AN APPROXIMATE DYNAMIC PROGRAMMING APPROACH TO THE SCHEDULING OF IMPATIENT JOBS IN A CLEARING SYSTEM

Dong Li, B.Eng.,M.Eng.

Submitted in Part Fulfilment

of the Requirements for the Degree of

Doctor of Philosophy

Department of Management Science

Lancaster University

October 2010

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

Abstract

A single server is faced with a collection of jobs of varying duration and urgency. Before service starts, all jobs are subject to an initial triage, i.e., an assessment of both their urgency and of their service requirement, and are allocated to distinct classes. Jobs in one class have independent and identically distributed lifetimes during which they are available for service. Should a job's lifetime ex[illegible]e before its service begins then it is lost from the system unserved. The goal is to schedule the jobs for service to maximise the expected number served to completion. Two heuristic policies have been proposed in the literature. One works well in a "no loss" limit while the other does so when lifetimes are short. Both can exhibit poor performance for problems at some distance from the regimes for which they were designed. We develop a robustly good heuristic by an approximative approach to the application of a single policy improvement step to the first policy above, in which we use a fluid model to obtain an approximation for its value function. The performance of the proposed heuristic is investigated in an extensive numerical study. This problem is substantially complicated if the initial triage is subject to error. We take a Bayesian approach to this additional uncertainty and discuss the design of heuristic policies to maximise the Bayes' return. We identify problem features for which a high price is paid for poor initial triage and for which improvements in initial job assessment yield significant improvements in service outcomes. An analytical upperbound for the cost of imperfect classification is developed for exponentially distributed lifetime cases. An extensive numerical study is conducted to explore the behaviour of the cost in more general situations.

Acknowledgements

I would like to express my profound gratitude to my supervisor, Professor Kevin Glazebrook, for his invaluable guidance and support over the years. His knowledge, experience, and encouragement are all things I very much appreciated and needed.

My sincere thanks are conveyed to the Department of Management Science in Lancaster University for providing financial support for my research and to staff members of the Department, especially Professor Adam Letchford, Professor Richard Eglese, Dr Dave Worthington and Dr Chris Kirkbride, who have provided their help and contributed in one way or another towards the fulfilment of this work. I would like to extend my thanks to Gay Bentinck and Christine Fletcher who have been always supportive in all the administrative matters and have made my life much easier.

My officemate Kevin Martin deserves a lot of credit for the final results of this work. He has been a valuable resource for helping to work out many technical problems. I am indebted to all of my colleagues and friends in Lancaster, who had helped to make my stay there pleasurable and exciting.

Finally, I would like to thank my family and in-laws for their understanding, support and encouragement over the years. I owe my deepest gratitude to my wife, Wenting, who has taken the main responsibilities to take care of our little daughter so that I can always concentrate on this work. Without her this thesis would not have been possible. Our lovely daughter, Ruohan, has brought us so many joyful moments, even in those very tough days.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

This January when I came back from vacation, I was stranded in London Heathrow airport for two days. Due to the very bad weather, all domestic flights and many international ones were cancelled. The passengers had to wait in massive queues for accommodation vouchers, alternative transportation arrangements, or some other purposes. The amount of passengers was so huge that they significantly outnumbered the staff, resulting in excessive waiting times everywhere. Many people left the queues, taking with them lots of unhappiness and complaints. A challenging question faced by the British Airways management is then how to optimally deploy the very limited resources (staff, aircraft, coaches, hotel rooms, etc.) so as to improve the service level as much as possible.

Similar situations are common in our daily lives. You may experience excessive waiting for service and run out of patience somewhere and sometime; for instance, in hospitals, when ringing call centres, or when checking out at the supermarket. All these situations are characterised by services provided by relatively scarce resources to impatient customers. Key features (eg. processing time, patience) are random, and the management challenge concerns the deployment of these resources over time to optimize some cost criterion. There are some other circumstances, which though very rare, are of crucial importance. One example relevant to the

theme of this thesis is the management of scarce medical resource after mass casualty incidents (MCI), like an earthquake or a terrorist bombing. After such an incident, a significant number of injuries are caused and these may overwhelm existing medical resources immediately. To support efficient resource allocation, all patients are subject to an initial *triage* at the scene, namely an assessment of the severity of their conditions, and are classified into distinct priority groups. Following triage, a central challenge concerns how the patients should be scheduled for treatment such that the total expected number of patients served successfully is maximised. The decisions must deal with the uncertainties associated with the patients' conditions, like the criticality of their condition and the amount of resource they will consume. Poor schedules may lead to preventable deaths. In many proposals, patients are simply treated according to a static order of their priority as determined at the outset. Recent literature on emergency response has argued the importance of developing dynamic scheduling policies (Arnold et al. [2004], Frykberg [2002], and Argon et al. [2008]).

Triage following an MCI must necessarily be undertaken speedily. As Frykberg [2002] has commented: " the more time it may take to find those needing immediate care, ...the greater the likelihood of preventable deaths caused by delay in treatment of the most severely injured." However, it may well be the case that determination of the actual criticality level of a patient in a short timeframe is challenging, and that the initial triage is subject to significant levels of error. Indeed, in a review related to terrorist bombing events, Frykberg and Tepas [1988] found that on average 59% of those classified as critical were actually non-critical, with a 0.05% error rate for the reverse. In medical terminology, these two triage errors are called *overtriage* and *undertriage*, respectively. Recently Turégano-Fuentes et al. [2008] mentioned in their bomb response assessment that " it is difficult to distinguish between casualties requiring immediate and delayed treatment by a rapid examination in the field " and reported inital chaos due to overtriage. It has been shown by Frykberg and Tepas [1988] that the accuracy of triage can have a

significant impact on casualty survival in MCIs.

1.2 Challenges and Objectives

The situation envisaged in the previous section can be thought of as an application of *stochastic scheduling of impatient jobs* in a clearing system. A simple mathematical representation can be developed as follows. Please note that we shall use the terms jobs and customers interchangeably in this thesis to denote the objects seeking service. A collection of jobs is seeking service which is provided by a single server. There are two major sources of randomness related to each job. Firstly, its service is of uncertain duration. Second, the jobs *lifetime*, namely the period of time during which it is available for service, is also uncertain. In most of our models, we assume that a job abandons the system unserved if its service does not begin before the expiration of its lifetime. Further, all jobs whose service begins are guaranteed to be served to completion. No service preemptions are allowed. Each job is subject to an initial triage on its service requirement and its urgency at time zero, and is placed in one of J *classes*. Suppose for the moment that the classification is without error. Jobs in each class j are assumed to have independent and identically distributed (i.i.d.) lifetimes (denoted $X_j \sim F_j$) and i.i.d. service times ($Y_j \sim G_j$). The goal is to optimally schedule service such that the total number of successful service completions achieved until the system is cleared is maximized. Intuitively, it seems clear that priority should be given to jobs with small service times (since these delays others less) and/or small lifetimes (since these are most urgent). Precisely how this should be done to achieve optimality is far from clear, however.

Argon et al. [2008] have shown that when lifetimes and service times are suitably agreeable (jobs with the shortest lifetimes also have the shortest service times) then the optimal policy always gives highest priority to the time-critical job regardless of the system state. Such special results notwithstanding, the central challenge

concerns the development and evaluation of strongly performing *heuristic policies*. The literature contains discussions of two candidate policy classes. Glazebrook et al. [2004] develop a simple static policy (hereafter denoted π^S) which operates a fixed priority among the job classes. The class with the smallest associated value of $E(X_j)E(Y_j)$ is accorded the highest priority (jobs scheduled first) while that with the largest associated value has lowest priority (jobs scheduled last). This simple, intuitive policy is shown to be asymptotically optimal for problems with exponentially distributed lifetimes in a "no premature job loss" limit ($\min_j E(X_j) \to \infty$). In contrast, Argon et al. [2008] develop a myopic heuristic (hereafter denoted π^M) which performs well in a "heavy premature job loss" limit ($\max_j E(X_j) \to 0$). While these policies perform satisfactorily in the neighbourhood of the regimes for which they were designed, they are non-robust and can exhibit poor performance more generally.

The textbook approach to solve this problem exactly is to model it as *Semi-Markov Decision Process* (SMDP) and to apply the methods of stochastic *dynamic programming* (DP). However, since the problem size increases exponentially fast with the number of job classes, the computational effort required is prohibitive in most cases of practical size and so the implementation of DP methods is infeasible. Strongly performing heuristic solutions are thus in order in such circumstances. An important stream of techniques to aid this quest falls under the title of *approximate dynamic programming* (ADP) (Powell [2007]). As far as we know, there is not any universal approximation strategy that works for every problem. Instead customised ADP algorithms are usually developed by exploiting specific problem structure. In this regard, one primary objective of this thesis is to design efficient ADP algorithms to render feasible the task of the development of robustly good heuristic policies for the triage problem.

This problem is complicated very substantially by the introduction of classification errors. The jobs which are assessed and placed into one class could in fact have many different characteristics. Due to this uncertainty, the distributions F_j

and G_j are no longer appropriate descriptions of the lifetimes and service times of all jobs assessed as belonging to class j. We shall use i to index the true class a job belongs to, and accordingly $X_i(Y_i)$ their lifetimes(service times). Moreover, the consequences resulting from any action are more uncertain. Any error-prone triage problem is much harder to analyse than triage with perfect classification and has been rarely studied in the literature.

There are two important contributions which address classification errors when controlling multiclass queueing systems. In neither case does the model incorporate impatience. Van der Zee and Theil [1961] consider a single server queue with two priority classes of jobs. Incoming jobs can be misclassified. To minimize the expected waiting time, they propose a threshold assignment rule to allocate the uncertain job to either class 1, class 2 or some mixed class. After classification, jobs are served in the fixed order of class 1 first, followed by the mixed class, and then class 2. The other contribution is due to Argon and Ziya [2009], who consider a similar priority assignment problem. Each arriving job sends out a signal which gives partial information (of a stochastic nature) about its true identity. On this basis the job is placed into a class. The authors argue that if jobs are partitioned into more distinct classes, the long-run average waiting cost achieved will be decreased. Both contributions focus on optimal priority assignment strategies to alleviate the possible impact of imperfect information. Neither considers job scheduling afterwards and simply offers service in some static priority order. As we have stressed earlier, static service policies can have very poor performance, especially when impatience is a model feature.

To this end, the thesis aims at the development of strong performing heuristic service policies in the presence of classification errors. Solving such a problem is far from a trivial task. Further, we are very interested in understanding the nature of the additional cost incurred by these errors, and exploring system features for which this penalty may be considerable.

1.3 Literature Review

We first explore the general literature on the scheduling of impatient jobs. Particular attention is then given to clearing systems which are assumed to have no new arrivals over time. We also include contributions on admission control and dynamic routing of impatient jobs. These two decisions are often crucial components and precede job scheduling in the three levels of dynamic queueing systems control. Since such problems can usually be solved by ADP, a review on this theme is presented next from the algorithmic perspective. To conclude this section, we summarize previous contributions on *sequential decision making problems with unknown system parameters*, which is the broad problem class in which our error-prone triage problem sits. ADP approaches are an important source of solution methods for these problems also.

1.3.1 Stochastic Scheduling of Impatient Jobs

Impatience can be seen in various real life situations. A typical example is the above mentioned management of medical resource in the aftermath of a MCI. Another example in the healthcare setting concerns a hospital blood bank. If the blood stored in the bank is not used within a certain time of its collection, it may be unusable. In call centres, people will hang up if they are required to wait excessively for service. In banks or supermarkets, customers may abandon a queue if not served within some time of their arrival. In telecommunications, a message is considered lost if its transmission is not completed before some deadline. Interesting examples in a military context concern enemy targets which may move out of reach if not dealt with promptly.

Garnett et al. said in their work on a call centre design problem that "a major drawback of models that ignore abandonment is that they either distort or fail to provide information which is important to call centre managers." (Garnett et al. [2002]). A lot of research attention has been paid to the incorporation of impatience

in the modelling of manufacturing and service systems, and particularly, in resource management problems where limited resources are allocated to different tasks over time. Indeed there is now an extensive literature concerning the scheduling of impatient jobs in various application domains.

Two quite distinct approaches to the modelling of impatience can be found in the literature. One concerns a deadline to the end of service; the other a deadline to the beginning of service. The latter is also referred to as the lifetime or availability time. Under this approach, a common assumption is that jobs will not abandon the system once their service commences. In this thesis, we primarily adopt the second definition. Unless otherwise specified, the terms lifetime and availability time are used interchangeably throughout.

Before proceeding further, we pause to remark that the situation envisaged in this thesis should be clearly distinguished from cases where jobs do not abandon the system even though some deadline may have already passed. The objective in such cases is usually to minimize the total or long run average number of tardy jobs, or to minimize some cost function based on job tardiness. For important examples, see Glazebrook [1983], Pinedo [1983], Boxma and Forst [1986], Emmons and Pinedo [1990], Bhattacharya and Ephremides [1991], Jiang et al. [1996], Doytchinov et al. [2001], Van Mieghem [2003], and Pinedo [2008]. Moreover, all the literature covered in this thesis concerns stochastic scheduling problems. We shall not address deterministic job scheduling. Interested readers are directed to Pinedo [2008] who provides a comprehensive account of job scheduling problems in various contexts.

An early work related to impatient job scheduling is due to Panwar et al. [1988], who consider the transmission of voice packets over a packet-switched network. If the customer does not receive packets in time, they become useless and are lost. The objective is to maximize the long run fraction of successful customer services. It is assumed that upon arrival each packet declares an exact value of its deadline, namely the time available from its arrival to the beginning of its service. Both the

inter-arrival times and the service times are assumed to be independent and identically distributed. No service preemptions are allowed. The authors show that when enforced idle times are prohibited, the shortest time to extinction (STE) policy is optimal for $M/G/1$ queues and for a special case of $G/D/1$ queues. In the cases where enforced idle times are allowed, optimal policies, if they exist, must use the shortest time to extinction with inserted idle times ($STEI$). This problem has been further studied by Bhattacharya and Ephremides [1989]. They claim that when service times are exponentially distributed, the STE policy is, among all nonpreemptive and nonidling policies, the one to minimize the expected number of lost customers over any time interval. A similar argument holds for $STEI$ policies when enforced idling is allowed. They also assert that when there are no new arrivals, idling is not worthwhile and the STE policy is optimal among all nonpreemptive policies. When service preemptions are allowed, a preemptive version of STE is optimal in the general class of nonanticipative policies. These results are true whether the job deadline is defined as the latest service commence time or the latest service completion time. More recently, Shakkottai and Srikant [2002] have studied the scheduling of packets over a multiple channel wireless network. They argue that the results from wireline networks cannot be carried over to wireless networks. The STE policy is not necessarily optimal in the wireless domain. The main reasons are that the wireless channel is not perfect and that errors are location dependent. Packets cannot be transmitted through a bad channel, so the channel state (good, bad) must be considered in the scheduling policy. Assuming that the channel state is perfectly known, the authors show that a STE policy implemented only to good channels is nearly optimal for a class of deterministic arrival processes. Further, when there are no new arrivals, this channel state dependent STE policy is indeed optimal in most cases.

The three works mentioned in the preceding paragraph all assume that job deadlines are deterministic. This may be a reasonable assumption for communication network problems, but the degree of job impatience is generally uncertain.

The exact lifetime of a patient cannot be known *a priori*; the waiting time before a customer leaves a service queue or hangs up an unanswered phone call is not fixed, and so on. Therefore static policies like STE do not work well any more, except in some special cases. The problems in the stochastic impatience setting are usually very hard and optimal policies are not readily available in many cases. Attention has thus been turned to the development of effective heuristics.

Glazebrook et al. [2004] propose a dynamic heuristic policy for a Markovian multiclass single server queueing system with abandonment. Customers arrive in independent Poisson streams. Each customer class has service requirements which conform to some known exponential distribution and availability times are also exponential. Customers abandon the system as soon as their availability times have expired, whether in service or not. A reward is earned upon each successful service completion. The objective is to schedule customers for service to maximize the average reward rate. Gaver et al. [2006] consider a very interesting scheduling problem in a military setting where service completions cannot be perfectly observed. They consider a single server system with multiple classes of uncertain time-critical tasks (enemy targets). The service is to detect, classify and attack these hostile threats. Each task in the system has a class dependent, exponentially distributed availability time for service. The server must process the tasks effectively and efficiently. As a result, it is necessary to control the amount of service given to each. They propose one myopic policy and one Markovian priority heuristic which allocates a fixed amount of processing time to jobs within each class. A special case of this problem that has only a single class of tasks is studied by Glazebrook and Punton [2008]. They propose two dynamic heuristic policies for the determination of processing times. In a different situation, the customers whose deadlines have expired will not abandon the system automatically. Instead they are removed from the queue by the scheduler to avoid wasting server resources. Zhao et al. [1991] consider such a problem and assume that availability times have a concave cumulative distribution function. A scheduling policy must

decide not only how customers should be served, but also how and when customers should be rejected from the system.

All studies reviewed so far concern situations in which a stream of new jobs arrive for service over time. The clearing system model alluded to in Section 1.2 posits an amount (possibly large) of urgent work, arising perhaps as the result of some natural or man-made disaster, which is present *ab initio* and which must be accomplished with limited resources. Direct precursors of this thesis include the studies of Glazebrook et al. [2004] and Argon et al. [2008]. The former considers the scheduling of a batch of impatient jobs in a single server clearing system. Each job has an exponentially distributed availability time, but its processing time distribution can be arbitrary. Every successful service completion yields a job dependent reward, and the objective is to maximize the expected total reward received until the system is cleared. They propose a static permutation policy and prove its convergence to optimal in a no loss limit via an interchange argument. The latter contribution emphasises applications concerning the use of limited medical resources after MCIs. Patients are placed into different priority classes after a triage process, namely an assessment of their urgency for medical attention. Lifetimes and service requirements are class specific and are both exponentially distributed. The authors develop myopic service polices which have been shown to work well when loss rates are high relative to service rates. A slightly different version of this problem is due to Glazebrook and Mitchell [2002], who consider the scheduling of improving/deteriorating jobs in a clearing system. Jobs improve (namely, more close to completion) while being processed, but deteriorate and can even abandon the system whenever service is allocated elsewhere. The goal of scheduling is to maximize the total expected discounted reward.

We now move beyond scheduling and give a review of contributions on admission control and dynamic routing of impatient jobs for service. Admission control concerns decisions on whether jobs are allowed to enter a queueing system. In a multi-queue situation, a routing decision is then made to send the admitted jobs

to specific service stations. Ward and Kumar [2008] consider the admission control of a $GI/GI/1$ queue with impatient customers who have a common exponentially distributed deadline before service completion. The controller must decide whether to admit an arriving customer for which some payment is received. However, a much larger refund must be paid to any admitted customers who subsequently renege after waiting too long without being served. The authors have shown that a simple barrier policy is asymptotically optimal under some stated conditions on the arrival process. Movaghar [2005] considers the problem of dynamically routing arriving impatient customers to parallel queues with identical servers. No jockeying among queues is allowed, and within each queue customers are served in a first-come-first-serve ($FCFS$) fashion. Customer deadlines are i.i.d. and generally distributed. It is shown that when the deadline distribution meets a certain condition, the policy of joining the shortest queue minimizes the expected number of lost customers during any finite interval in the long run. Recently, Glazebrook et al. [2009] have developed heuristic policies for both the admission control and subsequent routing of impatient customers seeking service. They assume a Markovian model, where interarrival times, service times and availability times are all exponentially distributed.

Another contribution due to Lillo [2001] considers the optimal control of an $M/G/1$ queue with impatient customers by means of turning the server on and off. Customers are segmented into two priority classes. The higher priority customers are highly impatient in that they only enter the system if the server is on and idle upon their arrival; otherwise they leave the system immediately. The lower priority customers have zero impatience. The paper shows that, in the class of policies which always turns the server off when the system is empty, the optimal one is to turn the server back on when both of two linear functions of the number of lower priority customers present in the system attain non-positive values.

There is now a considerable literature tailored to call centre applications. Bassamboo et al. [2005] consider the admission control and dynamic routing of mul-

ticlass incoming customers to a group of server pools, each of which is qualified to process only a certain customer class(es). Arrival rates are subject to random changes. According to this paper, if appropriate scaling is applied to the system parameters (arrival rate, service rate, abandonment rate) or the number of server pools, the original problem can be well approximated by a stochastic fluid model. The routing and admission control decisions are then determined by a simple linear program derived from the fluid approximation. The authors are able to derive a control policy for the original system and prove that it is asymptotically optimal for the minimisation of the expected cost over a finite horizon. This problem is extended in a subsequent paper (Bassamboo et al. [2006]) to incorporate decisions concerning the number of servers to be employed in each server pool. In previous literature, the staffing problem is considered separately from the admission and routing problems due to the complexity of addressing them together. The paper assumes that all incoming customers are admitted and that the system consists of two types of costs, namely personnel costs and abandonment costs. Again, they propose strongly performing staffing and routing policies based on an asymptotical analysis in a limiting parameter regime. Helber and Henken [2010] address simultaneously staffing and the shift scheduling of multiskill agents in a contact centre. In contrast with telephone call centres, contact centres can be reached by customers over a variety of media, like phone, email, instant message and so on. They propose simulation optimization approaches and develop policies which are shown to work best for medium to large sized contact centres.

1.3.2 Approximate Dynamic Programming

A standard approach to stochastic scheduling problems is to model them as Markov Decision Processes (MDP) and then apply the methods of DP. Extensive and mathematically rigorous treatments of MDPs can be found in Puterman [1994] and Tijms [1994]. The foundation of dynamic programming is the well known

Bellman equation, which enables computation of the *cost-to-go function* or *value function* in a recursive fashion. Expressed in detail, it expresses the maximum total reward (for finite horizon problems) or the maximum total discounted/long run average reward (for infinite horizon problems) at a system state in terms of the payoff from some immediate action and the maximum reward at the next state which results from these actions. In the infinite horizon discounted reward, discrete state space case, it can take the form

$$V(s) = \max_{a \in A(s)} \left\{ R(s,a) + \beta \sum_{s' \in \Omega} p(s'|s,a) V(s') \right\}, \tag{1.1}$$

where s is the current system state, a the action taken at this state, $p(s'|s,a)$ the one step transition probability, and $R(s,a)$ the reward earned at state s if action a is applied. We use Ω and $A(s)$ for the state space and admissible action space respectively. $V(s)$ is the value function evaluated at state s. Note that β is the discount factor and lies between 0 and 1.

It is clear from (1.1) that there is one Bellman equation for each state. By solving these equations, we compute the value at all states, and simultaneously an optimal policy which determines the best action to take in each state. Solution methods for the Bellman equation include *policy iteration, value iteration, and linear programming.* However, the notorious drawback of DP, the *curse of dimensionality* (Bellman [1961]), means that the exact solution of the Bellman equation, or even storage of the results, is infeasible for a very wide range of problems. The computational and storage requirements grow exponentially with the dimensionality of the state space. In such situations, good suboptimal heuristic solutions are thus in order. Various methods have been proposed for this quest and have been proven to work successfully in specific applications.

In this thesis, we primarily focus on methods which centre on the development of an approximation to the value function. This stream of methods have been well studied and have earned different names. The control theory community uses the term "neuro-dynamic programming", as named after the neural network which is

used in the value function approximation. An important book on this theme is due to Bertsekas and Tsitsiklis [1996]. The artificial intelligence community calls it "reinforcement learning" (Sutton and Barto [1998]). In this thesis we will use the term "approximate dynamic programming", which has been adopted by the operational research community. A recent book due to Powell [2007] provides a comprehensive account on models, algorithms and applications of ADP.

Among various approximation schemes, the simplest one is known as the *lookup table representation*, in which the approximate values, say $\bar{V}(s)$, are stored in a table for each discrete system state s. Such a simple representation still suffers from the curse of dimensionality. For a problem with a fairly large state space, the resulting lookup table will be huge, even though the values in many states are unlikely to be used at all under optimal policies. To get around this issue a simulation based value iteration algorithm has been developed. This algorithm updates the approximation iteratively for the states on the basis of simulated trajectories. The use of simulation here will help to generate representative states so that the computational effort is concentrated on them rather than on the entire state space. Detailed discussion of such algorithms is given in Bertsekas and Tsitsiklis [1996]. Another widely used technique is to aggregate states so as to have a simpler lookup table. Bean et al. [1987] use a fixed level of aggregation. Adaptive state aggregation is considered by Bertsekas and Castanon [1989] and Singh et al. [1995]. The latter also propose a soft state aggregation strategy. The contribution due to Lambert III et al. [2004] proposes an aggregation method by which the solutions produced can be directly implemented to the original problem. The otherwise essential step, disaggregation, is thus not required.

A more efficient scheme, which is called *compact representation*, maps the state by an approximate value function by constructing some parametrized functions $\bar{V}(s, \theta)$, where θ is a vector of parameters. To be attractive the size of the parameter vector needs to be necessarily much less than that of the state space. Only these parameters and the general structure of the function are stored, and the

approximation is generated only as required.

There are two ingredients in this scheme. First, an approximation architecture needs to be selected. In a general sense there are two categories of architectures, namely, linear and non-linear. An important example of the latter is a neural network (Bertsekas and Tsitsiklis [1996]). A notable success story of using neural networks is an application to develop a policy for backgammon by Tesauro [1992]. In a linear architecture, the approximation is expressed as a linear combination of a set of *basis functions* which are weighted by the parameter vector θ. Basis functions are mappings from state variables to the real line and can be nonlinear. They are selected to capture system *features*, namely, important aspects of the system state. The determination of the architecture or basis functions requires a lot of insight on the problem structure. A nice discussion is given in Bertsekas and Tsitsiklis [1996], who have suggested that linear architectures should be used whenever possible. Keller et al. [2006] explore strategies for the automatic construction of basis functions.

Once the architecture is fixed, the parameters are tuned by some statistical methods. Please be aware that a variant proposed by Preux et al. [2009] also tunes the architecture itself. There are a wide range of parameter fitting methods whose performance varies for different approximation architectures. Bertsekas and Tsitsiklis [1996] provide a comparison study of alternative methods. Some important examples introduced in this book include gradient algorithms, least squares methods and Kalman filtering (see also Choi and Van Roy [2006]). For a comprehensive discussion of statistical learning methods readers are referred to an excellent book due to Hastie et al. [2009].

We are still one step away from developing practical approximate DP algorithms. There are no training data pairs available to perform the parameter fitting (in fact, our objective is to generate those data pairs). We neither know if the approximation converges, and if it does, how close the approximation is to the real optimum. In this regard, an important type of algorithm must be syn-

thesized into dynamic programming, namely the *stochastic approximation method.* These methods provide a theoretical framework for iterative estimation of the value functions (in the lookup representation) or the parameter vector (in the compact representation) based on randomly sampled information. They are fundamental tools to analyze the convergence properties of approximate DP algorithms. More details are given in Bertsekas and Tsitsiklis [1996]. A challenging question in these methods concerns the selection of optimal stepsizes, which are used to smooth old estimates with new observations. Powell [2007] gives a thorough discussion of this challenge. A more comprehensive treatment of general stochastic iterative algorithms can be found in Kushner and Clark [1978] and Benveniste et al. [1990].

We are now ready to introduce *approximate value iteration* algorithms (AVI). In a typical iteration of such an algorithm, the approximate value functions are updated at a selected subset of representative states. We write

$$\tilde{V}_{k+1}(s) = \max_{a \in A(s)} \left\{ R(s,a) + \beta \sum_{s' \in \Omega} p(s'|s,a) \bar{V}(s', \theta_k) \right\}, \forall s \in S_k. \qquad (1.2)$$

The subset S_k can be generated by either simulation or state sampling. Some strategies for the latter are presented in Powell [2007]. If the expectation in (1.2) is tractable, the update can be done exactly, otherwise simulation is used to estimate the expectation and the update is done approximately. Based on the values of $\tilde{V}_{k+1}(s), s \in S_k$, a new set of parameters θ_{k+1} for the approximation function are fitted by (for example) least square methods. The above steps are then repeated recursively until the algorithm converges.

Such AVI algorithms were proposed almost as early as dynamic programming itself. Bellman and Dreyfus [1959] approximate the value function by polynomials to accommodate the very small amount of computer memory available then. Whitt [1978] reduces a large scale MDP model to a smaller one by compact representation. See also Reetz [1977]. A more recent contribution due to Choi and Reveliotis [2005] considers a relative value function approximation for long run average reward problems. The application is to a job scheduling problem in a capacitated

re-entrant line, which is modelled as a continuous time Markov decision process. The authors use a linear approximation architecture and demonstrate that it is a promising solution method for the problem considered. A discussion on feature selection strategies is also presented. For AVI algorithms, a known problem is the lack of guaranteed convergence. In other words, there may not exist a unique fixed point. De Farias and Van Roy [2000] show that approximate value iteration should not be expected to converge and present two counter examples. Nevertheless, they propose a variant of AVI which exploits temporal difference (TD) learning and prove that this variant is guaranteed to possess at least one fixed point. However, this property does not ensure convergence. Temporal difference learning is a simulation based policy evaluation method. More details will be given later. Tsitsiklis and Van Roy [1996] also propose two variants of their AVI algorithms. One is based on a lookup table representation in feature space, while the other employs a linear feature based architecture. A proof of convergence of both algorithms is provided. Bounds from the optimal performance are developed to assess their accuracy. Roubos and Bhulai [2010] approach a problem concerning the control of a time-vary queueing system by AVI. A counter-intuitive result is that state disaggregation is preferred to state aggregation in this problem. They argue that their approximation is more accurate when information from more state variables is captured.

Another type of algorithm, called *approximate linear programming* (ALP) and first introduced by Schweitzer and Seidmann [1985], endeavours to find a value function approximation directly by solving a linear program. They generalize the exact linear programming approach by replacing the value functions by linear parametric approximations. The obvious advantage is a drastic reduction in the number of variables, from the size of state space to that of the parameter vector. However, the number of constraints is still prohibitive. A natural response to this challenge is again to concentrate on a subset of states. A second approach is to apply general cutting plane algorithms.

De Farias and Van Roy [2004] propose a sampling scheme to generate a much smaller subset of constraints. The resulting problem is called a reduced linear program (RLP), and is shown to be not only practically solvable, but also to have optimal solutions adequately close to those of the original ALP. Under certain conditions, the constraints to be sampled only grow polynomially in the number of parameters and are independent of the original number of constraints. Trick and Zin [1993] approach a continuous state space stochastic dynamic program by discretization, and solve the discretized problem by ALP with constraint generation methods. They claim that one benefit of using ALP in this situation is the availability of the shadow prices, which are used to generate an efficient discrete grid at no extra computational cost. Much literature on ALP exploits specific problem structure. Morrison and Kumar [1999] investigate the special features of transition probabilities of a queueing network and construct a new ALP in which only a small number of constraints are active. Guestrin et al. [2003] exploit two structures in a factored MDP problem, "additive" and "context-specific", which yield an efficient and accurate linear approximation architecture to the value functions. Further, the original problem can be represented exactly by another LP with exponentially less constraints.

It is a known fact that the state-relevant weights appearing in the objective function have no influence on the solutions in the exact LP algorithms, and thus can take arbitrary positive values. This property, however, does not carry over to their approximate counterparts. The role of these weights is explicitly explored by De Farias and Van Roy [2003]. They have shown that these values have significant impact on the scalability of the ALP algorithms. Guidance on the weight selection strategies for practical problems is provided. This contribution is also the first to evaluate approximation quality by developing error bounds against best possible approximations.

Both AVI or ALP aim at approximating the value functions associated with the unknown optimal policy, which is a non-trivial task in many cases. In contrast,

approximating the value functions associated with a given policy can be much more straightforward. We call these policy value functions. This observation yields a very rich type of algorithm, *approximate policy iteration* (API). Generic API starts with a chosen policy and a fixed approximation architecture to the policy value function. The value functions of this policy are estimated by Monte Carlo simulation or some other method, followed by a parameter fitting procedure. After that, a policy improvement step is performed, which uses the approximate policy value functions obtained by using the latest fitted parameters. These two steps (the policy evaluation step and the policy improvement step) are then applied alternately to the newly constructed policy. This is repeated until convergence is achieved.

A variant of API is based on temporal difference learning. It differs from the generic simulation enabled API in that the policy value function estimation is updated incrementally after each transition, while the latter only updates at the end of one simulation run. TD learning was first introduced in the PhD thesis of Sutton [1984] and since then has been widely used in ADP. The convergence of TD(0) is established by Sutton [1988]. Dayan [1992] extends the result to more general TD(λ) learning methods, where $\lambda \leq 1$ is a discount factor by which the differences of future visited states are exponentially discounted. A stronger convergence result is given by Dayan and Sejnowski [1994]. Jaakkola et al. [1994] relate TD(λ) learning to stochastic approximation theory and provide a rigorous proof of convergence. TD(λ) learning is extended to long run average reward problems by Tsitsiklis and Van Roy [1997]. It is applied by Marbach et al. [2000] to a call admission control and routing problem for integrated service networks.

Policy iteration algorithms usually achieve the greatest improvement in the first few iterations (Tijms [1994]). This observation has motivated the *single step policy improvement* algorithm, in which only one policy iteration is executed. Such algorithms have been widely applied in cases where the policy value functions can be computed exactly. Examples include Glazebrook et al. [2004], Ott and

Krishnan [1992], Bhulai and Koole [2003], and Opp et al. [2005]. If exact policy evaluation is infeasible, approximation methods are employed and this leads to a very important variant, *single step approximate policy improvement* (SSAPI). This has been applied by Roubos and Bhulai [2007] to a problem of admission control to two single server tandem queues over a long period. They construct a very simple initial policy which admits all jobs into the system. The relative value function of this policy is approximated by a second order polynomial. Bhulai [2009] develops a variant of SSAPI for a call routing problem in a multi-skilled call centre. For either of the two settings studied, he proposes an initial policy whose special features render feasible a good approximation to the relative value function by a non-parametrized architecture.

ADP algorithms (except API) eventually lead to the best estimate of the value functions $\bar{V}^*(s)$, if convergence is achieved. There is one remaining step, which is to construct implementable policies based on these values. For API, an updated policy must be constructed at every iteration. In either case, it can be done by solving the following equation (replace $\bar{V}^*(s)$ by the policy value function approximation for API)

$$\pi(s) = \arg\max_{a \in A(s)} \left\{ R(s,a) + \beta \sum_{s' \in \Omega} p(s'|s,a) \bar{V}^*(s') \right\}. \tag{1.3}$$

Unfortunately, the exact solution to this equation remains challenging if the action space is large and/or the calculation of the expectation is intractable.

It is not hard to find practical problems in which the action space has more than thousands of dimensions. A typical example mentioned in Powell [2007] is the blood inventory management problem. The decision concerns the allocation of each available blood type to meet the demand of another type. The number of decisions increases exponentially with the number of valid substitution pairs. To deal with this difficulty, Bertsekas and Tsitsiklis [1996] suggest the incorporation of actions into states and then to solve the augmented state space problem by value function approximation methods. Powell [2007] proposes a more systematic

technique, namely a synthesization of mathematical programming into ADP. It is the first work to integrate these two techniques. To make this approach workable, the classical pre-decision state variables must be replaced by *post-decision* ones. More details can be found in the book and the references therein.

As for the expectation in (1.3), we can always approximate it by use of Monte Carlo simulation. In the ADP literature, there is an important quantity whose use enables this difficulty to be bypassed. It is called the *Q-factor*, and is defined for a state-action pair (s, a) and a policy π. In particular, the optimal Q-factor denotes the total expected reward obtained when action a is applied in state s and the optimal policy is followed thereafter. This is written as

$$Q(s,a) = R(s,a) + \beta \sum_{s' \in \Omega} p(s'|s,a) V(s').$$

Together with the Bellman equation (1.1), a recursive equation to compute optimal Q-factors is derived as follows,

$$Q(s,a) = R(s,a) + \beta \sum_{s' \in \Omega} p(s'|s,a) \left(\max_{a' \in A(s')} Q(s',a') \right).$$

A strategy called *Q-learning* to approximate the optimal Q-factors was introduced by Watkins and Dayan [1992]. The power of Q-learning lies in its ability to break the so-called *curse of modeling*, which describes the difficulty of explicitly calculating the transition probabilities for complex systems with multiple governing random variables. However, this viewpoint has been challenged by Powell [2007], who argues that the real value of Q-learning is the release from expectation computation. Indeed, once the approximate optimal Q-factors, $\bar{Q}^*(s,a)$, have been obtained, the computation of the policy by the equation

$$\pi(s) = \arg\max_{a \in A(s)} \left\{ \bar{Q}^*(s,a) \right\}.$$

is trivial. This is in sharp contrast with (1.3). A convergence proof of Q-learning

can be found in Tsitsiklis [1994] and Jaakkola et al. [1994]. Crites and Barto [1998] apply Q-learning to an elevator control problem. An online Q-learning algorithm is proposed by Levy et al. [2006] and is used to control a non-stationary Markov decision process. The learning rate is updated adaptively to track parameter changes. Leslie and Collins [2005] propose an individual Q-learning approach to deal with the *iterated normal form game*. They introduce player dependent learning rates for which convergence results can be proved in a large number of cases. More details of this algorithm are given in Bertsekas and Tsitsiklis [1996] and Sutton and Barto [1998].

We conclude this section with reference to a fundamental question in ADP. Recall that in all ADP algorithms, a subset of representative states are generated and computational effort is concentrated on them. This is essential to break the curse of dimensionality. A natural question to ask is, should we make decisions on just these states, or we should instead try some new ones? Because no *a prior* knowledge about the optimal policy is available, we do not know which subset we should concentrate on. It may well be the case that some important states are left out, and the algorithm could be trapped in a local optimum. It is thus necessary to do some exploration of the state space, which however could be costly and time consuming. A tradeoff between the cost of exploration and their future values must be considered. This issue is referred to as "exploration vs. exploitation" and is discussed in Powell [2007]. Singh et al. [2000] consider different exploration strategies and provide corresponding convergence results. A nice survey and discussion on exploration schemes in learning control is due to Thrun [1992].

1.3.3 Sequential Decision Making with Unknown System Parameters

Even though only two direct precursors of the error-prone triage problem have appeared in the literature, the general problem of sequential decision making with unknown system parameters has been studied extensively. Please note that these are called *stochastic adaptive control* problems by the control theory community. In such a problem, the system under study has some unknown elements. Taking job scheduling as one example, the distributions of service times may not be completely known. Instead, they may depend upon an unknown parameter.

The relevant literature can be divided into two distinct threads. One concerns *Bayesian sequential decision problems* (BSDPs), in which a prior distribution on the unknown parameters is given. The other deals with non-Bayesian approaches in which no prior distribution is available. Instead, the decision maker is usually given a set which is believed to contain the unknown parameter. In this thesis we shall be concerned with Bayesian sequential decision problems. For non-Bayesian approaches readers are referred to a broad survey due to Kumar [1985].

In BSDPs, information regarding the unknown parameters is obtained and is available to the decision maker over time. The objective is to design control policies to optimize some pre-specified cost function. A problem of this kind is made difficult not only because of the uncertainty around the system evolution, but also because of changing beliefs in the unknown parameters. The standard solution approach is to convert such problems into an equivalent DP. Then the rich DP theory and methods are available. Much attention had been devoted in the 1960s-70s to resolving delicate questions associated with this conversion and to obtaining well formulated DP approaches. For important examples see Bellman [1961], Martin [1967], Hinderer [1970], Furukawa [1970], Rieder [1975]. and Kumar [1985]. Sadly, as we have already seen, DP problems can rarely be solved to optimality. Hence ADP algorithms are sought to tackle these problems.

Some early contributions tried to bound the optimal solution. Van Hee [1978]

proposed an approach to compute lower and upper bounds on the optimal discounted cost function. Calculation of the bounds involves the optimal solution to two sets of nonadaptive control problems for which the parameters are assumed known. The solutions of both problem sets are then manipulated with a DP operator in an iterative fashion to generate the lower and upper bounds. It has been shown that as the number of iterations diverges to infinity then both bounds converge monotonically to the real optimum of the original problem. However, this calculation procedure can be quite clumsy. Some special cases described in Van Hee [1978] allow much simpler procedures. Other lower and upper bounds can be found in Martin [1967], Satia and Lave [1973] and Waldmann [1985].

There is one class of BSDPs whose special structure enables the development of rigorous theory and exact optimal solutions. These are the *multi-armed bandit* problems, for which the policy induced by the celebrated *Gittins Index* is proven to be optimal. The Gittins Index was first proposed by Gittins and Jones [1974] under the name of the *Dynamic Allocation Index* (*D.A.I.*). Later, Whittle [1980] gave a simpler proof of the optimality of Gittins Index policies than provided by Gittins and Jones [1974] and used the term Gittins Index. The index policy has a particularly simple form, which can be obtained by computing an index function for each alternative bandit. There are extensive contributions on this theme, with broad applications.

Stochastic Job Scheduling

An early paper is due to Gittins and Glazebrook [1977], who apply the Gittins Index theorem to stochastic scheduling problems where the distributions of the jobs' service times are dependent upon some unknown parameters. Discounted rewards are obtained when a job's service is completed and the objective is to achieve the total maximal expected reward. With the knowledge of the prior distributions of these parameters, each job is modelled as a *Bayesian bandit process*, and a *memoryless Bayesian bandit process* if the posterior distributions of the parameters

depend only on the current state. It has been shown that, by incorporating the parameters and their probabilities into the law of motion, this generalized problem can be always reduced to the problem of allocating resources for a multi-armed bandit. Hence the Gittins Index theorem applies.

Hamada and Glazebrook [1993] consider a Bayesian sequential single machine scheduling problem, where the objective is to minimize the expected sum of weighted flowtimes. Jobs are grouped into classes. Within each class the processing times are i.i.d. and exponentially distributed with an unknown parameter. The conjugate priors on the parameters are gamma. The system state is augmented by including the parameters of the gamma distribution and the problem is formulated as a DP. The optimal strategy is then obtained by applying the Gittins Index theorem.

Recent work by Cai et al. [2009] considers the scheduling of a batch of jobs on a single machine that is subject to breakdown. If the machine breaks down in the middle of processing a job, all work done to date is lost and the job must be processed again from the beginning once the machine is fixed. Both the distributions of the processing times and the machine up/downtimes are unknown. The authors develop the Gittins Index from the posterior distributions of the unknown parameters and so generate optimal dynamic policies.

Clinical Trials

Glazebrook [1978] studies the allocation of multiple treatments to patients in a series of clinical experiments. The set of outcomes of an experimental treatment can be fairly large. The probabilities of the individual outcomes are unknown. This treatment allocation problem is modelled as a multi-armed bandit and an index function is developed for each bandit (treatment). The optimal strategy is to choose the treatment with the smallest index at each decision point in the trial.

Optimal Exploration

Benkherouf et al. [1992] consider a Bayesian model of oil exploration. There are a number of candidate areas to explore and each area contains an unknown number of oil fields. The oil company needs to know which area to drill next or whether to choose to drill no well, based on the knowledge learned from earlier exploration. The optimal strategy is established and is based on the Gittins Index. Unfortunately, according to the authors the optimal strategy may be difficult to apply, since it may involve a lot of switching between areas. In this case, the Gittins indices can be used to evaluate heuristics by bounding the lost revenue when choosing them instead of an optimal policy.

Glazebrook and Boys [1995] extend the oil exploration model to a general search problem. This concerns the determination of an optimal strategy to search objects in several locations. Each location has an unknown number of objects of value. A single search can lead to the discovery of multiple objects and a reward is earned accordingly. The discovered objects are then removed from the location before the next search. A binomial distribution is assumed as a conditional model of the number of objects discovered in a single search. The authors show that a Gittins Index policy is optimal for this problem and that the nature of this policy depends critically on the the prior distributions on the number of unknown objects in each location. If these priors are either Poisson or have a lighter tail than the Poisson, the optimal policy is myopic and searches whichever location yields the largest immediate expect reward. In this case a lot of switching between locations may be involved. On the other hand, if all priors have heavier tails than the Poisson, the optimal policy becomes a kind of "stay with the winner" rule.

There is also a rich literature on more general BSDPs, in cases where heuristics are developed because of the difficulty of finding the optimum.

Screening Design

Boys et al. [1996b] study a screen design problem where both the therapeutic effect and the toxicity of pharmaceutical compounds are tested by passing them through two series of screens. Each kind of test must be performed in a specified order but between them they can be interleaved arbitrarily. The authors propose a Bayesian formulation and specify a joint prior density for measures of activity and toxicity. The heuristic which always chooses as the next screen the one with larger failure probability per unit cost is shown to work well under some simple conditions. Boys et al. [1996a] extend this problem to a more general setting, where the question is how to filter out, in a cost effective way, items with acceptable attributes. It is very expensive to measure the attributes, but there are some associated covariates whose measurement is essentially free. They construct a Bayes-optimal two stage screen. At stage 1 screening is via the covariates and only indecisive items are passed over to stage 2 when the attributes are measured. The authors suggest that even though the measurement of the attributes is very expensive, it may be worth doing in individual cases.

Inventory Control

Azoury [1985] considers periodic review inventory problems where the demand distribution is dependent upon some unknown parameters. Two inventory models are analysed, namely a depletive model of consumable items and a nondepletive model for repairable items, both of which are formulated as Bayesian dynamic programs. By imposing certain conditions on the demand distribution and on the prior for the unknown parameters, they show that the development of the optimal ordering policy in either model can be reduced to the solution of a dynamic program with one variable only.

Optimal Search

A decentralized multiagent search problem is studied by Zhao et al. [2008]. There are several agents searching for targets over a network. Communication and coordination among agents are limited, a feature which is referred to as a *coordination dilemma.* The consequence of the coordination dilemma is the loss of a global optimum even though an individual agent could follow an optimal strategy himself. The authors claim that it is always favourable to implement the same randomized policy for all agents individually. Three heuristics are proposed and one of them is based on DP policy iteration. The starting point for this is an optimal policy for the corresponding centralized multiagent search problem which can be formulated and solved as a dynamic program.

As can be seen from the above, BSDP problems are usually computationally demanding to solve and/or close to optimal adaptive policies may be difficult to implement. A natural question concerns whether or not it is worth developing adaptive policies. This question is investigated by Glazebrook and Owen [1995]. They introduce the value of adaptive solutions (VAS) to quantify the benefit brought about by learning about unknown parameters. The VAS is defined as the loss experienced when deeming the system's unknown parameters to be known and then developing an optimal policy for the known case. The authors are able to relate the VAS to two model features, namely, the degree of peakedness of the prior distribution and the sensitivity of optimal policies to the assumed values of the unknown parameters. According to the authors, the VAS is small if the prior is peaked and/or optimal polices are insensitive for the scheduling models under consideration.

We would like to mention that a slightly different body of literature deals with partially observable Markov Decision Processes (POMDP). In this literature all elements of the systems are known. The uncertainty relates to knowledge of the current system state. It could be costly or, indeed, impossible to observe

the current state completely and without error. Only information relating to the current state is obtained. Despite this difference, the solution methodology is somewhat similar. By redefining the state space to be a space of distributions over the actual system states, any POMDP can be reformulated as a completely observed, continuous MDP problem. General techniques for continuous MDPs can then be applied. However, some special properties of this converted problem allow tailored techniques to be developed. For comprehensive and rigorous treatments of POMDPs, readers are referred to Sawaragi and Yoshikawa [1970], Monahan [1982], Sondik [1978], and the PhD dissertation of Cassandra [1998] and the references therein.

1.4 Contributions

We summarise the primary contributions of the thesis as follows:

1. For the perfect triage problem, we exploit the simplicity (especially the static nature) of the heuristic π^S proposed by Glazebrook et al. [2004] to develop a new class of heuristic policies with robustly strong performance via a two stage procedure. At stage 1, we use a fluid model to approximate the policy value function of the system operating under π^S. We then adopt an approximate DP approach and design in stage 2 a dynamic heuristic by using the approximate policy value function from stage 1 in a single step policy improvement algorithm.

2. Taking advantage of the special structure of the policy π^S, the fluid model we have developed has a very simple, deterministic, and non-parametric architecture. This architecture enjoys the advantages of compact representation, yet avoids the non-trivial task of parameter fitting. The solution to the fluid model is very fast and straightforward, nothing more than solving an array of ordinary differential equations, one for each job class. Numerical results

demonstrate that the approximated value functions are very close to the exact ones for all the states examined.

3. We conduct extensive numerical experiments to investigate the performance of our proposed heuristics for the perfect triage problem in three general scenarios. These are (I) lifetimes and service times are both exponentially distributed, (II) Weibull lifetimes and deterministic service times, and (III) Weibull lifetimes and exponential service times. In the first two scenarios we are able to compute optimal policies using exact DP methods, though this is very expensive of computing time, except for very small problem instances. It is thus possible to assess the quality of our proposed heuristics by direct comparison with the optimum. For the third scenario, it has not proved possible to develop optimal policies for problems of even modest size in reasonable time. However, the way that the proposed heuristic policies are developed means that their on-line implementation is straightforward. Hence, we chose to assess the *relative performance* of alternative heuristics by means of Monte Carlo simulation. Numerical results have shown that our proposed heuristics perform extremely well, comfortably outperforming competitors, in all the testing instances considered. This work on the perfect triage problem has appeared as Li and Glazebrook [2010a].

4. To explore the error-prone triage problem, we propose a simple analytical model and adopt a *Bayesian* approach to address the uncertainty of the true identity of each job. Hence immediately after triage ($t = 0$) each job has a *prior distribution* which summarises the decision maker's beliefs about its true identity before service begins. As time passes, these beliefs are modified at every time $t > 0$ and *posterior distributions*, which condition on the event that a surviving job's lifetime exceeds t, are computed using Bayes' Theorem.

5. We formulate this Bayesian sequential decision problem as a dynamic program. The ADP approach proposed for the perfect triage situation is further

developed to yield effective solutions to our Bayesian model. We successfully extend the fluid model approach to accommodate triage errors and the approach still generates high quality policy value function approximations in this case. A numerical study testifies to the strong performance of the resulting heuristic service policy.

6. We then explore the question of whether it is possible to identify problem features in which poor triage is particularly costly in terms of the number of service completions lost. To this end, we introduce the *(relative) cost of imperfect classification*, denoted $(R)CIC$, as a natural measure of this. We are able to develop an analytical upperbound for $(R)CIC$ for the case in which the random lifetimes X_i are exponentially distributed. This bound tells us that in the exponential lifetime case, $(R)CIC$ is small whenever the system parameter

$$\Delta \equiv \max_i E(X_i)E(Y_i) - \min_i E(X_i)E(Y_i)$$

is small. In such cases the triage process is *relatively* unimportant for the scheduling problems. Numerical studies indicate that these insights extend beyond the exponential case and suggest strongly that there is most to be gained for the scheduling problem from improving the quality of triage when Δ is large. A paper (Li and Glazebrook [2010b]) describing these contributions to the error-prone triage problem has been submitted for publication.

1.5 Outline of the Thesis

Chapter 2 investigates the scheduling of impatient jobs in a clearing system with perfect classification. A SMDP is constructed to model this problem. We present an efficient ADP approach to the development of dynamic heuristic policies via a fluid model approximation. An extensive numerical investigation is carried out to

compare the performances of our proposed heuristic policies to earlier proposals in the literature and (where possible) to optimal.

We extend to the error-prone triage problem with imperfect classification in Chapter 3. After the introduction of some additional notation this problem is modelled as a Bayesian sequential decision problem which is then formulated as a dynamic program. The solution approach and the fluid model proposed in Chapter 2 are further developed so that they can yield effective service policies in the presence of classification errors. Again, a numerical analysis is conducted to assess the quality of the heuristics.

In Chapter 4 the cost caused by imperfect classification is studied. We propose a measure ($RCIC$) to quantify the cost. A $J = 2$ example is presented to illustrate the sensitivity of $RCIC$ to the error rates. For the exponential lifetime cases, we find an analytical upperbound for $RCIC$ that depends heavily on an identified system parameter. A detailed proof is given. For more general problems, a worst case numerical study is conducted to explore thoroughly the impact of this system parameter.

We conclude in Chapter 5 with a summary and a discussion of possible future research directions.

Chapter 2

Scheduling of Impatient Jobs with Perfect Classification

This chapter considers the scheduling of impatient jobs with perfect classification. It proceeds as follows: the problem is modelled as a semi-Markov decision process in Section 2.1. In Section 2.2 we describe an approach to the development of heuristic policies via an approximating fluid model. Our proposed heuristics are implemented in three general scenarios and subject to extensive numerical investigation in Section 2.3 where they are compared to earlier proposals in the literature and (where possible) to optimal. A conclusion is given in Section 2.4.

2.1 The Model

A clearing system has a single server and a collection of impatient jobs (or customers) awaiting service. Before any service starts, each job is allocated to one of J classes after a triage process. We use the pair jk to denote the job which is the k^{th} member of class j, $1 \leq k \leq L_j, 1 \leq j \leq J$. Observe that L_j is the number of class j jobs present at time 0. Associated with each job jk are two positive valued random variables, namely its *lifetime* X_{jk} and its *service time* Y_{jk}. Class j lifetimes $X_{j1}, X_{j2}, \cdots, X_{jL_j}$ are independent and identically distributed (i.i.d.), having the

same distribution as X_j whose distribution function is F_j. Similarly, the collection $\{Y_{j1}, Y_{j2}, \cdots, Y_{jL_j}\}$ of class j service times are independent and identically distributed, having the same distribution as Y_j whose distribution function is G_j. All lifetimes and service times have finite expectation and are independent of each other.

The single server processes individual jobs nonpreemptively. Job jk will abandon the system unserved if its service has not begun before X_{jk}. However, once a job has begun service, it will be served through to completion. Let π denote a service policy (a nonanticipative rule for allocating the server to waiting jobs) and $T_{jk}(\pi)$ the random time at which policy π begins to process job jk. If jk is not served by π then we write $T_{jk}(\pi) = \infty$. The number of jobs served to completion under π is denoted $N(\pi)$ and is given by

$$N(\pi) = \sum_{j=1}^{J} \sum_{k=1}^{L_j} \mathbb{I}\{T_{jk}(\pi) \leq X_{jk}\}. \tag{2.1}$$

In (2.1), $\mathbb{I}$ is an indicator. The goal of analysis is the determination of a policy π to maximise $E\{N(\pi)\}$. Argon et al. [2008] argue that under the optimal policy the server will never idle, while the theory of stochastic dynamic programming (see, for example, Puterman [1994]) guarantees the existence of an optimal policy which takes actions which depend only upon the current system state.

We model this problem as a semi-Markov decision process as follows:

1. Decision epochs are at time zero and at all service completion times. The *state of the process* at decision epoch $t \geq 0$ is denoted $\{n_j(t), 1 \leq j \leq J; t\} \equiv \{\mathbf{n}(t), t\}$ where $n_j(t)$ is the number of class j jobs which at time t have not yet been served and have not abandoned the system. Generic states of the system are denoted $(\mathbf{n}, t), (\mathbf{n}', s)$. Note that the number of *effective* states decreases as time passes. A state is effective if it has a positive probability ever to be visited given the initial condition.

2. At each decision epoch, one of the jobs remaining in the system is chosen for

processing. In any system state $(\mathbf{n}, t)$, the collection of admissible actions is written $A(\mathbf{n})$ and is given by

$$A(\mathbf{n}) = \{j; n_j \geq 1, 1 \leq j \leq J\}. \tag{2.2}$$

In (2.2) the action j is identified with the class of the job chosen for processing.

3. Let t be a decision epoch and $(\mathbf{n}, t)$ the system state then. If action $j \in A(\mathbf{n})$ is taken and results in a service time (realized value of Y_j) equal to s then the system at the next decision epoch $t+s$ will be $(\mathbf{n}', t+s)$ with probability $p(\mathbf{n}'|\mathbf{n}, t, j, s)$ given by

$$\begin{aligned} & p(\mathbf{n}'|\mathbf{n}, t, j, s) \\ & = \prod_{i=1}^{J} \binom{n_i - \delta_{ij}}{n_i'} \{P[X_i \geq t+s | X_i > t]\}^{n_i'} \{P[X_i < t+s | X_i > t]\}^{n_i - \delta_{ij} - n_i'} \\ & = \prod_{i=1}^{J} \binom{n_i - \delta_{ij}}{n_i'} \left\{\frac{1 - F_i(t+s)}{1 - F_i(t)}\right\}^{n_i'} \left\{\frac{F_i(t+s) - F_i(t)}{1 - F_i(t)}\right\}^{n_i - \delta_{ij} - n_i'}, \\ & 0 \leq n_j' \leq n_j - \delta_{ij}, 1 \leq i \leq J. \end{aligned} \tag{2.3}$$

In (2.3), δ_{ij} is the Kronecker delta which is equal to one when $i = j$ and is otherwise zero.

4. A *policy* π is any nonanticipative rule for choosing admissible actions. Our goal is the determination of a policy to maximise the expected number of jobs served from initial state $(\mathbf{L}, 0)$.

In principle, an optimal policy could be developed with the tools of stochastic dynamic programming. Write

$$\Omega = \{(\mathbf{n}, t); 0 \leq n_j \leq L_j, 1 \leq j \leq J, t \in \mathbb{R}^+\}$$

for the system's state space and develop the value function $V : \Omega \rightarrow \left[0, \sum_{j=1}^{J} L_j\right]$

where $V(\mathbf{n}, t)$ is the maximal expected number of service completions from state $(\mathbf{n}, t)$. Assuming sufficient regularity, V satisfies the optimality equations

$$V(\mathbf{n}, t) = 1 + \max_{j \in A(\mathbf{n})} \left\{ \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, j, s) V(\mathbf{n}', t+s) dG_j(s) \right\}, \mathbf{n} \neq \mathbf{0},$$
$$V(\mathbf{0}, t) = 0. \tag{2.4}$$

It is trivial that $V(\mathbf{n}, t)$ is monotonically increasing componentwise in $\mathbf{n}$, for fixed t. Moreover, from our numerical experiments, we observe that in many cases the optimal policy is also monotone componentwise in $\mathbf{n}$ for fixed t. Specifically, if the optimal policy moves from one action to another as the number of jobs in one class increases/decreases, it will stay on the latter action if the number of jobs in this class is further increased/decreased.

As we have stressed repeatedly, computational approaches to the determination of optimal policies built around the recursive scheme in (2.4) are not practical for problems of realistic size. We seek to develop *heuristic* approaches which are close to reward maximising.

Remark 2.1. *The heuristic approaches to be developed in the coming section are generic. The implementation in scenarios where the state space is discrete and finite is straightforward. In general situations, however, the state space Ω is continuous and infinite. This is attributed to the inclusion of the element t in the state space. The standard approach is to discretize the continuous time axis and develop heuristics for the resulting discrete problem. Unfortunately, this treatment may blow up the state space which may be already intractable in many cases. A scenario of such is dealt with in this thesis and a detailed account is given in the numerical study section 2.3. For general approaches to continuous MDP problems, see for example Boyan and Littman [2001], Li and Littman [2005] and Marecki et al. [2006].*

Two heuristics have been proposed in the literature and both will play a role in

our narrative. Glazebrook et al. [2004] proposed a *static priority rule* π^S. Suppose that the job classes are numbered in *increasing order* of the quantity $E(X_j)E(Y_j)$, i.e. such that

$$E(X_1)E(Y_1) \leq E(X_2)E(Y_2) \leq \cdots \leq E(X_J)E(Y_J). \tag{2.5}$$

In any state $(\mathbf{n}, t)$, π^S chooses action $\pi^S(\mathbf{n}, t)$ where

$$\pi^S(\mathbf{n}, t) = \min\{j; n_j \geq 1\}.$$

Hence π^S implements a class priority according to the ordering $1 \rightarrow 2 \rightarrow \cdots \rightarrow J$. It favours jobs with small mean service times and/or small mean lifetimes, and specifically, jobs which have large probabilities of abandonment if the service is allocated to any of the others. It was shown by Glazebrook et al. [2004] to be optimal in a "no premature job loss" limit when job lifetimes are exponentially distributed.

Argon et al. [2008] propose a myopic heuristic policy π^M which takes the following form: in state $(\mathbf{n}, t)$, π^M chooses action $\pi^M(\mathbf{n}, t)$ to be the non-empty class j with smallest associated value of

$$E(Y_j)\left[\sum_{i=1}^{J}(n_i - \delta_{ij})\{E(X_i - t|X_i > t)\}^{-1}\right]. \tag{2.6}$$

This quantity can be understood as an approximation to the mean number of abandonments while serving a class j job. Therefore, policy π^M gives priority to the actions for which the mean number of abandonments during the next service is close to minimal. It works well in a "heavy premature job loss" limit. In such cases most jobs will be lost at an early stage and the value function is determined by the very first few actions, which are "optimised" by π^M in some sense.

As we shall see, both heuristics π^S and π^M perform well on occasion, but exhibit a lack of robustness in performance. Namely there are problems for which

they do not work at all well. In the next section we propose an approach to the development of a heuristic policy with associated performance which is robust and stronger than either.

Remark 2.2. *Minor adjustments to the above are required for simple variants of our model, such as (a) incorporating the possibility of loss during service, and/or (b) different returns earned upon completion of services of jobs from different classes. For example, scenario (a) requires an adjustment of the optimality equations in (2.4) to*

$$\begin{aligned}\bar{V}(\boldsymbol{n},t) = \max_{j\in A(\boldsymbol{n})}\Bigg\{ & P(X_j > t + Y_j | X_j > t) \\ & + \int_0^\infty \sum_{\boldsymbol{n}'} p(\boldsymbol{n}'|\boldsymbol{n},t,j,s)\bar{V}(\boldsymbol{n}',t+s)dG_j(s)\Bigg\}, \boldsymbol{n} \neq \boldsymbol{0}, \\ \bar{V}(\boldsymbol{0},t) = 0. & \end{aligned} \tag{2.7}$$

In (2.7) we make an assumption that service times are delivered in full even when premature loss occurs. There are other modelling possibilities. It is evidently the case that $\bar{V}(\boldsymbol{n},t) \leq V(\boldsymbol{n},t)$ *for all choices of* $\boldsymbol{n},t$.

2.2 Heuristic Policy Development - a Single Step Approximate Policy Improvement Algorithm via Fluid Models

Of the heuristic policies described at the conclusion of the preceding section, π^S enjoys the benefits of a very simple structure. It seems reasonable to explore the possibility of designing effective dynamic heuristics for our problem by strengthening the performance of this *static* policy via the implementation of a single DP policy improvement step. Write $V_{\pi^S} : \Omega \to \left[0, \sum_{j=1}^J L_j\right]$ for the value function

for π^S, namely $V_{\pi^S}(\mathbf{n}, t)$ is the expected number of service completions from state $(\mathbf{n}, t)$ under policy π^S. The function V_{π^S} satisfies the recursion

$$V_{\pi^S}(\mathbf{n}, t) = 1 + \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, \pi^S(\mathbf{n}, t), s) V_{\pi^S}(\mathbf{n}', t+s) dG_{\pi^S(\mathbf{n},t)}(s), \mathbf{n} \neq \mathbf{0},$$

$$V_{\pi^S}(\mathbf{0}, t) = 0. \tag{2.8}$$

A single DP policy improvement step applied to π^S will result in a new dynamic policy π^{SPI} determined as follows:

$$\pi^{SPI}(\mathbf{n}, t) = \arg\max_j \left\{ \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, j, s) V_{\pi^S}(\mathbf{n}', t+s) dG_j(s) \right\} \tag{2.9}$$

with the argmax in (2.9) being taken over the admissible set $A(\mathbf{n})$. In words, policy π^{SPI} makes optimal decisions under an assumption that all future decisions are made according to π^S.

Our experience is that policy π^{SPI} performs very strongly when it is available. In support of this claim we make reference to 2,000 randomly generated problems in which $J = 2$ and both lifetimes and service times are exponentially distributed. The performance of π^S and π^{SPI} when applied to these problems is given in Table 2.1(a), while the details of the problems themselves are given in Section 2.3. Similar results for 2,000 randomly generated problems with $J = 5$ and exponentially distributed lifetimes and service times are given in Table 2.1(b). In both tables, the results are presented in four groups (500 problems in each group) labelled A, B, C and D according to the *relative* lengths of lifetimes and service times in the generated problems. In Table 2.1(a), the worst performances of heuristic π^S within each group, as measured by the percentage deviation from optimum, are 23.79% (group A), 24.33% (B), 16.52% (C) and 5.17% (D). Once a policy improvement step is applied to π^S as in (2.9) above, the corresponding worst case percentages for π^{SPI} are 0% for all four groups. In Table 2.1(b), the worst case percentages for π^S are 15.95% (A), 11.86% (B), 3.87% (C) and 0.33% (D) while those for π^{SPI} are 0.48%

(A), 0.88% (B), 1.04% (C) and 0.05% (D). In all groups and for both tables the median percentage suboptimality for π^{SPI} was 0%. Further, for non-exponential cases, results for 2,000 randomly generated problems in which $J = 2$, lifetimes are Weibull distributed and service times are deterministic are given in Table 2.1(c). Full problem details may be found in Section 2.3. The design of the study is along the lines of the exponential cases above. In Table 2.1(c), the worst case percentage for heuristic π^S are 36.34% (group A'), 26.16% (B'), 20.21% (C') and 4.13% (D'). The corresponding percentage for π^{SPI} are 0.00% (A'), 0.24% (B'), 1.46% (C') and 0.20% (D').

Remark 2.3. *A dynamic version of the priority policy π^S recalculates the expected remaining lifetimes at every decision epoch and updates the priority list. We have also implemented a single DP policy improvement step for this policy in all the 2,000 problems for Weibull lifetimes and deterministic service times when $J = 2$. The resulting policy is referred to as π^{DSPI} and its performance is summarized in Table 2.2, in which we have also included the results for policy π^{SPI} for the reader's convenience. It is shown clearly that π^{DSPI} does not offer any significant improvement on π^{SPI}. In some cases its performance is weaker. It is worth mentioning that in some special cases the priority sequence does not change over time and hence π^{DSPI} reduces to π^{SPI}. An obvious example is when lifetimes are exponentially distributed. Another example could be when the lifetime distributions satisfy a certain conditions such that the expected remaining lifetimes remain a fixed ordering between classes.*

The strong performance of π^{SPI} notwithstanding, its development via (2.8) and (2.9) is computationally prohibitive other than for small problems and special cases. In light of this computational intractability we proceed as follows: we shall develop an approximation $V_{\pi^S}^{app} : \Omega \to \left[0, \sum_{j=1}^{J} L_j\right]$ to the policy value function V_{π^S}. The dynamic heuristic which then results is obtained by using the approxi-

Category		$\Delta(\pi^{SPI}, opt)$	$\Delta(\pi^{S}, opt)$
	MIN	0.00	0.00
A	MEAN	0.00	3.01
(very short lifetimes)	MAX	0.00	23.79
	MIN	0.00	0.00
B	MEAN	0.00	2.52
(short lifetimes)	MAX	0.00	24.33
	MIN	0.00	0.00
C	MEAN	0.00	1.23
(moderate lifetimes)	MAX	0.00	16.52
	MIN	0.00	0.00
D	MEAN	0.00	0.12
(long lifetimes)	MAX	0.00	5.17

(a) Exponential lifetimes and service times when $J = 2$.

Category		$\Delta(\pi^{SPI}, opt)$	$\Delta(\pi^{S}, opt)$
	MIN	0.00	0.00
A	MEAN	0.00	1.36
	MAX	0.48	15.95
	MIN	0.00	0.00
B	MEAN	0.01	0.84
	MAX	0.88	11.86
	MIN	0.00	0.00
C	MEAN	0.01	0.13
	MAX	1.04	3.87
	MIN	0.00	0.00
D	MEAN	0.00	0.00
	MAX	0.05	0.33

(b) Exponential lifetimes and service times when $J = 5$.

Category		$\Delta(\pi^{SPI}, opt)$	$\Delta(\pi^{S}, opt)$
	MIN	0.00	0.00
A'	MEAN	0.00	4.27
(very short lifetimes)	MAX	0.00	36.34
	MIN	0.00	0.00
B'	MEAN	0.00	2.53
(short lifetimes)	MAX	0.24	26.16
	MIN	0.00	0.00
C'	MEAN	0.02	0.71
(moderate lifetimes)	MAX	1.46	20.21
	MIN	0.00	0.00
D'	MEAN	0.00	0.22
(long lifetimes)	MAX	0.20	4.13

(c) Weibull lifetimes and deterministic times when $J = 2$.

Table 2.1: Percentage deviation from optimal performance of heuristic policies π^S and π^{SPI}.

Category		$\Delta(\pi^{SPI}, opt)$	$\Delta(\pi^{DSPI}, opt)$
A' (very short lifetimes)	MIN	0.00	0.00
	MEAN	0.00	0.00
	MAX	0.00	0.00
B' (short lifetimes)	MIN	0.00	0.00
	MEAN	0.00	0.01
	MAX	0.24	2.94
C' (moderate lifetimes)	MIN	0.00	0.00
	MEAN	0.02	0.01
	MAX	1.46	1.59
D' (long lifetimes)	MIN	0.00	0.00
	MEAN	0.00	0.00
	MAX	0.20	0.21

Table 2.2: Percentage deviation from optimal performance of heuristic policies π^{SPI} and π^{DSPI} for Weibull lifetimes and deterministic service times when $J = 2$.

mation $V_{\pi^S}^{app}$ within (2.9). Hence we have

$$\pi^{SF}(\mathbf{n},t) = \arg\max_j \left\{ \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n},t,j,s) V_{\pi^S}^{app}(\mathbf{n}',t+s) dG_j(s) \right\}. \quad (2.10)$$

Remark 2.4. *For the determination of policy π^{SPI} via equation (2.9), we do not need to compute the quantity within the brackets for $j = \pi^S(\boldsymbol{n},t)$, because $V_{\pi^S}(\boldsymbol{n},t)$ should have already been computed. The computational effort is thus significantly reduced. However, this result does not carry over to the fluid heuristic π^{SF} that is determined by equation (2.10). We must compute the quantity in the brackets for every action $j \in A(\boldsymbol{n})$, including $\pi^S(\boldsymbol{n},t)$.*

The approximating policy value function $V_{\pi^S}^{app}$ is obtained by developing a suitable fluid (deterministic) analogue of the stochastic system emptying under policy π^S. In this approximating model the (random) number of jobs remaining is represented by a fluid level which diminishes at a suitable deterministic rate to reflect *both* service completions and losses from the system of unserved jobs under π^S.

2.2.1 Fluid Model: No Losses During Service

We proceed to discuss how to develop $V_{\pi^S}^{app}(\mathbf{n},t)$, an approximation to $V_{\pi^S}(\mathbf{n},t)$ based on a fluid model which drains fluid in a way which is appropriate given our assumption that jobs in service cannot experience premature loss.

Remark 2.5. *Fluid approximations have been widely used in the study of performance evaluation and optimal control of queueing systems. Almost all the literature approximates the original queueing system via a fluid limit that is obtained through appropriate scaling of system parameters. Contributions of this sort are covered at length in earlier works by Mandelbaum et al. [1998], Gajrat and Hordijk [2000], Whitt [2006], Decreusefond and Moyal [2006], and Bassamboo et al. [2006],*

among others. Fluid limits have also been used in the development of ADP algorithms. Moallemi et al. [2006] obtain valuable insights on system behaviour by means of an asymptotic analysis in a fluid limit. These analyses are then used to select basis functions in an ALP algorithm. Similarly, Veatch [2009] approximates the relative value functions in an ALP algorithm by quadratic value functions obtained from fluid models. Another contribution due to Chen et al. [2009] constructs basis functions by exploiting knowledge of associated fluid and diffusion approximations in a TD learning algorithm.

Distinct from previous literature which centres on asymptotic analyses in a limiting regime, the fluid approximation model proposed in this section has a very simple structure and does not require any parameter scaling. The solutions to the fluid model are immediate approximations to the value functions rather than just guidelines to basis function selection as in the previous contributions. These features are essential for the development of our efficient ADP algorithms.

We focus initially on the contribution to $V_{\pi^S}^{app}(\mathbf{n}, t)$ from a single job class. We drop the class identifier and use X, Y and $\theta(\cdot)$ for the class lifetime, service time (both assumed absolutely continuous in this account) and lifetime hazard respectively. Note that the lifetime hazard is given by $\theta(t) = F'(t)\{1 - F(t)\}^{-1}$ where F is the distribution function of X and $'$ denotes derivative. We also write $E(Y) = \mu^{-1}$.

We use the pair (m, s) to denote the fact that an amount of fluid (number of jobs) m is present when the processing of some class begins at time s. Because of the way in which π^S imposes static priorities among the classes, this class will be served continually from s until all of the corresponding fluid is drained. We use $N(m, s)$ for the number of services completed (which in the fluid model may be non-integer) during the processing of the class. It will then follow that $\mu^{-1}N(m, s)$ is the time taken to process the class under the fluid model.

The fluid is drained as follows: if $m \geq 1$ then a single unit of fluid is removed instantaneously (to signify the guaranteed service completion of one job) at time

s. Loss of fluid is then experienced at a rate determined by the hazard rate $\theta(\tau)$ during the time period $s < \tau \leq s + \mu^{-1}$. Note that this period is the time occupied by the processing of the first job in the fluid model. If the amount of fluid remaining at time $s + \mu^{-1}$ exceeds one then a further single unit of fluid is removed instantaneously at time $s + \mu^{-1}$ and signifies the guaranteed service completion of a second job. Loss of fluid is then expected at a rate determined by the hazard rate $\theta(\tau)$ during the period $s + \mu^{-1} < \tau \leq s + 2\mu^{-1}$, and so on. If we write $R(\tau)$ for the amount of fluid remaining at time τ, we have for $\tau \geq s$

$$\begin{aligned} R'(\tau) &= -\theta(\tau)R(\tau), \tau \neq s + k\mu^{-1}, k \in \mathbb{N}, \\ R(\{s + k\mu^{-1}\}^+) &= \max\{R(s + k\mu^{-1}) - 1, 0\}, k \in \mathbb{N}, \\ R(s) &= m. \end{aligned} \tag{2.11}$$

If we define $k(m, s)$ by

$$k(m, s) = \min\{k; R(\{s + k\mu^{-1}\}^+) = 0\} \tag{2.12}$$

we then have

$$N(m, s) = k(m, s) + R(s + k(m, s)\mu^{-1}). \tag{2.13}$$

Note from (2.12) that $k(m, s)$ is the (integer) number of fully completed jobs under the fluid model while $R(s + k(m, s)\mu^{-1})$ is a fractional amount of fluid remaining after those completions and is deemed to yield a further fractional completion within the approximating fluid model.

The fluid model is illustrated in Figure 2.1.

In fact, the system (2.11) is straightforward to solve explicitly. In order to state the solution with a minimum of notation we develop the sequence

$$m_r(s) = \begin{cases} 1, r = 0 \\ 1 + \sum_{u=0}^{r-1} \exp\left\{\int_0^{(u+1)\mu^{-1}} \theta(s + v)dv\right\}, r \in \mathbb{Z}^+. \end{cases}$$

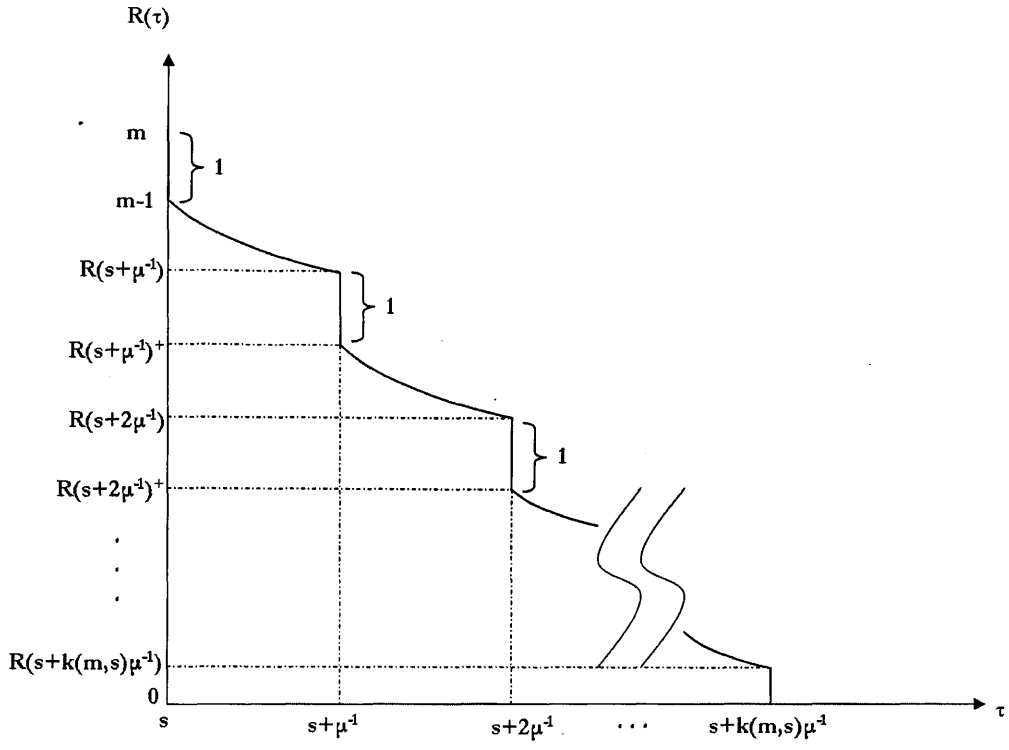

Figure 2.1: Fluid Model.

The quantity $m_r(s)$ may be understood as the amount of fluid at time s to achieve $r+1$ service completions. Please note that the assumption that $E(X) < \infty$ implies that the hazard rate θ has an infinite integral over $\mathbb{R}^+$ and hence that $m_r(s) \to \infty, r \to \infty$ for all choices of s.

Proposition 1. (a) If $m_{r-1}(s) < m \leq m_r(s)$ for some $r \in \mathbb{Z}^+$ then

$$N(m,s) = r + \{m - m_{r-1}(s)\} \exp\left\{ - \int_0^{r\mu^{-1}} \theta(s+v)dv \right\}; \tag{2.14}$$

(b) If $m \leq m_0(s) = 1$ then

$$N(m,s) = m.$$

Proof. From (2.11), we have that when $R(\{s + k\mu^{-1}\}^+) > 0$ it then follows that

$$R'(\tau) = -\theta(\tau)R(\tau), s + k\mu^{-1} < \tau < s + (k+1)\mu^{-1},$$

and hence that

$$R(s+(k+1)\mu^{-1}) = R(\{s+k\mu^{-1}\}^{+}) \exp\left\{-\int_{k\mu^{-1}}^{(k+1)\mu^{-1}} \theta(s+v)dv\right\}. \quad (2.15)$$

We now develop the sequence $\{\hat{R}(k), k \in \mathbb{N}\}$ as follows:

$$\hat{R}(0) = m,$$
$$\hat{R}(k+1) = \{\hat{R}(k)-1\} \exp\left\{-\int_{k\mu^{-1}}^{(k+1)\mu^{-1}} \theta(s+v)dv\right\}, k \in \mathbb{N}. \quad (2.16)$$

Substituting m into the sequence we have for $k \in \mathbb{N}$ that

$$\hat{R}(k) = m \exp\left\{-\int_{0}^{k\mu^{-1}} \theta(s+v)dv\right\} - \sum_{u=0}^{k-1} \exp\left\{-\int_{u\mu^{-1}}^{k\mu^{-1}} \theta(s+v)dv\right\}. \quad (2.17)$$

In (2.17) and elsewhere we use the convention that an empty sum is zero. From (2.11) and (2.15) it is straightforward that if $\hat{R}(l) > 0, 0 \leq l \leq k$, then

$$\hat{R}(k) = R(s+k\mu^{-1}). \quad (2.18)$$

Now consider m in the range $m_{r-1}(s) < m \leq m_r(s)$ where $r \in \mathbb{Z}^{+}$. We write m in the form

$$m = 1 + \sum_{u=0}^{r-2} \exp\left\{\int_{0}^{(u+1)\mu^{-1}} \theta(s+v)dv\right\} + \gamma \exp\left\{\int_{0}^{r\mu^{-1}} \theta(s+v)dv\right\} \quad (2.19)$$

where $\gamma \in (0,1]$. It is straightforward from (2.17),(2.19) and an induction argument that $\hat{R}(k)$ decreases as k increases from 0 to r with

$$\hat{R}(k) = 1 + \sum_{u=k}^{r-2} \exp\left\{\int_{k\mu^{-1}}^{(u+1)\mu^{-1}} \theta(s+v)dv\right\} + \gamma \exp\left\{\int_{k\mu^{-1}}^{r\mu^{-1}} \theta(s+v)dv\right\}, 0 \leq k \leq r-1. \quad (2.20)$$

It now follows from (2.18) and (2.20) that

$$\hat{R}(k) = R(s+k\mu^{-1}), 0 \leq k \leq r-1,$$

and in particular that

$$\hat{R}(r-1) = R(s+(r-1)\mu^{-1}) = 1+\gamma \exp\left\{\int_{(r-1)\mu^{-1}}^{r\mu^{-1}} \theta(s+v)dv\right\}. \qquad (2.21)$$

It follows from (2.16), (2.18) and (2.21) that

$$\hat{R}(r) = R(s+r\mu^{-1}) = \gamma. \qquad (2.22)$$

From (2.11)-(2.13) we now infer that

$$\begin{aligned} N(m,s) &= r+\gamma \\ &= r+\{m-m_{r-1}(s)\}\exp\left\{-\int_0^{r\mu^{-1}} \theta(s+v)dv\right\}. \end{aligned} \qquad (2.23)$$

Equation (2.23) is recovered from (2.19) by solving for γ and using the expression for $m_r(s)$ in the paper. This completes the proof of Proposition 1(a). Proposition 1(b) follows trivially from the definition of the quantities concerned. □

In order to obtain $V_{\pi^S}^{app}(\mathbf{n},t)$, we need to restore the class identifier to the notation and write $N_j(m_j,s_j)$ for the above fluid approximation for the number of class j services completed from an initial state (m_j,s_j). We now suppose that the classes are numbered according to their ordering by π^S, with class 1 processed first and class J last.

For fixed system state $(\mathbf{n},t)$ and $1 \le j \le J$, we inductively develop the quantities $\nu_j(\mathbf{n},t)$ which record the number of class j services completed under the fluid model when static policy π^S is applied from this state, as follows:

$$\nu_1(\mathbf{n},t) = N_1(n_1,t),$$

and

$$\nu_j(\mathbf{n},t) = N_j\left(n_j \exp\left\{-\int_0^{\sum_{i=1}^{j-1}\mu_i^{-1}\nu_i(\mathbf{n},t)} \theta_j(t+v)dv\right\},\right.$$
$$\left. t + \sum_{i=1}^{j-1}\mu_i^{-1}\nu_i(\mathbf{n},t)\right), 1 \le j \le J. \qquad (2.24)$$

The first argument of N_j on the right hand side of (2.24) is the number of class j jobs present when the processing of that class begins. The original number n_j (present at t) is diminished by losses occurring over the time period $[t, t + \sum_{i=1}^{j-1}\mu_i^{-1}\nu_i(\mathbf{n},t))$ during which the first $j-1$ classes are processed. We now use the quantities in (2.24) to develop the needed approximating policy value function as

$$V_{\pi^S}^{app}(\mathbf{n},t) = \sum_{j=1}^{J}\nu_j(\mathbf{n},t).$$

Dynamic heuristic π^{SF} is then developed from (2.10).

Remark 2.6. *It is straightforward to establish from the expression on the right hand side of (2.14), together with the preceding expressions for the $m_r(s)$, that the quantity $N(m,s)$, regarded as a function of m only (fixed s) is continuous, increasing, piecewise linear and concave. It will then follow that the derived approximating value $V_{\pi^S}^{app}(\boldsymbol{n},t)$ is increasing and concave componentwise in $\boldsymbol{n}$, for fixed t. This is exemplified in Figure 2.2 below where values of $V_{\pi^S}^{app}(n_1,n_2,0)$ are plotted for a two class problem whose details are as in the following example.*

Example 2.1. *Consider a two class example in which the lifetimes X_j are i.i.d. Weibull with hazard rate given by*

$$\theta_j(t) = \alpha_j\beta_j^{-\alpha_j}t^{\alpha_j-1}, t \in \mathbb{R}^+, j = 1,2. \qquad (2.25)$$

In this illustrative example we set the parameter values to be $\alpha_1 = 1.06, \beta_1 = 56.77, \alpha_2 = 1.81, \beta_1 = 81.22$. We further assume that the service times are de-

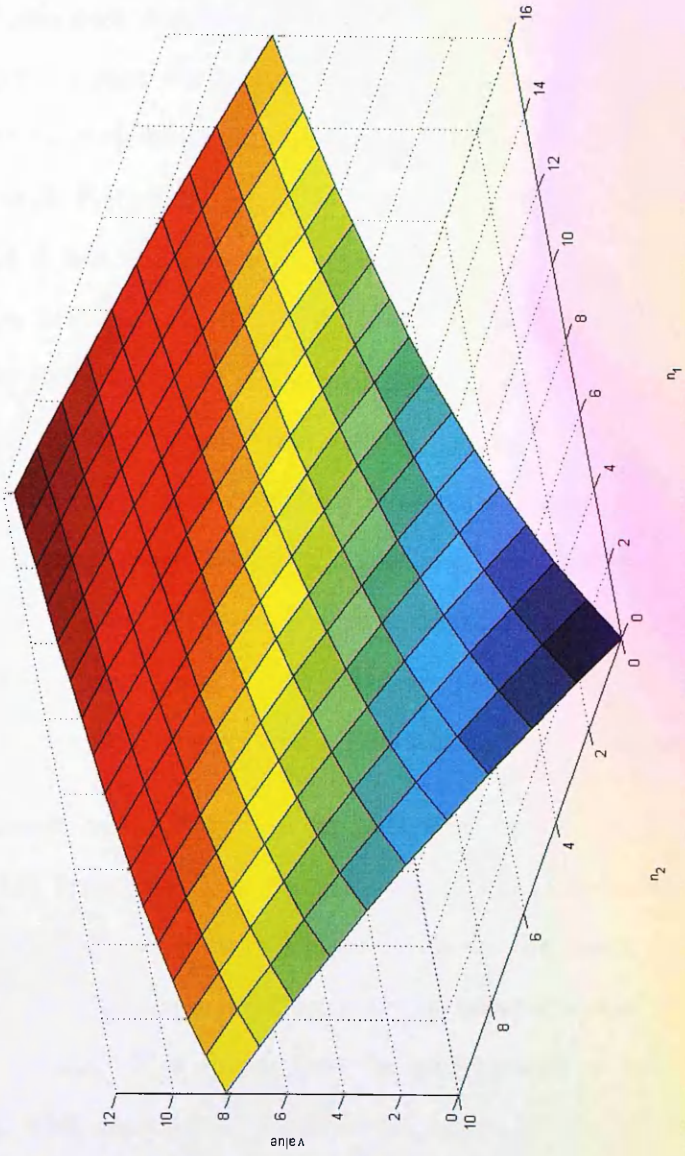

Figure 2.2: Values of $V_{\pi^S}^{app}(n_1, n_2, 0)$ where $0 \leq n_1 \leq 16, 0 \leq n_2 \leq 10$.

terministic with rates $\mu_1 = 0.11, \mu_2 = 0.13$. *In Figure 2.3 find a summary of the decisions (which job class to process) taken by dynamic heuristic* π^{SF} *at time points 0, 18.15, 30.85 and 45.39, should decisions be required then. These time points are chosen to be representative of likely decision times. Indeed, the three positive time points are the decision epochs when two class 1 jobs, four class 2 jobs, and five class 1 jobs have just finished service, respectively.*

In Figure 2.3 each filled circle indicates a decision in favour of class 1, and each diamond a decision in favour of class 2. Please note that $\theta_1(t) \geq \theta_2(t), 0 \leq t \leq 66.02$, *while* $\theta_2(t)/\theta_1(t)$ *increases with t (refer to Figure 2.4). Hence at time* $t = 0$, *class 1 jobs will appear more urgent and in most states this is reflected in a decision to process this class. As time increases all residual lifetimes of jobs decrease, but those of class 2 jobs decrease more rapidly. Hence decisions taken by* π^{SF} *increasingly favour class 2 jobs as time elapses.*

We would like to point out that in the top right diagram π^{SF} *switches back and forth between class 1 and 2 with* n_2 *increasing, for fixed* $n_1 = 9$ *or* 10. *This is also found in the top left diagram when* $n_1 = 16$. *This kind of pattern is due to approximation errors. even though these are fairly small. We have checked the optimal policy for this example and it does not have such behaviour.*

Even though we do not have an analytical bound on the accuracy of the proposed fluid approximation, we can check its performance numerically. For the above example, we plot in Figure 2.5 both the exact value function V_{π^S} and its fluid approximation $V_{\pi^S}^{app}$ for a set of selected states at four representative decision times. It is shown that the performance of the approximation is outstanding, with consistently small errors across all the states examined. We conjecture that it also works well for the other states. Indeed, the results summarised in Table 2.3 support our conjecture. The percentage approximation errors, $\Delta(V_{\pi^S}, V_{\pi^S}^{app}) = 100|1 - V_{\pi^S}^{app}/V_{\pi^S}|\%$, have a small average of 1.27% and a worst performance of just 3.34% over 19,354 points in the effective state space. Such a space excludes all the states which will never be visited given the initial condition.

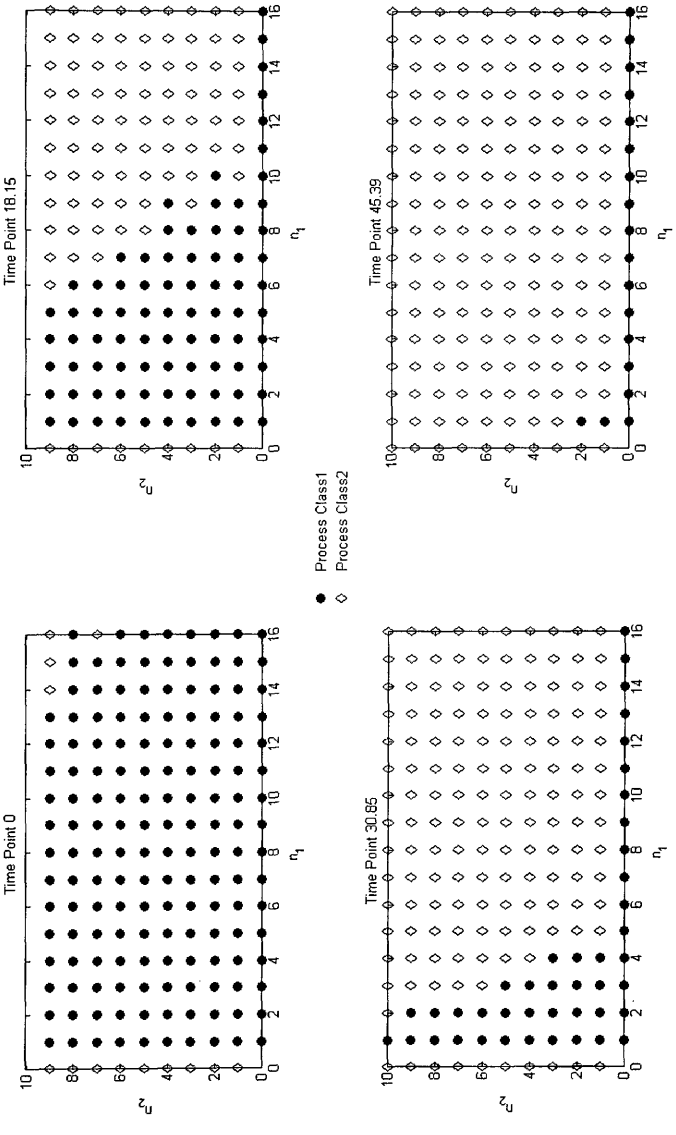

Figure 2.3: Decisions taken by policy π^{SF} in states (n_1, n_2, t) where $0 \leq n_1 \leq 16, 0 \leq n_2 \leq 10$ and t=0, 18.15, 30.85, 45.39.

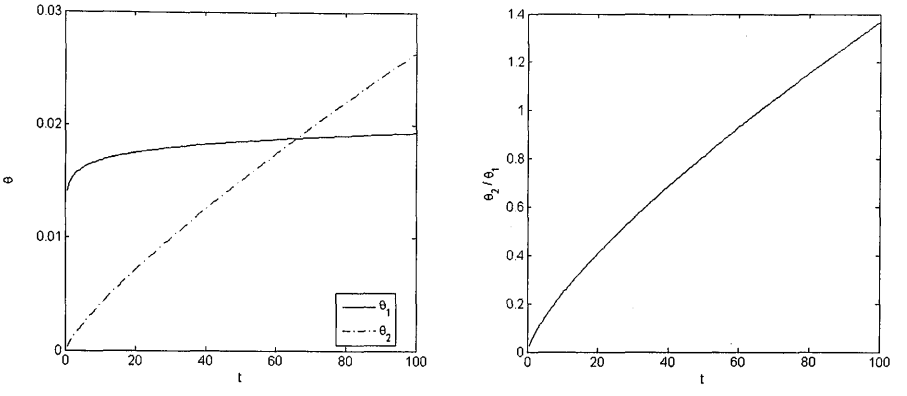

Figure 2.4: Hazard Rates.

For this example, a state (n_1, n_2, t) is in the effective sample space if

1. t can be expressed in terms of

$$t = \frac{r_1}{\mu_1} + \frac{r_2}{\mu_2},$$

for some r_1, r_2 which are non-negative integers and satisfy $0 \leq r_1 \leq 16, 0 \leq r_2 \leq 10$;

2. and the following condition holds:

$$t \leq \frac{16 - n_1}{\mu_1} + \frac{10 - n_2}{\mu_2}.$$

MEAN	1.27%
MIN	0.00%
1ST QUARTILE	1.01%
MEDIAN	1.35%
3RD QUARTILE	1.61%
MAX	3.34%

Table 2.3: Percentage approximation errors $\Delta(V_{\pi^S}, V_{\pi^S}^{app})$ for Example 2.1.

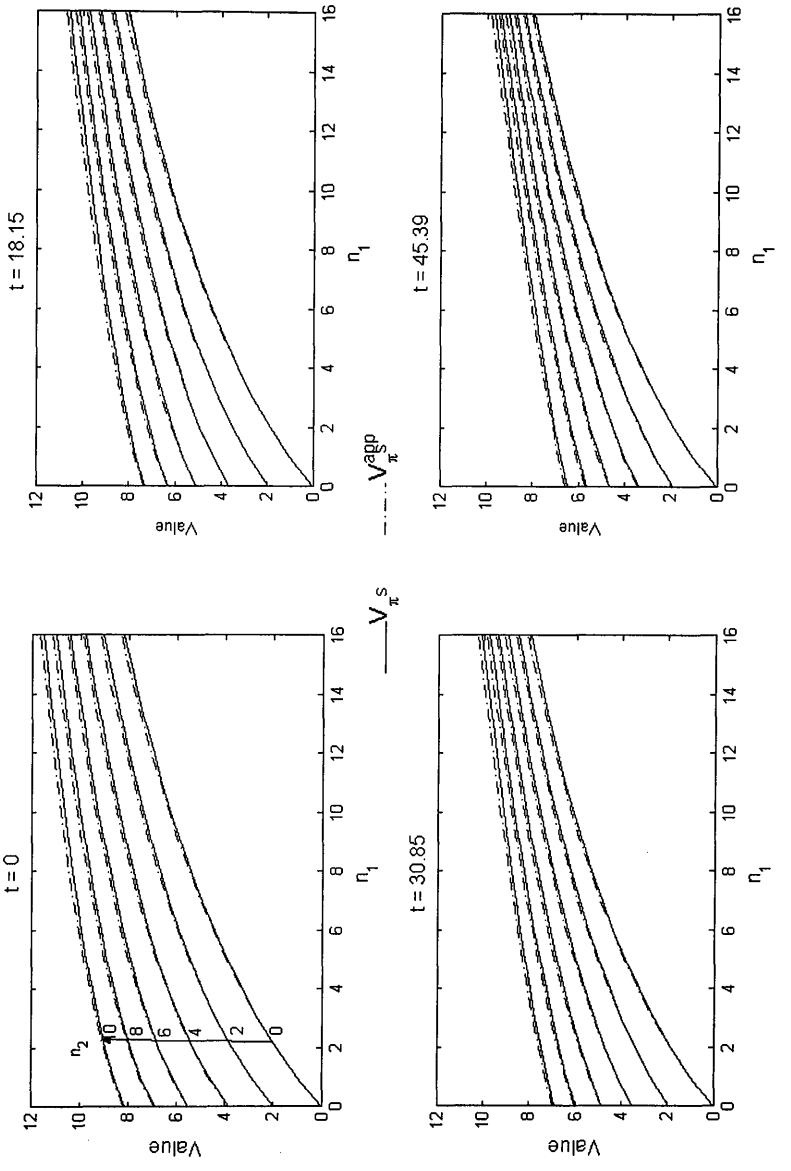

Figure 2.5: Exact value functions V_{π^S} verses the fluid approximations $V_{\pi^S}^{app}$ in states (n_1, n_2, t) where $0 \leq n_1 \leq 16, n_2 = 0, 2, 4, 6, 8, 10$ and t=0, 18.15, 30.85, 45.39.

2.2.2 Fluid Model: Losses During Service

We now give a brief account of how the above discussion should be modified for the variant of our model in which losses are allowed during service. See (2.7) above. It is clear that our initial policy π^S will continue to be asymptotically optimal in the sense discussed in Glazebrook et al. [2004] for this variant of the basic model discussed above. We write $\bar{V}_{\pi^S} : \Omega \to \left[0, \sum_{j=1}^{J} L_j\right]$ for the policy value function for π^S now given by

$$\begin{aligned}\bar{V}_{\pi^S}(\mathbf{n}, t) &= P\left\{X_{\pi^S(\mathbf{n},t)} > t + Y_{\pi^S(\mathbf{n},t)} | X_{\pi^S(\mathbf{n},t)} > t\right\} \\ &\quad + \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, \pi^S(\mathbf{n}, t), s)\bar{V}_{\pi^S}(\mathbf{n}', t+s) dG_{\pi^S(\mathbf{n},t)}(s), \\ \bar{V}_{\pi^S}(\mathbf{0}, t) &= 0. \end{aligned} \tag{2.26}$$

Direct computation of $\bar{V}_{\pi^S}$ from (2.26) is computationally prohibitive other than for small problems and special cases. Hence we again deploy a fluid approximation to develop an approximating value function $\bar{V}_{\pi^S}^{app} : \Omega \to \left[0, \sum_{j=1}^{J} L_j\right]$. The dynamic heuristic which results is given by

$$\begin{aligned}\bar{\pi}^{SF}(\mathbf{n}, t) = \arg\max_j \Bigg\{ & P(X_j > t + Y_j | X_j > t) \\ & + \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, j, s)\bar{V}_{\pi^S}^{app}(\mathbf{n}', t+s) dG_j(s) \Bigg\}. \end{aligned} \tag{2.27}$$

Consider now a single class with X, Y, and θ as in subsection (2.2.1) above. Under the fluid model suppose that π^S begins processing the class when in state (m, s). If we write $\bar{R}(\tau)$ for the amount of fluid remaining at time τ, we have that for $\tau \geq s$

$$\begin{aligned}\bar{R}'(\tau) &= -\theta(\tau)\bar{R}(\tau) - \mu \mathbb{I}\{\bar{R}(\tau) > 0\}, \\ \bar{R}(s) &= m. \end{aligned} \tag{2.28}$$

Hence, according to (2.28) fluid is now drained continuously under the impact of

both losses from the system and service effort. If we define $\bar{t}(m,s)$ by

$$\bar{t}(m,s) = \inf\{\tau - s; \tau \geq s \text{ and } \bar{R}(\tau) = 0\}$$

then it is not difficult to show from (2.28) that $\bar{t}(m,s)$ satisfies the integral equation

$$\int_0^{\bar{t}(m,s)} \left\{ \exp\left[\int_0^u \theta(s+v)dv \right] \right\} du = m\mu^{-1}, \tag{2.29}$$

which has the solution

$$\bar{t}(m,s) = \theta^{-1} \ln(1 + m\theta\mu^{-1})$$

in the special exponential lifetime case in which the hazard rate is constant, namely $\theta(\cdot) \equiv \theta$. Since $\bar{t}(m,s)$ is the time for the fluid to be drained, the corresponding number of service completions is given by

$$\bar{N}(m,s) = \mu\bar{t}(m,s).$$

To obtain $\bar{V}_{\pi^S}^{app}(\mathbf{n},t)$ we develop quantities $\bar{\nu}_j(\mathbf{n},t), 1 \leq j \leq J$, inductively as follows:

$$\bar{\nu}_1(\mathbf{n},t) = \bar{N}_1(n_1,t),$$

and

$$\bar{\nu}_j(\mathbf{n},t) = \bar{N}_j \left(n_j \exp\left\{ - \int_0^{\sum_{i=1}^{j-1} \mu_i^{-1} \bar{\nu}_i(\mathbf{n},t)} \theta_j(t+v)dv \right\}, \right.$$
$$\left. t + \sum_{i=1}^{j-1} \mu_i^{-1} \bar{\nu}_i(\mathbf{n},t) \right), 1 \leq j \leq J. \tag{2.30}$$

Please note that in (2.30) the job classes are numbered in order of their processing by π^S. We now use the quantities in (2.30) to develop the approximating policy

value function as

$$\bar{V}_{\pi^S}^{app}(\mathbf{n}, t) = \sum_{j=1}^{J} \bar{\nu}_j(\mathbf{n}, t).$$

Dynamic heuristic $\bar{\pi}^{SF}$ is then developed from (2.27).

2.3 Numerical Study

In the following discussion we consider three general scenarios. These are (I) lifetimes and service times are both exponentially distributed, (II) Weibull lifetimes and deterministic service times, and (III) Weibull lifetimes and exponential service times. As we shall see, our problem formulations for scenarios (I) and (II) have discrete and finite state spaces. In both cases it is possible (though expensive) to compute optimal policies for problems of modest size using dynamic programming by exploiting special features of the structure of the value iteration algorithms concerned. It is thus possible to assess the quality of heuristic π^{SF} by direct comparison of the expected number of service completions achieved with the optimum. For scenario (III), the state space is continuous and infinite. In these cases, it has not proved possible to develop optimal policies for problems of even modest size in reasonable time. In sharp contrast, it is a straightforward matter to perform the computations necessary to implement heuristic π^{SF} *on-line*; namely, to obtain those $\pi^{SF}(\mathbf{n}, t)$ which are required in any realisation of the system. Hence, in scenario (III) it is natural to assess the *relative performance* of π^{SF} and the competitor heuristics π^S (see (2.5)) and π^M (see (2.6)) by means of Monte Carlo simulation. In the numerical results we have also included the heuristic π^{SPI} for comparison when it has proved possible to do so.

2.3.1 Scenario (I): lifetimes and service times exponentially distributed

In this scenario, we have

$$F_j(s) = 1 - e^{-\theta_j s},$$
$$G_j(s) = 1 - e^{-\mu_j s},$$

and crucially, the lifetime hazard rates θ_j are all constant functions. Hence the time dependence in the policy value function $V_\pi(\mathbf{n}, t)$ for any stationary policy (i.e. one which makes decisions in any state $(\mathbf{n}, t)$ which depend upon $\mathbf{n}$ but not on t) disappears. In such cases, the optimality equation in (2.4) reduces to

$$V(\mathbf{n}) = 1 + \max_{j \in A(\mathbf{n})} \left\{ \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, j, s) V(\mathbf{n}') \mu_j e^{-\mu_j s} ds \right\}, \mathbf{n} \neq \mathbf{0},$$
$$V(\mathbf{0}) = 0, \tag{2.31}$$

and the transition probability reduces from (2.3) to

$$p(\mathbf{n}'|\mathbf{n}, j, s) = \prod_{i=1}^{J} \binom{n_i - \delta_{ij}}{n_i'} e^{-\theta_i s n_i'} \left(1 - e^{-\theta_i s}\right)^{n_i - \delta_{ij} - n_i'}, \tag{2.32}$$
$$0 \leq n_j' \leq n_j - \delta_{ij}, 1 \leq i \leq J.$$

The policy π^{SF} can be obtained from the expression

$$\pi^{SF}(\mathbf{n}) = \arg\max_j \left\{ \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, j, s) V_{\pi^S}^{app}(\mathbf{n}') \mu_j e^{-\mu_j s} ds \right\}. \tag{2.33}$$

Equation (2.31) enables the development of optimal policies in this case along with the value $V(\mathbf{L})$ which is the expected number of service completions achieved from initial state $\mathbf{L}$. It is a straightforward matter to check that in this case heuristics $\pi^{SPI}, \pi^{SF}, \pi^S$ and π^M are all stationary (have no explicit time dependence). Here, the appropriate value iteration algorithm for the computation of the num-

ber of expected services achieved under any stationary policy π is developed from recursions of the form

$$V_\pi(\mathbf{n}) = 1 + \int_0^\infty \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, \pi(\mathbf{n}), s) V_\pi(\mathbf{n}') \mu_j e^{-\mu_j s} ds, \mathbf{n} \neq \mathbf{0},$$
$$V_\pi(\mathbf{0}) = 0. \tag{2.34}$$

Please note that the integrals in equations (2.31), (2.33) and (2.34) need to be estimated by numerical approximation. In this thesis we employ the Matlab numerical integration toolbox for this purpose. The function being used is *quadl*, which is based on a recursive adaptive Lobatto quadrature. To use this function we need to specify an integration interval of the service time s. The lower bound is obviously zero. The upperbound must be a finite positive real number rather than infinity as shown in these equations. Proper selection of the upperbound is critical to the computation outcome. A value which is too small will lead to incorrect integration results, while a too big value will waste a significant amount of computational time. Experiments show that the value which covers 99.9% of all the service time possibilities finds the balance. For exponential service times, this value is $3\mu^{-1} \ln 10$.

It is worth mentioning that because the lifetime distributions are exponential and memoryless in this scenario, the quantity (2.6) that determines policy π^M takes a very simple closed form, as shown below:

$$\mu_j^{-1} \left[\sum_{i=1}^{J} (n_i - \delta_{ij}) \theta_i \right].$$

As we shall see, however, this quantity is much more difficult to compute when lifetimes are Weibull distributed. In the next two scenarios, numerical methods are used instead.

Problems are generated at random for two class ($J = 2$) and five class ($J = 5$) cases for each of four distinct assumptions (A, B, C, D) about the relative lengths

of lifetimes and service times. We sample the key problem features $\mu_j^{-1}, \theta_j^{-1}, L_j$ as follows:

$$\mu_j^{-1} \sim U[1, 10] \quad \text{(all cases)}; \tag{2.35a}$$

$$\theta_j^{-1}\mu_j|\mu_j^{-1} \sim U[0.1, 0.5] \quad \text{(very short lifetimes, } A\text{)}; \tag{2.35b}$$

$$\theta_j^{-1}\mu_j|\mu_j^{-1} \sim U[0.5, 2.0] \quad \text{(short lifetimes, } B\text{)}; \tag{2.35c}$$

$$\theta_j^{-1}\mu_j|\mu_j^{-1} \sim U[2.0, 10.0] \quad \text{(moderate lifetimes, } C\text{)}; \tag{2.35d}$$

$$\theta_j^{-1}\mu_j|\mu_j^{-1} \sim U[10.0, 100.0] \quad \text{(long lifetimes, } D\text{)}; \tag{2.35e}$$

$$L_j \sim DU[1, 50] \quad (J = 2 \text{ cases}); \tag{2.35f}$$

$$L_j \sim DU[1, 6] \quad (J = 5 \text{ cases}). \tag{2.35g}$$

In (2.35a)-(2.35g), $U[a, b]$ is a continuous uniform distribution on the range $[a, b]$ while $DU[a, b]$ is a discrete uniform distribution on $[a, b]$. Note that in category A, (2.35b) indicates that the ratio of mean lifetime to mean service time lies in the range $[0.1, 0.5]$ and hence few service completions are likely. In category D this ratio lies in the range $[10.0, 100.0]$ and hence many service completions are likely. We observe that the myopic nature of heuristic π^M suggests that it is likely to perform well in category A where the current decision in any state need take little account of the future. Further, the asymptotic optimality of π^S in a "no premature job loss" limit suggests that it should perform well in category D.

Note in this section and the numerical study elsewhere, that we deliberately avoid those special problem instances where the jobs with smaller mean lifetimes also have smaller mean service times. In these cases the optimal policy is obvious and all the three heuristics can easily find the optimum. This may disguise the differences in performance between them.

For each of $J = 2$ and $J = 5$ and each of the categories A, B, C and D, 500 problems were generated at random according to (2.35a)-(2.35g). For each problem value iteration was deployed to compute the mean number of service completions achieved under the heuristics π^{SPI}, π^{SF}, π^S and π^M and under an

optimal policy. In every problem the percentage suboptimality $\Delta(\pi, opt)$ of each heuristic $\pi = \pi^{SPI}, \pi^{SF}, \pi^{S}, \pi^{M}$ was computed. Further, for each collection of 500 problems, the minimum, mean and maximum values of $\Delta(\pi, opt)$ were recorded for each heuristic. These values may be found in Table 2.4(a) ($J = 2$) and Table 2.5(a) ($J = 5$).

From Table 2.4(a), and as indicated above, the performance of the asymptotically optimal heuristic π^{S} improves steadily from category A to category D while for myopic heuristic π^{M} the reverse is the case. Serious suboptimalities can occur especially in those problem instances for which these heuristics are not designed. The position is similar in Table 2.5(a) though the fact that in the $J = 5$ cases the values of the generated L_j are much smaller (see (2.35g)) means that the maximum suboptimalities for the myopic heuristic are substantially reduced. In sharp contrast, the heuristic π^{SF} is robust; it performs well in all scenarios. It outperforms π^{S} and π^{M} in all cases with the single exception of category $D, J = 5$ where the asymptotic optimality of π^{S} confers on the latter a slight advantage. As reported in Section 2.2, π^{SPI} performs outstandingly well and may be readily computed in exponential cases of modest size.

We also calculated, for each of the categories and $J = 2$ and $J = 5$, the number of problem instances in which each heuristics $\pi = \pi^{SF}, \pi^{S}, \pi^{M}$ provides the best performance. Moreover, a Friedman test was conducted to test if the differences across the heuristics are significant. These results can be found in Table 2.4(b) ($J = 2$) and Table 2.5(b) ($J = 5$). It is worth mentioning that here and elsewhere, the total number of winners exceeds the number of problems. This is because of ties in performance between policies. It is shown in Table 2.4(b) that π^{SF} provides the best performance in many more instances than the other two when $J = 2$. The only exception is category D where π^{S} wins in 24 more instances. The result is similar in Table 2.5(b). π^{S} wins again in category D. In category B the myopic policy π^{M} provides the best performance in 33 more instances than π^{SF}, which could be due to the smaller values of L_j when $J = 5$. In either table, the p-values

Category		$\Delta(\pi^{SPI}, opt)$	$\Delta(\pi^{SF}, opt)$	$\Delta(\pi^{S}, opt)$	$\Delta(\pi^{M}, opt)$
	MIN	0.00	0.00	0.00	0.00
A	MEAN	0.00	0.01	3.01	0.02
(very short lifetimes)	MAX	0.00	0.55	23.79	1.94
	MIN	0.00	0.00	0.00	0.00
B	MEAN	0.00	0.02	2.52	0.14
(short lifetimes)	MAX	0.00	0.18	24.33	4.29
	MIN	0.00	0.00	0.00	0.00
C	MEAN	0.00	0.00	1.23	0.90
(moderate lifetimes)	MAX	0.00	0.12	16.52	9.32
	MIN	0.00	0.00	0.00	0.00
D	MEAN	0.00	0.02	0.12	2.23
(long lifetimes)	MAX	0.00	0.71	5.17	18.86

(a) Percentage deviation from optimal performance.

	π^{SF}	π^{S}	π^{M}	Friedman's p-value
A	472	241	332	0.000
B	388	262	332	0.000
C	465	327	258	0.000
D	417	441	232	0.000

(b) Number of win instances out of a total of 500, with p-values for Friedman test of difference.

Table 2.4: Numerical study results for exponential lifetimes and service times when $J = 2$.

Category		$\Delta(\pi^{SPI}, opt)$	$\Delta(\pi^{SF}, opt)$	$\Delta(\pi^{S}, opt)$	$\Delta(\pi^{M}, opt)$
	MIN	0.00	0.00	0.00	0.00
A	MEAN	0.00	0.01	1.36	0.04
(very short lifetimes)	MAX	0.48	1.10	15.95	2.42
	MIN	0.00	0.00	0.00	0.00
B	MEAN	0.01	0.02	0.84	0.03
(short lifetimes)	MAX	0.88	1.02	11.86	1.80
	MIN	0.00	0.00	0.00	0.00
C	MEAN	0.01	0.02	0.13	0.34
(moderate lifetimes)	MAX	1.04	1.27	3.87	4.39
	MIN	0.00	0.00	0.00	0.00
D	MEAN	0.00	0.02	0.00	0.49
(long lifetimes)	MAX	0.05	0.46	0.33	4.36

(a) Percentage deviation from optimal performance.

	π^{SF}	π^{S}	π^{M}	Friedman's p-value
A	475	198	265	0.000
B	348	265	381	0.000
C	441	369	252	0.000
D	361	471	169	0.000

(b) Number of win instances out of a total of 500, with p-values for Friedman test of difference.

Table 2.5: Numerical study results for exponential lifetimes and service times when $J = 5$.

of Friedman test are all zeroes, which means that we have very strong evidence of differences in performance between the heuristics.

It is worthy of mention that the scale of computational effort required in the analysis necessitated the use of high performance computing (HPC) facilities. The most computationally intensive tasks include development of the optimal policy, development of policy π^{SPI}, and computation of the mean number of service completions achieved under each heuristic. In contrast, development of heuristic π^{SF} takes only a small percentage of the total computation time. It is trivial to develop π^{S} and π^{M}. To appreciate the computing challenge of this numerical analysis, we can estimate the time which would have been required if all the 4,000 problem instances were solved on a single PC. Experiments show that the average run time is 12 hours for a single problem on my desktop, which has a 3.00GHz CPU and 1GB RAM. The total time needed would be 2,000 days, or 5 years and a half. In contrast, HPC facilities in Lancaster University have the capability to solve 60 problems in parallel, with an average run time of 2 hours per problem. The total time required is then 5 days and a half. Similar results also hold for the next two scenarios.

2.3.2 Scenario (II): Weibull lifetimes and deterministic service times

The Weibull family of distributions yields a flexible way of modelling lifetimes. Note that here and elsewhere, we shall use Weibull(α_j, β_j) to denote the distribution function

$$F_j(s) = 1 - e^{-(s/\beta_j)^{\alpha_j}}, \tag{2.36}$$

where α_j is the shape parameter and β_j is the scale parameter for class j. This reduces to the exponential distribution when $\alpha_j = 1$. The mean of the Weibull random variable is $\beta_j \Gamma(1 + \alpha_j^{-1})$. The hazard rate takes the form given in (2.25).

In order for the class j hazard to be increasing, which is a natural assumption in many applications, we require that $\alpha_j > 1$.

In cases in which class j service times are deterministic of value μ_j^{-1} the value iteration procedure need only compute value functions V, V_π at states $(\mathbf{n}, t)$ for t-values of the form $t = \sum_{j=1}^{J} m_j \mu_j^{-1}$ where the m_j are non-negative integers. The optimality equation (2.4) now takes the form

$$V(\mathbf{n}, t) = 1 + \max_{j \in A(\mathbf{n})} \left\{ \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, j, \mu_j^{-1}) V(\mathbf{n}', t + \mu_j^{-1}) \right\}, \mathbf{n} \neq \mathbf{0},$$

$$V(\mathbf{0}, t) = 0, \tag{2.37}$$

and the transition probability (2.3) becomes

$$p(\mathbf{n}'|\mathbf{n}, t, j, \mu_j^{-1}) = \prod_{i=1}^{J} \binom{n_i - \delta_{ij}}{n_i'} \left(1 - F_i^t(\mu_j^{-1})\right)^{n_i'} \left(F_i^t(\mu_j^{-1})\right)^{n_i - \delta_{ij} - n_i'}, \tag{2.38}$$

in which the remaining lifetime distribution $F_j^t(s)$ is given by

$$F_j^t(s) = 1 - \exp\left[\left(\frac{t}{\beta_j}\right)^{\alpha_j} - \left(\frac{t+s}{\beta_j}\right)^{\alpha_j}\right]. \tag{2.39}$$

Also, we have

$$\pi^{SF}(\mathbf{n}, t) = \arg\max_j \left\{ \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, j, \mu_j^{-1}) V_{\pi^S}^{app}(\mathbf{n}', t + \mu_j^{-1}) \right\}, \tag{2.40}$$

and

$$V_\pi(\mathbf{n}, t) = 1 + \left\{ \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, \pi(\mathbf{n}, t), \mu_{\pi(\mathbf{n},t)}^{-1}) V_\pi(\mathbf{n}', t + \mu_{\pi(\mathbf{n},t)}^{-1}) \right\}, \mathbf{n} \neq \mathbf{0},$$

$$V_\pi(\mathbf{0}, t) = 0. \tag{2.41}$$

The effective discretisation of the time axis which results radically simplifies value iteration procedures.

We now give the method to compute the quantity (2.6), which has been copied

over here for the reader's convenience:

$$E(Y_j)\left[\sum_{i=1}^{J}(n_i-\delta_{ij})\{E(X_i-t|X_i>t)\}^{-1}\right].$$

In the above expression, the first term $E(Y_j)$ is as simple as in the previous scenario. The tricky part is to compute $E(X_i-t|X_i>t)$, which is referred to as the *mean residual lifetime* (MRL) in the literature. Recall that in the previous scenario MRL reduces to $E(X)$ or θ^{-1} due to the memoryless property of exponential lifetimes. For Weibull lifetimes that are defined by (2.36), $E(X_i-t|X_i>t)$, or $MRL_i(t)$, can be calculated by the following equation. For presentation purpose, we discard the subscript i in the following account.

$$E(X-t|X>t)=MRL(t)=\frac{\beta\Gamma(1/\alpha,(t/\beta)^{\alpha})}{\alpha}e^{(t/\beta)^{\alpha}}, \tag{2.42}$$

where $\Gamma(1/\alpha,(t/\beta)^{\alpha})$ is the upper incomplete Gamma function which takes the form

$$\Gamma(1/\alpha,(t/\beta)^{\alpha})=\int_{(t/\beta)^{\alpha}}^{\infty}x^{1/\alpha-1}e^{-x}dx. \tag{2.43}$$

Proof. By definition

$$MRL(t)=\frac{\int_t^{\infty}(1-F(s))ds}{1-F(t)}.$$

Substituting $F(\cdot)$ by equation (2.36), we have

$$MRL(t)=\frac{\int_t^{\infty}e^{-(s/\beta)^{\alpha}}ds}{e^{-(t/\beta)^{\alpha}}}. \tag{2.44}$$

Now, expand the upper incomplete gamma function in (2.42) by (2.43), we have

$$MRL(t)=\frac{\beta\int_{(t/\beta)^{\alpha}}^{\infty}(x^{1/\alpha-1}e^{-x}dx)}{\alpha}e^{(t/\beta)^{\alpha}}. \tag{2.45}$$

Let $x = (s/\beta)^\alpha$, we have

$$dx = \alpha s^{\alpha-1}/\beta^\alpha ds,$$
$$x^{1/\alpha-1} = (\beta/s)^{\alpha-1}. \tag{2.46}$$

Substituting these values back into equation (2.45), we have the resulting equation exactly as in (2.44). □

The calculation seems quite straightforward via equation (2.42). The only difficulty is that C++ does not have a pre-defined incomplete Gamma function in its standard libraries. Luckily, such a function is provided in the Boost C++ libraries, which are a collection of free libraries that extend the functionality of C++.

Unfortunately, during the analysis we have found that the equation (2.42) is not numerically robust. When time t is large, the incomplete Gamma function approaches zero while on the contrary the exponential term approaches infinity. The result is a NaN (Not a Number). To get around this issue, we opt instead to compute MRL by the following equation:

$$MRL(t) = \int_0^\infty s\, dF^t(s)$$
$$MRL(t) = \int_0^\infty s\, dF^t(s),$$

where $F^t(s)$ is the remaining lifetime distribution function given in (2.39). Therefore, we have

$$MRL(t) = \int_0^\infty s\frac{\alpha}{\beta}\left(\frac{t+s}{\beta}\right)^{\alpha-1} \exp\left[\left(\frac{t}{\beta}\right)^\alpha - \left(\frac{t+s}{\beta}\right)^\alpha\right] ds. \tag{2.47}$$

There is no closed form solution to the integration. Again, it is computed numerically by Matlab function *quadl*. The upperbound for the integral is determined as

$$(t^\alpha + 7\beta^\alpha \ln 10)^{1/\alpha} - t, \tag{2.48}$$

which covers 99.99999% of all the remaining lifetime possibilities.

In our numerical study, problems are generated at random for two class ($J = 2$) and five class problems ($J = 5$) under each of four distinct assumptions (A', B', C', D') about the relative lengths of lifetimes and service times. We sample the key problem features $\mu_j^{-1}, \alpha_j, \beta_j, L_j$ as follows:

$$\mu_j^{-1} \sim U[1, 10] \quad \text{(all cases)}; \tag{2.49a}$$

$$\alpha_j \sim U[1.0, 2.0] \quad \text{(all cases)}; \tag{2.49b}$$

$$\beta_j \Gamma(1 + \alpha_j^{-1})\mu_j | \mu_j^{-1}, \alpha_j \sim U[0.1, 0.5] \quad \text{(very short lifetimes, } A'\text{)}; \tag{2.49c}$$

$$\beta_j \Gamma(1 + \alpha_j^{-1})\mu_j | \mu_j^{-1}, \alpha_j \sim U[0.5, 2.0] \quad \text{(short lifetimes, } B'\text{)}; \tag{2.49d}$$

$$\beta_j \Gamma(1 + \alpha_j^{-1})\mu_j | \mu_j^{-1}, \alpha_j \sim U[2.0, 10.0] \quad \text{(moderate lifetimes, } C'\text{)}; \tag{2.49e}$$

$$\beta_j \Gamma(1 + \alpha_j^{-1})\mu_j | \mu_j^{-1}, \alpha_j \sim U[10.0, 100.0] \quad \text{(long lifetimes, } D'\text{)}; \tag{2.49f}$$

$$L_j \sim DU[1, 20] \quad (J = 2 \text{ cases}); \tag{2.49g}$$

$$L_j \sim DU[1, 5] \quad (J = 5 \text{ cases}). \tag{2.49h}$$

Note that values of β_j are derived from the values of μ_j^{-1} and α_j obtained from the draws in (2.49a) and (2.49b) and the value of $\beta_j \Gamma(1 + \alpha_j^{-1})$ obtained from whichever is appropriate of the draws in (2.49c)-(2.49f). Note also that the values of the L_j drawn from (2.49g)-(2.49h) will tend to be smaller than those in (2.35f)-(2.35g). This is forced upon us by the added complexity of the recursions (2.37), (2.40) and (2.41) in comparison with (2.31), (2.33) and (2.34) . As with scenario (I) comparisons between the heuristics are based on 500 problems randomly generated as in (2.49a)-(2.49h) above for each of $J = 2$ and $J = 5$ and each of the categories A', B', C' and D'. Note that development of policy π^{SPI} is possible for the $J = 2$ cases, though computationally expensive. Hence this heuristic has been included in the $J = 2$ study but not in the $J = 5$ study since it was not possible to get results for the latter in reasonable computational time. The results are presented in Table 2.6 and Table 2.7.

The evidence provided by Tables 2.6(a) and 2.7(a) yields similar conclusions

Category		$\Delta(\pi^{SPI}, opt)$	$\Delta(\pi^{SF}, opt)$	$\Delta(\pi^{S}, opt)$	$\Delta(\pi^{M}, opt)$
	MIN	0.00	0.00	0.00	0.00
A'	MEAN	0.00	0.00	4.27	0.06
(very short lifetimes)	MAX	0.00	0.00	36.34	2.80
	MIN	0.00	0.00	0.00	0.00
B'	MEAN	0.00	0.01	2.53	0.20
(short lifetimes)	MAX	0.24	0.94	26.16	6.14
	MIN	0.00	0.00	0.00	0.00
C'	MEAN	0.02	0.03	0.71	1.30
(moderate lifetimes)	MAX	1.46	1.81	20.21	16.29
	MIN	0.00	0.00	0.00	0.00
D'	MEAN	0.00	0.00	0.22	2.31
(long lifetimes)	MAX	0.20	0.38	4.13	21.36

(a) Percentage deviation from optimal performance.

	π^{SF}	π^{S}	π^{M}	Friedman's p-value
A'	500	206	414	0.000
B'	472	200	284	0.000
C'	470	310	219	0.000
D'	497	385	172	0.000

(b) Number of win instances out of a total of 500, with p-values for Friedman test of difference.

Table 2.6: Numerical study results for Weibull lifetimes and deterministic service times when $J = 2$.

Category		$\Delta(\pi^{SF}, opt)$	$\Delta(\pi^{S}, opt)$	$\Delta(\pi^{M}, opt)$
	MIN	0.00	0.00	0.00
A'	MEAN	0.00	2.63	0.11
(very short lifetimes)	MAX	0.02	33.42	5.39
	MIN	0.00	0.00	0.00
B'	MEAN	0.02	0.88	0.07
(short lifetimes)	MAX	1.79	10.57	2.13
	MIN	0.00	0.00	0.00
C'	MEAN	0.02	0.17	0.65
(moderate lifetimes)	MAX	0.84	4.53	6.25
	MIN	0.00	0.00	0.00
D'	MEAN	0.03	0.22	0.67
(long lifetimes)	MAX	0.63	2.48	5.66

(a) Percentage deviation from optimal performance.

	π^{SF}	π^{S}	π^{M}	Friedman's p-value
A'	500	162	274	0.000
B'	458	165	219	0.000
C'	458	264	151	0.000
D'	481	137	76	0.000

(b) Number of win instances out of a total of 500, with p-values for Friedman test of difference.

Table 2.7: Numerical study results for Weibull lifetimes and deterministic service times when $J = 5$.

to those drawn from Tables 2.4(a) and 2.5(a). The heuristics π^S and π^M continue to have poor worst case performance in settings for which they were not designed. The uniform excellence of the performance of π^{SF} is in clear contrast. From Tables 2.6(b) and 2.7(b), it is clear that π^{SF} wins easily in all cases. It provides the best performance in nearly all the problem instances, especially category A in which it is the best for every single instance. The zero p-values again indicate strong evidence of differences in performance among the heuristics.

2.3.3 Scenario (III): Weibull lifetimes and exponential service times

In both scenarios (I) and (II) it was possible to exploit model features to develop exact analyses based on DP value iteration for problems of modest size. In this way, it was possible (though expensive) to develop optimal policies and calculate the expected number of service completions for the heuristics of interest and for the optimal policy. This is no longer possible in scenario (III).

As the service times are now continuous random variables, the decision epochs after time zero could be any positive real values in the set $\mathbb{R}^+$. The resulting state space is continuous and infinite. It is impossible to implement direct DP methods for such problems. The continuous state space must be discretized. This can be done by defining a small enough time increment δ and a large enough termination time point T_{max}. We assume that after T_{max} no more customers will be sent for service and no more rewards will be received. Let $E = \{0, \delta, 2\delta, \cdots, k\delta, \cdots, K\delta\}$ be the set of all time points on the discretized time axis, where $K\delta$ is the last one within T_{max} and $K = \lfloor T_{max}/\delta \rfloor$. For any reasonable discretisation, the value of K is huge and the size of the state space, $(K+1)\prod_1^J L_j$, is far beyond tractability even for cases with a modest number of initial jobs. Development of the optimal policy, or even the exact evaluation of any given policy, is not a realistic option.

Nonetheless, it is still straightforward to develop on-line applications of the

three heuristics π^{SF}, π^{S} and π^{M} (though not of π^{SPI}). By this we mean that it is straightforward to perform the computations required to determine the action prescribed by each heuristic in any given state. This is trivial for the heuristics π^{S} and π^{M}. We show now how to determine the action specified by the heuristic policy π^{SF} for any given state.

After discretisation, the service completions can only happen at the time points in set E, and the service time variables are transformed from exponential to their discrete counterpart, geometric. We allow customers to abandon the system at any time and the lifetime distributions remain unchanged, still Weibull. Let Y_j^d be a geometric random variable describing the discretized service time for class j jobs. The pmf (probability mass function) for Y_j^d is

$$\begin{aligned} \mathrm{P}\{Y_j^d = k\delta\} &= (1 - G_j(\delta))^{k-1} G_j(\delta) \\ &= e^{-\mu_j \delta(k-1)}(1 - e^{-\mu_j \delta}), 0 \leq k \leq K. \end{aligned} \tag{2.50}$$

Remember $G_j(\delta) = 1 - e^{-\mu_j \delta}$ is the probability that a service completes within the interval δ before discretization.

Denote by $E(t) = \{k_t\delta, (k_t + 1)\delta, \cdots, K\delta\}$ the set of time points lying within the interval $[t, T_{max}]$. Obviously we have $k_t = \lceil t/\delta \rceil$ and in particular $k_0 = 0$ and $E(0) = E$. We then have

$$\pi^{SF}(\mathbf{n}, t) = \arg\max_j \left\{ \sum_{s \in E(t)} \mathrm{P}(Y_j^d = s) \sum_{\mathbf{n}'} p(\mathbf{n}'|\mathbf{n}, t, j, s) V_{\pi^S}^{app}(\mathbf{n}', t + s) \right\}, \tag{2.51}$$

where the transition probability $p(\mathbf{n}'|\mathbf{n}, t, j, s)$ takes the same form as in (2.38).

To find a proper T_{max} value, we firstly define by $\mathbf{Y}(\mathbf{L})$ a random variable for the cumulative service times of all the initial jobs, written as

$$\mathbf{Y}(\mathbf{L}) = \sum_{j=1}^{J} \sum_{k=1}^{L_j} Y_{jk}. \tag{2.52}$$

Then we let

$$\begin{aligned} T_{max} &= E(\mathbf{Y}(\mathbf{L})) + 3\sqrt{Var(\mathbf{Y}(\mathbf{L}))} \\ &= \sum_{j=1}^{J} E(Y_j)L_j + 3\sqrt{\sum_{j=1}^{J} Var(Y_j)L_j} \\ &= \sum_{j=1}^{J} \frac{L_j}{\mu_j} + 3\sqrt{\sum_{j=1}^{J} \frac{L_j}{\mu_j^2}}. \end{aligned} \tag{2.53}$$

The resulting T_{max} serves as a reasonable (actually rather conservative) termination point, by which most jobs (if not all) will have either been served or lost from the system. There is not a single best choice of the value of δ. When making the selection we need to consider the balance between the accuracy of the approximation and the computational effort required.

Even though the determination of the π^{SF} action by equation (2.51) still takes some time, especially at the early stage, it does not matter at all if this is only required for a few states. In light of this, a natural approach is then to use Monte Carlo simulation to conduct a comparative study of the performance of π^{SF}, π^{S} and π^{M}. For each simulation run, the computation of (2.51) is only needed for those states visited (not many at all) by the simulation trajectory. The computational time taken in the rest of the simulation is negligible.

In the exact approach, we do not need to differentiate individual jobs within one class as they are identically distributed. This is not the case any more in the simulation. Each job has a distinct sampled lifetime and a distinct sampled service time. An extra decision is then required to select, among the others, a specific job for service. This decision needs to be necessarily a random pick up process as otherwise the simulation will be biased from the exact approach. To simulate this, we assign each job a priority which is sampled uniformly between 0 and 1. If a class j job is to be served next, the choice is given to the remaining job that has the highest priority in that class.

The other two random variables, the lifetimes x_{jk} and the discretized service

times y^d_{jk}, are generated as below.

$$x_{jk} = \beta_j(-\ln U)^{1/\alpha_j},$$
$$y^d_{jk} = \left\lceil \frac{\ln U}{-\mu_j \delta} \right\rceil,$$

where U is a random variable uniformly distributed between 0 and 1 and is sampled using the Mersenne Twister random number generator (Matsumoto and Nishimura [1998]). There is plenty of open source code available which implements this algorithm. The one we use in this thesis is due to Fog [2010].

A key issue to be addressed in any simulation is variance reduction. In this thesis we use the method of common random numbers (or matched sampling) to reduce the variance, by sharing random numbers between the simulations for the heuristic policies π^S, π^M and π^{SF}. Expressed in detail, at the beginning of each replication, the lifetimes, service times and priorities are generated and stored for all jobs. Under the same experimental scenario, the system is then simulated repeatedly three times, one for each heuristic. The difference between the number of successful service completions achieved by π^{SF} and that by π^S or π^M is recorded as one output sample, the averages of which over all the replications are used to generate the final report.

For each problem instance, the simulation is replicated a large number of times until the variance of the samples falls below an acceptable tolerance. Denote the sample variance in our simulation by $s^2(\pi^S, \pi^{SF})$ and $s^2(\pi^M, \pi^{SF})$, respectively. The simulation stops after M replications where M is the smallest m which satisfies the following stopping criterion.

$$\max\left\{1.96\frac{s(\pi^S, \pi^{SF})}{\sqrt{m}}, 1.96\frac{s(\pi^M, \pi^{SF})}{\sqrt{m}}\right\} < d,$$

where d is the acceptable tolerance which is set to 0.001. To ensure that the simulation has a fairly large coverage of the sample space, we require that $M >$ 100. In other words, the simulation is replicated at least 100 times and then the

stopping criterion begins to apply. We also specify an upper limit for M so that the simulation does not run forever.

With the simulation progressing, the sample mean and sample variance are updated dynamically, whenever a new replication is done and new samples are available. We could keep a record of all the samples to date for the calculation, but they would consume a lot of memory. A better way is to update the sample mean and variance by the following on-line algorithm. We have used generic notations x_m, $\bar{x}_m$ and s_m^2 for the latest sample after the m^{th} replication, the latest sample mean and the latest sample variance, respectively. We have

$$\bar{x}_m = \bar{x}_{m-1} + \frac{x_m - \bar{x}_{m-1}}{m},$$
$$s_m^2 = (1 - \frac{1}{m-1})s_{m-1}^2 + m(\bar{x}_m - \bar{x}_{m-1})^2.$$

As we can see, there is no need to store any samples in this algorithm.

Problem parameters were chosen as in scenario (I) (service times) and scenario (II) (lifetimes) though time constraints limited the study to the $J = 2$ case, with 100 problems generated in each category. In Table 2.8(a) find information on the estimated values of $\Delta(\pi^S, \pi^{SF})$ and $\Delta(\pi^M, \pi^{SF})$, the percentage excess expected number of successes achieved by π^{SF} over π^S and π^M respectively. Hence positive values indicate stronger performance by π^{SF} while negative values indicate stronger performance by the competitor heuristic.

While we encountered occasional problem instances in which the estimated expected number of successes achieved by the competitor heuristic exceeded that of π^{SF} these were rare and the differences usually very small. In all categories the average performance of π^{SF} was superior. There continued to be problem instances in which π^S, π^M performed very poorly in comparison with π^{SF}.

In Table 2.8(b), the number of win instances for π^{SF} is many more than π^S and π^M in all the cases considered. The p-values of Friedman test are again zeroes throughout and indicate strong evidence of differences in performance among these

Category		$\Delta(\pi^S, \pi^{SF})$	$\Delta(\pi^M, \pi^{SF})$
	MIN	-0.57	-0.57
A'	MEAN	2.18	0.07
(very short lifetimes)	MAX	14.47	1.16
	MIN	-0.60	-2.07
B'	MEAN	1.43	0.15
(short lifetimes)	MAX	19.63	8.86
	MIN	-0.41	-0.56
C'	MEAN	0.25	1.02
(moderate lifetimes)	MAX	5.74	10.47
	MIN	-0.01	0.00
D'	MEAN	0.16	0.94
(long lifetimes)	MAX	3.39	8.24

(a) Percentage difference between the performance of heuristics π^S, π^M and that of heuristic π^{SF}.

	π^{SF}	π^S	π^M	Friedman's p-value
A'	92	49	70	0.000
B'	77	56	74	0.000
C'	91	74	55	0.000
D'	97	79	50	0.000

(b) Number of win instances out of a total of 100, with p-values for Friedman test of difference.

Table 2.8: Numerical study results for Weibull lifetimes and exponential service times when $J = 2$.

heuristics.

2.3.4 Implementation Notes

In this section we put together a selection of important procedures that have been implemented in this thesis to deal with the very intensive computational requirements.

1. The numerical study in this chapter and in the next two is implemented in the C++ programming language for its high efficiency in scientific computation. The only exception is some numerical integrations which are calculated by the Matlab numerical toolbox. To import Matlab functions into C++, they are compiled into wrapper files for C++ by invoking the Matlab command *mcc*, and then the generated files are included in C++ as resources files.

2. For each of the three scenarios, we group all the analysis processes into one *class*. For the second and third scenario, since most of the data and functions are similar, only the class for the second scenario is created from scratch and the one for the third scenario is just a derived class. This procedure greatly improves code efficiency.

3. The binomial coefficients $\binom{n}{k}$ are used very frequently and repeatedly in all problem instances. Even though a single computation of it is momentary, millions of times becomes very significant. To remove this unnecessary computational effort, we save the pre-calculated binomial coefficients into a disk file and load it into memory when the program is initialized. The data stored in the file take the form of a matrix, where the columns are n and the rows are k. Because $\binom{n}{k} = \binom{n}{n-k}$, the matrix is actually lower triangular. We have chosen $0 \leq n \leq 50$, of which the resulting matrix is enough for all of our numerical analyses in this thesis.

4. Similarly, some intermediate quantities are used repeatedly, such as one step transition probabilities or MRL. Instead of computing them repeatedly when-

ever needed, they are just computed once and then stored into memory for future usage.

5. In our C++ implementation, each state is defined as an object of a state class. For Weibull lifetime cases, the state space is huge. As a result the objects created will consume a large amount of memory. To save more memory for the actual computation, we implement state in a different way. We create an object for each $\mathbf{n}$ rather than for each $(\mathbf{n}, t)$, and in each of these objects, declare an array for all possible decision epochs. We have found this technique works very well for efficient memory allocation.

6. A parameter data file is loaded to memory in the initialization. This file contains a set of parameters which change frequently from one problem instance to another, such as the number of job classes, or the distribution of lifetimes and service times. This approach allows a single compilation for all problem instances. Therefore, we do not need to re-compile the program every time when a change of parameters occurs. Only the parameter values need to be updated in the data file.

2.4 Conclusion

A batch of impatient jobs is present at time zero in a single server clearing system. Before any service starts they are subject to a perfect triage process and are placed into distinct classes. Jobs placed into one class are assumed to have i.i.d. lifetimes and i.i.d. service times. The objective is to schedule the service to maximize the total number of successful service completions.

We model this problem as a SMDP and hence standard DP approaches can be applied to develop optimal policies. However, any problem of practical size cannot be solved to exact optimality due to the curse of dimensionality. Instead, we propose to generate effective heuristic policies by a single policy improvement

step from a static permutation heuristic π^S. Moreover, the value functions of π^S are approximated by a deterministic fluid model to deal with the intractability of exact policy evaluation. The constructed fluid model not only has a simple form that permits fast solution, but also its quality of approximation is outstanding.

Our proposed heuristic is tested in an extensive numerical study in three scenarios, namely exponential lifetimes and service times, Weibull lifetimes and deterministic service times, and Weibull lifetimes and exponential service times. In the first two scenarios we are able to compute the exact optimum for small problems and thus to compare competing heuristics by means of suboptimality. In the third scenario it is not possible to develop optimal policies in reasonable time. Instead, a simulation based comparative study is conducted. In all three scenarios, our heuristic works robustly well. It comfortably outperforms the other two heuristics proposed in the literature (one is π^S), both of which can exhibit poor performance outside of the domains for which they were designed.

Chapter 3

Scheduling of Impatient Jobs with Imperfect Classification

In this chapter we extend the work of Chapter 2 to impatient job scheduling with imperfect classification. It is organized into four major topics. In Section 3.1 our job scheduling problem is formulated as a Bayes sequential decision problem and in Section 3.2 an exact approach to its solution via dynamic programming is described. In Section 3.3, the approximate DP methodology and the fluid model proposed in Chapter 2 are further developed so that they can yield effective solutions to our Bayesian model. A numerical study in Section 3.4 testifies to the strong performance of the resulting heuristic scheduling policy. It is concluded in Section 3.5.

3.1 The Model

A clearing system has a single server and a collection of N jobs (or customers) awaiting service, which starts at time 0. Each job is one of J types, each type being identified by an integer $i \in \{1, 2, \ldots, J\}$. Each type i job has associated with it two positive-valued random variables (r.v.s). One of these is its lifetime, namely the period during which the job is available for service, which is deemed to have the distribution of some r.v. X_i with distribution function F_i. A job will

leave the system unserved if its lifetime expires before it is taken into service. The other job-related r.v. is its service time, which, for a type i job, is deemed to have the distribution of some r.v. Y_i with distribution function G_i. A job leaves the system when its service is complete. The lifetimes and service times of all jobs form a mutually independent collection. We assume that service is nonpreemptive. Up to this point the model duplicates the one in the previous chapter.

At time 0 all jobs are subject to an error-prone triage and thus the type of each job is observed with error. Should a type i job be assessed as type j, we shall say that it becomes a member of *class* j. Hence throughout, we shall use the term *class* to denote the *assessed type* of a job. Only job class is observed. We shall adopt the following simple probabilistic model of job (mis)classification. We write p_i for the (unconditional) probability that a job is of type i and ϵ_{ij} for the (conditional) probability that a type i job is assessed as j. If $\epsilon_{ii} < 1$ for any i then some misclassification is possible. By deployment of Bayes' Theorem we infer the conditional probability that, in advance of any service, a class j job is actually of type i to be

$$p_{ij}(0) \triangleq P\left(type\ i \mid class\ j\right) = \frac{\epsilon_{ij} p_i}{\sum_{k=1}^{J} \epsilon_{kj} p_k}. \tag{3.1}$$

We shall call $\{p_{ij}(0), 1 \leq i \leq J\}$ the *prior distribution* for each job classified as j. This summarises the decision maker's beliefs about its true identity before service begins.

As time passes, jobs leave the system as services are completed and lifetimes expire. It is also true, that at some time $t > 0$ our beliefs with regard to the (true) type of the remaining jobs need to be updated in light of their survival beyond t. Again applying Bayes' Theorem, we compute *posterior distributions* $\{p_{ij}(t), 1 \leq i \leq J\}$ for each class j job as follows,

$$p_{ij}\left(t\right) \triangleq P\left(type\ i \mid class\ j, lifetime > t\right) = \frac{\epsilon_{ij} p_i \left\{1 - F_i\left(t\right)\right\}}{\sum_{k=1}^{J} \epsilon_{kj} p_k \left\{1 - F_k\left(t\right)\right\}}. \tag{3.2}$$

Proof. Define the following events.

- A_i: a job is of type i. Generally define event A_k: a job is of type $k, 1 \leq k \leq J$. It is obvious that all these events are mutually exclusive.
- B: a job is assessed as class j.
- C: a job has survived up to time t.

It is not difficult to derive the Bayes' formula for three events.

$$P(A_i|B \cap C) = \frac{P(B \cap C|A_i)P(A_i)}{P(B \cap C)} = \frac{P(A_i)P(B|A_i)P(C|A_i \cap B)}{P(B \cap C)}.$$

By definition event C is independent of event B given any event A_k, as a job's lifetime distribution is solely determined by its type. We then have

$$P(A_i|B \cap C) = \frac{P(A_i)P(B|A_i)P(C|A_i)}{\sum_{k=1}^{J} P(B|A_k)P(C|A_k)P(A_k)}.$$

Substituting all the probabilities by their definitions gives the equation (3.2) immediately. □

Please note that equation (3.2) is not numerically robust for large t. When time t is large enough, the survival probability for all jobs will usually approach zero. This results in a zero numerator and a zero denominator, and thus a NaN issue. We fix this by multiplying both expressions by $\{1 - F_i(t)\}^{-1}$. In the Weibull distributed lifetime scenario, the rearranged equation is,

$$\begin{aligned} p_{ij}(t) &= \frac{\epsilon_{ij} p_i}{\sum_{k=1}^{J} \epsilon_{kj} p_k \exp\left[(t/\beta_i)^{\alpha_i} - (t/\beta_k)^{\alpha_k}\right]} \\ &= \frac{\epsilon_{ij} p_i}{\epsilon_{ij} p_i + \sum_{k=1, k\neq i}^{J} \epsilon_{kj} p_k \exp\left[(t/\beta_i)^{\alpha_i} - (t/\beta_k)^{\alpha_k}\right]}. \end{aligned} \tag{3.3}$$

It is trivial to show that if all of the $\epsilon_{..}, p_.$ are strictly positive and, further, that

there exists some type i^* which *outlasts the others* in the sense that

$$\lim_{t\to\infty} \frac{1-F_i(t)}{1-F_{i^*}(t)} = 0, i \neq i^*, \tag{3.4}$$

then we must have

$$\lim_{t\to\infty} p_{i^*j}(t) = 1, \forall j,$$

and consequently

$$\lim_{t\to\infty} p_{ij}(t) = 0, i \neq i^*, \forall j.$$

Hence survival information can be very informative for (true) type identity.

Example 3.1. *Consider an example with $J = 2$ and with the following parameters:*

- $p_1 = 0.3, p_2 = 0.7$,
- $\epsilon_{12} = 0.3, \epsilon_{21} = 0.4$,
- $X_1 \sim Weibull(1.68, 1.62)$, $X_2 \sim Weibull(1.16, 13.43)$.

At time zero, the probability that a class 1(2) job is indeed of type 1(2) is 0.43(0.82) from equation (3.1). As time passes these conditional probabilities are updated according to equation (3.2). Their values at times 2 and 5 are given below:

$$p_{11}(2) = 0.17, p_{22}(2) = 0.95;$$
$$p_{11}(5) = 0.00, p_{22}(5) = 1.00.$$

Since type 2 jobs have much longer mean lifetimes (with mean 12.75) than type 1 (mean 1.45), it is unsurprising that we have $i^* = 2$ here and that type 2 jobs outlast type 1. As time passes, all surviving jobs are increasingly likely to be of type 2.

The goal of analysis is the development of an approach to the allocation of service to surviving jobs (i.e., those still in the system) to maximise the expected

number of jobs which leave the system served. Equivalently, we seek to minimise the expected number of jobs which leave the system on the occasion of the expiry of their lifetimes. Decision epochs occur whenever a service completion occurs. Suppose that t is such an epoch and that at time t, n_j class j jobs survive. We write $(\mathbf{n}, t)$ for the corresponding system state, where $\mathbf{n} \triangleq (n_1, n_2, \ldots, n_J)$ is the vector summarising the class membership of surviving jobs. Write

$$A(\mathbf{n}) \triangleq \{j; n_j > 0\}$$

for the set of admissible actions in state $(\mathbf{n}, t)$. A service policy π maps each system state $(\mathbf{n}, t)$ to the action set $A(\mathbf{n})$. We shall conventionally use $(\mathbf{L},0)$ for the initial system state, where $\mathbf{L} = \{L_j, 1 \leq j \leq J\}$. Hence at time 0, L_j jobs are placed in class $j, 1 \leq j \leq J$.

With initial system state $(\mathbf{L},0)$ we shall use $\mathbf{L}_{..} \equiv \{L_{ij}, 1 \leq i, j \leq J\}$ to denote the *unobservable true state*, where L_{ij} denotes the number of type i jobs classified initially as j. Since all job classes are determined independently, we have $\sum_{i=1}^{J} L_{ij} = L_j, 1 \leq j \leq J$, with

$$\{L_{1j}, L_{2j}, \ldots L_{Jj}\} \mid \mathbf{L} \sim Multinomial\left(L_j; p_{1j}(0), p_{2j}(0), \ldots, p_{Jj}(0)\right), 1 \leq j \leq J. \tag{3.5}$$

Conditionally upon the true state $(\mathbf{L}_{..}, 0)$ we use a triple ijk to label the jobs, where ijk denotes the kth type i job to be classified as j. The range of k is $1 \leq k \leq L_{ij}$. Denoting the lifetime of ijk by $X_{ijk} \sim F_i$, we use $T_{ijk}(\pi)$ to denote the time at which service policy π begins to process job ijk. Conditional upon the true state $(\mathbf{L}_{..}, 0)$ we write $N(\pi \mid \mathbf{L}_{..}, 0)$ for the number of jobs to be served to completion under policy π. We have

$$N(\pi \mid \mathbf{L}_{..}, 0) = \sum_{i=1}^{J} \sum_{j=1}^{J} \sum_{k=1}^{L_{ij}} \mathbb{I}\{T_{ijk}(\pi) \leq X_{ijk}\}, \tag{3.6}$$

where $\mathbb{I}$ is an indicator. The goal of analysis is the determination of a service policy

π to maximise the quantity

$$V_\pi^e(\mathbf{L},0) = E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[E\left\{N\left(\pi \mid \mathbf{L}_{..},0\right)\right\}\right]. \tag{3.7}$$

In (3.7), the inner expectation is taken over realisations of the system, evolving under policy π from initial true state $\mathbf{L}_{..}$. The outer expectation is taken with respect to the conditional distribution for $\mathbf{L}_{..} \mid (\mathbf{L},0)$ whose marginal distributions are given in (3.5). We call $V_\pi^e(\mathbf{L},0)$ the *Bayes' return* generated by policy π from initial state $(\mathbf{L},0)$. Please note that in (3.7) and in what follows we shall use a superscript e to denote the fact that we are dealing with an object associated with a classification which is prone to error (and hence uncertainty).

At this point, we want to make a distinction between our problem and partially observable Markov Decision Processes. POMDPs are differentiated from standard MDPs only in that the state of the system is not directly observable. In our problem, not only is the true state $\mathbf{L}_{..}$ unobservable, but so are "true" actions as the type of the job chosen by an action is unknown. Therefore, the problem considered in this chapter is actually more difficult than standard POMDP problems.

Remark 3.1. *This model generalizes the job scheduling problem with perfect classification that we have studied in Chapter 2. To see this, consider a no error limit where* $\epsilon_{ij} \to 0, \forall i \neq j$. *For any non-negative time point, we have by equation (3.1) and (3.2)*

$$p_{ij}(t) = 0, \forall i \neq j,$$
$$p_{jj}(t) = 1,$$

and by equation (3.5)

$$L_{ij} = 0, \forall i \neq j,$$
$$L_{jj} = L_j.$$

The jobs assessed as class j are indeed of type j and the observed initial state is exactly the true state.

3.2 Formulation of the Bayes Sequential Decision Problem as a Dynamic Program

We now formulate the Bayes sequential decision problem

$$V^e(\mathbf{L},0) \equiv \sup_{\pi} V_\pi^e(\mathbf{L},0) \tag{3.8}$$

as a dynamic program (DP). Any policy achieving the supremum in (3.8) is a *Bayes' policy.* In order to formulate the associated DP we require additional notation. We shall use X_j^e to denote the random lifetime of a job classified as j at time 0. Using (3.1) above, the associated distribution function is given by

$$F_j^e(s) = \sum_{i=1}^{J} p_{ij}(0) F_i(s), s \in \mathbb{R}^+, 1 \leq j \leq J, \tag{3.9}$$

and the corresponding survival function by

$$\begin{aligned} \bar{F}_j^e(s) &= 1 - F_j^e(s) \\ &= \sum_{i=1}^{J} p_{ij}(0) \bar{F}_i(s), s \in \mathbb{R}^+, 1 \leq j \leq J, \end{aligned} \tag{3.10}$$

where $\bar{F}_i(s) = 1 - F_i(s)$ is the survival function for type i lifetimes.

We shall assume that each F_i has an associated absolutely continuous density f_i. Hence X_j^e has the density

$$f_j^e(s) = \sum_{i=1}^{J} p_{ij}(0) f_i(s), s \in \mathbb{R}^+, 1 \leq j \leq J, \tag{3.11}$$

and the continuous hazard

$$\theta_j^e(s) = f_j^e(s)\left\{1 - F_j^e(s)\right\}^{-1} = \sum_{i=1}^{J} p_{ij}(s)\,\theta_i(s), s \in \mathbb{R}^+, 1 \le j \le J, \tag{3.12}$$

where in (3.12), $\theta_i = f_i\{1 - F_i\}^{-1}$ denotes the continuous hazard associated with the type i lifetime X_i.

Proof of (3.12). From (3.9), (3.11), and the definition of hazard rates we have

$$\theta_j^e(s) = \frac{\sum_{i=1}^{J} p_{ij}(0) f_i(s)}{\sum_{i=1}^{J} p_{ij}(0)(1 - F_i(s))}. \tag{3.13}$$

Substitute $p_{ij}(0)$ by equation (3.1), and cancel the common term $\sum_{k=1}^{J} \epsilon_{kj} p_k$, then we have

$$\theta_j^e(s) = \frac{\sum_{i=1}^{J} \epsilon_{ij} p_i (1 - F_i(s))(f_i(s)/(1 - F_i(s)))}{\sum_{i=1}^{J} \epsilon_{ij} p_i (1 - F_i(s))}. \tag{3.14}$$

By changing the index from i to k in the denominator, the equation above can be transformed to

$$\theta_j^e(s) = \sum_{i=1}^{J} \frac{\epsilon_{ij} p_i (1 - F_i(s))}{\sum_{k=1}^{J} \epsilon_{kj} p_k (1 - F_k(s))} \frac{f_i(s)}{1 - F_i(s)}.$$

Note that the first multiplier within the summation is just $p_{ij}(s)$ (See equation (3.2)), and the second multiplier is the hazard rate θ_i for type i. The result then follows trivially. □

Should type i^* outlast the others in the sense of (3.4) above then we will have

$$\lim_{s \to \infty} \left\{\theta_j^e(s) - \theta_{i^*}(s)\right\} = 0, 1 \le j \le J.$$

Concerning job service times, consider a situation in which a job, originally classified as j, is still in the system and is scheduled for service at time t. We shall use

$Y^e_{j,t}$ for the corresponding service time, whose distribution function is given by

$$G^e_{j,t}(s) = \sum_{i=1}^{J} p_{ij}(t)\, G_i(s), s \in \mathbb{R}^+, 1 \leq j \leq J, \tag{3.15}$$

and the mean value is given by

$$E(Y^e_{j,t}) = \sum_{i=1}^{J} p_{ij}(t) E(Y_i). \tag{3.16}$$

Please note that service time distributions are now time dependent. In the important special case that the true service times Y_i are deterministic and distinct (in which case we use S_i for the type i service time), we have

$$P\left\{Y^e_{j,t} = S_i\right\} = p_{ij}(t), t \in \mathbb{R}^+, 1 \leq i, j \leq J. \tag{3.17}$$

Now let t be a decision epoch for the problem and let $(\mathbf{n},t)$ be the system state then. If action $j \in A(\mathbf{n})$ is taken and results in a service time (realised value of $Y^e_{j,t}$) equal to s then the system state at the next decision epoch will be $(\mathbf{n}',t+s)$ with probability $p^e(\mathbf{n}' \mid \mathbf{n},t,j,s)$ given by

$$\begin{aligned}
&p^e(\mathbf{n}' \mid \mathbf{n},t,j,s) \\
&= \prod_{m=1}^{J} \binom{n_m - \delta_{mj}}{n'_m} \left\{P\left[X^e_m \geq t+s \mid X^e_m > t\right]\right\}^{n'_m} \left\{P\left[X^e_m < t+s \mid X^e_m > t\right]\right\}^{n_m - \delta_{mj} - n'_m} \\
&= \prod_{m=1}^{J} \binom{n_m - \delta_{mj}}{n'_m} \left\{\frac{1 - F^e_m(t+s)}{1 - F^e_m(t)}\right\}^{n'_m} \left\{\frac{F^e_m(t+s) - F^e_m(t)}{1 - F^e_m(t)}\right\}^{n_m - \delta_{mj} - n'_m},
\end{aligned}$$

$$0 \leq n'_m \leq n_m - \delta_{mj}, 1 \leq m \leq J. \tag{3.18}$$

In (3.18), δ_{mj} is the Kronecker delta which is equal to one when $m = j$ and is zero otherwise. We can re-express the transition probability in (3.18) using the hazard

in (3.12). We have

$$p^e(\mathbf{n}' \mid \mathbf{n},t,j,s)$$

$$= \prod_{m=1}^{J} \binom{n_m - \delta_{mj}}{n'_m} \left\{ \exp\left(-\int_t^{t+s} \theta_m^e(\tau)\, d\tau \right) \right\}^{n'_m} \left\{ 1 - \exp\left(-\int_t^{t+s} \theta_m^e(\tau)\, d\tau \right) \right\}^{n_m - \delta_{mj} - n'_m},$$

$$0 \leq n'_m \leq n_m - \delta_{mj}, 1 \leq m \leq J. \tag{3.19}$$

In order to formulate the optimality equation for our Bayes sequential decision problem, we require the value function $V^e : \Omega \rightarrow \left[0, \sum_{j=1}^{J} L_j\right]$, where Ω is the system's state space, given by

$$\Omega = \left\{ (\mathbf{n},t); 0 \leq n_j \leq L_j, t \in \mathbb{R}^+ \right\}.$$

The quantity $V^e(\mathbf{n},t)$ is the maximal expected number of service completions which can be delivered from system state $(\mathbf{n},t)$. Note that the quantity $V^e(\mathbf{L},0)$ is developed in (3.6),(3.7) and (3.8) above. The observed state $(\mathbf{n},t)$ implies the marginal conditional distributions

$$\{n_{1j}, n_{2j}, \ldots n_{Jj}\} \mid \mathbf{n} \sim Multinomial\left(n_j; p_{1j}(t), p_{2j}(t), \ldots, p_{Jj}(t)\right), 1 \leq j \leq J \tag{3.20}$$

over true (but unobservable) states $(\mathbf{n}_{..},t)$, where $\mathbf{n}_{..}$ is the unobserved true state associated with $\mathbf{n}$. These are marginals for the *posterior distribution* over true states which apply when the system finds itself in state $(\mathbf{n},t)$. For any service policy π, we then develop the quantity $N(\pi \mid \mathbf{n},t)$ by natural extension from (3.6) and write

$$V^e(\mathbf{n},t) \equiv \sup_{\pi} V_\pi^e(\mathbf{n},t), \tag{3.21}$$

where

$$V_\pi^e(\mathbf{n},t) = E_{\mathbf{n}_{..}|(\mathbf{n},t)}\left[E\left\{N(\pi \mid \mathbf{n}_{..},t)\right\}\right], \tag{3.22}$$

with the outer expectation in (3.22) being taken with respect to the posterior

distribution over true states. With the above in place we may write the DP optimality equations as

$$V^e(\mathbf{n},t) = 1 + \max_{j \in A(\mathbf{n})} \left\{ \int_0^\infty \sum_{\mathbf{n}'} p^e(\mathbf{n}' \mid \mathbf{n},t,j,s)\, V^e(\mathbf{n}',t+s)\, dG^e_{j,t}(s) \right\}, \mathbf{n} \neq \mathbf{0}, \quad (3.23)$$

and

$$V^e(\mathbf{0},t) = 0.$$

In the special case of deterministic service times considered in (3.17), we can specialise (3.23) to

$$V^e(\mathbf{n},t) = 1 + \max_{j \in A(\mathbf{n})} \left\{ \sum_{i=1}^{J} \sum_{\mathbf{n}'} p_{ij}(t)\, p^e(\mathbf{n}' \mid \mathbf{n},t,j,S_i)\, V^e(\mathbf{n}',t+S_i) \right\}, \mathbf{n} \neq \mathbf{0}. \quad (3.24)$$

We further note that DP value iteration is available to us to compute the Bayes' returns associated with any specified service policy π. In this event we use the recursion

$$V^e_\pi(\mathbf{n},t) = 1 + \left\{ \int_0^\infty \sum_{\mathbf{n}'} p^e(\mathbf{n}' \mid \mathbf{n},t,\pi(\mathbf{n},t),s)\, V^e_\pi(\mathbf{n}',t+s)\, dG^e_{\pi(\mathbf{n},t),t}(s) \right\}, \mathbf{n} \neq \mathbf{0}, \quad (3.25)$$

which in the case of deterministic service times becomes

$$V^e_\pi(\mathbf{n},t) = 1 + \left\{ \sum_{i=1}^{J} \sum_{\mathbf{n}'} p_{i\pi(\mathbf{n},t)}(t)\, p^e(\mathbf{n}' \mid \mathbf{n},t,\pi(\mathbf{n},t),S_i)\, V^e_\pi(\mathbf{n}',t+S_i) \right\}, \mathbf{n} \neq \mathbf{0}. \quad (3.26)$$

In Chapter 2 we have shown that in the case of perfect classification, the simple static heuristic policy π^S works well when loss rates are low. We now adapt this policy to the imperfect classification case as follows: list the job classes in increasing order of the quantity $E(X^e_j)\, E(Y^e_{j,0})$, i.e such that

$$E(X^e_1)\, E(Y^e_{1,0}) \leq E(X^e_2)\, E(Y^e_{2,0}) \leq \ldots E(X^e_J)\, E(Y^e_{J,0}). \quad (3.27)$$

In any state $(\mathbf{n},t)$, the adapted policy π^S chooses action $\pi^S(\mathbf{n},t)$ where

$$\pi^S(\mathbf{n},t) = \min\{j; n_j \geq 1\}.$$

In what follows, the policy π^S will be both assessed as a policy in its own right and also used as a building block in the construction of strongly performing heuristic policies for our Bayes sequential decision problems. How we do this is described in the next section.

Remark 3.2. *We have demonstrated in Chapter 2 that the scenario of exponentially distributed lifetimes and service times is relatively easy to analyse in the perfect classification case because of the memoryless property. The time dependence disappears and the state reduces to just* $\mathbf{n}$. *This simplification is not available when classification is imperfect. To see this, suppose that lifetimes and service times for type i jobs are exponentially distributed as* $F_i(s) = 1 - e^{-\theta_i s}$ *and* $G_i(s) = 1 - e^{-\mu_i s}$, *respectively. According to equation (3.9) and (3.15), we have for each class* j *the following distributions*

$$F_j^e(s) = 1 - \sum_{i=1}^{J} p_{ij}(0)\, e^{-\theta_i s},$$

$$G_{j,t}^e(s) = 1 - \sum_{i=1}^{J} p_{ij}(t)\, e^{-\mu_i s},$$

both of which have now lost the memoryless property. This can be also seen from hazard rates which are now a function of time rather than constant as before, namely,

$$\theta_j^e(s) = \sum_{i=1}^{J} p_{ij}(s)\, \theta_i.$$

As a result, the time dimension cannot be discarded from the state and in this case the state space is therefore continuous and infinite. It is not easy at all to solve such problems exactly. Due to the computational constraints, we shall focus instead

on the Weibull lifetime and deterministic service time scenario in the numerical analysis. As we shall see in section 3.4, this scenario permits one to compute optimal policies via an exact value iteration.

3.3 On the Development of Effective Heuristic Policies

For problems of realistic size, the utilisation of full DP to develop optimal service policies via suitable deployment of optimality equations (3.23) is computationally intractable. In light of the results from the previous chapter, one possible route to the development of effective policies would be to apply a single DP policy improvement step to the simple static proposal π^S in (3.27). The resulting policy, which is again referred to as π^{SPI}, is determined as follows:

$$\pi^{SPI}(\mathbf{n},t) = \underset{j\in A(\mathbf{n})}{\arg\max} \left\{ \int_0^\infty \sum_{\mathbf{n}'} p^e(\mathbf{n}' \mid \mathbf{n},t,j,s)\, V^e_{\pi^S}(\mathbf{n}',t+s)\, dG^e_{j,t}(s) \right\}, \mathbf{n} \neq \mathbf{0}. \tag{3.28}$$

Sadly, the computation of $V^e_{\pi^S}$ is in many cases not tractable. Instead, we develop an approximation $V^{e,app}_{\pi^S}$ to $V^e_{\pi^S}$ by the deployment of an appropriate fluid approximation to the stochastic service system. This approach, which extends that described in Chapter 2 for the perfect classification case, is now described.

In our fluid approximation, we fix $j \in \{1,2,\ldots,J\}$ and represent the class j situation when its processing begins under static policy π^S by the pair (m_j, s_j) In this representation, s_j is the time at which class j service begins under π^S and m_j is an amount of fluid representing the number of class j jobs surviving then. The nature of policy π^S means that class j will be served continually from (m_j, s_j) until all of the class j jobs are completed, namely until all of the corresponding fluid is drained in the approximating model. The process of draining class j fluid is as follows: if $m_j \geq 1$ a single unit of fluid is removed instantaneously at time

$t_{j0} = s_j$ and signifies the guaranteed completion of a single job's service. Loss of fluid is thereafter experienced at rate $\theta_j^e(\tau)$ during the period of this initial class j service, which in the fluid model occupies the time interval $[t_{j0}, t_{j1})$, where $t_{j1} = t_{j0} + E\left(Y^e_{j,t_{j0}}\right)$. Should the amount of class j fluid remaining at t_{j1} exceed one then a further single unit of fluid is removed instantaneously then and signifies the guaranteed completion of a second class j service, and so on. In what follows, t_{jk} denotes the time of the kth class j service completion while $R_j(\tau)$ denotes the amount of class j fluid remaining at time τ. Class j fluid draining as service is offered to class j continuously from (m_j, s_j) is modelled as follows:

$$t_{j0} = s_j; t_{jk+1} = t_{jk} + E\left(Y^e_{j,t_{jk}}\right), k \in \mathbb{N}, \tag{3.29}$$

$$\begin{aligned} R_j(t_{j0}) &= m_j, \\ R_j'(\tau) &= -\theta_j^e(\tau) R_j(\tau), \tau \notin \{t_{jk}, k \in \mathbb{N}\}, \\ R_j\left(t_{jk}^+\right) &= \{R_j(t_{jk}) - 1\}^+, k \in \mathbb{N}. \end{aligned} \tag{3.30}$$

The illustration to the above model is very similar to Figure 2.1 in Chapter 2, except that the time interval between two service completions is now a variable quantity ($E(Y^e_{t_{jk}})$) instead of the constant (μ_j^{-1}). This model is solved recursively using:

$$\begin{aligned} R_j(t_{jk+1}) &= R_j\left(t_{jk}^+\right) \exp\left\{\int_{t_{jk}}^{t_{jk+1}} -\theta_j^e(\tau) d\tau\right\}, \\ R_j(t_{j0}) &= m_j, \end{aligned} \tag{3.31}$$

where t_{jk} is computed by (3.29) and $R_j(t_{jk}^+)$ from the last equation in (3.30). The solution process terminates as soon as the first zero value of $R_j\left(t_{jk}^+\right)$ is encountered.

We now introduce the quantities

$$K_j(m_j, s_j) = \min\left\{k; R_j\left(t_{jk}^+\right) = 0\right\} \tag{3.32}$$

and

$$N_j\left(m_j, s_j\right) = K_j\left(m_j, s_j\right) + R_j\left(t_{jK_j(m_j,s_j)}\right). \tag{3.33}$$

From (3.32), $K_j\left(m_j, s_j\right)$ is the (integer) number of fully completed class j jobs under the fluid model while $R_j\left(t_{jK_j(m_j,s_j)}\right)$ is the fractional amount of class j fluid remaining at the conclusion of the class j processing and which is deemed to yield a further fractional completion within the approximating fluid model. In the fluid model, we take the total processing time of class j to be

$$T_j\left(m_j, s_j\right) = \sum_{k=0}^{K_j(m_j,s_j)-1} E\left(Y^e_{j,t_{jk}}\right) + R_j\left(t_{jK_j(m_j,s_j)}\right) E\left(Y^e_{j,t_{jK_j\left(m_j,s_j\right)}}\right). \tag{3.34}$$

We now fix system state $(\mathbf{n},t)$ and use the quantities developed in (3.29)-(3.34) to develop $V^{e,app}_{\pi^S}(\mathbf{n},t)$, the estimate of the expected number of job completions secured under static policy π^S from state $(\mathbf{n},t)$ obtained from our approximating fluid model. We define the quantities $\zeta_j\left(\mathbf{n},t\right), \psi_j\left(\mathbf{n},t\right), 1 \le j \le J$, inductively as follows:

$$\zeta_1\left(\mathbf{n},t\right) = N_1\left(n_1, t\right), \psi_1\left(\mathbf{n},t\right) = T_1\left(n_1, t\right),$$

and

$$\zeta_j\left(\mathbf{n},t\right) = N_j\left(n_j \exp\left\{-\int_0^{\sum_{k=1}^{j-1}\psi_k(\mathbf{n},t)} \theta^e_j\left(t+v\right)dv\right\}, t + \sum_{k=1}^{j-1}\psi_k\left(\mathbf{n},t\right)\right), 1 \le j \le J, \tag{3.35}$$

$$\psi_j\left(\mathbf{n},t\right) = T_j\left(n_j \exp\left\{-\int_0^{\sum_{k=1}^{j-1}\psi_k(\mathbf{n},t)} \theta^e_j\left(t+v\right)dv\right\}, t + \sum_{k=1}^{j-1}\psi_k\left(\mathbf{n},t\right)\right), 1 \le j \le J. \tag{3.36}$$

The quantity $\zeta_j(\mathbf{n},t)$ records the number of class j services completed under the fluid model when π^S is applied from state $(\mathbf{n},t)$, and the quantity $\psi_j(\mathbf{n},t)$ the processing time taken on this class. The first argument of N_j, T_j on the right hand side of (3.35) and (3.36) is the number of class j jobs present when the processing of class j begins. The second argument is the time at which the processing of class

j begins. The original number n_j (present at t) is diminished by losses occurring over the time period $[t, t + \sum_{k=1}^{j-1} \psi_k(\mathbf{n},t))$ during which the first $j-1$ classes are processed.

We now obtain the approximating value function as

$$V_{\pi^S}^{e,app}(\mathbf{n},t) = \sum_{j=1}^{J} \zeta_j(\mathbf{n},t). \tag{3.37}$$

Example 3.2. *For the problem with deterministic service times, and the p_i and lifetime distributions given in Example 3.1, and the following additional problem parameters, namely*

- $L_1 = L_2 = 5$,
- $S_1 = 1.41, S_2 = 4.76$,

we computed both the exact value function $V_{\pi^S}^e$ and the fluid approximation $V_{\pi^S}^{e,app}$. As in the perfect classification situation, we plot in Figure 3.1 fluid approximation values at time zero and in Figure 3.2 both exact and approximate ones at a set of states at four representative decision epochs. A summary of the percentage approximation errors, $\Delta(V_{\pi^S}^e, V_{\pi^S}^{e,app}) = 100|1 - V_{\pi^S}^{e,app}/V_{\pi^S}^e|\%$, over the effective state space is presented in Table 3.1 below. For this example, a state (n_1, n_2, t) is in the effective sample space if

1. *t can be expressed in terms of*

$$t = r_1 S_1 + r_2 S_2,$$

for some r_1, r_2 which are non-negative integers and satisfy $0 \leq r_1, r_2 \leq (L_1 + L_2) = 10$;

2. *and the following condition holds:*

$$t \leq (10 - n_1 - n_2) \max\{S_1, S_2\}.$$

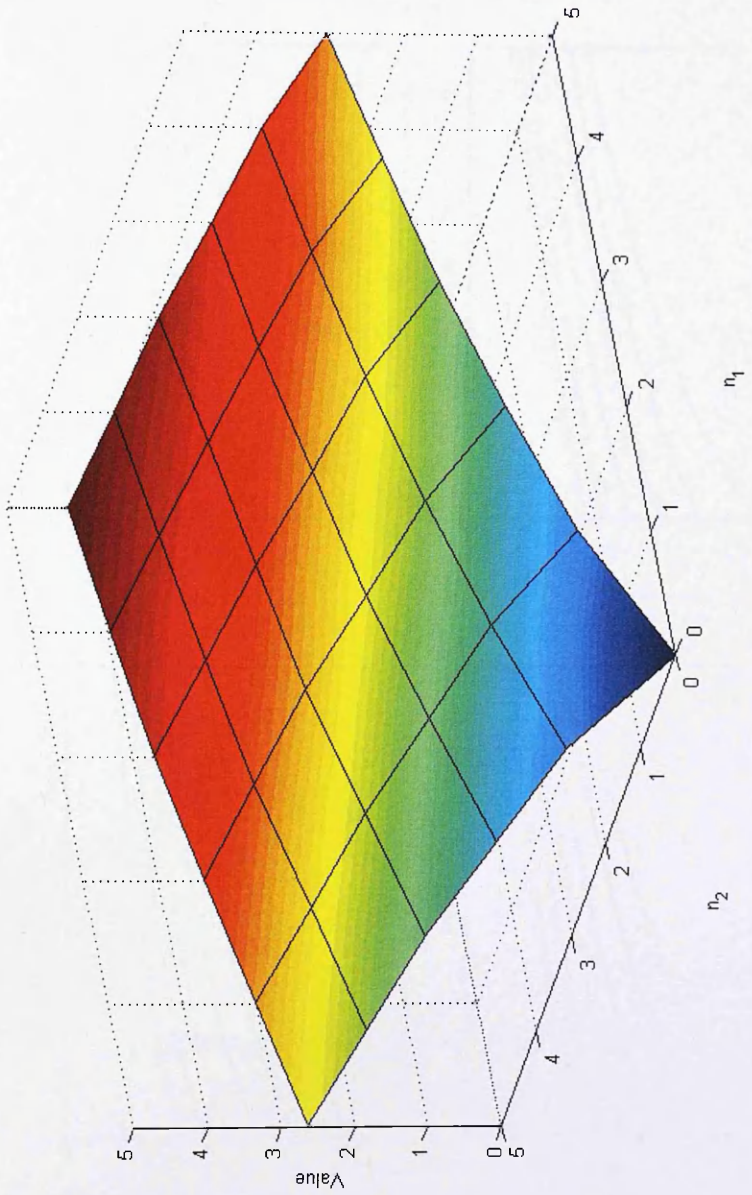

Figure 3.1: Values of $V_{\pi^S}^{e,app}(n_1, n_2, 0)$ where $0 \leq n_1, n_2 \leq 5$.

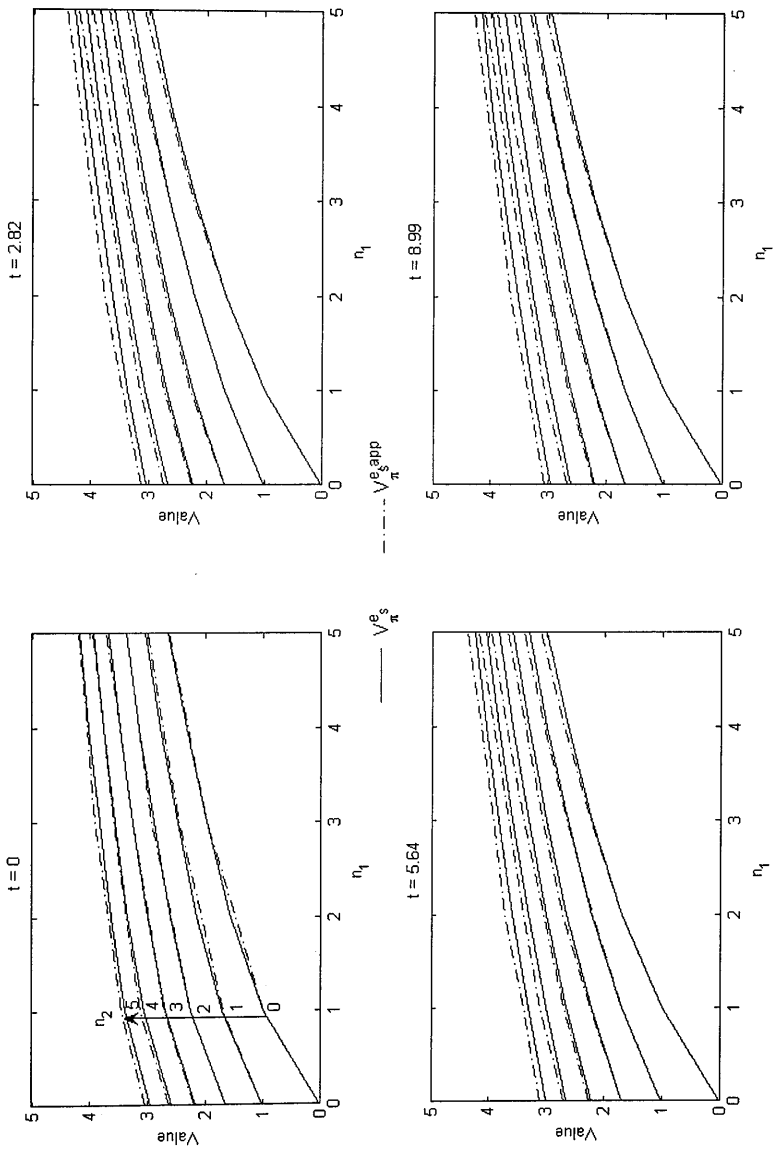

Figure 3.2: Exact value functions $V^e_{\pi^S}$ verses the fluid approximations $V^{e,app}_{\pi^S}$ in states (n_1, n_2, t) where $0 \leq n_1, n_2 \leq 5$ and t=0, 2.82, 5.64, 8.99.

MEAN	1.41%
MIN	0.00%
1ST QUARTILE	0.00%
MEDIAN	1.48%
3RD QUARTILE	2.35%
MAX	4.55%

Table 3.1: Percentage approximation errors $\Delta(V^{e}_{\pi^S}, V^{e,app}_{\pi^S})$ for Example 3.2.

It is worth mentioning that due to the uncertainty of each job's true identity, the effective state space tends to be larger in the imperfect triage case compared to the perfect triage of Chapter 2. This can be seen from the more relaxed upper bounds on r_1, r_2 and on time t.

We can see that the approximating value function is again increasing and concave componentwise in $\mathbf{n}$, for fixed t. Further, the fluid approximation still has robustly outstanding performance, with an average error of 1.41% and a worst performance of 4.55%.

Our proposed sequential decision rule π^{SF} in the case of error-prone triage can now be obtained by deploying the approximate value function from (3.37) within (3.28). We write,

$$\pi^{SF}(\mathbf{n},t) = \underset{j\in A(\mathbf{n})}{\arg\max} \left\{ \int_0^{\infty} \sum_{\mathbf{n}'} p^{e}(\mathbf{n}' \mid \mathbf{n},t,j,s)\, V^{e,app}_{\pi^S}(\mathbf{n}',t+s)\, dG^{e}_{j,t}(s) \right\}, \mathbf{n} \neq \mathbf{0}, \tag{3.38}$$

When service times are deterministic, we have the form

$$\pi^{SF}(\mathbf{n},t) = \underset{j\in A(\mathbf{n})}{\arg\max} \left\{ \sum_{i=1}^{J} \sum_{\mathbf{n}'} p_{ij}(t) p^{e}(\mathbf{n}' \mid \mathbf{n},t,j,S_i)\, V^{e,app}_{\pi^S}(\mathbf{n}',t+S_i) \right\}, \mathbf{n} \neq \mathbf{0}. \tag{3.39}$$

3.4 Numerical Study

In what follows we compare the performance of the heuristic π^{SF} developed in the preceding section with that of the static proposal π^{S} described around (3.27).

We shall also consider π^M, an adaptation of the myopic policy proposed by Argon et al. [2008] for the perfect classification case. In state $(\mathbf{n},t)$, π^M chooses the action from $A(\mathbf{n})$ to be the non-empty class j with the smallest associated value of

$$E\left(Y_{j,t}^e\right)\left[\sum_{k=1}^{J}\left(n_k-\delta_{kj}\right)\left\{E\left(X_k^e-t \mid X_k^e>t\right)\right\}^{-1}\right].$$

In the above expression, $E\left(X_k^e-t \mid X_k^e>t\right)$ or MRL for the class k jobs, can be obtained by

$$E(X_k^e-t|X_k^e>t)=\sum_{i=1}^{J}p_{ik}(t)E(X_i-t|X_i>t), \tag{3.40}$$

in which $E(X_i-t|X_i>t)$ is the MRL for type i jobs and can be calculated by equation (2.47).

Proof of (3.40). By definition, we have

$$E(X_k^e-t|X_k^e>t)=\frac{\int_t^\infty(1-F_k^e(s))ds}{1-F_k^e(t)}=\frac{\int_t^\infty \bar{F}_k^e(s)ds}{\bar{F}_k^e(t)}. \tag{3.41}$$

From equation (3.10),

$$E(X_k^e-t|X_k^e>t)=\frac{\int_t^\infty\sum_{i=1}^{J}p_{ik}(0)\bar{F}_i(s)ds}{\sum_{i=1}^{J}p_{ik}(0)\bar{F}_i(t)}. \tag{3.42}$$

Substitute $p_{ik}(0)$ by equation (3.1), simplify the resulting expression, and switch the order of integration and summation, to obtain

$$E(X_k^e-t|X_k^e>t)=\frac{\sum_{i=1}^{J}\epsilon_{ik}p_i\int_t^\infty\bar{F}_i(s)ds}{\sum_{i=1}^{J}\epsilon_{ik}p_i\bar{F}_i(t)}. \tag{3.43}$$

Change the index from i to m in the denominator, and move it into the summation

in the numerator; the above equation then becomes

$$E(X_k^e - t|X_k^e > t) = \sum_{i=1}^{J} \frac{\epsilon_{ik}p_i}{\sum_{m=1}^{J} \epsilon_{mk}p_m\bar{F}_m(t)} \int_t^\infty \bar{F}_i(s)ds.$$
$$= \sum_{i=1}^{J} \frac{\epsilon_{ik}p_i\bar{F}_i(t)}{\sum_{m=1}^{J} \epsilon_{mk}p_m\bar{F}_m(t)} \frac{\int_t^\infty \bar{F}_i(s)ds}{\bar{F}_i(t)}. \quad (3.44)$$

Note that the first term in the summation is just $p_{ik}(t)$ (refer back to equation 3.2), and the second term $E(X_i - t|X_i > t)$. The result then follows immediately. □

The Bayes' returns for each of π^{SF}, π^S and π^M are compared to the optimum for 18,000 randomly generated problems in each of which job lifetimes are Weibull and service times are deterministic. As we can see from (3.24), (3.26), and (3.39) in the deterministic service time case the optimality equation, value function equation and approximate single step policy improvement equation are computed via summations over a finite number of terms, followed by an argmax over a finite set. For problems of modest size, it is thus possible, even though very expensive, to compute optimal policies for such cases and compare competing heuristics against the optimal performance.

The problems are generated at random under four different sets of assumptions (represented by categories A,B,C and D) regarding the relative lengths of service times and lifetimes of individual jobs and under three different sets of assumptions (poor, medium and good) regarding the quality of the initial job classification. Some problems involve just two job types ($J = 2$) while for others there are four ($J = 4$). The key problem features S_i, α_i, β_i and N are sampled/chosen as follows:

$$S_i \sim U[1,10] \qquad (J=2 \text{ cases}); \tag{3.45a}$$

$$S_i \sim U[1,5] \qquad (J=4 \text{ cases}); \tag{3.45b}$$

$$\alpha_i \sim U[1,2]\,; \tag{3.45c}$$

$$\beta_i \Gamma\left(1+\alpha_i^{-1}\right) S_i^{-1} \mid S_i, \alpha_i \sim U[0.1,0.5] \qquad \text{(very short lifetimes, } A); \tag{3.45d}$$

$$\beta_i \Gamma\left(1+\alpha_i^{-1}\right) S_i^{-1} \mid S_i, \alpha_i \sim U[0.5,2] \qquad \text{(short lifetimes, } B); \tag{3.45e}$$

$$\beta_i \Gamma\left(1+\alpha_i^{-1}\right) S_i^{-1} \mid S_i, \alpha_i \sim U[2,10] \qquad \text{(moderate lifetimes, } C); \tag{3.45f}$$

$$\beta_i \Gamma\left(1+\alpha_i^{-1}\right) S_i^{-1} \mid S_i, \alpha_i \sim U[10,100] \qquad \text{(long lifetimes, } D); \tag{3.45g}$$

$$N=20 \qquad (J=2 \text{ cases}); \tag{3.45h}$$

$$N=10 \qquad (J=4 \text{ cases}). \tag{3.45i}$$

Further, for each problem the p_i are obtained by first sampling independently from $U[0.1,0.9]$ and then normalising. The (mis)classification probabilities ϵ_{ij} are obtained as follows: first obtain the probabilities of correct classification ϵ_{ii} by sampling as follows:

$$\epsilon_{ii} \sim U[0.5,0.65] \qquad \text{(poor classification)}; \tag{3.45j}$$

$$\epsilon_{ii} \sim U[0.65,0.85] \qquad \text{(medium classification)}; \tag{3.45k}$$

$$\epsilon_{ii} \sim U[0.85,1] \qquad \text{(good classification)}. \tag{3.45l}$$

Then obtain the ϵ_{ij} , $i \neq j$, by sampling independently from $U[0,1]$ and normalising suitably.

Please note that the number of initial jobs N here is much less than in the counterpart scenario when the classification is perfect (see (2.49g)-(2.49h) in section 3.4). The MDP model and the DP recursions in this chapter are much more challenging. Moreover, the state space is far larger, which can be seen from the fact that the next decision epoch resulted from an action can have J possibilities, rather than just one in the perfect classification situation. We found that it is not

possible to solve larger problem instances to optimality in reasonable time.

Each sampled instance from the above is called a *profile.* For each profile, we generate a range of problems with different initial states (**L**,0). This is done in two steps as follows: first, the number K_i of jobs of type i is obtained by sampling from the multinomial distribution

$$\{K_1, K_2, \ldots, K_J\} \sim Multinomial\,(N; p_1, p_2, \ldots p_J)\,. \tag{3.46}$$

Second, for each fixed i, the L_{ij}, $1 \leq j \leq J$, namely the number of type i jobs classified initially as j is then obtained by sampling from the multinomial distribution

$$\{L_{i1}, L_{i2,} \ldots, L_{iJ}\} \sim Multinomial\,(K_i; \epsilon_{i1}, \epsilon_{i2}, \ldots, \epsilon_{iJ})\,. \tag{3.47}$$

These samples are drawn independently for distinct i. We obtain the components of initial state **L** by setting L_j, the total number of jobs initially classified as j, equal to $\sum_{i=1}^{J} L_{ij}$. Please note that our sampling scheme is such that the L_j will tend to be smaller for the $J = 4$ cases studied than for the $J = 2$ ones. This choice is dictated by the computational requirements of the value iteration scheme needed for the determination of the maximal Bayes' return and the associated optimal policies. Please note that our heuristic policies can themselves be computed easily for much larger problems.

Please also note that service times are sampled from a smaller range for $J = 4$ cases. The large variability of service times will lead to a huge number of decision epochs and thus unreasonably long computational times. Since the true identity of each job under service is unknown, the time between two consecutive decision epochs could be the service time of any job type. The longest service time determines the last possible decision epoch, while the shortest service time determines the gap between two consecutive epochs. When these two extreme values are far apart, the number of decision epochs is very large. A smaller sample interval can greatly reduce the number of decision epochs and thus the computational times,

yet has little impact on the analysis. It is just a scale change. The parameter that really makes a difference is the relative length between lifetimes and service times, which has not been changed at all.

For $J = 2$, 200 profiles were generated according to the above sampling scheme in (3.45a)-(3.45l) for each of the 12 combinations of problem category (A,B,C,D) and classification quality (poor, medium, good). For each profile, 5 problems (ie, initial states) were generated according to (3.46) and (3.47). Thus the total number of problems generated for each problem category/classification quality combination quality was $1,000$, making $12,000$ problems overall. For each problem, the quantities $V^e_{\pi^{SF}}(\mathbf{L},0)$, $V^e_{\pi^S}(\mathbf{L},0)$ and $V^e_{\pi^M}(\mathbf{L},0)$, the Bayes' returns respectively for the three heuristics π^{SF}, π^S and π^M were computed along with the maximal return $V^e(\mathbf{L},0)$. All computations used an appropriate form of DP value iteration from (3.24) and (3.26). For each heuristic $\pi = \pi^{SF}, \pi^S, \pi^M$ and each problem $(\mathbf{L},0)$ generated , the percentage suboptimality

$$\Delta_\pi(\mathbf{L},0) \equiv 100\{V^e(\mathbf{L},0) - V^e_\pi(\mathbf{L},0)\}\{V^e(\mathbf{L},0)\}^{-1}$$

was computed. Further, for each subcollection of $1,000$ problems corresponding to a problem category/classification quality combination, the minimum, mean and maximum values of $\Delta_\pi(\mathbf{L},0)$ were computed for each heuristic. These values may be found in Table 3.2(a). As in Chapter 2, we calculated, for each of the categories, the number of instances in which each heuristic $\pi = \pi^{SF}, \pi^S, \pi^M$ provides the best performance. In Table 3.2(b) find these results, together with p-values of the Friedman test on the differences in performance.

From Table 3.2(a), we observe that the policy π^{SF} developed by utilising an approximating fluid model within a single step DP policy improvement performs robustly well throughout. Its mean percentage suboptimality never exceeds 0.03% with a worst case, among all $12,000$ problems of just 1.21% suboptimal. It comfortably outperforms π^S and π^M excepting only the category B/poor classification case where it is marginally outperformed by π^M. Serious suboptimalities are ob-

	Classification	poor			medium			good		
Category		π^{SF}	π^{S}	π^{M}	π^{SF}	π^{S}	π^{M}	π^{SF}	π^{S}	π^{M}
A	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.00	0.32	0.01	0.00	1.15	0.02	0.00	1.95	0.05
	Max	0.00	4.39	0.43	0.00	20.65	2.32	0.00	26.08	5.35
B	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.03	0.12	0.01	0.02	0.89	0.05	0.01	1.53	0.12
	Max	0.76	4.15	0.27	0.91	11.36	2.06	0.69	20.93	7.31
C	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.02	0.11	0.17	0.02	0.21	0.53	0.02	0.54	0.89
	Max	0.58	1.64	3.46	0.85	4.59	8.82	1.21	11.35	15.74
D	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.00	0.04	0.18	0.00	0.12	0.62	0.01	0.25	1.24
	Max	0.07	0.73	3.77	0.12	1.64	7.88	0.99	3.24	13.24

(a) Percentage deviation from optimal performance.

	poor				medium				good			
	π^{SF}	π^{S}	π^{M}	Friedman's p-value	π^{SF}	π^{S}	π^{M}	Friedman's p-value	π^{SF}	π^{S}	π^{M}	Friedman's p-value
A	1000	629	937	0.000	1000	610	935	0.000	1000	557	912	0.000
B	610	617	776	0.000	824	601	756	0.000	911	540	631	0.000
C	501	717	665	0.001	888	690	566	0.000	938	652	526	0.000
D	924	735	538	0.000	972	769	556	0.000	988	735	510	0.000

(b) Number of win instances out of a total of 1000, with p-values for Friedman test of difference.

Table 3.2: Numerical study results for Weibull lifetimes and deterministic service times when $J = 2$, imperfect classification.

served for the policies π^S and π^M, especially for problem configurations for which they were not designed. As before, policy π^S works well when jobs have long lifetimes and deteriorates as lifetimes decrease. For policy π^M the reverse is the case. From Table 3.2(b) we find that, in most category/classification quality combinations, policy π^{SF} provides the best performance more often than the other two. Particularly in category A, it is the best policy in all the 1,000 problems regardless of the classification quality. It only loses to π^S and π^M at a small disadvantage in categories B and C when the classification is poor. The zero p-values nearly everywhere show that these policies have statistically significant differences in performance. The single exception is category C/poor classification in which the p-value is 0.001. This is still very strong evidence of differences in performance among the heuristics.

Note that, within problem categories, percentage suboptimalities tend to increase with classification quality. Take category A as one example. The mean and worst percentage for policy π^S are 0.32% and 4.39% when the classification is poor, but they increase to 1.95% and 26.08% if the classification is good. Similarly, for policy π^M, its mean(worst) percentage increases from 0.01%(0.43%) to 0.05%(5.35%) when the classification quality improves from poor to good. The same pattern can be found in the other categories.

To understand this, consider the worst case for classification in which $\epsilon_{ij} = J^{-1} \forall i, j$ and the classification process randomly allocates jobs to classes. In this case, the assessment process fails to offer useful information on job type and posterior probabilities (of true type) are independent of class. All jobs are effectively members of a single undifferentiated class and service policies which make use of class information are indistinguishable. As the classification improves from this worst case, the classes become more distinct, information on class membership more informative, policies more distinct and hence the choice of policy more important. It is thus unsurprising that the differences between the heuristic service policies is most pronounced when classification is reasonable or good. A cautionary

note is that the small suboptimalities when classification is poor do not necessarily indicate satisfactory service outcomes. It may well be that even the optimal policy cannot achieve a high Bayes' return when the classification errors are significant. There is little value in investing lots of resource to develop optimal policies or the fluid heuristic which are just marginally better than these simple ones. The only way is to improve the classification process itself. However, as we shall see in the next chapter, there are some special situations in which the classification outcomes really are immaterial and hence the difference between policies small. In such cases, effective scheduling is not reliant upon an accurate initial triage. A detailed account on this theme is presented in the next chapter.

The study for problems with $J = 4$ was conducted in a similar fashion, except that its computational demands were such that only 100 profiles were generated for each of the problem category/classification quality combinations. Hence in this part of the study a further $6,000$ problems were investigated.

The results summarized in Table 3.3(a) are qualitatively very similar to those for the $J = 2$ cases. However, the performance pattern with regard to the lifetime category or the classification quality is not as clear as in Table 3.2(a). In some cases the heuristic π^S works better when jobs leave faster, and in some others the myopic policy π^M works better when jobs leave more slowly. This is due to the relatively small sample size which is forced upon us by the computational complexity when $J = 4$. To uncover the real trend, we calculated the moving average of the mean percentages over two adjacent categories and two adjacent classification qualities. The results presented in Table 3.3(b) more clearly show the consistent pattern as before.

In comparison with Table 3.2(b), Table 3.3(c) shows stronger performance of π^{SF} in that it wins in all the scenarios. Moreover, the number of win instances is considerably more than that of π^S and π^M. The p-values are now all zeroes and thus there is strong evidence of differences in performance among the heuristics.

Remark 3.3. *Analysis has been conducted to understand how the optimal policy*

	Classification	poor			medium			good		
Category		π^{SF}	π^{S}	π^{M}	π^{SF}	π^{S}	π^{M}	π^{SF}	π^{S}	π^{M}
A	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.00	0.92	0.28	0.00	1.17	0.23	0.00	1.25	0.18
	Max	0.00	11.83	5.97	0.33	15.45	6.91	0.00	19.34	9.64
B	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.06	1.00	0.08	0.07	1.49	0.10	0.06	1.63	0.14
	Max	2.01	9.82	1.87	3.14	10.83	3.66	2.02	17.06	6.26
C	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.05	0.30	0.44	0.03	0.27	0.60	0.04	0.24	0.60
	Max	1.13	5.11	3.67	0.69	3.82	5.89	0.84	2.88	6.05
D	Min	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Mean	0.03	0.23	0.42	0.01	0.29	0.40	0.02	0.36	0.69
	Max	0.96	2.74	3.19	0.41	2.87	3.26	0.60	2.63	4.87

(a) Percentage deviation from optimal performance.

	poor + medium			medium + good		
	π^{SF}	π^{S}	π^{M}	π^{SF}	π^{S}	π^{M}
$A+B$	0.03%	1.15%	0.17%	0.03%	1.39%	0.16%
$B+C$	0.05%	0.76%	0.31%	0.05%	0.91%	0.36%
$C+D$	0.03%	0.27%	0.46%	0.03%	0.29%	0.57%

(b) Moving average of mean percentage deviation from optimal performance.

	poor				medium				good			
	π^{SF}	π^{S}	π^{M}	Friedman's p-value	π^{SF}	π^{S}	π^{M}	Friedman's p-value	π^{SF}	π^{S}	π^{M}	Friedman's p-value
A	498	184	368	0.000	500	159	382	0.000	500	227	400	0.000
B	414	104	234	0.000	442	79	190	0.000	444	141	238	0.000
C	398	180	134	0.000	440	232	127	0.000	460	292	199	0.000
D	479	168	83	0.000	486	164	133	0.000	491	209	133	0.000

(c) Number of win instances out of a total of 500, with p-values for Friedman test of difference.

Table 3.3: Numerical study results for Weibull lifetimes and deterministic service times when $J = 4$, imperfect classification.

for the perfect classification case performs in the imperfect classification case. By deeming the classification errors to be zero, the optimal policy for the perfect classification was developed via the method described in Chapter 2. Then the Bayes' return for this policy was computed and compared against that of each of the heuristics and the maximal return $V^e(\boldsymbol{L}, 0)$. *Results show that even though sometimes, especially when the errors are small, this policy is also optimal in the imperfect classification case, it is in general suboptimal. In some cases its performance is weak and is outperformed by* π^S.

To conclude this section, we list below several implementation features in this chapter, which complement those mentioned in Section 2.3.4 in Chapter 2.

1. The prior probabilities $p_{ij}(0)$ and posterior probabilities $p_{ij}(t), t > 0$ are calculated only once. Whenever needed, we just retrieve the value from memory. Similarly, the remaining lifetime distribution for class j jobs, $P(X_j^e < t + s \mid X_j^e > t)$ appearing in equation (3.18), is calculated and stored in memory. They are used frequently to compute transition probabilities.

2. There is no closed form solution to the integrations in equation (3.19) and (3.31) in most cases, they are thus calculated numerically by Matlab function *quadl*, as in Chapter 2.

3. The main C++ class for this numerical study is also derived from the one for the Weibull lifetime and deterministic service time scenario in Chapter 2.

3.5 Conclusion

In this chapter we consider an error-prone triage problem in which jobs assessed as one class could actually have many different characteristics. To deal with this additional uncertainty, we propose a Bayesian sequential decision model for this

problem. Our beliefs on the true identity of the remaining jobs are updated over time in light of the job's survival.

To solve this Bayesian model, we reformulate it as a SMDP. The approximate single step policy improvement algorithm and the fluid approximation model proposed in Chapter 2 are further developed to generate effective heuristics here when triage is with error. Even though the fluid model is now more complex, and does not have closed form solutions, it can still be quickly solved numerically. The resulting approximation is again very close to the corresponding exact value function.

The proposed heuristic is subject to extensive numerical investigation. Unlike in Chapter 2 where three scenarios have been tested, in this chapter we can only test problems with Weibull lifetimes and deterministic service times, since only in such cases are the optimal policies available in reasonable time. Performances of competing heuristics are compared in terms of suboptimality. The results show that our proposed heuristic has outstanding performance throughout and outperforms the alternative heuristics in almost all problem instances.

An interesting observation is that, for all the heuristics, the suboptimality tends to decrease as the triage quality deteriorates. This does not necessarily mean that the heuristics work well when the classification is poor. On the contrary, it may well be that even optimal policies cannot achieve good outcomes. When the triage process is improved, the policies become more distinct, and more is to be gained by choosing a good scheduling policy.

Chapter 4

Cost of Imperfect Classification

In Chapter 3, we have described tools to develop π^{SF}, an effective and easily computed policy for the triage problem with imperfect classification which has been seen to achieve a Bayes' return close to the optimum in a large number of problems. We now explore the complementary question of the price paid in reduced service completions for our inability to classify perfectly. *Inter alia,* this will give us insight concerning situations where there is most to be gained from improving the quality of classification. Section 4.1 introduces a measure to quantify the classification cost. An analytical upperbound for the cost is developed in Section 4.2 for exponential lifetime cases. To investigate the behaviour of the cost in more general situations, a comprehensive numerical study is conducted in 4.3. We conclude this chapter in Section 4.4.

4.1 Introduction

Recall from our development in Section 3.1 that for imperfect classification, we used $\mathbf{L}_{\cdot\cdot}$ for the true state corresponding to the observed state $\mathbf{L}$. We write ν for a service policy which is able to take decisions on the basis of the (unobservable) true state as the system evolves and

$$\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{\cdot\cdot}, 0\right)\right\}$$

for the maximal return available when access to the true system state is available throughout.

Definition 4.1. *The cost of imperfect classification for initial state* $(\mathbf{L},0)$ *is given by*

$$CIC(\mathbf{L},0) \equiv E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{..},0\right)\right\}\right] - V^{e}\left(\mathbf{L},0\right). \tag{4.1}$$

The relative cost of imperfect classification is given by,

$$RCIC\left(\mathbf{L},0\right) \equiv \frac{CIC(\mathbf{L},0)}{V^{e}\left(\mathbf{L},0\right)}. \tag{4.2}$$

An alternative to $RCIC$ can be obtained by

$$\widehat{RCIC} = \frac{RCIC}{1+RCIC} = \frac{CIC(\mathbf{L},0)}{E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{..},0\right)\right\}\right]},$$

which takes values in $[0,1]$. We choose to use $RCIC$ as it is a natural measure of the relative increase in the number of service completions which could have been secured if the true states were able to be observed. This ratio can be more than one if the classification errors are significant.

Example 4.1. *For the problem with the* p_i *and lifetime distributions given in Example 3.1 (page 83), we computed the relative cost* $RCIC$ *with respect to different misclassification probabilities in the range 0.0 to 0.5. Note that* $\epsilon_{ij} = 0.5, i, j = 1, 2$ *is the worst case when there are only two types of jobs. Figure 4.1 below plots* $RCIC$.

In Figure 4.1 we see that the relative cost $RCIC$ is continuous, increasing and concave componentwise in the misclassification probabilities ϵ_{12} and ϵ_{21} and is much more sensitive to the latter. A major factor here is the fact that type 1 jobs have much shorter lifetimes (with mean 1.45) than type 2 (mean 12.75), and are thus lost from the system much more quickly in any event. Should type 2 jobs be classified correctly and scheduled for processing appropriately, they are much

more likely to be served to completion and contribute to the system's return. Mathematically, the quantity $E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{..},0\right)\right\}\right]$ in the expression for $CIC(\mathbf{L},0)$ increases with ϵ_{21} but decreases with ϵ_{12}. For any initial state $(\mathbf{L},0)$, the higher the value of ϵ_{21}, the more jobs are likely to actually be of type 2, and the higher are likely to be the number of successful completions. The quantity $V^{e}(\mathbf{L},0)$ decreases with both misclassification probabilities.

4.2 The Cost of Imperfect Classification - Analytical Insight

In the special case of exponentially distributed lifetimes, it is possible to gain analytical insight into system characteristics which impact the cost of imperfect classification CIC and which will inform our upcoming numerical study. We first state a simple result which will be of use in the analysis.

Lemma 4.1.

$$E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{..},0\right)\right\}\right] \geq V^{e}(\mathbf{L},0) \geq E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[\min_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{..},0\right)\right\}\right]. \tag{4.3}$$

In order to state our main result, we need some additional notation. In the exponential lifetime case we shall write the distribution of type i lifetimes as $X_i \sim \exp\left(\rho\theta_i\right), 1 \leq i \leq J$, where the θ_i are taken to be fixed, and we shall be interested in the 'no loss' limit $\rho \to 0$. Further, we write $\mathbf{Y}\left(\mathbf{L}_{..}\right)$ for the total of the service times associated with the true but unobservable system state $\left(\mathbf{L}_{..},0\right)$. Conditionally upon this true state, as in Section 3.1 we use the triple ijk for the kth type i job to be classified as j. If Y_{ijk} denotes the service time of ijk then $Y_{ijk} \sim G_i$ and we write

$$\mathbf{Y}\left(\mathbf{L}_{..}\right) = \sum_{i=1}^{J}\sum_{j=1}^{J}\sum_{k=1}^{L_{ij}} Y_{ijk.}$$

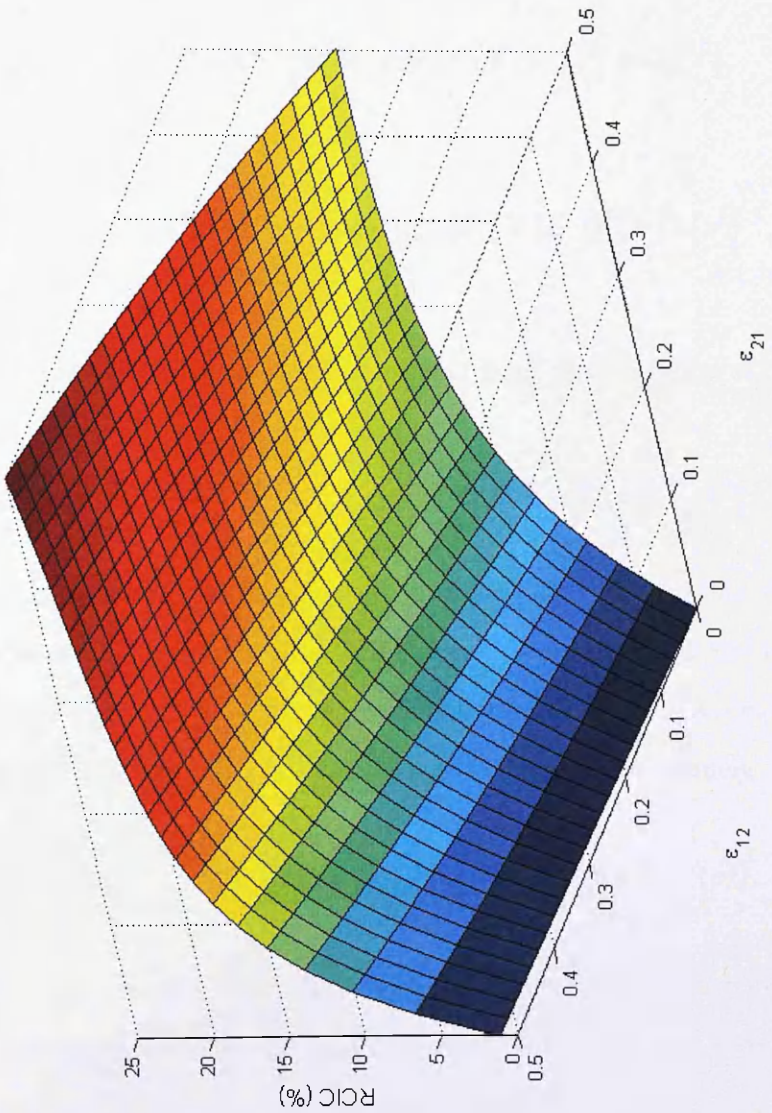

Figure 4.1: The relative costs $RCIC$ for a problem with $J = 2$, Weibull lifetimes and deterministic service times.

Finally, we need the system parameter δ, defined by

$$\delta \equiv \max_i \frac{\theta_i}{E(Y_i)} - \min_i \frac{\theta_i}{E(Y_i)} = \rho^{-1}\left[\max_i \{E(X_i)\,E(Y_i)\}^{-1} - \min_i \{E(X_i)\,E(Y_i)\}^{-1}\right].$$

Theorem 4.1. *For the case of exponential lifetimes, we have*

$$CIC(\mathbf{L},0) \leq \frac{1}{2}\delta\rho E_{\mathbf{L}_{..}|(\mathbf{L},0)}[E\{\mathbf{Y}(\mathbf{L}_{..})\}^2] + O(\rho^2)$$

and

$$RCIC(\mathbf{L},0) \leq \frac{1}{2N}\delta\rho E_{\mathbf{L}_{..}|(\mathbf{L},0)}[E\{\mathbf{Y}(\mathbf{L}_{..})\}^2] + O(\rho^2).$$

Proof. It follows from the above Lemma and from the definition of CIC that

$$CIC(\mathbf{L},0) \leq E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[\max_\nu E\{N(\nu \mid \mathbf{L}_{..},0)\} - \min_\nu E\{N(\nu \mid \mathbf{L}_{..},0)\}\right]. \quad (4.4)$$

It further follows from the analysis of Glazebrook et al. [2004], that when the true state is observable and has initial value $(\mathbf{L}_{..},0)$ then the static service policy ν^{SI} which serves the jobs in increasing order of the quantity $E(X_i)\,E(Y_i)$ secures a return which is within an $O(\rho^2)$ quantity of the maximum, namely

$$\max_\nu E\{N(\nu \mid \mathbf{L}_{..},0)\} - E\{N(\nu^{SI} \mid \mathbf{L}_{..},0)\} \leq O(\rho^2). \quad (4.5)$$

Similarly, it can easily be established that the static policy ν^{SD} which serves the jobs in decreasing order of the quantity $E(X_i)\,E(Y_i)$ secures a return which is within an $O(\rho^2)$ quantity of the minimum, namely

$$E\{N(\nu^{SD} \mid \mathbf{L}_{..},0)\} - \min_\nu E\{N(\nu \mid \mathbf{L}_{..},0)\} \leq O(\rho^2). \quad (4.6)$$

To simplify the argument at this point, we relabel the jobs 1 to N such that, conditional upon $(\mathbf{L}_{..},0)$, the static policy ν^{SI} is identified with the permutation

$(1, 2, \ldots, N)$ and the static policy ν^{SD} with $(N, N-1, \ldots, 1)$. Now, the permutation $(1, 2, \ldots, N)$ can be obtained from $(N, N-1, \ldots, 1)$ by a series of $\binom{N}{2}$ pairwise interchanges. At each stage, a permutation of the form $(\ldots, l, m, \ldots), l > m$, is transformed via a single interchange to $(\ldots, m, l, \ldots)$. From Glazebrook et al. [2004], the static service policies corresponding to these permutations have associated returns such that

$$E\{N((\ldots, m, l, \ldots) \mid \mathbf{L}_{..}, 0)\} - E\{N((\ldots, l, m, \ldots) \mid \mathbf{L}_{..}, 0)\}$$

$$= \left\{ \frac{\rho\theta_m}{E(Y_m)} - \frac{\rho\theta_l}{E(Y_l)} \right\} E(Y_l)E(Y_m) + O(\rho^2) \leq \delta\rho E(Y_l)E(Y_m) + O(\rho^2). \quad (4.7)$$

Note that, in (4.7), by slight abuse of our notation, we have used subscripts l, m to identify quantities (hazard rates, service times) identified with particular jobs. If we now aggregate the impact on returns from all $\binom{N}{2}$ pairwise interchanges, we infer from (4.7) that

$$\begin{aligned} E\{N(\nu^{SI} \mid \mathbf{L}_{..}, 0)\} - E\{N(\nu^{SD} \mid \mathbf{L}_{..}, 0)\} &\leq \delta\rho \sum_{l>m} E(Y_l) E(Y_m) + O(\rho^2) \\ &\leq \frac{\delta}{2}\rho E\{\mathbf{Y}(\mathbf{L}_{..})\}^2 + O(\rho^2). \quad (4.8) \end{aligned}$$

We now infer from (4.5), (4.6) and (4.8) that

$$\max_{\nu} E\{N(\nu \mid \mathbf{L}_{..}, 0)\} - \min_{\nu} E\{N(\nu \mid \mathbf{L}_{..}, 0)\} \leq \frac{\delta}{2}\rho E\{\mathbf{Y}(\mathbf{L}_{..})\}^2 + O(\rho^2)$$

and the bound for $CIC(\mathbf{L},0)$ now follows from (4.4). The bound for $RCIC(\mathbf{L},0)$ uses that for $CIC(\mathbf{L},0)$ together with the fact that

$$V^e(\mathbf{L},0) = N + O(\rho).$$

This concludes the proof. □

Remark 4.1. *If we suppose that service time Y_i has mean and variance μ_i and σ_i^2*

respectively then it is straightforward to show that the key quantity in the bounds given in Theorem 4.1 is given by

$$E_{\mathbf{L}_{..}|(\mathbf{L},0)}[E\{\mathbf{Y}(\mathbf{L}_{..})\}^2] = \sum_{j=1}^{J}\sum_{i=1}^{J} L_j p_{ij}(0)\left[\sigma_i^2 + \mu_i^2\{1 - p_{ij}(0)\}\right] - \sum_{j=1}^{J}\sum_{i\neq k} L_j p_{ij}(0) p_{kj}(0)\mu_i\mu_k.$$

It follows from the above theoretical results that when job lifetimes are exponentially distributed and, moreover, long, the cost of imperfect classification will be small for problems in which the key quantity $E(X_i)E(Y_i)$ varies little across distinct job types. We shall see in the upcoming numerical study that the insight afforded by these cases has much broader application. Note, for example, the following development of Example 4.1 above. We first introduce the quantity

$$R_{i_1 i_2} \equiv \frac{E(X_{i_1})E(Y_{i_1})}{E(X_{i_2})E(Y_{i_2})}$$

as the index ratio between job types i_1 and i_2.

Example 4.2. *In Example 4.1, should the service time for types of job 2 be decreased to* 0.16 *then* $R_{12} = R_{21} = 1$ *and the relative cost of imperfect classification is drastically reduced. See Figure 4.2, where the maximum of RCIC over the displayed range for the adjusted problem is now just* 2.32%.

We now consider further the insights afforded by the above material in the next section.

Figure 4.2: The relative costs $RCIC$ for a problem with $J = 2$, Weibull lifetimes and deterministic service times, and two different service rates of type 2 jobs.

4.3 The Cost of Imperfect Classification in the Worst Case - a Numerical Study

We shall now explore the role of the above index ratio by numerically investigating the behaviour of the key quantities CIC and $RCIC$ in the worst case, namely when $\epsilon_{ij} = J^{-1} \forall i, j$. We shall thus adopt a conservative viewpoint and ask questions about how much damage is done by a failure of the classification process to achieve anything better than random allocation of jobs to classes. The reader is referred back to the above definitions of $CIC(\mathbf{L}, 0)$ and $RCIC(\mathbf{L}, 0)$ around (4.1). Remember that in this worst case, the quantity $V^e(\mathbf{L}, 0)$ which plays a key role in (4.1), can be obtained by computing the Bayes' return for *any* service policy. In the computations below, we shall in fact use the static proposal π^S for this purpose.

One approach to the computation of the key quantity

$$E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{..}, 0\right)\right\}\right]$$

is to estimate it via repeated sampling from the multinomial conditional distribution for $\mathbf{L}_{..} \mid (\mathbf{L},0)$. We opted instead for an exact approach which computed $[\max_{\nu} E\{N(\nu \mid \mathbf{L}_{..}, 0)\}]$ for each $\mathbf{L}_{..}$ in the support of this distribution and then computed the exterior expectation. Remember that to calculate the number of expected service completions in the perfect classification situation, we need to know the initial number of jobs in each type. To derive them from $\mathbf{L}_{..}$, we define the following two vectors. One is $\mathbf{L}' \equiv (L'_1, \cdots, L'_j)$ the number of jobs of each type i, and the other $\mathbf{L}_{.\mathbf{j}} \equiv (L_{1j}, \cdots, L_{Jj})$ the number of class j jobs which are actually of type $i, 1 \leq i \leq J$. It is clear that $\mathbf{L}_{.j}$ has components which form a subset of those of $\mathbf{L}_{..}$ while $\mathbf{L}'$ can be obtained from the equation below.

$$\mathbf{L}' = \sum_{j=1}^{J} \mathbf{L}_{.j}. \tag{4.9}$$

We then have

$$E_{\mathbf{L}_{\cdot\cdot}|(\mathbf{L},0)}\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{\cdot\cdot}, 0\right)\right\}\right] = E_{\mathbf{L}'|(\mathbf{L},0)}\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}', 0\right)\right\}\right]$$
$$= E_{\mathbf{L}'|(\mathbf{L},0)}\left[V(\mathbf{L}',0)\right].$$

The inner quantity on the right hand side of the second equal sign can be computed via the optimality equations in (2.4) (page 36) in Chapter 2. To calculate the outer expectation, we need to compute the probability mass function of $\mathbf{L}' \mid (\mathbf{L},0)$, which is however a far from trivial task for problems of practical size. We first write down below the probability mass function of $\mathbf{L}_{\cdot j} \mid (\mathbf{L},0)$ (denoted as h_j) for class j, which has been shown to be multinomial (see (3.5)).

$$h_j(\mathbf{L}_{\cdot\mathbf{j}}) = \begin{cases} \frac{L_j!}{L_{ij}!\cdots L_{Jj}!} p_{1j}(0)^{L_{1j}} \cdots p_{Jj}(0)^{L_{Jj}}, \text{if} \sum_{i=1}^{J} L_{ij} = L_j, \\ 0, \text{otherwise.} \end{cases} \tag{4.10}$$

It then follows from (4.9) and (4.10) that H, the probability mass function of $\mathbf{L}' \mid (\mathbf{L},0)$, can be computed via a convolution of all the h_j, written as

$$H = h_1 * h_2 * \cdots * h_J. \tag{4.11}$$

Convolution of two or more probability mass functions of one dimensional discrete random variables is straightforward and a detailed account can be found in Grinstead and Snell [1997]. However, this is certainly not the case for multi-dimensional random variables, or vectors. In our problem, there are J multinomial random variables, each of which is a J dimensional vector whose components sum to L_j. The convolution of any two such distributions requires summing over the support of either of them. For each class j, the support is the set

$$\{(n_1, \cdots, n_J) \in \mathbb{N}^J | n_1 + \cdots + n_J = L_j\}. \tag{4.12}$$

Its number of elements is

$$\binom{L_j + J - 1}{L_j}, \tag{4.13}$$

which is exponential with regard to J and polynomial of degree $J-1$ with regard to L_j. Even for moderate J and L_j, the size of the support is already very large. To make the situation worse, the convolution needs to be performed recursively over all h_j.

In light of this, we explored the question of how this task should be approached for maximum computational efficiency. It has been found that the order of convolution makes a big difference in the run time. It is always preferable to convolve the variables with smaller support first, and in each convolution, to sum over the support with smaller size. Experiments show that the best convolution order could lead to a run time reduction of as much as 80% in some cases in which $J = 5$ and L_j is less than 10.

Remark 4.2. *The reader should note that even though the quantity*

$$\left[\max_{\nu} E\left\{N\left(\nu \mid \mathbf{L}_{..}, 0\right)\right\}\right]$$

is available in the numerical study reported here, it is challenging to obtain in general. Instead we can approximate it by computing returns for our heuristic policy π^{SF}, whose design is described in Section 2.2, in cases with perfect classification and whose initial state is summarised by $(\mathbf{L}_{..}, 0)$. We know from Li and Glazebrook [2010a] that the performance of π^{SF} is very close to optimal in such cases and that any underestimate of the quantities $CIC(\mathbf{L}, 0)$ and $RCIC(\mathbf{L}, 0)$ which results will be small.

Our numerical study considers problems with Weibull lifetimes and deterministic service times and $J = 2, 4$. Problems are created as in (3.45a)-(3.45i) above

except for the sampling of β_j, which is obtained as follows. Firstly we sample β_1 for the type 1 from one of (3.45d)-(3.45g), according to the specified category. The remaining β_i's are determined such that, conditioned on the sampled values of the α_i, S_i, and β_1 the index ratios R_{ii+1} are all equal to some R, say, where without loss of generality we take $R > 1$. We then check the relative length of service times and the lifetimes for every type other than the first one. If all of them are in the same category as of type 1, one problem instance is created. Otherwise the entire process is repeated. We take four cases for the setting of an $R - value$ which are

$$R \sim U[1, 1.1] \qquad \text{(Range 1)};$$

$$R \sim U[1.1, 1.5] \qquad \text{(Range 2)};$$

$$R \sim U[1.5, 2] \qquad \text{(Range 3)};$$

$$R \sim U[2, 4] \qquad \text{(Range 4)}.$$

Please note that Range 1(respectively, 4) allows the quantity $E(X_4)E(Y_4)$ to be between 1(respectively, 8) and 1.331(respectively, 64) times as large as $E(X_1)E(Y_1)$. We thus investigate a wide range of cases, including some in which the values of $E(X_i)E(Y_i)$ are nearly equal $\forall i$ to others in which there can be very large differences. As above, the p_i will be obtained by first sampling independently from $U[0.1, 0.9]$ and then normalising. However, we replace (3.45j)-(3.45l) by the choice $\epsilon_{ij} = J^{-1} \forall i, j$.

The computational effort needed here is very considerably greater than for the studies reported in Section 3.4. Happily, for the worst classification case we are considering the classification outcomes $\mathbf{L}$ are immaterial. Nonetheless, for each imperfect classification problem we need to solve to obtain $V^e(\mathbf{L}, 0)$, there are $\binom{N+J-1}{N}$ corresponding perfect classification problems whose solution contributes to the quantity $E_{\mathbf{L}_{..}|(\mathbf{L},0)}[\max_\nu E\{N(\nu \mid \mathbf{L}_{..}, 0)\}]$. This number increases very rapidly with N in the $J = 4$ cases.

For $J = 2$, and each category (A,B,C,D)/range $(1, 2, 3, 4)$ combination, 500

profiles were generated according to the above scheme, making 8,000 problems in total. Values of the relative costs $RCIC$ are presented in Figure 4.3 in the form of a boxplot for each of the 16 category/range combinations.

Computational demands are such that for the $J = 4$ study, the number of profiles for each category/range combination was reduced to 100. Hence 1,600 problems with $J = 4$ were investigated. The results are given in Figure 4.4.

It is shown very clearly in both figures that, within each category, the values of $RCIC$ do indeed increase markedly with R. Take Category D in Figure 4.3 as an example. The average and the maximum $RCIC$ for Range 1 is just 0.68% and 3.05%, respectively. The values are still quite small for Range 2, with a mean of 0.98% and a maximum of 4.98%. However, when R continues to increase to the next two ranges, the $RCIC$ values can get quite significant. The mean is 3.65% in Range 4, with a maximum as large as 10.52%. This increase is rather more dramatic for the $J = 4$ cases. This is as to be expected since the latter cases accommodate much greater variability in the type-specific values $E(X_i)E(Y_i)$.

Another clear feature is the tendency of $RCIC$ to decrease as lifetimes grow. To see why this might be expected, observe that in the limit in which no jobs are lost (lifetimes are infinite), all jobs will ultimately be served and no costs incurred by any misclassification. The only exception is Category A, which has smaller $RCIC$ than category B in most cases. This is not surprising as category A jobs have very short lifetimes. Most of them abandon the system in the very early stage and thus the impact of the misclassification is small.

The median values for $RCIC$ are considerably larger for the $J = 4$ cases , reflecting the fact that misclassification has more impact when the number of job types is greater. That the maximum values are nevertheless reduced for $J = 4$ is almost certainly due to the fact that the initial number of jobs in these problems is rather smaller (10 rather than 20), depressing the variability of the outcomes.

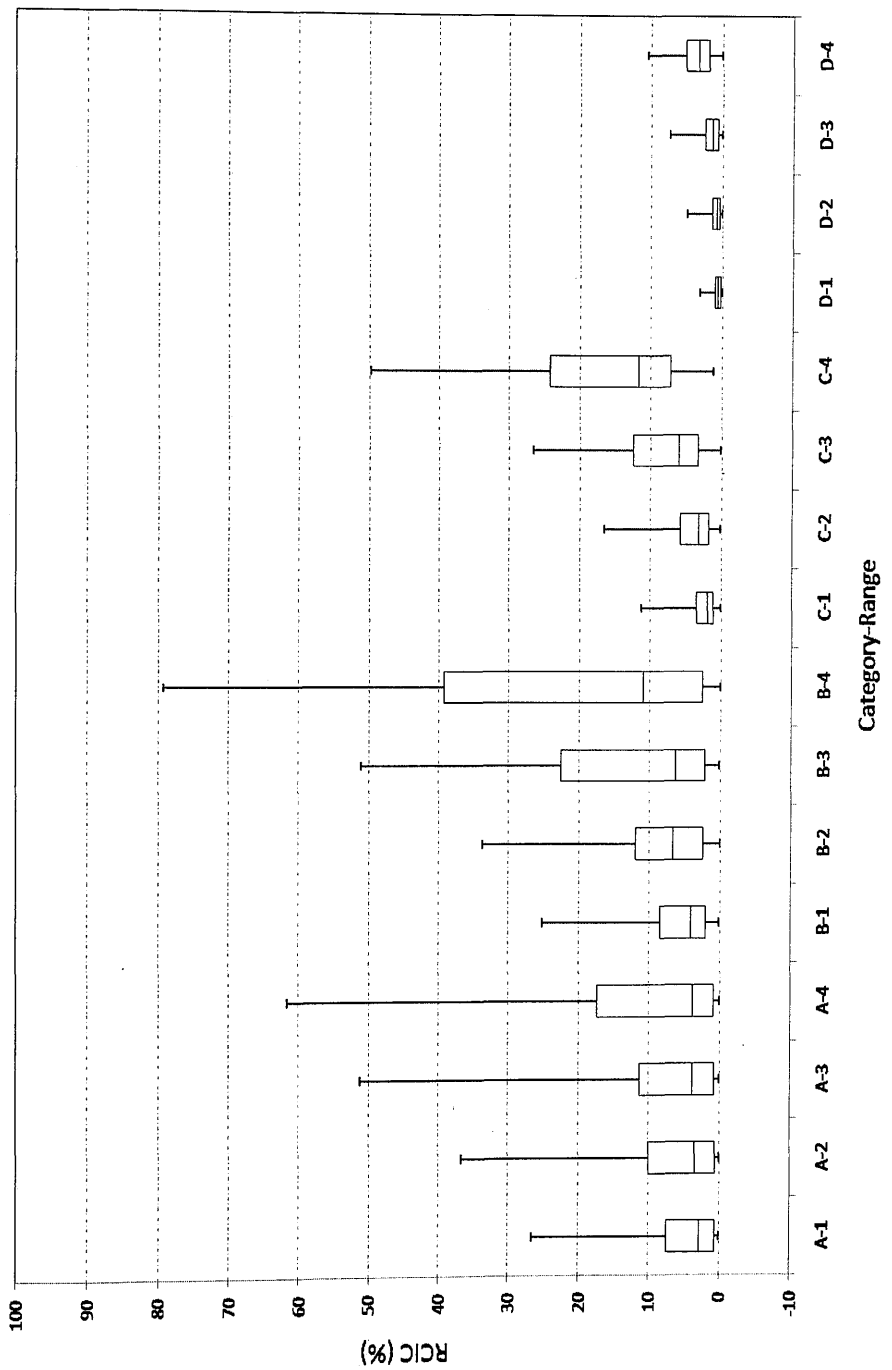

Figure 4.3: Boxplot of the worst case relative costs $RCIC$ for Weibull lifetimes and deterministic service times when $J = 2$.

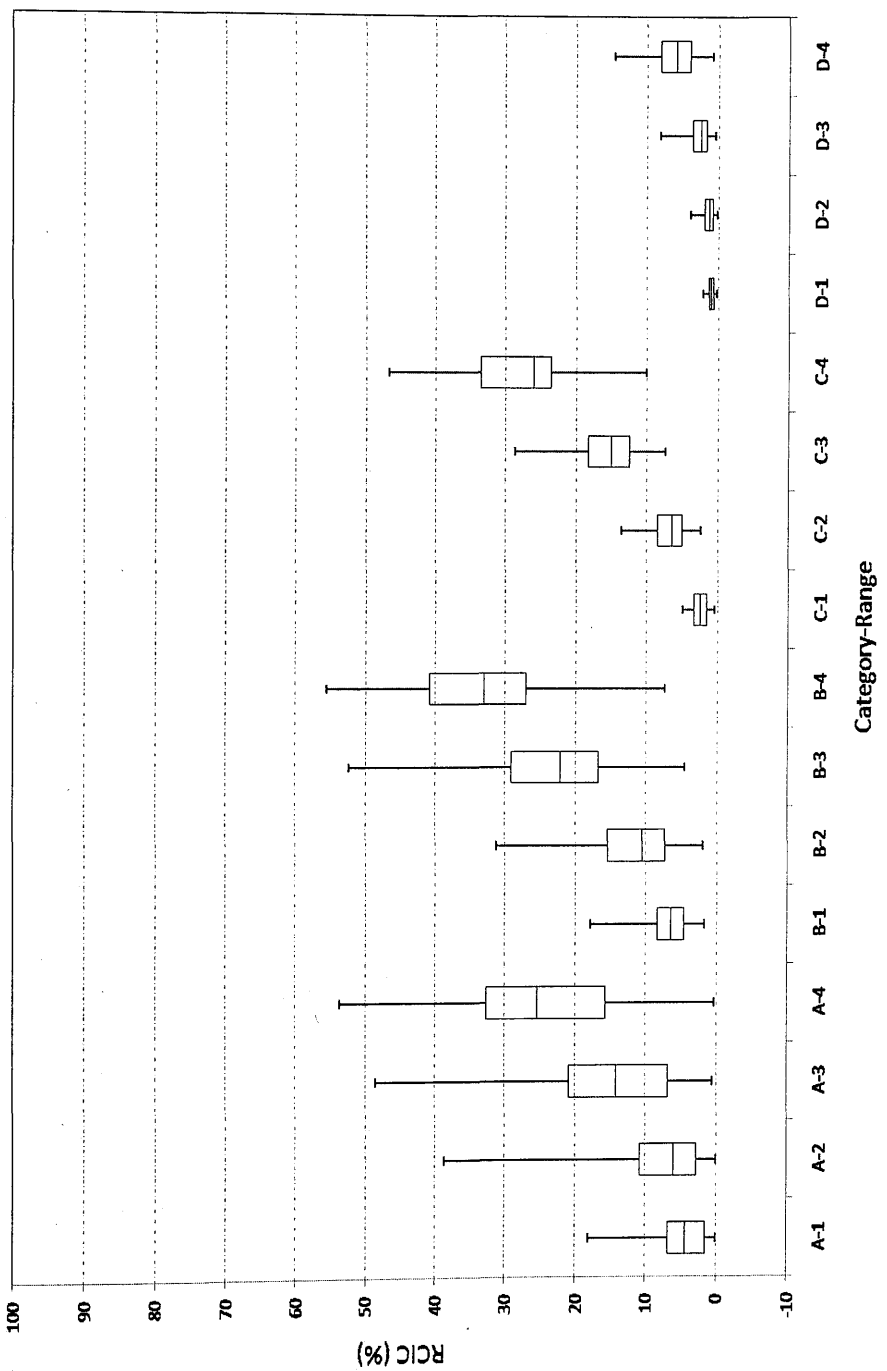

Figure 4.4: Boxplot of the worst case relative costs $RCIC$ for Weibull lifetimes and deterministic service times when $J = 4$.

4.4 Conclusion

In this chapter we introduce a quantity $RCIC$ to measure the cost of imperfect classification. An upperbound for $RCIC$ is then developed for problems with exponentially distributed lifetimes. It follows that for such problems the value of $RCIC$ is small when jobs' lifetimes are long and when the index ratio R_{i1i2} is close to one across all job types. In light of this result, a comprehensive numerical study is conducted to explore the behaviour of $RCIC$ in a more general scenario in which lifetimes are Weibull and service times are deterministic. We focus on the worst case in which the classification process randomly allocates jobs to classes. It has been found that $RCIC$ does increase significantly as the index ratio increases from one in all the problem instances considered. Further, the misclassification has more impact when there are more distinct job types, in which cases more uncertainty is accommodated. An interesting observation is that the cost tends to decrease with jobs' lifetimes. This can be understood by considering a limit situation in which all jobs have infinite lifetimes and there will be no cost incurred by any misclassification as all of them will be served sooner or later.

It is worth mentioning that in the cases where $RCIC$ is small, the optimal policy that takes decisions based on the observed state will be almost as good as the optimal policy that is able to take decisions based on the unobservable true state. The classification errors thus hardly have impact on effective scheduling of the system. This result extends to any policy if $RCIC$ is small even in the worst case. In such situations, an arbitrary policy will achieve the same good results.

Chapter 5

Conclusions and Future Research

5.1 Summary and Conclusions

We consider in this thesis the scheduling of impatient jobs in a clearing system, which is originally motivated by the medical resource management problem after MCIs. In such situations the injuries significantly and suddenly overwhelm the available resource. To support efficient resource allocation, all the patients are subject to an initial triage and are placed into distinct classes based on the severity of their conditions. Following triage, the central challenge is to develop effective service policies such that the expected number of successful treatments is maximized.

A simple single server version of this problem is addressed in Chapter 2, where it is modelled as a SMDP and explicit optimality equations are constructed by means of which optimal policies can in principle be developed. Sadly, this exact DP approach is not a realistic option for problems of practical size. The computational complexity increases exponentially fast with respect to the problem size. We opt instead to develop effective heuristic service policies by implementing a single step approximate policy improvement algorithm to a static permutation policy π^S which was proposed by Glazebrook et al. [2004]. The value functions of π^S are well approximated by a fluid model, which is a deterministic analogue of the stochastic system when operated under π^S. It has a very simple form and can be solved

trivially. This feature is essential for the efficiency of our heuristic algorithm and the effectiveness of the resulting policy.

An extensive numerical study has been conducted to explore the performance of our proposed heuristic policy. In the scenario of exponential lifetimes and service times, and that of Weibull lifetimes and deterministic service times, we are able to develop optimal policies and thus to evaluate the heuristics' performance by means of reward suboptimality. We also consider a more challenging scenario of Weibull lifetimes and exponential service times. In this case it has not proved possible to compute optimal policies (even for fairly small problems) in reasonable time. A simulation study is carried out instead to compare the performance of competing heuristics. It is shown clearly that, in all the three scenarios, our heuristic works robustly well. It comfortably outperforms the two alternative heuristics proposed in the literature (one is π^S) in most problem instances.

This problem is extended in Chapter 3 to accommodate classification errors. Jobs placed into one class could actually have many different characteristics. This is especially the case for triage after MCIs, which has been shown in the literature to be subject to significant levels of error. Due to this additional uncertainty, the resulting problem is substantially more complex. We propose a simple analytical model and adopt a Bayesian approach to deal with the uncertainty arising from possible misclassification. This Bayesian sequential decision problem is then formulated as a dynamic program and optimal policies can in principle be developed by standard DP methods. However, this is again infeasible for a wide range of problems. In light of the results from Chapter 2, we implement a single policy improvement step to an adapted version of policy π^S. We successfully extend the fluid approach to generate high quality approximations to the value functions of the adapted π^S.

The resulting heuristic policy is subject to a similar numerical study to that of Chapter 2. Due to the complexity of the error-prone triage problem, we are only able to test the scenario of Weibull lifetimes and deterministic service times.

Numerical results indicate that our proposed heuristic policy has robustly strong performance in all problem instances considered. An interesting observation is that the suboptimalities of all the heuristic policies considered in this numerical analysis tend to decrease with the classification errors. Indeed, in the worst case in which the triage process randomly allocates jobs to classes, all heuristics have the same performance as the optimal policy. This is however not good news at all. Instead, it simply means that even the optimal policy cannot achieve good results in these cases. The only way to improve the performance is to make the classification more accurate.

However, there are some special cases in which effective scheduling is not reliant upon an accurate initial triage. The issue is explored in detail in Chapter 4. We first propose a measure to quantify the cost incurred by classification errors. An analytical upperbound is then established for exponential lifetime cases. This upperbound approaches zero when the lifetimes are long and when the index ratio R_{i1i2} is close to one across all jobs. To explore the behaviour of the cost, a worst case numerical study is conducted for problems of Weibull lifetimes and deterministic service times. It is shown that the cost does decrease rapidly as the index ratio approaches one from above in all problem instances. Moreover, the misclassification has more impact when there are more job types. This is not surprising as greater variability is accommodated in these cases. Finally, the cost decreases with jobs' lifetimes. To understand this, observe that in a no loss limit, all jobs will be served eventually and no cost is incurred by any misclassification.

5.2 Future Research

More research is needed to investigate the behaviour of the cost of imperfect classification in the exponential service time scenario. Because of the computational complexity exact optimal policies cannot be developed in these cases and thus $(R)CIC$ is not available. However, it can be approximated by the following method. It has

been shown in Chapter 2 and 3 that under both perfect and imperfect classification, our proposed heuristic policy has very strong performance and the deviation from optimal is very small. Therefore, a close approximation to CIC is given by the following quantity,

$$E_{\mathbf{L}_{..}|(\mathbf{L},0)}\left[V_{\pi^{SF}}\left(\mathbf{L}_{..},0\right)\right]-V^{e}_{\pi^{SF}}\left(\mathbf{L},0\right).$$

Unfortunately, in the Weibull lifetime and exponential service time scenario, $V_{\pi^{SF}}\left(\mathbf{L}_{..},0\right)$ and $V^{e}_{\pi^{SF}}\left(\mathbf{L},0\right)$ are not available either. A possible way is to estimate them by Monte Carlo simulation. The quantity $V_{\pi^{SF}}\left(\mathbf{L}_{..},0\right)$ can be readily obtained as we have already done this in Scenario (III), Section 2.3, Chapter 2. The main work then required is to simulate the true states from the observed states, and to calculate $V^{e}_{\pi^{SF}}\left(\mathbf{L},0\right)$ for every sampled true state.

A key problem feature in this thesis is that all jobs are present at time zero and there are no new arrivals into the system. This is indeed the case (at least approximately) for the triage problem in the aftermath of MCIs, as all the injuries are present immediately after the incident. However, it does take time to identify all of them. The triage process and the following treatment would start straightaway rather than wait until all injuries are collected. In this regard, one possible topic for future research could be the incorporation of an incoming stream with a time dependent arrival rate that decreases gradually to zero.

A practical research topic could address the hospital Accident & Emergency triage problem. In this problem, patients must be seen within a few hours (say 4) after arrival or they have to be admitted to hospital. The cost incurred in the latter case is significantly more than the former. Therefore, one major objective of the A&E triage is to maximize the number of patients attended within 4 hours. Different from the triage problem in the aftermath of MCIs, new patients come to A&E continuously and randomly, and hence multiple stochastic arrival processes must be considered. The lifetimes are however known and deterministic in this case.

We conclude this section with a very interesting and challenging future research problem. Consider a scenario in which each impatient job requires a range of different resource(s). Take the medical service as an example. Some patients may need a doctor, a nurse and an anaesthetist at the same time to complete the treatment, while some others may need only a doctor and/or a nurse. There are some multifunctional staff who can play different roles, while the rest can only perform a single specialised function. The question is how to allocate staff to patients and in what sequence to achieve the maximum number of expected service completions. A promising method to deal with this very complex problem is to model each job as a restless bandit process and then the rich theory of Whittle Indices (Whittle [1988]) applies. The recent extension of Gittins Index heuristics to accommodate more general resource distribution among multiarmed bandits (Glazebrook and Minty [2009]) may shed some light on this issue. Note that if all patients need only one and the same type of resource, this problem reduces to a multi-server version of the problem considered in Chapter 2.

Bibliography

N. T. Argon and S. Ziya. Priority assignment under imperfect information on customer type identities. Manufacturing and Service Operations Management, 11(4):674–693, 2009.

N. T. Argon, S. Ziya, and R. Righter. Scheduling impatient jobs in a clearing system with insights on patients triage in mass casualty incidents. Probability in the Engineering and Informational Sciences, 22(3):301–332, 2008.

J. L. Arnold, M.-C. Tsai, P. Halpern, H. Smithline, E. Stok, and G. Ersoy. Mass-casualty, terrorist bombings: Epidemiological outcomes, resource utilization, and time course of emergency needs (Part I). Prehospital and Disaster Medicine, 18:220–234, 2004.

K. S. Azoury. Bayes solution to dynamic inventory models under unknown demand distribution. Management Science, 31(9):1150–1160, 1985.

A. Bassamboo, J. M. Harrison, and A. Zeevi. Dynamic routing and admission control in high-volume service systems: asymptotic analysis via multi-scale fluid limits. Queueing Systems, 51:249–285, 2005.

A. Bassamboo, J. M. Harrison, and A. Zeevi. Design and control of a large call center: Asymptotic analysis of an LP-based method. Operations Research, 54 (3):419–435, 2006.

J. C. Bean, J. R. Birge, and R. L. Smith. Aggregation in dynamic-programming. Operations Research, 35(2):215–220, 1987.

R. Bellman. Adaptive Control Processes: A Guided Tour. Princeton University Press, Princeton, NJ, 1961.

R. Bellman and S. Dreyfus. Functional approximxations and dynamic programming. Mathematical Tables and Other Aids to Computation, 13:247–251, 1959.

L. Benkherouf, K. D. Glazebrook, and R. W. Owen. Gittins indices and oil exploration. Journal of the Royal Statistical Society. Series B (Methodological), 54 (1):229–241, 1992.

A. Benveniste, M. Metivier, and P. Priouret. Adaptive algorithms and stochastic approximations. Springer-Verlag, New York, 1990.

D. P. Bertsekas and D. A. Castanon. Adaptive aggregation methods for infinite horizon dynamic-programming. IEEE Transactions on Automatic Control, 34 (6):589–598, 1989.

D. P. Bertsekas and J. N. Tsitsiklis. Neuro-Dynamic Programming. Athena Scientific, Belmont, Massachusetts, 1996.

P. P. Bhattacharya and A. Ephremides. Optimal scheduling with strict deadlines. IEEE Transactions on Automatic Control, 34(7):721–728, 1989.

P. P. Bhattacharya and A. Ephremides. Optimal allocation of a server beween two queues with due times. IEEE Transactions on Automatic Control, 36:1417–1423, 1991.

S. Bhulai. Dynamic routing policies for multiskill call centers. Probability in the Engineering and Informational Sciences, 23(1):101–119, 2009.

S. Bhulai and G. Koole. On the structure of value functions for threshold policies in queueing models. Journal of Applied Probability, 40(3):613–622, 2003.

O. J. Boxma and F. G. Forst. Minimizing the expected weighted number of tardy jobs in stochastic flow shops. Operations Research Letters, 5(3):119–126, 1986.

J. A. Boyan and M. L. Littman. Exact solutions to time-dependent MDPs. In T. K. Leen, T. G. Dietterich, and V. Tresp, editors, Advances in Neural Information Processing Systems 13, volume 13 of Advances in Neural Information Processing Systems, pages 1026–1032. 2001.

R. J. Boys, K. D. Glazebrook, and D. J. Laws. A class of Bayes-optimal two-stage screens. Naval Research Logistics, 43(8):1109–1125, 1996a.

R. J. Boys, K. D. Glazebrook, and C. M. McCrone. A Bayesian model for the optimal ordering of a collection of screens. Biometrika, 83(2):472–476, 1996b.

X. Cai, X. Wu, and X. Zhou. Stochastic scheduling subject to preemptive-repeat breakdowns with incomplete information. Operations Research, 57(5):1236–1249, 2009.

A. R. Cassandra. Exact and Approximate Algorithms for Partially Observable Markov Decision Processes. PhD thesis, 1998.

W. Chen, D. Huang, A. Kulkarni, J. Unnikrishnan, Q. Zhu, P. Mehta, S. Meyn, and A. Wierman. Approximate dynamic programming using fluid and diffusion approximations with applications to power management. In 48th IEEE Conference on Decision and Control, 2009.

D. Choi and B. Van Roy. A generalized Kalman filter for fixed point approximation and efficient temporal-difference learning. Discrete Event Dynamic Systems, 16 (2):207–239, 2006.

J. Y. Choi and S. Reveliotis. Relative value function approximation for the capacitated re-entrant line scheduling problem. IEEE Transactions on Automation Science and Engineering, 2(3):285–299, 2005.

R. H. Crites and A. G. Barto. Elevator group control using multiple reinforcement learning agents. Machine Learning, 33(2):235–262, 1998.

P. Dayan. The convergence of TD(λ) for general λ. Machine Learning, 8(3): 341–362, 1992.

P. Dayan and T. J. Sejnowski. TD(λ) converges with probability 1. Machine Learning, 14(3):295–301, 1994.

D. P. De Farias and B. Van Roy. On the existence of fixed points for approximate value iteration and temporal-difference learning. Journal of Optimization Theory and Applications, 105(3):589–608, 2000.

D. P. De Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. Operations Research, 51(6):850–865, 2003.

D. P. De Farias and B. Van Roy. On constraint sampling in the linear programming approach to approximate dynamic programming. Mathematics of Operations Research, 29(3):462–478, 2004.

L. Decreusefond and P. Moyal. Fluid limit of a heavily loaded EDF queue with impatient customers. Markov Process and Related Fields, 14:131–158, 2006.

B. Doytchinov, J. P. Lehoczky, and S. Shreve. Real-time queues in heavy traffic with earliest-deadline-first queue discipline. The Annals of Applied Probability, 11(2):332–378, 2001.

H. Emmons and M. Pinedo. Scheduling stochastic jobs on parallel machines with due dates. European Journal of Operational Research, 47:49–55, 1990.

A. Fog. Pseudo random number generators - uniform and non-uniform distributions. <http://www.agner.org/random/>, Last modified: Aug 03, 2010.

E. R. Frykberg. Medical management of disasters and mass casualties from terrorist bombings: How can we cope? The Journal of Trauma, 53(2):201–212, 2002.

E. R. Frykberg and J. J. Tepas. Terrorist bombings. Lessons learned from Belfast to Beirut. Annals of Surgery, 208(5):569–576, 1988.

N. Furukawa. Fundamental theorems in a Bayes controlled process. Bulletin of Mathematical Statistics, 14:103–110, 1970.

A. Gajrat and A. Hordijk. Fluid approximation of a controlled multiclass tandem network. Queueing Systems, 35:349–380, 2000.

O. Garnett, A. Mandelbaum, and M. I. Reiman. Designing a call center with impatient customers. Manufacturing and Service Operations Management, 4: 208–227, 2002.

D. P. Gaver, P. A. Jacobs, G. Samorodnitsky, and K. D. Glazebrook. Modeling and analysis of uncertain time-critical tasking problems. Naval Research Logistics, 53:588–599, 2006.

J. C. Gittins and K. D. Glazebrook. Bayesian models in stochastic scheduling. Journal of Applied Probability, 14(3):556–565, 1977.

J. C. Gittins and D. M. Jones. A dynamic allocation index for the sequential design of experiments. Progress in Statistics, pages 241–266, North-Holland, Amsterdam, 1974.

K. D. Glazebrook. On the optimal allocation of two or more treatments in a controlled clinical trial. Biometrika, 65(2):335–340, 1978.

K. D. Glazebrook. On stochastic scheduling problems with due dates. International Journal of Systems Science, 14(11):1259–1271, 1983.

K. D. Glazebrook and R. J. Boys. A class of Bayesian models for optimal exploration. Journal of the Royal Statistical Society, Series B (Methodological), 57 (4):705–720, 1995.

K. D. Glazebrook and R. Minty. A generalized Gittins index for a class of multi-armed bandits with general resource requirements. Mathematics of Operations Research, 34(1):26–44, 2009.

K. D. Glazebrook and H. M. Mitchell. An index policy for a stochastic scheduling model with improving/deteriorating jobs. Naval Research Logistics, 49:706–721, 2002.

K. D. Glazebrook and R. W. Owen. On the value of adaptive solutions to stochastic scheduling problems. Mathematics of Operations Research, 20(1):65–89, 1995.

K. D. Glazebrook and E. L. Punton. Dynamic policies for uncertain time-critical tasking problems. Naval Research Logistics, 55:142–155, 2008.

K. D. Glazebrook, P. S. Ansell, R. T. Dunn, and R. R. Lumley. On the optimal allocation of service to impatient tasks. Journal of Applied Probability, 41: 51–72, 2004.

K. D. Glazebrook, C. Kirkbride, and J. Ouenniche. Index policies for the admission control and routing of impatient customers to heterogeneous service stations. Operations Research, 57(4):975–989, 2009.

C. M. Grinstead and J. L. Snell. Introduction to probability. AMS Bookstore, 2nd edition, 1997.

C. Guestrin, D. Koller, R. Parr, and S. Venkataraman. Efficient solution algorithms for factored MDPs. Journal of Artificial Intelligence Research, 19:399–468, 2003.

T. Hamada and K. D. Glazebrook. A Bayesian sequential single-machine scheduling problem to minimize the expected weighted sum of flowtimes of jobs with exponential processing times. Operations Research, 41(5):924–934, 1993.

T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning: Data Mining, Inference, and Prediction. Series in Statistics. Springer, New York, NY, 2nd edition, 2009.

S. Helber and K. Henken. Profit-oriented shift scheduling of inbound contact centers with skills-based routing, impatient customers, and retrials. OR Spectrum, 32(1):109–134, 2010.

K. Hinderer. Foundations of non-stationary dynamic programming with discrete time parameter. Springer-Verlag, New York, 1970.

T. Jaakkola, M. I. Jordan, and S. P. Singh. On the convergence of stochastic iterative dynamic programming algorithms. Neural Computation, 6(6):1185–1201, 1994.

Z. Jiang, T. G. Lewis, and J. Y. Colin. Scheduling hard real-time constrained periodic tasks on multiple processors. Journal of Systems Software, 19:102–118, 1996.

P. W. Keller, S. Mannor, and D. Precup. Automatic basis function construction for approximate dynamic programming and reinforcement learning. In Proceedings of the 23rd international conference on Machine learning, pages 449–456, Pittsburgh, Pennsylvania, 2006. ACM.

P. R. Kumar. A survey of some results in stochastic adaptive control. SIAM Journal on Control and Optimization, 23(3):329–380, 1985.

H. J. Kushner and D. S. Clark. Stochastic approximation methods for constrained and unconstrained systems. Springer-Verlag, Berlin, 1978.

T. J. Lambert III, M. A. Epelman, and R. L. Smith. Aggregation in stochastic dynamic programming. Technical Report 04-07, University of Michigan, 2004.

D. S. Leslie and E. J. Collins. Individual Q-learning in normal form games. SIAM Journal on Control and Optimization, 44(2):495–514, 2005.

K. Levy, F. J. Vazquez-Abad, and A. Costa. Adaptive stepsize selection for online Q-learning in a non-stationary environment. WODES 2006: Eighth International Workshop on Discrete Event Systems, Proceedings. IEEE, New York, 2006.

D. Li and K. D. Glazebrook. An approximate dynamic programing approach to

the development of heuristics for the scheduling of impatient jobs in a clearing system. Naval Research Logistics, 57(3):225–236, 2010a.

D. Li and K. D. Glazebrook. A Bayesian approach to the triage problem with imperfect classication. Submitted, 2010b.

L. Li and M. L. Littman. Lazy approximation for solving continuous finite-horizon mdps. In Twentieth National Conference on Artificial Intelligence, 2005.

R. E. Lillo. Optimal control of an M/G/1 queue with impatient priority customers. Naval Research Logistics, 48:201–209, 2001.

A. Mandelbaum, W. A. Massey, and M. I. Reiman. Strong approximations for Markovian service networks. Queueing Systems, 30:149–201, 1998.

P. Marbach, O. Mihatsch, and J. N. Tsitsiklis. Call admission control and routing in integrated services networks using neuro-dynamic programming. IEEE Journal on Selected Areas in Communications, 18(2):197–208, 2000.

J. Marecki, Z. Topol, and M. Tambe. A fast analytical algorithm for MDPs with continuous state spaces. In AAMAS-06 Proceedings of 8th Workshop on Game Theoretic and Decision Theoretic Agents, 2006.

J. J. Martin. Bayesian Decision Problems and Markov Chains. John Wiley, New York, 1967.

M. Matsumoto and T. Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modeling and Computer Simulation, 8(1):3–30, 1998.

C. Moallemi, S. Kumar, and B. V. Roy. Approximate and data-driven dynamic programming for queueing networks. Technical report, Graduate School of Business, Columbia University, 2006.

G. E. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. Management Science, 28(1):1–16, 1982.

J. R. Morrison and P. R. Kumar. New linear program performance bounds for queueing networks. Journal of Optimization Theory and Applications, 100(3): 575–597, 1999.

A. Movaghar. Optimal control of parallel queues with impatient customers. Performance Evaluation, 60:327–343, 2005.

M. Opp, K. Glazebrook, and V. G. Kulkarni. Outsourcing warranty repairs: Dynamic allocation. Naval Research Logistics, 52:381–396, 2005.

T. J. Ott and K. R. Krishnan. Separable routing: A scheme for state-dependent routing of circuit switched telephone traffic. Annals of Operations Research, 35 (1):43–68, 1992.

S. S. Panwar, D. Towsley, and J. K. Wolf. Optimal scheduling policies for a class of queues with customer deadlines to the beginning of service. Journal of the Association for Computing Machinery, 35(4):832–844, 1988.

M. L. Pinedo. Stochastic scheduling with release dates and due dates. Operations Research, 31(3):559–572, 1983.

M. L. Pinedo. Scheduling: Theory, Algorithms, and Systems. Springer, 3rd edition, 2008.

W. B. Powell. Approximate Dynamic Programming: Solving the curses of dimensionality. John Wiley and Sons, Hoboken, New Jersey, 2007.

P. Preux, S. Girgin, M. Loth, and Ieee. Feature Discovery in Approximate Dynamic Programming. ADPRL: 2009 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning. IEEE, New York, 2009.

M. L. Puterman. Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons, New York, 1994.

D. Reetz. Approximate solutions of a discounted Markovian decision process. Bonner Mathematische Schriften, 98:77–92, 1977.

U. Rieder. Bayesian dynamic programming. Advances in Applied Probability, 7 (2):330–348, 1975.

D. Roubos and S. Bhulai. Average-cost approximate dynamic programming for the control of birth-death processes. Technical report, VU University Amsterdam, NL, 2007.

D. Roubos and S. Bhulai. Approximate dynamic programming techniques for the control of time-varying queueing systems applied to call centers with abandonments and retrials. Probability in the Engineering and Informational Sciences, 24:27–45, 2010.

J. K. Satia and R. E. Lave. Markovian decision processes with uncertain transition probabilities. Operations Research, 21(3):728–740, 1973.

Y. Sawaragi and T. Yoshikawa. Discrete-time Markovian decision processes with incomplete state observation. The Annals of Mathematical Statistics, 41(1): 78–86, 1970.

P. J. Schweitzer and A. Seidmann. Generalized polynomial approximations in Markovian decision processes. Journal of Mathematical Analysis and Applications, 110(2):568–582, 1985.

S. Shakkottai and R. Srikant. Scheduling real-time traffic with deadlines over a wireless channel. Wireless Network, 8(1):13–26, 2002.

S. Singh, T. Jaakkola, M. L. Littman, and C. Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. Machine Learning, 38 (3):287–308, 2000.

S. S. Singh, T. Jaakkola, and M. I. Jordan. Reinforcement learning with soft state aggregation. In G. Tesauro, D. Touretzky, and T. K. Leen, editors, Advances in Neural Information Processing Systems 7. MIT Press, Cambridge, MA, 1995.

E. J. Sondik. The optimal control of partially observable Markov processes over the infinite horizon: Discounted costs. Operations Research, 26(2):282–304, 1978.

R. Sutton and A. Barto. Reinforcement learning. The MIT Press, Cambridge, Massachusetts, 1998.

R. S. Sutton. Temporal Credit Assignment in Reinforcement Learning. PhD thesis, 1984.

R. S. Sutton. Learning to predict by the methods of temporal differences. Machine Learning, 3(1):9–44, 1988.

G. Tesauro. Practical issues in temporal difference learning. Machine Learning, 8 (3-4):257–277, 1992.

S. B. Thrun. The role of exploration in learning control. In D. A. White and D. A. Sofge, editors, Handbook of intelligent control: neural, fuzzy, and adaptive approaches. Van Nostrand Reinhold, New York, NY, 1992.

H. C. Tijms. Stochastic Models: an Algorithmic Approach. John Wiley and Sons, Chichester, 1994.

M. Trick and S. Zin. A linear programming approach to solving stochastic dynamic programs. unpublished manuscript, 1993.

J. N. Tsitsiklis. Asynchronous stochastic approximation and Q-learning. Machine Learning, 16(3):185–202, 1994.

J. N. Tsitsiklis and B. Van Roy. Feature-based methods for large scale dynamic programming. Machine Learning, 22(1):59–94, 1996.

J. N. Tsitsiklis and B. Van Roy. Average cost temporal difference learning. Technical Report Laboratory for Information and Decision Systems, LIDS-P-2390, Massachusetts Institute of Technology, 1997.

F. Turégano-Fuentes, D. Pérez-Díaz, M. Sanz-Sánchez, and J. Ortiz Alonso. Overall asessment of the response to terrorist bombings in trains, Madrid, 11 March 2004. European Journal of Trauma and Emergency Surgery, 34(5):433–441, 2008.

S. P. Van der Zee and H. Theil. Priority assignment in waiting-line problems under conditions of misclassification. Operations Research, 9:875–885, 1961.

K. M. Van Hee. Bayesian control of Markov chains. Mathematical Center Tracts 95. Mathematisch Centrum, Amsterdam, 1978.

J. Van Mieghem. Due-date scheduling: asymptotic optimality of generalized longest queue and generalized largest delay rules. Operations Research, 51(1): 113–122, 2003.

M. H. Veatch. Approximate dynamic programming for networks: Fluid models and constraint reduction. Technical report, Department of Mathematics, Gordon College, 2009.

K. H. Waldmann. On bounds for dynamic programs. Mathematics of Operations Research, 10(2):220–232, 1985.

A. R. Ward and S. Kumar. Asymptotically optimal admission control of a queue with impatient customers. Mathematics of Operations Research, 33(1):167–202, 2008.

C. J. C. H. Watkins and P. Dayan. Q-learning. Machine Learning, 8(3):279–292, 1992.

W. Whitt. Approximations of dynamic programs. Mathematics of Operations Research, 3(3):231–243, 1978.

W. Whitt. Fluid model for multiserver queues with abandonments. Operations Research, 54(1):37–54, 2006.

P. Whittle. Multi-armed bandits and the Gittins index. Journal of the Royal Statistical Society, Series B (Methodological), 42(2):143–149, 1980.

P. Whittle. Restless bandits: Activity allocation in a changing world. In Applied Probability Trust. 1988.

Y. J. Zhao, S. D. Patek, and P. A. Beling. Decentralized Bayesian search using approximate dynamic programming methods. IEEE Transactions on Systems Man and Cybernetics Part B-Cybernetics, 38(4):970–975, 2008.

Z.-X. Zhao, S. S. Panwar, and D. Towsley. Queueing performance with impatient customers. Proceedings of IEEE INFOCOM91, 1:400–409, 1991.

Appendices

Appendix A

Contents in the Accompanying CD

The main directory in the CD is *".\DSICS"*, which includes all the C++ header files (*.h*) and source files (*.cpp*). They are organized into a Microsoft Visual C++ project, which can be accessed by double clicking the file *DSICS.sin*.

Other files include the parameter file *controlParas.dat*, the binomial coefficient data file *combMatrix.dat*, and a sample input data file *wd_sample.dat*. For the reader's reference, we have also copied two resource files into this folder. One is *libcalcProb.lib* which is generated by the method mentioned in Appendix D, and the other a Matlab run time library, *mclmcrrt.lib*. Both resource files must be included into the project.

There are two other data files (*regression.dat, fluidApp.dat*) in this directory, but they are not actually used. However they cannot be deleted as the program still reads them in the initialization.

The second directory is *".\problem instances"*, which contains all the problem instance data files used in this thesis. The file format must be strictly followed for any new instances. Note that the selected data file needs to be copied over to the working folder before kicking off a solve.

The third directory is *".\matlabFunctions"*, for the three Matlab functions to be mentioned in Appendix D.

The last directory is *".\thesis"*, which contains an electronic copy of this thesis in pdf format.

Appendix B

Instructions to Use HPC

The following steps can be followed to use HPC.

1. Log into HPC.
2. Copy all the header files, source files, data files and Matlab files from corresponding directories in the accompanying CD into HPC fileserver.
3. Load Matlab module and launch it. Execute the command to be mentioned in Appendix D to compile all the Matlab functions into a C++ library.
4. Still in Matlab, execute the following command to build an executable: *mbuild CuImprove.cpp ExpoExpo.cpp WeibDetermnc.cpp WeibExpo.cpp BayesianWeibDeter.cpp convolution.cpp mersenne.cpp SimuPolicies.cpp stdafx.cpp -L. -lcalcProb -I. -output triage.*
5. Change parameter values in *controlParas.dat* for the selected problem instance.
6. Create a batch job control script and submit jobs by issuing a *qsub* command. Remember to load Matlab and Boost modules in the script.
7. More information can be found in http://www.lancs.ac.uk/iss/hpc/.

Appendix C

C++ Code for Key Classes and Functions

This appendix contains a selection of C++ code for the key classes and functions in this thesis. They are organized in the following order.

- *ExpoExpo.cpp*: the source file for the exponential lifetime and service time scenario. The key functions which are included in this appendix are:
  - *createStateSpace()*: create all the states.
  - *probCalculation()*: calculate transition probabilities.
  - *altFluidApprxForThetaMu()*: fluid model in this scenario.
  - *policyImprovementFluid()*: the approximate single step policy improvement algorithm using the fluid model.
  - *geneOptActions()*: development of the optimal policy.
- *WeibDetermnc.cpp*: the source file for the Weibull lifetime and deterministic service time scenario. The key functions included in this appendix are:
  - *geneDeciEpochs()*: generate all possible decision epochs.
  - *calcAllMRLTs()*: calculate MRL for all job classes at all decision epochs.

- ***WeibExpo.cpp***: the source file for the Weibull lifetime and exponential service time scenario. The key function is:
  - ***calcBySimulation()***: simulation steps for this scenario.
- ***BayesianWeibDeter.cpp***: the source file for the Weibull lifetime and deterministic service time scenario in the imperfect classification situation. The key functions included in this appendix are:
  - ***calcPostProbs()***: calculate the prior and the posterior probabilities.
  - ***calcRemSurProb()***: calculate remaining survival probabilities.
- ***CuImprove.cpp***: the main function. It is where the program starts to execute. This code should be the first to read as it contains detailed steps to run this program. Due to the length of this code, it is not included in this appendix. Interested readers can find it in the accompanying CD. Refer to Appendix A for more information.

ExpoExpo.cpp

```
void Problem::createStateSpace ()
{
      State* state;
      //Create two NUMCLASS elements integer array
      int numJobIndex[MAXNUMCLASS];
      int numJobsMax[MAXNUMCLASS];

      int startClass = MAXNUMCLASS - _numClasses;//Calculate the starting class index

      for(int i = 0; i<startClass; i++)
            numJobsMax[i] = 0; // Set the initial several class jobsMax to be zero

    //Copy the actual sizeClass to the array numJobsMax
      for(int i = startClass; i < MAXNUMCLASS; i++)
            numJobsMax[i] = _sizeClasses[i - startClass];

      //Create the state space for each problem
      for (numJobIndex[0] = 0; numJobIndex[0] <= numJobsMax[0]; numJobIndex[0] ++)
      for (numJobIndex[1] = 0; numJobIndex[1] <= numJobsMax[1]; numJobIndex[1] ++)
      for (numJobIndex[2] = 0; numJobIndex[2] <= numJobsMax[2]; numJobIndex[2] ++)
      for (numJobIndex[3] = 0; numJobIndex[3] <= numJobsMax[3]; numJobIndex[3] ++)
      for (numJobIndex[4] = 0; numJobIndex[4] <= numJobsMax[4]; numJobIndex[4] ++)
      for (numJobIndex[5] = 0; numJobIndex[5] <= numJobsMax[5]; numJobIndex[5] ++)
      for (numJobIndex[6] = 0; numJobIndex[6] <= numJobsMax[6]; numJobIndex[6] ++)
      for (numJobIndex[7] = 0; numJobIndex[7] <= numJobsMax[7]; numJobIndex[7] ++)
      for (numJobIndex[8] = 0; numJobIndex[8] <= numJobsMax[8]; numJobIndex[8] ++)
      for (numJobIndex[9] = 0; numJobIndex[9] <= numJobsMax[9]; numJobIndex[9] ++)
      {
            VecInt numJobs;
            for (int i = startClass; i < MAXNUMCLASS;i++)
                  numJobs.push_back (numJobIndex[i]);
            state = new State(numJobs);
            _stateSpace.push_back (state);
      }
};

void Problem::probCalculation ()
{
      // Loop over each state
      for (int id = 0; id < _stateSpace.size(); id ++)
      {
            State* state = _stateSpace[id]; // Get the current state
```

ExpoExpo.cpp

```
VecInt numJobs = state->getNumJobs (); // Get the current number of jobs vector

// Loop over each class
for(int classIndex = 0; classIndex < _numClasses; classIndex++)
{
      // For debugging
      double totalProb = 0.0;

      // If there is 1+ jobs left, calculate the probb under this action
      if(numJobs[classIndex] >= 1)
      {
            //Create two NUMCLASS elements integer array
            int numJobIndex[MAXNUMCLASS];
            int numJobsMax[MAXNUMCLASS];

            int startClass = MAXNUMCLASS - _numClasses;//Calculate the starting class index

            for(int i = 0; i<startClass; i++)
                  numJobsMax[i] = 0; // Set the initial several class jobsMax to be zero

            //Copy the actual sizeClass to the array numJobsMax
            for(int i = startClass; i < MAXNUMCLASS; i++)
                  numJobsMax[i] = numJobs[i - startClass];

            // Minus 1 to this class
            numJobsMax[startClass + classIndex] = numJobsMax[startClass + classIndex]-1;

            //Loop over all states
            for (numJobIndex[0] = 0; numJobIndex[0] <= numJobsMax[0]; numJobIndex[0] ++)
            for (numJobIndex[1] = 0; numJobIndex[1] <= numJobsMax[1]; numJobIndex[1] ++)
            for (numJobIndex[2] = 0; numJobIndex[2] <= numJobsMax[2]; numJobIndex[2] ++)
            for (numJobIndex[3] = 0; numJobIndex[3] <= numJobsMax[3]; numJobIndex[3] ++)
            for (numJobIndex[4] = 0; numJobIndex[4] <= numJobsMax[4]; numJobIndex[4] ++)
            for (numJobIndex[5] = 0; numJobIndex[5] <= numJobsMax[5]; numJobIndex[5] ++)
            for (numJobIndex[6] = 0; numJobIndex[6] <= numJobsMax[6]; numJobIndex[6] ++)
            for (numJobIndex[7] = 0; numJobIndex[7] <= numJobsMax[7]; numJobIndex[7] ++)
            for (numJobIndex[8] = 0; numJobIndex[8] <= numJobsMax[8]; numJobIndex[8] ++)
            for (numJobIndex[9] = 0; numJobIndex[9] <= numJobsMax[9]; numJobIndex[9] ++)
            {
                  VecInt numRemJobs;
                  for (int i = startClass; i < MAXNUMCLASS;i++)
                        numRemJobs.push_back (numJobIndex[i]);
```

```
                                    // Call Matlab function. Note the the action is from one, so it equals to (classIndex
+ 1)
                                    double prob =  calcPrbb(_serRate, _lossRate, numJobs, numRemJobs, _numClasses,
classIndex + 1);
                                    Probb *probability = new Probb(numRemJobs, classIndex, prob);
                                    state->addTransProbs (probability);
                                    totalProb += prob;
                              };
                              if (totalProb - 1> TOTALPROBTOL || 1 - totalProb > TOTALPROBTOL )
                              {
                                    cout << "state (" ;
                                    for(int jjj = 0; jjj < _numClasses;jjj++)
                                          cout << numJobs[jjj] <<", ";
                                    cout <<") - Done "      << "action"<< classIndex+1 <<", " << totalProb <<endl;
                              };
                        };//end of if
                  } // end of loop over classes
            } // End of loops over states
            //debug_file.close ();
};

void Problem::altFluidApprxForThetaMu ()
{
      //Since the service times will be used repeatedly, we calculate them and store in a vector
      double *serTime;
      serTime= new double[_numClasses];
      for (int j = 0; j <_numClasses; j++)
            serTime[j] = 1.0 / _serRate[j];

      // Loop over each state
      for (int id = 0; id < _stateSpace.size(); id ++)
      {
            State* state = _stateSpace[id]; // Get the current state
            VecInt numJobs = state->getNumJobs ();

            int totalJobs = 0;
            for(int j = 0; j < _numClasses; j++)
                  totalJobs += numJobs[j];

            // If no job left in the system, skip to next state.
            if(totalJobs == 0)
```

ExpoExpo.cpp

```
      continue; // it has been handled in the state constructor
if(totalJobs == 1)
{
      state->setRewFluidApprForThetaMu (1.0);
      continue;
}

// Define three arraies to store the results of every class
double *tStart, *expRew, *levelFluid;
tStart = new double[_numClasses];
expRew = new double[_numClasses];
levelFluid = new double[_numClasses];
for (int wt = 0; wt <_numClasses; wt++)
      expRew[wt] = 0.0;

double expRewTotal = 0.0;
double serStartTime = 0.0; // set the initial starting time as curTim
double serEndTime;

for(int j = 0; j < _numClasses; j++) // loop over each class
{
      if (numJobs[j] == 0)// if there is no job in this class, jump to the next one
            continue;

      int thetaMuAct = j;

      // Set the starting parameters for this class
      tStart[thetaMuAct] = serStartTime; // the service starting time
      levelFluid[thetaMuAct] = numJobs[thetaMuAct] * // the remaining fluid at the starting time
                        exp( - _lossRate[thetaMuAct] * tStart[thetaMuAct]);

      // Wherever more than one job left, goto the calculation below
      while (levelFluid[thetaMuAct] >= 1)
      {
            expRew[thetaMuAct]++; // add one to the reward
            serEndTime = serStartTime + serTime[thetaMuAct];

            levelFluid[thetaMuAct] = (levelFluid[thetaMuAct]-1)*
                        exp( - _lossRate[thetaMuAct] * serTime[thetaMuAct]);
            serStartTime = serEndTime;
      }
```

ExpoExpo.cpp

```
                // If only a fractional job left, goto the calculation below
                expRew[thetaMuAct] = expRew[thetaMuAct] + levelFluid[thetaMuAct];
                serEndTime = serStartTime + levelFluid[thetaMuAct] * serTime[thetaMuAct];
                serStartTime = serEndTime; // update the service start time for next class

                // increase the total return
                expRewTotal += expRew[thetaMuAct];
          };

          //Set the total rewards
          state->setRewFluidApprForThetaMu (expRewTotal);

          //Free up the heap memories
          delete []tStart;
          delete []expRew;
          delete []levelFluid;
          tStart = NULL;
          expRew = NULL;
          levelFluid = NULL;
     }; // end of loop over states
};

void Problem::policyImprovementFluid ()
{
     // Loop over the fluid algorithms
     for (int i = 0; i < FLUIDCONTROLMATRIX.size (); i++)
     {
          string fluidAlgorithm = FLUIDCONTROLMATRIX[i]->getAlgorithm ();
          int fluidFlag = FLUIDCONTROLMATRIX[i]->getFlag ();
          if (fluidFlag != 1) // if the flag is off, skip to next
               continue;
          else
          {
               // Loop over each state
               for (int id = 0; id < _stateSpace.size(); id ++)
               {
                    State* state = _stateSpace[id]; // Get the current state
                    int action = state->getCURuleAction (); // Get the class served under thetamu rule
                    VecInt numJobs = state->getNumJobs (); // Get the number of jobs in each class
                    double expRewFluThetaMu = state->getRewFluidApprForThetaMu ();// Get the fluid exp reward
under thetamu rule
```

ExpoExpo.cpp

actions

maxExpRewNewAction

return

```
double maxExpRewNewAction = 0.0; // Define a variable to store the maximum return under new

int maxNewAction = -1; // Define a variable to store the new action correspondin to

double delta = 0.0; // Delta between the above two rewards

// Loop over classes.
int startClass = action + 1;
if (EXTRACALCIND == 1)
      startClass = 0;
for (int classIndex = startClass; classIndex < _numClasses; classIndex++)
{
      if (numJobs[classIndex]== 0 ) // If the number of jobs is zero, go to next class
            continue;
      else // Otherwise set the action to the current class and calculate the expected

      {
            int newAction = classIndex;
            double expRewNewAction = 0.0;

            //Create two NUMCLASS elements integer array
            int numJobIndex[MAXNUMCLASS];
            int numJobsMax[MAXNUMCLASS];

            int startClass = MAXNUMCLASS - _numClasses;//Calculate the starting class index

            for(int i = 0; i<startClass; i++)
                  numJobsMax[i] = 0; // Set the initial several class jobsMax to be zero

            //Copy the actual sizeClass to the array numJobsMax
            for(int i = startClass; i < MAXNUMCLASS; i++)
                  numJobsMax[i] = numJobs[i - startClass];

            // Minus 1 to this class
            numJobsMax[startClass + newAction] = numJobsMax[startClass + newAction]-1;

            //Loop over all states
            for (numJobIndex[0] = 0; numJobIndex[0] <= numJobsMax[0]; numJobIndex[0] ++)
            for (numJobIndex[1] = 0; numJobIndex[1] <= numJobsMax[1]; numJobIndex[1] ++)
            for (numJobIndex[2] = 0; numJobIndex[2] <= numJobsMax[2]; numJobIndex[2] ++)
            for (numJobIndex[3] = 0; numJobIndex[3] <= numJobsMax[3]; numJobIndex[3] ++)
            for (numJobIndex[4] = 0; numJobIndex[4] <= numJobsMax[4]; numJobIndex[4] ++)
```

ExpoExpo.cpp

```
                                    for (numJobIndex[5] = 0; numJobIndex[5] <= numJobsMax[5]; numJobIndex[5] ++)
                                    for (numJobIndex[6] = 0; numJobIndex[6] <= numJobsMax[6]; numJobIndex[6] ++)
                                    for (numJobIndex[7] = 0; numJobIndex[7] <= numJobsMax[7]; numJobIndex[7] ++)
                                    for (numJobIndex[8] = 0; numJobIndex[8] <= numJobsMax[8]; numJobIndex[8] ++)
                                    for (numJobIndex[9] = 0; numJobIndex[9] <= numJobsMax[9]; numJobIndex[9] ++)
                                    {
                                          VecInt numRemJobs;
                                          for (int i = startClass; i < MAXNUMCLASS;i++)
                                                numRemJobs.push_back (numJobIndex[i]);

                                          // Calculate the probability to next state
                                          double prob = state->getTransProbs (numRemJobs, newAction);

                                          // Get the next state with numRemjobs left
                                          State* nextState = getState(numRemJobs);

                                          // Get the total weighted reward in the route to the state above
                                          expRewNewAction += prob * nextState->getRewFluidApprForThetaMu ();
                                    }; // End of loop over all next states

                                    // After looping over all possible next state, add in the immediate reward "1"
                                    expRewNewAction += 1;

                                    if (expRewNewAction > maxExpRewNewAction)
                                    {
                                          maxExpRewNewAction = expRewNewAction;
                                          maxNewAction = newAction;
                                    }
                              }// end of if -else
                        } // End of loop over classes

                        // Calculate the delta of the thetamu rule return v.s. the max of new action returns
                        delta = expRewFluThetaMu - maxExpRewNewAction;

                        // Set the Improved action after checking the delta
                        if ( delta < 0 || EXTRACALCIND == 1)
                              state->setImprovedActionFluid (maxNewAction);
                  } //end of loop over states
            }; // end of if-else on the flags on/off
      };// end loop over algorithms
};
```

ExpoExpo.cpp

```
void Problem::geneOptActions ()
{
      for (int id = 0; id < _stateSpace.size(); id ++)
      {
            State* state = _stateSpace[id]; // Get the current state
            VecInt numJobs = state->getNumJobs (); // Get the number of jobs in each class
            double maxExpRewNewAction = 0.0; // Define a variable to store the maximum return under new actions
            int maxNewAction = -1; // Define a variable to store the new action corresponding to maxExpRewNewAction

            // Loop over the classes.
            for (int classIndex = 0; classIndex < _numClasses; classIndex++)
            {
                  if (numJobs[classIndex] == 0 ) // If the number of jobs is zero, go to next class
                        continue;
                  else // Otherwise set the action to the current class and calculate the expected return
                  {
                        int newAction = classIndex;
                        double expRewNewAction = 0.0;

                        //Create two NUMCLASS elements integer array
                        int numJobIndex[MAXNUMCLASS];
                        int numJobsMax[MAXNUMCLASS];

                        int startClass = MAXNUMCLASS - _numClasses;//Calculate the starting class index

                        for(int i = 0; i<startClass; i++)
                              numJobsMax[i] = 0; // Set the initial several class jobsMax to be zero

                        //Copy the actual sizeClass to the array numJobsMax
                        for(int i = startClass; i < MAXNUMCLASS; i++)
                              numJobsMax[i] = numJobs[i - startClass];

                        // Minus 1 to served class
                        numJobsMax[startClass + newAction] = numJobsMax[startClass + newAction]-1;

                        //Loop over all states
                        for (numJobIndex[0] = 0; numJobIndex[0] <= numJobsMax[0]; numJobIndex[0] ++)
                        for (numJobIndex[1] = 0; numJobIndex[1] <= numJobsMax[1]; numJobIndex[1] ++)
                        for (numJobIndex[2] = 0; numJobIndex[2] <= numJobsMax[2]; numJobIndex[2] ++)
                        for (numJobIndex[3] = 0; numJobIndex[3] <= numJobsMax[3]; numJobIndex[3] ++)
                        for (numJobIndex[4] = 0; numJobIndex[4] <= numJobsMax[4]; numJobIndex[4] ++)
                        for (numJobIndex[5] = 0; numJobIndex[5] <= numJobsMax[5]; numJobIndex[5] ++)
```

ExpoExpo.cpp

```
                    for (numJobIndex[6] = 0; numJobIndex[6] <= numJobsMax[6]; numJobIndex[6] ++)
                    for (numJobIndex[7] = 0; numJobIndex[7] <= numJobsMax[7]; numJobIndex[7] ++)
                    for (numJobIndex[8] = 0; numJobIndex[8] <= numJobsMax[8]; numJobIndex[8] ++)
                    for (numJobIndex[9] = 0; numJobIndex[9] <= numJobsMax[9]; numJobIndex[9] ++)
                    {
                          // Create a vector for the remained jobs
                          VecInt numRemJobs;
                          for (int i = startClass; i < MAXNUMCLASS;i++)
                                numRemJobs.push_back (numJobIndex[i]);

                          // Calculate the probability to next state
                          double prob = state->getTransProbs (numRemJobs, newAction);

                          // Get the next state with numRemjobs left
                          State* nextState = getState(numRemJobs);

                          // Get the total weighted reward in the route to the state above
                          expRewNewAction += prob * nextState->getExpRewForOpt ();
                    }; // End of loop over all next states

                    // After looping over all possible next state, add in the immediate reward "1"
                    expRewNewAction += 1;

                    if (expRewNewAction > maxExpRewNewAction)
                    {
                          maxExpRewNewAction = expRewNewAction;
                          maxNewAction = newAction;
                    }
              }// end of if -else
        } // End of loop over classes
        state->setOptAction (maxNewAction);
        state->setExpRewForOpt (maxExpRewNewAction);
    } //end of loop over state
};
```

WeibDetermnc.cpp

```
void WProblem::geneDeciEpochs ()
{
      list<double> epochs;

      // Calculate the maximum time length
      double maxTime = scalarProduct(_sizeClasses, _serRate, _numClasses);

      //Create two NUMCLASS elements integer array
      int numJobIndex[MAXNUMCLASS];
      int numJobsMax[MAXNUMCLASS];

      int startClass = MAXNUMCLASS - _numClasses;//Calculate the starting class index

      for(int i = 0; i<startClass; i++)
            numJobsMax[i] = 0; // Set the initial several class jobsMax to be zero

      //Based on the degree of results details, choose how many epochs to generate
      if(WeibDeterAct)
      {
            // If the detailed actions are needed, generate all possible epochs
            // Set the number of jobs in each class to be a very big number so that all the possible
            // decision epochs can be generated
            for(int i = startClass; i < MAXNUMCLASS; i++)
                  numJobsMax[i] = MAXNUMJOBS;
      }
      else
      {
            // If simply the total rewards from the initial state are needed, generate
            // only reachable states from beginning
            for(int i = startClass; i < MAXNUMCLASS; i++)
                  numJobsMax[i] = _sizeClasses[i - startClass];
      }

      //Create the state space for each problem
      for (numJobIndex[0] = 0; numJobIndex[0] <= numJobsMax[0]; numJobIndex[0] ++)
      for (numJobIndex[1] = 0; numJobIndex[1] <= numJobsMax[1]; numJobIndex[1] ++)
      for (numJobIndex[2] = 0; numJobIndex[2] <= numJobsMax[2]; numJobIndex[2] ++)
      for (numJobIndex[3] = 0; numJobIndex[3] <= numJobsMax[3]; numJobIndex[3] ++)
      for (numJobIndex[4] = 0; numJobIndex[4] <= numJobsMax[4]; numJobIndex[4] ++)
      for (numJobIndex[5] = 0; numJobIndex[5] <= numJobsMax[5]; numJobIndex[5] ++)
      for (numJobIndex[6] = 0; numJobIndex[6] <= numJobsMax[6]; numJobIndex[6] ++)
      for (numJobIndex[7] = 0; numJobIndex[7] <= numJobsMax[7]; numJobIndex[7] ++)
```

WeibDetermnc.cpp

```
for (numJobIndex[8] = 0; numJobIndex[8] <= numJobsMax[8]; numJobIndex[8] ++)
for (numJobIndex[9] = 0; numJobIndex[9] <= numJobsMax[9]; numJobIndex[9] ++)
{
      VecInt currentNumJobs;
      for (int i = startClass; i < MAXNUMCLASS;i++)
            currentNumJobs.push_back (numJobIndex[i]);

      double decisionEpoch = scalarProduct(currentNumJobs,_serRate, _numClasses);
      if (decisionEpoch <= maxTime)
      {
            epochs.push_back (decisionEpoch);
            //cout << decisionEpoch<<endl;
      }
      else
            break;
}

//Sort the list in ascending order and then remove duplicated values
epochs.sort ();
epochs.unique (); // this method cannot remove all the duplicated values

// Copy the epochs to a vector and delete remained duplicat time points
list <double>::iterator c1_Iter, c2_Iter;

c1_Iter = epochs.begin (); // The left hand side of the pairwise comparison
c2_Iter = epochs.begin (); // The right hand side of the pairwise comparison
c2_Iter ++; // Move one position from the beginning
do
{
      if(*c2_Iter - *c1_Iter > VALUE_ZERO) // If they are not equal, push LHS into the vector
      {
            _decisionEpochs.push_back (*c1_Iter);
            c1_Iter = c2_Iter; // Move LHS to the current position of RHS
            c2_Iter ++; // Move RHS one position
      }
      else // If they are equal, just move one position for RHS
            c2_Iter ++;

}while ( c2_Iter != epochs.end()); // When c2_Iter is the end, stop the loop

// Push back the last element into the vector
_decisionEpochs.push_back (*(--c2_Iter));
```

WeibDetermnc.cpp

```
      _numDecisionEpochs = _decisionEpochs.size ();
      _maxTime = _decisionEpochs[_numDecisionEpochs - 1];
};

void WProblem::calcAllMRLTs ()
{
      for(int classIndex = 0; classIndex < _numClasses; classIndex++)
      {
            VecDouble MRLTCls;
            double alpha = _alpha[classIndex];
            double beta =  _beta[classIndex];

            for(int timeIndex = 0; timeIndex < _numDecisionEpochs; timeIndex++)
            {
                  double curTime = getEpochTimePoint(timeIndex);
                  double MRLT = calcMRLT_matlab(alpha, beta, curTime);
                  /*************************************************************/
                  /* Updated 15/01/09                                          */
                  /* The formula below is not numerically stable, so discarded. */
                  /*************************************************************/
                  ////Calculate the updated mean remaing life time, by Argon's formula (Argon2007)
                  //double temp = pow(curTime/beta, alpha);
                  //double MRLT = beta * boost::math::tgamma (oneOverAlpha, temp) * exp(temp) / alpha ;

                  MRLTCls.push_back(MRLT);
            }
            _MRLTs.push_back (MRLTCls);
      }
};
```

WeibExpo.cpp

```
void WESProblem::calcBySimulation ()
{
      //Open a file for simulation output
      char buffer[50];
      sprintf(buffer,"wesimulation%d.txt",_problemIndex);

      ofstream out_file(buffer);
      if(out_file.fail())
      {
            cerr<<"error opening output file wesimulation.txt \n";
      }

      // Get the mean sertime to save computation when generating random service times
      VecDouble meanSerTime(_numClasses);
      for(int j = 0; j< _numClasses; j++)
            meanSerTime[j] = 1 / (_serRate[j] * _stepSize);

      // Choose seed for the random number generators
    int32 seed = (int32)time(0);        // random seed
      seed += 3600 * getProbIndex (); // perturbed by the number of hours of the problem index
      out_file<<seed<<endl;

    // choose one of the random number generators:
      CRandomMersenne rgLifeTimes(seed);          // Random number generator for life times
      CRandomMersenne rgSerTimes(seed+100);         // Random number generator for service times
      CRandomMersenne rgJobsPriority(seed+200);   // Random number generator for the jobs' priorities in each class

      // three two dimention vectors to store the random numbers in even number runs, for the purpose of antithetic
variables
      VecDubTwoDimens arrayRandLifeTimes;
      VecDubTwoDimens arrayRandSerTimes;
      VecDubTwoDimens arrayRandPriorities;
      for(int j = 0; j<_numClasses; j++)
      {
            VecDouble temp (_sizeClasses[j]);
            arrayRandLifeTimes.push_back (temp);
            arrayRandSerTimes.push_back (temp);
            arrayRandPriorities.push_back (temp);
      };

      int simRuns; // index of simulation runs. To get the number of runs, add one to it (simRuns + 1)
      // or:
```

WeibExpo.cpp

```
// CRandomMother rg(seed);          // make instance of random number generator

// Defined statistics to be used
double thetaMu_fluid_mean_old = 0.0;
double thetaMu_fluid_mean_new = 0.0;
double myopic_fluid_mean_old = 0.0;
double myopic_fluid_mean_new = 0.0;
double fluid_mean_old = 0.0;
double fluid_mean_new = 0.0;
double thetaMu_fluid_var = 0.0;
double myopic_fluid_var = 0.0;

// Two flags to control simulation
bool thetaMuDone = false;
bool myopicDone = false;

// The number of job served under each policy
double fluidJobServed, thetaMuJobServed, myopicJobServed;

// Find the first fluid PI action for the problem. It is used repeatedly so by doing this the time is saved.
WState* initState = getState(_sizeClasses);
int firstFluidPIAction = singleStateFluidPI(initState, 0.0);

for(simRuns = 0; simRuns < MAXNUMSIMURUNS; simRuns++)
{
      // Set up the simulation environment
      vector<ListDouble> arrayOfLifeTimes; //The life times of jobs in each class
      vector<ListDouble> arrayOfServiceTimes; // The service times of jobs in each class
      vector<ListDouble> arrayOfPriorities; // The jobs' prioritie in each class

      ListDouble::iterator iterOfLifeTime, iterOfServiceTime, iterOfPriority; //The list iterator access
randomly the elements
      ListDouble::iterator servedJob;
      ListDouble::difference_type gap;

      // Define the state and the action at this state
      WState* curState;
      int action;

      if(simRuns % 2 == 0)
      {
            // Generate new random numbers for even indexed runs
```

WeibExpo.cpp

```
                for(int j = 0; j<_numClasses; j++)
                {
                      for(int numJobs = 0; numJobs < _sizeClasses[j]; numJobs++)
                      {
                            arrayRandLifeTimes[j][numJobs]= rgLifeTimes.Random (); // generate random numbers for
lifetimes
                            arrayRandSerTimes[j][numJobs] = rgSerTimes.Random (); // generate random numbers for
service times
                            arrayRandPriorities[j][numJobs]= rgJobsPriority.Random (); // generate the random
numbers for priorities
                      }
                }
          }
          else
          {
                // Use 1 - U for odd indexed runs
                for(int j = 0; j<_numClasses; j++)
                {
                      for(int numJobs = 0; numJobs < _sizeClasses[j]; numJobs++)
                      {
                            arrayRandLifeTimes[j][numJobs] = 1 - arrayRandLifeTimes[j][numJobs]; // use 1 - U_old
                            arrayRandSerTimes[j][numJobs] = 1 - arrayRandSerTimes[j][numJobs]; // use 1 - U_old
                            arrayRandPriorities[j][numJobs] = 1 - arrayRandPriorities[j][numJobs]; // use 1 -
U_old
                      }
                }
          };

          // Generate the life times, service times and priorities for every job and then sorted in ascending
order according to life times
          for(int j = 0; j<_numClasses; j++)
          {
                ListDouble jobsLifeTimes, jobsServiceTimes, jobsPriorities;
                if (_sizeClasses[j] > 0) // If the number of jobs is not zero
                {
                      for(int numJobs = 0; numJobs < _sizeClasses[j]; numJobs++)
                      {
                            jobsLifeTimes.push_back ( _beta[j]* pow(- log(arrayRandLifeTimes[j][numJobs]),
1/_alpha[j])); // generate random lifetimes
                            jobsServiceTimes.push_back (ceil(- meanSerTime[j]*log(arrayRandSerTimes[j][numJobs]))
* _stepSize); // generate random service times
```

WeibExpo.cpp

```
                        jobsPriorities.push_back (arrayRandPriorities[j][numJobs]); // generate the priorities
in (0,1)
                    };
                    jobsLifeTimes.sort (); // sort the life times
                };
                arrayOfLifeTimes.push_back (jobsLifeTimes); // add into the array
                arrayOfServiceTimes.push_back (jobsServiceTimes); // add into the array
                arrayOfPriorities.push_back (jobsPriorities);
        };

        /*****************************************************************************/
        /***************************** Fluid PI Simulation ***************************/
        /*****************************************************************************/
        if(printSimuSteps)
        {
                // setup the headers of the output file
                out_file<<"systemTime "<<"Event ";
                for(int j=0; j<_numClasses;j++)
                        out_file<<"jobRemOfClass"<<j+1<<"    ";
                out_file<<"ServerStatus"<<endl;
        }

        // Initialize
        VecInt jobRem = _sizeClasses; // set the number of jobs remained as the initial numbers
        int serverStatus = -1; //set the server status to idle
        double sysTime = 0.0; // set the system time as zero
        double nextSerEndTime = VALUE_INFINITY; // Define the event time of service completion
        double nextAbandonTime = VALUE_INFINITY; // Define the event time of customer abandment
        int nextAbandonClass; // Define which class customer is lost next
        int totalJobs = 0; // Count the total number of jobs at epochs
        int jobServed = 0; // Count the number of customers served
        VecInt jobLost(_numClasses,0); // Count the number of customers lost by class

        vector<ListDouble> FluidArrayOfLifeTimes(arrayOfLifeTimes);
        vector<ListDouble> FluidArrayOfSerTimes (arrayOfServiceTimes);
        vector<ListDouble> FluidArrayOfPriorities (arrayOfPriorities);

        // Find the first fluid PI action
        // Note again the action in the c++ codes is zero based.
        action = firstFluidPIAction;

        // Select which job is sent for service this time, and remove it from the life times array
```

WeibExpo.cpp

```
// Option 2, select the one for service based on their random priorities
if(FluidArrayOfLifeTimes[action].size () == 1)
{
      // if only one job left, just select it
      iterOfServiceTime = FluidArrayOfSerTimes[action].begin ();
      // Get the service time of it, and get the next service completion time
      double serTime = *iterOfServiceTime;
      nextSerEndTime = serTime;

      // Then remove the elements from all lists
      FluidArrayOfLifeTimes[action].pop_front ();
      FluidArrayOfSerTimes[action].pop_front ();
      FluidArrayOfPriorities[action].pop_front ();
}
else
{
      iterOfLifeTime = FluidArrayOfLifeTimes[action].begin ();
      iterOfServiceTime = FluidArrayOfSerTimes[action].begin ();

      // choose the highest priority job available for service
      servedJob = max_element(FluidArrayOfPriorities[action].begin (),
            FluidArrayOfPriorities[action].end());

      // Move the iterators to the right location fo find the service times and life times
      gap = distance(FluidArrayOfPriorities[action].begin (), servedJob);
      advance(iterOfLifeTime, gap);
      advance(iterOfServiceTime, gap);

      // Get the service time of it and the next service completion time
      double serTime = *iterOfServiceTime;
      nextSerEndTime = serTime;

      // Then remove the elements from all three lists
      FluidArrayOfLifeTimes[action].erase(iterOfLifeTime);
      FluidArrayOfSerTimes[action].erase(iterOfServiceTime);
      FluidArrayOfPriorities[action].erase (servedJob);
}

// Decrease the number of remaining jobs from the action class and update the counter variable
jobRem[action]--;
serverStatus = action;
jobServed++;
```

WeibExpo.cpp

```
// If the next service ends after _maxTime, stop the simulation and do not count the reward onwards
if(nextSerEndTime >= _maxTime)
{
      if(printSimuSteps)
      {
            out_file<<sysTime<<"    "<<"SimuEnds, _maxTimeReached"<<"    ";
            for(int j=0; j<_numClasses;j++)
                  out_file<<jobRem[j]<<"  ";
            out_file<<0<<endl;
      }
      goto FLUIDPI_STOP;
};

// Schedule the next abandonment after making the selection.
for(int j = 0; j<_numClasses; j++)
{
      if (FluidArrayOfLifeTimes[j].empty ())
            continue; // If there is no job left, just go to the next class

      // The first element is always the one to be lost next in each class
      iterOfLifeTime = FluidArrayOfLifeTimes[j].begin ();
      if(*iterOfLifeTime < nextAbandonTime)
      {
            nextAbandonTime = *iterOfLifeTime;
            nextAbandonClass = j;
      };
};

// Output this first event
if(printSimuSteps)
{
      out_file<<sysTime<<"    "<<"SendTheFirstJobForService"<<"    ";
      for(int j=0; j<_numClasses;j++)
            out_file<<jobRem[j]<<"  ";
      out_file<<serverStatus+1<<endl;
}

for(int j = 0; j<_numClasses; j++)
      totalJobs += jobRem[j];

while(totalJobs > 0)
```

WeibExpo.cpp

```
{
      if(nextSerEndTime <= nextAbandonTime) // if next event is service completion
      {
            sysTime = nextSerEndTime;
            curState = getState(jobRem);

            // Get the action of this state
            int singleClsFlag = curState->getSingleClsFlag ();
            if (singleClsFlag >= 0)
                  action = singleClsFlag;
            else
                  action = singleStateFluidPI(curState, sysTime);

            // Select which job is sent for service from the action class,
            //// Option 1, Send the one to be lost next for service
            //FluidArrayOfLifeTimes[action].pop_front ();

            // Option 2, select the one for service based on their random priorities
            if(FluidArrayOfLifeTimes[action].size () == 1)
            {
                  // if only one job left, just select it
                  iterOfServiceTime = FluidArrayOfSerTimes[action].begin ();
                  // Get the service time of it, and get the next service completion time
                  double serTime = *iterOfServiceTime;
                  nextSerEndTime = serTime + sysTime;

                  // Then remove the elements from all lists
                  FluidArrayOfLifeTimes[action].pop_front ();
                  FluidArrayOfSerTimes[action].pop_front ();
                  FluidArrayOfPriorities[action].pop_front ();
            }
            else
            {
                  iterOfLifeTime = FluidArrayOfLifeTimes[action].begin ();
                  iterOfServiceTime = FluidArrayOfSerTimes[action].begin ();

                  // choose the highest priority job available for service
                  servedJob = max_element(FluidArrayOfPriorities[action].begin (),
                        FluidArrayOfPriorities[action].end());

                  //cout<<*servedJob<<endl;
```

WeibExpo.cpp

```
        // Move the iterators to the right location fo find the service times and life times
        gap = distance(FluidArrayOfPriorities[action].begin (), servedJob);
        advance(iterOfLifeTime, gap);
        advance(iterOfServiceTime, gap);

        ////for debug
        //cout<<"The one sent for service is No"<<theOneSentService+1<<endl;

        // Get the service time of it and the next service completion time
        double serTime = *iterOfServiceTime;
        nextSerEndTime = serTime + sysTime;

        // Then remove the elements from all three lists
        FluidArrayOfLifeTimes[action].erase(iterOfLifeTime);
        FluidArrayOfSerTimes[action].erase(iterOfServiceTime);
        FluidArrayOfPriorities[action].erase (servedJob);
}

jobRem[action]--;
serverStatus = action; // update the server status
jobServed++;

// If the service ends after _maxTime, stop the simulation and do not count the reward
onwards

if(nextSerEndTime >= _maxTime)
{
      if(printSimuSteps)
      {
            out_file<<sysTime<<"    "<<"SimuEnds, _maxTimeReached"<<"    ";
            for(int j=0; j<_numClasses;j++)
                  out_file<<jobRem[j]<<"  ";
            out_file<<0<<endl;
      }
      goto FLUIDPI_STOP;
};

// Reschedule the next abandonment after making the selection.
nextAbandonTime = VALUE_INFINITY;
for(int j = 0; j<_numClasses; j++)
{
      if (FluidArrayOfLifeTimes[j].empty ())
            continue;
```

WeibExpo.cpp

```
            iterOfLifeTime = FluidArrayOfLifeTimes[j].begin ();
            if(*iterOfLifeTime < nextAbandonTime)
            {
                  nextAbandonTime = *iterOfLifeTime;
                  nextAbandonClass = j;
            };
      };

      if(printSimuSteps)
      {
            out_file<<sysTime<<"    "<<"serEndAndSendNext"<<"     ";
            for(int j=0; j<_numClasses;j++)
                  out_file<<jobRem[j]<<"  ";
            out_file<<serverStatus+1<<endl;
      }
}
else //otherwise next event is customer loss
{
      sysTime = nextAbandonTime;
      jobRem[nextAbandonClass]--;
      jobLost[nextAbandonClass]++;
      nextAbandonTime = VALUE_INFINITY;

      if(printSimuSteps)
      {
            out_file<<sysTime<<"    "<<"Class"<<nextAbandonClass+1<<"Abandonment"<<"      ";
            for(int j=0; j<_numClasses;j++)
                  out_file<<jobRem[j]<<"  ";
            out_file<<serverStatus+1<<endl;
      }

      //Update the next lost job location by removing the one just lost
      FluidArrayOfLifeTimes[nextAbandonClass].pop_front ();
      FluidArrayOfSerTimes[nextAbandonClass].pop_front();
      FluidArrayOfPriorities[nextAbandonClass].pop_front ();

      //Find the time of next abandonment and which class
      for(int j = 0; j<_numClasses; j++)
      {
            if (FluidArrayOfLifeTimes[j].empty ())
                  continue;
            iterOfLifeTime = FluidArrayOfLifeTimes[j].begin ();
```

WeibExpo.cpp

```
                if(*iterOfLifeTime < nextAbandonTime)
                {
                      nextAbandonTime = *iterOfLifeTime;
                      nextAbandonClass = j;
                };
          };
      }
      // calcualte the total remained jobs
      totalJobs = 0;
      for(int j = 0; j<_numClasses; j++)
            totalJobs += jobRem[j];
}; // end of while loop

sysTime = nextSerEndTime;
serverStatus = -1;
if(printSimuSteps)
{
      out_file<<sysTime<<"    "<<"LastJobEndService"<<"     ";
      for(int j=0; j<_numClasses;j++)
                  out_file<<jobRem[j]<<"  ";
            out_file<<serverStatus+1<<endl;
}

FLUIDPI_STOP: //the label for the stopage of fluid PI policy
fluidJobServed = jobServed;

// update real time the sample mean of fluid PI;
fluid_mean_new = fluid_mean_old + (fluidJobServed - fluid_mean_old) / (simRuns + 1);
fluid_mean_old = fluid_mean_new;

/*****************************************************************************/
/***************************** ThetaMu Simulation ****************************/
/*****************************************************************************/
if(!thetaMuDone)
{
      if(printSimuSteps)
      {
            // setup the headers of the output file
            out_file<<"systemTime "<<"Event ";
            for(int j=0; j<_numClasses;j++)
                  out_file<<"jobRemOfClass"<<j+1<<"   ";
            out_file<<"ServerStatus"<<endl;
```

WeibExpo.cpp

```
}

// Re-initialize
jobRem.assign(_sizeClasses.begin(), _sizeClasses.end()); // set the number of jobs remained as the
initial numbers
serverStatus = -1; //set the server status to idle
sysTime = 0.0; // set the system time as zero
nextSerEndTime = VALUE_INFINITY; // Define the event time of service completion
nextAbandonTime = VALUE_INFINITY; // Define the event time of customer abandment
nextAbandonClass; // Define which class customer is lost next
totalJobs = 0; // Count the total number of jobs at epochs
jobServed = 0; // Count the number of customers served
jobLost.assign(_numClasses,0); // Count the number of customers lost by class

vector<ListDouble> ThetaMuArrayOfLifeTimes(arrayOfLifeTimes);
vector<ListDouble> ThetaMuArrayOfSerTimes(arrayOfServiceTimes);
vector<ListDouble> ThetaMuArrayOfPriorities (arrayOfPriorities);

// Get the starting state
curState = getState(jobRem);

// Find the first ThetaMu action
// Note again the action in the c++ codes is zero based.
action = curState->getThetaMuAction (0);


// Option 2, select the one for service based on their random priorities
if(ThetaMuArrayOfLifeTimes[action].size () == 1)
{
      // if only one job left, just select it
      iterOfServiceTime = ThetaMuArrayOfSerTimes[action].begin ();
      // Get the service time of it, and get the next service completion time
      double serTime = *iterOfServiceTime;
      nextSerEndTime = serTime;

      // Then remove the elements from all lists
      ThetaMuArrayOfLifeTimes[action].pop_front ();
      ThetaMuArrayOfSerTimes[action].pop_front ();
      ThetaMuArrayOfPriorities[action].pop_front ();
}
else
{
```

WeibExpo.cpp

```
        iterOfLifeTime = ThetaMuArrayOfLifeTimes[action].begin ();
        iterOfServiceTime = ThetaMuArrayOfSerTimes[action].begin ();

        // choose the highest priority job available for service
        servedJob = max_element(ThetaMuArrayOfPriorities[action].begin (),
              ThetaMuArrayOfPriorities[action].end());

        // Move the iterators to the right location fo find the service times and life times
        gap = distance(ThetaMuArrayOfPriorities[action].begin (), servedJob);
        advance(iterOfLifeTime, gap);
        advance(iterOfServiceTime, gap);

        ////for debug
        //cout<<"The one sent for service is No"<<theOneSentService+1<<endl;

        // Get the service time of it and the next service completion time
        double serTime = *iterOfServiceTime;
        nextSerEndTime = serTime;

        // Then remove the elements from all three lists
        ThetaMuArrayOfLifeTimes[action].erase(iterOfLifeTime);
        ThetaMuArrayOfSerTimes[action].erase(iterOfServiceTime);
        ThetaMuArrayOfPriorities[action].erase (servedJob);
  }

  // Decrease the number of remaining jobs from the action class and update the counter variable

  jobRem[action]--;
  serverStatus = action;
  jobServed++;

  // If the next service ends after _maxTime, stop the simulation and do not count the reward
onwards
  if(nextSerEndTime >= _maxTime)
  {
        if(printSimuSteps)
        {
              out_file<<sysTime<<"    "<<"SimuEnds, _maxTimeReached"<<"   ";
              for(int j=0; j<_numClasses;j++)
                    out_file<<jobRem[j]<<"  ";
              out_file<<0<<endl;
        }
```

WeibExpo.cpp

```
        goto THETAMU_STOP;
};

// Schedule the next abandonment after making the selection.
for(int j = 0; j<_numClasses; j++)
{
        if (ThetaMuArrayOfLifeTimes[j].empty ())
                continue; // If there is no job left, just go to the next class

        // The first element is always the one to be lost next in each class
        iterOfLifeTime = ThetaMuArrayOfLifeTimes[j].begin ();
        if(*iterOfLifeTime < nextAbandonTime)
        {
                nextAbandonTime = *iterOfLifeTime;
                nextAbandonClass = j;
        };
};

// Output this first event
if(printSimuSteps)
{
        out_file<<sysTime<<"    "<<"SendTheFirstJobForService"<<"   ";
        for(int j=0; j<_numClasses;j++)
                out_file<<jobRem[j]<<"  ";
        out_file<<serverStatus+1<<endl;
}

for(int j = 0; j<_numClasses; j++)
        totalJobs += jobRem[j];

while(totalJobs > 0)
{
        if(nextSerEndTime <= nextAbandonTime) // if next event is service completion
        {
                sysTime = nextSerEndTime;
                curState = getState(jobRem);
                int epochIndex = int(sysTime / _stepSize);
                action = curState->getThetaMuAction (epochIndex);

                // Select which job is sent for service from the action class,
                //// Option 1, Send the one to be lost next for service
                //ThetaMuArrayOfLifeTimes[action].pop_front ();
```

WeibExpo.cpp

```
// Option 2, select the one for service based on their random priorities
if(ThetaMuArrayOfLifeTimes[action].size () == 1)
{
      // if only one job left, just select it
      iterOfServiceTime = ThetaMuArrayOfSerTimes[action].begin ();
      // Get the service time of it, and get the next service completion time
      double serTime = *iterOfServiceTime;
      nextSerEndTime = serTime + sysTime;

      // Then remove the elements from all lists
      ThetaMuArrayOfLifeTimes[action].pop_front ();
      ThetaMuArrayOfSerTimes[action].pop_front ();
      ThetaMuArrayOfPriorities[action].pop_front ();
}
else
{
      iterOfLifeTime = ThetaMuArrayOfLifeTimes[action].begin ();
      iterOfServiceTime = ThetaMuArrayOfSerTimes[action].begin ();

      // choose the highest priority job available for service
      servedJob = max_element(ThetaMuArrayOfPriorities[action].begin (),
            ThetaMuArrayOfPriorities[action].end());

      // Move the iterators to the right location fo find the service times and life
times

      gap = distance(ThetaMuArrayOfPriorities[action].begin (), servedJob);
      advance(iterOfLifeTime, gap);
      advance(iterOfServiceTime, gap);

      ////for debug
      //cout<<"The one sent for service is No"<<theOneSentService+1<<endl;

      // Get the service time of it and the next service completion time
      double serTime = *iterOfServiceTime;
      nextSerEndTime = serTime + sysTime;

      // Then remove the elements from all three lists
      ThetaMuArrayOfLifeTimes[action].erase(iterOfLifeTime);
      ThetaMuArrayOfSerTimes[action].erase(iterOfServiceTime);
      ThetaMuArrayOfPriorities[action].erase (servedJob);
}
```

WeibExpo.cpp

```
                jobRem[action]--;
                serverStatus = action; // update the server status
                jobServed++;

                // If the service ends after _maxTime, stop the simulation and do not count the reward
onwards
                if(nextSerEndTime >= _maxTime)
                {
                      if(printSimuSteps)
                      {
                            out_file<<sysTime<<"    "<<"SimuEnds, _maxTimeReached"<<"   ";
                            for(int j=0; j<_numClasses;j++)
                                  out_file<<jobRem[j]<<"  ";
                            out_file<<0<<endl;
                      }
                      goto THETAMU_STOP;
                };

                // Reschedule the next abandonment after making the selection.
                nextAbandonTime = VALUE_INFINITY;
                for(int j = 0; j<_numClasses; j++)
                {
                      if (ThetaMuArrayOfLifeTimes[j].empty ())
                            continue;
                      iterOfLifeTime = ThetaMuArrayOfLifeTimes[j].begin ();
                      if(*iterOfLifeTime < nextAbandonTime)
                      {
                            nextAbandonTime = *iterOfLifeTime;
                            nextAbandonClass = j;
                      };
                };

                if(printSimuSteps)
                {
                      out_file<<sysTime<<"    "<<"serEndAndSendNext"<<"     ";
                      for(int j=0; j<_numClasses;j++)
                            out_file<<jobRem[j]<<"  ";
                      out_file<<serverStatus+1<<endl;
                }
          }
          else //otherwise next event is customer loss
```

WeibExpo.cpp

```
        {
                sysTime = nextAbandonTime;
                jobRem[nextAbandonClass]--;
                jobLost[nextAbandonClass]++;
                nextAbandonTime = VALUE_INFINITY;

                if(printSimuSteps)
                {
                        out_file<<sysTime<<"    "<<"Class"<<nextAbandonClass+1<<"Abandonment"<<"      ";
                        for(int j=0; j<_numClasses;j++)
                                out_file<<jobRem[j]<<"  ";
                        out_file<<serverStatus+1<<endl;
                }

                //Update the next lost job location by removing the one just lost
                ThetaMuArrayOfLifeTimes[nextAbandonClass].pop_front ();
                ThetaMuArrayOfSerTimes[nextAbandonClass].pop_front();
                ThetaMuArrayOfPriorities[nextAbandonClass].pop_front ();

                //Find the time of next abandonment and which class
                for(int j = 0; j<_numClasses; j++)
                {
                        if (ThetaMuArrayOfLifeTimes[j].empty ())
                                continue;
                        iterOfLifeTime = ThetaMuArrayOfLifeTimes[j].begin ();
                        if(*iterOfLifeTime < nextAbandonTime)
                        {
                                nextAbandonTime = *iterOfLifeTime;
                                nextAbandonClass = j;
                        };
                };
        }
        // calcualte the total remained jobs
        totalJobs = 0;
        for(int j = 0; j<_numClasses; j++)
                totalJobs += jobRem[j];
}; // end of while loop

sysTime = nextSerEndTime;
serverStatus = -1;
if(printSimuSteps)
{
```

WeibExpo.cpp

```
                        out_file<<sysTime<<"    "<<"LastJobEndService"<<"     ";
                        for(int j=0; j<_numClasses;j++)
                                    out_file<<jobRem[j]<<"  ";
                              out_file<<serverStatus+1<<endl;
                  }

                  THETAMU_STOP: //the label for the stopage of thetamu policy
                  thetaMuJobServed = jobServed;

                  // Update real time the sample mean and variance of the difference between fluid PI and thetaMu
                  double thetaMu_fluid_delta = thetaMuJobServed - fluidJobServed;
                  thetaMu_fluid_mean_new = thetaMu_fluid_mean_old + (thetaMu_fluid_delta - thetaMu_fluid_mean_old) /
(simRuns + 1);
                  double delta = thetaMu_fluid_mean_new - thetaMu_fluid_mean_old;
                  if(simRuns > 0)
                        thetaMu_fluid_var = (1- 1.0/double(simRuns)) * thetaMu_fluid_var
                        + (simRuns + 1) * delta * delta;
                  thetaMu_fluid_mean_old = thetaMu_fluid_mean_new;

                  // If the std dev is within the allowable limit, stop thetamu simulation after 101 runs
                  if(1.96 * (sqrt(thetaMu_fluid_var/(simRuns+1))) < ACCEPTEDRELDIFF && simRuns > 99)
                  {
                        cout<<"ThetaMu policy simulation stop criteria reached after "<< simRuns+1<<" runs"<<endl;
                        thetaMuJobServed = 0.0;
                        thetaMuDone = true;
                  }
            };

            /*****************************************************************************/
            /****************************** Myopic Simulation ****************************/
            /*****************************************************************************/
            if(!myopicDone)
            {
                  if(printSimuSteps)
                  {
                        // setup the headers of the output file
                        out_file<<"systemTime "<<"Event ";
                        for(int j=0; j<_numClasses;j++)
                              out_file<<"jobRemOfClass"<<j+1<<"   ";
                        out_file<<"ServerStatus"<<endl;
                  }
```

WeibExpo.cpp

```
                // Re-initialize
                jobRem.assign(_sizeClasses.begin(), _sizeClasses.end()); // set the number of jobs remained as the
initial numbers
                serverStatus = -1; //set the server status to idle
                sysTime = 0.0; // set the system time as zero
                nextSerEndTime = VALUE_INFINITY; // Define the event time of service completion
                nextAbandonTime = VALUE_INFINITY; // Define the event time of customer abandment
                nextAbandonClass; // Define which class customer is lost next
                totalJobs = 0; // Count the total number of jobs at epochs
                jobServed = 0; // Count the number of customers served
                jobLost.assign(_numClasses,0); // Count the number of customers lost by class

                vector<ListDouble> MyopicArrayOfLifeTimes(arrayOfLifeTimes);
                vector<ListDouble> MyopicArrayOfSerTimes(arrayOfServiceTimes);
                vector<ListDouble> MyopicArrayOfPriorities(arrayOfPriorities);

                // Get the starting state
                curState = getState(jobRem);

                // Find the first myopic action
                action = curState->getTriAction (0);

                // Option 2, select the one for service based on their random priorities
                if(MyopicArrayOfLifeTimes[action].size () == 1)
                {
                      // if only one job left, just select it
                      iterOfServiceTime = MyopicArrayOfSerTimes[action].begin ();
                      // Get the service time of it, and get the next service completion time
                      double serTime = *iterOfServiceTime;
                      nextSerEndTime = serTime;

                      // Then remove the elements from all lists
                      MyopicArrayOfLifeTimes[action].pop_front ();
                      MyopicArrayOfSerTimes[action].pop_front ();
                      MyopicArrayOfPriorities[action].pop_front ();
                }
                else
                {
                      iterOfLifeTime = MyopicArrayOfLifeTimes[action].begin ();
                      iterOfServiceTime = MyopicArrayOfSerTimes[action].begin ();

                      // choose the highest priority job available for service
```

WeibExpo.cpp

```
        servedJob = max_element(MyopicArrayOfPriorities[action].begin (),
              MyopicArrayOfPriorities[action].end());

        // Move the iterators to the right location fo find the service times and life times
        gap = distance(MyopicArrayOfPriorities[action].begin (), servedJob);
        advance(iterOfLifeTime, gap);
        advance(iterOfServiceTime, gap);

        // Get the service time of it and the next service completion time
        double serTime = *iterOfServiceTime;
        nextSerEndTime = serTime;

        // Then remove the elements from all three lists
        MyopicArrayOfLifeTimes[action].erase(iterOfLifeTime);
        MyopicArrayOfSerTimes[action].erase(iterOfServiceTime);
        MyopicArrayOfPriorities[action].erase (servedJob);
}

// Decrease the number of remaining jobs from the action class and update the counter variable

jobRem[action]--;
serverStatus = action;
jobServed++;

// If the next service ends after _maxTime, stop the simulation and do not count the reward
onwards
if(nextSerEndTime >= _maxTime)
{
      if(printSimuSteps)
      {
            out_file<<sysTime<<"    "<<"SimuEnds, _maxTimeReached"<<"    ";
            for(int j=0; j<_numClasses;j++)
                  out_file<<jobRem[j]<<"  ";
            out_file<<0<<endl;
      }
      goto MYOPIC_STOP;
};

// Schedule the next abandonment after making the selection.
for(int j = 0; j<_numClasses; j++)
{
      if (MyopicArrayOfLifeTimes[j].empty ())
```

WeibExpo.cpp

```
            continue; // If there is no job left, just go to the next class

      // The first element is always the one to be lost next in each class
      iterOfLifeTime = MyopicArrayOfLifeTimes[j].begin ();
      if(*iterOfLifeTime < nextAbandonTime)
      {
            nextAbandonTime = *iterOfLifeTime;
            nextAbandonClass = j;
      };
};

// Output this first event
if(printSimuSteps)
{
      out_file<<sysTime<<"    "<<"SendTheFirstJobForService"<<"   ";
      for(int j=0; j<_numClasses;j++)
            out_file<<jobRem[j]<<"  ";
      out_file<<serverStatus+1<<endl;
}

for(int j = 0; j<_numClasses; j++)
      totalJobs += jobRem[j];

while(totalJobs > 0)
{
      if(nextSerEndTime <= nextAbandonTime) // if next event is service completion
      {
            sysTime = nextSerEndTime;
            curState = getState(jobRem);
            int epochIndex = int(sysTime / _stepSize);
            action = curState->getTriAction (epochIndex);

            // Option 2, select the one for service based on their random priorities
            if(MyopicArrayOfLifeTimes[action].size () == 1)
            {
                  // if only one job left, just select it
                  iterOfServiceTime = MyopicArrayOfSerTimes[action].begin ();
                  // Get the service time of it, and get the next service completion time
                  double serTime = *iterOfServiceTime;
                  nextSerEndTime = serTime + sysTime;

                  // Then remove the elements from all lists
```

WeibExpo.cpp

```
                MyopicArrayOfLifeTimes[action].pop_front ();
                MyopicArrayOfSerTimes[action].pop_front ();
                MyopicArrayOfPriorities[action].pop_front ();
        }
        else
        {
                iterOfLifeTime = MyopicArrayOfLifeTimes[action].begin ();
                iterOfServiceTime = MyopicArrayOfSerTimes[action].begin ();

                // choose the highest priority job available for service
                servedJob = max_element(MyopicArrayOfPriorities[action].begin (),
                        MyopicArrayOfPriorities[action].end());

                // Move the iterators to the right location fo find the service times and life
times

                gap = distance(MyopicArrayOfPriorities[action].begin (), servedJob);
                advance(iterOfLifeTime, gap);
                advance(iterOfServiceTime, gap);

                // Get the service time of it and the next service completion time
                double serTime = *iterOfServiceTime;
                nextSerEndTime = serTime + sysTime;

                // Then remove the elements from all three lists
                MyopicArrayOfLifeTimes[action].erase(iterOfLifeTime);
                MyopicArrayOfSerTimes[action].erase(iterOfServiceTime);
                MyopicArrayOfPriorities[action].erase (servedJob);
        }

    jobRem[action]--;
        serverStatus = action; // update the server status
        jobServed++;

        // If the service ends after _maxTime, stop the simulation and do not count the reward
onwards
        if(nextSerEndTime >= _maxTime)
        {
                if(printSimuSteps)
                {
                        out_file<<sysTime<<"    "<<"SimuEnds, _maxTimeReached"<<"   ";
                        for(int j=0; j<_numClasses;j++)
                                out_file<<jobRem[j]<<"  ";
```

WeibExpo.cpp

```
                out_file<<0<<endl;
            }
            goto MYOPIC_STOP;
        };

        // Reschedule the next abandonment after making the selection.
        nextAbandonTime = VALUE_INFINITY;
        for(int j = 0; j<_numClasses; j++)
        {
            if (MyopicArrayOfLifeTimes[j].empty ())
                continue;
            iterOfLifeTime = MyopicArrayOfLifeTimes[j].begin ();
            if(*iterOfLifeTime < nextAbandonTime)
            {
                nextAbandonTime = *iterOfLifeTime;
                nextAbandonClass = j;
            };
        };

        if(printSimuSteps)
        {
            out_file<<sysTime<<"    "<<"serEndAndSendNext"<<"      ";
            for(int j=0; j<_numClasses;j++)
                out_file<<jobRem[j]<<"  ";
            out_file<<serverStatus+1<<endl;
        }
    }
    else //otherwise next event is customer loss
    {
        sysTime = nextAbandonTime;
        jobRem[nextAbandonClass]--;
        jobLost[nextAbandonClass]++;
        nextAbandonTime = VALUE_INFINITY;

        if(printSimuSteps)
        {
            out_file<<sysTime<<"    "<<"Class"<<nextAbandonClass+1<<"Abandonment"<<"      ";
            for(int j=0; j<_numClasses;j++)
                out_file<<jobRem[j]<<"  ";
            out_file<<serverStatus+1<<endl;
        }
```

WeibExpo.cpp

```
                        //Update the next lost job location by removing the one just lost
                        MyopicArrayOfLifeTimes[nextAbandonClass].pop_front ();
                        MyopicArrayOfSerTimes[nextAbandonClass].pop_front();
                        MyopicArrayOfPriorities[nextAbandonClass].pop_front();

                        //Find the time of next abandonment and which class
                        for(int j = 0; j<_numClasses; j++)
                        {
                              if (MyopicArrayOfLifeTimes[j].empty ())
                                    continue;
                              iterOfLifeTime = MyopicArrayOfLifeTimes[j].begin ();
                              if(*iterOfLifeTime < nextAbandonTime)
                              {
                                    nextAbandonTime = *iterOfLifeTime;
                                    nextAbandonClass = j;
                              };
                        };
                  }
                  // calcualte the total remained jobs
                  totalJobs = 0;
                  for(int j = 0; j<_numClasses; j++)
                        totalJobs += jobRem[j];
            }; // end of while loop

            sysTime = nextSerEndTime;
            serverStatus = -1;
            if(printSimuSteps)
            {
                  out_file<<sysTime<<"    "<<"LastJobEndService"<<"     ";
                  for(int j=0; j<_numClasses;j++)
                              out_file<<jobRem[j]<<"  ";
                        out_file<<serverStatus+1<<endl;
            }

            MYOPIC_STOP: //the label for the goto statement above
            myopicJobServed = jobServed;

            // Update real time the sample mean and variance of the difference between fluid PI and myopic
            double myopic_fluid_delta = myopicJobServed - fluidJobServed;
            myopic_fluid_mean_new = myopic_fluid_mean_old + (myopic_fluid_delta - myopic_fluid_mean_old) /
(simRuns + 1);
            double delta = myopic_fluid_mean_new - myopic_fluid_mean_old;
```

WeibExpo.cpp

```
            if(simRuns > 0)
                  myopic_fluid_var = (1- 1.0/double(simRuns)) * myopic_fluid_var
                  + (simRuns + 1) * delta * delta;
            myopic_fluid_mean_old = myopic_fluid_mean_new;

            if(1.96 * (sqrt(myopic_fluid_var/(simRuns+1))) < ACCEPTEDRELDIFF && simRuns > 99)
            {
                  //simRuns + 1 is the number of samples or runs
                  cout<<"Myopic policy simulation stop criteria reached after "<< simRuns + 1<<" runs"<<endl;
                  myopicJobServed = 0.0; // Set the job served to be zero
                  myopicDone = true;
            }
      };

      // simRuns is the run index, which starts from zero and ia one less than the number of runs
      out_file<<simRuns<<"    "<<thetaMuJobServed<<"  "<<fluidJobServed<<"    "<<myopicJobServed<<"   "
            //<<thetaMu_fluid_var<<"      "<<myopic_fluid_var
            <<endl;

      // Stop critieria
      if ( thetaMuDone && myopicDone)
      {
            cout<<"The simulation stop after "<<simRuns+1 <<" runs" <<endl;
            simRuns++; //Due to the break, simRuns is not the actual number of samples, but one less.
            break;
      }
} // end of loop of multi-run of simulation

// Set the final results
setMeanFluidPI(fluid_mean_old);
setMeanThetaMuFluid(thetaMu_fluid_mean_old);
setMeanMyopicFluid(myopic_fluid_mean_old);
setStdDevThetaMuFluid(sqrt(thetaMu_fluid_var/simRuns)); // Out of the loop, simRuns is added by one, so it is
the number of runs at this point
setStdDevMyopicFluid(sqrt(myopic_fluid_var/simRuns));

//out_file<<simRuns<<"  "<<thetaMu_fluid_mean_old<<"  "<<fluid_mean_old<<"    "<<myopic_fluid_mean_old<<"   "
//    <<thetaMu_fluid_var<<"  "<<myopic_fluid_var<<endl;

out_file.close();
};
```

BayesianWeibDeter.cpp

```
void BWProblem::calcPostProbs ()
{
      //loop over all epochs
      for(int timeIndex = 0; timeIndex < _numDecisionEpochs; timeIndex++)
      {
            double timePoint=getEpochTimePoint(timeIndex);
            VecDubTwoDimens postProbAtTimePoint;

            //Loop over all classes
            for(int clsIndex = 0; clsIndex < _numClasses; clsIndex++)
            {
                  VecDouble postProbForThisCls;
                  double sumProbs = 0.0; //sum over all the types but the last one, which is just (1-sumProbs)

                  //Loop over all types
                  for(int typeIndex = 0; typeIndex < _numClasses; typeIndex++)
                  {
                        double postProbForThisType;
                        if(typeIndex < _numClasses - 1) //Calculate when the type is not the last one
                        {
                              double denominator = 0.0;
                              for(int m = 0; m <_numClasses; m++)
                              {
                                    //if the error is zero, just jump to the next m
                                    if(_clsErrors[m][clsIndex] < VALUE_ZERO)
                                          continue;
                                    denominator += _clsErrors[m][clsIndex] * _priorProbs[m]
                                                * exp(pow(timePoint/_beta[typeIndex],_alpha[typeIndex])
                                                - pow(timePoint/_beta[m],_alpha[m]));
                              }
                              if(_clsErrors[typeIndex][clsIndex] < VALUE_ZERO)
                                    postProbForThisType = 0.0;
                              else
                                    postProbForThisType = _clsErrors[typeIndex][clsIndex]*_priorProbs[typeIndex]
                                                      /denominator;

                              sumProbs += postProbForThisType; // get the summation over types
                        }
                        else //This last type is not independent, so just use (1.0-sumprobs)
                        {
                              //If the first J-1 type probs greater than one, report an error.
                              if(sumProbs - 1.0 > VALUE_ZERO)
```

BayesianWeibDeter.cpp

```
                        {
                              cout<<"The sum of J-1 post prob is larger than 1. The program is
terminated!!"<<endl;
                              abort();
                        }
                        postProbForThisType = 1.0 - sumProbs;
                  }

                  //Check the integrity of the resulted value
                  if((boost::math::isnan)(postProbForThisType))
                  {
                        cout<<"NaN issue encountered, the program is terminated!!"<<endl;
                        abort();
                  }
                  if(postProbForThisType - 1.0 > VALUE_ZERO)
                  {
                        cout<<"The post prob is larger than 1. The program is terminated!!"<<endl;
                        abort();
                  }
                  //if(postProbForThisType < VALUE_DUBMIN)
                  //    postProbForThisType = 0.0; //force it to 0.0

                  postProbForThisCls.push_back (postProbForThisType);
            }
            postProbAtTimePoint.push_back (postProbForThisCls);
      }
      _postProbs.push_back (postProbAtTimePoint);
   };
};

void BWProblem::calcRemSurProb ()
{
      // Loop for each class
      for(int clsIndex = 0; clsIndex < _numClasses; clsIndex ++)
      {
            VecDubTwoDimens probOfThisCls;

            // Loop over each true type, of which the purpose is to get service times
            for(int trueTypeIndex = 0; trueTypeIndex < _numClasses; trueTypeIndex ++)
            {
                  double serTime = 1 / _serRate[trueTypeIndex];
                  VecDouble temp;
```

BayesianWeibDeter.cpp

```
                //Loop over all decision epochs
                for(int timeIndex = 0; timeIndex < _numDecisionEpochs; timeIndex ++)
                {
                      double curTime = getEpochTimePoint(timeIndex);
                      double endTime = curTime + serTime;

                      ////for debug
                      //cout<<curTime<<","<<endTime<<","<<clsIndex<<endl;
                      temp.push_back(BayesianFluidODFSol(curTime, endTime, clsIndex));
                }
            probOfThisCls.push_back (temp);
        }
        _remSurProb.push_back (probOfThisCls);

    }
};
```

Appendix D

Matlab Functions

The following Matlab functions are included in this appendix.

- ***calcProb.m***: the function to calculate the one step transition probability in the exponential lifetime and exponential service time scenario in Chapter 2. Refer to equation (2.32).
- ***MRLT.m***: the function to calculate MRL for Weibull lifetimes. Refer to equation (2.47).
- ***BayesianFluid.m***: the function to calculate the integration in equation (3.31) to solve the fluid model when classification is imperfect. It is also used in (3.19) to compute the one step transition probability in these cases.

The command to compile these Matlab functions is ***mcc -W cpplib:libcalcProb -T link:lib calcProb.m MRLT.m BayesianFluid.m binopdf.m***, the result of which is a C++ library file named *libcalcProb.lib*. By importing this file to C++, we can then access any of these Matlab functions from C++. Note that *binopdf.m* is a native Matlab function to calculate pdf (probability density function) values for binomial random variables.

calcProb.m

```
function z = calcProb(combs, mu, theta, L, k, numClass, action, N, b)
warning on all
warning off MATLAB:nchoosek:LargeCoefficient
z = quadl(@probFunc1,0,b);

    function z = probFunc1(x)
    % Intermidate variables
    y = 1;
    c = 0;
    n = L - k;
    n(action) = n(action)-1;
    for i = 1 : numClass
        y = combs(i) .* y .* (1- exp( -1 .* theta(i).* x)).^ n(i);
        c = c + theta(i) .* k(i);
    end
    y =  y .* (exp(-1 .* x .* (c + mu(action))));

    z = mu(action) .* y .* N;
    end
end
```

MRLT.m

```
% Mean Remaining Life Time Or time to failure
function rel = MRLT(alpha, beta, t)
% calcualtion of b- the integration upperbound
b = (t .^ alpha - beta .^ alpha .* log(1e-7)) .^ (1.0 ./ alpha) - t;
rel = quadl(@meanRemLifeTimeFunc,0,b);

    function z = meanRemLifeTimeFunc(s)
          z = exp((t ./ beta).^ alpha - ((t+s) ./ beta) .^ alpha) ...
              .* s .* alpha ./ beta .*(((t+s) ./ beta) .^ (alpha-1));
    end
end
```

BayesianFluid.m

```
% Computation of the integration of hazard rate over a time period [startTime, endTime]
function z = BayesianFluid(numClasses, alpha, beta, priorProb, clsErrorsByCls, clsIndex, startTime, endTime)
z = quadl(@fluidModel,startTime,endTime);
      .
    function z = fluidModel(t)
        z = 0.0;
        sumProb = 0.0; % Calculate the sum over posterior probabilities of the first J-1 types

        %Loop over each type except the last one
        for typeIndex = 1 : (numClasses - 1)
            % Hazard rate of this type at time t
            hazardRate = alpha (typeIndex) .* t .^ (alpha(typeIndex) - 1) ./ beta(typeIndex) .^ alpha(typeIndex);

            %Calculate the posterior probability p_(typeIndex)(clsIndex)(t)
            denom = 0.0;
            for m = 1 : numClasses
                if(clsErrorsByCls(m) <= 0.0) % if the error is zero, jump to the next one.
                    continue;
                end
                denom = denom + clsErrorsByCls(m) .* priorProb(m) .* exp((t ./beta(typeIndex)) .^ alpha(typeIndex) -
(t ./beta(m)).^ alpha(m));
            end
            if clsErrorsByCls(typeIndex) < 1e-5
                postProb = 0.0;
            else
                postProb = clsErrorsByCls(typeIndex).* priorProb(typeIndex) ./ denom;
            end
            sumProb = sumProb + postProb;
            z = z +  postProb .* hazardRate;
        end

        %Include the last type
        hazardRate = alpha (numClasses) .* t .^ (alpha(numClasses) - 1) ./ beta(numClasses) .^ alpha(numClasses);
        z = z + (1 - sumProb) .* hazardRate;
    end
end
```