on Q), and Q is updated using fixed P-values:

$$Q(\mathbf{x}_t, a_t) \leftarrow (1-\alpha_t)Q(\mathbf{x}_t, a_t)+\alpha_t \left[c(\mathbf{x}_t, a_t) + Q\big(\mathbf{x}_{t+1}, \arg\min_{a' \in A_{\mathbf{x}_{t+1}}} P(\mathbf{x}_{t+1}, a')\big) - Q(\mathbf{x}^*, a^*)\right].$$

Over K outer iterations, P is periodically updated by copying Q (i.e., $P \leftarrow Q$), stabilizing the learning process. At the final iteration, Q yields a greedy policy θ, and g is estimated empirically under θ.

The convergence of these algorithms to the optimal average cost g and associated action-value function Q has been established under standard conditions, including sufficient exploration of all state-action pairs and diminishing step sizes. Exploration strategies such as ϵ-greedy, softmax, and upper confidence bound (UCB) ensure broad state-action sampling (Sutton and Barto (2018)).

1.4.5 Index heuristics

In the context of large-scale stochastic control problems, such as the dynamic allocation of mobile servers within a networked environment, the state space associated with the underlying MDP typically grows combinatorially with the number of system components (e.g., servers, demand points, and node locations). While ADP and RL methods offer scalable alternatives to DP, these methods often face limitations when rapid decision-making and interpretability are required. Index heuristics form an alternative class of methods that aim to balance computational tractability and policy quality by assigning state-dependent scalar values, referred to as *indices*, to tasks. The indices serve as a measure of the relative priority or urgency of each task, based on the current state of the system. By evaluating these indices, decision-maker can prioritize tasks dynamically, enabling efficient and decentralized control strategies.

Index heuristics were first proposed in the context of *multi-armed bandit* (MAB) problems, in which multiple stochastic projects (or 'arms') compete for a limited pool of resources (typically a single active slot). Each project yields a stochastic reward, and the decision-maker must select one project at each time to maximize expected cumulative discounted reward. Gittins (1979) demonstrated that in the MAB setting, the optimal policy is indexable; that is, each project in state $\mathbf{x}$ can be assigned an index $\gamma(\mathbf{x})$ representing the expected reward rate achievable from that state under an optimal stopping rule. The optimal action at each decision epoch is to activate the project with

the highest index. The *Gittins index* is defined as:

$$\gamma(\mathbf{x}) = \sup_{\tau \geq 1} \frac{\mathbb{E}_{\mathbf{x}} \left[\sum_{t=0}^{\tau-1} \beta^t R(\mathbf{x}_t) \right]}{\mathbb{E}_{\mathbf{x}} \left[\sum_{t=0}^{\tau-1} \beta^t \right]}, \tag{1.17}$$

where $\beta \in (0, 1)$ is the discount factor, $R(\mathbf{x}_t)$ denotes the reward at time t, and τ is a stopping time. A detailed exposition of index theory in the context of multi-armed bandit problems, including algorithmic developments and applications, is provided in the monograph by Gittins et al. (2011).

While the Gittins index is optimal in the classical setting where only active arms evolve, many practical systems, including the one considered in this thesis, involve more complex dynamics in which all components evolve over time regardless of whether they are actively served. These are known as *restless bandit* problems. Whittle (1988) extended the index approach to this setting by introducing a Lagrangian relaxation that penalizes resource usage through a subsidy for passivity. A project is said to be *indexable* if the set of passive-optimal states increases monotonically as the subsidy increases. When indexability holds, each state can be assigned a *Whittle index*, denoting the critical subsidy at which it becomes optimal to activate the project.

Although Whittle index policies are not generally optimal, they are known to be asymptotically optimal under certain limiting regimes (Weber and Weiss (1990)) and have demonstrated strong empirical performance across diverse applications (see, for example, Ansell et al. (2003) and Glazebrook et al. (2005)). Their structure naturally enables scalable implementation: at each time, the controller computes or retrieves the index associated with each decision project and selects those with the highest indices, subject to resource constraints. This decentralized decision rule avoids the curse of dimensionality and leads to interpretable and computationally efficient policies. Further theoretical developments in indexable families of restless bandit problems, with particular emphasis on structural properties are presented in Glazebrook et al. (2006).

Index heuristics have proven effective in a range of stochastic resource allocation problems. For example, Shone et al. (2019) demonstrated how index heuristics facilitated real-time decision-making in congested queueing systems with time-varying demand, leading to significant reductions in computational complexity. Clarkson et al. (2020) applied index heuristics in a search problem with discrete locations, achieving scalable solutions that improved task prioritization. Consequently, index heuristics serve as an important bridge between the theoretical optimality of DP and the scalable adaptability

of ADP and RL. Their decentralized nature and interpretability make them especially well-suited for real-time, interruptible control systems, such as those explored in this thesis.

1.5 Research contributions

This thesis combines three related papers that study how to schedule mobile servers in systems with random demand across a network. These problems arise when servers must respond in real time to spatially and temporally distributed demand, with setup and service times that are random, interruptible and sequence-dependent. The decision-making process is formulated as an MDP, covering both finite-state and infinite-state models.

The specific contributions of each paper are outlined below:

1. **Paper 1: Stochastic Dynamic Job Scheduling with Interruptible Setup and Processing Times**

   This paper introduces a novel infinite-state MDP framework for modeling dynamic job scheduling by a single server over a network. Jobs require setup and processing times that are both interruptible and sequence-dependent, and the server must make real-time, location-aware decisions. A key theoretical contribution is the proof of system stability and the existence of stationary optimal policies under a long-run average cost criterion, despite the challenges posed by an infinite state space and frequently occurring decision epochs. To address the network-based scheduling problem, we propose two families of long-sighted, index-based heuristics: the K-stop and (K from L)-stop policies. These heuristics exploit the network topology to guide scheduling decisions while allowing flexible interruption operations. The K-stop policy satisfies a pathwise consistency property, ensuring that the server reaches demand points in finite time, while the (K from L)-stop policies are designed to scale effectively to larger networks. Extensive numerical experiments validate the effectiveness of these heuristics. In small problem instances, both the K-stop and (K from L)-stop policies achieve performance much closer to optimality than an alternative heuristic based on the work of Duenyas and Van Oyen (1996).

2. **Paper 2: Dynamic Repair and Maintenance of Heterogeneous Machines Distributed on a Network: A Reinforcement Learning Approach**

   This work adapts the methods in Paper 1 to a finite-state MDP model with more general cost functions, representing machine degradation and maintenance requirements. The system features interruptible and sequence-dependent setups alongside state transitions due to stochastic wear and repair. We design index-based heuristics based on an equivalence between cost and reward formulations and further enhance these using reinforcement learning techniques. Specifically, we apply value function approximation to implement one-step policy improvement in an online setting. We also use theoretical analysis to establish conditions under which our heuristics become optimal. Extensive simulation experiments demonstrate the effectiveness of our heuristics across diverse system configurations.

3. **Paper 3: Stochastic Dynamic Job Scheduling with Interruptible Setup and Processing Times for Multiple Servers**

   Building on the single-server infinite-state model in Paper 1, this paper generalizes the framework to multiple mobile servers operating over a shared network. We develop both local heuristics, based on a novel demand-allocation scheme using gradient descent to partition the network into regions, and global coordination strategies, where servers dynamically adapt routing and job assignments in response to system-wide load conditions. To enable scalable coordination, we develop algorithms that generalize the single-server scheduling policies by adapting the 1-stop rule from the K-stop policy in Paper 1. These algorithms ensure that each server primarily operates within its designated region while allowing for occasional cross-region service when beneficial. Through comprehensive computational experiments, we validate the effectiveness of our methods and highlight the trade-offs between decentralized simplicity and centralized efficiency.

The three papers present a framework for dynamic scheduling in stochastic networks. They tackle both infinite-state and finite-state MDPs under realistic constraints and develop scalable heuristics with strong performance.

1.6 Thesis outline

As shown in Figure 1.2, this thesis is organised into five chapters. Chapters 2, 3, and 4 are based on three original research papers that address related stochastic dynamic scheduling problems involving mobile servers in networked environments. Each chapter develops models and heuristics suited to the problem setting, contributing to a coherent framework for decision-making in complex service systems.

- **Chapter 1: Introduction**
  Introduces the motivation, problem overview, practical applications, the theoretical framework (MDPs, DP, ADP, RL, and index heuristics), and the thesis contributions.

- **Chapter 2: Single-server, infinite-state MDP with linear costs**
  Based on Paper 1, this chapter studies a single-server scheduling problem modeled as an infinite-state MDP with linear cost functions. It introduces K-stop and (K from L)-stop index heuristics that account for network structure to reduce scheduling costs.

- **Chapter 3: Single-server, finite-state MDP with general costs**
  Based on Paper 2, this chapter develops index heuristics for a finite-state maintenance scheduling MDP with general cost functions. Reinforcement learning methods improve these heuristics to better manage complex cost structures.

- **Chapter 4: Multi-server, infinite-state MDP with linear costs**
  Based on Paper 3, this chapter extends the single-server model to multiple servers. A gradient-based demand allocation balances workload and cost across servers, combined with local heuristics adapted from Chapter 2.

- **Chapter 5: Conclusions and future work**
  Summarizes contributions and outlines directions for further research.

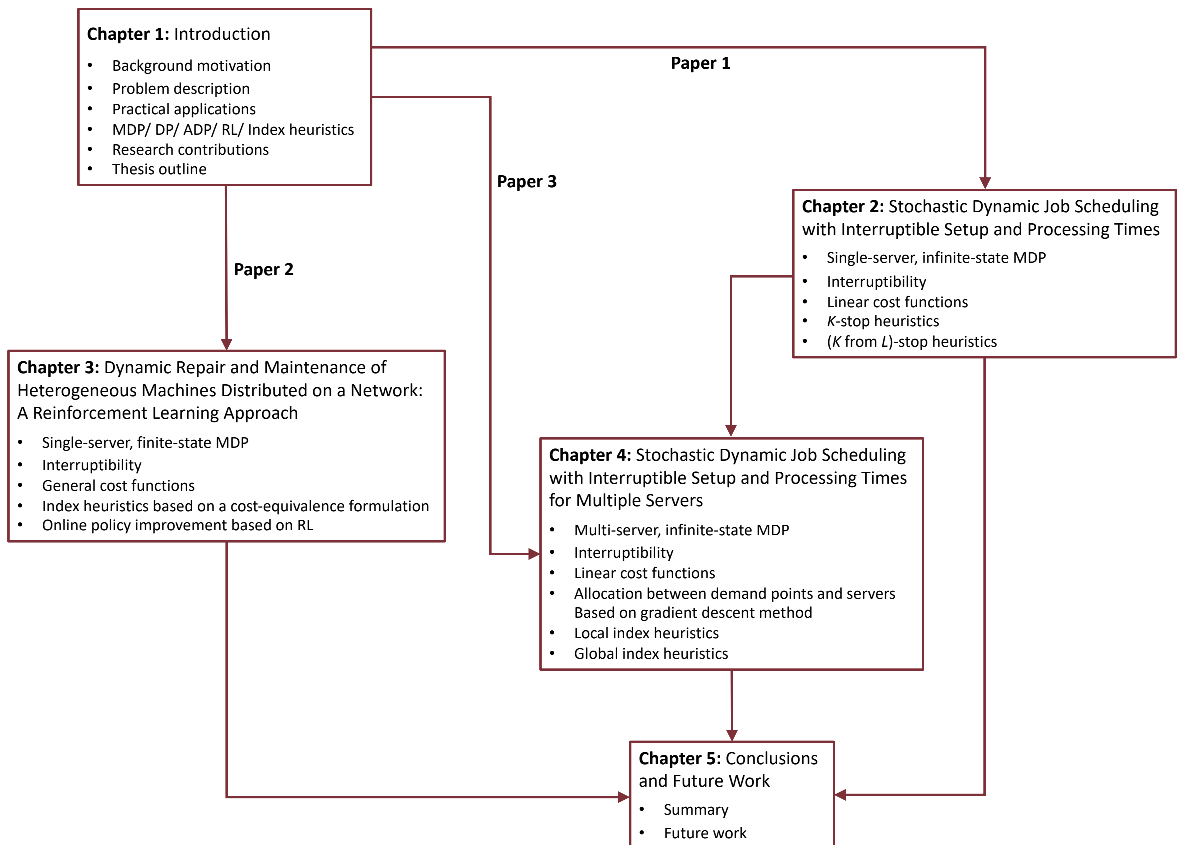

FIGURE 1.2: A structural overview of this thesis, outlining how the research is organised across chapters.

Chapter 2

Stochastic Dynamic Job Scheduling with Interruptible Setup and Processing Times

We consider a stochastic, dynamic job scheduling problem in which a single server processes jobs of different types that arrive according to independent Poisson processes. The problem is formulated 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

2.1 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 can be processed at a rate of μ_i, $i = 1, ..., N$, with a holding cost of c_i incurred per unit time until the processing of the task is complete. 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 generalised 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. Lee and Vojnovic (2021) established a learning-based variant of the $c\mu$-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. More recently, Long et al. (2024) proposed a generalized $c\mu$-rule that accounts for variations in service rates and costs across heterogeneous servers.

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. When a server moves from one node to another, the switching time is assumed to follow an exponential distribution with a constant rate. More generally, if the distribution of the switching time between any two types of jobs is known (estimated from historical operational data for example) then an appropriate number of intermediate stages between the corresponding demand points can be introduced in the network so as to match key moments, such as the mean and variance, of that distribution. We acknowledge that representing all job pairs simultaneously in this way may require edge-dependent switching rates. While such heterogeneity is not included in the current model for clarity and tractability, it can be incorporated without fundamental difficulty and represents a natural extension of the proposed network-based framework.

As an example of our network-based approach, consider the situation shown in Figure 2.1. 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 2.2.

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 2.1, 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

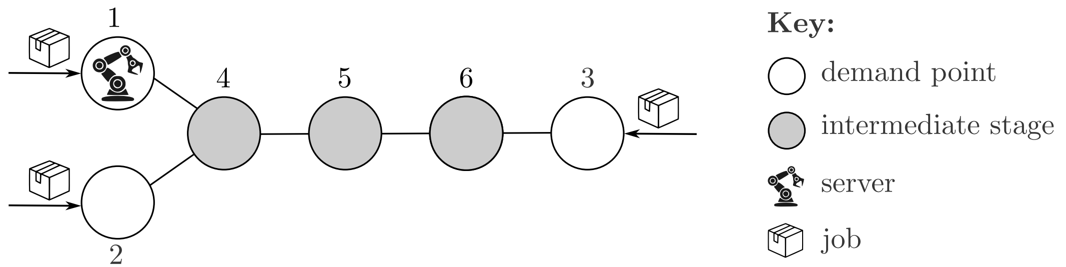

Figure 2.1: A network with 3 demand points, 3 intermediate stages and a single server. White-colored nodes represent demand points at which new jobs arrive, and gray-colored nodes represent intermediate stages.

in Section 2.2). 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 2.1, 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. Thus, we may regard jobs of type 3 as having quite unique setup requirements.

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, or a security problem

in which a defensive agent responds to threats that appear in a computer network. In reality, certain jobs may have emergency or high-priority status, which allows them to pre-empt lower-priority jobs that are currently being processed. While most service tasks are non-interruptible once started, the proposed generic decision framework can accommodate such situations, enabling a higher-priority job to interrupt an ongoing lower-priority job if necessary. Similar considerations apply to other potential applications of the framework.

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' (i.e., not moveable) in this setting, rather than being assigned dynamically to different tasks. Luo (2020) 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.

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 2.3). 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 2.2, 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, network-based formulation of a stochastic, dynamic job scheduling problem in which setup and processing times are 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. Moreover, we prove that these heuristics possess a pathwise consistency property which ensures that the server always proceeds to a demand point in finite time.

- 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 2.2 we formulate our stochastic, dynamic job scheduling problem as an MDP and prove an important stability property. In Section 2.3 we derive index heuristics and prove some useful properties of these heuristics, such as pathwise consistency. In Section 2.4 we present the results of our numerical study. Finally, our concluding remarks can be found in Section 2.5.

2.2 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 2.1). Let $N := V \setminus D$ denote the other nodes (the gray nodes in Figure 2.1), 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 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 while jobs are waiting or being 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 μ_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*. The parameters μ and τ are used to parametrise exponential distributions for the switching and processing operations (respectively). 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. A couple of further remarks should be made regarding the switching times in our model. Firstly, although the switching time between two adjacent nodes has the memoryless property, 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. Secondly, although we assume that the switching rate τ is the same between any two adjacent nodes, the results and index heuristics that we develop in this paper can be generalized without difficulty to the case of edge-dependent and also direction-dependent switching rates, in which case the time for the server to switch from one demand point to another belongs to the more general class of phase-type distributions. We use a constant switching rate τ only for ease of exposition. It is well-known that phase-type distributions can approximate any continuous distribution to an arbitrary degree of accuracy (Asmussen (2003)).

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. 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 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

$$\Delta := \left(\sum_{i\in D}\lambda_i + \max\{\mu_1, ..., \mu_d, \tau\}\right)^{-1}. \tag{2.1}$$

Let $R(\mathbf{x})$ be the set of nodes adjacent to node $v(\mathbf{x})$ 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

$$p_{\mathbf{x},\mathbf{y}}(a) := \begin{cases} \lambda_i\Delta, & \text{if } y_i = x_i + 1 \text{ and } y_j = x_j \text{ for } j \neq i, \\ \mu_i\Delta, & \text{if } y_i = x_i - 1,\ y_j = x_j \text{ for } j \neq i \text{ and } a = v(\mathbf{x}) = i, \\ \tau\Delta, & \text{if } y_j = x_j \text{ for all } j \in D,\ v(\mathbf{y}) \in R(\mathbf{x}) \text{ and } a = v(\mathbf{y}), \\ 0, & \text{otherwise,} \end{cases} \tag{2.2}$$

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. 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

$$f(\mathbf{x}) := \sum_{i\in D} c_i x_i. \tag{2.3}$$

Let θ denote a decision-making policy for the MDP. In a general sense, θ 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

(or *average cost* for short) under policy θ, given that the initial state of the system is $\mathbf{x}_0 \in S$, can be expressed as

$$g_\theta(\mathbf{x}) = \liminf_{t\to\infty} t^{-1}\mathbb{E}_\theta\left[\sum_{k=0}^{t-1} f(\mathbf{x}_k) \;\middle|\; \mathbf{x}_0 = \mathbf{x}\right] \tag{2.4}$$

where $\mathbf{x}_k$ denotes the state of the system at time step $k \in \mathbb{N}_0$. The theory of uniformization implies that under a stationary policy θ, 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. The objective of the problem is to minimize the long-run average cost $g_\theta(\mathbf{x})$; that is, to find a policy θ^* such that

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

where Θ is the set of all admissible policies.

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 θ, in the sense that θ induces an ergodic Markov chain on S. Let ρ 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 2.1. ***(Stability.)*** *Suppose $\rho < 1$. Then there exists a deterministic stationary policy θ 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.1. 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 θ 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 2.2. 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.

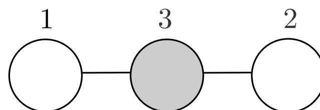

FIGURE 2.2: A network with 2 demand points and 1 intermediate stage.

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 θ^* 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 2.1 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 τ 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 λ_i/μ_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 2.1 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 2.3.

2.3 Index heuristics

As mentioned in Section 2.1, 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) 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. Specifically, let the reward function $r(\mathbf{x}, a)$ be defined for $\mathbf{x} = (v, (x_1, ..., x_d)) \in S$, $a \in A_{\mathbf{x}}$ as follows:

$$r(\mathbf{x}, a) = \begin{cases} c_i\mu_i, & \text{if } v = i \text{ for some } i \in D,\ x_i \geq 1 \text{ and } a = v, \\ 0, & \text{otherwise.} \end{cases} \tag{2.5}$$

It is shown in Duenyas and Van Oyen (1996) (using the results of Bell (1971), who studied a similar equivalence in the context of discounted cost problems) that the minimization of the long-run average cost (2.4) is equivalent to the maximization of a similar function based on long-run average rewards, in which the single-step cost $f(\mathbf{x})$ is replaced by the reward $r(\mathbf{x}, a)$ defined above. The use of the reward-based formulation (2.5) implies that the system earns rewards based only on the server's current location and choice of action. On the other hand, the cost formulation (2.3), 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 the reward formulation 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 formulation (2.5) rather than the cost formulation (2.3). However, in our numerical experiments in Section 2.4, we evaluate and compare the performances of different policies according to the original cost formulation presented in Section 2.2.

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 2.4.

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 A.2.

2.3.2 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:

$$T_j(\mathbf{x}, s, t) = \frac{x_{s_j} + \lambda_{s_j}\left[t + \sum_{k=1}^{j-1}\left(\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, t)\right) + \delta(s_{j-1}, s_j)/\tau\right]}{\mu_{s_j} - \lambda_{s_j}}, \quad t \geq 0, \tag{2.6}$$

$$R_j(\mathbf{x}, s, t) = c_{s_j}\mu_{s_j}T_j(\mathbf{x}, s, t), \qquad t \geq 0. \tag{2.7}$$

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 λ_i and are processed at a continuous rate μ_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 (2.6), 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 $\mu_{s_j} - \lambda_{s_j}$.

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. Waiting before moving can sometimes be beneficial since it can cause the average reward to increase due to the greater number of jobs processed by the server upon arrival. This is somewhat contrary to our MDP formulation in Section 2.2, 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

$$\psi(\mathbf{x}, s, t) = \frac{\sum_{k=1}^{|s|} R_k(\mathbf{x}, s, t)}{t + \sum_{k=1}^{|s|} [\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, t)]}, \quad t \geq 0, \tag{2.8}$$

$$\phi_j(\mathbf{x}, s, t) = \frac{\sum_{k=1}^{j} R_k(\mathbf{x}, s, t)}{t + \sum_{k=1}^{j} [\delta(s_{k-1}, s_k)/\tau + T_k(\mathbf{x}, s, t)] + \delta(s_j, v)/\tau}, \quad t \geq 0,\ j \in \{1, ..., |s|\}. \tag{2.9}$$

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. The reason for returning to the starting node v will be explained following the presentation of the steps of the K-stop heuristic algorithm. 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 (2.8) and (2.9) 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 2.2. *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 (2.6) are linear functions of t, for $j = 1, ..., |s|$. Hence, the average rewards $R_j(\mathbf{x}, s, t)$ in (2.7) are also linear in t. It follows that the quantity $\psi(\mathbf{x}, s, t)$ in (2.8) 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 an empty set $\sigma = \emptyset$.

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

$$\beta_j(\mathbf{x}, s, t) := \begin{cases} \dfrac{\sum_{k=1}^{j} R_k(\mathbf{x}, s, t)}{\sum_{k=1}^{j} T_k(\mathbf{x}, s, t)}\rho + c_v\mu_v(1-\rho), & \text{if } v \notin \{s_1, s_2, ..., s_j\}, \\ 0, & \text{otherwise.} \end{cases} \tag{2.10}$$

   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 σ. Otherwise, do not make any change to the set σ.

   (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 σ 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. 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 an empty demand point ($v \in D$ and $x_v = 0$) then we perform the following steps:

   (a) Let $\mathcal{S}$ be defined in the same way as in step 1(a). Initialize three empty sets: $\sigma_1 = \emptyset$, $\sigma_2 = \emptyset$ and $\sigma = \emptyset$.

   (b) For each sequence $s \in \mathcal{S}$, add s to σ_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

$$\gamma(\mathbf{x}, s, t) := \frac{\sum_{k=1}^{|s|} R_k(\mathbf{x}, s, t)}{\sum_{k=1}^{|s|} T_k(\mathbf{x}, s, t)}\rho, \quad t \geq 0. \tag{2.11}$$

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 σ_2 and add it to σ_1. Otherwise, do not make any changes.

(d) If σ_1 is non-empty, let $\sigma = \sigma_1$. Otherwise, let $\sigma = \sigma_2$.

(e) Carry out step 1(c).

3. If the server is at an intermediate stage ($v \notin D$) then carry out steps 2(a)-2(e).

The steps presented above require some explanation. Consider the first case, where the server is at a non-empty demand point. In this case, σ 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 2.2, 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 ρ 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 ρ. This is an important condition for system stability. On the other hand, if ρ 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 ε is some small positive number. This approach is justified by Lemma 2.2, 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 an empty demand point. In this case, by using two different sets σ_1 and σ_2, we effectively separate the eligible sequences into two different subsets, with sequences in σ_1 being given a higher priority for selection than those in σ_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 σ_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 σ_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 2.3, where it is shown that they are sufficient to ensure that the sequence s remains included in the set σ_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 σ_1, it may still be included in σ_2, in which case it may be selected in step 2(e) if σ_1 is empty. The conditions for inclusion in σ_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 is not necessarily helpful for maintaining stability, so following sequence s should not be seen as a worse option in this regard.

Finally, in case 3, we use the same rules as in case 2. This implies that (for the purposes of the K-stop heuristic) we treat intermediate stages as being similar to empty demand points. The server cannot earn any immediate reward by remaining at node v, so it uses the same conditions for switching used in case 2.

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 2.3. ***(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 2.3 can be found in Appendix A.3. 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 A.4.

Corollary 2.4. *Suppose the conditions of Theorem 2.3 apply. Then the expected amount of time until the server arrives at a demand point is finite. More specifically, if T_{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.

It should be noted that if $K \geq 2$ then the number of sequences in $\mathcal{S}$ increases exponentially 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.

2.3.3 (K from L)-stop heuristic

As explained in Section 2.3.2, 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 2.4 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 (2.11). 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 2.3.2. The sequences are required to satisfy the same eligibility conditions described in Section 2.3.2 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 2.3, 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.

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,

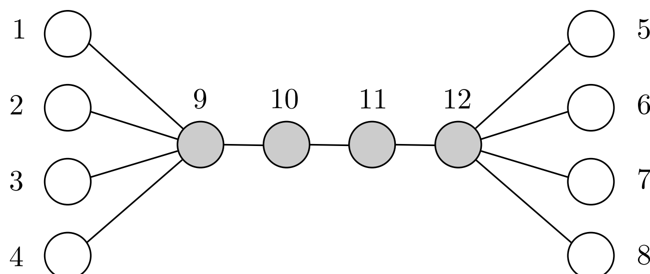

FIGURE 2.3: A network with 4 demand points on the left, 4 demand points on the right and 4 intermediate stages.

in the next section, we show that it may be able to achieve a similar performance at smaller computational expense.

2.4 Numerical results

In this section we report the results of numerical experiments in order to compare the performances of the heuristics described in Section 2.3. We consider a relatively simple network layout, shown in Figure 2.4, 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 $10,000$ 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 ρ, τ, λ_i, μ_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

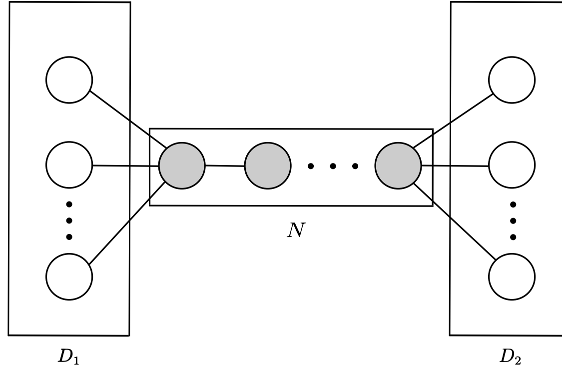

FIGURE 2.4: A diagrammatic representation of the network with d_1 demand points on the left, d_2 demand points on the right and n intermediate stages.

parameters. Two parameters of particular interest in our study are ρ, the overall traffic intensity rate for the system, and $\eta := \tau/(\sum_i \lambda_i)$, which represents the rate at which the server is able to switch between adjacent nodes relative to the total rate that jobs are arriving. Our sampling method works by generating the values of ρ and η first and then scaling the values of other parameters in order to conform to the required values of ρ and η. For full details, please refer to Appendix A.5.

In each problem instance, we test the performances of the heuristic policies in Section 2.3 using simulation experiments. The DVO heuristic described in Section 2.3.1 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 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. For implementation details of our simulation experiments, please refer to Appendix A.6.

2.4.1 General performance of the heuristics

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 2.1). In each problem instance we have used an iterative method, described in Appendix A.7, 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 ρ is not too large. In total, we have been able to compute g^* in 1229 of the 10,000 instances.

Table 2.1 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 2.1: The percentage suboptimalities of the DVO, 1-stop, 2-stop and 3-stop heuristics with respect to the optimal policy given by relative value iteration, computed across 1229 systems.

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

Table 2.2 shows a comparison between the heuristic policies across all 10,000 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 9% to 11%. 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 2.2: The percentage improvements of the 1-stop, 2-stop, (2 from 4)-stop, 3-stop and 4-stop heuristics with respect to the DVO heuristic, computed across 10,000 systems.

Heuristic	Mean	10th pct.	25th pct.	50th pct.	75th pct.	90th pct.
1-stop	9.48 ± 0.16	0.17	4.69	9.45	14.59	19.35
2-stop	10.50 ± 0.15	1.69	5.64	10.24	15.31	20.08
(2 from 4)-stop [imp.]	10.48 ± 0.15	1.68	5.63	10.24	15.31	20.05
(2 from 4)-stop [str.]	10.30 ± 0.15	1.42	5.43	10.08	15.24	19.97
3-stop	10.92 ± 0.14	2.25	6.05	10.63	15.73	20.23
4-stop	11.14 ± 0.14	2.76	6.41	10.85	15.79	20.29

2.4.2 Performance under specific parameters

Next, we investigate the effect of varying the number of intermediate stages n. Table 2.3 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). In order to explain the trends in Table 2.3 we will also refer to Table 2.4, which shows how often the server changes its direction of movement under the various policies. We can define a 'direction change' as follows: let ω_{switch} denote the total number of time steps at which the server chooses to switch to an adjacent node (rather than remaining at its current node) during a simulation run under a particular policy, and let ω_{change} denote the number of occasions during the simulation in which the server is located at the same intermediate stage for two consecutive time steps, but chooses two different adjacent nodes to switch to during these steps. Then the percentage of 'direction changes' in the simulation run is calculated as

$$100 \times \omega_{\text{change}}/\omega_{\text{switch}}.$$

Table 2.3: The mean percentage improvements (with 95% confidence intervals) 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) shown for different values of n over all 10,000 instances.

Heuristic	$n = 1$ [1671 instances]	$n = 2$ [1642]	$n = 3$ [1654]	$n = 4$ [1706]	$n = 5$ [1663]	$n = 6$ [1664]
1-stop (vs. DVO)	10.58 ± 0.50	10.71 ± 0.36	10.18 ± 0.36	9.14 ± 0.36	8.46 ± 0.38	7.86 ± 0.39
2-stop (vs. 1-stop)	1.56 ± 0.15	0.88 ± 0.10	0.80 ± 0.13	0.76 ± 0.14	1.04 ± 0.16	1.15 ± 0.19
(2 from 4)-stop [imp.] (vs. 1-stop)	1.45 ± 0.14	0.85 ± 0.10	0.86 ± 0.12	0.82 ± 0.13	1.05 ± 0.16	1.08 ± 0.19
(2 from 4)-stop [str.] (vs. 1-stop)	1.31 ± 0.13	0.78 ± 0.09	0.65 ± 0.12	0.56 ± 0.17	0.77 ± 0.19	0.88 ± 0.20
3-stop (vs. 1-stop)	2.09 ± 0.17	1.28 ± 0.11	1.28 ± 0.13	1.25 ± 0.15	1.38 ± 0.18	1.52 ± 0.20
4-stop (vs. 1-stop)	2.23 ± 0.19	1.51 ± 0.12	1.46 ± 0.14	1.44 ± 0.15	1.63 ± 0.18	1.86 ± 0.20

Table 2.4: The mean 'direction change' percentages (with 95% confidence intervals) for each of the K-stop and (K from L)-stop heuristics shown for different values of n over all 10,000 instances.

Heuristic	$n = 1$ [1671 instances]	$n = 2$ [1642]	$n = 3$ [1654]	$n = 4$ [1706]	$n = 5$ [1663]	$n = 6$ [1664]
1-stop	4.95 ± 0.10	4.24 ± 0.09	3.85 ± 0.09	3.53 ± 0.08	3.41 ± 0.07	3.25 ± 0.07
2-stop	5.00 ± 0.10	4.26 ± 0.08	3.84 ± 0.08	3.50 ± 0.07	3.34 ± 0.07	3.15 ± 0.07
(2 from 4)-stop [imp.]	5.03 ± 0.10	4.29 ± 0.08	3.85 ± 0.08	3.52 ± 0.07	3.33 ± 0.07	3.13 ± 0.07
(2 from 4)-stop [str.]	5.03 ± 0.10	4.29 ± 0.08	3.84 ± 0.08	3.51 ± 0.07	3.34 ± 0.07	3.12 ± 0.07
3-stop	5.03 ± 0.10	4.28 ± 0.08	3.85 ± 0.08	3.50 ± 0.07	3.32 ± 0.07	3.11 ± 0.07
4-stop	5.05 ± 0.10	4.30 ± 0.08	3.86 ± 0.08	3.50 ± 0.07	3.31 ± 0.07	3.10 ± 0.07

The first row of Table 2.3 appears to suggest a trend for the improvements given by the 1-stop policy (compared to the DVO heuristic) to decrease as n increases. We can also refer to Table 2.4, which shows that the percentage of 'direction changes' under the 1-stop policy tends 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 ρ, 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 single intermediate stage may be seen as a convenient location for the server, as from here it can switch to any of the demand points in a single transition. Suppose it chooses to switch from the intermediate stage to some demand point $i \in D$ at one time step, but the switch is unsuccessful and it then chooses to switch to a different demand point $j \neq i$ at the next time step. This change of direction is essentially 'costless', as the distance to node j has not increased as a result of choosing to move towards i on the previous step. On the other hand, in the $n = 6$ case, the server must pass through a series of intermediate stages in order to move from D_1 to D_2 or vice versa. If it takes a step towards D_2, then it may be incentivized to continue moving towards D_2 rather than reversing direction and moving back towards D_1 (as this would require some additional switching time). To summarize, smaller values of n enable the server to change direction in a more costless manner while it is located at the intermediate stages, and this results in larger improvements for the 1-stop policy compared to the DVO heuristic (under which the server never changes direction).

The remaining rows in Table 2.4 show that the other K-stop and (K from L)-stop heuristics also tend to change direction less frequently as n increases. However, the corresponding rows in Table 2.3 appear to show a more interesting pattern. Recall that in these rows, we are comparing the $K \geq 2$ policies with the $K = 1$ policy in order to show the benefits of considering longer sequences and therefore taking the network topology into account. These benefits appear to decrease as n increases from 1 to 4, but then increase again as n increases beyond 4. To make sense of this, we observe that smaller values of n can actually work to the advantage of longer-sequence policies, because in these systems the average number of jobs in the system tends to be smaller and the server does not need to spend as long at any particular demand point in order to serve it exhaustively. This means that the longer sequences selected by the heuristics are more likely to be followed through to completion, whereas when n becomes larger, it becomes unlikely that the sequences selected will be followed through to completion (as this requires a greater commitment of time, during which the server may react

dynamically to the latest stochastic system events). On the other hand, larger values of n also offer advantages for the longer-sequence policies, because the 'clustering' of the network takes a greater effect in these situations. The longer-sequence policies (unlike the 1-stop policy) are able to assess the benefits of moving to specific demand points by taking into account their proximities to other demand points that might be visited next, and this becomes more important as the distance between the clusters increases.

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.

Next, we investigate the effect of varying the switching rate parameter τ. Let $\eta := \tau/(\sum_i \lambda_i)$ denote the ratio of the switching rate to the sum of the job arrival rates. Thus, larger values of η imply that setup times are relatively short compared to the times in between new job arrivals, whilst smaller values of η imply the opposite. Our method for randomly generating the system parameter values (detailed in Appendix A.5) implies that, in any problem instance, η is equally likely to fall within any of the intervals $[0.1, 0.4)$, $[0.4, 0.7)$, $[0.7, 1)$, $[1, 4)$, $[4, 7)$ and $[7, 10)$. These intervals are shown as columns in Tables 2.5 and 2.6, and the rows in these tables are presented in the same format as Tables 2.3 and 2.4.

Table 2.5: The mean percentage improvements (with 95% confidence intervals) 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) shown for different values of η over all 10,000 instances.

Heuristic	$0.1 \leq \eta < 0.4$ [1628 instances]	$0.4 \leq \eta < 0.7$ [1688]	$0.7 \leq \eta < 1$ [1599]	$1 \leq \eta < 4$ [1737]	$4 \leq \eta < 7$ [1680]	$7 \leq \eta < 10$ [1668]
1-stop (vs. DVO)	7.52 ± 0.47	9.01 ± 0.42	9.67 ± 0.39	11.58 ± 0.38	9.96 ± 0.36	9.03 ± 0.33
2-stop (vs. 1-stop)	1.56 ± 0.20	1.43 ± 0.17	1.39 ± 0.16	0.99 ± 0.12	0.53 ± 0.10	0.32 ± 0.08
(2 from 4)-stop [imp.] (vs. 1-stop)	1.53 ± 0.19	1.44 ± 0.16	1.41 ± 0.15	0.99 ± 0.12	0.48 ± 0.10	0.27 ± 0.10
(2 from 4)-stop [str.] (vs. 1-stop)	1.21 ± 0.22	1.17 ± 0.17	1.12 ± 0.17	0.77 ± 0.15	0.44 ± 0.10	0.27 ± 0.08
3-stop (vs. 1-stop)	2.36 ± 0.21	2.02 ± 0.19	1.94 ± 0.17	1.28 ± 0.13	0.76 ± 0.10	0.49 ± 0.09
4-stop (vs. 1-stop)	2.83 ± 0.22	2.38 ± 0.19	2.22 ± 0.18	1.43 ± 0.13	0.86 ± 0.10	0.47 ± 0.10

By looking at the first rows of Tables 2.5 and 2.6, we are able to observe a clear correlation between the mean percentage improvement of the 1-stop policy (vs. the DVO heuristic) and the percentage of direction changes under the 1-stop policy. Both of these attain

TABLE 2.6: The mean 'direction change' percentages (with 95% confidence intervals) for each of the K-stop and (K from L)-stop heuristics shown for different values of η over all 10,000 instances.

Heuristic	$0.1 \le \eta < 0.4$ [1628 instances]	$0.4 \le \eta < 0.7$ [1688]	$0.7 \le \eta < 1$ [1599]	$1 \le \eta < 4$ [1737]	$4 \le \eta < 7$ [1680]	$7 \le \eta < 10$ [1668]
1-stop	2.83 ± 0.08	3.22 ± 0.08	3.58 ± 0.10	4.81 ± 0.10	4.76 ± 0.07	3.96 ± 0.04
2-stop	2.76 ± 0.08	3.16 ± 0.08	3.52 ± 0.09	4.81 ± 0.10	4.78 ± 0.07	3.97 ± 0.04
(2 from 4)-stop [imp.]	2.73 ± 0.08	3.15 ± 0.08	3.52 ± 0.09	4.87 ± 0.10	4.83 ± 0.06	3.98 ± 0.04
(2 from 4)-stop [str.]	2.71 ± 0.08	3.13 ± 0.08	3.50 ± 0.09	4.86 ± 0.10	4.83 ± 0.06	3.98 ± 0.04
3-stop	2.75 ± 0.08	3.17 ± 0.08	3.53 ± 0.09	4.81 ± 0.10	4.77 ± 0.06	3.96 ± 0.04
4-stop	2.75 ± 0.08	3.18 ± 0.08	3.55 ± 0.09	4.82 ± 0.10	4.77 ± 0.06	3.96 ± 0.04

their highest values when η falls within the interval $[1, 4)$. Thus, it seems that the 1-stop policy performs better (relative to the DVO heuristic) when it perceives an advantage in changing direction more often. It is also interesting to note that the improvements (and also the direction change percentages) tend to decrease as η becomes small, and also as η becomes large. Indeed, when η 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 η is large, switching between nodes is relatively fast and in this situation it becomes less likely that any arrivals will occur while the server is traversing the intermediate stages, so the server has no reason to change its course. Indeed, the latter point is consistent with Theorem 2.3, which states that the server proceeds along the shortest path to a particular demand point as long as the number of jobs in the system remains unchanged.

By examining the remaining rows in Table 2.5, we observe that the K-stop and (K from L)-stop heuristics (for $K \ge 2$) are able to achieve greater improvements over the 1-stop policy when η 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 η 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 is more difficult for the longer-sighted heuristics to achieve improvements over the 1-stop policy in this situation. On the other hand, when η is small, choosing to move from one demand point to another may be seen as a greater commitment, as switching requires a greater investment of time. Thus, the longer-sighted heuristics are able to outperform the 1-stop policy by planning to follow sequences that allow efficient movement between the demand points in the network.

Finally, we investigate the effects of varying the traffic intensity, $\rho = \sum_i \lambda_i/\mu_i$. Our method for randomly generating the system parameter values (Appendix A.5) implies

that, in any problem instance, ρ is equally likely to fall within any of the 4 intervals shown as columns in Tables 2.7 and 2.8. Once again, we present these tables in the same format as Tables 2.5 and 2.6. Thus, Table 2.7 shows how the relative performances of the heuristics vary with ρ, while Table 2.8 shows how the 'direction change' percentages vary with ρ.

TABLE 2.7: The mean percentage improvements (with 95% confidence intervals) 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) shown for different values of ρ over all 10,000 instances.

Heuristic	$0.1 \le \rho < 0.3$ [2448 instances]	$0.3 \le \rho < 0.5$ [2514]	$0.5 \le \rho < 0.7$ [2498]	$0.7 \le \rho < 0.9$ [2540]
1-stop (vs. DVO)	16.64 ± 0.25	11.26 ± 0.25	7.00 ± 0.27	3.27 ± 0.26
2-stop (vs. 1-stop)	0.55 ± 0.09	0.79 ± 0.11	1.37 ± 0.13	1.40 ± 0.13
(2 from 4)-stop [imp.] (vs. 1-stop)	0.57 ± 0.09	0.83 ± 0.11	1.33 ± 0.13	1.33 ± 0.13
(2 from 4)-stop [str.] (vs. 1-stop)	0.53 ± 0.09	0.70 ± 0.11	1.17 ± 0.13	0.90 ± 0.17
3-stop (vs. 1-stop)	0.87 ± 0.09	1.26 ± 0.12	1.84 ± 0.15	1.88 ± 0.15
4-stop (vs. 1-stop)	1.05 ± 0.10	1.45 ± 0.13	2.09 ± 0.15	2.14 ± 0.15

TABLE 2.8: The mean 'direction change' percentages (with 95% confidence intervals) for each of the K-stop and (K from L)-stop heuristics shown for different values of ρ over all 10,000 instances.

Heuristic	$0.1 \le \rho < 0.3$ [2448 instances]	$0.3 \le \rho < 0.5$ [2514]	$0.5 \le \rho < 0.7$ [2498]	$0.7 \le \rho < 0.9$ [2540]
1-stop	2.96 ± 0.06	4.10 ± 0.07	4.36 ± 0.07	4.03 ± 0.07
2-stop	2.95 ± 0.06	4.08 ± 0.07	4.33 ± 0.07	4.00 ± 0.06
(2 from 4)-stop [imp.]	2.92 ± 0.06	4.10 ± 0.07	4.36 ± 0.07	4.03 ± 0.06
(2 from 4)-stop [str.]	2.92 ± 0.06	4.09 ± 0.07	4.35 ± 0.07	4.01 ± 0.06
3-stop	2.95 ± 0.06	4.08 ± 0.07	4.34 ± 0.07	4.00 ± 0.06
4-stop	2.95 ± 0.06	4.08 ± 0.07	4.35 ± 0.07	4.02 ± 0.06

By examining the first row in Table 2.7, we observe that the improvements given by the 1-stop policy (vs. the DVO heuristic) are much greater when ρ is small than when it is large. It is useful to recall that, under any stable policy, ρ corresponds to the proportion of time spent processing jobs. If ρ 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 ρ 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. In such situations, although Table 2.8 suggests that the 1-stop policy may change direction more often (due to the frequent arrivals of new jobs while the server is switching), these changes of direction are less effective in yielding improvements over the DVO heuristic. 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 ρ is large), whereas it does not have the ability to make new decisions while the server is switching, so in this sense larger ρ

values work in its favor. Therefore, it is more difficult for the 1-stop policy to gain an advantage when ρ is large.

The remaining rows in Table 2.7 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 increase as ρ increases. This makes sense intuitively, as the longer-sighted heuristics are able to assess the congestion levels at multiple demand points when choosing a sequence to follow. In heavy traffic systems, it is more important to recognize situations where switching to a different part of the network allows the server to process jobs at a number of demand points in succession, and these longer sequences are more likely to satisfy the system stability conditions described in the steps of the K-stop algorithm (see Section 2.3.2) because the amount of time spent processing jobs is sufficient to counterbalance the amount of time spent switching between nodes.

2.4.3 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 2.3.3.

In Table 2.9 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 2.3.2. 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 2.9: 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	7.19×10^{-5}	1.72×10^{-6}	7.21×10^{-6}	3.17×10^{-6}	6.36×10^{-5}	4.61×10^{-4}

2.5 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 2.2, 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 2.3 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. Although these heuristics are not provably guaranteed to yield stable policies, we have not encountered any unstable instances in our numerical experiments.

The numerical results in Section 2.4 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.

Chapter 3

Dynamic Repair and Maintenance of Heterogeneous Machines Distributed on a Network: A Reinforcement Learning Approach

We consider a problem in which a single repairer is responsible for the maintenance and repair of a collection of machines, positioned at different locations on a network of nodes and edges. Machines deteriorate according to stochastic processes and incur increasing costs as they approach complete failure. The times needed for repairs to be performed, and the amounts of time needed for the repairer to switch between different machines, are random and machine-dependent. The problem is formulated as a Markov decision process (MDP) in which the objective is to minimize long-run average costs. We prove the equivalence of an alternative formulation based on rewards and use this to develop an index heuristic policy, which is shown to be optimal in certain special cases. We then use reinforcement learning techniques to develop a novel approximate policy improvement (API) approach, which uses the index heuristic as a base policy and also as an insurance option at decision epochs where the best action cannot be selected with sufficient confidence. Results from extensive numerical experiments, involving randomly-generated network layouts and parameter values, show that the API heuristic is able

to achieve close-to-optimal performance in fast-changing systems with state transitions occurring 100 times per second, suggesting that it is suitable for online implementation.

Keywords: Repair and maintenance, policy improvement, reinforcement learning, dynamic programming

3.1 Introduction

A classical problem in operations research concerns the dynamic scheduling of maintenance and repair operations for a collection of machines that are prone to failure. These problems fall within the broader framework of reliability modeling, which is concerned with the performance of systems that degrade or fail over time due to random influences. Related problems have been studied extensively due to their numerous applications in computer systems, telecommunications, transportation infrastructure, manufacturing systems, and other areas (see, e.g., Seyedshohadaie et al. (2010); Ahmad et al. (2017); Poonia (2021); Geurtsen et al. (2023)). In the context of machine failures, it is often assumed that each machine has a finite number of possible states, ranging from the pristine or 'as-good-as-new' state (in which it functions at its optimal performance level) to the 'failed' state (in which it performs at its worst level, or does not function at all). If a machine is left unattended, a *failure process* causes it to pass through a sequence of increasingly undesirable states until it eventually reaches the failed state. The times of state transitions are random, so that the speed of the failure process is unpredictable. The system controller has a set of resources available (in the form of repair engineers, for example) to stall or reverse the failure process by effecting transitions in the opposite direction, i.e. towards the pristine state, and must allocate these resources efficiently over time in order to optimize a pre-specified measure of performance. These types of model assumptions, which combine stochastic degradation with resource-constrained control, have been extensively studied in the literature (see, e.g., Nino-Mora (2001); Glazebrook et al. (2005); Abbou and Makis (2019)).

The dynamic maintenance and repair problem can be extended and generalized in various ways. For example, one can assume that different kinds of maintenance and repair operations (known as 'rate-modifying activities') are available (Lee and Lin (2001); Woo et al. (2017); Mosheiov and Oron (2021)), or allow machine failures to be governed by multiple dependent failure processes (Yousefi et al. (2020); Ogunfowora and Najjaran

(2023); Ward et al. (2024). In this paper we consider a version of the problem that, to the best of our knowledge, is original and enables insightful analyses based on a network model. We assume that there is a single repair operative on duty (referred to throughout the paper as a 'repairer') who bears sole responsibility for the maintenance and repair of $m \geq 2$ machines. Although the repairer can switch from repairing one machine to another, this cannot be done instantaneously, and the decision-making process should therefore take into account the time required to re-allocate effort from one machine to another in addition to the time needed to perform repairs. Indeed, the machines might be in different physical locations, in which case some travel time would be needed to switch from one machine to another. Alternatively, even if travel times are negligible, there might be setup requirements for repairing individual machines, so that the repairer must expend some time to perform the necessary preliminary tasks before commencing the next repair. We refer to the time needed to re-allocate effort from one machine to another as a 'switching time', and introduce further complexity by allowing these switching times to be interruptible, so that the repairer may begin the process of switching to one machine but then take another action (such as changing course and switching to a different machine) before completing the switch.

To motivate our network-based model, Figure 3.1 shows an example with 4 machines ($m = 4$). The machines are represented as gray-colored nodes in the network, labeled 1, 2, 3 and 4. At any given time, the repairer occupies one node and can choose either to remain at the current node or attempt to switch to one of the adjacent nodes (connected by edges to the current node). In order to repair a particular machine, the repairer must occupy the corresponding node. Thus, in order to switch from repairing one machine to another, it must traverse an appropriate path through the network. The white-colored nodes represent intermediate 'stages' that must be completed during a switch. For example, to switch from machine 1 to 3, the repairer would need to pass through nodes 5, 6 and 7. However, the direction of movement can be changed at any time, so the repairer might follow a path such as $(1, 5, 6, 5, 2)$. This would represent a switch from machine 1 to 3 that gets 'interrupted' in favor of a switch to machine 2 instead. By designing the network appropriately, we can model intricate relationships between the travel times or setup requirements of different machines, including the effects on switching times when the repairer makes partial progress in switching from one machine to another. In Section 3.2 we provide details of how the system can be modeled as a Markov decision process (MDP).

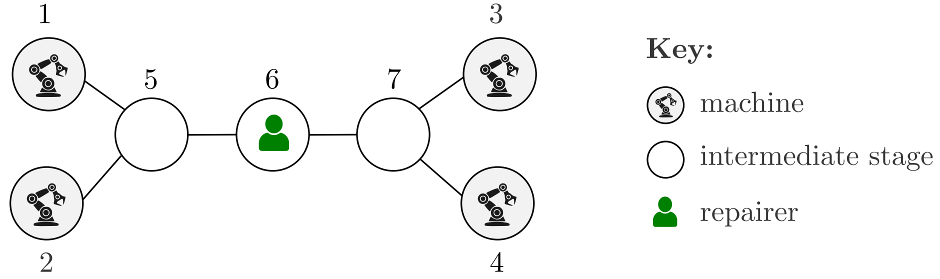

FIGURE 3.1: A network with 4 machines, 3 intermediate stages and a single repairer. Gray-colored nodes represent machines, and white-colored nodes represent intermediate stages.

Several previous studies have modeled repair and maintenance problems using restless bandit formulations, based on the principle that only a certain subset of machines can be attended to at any given time, and those that are unattended are liable to change state (i.e. degrade) rather than remain in the same condition. Glazebrook et al. (2005) considered a discrete-time formulation with multiple repairers and used the Whittle's index method (see Whittle (1988)) to derive index heuristics in order to minimize expected discounted costs. Abbou and Makis (2019) also used Whittle indices to minimize discounted costs but proposed a group maintenance formulation, in which each 'bandit' is a production facility consisting of several machines with different possible states. Ruiz-Hernandez et al. (2020) considered a scenario with *imperfect* maintenance, in which repairs return a machine to an uncertain state of wear rather than the pristine state, and also used Whittle indices with discounted costs. Ayesta et al. (2021) developed computational methods based on general expressions for Whittle indices in Markovian bandit models and, as a special case, considered a machine repairman problem in continuous time under the average cost criterion. The restless bandits approach has some advantages in that, by considering a decomposition of the problem based on Lagrangian relaxation, one can easily extend the analyses to the case of multiple repairers. On the other hand, the aforementioned studies do not consider switching times (interruptible or otherwise) and indeed bandit-type models become much less tractable in general when the assumption of immediate, costless switching is removed.

One study of particular relevance to ours is that of Duenyas and Van Oyen (1996), which considers the dynamic assignment of a single server in a system of parallel, heterogeneous queues. Although their model is presented in a different context from ours (i.e. servicing of customers in a queueing system rather than repair of machines), there are similarities in its mathematical formulation, including the assumption of Poisson customer arrivals (analogous to machine degradations in our model) and the incorporation of random

switching times. They also consider an infinite state space and allow service times to have arbitrary distributions, whereas our model relies on Markovian distributions in order to enable an MDP formulation. However, the switching and service times in Duenyas and Van Oyen (1996) are non-interruptible, and they also assume that the distribution of the switching time only depends on the queue being 'switched to', implying that there is no need to consider a network formulation.

In a concurrent study (Tian and Shone (2026b)), we consider an extension of the job scheduling problem in Duenyas and Van Oyen (1996) that allows the server to interrupt processing and switching times and also uses a network formulation to encode the amounts of effort needed to switch between different tasks. The formulation in Tian and Shone (2026b) has similarities to the one that we consider in the current paper, but there are two main differences. Firstly, in Tian and Shone (2026b) it is assumed that there can be any number of jobs waiting to be processed at a demand point and this leads to the formulation of an MDP with an infinite state space, in which system stability is an important consideration. In this paper we obtain a finite-state MDP because the set of possible conditions for any machine i is finite. While this removes the need to consider questions involving stability, it introduces other complications related to multichain policies and irregular value functions that we elaborate on in Sections 3.3 and 3.4. Secondly, in Tian and Shone (2026b) it is assumed that the cost of having k jobs waiting at a particular demand point is linear in k, but in this paper we allow the cost function $f(k)$ to be more general, as the effect on a machine's performance may be non-linear in the number of degradation events that has occurred. We also note that the solution methods in Tian and Shone (2026b) are focused on index policies, whereas in this paper we begin by developing an index policy but subsequently use an altered version of this as a base policy in a novel online policy improvement algorithm.

The main contributions of our paper are as follows:

- We consider an original model of a repair and maintenance problem in which repair and switching times not only follow random distributions but are also interruptible. In addition, our proposed network formulation allows switching times to depend not only on the machines involved in the switch, but also on the amount of progress made during any switches that have been interrupted.
- We develop an index-based heuristic for choosing which machine to switch to next, with decisions being continuously re-calculated during repair and switching times

in order to allow interruptions. In order to develop the heuristic, we first prove an equivalence between two cost formulations in the underlying MDP formulation. We then prove that the index heuristic is optimal in certain special cases of our problem.

- In order to obtain a stronger-performing heuristic policy we apply reinforcement learning techniques and approximate the value function for the index heuristic. We then use one-step policy improvement to search for improved decisions during online implementation. Our online policy improvement approach includes a novel safety mechanism, based on reliability weights, which mandates that the decision given by the index policy is followed in situations where the best action cannot be identified with sufficient confidence.

- We present the results of computational experiments with randomly-generated network layouts and parameter values in order to test the performances of our heuristics and investigate the effects of adjusting key parameter values.

Section 3.2 provides details of our mathematical formulation. Section 3.3 introduces our index heuristic and also includes proofs of its optimality in certain special cases. Section 3.4 explains the use of online policy improvement for obtaining a stronger-performing heuristic policy. Section 3.5 provides details of results from our numerical experiments. Finally, our concluding remarks are given in Section 3.6.

3.2 Problem formulation

Let $G = (V, E)$ be a connected graph, referred to as a *network*, where V and E are the sets of nodes and edges respectively. Let $M \subseteq V$ denote a subset of nodes referred to as *machines*, and let $N := V \setminus M$ denote the set of other nodes, referred to as *intermediate stages*. We use $m := |M|$ (resp. $n := |N|$) to denote the number of machines (resp. intermediate stages). As a convention we label the machines $1, 2, ..., m$, while the stages in N are labeled $m + 1, m + 2, ..., m + n$.

At any given time, machine $i \in M$ is in state $x_i \in \{0, 1, ..., K_i\}$, where 0 is the pristine or 'as-good-as-new' state and K_i is the 'failed' state. When machine i is in state $x_i \leq K_i - 1$, transitions to state $x_i + 1$ occur at an exponential rate $\lambda_i > 0$; this is the *degradation*

rate for machine i. All machines in M are assumed to degrade independently of each other. In addition, a cost $f_i(x_i)$ is incurred per unit time while machine i is in state x_i. We assume that the functions f_i are strictly increasing and that $f_i(0) = 0$ for each $i \in M$.

There is a single repairer who occupies a single node in V at any given time. At each point in time, the repairer can decide either to remain at the currently-occupied node or attempt to move to one of the adjacent nodes (connected by edges to the current node). In the latter case, it must also choose which node to move to. Therefore, if the number of adjacent nodes is l then the number of possible decisions for the repairer is $l+1$. If it remains at a node $i \in M$ (i.e. at a machine) and $x_i \geq 1$ then it is assumed to be *repairing* the machine, in which case the state of the machine transitions from x_i to $x_i - 1$ at a rate $\mu_i > 0$, where μ_i is the *repair rate* for machine i. We note that degradations can still happen at a machine while it's being repaired, so that $\lambda_i + \mu_i$ is the total transition rate for a non-failed machine i while it is under repair. If the repairer remains at a node i with $x_i = 0$, then the repairer is said to be *idle*. Similarly, idleness can also occur if the repairer chooses to remain at some stage $j \in N$ rather than attempting to move. On the other hand, if the repairer chooses to move to an adjacent node, then it moves to that node at an exponential rate $\tau > 0$, referred to as the *switching rate*. Repair times and switching times are assumed to be independent of the degradation processes for machines in M. We assume that the switching rate is the same between any adjacent pair of nodes, so that the expected time to move from one node to another is proportional to the number of edges that must be traversed. This implies that the amount of time to switch from one machine to another is distributed as a sum of i.i.d. exponential random variables and therefore follows an Erlang distribution.

The above assumptions allow the system to be formulated as a finite-state, continuous-time Markov decision process (MDP). Herein, we will follow similar notational conventions to those in Tian and Shone (2026b). Let S be the state space, given by

$$S := \{(i, (x_1, ..., x_m)) \mid i \in V,\ x_i \in \{0, 1, ..., K_i\} \text{ for } i \in M\},$$

where i indicates the node currently occupied by the repairer. As a notational convention, we will tend to use i for the repairer's current location throughout this paper, while j will be used more generally for nodes in V. Also, k will be used to represent the condition of an individual machine, which we will sometimes refer to as the state of the machine (not to be confused with the system state). Given that the repairer cannot be repairing and switching at the same time (and also cannot be carrying out

more than one repair), the sum of the transition rates under any state cannot exceed $\sum_{j\in M}\lambda_j+\max\{\mu_1,...,\mu_m,\tau\}$ and we can therefore use the technique of uniformization (Lippman (1975); Serfozo (1979)) to consider an equivalent discrete-time formulation which evolves in time steps of size $\Delta:=\left(\sum_{j\in M}\lambda_j+\max\{\mu_1,...,\mu_m,\tau\}\right)^{-1}$.

For each state $\mathbf{x}=(i,(x_1,...,x_m))\in S$, let $R(\mathbf{x})$ denote the set of nodes adjacent to node i and let $A_{\mathbf{x}}=\{i\}\cup R(\mathbf{x})$ denote the set of actions available under state $\mathbf{x}$. We can interpret action $a\in A_{\mathbf{x}}$ as the node that the repairer attempts to move to next, given that it is currently in state $\mathbf{x}$ (with $a=i$ indicating that the repairer remains at the current node). Then, for two distinct states $\mathbf{x}:=(i,(x_1,...,x_m))$ and $\mathbf{x}':=(i',(x_1',...,x_m'))$, the transition probability of moving from state $\mathbf{x}$ to $\mathbf{x}'$ in the uniformized MDP can be expressed as

$$p_{\mathbf{x},\mathbf{x}'}(a):=\begin{cases}\lambda_j\Delta, & \text{if } i'=i,\ x_j'=x_j+1 \text{ for some } j\in M \text{ and } x_l'=x_l \text{ for } l\neq j,\\ \mu_i\Delta, & \text{if } i'=i,\ x_i'=x_i-1,\ x_j'=x_j \text{ for } j\neq i \text{ and } a=i,\\ \tau\Delta, & \text{if } i'\in R(\mathbf{x}),\ x_j'=x_j \text{ for all } j\in M \text{ and } a=i',\\ 0, & \text{otherwise.}\end{cases} \quad (3.1)$$

The first line in (3.1) corresponds to a degradation event at machine j, the second line corresponds to a completed (partial) repair at the repairer's current node i and the third line represents switching from node i to i'. We also have $p_{\mathbf{x},\mathbf{x}}(a)=1-\sum_{\mathbf{y}\neq\mathbf{x}}p_{\mathbf{x},\mathbf{y}}(a)$ as the probability of remaining in the same state. Since the units of time are arbitrary, in the rest of the paper we will assume $\Delta=1$ without loss of generality. This implies that we can use quantities such as λ_j, μ_j, τ to represent transition probabilities in the uniformized MDP.

For each time step that the system spends in state $\mathbf{x}$ a cost $c(\mathbf{x})$ is incurred, given by

$$c(\mathbf{x})=\sum_{i\in M}f_i(x_i). \quad (3.2)$$

Let θ denote the decision-making policy to be followed, i.e. the method of choosing actions in our MDP. If the same action $\theta(\mathbf{x})\in A_{\mathbf{x}}$ is chosen every time the system is in state $\mathbf{x}\in S$, then the policy is referred to as *stationary* and *deterministic* (Puterman (1994)). The expected long-run average cost (or *average cost* for short) under policy θ,

given that the system begins in state $\mathbf{x}_0 \in S$, is

$$g_\theta(\mathbf{x}) = \liminf_{t\to\infty} t^{-1}\mathbb{E}_\theta\left[\sum_{n=0}^{t-1} c(\mathbf{x}_n) \;\middle|\; \mathbf{x}_0 = \mathbf{x}\right], \tag{3.3}$$

where $\mathbf{x}_n$ is the system state at time step $n \in \mathbb{N}_0$. The theory of uniformization implies that if θ is a stationary policy then the average cost $g_\theta(\mathbf{x})$ in the discretized system is equal to the average cost per unit time incurred under the same policy in the continuous-time system (in which we suppose that actions are chosen at the beginning of each new sojourn, i.e. every time the system transitions from one state to another). The objective of the problem is to find a policy θ^* such that

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

where Θ is the set of all admissible policies. In other words, we aim to find a policy that minimizes the average cost.

Since S is finite, the question of system stability does not arise. However, our MDP formulation does have some particular subtleties. For one thing, we cannot claim that the MDP is *unichain* (a common property of MDPs based on queueing systems, for example), since one can easily define a stationary policy θ under which the resulting Markov chain has more than one positive recurrent class. The simplest way to do this is to define the actions $\theta(\mathbf{x})$ in such a way that the repairer always remains at whichever node it currently occupies. This policy allows all machines except one (if the machine is initially located at a node in M) to remain permanently in their failed states. The average cost under such a policy is finite, but clearly depends on the initial state. However, we can show that our MDP belongs to the 'communicating' class of multichain MDP models, as discussed in Puterman (1994) (p. 348), from which it follows that the average cost under an optimal policy must be independent of the initial state. We state this formally below and provide a proof in Appendix B.1.

Proposition 3.1. *Let θ^* be an optimal policy. Then the average cost $g_{\theta^*}(\mathbf{x})$ is independent of the initial state $\mathbf{x}$, and can be expressed as a constant g^*.*

The theory in Puterman (1994) then implies that there exists a function $h : S \to \mathbb{R}$ satisfying the optimality equations

$$g^* + h(\mathbf{x}) = c(\mathbf{x}) + \min_{a\in A_{\mathbf{x}}}\left\{\sum_{\mathbf{y}\in S} p_{\mathbf{x},\mathbf{y}}(a)h(\mathbf{y})\right\} \qquad \forall \mathbf{x} \in S. \tag{3.4}$$

In theory, we can use dynamic programming (DP) techniques such as value iteration and policy improvement to compute a constant g^* and function h satisfying the optimality equations. These algorithms are made more complicated by the existence of policies with multichain structure, but the 'communicating' property ensures their theoretical feasibility (see Puterman (1994), pp. 478-484). Although DP algorithms are useful for finding optimal solutions in small problem instances, they become computationally intractable as S increases in size. We therefore propose the use of index heuristics and reinforcement learning in Sections 3.3 and 3.4 respectively in order to find strong-performing policies in systems with large state spaces.

Before closing this section, we present a short example to show that the finite-state nature of our problem prevents direct application of certain results that have been proved for infinite-state problems with similar formulations. As mentioned in our introduction, Duenyas and Van Oyen (1996) considers an infinite-state problem in which a server must be allocated dynamically between different queues. Their formulation is different from ours in several respects; for example, it assumes that switching and service times are non-interruptible. They also assume linear holding costs and, under this assumption, are able to show that different queues can be assigned priorities based on the values of $c_i\mu_i$, where c_i is the linear holding cost for customers at queue i and μ_i is the service rate. More specifically, they show that any queue i belonging to the set $\arg\max_i\{c_i\mu_i\}$ can be defined as a 'top-priority' queue and, under an optimal policy, the server should never switch away from such a queue if it is non-empty. The next example, which considers the case of linear holding costs, shows that the analogous property does not hold in our system.

Example 3.1. *Consider a network consisting of only 2 nodes connected by a single edge. Both nodes are machines; that is,* $M = \{1, 2\}$*. We assume that* $\lambda_1 = \lambda_2 = 0.4$*,* $K_1 = K_2 = 2$ *and* $f_1(x) = f_2(x) = x$ *for* $x \in \{0, 1, 2\}$*; that is, the degradation rates for both machines are the same, they both have 3 possible states (with state 2 being the 'failed' state) and they both have the same cost function, which is a linear function of the degree of degradation. The repair rates are* $\mu_1 = 1.1$ *and* $\mu_2 = 1$*, so machine 1 can be repaired slightly faster than machine 2. The switch rate is* $\tau = 100$*. Table 3.1 shows an optimal policy for this system, computed using dynamic programming. To clarify, part (a) of Table 3.1 shows the decisions specified by the optimal policy at the 9 possible states where the repairer is at machine 1, and part (b) shows the decisions specified at the 9 possible states where the repairer is at machine 2.*

Table 3.1: Decisions made under an optimal policy for Example 3.1

(a) when the repairer is at machine 1

	$x_2 = 0$	$x_2 = 1$	$x_2 = 2$
$x_1 = 0$	1	2	2
$x_1 = 1$	1	1	1
$x_1 = 2$	1	2	1

(b) when the repairer is at machine 2

	$x_2 = 0$	$x_2 = 1$	$x_2 = 2$
$x_1 = 0$	1	2	2
$x_1 = 1$	1	1	1
$x_1 = 2$	1	2	1

Under the optimal policy, the decision under state $\mathbf{x} = (1, (2, 1))$ *is to switch to machine 2. Thus, the repairer moves away from machine 1 despite the fact that* $c_1\mu_1 > c_2\mu_2$ *and machine 1 requires repair* ($x_1 > 0$)*. This shows that in a finite-state problem, the 'top-priority' rule described in Duenyas and Van Oyen (1996) no longer applies.*

3.3 Index heuristic

In order to develop an index-based heuristic for our dynamic repair and maintenance problem, we begin by proving that the cost function (3.2) is equivalent (in terms of average cost incurred under a fixed stationary policy) to an alternative function that focuses attention on the repairer's current location. For a general state $\mathbf{x} = (i, (x_1, ..., x_m)) \in S$ and action $a \in A_{\mathbf{x}}$, let $\tilde{c}(\mathbf{x}, a)$ be defined as follows:

$$\tilde{c}(\mathbf{x}, a) = \begin{cases} \sum_{j \in M} f_j(K_j) - \frac{\mu_i}{\lambda_i}[f_i(K_i) - f_i(x_i - 1)], & \text{if } i \in M,\ x_i \geq 1 \text{ and } a = i, \\ \sum_{j \in M} f_j(K_j), & \text{otherwise.} \end{cases} \tag{3.5}$$

Notably, the function $\tilde{c}$ (unlike the original cost function c) depends on both the state $\mathbf{x}$ and the action a chosen under $\mathbf{x}$. We observe that $\tilde{c}(\mathbf{x}, a)$ includes a negative term if and only if the repairer is located at a damaged machine i and chooses to perform repairs. This negative term increases in magnitude as the machine approaches the pristine state. The next result establishes the equivalence between the cost functions c and $\tilde{c}$ under stationary policies. We provide a proof in Appendix B.2.

Proposition 3.2. *Let* θ *be a stationary policy, and let*

$$\tilde{g}_\theta(\mathbf{x}) := \liminf_{t \to \infty} t^{-1} \mathbb{E}_\theta \left[\sum_{n=0}^{t-1} \tilde{c}(\mathbf{x}_n, \theta(\mathbf{x}_n)) \,\middle|\, \mathbf{x}_0 = \mathbf{x} \right], \quad \mathbf{x} \in S, \tag{3.6}$$

where $\mathbf{x}_n$ is the system state at time step $n \in \mathbb{N}_0$. That is, $\tilde{g}_\theta(\mathbf{x})$ is the average cost under policy θ given that we replace the cost function $c(\mathbf{x})$ by $\tilde{c}(\mathbf{x}, a)$. Then

$$\tilde{g}_\theta(\mathbf{x}) = g_\theta(\mathbf{x}) \quad \forall \mathbf{x} \in S, \tag{3.7}$$

where $g_\theta(\mathbf{x})$ is the average cost defined in (3.3).

The intuition for Proposition 3.2 is as follows. If the repairer adopts a completely passive policy and never repairs any machines, then the long-run average cost is trivially equal to $\sum_{j \in M} f_j(K_j)$. On the other hand, suppose we modify the passive policy by choosing a specific state $\mathbf{x} = (i, (x_1, ..., x_m))$ with $x_i \geq 1$, and specifying that the repairer should always choose to repair machine i under this state. Given that the repairer chooses this action at a particular time step, there is a probability μ_i that the machine's state transitions to $x_i - 1$ at the next time step. Following this transition, we obtain a cost rate advantage of $[f_i(x_i) - f_i(x_i - 1)]$ compared to the passive policy (under which the machine would have remained in state x_i). This advantage persists until the machine next degrades, which is expected to happen $1/\lambda_i$ time steps later. At that point, the cost rate advantage becomes $[f_i(x_i + 1) - f_i(x_i)]$ and this also continues for an expected $1/\lambda_i$ time steps. By considering a sequence of further degradations until machine i eventually degrades to state K_i, we can quantify the advantage of choosing the repair action under state $\mathbf{x}$ as $\mu_i[f_i(K_i) - f_i(x_i - 1)]/\lambda_i$. It follows that this quantity should be deducted from the average cost of the passive policy every time the repair action is chosen at $\mathbf{x}$. In Appendix B.2 we give a proof using more rigorous arguments, by considering the detailed balance equations of the continuous-time Markov chain induced by a fixed stationary policy.

Proposition 3.2 enables us to consider an equivalent MDP formulation based on maximizing rewards, rather than minimizing costs. The reward under a particular state $\mathbf{x} \in S$ is equal to the absolute value of the negative term in (3.5) if the repairer performs a repair under $\mathbf{x}$, and zero otherwise. The next result formalizes this equivalence.

Corollary 3.3. *Consider an MDP formulation in which the single-step reward for being in state $\mathbf{x} = (i, (x_1, ..., x_m))$ and choosing action $a \in A_\mathbf{x}$, denoted by $r(\mathbf{x}, a)$, is given by*

$$r(\mathbf{x}, a) = \begin{cases} \dfrac{\mu_i}{\lambda_i}[f_i(K_i) - f_i(x_i - 1)], & \text{if } i \in M,\ x_i \geq 1 \text{ and } a = i, \\ \\ 0, & \text{otherwise,} \end{cases} \tag{3.8}$$

and the objective is to maximize the long-run average reward, defined as

$$u_\theta(\mathbf{x}) := \limsup_{t\to\infty} t^{-1}\mathbb{E}_\theta\left[\sum_{n=0}^{t-1} r(\mathbf{x}_n, \theta(\mathbf{x}_n)) \,\Big|\, \mathbf{x}_0 = \mathbf{x}\right], \quad \mathbf{x} \in S. \tag{3.9}$$

All other aspects of the MDP formulation, including states, actions and transition probabilities, are the same as described in Section 3.2. Then any stationary policy which maximizes the long-run average reward (3.9) in the new MDP formulation also minimizes the long-run average cost (3.3) in the previous MDP formulation.

Proof. By Proposition 3.2 it is sufficient to show that maximizing the long-run average reward (3.9) is equivalent to minimizing the modified average cost (3.6) in which the alternative cost function $\tilde{c}(\mathbf{x}, a)$ is used. Indeed, the cost function $\tilde{c}$ defined in (3.5) includes a constant term $\sum_{j\in M} f_j(K_j)$ that is incurred under all states. It follows that $\tilde{g}_\theta(\mathbf{x}) = \sum_{j\in M} f_j(K_j) - u_\theta(\mathbf{x})$, and hence minimizing $\tilde{g}_\theta(\mathbf{x})$ is equivalent to maximizing $u_\theta(\mathbf{x})$. □

Recall that, by Proposition 3.1, the optimal average cost g^* is independent of the initial state. Therefore Corollary 3.3 implies that the optimal average reward $u^* = \sup_\theta u_\theta(\mathbf{x})$ also has no dependence on $\mathbf{x}$.

The main advantage of using the reward-based formulation (3.9) is that it allows the reward under a particular state $\mathbf{x}$ to be computed purely in terms of the parameters for the repairer's current location i (with the reward being zero if i is an intermediate stage, rather than a machine), whereas the cost formulation (3.3) involves aggregating cost rates over all of the damaged machines under a particular state. This greatly assists the development of an index-based heuristic policy. A similar approach was used by Duenyas and Van Oyen (1996), who formulated a problem based on minimizing average costs but then developed an index heuristic based on maximizing rewards. In the infinite-state problem considered by Duenyas and Van Oyen (1996), the reward function takes the much simpler form $c_i\mu_i$ (where, in their notation, c_i is the linear cost rate associated with queue i). The reward function (3.9) in our model is more complicated as it depends on the level of degradation, x_i, and also on the degradation rate λ_i.

We now proceed to describe the development of an index-based heuristic for our MDP, focusing on the single-step reward function $r(\mathbf{x}, a)$ rather than the cost function $c(\mathbf{x})$ from Section 3.2. The basic principle underlying the heuristic is that, under any given state $\mathbf{x} \in S$, the repairer calculates 'average reward rates' (referred to as *indices*) associated

with different possible actions, and then chooses an action that maximizes the index. These indices are calculated over a short time horizon, by assuming that the repairer will commit to fully repairing the next machine that it visits. More specifically, if the repairer is located at a machine $i \in M$ then the index for remaining at node i is calculated by assuming that the machine remains there until $x_i = 0$, whereas the index for switching to an alternative node $j \neq i$ is calculated by assuming that the machine moves directly to node j and then remains there until $x_j = 0$. If the repairer is at an intermediate stage $i \in N$ then the procedure is similar, except we only calculate indices for moving to machines in $j \in M$ and do not calculate an index for remaining at i. These indices inform the action chosen by the repairer at the current time step, but the indices are recalculated at every time step, so (for example) the repairer might begin moving towards a particular machine j but change direction and move towards a different machine l if the latter machine experiences a degradation event.

Firstly, for a particular machine $i \in M$, let $T_i(k)$ denote the random amount of time taken for the machine to return to its pristine state, given that it begins in state $k \in \{0, 1, ..., K_i\}$ and the repairer begins to repair it immediately and continues without any interruption. Also, let $R_i(k)$ denote the cumulative reward earned while these repairs are in progress, i.e. until the machine returns to the pristine state. We define both $T_i(0)$ and $R_i(0)$ to be zero. Expressions for $\mathbb{E}[T_i(k)]$ and $\mathbb{E}[R_i(k)]$ for $k \geq 1$ can be derived quite easily in terms of the system parameters, by considering the dynamics of an $M/M/1$ queue with finite capacity. These expressions are given in the next lemma.

Lemma 3.4. *For each machine $i \in M$ and $k \in \{1, ..., K_i\}$, we have*

$$\mathbb{E}[R_i(k)] = \sum_{p=1}^{K_i} C_{ip}(k) s_i(k), \qquad \mathbb{E}[T_i(k)] = \sum_{p=1}^{K_i} C_{ip}(k), \tag{3.10}$$

where $s_i(k) := \mu_i[f_i(K_i) - f_i(k-1)]/\lambda_i$ is the reward rate in state k and the coefficients $C_{ip}(k)$ are given by

$$C_{ip}(k) = \begin{cases} \dfrac{1}{\mu_i^p} \sum_{r=0}^{p-1} \lambda_i^{p-1-r} \mu_i^r, & p = 1, 2, ..., k, \\ \\ \dfrac{1}{\mu_i^p} \sum_{r=0}^{k-1} \lambda_i^{p-1-r} \mu_i^r, & p = k+1, k+2, ..., K_i. \end{cases}$$

Proof. Given that the repairer performs uninterrupted repairs at machine i, standard arguments based on the dynamics of continuous-time Markov chains imply that

$$\mathbb{E}[R_i(k)] = \begin{cases} \dfrac{s_i(k)}{\lambda_i+\mu_i} + \dfrac{\lambda_i}{\lambda_i+\mu_i}\mathbb{E}[R_i(k+1)] + \dfrac{\mu_i}{\lambda_i+\mu_i}\mathbb{E}[R_i(k-1)], & k=1,2,...,K_i-1, \\ \\ \dfrac{s_i(K_i)}{\mu_i} + \mathbb{E}[R_i(K_i-1)], & k=K_i. \end{cases} \tag{3.11}$$

Thus, we have a system of K_i equations in K_i unknowns and can solve these to find the values of $\mathbb{E}[R_i(1)], ..., \mathbb{E}[R_i(K_i)]$ stated in the lemma.

For the $\mathbb{E}[T_i(k)]$ values, the equations (3.11) take exactly the same form except that we replace each $R_i(k)$ by $T_i(k)$ and each reward rate $s_i(k)$ should be replaced by 1. The equations can then be solved to give the results stated in the lemma. □

Suppose the system is in state $\mathbf{x} = (i, (x_1, ..., x_m))$, where $i \in M$. We define the index for remaining at machine i, denoted by $\Phi_i^{\text{stay}}(x_i)$, as

$$\Phi_i^{\text{stay}}(x_i) = \begin{cases} \dfrac{\mathbb{E}[R_i(x_i)]}{\mathbb{E}[T_i(x_i)]}, & \text{if } x_i \in \{1, ..., K_i\}, \\ \\ 0, & \text{if } x_i = 0. \end{cases} \tag{3.12}$$

Next, let the repairer's current location i be an arbitrary node in V, and let D_{ij} denote the random amount of time needed for the repairer to move from node i to a machine $j \neq i$, assuming that it follows a shortest path in the network. We define the index for moving to machine j, denoted by $\Phi_{ij}^{\text{move}}(x_j)$, as

$$\Phi_{ij}^{\text{move}}(x_j) := \sum_{k=x_j}^{K_j} \mathbb{P}(X_j = k)\, \frac{\mathbb{E}[R_j(k)]}{\mathbb{E}[D_{ij}|X_j = k] + \mathbb{E}[T_j(k)]}, \tag{3.13}$$

where X_j is the (random) state of machine j upon arrival of the repairer at node j, suppressing the dependence on x_j for convenience. We note that X_j is distributed as a sum of i.i.d. geometric random variables and therefore follows a truncated version of

the negative binomial distribution, as follows:

$$\mathbb{P}(X_j = k) = \begin{cases} \binom{d_{ij}+k-x_j-1}{d_{ij}-1} \left(\frac{\tau}{\lambda_j+\tau}\right)^{d_{ij}} \left(\frac{\lambda_j}{\lambda_j+\tau}\right)^{k-x_j}, & k = x_j, ..., K_j - 1, \\ 1 - \sum_{r=x_j}^{K_j-1} \mathbb{P}(X_j = r), & k = K_j, \end{cases}$$

where d_{ij} is the number of nodes that the repairer must traverse on a shortest path from i to j. For $k = x_j, ..., K_j - 1$ we have

$$\mathbb{E}[D_{ij}|X_j = k] = \frac{d_{ij} + k - x_j}{\tau + \lambda_j},$$

while for $k = K_j$ we use the law of total expectation, which implies that

$$\mathbb{E}[D_{ij}] = \frac{d_{ij}}{\tau} = \sum_{k=x_j}^{K_j} \mathbb{P}(X_j = k)\mathbb{E}[D_{ij}|X_j = k],$$

and hence

$$\mathbb{E}[D_{ij}|X_j = K_j] = \frac{(d_{ij}/\tau) - \sum_{k=x_j}^{K_j-1} \mathbb{P}(X_j = k)\mathbb{E}[D_{ij}|X_j = k]}{1 - \sum_{k=x_j}^{K_j-1} \mathbb{P}(X_j = k)}.$$

We may interpret the index $\Phi_i^{\text{stay}}(x_i)$ as an approximate measure of the average reward per unit time that the repairer can obtain by remaining at machine i until $x_i = 0$, while $\Phi_{ij}^{\text{move}}(x_j)$ is the approximate average reward per unit time for switching to machine j and remaining there until $x_j = 0$. These are only approximate measures because, in fact, random variables such as $R_j(k)$ and $T_j(k)$ are not independent of each other and so the quotients that appear in (3.12) and (3.13) cannot be regarded as exact closed forms for the average rewards that would be obtained during the relevant time periods.

There is one further index quantity that should be defined. Suppose the system is in state $\mathbf{x} = (i, (x_1, ..., x_m))$ and let $\Phi_{ij}^{\text{wait}}(x_j)$ be defined in a similar way to $\Phi_{ij}^{\text{move}}(x_j)$ except that it represents the (approximate) average reward that the repairer would gain by waiting for one further degradation event to occur at machine $j \neq i$, then carrying out the same actions as in the case of the previous index (that is, switching to j and repairing it until $x_j = 0$). If $x_j = K_j$ then no further degradations at j are possible, but we still suppose that the repairer waits for $1/\lambda_j$ time units before moving to j.

Specifically, we define

$$\Phi_{ij}^{\text{wait}}(x_j) := \sum_{k=x_j}^{K_j-1} \mathbb{P}(X_j = k)\, \frac{\mathbb{E}[R_j(k+1)]}{1/\lambda_j + \mathbb{E}[D_{ij}|X_j = k] + \mathbb{E}[T_j(k+1)]}, + \mathbb{P}(X_j = K_j)\, \frac{\mathbb{E}[R_j(K_j)]}{1/\lambda_j + \mathbb{E}[D_{ij}|X_j = K_j] + \mathbb{E}[T_j(K_j)]}. \quad (3.14)$$

Note that in (3.14), X_j has the same definition as before (i.e. the state of machine j when the repairer arrives, given that the repairer begins moving immediately), but we calculate the index using $R_j(k+1)$ and $T_j(k+1)$ rather than $R_j(k)$ and $T_j(k)$ because the repairer waits for an extra degradation event before beginning to move. If $\Phi_{ij}^{\text{move}}(x_j) < \Phi_{ij}^{\text{wait}}(x_j)$ then we can interpret this as an incentive for the repairer to wait longer before moving to node j, since the occurrence of a further degradation event will increase the average reward that can be earned by switching to j and repairing the machine fully. Note that if $x_j = K_j$ then it is guaranteed that $\Phi_{ij}^{\text{move}}(x_j) > \Phi_{ij}^{\text{wait}}(x_j)$, so there is no incentive to wait in that case. In our index heuristic, we prevent the repairer from switching from machine i to machine j if $\Phi_{ij}^{\text{move}}(x_j) < \Phi_{ij}^{\text{wait}}(x_j)$.

Before defining the index heuristic, we discuss how it should make decisions under states with $x_j = 0$ for all $j \in M$. States of this form are quite different from others in S, as the repairer does not have any work to do. However, one might suppose that the repairer should still take some kind of preparatory action in order to ensure that it is able to respond to any future jobs in the most efficient manner. Accordingly, we aim to identify a particular node in the network that can be regarded as an 'optimal position' for the repairer while it is waiting for new tasks to materialize. Specifically, for each node $i \in V$, let $\Psi(i)$ be defined as follows:

$$\Psi(i) := \sum_{j \in M} \left(\frac{\lambda_j}{\lambda_1 + \ldots + \lambda_m} \right) \mathbb{E}[D_{ij}], \quad (3.15)$$

Thus, $\Psi(i)$ is the expected amount of time spent moving to the next machine that degrades, where the expectation is taken with respect to which machine this will be. We define the repairer's optimal idling position as the node $i \in V$ that minimizes $\Psi(i)$. If there are multiple such nodes, then an arbitrary choice of idling position can be made.

The next definition specifies how actions are chosen by the index heuristic.

Definition 3.5. *(Index policy.) Let θ^{IND} be a stationary policy that uses the following rules to select an action under state $\mathbf{x} = (i, (x_1, ..., x_m)) \in S$:*

1. *If $x_j = 0$ for all $j \in M$ (that is, all machines are in the pristine condition) then let $i^* \in \arg\min_{i \in V} \Psi(i)$. The repairer's action should be to move to the first node on the shortest path from i to i^* (if $i \neq i^*$), or to remain at i^* (if $i = i^*$).*
2. *If $x_j \geq 1$ for at least one $j \in M$, then consider two possible cases:*
   (a) *If $i \in M$ (that is, the repairer is at a machine) then let the set J be defined by*
   $$J := \left\{ j \in M \setminus \{i\} \ : \ \Phi_{ij}^{\text{move}}(x_j) \geq \Phi_{ij}^{\text{wait}}(x_j) \right\}. \tag{3.16}$$
   If J is non-empty, then let j^* be the machine in J that maximizes $\Phi_{ij}^{\text{move}}(x_j)$. If $\Phi_{ij^*}^{\text{move}}(x_{j^*}) > \Phi_i^{\text{stay}}(x_i)$ then the repairer's action should be to move to the first node on the shortest path from i to j^*. On the other hand, if either J is empty or $\Phi_{ij^*}^{\text{move}}(x_{j^*}) \leq \Phi_i^{\text{stay}}(x_i)$, then the repairer's action should be to remain at node i.
   (b) *If $i \in N$ (that is, the repairer is at an intermediate stage), then let j^* be the machine that maximizes $\Phi_{ij}^{\text{move}}(x_j)$. The repairer's action should be to move to the first node on the shortest path from i to j^*.*

In cases where two or more actions are viable choices according to the above rules, we assume that the action chosen under θ^{IND} is based on a pre-determined priority ordering of the nodes in V. Alternative tie-breaking rules could be considered based on the relative positions of demand points to each other in the network, but in many practical cases, ties will rarely occur, so the tie-breaking rule should have a relatively small impact. We refer to θ^{IND} as the index heuristic policy.

It is clearly of interest to investigate how well the index heuristic policy performs in comparison to an optimal policy, and also how well it performs relative to other heuristic policies. In Section 3.5 we provide results from numerical experiments in order to demonstrate its performance empirically. Theoretical performance guarantees (e.g. suboptimality bounds) are difficult to establish in the full generality of our model. However, we are able to show that the index policy is optimal in certain special cases of our problem. These special cases are discussed in the next two propositions.

Proposition 3.6. *Suppose the following 3 conditions are satisfied:*

1. *All machines are directly connected to each other; that is, V is a complete graph.*
2. *Each machine has only two possible states; that is, $K_j = 1$ for each $j \in M$.*
3. *The degradation rates, repair rates and cost functions are the same for all machines. That is, we have $\lambda_1 = ... = \lambda_m$, $\mu_1 = ... = \mu_m$ and $f_1(1) = ... = f_m(1)$.*

Then the index policy is optimal.

The proof of Proposition 3.6 can be found in Appendix B.3. This proposition describes a homogeneous type of problem in which all machines have the same parameter values, and furthermore assumes that the state of a machine is a binary variable, so that it is effectively either 'working' or 'not working'. In this type of binary model, the cost function $f_j(k)$ for machine $j \in M$ is determined by a single positive number, $f_j(1)$. While binary models such as these might appear quite simplistic, they are still interesting and potentially relevant to real-world applications, and we include them in our numerical experiments in Section 3.5. Another significant assumption in Proposition 3.6 is that all nodes are connected to each other, so that we can assume there are no intermediate stages (as if the network has intermediate stages, these will clearly never be visited under an optimal policy).

We have been able to confirm using experiments that all three assumptions in Proposition 3.6 are necessary; that is, if any of them are omitted then the index policy may not be optimal. However, we have also found that the index policy may be optimal in another type of network, known as a 'star network', which we define below.

Definition 3.7. *Suppose there exists an intermediate stage $s \in N$ such that, for any two distinct machines $i, j \in M$, the unique shortest path from i to j is a path of length $2r$ (where $r \in \mathbb{N}$) that passes through node s. Then the network V is called a* star network*, with s and r referred to as the* center *and* radius *respectively.*

In a star network, all machines are equidistant from each other (see Figure 3.2). The next proposition shows that the index policy is optimal in a star network under certain conditions, including two of the conditions from Proposition 3.6.

Proposition 3.8. *Let V be a star network with radius r, and suppose that Assumptions 2 and 3 from Proposition 3.6 hold. Then the index policy is optimal, provided that*

$$\tau > 2r\lambda, \tag{3.17}$$

where λ is the common degradation rate for all machines.

Proof of Proposition 3.8 can be found in Appendix B.4. We note that the condition in 3.17 is equivalent to $2r/\tau < 1/\lambda$, which states that the expected amount of time for the repairer to move from one machine to another is smaller than the expected amount of time for a degradation event to occur at a non-degraded machine. This condition should often be satisfied in realistic applications. Under the stated assumptions, the index policy is a 'greedy' policy that always prioritizes the nearest degraded machine (or moves towards the center of the network if there are no degraded machines), and we can show that such a policy is also optimal. Please see B.4 for full details of the proof.

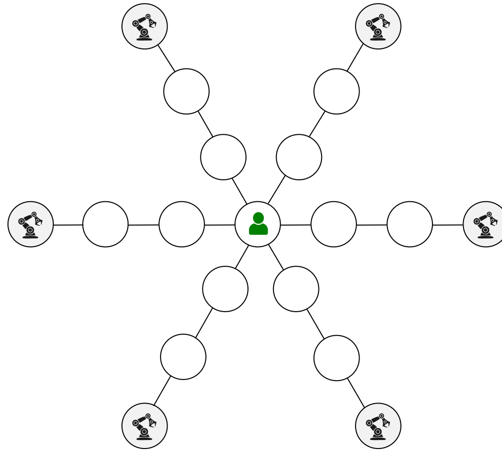

FIGURE 3.2: A star network with 6 machines and a radius $r = 3$.

3.4 Online policy improvement

The index heuristic developed in Section 3.3 yields a dynamic policy that, at any given time, either chooses a machine to prioritize next (if at least one machine requires repair), or directs the repairer to an optimal idling position (if no machines require repair). Importantly, in the former case, the indices in (3.12)-(3.14) are recalculated at each time step, so that the repairer can take advantage of the ability to interrupt switching and repair times as permitted under our problem formulation. However, a weakness of the index heuristic is that it works in a 'short-sighted' (myopic) way, as the decision to prioritize a particular machine is made without taking into account that machine's proximity to other machines in the network. It would be preferable to use a more sophisticated approach that takes the network topology into account when making decisions.

As discussed in Section 3.2, an optimal policy can be characterized by a set of values $\{h(\mathbf{x})\}_{\mathbf{x}\in S}$ satisfying the optimality equations in (3.4). The well-known method of *policy improvement* (PI), applicable to finite-state MDPs under the average cost criterion, works by starting from an arbitrary base policy and performing successive improvement steps, yielding a sequence of policies that eventually converges to an optimal one (Puterman (1994)). An optimal policy will naturally make long-sighted decisions that depend on the network topology, and any step of PI is therefore helpful in obtaining a policy that avoids short-sighted behavior. Although multiple improvement steps may be needed in order to obtain an optimal policy, it has been observed that the first improvement step often yields the largest improvement (Tijms (2003)). Furthermore, heuristics based on applying one step of policy improvement have been successfully applied to various types of queueing control problems (Krishnan (1990); Argon et al. (2009); Shone et al. (2020)). In this section we propose an approximate one-step PI method that works by simulating the random evolution of the system, in the style of a reinforcement learning algorithm. The method is approximate because, by using simulation, we are only able to acquire estimates of the value function h_θ under any base policy θ. The exact evaluation of h_θ would require dynamic programming, which (as discussed earlier) becomes computationally intractable when the state space S is large.

As stated in Proposition 3.1, an optimal policy for our system is always unichain. However, the policy given by the index heuristic in Section 3.3 is not guaranteed to be unichain and indeed we have found that it can have a multichain structure in some problem instances. It will therefore be desirable to introduce a modified version of the index heuristic in which the resulting policy must be unichain, in order to avoid certain complications that arise when applying PI to a multichain base policy. The modified index heuristic is defined below.

Definition 3.9. *(Modified index policy.) Let $\theta^{\text{M-IND}}$ denote a stationary policy that uses the following rules to select an action under state $\mathbf{x} = (i, (x_1, ..., x_m)) \in S$:*

1. *If $x_j = K_j$ for all $j \in M$ (that is, all machines are at their maximum state of degradation), then let $j^* \in M$ be the smallest-indexed machine satisfying*

$$j^* \in \arg\max_{j\in M} \left\{ \frac{\mathbb{E}[R_j(K_j)]}{\mathbb{E}[T_j(K_j)]} \right\}. \tag{3.18}$$

   Then the action chosen under state $\mathbf{x}$ should either be to move to the first node on the shortest path from i to j^* (if $i \neq j^*$), or to stay at node i (if $i = j^*$).

2. *Otherwise, the action chosen under state* $\mathbf{x}$ *should be the same action given by the unmodified index policy* θ^{IND} *in Definition 3.5.*

Note that under any admissible policy, it is always possible for the system to reach a state with $x_j = K_j$ for all $j \in M$, since this can occur randomly via an unbroken sequence of machine degradations. Under the modified index policy $\theta^{\text{M-IND}}$, the machine j^* satisfying (3.18) is unique and has no dependence on the system state $\mathbf{x}$. Thus, the state $(j^*, (K_1, ..., K_m))$ is accessible from any state under $\theta^{\text{M-IND}}$. This ensures that there cannot be more than one positive recurrent class of states under $\theta^{\text{M-IND}}$, and therefore the policy is indeed unichain.

In general, the performance of $\theta^{\text{M-IND}}$ may be worse than that of θ^{IND}, since the rule that it uses under states with $x_j = K_j$ for all $j \in M$ is not consistent with the rules used under other states. The only purpose of introducing $\theta^{\text{M-IND}}$ is so that it can be used as a convenient base policy in our PI method. In the computational experiments that we report in Section 3.5 we use the original definition of the index policy (Definition 3.5) when evaluating the performance of the index heuristic, and compare its performance with that of the simulation-based PI method.

Our PI method consists of an offline part and an online part. The purpose of the offline part is to evaluate the base policy, $\theta^{\text{M-IND}}$, and this is done by using extensive simulation experiments to estimate the average cost $g_{\theta^{\text{M-IND}}}$ and store an array of estimated values $h_{\theta^{\text{M-IND}}}(\mathbf{x})$ for states $\mathbf{x}$ that are visited often under $\theta^{\text{M-IND}}$. Subsequently, in the online part, we simulate the evolution of the system and, at each discrete time step, choose an action by applying an improvement step to the base policy. Actions must be selected at the beginning of each time step, but we assume that after an action is selected, the next transition of the system does not occur until a further δ time units have elapsed (here, δ represents the length of a time step and is analogous to the uniformization parameter Δ used in Section 3.2). During this δ-length time period we run nested simulation experiments (within the main simulation) with the aim of improving the estimates of $h_{\theta^{\text{M-IND}}}(\mathbf{x}')$ for states $\mathbf{x}'$ that are near the current state $\mathbf{x}$. The results of the nested online simulations supplement the results of the offline simulations in order to estimate the value function as accurately as possible within a neighborhood of the current state. Once the time limit δ is reached, we use a criterion based on a confidence interval to determine whether to select the best 'improving' action identified during the δ-length period or (instead) whether to select the action prescribed by the base policy (the latter

option is chosen in situations where we cannot say with sufficient confidence which action is best).

Some additional explanations of how the offline and online parts work are provided in the subsections below. To simplify notation we use θ rather than $\theta^{\text{M-IND}}$ to denote the modified index policy, referred to as the base policy. Due to space restrictions, we defer full details of the algorithms (including pseudocode) to Appendix B.5.

3.4.1 Offline part

In the offline part we simulate the evolution of the discrete-time MDP formulated in Section 3.2. In the preparatory stage, we conduct an initial simulation in order to identify a subset of states in $Z \subset S$ that are visited often under the base policy θ, and also obtain an estimate $\hat{g}_\theta$ of the average cost under this policy. We then initialize an array U, used to store information about states visited during the simulation. Initially, the array U contains a single state which we denote by $\mathbf{u}_0$.

Next, we conduct a large number of variable-length simulations, where each simulation begins from one of the states in Z and continues until the system reaches one of the states in U. In general, suppose the rth variable-length simulation (for $r = 1, 2, ...$) begins from state $\mathbf{z}_r \in Z$ and ends in state $\mathbf{u}_r \in U$. Also, let $\mathbf{x}$ be one of the first p states visited during this simulation, where p is a small integer (we use $p = 5$ in our experiments). Then we calculate an estimate of $h_\theta(\mathbf{x})$, denoted by $\hat{h}_\theta^{(s)}(\mathbf{x})$ (where s is a counter of how many estimates have been obtained for $h_\theta(\mathbf{x})$ so far), as follows:

$$\hat{h}_\theta^{(s)}(\mathbf{x}) = C_r(\mathbf{z}_r, \mathbf{u}_r) + \hat{h}_\theta(\mathbf{u}_r) - \hat{g}_\theta T_r(\mathbf{z}_r, \mathbf{u}_r), \tag{3.19}$$

where $C_r(\mathbf{z}_r, \mathbf{u}_r)$ is the total cost incurred during the variable-length simulation, $T_r(\mathbf{z}_r, \mathbf{u}_r)$ is the total number of time steps in the simulation, and $\hat{h}_\theta(\mathbf{u}_r)$ is the latest available estimate of $h_\theta(\mathbf{u_r})$ which is calculated using all of the estimates $h_\theta^{(1)}(\mathbf{u}_r), h_\theta^{(2)}(\mathbf{u}_r)$, etc. Note that equation (3.19) is derived using the well-known policy evaluation equations in dynamic programming, which are similar to the optimality equations (3.4). It can be explained intuitively as follows: in order to obtain an estimate of the relative value of being in state $\mathbf{x}$, we take the total cost incurred from starting in that state, which is $C_r(\mathbf{z}_r, \mathbf{u}_r) + \hat{h}_\theta(\mathbf{u}_r)$ (where $\hat{h}_\theta(\mathbf{u}_r)$ represents the 'terminal cost' associated with finishing in state $\mathbf{u}_r$) and subtract the cost that would have been earned if we had incurred the long-run average cost $\hat{g}_\theta$ over $T_r(\mathbf{z}_r, \mathbf{u}_r)$ time steps.

After obtaining the new estimate $\hat{h}_{\theta}^{(s)}(\mathbf{x})$ we add state $\mathbf{x}$ to the array U if it is not included already. We also store some other information about $\mathbf{x}$ in U, including the sum of the squared estimates $[\hat{h}_{\theta}^{(s)}(\mathbf{x})]^2$, which can be used to calculate a variance; this is needed in the online part of the algorithm (please see Appendix B.5 for full details). As the offline part of the algorithm continues, the array U becomes populated with more states and more information about those states. After a large number of iterations, the offline part ends. The intended outcome is that U should include a subset of states that are visted often under the base policy and, for each state $\mathbf{x} \in U$, an estimate $\hat{h}_{\theta}(\mathbf{x})$ which should be a reasonably accurate estimate of the true value $h_{\theta}(\mathbf{x})$. We have been able to verify using experiments that the algorithm is able to output estimates $\hat{h}_{\theta}(\mathbf{x})$ that are very close to the true values $h_{\theta}(\mathbf{x})$ given by dynamic programming.

3.4.2 Online part

In this part we take the array U obtained from the offline part as an input and simulate the evolution of the system again, but instead of using the actions given by the base policy θ, we apply an approximate policy improvement step. This means that if $\mathbf{x}$ is the current state, we aim to select the action a that minimizes

$$\sum_{\mathbf{y} \in S} p_{\mathbf{x},\mathbf{y}}(a)\hat{h}_{\theta}(\mathbf{y}), \tag{3.20}$$

where $\hat{h}_{\theta}(\mathbf{y})$ is the latest available estimate of $h_{\theta}(\mathbf{y})$. This implies that if $\mathbf{x} = (i, (x_1, ..., x_m))$ denotes the current state then we require estimates $\hat{h}_{\theta}(\mathbf{y})$ for all possible 'neighboring' states, i.e. all states $\mathbf{y}$ such that $p_{\mathbf{x},\mathbf{y}}(a) > 0$ for some $a \in A_{\mathbf{x}}$. In general, let $\mathbf{x}^{j+}$ (resp. $\mathbf{x}^{j-}$) denote a state identical to $\mathbf{x}$ except that the component x_j is increased (resp. decreased) by one, and let $\mathbf{x}^{\to j}$ denote a state identical to $\mathbf{x}$ except that the repairer's location is changed to j, where $j \neq i$. We also write $\mathbf{x}^{\to i} := \mathbf{x}$. Note that degradation events occur independently of the action chosen, so any state of the form $\mathbf{x}^{j+}$ for some $j \in M$ can be ignored when it comes to minimizing (3.20), because it occurs with the same probability under any action. Therefore, to find the action that minimizes (3.20), we only need estimates $\hat{h}_{\theta}(\mathbf{y})$ for states $\mathbf{y}$ that are either (i) equal to $\mathbf{x}$, (ii) of the form $\mathbf{y} = \mathbf{x}^{\to j}$ where j is a node adjacent to i, or (iii) of the form $\mathbf{x}^{i-}$ where $i \in M$ is the repairer's current location and $x_i \geq 1$. Let $F(\mathbf{x})$ denote the set of states satisfying either (i), (ii) or (iii).

Immediately after the system enters state $\mathbf{x}$, we identify the action that minimizes (3.20) by using the latest available estimates $\hat{h}_\theta(\mathbf{y})$ for states $\mathbf{y} \in F(\mathbf{x})$. However, rather than simply selecting the action that minimizes this expression, instead we use the information in U to calculate a confidence interval of the form $[\hat{h}_\theta^-(\mathbf{y}), \hat{h}_\theta^+(\mathbf{y})]$ for each $\mathbf{y} \in F(\mathbf{x})$. We then say that an action $a_1 \in A_\mathbf{x}$ is better than another action $a_2 \in A_\mathbf{x}$ if and only if

$$\sum_{\mathbf{y}\in S} p_{\mathbf{x},\mathbf{y}}(a_1)\hat{h}_\theta(\mathbf{y}) < \sum_{\mathbf{y}\in S} p_{\mathbf{x},\mathbf{y}}(a_2)\hat{h}_\theta(\mathbf{y}) \qquad \forall \hat{h}_\theta(\mathbf{y}) \in [\hat{h}_\theta^-(\mathbf{y}), \hat{h}_\theta^+(\mathbf{y})],\ \mathbf{y} \in F(\mathbf{x}). \quad (3.21)$$

In practice, (3.21) yields some simplifications. For example, if the repairer is at an intermediate stage of the network ($i \notin M$) then the only possible actions are to switch to an adjacent node or to idle at the current node. After substituting the relevant transition probabilities into (3.21), we find that a_1 should be preferred to a_2 if

$$\hat{h}_\theta^+(\mathbf{x}^{\to a_1}) < \hat{h}_\theta^-(\mathbf{x}^{\to a_2}),$$

which means that the confidence interval for $h_\theta(\mathbf{x}^{\to a_1})$ is non-overlapping with that of $h_\theta(\mathbf{x}^{\to a_2})$ and lies entirely below it. On the other hand, if the repairer is at a machine $i \in M$ with $x_i \geq 1$ then we also need to consider whether remaining at node i (i.e. carrying out repairs) should be preferred to switching to $j \neq i$. In this case, remaining at i is better than switching to j if the inequality

$$\mu_i\left(\hat{h}_\theta(\mathbf{x}^{i-}) - \hat{h}_\theta(\mathbf{x})\right) < \tau\left(\hat{h}_\theta(\mathbf{x}^{\to j}) - \hat{h}_\theta(\mathbf{x})\right)$$

holds for all $\hat{h}_\theta(\mathbf{x}^{i-}) \in [\hat{h}_\theta^-(\mathbf{x}^{i-}), \hat{h}_\theta^+(\mathbf{x}^{i-})]$, $\hat{h}_\theta(\mathbf{x}^{\to j}) \in [\hat{h}_\theta^-(\mathbf{x}^{\to j}), \hat{h}_\theta^+(\mathbf{x}^{\to j})]$ and $\hat{h}_\theta(\mathbf{x}) \in [\hat{h}_\theta^-(\mathbf{x}), \hat{h}_\theta^+(\mathbf{x})]$. If $\tau \geq \mu_i$ then this is equivalent to

$$\mu_i\hat{h}_\theta^+(\mathbf{x}^{i-}) - \tau\hat{h}_\theta^-(\mathbf{x}^{\to j}) + (\tau - \mu_i)\hat{h}_\theta^+(\mathbf{x}) < 0. \quad (3.22)$$

On the other hand, if $\tau < \mu_i$ then (3.22) still applies except that $(\tau - \mu_i)$ is negative and therefore $\hat{h}_\theta^+(\mathbf{x})$ should be replaced with $\hat{h}_\theta^-(\mathbf{x})$ in order to obtain an upper bound for the left-hand side. The confidence intervals are calculated using the method of reliability weights (Galassi et al. (2011), Sec. 21.7); please see Appendix B.5 for full details. If there exists an action $a^* \in A_\mathbf{x}$ which is better than all other actions according to the criteria described above, then this action is chosen by the improving policy. Otherwise, we cannot say with sufficient confidence which action should be chosen, and therefore as a 'safety measure' we set $a^* = \theta(\mathbf{x})$, so that the action prescribed by the base policy is chosen. By reverting to the base policy in the latter situation, we take a conservative

approach and aim to ensure that improving actions are chosen only when they are statistically likely to be better than the action of the base policy, which helps to ensure that the policy given by approximate PI is a genuine improvement over the base policy. This is akin to the method of 'safe policy improvement', for which there is some existing literature (Laroche et al. (2019); Simão et al. (2023)).

After an action $a^* \in A_{\mathbf{x}}$ has been chosen according to the rules above, the next transition of the system does not occur until a further δ time units have elapsed, and we perform as many nested simulations as possible until the time limit δ is reached. (In our numerical experiments in Section 3.5 we set $\delta = 0.01$ seconds, so that state transitions occur very frequently.) Each nested simulation proceeds as follows: first we simulate a single transition of the system under the chosen action a^* and observe a new state $\mathbf{x}'$. Then, for each of the states $\mathbf{y} \in F(\mathbf{x}')$, we simulate the evolution of the system starting from $\mathbf{y}$, assuming that the base policy θ is used for selecting actions, until eventually a state contained in the array U is reached. At this point an estimate is obtained for $h_\theta(\mathbf{y})$ using equation (3.19), as in the offline part. The estimates $\hat{h}_\theta(\mathbf{y})$ obtained from the nested simulations supplement the estimates already obtained from the offline part. Thus, the purpose of the nested simulations is to anticipate what the next state of the system might be (by sampling a new state $\mathbf{x}'$) and enhance the PI policy's ability to make a good choice of action at the next state (by improving the estimates $\hat{h}_\theta(\mathbf{y})$ for states $\mathbf{y} \in F(\mathbf{x}')$). The nested simulations are carried out repeatedly until δ time units have elapsed, at which point the 'actual' realization of the next state $\mathbf{x}'$ is observed and the main simulation continues.

By simulating a large number of time steps, with actions chosen using the rules above, we can evaluate the performance of the approximate PI policy. It should be noted that as the simulation progresses we continue to acquire new estimates $\hat{h}_\theta(\mathbf{x})$ for states $\mathbf{x}$ that are visited and the confidence intervals $[\hat{h}_\theta^-(\mathbf{x}), \hat{h}_\theta^+(\mathbf{x})]$ become narrower, which means that we revert to the base policy less often. Thus, as in many reinforcement learning algorithms, the policy becomes stronger as we acquire more experience from interacting with the environment.

3.5 Numerical results

In this section we report the results of computational experiments involving 1000 randomly generated problem instances. Each problem instance includes a randomly generated network layout and also a randomly sampled set of system parameters. To generate the network layout, we begin by constructing a 5×5 integer lattice with 25 nodes connected by horizontal and vertical edges. We then select a random subset of m nodes, where $2 \leq m \leq 8$, and designate these as machines. Thus, each machine $i \in M$ is assigned coordinates (a_i, b_i) with $a_i, b_i \in \{1, ..., 5\}$. The remaining (unselected) nodes serve as intermediate stages, although some of these may be redundant (in the sense that they would never be visited under an optimal policy) and can be ignored. Figure 3.3 shows an example in which machines are located at positions $(1, 3)$, $(2, 5)$ and $(3, 1)$. It is clear from the figure that, given these machine locations, the intermediate stages at $(2, 1)$, $(2, 2)$, $(2, 3)$ and $(2, 4)$ are the only ones that the repairer needs to visit in order to move between the machines in the most efficient manner possible, so there is no need to include intermediate stages at any other lattice points. The length of the shortest path between machines i and j is the Manhattan distance $|a_i - a_j| + |b_i - b_j|$.

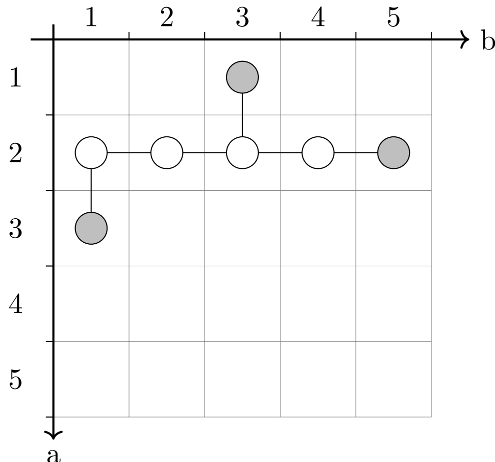

FIGURE 3.3: A network with machines at $(1, 3)$, $(2, 5)$ and $(3, 1)$ and intermediate stages at $(2, 1)$, $(2, 2)$, $(2, 3)$ and $(2, 4)$.

After the network layout has been generated for a specific problem instance, values of the system parameters are sampled randomly. We give a brief summary of this procedure here, but defer full details to Appendix B.6. We assume the number of possible degradation states, K_i, is the same for each machine $i \in M$, but sample this value randomly from the set $\{1, 2, 3, 4, 5\}$. We also consider three different cases for cost functions $f_i(x_i)$, as follows: (i) $f_i(x_i)$ increases linearly with the degradation state x_i;

(ii) $f_i(x_i)$ is convex and increases quadratically with x_i; (iii) $f_i(x_i)$ increases linearly for $x_i \in \{0, ..., K_i - 1\}$, but then increases much more sharply when x_i reaches the 'failed' state K_i. The type of cost function is the same for each machine within a particular problem instance, but the specific parameters used in the cost functions are machine-dependent, so some machines are associated with higher degradation costs than others. The degradation rates λ_i and repair rates μ_i are randomly generated in such a way that the overall traffic intensity, given by $\rho = \sum_i \lambda_i/\mu_i$, lies within the range $[0.1, 1.5]$. Thus, we consider systems where the repairer is very under-utilized (i.e. ρ is near zero) and also systems where it is in very high demand (i.e. $\rho > 1$). We also define the parameter $\eta := \tau/(\sum_{i \in M} \lambda_i)$ as a measure of how fast the repairer is able to switch between adjacent nodes relative to the total of the machine degradation rates, and generate the value of τ in order to consider a wide range of cases for the value of η. Please see Appendix B.6 for full details of our parameter generation methods.

Recall that θ^{IND} denotes the index policy defined in Section 3.3 (see Definition 3.5). We use θ^{API} to denote the policy followed during online implementation of the approximate policy improvement (API) method discussed in Section 3.4, with a time limit $\delta = 0.01$ used for decision-making; that is, the system changes state every 0.01 seconds and the API heuristic is forced to make a decision within that time (please see Appendix B.5 for further details of how the API heuristic is implemented in our experiments). Within each problem instance, we use simulation experiments to assess the performances of θ^{IND} and θ^{API}. Also, in instances with $2 \leq m \leq 4$, we use dynamic programming (specifically, exact policy improvement) to calculate optimal long-run average costs and rewards and use these to evaluate suboptimality percentages for these heuristics. In this setting, we also evaluate a simple benchmark heuristic, denoted by θ^{POL}, under which the server tours a subset of the m machines in a fixed cyclic order and always completes the full repair of a machine before moving on to the next one (see, Section 3.5.1, for details). As m increases, it becomes computationally infeasible to calculate an optimal policy using dynamic programming, so we do not calculate suboptimality percentages for systems with $5 \leq m \leq 8$.

Our experiments were carried out on an Intel(R) Core(TM) i5-6500T desktop computer, with 8GB of RAM, running Windows 10. The software used was Python 3.9.10, with the PyPy just-in-time compiler (http://pypy.org) used to speed up computations. During online implementation, the θ^{POL} and θ^{IND} policies require a negligible amount of time to make a decision at any individual time step. On the other hand, θ^{API} carries out as many nested simulations as it can within the time limit δ in order to improve the value function

estimates for nearby states, as described in Section 3.4. Since all of our heuristics can make decisions quickly in an online setting, the number of experiments that we are able to perform is limited not by the decision-making processes of the heuristics themselves, but rather by the number of time steps that we need to simulate in order to accurately evaluate the performances of the heuristics. We simulate each heuristic over $500,000$ time steps within each problem instance (please see Appendix B.7 for further details).

The remainder of this section is organized as follows. In Section 3.5.1 we describe a simple heuristic based on polling systems that can be used to provide a useful benchmark for assessing the performances of the index and API heuristics. In Section 3.5.2, we evaluate the performances of our heuristic policies by calculating their suboptimalities relative to the optimal values in systems of modest size ($2 \leq m \leq 4$) and also comparing them with the polling system policies. We also report the improvements that the API heuristic is able to achieve over the index heuristic in larger systems ($5 \leq m \leq 8$). In Section 3.5.3 we investigate the effects of varying the system parameters on the performances of our heuristic policies.

3.5.1 Polling system heuristic

In order to provide an additional benchmark for comparisons, we introduce a simple 'polling system' heuristic that works by touring a subset of the m machines in a cyclic pattern and always fully repairing each machine it visits before moving to the next one. Consider a non-empty subset of machines $\mathcal{M} \subseteq M$ and let $\theta^{\text{POL}}(\mathcal{M})$ denote a policy under which the repairer visits the machines in a cyclic pattern $(i_1, ...i_{|\mathcal{M}|}, i_1, ...)$, where $i_1, ..., i_{|\mathcal{M}|}$ are distinct nodes in $\mathcal{M}$. We assume that the order of visiting the machines is chosen in order to minimize the total travel distance,

$$D(i_1, ..., i_{|\mathcal{M}|}) := \sum_{j=1}^{|\mathcal{M}|-1} d(i_j, i_{j+1}) + d(i_{|\mathcal{M}|}, i_1), \tag{3.23}$$

where $d(i, j)$ is the length of a shortest path between machines i and j. In order to minimize this travel distance we must solve a 'traveling salesman' problem (TSP), but since we only implement the polling system heuristic for systems with $m \leq 4$ the solution to the TSP is easy to find using brute force. Thus, for any given subset $\mathcal{M} \subseteq M$ we calculate the performance of $\theta^{\text{POL}}(\mathcal{M})$ by first solving a TSP to find a sequence $(i_1, ..., i_{|\mathcal{M}|})$ that minimizes the travel distance (3.23), then simulating the average cost incurred when the repairer follows the resulting cyclic pattern, enforcing the rule that damaged machines

are always fully repaired before the repairer moves to the next machine. For any given system, we can obtain a useful benchmark for our index and API heuristics by simulating the performances of all possible 'polling system' policies $\theta^{\text{POL}}(\mathcal{M})$ (taking into account all possible subsets $\mathcal{M} \subseteq M$) and then reporting the lowest average cost among all of these. We use $\theta^{\text{POL}} \in \arg\min_{\mathcal{M}}\{g_{\theta^{\text{POL}}(\mathcal{M})}(\mathbf{x})\}$ to denote the best-performing polling system heuristic within a particular instance (with ties broken arbitrarily).

3.5.2 Performances of heuristic policies in systems with $2 \leq m \leq 8$

In systems with $2 \leq m \leq 4$ we measure the percentage suboptimalities of the heuristic policies θ^{POL}, θ^{IND} and θ^{API} by comparing the average costs under these policies with the optimal values given by dynamic programming. In addition, we report the percentage suboptimalities when the average reward $u_\theta(\mathbf{x})$ defined in (3.9) is used to measure the performance of a policy, rather than the average cost $g_\theta(\mathbf{x})$. Note that by Proposition 3.2 and Corollary 3.3 we have $g_\theta(\mathbf{x}) = \sum_{j\in M} f_j(K_j) - u_\theta(\mathbf{x})$, and therefore the *absolute* suboptimality is the same regardless of whether it is measured using costs or rewards. That is, for any policy θ and $\mathbf{x} \in S$ we have

$$g_\theta(\mathbf{x}) - g^* = u^* - u_\theta(\mathbf{x}),$$

where g^* and u^* denote the average cost and average reward, respectively, under the optimal policy. However, the *percentage* suboptimalities under the cost and reward formulations are $100 \times (g_\theta(\mathbf{x}) - g^*)/g^*$ and $100 \times (u^* - u_\theta(\mathbf{x}))/u^*$ respectively, and these are different because $g^* \neq u^*$ in general. Although our original problem formulation in Section 3.2 was given in terms of costs, we would argue that measuring percentage suboptimalities in terms of rewards is also useful, because u^* is more robust than g^* with respect to the problem size (represented by the number of machines m). To see this, note that as m increases, it becomes increasingly likely that some machines are never visited by the repairer; that is, they are left permanently broken. These non-visited machines cause the optimal average cost g^* to increase, but they have no effect on u^*. Hence, the ratio $(g_\theta(\mathbf{x}) - g^*)/g^*$ decreases purely because of the inflation in g^*, whereas the ratio $(u^* - u_\theta(\mathbf{x}))/u^*$ remains unaffected. The cost formulation measures average cost relative to the situation where all machines are pristine (working perfectly), while the reward formulation measures reward relative to the situation where are machines are completely failed. Thus the choice of cost or reward formulation could depend on whether machines are pristine most of the time, or whether there are often several

machines that are completely failed. In truth, both measures of suboptimality could be insightful depending on how one chooses to view the problem, so we consider both in this section.

Tables 3.2 and 3.3 show, for each heuristic policy, 95% confidence intervals for the mean percentage suboptimality relative to the optimal value g^* under the cost-based formulation and u^* under the reward-based formulation, respectively, based on results from 412 'small' problem instances. We also report the 10th, 25th, 50th, 75th, and 90th percentiles of the percentage suboptimality distributions for each heuristic. These results indicate that the approximate policy improvement heuristic (θ^{API}) consistently outperforms the other two heuristics, with an average percentage suboptimality of 2.51% under the cost formulation and 1.02% under the reward formulation. The index heuristic (θ^{IND}) performs well in many instances, but is clearly inferior to θ^{API}, most likely due to its short-sighted method of making decisions (as discussed in Section 3.3). Of course, we would expect θ^{API} to improve upon θ^{IND} because the method of constructing θ^{API} involves using θ^{IND} as a base and then applying approximate policy improvement. The polling system heuristic (θ^{POL}) is clearly weaker than θ^{IND} and θ^{API}, despite the fact that θ^{POL} is a type of 'ensemble' heuristic that involves taking the best average cost (or reward) from all of the possible polling heuristics $\theta^{\text{POL}}(\mathcal{M})$ within each problem instance. The weak performance of θ^{POL} essentially shows the limitations of failing to respond dynamically to the latest machine degradation levels.

Table 3.2: Percentage suboptimalities of heuristic policies in 412 'small' problem instances with $2 \leq m \leq 4$, evaluated under the cost-based formulation.

Heuristic	Mean	10th pct.	25th pct.	50th pct.	75th pct.	90th pct.
POL	29.64 ± 3.72	0.00	5.78	17.36	35.25	71.14
IND	8.11 ± 1.04	0.01	0.93	4.60	10.81	21.23
API	2.51 ± 0.55	0.00	0.03	0.66	2.45	6.60

Table 3.3: Percentage suboptimalities of heuristic policies in 412 'small' problem instances with $2 \leq m \leq 4$, evaluated under the reward-based formulation.

Heuristic	Mean	10th pct.	25th pct.	50th pct.	75th pct.	90th pct.
POL	8.36 ± 0.72	0.00	2.07	7.43	12.64	18.95
IND	3.01 ± 0.45	0.00	0.37	1.70	3.89	7.30
API	1.02 ± 0.32	0.00	0.02	0.25	0.83	2.09

Tables 3.4 and 3.5 show the percentage improvements achieved by θ^{API} against θ^{IND} over all 1000 problem instances, with the results categorized according to the number of machines m, under the cost and reward formulations respectively. In larger problem sizes (with $m \geq 5$) it becomes impractical to compute exact percentage suboptimalities,

but the results in these tables provide some assurance that the extra benefit given by applying approximate policy improvement remains fairly consistent as the size of the problem increases.

TABLE 3.4: Percentage improvements of API over the index policy for different values of m, evaluated under the cost-based formulation.

	$m = 2$ [132 instances]	$m = 3$ [148]	$m = 4$ [132]	$m = 5$ [146]	$m = 6$ [148]	$m = 7$ [149]	$m = 8$ [145]
API vs. IND	3.45 ± 1.13	5.43 ± 1.13	5.00 ± 0.91	5.96 ± 0.97	5.12 ± 0.76	6.33 ± 1.02	4.98 ± 0.94

TABLE 3.5: Percentage improvements of API over the index policy for different values of m, evaluated under the reward-based formulation.

	$m = 2$ [132 instances]	$m = 3$ [148]	$m = 4$ [132]	$m = 5$ [146]	$m = 6$ [148]	$m = 7$ [149]	$m = 8$ [145]
API vs. IND	1.83 ± 1.23	2.97 ± 1.53	2.12 ± 0.42	2.37 ± 0.37	2.32 ± 0.42	2.12 ± 0.36	2.08 ± 0.39

Table 3.6 shows, for each $m \in \{2, ..., 8\}$, the percentage of time steps at which the heuristic θ^{API} resorts to a 'safe' action choice by selecting the same action given by the index heuristic θ^{IND}. Recall from Section 3.4 that, in the online implementation of the API heuristic, the policy makes a 'safe' choice of action if it cannot identify the best choice of action with a sufficient level of confidence. Table 3.6 shows that 'safe' action choices are selected with increasing frequency as m increases. This is due to the fact that, as the number of dimensions in the state space increases, it becomes more difficult to obtain reliable estimates for the values at individual states during the offline stage because there are relatively few states that are visited sufficiently often. This problem could be addressed by allocating more time to the offline stage as m becomes larger, or increasing the online time limit δ (currently set at 0.01 seconds) so that the algorithm can gather more information about nearby states at each time step during the online stage.

TABLE 3.6: Percentage of 'safe' action choices under θ^{API} for different values of m.

	$m = 2$ [132 instances]	$m = 3$ [148]	$m = 4$ [132]	$m = 5$ [146]	$m = 6$ [148]	$m = 7$ [149]	$m = 8$ [145]
Pct. of 'safe' action choices	0.27 ± 0.16	0.86 ± 0.35	1.81 ± 0.47	4.53 ± 0.93	5.76 ± 0.77	8.37 ± 1.12	11.59 ± 1.21

3.5.3 Effects of varying the system parameters

Next, we investigate how the results from our experiments depend on the values of important system parameters. In Table 3.7 we categorize our results according to the

traffic intensity, $\rho = \sum_{i \in M} \lambda_i/\mu_i$. The first three rows of the table show percentage suboptimalities of the heuristic policies θ^{POL}, θ^{IND} and θ^{API} over the 412 instances with $2 \leq m \leq 4$, and the fourth row shows the percentage improvements of θ^{API} over θ^{IND} over all 1000 instances with $2 \leq m \leq 8$. The columns correspond to disjoint, equal-length intervals for ρ. Within the table cells we report 95% confidence intervals for the relevant values under both the cost formulation (shown in regular font) and the reward formulation (shown in bold font). Table 3.8 follows the same format as Table 3.7 except that the parameter of interest is $\eta = \tau/(\sum_{i \in M} \lambda_i)$, which measures the relative switching speed compared to the sum of the machine degradation rates. Table 3.9 again follows the same format except that we categorize the results according to the cost function $f_i(x_i)$. The three cost functions that we consider are: linear, quadratic and piecewise linear (see Appendix B.6 for full details). Finally, Table 3.10 again follows the same format except that the parameter of interest is K, which is the maximum degradation state used for all machines in the system.

Table 3.7: Summary of numerical results categorized according to the value of $\rho = \sum_i \lambda_i/\mu_i$, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font)

	$0.1 \leq \rho < 0.3$	$0.3 \leq \rho < 0.5$	$0.5 \leq \rho < 0.7$	$0.7 \leq \rho < 0.9$	$0.9 \leq \rho < 1.1$	$1.1 \leq \rho < 1.3$	$1.3 \leq \rho < 1.5$
θ^{POL} Subopt. [412 instances]	72.03 ± 13.88 [$\mathbf{9.11 \pm 2.09}$]	42.98 ± 10.65 [$\mathbf{9.91 \pm 2.30}$]	24.16 ± 5.09 [$\mathbf{8.66 \pm 2.23}$]	19.57 ± 4.33 [$\mathbf{7.28 \pm 1.75}$]	16.56 ± 4.10 [$\mathbf{8.96 \pm 1.97}$]	13.96 ± 3.61 [$\mathbf{8.96 \pm 1.87}$]	8.13 ± 1.66 [$\mathbf{6.27 \pm 1.21}$]
θ^{IND} Subopt. [412 instances]	8.92 ± 2.31 [$\mathbf{1.78 \pm 0.73}$]	13.29 ± 3.87 [$\mathbf{3.23 \pm 1.24}$]	11.32 ± 3.70 [$\mathbf{3.73 \pm 1.37}$]	9.20 ± 2.91 [$\mathbf{2.95 \pm 1.43}$]	6.46 ± 3.24 [$\mathbf{2.94 \pm 0.99}$]	5.74 ± 1.80 [$\mathbf{4.30 \pm 1.81}$]	2.86 ± 0.76 [$\mathbf{2.74 \pm 0.86}$]
θ^{API} Subopt. [412 instances]	2.90 ± 1.27 [$\mathbf{0.92 \pm 0.79}$]	3.51 ± 1.63 [$\mathbf{0.64 \pm 0.31}$]	3.35 ± 1.54 [$\mathbf{1.23 \pm 0.91}$]	2.98 ± 1.51 [$\mathbf{0.80 \pm 0.36}$]	2.87 ± 3.04 [$\mathbf{1.29 \pm 1.33}$]	1.51 ± 0.63 [$\mathbf{1.04 \pm 0.51}$]	0.87 ± 0.47 [$\mathbf{1.27 \pm 1.18}$]
θ^{API} improv. vs. θ^{IND} [1000 instances]	6.26 ± 1.02 [$\mathbf{1.21 \pm 0.55}$]	7.39 ± 1.34 [$\mathbf{2.35 \pm 0.71}$]	6.82 ± 1.04 [$\mathbf{2.67 \pm 0.50}$]	5.80 ± 1.05 [$\mathbf{2.49 \pm 1.09}$]	3.81 ± 0.81 [$\mathbf{2.13 \pm 0.60}$]	3.74 ± 0.75 [$\mathbf{3.06 \pm 1.29}$]	2.61 ± 0.51 [$\mathbf{2.19 \pm 0.80}$]

Table 3.8: Summary of numerical results categorized according to the value of $\eta = \tau/(\sum_{i \in M} \lambda_i)$, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font)

	$0.1 \leq \eta < 0.4$	$0.4 \leq \eta < 0.7$	$0.7 \leq \eta < 1$	$1 \leq \eta < 4$	$4 \leq \eta < 7$	$7 \leq \eta < 10$
θ^{POL} Subopt. [412 instances]	5.89 ± 2.16 [$\mathbf{4.89 \pm 1.66}$]	16.96 ± 5.09 [$\mathbf{10.50 \pm 2.38}$]	23.78 ± 6.19 [$\mathbf{11.57 \pm 2.18}$]	41.71 ± 11.45 [$\mathbf{10.72 \pm 1.61}$]	37.87 ± 10.52 [$\mathbf{7.17 \pm 1.05}$]	50.90 ± 10.82 [$\mathbf{5.88 \pm 0.82}$]
θ^{IND} Subopt. [412 instances]	4.09 ± 1.82 [$\mathbf{4.26 \pm 1.92}$]	5.33 ± 1.54 [$\mathbf{3.91 \pm 1.22}$]	6.26 ± 2.27 [$\mathbf{3.62 \pm 0.97}$]	10.43 ± 2.62 [$\mathbf{3.02 \pm 0.60}$]	9.85 ± 3.15 [$\mathbf{2.03 \pm 0.53}$]	12.43 ± 3.10 [$\mathbf{1.31 \pm 0.29}$]
θ^{API} Subopt. [412 instances]	1.59 ± 1.07 [$\mathbf{2.18 \pm 1.55}$]	1.24 ± 0.49 [$\mathbf{0.71 \pm 0.25}$]	2.13 ± 1.11 [$\mathbf{1.35 \pm 0.87}$]	2.84 ± 1.10 [$\mathbf{0.90 \pm 0.37}$]	2.95 ± 1.45 [$\mathbf{0.55 \pm 0.22}$]	4.20 ± 2.12 [$\mathbf{0.39 \pm 0.19}$]
θ^{API} improv. vs. θ^{IND} [1000 instances]	2.37 ± 0.78 [$\mathbf{3.26 \pm 1.66}$]	4.09 ± 0.63 [$\mathbf{3.33 \pm 0.59}$]	4.02 ± 0.89 [$\mathbf{2.58 \pm 0.60}$]	7.76 ± 0.95 [$\mathbf{2.29 \pm 0.32}$]	6.75 ± 1.15 [$\mathbf{1.27 \pm 0.22}$]	5.83 ± 0.79 [$\mathbf{0.88 \pm 0.13}$]

From Table 3.7 we can see that the polling system heuristic θ^{POL} is very weak when ρ is small, but improves as ρ increases. Indeed, when ρ is small, we often have a situation where only one machine is degraded (that is, $x_j > 0$ for only one $j \in M$) and the repairer should go directly to that machine under an optimal policy, but the polling heuristic

Table 3.9: Summary of numerical results categorized according to the type of cost function $f_i(x_i)$ used, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font)

	Linear	Quadratic	Piecewise linear
θ^{POL} Subopt.	26.47 ± 4.96	32.76 ± 7.41	29.09 ± 6.31
[412 instances]	[$\mathbf{10.16 \pm 1.42}$]	[$\mathbf{7.55 \pm 1.07}$]	[$\mathbf{7.55 \pm 1.27}$]
θ^{IND} Subopt.	5.38 ± 1.55	9.19 ± 1.99	9.50 ± 1.72
[412 instances]	[$\mathbf{2.98 \pm 1.05}$]	[$\mathbf{2.82 \pm 0.59}$]	[$\mathbf{3.27 \pm 0.70}$]
θ^{API} Subopt.	1.42 ± 0.54	3.20 ± 1.23	2.78 ± 0.83
[412 instances]	[$\mathbf{0.95 \pm 0.49}$]	[$\mathbf{0.77 \pm 0.33}$]	[$\mathbf{1.38 \pm 0.80}$]
θ^{API} improv. vs. θ^{IND}	3.58 ± 0.52	5.55 ± 0.64	6.49 ± 0.73
[1000 instances]	[$\mathbf{2.39 \pm 0.72}$]	[$\mathbf{2.19 \pm 0.30}$]	[$\mathbf{2.23 \pm 0.52}$]

Table 3.10: Summary of numerical results categorized according to the maximum degradation state K, with 95% confidence intervals shown under both the cost formulation (regular font) and the reward formulation (bold font)

	$K = 1$	$K = 2$	$K = 3$	$K = 4$	$K = 5$
θ^{POL} Subopt.	21.81 ± 5.95	33.79 ± 11.74	28.90 ± 7.34	33.91 ± 8.89	29.48 ± 7.74
[412 instances]	[$\mathbf{11.64 \pm 2.29}$]	[$\mathbf{9.21 \pm 1.81}$]	[$\mathbf{8.12 \pm 1.51}$]	[$\mathbf{7.12 \pm 1.30}$]	[$\mathbf{6.39 \pm 1.04}$]
θ^{IND} Subopt.	3.26 ± 1.11	4.74 ± 1.58	6.94 ± 1.52	12.01 ± 2.66	12.06 ± 3.09
[412 instances]	[$\mathbf{2.86 \pm 1.11}$]	[$\mathbf{2.01 \pm 0.86}$]	[$\mathbf{2.85 \pm 0.78}$]	[$\mathbf{3.66 \pm 1.15}$]	[$\mathbf{3.42 \pm 1.04}$]
θ^{API} Subopt.	1.19 ± 0.89	0.96 ± 0.47	1.83 ± 0.96	4.10 ± 1.36	3.93 ± 1.70
[412 instances]	[$\mathbf{1.57 \pm 1.33}$]	[$\mathbf{0.50 \pm 0.33}$]	[$\mathbf{0.95 \pm 0.73}$]	[$\mathbf{1.07 \pm 0.53}$]	[$\mathbf{0.98 \pm 0.27}$]
θ^{API} improv. vs. θ^{IND}	2.16 ± 0.58	4.33 ± 0.63	5.08 ± 0.67	7.02 ± 0.84	7.03 ± 1.09
[1000 instances]	[$\mathbf{1.71 \pm 0.86}$]	[$\mathbf{1.89 \pm 0.41}$]	[$\mathbf{2.18 \pm 0.48}$]	[$\mathbf{2.75 \pm 0.78}$]	[$\mathbf{2.70 \pm 0.73}$]

forces the repairer to make wasteful visits to all machines according to a fixed cyclic pattern. The other heuristics θ^{IND} and θ^{API} also seem to achieve lower suboptimality values as ρ increases under the cost formulation, but this may be caused by the inflation in g^* that occurs when the overall degradation burden increases (as discussed earlier). The bold-font figures in Table 3.7 show that when the reward formulation is used, there is no clear trend for the performances of θ^{IND} and θ^{API} to get better or worse as ρ increases. Instead, θ^{IND} is consistently within 5% of optimality, while θ^{API} is consistently within 2%.

Table 3.8 shows that, for all three heuristic policies, there is a general trend for suboptimality to increase with η when the cost formulation is used. The same reasoning given above also applies in this case, since smaller η values imply that machine degradation happens more frequently (relative to the repairer's movement speed), so the optimal value g^* will tend to increase. Under the reward formulation, however, it seems that the trend for θ^{IND} and θ^{API} is the opposite: as η increases, these heuristics seem to

improve their performance. This may be due to the fact that the indices used by θ^{IND} (which also influence the decisions of θ^{API}) are calculated under the assumption that the repairer will move directly to a particular demand point without changing direction, which is more likely to be an optimal course of action if η is large (since there is less chance of new degradation events occurring while the repairer is moving).

Table 3.9 shows that under the cost formulation, the three heuristic policies seem to achieve the lowest suboptimality values when the linear cost function is used. However, under the reward formulation, this trend is no longer apparent. Recall that the linear and quadratic cost functions are defined by $f_i(x_i) = c_i x_i$ and $f_i(x_i) = c_i x_i^2$ respectively, whereas the piecewise linear cost function $f_i(x_i) = c_i(x_i + 10 \cdot \mathbb{I}(x_i = K_i))$ assumes an extra penalty for any machine that has reached the state of complete failure. One might imagine that the piecewise linear case would be a difficult case to optimize, as even the machines with relatively low service priorities should still (ideally) be visited regularly enough to make complete failure unlikely. From Table 3.9, however, there is no clear evidence that the heuristics perform worse in this case than in the other two.

Finally, Table 3.10 shows that the effect of measuring machine degradation on a finer scale (in other words, increasing the number of possible degradation states) is rather difficult to pinpoint based on the suboptimality percentages. When the cost formulation is used, the performances of θ^{IND} and θ^{API} appear to become slightly worse as K increases, which may be expected since the problem becomes intrinsically more complex. However, under the reward formulation, this trend seems to disappear. Indeed, even in the $K = 5$ case, θ^{IND} and θ^{API} are able to achieve mean suboptimalities of less than 3.5% and 1% respectively. Encouragingly, in all of the cases shown in Tables 3.7-3.10, θ^{API} is able to obtain healthy improvements over θ^{IND}.

3.6 Conclusions

This paper considers an original model of a dynamic, network-based repair and maintenance problem in which both repair and switching times are interruptible. Each machine evolves through a finite set of degradation states, and switching times depend not only on the machines involved but also on the progress made during any interrupted operations. We develop an index-based heuristic by firstly proving a useful equivalence between the cost function (3.2) and an alternative cost function (3.5) that focuses on the repairer's

current location. The heuristic is proven to be optimal in certain special cases of the problem.

The index-based heuristic is limited by the fact that it makes decisions in a myopic manner, and does not consider the relative positions of machines in the network when deciding which one to prioritize next. In order to develop a more long-sighted approach that can exploit the network topography, we use reinforcement learning methods and approximate the value function of the index policy by simulating trajectories of the system. We then use one-step policy improvement during online operation, with the index policy's decisions used as a fall-back option in situations where the policy cannot identify the best 'improving' action with sufficient confidence. The results in Section 3.5, based on randomly-generated network layouts, show that the approximate policy improvement (API) approach yields a heuristic policy with close-to-optimal performance in smaller systems (with between 2 and 4 machines), and also maintains consistent improvements over the index policy in larger systems (with between 5 and 8 machines). These results were obtained by simulating fast-changing systems in which transitions occur at a rate of 100 per second, suggesting that the method is suitable for real-time implementation.

Future research could explore alternative approximation schemes, more general cost or reward functions, and extensions to multiple repairers or parallel maintenance networks, with applications to manufacturing plants, utility networks, and fleet maintenance operations.

Chapter 4

Stochastic Dynamic Job Scheduling with Interruptible Setup and Processing Times for Multiple Servers

We consider a stochastic, dynamic job scheduling problem on a network with multiple homogeneous servers, formulated as a queueing control problem. Jobs of various types arrive at designated demand points according to independent Poisson processes and wait in queues for service. Each server can be controlled independently of the others and it is possible for multiple servers to occupy the same node and provide service simultaneously, in which case they can speed up the processing of individual jobs. Setup and processing times are assumed to be interruptible, and the objective is to minimize the long-run average holding cost. We formulate the problem as a Markov decision process and propose two classes of heuristic policies: local partitioning heuristics, which assign demand points to servers using an allocation scheme given by iterated gradient descent, and global scheduling heuristics, which allow greater server flexibility. We conduct extensive numerical experiments to compare the performances of the heuristics across different network topologies and parameter regimes. Results show that the local partitioning approach, particularly the modified 1-stop heuristic, achieves strong performance while maintaining scalability to large problem sizes.

Keywords: Job scheduling, dynamic programming, index heuristics

4.1 Introduction

Queueing control problems have been extensively studied, with the goal of optimally allocating customers or servers in order to optimize a relevant performance measure. However, it is not always possible to find a simple policy that guarantees optimal performance (Biswas (2017)). In this paper we introduce a novel formulation for a multiple-server scheduling problem, in which servers are routed dynamically around a network to process jobs of different types that arrive at random locations. Our primary application area of interest is job scheduling. In this context, the random travel times needed for servers to move between nodes in the network represent setup times to switch between processing different job types, and the objective is to minimize the long-run average holding cost associated with keeping jobs waiting.

Many previous studies have examined the performance of simple server scheduling rules based on index quantities, or *indices*, calculated from the problem parameters. In several classical problems, an optimal policy is a type of *$c\mu$-rule*. This rule prioritizes job classes based on the index $c_i\mu_i$, where c_i is the holding cost per unit time for class i jobs and $1/\mu_i$ represents the mean service time for class i jobs (Buyukkoc et al. (1985); Saghafian and Veatch (2015); Krishnasamy et al. (2018); Lee and Vojnovic (2021); Ozkan (2022)) In a far-reaching extension of the $c\mu$-rule, Van Mieghem (1995) considered a general single-server multiclass queueing system where the unit cost c_i is defined as a convex function of the waiting time, leading to a scheduling policy, following the generalized $c\mu$, that minimizes the total cumulative delay cost over a finite time horizon. Building on this, Mandelbaum and Stolyar (2004) extended the analysis to multi-server systems under heavy traffic, demonstrating that a simple generalized $c\mu$-rule ($Gc\mu$) is asymptotically optimal for delay costs that increase convexly with queue lengths or sojourn times. Subsequently, Atar et al. (2010) and Atar et al. (2011) introduced the $c\mu/\theta$-policy for multi-server systems, an extension of the well-known $Gc\mu$ rule that incorporates abandonment. In this policy, jobs are prioritized based on the index $c_i\mu_i/\theta_i$, where θ_i denotes the abandonment rate for class i. Further extending these results, Larrañaga et al. (2015) and Long et al. (2020) presented a fluid index model for multi-class servers with general convex queue length cost functions. The model generalizes

to the $Gc\mu/\theta$-rule and is shown to be optimal for the case of minimizing the long-run average queue length costs and abandonment penalties.

More recently, Xia et al. (2022) studied a dynamic job assignment problem in queueing systems with a single class of Poisson arrivals and heterogeneous server pools, aiming to minimize the long-run average cost. They proposed a c/μ-rule and proved its optimality for linear cost functions. This policy prioritizes pools with lower operating costs and faster service rates. Expanding on this, Long et al. (2024) generalized the cost functions, leading to the development of the Gc/μ rule, a dynamic priority policy that adjusts priorities across server pools. This approach ensures that customers are routed efficiently, with fair utilization of all pools. The Gc/μ rule is proven to be asymptotically optimal for nonlinear convex costs, and is a simple, effective policy that requires no information about arrival rates or service capacities.

In various domains, multiple-server models are widely applied to optimize resource allocation, enhance efficiency and minimize costs. In manufacturing, Buzacott and Shanthikumar (1993) established foundational work on stochastic models to improve workflows in production lines, focusing on resource allocation and task scheduling. In cloud computing, Armbrust et al. (2010) introduced cloud resource management principles, emphasizing scalable solutions for distributing workloads across multiple servers. Recent studies, such as Yadav et al. (2021), further advanced this area by introducing energy-efficient multi-machine scheduling algorithms, balancing energy consumption and service quality. In call centers, Gans et al. (2003) investigated how to optimize agent utilization through multi-server queueing models. This was later expanded upon by Zhang (2023), who incorporated reinforcement learning to dynamically route calls for better service. Healthcare systems also benefit from these models, with Chan et al. (2021) showing how flexible nurse staffing in emergency departments reduces wait times, while Mandelbaum et al. (2012) proposed a fairness-focused model for bed allocation using real-time data and analytics. In logistics and transportation, Desaulniers et al. (1998) pioneered multi-server models for scheduling vehicles from multiple depots, a concept further extended by Lim et al. (2021) in multi-depot split-delivery vehicle routing problems. More recently, Bhatnagar and Shukla (2023) introduced an optimized multi-server queueing model for real-time transportation, focusing on minimizing costs in unloading and loading service centers. These diverse applications of multi-server models reflect their versatility in improving operational efficiency and service delivery across industries.

Models with multiple queues served by one or more servers have been well-studied in

the literature on *polling systems* (Levy and Sidi (1990)). There are many possible applications of such models in fields such as transportation, communications and production systems (Boon et al. (2011)). However, the formulation of the multi-server scheduling problem as a polling system has garnered relatively little attention (Antunes et al. (2011)). Early studies, such as Kamal and Hamacher (1989) and Marsan et al. (1990), focused on symmetric multiqueue polling systems with multiple identical servers providing non-exhaustive service, handling at most one customer per visit. Both studies considered non-zero switching times between queues, with Marsan et al. (1990) placing particular emphasis on systems with Poisson arrivals. Borst (1995) explored coupled server systems that jointly serve queues and provided exact analysis of waiting times and queue lengths across various configurations, including single-queue systems with varying server numbers, two-queue two-server systems with exhaustive service and exponential service times, and infinite-server systems with multiple queues and deterministic service times.

Several more complex polling models have also been studied. For example, van der Mei and Borst (1997) investigated a polling model where each server visits queues based on its own service order. They employed the power-series algorithm to numerically evaluate and optimize system performance due to the complexity of the model. Lee (2003) examined random polling systems with infinite coupled servers, assuming correlated customer arrivals. The study compared exhaustive and gated service disciplines and found that in systems with many busy servers, gated service outperforms exhaustive service in reducing waiting times. Robert and Roberts (2010) introduced a mean field approximation to evaluate the capacity and stability limits of a multi-server polling systems with constraints on the number of servers attending a queue, and demonstrated asymptotic accuracy of the method. Matveev (2018) presented a fluid model for a polling system with unlimited buffers and divided the system into separate zones for each server. In their model, servers operate within their zones and serve one buffer at a time but can also switch between buffers, with non-zero switch-over times. The study provides a criterion for a scheduling protocol that ensures system stability by keeping the total amount of work in the buffers bounded. In a recent study, Yang et al. (2024) applied a Bi-directional Long Short Term Memory (BiLSTM) neural network to a multi-server polling system for 5G network slicing. This approach optimizes base station selection based on average queue length and delay and can effectively reduce resource wastage without requiring difficult mathematical analysis.

In this paper, we consider a stochastic, dynamic job scheduling problem that extends our

previous work in Tian and Shone (2026b). A key distinction is that, instead of a single server, we now consider a batch of homogeneous servers operating within a network. The network consists of nodes and edges, where designated demand points serve as entry locations for specific job types. Each server processes jobs one at a time, and the time required for a server to move between demand points reflects the setup time needed to switch from processing one job type to another. The network's structure inherently encodes transition costs between different job types, capturing the effort required for a server to adapt to a new type of job.

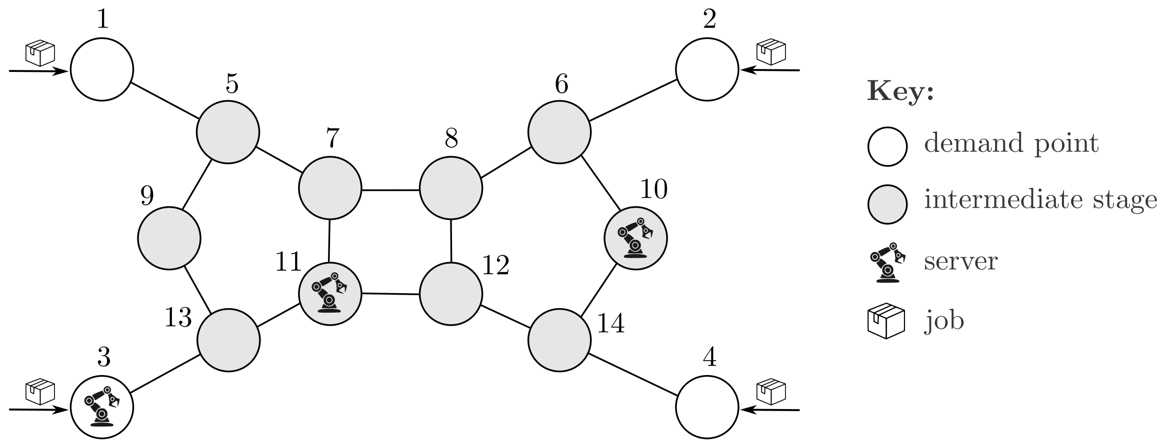

FIGURE 4.1: A network with 4 demand points, 10 intermediate stages and 3 servers. White-colored nodes represent demand points at which new jobs arrive, and gray-colored nodes represent intermediate stages.

Figure 4.1 illustrates an example of our network-based framework. In this figure, the white-colored nodes represent demand points where jobs arrive according to independent Poisson processes, while the gray-colored nodes act as intermediate stages that servers must traverse in order to move between demand points. The intermediate stages are used to abstract distances between demand points. A shortest path may pass through different intermediate nodes (e.g., node 2 to node 3 via node 7 or node 12), representing alternative movement sequences. These stages are not physical locations but encode the distance structure, such as setup requirements, so closer demand points correspond to more similar tasks. Multiple paths therefore allow alternative movements without altering the switching or service logic. As a manufacturing example, suppose a server is set up to process jobs of type“AB” and it needs to be changed to“CD”, which involves changing A to C and changing B to D (these could represent different tools or machinery parts for example). We could do the A-C switch first in which case we go AB-CB-CD, or do the B-D switch first in which case we go AB-AD-CD. In the first case the intermediate

stage is closer to some other job CX (say), while in the second case it is closer to some other job AY. Moreover, the system includes three servers, each occupying a single node at any given time. A server can either stay at its current location or move to an adjacent node. To process a job of type i (for $i \in \{1, 2, 3, 4\}$), a server must be located at node i. We allow two or more servers to occupy the same node, and if multiple servers are at the same demand point then the demand at that location can be processed at an increased rate. Processing times and server switching times between nodes are stochastic, and jobs waiting in the system incur linear holding costs. A formal problem formulation is provided in Section 4.2.

A defining characteristic of our model, as established in Tian and Shone (2026b), is that servers are not bound to predetermined actions and can dynamically adjust their course at any time. This flexibility enables servers to interrupt job processing or alter their transition between job types in response to changing system conditions. Additionally, our network formulation accounts for partial progress made during interrupted setups, which can either reduce or prolong the setup time for a new job type, depending on the network's design. For example, consider a server currently at node 11 (shown in Figure 4.1), intending to move to node 4 to process jobs of type 4. To reach node 4, the server must first pass through nodes 12 and 14 (assuming that it follows the shortest path). The time to switch between adjacent nodes is assumed to be randomly distributed (as detailed in Section 4.2). However, if the server arrives at node 12 and decides to change its course toward node 2, perhaps because new jobs have arrived at node 2, the setup for type 4 jobs is interrupted. The server now shifts its focus to preparing for type 2 jobs. Despite this change in course, the progress made toward node 4 (i.e. the switch to node 12) still enables the server to arrive at node 2 more quickly than if it had simply remained at node 11. In contrast, if the server moves toward node 3 instead, the progress made in setting up type 4 jobs delays the setup for type 3 jobs, as the earlier progress does not align with the new job type's requirements.

In this paper we adopt a Markov Decision Process (MDP) framework, which naturally models sequential decision-making in uncertain environments. Nonetheless, exact solution methods are infeasible for large-scale settings due to state space explosion, so we require scalable heuristics. Prior research has employed MDPs across diverse scheduling and allocation problems, though often under more restrictive or structurally different assumptions. For instance, Zhang et al. (2017) examined a queueing system where jobs with heterogeneous service requirements arrive stochastically, and decisions focus on job

selection at each time step. Their machines, however, remain fixed rather than being reassigned dynamically across tasks. Zheng et al. (2019) explored job scheduling in mobile cloud networks using a bilayer MDP, aiming to reduce latency and service costs by controlling server activation and job routing. Their servers differ in processing power, and while they introduce dynamic job scheduling, server positions are static. In a different application domain, He et al. (2020) developed a generic MDP framework and reinforcement learning approach for scheduling agile Earth observation satellites. In their model, satellites with fixed imaging capacities are assigned to incoming observation tasks based on predefined reward rules and orbital constraints. While their work addresses stochastic task arrivals and uses approximate dynamic programming to cope with policy complexity, the model assumes a finite state space and task assignment structure, differing from our setting involving continuous arrivals, flexible resource movement, and an infinite state space. More recently, O'Reilly et al. (2024) considered hospital ward allocation as an MDP, optimizing patient-to-ward assignments to reduce long-term costs. Their approach models heterogeneity across service units via priority tiers, while our formulation allows for more adaptive and homogeneous server deployment.

In our previous work (Tian and Shone (2026b)), we developed a single-server dynamic scheduling model with similarities to the earlier model of Duenyas and Van Oyen (1996), which assumes a system of parallel queues with independent Poisson job arrivals. While the model in Duenyas and Van Oyen (1996) incorporates random processing and switching times over an infinite time horizon, it does not allow for interruptions or a network-based representation. By contrast, our current work builds on the flexibility introduced in Tian and Shone (2026b) while expanding the scope to multiple servers, requiring a more structured allocation strategy. This extension necessitates new methodologies, such as network decomposition to manage job allocations and heuristic strategies for simultaneous server movement. Despite these differences, the index heuristics developed in Tian and Shone (2026b) provide the foundation for our index-based approach in Section 4.3, where we partition the network into multiple single-server scheduling areas. Specifically, we adapt and extend the 1-stop heuristic framework from Tian and Shone (2026b) to handle the added complexity of multiple servers. Within this local partitioning scheme, we also introduce a cycle-based polling heuristic as a benchmark, where each server operates within its designated area, following a predefined sequence of demand points based on shortest distances.

The main contributions of our paper are as follows:

- We provide a novel, network-based formulation for a multiple-server stochastic, dynamic job scheduling problem, where setup and processing times are sequence-dependent and interruptible.
- We propose an allocation approach that partitions the overall system demand by assigning demand points to servers according to proportions that we aim to optimize using iterated gradient descent.
- We propose two heuristics from the local partitioning perspective: the *polling heuristic* and *modified 1-stop heuristic*.
- We propose two heuristics from the global scheduling perspective, referred to as the *flexible heuristic* and the *exclusive heuristic*.
- We present the results of extensive numerical experiments comparing the performances of the various heuristics under different network layouts and parameter settings. We show that, overall, the local partitioning method with the modified 1-stop heuristic achieves the strongest performance.

The rest of the paper is organized as follows. In Section 4.2 we formulate our stochastic, dynamic job scheduling problem as an MDP. In Section 4.3, we provide details of our heuristic allocation method based on local partitioning and introduce two separate heuristics based on this method. In Section 4.4 we introduce two simpler heuristics from the global scheduling perspective and use these as benchmarks within our numerical study. Finally, our concluding remarks can be found in Section 4.5.

4.2 Problem formulation

The problem is formulated on a connected graph, referred to as a *network*, which consists of a node set V and an edge set E. The nodes in V are partitioned into two subsets: the *demand points*, denoted by $D \subseteq V$, where service is required, and the *intermediate stages*, denoted by $N := V \setminus D$, where no service is needed but servers may traverse. The number of demand points is $d = |D|$, and the number of intermediate stages is $n = |N|$. Demand points are indexed as $1, 2, \ldots, d$, and intermediate stages are indexed as $d + 1, d + 2, \ldots, d + n$.

The network is served by a set of homogeneous servers, denoted by M, where the total number of servers is $m = |M|$, labeled $1, 2, \ldots, m$. We assume that $m < d$. Each server independently moves through the network and is always positioned at a single node in V. It is possible for multiple servers to be located at the same node. At each decision epoch, a server can either remain at its current location or move to an adjacent node. If a server is located at a node with l neighbors, it has $l + 1$ possible actions, including staying in place.

Jobs arrive at each demand point $i \in D$ according to a Poisson process with an arrival rate $\lambda_i > 0$, and arrivals at different demand points are independent. Each job that arrives at demand point $i \in D$ remains there until it is served, accumulating a *holding cost* $c_i > 0$ per unit of time. If $k \geq 1$ servers are located at a demand point i with $x_i \geq 1$, then these servers are assumed to be providing service to jobs of type i. We assume that jobs are processed in a strict first-come-first-served order, and this means that if multiple servers are at a non-empty demand point then they are collectively processing the same job (i.e. the one that arrived first). If only one server is at node i then jobs are processed at an instantaneous *service rate* $\mu_i > 0$, but if multiple servers are present then the rth server (for $r \geq 1$) increases the total service rate by $(1-\alpha)^{r-1}\mu_i$, where $\alpha \in [0, 1]$ is a fixed parameter that reflects the diminishing service efficiency as more servers are allocated to the same demand point. This implies that the total service rate at demand point i with k servers present is

$$\sum_{r=1}^{k} (1-\alpha)^{r-1}\mu_i = \begin{cases} \alpha^{-1}\mu_i[1-(1-\alpha)^k], & \text{if } \alpha > 0, \\ k\mu_i, & \text{if } \alpha = 0. \end{cases}$$

which is a concave function of k. For example, if $\mu_i = 2$ and $\alpha = 0.1$ then the total service rates with 1, 2 and 3 servers present are 2, 3.8 and 5.42 respectively. In many practical systems, choosing $\alpha > 0$ is realistic, as parallel service at a single queue may lead to coordination overhead, interference, or contention for shared resources. For example, in healthcare triage, multiple staff members may experience diminishing productivity due to limited physical space or task duplication, while in logistics or technical support, assigning additional personnel to the same task often yields only marginal improvements. Hence, $\alpha > 0$ captures such realistic inefficiencies and encourages more spatially distributed deployments. A server is said to be *idle* if it stays at a demand point with no queued jobs or remains at an intermediate stage.

Servers move between adjacent nodes at an instantaneous rate $\tau > 0$, referred to as the *switching rate*. Switching and processing times are independent of the job arrival

processes at demand points. For simplicity, we assume a uniform switching rate across all adjacent nodes, meaning that the expected switching time between two arbitrary nodes in the network is proportional to the number of edges that must be traversed. If a server moves from one node to a non-adjacent node without any interruption, then the assumption of instantaneous switching rates implies that the times to switch between pairs of adjacent nodes are independent and exponentially distributed and hence the total switching time follows an Erlang distribution. We also note that the results and heuristic methods in this paper can be extended to cases where the switching rates are edge-dependent, in which case we have a more general phase-type distribution for the (uninterrupted) amount of time to switch between non-adjacent nodes.

We now formulate the system as a continuous-time Markov decision process (MDP) based on the assumptions described earlier. Herein, we will follow similar notational conventions to those in Tian and Shone (2026b). The state space of the MDP is given by

$$S := \{((v_1, \dots, v_m), (x_1, \dots, x_d)) \mid v_s \in V \text{ for } s \in M,\ x_i \geq 0 \text{ for } i \in D\},$$

where v_s denotes the node currently occupied by server s, and x_i represents the number of jobs in the queue at demand point $i \in D$. The first component, $(v_1, \dots, v_m)$, captures the positions of all m servers in the network, while the second component, $(x_1, \dots, x_d)$, represents the queue lengths at the demand points.

For simplicity, we will use vectors such as $\mathbf{x}$ to represent generic states in S. The notation $\mathbf{v}(\mathbf{x}) = (v_1(\mathbf{x}), ..., v_m(\mathbf{x}))$ denotes the locations of all servers under state $\mathbf{x} \in S$, while $v_s(\mathbf{x})$ specifically represents the location of server s. We sometimes write $\mathbf{v}$ instead of $\mathbf{v}(\mathbf{x})$ and v_s instead of $v_s(\mathbf{x})$ when the state associated with $\mathbf{v}$ or v_s is clear. Since a server cannot process jobs while switching and can serve at most one demand point at a time, the total transition rate under any state can be bounded above by

$$\sum_{i \in D} \lambda_i + m \times \max\{\mu_1, \dots, \mu_d, \tau\}.$$

Using the technique of uniformization (Serfozo (1979)), we transform the system into a discrete-time counterpart that evolves in time steps of size

$$\Delta := \left(\sum_{i \in D} \lambda_i + m \times \max\{\mu_1, \dots, \mu_d, \tau\} \right)^{-1}. \tag{4.1}$$

Let $R_s(\mathbf{x})$ denote the set of nodes adjacent to $v_s(\mathbf{x})$, the location of server s under state $\mathbf{x}$. The action set available in state $\mathbf{x} \in S$ is given by

$$A_{\mathbf{x}} = \prod_{s \in M} A_{\mathbf{x}}^s, \quad \text{where} \quad A_{\mathbf{x}}^s = \{v_s(\mathbf{x})\} \cup R_s(\mathbf{x}).$$

Each action vector $\mathbf{a}(\mathbf{x}) = (a_1(\mathbf{x}), \ldots, a_m(\mathbf{x})) \in A_{\mathbf{x}}$ represents the movement decision for all m servers, where $a_s(\mathbf{x})$ specifies the node that server s moves to next, with $a_s(\mathbf{x}) = v_s(\mathbf{x})$ indicating that server s remains at its current position. Then, at each time step, for any two distinct states

$$\mathbf{x} = (\mathbf{v}(\mathbf{x}), (x_1, \ldots, x_d)), \quad \mathbf{y} = (\mathbf{v}(\mathbf{y}), (y_1, \ldots, y_d)),$$

the transition probability from $\mathbf{x}$ to $\mathbf{y}$ under action $\mathbf{a}(\mathbf{x})$ is given by

$$p_{\mathbf{x},\mathbf{y}}(\mathbf{a}(\mathbf{x})) := \begin{cases} \lambda_i \Delta, & \text{if } y_i = x_i + 1 \text{ for some } i \in D \text{ and } y_j = x_j \text{ for all } j \neq i, \\ & \text{and } v_s(\mathbf{x}) = v_s(\mathbf{y}) \text{ for all } s \in M, \\ \sum_{r=1}^{k} (1-\alpha)^{r-1} \mu_i \Delta, & \text{if } y_i = x_i - 1 \text{ for some } i \in D \text{ and } y_j = x_j \text{ for all } j \neq i, \\ & \text{and } |\{s \in M \mid v_s(\mathbf{x}) = i,\ a_s(\mathbf{x}) = v_s(\mathbf{x})\}| = k \geq 1, \\ & \text{and } v_s(\mathbf{x}) = v_s(\mathbf{y}) \text{ for all } s \in M, \\ \tau \Delta, & \text{if } y_i = x_i \text{ for all } i \in D, \\ & \text{and } |\{s^* \in M \mid v_{s^*}(\mathbf{x}) \neq v_{s^*}(\mathbf{y}),\ a_{s^*}(\mathbf{x}) = v_{s^*}(\mathbf{y})\}| = 1, \\ & \text{and } v_s(\mathbf{x}) = v_s(\mathbf{y}) \text{ for } s \in M \setminus \{s^*\}, \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

The probability of remaining in the same state is $p_{\mathbf{x},\mathbf{x}}(\mathbf{a}(\mathbf{x})) = 1 - \sum_{\mathbf{y} \neq \mathbf{x}} p_{\mathbf{x},\mathbf{y}}(\mathbf{a}(\mathbf{x}))$. We note that the transition probabilities in (4.2) imply that at most one event (either a new job arrival, or a job completion at one demand point, or a successful switch by one server) can happen at any discrete time step. This assumption is consistent with the theory of uniformization (Serfozo (1979)) and is analogous to the property of continuous-time systems that there is always a random, non-zero amount of time between any two state transitions.

At each time step, the system incurs a single-step cost, denoted as $f(\mathbf{x})$, which is calculated by summing the holding costs of jobs that are either still waiting to be processed or currently being processed. Specifically, the cost is expressed as:

$$f(\mathbf{x}) := \sum_{i \in D} c_i x_i. \tag{4.3}$$

Let θ represent a decision-making policy for our MDP. If the action $\theta(\mathbf{x}) \in A_{\mathbf{x}}$ is consistently selected for each state $\mathbf{x} \in S$ and non-randomized, then the policy is termed *stationary* and *deterministic* (Puterman (1994)). The expected long-run average cost per unit time, or simply *average cost*, under policy θ, with initial state $\mathbf{x}_0 \in S$, is defined as:

$$g_\theta(\mathbf{x}) = \liminf_{t \to \infty} t^{-1} \mathbb{E}_\theta \left[\sum_{k=0}^{t-1} f(\mathbf{x}_k) \,\middle|\, \mathbf{x}_0 = \mathbf{x} \right], \tag{4.4}$$

where $\mathbf{x}_k$ denotes the state of the system at time step $k \in \mathbb{N}_0$. According to the theory of uniformization, the discretized system under a stationary policy θ will yield the same average cost $g_\theta(\mathbf{x})$ as the continuous-time system, in which actions are taken every time the system transitions to a new state. The goal of the problem is to minimize the long-run average cost $g_\theta(\mathbf{x})$, which means finding a policy θ^* such that:

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

where Θ is the set of all admissible policies.

Since S is infinite, the average cost $g_\theta(\mathbf{x})$ remains finite only if the system is stable under policy θ. Stability, in this context, requires that θ induces an ergodic Markov chain on S, ensuring that the system does not grow unbounded over time. We define the *worst case traffic intensity* of the system as:

$$\begin{aligned} \rho &= \sum_{i \in D} \left(\frac{\lambda_i}{\sum_{r=1}^{m} (1-\alpha)^{r-1} \mu_i} \right) \\ &= \begin{cases} \sum_{i \in D} \dfrac{\lambda_i}{m\mu_i}, & \text{if } \alpha = 0, \\ \sum_{i \in D} \dfrac{\alpha \lambda_i}{(1-(1-\alpha)^m)\mu_i}, & \text{if } \alpha > 0. \end{cases} \end{aligned}$$

We refer to ρ as the 'worst case' traffic intensity because it is derived using a conservative assumption that, at any given time step, jobs at demand point $i \in D$ are either processed

at a rate $\sum_{r=1}^{m}(1-\alpha)^{r-1}\mu_i$ or not at all. The former rate can only occur if all servers are located at i, in which case the total service rate is discounted by the maximum possible factor. We conjecture that if $\rho < 1$ then there exists a stationary policy under which the system is stable, and that it should be possible to prove this statement by generalizing the stability proof in Tian and Shone (2026b). However, we do not yet have a rigorous proof of this.

In theory, it is possible to use the 'approximating sequences' technique from Sennott (1997) to compute an optimal policy for our infinite-state MDP using value iteration, provided that there exists such a policy under which the system is stable. However, this method faces significant challenges in practice. Specifically, the "curse of dimensionality" is a major obstacle, making it infeasible to compute optimal policies unless the dimensions of the state space are very small. As the number of demand points and/or servers increases, the state space expands exponentially, raising the computational burden to an impractical level. In the next section we propose heuristic methods, which provide efficient approximations to the optimal policy. These heuristics are designed to be computationally manageable while still delivering strong performance across a wide range of system configurations.

4.3 Index heuristics

To address the complexities introduced by multiple servers in the dynamic job scheduling problem (as depicted in Figure 4.1), we develop a heuristic approach in two phases. The first phase focuses on allocating demand to servers in such a way that each demand point $i \in D$ has its total demand rate (represented by λ_i) divided between the m servers according to a set of proportions, which we aim to heuristically optimize. This effectively divides the network into a set of m 'regions', with each region assigned to a single server. The regions may be overlapping because demand points may be shared between two or more servers. In the second phase, heuristics are developed to schedule the servers dynamically within their assigned regions.

We define $\mathbf{p} := (\mathbf{p}_1, \mathbf{p}_2, ..., \mathbf{p}_m)$ as an *allocation vector*, where each element $\mathbf{p}_s$ is itself a vector of the form $(p_{s1}, ..., p_{sd})$, specifying the proportion of demand from each point

$i \in D$ assigned to server $s \in M$. This allocation must satisfy the following constraints:

$$\text{(i) Bounded proportions:} \quad 0 \le p_{si} \le 1, \qquad \forall s \in M,\ i \in D, \tag{4.5}$$

$$\text{(ii) Full allocation:} \quad \sum_{s \in M} p_{si} = 1, \qquad \forall i \in D, \tag{4.6}$$

$$\text{(iii) Server capacity:} \quad \sum_{i \in D} \frac{p_{si}\lambda_i}{\mu_i} < 1, \qquad \forall s \in M, \tag{4.7}$$

$$\text{(iv) Sharing degree:} \quad m_i := \left|\{s \in M \mid p_{si} > 0\}\right|, \qquad \forall i \in D, \tag{4.8}$$

$$\text{(v) System stability:} \quad \sum_{i \in D} \left(\frac{\lambda_i}{\sum_{r=1}^{m_i} (1-\alpha)^{r-1}\mu_i} \right) < 1, \qquad \text{if } \sum_{i \in D} m_i > d. \tag{4.9}$$

Constraints (4.5)–(4.6) ensure that the allocation proportions are well-defined and that each demand point is fully distributed among the servers. Constraint (4.7) is necessary in order to prevent individual servers from becoming overloaded. The quantity m_i in (4.8) counts the number of servers that share demand point i, and the system-wide constraint (4.9) limits the worst case traffic intensity under decaying efficiency when demand is split across multiple servers.

Given an allocation vector $\mathbf{p}$, we then define a *local* system state for each server $s \in M$ as

$$\mathbf{x}_s = (v_s(\mathbf{x}), (x_{s1}, \ldots, x_{sd})), \tag{4.10}$$

where $v_s(\mathbf{x})$ is the server's location and, for each $i \in D$, x_{si} is the number of jobs at i that have been assigned to server s. These local states are updated dynamically during the system's evolution in the following way. Each time a new job arrives at demand point $i \in D$, we increase exactly one of the variables x_{si} by one. This is done according to a random draw, where the probability of x_{si} being increased (for $s = 1, ..., m$) is p_{si}. Furthermore, each time a service completion occurs at demand point i, we decrease exactly one of the x_{si} variables by one. This is done in such a way that the servers located at i are ranked according to a pre-defined ordering of the servers, and the probability that the jth server in the ranking has its local variable decreased by one is $(1-\alpha)^{j-1}\mu_i / [\sum_{r=1}^{k} (1-\alpha)^{r-1}\mu_i]$, where $j \in \{1, ..., k\}$ and k is the number of servers present. Updating the variables x_{si} in this way ensures that the constraint $\sum_{s \in M} x_{si} = \mathbf{x}_i$ holds at all times during the evolution of the system.

One point that should be clarified is that, although our heuristic method involves dynamically distributing the jobs in the system among the servers, we do not require that

specific jobs must be served by specific servers according to one-to-one matches. So if a local variable x_{si} increases by one (following the arrival of a job at demand point i), this does not mean that the newly-arrived job can only be processed by server s. Instead, it just means that the total number of type-i jobs allocated to server s increases by one. Indeed, recall from Section 4.2 that we assume a 'first-come-first-served' rule, under which the next job to be processed at demand point $i \in D$ must be the job that arrived first. Therefore, it is not possible to allow a server to 'skip' a job in order to serve one of the specific jobs allocated to it.

Given any allocation that satisfies the constraints (4.5)-(4.9), we can define $\mathcal{D}_s$ (for each $s \in M$) as the set of demand points that server s has partial or full responsibility for, which we refer to as the *region* for server s:

$$\mathcal{D}_s := \{i \in D \mid p_{si} > 0\}.$$

By updating 'local states' for the different servers as described above, we have a way of dynamically distributing the jobs in the system among the servers. However, we have not yet specified how servers should dynamically process the jobs within their own regions. In this section we propose two different methods for routing the servers within their assigned regions. The first method is a simple 'polling' heuristic, under which each server visits the demand points within its region in a fixed cyclic order. The second method is a more sophisticated index heuristic, under which each server moves between the demand points in its region according to the 1-stop heuristic introduced in Tian and Shone (2026b), and we also allow servers to temporarily visit other regions if there are no jobs waiting within their own regions.

Our heuristic approach aims to find the allocation vector $\mathbf{p}$ that enables the best system performance. However, there are two major difficulties with this: (i) the set of possible allocations is infinite and uncountable; (ii) after deciding a particular allocation $\mathbf{p}$, it is computationally expensive to evaluate the performance of the system under that allocation (even if we assume that the servers follow a simple pattern of behavior, such as a cyclic polling heuristic), since this can only be done approximately using simulation experiments. To address the second difficulty, we consider a fluid approximation of the system dynamics. Under the fluid approximation, we take an allocation vector $\mathbf{p}$ as an input and then decompose the problem into m independent single-server subproblems. For $s \in M$, the corresponding subproblem has the following properties:

1. The state variables x_{si} for $i \in \mathcal{D}_s$ are allowed to be non-integer;
2. For each $i \in \mathcal{D}_s$, the variable x_{si} increases at a continuous rate λ_{si} per unit time, where $\lambda_{si} = p_{si}\lambda_i$;
3. If the server is at a demand point $i \in \mathcal{D}_s$ with $x_{si} > 0$, then it provides service at a rate μ_i, implying that the net rate of change in x_{si} is $\mu_i - \lambda_{si}$ per unit time;
4. If the server switches between two adjacent nodes then the switch takes exactly $1/\tau$ time units to complete.

We also assume that server s visits the demand points $\mathcal{D}_s$ in a fixed cyclic pattern, serving each demand point exhaustively on each visit, so that switching and processing times are never interrupted. By making these assumptions we can evaluate the performance of a system under a particular allocation with minimal computational effort. This can be done as follows: firstly, in order to determine the cyclic pattern (i.e. the order of visiting demand points) that a server $s \in M$ should follow within its own region $\mathcal{D}_s$, we solve a Traveling Salesman Problem (TSP) in order to minimize the total switch distance. Let $d_s = |\mathcal{D}_s|$ be the number of demand points assigned to server s, and let $\delta(i, j)$ denote the length of the shortest path (in terms of the number of nodes that must be traversed) between two points $i, j \in \mathcal{D}_s$. To model the server's route, we define binary decision variables y_{ij}, where $y_{ij} = 1$ if the server travels directly from demand point i to demand point j, and $y_{ij} = 0$ otherwise. The TSP objective can be expressed as

$$\text{Minimize} \sum_{i \in \mathcal{D}_s} \sum_{j \in \mathcal{D}_s} \delta(i, j) y_{ij}$$

such that

$$\sum_{\substack{j \in \mathcal{D}_s \\ j \neq i}} y_{ij} = 1, \quad \forall i \in \mathcal{D}_s, \tag{4.11}$$

$$\sum_{\substack{j \in \mathcal{D}_s \\ j \neq i}} y_{ji} = 1, \quad \forall i \in \mathcal{D}_s, \tag{4.12}$$

$$1 \leq l_i \leq d_s - 1, \quad \forall i \in \mathcal{D}_s \setminus \{i_1\}, \tag{4.13}$$

$$l_i - l_j + d_s \cdot y_{ij} \leq d_s - 1, \quad \forall i, j \in \mathcal{D}_s, \ \ i \neq j, \tag{4.14}$$

$$y_{ij} \in \{0, 1\}, \quad \forall i, j \in \mathcal{D}_s, \ i \neq j. \tag{4.15}$$

where (4.11) ensures that the server leaves each demand point once, (4.12) ensures that the server visits each demand point exactly once, (4.13) introduces auxiliary integer variables l_i representing the relative position of demand point i in the tour (excluding a designated reference node i_1 that anchors the tour and breaks symmetry), (4.14) prevents the formation of subtours by enforcing an ordering of the positions of demand points in the tour, and (4.15) ensures that the decision variables y_{ij} are binary.

The solution to this problem yields an optimal sequence of demand points $\mathcal{O}_s = (i_1, i_2, \ldots, i_{d_s})$, and the total travel distance can be calculated as

$$q_s := \sum_{k=1}^{d_s - 1} \delta(i_k, i_{k+1}) + \delta(i_{d_s}, i_1).$$

Once the optimal sequence $\mathcal{O}_s$ for server s is determined, we can calculate the maximum number of waiting jobs at demand point i_k (denoted by $\bar{x}_{i_k}$), and the corresponding amount of service time (denoted by T_{i_k}) in the fluid approximation as follows:

$$\bar{x}_{i_k} = \lambda_{s i_k} \left(\sum_{\substack{j=1 \\ j \neq k}}^{d_s} T_{i_j} + q_s/\tau \right), \quad T_{i_k} = \frac{\bar{x}_{i_k}}{\mu_{i_k} - \lambda_{s i_k}},$$

where T_{i_j} is the amount of time needed to process all jobs at demand point $i_j \neq i_k$ on the server's route $\mathcal{O}_s$ after server s arrives at i_j. Although $\bar{x}_{i_k}$ and T_{i_k} appear to depend on each other, these equations should be interpreted as a system of fixed-point equations rather than as a sequential computation. Substituting the expression for T_{i_k} into the first equation yields a set of coupled nonlinear equations in $\{\bar{x}_{i_k}\}_{k=1}^{d_s}$.

In the fluid approximation, this system can be solved either analytically in simple cases or numerically using fixed-point iteration. Specifically, one may initialize $\bar{x}_{i_k}$ (or equivalently T_{i_k}) with feasible values and iteratively update the quantities until convergence. Under the stability condition $\lambda_{si_k} < \mu_{i_k}$, the iteration converges to a unique solution.

It is clear from Figure 4.2 that the average number of jobs at node i_k is $\bar{x}_{i_k}/2$. Hence, the fluid cost incurred by server s is given by

$$C_s = \sum_{k=1}^{d_s} c_{i_k} \bar{x}_{i_k}/2.$$

The total operation cost under allocation $\mathbf{p}$ is then computed as

$$C = \sum_{s \in M} C_s.$$

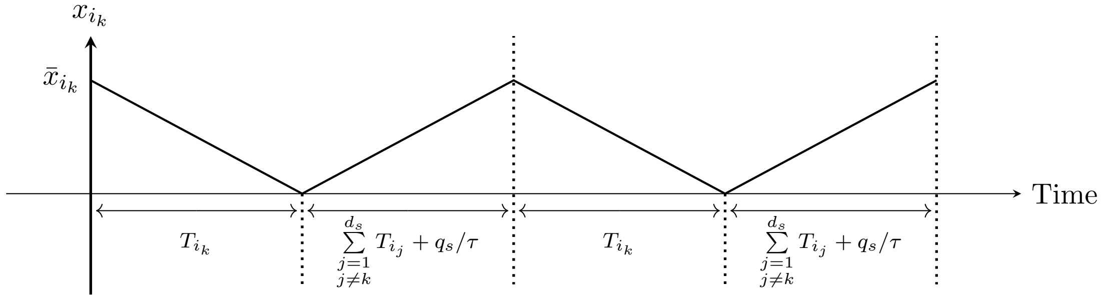

FIGURE 4.2: The number of waiting jobs at demand point i_k evolves with time.

Algorithm 4.1 Optimization of Allocation between Servers and Demand Points

1: **Input:** $D = \{1, 2, \ldots, d\}$, $M = \{1, 2, \ldots, m\}$, σ
2: **for** each server $s \in M$ **do**
3: $\quad \mathbf{p}_s = (p_{s1}, \ldots, p_{sd}) \leftarrow$ initial allocation
4: $\quad \mathcal{D}_s \leftarrow \{i \mid p_{si} > 0,\ i \in D\}$
5: $\quad \mathcal{O}_s, C_s \leftarrow \text{TSP}(\mathcal{D}_s), \text{fluid-cost-approx}(\mathcal{O}_s)$
6: **end for**
7: $C \leftarrow \sum_{s \in M} C_s$
8: **while** True **do**
9: $\quad$ Select $i^* \in D$
10: $\quad$ Select $s, s' \in M$ with $p_{si^*} > 0$, $p_{s'i^*} < 1$
11: $\quad$ **for** each $i \in D$ **do**
12: $\quad\quad$ **if** $i = i^*$ **then**
13: $\quad\quad\quad p'_{si^*} \leftarrow p_{si^*} - \sigma$
14: $\quad\quad\quad p'_{s'i^*} \leftarrow p_{s'i^*} + \sigma$
15: $\quad\quad$ **else**
16: $\quad\quad\quad p'_{si} \leftarrow p_{si}$
17: $\quad\quad\quad p'_{s'i} \leftarrow p_{s'i}$
18: $\quad\quad$ **end if**
19: $\quad$ **end for**
20: $\quad \mathbf{p'_s} \leftarrow (p'_{s1}, \ldots, p'_{sd})$, $\mathbf{p'_{s'}} \leftarrow \left(p'_{s'1}, \ldots, p'_{s'd}\right)$
21: $\quad \mathcal{D}'_s \leftarrow \{i \mid p'_{si} > 0,\ i \in D\}$, $\mathcal{D}'_{s'} \leftarrow \{i \mid p'_{s'i} > 0,\ i \in D\}$
22: $\quad$ **if** Constraints (4.5)-(4.9) are satisfied under $\mathbf{p'_s}$, $\mathbf{p'_{s'}}$ **then**
23: $\quad\quad \mathcal{O}'_s, C'_s \leftarrow \text{TSP}(\mathcal{D}'_s), \text{fluid-cost-approx}(\mathcal{O}'_s)$
24: $\quad\quad \mathcal{O}'_{s'}, C'_{s'} \leftarrow \text{TSP}(\mathcal{D}'_{s'}), \text{fluid-cost-approx}(\mathcal{O}'_{s'})$
25: $\quad\quad C' \leftarrow C'_s + C'_{s'} + \sum_{s \in M \setminus \{s, s'\}} C_s$
26: $\quad\quad$ **if** $C' < C$ **then**
27: $\quad\quad\quad \mathbf{p_s} \leftarrow \mathbf{p'_s}$, $\mathbf{p_{s'}} \leftarrow \mathbf{p'_{s'}}$
28: $\quad\quad\quad C \leftarrow C'$, $\mathcal{D}_s \leftarrow \mathcal{D}'_s$, $\mathcal{D}_{s'} \leftarrow \mathcal{D}'_{s'}$
29: $\quad\quad\quad \mathcal{O}_s \leftarrow \mathcal{O}'_s$, $\mathcal{O}_{s'} \leftarrow \mathcal{O}'_{s'}$
30: $\quad\quad$ **end if**
31: $\quad$ **end if**
32: $\quad$ **if** no improvement after 200 iterations **then**
33: $\quad\quad$ **stop**
34: $\quad$ **end if**
35: **end while**
36: **Output:** $\mathbf{p}_s$, $\mathcal{D}_s$, $\mathcal{O}_s$ for all $s \in M$

Building on the fluid model for evaluating allocation costs, Algorithm 4.1 implements an iterative improvement process through gradient descent. In each iteration, a demand point is randomly selected, and a small proportion (e.g. $\sigma = 0.01$) of its allocation is swapped between two servers. After the swap, the operation cost is recalculated using the fluid approximation. If the new allocation reduces the total cost, it is accepted; otherwise, it is discarded. This iterative process continues until no cost improvements are observed for K consecutive iterations (we use $K = 200$ in our experiments). To reduce the risk that the algorithm converges to a local optimum but not a global one, the gradient descent procedure is repeated 10 times with different initializations, and the best allocation (giving the lowest cost C) is chosen from among the 10 runs.

Both of the heuristic methods that we propose in this section involve computing an optimal allocation vector $\mathbf{p}$ using Algorithm 4.1 as a starting point. We describe the two different heuristic approaches in the next two subsections.

4.3.1 Polling heuristic

In this heuristic, we begin by obtaining an allocation vector $\mathbf{p}$ using Algorithm 4.1. This yields a set of demand points $\mathcal{D}_s$ and a sequence $\mathcal{O}_s$ for each server $s \in M$. We then simply allow each server to follow the sequence $\mathcal{O}_s$ obtained from the algorithm. We assume that servers only serve the jobs included in their local state $\mathbf{x}_s$, and they serve these jobs in an exhaustive way, which means that server s remains at demand point $i \in \mathcal{D}_s$ if and only if $x_{si} > 0$. If a server is at an empty demand point or at an intermediate stage, then it moves to the next demand point in the sequence. For example, suppose there are 4 demand points ($d = 4$), $\mathcal{D}_s = \{2, 3, 4\}$ and $\mathcal{O}_s = (2, 4, 3)$ for some $s \in M$. Then if the local state is $(2, (0, 1, 2, 1))$, the server remains at node 2, but if the local state is $(2, (0, 0, 2, 1))$ then it takes the first step towards node 4.

We make the following remarks about the polling heuristic:

1. The polling heuristic gives a nonstationary policy, in the sense that even if the allocation vector $\mathbf{p}_s$ and the local state $\mathbf{x}_s$ are given, the action of server s still depends on the history of the process. This is because of the issue noted in Tian and Shone (2026b) that if the server is at an intermediate stage, then it may need to know which demand point was visited last in order to decide which demand point to move to next.

2. The cost C_s given by Algorithm 4.1 is generally not a good estimate of the 'true' system cost for server s, i.e. the long-run average holding cost for jobs assigned to it. This is because C_s is calculated using a fluid approximation that does not model the true stochastic dynamics of the system and also ignores the parameter α, which captures the effect of simultaneous service by multiple servers at the same demand point. However, the polling system may still perform well if there is a good correlation between the costs given by the fluid approximation and the 'true' system costs.

4.3.2 Modified 1-stop heuristic

As discussed in Section 4.1, the multiple-server problem that we consider in this paper is an extension of our previous work (Tian and Shone (2026b)), which considered a single-server scheduling problem defined on a network. The heuristic method that we propose in this subsection is a generalization of the 1-stop heuristic in the aforementioned work (which was based on the earlier work of Duenyas and Van Oyen (1996)) to multiple servers. As explained in Tian and Shone (2026b), in order to motivate the heuristic approach it is necessary to establish a formal equivalence between minimizing costs and maximizing rewards in the context of discounted problems, and then use this to define a reward function for the undiscounted problem.

Given the independence and homogeneity of servers, we define a reward function for each server $s \in M$ as follows. For a system state $\mathbf{x} = (\mathbf{v}(\mathbf{x}), (x_1, \ldots, x_d))$ and an action $a_s \in A_{\mathbf{x}}^s$, the reward function $r_s(\mathbf{x}, a_s)$ is given by:

$$r_s(\mathbf{x}, a_s) = \begin{cases} c_i(1-\alpha)^{k-1}\mu_i, & \text{if } v_s = i \in D,\ x_i \geq 1,\ a_s = v_s, \text{ and } \sum_{s' \leq s,\, s' \in M} \mathbb{I}(v_{s'} = i) = k, \\ 0, & \text{otherwise.} \end{cases} \tag{4.16}$$

Here, k denotes the position of server s among servers currently located at demand point i, according to the predefined order of servers in M. For example, if servers 2 and 4 are located at node i then we refer to these as the 1st and 2nd servers at i respectively. The total single-step reward function for all servers is then given by

$$r(\mathbf{x}, \mathbf{a}) = \sum_{s \in M} r_s(\mathbf{x}, a_s). \tag{4.17}$$

The reward-based formulation (4.17) is advantageous because it depends only on the servers' current locations and actions, making it computationally efficient. In contrast, the cost-based formulation (4.3) requires tracking the number of jobs at each demand point. The reward-based approach is therefore more convenient when developing index heuristics, which assign easily computable scores (or indices) to decision options in any given state. In this subsection, the modified 1-stop heuristic is derived using the reward-based formulation (4.17). However, in Section 4.4, we compare different policies based on the original cost formulation from Section 4.2.

The modified 1-stop heuristic that we propose in this subsection works by obtaining an allocation vector $\mathbf{p}$ using Algorithm 4.1 as a starting point. This yields a set of demand points $\mathcal{D}_s$ for each $s \in M$. Then, at each time step, server $s \in M$ calculates indices ψ_j for each demand point $j \in \mathcal{D}_s$, and selects an action by prioritizing the demand point j that maximizes ψ_j (provided that it meets eligibility conditions). The indices are calculated in a similar way to those in (Tian and Shone (2026b)), except that for the multiple-server case we incorporate the following modifications: (i) the index for traveling to a demand point which already has at least one server present is modified in order to take into account the reduction in the total service rate (determined by the parameter α); (ii) if $x_{si} = 0$ for all demand points in $i \in \mathcal{D}_s$, but there is at least one demand point $j \notin \mathcal{D}_s$ such that $\mathbf{x}_j > 0$, then it moves to the nearest non-empty demand point (call this j^*) and begins processing a job there (even though $x_{sj^*} = 0$); (iii) if server s is processing a job at a demand point j^* such that $x_{sj^*} = 0$ (as described in (ii)) but a new job arrives and causes x_{si} to be positive for some $i \in \mathcal{D}_s$, then server s immediately reverts to processing jobs within its own region according to the standard rules. In case (iii), if the new job arrival is at a different demand point from the server's current location, then the current location should be treated as an intermediate stage. The intuitive explanation for including the modifications (ii) and (iii) is that if a server has no work to do in its own region, but there is work to do in another region, then instead of remaining idle it should go to assist in one of the other regions - but return to its own region as soon as new jobs arrive there.

For notational convenience, let $\nu_i(\mathbf{x}) = \sum_{s \in M} \mathbb{I}(v(\mathbf{x}) = i)$ denote the total number of servers present at demand point $i \in D$ under state $\mathbf{x} \in S$. We will simply write ν_i instead of $\nu_i(\mathbf{x})$ when the state associated with this quantity is obvious. Also, let $\mu_i^k = \sum_{r=1}^{k} (1 - \alpha)^{r-1} \mu_i$ denote the total service rate at demand point $i \in D$ with k servers there. The algorithm below shows how each server $s \in M$ selects an action under state $\mathbf{x} \in S$ under the modified 1-stop heuristic. We note that, as in the polling

heuristic, it is necessary to use Algorithm 4.1 first in order to obtain the allocation vector $\mathbf{p}$, and to update the local state $\mathbf{x}_s$ by randomly updating the local variables x_{si} according to random draws when new job arrivals or service completions occur.

Modified 1-stop heuristic algorithm

For server $s \in M$ use the following rules to select an action under state $\mathbf{x} \in S$.

1. If the server is at a demand point $i \in \mathcal{D}_s$ with $x_{si} > 0$ then

   (a) Initialize an empty set $\sigma = \emptyset$.

   (b) For each demand point $j \in \mathcal{D}_s$ with $c_j \mu_j^{\nu_j+1} \geq c_i \mu_i^{\nu_i}$, estimate the reward rate ψ_j that would be earned by switching to node j, and serving it exhaustively, and also the reward rate ϕ_j that would be earned by switching to node j, serving it exhaustively and then returning to i, using

   $$\psi_j = c_j \mu_j^{\nu_j+1} \frac{x_{sj} + \lambda_{sj}\delta(i,j)/\tau}{x_{sj} + \mu_j^{\nu_j+1}\delta(i,j)/\tau},$$

   $$\phi_j = c_j \mu_j^{\nu_j+1} \frac{x_{sj} + \lambda_{sj}\delta(i,j)/\tau}{x_{sj} + \mu_j^{\nu_j+1}\delta(i,j)/\tau + (\mu_j^{\nu_j+1} - \lambda_{sj})\delta(j,i)/\tau}. \tag{4.18}$$

   If $\phi_j \geq c_j \mu_j^{\nu_j+1} \rho + c_i \mu_i^{\nu_i}(1-\rho)$, then add j to the set σ.

   (c) If σ is non-empty, then switch to the first node on a shortest path from i to the demand point j^* in σ with the highest index ψ_{j^*} (with ties broken arbitrarily). Otherwise, the action chosen should be to remain at node i.

2. If the server is at a node i such that either i is an empty demand point (that is, $i \in \mathcal{D}_s$ and $x_{si} = 0$) or i is an intermediate stage (that is, $i \in N$) or i is a demand point not included in the server's region (that is, $i \notin \mathcal{D}_s$), and there is at least one demand point $j \in \mathcal{D}_s$ with $x_{sj} > 0$, then

   (a) Initialize three empty sets: $\sigma_1 = \emptyset$, $\sigma_2 = \emptyset$ and $\sigma = \emptyset$.

   (b) For each demand point $j \in \mathcal{D}_s$ with $j \neq i$, calculate the reward rate ψ_j the same way as in part 1(a). If $\psi_j \geq c_j \mu_j^{\nu_j+1} \rho$, then add j to σ_1; otherwise, add j to σ_2.

   (c) If σ_1 is non-empty, let $\sigma = \sigma_1$. Otherwise, let $\sigma = \sigma_2$.

   (d) Carry out step 1(c).

3. If $x_{sj} = 0$ for all $j \in D$ then

   (a) If there is at least one demand point $i \in D$ with $x_i > 0$, then switch to the first node on a shortest path from the server's current location v_s to the nearest non-empty demand point in the network (with ties broken arbitrarily).

   (b) If $x_i = 0$ for all $i \in D$, carry out steps 2(a)-2(d).

In the next section we present the results of numerical experiments to show how these heuristics perform in randomly-generated problem instances.

4.4 Numerical results

Our numerical experiments are based on randomly-generated network layouts. In each problem instance we generate a network layout within a 5×5 grid, following the methodology established in Chapter 3 of this thesis. The network construction begins with the random placement of demand points on the grid, where each demand point $i \in D$ is assigned coordinates (a_i, b_i), with $a_i, b_i \in \{1, 2, 3, 4, 5\}$. The grid nodes are interconnected by horizontal and vertical edges, forming a Manhattan distance-based structure where the shortest path between any two nodes i and j is given by $|a_i - a_j| + |b_i - b_j|$. Nodes that do not serve as demand points function as intermediate stages, though some may be redundant.

Within each problem instance, we identify and eliminate redundant intermediate stages using the *dominated stages* method. An intermediate stage $k \in N$ is classified as dominated if there is at least one other intermediate stage $j \in N$ such that $\delta(i, k) \geq \delta(i, j)$ for all demand points $i \in D$, with this inequality being strict for at least one $i \in D$. By removing dominated intermediate stages, we simplify the network while preserving necessary routing options. Figure 4.3 shows 4 of the randomly-generated network layouts from our experiments. For any pair of nodes in the resulting network, a shortest path exists and may pass through demand points. The index calculations depend only on the path length, defined as the number of nodes traversed excluding the starting node, and are independent of the types of nodes along the path. Passing through a demand point incurs no additional delay, since the server performs service only when explicitly selected and may switch immediately upon arrival.

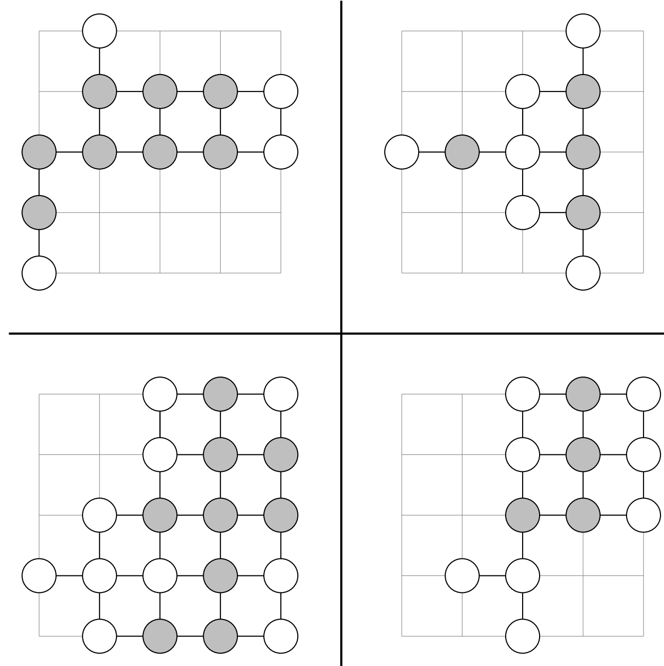

FIGURE 4.3: 4 randomly-generated network layouts with demand points shown in white and intermediate stages shown in gray, after removal of redundant intermediate stages.

To introduce variability in our experiments, each problem instance has a randomly selected number of servers m from the set $\{2, 3, 4\}$ and number of demand points d from the set $\{4, 5, 6, 7, 8, 9, 10\}$. We also generate the values of ρ, τ, λ_i, μ_i and c_i (for each $i \in D$) randomly, using a method that accounts for a broad range of possible system scenarios. Two key parameters in our study are ρ, representing the overall worst case traffic intensity of the system, and $\eta := \tau/(\sum_i \lambda_i)$, which measures the server's ability to switch between adjacent nodes relative to the total rate at which jobs arrive. The parameter α, which adjusts the service rate based on the number of servers at i, takes values from the set $\{0, 0.1, ..., 0.9\}$ and is applied consistently across all heuristics. Our sampling method first generates the values for ρ and η, then considers each possible value of α in turn and scales the values of other parameters to conform to the required values of ρ, η and α. Further details regarding the parameter generation method can be found in Appendix C.1.

In the multiple-server problem, it is not feasible to compute optimal solutions using dynamic programming unless the problem size is trivially small. Therefore we focus on comparisons between heuristic policies in this section. In each problem instance, we evaluate the performances of four heuristics through simulation experiments. The two heuristics introduced in Section 4.3, namely the polling heuristic and the modified 1-stop

heuristic, allocate demand proportions to servers according to Algorithm 4.1 and then direct the servers to serve these demands according to the rules described in Sections 4.3.1 and 4.3.2.

We compare the performances of the polling and modified 1-stop heuristics with those of two other heuristics, referred to as 'global scheduling' methods, which allow servers to move freely across the entire network (rather than being restricted to local regions). The first of these methods, referred to as the *flexible* heuristic, allows multiple servers to simultaneously serve the same demand point. The second, referred to as the *exclusive* heuristic, restricts each demand point to being served by only one server at a time. These methods are described in the next two subsections.

4.4.1 Flexible heuristic

This is a simple distance-based heuristic which allows multiple servers to serve a demand point simultaneously. The action for server $s \in M$ at state $\mathbf{x} \in S$ is selected according to the rules below.

1. If server s is at a demand point $i \in D$ with $x_i > 0$ then the action is to remain at node i.
2. If server s is at a node $i \in V$ such that either i is an empty demand point (that is, $i \in D$ and $x_i = 0$) or i is an intermediate stage (that is, $i \in N$), and there is at least one demand point $j \in D$ such that $x_j > 0$, then
   (a) Initialize an empty set $\sigma = \emptyset$.
   (b) Add to σ all demand points $j \in D$ with $x_j > 0$ that are closest to i. Formally,
   $$\sigma \leftarrow \left\{ j \in D : x_j > 0 \ \text{ and } \ \delta(i,j) = \min_{k \in D,\, x_k > 0} \delta(i,k) \right\}.$$
   (c) Let j^* denote the demand point in σ with the largest value of $c_j \mu_j^{\nu_j+1}$ (with ties broken arbitrarily). The server should switch to the first node on a shortest path from i to j^*.
3. If $x_j = 0$ for all $j \in D$, then server s is to remain at node i.

4.4.2 Exclusive heuristic

The main purpose of this heuristic is to ensure that a demand point is never served by more than one server at the same time. At each time step, any server that is not already at a non-empty demand point is assigned to one of the demand points that are currently unoccupied, and these assignments are made in such a way that no two servers are assigned to the same demand point.

To implement this procedure, we first generate all possible pairs between idle servers and non-empty, non-occupied demand points. These pairs are then sorted in ascending order of distance. We iterate through the sorted list and, for each pair (s, j), if both server s and demand point j remain unpaired, we record the match and remove them from further consideration. This greedy matching process continues until no eligible pairs remain. We denote by y_s the demand point assigned to server s (if any). The pairing must satisfy the following constraints:

$$\sum_{j \in D} \mathbb{I}(s,j) \leq 1, \quad \forall s \in M, \quad \text{and} \quad \sum_{s \in M} \mathbb{I}(s,j) \leq 1, \quad \forall j \in D,$$

where $\mathbb{I}(s, j)$ is an indicator function equal to 1 if server s is paired with demand point j, and 0 otherwise. The paired demand point for server s is given by:

$$y_s = \begin{cases} j, & \text{if } \exists\, j \in D \text{ such that } \mathbb{I}(s,j) = 1, \\ 0, & \text{otherwise.} \end{cases}$$

The action for server $s \in M$ at state $\mathbf{x} \in S$ is selected according to the following rules:

1. If server s is at a demand point i with $x_i > 0$, then the action is to remain at i.
2. If server s is at a node $i \in V$ such that either i is an empty demand point (that is, $i \in D$ and $x_i = 0$) or i is an intermediate stage (that is, $i \in N$), and the system is non-empty (that is, $\sum_{i \in D} x_i > 0$), then
   (a) If $y_s = 0$, server s remains at i.
   (b) If $y_s = j$, server s moves from i to the first node on a shortest path from i to j.
3. If $x_j = 0$ for all $j \in D$, then server s remains at node i.

4.4.3 Results of experiments

Table 4.1 and 4.2 compares the performances of all heuristic policies across 1000 instances for various service discount rates α, with the polling heuristic as the benchmark. For each instance, the table reports 95% confidence intervals for the mean percentage improvements over the polling policy. The modified 1-stop heuristic generally outperforms the others in most values of α, with improvements ranging from 47.39% at $\alpha = 0$ to 33.34% as α approaches 1, reflecting the diminishing benefit of cross-regional joint service under discounting. At high α values, its performance falls slightly below that of the exclusive heuristic, which remains largely stable between 36% and 39%, as the exclusive heuristic prevents multiple servers from serving the same point simultaneously. Minor variations in the exclusive heuristic relative to the benchmark arise because, in the polling heuristic, servers occasionally converge on the same location during their cyclic routes, allowing limited joint service affected by α. The flexible heuristic, on the other hand, which permits multiple servers to visit the same point, exhibits a sharp decline as α increases, ultimately resulting in negative improvements at high values of α.

Table 4.1: The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for $\alpha \in \{0, 0.1, 0.2, 0.3, 0.4\}$, with 100 instances evaluated for each α across a total of 1000 instances.

Heuristic	$\alpha = 0$	$\alpha = 0.1$	$\alpha = 0.2$	$\alpha = 0.3$	$\alpha = 0.4$
modified 1-stop	47.39 ± 3.59	46.98 ± 3.07	45.09 ± 2.88	44.12 ± 2.78	42.69 ± 2.50
exclusive	36.79 ± 5.88	39.09 ± 2.83	38.79 ± 2.65	39.02 ± 2.70	38.80 ± 2.74
flexible	27.12 ± 5.83	22.13 ± 4.90	16.97 ± 4.80	13.40 ± 4.81	9.18 ± 5.22

Table 4.2: The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for $\alpha \in \{0.5, 0.6, 0.7, 0.8, 0.9\}$, with 100 instances evaluated for each α across a total of 1000 instances.

Heuristic	$\alpha = 0.5$	$\alpha = 0.6$	$\alpha = 0.7$	$\alpha = 0.8$	$\alpha = 0.9$
modified 1-stop	40.76 ± 2.46	38.03 ± 2.29	36.46 ± 2.47	34.60 ± 2.74	33.34 ± 2.95
exclusive	38.94 ± 2.70	36.20 ± 2.74	36.54 ± 2.86	36.90 ± 2.82	36.66 ± 3.03
flexible	5.54 ± 6.06	-3.57 ± 7.64	-7.67 ± 9.13	-14.19 ± 12.59	-18.62 ± 13.98

Table 4.3 provides comparisons for heuristics under systems with different numbers of demand points, $d \in \{4, 5, 6, 7, 8, 9, 10\}$. As in Tables 4.1 and 4.2, each row in Table 4.3 reports 95% confidence intervals for the mean percentage improvements over the polling policy. The results indicate that all heuristics tend to improve as d increases,

reflecting the greater flexibility of these policies compared with the fixed-order polling heuristic in handling the growing complexity associated with more demand points. As intuition would suggest, the modified 1-stop heuristic performs better as d increases, as it can balance the workload between servers through an initial allocation scheme. This is reflected in the results, with an improvement of 31.17% at $d = 4$ rising to 49.32% at $d = 10$. The exclusive heuristic also shows steady improvements, increasing from 28.61% to 46.73%, as its restriction on multiple servers serving the same point promotes efficient utilization of each server, which becomes increasingly beneficial as the number of demand points grows. We note that the gap between the exclusive heuristic and the modified 1-stop heuristic is larger when $d = 5$ than in the other cases. This happens because when $d = 5$ we sample m from the set $\{2, 3, 4\}$, and if $m = 4$ then the number of servers is almost equal to the number of demand points, implying that the system has plenty of capacity and should often allow multiple servers to occupy the same demand point in order to maximize efficiency - but the exclusive heuristic does not allow this. When $d \in \{6, 7, ..., 10\}$ we still sample m from the set $\{2, 3, 4\}$, so we do not encounter a similar situation with m being so close to d. The flexible heuristic performs worse than the polling policy when d is small because, in these situations, the distances between demand points tend to be larger and it is more efficient for servers to remain within their own regions, as they do under the polling policy. As d increases, the flexible heuristic performs better because the grid becomes more densely populated with demand points and the distances between demand points become smaller on average.

Table 4.3: The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for different values of d over all 1000 instances.

Heuristic	$d = 4$ [130 instances]	$d = 5$ [150]	$d = 6$ [120]	$d = 7$ [170]	$d = 8$ [150]	$d = 9$ [140]	$d = 10$ [140]
modified 1-stop	31.17 ± 2.40	37.66 ± 3.09	37.16 ± 3.20	45.38 ± 1.75	38.89 ± 1.65	45.23 ± 1.73	49.32 ± 1.92
exclusive	28.61 ± 2.37	24.25 ± 3.97	37.66 ± 1.56	40.84 ± 2.05	42.45 ± 2.13	43.19 ± 1.55	46.73 ± 1.65
flexible	-13.46 ± 9.57	-12.18 ± 9.51	9.23 ± 5.75	7.05 ± 7.13	12.27 ± 4.08	8.95 ± 4.84	22.91 ± 4.99

Table 4.4 shows the mean percentage improvements over the polling policy (with 95% confidence intervals) of the heuristics for systems with varying numbers of servers $m \in \{2, 3, 4\}$. As shown, the modified 1-stop heuristic steadily improves as m increases, reflecting its ability to coordinate server effort efficiently by scheduling servers within their own areas or to assist other areas when idle. The exclusive heuristic performs consistently for smaller m, but its improvement over the polling policy is slightly smaller at $m = 4$, as the restriction against multiple servers serving the same point reduces flexibility when more servers are available. The flexible heuristic, in contrast, shows a clear

decline as m increases, moving from positive improvement at $m = 2$ to slightly negative at $m = 4$, indicating that allowing multiple servers to visit the same point becomes progressively less effective with more servers, as their efforts may be underutilized.

Table 4.4: The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for different values of m over all 1000 instances.

Heuristic	$m = 2$ [340 instances]	$m = 3$ [360]	$m = 4$ [300]
modified 1-stop	38.39 ± 1.29	41.19 ± 1.40	43.55 ± 2.09
exclusive	37.35 ± 1.28	41.04 ± 1.55	34.34 ± 2.34
flexible	13.50 ± 2.75	2.80 ± 5.07	-1.90 ± 5.73

Table 4.5 shows the effects of varying the traffic intensity, $\rho = \sum_{i \in D} \left(\lambda_i / \sum_{r=1}^{m} (1 - \alpha)^{r-1} \mu_i \right)$. The method for randomly generating system parameter values (see Appendix C.1) ensures that, in any given problem instance, ρ is equally likely to fall within any of the four intervals shown as columns in the table. As with the previous tables, we present the results in the same format, with 95% confidence intervals for the relative performance of the heuristics over the polling heuristic. It can be seen that the modified 1-stop and exclusive heuristics both maintain relatively stable improvements across the different traffic intensity ranges, reflecting their efficient use of servers, while the flexible heuristic deteriorates sharply as ρ increases, due to the growing impact of inefficient server utilization under heavier system workloads. More specifically, the flexible heuristic tends to allow multiple servers to occupy the same demand point more often than the other heuristics, and this becomes increasingly inefficient as ρ increases.

Table 4.5: The mean percentage improvements (with 95% confidence intervals) of the modified 1-stop policy, exclusive policy and flexible policy over the polling policy shown for different values of ρ over all 1000 instances.

Heuristic	$0.1 \le \rho < 0.3$ [280 instances]	$0.3 \le \rho < 0.5$ [260]	$0.5 \le \rho < 0.7$ [230]	$0.7 \le \rho < 0.9$ [230]
modified 1-stop	42.00 ± 1.53	38.73 ± 1.59	41.43 ± 2.03	41.68 ± 2.29
exclusive	40.15 ± 1.66	33.68 ± 1.81	37.82 ± 1.63	39.46 ± 2.83
flexible	26.80 ± 2.25	12.51 ± 3.12	1.81 ± 4.32	-26.71 ± 8.50

4.5 Conclusions

This paper is an extension of our previous work (Tian and Shone (2026b)) and focuses on the multi-server scheduling problem. The originality of the model lies in formulating a dynamic problem in which a batch of m homogeneous servers are scheduled across d

demand points in a network, where intermediate stages exist between demand points. This structure allows the model to accurately capture interruptions in both setup and processing times, while also enabling the representation of complex dependencies among the setup requirements of different job types at the demand points. The arrival times of new jobs at demand points and the servers' setup and processing times are random, making the problem highly dynamic and stochastic. We conjecture that the system is stable provided that $\rho < 1$, where the traffic intensity ρ depends on the joint-service effect among servers controlled by the parameter α.

The main idea in developing approaches for the problem is, firstly, to introduce an allocation scheme that proportionally assigns demand points to servers by partitioning them into separate regions for each server, and secondly, to modify the 1-stop heuristic established in Tian and Shone (2026b) by allowing idle servers to be 'borrowed' to assist other regions. As a benchmark, each server applies a polling heuristic within its own region, and the overall performance is assessed in aggregate. Two other heuristics (referred to as the 'global scheduling' methods) are introduced for comparison, with the exclusive heuristic restricting each demand point to being served by only one server at a time, and the flexible heuristic allowing multiple servers to serve the same demand point simultaneously. Except for the exclusive heuristic, these heuristics allow the joint processing of jobs by multiple servers, with larger values of α corresponding to greater losses in total service rate.

The numerical results in Section 4.4 show the performances of the heuristics under various parameter settings, categorized according to the parameter α, the number of demand points d, the number of servers m, and the traffic intensity ρ, all considered over uniform ranges. The modified 1-stop demonstrates consistently strong performance compared to the benchmark polling policy and also outperforms the two global heuristics in most cases, suggesting that this local scheduling heuristic is a strong candidate for practical use in complex multi-server scheduling problems. Future research could explore alternative allocation schemes for partitioning demand among servers, as well as the incorporation of long-sighted approaches such as the K-stop with $K \geq 2$ (Tian and Shone (2026b)) within the local scheduling heuristics.

Chapter 5

Conclusions and Future Work

This chapter concludes the thesis by providing a summary of its main contributions and research outcomes, and by outlining directions for future work.

5.1 Summary

This thesis has investigated stochastic dynamic scheduling problems in which mobile servers respond to spatially distributed demand that evolves over time. Across the three studies presented in Chapters 2 to 4, heuristic methods are developed and analysed for single-server and multi-server systems, with attention to both theoretical properties and practical applicability. A distinctive feature of the problem setting is the explicit representation of networks with both demand points and intermediate stages, which allows server movements, setup requirements for different demand types, and structural similarities or differences across such requirements to be represented. The analysis shows that decision rules combining long-sighted planning with the ability to interrupt service or switch can reduce congestion level in the system and lower operational costs.

Chapter 2 focuses on infinite-state, single-server systems and develops two heuristics: the K-stop, which anticipates future server movements, and its scalable variant, the (K from L)-stop. The pathwise consistency theorem for the K-stop shows that the server follows coherent paths through intermediate stages, avoiding unnecessary travel, and a corollary guarantees that a demand point is reached in finite expected time. Increasing K improves performance by extending foresight, albeit at greater computational cost, while the (K from L)-stop reduces effort by pre-selecting candidate demand points

without significant loss of accuracy. These theoretical properties are borne out in numerical experiments: in 1,229 of 10,000 instances with at most three demand points, where optimal performances were known, the K-stop heuristics achieved substantially lower suboptimality than the DVO heuristic (which prohibits interruptible switching or processing), with the $K = 2$ and $K = 3$ policies on average within 4% of optimality. Across all 10,000 instances, the K-stop and (K from L)-stop heuristics improved mean performance over DVO by 9–11%, with the (2 from 4)-stop producing results close to the 2-stop heuristic.

Chapter 3 considers finite-state, single-server systems, motivated by machine repair and maintenance applications where each demand point corresponds to a machine with discrete degradation levels. Representing deterioration stages with general cost functions captures variation in severity and failure risk across machines. An index-based heuristic is developed that focuses on the next repair decision and is based on a provably equivalent cost decomposition separating fixed from server-dependent components. This decomposition allows indices for each machine to be calculated by approximating average reward from server operations. The resulting index policy is optimal in certain cases and provides a practical baseline for decision-making. To further enhance performance, the index policy is refined through a procedure that combines value function approximation with one-step policy improvement using reinforcement learning. This refinement, referred to as approximate policy improvement, yields a forward-looking yet computationally efficient strategy that achieves near-optimal performance even in fast-changing systems with transition rates on the order of 100 per second by setting time-step length $\delta = 0.01$, making it suitable for online implementation in real time. A unichain modification is introduced to ensure that the induced Markov chain is unichain, thereby guaranteeing well-defined average costs and enabling iterative refinement through approximate policy improvement. The effectiveness of these developments is demonstrated numerically: across 412 of 1,000 instances with 2–4 machines, approximate policy improvement reduced the average suboptimality from 8.11% under the index-based heuristic to 2.51% under the cost formulation, and from 3.01% to 1.02% under the reward formulation, respectively, the latter reflecting the lower sensitivity of the reward formulation to the likely increase in degraded machines as the total number of machines grows. Across all 1,000 instances, the additional benefit of approximate policy improvement over the index policy remains fairly consistent as the problem size increases.

Chapter 4 extends the analysis of Chapter 2 to multi-server systems, where coordination and workload balancing are central. A proportional allocation method first assigns

demand points to servers, defining service areas with varying responsibilities. Within each area, a modified 1-stop heuristic is applied, improving performance by reducing congestion and better utilising server capacity compared with purely global strategies. The heuristic also allows idle servers to provide temporary joint service to demand points outside their own areas, reflecting practical operational flexibility. The effect of multiple servers serving the same point is controlled by the parameter α, which models diminishing returns. Numerical results show that smaller α values improve the relative performance of the modified 1-stop heuristic compared to a benchmark polling heuristic, from 33.37% at $\alpha = 0.9$ to 46.9% at $\alpha = 0$, highlighting the benefits of joint service and server reassignment. The influence of α is further illustrated by two global heuristics: exclusive, where each demand point is served by only one server, and flexible, where multiple servers can serve the same point. When α is small, the exclusive heuristic suffers because limiting each demand point to a single server prevents efficiency gains, whereas the flexible heuristic benefits from shared service without significant penalty.

The numerical experiments across all studies indicate that the proposed heuristics generate effective policies. Anticipating future server movements can reduce congestion even over short horizons, exploiting network structure and allowing interruptible operations reduces unnecessary travel, adaptive policies that reevaluate actions respond more effectively to uncertainty, and effective assignment of areas to servers further enhances multi-server performance. A common feature across the studies is the use of index heuristics, introduced in Section 1.4.5, with each study designing a version tailored to its specific problem, resulting in strong performance as reflected in the numerical results. By assigning a simple index to each task, these heuristics enable fast and more interpretable decision-making without evaluating the full state space. This provides practical advantages over alternative methods such as dynamic programming, which suffers from the curse of dimensionality, and reinforcement learning, which requires extensive training and cannot easily adjust to changing parameters.

In summary, this thesis develops heuristic approaches for multi-server scheduling systems. In Chapter 2, K-stop and (K from L)-stop heuristics are introduced for infinite-state, single-server systems; in Chapter 3, index-based heuristics and approximate policy improvement methods using reinforcement learning are applied to finite-state, single-server systems; and in Chapter 4, proportional allocation combined with a modified index heuristic is employed for multi-server systems. The results illustrate how heuristic performance varies with traffic intensity, switching rate, and penalty parameters, highlighting the roles of foresight, interruptibility, and cooperation in shaping effective

scheduling policies.

5.2 Future work

Several avenues for future research can build on this thesis. A key direction is establishing formal performance guarantees for heuristic policies. Approximation ratios, quantifying proximity to the optimal policy, and regret bounds, measuring cumulative loss over time, would provide stronger theoretical foundations. Another is to extend reinforcement learning techniques to very large or infinite state spaces. Function approximation and related methods could scale learning to design adaptive and computationally tractable policies for complex service systems.

Further work could address system-level challenges in multi-server environments. Alternative allocation strategies may improve workload balancing and reduce congestion. Extending the models to allow for dynamic fleet sizing, in which the number of active servers adapts to real-time demand, would provide greater flexibility in resource management. Incorporating demand feedback, where service actions influence future patterns of demand, would enrich the modelling framework. For example, in maintenance applications, repairing a machine reduces the likelihood of immediate breakdowns, while in service networks, server movements can alter congestion and future arrival rates. Other possible extensions include heterogeneous server capabilities, communication delays, partial observability, and non-stationary demand.

These directions have practical implications across domains and would build on the foundations laid in this thesis. In mobile healthcare, adaptive routing that accounts for feedback could improve patient outcomes. In emergency response, heuristics robust to uncertainty would increase reliability. In logistics and infrastructure maintenance, reinforcement learning combined with advanced allocation and fleet management strategies could enhance efficiency and support real-time decision-making.

Bibliography

Abbou, A. and Makis, V. (2019). Group maintenance: A restless bandits approach. *INFORMS Journal on Computing*, 31(4):719–731.

Ahmad, W., Hasan, O., Pervez, U., and Qadir, J. (2017). Reliability modeling and analysis of communication networks. *Journal of Network and Computer Applications*, 78:191–215.

Al-Hussaini, S., Thakar, S., Kim, H., Rajendran, P., Shah, B. C., Marvel, J. A., and Gupta, S. K. (2020). Human-supervised semi-autonomous mobile manipulators for safely and efficiently executing machine tending tasks. *arXiv preprint arXiv:2010.04899*.

Aldossary, M. (2021). A review of dynamic resource management in cloud computing environments. *Computer Systems Science & Engineering*, 36(3):461–476.

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.

Annem, V., Rajendran, P., Thakar, S., and Gupta, S. K. (2019). Towards remote teleoperation of a semi-autonomous mobile manipulator system in machine tending tasks. In *International Manufacturing Science and Engineering Conference*, volume 58745, page V001T02A027. American Society of Mechanical Engineers.

Ansell, P., Glazebrook, K. D., Nino-Mora, J., and O'Keeffe, M. (2003). Whittle's index policy for a multi-class queueing system with convex holding costs. *Mathematical Methods of Operations Research*, 57:21–39.

Antunes, N., Fricker, C., and Roberts, J. (2011). Stability of multi-server polling system with server limits. *Queueing Systems*, 68:229–235.

Argon, N., Ding, L., Glazebrook, K., and Ziya, S. (2009). Dynamic routing of customers with general delay costs in a multiserver queuing system. *Probability in the Engineering and Informational Sciences*, 23(2):175–203.

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A., Stoica, I., et al. (2010). A view of cloud computing. *Communications of the ACM*, 53(4):50–58.

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

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

Atar, R., Giat, C., and Shimkin, N. (2011). On the asymptotic optimality of the cμ/θ rule under ergodic cost. *Queueing Systems*, 67:127–144.

Ayesta, U., Gupta, M., and Verloop, I. (2021). On the computation of Whittle's index for Markovian restless bandits. *Mathematical Methods of Operations Research*, 93:179–208.

Bandara, D., Mayorga, M. E., and McLay, L. A. (2012). Optimal dispatching strategies for emergency vehicles to increase patient survivability. *International Journal of Operational Research*, 15(2):195–214.

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

Baron, R. and Haick, H. (2024). Mobile diagnostic clinics. *ACS sensors*, 9(6):2777–2792.

Baubaid, A., Boland, N., and Savelsbergh, M. (2023). The dynamic freight routing problem for less-than-truckload carriers. *Transportation Science*, 57(3):717–740.

Belgacem, A. (2022). Dynamic resource allocation in cloud computing: analysis and taxonomies. *Computing*, 104(3):681–710.

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.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

Berbeglia, G., Cordeau, J.-F., and Laporte, G. (2010). Dynamic pickup and delivery problems. *European Journal of Operational Research*, 202(1):8–15.

Bertels, S. and Fahle, T. (2006). A hybrid setup for a hybrid scenario: combining heuristics for the home health care problem. *Computers & Operations Research*, 33(10):2866–2890.

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

Bertsekas, D. P. and Tsitsiklis, J. N. (1996). *Neuro-Dynamic Programming*. Athena Scientific.

Bhatnagar, A. and Shukla, A. K. (2023). An optimized multi-server queuing model for transportation problems. In *Computational Intelligence in Analytics and Information Systems*, pages 283–293. Apple Academic Press.

Biswas, A. (2017). An ergodic control problem for many-server multiclass queueing systems with cross-trained servers. *Stochastic Systems*, 7(2):264–288.

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

Bonfim, M. S., Dias, K. L., and Fernandes, S. F. (2019). Integrated nfv/sdn architectures: A systematic literature review. *ACM Computing Surveys (CSUR)*, 51(6):1–39.

Boon, M. A., van der Mei, R. D., and Winands, E. M. (2011). Applications of polling systems. *Surveys in Operations Research and Management Science*, 16(2):67–82.

Borst, S. C. (1995). Polling systems with multiple coupled servers. *Queueing Systems*, 20:369–393.

Botteghi, N., Grefte, L., Poel, M., Sirmacek, B., Brune, C., Dertien, E., and Stramigioli, S. (2021). Towards autonomous pipeline inspection with hierarchical reinforcement learning. In *International Conference on Robot Intelligence Technology and Applications*, pages 259–271. Springer.

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

Bravo, J. J. and Vidal, C. J. (2013). Freight transportation function in supply chain optimization models: A critical review of recent trends. *Expert Systems with Applications*, 40(17):6742–6757.

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

Büsing, C., Comis, M., Schmidt, E., and Streicher, M. (2021). Robust strategic planning for mobile medical units with steerable and unsteerable demands. *European Journal of Operational Research*, 295(1):34–50.

Butt, S. A., Naseer, M., Ali, A., Khalid, A., Jamal, T., and Naz, S. (2024). Remote mobile heath monitoring frameworks and mobile applications: Taxonomy, open challenges, motivation, and recommendations. *Engineering Applications of Artificial Intelligence*, 133:108233.

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

Buzacott, J. A. and Shanthikumar, J. G. (1993). Stochastic models of manufacturing systems. *Prentice-Hall international ed.*

Byner, C., Matthias, B., and Ding, H. (2019). Dynamic speed and separation monitoring for collaborative robot applications–concepts and performance. *Robotics and Computer-Integrated Manufacturing*, 58:239–252.

Cai, J., Zhu, Q., Lin, Q., Ma, L., Li, J., and Ming, Z. (2023). A survey of dynamic pickup and delivery problems. *Neurocomputing*, 554:126631.

Calvo, A., Silano, G., and Capitán, J. (2022). Mission planning and execution in heterogeneous teams of aerial robots supporting power line inspection operations. In *2022 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1644–1649. IEEE.

Caunhye, A. M., Nie, X., and Pokharel, S. (2012). Optimization models in emergency logistics: A literature review. *Socio-economic Planning Sciences*, 46(1):4–13.

Cerrato, I., Palesandro, A., Risso, F., Suñé, M., Vercellone, V., and Woesner, H. (2015). Toward dynamic virtualized network services in telecom operator networks. *Computer Networks*, 92:380–395.

Chan, C. W., Huang, M., and Sarhangian, V. (2021). Dynamic server assignment in multiclass queues with shifts, with applications to nurse staffing in emergency departments. *Operations Research*, 69(6):1936–1959.

Choi, M., Molisch, A. F., and Kim, J. (2020). Joint distributed link scheduling and power allocation for content delivery in wireless caching networks. *IEEE Transactions on Wireless Communications*, 19(12):7810–7824.

Cinlar, E. (1975). *Introduction to Stochastic Processes*. Prentice-Hall, Englewood Cliffs, New Jersey.

Clarkson, J., Glazebrook, K. D., and Lin, K. Y. (2020). Fast or slow: Search in discrete locations with two search modes. *Operations Research*, 68(2):552–571.

Desaulniers, G., Lavigne, J., and Soumis, F. (1998). Multi-depot vehicle scheduling problems with time windows and waiting costs. *European Journal of Operational Research*, 111(3):479–494.

Dey, P. K. (2004). Decision support system for inspection and maintenance: a case study of oil pipelines. *IEEE Transactions on Engineering Management*, 51(1):47–56.

Duenyas, I. and Van Oyen, M. (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. *Comput. Mater. Contin*, 73(3):5103–5120.

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

Feng, J., Pei, Q., Yu, F. R., Chu, X., Du, J., and Zhu, L. (2020). Dynamic network slicing and resource allocation in mobile edge computing systems. *IEEE Transactions on Vehicular Technology*, 69(7):7863–7878.

Fikar, C. and Hirsch, P. (2017). Home health care routing and scheduling: A review. *Computers & Operations Research*, 77:86–95.

Firoozabadi, S. M. K., Soleimani, G., Amiri, M., and Moradian, M. (2017). Review of emergency response methods in disaster management, dispatch and control of forces in emergencies. *Journal of Economic & Management Perspectives*, 11(3):1737–1747.

Galassi, M., Davies, J., Theiler, J., Gough, B., Jungman, G., Alken, P., Booth, M., and Rossi, F. (2011). GNU Scientific Library Reference Manual-Edition 1.15, for GSL Version 1.15.

Gans, N., Koole, G., and Mandelbaum, A. (2003). Telephone call centers: Tutorial, review, and research prospects. *Manufacturing & Service Operations Management*, 5(2):79–141.

Gendreau, M., Laporte, G., and Semet, F. (2001). A dynamic model and parallel tabu search heuristic for real-time ambulance relocation. *Parallel Computing*, 27(12):1641–1653.

Geurtsen, M., Didden, J. B., Adan, J., Atan, Z., and Adan, I. (2023). Production, maintenance and resource scheduling: A review. *European Journal of Operational Research*, 305(2):501–529.

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.

Gittins, J., Glazebrook, K., and Weber, R. (2011). *Multi-armed bandit allocation indices.* John Wiley & Sons.

Gittins, J. C. (1979). Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 41(2):148–164.

Glazebrook, K. D., Mitchell, H., and Ansell, P. (2005). Index policies for the maintenance of a collection of machines by a set of repairmen. *European Journal of Operational Research*, 165(1):267–284.

Glazebrook, K. D., Ruiz-Hernandez, D., and Kirkbride, C. (2006). Some indexable families of restless bandit problems. *Advances in Applied Probability*, 38(3):643–672.

Gosavi, A. (2003). *Simulation-Based Optimization: Parametric Optimization Techniques and Reinforcement Learning*, volume 25 of *Operations Research/Computer Science Interfaces Series.* Springer.

Gu, R., Liu, Y., and Poon, M. (2023). Dynamic truck–drone routing problem for scheduled deliveries and on-demand pickups with time-related constraints. *Transportation Research Part C: Emerging Technologies*, 151:104139.

He, Y., Xing, L., Chen, Y., Pedrycz, W., Wang, L., and Wu, G. (2020). A generic Markov decision process model and reinforcement learning method for scheduling agile earth observation satellites. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 52(3):1463–1474.

Higgins, A. (1998). Scheduling of railway track maintenance activities and crews. *Journal of the Operational Research Society*, 49(10):1026–1033.

Ifrah, S. (2024). *Azure Kubernetes Service*, pages 121–149. Apress, Berkeley, CA.

Iqbal, H., Tesfamariam, S., Haider, H., and Sadiq, R. (2017). Inspection and maintenance of oil & gas pipelines: a review of policies. *Structure and Infrastructure Engineering*, 13(6):794–815.

Jia, F., Afaq, M., Ripka, B., Huda, Q., and Ahmad, R. (2023). Vision-and lidar-based autonomous docking and recharging of a mobile robot for machine tending in autonomous manufacturing environments. *Applied Sciences*, 13(19):10675.

Jia, F., Ma, Y., and Ahmad, R. (2024). Review of current vision-based robotic machine-tending applications. *The International Journal of Advanced Manufacturing Technology*, 131(3):1039–1057.

Jun, S., Lee, S., and Yih, Y. (2021). Pickup and delivery problem with recharging for material handling systems utilising autonomous mobile robots. *European Journal of Operational Research*, 289(3):1153–1168.

Kamal, A. E. and Hamacher, V. (1989). Approximate analysis of non-exhaustive multi-server polling systems with applications to local area networks. *Computer Networks and ISDN Systems*, 17(1):15–27.

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.

Kim, J.-H., Sharma, G., Boudriga, N., and Iyengar, S. S. (2010). Spamms: A sensor-based pipeline autonomous monitoring and maintenance system. In *2010 Second International Conference on COMmunication Systems and NETworks (COMSNETS 2010)*, pages 1–10. IEEE.

Krishnan, K. (1990). Joining the right queue: A state-dependent decision rule. *IEEE Transactions on Automatic Control*, 35:104–108.

Krishnasamy, S., Arapostathis, A., Johari, R., and Shakkottai, S. (2018). On learning the cμ rule: Single and multi-server settings. *Available at SSRN 3123545*.

Kundu, T., Sheu, J.-B., and Kuo, H.-T. (2022). Emergency logistics management—review and propositions for future research. *Transportation Research Part E: Logistics and Transportation Review*, 164:102789.

Kushner, H. J. and Yin, G. G. (2003). *Stochastic approximation and recursive algorithms and applications*. Springer.

Laroche, R., Trichelair, P., and Des Combes, R. T. (2019). Safe policy improvement with baseline bootstrapping. In *International Conference on Machine Learning*, pages 3652–3661. PMLR.

Larrañaga, M., Ayesta, U., and Verloop, I. M. (2015). Asymptotically optimal index policies for an abandonment queue with convex holding cost. *Queueing Systems*, 81(2):99–169.

Lee, C.-Y. and Lin, C.-S. (2001). Single-machine scheduling with maintenance and repair rate-modifying activities. *European Journal of Operational Research*, 135(3):493–513.

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

Lee, J., Bagheri, B., and Kao, H.-A. (2015). A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3:18–23.

Lee, T. Y. S. (2003). Analysis of random polling system with an infinite number of coupled servers and correlated input process. *Computers & Operations Research*, 30(13):2003–2020.

Lei, L., Yuan, Y., Vu, T. X., Chatzinotas, S., and Ottersten, B. (2019). Learning-based resource allocation: Efficient content delivery enabled by convolutional neural network. In *2019 IEEE 20th International Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pages 1–5. IEEE.

Letnik, T., Farina, A., Mencinger, M., Lupi, M., and Božičnik, S. (2018). Dynamic management of loading bays for energy efficient urban freight deliveries. *Energy*, 159:916–928.

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

Levy, H. and Sidi, M. (1990). Polling systems: applications, modeling, and optimization. *IEEE Transactions on Communications*, 38(10):1750–1760.

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, H., Parikh, D., He, Q., Qian, B., Li, Z., Fang, D., and Hampapur, A. (2014). Improving rail network velocity: A machine learning approach to predictive maintenance. *Transportation Research Part C: Emerging Technologies*, 45:17–26.

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.

Lim, H., Lee, G. M., and Singgih, I. K. (2021). Multi-depot split-delivery vehicle routing problem. *IEEE Access*, 9:112206–112220.

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

Liu, J., Li, C., and Luo, Y. (2024). Efficient resource allocation for IoT applications in mobile edge computing via dynamic request scheduling optimization. *Expert Systems with Applications*, 255:124716.

Liu, S., Huang, Y., and Shi, L. (2022). Autonomous mobile clinics: empowering affordable anywhere, anytime healthcare access. *IEEE Engineering Management Review*, 50(4):147–154.

Liu, S. and Luo, Z. (2023). On-demand delivery from stores: Dynamic dispatching and routing with random demand. *Manufacturing & Service Operations Management*, 25(2):595–612.

Long, Z., Shimkin, N., Zhang, H., and Zhang, J. (2020). Dynamic scheduling of multiclass many-server queues with abandonment: The generalized cμ/h rule. *Operations Research*, 68(4):1218–1230.

Long, Z., Zhang, H., Zhang, J., and Zhang, Z. G. (2024). The generalized c/μ rule for queues with heterogeneous server pools. *Operations Research*, 72(6):2488–2506.

Lu, L. and Wang, S. (2019). Literature review of analytical models on emergency vehicle service: location, dispatching, routing and preemption control. In *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*, pages 3031–3036. IEEE.

Luo, Q., Hu, S., Li, C., Li, G., and Shi, W. (2021). Resource scheduling in edge computing: A survey. *IEEE Communications Surveys & Tutorials*, 23(4):2131–2165.

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

Malazi, H. T., Chaudhry, S. R., Kazmi, A., Palade, A., Cabrera, C., White, G., and Clarke, S. (2022). Dynamic service placement in multi-access edge computing: A systematic literature review. *IEEE Access*, 10:32639–32688.

Mamane, A., Fattah, M., El Ghazi, M., El Bekkali, M., Balboul, Y., and Mazer, S. (2022). Scheduling algorithms for 5g networks and beyond: Classification and survey. *IEEe Access*, 10:51643–51661.

Mandelbaum, A., Momčilović, P., and Tseytlin, Y. (2012). On fair routing from emergency departments to hospital wards: QED queues with heterogeneous servers. *Management Science*, 58(7):1273–1291.

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

Mankowska, D. S., Meisel, F., and Bierwirth, C. (2014). The home health care routing and scheduling problem with interdependent services. *Health Care Management Science*, 17:15–30.

Manzoor, M. F., Abid, A., Farooq, M. S., Nawaz, N. A., and Farooq, U. (2020). Resource allocation techniques in cloud computing: A review and future directions. *Elektronika ir Elektrotechnika*, 26(6):40–51.

Marbach, P. and Tsitsiklis, J. N. (2002). Simulation-based optimization of Markov reward processes. *IEEE Transactions on Automatic Control*, 46(2):191–209.

Marsan, M. A., De Moraes, L., Donatelli, S., and Neri, F. (1990). Analysis of symmetric nonexhaustive polling with multiple servers. In *IEEE INFOCOM'90*, pages 284–285. IEEE Computer Society.

Martin, A., Egaña, J., Flórez, J., Montalban, J., Olaizola, I. G., Quartulli, M., Viola, R., and Zorrilla, M. (2018). Network resource allocation system for qoe-aware delivery of media services in 5g networks. *IEEE Transactions on Broadcasting*, 64(2):561–574.

Martin, X. A., Hatami, S., Calvet, L., Peyman, M., and Juan, A. A. (2023). Dynamic reactive assignment of tasks in real-time automated guided vehicle environments with potential interruptions. *Applied Sciences*, 13(6):3708.

Matveev, A. (2018). On stabilizability of fluid multi-server polling systems with setups. *Cybernetics Physics*, 7(1):26–34.

Maxwell, M. S., Restrepo, M., Henderson, S. G., and Topaloglu, H. (2010). Approximate dynamic programming for ambulance redeployment. *INFORMS Journal on Computing*, 22(2):266–281.

Meyn, S. (2008). *Control techniques for complex networks*. Cambridge University Press.

Morshedlou, N., González, A. D., and Barker, K. (2018). Work crew routing problem for infrastructure network restoration. *Transportation Research Part B: Methodological*, 118:66–89.

Mosheiov, G. and Oron, D. (2021). A note on scheduling a rate modifying activity to minimize total late work. *Computers & Industrial Engineering*, 154:107138.

Nasrollahzadeh, A. A., Khademi, A., and Mayorga, M. E. (2018). Real-time ambulance dispatching and relocation. *Manufacturing & Service Operations Management*, 20(3):467–480.

Nino-Mora, J. (2001). Restless bandits, partial conservation laws and indexability. *Advances in Applied Probability*, 33(1):76–98.

Ogunfowora, O. and Najjaran, H. (2023). Reinforcement and deep reinforcement learning-based solutions for machine maintenance planning, scheduling policies, and optimization. *Journal of Manufacturing Systems*, 70:244–263.

O'Reilly, M. M., Krasnicki, S., Montgomery, J., Heydar, M., Turner, R., Van Dam, P., and Maree, P. (2024). Markov decision process and approximate dynamic programming for a patient assignment scheduling problem. *arXiv preprint arXiv:2406.18618*.

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

Paintsil, E. (2025). *Introduction to the Amazon Elastic Kubernetes Service*, pages 3–11. Apress, Berkeley, CA.

Pan, W. and Liu, S. Q. (2023). Deep reinforcement learning for the dynamic and uncertain vehicle routing problem. *Applied Intelligence*, 53(1):405–422.

Park, K., Jo, S., Shin, Y., and Moon, I. (2025). Flexible material handling system for multi-load autonomous mobile robots in manufacturing environments: a hierarchical reinforcement learning approach. *International Journal of Production Research*, pages 1–21.

Pillac, V., Gendreau, M., Guéret, C., and Medaglia, A. L. (2013). A review of dynamic vehicle routing problems. *European Journal of Operational Research*, 225(1):1–11.

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

Poonia, P. K. (2021). Performance assessment of a multi-state computer network system in series configuration using copula repair. *International Journal of Reliability and Safety*, 15(1-2):68–88.

Powell, W. B. (2007). *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons.

Powell, W. B. (2022). *Reinforcement Learning and Stochastic Optimization: A Unified Framework for Sequential Decisions: by Warren B. Powell (ed.), Wiley (2022). Hardback. ISBN 9781119815051.*, volume 22. Taylor & Francis.

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

Robert, P. and Roberts, J. (2010). A mean field approximation for the capacity of server-limited, gate-limited multi-server polling systems. *ACM SIGMETRICS Performance Evaluation Review*, 38(2):24–26.

Rocchetta, R., Bellani, L., Compare, M., Zio, E., and Patelli, E. (2019). A reinforcement learning framework for optimal operation and maintenance of power grids. *Applied Energy*, 241:291–301.

Ruiz-Hernandez, D., Pinar-Perez, J., and Delgado-Gomez, D. (2020). Multi-machine preventive maintenance scheduling with imperfect interventions: A restless bandit approach. *Computers & Operations Research*, 119:104927.

Sabharwal, N. and Pandey, P. (2020). *Introduction to GKE*, pages 1–23. Apress, Berkeley, CA.

Sabry, A. H. and Amirulddin, U. A. B. U. (2024). A review on fault detection and diagnosis of industrial robots and multi-axis machines. *Results in Engineering*, page 102397.

Saghafian, S. and Veatch, M. H. (2015). A c-mu rule for two-tiered parallel servers. *IEEE Transactions on Automatic Control*, 61(4):1046–1050.

Salahuddin, M. A., Sahoo, J., Glitho, R., Elbiaze, H., and Ajib, W. (2017). A survey on content placement algorithms for cloud-based content delivery networks. *IEEE Access*, 6:91–114.

Schmid, V. (2012). Solving the dynamic ambulance relocation and dispatching problem using approximate dynamic programming. *European Journal of Operational Research*, 219(3):611–621.

Sennott, L. I. (1989). Average cost optimal stationary policies in infinite state markov decision processes with unbounded costs. *Operations Research*, 37(4):626–633.

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

Seyedshohadaie, S. R., Damnjanovic, I., and Butenko, S. (2010). Risk-based maintenance and rehabilitation decisions for transportation infrastructure networks. *Transportation Research Part A: Policy and Practice*, 44(4):236–248.

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.

Shone, R., Glazebrook, K., and Zografos, K. G. (2019). Resource allocation in congested queueing systems with time-varying demand: An application to airport operations. *European Journal of Operational Research*, 276(2):566–581.

Shone, R., Knight, V., and Harper, P. (2020). A conservative index heuristic for routing problems with multiple heterogeneous service facilities. *Mathematical Methods of Operations Research*, 92:511–543.

Silano, G., Bednar, J., Nascimento, T., Capitan, J., Saska, M., and Ollero, A. (2021). A multi-layer software architecture for aerial cognitive multi-robot systems in power line inspection tasks. In *2021 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 1624–1629. IEEE.

Simão, T. D., Suilen, M., and Jansen, N. (2023). Safe policy improvement for pomdps via finite-state controllers. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pages 15109–15117.

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

Spall, J. C. (2005). *Introduction to stochastic search and optimization: estimation, simulation, and control*. John Wiley & Sons.

Su, R., Zhang, D., Venkatesan, R., Gong, Z., Li, C., Ding, F., Jiang, F., and Zhu, Z. (2019). Resource allocation for network slicing in 5g telecommunication networks: A survey of principles and models. *IEEE Network*, 33(6):172–179.

Sun, Y., Lang, M., and Wang, D. (2015). Optimization models and solution algorithms for freight routing planning problem in the multi-modal transportation networks: A review of the state-of-the-art. *The Open Civil Engineering Journal*, 9:714–723.

Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition.

Tang, G. Q. (2011). Smart grid management & visualization: Smart power management system. In *2011 8th International Conference & Expo on Emerging Technologies for a Smarter World*, pages 1–6. IEEE.

Tavasszy, L., de Bok, M., Alimoradi, Z., and Rezaei, J. (2020). Logistics decisions in descriptive freight transportation models: A review. *Journal of Supply Chain Management Science*, 1(3-4):74–86.

Tian, D. and Shone, R. (2026a). Dynamic repair and maintenance of heterogeneous machines dispersed on a network: A rollout method for online reinforcement learning. *Computers & Operations Research*. In press.

Tian, D. and Shone, R. (2026b). Stochastic dynamic job scheduling with interruptible setup and processing times: An approach based on queueing control. *European Journal of Operational Research*, 329(3):920–934.

Tijms, H. (2003). *A First Course in Stochastic Models*. Wiley & Sons, Chichester.

Ulmer, M. W. (2020). Dynamic pricing and routing for same-day delivery. *Transportation Science*, 54(4):1016–1033.

Um, T.-W., Lee, H., Ryu, W., and Choi, J. K. (2014). Dynamic resource allocation and scheduling for cloud-based virtual content delivery networks. *Etri Journal*, 36(2):197–205.

van der Mei, R. D. and Borst, S. C. (1997). Analysis of multiple-server polling systems by means of the power-series algorithm. *Stochastic Models*, 13(2):339–369.

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

Villani, V., Pini, F., Leali, F., and Secchi, C. (2018). Survey on human–robot collaboration in industrial settings: Safety, intuitive interfaces and applications. *Mechatronics*, 55:248–266.

Vlachos, I., Pascazzi, R. M., Ntotis, M., Spanaki, K., Despoudi, S., and Repoussis, P. (2024). Smart and flexible manufacturing systems using Autonomous Guided Vehicles (AGVs) and the Internet of Things (IoT). *International Journal of Production Research*, 62(15):5574–5595.

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.

Wang, S., Wan, J., Li, D., and Zhang, C. (2016). Implementing smart factory of industrie 4.0: an outlook. *International Journal of Distributed Sensor Networks*, 12(1):3159805.

Wang, X., Wang, K., Wu, S., Di, S., Jin, H., Yang, K., and Ou, S. (2018). Dynamic resource scheduling in mobile edge cloud with cloud radio access network. *IEEE Transactions on Parallel and Distributed Systems*, 29(11):2429–2445.

Wang, X., Wang, X., Che, H., Li, K., Huang, M., and Gao, C. (2015). An intelligent economic approach for dynamic resource allocation in cloud services. *IEEE Transactions on Cloud Computing*, 3(3):275–289.

Ward, T., Jenab, K., Ortega-Moody, J., and Staub, S. (2024). A comprehensive review of machine learning techniques for condition-based maintenance. *International Journal of Prognostics and Health Management*, 15(2):3850–3870.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.

Weber, R. R. and Weiss, G. (1990). On an index policy for restless bandits. *Journal of Applied Probability*, 27(3):637–648.

Whittle, P. (1988). Restless bandits: Activity allocation in a changing world. *Journal of Applied Probability*, 25(A):287–298.

Woo, Y.-B., Jung, S., and Kim, B. S. (2017). A rule-based genetic algorithm with an improvement heuristic for unrelated parallel machine scheduling problem with time-dependent deterioration and multiple rate-modifying activities. *Computers & Industrial Engineering*, 109:179–190.

Xia, L., Zhang, Z. G., and Li, Q.-L. (2022). A c/μ-rule for job assignment in heterogeneous group-server queues. *Production and Operations Management*, 31(3):1191–1215.

Xiao, Z., Song, W., and Chen, Q. (2012). Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE Transactions on Parallel and Distributed Systems*, 24(6):1107–1117.

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

Xu, S., Panwar, S. S., Kodialam, M., and Lakshman, T. (2020). Deep neural network approximated dynamic programming for combinatorial optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 1684–1691.

Yadav, R., Zhang, W., Li, K., Liu, C., and Laghari, A. A. (2021). Managing overloaded hosts for energy-efficiency in cloud data centers. *Cluster Computing*, pages 1–15.

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.

Yang, X. and Strauss, A. K. (2017). An approximate dynamic programming approach to attended home delivery management. *European Journal of Operational Research*, 263(3):935–945.

Yang, Z., Huang, W., and Ding, H. (2024). BiLSTM-based selection and prediction of optimal polling systems for multiple server numbers. *Multimedia Tools and Applications*, pages 1–21.

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

Yousefi, N., Tsianikas, S., and Coit, D. W. (2020). Reinforcement learning for dynamic condition-based maintenance of a system with individually repairable components. *Quality Engineering*, 32(3):388–408.

Zhang, Q. (2023). Dynamic Routing Policies for Multi-Skill Call Centers Using Deep Q Network. *Mathematics*, 11(22):4662.

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.

Zheng, R., Liu, K., Zhu, J., Zhang, M., and Wu, Q. (2019). Stochastic resource scheduling via bilayer dynamic Markov decision process in mobile cloud networks. *Computer Communications*, 145:234–242.

Zolfaghari, B., Srivastava, G., Roy, S., Nemati, H. R., Afghah, F., Koshiba, T., Razi, A., Bibak, K., Mitra, P., and Rai, B. K. (2020). Content delivery networks: State of the art, trends, and future roadmap. *ACM Computing Surveys (CSUR)*, 53(2):1–34.

Appendix A

Appendices for Chapter 2

A.1 Proof of Theorem 2.1.

Proof. Consider a modified MDP in which the state space is

$$\tilde{S} := \{(v, w, (x_1, ..., x_d)) \mid v \in V,\ w \in D,\ x_i \geq 0 \text{ for } i \in D\}, \tag{A.1}$$

where the extra variable w represents the most recent demand point at which the server witnessed no jobs present. More precisely, if $w = i$ then this indicates that at a certain time step t_0 in the history of the process the system was in a state with $v = i$ and $x_i = 0$, and none of the states visited in the more recent time steps (between t_0 and the present time) had the server at another demand point $j \in D \setminus \{i\}$ at which there were no jobs present. (We can set w to an arbitrary value when the process is initialized.) All other aspects of the modified MDP formulation (e.g. actions and costs) are the same as in the original version. The transitions of the process do not lose their memoryless property when w is included, since the knowledge that $w = i$ at a particular time step is sufficient to specify the probability distribution for its value at the next step; specifically, w is guaranteed to remain unchanged unless either of the following two cases applies:

1. The server is at a demand point $j \in D \setminus \{i\}$ with $x_j = 1$ and chooses action j, in which case there is a probability of $\mu_j\Delta$ that we have $w = j$ at the next time step and a probability of $1 - \mu_j\Delta$ that we still have $w = i$.

2. The server is at an intermediate stage $k \in N$ adjacent to a demand point $j \in D\setminus\{i\}$ with $x_j = 0$ and chooses action j, in which case there is a probability of $\tau\Delta$ that

we have $w = j$ at the next time step and a probability of $1 - \tau\Delta$ that we still have $w = i$.

Consider a 'polling system' policy $\theta^{[P]}$ under which the server visits the demand points in a repeating sequence $(1, 2, ..., d, 1, 2, ..., d, 1, 2, ...)$ and, upon arriving at any demand point i, remains there until the number of jobs has been reduced to zero ($x_i = 0$) before moving to the next demand point in the sequence. If we already have $x_i = 0$ when the server arrives at node i, then the server immediately moves to the next demand point. It should be noted that, since switches require an exponentially distributed amount of time, it may happen that the server decides to move away from demand point i when $x_i = 0$ but a new job arrives at i while the server is still located at i. In this case, under our proposed policy, the server continues trying to move to the next demand point rather than processing the new job. We assume that the server always chooses the shortest path (in terms of the number of intermediate stages that must be traversed) between two demand points and, in the case where two or more paths are tied for the shortest length, the path is selected according to some fixed priority ordering of the nodes. This policy can be represented as a stationary policy in our modified MDP by specifying actions according to the simple rule that if $w = i \leq d - 1$ then the server attempts to move towards demand point $i + 1$ by taking the next step on the shortest path to that node; or, if it is already at $i + 1$, then it remains there. Similarly if $w = d$ then the server attempts to move to demand point 1 (or remains there).

Under the proposed policy $\theta^{[P]}$, the system behaves as a polling system with an exhaustive polling regime (meaning that demand points are served until they are empty). Given that $\rho < 1$, Lemma 3.1 in Altman et al. (1992) implies that the system is stable and there exists a probability distribution $\{\pi_{\theta^{[P]}}(\mathbf{x})\}_{\mathbf{x} \in \tilde{S}}$ such that $\pi_{\theta^{[P]}}(\mathbf{x})$ is the long-run proportion of time spent in state $\mathbf{x} \in \tilde{S}$. In the next part of the proof we show that it is possible to use value iteration to compute an optimal policy for the modified MDP. Given that the state space S is infinite, this requires the use of the 'approximating sequences' method developed in Sennott (1999). Let Ψ denote the modified MDP with state space $\tilde{S}$ defined in (A.1) and let $(\Psi_0, \Psi_1, \Psi_2, ...)$ denote a sequence of MDPs that are defined on finite spaces, so that the MDP Ψ_m (for $m \in \mathbb{N}_0$) has state space $\tilde{S}_m$ given by

$$\tilde{S}_m := \{(v, w, (x_1, ..., x_d)) \mid v \in V,\ w \in D,\ 0 \leq x_i \leq m \text{ for } i \in D\}.$$

Thus, in the MDP Ψ_m, we do not allow the number of jobs at any node $i \in D$ to be greater than m. We do this by modifying the transition probabilities so that if $\mathbf{x}$ is a

state with $x_i = m$ for some $i \in D$, then the arrival of a new job at node i is impossible and instead the 'self-transition' probability $p_{\mathbf{x},\mathbf{x}}(a)$ is increased by $\lambda_i \Delta$. Let θ^*_m be an optimal policy for the MDP Ψ_m. We can show that the sequence $(\theta^*_0, \theta^*_1, ...)$ converges to an optimal policy for the infinite-state MDP Ψ, but this requires certain conditions to be verified. Specifically, we must show that the assumptions (AC1)-(AC4) described on p. 169 of Sennott (1999) hold for the sequence $(\Psi_0, \Psi_1, \Psi_2, ...)$. (See also Sennott (1997), pp. 117-118 for an equivalent set of assumptions.)

Assumption (AC1) in Sennott (1999) states that, for the finite-state MDP Ψ_m, there exists a constant g_m and a function $h_m : \tilde{S}_m \to \mathbb{R}$ satisfying

$$g_m + h_m(\mathbf{x}) = f(\mathbf{x}) + \min_{a \in A_{\mathbf{x}}} \left\{ \sum_{\mathbf{y} \in \tilde{S}} p_{\mathbf{x},\mathbf{y}}(a) h_m(\mathbf{y}) \right\} \qquad \forall \mathbf{x} \in \tilde{S}_m. \tag{A.2}$$

We can easily show that, given any two states $\mathbf{x}, \mathbf{y} \in \tilde{S}_m$, there exists a stationary policy θ such that $\mathbf{y}$ is accessible from $\mathbf{x}$ in the Markov chain induced by θ. This can be achieved by considering arbitrary states $\mathbf{x}, \mathbf{y} \in \tilde{S}_m$ and identifying a sequence of random transitions that would cause the system to transition from $\mathbf{x}$ to $\mathbf{y}$ under policy θ (which can be specified differently for each pair of states $(\mathbf{x}, \mathbf{y})$). It then follows from Puterman (1994) (p. 478) that Ψ_m belongs to the 'communicating' class of multichain MDP models, and the constant g_m in (A.2) is the optimal long-run average cost for Ψ_m. Moreover, the values $h_m(\mathbf{x})$ for states $\mathbf{x} \in \tilde{S}_m$ can be computed using the well-known method of value iteration, in which we define $v_m^{(k)}(\mathbf{x})$ as the optimal (minimal) total cost in a finite-horizon problem with k stages initialized in state $\mathbf{x}$ and then compute $h_m(\mathbf{x}) = \lim_{k\to\infty}(v_m^{(k)}(\mathbf{x}) - v_m^{(k)}(\mathbf{z}))$, with the reference state $\mathbf{z} \in \tilde{S}_m$ chosen arbitrarily.

In order to verify assumptions (AC2)-(AC4) we will need to use several properties of the functions $v_m^{(k)}(\mathbf{x})$. Let $\mathbf{x}^{i+}$ denote a state identical to $\mathbf{x}$ except that one extra job is present at demand point $i \in D$. The required properties are:

1. $v_m^{(k)}(\mathbf{x}) \le v_m^{(k+1)}(\mathbf{x}) \quad \forall\, m, k \in \mathbb{N}_0,\ \mathbf{x} \in \tilde{S}_m.$ (A.3)
2. $v_m^{(k)}(\mathbf{x}) \le v_m^{(k)}(\mathbf{x}^{i+}) \quad \forall\, m, k \in \mathbb{N}_0,\ i \in D,\ \mathbf{x} \in \tilde{S}_m$ such that $x_i < m$. (A.4)
3. $v_m^{(k)}(\mathbf{x}) \le v_{m+1}^{(k)}(\mathbf{x}) \quad \forall\, m, k \in \mathbb{N}_0,\ \mathbf{x} \in \tilde{S}_m.$ (A.5)
4. Fix $m \in \mathbb{N}_0$ and let $\mathbf{x} = (v, w, (x_1, ..., x_d))$ and $\mathbf{x}' = (v', w', (x'_1, ..., x'_d))$ be two states in $\tilde{S}_m$ with $v = v'$ and $x_i = x'_i$ for $i = 1, ..., d,$ but $w \ne w'$. Then $v_m^{(k)}(\mathbf{x}) = v_m^{(k)}(\mathbf{x}')$ for $k \in \mathbb{N}_0$. (A.6)

All of the properties above are logical and can be proved using induction on k. We have omitted details of the induction arguments in order to avoid making this proof excessively long, but they are quite straightforward and only require some care in considering the different possible actions that might be chosen by the relevant finite-horizon optimal policies. Property (A.3) states that the optimal expected total cost is increasing with the number of stages remaining, k. Property (A.4) states that this cost is increasing with the number of jobs initially present at any demand point. Property (A.5) states that this cost is increasing with the maximum number of jobs, m, allowed to be present at any demand point. Finally, property (A.6) states that the variable w has no effect on the optimal expected total cost, which makes sense as it does not impose any constraints on the actions that may be chosen in the k remaining stages.

We proceed to verify (AC2)-(AC4). Assumption (AC2) states that $\limsup_{m\to\infty} h_m(\mathbf{x}) < \infty$ for each $\mathbf{x} \in \tilde{S}$. Consider the polling-type policy $\theta^{[P]}$ described earlier. It is clear that the Markov chain induced by $\theta^{[P]}$ has a unichain structure on the state space $\tilde{S}$, since the state $\mathbf{z} = (1, 1, (0, 0, ..., 0))$ is accessible from any other state under this policy (this can be seen from the fact that, in between consecutive visits to demand point 1, it is always possible for no new jobs to arrive at any demand points). Let $J_{\theta^{[P]}}(\mathbf{x}, \mathbf{z})$ denote the expected total cost incurred until the system enters state $\mathbf{z}$, given that it is initialized in state $\mathbf{x}$ and follows policy $\theta^{[P]}$. Since $\mathbf{z}$ is positive recurrent, it follows from standard theory (see Sennott (1999), pp. 298-302) that $J_{\theta^{[P]}}(\mathbf{x}, \mathbf{z}) < \infty$ for all $\mathbf{x} \in \tilde{S}$. Next, for $m \in \mathbb{N}_0$, let $J_{m,\theta^{[P]}}(\mathbf{x}, \mathbf{z})$ be defined in an analogous way to $J_{\theta^{[P]}}(\mathbf{x}, \mathbf{z})$ except that we consider applying the policy $\theta^{[P]}$ to the finite-state MDP Ψ_m instead of the infinite-state MDP Ψ. It can easily be shown that $J_{m,\theta^{[P]}}(\mathbf{x}, \mathbf{z}) < J_{\theta^{[P]}}(\mathbf{x}, \mathbf{z})$ for all $m \in \mathbb{N}$, since the amount of time that the server spends at any demand node (and, hence, the cost incurred) is stochastically smaller under Ψ_m than under Ψ. We now follow similar arguments to those in the proof of Proposition 8.2.1, p. 171 in Sennott (1999). By using property (A.3) and the fact that $v_m^{(k)}(\mathbf{x})$ is defined as the expected k-stage cost under an optimal policy, we have

$$v_m^{(k)}(\mathbf{x}) \leq v_m^{(p)}(\mathbf{x}) \leq v_{m,\theta}^{(p)}(\mathbf{x}) \quad \forall \mathbf{x} \in \tilde{S},\ m \in \mathbb{N}_0,\ p \geq k, \tag{A.7}$$

where $v_{m,\theta}^{(k)}$ is the expected k-stage cost under an arbitrary policy θ. Let θ be defined to mimic policy $\theta^{[P]}$ until the system reaches state $\mathbf{z}$ and then follow an optimal finite-horizon policy for k steps. Then, from (A.7) it follows that $v_m^{(k)}(\mathbf{x}) \leq J_{m,\theta^{[P]}}(\mathbf{x}, \mathbf{z}) +$

$v_m^{(k)}(\mathbf{z})$. Hence, using the previous arguments, we have

$$v_m^{(k)}(\mathbf{x}) - v_m^{(k)}(\mathbf{z}) \leq J_{m,\theta^{[P]}}(\mathbf{x},\mathbf{z}) \leq J_{\theta^{[P]}}(\mathbf{x},\mathbf{z}) \quad \forall m \in \mathbb{N}_0,\ \mathbf{x} \in \tilde{S}.$$

Since $h_m(\mathbf{x}) = \lim_{k\to\infty}(v_m^{(k)}(\mathbf{x}) - v_m^{(k)}(\mathbf{z}))$, this establishes that the sequence $\{h_m(\mathbf{x})\}_{m\in\mathbb{N}}$ is bounded above and hence $\limsup_{m\to\infty} h_m(\mathbf{x}) < \infty$ as required.

Assumption (AC3) states that there exists a constant $Q \geq 0$ such that $-Q \leq \liminf_{m\to\infty} h_m(\mathbf{x})$ for all $\mathbf{x} \in \tilde{S}$. By using property (A.4) and taking limits as $k \to \infty$, we obtain

$$h_m(\mathbf{x}) \leq h_m(\mathbf{x}^{i+}) \quad \forall i \in D,\ \mathbf{x} \in \tilde{S}_m \text{ such that } x_i < m. \tag{A.8}$$

This shows that the function h_m attains a minimum on the subset of states with no jobs present, which we denote as U. That is,

$$\arg\min_{\mathbf{x}\in\tilde{S}_m} h_m(\mathbf{x}) \in \{(v, w, (x_1, ..., x_d)) \mid v \in V,\ w \in D,\ x_i = 0 \text{ for all } i \in D\} =: U.$$

Let $\mathbf{u}^*$ be a state that attains the minimum above. We will assume that $\mathbf{u}^*$ is positive recurrent under $\theta^{[P]}$. However, this requires some justification. Recall that the server visits the demand points according to the sequence $(1, 2, ..., d, 1, 2, ..., d, 1, 2, ...)$ under policy $\theta^{[P]}$. Therefore, at any given time, the server's current node v must lie somewhere on the path between demand points w and $w + 1$ (if $w \leq d - 1$) or d and 1 (if $w = d$). Using property (A.6), we can assume that the variables v and w associated with state $\mathbf{u}^*$ do indeed satisfy these constraints, as it is always possible to change the value of w without making any difference to the optimal finite-stage expected cost. Also, if $\mathbf{u}^*$ is a state with $v \notin D$ (i.e. the server is at an intermediate stage rather than a demand point), it is reasonable to assume that node v is visited during the server's cyclic route under policy $\theta^{[P]}$, since if this is not the case, we can always modify the policy $\theta^{[P]}$ slightly so that node v is visited at some point during the server's route.

Using similar arguments to those given for (AC2), we then have that $\mathbf{u}^*$ is positive recurrent under $\theta^{[P]}$ and $J_{m,\theta^{[P]}}(\mathbf{x},\mathbf{u}^*) \leq J_{\theta^{[P]}}(\mathbf{x},\mathbf{u}^*) < \infty$ for any $\mathbf{x} \in \tilde{S}_m$. In particular let us consider the state $\mathbf{z} = (1, 1, (0, 0, ..., 0))$. Repeating previous arguments, we have $v_m^{(k)}(\mathbf{z}) \leq J_{\theta^{[P]}}(\mathbf{z},\mathbf{u}^*) + v_m^{(k)}(\mathbf{u}^*)$. Taking limits as $k \to \infty$, we obtain $h_m(\mathbf{u}) \geq h_m(\mathbf{u}^*) \geq -J_{\theta^{[P]}}(\mathbf{z},\mathbf{u}^*)$ for all $\mathbf{u} \in U$. This establishes the required lower bound for states in U, and it follows from (A.8) that the same lower bound also works for all $\mathbf{x} \in \tilde{S}_m$. Since this argument can be repeated for each $m \in \mathbb{N}$ (establishing a uniform lower bound independent of m), we have $-J_{\theta^{[P]}}(\mathbf{z},\mathbf{u}^*) \leq \liminf_{m\to\infty} h_m(\mathbf{x})$ for all $\mathbf{x} \in \tilde{S}$ as required.

Assumption (AC4) states that $\limsup_{m\to\infty} g_m =: g^* < \infty$ and $g^* \le g(\mathbf{x})$ for $\mathbf{x} \in \tilde{S}$, where g_m is the constant that appears in (A.2) and the notation $g(\mathbf{x})$ allows for a possible dependence of the long-run average cost on the initial state $\mathbf{x}$. By Proposition 8.2.1, Step 3(i) in Sennott (1999), it is sufficient to show that

$$v_m^{(k)}(\mathbf{x}) \le \lim_{p\to\infty} v_p^{(k)}(\mathbf{x}) \quad \forall\, m, k \in \mathbb{N}_0,\ \mathbf{x} \in \tilde{S}_m, \tag{A.9}$$

which follows immediately from property (A.5).

Having verified that assumptions (AC1)-(AC4) hold for the modified MDP Ψ, we can use the results in Sennott (1999) to conclude that any limit point of a sequence of stationary optimal policies for the finite-state MDPs $(\Psi_0, \Psi_1, \Psi_2, ...)$ is optimal for Ψ (Theorem 8.1.1) and furthermore a limit point is guaranteed to exist (Proposition B.5). By the previous arguments, we can compute an optimal policy for any finite-state MDP Ψ_m using value iteration. During the process of value iteration, the functions $v_m^{(k)}(\mathbf{x})$ for $\mathbf{x} \in \tilde{S}_m$ are computed using the rule

$$v_m^{(k+1)}(\mathbf{x}) = f(\mathbf{x}) + \min_{a\in A_{\mathbf{x}}} \left\{ \sum_{\mathbf{y}\in\tilde{S}_m} p_{\mathbf{x},\mathbf{y}}(a) v_m^{(k)}(\mathbf{y}) \right\}, \quad k \in \mathbb{N}_0.$$

Property (A.6) implies that if $\mathbf{x}$ and $\mathbf{x}'$ are two states in $\tilde{S}_m$ that differ from each other only in the variable w, then any action $a \in A_{\mathbf{x}}$ that attains the minimum in the equation above for state $\mathbf{x}$ is also a feasible action that attains the minimum in the corresponding equation for state $\mathbf{x}'$. Essentially, this means that it is possible to find an optimal stationary policy for Ψ_m that chooses actions independently of the variable w. By the previous arguments, the same property also applies to an optimal stationary policy for the infinite-state MDP Ψ (obtained as a limit of the optimal finite-state policies). However, if we have an optimal stationary policy that chooses actions independently of w, then the same policy must also be admissible for the MDP formulated in Section 2.2 with state space S. From (AC4) we also know that the long-run average cost under such a policy is finite, implying stability. This completes the proof. □

A.2 DVO Heuristic

In this appendix we describe the steps of the DVO heuristic as presented in Duenyas and Van Oyen (1996). Note that, where appropriate, we adapt the authors' notation so

that it is consistent with the notation used in our paper.

1. If the server has just finished processing a job at some demand point $i \in D$ then there are two possible cases: either (a) there are still some jobs remaining at i ($x_i > 0$) or (b) there are no jobs remaining at i ($x_i = 0$).

   (a) In the first case, the server can either continue processing jobs at i or switch to some other demand point $j \neq i$. We carry out the following steps:

      i. Initialize an empty set $\sigma = \emptyset$.

      ii. For each demand point j with $c_j\mu_j \geq c_i\mu_i$, calculate the reward rate ψ_j that would be earned by switching to node j, serving it exhaustively, then switching back to node i, using

      $$\psi_j = c_j\mu_j \frac{x_j + \lambda_j\delta(i,j)/\tau}{x_j + \mu_j\delta(i,j)/\tau + (\mu_j - \lambda_j)\delta(j,i)/\tau}.$$

      If $\psi_j \geq c_j\mu_j\rho + c_i\mu_i(1-\rho)$, then add j to the set σ.

      iii. If σ is non-empty, then switch to the demand point j with the highest index ψ_j (with ties broken arbitrarily). Otherwise, process one more job at node i.

   (b) In the second case, the server can either remain idle at i or switch to some other demand point $j \neq i$. We carry out the following steps:

      i. Initialize three empty sets: $\sigma_1 = \emptyset$, $\sigma_2 = \emptyset$ and $\sigma = \emptyset$.

      ii. For each demand point $j \neq i$, calculate the reward rate ϕ_j in a similar way to part (a) but without including the time taken to switch back from j to i, using

      $$\phi_j = c_j\mu_j \frac{x_j + \lambda_j\delta(i,j)/\tau}{x_j + \mu_j\delta(i,j)/\tau}.$$

      If $\phi_j > c_j\mu_j\rho$, then add j to σ_1; otherwise, add j to σ_2.

      iii. If σ_1 is non-empty, let $\sigma = \sigma_1$. Otherwise, let $\sigma = \sigma_2$.

      iv. Let j^* denote the demand point in σ with the highest reward rate ϕ_{j^*}, with ties broken arbitrarily. If $x_{j^*} > \lambda_{j^*}\delta(j^*,i)/\tau$, then switch to j^*. Otherwise, remain idle at i.

2. If the server has just arrived at a demand point then it immediately begins processing jobs there if there is at least one job waiting. This ensures that, after switching to a new demand point, it must process at least one job there before

switching somewhere else. If there are no jobs waiting, then the rule for idling described in step 1(b) is used.

3. If the server is idle at a demand point and a new job arrives in the system, then the rule for idling described in step 1(b) is used.

As mentioned in Section 2.3, there is no need to specify the rule used by the DVO heuristic when the server is at an intermediate stage $i \in N$, since the server is required to continue moving towards a particular demand point (chosen at the previous decision epoch) in this case. Similarly, if the server is at a non-empty demand point $i \in D$ and has *not* just finished processing a job, then this indicates that a service is in progress and the server is required to remain at node i.

A.3 Proof of Theorem 2.3.

Proof. Let $\mathbf{x} = (v, (x_1, ..., x_d))$ be the current state of the system, where $v \in N$. We will use T_{arr} to denote the random amount of time until the next job arrives in the system, and T_{switch} to denote the amount of time until the server reaches a demand point. According to the rules of the K-stop heuristic, a sequence s can only be added to the set σ if it satisfies the condition $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$. We begin by showing that there always exists at least one sequence that satisfies this condition, and therefore σ_2 (and, hence, σ) must be non-empty. Indeed, the set $\mathcal{S}$ is defined to include all sequences of length m, for each $1 \leq m \leq K$. Therefore it includes the sequences of length one, (j), for each $j \in D$. Let $s = (j)$, where $j \in D$ is arbitrary. Then, using (2.8), we have

$$\begin{aligned}
\psi(\mathbf{x}, (j), t) &= c_j \mu_j \left\{ \frac{T_1(\mathbf{x}, (j), t)}{t + \delta(v, j)/\tau + T_1(\mathbf{x}, (j), t)} \right\} \\
&= c_j \mu_j \left\{ \frac{[x_j + \lambda_j(t + \delta(v, j)/\tau)]/(\mu_j - \lambda_j)}{t + \delta(v, j)/\tau + [x_j + \lambda_j(t + \delta(v, j)/\tau)]/(\mu_j - \lambda_j)} \right\} \\
&= c_j \mu_j \left\{ \frac{x_j + \lambda_j(t + \delta(v, j)/\tau)}{x_j + \mu_j(t + \delta(v, j)/\tau)} \right\}.
\end{aligned}$$

After differentiating, we obtain

$$\frac{\partial}{\partial t}\psi(\mathbf{x}, (j), t) = -c_j \mu_j \left\{ \frac{x_j(\mu_j - \lambda_j)}{[x_j + \mu_j(t + \delta(v, j)/\tau)]^2} \right\}, \tag{A.10}$$

which equals zero if $x_j = 0$, and is negative otherwise. Hence, the sequence $s = (j)$ satisfies the condition $\frac{\partial}{\partial t}\psi(\mathbf{x}, s, t)\big|_{t=0} \leq 0$. We can therefore be sure that σ_2 is non-empty, but in order to proceed we must consider two possible subcases: either (a) the set σ_1 is non-empty and we choose the sequence in σ_1 with the highest value of $\psi(\mathbf{x}, s, 0)$, or (b) the set σ_1 is empty, but σ_2 is non-empty and we choose the sequence in σ_2 with the highest value of $\psi(\mathbf{x}, s, 0)$. In the remainder of this proof we consider these two subcases separately.

Subcase (a): σ_1 is non-empty

For convenience, we will use $\xi(\mathbf{x}, s, t)$ to denote the proportion of time spent processing jobs (as opposed to switching between nodes or idling) while following sequence s. That is:

$$\xi(\mathbf{x}, s, t) := \frac{\sum_{j=1}^{|s|} T_j(\mathbf{x}, s, t)}{t + \sum_{j=1}^{|s|} [\delta(s_{j-1}, s_j)/\tau + T_j(\mathbf{x}, s, t)]}, \quad t \geq 0. \tag{A.11}$$

Given that $T_j(\mathbf{x}, s, t) \equiv R_j(\mathbf{x}, s, t)/(c_{s_j}\mu_{s_j})$ for each $j = 1, ..., |s|$, we can use identical arguments to those in the proof of Lemma 2.2 to show that $\xi(\mathbf{x}, s, t)$ is a monotonic function of t. Observe that the condition $\psi(\mathbf{x}, s, 0) \geq \gamma(\mathbf{x}, s, 0)$ is equivalent to

$$\xi(\mathbf{x}, s, 0) \geq \rho. \tag{A.12}$$

Let s^* denote the sequence in σ_1 that maximizes $\psi(\mathbf{x}, s, 0)$. In this case, according to the rules of the heuristic, the server attempts to take one step along a shortest path to demand point s_1^*, where s_1^* is the first element of s^*. Let $(i_0, i_1, ..., i_k)$ denote a shortest path from v to s_1^*, where $k := \delta(v, s_1^*)$, $i_0 := v$ and $i_k := s_1^*$. Also let $\mathbf{x}_j$ denote a state identical to $\mathbf{x}$ except that the server is located at node i_j, for $j = 1, 2, ..., k$. It is useful to note that

$$\xi(\mathbf{x}_j, s^*, 0) = \xi(\mathbf{x}_k, s^*, (k-j)/\tau) \quad \forall j \in \{1, 2, ..., k-1\}. \tag{A.13}$$

This is because $\xi(\mathbf{x}_j, s^*, 0)$ represents the proportion of time spent processing jobs given that the server begins at node i_j and immediately begins traveling along a path of length $(k-j)$ in order to reach node s_1^*, while $\xi(\mathbf{x}_k, s^*, (k-j)/\tau)$ is the corresponding proportion given that the server begins at node $i_k = s_1^*$ but waits for $(k-j)/\tau$ time units before beginning to process jobs there. In terms of the proportion of time spent processing jobs while following sequence s^*, these two quantities are the same. In the case $|s^*| = 1$

(that is, $s^* = (j)$ for some $j \in D$) we infer from (A.10) that

$$\frac{\partial}{\partial t}\xi(\mathbf{x},(j),t) = \frac{1}{c_j\mu_j}\frac{\partial}{\partial t}\psi(\mathbf{x},(j),t) \le 0, \tag{A.14}$$

and hence, using (A.13), we can be assured that the condition $\xi(\mathbf{x}, s^*, 0) \ge \rho$ (equivalent to $\psi(\mathbf{x}, s^*, 0) \ge \gamma(\mathbf{x}, s^*, 0)$) remains satisfied as the server moves towards s_1^*. Therefore, according to the rules of the heuristic, s^* remains included in σ_1 at all stages while the server moves from v to s_1^*. On the other hand, consider the case $|s^*| \ge 2$. In this case the rules of the heuristic imply that the extra condition $\xi(\mathbf{y}, s^*, 0) \ge \rho$ must be satisfied, where $\mathbf{y}$ is equivalent to $\mathbf{x}_k$ in the notation of this proof. By the same reasoning used to derive (A.13), we have

$$\xi(\mathbf{x}, s^*, 0) = \xi(\mathbf{x}_k, s^*, k/\tau). \tag{A.15}$$

Since $\xi(\mathbf{x}_k, s^*, t)$ is a monotonic function of t, we can infer from (A.13) and (A.15) that

$$\min\{\xi(\mathbf{x}, s^*, 0),\ \xi(\mathbf{x}_k, s^*, 0)\} \le \xi(\mathbf{x}_j, s^*, 0) \le \max\{\xi(\mathbf{x}, s^*, 0),\ \xi(\mathbf{x}_k, s^*, 0)\} \quad \forall j \in \{1, 2, ..., k\}. \tag{A.16}$$

Given that $s^* \in \sigma_1$, the rules of the heuristic imply that $\xi(\mathbf{x}, s^*, 0) \ge \rho$ and $\xi(\mathbf{x}_k, s^*, 0) \ge \rho$, so from (A.16) it follows that $\xi(\mathbf{x}_j, s^*, 0) \ge \rho$ for each $j = 1, 2, ..., k$. Also, given that $s^* \in \sigma_1$, the heuristic rules imply $\frac{\partial}{\partial t}\psi(\mathbf{x}, s^*, t)\big|_{t=0} \le 0$. Applying the same reasoning to the function ψ that we used for ξ, we have

$$\psi(\mathbf{x}_j, s^*, 0) = \psi(\mathbf{x}_k, s^*, (k-j)/\tau) \quad \forall j \in \{1, 2, ..., k\}, \tag{A.17}$$

$$\psi(\mathbf{x}, s^*, 0) = \psi(\mathbf{x}_k, s^*, k/\tau). \tag{A.18}$$

From the proof of Lemma 2.2 we know that $\frac{\partial}{\partial t}\psi(\mathbf{x}, s^*, t)$ has the same sign for all $t \ge 0$. Hence, from (A.18) it follows that $\frac{\partial}{\partial t}\psi(\mathbf{x}_k, s^*, t)\big|_{t=0} \le 0$ and then from (A.17) it follows that $\frac{\partial}{\partial t}\psi(\mathbf{x}_j, s^*, t)\big|_{t=0} \le 0$ for each $j = 1, 2, ..., k$. By collating the arguments given thus far, we can say that as the server travels from v to s_1^* along the shortest path $(i_0, i_1, ..., i_k)$, the sequence s^* always satisfies the necessary conditions to be included in σ_1.

Now suppose that the server arrives at node i_1 before time T_{arr}. If $i_1 = s_1^*$ (i.e. the shortest path from v to s_1^* is of length 1), then there is nothing further to prove, as the server has completed its journey to demand point s_1^* via the shortest possible path. On the other hand, if $i_1 \ne s_1^*$ (i.e. the shortest path from v to s_1^* has length greater than 1), then we must continue. Given that $\frac{\partial}{\partial t}\psi(\mathbf{x}, s^*, t)\big|_{t=0} \le 0$ and using Lemma 2.2 again, it must be the case that $\psi(\mathbf{x}, s^*, 0) \le \psi(\mathbf{x}_1, s^*, 0)$; in other words, the average reward

does not decrease after the server takes a step along the shortest path to s_1^*. By the previous arguments, s^* is still included in σ_1 when the server is located at i_1. However, we cannot be sure that s^* is chosen by the heuristic at state $\mathbf{x}_1$.

If s^* is chosen by the heuristic at $\mathbf{x}_1$, then the server continues attempting to move along the shortest path to node s_1^*. On the other hand, suppose the heuristic chooses an alternative sequence $\tilde{s} \in \sigma_1$ at $\mathbf{x}_1$, and let $\tilde{s}_1$ be the first demand point in sequence $\tilde{s}$. If $\tilde{s}_1 = s_1^*$, then (once again) we have no difficulties, as the server continues attempting to move along the shortest path to s_1^*. In the rest of this subcase we assume a non-trivial case where $\tilde{s}_1 \neq s_1^*$.

We can show that although $\tilde{s}_1 \neq s_1^*$, the server's movement from node v to i_1 still qualifies as a step along a shortest path from v to $\tilde{s}_1$. In other words, even if the server now prefers to move towards a different demand point, the step from v to i_1 was still a step in the right direction. To see this, first note that

$$\psi(\mathbf{x}, \tilde{s}, 0) \leq \psi(\mathbf{x}, s^*, 0) \leq \psi(\mathbf{x}_1, s^*, 0) \leq \psi(\mathbf{x}_1, \tilde{s}, 0), \tag{A.19}$$

where the first inequality is due to the fact that s^* is preferred to $\tilde{s}$ at state $\mathbf{x}$, the second inequality is due to the fact that $\frac{\partial}{\partial t}\psi(\mathbf{x}, s^*, t)\big|_{t=0} \leq 0$ (as stated earlier) and the third inequality is due to the fact that $\tilde{s}$ is preferred to s^* under the new state $\mathbf{x}_1$. The rules of the heuristic state that if the average rewards for two sequences are equal, then a sequence is chosen according to a fixed priority ordering of the demand points in D. Therefore either the first inequality or the third inequality in (A.19) must be strict, as if they both hold with equality then the same sequence (either s^* or $\tilde{s}$) must be chosen at both $\mathbf{x}$ and $\mathbf{x}_1$. We conclude that

$$\psi(\mathbf{x}, \tilde{s}, 0) < \psi(\mathbf{x}_1, \tilde{s}, 0), \tag{A.20}$$

implying that the average reward for sequence $\tilde{s}$ has increased after moving from v to i_1. Let $\tilde{\mathbf{x}}$ denote a state identical to $\mathbf{x}$ except that the server is located at node $\tilde{s}_1$ instead of v. By definition, we have

$$\psi(\mathbf{x}, \tilde{s}, 0) = \psi(\tilde{\mathbf{x}}, \tilde{s}, \delta(v, \tilde{s}_1)/\tau), \tag{A.21}$$

$$\psi(\mathbf{x}_1, \tilde{s}, 0) = \psi(\tilde{\mathbf{x}}, \tilde{s}, \delta(i_1, \tilde{s}_1)/\tau). \tag{A.22}$$

Suppose (for a contradiction) that $\delta(i_1, \tilde{s}_1) \geq \delta(v, \tilde{s}_1)$. If these two distances are equal then from (A.21)-(A.22) we have $\psi(\mathbf{x}, \tilde{s}, 0) = \psi(\mathbf{x}_1, \tilde{s}, 0)$, giving a contradiction with

(A.20). On the other hand, if the inequality is strict (that is, $\delta(i_1, \tilde{s}_1) > \delta(v, \tilde{s}_1)$) then using Lemma 2.2 we infer that $\frac{\partial}{\partial t}\psi(\mathbf{x}_1, \tilde{s}, t)\big|_{t=0} > 0$. Hence, according to the rules of the heuristic, the sequence $\tilde{s}$ could not have been chosen under state $\mathbf{x}_1$. We conclude that $\delta(i_1, \tilde{s}_1) < \delta(v, \tilde{s}_1)$ and therefore the server's step from v to i_1 represents a step along a shortest path from v to $\tilde{s}_1$.

Subcase (b): σ_1 is empty

From the arguments at the beginning of the proof we know that even if σ_1 is empty, σ_2 must be non-empty. As in case (a), let s^* denote the sequence chosen by the heuristic under state $\mathbf{x}$, and let $(i_0, i_1, ..., i_k)$ denote a shortest path from v to s_1^*, where $k = \delta(v, s_1^*)$, $i_0 = v$ and $i_k = s_1^*$. Also let $\mathbf{x}_j$ denote a state identical to $\mathbf{x}$ except that the server is located at node i_j, for $j = 1, 2, ..., k$. After the server moves from node v to i_1, there are two possible scenarios: either σ_1 remains empty, or it becomes non-empty. In the first scenario, we can simply repeat the relevant arguments used in subcase (a) to show that, even if the heuristic chooses some alternative sequence $\tilde{s} \in \sigma_2$ under state $\mathbf{x}_1$, it must be the case that $\delta(i_1, \tilde{s}_1) < \delta(v, \tilde{s}_1)$, and therefore the server's movement from v to i_1 qualifies as a step along a shortest path from v to $\tilde{s}_1$. This is due to the fact that $\frac{\partial}{\partial t}\psi(\mathbf{x}_1, \tilde{s}, t)\big|_{t=0}$ must be non-positive in order for $\tilde{s}$ to be chosen. In the rest of this part we consider the second scenario, where σ_1 becomes non-empty.

Suppose $\tilde{s} \in \sigma_1$ and the heuristic chooses sequence $\tilde{s}$ under state $\mathbf{x}_1$. It may be the case that $\tilde{s}_1 = s_1^*$, in which case the server simply keeps following the same path. However, if $\tilde{s}_1 \neq s_1^*$, we can again show that $\delta(i_1, \tilde{s}_1) < \delta(v, \tilde{s}_1)$. To see this, let $\tilde{\mathbf{x}}$ denote the state identical to $\mathbf{x}$ except that the server is located at node $\tilde{s}_1$. Due to the rules of the heuristic, we must have

$$\xi(\mathbf{x}_1, \tilde{s}, 0) \geq \rho \tag{A.23}$$

and additionally, if $|\tilde{s}| \geq 2$, then

$$\xi(\tilde{\mathbf{x}}, \tilde{s}, 0) \geq \rho. \tag{A.24}$$

Note that $\xi(\mathbf{x}_1, \tilde{s}, 0) = \xi(\tilde{\mathbf{x}}, \tilde{s}, \delta(i_1, \tilde{s}_1)/\tau)$. Suppose (for a contradiction) that $\delta(i_1, \tilde{s}_1) \geq \delta(v, \tilde{s}_1)$. Given that sequence $\tilde{s}$ was not included in σ_1 when the server was at state $\mathbf{x}$, at least one of the conditions $\frac{\partial}{\partial t}\psi(\mathbf{x}, \tilde{s}, t)\big|_{t=0} \leq 0$, $\xi(\mathbf{x}, \tilde{s}, 0) \geq \rho$ and $\xi(\tilde{\mathbf{x}}, \tilde{s}, 0) \geq \rho$ (where the latter only applies if $|\tilde{s}| \geq 2$) must fail to hold. However, given that $\tilde{s}$ is chosen at $\mathbf{x}_1$ and $\psi(\mathbf{x}_1, \tilde{s}, 0) \equiv \psi(\mathbf{x}, \tilde{s}, 1/\tau)$, it must be the case (using Lemma 2.2) that $\frac{\partial}{\partial t}\psi(\mathbf{x}_1, \tilde{s}, t)\big|_{t=0}$ has the same sign as $\frac{\partial}{\partial t}\psi(\mathbf{x}, \tilde{s}, t)\big|_{t=0}$, so the derivative condition holds. We also have $\xi(\tilde{\mathbf{x}}, \tilde{s}, 0) \geq \rho$ when $|\tilde{s}| \geq 2$ from (A.24), so we can proceed to assume that

$\xi(\mathbf{x}, \tilde{s}, 0) < \rho$, which is equivalent to $\xi(\tilde{\mathbf{x}}, \tilde{s}, \delta(v, \tilde{s}_1)/\tau) < \rho$. Hence, we have

$$\xi(\tilde{\mathbf{x}}, \tilde{s}, \delta(v, \tilde{s}_1)/\tau) < \rho \leq \xi(\tilde{\mathbf{x}}, \tilde{s}, \delta(i_1, \tilde{s}_1)/\tau) \tag{A.25}$$

but also (by assumption)

$$0 < \delta(v, \tilde{s}_1) \leq \delta(i_1, \tilde{s}_1), \tag{A.26}$$

implying that $\xi(\tilde{\mathbf{x}}, \tilde{s}, t)$ is increasing with t. If $|\tilde{s}| = 1$ then this gives a contradiction, since it was shown in (A.14) that $\xi(\cdot)$ is non-increasing with t for sequences of length one. On the other hand, if $|\tilde{s}| \geq 2$, we modify the right-hand side of (A.25) and combine (A.23)-(A.24) to give

$$\xi(\tilde{\mathbf{x}}, \tilde{s}, \delta(v, \tilde{s}_1)/\tau) < \rho \leq \min\{\xi(\tilde{\mathbf{x}}, \tilde{s}, 0),\ \xi(\tilde{\mathbf{x}}, \tilde{s}, \delta(i_1, \tilde{s}_1)/\tau)\},$$

which, along with (A.26), contradicts the fact that $\xi(\tilde{\mathbf{x}}, \tilde{s}, t)$ is a monotonic function of t. We conclude that $\delta(i_1, \tilde{s}_1) < \delta(v, \tilde{s}_1)$ as required.

We can repeat the arguments given in subcases (a) and (b) to show that any movement of the server from one intermediate stage to another (prior to $\min\{T_{\text{arr}},\ T_{\text{switch}}\}$) qualifies as a step along a shortest path from v to some particular demand point j^*. The key point is that even though the sequences chosen at the various intermediate stages may not always begin with node j^*, the distance from the currently-occupied node to j^* is always reduced after each step, so the complete path is indeed a shortest path from v to j^*. Given that the number of nodes in the network is finite, the demand point j^* must eventually be reached if no new jobs arrive in the meantime. This completes the proof. □

A.4 Proof of Corollary 2.4.

Proof. We assume that the server is initially located at some intermediate stage $v \in N$ and use $M = \max_{\{i \in N,\ j \in D\}} \delta(i, j)$ to denote the maximum distance between an intermediate stage and a demand point. Let α denote the number of new jobs that arrive in the system before the server reaches a demand point, given that the K-stop heuristic is followed. By conditioning on α, we have

$$\mathbb{E}[T_{\text{switch}}] = \sum_{k=0}^{\infty} \mathbb{E}[T_{\text{switch}} \mid \alpha = k]\ \mathbb{P}(\alpha = k). \tag{A.27}$$

Due to Theorem 2.3 we know that if the server is at an intermediate stage, then until the next new job arrives it attempts to move along a shortest path to some particular demand point j^*. Since (prior to T_{switch}) the server is always attempting to move, the expected amount of time until the next event (either a switch or the arrival of a new job in the system) is always $(\Lambda + \tau)^{-1}$, where $\Lambda = \sum_{i \in D} \lambda_i$. In the worst case, the number of nodes that must be traversed in order to reach j^* is M. Hence, we can form an upper bound for $\mathbb{E}[T_{\text{switch}} \mid \alpha = 0]$:

$$\mathbb{E}[T_{\text{switch}} \mid \alpha = 0] \leq \frac{M}{\Lambda + \tau}.$$

Extending this argument, we can obtain an upper bound for $\mathbb{E}[T_{\text{switch}} \mid \alpha = k]$ (for $k \geq 1$) by supposing that every time a new job arrives in the system, the server changes direction and attempts to move to a demand point M nodes away, and manages to complete $(M - 1)$ of these switches before a new job arrives in the system and forces it to change direction again. Suppose this pattern continues until k arrivals have occurred, at which point it manages to complete M switches without interruption and reaches a demand point. In this scenario, the total number of switches made is $k(M - 1) + M$ and the total number of new jobs arriving is k, so the total number of system events is $(k + 1)M$. Hence:

$$\mathbb{E}[T_{\text{switch}} \mid \alpha = k] \leq \frac{(k+1)M}{\Lambda + \tau}, \quad k \geq 0.$$

Next, consider the probabilities $\mathbb{P}(\alpha = k)$. Each time a system event occurs (prior to T_{switch}), there is a probability of $\tau/(\Lambda + \tau)$ that this is a switch rather than the arrival of a new job. We can obtain an upper bound for $\mathbb{P}(\alpha = 0)$ by supposing that the server only needs to complete one switch in order to reach its intended demand point. Hence:

$$\mathbb{P}(\alpha = 0) \leq \frac{\tau}{\Lambda + \tau}.$$

On the other hand, the largest possible probability of an arrival occurring before the server reaches a demand point is $1 - (\tau/(\Lambda + \tau))^M$, since this represents the case where the server must complete M switches in order to reach its intended demand point. Putting these arguments together, we have the following upper bound:

$$\mathbb{P}(\alpha = k) \leq \left[1 - \left(\frac{\tau}{\Lambda + \tau}\right)^M\right]^k \left(\frac{\tau}{\Lambda + \tau}\right), \quad k \geq 0.$$

Let $p := 1 - (\tau/(\Lambda+\tau))^M$ for notational convenience. Then, using (A.27), we have

$$\begin{aligned}
\mathbb{E}[T_{\text{switch}}] &\leq \sum_{k=0}^{\infty} \frac{(k+1)M}{\Lambda+\tau}\, p^k \left(\frac{\tau}{\Lambda+\tau}\right) \\
&= \frac{\tau M}{(\Lambda+\tau)^2} \sum_{k=0}^{\infty} (k+1)p^k \\
&= \frac{\tau M}{(\Lambda+\tau)^2} \cdot \frac{1}{(1-p)^2} \\
&= \frac{\tau M}{(\Lambda+\tau)^2} \left(\frac{\Lambda+\tau}{\tau}\right)^{2M} \\
&= \frac{M}{\tau} \left(\frac{\Lambda+\tau}{\tau}\right)^{2(M-1)}. \qquad \text{(A.28)}
\end{aligned}$$

This completes the proof. We also note that the bound holds with equality if and only if $M = 1$. □

A.5 Methods for generating the parameters for the numerical experiments in Section 2.4

For each of the 10,000 instances considered in Section 2.4, the system parameters are randomly generated as follows:

- The size of the left-hand cluster, d_1, is sampled unbiasedly from the set $\{1, 2, 3, 4\}$.
- The size of the right-hand cluster, d_2, is sampled unbiasedly from the set $\{1, 2, 3, 4\}$.
- The number of intermediate stages, n, is sampled unbiasedly from the set $\{1, 2, 3, 4, 5, 6\}$.
- The overall traffic intensity, ρ, is sampled from a continuous uniform distribution between 0.1 and 0.9. Subsequently, the job arrival rates λ_i and processing rates μ_i for $i \in D$ are generated as follows:
  - Each processing rate μ_i is initially sampled from a continuous uniform distribution between 0.1 and 0.9.
  - For each demand point $i \in D$, an initial value for the job arrival rate λ_i' is sampled from a continuous uniform distribution between $0.1\mu_i$ and μ_i.

- For each demand point $i \in D$, the actual traffic intensity ρ_i is obtained by re-scaling the initial traffic intensity λ_i'/μ_i, as follows:

$$\rho_i := \frac{\lambda_i'/\mu_i}{\sum_{i\in D} \lambda_i'/\mu_i}\rho.$$

This ensures that $\sum_{i\in D} \rho_i = \rho$.

- For each demand point $i \in D$, the actual job arrival rate λ_i is obtained as follows:

$$\lambda_i := \rho_i \mu_i.$$

- All of the λ_i and μ_i values are rounded to 2 significant figures (also causing a small change to the value of ρ).

• For each demand point $i \in D$, the holding cost c_i is sampled from a continuous uniform distribution between 0.1 and 0.9.

• In order to generate the switching rate τ, we first define $\eta := \tau/(\sum_{i\in D} \lambda_i)$ and generate the value of η as follows:

  - Sample a value p from a continuous uniform distribution between 0 and 1.
  - If $p < 0.5$, sample η from a continuous uniform distribution between 0.1 and 1.
  - If $p \geq 0.5$, sample η from a continuous uniform distribution between 1 and 10.

  We then define $\tau := \eta \sum_{i\in D} \lambda_i$.

A.6 Simulation methods for the numerical experiments in Section 2.4

In each of the 10,000 problem instances, the performances of the K-stop and (K from L)-stop heuristics are estimated by simulating the discrete-time evolution of the uniformized MDP described in Section 2.2. We also use a 'common random numbers' method to ensure that job arrivals occur at the same times under each of these policies. More specifically, the simulation steps are as follows:

1. Generate a set of random system parameters as described in Appendix A.5.
2. Set $N_0 := 10,000$ as the length of the warm-up period and $N_1 := 1,000,000$ as the length of the simulation.
3. Generate a list Z of length $N_0+N_1 = 1,010,000$, consisting of uniformly-distributed random numbers between 0 and 1.
4. Consider each of the K-stop and (K from L)-stop heuristics in turn. For each one, set $\mathbf{x}_0 := (1,(0,0,...,0))$ as the initial state and use the first N_0 random numbers in Z to simulate events in the first N_0 time steps (this is done by sampling from the transition probability distribution described in (2.2)). This is the 'warm-up period'. Let $\mathbf{y}$ denote the state reached at the end of the warm-up period. Then, use the remaining N_1 random numbers in Z to simulate events during the next N_1 time steps, with the system beginning in state $\mathbf{y}$, and use the statistics collected during these time steps to quantify the heuristic's performance.

The method for simulating the DVO heuristic is different, because (as discussed in Section 2.3) this heuristic is not directly compatible with the MDP formulation in the paper. To simulate the DVO heuristic, we simulate in continuous time as follows:

1. Use the same set of randomly-generated system parameters used for the other heuristics.
2. Let $N_0 = 10,000$ and $N_1 = 1,000,000$ (as for the other heuristics). Set $T_0 := N_0\Delta$ and $T_1 := N_1\Delta$ (where Δ is defined in (2.1)) as the warm-up period length and the main simulation length, respectively.
3. Set $\mathbf{x}_0 := (1,(0,0,...,0))$ as the initial state. Note that, under the DVO heuristic, the set of decision epochs is not the same as under the other heuristics (see Section 2.3.1). At each decision epoch we make a decision using the rules of the DVO heuristic and then simulate the time until the next decision epoch, which requires simulating from an exponential distribution (if the server processes a job or idles at an empty demand point) or an Erlang distribution (if the server switches to another demand point). Costs are accumulated based on the job counts x_i at the various demand points in between decision epochs. This process continues until the total time elapsed exceeds $N_0\Delta$, at which point the warm-up period ends. Let $\mathbf{y}$ denote the state reached at the end of the warm-up period. We then carry out

the main simulation in the same way, starting from state $\mathbf{y}$ and continuing for a further $N_1\Delta$ time units, at which point the process ends and the performance of the DVO heuristic is estimated by dividing the total costs incurred during the main simulation by the total time elapsed.

A.7 Method for testing the feasibility of dynamic programming (DP) for the numerical experiments in Section 2.4

As explained in Section 2.4, we aim to use DP (specifically, relative value iteration) to compute the optimal average cost g^* whenever it is computationally feasible to do so. For a particular problem instance, we carry out the following steps in order to classify it as either 'feasible' or 'infeasible' and (for the feasible instances only) estimate the optimal average cost:

1. If $d \geq 4$, classify this instance as infeasible.
2. Otherwise (if $d \leq 3$), carry out the following steps:
   (a) Set $m = 10$, $t = 0$ and $g^* = 0$.
   (b) Consider a finite-state MDP in which the number of jobs present at any demand point is not allowed to exceed m, as described in the proof of Theorem 2.1. Let M denote the size of the state space, given by

$$M := (d + n)(m + 1)^d.$$

   (Recall that d and n are the numbers of demand points and intermediate stages in the network, respectively.) If $M \geq 1,000,000$, go to step (d). Otherwise, solve the finite-state MDP using DP, let t denote the time taken (in seconds) and let g^* denote the optimal average cost.
   (c) If $t < 600$, increase m by 10 and return to step (b). Otherwise, continue to step (d).

(d) If either (i) $g^* = 0$, or (ii) the latest value of g^* exceeds the previous value by more than ϵ (where we set $\epsilon = 0.001$), then classify this instance as infeasible. Otherwise, classify it as feasible and let the latest value of g^* be an approximation for the optimal average cost in the infinite-state MDP.

Appendix B

Appendices for Chapter 3

B.1 Proof of Proposition 3.1.

We prove the result by showing that our finite-state MDP belongs to the *communicating* class of MDP models, as defined in Puterman (1994) (p. 348). To this end, it will be sufficient to prove the lemma below.

Lemma B.1. *For any two states $\mathbf{x}, \mathbf{x}' \in S$ there exists a stationary policy θ such that $\mathbf{x}'$ is accessible from $\mathbf{x}$ in the Markov chain induced by θ.*

The lemma is proved as follows. Let $\mathbf{x} = (i, (x_1, ..., x_m))$ and $\mathbf{x}' = (i', (x'_1, ..., x'_m))$ be two given states in S. Let θ be a stationary policy that chooses an action under state $\mathbf{x}$ as follows:

- If $x_1 = x_2 = ... = x_m = 0$ then we identify the shortest path from i to i' and attempt to take the first step along that path. (If $i = i'$ then we just remain at i.)
- If $x_j > 0$ for at least one machine $j \in M$ then we identify the 'most-damaged' machine, i.e. the machine j with $x_j \geq x_l$ for all $l \in M$. In case of ties, we assume that j is selected according to some pre-determined priority ordering of the machines. If the repairer is already at node j (i.e. $j = i$), then it remains there. Otherwise, the repairer identifies the shortest path from its current location i to node j, and attempts to take the first step along that path.

For clarity, the 'shortest path' between two nodes $i, i' \in V$ means the shortest possible sequence of connected nodes from i to i'. In case two or more paths are tied for the shortest distance, the path can be chosen according to some fixed priority ordering of the nodes. Also, note that the rules above are implemented dynamically at each time step, so it may happen (for example) that the repairer begins moving towards a particular machine j with $x_j \geq x_l$ for all $l \in M$ but, during the switching process, another machine r degrades to the point that $x_j < x_r$ and therefore the repairer changes course and moves towards node r instead.

To prove the proposition, it suffices to show that there is a possible sequence of random events that causes the system to reach state $\mathbf{x}'$ when the policy θ described above is followed. In the first case ($x_1 = x_2 = ... = x_m = 0$), it is clear that the state $(i', (0, 0, ..., 0))$ can be reached if none of the machines degrade before the repairer arrives at node i'. If the system is in state $(i', (0, 0, ..., 0))$ and one of the machines (say machine 1) degrades, then the system transitions to state $(i', (1, 0, ..., 0))$ and according to the rules above, the repairer should attempt to move to the first node on the shortest path from i' to 1. However, with probability $1 - \tau$, the switch 'fails' and the repairer remains at i'. Furthermore, it is possible for other machines to deteriorate (possibly multiple times) before the switch eventually succeeds. (It may also happen that $i' = 1$, in which case the repairer remains at machine 1 and attempts to repair it, but degradations happen while the repair is in progress.) Using this reasoning, we can see that the state $\mathbf{x}' = (i', x'_1, ..., x'_m))$ (where each x'_j is an arbitrary value in $\{0, ..., K_j\}$) is accessible from $(i, (0, 0, ...0))$ via an appropriate sequence of random degradation events.

In the second case ($x_j > 0$ for at least one machine $j \in M$), suppose that no further degradations occur until the repairer has restored all of the machines to their 'pristine' states. This fortuitous sequence of events allows the system to eventually transition to state $(j, (0, 0, ..., 0))$, where in this case $j \in M$ denotes the last machine to be fully repaired. We can then simply repeat the arguments given for the previous case, replacing i with j, in order to show that state $\mathbf{x}' = (i', (x'_1, ..., x'_m))$ can eventually be reached. This completes the proof of the lemma, and the result follows from the theory in Puterman (1994) (see Theorem 8.3.2, p. 351). □

B.2 Proof of Proposition 3.2.

In this proof we consider an arbitrary stationary policy, θ. Since θ is not necessarily optimal, it may be the case that the average cost $g_\theta(\mathbf{x})$ depends on the initial state $\mathbf{x}$. Let P_θ denote the transition matrix under θ, consisting of the one-step transition probabilities $p_{\mathbf{x},\mathbf{y}}(\theta(\mathbf{x}))$ defined in (3.1), where $\theta(\mathbf{x})$ is the action chosen at state $\mathbf{x}$ under θ. Since the state space S is finite and P_θ is a stochastic matrix, it follows from Proposition 8.1.1 in Puterman (1994) (p. 333) that

$$g_\theta(\mathbf{x}) = \sum_{\mathbf{y}\in S} \pi_\theta(\mathbf{x},\mathbf{y})c(\mathbf{y}),$$

where $\pi_\theta(\mathbf{x},\mathbf{y}) = \lim_{k\to\infty} \mathbb{P}_\theta(\mathbf{x}_k = \mathbf{y} \mid \mathbf{x}_0 = \mathbf{x})$ denotes the limiting probability of moving from state $\mathbf{x}$ to state $\mathbf{y}$ in k transitions under policy θ, and this probability can be obtained as the entry in the row corresponding to state $\mathbf{x}$ and the column corresponding to state $\mathbf{y}$ in the matrix $P_\theta^* := \lim_{k\to\infty}(P_\theta)^k$. It follows that equation (3.7) is equivalent to

$$\sum_{\mathbf{y}\in S} \pi_\theta(\mathbf{x},\mathbf{y})c(\mathbf{y}) = \sum_{\mathbf{y}\in S} \pi_\theta(\mathbf{x},\mathbf{y})\tilde{c}(\mathbf{y},\theta(\mathbf{y})) \quad \forall \mathbf{x}\in S. \tag{B.1}$$

Let the initial state $\mathbf{x}$ be fixed. For each machine $j \in M$ and $k \in \{1, 2, ..., K_j\}$, let $G_{j,k}$ denote the set of states $\mathbf{y}$ under which the repairer is at node j, $y_j = k$ and the action chosen under θ is to remain at node j. By considering the costs incurred at each of the individual machines, it may be seen that equation (B.1) can be expressed as

$$\sum_{j\in M}\sum_{\mathbf{y}\in S} \pi_\theta(\mathbf{x},\mathbf{y})f_j(y_j) = \sum_{j\in M} f_j(K_j) - \sum_{j\in M}\sum_{k=1}^{K_j}\sum_{\mathbf{y}\in G_{j,k}} \pi_\theta(\mathbf{x},\mathbf{y})\frac{\mu_j}{\lambda_j}[f_j(K_j) - f_j(k-1)].$$

It will therefore be sufficient to show that, for an arbitrary fixed $j \in M$, we have

$$\sum_{\mathbf{y}\in S} \pi_\theta(\mathbf{x},\mathbf{y})f_j(y_j) = f_j(K_j) - \sum_{k=1}^{K_j}\sum_{\mathbf{y}\in G_{j,k}} \pi_\theta(\mathbf{x},\mathbf{y})\frac{\mu_j}{\lambda_j}[f_j(K_j) - f_j(k-1)]. \tag{B.2}$$

Using the detailed balance equations for ergodic Markov chains (see, for example, Cinlar (1975)), the rate at which machine j transitions from state y_j to y_{j+1} under θ must equal the rate at which it transitions from y_{j+1} to y_j. This implies that

$$\lambda_j \sum_{\mathbf{y}\in S: y_j = k} \pi_\theta(\mathbf{x},\mathbf{y}) = \mu_j \sum_{\mathbf{y}\in G_{j,k+1}} \pi_\theta(\mathbf{x},\mathbf{y}) \qquad \forall k \in \{0, 1, ..., K_j - 1\}.$$

Therefore the right-hand side of (B.2) can be written equivalently as

$$f_j(K_j) - \sum_{k=0}^{K_j-1} \sum_{\mathbf{y}\in S: y_j=k} \pi_\theta(\mathbf{x},\mathbf{y})[f_j(K_j) - f_j(k)]. \tag{B.3}$$

Then, using the fact that

$$\sum_{k=0}^{K_j} \sum_{\mathbf{y}\in S: y_j=k} \pi_\theta(\mathbf{x},\mathbf{y}) = 1,$$

we can write (B.3) in the form

$$\begin{aligned}
&\sum_{k=0}^{K_j} \sum_{\mathbf{y}\in S: y_j=k} \pi_\theta(\mathbf{x},\mathbf{y}) f_j(K_j) - \sum_{k=0}^{K_j-1} \sum_{\mathbf{y}\in S: y_j=k} \pi_\theta(\mathbf{x},\mathbf{y})[f_j(K_j) - f_j(k)] \\
&= \sum_{\mathbf{y}\in S: y_j=K_j} \pi_\theta(\mathbf{x},\mathbf{y}) f_j(K_j) + \sum_{k=0}^{K_j-1} \sum_{\mathbf{y}\in S: y_j=k} \pi_\theta(\mathbf{x},\mathbf{y}) f_j(k) \\
&= \sum_{\mathbf{y}\in S} \pi_\theta(\mathbf{x},\mathbf{y}) f_j(y_j)
\end{aligned}$$

which equals the left-hand side of (B.2) as required. This confirms that the average costs for machine j are equal under both cost formulations, and since this argument can be repeated for each machine $j \in M$ and initial state $\mathbf{x} \in S$, the proof is complete. □

B.3 Proof of Proposition 3.6

We assume that (i) there are m machines arranged in a complete graph with no intermediate stages, i.e., $V = M$; (ii) for all $j \in M$, $K_j = 1$; (iii) all m machines have identical parameter settings. Under these conditions, the state space has the form

$$S = \{(i, (x_1, ..., x_m)) \mid i \in M,\ x_j \in \{0, 1\} \text{ for } j \in M\}.$$

Machine $i \in M$ is working if $x_i = 0$, and has failed if $x_i = 1$. Throughout this proof we omit the subscripts on the degradation rates λ_j, the repair rates μ_j and the cost functions f_j, since the third assumption implies that these are the same for all $j \in M$. Also, given that each machine has only two possible conditions, we will depart from our usual notational convention and use k as a time step index rather than an indicator of a machine's condition.

The general strategy of this proof is to determine the decisions made by the index policy θ^{IND} under these conditions, and then show that these coincide with the decisions made by an optimal policy. First, suppose that the system is operating under the index policy. We can consider three possible cases: (a) the repairer is at a node i with $x_i = 1$; (b) the repairer is at a node i with $x_i = 0$ and there is at least one node $j \neq i$ with $x_j = 1$; (c) the repairer is at a node i with $x_i = 0$ and we also have $x_j = 0$ for all $j \neq i$. In case (a), the repairer has a choice between remaining at the failed machine i or switching to an alternative machine. Recall that the index policy makes decisions using the indices $\Phi_i^{\text{stay}}(x_i)$, $\Phi_{ij}^{\text{move}}(x_j)$ and $\Phi_{ij}^{\text{wait}}(x_j)$ defined in (3.12), (3.13) and (3.14) respectively. Given the assumption that $K_j = 1$ for all $j \in S$, the index for remaining at node i simplifies to

$$\Phi_i^{\text{stay}}(x_i) = \Phi_i^{\text{stay}}(1) = \frac{\mathbb{E}[R_i(1)]}{\mathbb{E}[T_i(1)]} = \frac{\mu}{\lambda} f(1), \tag{B.4}$$

which is simply equal to the reward rate earned while the machine is being repaired. Next, consider the index for moving to an alternative node $j \neq i$. According to the rules of the heuristic, the repairer should only consider switching to nodes belonging to the set J defined in (3.16). Thus, we need to check whether $\Phi_{ij}^{\text{move}}(x_j) \geq \Phi_{ij}^{\text{wait}}(x_j)$ for each $j \neq i$. We can express these indices as follows:

$$\Phi_{ij}^{\text{move}}(x_j) = \begin{cases} \mathbb{P}(X_j = 1)\,\dfrac{\mathbb{E}[R_j(1)]}{\mathbb{E}[D_{ij}|X_j = 1] + \mathbb{E}[T_j(1)]} = \dfrac{\mu\tau f(1)}{\tau^2 + (2\mu + \lambda)\tau + \mu\lambda}, & \text{if } x_j = 0, \\ \dfrac{\mathbb{E}[R_j(1)]}{\mathbb{E}[D_{ij}] + \mathbb{E}[T_j(1)]} = \left(\dfrac{\tau}{\mu + \tau}\right)\dfrac{\mu}{\lambda} f(1), & \text{if } x_j = 1, \end{cases} \tag{B.5}$$

$$\Phi_{ij}^{\text{wait}}(x_j) = \frac{\mathbb{E}[R_j(1)]}{1/\lambda + \mathbb{E}[D_{ij}] + \mathbb{E}[T_j(1)]} = \frac{\mu\tau f(1)}{(\mu + \lambda)\tau + \mu\lambda}. \tag{B.6}$$

By comparing the expressions in (B.5) and (B.6) we can see that $\Phi_{ij}^{\text{move}}(x_j) < \Phi_{ij}^{\text{wait}}(x_j)$ if $x_j = 0$, but $\Phi_{ij}^{\text{move}}(x_j) > \Phi_{ij}^{\text{wait}}(x_j)$ if $x_j = 1$. Therefore the repairer only considers switching to node j if $x_j = 1$. However, even in this case $\Phi_{ij}^{\text{move}}(x_j)$ is smaller than the index $\Phi_i^{\text{stay}}(x_i)$ in (B.4) (noting that $R_i(1) \overset{\text{dist}}{=} R_j(1)$ and $T_i(1) \overset{\text{dist}}{=} T_j(1)$ due to homogeneous parameters), so the repairer always prefers to remain at node i if $x_i = 1$, regardless of the x_j values at other nodes.

Next, consider case (b), in which $x_i = 0$ and there is a node $j \neq i$ with $x_j = 1$. In this case, from (3.12) we have $\Phi_i^{\text{stay}}(x_i) = 0$, while from (B.5) and (B.6) we can see that $\Phi_{ij}^{\text{move}}(x_j)$ is strictly positive and greater than $\Phi_{ij}^{\text{wait}}(x_j)$. Therefore the repairer chooses to switch to move j in this case.

Finally, consider case (c), in which $x_j = 0$ for all $j \in M$. In this case the repairer chooses to move to the 'optimal idling position', i^* (or remain if it is already there). Under the stated assumptions, the quantity $\Psi(i)$ in (3.15) is the same for all $i \in M$, so i^* can be chosen arbitrarily as any of the nodes in M. This is a rather trivial case and we will show later in the proof that any action by the repairer is optimal in this situation.

In the next stage of the proof we will show that there exists an optimal policy that makes the same decisions as the index policy. First, we introduce some terminology. Consider two distinct states $\mathbf{x} = (i, (x_1, ..., x_m))$ and $\mathbf{x}' = (i', (x'_1, ..., x'_m))$. We say that states $\mathbf{x}$ and $\mathbf{x}'$ are *symmetric* if $x_i = x'_{i'}$ and $\sum_{j \in M} x_j = \sum_{j \in M} x'_j$. That is, the total number of failed machines is the same under both states, and the repairer is either at a working machine under both states or at a failed machine under both states. For example, in a problem with 3 machines, we say that states $(1, (1, 0, 1))$ and $(2, (0, 1, 1))$ are symmetric, but the states $(1, (0, 0, 1))$ and $(2, (0, 1, 0))$ are not symmetric (because the repairer is at a working machine in the first case, but a failed machine in the second case).

Consider a finite-horizon version of the discrete-time MDP formulated in Section 3.2 (following uniformization), and let $w_k(\mathbf{x})$ denote the minimum possible expected total cost that can be achieved over k time steps, given that the initial state is $\mathbf{x} \in S$. In order to make progress we will need to use the following three properties of the function w_k, which we label (WN1)-(WN3) for convenience:

1. (WN1) If states $\mathbf{x}$ and $\mathbf{x}'$ are symmetric, then $w_k(\mathbf{x}) = w_k(\mathbf{x}')$ for all $k \in \mathbb{N}$.
2. (WN2) If states $\mathbf{x} = (i, (x_1, ..., x_m))$ and $\mathbf{x}' = (i', (x'_1, ..., x'_m))$ are identical except that $x_j = 0$ and $x'_j = 1$ for some $j \in M$, then $w_k(\mathbf{x}) \leq w_k(\mathbf{x}')$ for all $k \in \mathbb{N}$.
3. (WN3) Suppose states $\mathbf{x} = (i, (x_1, ..., x_m))$ and $\mathbf{x}' = (i', (x'_1, ..., x'_m))$ are identical except that $i \neq i'$. If $x_i = 1$ and $x_{i'} = 0$, then $w_k(\mathbf{x}) \leq w_k(\mathbf{x}')$ for all $k \in \mathbb{N}$.

Property (WN1) states that the optimal expected total cost remains the same if the initial state $\mathbf{x}$ is replaced by one of its symmetric states. Indeed, if $\mathbf{x}$ and $\mathbf{x}'$ are symmetric

then $w_k(\mathbf{x})$ and $w_k(\mathbf{x}')$ must be identical, since the decision-maker is faced with the same problem in both cases (one may observe that the states $\mathbf{x}$ and $\mathbf{x}'$ are identical up to a renumbering of the nodes in V). Property (WN2) states that it is better for machine j to be working than failed (if all else is equal), which is obvious because a failed machine will cause an extra cost to be incurred. Property (WN3) states that it is better for the repairer to be located at a failed machine than a working machine (if all else is equal), which makes sense because being at a failed machine allows the repairer to commence repairs without any delay. All three of these properties can be proved using induction on k. We omit the induction proofs for brevity.

Next, we consider the action that should be chosen in order to minimize the expected total cost over $(k+1)$ time steps, starting from state $\mathbf{x} = (i, (x_1, ..., x_m))$. In general, we have

$$w_{k+1}(\mathbf{x}) = \min_{j \in M} \{q_{k+1}(\mathbf{x}, j)\}, \tag{B.7}$$

where $q_{k+1}(\mathbf{x}, j)$ is the expected total cost for choosing action $j \in M$ and then following an optimal policy for the remaining k time steps. The expression for $q_{k+1}(\mathbf{x}, j)$ takes different forms depending on whether the action j is to remain at the current node i or to switch to an alternative node. In the former case, it is also useful to distinguish between the cases $x_i = 0$ and $x_i = 1$. Let $M_0(\mathbf{x})$ and $M_1(\mathbf{x})$ be the total numbers of machines that are working and failed under state $\mathbf{x}$, respectively. Then, using the transition probabilities in (3.1), we have

$$q_{k+1}(\mathbf{x}, j) = \begin{cases} M_1(\mathbf{x})f(1) + M_0(\mathbf{x})\lambda w_k(\mathbf{x}^{j+}) \\ + \left(1 - M_0(\mathbf{x})\lambda\right) w_k(\mathbf{x}), & \text{if } j = i \text{ and } x_i = 0, \\ M_1(\mathbf{x})f(1) + M_0(\mathbf{x})\lambda w_k(\mathbf{x}^{j+}) + \mu w_k(\mathbf{x}^{j-}) \\ + \left(1 - M_0(\mathbf{x})\lambda - \mu\right) w_k(\mathbf{x}), & \text{if } j = i \text{ and } x_i = 1, \\ M_1(\mathbf{x})f(1) + M_0(\mathbf{x})\lambda w_k(\mathbf{x}^{j+}) + \tau w_k(\mathbf{x}^{\to j}) \\ + \left(1 - M_0(\mathbf{x})\lambda - \tau\right) w_k(\mathbf{x}), & \text{if } j \neq i, \end{cases} \tag{B.8}$$

where $\mathbf{x}^{j+}$ denotes a state identical to $\mathbf{x}$ except that the component x_j is changed from 0 to 1, $\mathbf{x}^{j-}$ denotes a state identical to $\mathbf{x}$ except that the component x_j is changed from

1 to 0, and $\mathbf{x}^{\to j}$ denotes a state identical to $\mathbf{x}$ except that the repairer's location is changed to j.

Next, we consider the same three cases (a)-(c) discussed earlier and, for each case, show that the action chosen by the index policy (identified earlier) is also the action that should be chosen in order to minimize the finite-horizon expected total cost. Case (a) is where the repairer is at a machine i with $x_i = 1$. In order to show that remaining at machine i minimizes the expected total cost over $k+1$ stages, we need to show that $q_{k+1}(\mathbf{x}, i) \leq q_{k+1}(\mathbf{x}, j)$ for all $j \neq i$. Using the expressions in (B.8), it can be checked that

$$q_{k+1}(\mathbf{x}, i) - q_{k+1}(\mathbf{x}, j) = \mu(w_k(\mathbf{x}^{j-}) - w_k(\mathbf{x})) - \tau(w_k(\mathbf{x}^{\to j}) - w_k(\mathbf{x})). \tag{B.9}$$

By Property (WN2) we have $w_k(\mathbf{x}^{j-}) - w_k(\mathbf{x}) \leq 0$. If j is a node with $x_j = 1$ then x and $x^{\to j}$ are symmetric states, so it follows from (WN1) that $w_k(\mathbf{x}^{\to j}) - w_k(\mathbf{x}) = 0$. On the other hand, if j is a node with $x_j = 0$ then by (WN3) we have $w_k(\mathbf{x}^{\to j}) - w_k(\mathbf{x}) \geq 0$. It follows that the expression in (B.9) is non-positive in all cases, so an optimal policy will choose to remain at node i if $x_i = 1$. Next, consider case (b), where the repairer is at a node i with $x_i = 0$ and there is at least one node j with $x_j = 1$. In this case, from (B.8) we obtain

$$q_{k+1}(\mathbf{x}, i) - q_{k+1}(\mathbf{x}, j) = -\tau(w_k(\mathbf{x}^{\to j}) - w_k(\mathbf{x})), \tag{B.10}$$

and by (WN3) we have $w_k(\mathbf{x}^{\to j}) \leq w_k(\mathbf{x})$, so the expression in (B.10) is non-negative. Hence, an optimal policy will choose to switch to node j in this scenario. Finally, consider case (c), where the repairer is at a node i with $x_i = 0$ and we also have $x_j = 0$ for all $j \neq i$. The expression in (B.10) also applies to this case, but this time (WN1) implies that $w_k(\mathbf{x}^{\to j}) - w_k(\mathbf{x})$ is equal to zero because states $\mathbf{x}$ and $\mathbf{x}^{\to j}$ are symmetric. Essentially, this is a trivial case in which any action is optimal, including the action taken by the index policy.

In summary, we have shown that the actions chosen by the index policy in the various possible cases (a)-(c) coincide with the actions chosen by an optimal policy in a finite-horizon problem in which the objective is to minimize total cost over k stages (for arbitrary $k \in \mathbb{N}$). If we define $h(\mathbf{x}) := \lim_{k\to\infty}(w_k(\mathbf{x}) - w_k(\mathbf{z}))$, where $\mathbf{z}$ is a fixed reference state in S, then it follows from standard theory (see Puterman (1994), p. 373) that the function h satisfies the optimality equations (3.4). Hence, by our preceding arguments, the actions chosen by the index policy always attain the minimum on the

right-hand side in (3.4), and the index policy is indeed optimal. □

B.4 Proof of Proposition 3.8

Let V be a star network as defined in Definition 3.7, with radius $r \geq 1$. The other assumptions of the proposition are that $K_j = 1$ for all $j \in M$, all m machines have identical parameter settings, and $\tau > 2r\lambda$. The state space has the form

$$S = \{(i, (x_1, ..., x_m)) \mid i \in V,\ x_j \in \{0, 1\} \text{ for } j \in M\}.$$

Machine $i \in M$ is working if $x_i = 0$, and has failed if $x_i = 1$. As in the proof of Proposition 3.6 we will omit the subscripts on the degradation rates λ_j, repair rates μ_j and cost functions f_j, since these are assumed to be the same for all $j \in M$. Also, given that each machine has only two possible conditions, we will depart from our usual notational convention and use k as a machine or time step index rather than an indicator of a machine's condition.

The proof of Proposition 3.6 used dynamic programming arguments to show that the decisions made by the index policy are optimal, but this approach becomes more complicated in a network with intermediate stages, so instead we will use a slightly different approach. We will show that the decisions made by the index policy are *greedy*, in the sense that they always minimize the expected amount of time until the repairer is next repairing a machine, and then show that a greedy policy is optimal under the assumptions of the proposition.

The first step is to determine the decisions made under the index policy. In the arguments that follow, it will be useful to define $d(i, j)$ as the length of a shortest path between two nodes $i, j \in V$. We can consider five possible cases: (a) the repairer is at a machine $i \in M$ with $x_i = 1$; (b) the repairer is at a machine $i \in M$ with $x_i = 0$ and there is at least one machine $j \neq i$ with $x_j = 1$; (c) the repairer is at a machine $i \in M$ with $x_i = 0$ and we also have $x_j = 0$ for all $j \neq i$; (d) The repairer is at an intermediate stage $i \in N$ and there is at least one machine $j \in M$ with $x_j = 1$; (e) the repairer is at an intermediate stage $i \in N$ and we have $x_j = 0$ for all $j \in M$.

In case (a), the repairer has a choice between remaining at the failed machine i or switching to an alternative machine. Recall that the index policy makes decisions using the

indices $\Phi_i^{\text{stay}}(x_i)$, $\Phi_{ij}^{\text{move}}(x_j)$ and $\Phi_{ij}^{\text{wait}}(x_j)$ defined in (3.12), (3.13) and (3.14) respectively. As in the proof of Proposition 3.6, the index for remaining at node i simplifies to

$$\Phi_i^{\text{stay}}(x_i) = \Phi_i^{\text{stay}}(1) = \frac{\mathbb{E}[R_i(1)]}{\mathbb{E}[T_i(1)]} = \frac{\mu}{\lambda} f(1). \tag{B.11}$$

Next, consider the index for moving to an alternative node $j \neq i$. According to the rules of the heuristic, the repairer should only consider switching to nodes belonging to the set J defined in (3.16). The indices $\Phi_{ij}^{\text{move}}(x_j)$ and $\Phi_{ij}^{\text{wait}}(x_j)$ for $j \neq i$ can be expressed as

$$\Phi_{ij}^{\text{move}}(x_j) = \begin{cases} \mathbb{P}(X_j = 1) \, \dfrac{\mathbb{E}[R_j(1)]}{\mathbb{E}[D_{ij}|X_j = 1] + \mathbb{E}[T_j(1)]}, & \text{if } x_j = 0, \\ \dfrac{\mathbb{E}[R_j(1)]}{\mathbb{E}[D_{ij}] + \mathbb{E}[T_j(1)]}, & \text{if } x_j = 1, \end{cases} \tag{B.12}$$

$$\Phi_{ij}^{\text{wait}}(x_j) = \frac{\mathbb{E}[R_j(1)]}{1/\lambda + \mathbb{E}[D_{ij}] + \mathbb{E}[T_j(1)]}. \tag{B.13}$$

It can be seen that if $x_i = 1$ then $\Phi_i^{\text{stay}}(x_i) > \Phi_{ij}^{\text{move}}(x_j)$, regardless of whether or not node j is included in the set J. Hence, the repairer will remain at node i in this case. In case (b), where the repairer is at a node i with $x_i = 0$ and we also have $x_j = 1$ for some $j \neq i$, it can be seen from the expressions above that $\Phi_{ij}^{\text{move}}(x_j) > \Phi_{ij}^{\text{wait}}(x_j)$, so machine j is included in the set J and furthermore the index for remaining at node i is zero, so the repairer moves towards node j in this case (this implies taking a step towards the center of the star network).

In case (c), where the repairer is at a machine i with $i = 0$ and we also have $x_j = 0$ for all $j \neq i$, it is necessary to identify the node i^* that minimizes the quantity $\Psi(i)$ defined in (3.15). Given the layout of the star network (see Figure 3.2), it is easy to see that $\Psi(i)$ is minimized by setting $i = s$, where s is the center node. Hence, in this case (as in the previous case), the repairer should move from machine i towards the center.

In case (d), where the repairer is at an intermediate stage i and there exists $j \in M$ with $x_j = 1$, the rules of the heuristic imply that the repairer needs to identify the node j that minimizes $\Phi_{ij}^{\text{move}}(x_j)$. Let j be a fixed node in V such that $x_j = 1$. If $1 \leq d(i,j) \leq r$ (implying that the repairer is somewhere between the center node and j), then it is clear that $\mathbb{E}[D_{ij}] \leq \mathbb{E}[D_{ik}]$ for any machine $k \neq j$ and therefore from (B.12) we have $\Phi_{ij}^{\text{move}}(x_j) \geq \Phi_{ik}^{\text{move}}(x_k)$. Therefore the repairer should move towards node j in this situation. On the other hand, suppose we have $1 \leq d(i,k) < r$ for some machine k such

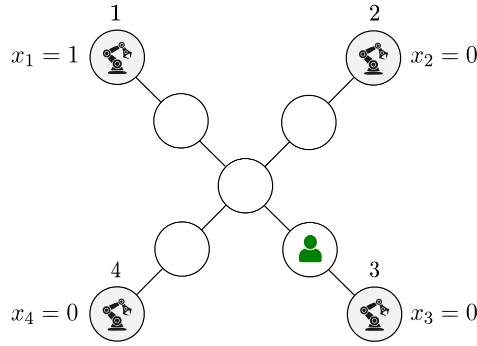

FIGURE B.1: A star network with 4 machines and a radius $r = 2$, where the repairer is at an intermediate stage, machine 1 has failed with $x_1 = 1$ and other machines are working with $x_2 = x_3 = x_4 = 0$.

that $x_k = 0$ (see Figure B.1). In this case, the decision is less clear. The repairer could move towards the nearest machine k, in the hope that a degradation event will occur at that machine before the repairer arrives. However, we will show that the decision under the index policy is to move towards the damaged machine j, which is further away. In order to show that $\Phi_{ij}^{\text{move}}(1) \geq \Phi_{ik}^{\text{move}}(0)$, we begin by establishing some useful bounds. Firstly, we use the general fact that $1 - x^p \leq k(1-x)$ for $k \in (0,1)$ and $p \in \mathbb{N}$ to show that

$$\mathbb{P}(X_k = 1) = 1 - \left(\frac{\tau}{\tau+\lambda}\right)^{d(i,k)} \leq \frac{d(i,k)\lambda}{\tau+\lambda}. \tag{B.14}$$

Notably, this bound becomes tighter as $\tau/(\tau+\lambda)$ approaches one, which is relevant to our case because we assume $\tau > 2r\lambda$. Secondly, we note that $\mathbb{E}[D_{ik}|X_k = 1]$ is the expected total amount of time for $d(i,k)+1$ events to occur, where each event is either a completed switch or a degradation event at node k, and the latter type of event can only occur once. The expected amount of time for each of these $d(i,k)+1$ events is either $1/(\tau+\lambda)$ or $1/\tau$, depending on whether or not the degradation event has occurred yet. Hence,

$$\mathbb{E}[D_{ik}|X_k = 1] \geq \frac{d(i,k)+1}{\tau+\lambda}. \tag{B.15}$$

Using the bounds in (B.14) and (B.15), we have

$$\begin{aligned}
\Phi_{ik}^{\text{move}}(0) &= \mathbb{P}(X_k = 1)\,\frac{\mathbb{E}[R_k(1)]}{\mathbb{E}[D_{ik}|X_k = 1] + \mathbb{E}[T_k(1)]} \\
&\leq \frac{d(i,k)\lambda}{\tau+\lambda}\cdot\frac{\mathbb{E}[R_k(1)]}{(d(i,k)+1)/(\tau+\lambda) + \mathbb{E}[T_k(1)]} \\
&= \frac{d(i,k)\mu f(1)}{d(i,k)\mu + \lambda + \mu + \tau}.
\end{aligned} \tag{B.16}$$

On the other hand, using $E[D_{ij}] = (2r - d(i,k))/\tau$, the index for switching to machine j is

$$\Phi_{ij}^{\text{move}}(1) = \frac{\mathbb{E}[R_j(1)]}{\mathbb{E}[D_{ij}] + \mathbb{E}[T_j(1)]} = \frac{\tau\mu f(1)}{\lambda(\tau + (2r - d(i,k))\mu)}. \tag{B.17}$$

Using (B.16) and (B.17), we find (after some algebra steps) that $\Phi_{ij}^{\text{move}}(1)$ is greater than or equal to the upper bound for $\Phi_{ik}^{\text{move}}(0)$ provided that

$$\tau^2 + (d(i,k)+1)\mu\tau \geq (d(i,k)-1)\lambda\tau + (2rd(i,k) - d(i,k)^2)\lambda\mu. \tag{B.18}$$

Given that $\tau > 2r\lambda$, it is sufficient to show that (B.18) holds with λ replaced by $\tau/(2r)$. After making this substitution and rearranging, we obtain

$$\frac{2r + d(i,k)^2}{2r}\mu\tau + \frac{2r - (d(i,k)-1)}{2r}\tau^2 \geq 0,$$

which holds because $1 \leq d(i,k) \leq r$. In conclusion, under the index policy the repairer chooses to move towards the damaged machine j rather than the (nearer) undamaged machine k.

Finally, consider case (e), where the repairer is at an intermediate stage and $x_j = 0$ for all $j \in M$. As noted in case (c), the quantity $\Psi(i)$ defined in (3.15) is minimized when $i = s$, so the repairer moves towards the center in this situation.

In the next part of the proof, we consider cases (a)-(e) again and show that, in each case, the repairer's action under the index policy also minimizes the expected amount of time until repair. For clarity, we define the 'time until repair' as the amount of time until the repairer chooses to remain at a damaged machine, and abbreviate it as TUR. In case (a), the TUR is trivially equal to zero under the index policy since the repairer remains at the damaged machine i.

In case (b), the repairer is at a machine i with $x_i = 0$ and there exists $j \neq i$ with $x_j = 1$. Suppose (for a contradiction) that the TUR is minimized by remaining at machine i. If machine i is still undamaged at the next time step then, logically, the TUR is still minimized by remaining at node i, since the repairer is still in the same situation. (It may happen that a degradation event occurs at another machine $k \notin \{i,j\}$, but this does not affect the minimization of the TUR, since there is no reason for the repairer to prefer machine k to machine j). Continuing this argument, it follows that the TUR is minimized by remaining at node i until a degradation event occurs there. However, the expected time until this happens is $1/\lambda$, while the expected amount of time to move directly to machine j is $2r/\tau$, and we have $2r/\tau < 1/\lambda$ by assumption. Therefore, in

fact, the repairer should not remain at machine i and should take the only possible alternative action, which is to move towards the center of the network. Note that we have not shown, at this stage, that the repairer should move to machine j without any interruption in order to minimize the TUR; we have only shown that if the repairer is at machine i then it should take the first step towards the center. The actions chosen at intermediate stages will be discussed in case (d).

Next, we address cases (c) and (e) at the same time. In both of these cases we have $x_j = 0$ for all $j \in M$. Let i denote the repairer's current location, which could be either a machine or an intermediate stage. We note that the quantity $\Psi(i)$ in (3.15) is the expected amount of time for the repairer to travel directly towards the next machine that degrades. It can easily be checked that $\Psi(i)$ increases with $d(i, s)$; that is, the further away the repairer gets from the center, the larger $\Psi(i)$ becomes. It follows that in order to minimize the TUR the repairer should travel towards the center node s. The index policy also directs the repairer to move to the center when all nodes are working, so again we find that it minimizes the TUR.

The only remaining case is (d), where the repairer is at an intermediate stage i and there is at least one node j with $x_j = 1$. If $d(i, j) \leq r$ then it is obvious that the TUR is minimized by moving to machine j without any interruption, which the index policy does. Thus, in order to avoid triviality, we should consider again the situation shown in Figure B.1, where the repairer's nearest node k has $x_k = 0$. In this case we can use an inductive argument to show that the TUR is minimized by moving towards the center. Initially, suppose $d(i, k) = 1$, so the repairer is only one step from machine k. The TUR cannot be minimized by remaining at the intermediate stage i, since (using similar reasoning to case (b)) this would imply that the repairer should remain at node i until a degradation event occurs at machine k, in which case the TUR is at least $1/\lambda$, which is greater than $(2r - 1)/\tau$ (the TUR given by moving directly to machine j). On the other hand, the TUR cannot be minimized by moving to machine k either, since we already showed in case (b) that if the repairer is at a machine k with $x_k = 0$ then it should move towards the center in order to minimize the TUR; therefore, moving from node i to machine k would result in back-and-forth movements between nodes i and k until a degradation occurs at machine k, and (again) the TUR would be at least $1/\lambda$. Hence, by process of elimination over the action set, we conclude that if $d(i, k) = 1$ then the TUR is minimized by moving towards the center. An identical argument can be used when $d(i, k) = 2$, using the fact that moving towards machine k results in back-and-forth behavior between two intermediate stages and a TUR of at least $1/\lambda$. Continuing

this argument, we conclude that the repairer should move towards the center and then continue towards machine j (or another damaged machine) in order to minimize the TUR.

In summary, we have shown that in all of the cases (a)-(e) the action chosen by the index policy also minimizes the expected time until repair (TUR). The final stage of the proof is to show that a 'greedy' policy that always minimizes the TUR is also optimal. In general, we can think of the evolution of the system as a sequence of time periods, where some of these periods are spent repairing a machine and some are spent performing other actions (such as moving between nodes in the network). Let these time periods be written as follows:

$$T_1, S_1, T_2, S_2, ..., T_k, S_k, ...$$

where each quantity S_k (for $k \in \mathbb{N}$) is an amount of time spent repairing a machine, and each quantity T_k is an amount of time spent performing other actions. For example, the repairer might begin at a machine i with $x_i = 0$, but then move towards a damaged machine j and repair it until it is fixed, in which case T_1 is the amount of time spent moving towards machine j and S_1 is the time spent performing the repair. The repairer earns a reward of $(\mu/\lambda)f(1)$ during each of the S_k periods, and earns no reward during the T_k periods. Thus, in order to maximize long-run average reward, the ratio $\sum_k S_k/(\sum_k(S_k+T_k))$ over a long time horizon should be as large as possible. It is clear from this argument that the repairer should never interrupt a repair in progress, since doing so would imply that (possibly) fewer degradation events would occur at other machines during the repair period, and therefore the repairer would not only earn a smaller reward from the current machine (due to interrupting the repair) but would also have fewer opportunities to earn rewards at other machines. We deduce that in order to maximize long-run average reward, the repairer should remain at a damaged machine if it is currently located there.

We will show that in order to maximize long-run average reward, the repairer should always greedily minimize the time until the next repair. This is not necessarily obvious, because if the repairer allows a particular non-repair time T_k to increase then this might enable the next non-repair time T_{k+1} to become smaller due to more degradation events occurring during the kth non-repair period and more opportunities for earning rewards becoming available.

Let the system be initialized in an arbitrary state and let j denote the first machine repaired by the index policy, given a particular sample path (i.e. a particular sequence

of random events). Hence, T_1 is the time until the repairer begins repairing machine j (this may be zero if the repairer is initially located at j). By our previous arguments, $\mathbb{E}[T_1]$ is minimized under the index policy, while $\mathbb{E}[S_1]$ is maximized (because the repair is not interrupted). When the repairer finishes repairing machine j, there are two possibilities: either (i) all machines are undamaged and the repairer has to wait until the next degradation event occurs, or (ii) there is at least one damaged machine, $j \neq i$, in the system. This idea is shown in Figure B.2, in which the second non-repair period has been divided into subperiods, A_2 and B_2. We use A_2 to denote the waiting time until there is at least one damaged machine in the system (which is zero in the case that there is already a damaged machine when the first repair finishes), and $B_2 = T_2 - A_2$ to denote the additional time until the second repair period begins. In Figure B.2 we also use t^{deg} to denote the time that the next degradation event occurs, given that there are no damaged machines at time $T_1 + S_1$.

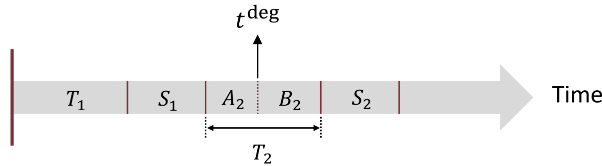

FIGURE B.2: Division of time into periods $T_1, S_1, T_2, S_2, ...$ under the index heuristic.

Given that we assume repairs are non-interrupted, $\mathbb{E}[S_1]$ is fixed and minimizing $\mathbb{E}[T_1]$ corresponds to maximizing $\mathbb{E}[A_2]$. If $A_2 > 0$ then the actions taken by the repairer during the time period $(T_1+S_1, T_1+S_1+A_2)$ ensure that the minimum TUR starting from time $T_1+S_1+A_2$ (denoted by B_2) is itself minimized. Intuitively, because the repairer moves towards the center of the network while it is waiting for the next degradation event to occur, it is in the best possible position to begin the second repair as early as possible. This ensures that the expected amount of non-repair time before starting the second repair, $\mathbb{E}[T_1 + T_2]$, is minimized. We emphasize the subtle point that the index policy does not necessarily minimize $\mathbb{E}[T_2]$ (because T_2 includes a possible 'waiting period' A_2), but the time spent in the waiting period works to the advantage of the index policy and ensures that the repairer spends as much time as possible moving towards the center, from which it follows that $\mathbb{E}[B_2]$ and $\mathbb{E}[T_1 + T_2]$ are both minimized.

This argument can be continued in an inductive manner. In general we can replace T_1 with $\sum_{k=1}^{K} T_k$ in Figure B.2, where $K \geq 1$, and use the inductive assumption that the index policy minimizes $\mathbb{E}[\sum_{k=1}^{K} T_k]$. Then the same arguments as above can be used to show that $\mathbb{E}[\sum_{k=1}^{K+1} T_k]$ is also minimized. Given that the S_k values are independent and

identically distributed (as noted above), it follows that the index policy maximizes the amount of time spent repairing over any finite time horizon, and therefore maximizes the expected total reward over a finite horizon. Therefore it must also maximize the long-run average reward. □

B.5 Details of Approximate Policy Improvement

In this appendix we provide further details and pseudocode for the approximate policy improvement (API) method described in Section 3.4. The method consists of an offline part, in which we acquire estimates of the value function $h_{\theta^{\text{M-IND}}}(\mathbf{x})$ under the modified index policy (referred to as the base policy) $\theta^{\text{M-IND}}$, and an online part, in which we apply an improvement step to the base policy (with the base policy also providing an insurance option in cases where an improving action cannot be identified with sufficient confidence). Both parts involve sampling trajectories of possible future events starting from a given state, and we present the pseudocode for this procedure as a separate module, called 'SampleTrajectory'. We also present the offline part as two separate modules (a preparatory phase and a main phase), and the online part as a single module. In summary, the API method is presented here as 4 separate modules:

- SampleTrajectory (called upon by other modules)
- Offline part - preparatory phase
- Offline part - main phase
- Online part

We briefly explain the steps of these modules in subsections B.5.1-B.5.4, and provide full pseudocode for all 4 modules in B.5.5.

To simplify notation, we will denote the modified index policy throughout this appendix by $\theta \equiv \theta^{\text{M-IND}}$. The offline and online parts use a dynamic array U to store and update estimated average cost values $\hat{h}_\theta(\mathbf{x})$ for states $\mathbf{x} \in S$. This array also stores other statistics, including a sum of squared costs $\widehat{SS}_\theta(\mathbf{x})$, sum of squared weights $\widehat{W}_\theta(\mathbf{x})$, and observation count $\hat{s}_\theta(\mathbf{x})$ for each $\mathbf{x} \in U$. For readability, we drop the hats and θ-subscripts in both the discussion and accompanying pseudocode, referring to these

statistics as h-values, SS-values, W-values and s-values.

B.5.1 SampleTrajectory module

The SampleTrajectory module requires certain inputs from the other modules, including a policy for selecting actions θ, an estimate for the average cost g_θ, an initial state $\mathbf{z}$, a reference state $\mathbf{u}_0$, an array U with the latest gathered statistics (as described above) and an integer p that determines how much information is recorded from each simulated trajectory. It generates a sample trajectory starting from $\mathbf{z}$ under policy θ, and uses this to update the value estimates $h(\mathbf{x})$ and other statistics for newly encountered states $\mathbf{x}$.

Trajectories are sampled by simulating the system evolution over a finite time, recording the cumulative cost C and the total number of elapsed time steps T, until a stopping condition is satisfied. This condition is met if the simulated trajectory reaches either (i) a designated reference state $\mathbf{u}_0$, or (ii) a state already stored in the value estimate array U, excluding the initial state $\mathbf{z}$. At this point, the simulation terminates and the stopping state is labeled as $\mathbf{u}$.

During the simulation, each newly encountered state $\mathbf{x}$ (i.e. one not previously visited in the same trajectory) is recorded in a `visited` list, along with the cumulative cost $C_\mathbf{x}$ and time step $T_\mathbf{x}$ when $\mathbf{x}$ was first visited. Here, $C_\mathbf{x}$ represents the total accumulated cost from the start of the trajectory up to the first visit to $\mathbf{x}$, and $T_\mathbf{x}$ denotes the number of steps taken to reach $\mathbf{x}$ from the initial state.

Once the simulated trajectory ends, the algorithm processes the first p visited states. For each such state $\mathbf{x}$, if it is not already included in U, a new entry $U(\mathbf{x})$ is initialized as $(h(\mathbf{x}), SS(\mathbf{x}), W(\mathbf{x}), s(\mathbf{x})) = (0, 0, 0, 0)$. Otherwise, the current values are extracted directly from U for update. Each update is computed using the bootstrapped estimate

$$h^{(s)}(\mathbf{x}) \leftarrow (C - C_\mathbf{x}) + h(\mathbf{u}) - g_\theta \cdot (T - T_\mathbf{x}),$$

where $s = s(\mathbf{x})$ is the observation count for state $\mathbf{x}$ and $h(\mathbf{u})$ denotes the latest available estimate for state $\mathbf{u}$, computed using the values $h^{(r)}(\mathbf{u})$ for $r = 1, ..., s(\mathbf{u})$. This corresponds to equation (3.19) (please see the accompanying explanation in that section). We then update the estimate $h(\mathbf{x})$ using the rule

$$h(\mathbf{x}) \leftarrow (1 - \alpha^{(s)}(\mathbf{x}))h(\mathbf{x}) + \alpha^{(s)}(\mathbf{x})h^{(s)}(\mathbf{x}), \tag{B.19}$$

where $\alpha^{(s)}(\mathbf{x})$ is a learning parameter, given by

$$\alpha^{(s)}(\mathbf{x}) = \frac{10}{10 + s - 1}.$$

We note that (B.19) is a standard exponential averaging rule used in reinforcement learning algorithms, which effectively states that the updated estimate $h(\mathbf{x})$ is computed as a weighted average of the old estimate and the newly-acquired observation (from the latest simulation). The second moment $SS(\mathbf{x})$, and sum of squared weights $W(\mathbf{x})$ are updated using a rule analogous to (B.19), and $U(\mathbf{x})$ is updated accordingly. The module returns the updated array U, the stopping state $\mathbf{u}$, and the total amount of time elapsed during the simulation, τ_δ.

B.5.2 Offline part - preparatory phase

In the preparatory phase we estimate the long-run average cost g_θ and construct a representative set of system states. This module is divided into three stages:

1. We begin by generating a core set of states Z^{core}, consisting of one state $\mathbf{z}_i$ for each machine $i \in M$. Specifically, for each $i \in M$, we simulate events starting from state $(i, (0, ..., 0))$ under the base policy θ and identify the state visited most frequently during this simulation, which we denote by $\mathbf{z}_i$.

2. Subsequently, a separate long-run simulation from initial state $(1, (0, ..., 0))$ is used to estimate the average cost g_θ. During this simulation, we track how often the repairer visits the machines in M. The machine j^* with the highest visitation count is identified, and its associated core state $\mathbf{z}_{j^*}$ is designated as the reference state $\mathbf{u}_0$. This reference state then becomes the initial member of the array U mentioned previously.

3. Next, we create a larger set $Z \supseteq Z^{\text{core}}$ to improve coverage of the state space. For each machine i we add the core state $\mathbf{z}_i \in Z^{\text{core}}$ to Z and also add all states of the form $\mathbf{z}_i^{\to j}$ for all nodes j adjacent to i in the network, where $\mathbf{z}_i^{\to j}$ is identical to $\mathbf{z}_i$ except that the repairer's location is changed to j. Also, if $\mathbf{z}_i$ is a state with i^{th} component greater than or equal to one, we include the state $\mathbf{z}_i^{i-}$ in Z, where $\mathbf{z}_i^{i-}$ is identical to $\mathbf{z}_i$ except that the ith component is reduced by one. Including states of the form $\mathbf{z}_i^{\to j}$ and $\mathbf{z}_i^{i-}$ in Z is beneficial for the online part that follows later.

B.5.3 Offline part - main phase

This module takes the set Z from the preparatory phase as an input. It is divided into two stages:

1. We consider each of the states $\mathbf{z} \in Z$ in turn. For each $\mathbf{z} \in Z$ we repeatedly sample trajectories of future events starting from $\mathbf{z}$ by using the SampleTrajectory module, and we update the array U with the statistics (h-values, SS-values, W-values, and s-values) gathered from these trajectories. We also set $p = 1$ within the SampleTrajectory module in this stage. This process continues (returning to the initial state $\mathbf{z}$ at the end of each trajectory) until either R^{off} trajectories have been simulated (where R^{off} is a large integer), or the cumulative simulation time exceeds a large threshold $\tau_{\max}$. At this point, we consider the next state in Z, or (if all states in Z have been considered) move on to the next stage.
2. We consider each of the core states $\mathbf{z} \in Z^{\text{core}} \subseteq Z$ in turn. For each state $\mathbf{z} \in Z^{\text{core}}$ we sample a trajectory of future events starting from $\mathbf{z}$ and update the array U with the statistics gathered, in the same way as in the previous stage. However, when the trajectory ends (which happens when the system reaches a state $\mathbf{u} \in U$), we use $\mathbf{u}$ as the initial state for the next trajectory, rather than returning to state $\mathbf{z}$. This creates a chain of successive trajectories, with the ending point of one trajectory acting as the starting point for the next. We also set $p = 5$ within the SampleTrajectory module in this stage. As in the previous stage, this process continues until either R^{off} trajectories have been simulated or the cumulative simulation time exceeds the threshold $\tau_{\max}$, at which point we consider the next state in Z^{core}, or (if all states in Z^{core} have been considered) end the offline part.

B.5.4 Online part

The online part is based on the paradigm of a 'real' system that we are able to observe and control in a continuous manner over time. The aim is to apply an improvement step to the base policy θ, making use of the statistical information gained from simulating the system evolving under θ in the offline part. In the absence of being able to observe a 'real' system, we conduct a simulation of the system, and perform 'nested' simulations within this in order to gather extra information about nearby states, as described in Section 3.4.

At each iteration, the current system state $\mathbf{x}$ is observed and a local neighborhood $F(\mathbf{x})$ is constructed. This neighborhood includes (i) the current state $\mathbf{x}$ itself, (ii) states reachable by relocating the repairer to adjacent nodes, and (iii) a state resulting from repair completion if the repairer is currently at machine i with $x_i \geq 1$. We then use the empirical data stored in U to obtain, for each $\mathbf{y} \in F(\mathbf{x})$, a confidence interval for $h_\theta(\mathbf{y})$. The confidence interval is given by

$$[h^-(\mathbf{y}), h^+(\mathbf{y})] = \left[h(\mathbf{y}) \pm z_{0.025} \sqrt{\left(\frac{SS(\mathbf{y}) - h(\mathbf{y})^2}{1 - W(\mathbf{y})} \right) W(\mathbf{y})} \right], \tag{B.20}$$

where $z_{0.025} \approx 1.96$ is a Normal distribution percentile. The formula in (B.20) is derived from the theory of weighted sample variances and reliability weights (cf. Sec. 21.7 of Galassi et al. (2011)). In the following lines, we provide some additional details of how the formula is derived. Let $h(\mathbf{y})$ be the h-value estimate for state $\mathbf{y}$ after $s(\mathbf{y})$ observations have been collected. It can be checked using (B.19) that

$$h(\mathbf{y}) = \sum_{r=1}^{s(\mathbf{y})} w^{(r)}(\mathbf{y})\, h^{(r)}(\mathbf{y}),$$

where the weights $w^{(r)}(\mathbf{y})$ for $r = 1, ..., s(\mathbf{y})$ satisfy

$$w^{(r)}(\mathbf{y}) = \alpha^{(r)}(\mathbf{y}) \prod_{j=r+1}^{s(\mathbf{y})} \Big(1 - \alpha^{(j)}(\mathbf{y})\Big).$$

We can compute the sample variance as

$$\begin{aligned} \hat{\sigma}(\mathbf{y})^2 &= \frac{\sum_{r=1}^{s(\mathbf{y})} w^{(r)}(\mathbf{y}) \big(h^{(r)}(\mathbf{y}) - h(\mathbf{y})\big)^2}{\sum_{r=1}^{s(\mathbf{y})} w^{(r)}(\mathbf{y})} \\ &= \sum_{r=1}^{s(\mathbf{y})} w^{(r)}(\mathbf{y}) \cdot h^{(r)}(\mathbf{y})^2 - h(\mathbf{y})^2 \\ &= SS(\mathbf{y}) - h(\mathbf{y})^2. \end{aligned}$$

Then, to obtain an unbiased estimate $\sigma(\mathbf{y})$, we use

$$\sigma(\mathbf{y})^2 = \frac{\hat{\sigma}(\mathbf{y})^2}{1 - (V_2/V_1^2)},$$

where

$$V_1 = \sum_{r=1}^{s(\mathbf{y})} w^{(r)}(\mathbf{y}), \qquad V_2 = \sum_{r=1}^{s(\mathbf{y})} w^{(r)}(\mathbf{y})^2.$$

As noted above, $V_1 = 1$ and we also have $V_2 = W(\mathbf{y})$. Hence,

$$\sigma(\mathbf{y})^2 = \frac{SS(\mathbf{y}) - h(\mathbf{y})^2}{1 - W(\mathbf{y})}.$$

Thus, the effective variance of the weighted sample mean is

$$\frac{\sigma(\mathbf{y})^2}{1/\sum_{r=1}^{s(\mathbf{y})} w^{(r)}(\mathbf{y})^2} = \left(\frac{SS(\mathbf{y}) - h(\mathbf{y})^2}{1 - W(\mathbf{y})}\right) W(\mathbf{y}),$$

which justifies (B.20). After computing the confidence intervals, the algorithm selects an action $a^* \in A_{\mathbf{x}}$ by, firstly, checking to see whether there is an action a^* satisfying the inequality

$$\sum_{\mathbf{y} \in S} p_{\mathbf{x},\mathbf{y}}(a^*)\, h(\mathbf{y}) < \sum_{\mathbf{y} \in S} p_{\mathbf{x},\mathbf{y}}(a')\, h(\mathbf{y})$$

for all alternative actions $a' \neq a^*$ and for all possible values of $h(\mathbf{y})$ (for $\mathbf{y} \in F(\mathbf{x})$) within their respective confidence intervals $[h^-(\mathbf{y}), h^+(\mathbf{y})]$. If no such a^* exists, we resort to the base policy as a 'safety measure' and set $a^* = \theta(\mathbf{x})$.

To continuously improve the accuracy of the value estimates, the SampleTrajectory module is called at every iteration. On each iteration, it is allowed to run for a maximum $\delta = 0.01$ seconds and the statistics in U are updated during this period. The motivation for this step is that in the 'real' system, there would be a certain amount of time (δ) in between any pair of state transitions, and therefore we can make use of this time to improve value function estimates for states that are near the current state. Finally, each iteration ends with a real-time transition from the current state $\mathbf{x}$ to a successor state based on the chosen action a^* (note that this is the action given by the 'improving' step, not the action given by the base policy). After a large number of iterations, R^{on}, have been completed, the online part ends and the average cost over all time steps is returned as a performance metric for the API policy.

B.5.5 Pseudocode of API

Please find pseudocode for the modules used in approximate policy improvement on the following pages.

Algorithm B.1 SampleTrajectory module

1: **Input:** State $\mathbf{z}$; reference state $\mathbf{u}_0$; array U; recording length p; policy θ; average cost g_θ
2: **Initialize:**
3: Total cost incurred $C \leftarrow 0$; total time steps $T \leftarrow 0$
4: Current state $\mathbf{z}_{\text{curr}} \leftarrow \mathbf{z}$; visited list `visited` $\leftarrow [(\mathbf{z}, 0, 0)]$
5: Start time: $t_{\text{start}} \leftarrow$ current time
6: **while** true **do**
7: $\quad C \leftarrow C + c(\mathbf{z}_{\text{curr}})$; $T \leftarrow T + 1$
8: $\quad$ Simulate transition: $\mathbf{z}_{\text{curr}} \xrightarrow{\theta(\mathbf{z}_{\text{curr}})} \mathbf{x}$
9: $\quad$ **if** ($\mathbf{x} \neq \mathbf{z}$ **or** $\mathbf{x} = \mathbf{u}_0$) **and** $\mathbf{x} \in U$ **then**
10: $\quad\quad$ Set stopping state $\mathbf{u} \leftarrow \mathbf{x}$; **break**
11: $\quad$ **else**
12: $\quad\quad \mathbf{z}_{\text{curr}} \leftarrow \mathbf{x}$
13: $\quad\quad$ **if** $\mathbf{x} \notin$ `visited` **then**
14: $\quad\quad\quad$ Append $(\mathbf{x}, C, T)$ to `visited`
15: $\quad\quad$ **end if**
16: $\quad$ **end if**
17: **end while**
18: **for** each $(\mathbf{x}, C_{\mathbf{x}}, T_{\mathbf{x}})$ in `visited` $[:p]$ **do** $\quad \triangleright$ first p visited during simulation
19: $\quad$ **if** $\mathbf{x} \notin U$ **then**
20: $\quad\quad$ Initialize: $U(\mathbf{x}) \leftarrow (0, 0, 0, 0)$
21: $\quad$ **end if**
22: $\quad$ Unpack: $(h(\mathbf{x}), SS(\mathbf{x}), W(\mathbf{x}), s(\mathbf{x})) \leftarrow U(\mathbf{x})$; $(h(\mathbf{u}), SS(\mathbf{u}), W(\mathbf{u}), s(\mathbf{u})) \leftarrow U(\mathbf{u})$
23: $\quad s \leftarrow s(\mathbf{x}) + 1$; $s(\mathbf{x}) \leftarrow s$
24: $\quad$ Compute learning parameter: $\alpha^{(s)}(\mathbf{x}) \leftarrow \frac{10}{10+s-1}$
25: $\quad$ Compute estimate: $h^{(s)}(\mathbf{x}) \leftarrow (C - C_{\mathbf{x}}) + h(\mathbf{u}) - g_\theta \cdot (T - T_{\mathbf{x}})$
26: $\quad$ Update:

$$h(\mathbf{x}) \leftarrow (1 - \alpha^{(s)}(\mathbf{x})) \cdot h(\mathbf{x}) + \alpha^{(s)}(\mathbf{x}) \cdot h^{(s)}(\mathbf{x})$$
$$SS(\mathbf{x}) \leftarrow (1 - \alpha^{(s)}(\mathbf{x})) \cdot SS(\mathbf{x}) + \alpha^{(s)}(\mathbf{x}) \cdot (h^{(s)}(\mathbf{x}))^2$$
$$W(\mathbf{x}) \leftarrow (1 - \alpha^{(s)}(\mathbf{x}))^2 \cdot W(\mathbf{x}) + (\alpha^{(s)}(\mathbf{x}))^2$$

27: $\quad U(\mathbf{x}) \leftarrow (h(\mathbf{x}), SS(\mathbf{x}), W(\mathbf{x}), s(\mathbf{x}))$
28: **end for**
29: End time: $t_{\text{end}} \leftarrow$ current time; simulation time $\tau_\delta \leftarrow t_{\text{end}} - t_{\text{start}}$
30: **Output:** Array U; stopping state $\mathbf{u}$; simulation time τ_δ

Note: For each state $\mathbf{x}$, $s(\mathbf{x})$ is the number of estimates so far. The current estimate $h^{(s)}(\mathbf{x})$ and learning parameter $\alpha^{(s)}(\mathbf{x})$ correspond to the s-th update.

Algorithm B.2 Offline part - preparatory phase

1: **Input:** Base policy θ; simulation lengths $R^{(1)}$, $R^{(2)}$
2: **Stage 1: Construct core state set Z^{core}**
3: Initialize $Z^{\text{core}} \leftarrow \emptyset$
4: **for** each machine $i \in M$ **do**
5: Initialize $r \leftarrow 0$, $\mathbf{x} \leftarrow (i, (0, ..., 0))$, $S_i \leftarrow \emptyset$; set $N_i(\mathbf{x}) \leftarrow 0$ upon adding $\mathbf{x}$ to S_i
6: **while** $r < R^{(1)}$ **do**
7: Simulate transition: $\mathbf{x} \xrightarrow{\theta(\mathbf{x})} \mathbf{y}$
8: $\mathbf{x} = (j, (x_1, ..., x_m)) \leftarrow \mathbf{y}$; $r \leftarrow r + 1$
9: **if** $j = i$ **then**
10: $S_i \leftarrow S_i \cup \{\mathbf{x}\}$; $N_i(\mathbf{x}) \leftarrow N_i(\mathbf{x}) + 1$
11: **end if**
12: **end while**
13: $\mathbf{z}_i \leftarrow \arg\max_{\mathbf{x} \in S_i} N_i(\mathbf{x})$; $Z^{\text{core}} \leftarrow Z^{\text{core}} \cup \{\mathbf{z}_i\}$
14: **end for**
15: **Stage 2: Estimate average cost g_θ and choose reference state $\mathbf{u}_0$**
16: Initialize $C \leftarrow 0$, $r \leftarrow 0$, $\mathbf{x} \leftarrow (1, (0, ..., 0))$, and counters $n_i \leftarrow 0$ for all $i \in M$
17: **while** $r < R^{(2)}$ **do**
18: $C \leftarrow C + c(\mathbf{x})$
19: Simulate transition: $\mathbf{x} \xrightarrow{\theta(\mathbf{x})} \mathbf{y}$
20: $\mathbf{x} = (i, (x_1, ..., x_m)) \leftarrow \mathbf{y}$; $r \leftarrow r + 1$
21: **if** $i \in M$ **then**
22: $n_i \leftarrow n_i + 1$
23: **end if**
24: **end while**
25: $g_\theta \leftarrow C/R^{(2)}$; $j^* \leftarrow \arg\max_i n_i$; $\mathbf{u}_0 \leftarrow \mathbf{z}_{j^*}$; move $\mathbf{z}_{j^*}$ to the front of Z^{core}
26: **Stage 3: Construct representative state set Z**
27: Initialize $Z \leftarrow \emptyset$
28: **for** each $\mathbf{z}_i = (i, (z_1, ..., z_m)) \in Z^{\text{core}}$ **do**
29: $Z \leftarrow Z \cup \{\mathbf{z}_i\} \cup \left\{\mathbf{z}_i^{\to j} : j \text{ adjacent to } i\right\}$
30: **if** $z_i \geq 1$ **then**
31: $Z \leftarrow Z \cup \{\mathbf{z}_i^{i-}\}$
32: **end if**
33: **end for**
34: **Output:** Average cost g_θ; reference state $\mathbf{u}_0$; core state set Z^{core}; representative state set Z

Algorithm B.3 Offline part - main phase

1: **Input:** Average cost g_θ; reference state $\mathbf{u}_0$; core state set Z^{core}; representative state set Z; simulation length R^{off}; time limit $\tau_{\max}$
2: Initialize array $U \leftarrow \emptyset$; set $U(\mathbf{u}_0) \leftarrow (0, 0, 1, 1)$
3: **Stage 1: Value estimation for representative states**
4: **for** each $\mathbf{z} \in Z$ **do**
5: Initialize $r \leftarrow 0$; $\tau \leftarrow 0$; recording length $p \leftarrow 1$
6: **while** $r < R^{\text{off}}$ **and** $\tau < \tau_{\max}$ **do**
7: Run **SampleTrajectory**$(\mathbf{z}) \to U$, stopping state $\mathbf{u}$, time τ_δ ▷ Ignore $\mathbf{u}$
8: $r \leftarrow r + 1$; $\tau \leftarrow \tau + \tau_\delta$
9: **end while**
10: **end for**
11: **Stage 2: System evolution from core states**
12: **for** each $\mathbf{z} \in Z^{\text{core}}$ **do**
13: Initialize $r \leftarrow 0$; $\tau \leftarrow 0$; recording length $p \leftarrow 5$
14: **while** $r < R^{\text{off}}$ **and** $\tau < \tau_{\max}$ **do**
15: Run **SampleTrajectory**$(\mathbf{z}) \to U$, stopping state $\mathbf{u}$, time τ_δ
16: $\mathbf{z} \leftarrow \mathbf{u}$; $r \leftarrow r + 1$; $\tau \leftarrow \tau + \tau_\delta$
17: **end while**
18: **end for**
19: **Output:** Array U

Algorithm B.4 Online part

1: **Input:** Average cost g_θ; reference state $\mathbf{u}_0$; array U; simulation length R^{on}; δ-length time limit
2: Initialize $C \leftarrow 0$, $r \leftarrow 0$, $\mathbf{x} \leftarrow \mathbf{u}_0$
3: **while** $r < R^{\text{on}}$ **do**
4: $C \leftarrow C + c(\mathbf{x})$
5: **Stage 1: Action selection**
6: Define local neighborhood $F(\mathbf{x})$
7: **for** each $\mathbf{y} \in F(\mathbf{x})$ **do**
8: Compute confidence interval $[h^-(\mathbf{y}), h^+(\mathbf{y})]$ from U
9: **end for**
10: Select $a^* \in A_\mathbf{x}$ satisfying:

$$\sum_{\mathbf{y}\in S} p_{\mathbf{x},\mathbf{y}}(a^*)\, h(\mathbf{y}) < \sum_{\mathbf{y}\in S} p_{\mathbf{x},\mathbf{y}}(a')\, h(\mathbf{y}), \quad \forall a' \neq a^*, \quad \forall h(\mathbf{y}) \in [h^-(\mathbf{y}), h^+(\mathbf{y})].$$

11: **if** no such a^* exists **then**
12: Set $a^* \leftarrow \theta(\mathbf{x})$
13: **end if**
14: **Stage 2: Information supplement**
15: Initialize $\tau \leftarrow 0$; recording length $p \leftarrow 1$
16: **while** $\tau < \delta$ **do**
17: Simulate transition: $\mathbf{x} \xrightarrow{a^*} \mathbf{x}'$
18: Define local neighborhood $F(\mathbf{x}')$
19: **for** each $\mathbf{y} \in F(\mathbf{x}')$ **do**
20: Run **SampleTrajectory**$(\mathbf{y}) \rightarrow U$, $\mathbf{u}$, τ_δ
21: $\tau \leftarrow \tau + \tau_\delta$
22: **end for**
23: **end while**
24: **Stage 3: Real-time transition**
25: Simulate transition: $\mathbf{x} \xrightarrow{a^*} \mathbf{x}_{\text{new}}$
26: $\mathbf{x} \leftarrow \mathbf{x}_{\text{new}}$; $r \leftarrow r + 1$
27: **end while**
28: $g \leftarrow C/R^{\text{on}}$
29: **Output:** Average cost g

Note: Confidence intervals are computed using the method of reliability weights (Galassi et al. (2011), Sec. 21.7)

B.6 Methods for generating the parameters for the numerical experiments in Section 3.5

For each of the 1,000 instances considered in Section 3.5, the system parameters are randomly generated as follows:

- The type of cost function is sampled unbiasedly from the set $\{1, 2, 3\}$, where:
  - Type 1 represents a linear cost function, i.e., the cost incurred per unit time for machine $i \in M$ in state x_i is given by $f_i(x_i) = c_i x_i$;
  - Type 2 represents a quadratic cost function, i.e., the cost incurred per unit time for machine $i \in M$ in state x_i is given by $f_i(x_i) = c_i x_i^2$;
  - Type 3 represents a piecewise linear cost function, i.e., the cost incurred per unit time for machine $i \in M$ in state x_i is given by $f_i(x_i) = c_i \left(x_i + 10 \cdot \mathbb{I}(x_i = K_i)\right)$, where $\mathbb{I}(x_i = K_i)$ is the indicator function that equals 1 if $x_i = K_i$, and 0 otherwise.
- The 'failed' state, K, is sampled unbiasedly from the set $\{1, 2, 3, 4, 5\}$.
- The number of machines, m, is sampled unbiasedly from the set $\{2, 3, 4, 5, 6, 7, 8\}$.
- The a-coordinate and b-coordinate of each machine $i \in M$, denoted as (a_i, b_i), are independently sampled unbiasedly from the set $\{1, 2, 3, 4, 5\}$, with resampling used if two or more machines share the same pair of coordinates.
- The machines in M are re-numbered according to the a-coordinate first and then (in case of ties) by the b coordinate.
- The overall traffic intensity, ρ, is sampled from a continuous uniform distribution between 0.1 and 1.5. Subsequently, the degradation rates λ_i and repair rates μ_i for $i \in M$ are generated as follows:
  - Each repair rate μ_i is initially sampled from a continuous uniform distribution between 0.1 and 0.9.
  - For each machine $i \in M$, an initial value for the degradation rate λ_i' is sampled from a continuous uniform distribution between $0.1\mu_i$ and μ_i.

  - For each machine $i \in M$, the actual traffic intensity ρ_i is obtained by re-scaling the initial traffic intensity λ_i'/μ_i, as follows:

$$\rho_i := \frac{\lambda_i'/\mu_i}{\sum_{i\in M} \lambda_i'/\mu_i}\rho.$$

    This ensures that $\sum_{i\in M} \rho_i = \rho$.

  - For each machine $i \in M$, the actual degradation rate λ_i is obtained as follows:

$$\lambda_i := \rho_i \mu_i.$$

  - All of the λ_i and μ_i values are rounded to 2 significant figures (also causing a small change to the value of ρ).

- For each machine $i \in M$, the cost rate, c_i, is sampled from a continuous uniform distribution between 0.1 and 0.9.

- In order to generate the switching rate τ, we first define $\eta := \tau/(\sum_{i\in M} \lambda_i)$ and generate the value of η as follows:

  - Sample a value p from a continuous uniform distribution between 0 and 1.
  - If $p < 0.5$, sample η from a continuous uniform distribution between 0.1 and 1.
  - If $p \geq 0.5$, sample η from a continuous uniform distribution between 1 and 10.

  We then define $\tau := \eta \sum_{i\in M} \lambda_i$.

B.7 Simulation methods for the numerical experiments in Section 3.5

In each of the 1,000 problem instances, the performances of the index policy and the policies given by the polling heuristic are estimated by simulating the discrete-time evolution of the uniformized MDP described in Section 3.2. We also use a 'common random numbers' method to ensure that machine degradations occur at the same times under each of these policies. More specifically, the simulation steps are as follows:

1. Generate a set of random system parameters as described in Appendix B.6.

2. Set $R^{\text{sim}} := 500,000$ as the number of time steps in the simulation.

3. Generate a list Z of length R^{sim}, consisting of uniformly-distributed random numbers between 0 and 1.

4. Consider each heuristic policy in turn. For each one, we set $\mathbf{x}_0 := (1, (0, 0, ..., 0))$ as the initial state, and then use the random numbers in Z to simulate events during these R^{sim} time steps, with the system beginning in state $\mathbf{x}_0$. The statistics collected during these time steps are used to quantify the heuristic's performance.

When implementing the API policy, we need to decide on the values of $R^{(1)}$, $R^{(2)}$, R^{off}, $\tau_{\max}$ and R^{on} used in the offline and online stages (see the pseudocode in Appendix B.5 for details). We use the values $R^{(1)} = 10,000$, $R^{(2)} = 500,000$, $R^{\text{off}} = 100,000$, $\tau_{\max} = 100$ and $R^{\text{on}} = 500,000$.

Appendix C

Appendices for Chapter 4

C.1 Methods for generating the parameters for the numerical experiments in Section 4.4

To generate the network layouts and parameter values for our numerical experiments, we begin by generating 100 different parameter sets, where each set includes randomly-generated values for the number of demand points d, the number of servers m, the coordinates of the servers (a_i, b_i) for $i \in D$, the cost rates c_i for $i \in D$, the traffic intensity ρ and the switching rate parameter η. For each of these 100 parameter sets we generate 10 different instances by considering each value $\alpha = 0, 0.1, \ldots, 0.9$ in turn, and generating the demand rates λ_i and service rates μ_i in order to preserve the required values of ρ and η. Thus, the total number of instances is $100 \times 10 = 1000$. In each of the 100 parameter sets we generate parameter values as follows:

- The number of demand points, d, is sampled unbiasedly from the set $\{4, 5, 6, 7, 8, 9, 10\}$.
- The number of servers, m, is sampled unbiasedly from the set $\{2, 3, 4\}$, resampling as necessary if $m = d$, as the trivial case of an equal number of servers and demand points is not considered.
- The a-coordinate and b-coordinate of each demand point $i \in D$ denoted as (a_i, b_i), are independently sampled in an unbiased way from the set $\{1, 2, 3, 4, 5\}$, resampling as necessary in order to avoid two demand points having the same pair of coordinates.

- Re-number demand points according to the a-coordinate first and then (in case of ties) by the b coordinate.

- For each demand point $i \in D$, the cost rate, c_i, is sampled from a continuous uniform distribution between 0.1 and 0.9.

- The overall traffic intensity, ρ, is sampled from a continuous uniform distribution between 0.1 and 0.9. Subsequently, the value of η, the arrival rates λ_i and service rates μ_i for $i \in D$ are generated as follows:

  - To generate η, first sample a value p from a continuous uniform distribution between 0 and 1, and then proceed as follows:

    (1) If $p < 0.5$, sample η from a continuous uniform distribution between 0.1 and 1.

    (2) If $p \geq 0.5$, sample η from a continuous uniform distribution between 1 and 10.

  - Each service rate μ_i is sampled from a continuous uniform distribution between 0.1 and 0.9.

  - For each demand point $i \in D$, an initial value for the arrival rate λ'_i is sampled from a continuous uniform distribution between $0.1\mu_i$ and μ_i.

  - For each value of $\alpha \in \{0, 0.1, \ldots, 0.9\}$, do the following:

    (1) For each demand point $i \in D$, the actual traffic intensity ρ_i is obtained by re-scaling the initial traffic intensity λ'_i/μ_i, as follows:

    $$\rho_i := \frac{\lambda'_i/\mu_i}{\sum_{i \in D} \lambda'_i/\mu_i} \rho.$$

    This ensures that $\sum_{i \in D} \rho_i = \rho$.

    (2) For each demand point $i \in D$, the actual arrival rate λ_i is obtained as follows:

    $$\lambda_i := \begin{cases} \rho_i \, m\mu_i, & \text{if } \alpha = 0, \\ \rho_i \, \dfrac{(1 - (1 - \alpha)^m)}{\alpha} \mu_i, & \text{if } \alpha \neq 0. \end{cases}$$

(3) The switching rate τ is generated as:

$$\tau := \eta \sum_{i \in D} \lambda_i.$$

(4) All of the λ_i, μ_i and τ values are rounded to 2 significant figures (also causing a small change to the value of ρ).

C.2 Simulation methods for the numerical experiments in Section 4.4

In each of the 1000 problem instances, we evaluate the performances of the heuristics described in Sections 4.3 and 4.4. Performances are estimated by simulating the discrete-time evolution of the uniformized MDP described in Section 4.2. To ensure consistency across policies, we employ the common random numbers (CRN) method, which synchronizes job arrival times across all simulations within a set of 10 instances. The simulation for each set of 10 instances (defined in Appendix C.1) proceeds as follows:

- Generate a list Z of length $R := 10,000,000$ with uniformly distributed random numbers between 0 and 1.
- Generate a set of random system parameters for each of the 10 instances, as described in Appendix C.1.
- For each instance, we evaluate the performance of each heuristic (polling heuristic, modified 1-stop heuristic, flexible heuristic, and exclusive heuristic) sequentially. Using R random numbers from Z, we simulate events over R time steps and collect statistics to quantify the performance of each heuristic. Since these heuristics follow distinct scheduling principles, as outlined in Sections 4.3 and 4.4, two separate simulation procedures are employed:

I. The procedure for local partitioning methods, including the polling heuristic and modified 1-stop heuristics:

1. Initialization: The system state is represented as $\mathbf{x} = (\mathbf{x}_1, \ldots, \mathbf{x}_m)$, where each server $s \in M$ has a local state vector $\mathbf{x}_s$. The local state vector components are as follows:

   - For the polling heuristic, an additional component $w_s(\mathbf{x})$ is introduced, representing the most recently visited demand point. The state vector of server s is:

   $$\mathbf{x}_s = (v_s(\mathbf{x}), w_s(\mathbf{x}), (x_{s1}, \ldots, x_{sd})),$$

     where:

     – $v_s(\mathbf{x})$ is the current node where server s is located.
     – $w_s(\mathbf{x})$ is the most recently visited demand point.
     – $(x_{s1}, \ldots, x_{sd})$ represents the number of jobs at each of d demand points for server s.

   - For the modified 1-stop heuristic, no component for the most recently visited demand point is needed, so the local state vector simplifies to:

   $$\mathbf{x}_s = (v_s(\mathbf{x}), (x_{s1}, \ldots, x_{sd})).$$

     Similarly, $v_s(\mathbf{x})$ represents the currently-located node, and $(x_{s1}, \ldots, x_{sd})$ represents the number of jobs at each demand point for server s.

   The initialization for both heuristics is as follows:

   (a) The state vector in both heuristics:
      - For the polling heuristic, the server's state vector is initialized as $\mathbf{x}_s = (i_1, i_1, \mathbf{0}_d)$, where both $v_s(\mathbf{x})$ and $w_s(\mathbf{x})$ are initialized to i_1, the first demand point in the server's designated set $\mathcal{D}_s$. The vector $\mathbf{0}_d$ is a zero vector of length d, representing zero jobs at each demand point.
      - For the modified 1-stop heuristic, the server's state vector is initialized as $\mathbf{x}_s = (i_1, \mathbf{0}_d)$, where i_1 and $\mathbf{0}_d$ match the ones in the polling heuristic for the first component and zero jobs for each demand point.

   (b) The corresponding action vector is defined as $\mathbf{a}(\mathbf{x}) = (a_1(\mathbf{x}), \ldots, a_m(\mathbf{x}))$, which specifies the actions to be taken by each server based on the state $\mathbf{x}$ under the heuristic (either the polling or modified 1-stop heuristic).

2. The simulation proceeds iteratively as follows:

   (a) Set the total cost $C = 0$ and the time step $r = 0$.

   (b) Update the total cost:

$$C \leftarrow C + \sum_{s\in M}\sum_{i\in D} c_i \cdot x_{si}.$$

   (c) Generate a random event by drawing $U = Z[r]$ and determine its type:

      i. Arrival event: If an arrival occurs at demand point i, determined by:

$$\Lambda_i = \sum_{k=1}^{i} \lambda_k \Delta, \quad \Lambda_0 = 0,$$

      with

$$\Lambda_{i-1} \le U < \Lambda_i,$$

      then exactly one of the variables $s_{1i}, ..., s_{mi}$ is increased by one, where p_{si} is the probability of x_{si} being increased, for $s = 1, ..., m$. Proceed to step (d).

      ii. Server event: If no arrival occurs, consider server events. Each server $s \in M$ has location $v_s(\mathbf{x})$ and action $a_s(\mathbf{x})$. Define

$$\mathcal{S}_0 := \Lambda_d.$$

      For $s \ge 1$, update the cumulative event probability:

$$\mathcal{S}_s = \begin{cases} \mathcal{S}_{s-1} + (1-\alpha)^{k-1}\mu_i\Delta, & \text{if } v_s(\mathbf{x}) = a_s(\mathbf{x}) = i \in D,\ \sum_{s\in M} x_{si} > 0 \\ & \text{and } k = \sum_{s' \le s,\, s' \in M} \mathbb{I}(v_{s'} = a_{s'} = i), \\ \mathcal{S}_{s-1} + \tau\Delta, & \text{if } v_s(\mathbf{x}) \ne a_s(\mathbf{x}), \\ \mathcal{S}_{s-1}, & \text{otherwise.} \end{cases}$$

      Server event check: If

$$\mathcal{S}_{s-1} \le U < \mathcal{S}_s,$$

      then:

      - If $v_s(\mathbf{x}) = a_s(\mathbf{x}) = i$ and $\sum_{s\in M} x_{si} > 0$, a service completion occurs at i, then:

- If $x_{si} > 0$, reduce x_{si} by one.
- If $x_{si} = 0$, a server j such that $x_{sj} > 0$ is probabilistically selected according to the workload weights, and the corresponding variable x_{sj} is reduced by one.

- If $v_s(\mathbf{x}) \neq a_s(\mathbf{x})$, server s moves to the next node en route to $a_s(\mathbf{x})$.

(d) Update the state $\mathbf{x}$ and action $\mathbf{a}(\mathbf{x})$. Then, increment the time step:

$$r \leftarrow r + 1.$$

(e) If $r < R$, repeat the simulation loop (from 2(b) to 2(e)). Otherwise, compute the average cost:

$$\bar{C} = C/R.$$

II. The procedure for global scheduling methods, including the flexible and exclusive heuristics:

1. Initialization: The system state in either exclusive or flexible heuristic is represented as

$$\mathbf{x} = (\mathbf{v}(\mathbf{x}), (x_1, \ldots, x_d)),$$

where

- $\mathbf{v}(\mathbf{x}) = (v_1(\mathbf{x}), \ldots, v_m(\mathbf{x}))$ represents the locations of all m servers in state $\mathbf{x}$.
- $(x_1, \ldots, x_d)$ represents the number of jobs at each of d demand points.

The initialization for both heuristics is as follows:

(a) The state vector is initialized as $\mathbf{x} = ((1, 2, \ldots, m), \mathbf{0}_d)$, where the servers are located at the demand points from 1 to m, and $\mathbf{0}_d$ is a zero vector of length d.

(b) The corresponding action vector is defined as $\mathbf{a}(\mathbf{x}) = (a_1(\mathbf{x}), \ldots, a_m(\mathbf{x}))$, which specifies the actions to be taken by each server based on the state $\mathbf{x}$ under the heuristic (either the exclusive or flexible heuristic).

2. The simulation proceeds iteratively as follows:

   (a) Set the total cost $C = 0$ and the time step $r = 0$.

   (b) Update the total cost:

   $$C \leftarrow C + \sum_{i \in D} c_i \cdot x_i,$$

   (c) Generate a random event by drawing $U = Z[r]$ and determine its type:

   i. Arrival event: If an arrival occurs at demand point i, determined by:

   $$\Lambda_i = \sum_{k=1}^{i} \lambda_k \Delta, \quad \Lambda_0 = 0,$$

   with

   $$\Lambda_{i-1} \leq U < \Lambda_i,$$

   then increase x_i by one. Proceed to step (d).

   ii. Server event: If no arrival occurs, consider server events. Each server $s \in M$ has location $v_s(\mathbf{x})$ and action $a_s(\mathbf{x})$. Define

   $$\mathcal{S}_0 := \Lambda_d.$$

   For $s \geq 1$, update the cumulative event probability:

   $$\mathcal{S}_s = \begin{cases} \mathcal{S}_{s-1} + (1-\alpha)^{k-1}\mu_i\Delta, & \text{if } v_s(\mathbf{x}) = a_s(\mathbf{x}) = i \in D,\ x_i > 0 \\ & \text{and } k = \sum_{s' \leq s,\, s' \in M} \mathbb{I}(v_{s'} = a_{s'} = i), \\ \mathcal{S}_{s-1} + \tau\Delta, & \text{if } v_s(\mathbf{x}) \neq a_s(\mathbf{x}), \\ \mathcal{S}_{s-1}, & \text{otherwise.} \end{cases}$$

   (Note: in the exclusive heuristic, k is no more than 1.)

   Server event check: If

   $$\mathcal{S}_{s-1} \leq U < \mathcal{S}_s,$$

   then:

   - If $v_s(\mathbf{x}) = a_s(\mathbf{x}) = i$ and $x_i > 0$, then reduce x_i by one.
   - If $v_s(\mathbf{x}) \neq a_s(\mathbf{x})$, server s moves to the next node en route to $a_s(\mathbf{x})$.

(d) Update the state $\mathbf{x}$ and action $\mathbf{a}(\mathbf{x})$. Then, increment the time step:

$$r \leftarrow r + 1.$$

(e) If $r < R$, repeat the simulation loop (from 2(b) to 2(e)). Otherwise, compute the average cost:

$$\bar{C} = C/R.$$