Approximate Dynamic Programming For Some Queueing Problems

by

Elias Vyzas

Bachelor of Engineering in Mechanical Engineering,
Imperial College, London (1994)

Submitted to the Department of Mechanical Engineering and the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Science in Mechanical Engineering

and

Master of Science in Electrical Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1997

© Massachusetts Institute of Technology 1997. All rights reserved.

Author

Department of Mechanical Engineering
January 31, 1997

Certified by

John N. Tsitsiklis
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by

Jean-Jacques E. Slotine
Associate Professor of Mechanical Engineering & Information Sciences
Thesis Reader

Accepted by

Arthur C. Smith
Chairman, Committee on Graduate Students
Department of Electrical Engineering and Computer Science

Accepted by

Ain A. Sonin
Graduate Officer
Department of Mechanical Engineering
Approximate Dynamic Programming For Some Queueing Problems

by

Elias Vyzas

Submitted to the Department of Mechanical Engineering and the
Department of Electrical Engineering and Computer Science
on January 31, 1997, in partial fulfillment of the
requirements for the degrees of
Master of Science in Mechanical Engineering
and
Master of Science in Electrical Engineering

Abstract

In this thesis we consider some queueing problems where the system incurs a cost for the customers waiting in the queues. Our objective is to determine a policy that minimizes the long-term cost in the system. In that context, we apply and compare several Approximate Dynamic Programming methods.

We first consider a simple 3-queue network. An exact optimal solution can be obtained, using Exact Dynamic Programming. An optimal solution under specific fluid model assumptions is also proposed by F. Avram, D Bertsimas and M. Ricard in their 1994 paper "Fluid Models of Sequencing Problems in Open Queueing Networks: An Optimal Control Approach." We find that the fluid model approximation gives results very close to the true optimal values. We also find that Approximate Policy Iteration methods using simulation and quadratic approximation give very satisfactory results, within 1% from the exact solution.

We then consider a much harder, 10-queue network. We again use simulation and quadratic function approximation, as part of the Iterative Approximate (Optimistic) Policy Iteration method. We also evaluate several heuristic methods: the First-Buffer-First-Serve, the Last-Buffer-First-Serve, the $C_\mu$-Rule as well as the newly introduced Ordinal Optimization. While local minima make it difficult for Approximate Dynamic Programming to perform a lot better than the existing heuristics, still it gives slightly improved results as does Ordinal Optimization.

Thesis Supervisor: John N. Tsitsiklis
Title: Professor of Electrical Engineering and Computer Science

Thesis Reader: Jean-Jacques E. Slotine
Title: Associate Professor of Mechanical Engineering & Information Sciences
Acknowledgments

I would like to thank my supervisor, Professor John Tsitsiklis for all his help, support and inspiration, as well as the reader of my thesis, Professor Jean-Jacques Slotine. I would also want to thank Professor Elias Gyftopoulos, for his advice and his belief in me.

Above all, I thank my parents for sacrificing everything they have for me. I deeply appreciate the support of all my friends, and especially the one joining me in the numerous allnighters for the thesis.

This research was supported by the NSF under grant ECS 9216531 and by EPRI under contract 8030-10.
Contents

1 Introduction ................................................................. 13
   1.1 Preliminary Discussion .............................................. 13
   1.2 Problem Statement .................................................. 13
   1.3 Research: The 3-Queue Network .................................... 14
       1.3.1 Exact Solutions ............................................. 14
       1.3.2 Approximate Architecture Considerations ................. 14
   1.4 Research: The 10-Queue Network ................................... 17
   1.5 Thesis Summary ................................................... 17

2 Background ................................................................. 19
   2.1 Overview .......................................................... 19
   2.2 Dynamic Programming .............................................. 19
       2.2.1 Exact Dynamic Programming ................................ 23
       2.2.2 Simulation Methods Using Lookup Table Representation . 25
       2.2.3 Approximate Dynamic Programming Using Compact Repre-
            sentations .................................................... 28
   2.3 Other Methods ..................................................... 30
       2.3.1 Heuristic Methods .......................................... 30
       2.3.2 The Fluid Model Approximation ............................ 31

3 Queueing Systems: A 3-Queue Network ................................. 33
   3.1 Continuous Markov Chains and their Uniformization ............ 33
   3.2 The 3-Queue Network Model ....................................... 34
3.2.1 Assumptions ........................................ 34
3.2.2 Uniformized DP Equation .......................... 37
3.3 The Fluid Model Approximation ......................... 40
3.4 Interesting Instances (Choice of Parameters) ..... 43
3.5 Comparison of Exact Solution to Fluid Model Approximation ... 44
   3.5.1 Implementation & Results ...................... 44
   3.5.2 Conclusions .................................... 45
3.6 Approximate Architectures for $V^*$ .................... 48
3.7 Supervised Fitting of $V^*$ .......................... 49
   3.7.1 Results ...................................... 50
   3.7.2 Conclusions .................................. 51
3.8 Approximate Policy Iteration ........................ 53
   3.8.1 Implementation & Results ...................... 53
   3.8.2 Conclusions .................................. 54
3.9 Further Discussion on Simulation Issues ................. 55
3.10 Results ............................................. 56
3.11 Conclusions ......................................... 57

4 Queueing Systems: A 10-Queue Network ...................... 65
4.1 The 10-Queue Network Model .......................... 65
   4.1.1 Assumptions ................................ 66
   4.1.2 Useful Conclusions from the 3-Queue Network Examined Previously (Chap. 3) ...... 69
4.2 Interesting Instance (Choice of Parameters) .......... 70
4.3 Heuristic Methods ................................... 71
4.4 Ordinal Optimization ................................ 72
4.5 Approximate Iterative (Optimistic) Policy Iteration ... 73
4.6 Implementation & Results ................................ 73
4.7 Conclusions ......................................... 78
List of Figures

3-1 The 3-queue network ................................................. 35
3-2 Deterministic choice of initial states: $MSE_2$ vs. number of sample
points per queue ....................................................... 60
3-3 Deterministic choice of initial states: $MSE_2$ vs. step size .......... 60
3-4 Deterministic choice of initial states: $MSE_2$ vs. number of repetitions
3-5 Random choice of initial states: $MSE_2$ vs. number of initial states 61
3-6 Random choice of initial states: $MSE_2$ vs. number of initial states
(detail) ................................................................. 62
3-7 Longer trajectories: $MSE_2$ vs. total number of estimates obtained 63
3-8 Longer trajectories: $MSE_2$ vs. length of trajectories .................. 63
3-9 Longer trajectories: $MSE_2$ vs. length of trajectories (detail) ...... 64
3-10 Combined Results: $MSE_2$ vs. total number of transitions .......... 64

4-1 The 10-queue network .................................................. 67
4-2 Average cost per time step vs. number of transitions .................. 74
4-3 Scatter plot of the average costs for the 2036 trials .................... 75
4-4 Histogram: The distribution of costs .................................. 76
4-5 Histogram: The distribution of costs (detail) ........................... 76
4-6 Parameter values vs. number of iterations ............................. 78
4-7 Average cost vs. number of iterations .................................. 79
List of Tables

3.1 Optimal policies (classes to be served) under the Fluid Model Approximation for \( x_1 > 0 \) .................................................. 41
3.2 Optimal policies (classes to be served) under the Fluid Model Approximation for \( x_2 = 0 \) .................................................. 42
3.3 Parameters resulting in different optimal policies. * threshold policies 43
3.4 Number of different decisions under light and under heavy load for different test cases (out of 9261 states) ......................... 46
3.5 Raw and normalized inputs and target outputs to be fitted ............ 49
3.6 Results: 2-layer Perceptron approximation ............................. 52
3.7 Exact Solution (ES), Approximate Policy Iteration (API), Simulation-Based Approximate Policy Iteration (SBAPI). Comparative results .......................... 54
3.8 Results: Random choice of starting states ................................ 57
3.9 Results: Deterministic choice of starting states ......................... 58
3.10 Results: Longer trajectories ............................................. 59
4.1 Average costs under different policies ................................. 74
Chapter 1

Introduction

1.1 Preliminary Discussion

A multiclass queueing network is one with several types of customers, differing in their arrival processes, service requirements, as well as costs per unit of waiting time. In such networks, the total cost incurred by the system can be greatly affected by the sequencing and routing policies implemented. Unfortunately, except in very few cases, no analytical solutions exist even for the simplest problems. As a result, simulation is the most common tool in evaluating any new methods.

1.2 Problem Statement

A large part of this thesis analyzes a simple queueing network, with two servers in tandem and three queues, two of which are competing for service in the first node. The customers arrive according to Poisson distributions and each server serves one customer at a time, with exponentially distributed service times. The system incurs a cost per time step per customer waiting in queue. The goal is to determine a policy that minimizes the cumulative discounted long-term cost in the system.

A much more complicated problem is then tackled: a reentrant line with ten queues and five nodes. Decisions have to be taken in three of the nodes. We allow for different service rates for different types of customers waiting to be served by the
same server. The goal is again to find a policy that performs better than the heuristic methods.

1.3 Research: The 3-Queue Network

1.3.1 Exact Solutions

For such a small-scale problem, exact solutions can be obtained using Exact Dynamic Programming (DP), under some basic assumptions like the existence of an upper limit in the number of customers any queue can hold at any time. One value for the cost-to-go is stored for every possible initial state in the system (the so-called "Lookup Table representation"). One of the methods that can yield the optimal solution is called Exact Policy Iteration, and works as follows: in the state, an arbitrary policy \( \mu \) is chosen. \( V^{\mu} \), the value function (or else the cost-to-go vector) under policy \( \mu \), is then evaluated using the equation ([2]):

\[
V^{\mu} = C_{\mu} + \beta P_{\mu} V^{\mu}
\]

(1.1)

where \( C_{\mu} \) is the one-step cost vector and \( P_{\mu} \) is the transition probability matrix under policy \( \mu \), while \( \beta \) is the discount factor. Then, a new policy is determined through a Policy Iteration step. This can be summarized as follows ([2]):

\[
\mu' = \arg\min (C_{\mu'} + \beta P_{\mu'} V^{\mu})
\]

(1.2)

where \( \mu \) is the current policy, \( \mu' \) is the new, improved policy, and the minimization takes place over all possible policies.

1.3.2 Approximate Architecture Considerations

When concerned with real-life problems, unfortunately, the state space becomes very large, and there is no way of storing one value for every state, or of obtaining an exact solution. As a result, it is thought very important that one tests and evaluates the
effectiveness of approximate methods. One class of such methods, Approximate policy Iteration, generally works as follows: First, assuming an arbitrary policy $\mu$, simulation is used to obtain enough information about the behavior of the system, and thus estimate $V^\mu$. Then, the results are fitted with some kind of function approximator, and this approximation is in turn used for the next step of the -approximate now-policy iteration.

**Choice of Starting State**

A direct advantage of using simulation is the fact that one does not have to obtain values for *every* single state. In addition to that, one does not have to obtain *very* accurate values either, because the function approximator will do the necessary averaging ("smoothing"). As a result, one can choose to start a simulation run from every possible state, or just use a grid of states (start a simulation from every other state for example). Also, one can choose the starting states at random. Another possibility is the use of a longer simulation (a trajectory through more states) which will provide results for the cost-to-go of several states instead of only one.

**Choice of Function Approximator**

Furthermore, one can choose from a large variety of function approximators, always keeping in mind that increased accuracy means increased computer time. As a guideline, some of the choices applicable to the particular problem are the typical quadratic or cubic least squares fitting, or the more sophisticated Multilayer Perceptron (MLP) ([6], [7]).

**Choice of the Approximate DP Method**

Last but not least comes the choice of optimization method. Here, things are more interesting: For some of the methods, there are no guarantees about reaching or even approaching an optimal solution. In some cases, there is even a possibility of divergence. Two optimization methods ([5]) tried here are the
• Approximate Policy Iteration,

• Approximate Iterative (Optimistic) Policy Iteration,

We would like to know if the above methods actually work, and also how well they work, relative to other, non-DP methods. The exact solutions are readily used as an objective basis for comparison and evaluation of the different approximate methods tested.

The Fluid Model Approximation

In several problems, a small assumption-approximation could make the problem a lot easier. There are times where analytical solutions exist for problems with very similar structure to the one of interest. Such is the case under the fluid model approximation of a queueing network, as is shown in [1]. Under this assumption, instead of integer number of customers, there are continuous fluid flows; instead of exponential rates of arriving customers, there are deterministic fluid velocities. Thus a good enough approximate optimal solution can be found for a large group of previously intractable queueing systems, including the one in interest here.

Consequently, one would like to compare these analytical results obtained from the approximate problem with the exact solutions mentioned above, to validate the exact solutions and at the same time to evaluate the quality of the fluid model assumption.

Use of Results

The experience drawn out of the 3-queue network, gives us some insight about the different approaches to queueing problems. This is then used in cases where the size of the problem does not allow one to examine all the possible methods, but rather choose the ones that seem most promising and hope for the best.
1.4 Research: The 10-Queue Network

Building on experience gained from the 3-queue system, we try to cope with a much more difficult problem, a 5-server, 10-queue network. In this problem, several complications exist:

- Removal of constraints.
  Nodes serving more than one type of customers are allowed to have different service rates and also different costs associated with each type. Also, the number of customers in each queue is now unlimited.

- Large amount of free variables.
  The same options concerning the function approximator, the initial state and the optimization method, still exist. In addition to them, the arrival rate, the service rates, the costs, the normalization factor, highly affect the nature of the problem, as well as the rate and/or the success of the training. The choice of at least some parameters will thus have to be arbitrary.

- Need for a new performance measure.
  Due to the size and the above mentioned difficulties of the system, no exact solutions exist. That creates the need for an alternative means of evaluating the performance of approximate methods. For that reason, heuristic policies similar to First-Come-First-Serve and Last-Come-First-Serve methods, are used as indicators.

1.5 Thesis Summary

In the chapters that follow, some background information is given about the various Exact and Approximate Dynamic Programming, and traditional heuristic methods which are relevant to the problems tackled (2). Several such methods are then tried on the 3-queue network which is studied in depth in 3. Finally, the more complex 10-queue network is examined in 4.
In this thesis, we see that several variations of Approximate Dynamic Programming produce very satisfactory results in the small 3-queue problem. We also check the validity of the Fluid Model approximation on the same problem and explore the effect of correlated estimates on the quality of the estimation. In addition to these, we see that similar methods produce results comparable to the heuristic ones that exist for the 10-queue network. We also find that Ordinal Optimization, a new heuristic method with broad applications gives equally good results.
Chapter 2

Background

2.1 Overview

In the following sections one can find a short analysis of the various theories, methods, and tools assumed as known in the rest of the thesis, together with the most important references.

The basic tool used is the theory of Dynamic Programming (DP for short). Some information is given about Exact DP, and then several aspects of Approximate DP are touched upon; approximate methods, approximation functions. Finally, other (non-DP) methods used are briefly discussed.

2.2 Dynamic Programming

The Dynamic Programming methods presented here concern a discrete-time finite-state system. They also need the existence of a cost, additive over time and dependent on the states visited and possibly on the controls chosen.

Let us assume that in time $k$ all the information about the current state of the system is summarized in variable $x_k$. Let us also assume that $u_k$ is the control chosen from the set $U_k$ of all the possible controls when in state $x_k$. The next state of the system $x_{k+1}$, depends not only on the current state $x_k$ and the control $u_k$, but also on a random disturbance $w_k$, unknown at the time $u_k$ is chosen, but belonging to a
known set $W_k$. Given that $N$ is the total number of state transitions in the system (horizon), the dynamics of the system can be described by:

$$x_{k+1} = f_k(x_k, u_k, w_k), \quad k = 0, 1, ..., N - 1,$$

(2.1)

The random disturbance mentioned above can be seen as follows: Given the state at time $k$ is $x_k = i$ and the control chosen is $u$, the probability that the next state $x_{k+1}$ will be $j$ is given by

$$p_{ij}(u) = p(x_{k+1} = j | x_k = i, u)$$

(2.2)

Thus we can equivalently write any dependence of the cost on $w_k$ as dependence on the next state $x_{k+1}$. Then, at the $k$th transition, we incur a cost $\beta^k g(i, u, j)$, where $g$ is a given function and $\beta$ is a positive scalar with $0 < \beta \leq 1$, called the discount factor. The cumulative cost-to-go starting from state $i$ is denoted by $V(i)$ and is the sum of the discounted costs incurred in every transition.

We can now make some simple distinctions between different sorts of problems:

- Existence of model vs. Monte Carlo simulation: When there is explicit knowledge of the cost structure and the transition probabilities, we say we have a model of the system. In many cases, such a model is not available or is too complicated to be analyzed, but instead, the system and the cost structure can be simulated. By the term simulation we generally mean a computer program which, given the necessary input (state $i$, control $u$) replicates the real world conditions (transition probabilities $p_{ij}$) and returns an output (transition cost $g(i, u, j)$). By repeatedly simulating and accumulating statistics, we can for example evaluate the average expected cost per stage for a given policy or other important information.

- Lookup Table vs. Compact representation: In problems with a moderate number of states, a Lookup Table representation of the cost-to-go function is used, in the sense that a separate variable $V(i)$ is kept in memory for each state $i$. In many cases though, the number of states is too large, making it impractical to
store a value for each. That is both because of memory limitations and because it is not useful to store so much information; it is redundant. In such cases, we make use of compact representations where \( V \) is approximated by a function of a smaller set of parameters.

- **Finite vs. Infinite Horizon problems:** We can also distinguish between finite horizon problems, where the cost accumulates over a finite number of stages, say \( N \), and infinite horizon problems where the cost accumulates indefinitely. We have \( \mu_k(x_k) \) as the set of decisions used for each possible \( x_k \). In other words, if at time step \( k \), \( x_k = i \), then the decision \( u_k \) is given by: \( u_k = \mu_k(x_k = i) \).

Then, in \( N \)-stage problems, policy \( \pi \) is the set \( \{\mu_0, \mu_1, \ldots, \mu_N\} \) and its expected cost starting from the initial state \( x_0 = i \) is:

\[
V^*_N(i) = V^*_N(x_0 = i) = E \left\{ \beta^N G(x_N) + \sum_{k=0}^{N-1} \beta^k g(x_k, u_k(x_k), x_{k+1}) \mid x_0 = i, \pi \right\}
\]  

(2.3)

where \( \beta^N G(x_N) \) is a terminal cost for ending up with final state \( x_N \), and the expected value is taken with respect to the probability distribution over all the possible state sequences \( \{x_0, x_1, \ldots, x_N\} \). This probability distribution depends only on the initial state \( x_0 = i \) and the policy \( \pi \). The optimal cost from state \( i \), that is, the minimum \( V^*_N(i) \) over all \( \pi \), is denoted by \( V^*_N(i) \).

In infinite horizon problems, the total expected cost associated with an initial state \( x_0 = i \) and a policy \( \pi = \{\mu_0, \mu_1, \ldots\} \) is

\[
V^*(i) = V^*(x_0 = i) = \lim_{N \to \infty} E \left\{ \sum_{k=0}^{N-1} \beta^k g(x_k, \mu_k(x_k), x_{k+1}) \mid x_0 = i, \pi \right\}
\]  

(2.4)

The optimal cost from state \( i \), that is, the minimum of \( V^*(i) \) over all \( \pi \), is denoted by \( V^*(i) \). Of particular interest in infinite horizon problems are stationary policies, which are policies of the form \( \pi = \{\mu, \mu, \ldots\} \). For brevity, we refer to \( \{\mu, \mu, \ldots\} \) as the stationary policy \( \mu \). The corresponding cost from state
$i$ is denoted by $V^\mu(i)$. We say that $\mu$ is optimal if $V^\mu(i) = V^*(i)$ for all states $i$. In infinite horizon problems, the optimal policies are typically stationary.

- The Discount Factor ($\beta$): The meaning of $\beta < 1$ is that future costs matter less than if the same costs incurred at the present time. Its use though has more advantages; the more we advance into the future, the less accurate our predictions are. So, we want to diminish the importance of costs occurring far into the future. In addition to that, by introducing a discount factor we can truncate the simulation after some point where terms become negligible, and thus tackle infinite horizon problems as if they were finite horizon ones.

In cases where we choose not to introduce a discount factor (or else $\beta = 1$), it may turn out that the additive cost is infinite. In most such cases the average cost per stage, defined by

$$\lim_{N \to \infty} \frac{1}{N} V_N^*(i)$$

is well defined and finite, and thus it is used instead.

Since the infinite horizon cost of a given policy is, by definition, the limit of the corresponding $N$-stage costs under this policy as $N \to \infty$, this will also hold for the optimal cost, i.e.

$$V^*(i) = \lim_{N \to \infty} V_N^*(i)$$

for all states $i$. Another way to see the above, is that the results of a finite horizon simulation typically approximate the infinite horizon cost and the longer the horizon the better the approximation. That is very useful as we will see later.

In finite state space problems, the following holds for all states $i$,

$$V^*(i) = \min_{u \in U(i)} E \{g(i, u, j) + \beta V^*(j)\} \quad \text{for all states } i.$$  

(2.7)

The above equation is known as the Bellman equation.
If \( \mu \) is such that:

\[
\mu(i) = \arg \min_{u \in U(i)} E \{ g(i, u, j) + \beta V^*(j) \}, \tag{2.8}
\]

for all states \( i \), then the policy \( \{ \mu, \mu, ... \} \) is optimal.

Most of the analysis and the methods mentioned in this thesis make use of the above issues, trying to compute \( V^* \) and an optimal policy efficiently. For their proofs and their exceptions one should refer to [3].

### 2.2.1 Exact Dynamic Programming

Methods presented in this section can be applied when an explicit model of the system is available and a Lookup Table representation is used for the problem, i.e. in small or moderate size problems.

**Exact Value Iteration**

Given an \( n \)-state problem, the *Value Iteration method* is based on the *Value Iteration algorithm*:

\[
V_{N+1}(i) = \min_{u \in U(i)} \left[ g(i, u) + \beta \sum_{j=1}^{n} p_{ij}(u) V_N(j) \right], \quad i = 1, \ldots, n, \tag{2.9}
\]

which can be derived from Eq. (2.3), and where:

\[
g(i, u) \equiv E \{ g(x_0, \mu_0(x_0), x_1) \mid x_0 = i, \pi \} = \sum_{j=1}^{n} p_{ij}(u) g(i, u, j) \tag{2.10}
\]

The above algorithm converges to the optimal costs \( V^*(1), V^*(n) \), starting from arbitrary initial conditions \( V_0(1), ..., V_0(n) \). The optimal costs \( V^*(1), ..., V^*(n) \) satisfy Bellman's equation (2.7), which can be also written as:

\[
V^*(i) = \min_{u \in U(i)} \left[ g(i, u) + \beta \sum_{j=1}^{n} p_{ij}(u) V^*(j) \right], \quad i = 1, \ldots, n, \tag{2.11}
\]
and are the unique solution of this equation. Generally, with some exceptions, the Value Iteration method in theory needs an infinite number of iterations. Nevertheless, in practice it tends to converge quickly for problems of moderate size, and it is an important method for calculating the optimal cost $V^*$. 

**Exact Policy Iteration**

An alternative to Value Iteration that always terminates finitely is *Policy Iteration*. We use the fact that the optimal policy is stationary, to only consider stationary policies. We also use the fact that for any stationary policy $\mu$ the costs $V^\mu(1),...,V^\mu(n)$ are the unique solution of the equation

$$V^\mu(i) = g(i, \mu(i)) + \beta \sum_{j=1}^{n} p_{ij}(\mu(i))V^\mu(j), \quad i = 1, ..., n, \quad (2.12)$$

The exact algorithm is as follows: We start with an arbitrary policy $\mu^0$ and generate a sequence of new policies $\mu^1, \mu^2,...$ as follows: Given policy $\mu^k$, we perform a *policy evaluation step*, that computes $V^{\mu^k}(i), i = 1, ..., n$, as the solution of the above system of linear equations (2.12). They can also be solved iteratively, using the DP iteration

$$V_{m+1}^{\mu^k}(i) = g(i, \mu^k(i)) + \beta \sum_{j=1}^{n} p_{ij}(\mu^k(i))V_m(j), \quad i = 1, ..., n, \quad (2.13)$$

converging to the cost $V^{\mu^k}(i)$ for each $i$. We then perform a *policy improvement step*, which computes a new policy $\mu^{k+1}$ as

$$\mu^{k+1}(i) = \arg \min_{\mu \in \hat{U}(i)} \left[ g(i, u) + \beta \sum_{j=1}^{n} p_{ij}(\mu(i))V^{\mu^k}(j) \right], \quad i = 1, ..., n. \quad (2.14)$$

It can be proven that the sequence of policies generated is always improving until we reach an optimal policy, where the algorithm terminates.
2.2.2 Simulation Methods Using Lookup Table Representation

Methods presented here involve the use of simulation in conjunction with a Lookup Table representation. Simulation can be a useful tool, either in substituting the model (if it is not known) or in gathering important statistics of the system (even when the model is known).

Given a trajectory \(x_0 = i, x_1 = j, \ldots\) under a policy \(\mu\), one can use it to estimate \(V^\mu(x_0 = i)\), but also \(V^\mu(x_1 = j)\), etc. This can happen as follows:

- In finite horizon problems without a termination state, say of horizon 10, we can run a trajectory 20 states long \((x_0, x_1, \ldots, x_{20})\), and get the subtrajectories \((x_0, x_1, \ldots, x_{10}), (x_1, x_2, \ldots, x_{11})\), etc. The same can apply to truncated infinite horizon discounted problems.

- In finite horizon problems with a termination state, we run a trajectory \(x_0, x_1, \ldots, x_N\), where \(x_N = T\) is the termination state, which provides an estimate for \(V^\mu(x_0 = i)\). This also gives us subtrajectories \((x_1, \ldots, x_N)\) that estimate \(V^\mu(x_1 = j)\), etc.

Every Visit vs. First Visit

It is very important to note that very often a state is encountered more than once in the same trajectory; say for example that \(x_0\) and \(x_5\) are the same state. Using what we said above, we get two cost samples for this state. Obviously these samples are not independent. The first-visit method only considers the first one, ignoring any subsequent visits to the same state, while the every-visit method takes under consideration all the visits to the state.

Value Iteration Using Monte Carlo Simulation

There are cases where an explicit model is known, but the state space is many orders of magnitude larger than the set of interesting states. By interesting states, we mean
states visited often, states that may arise while following an optimal policy. Then, we may want to have more accurate values for this subset, while we are less interested in the rest of the states. Of course this requires prior knowledge of this interesting subset, but that is beyond our current scope.

Then, we use simulation to get a trajectory, and then update only the \( V(i) \) values of the states \( i \) which were visited, using the Exact Value Iteration formula (2.9) ([3], Chap. 7).

There is also a form of simulation-based Value Iteration without the use of a model, called \textit{Q-learning}. It stores one value for each \textit{state-action pair}, instead of storing one for every state. Further details on the Q-learning method can be found in [3], [5] Chap. 5, and others.

\textbf{Policy Iteration Using Monte Carlo Simulation}

The policy iteration method presented in the previous subsection made use of a model. It can be also applied when a model is not available, with some changes in the way the cost-to-go \( V^\mu(i) \) is obtained. Given the \( m \)th trajectory in which state \( i \) occurred, and assuming the first-visit method, the accumulated cost from that trajectory is

\[ c(x_0 = i, m) = \left[ \sum_{k=0}^{N} \beta^k g(x_k, \mu(x_k), x_{k+1}) | x_0 = i \right] \]  \hspace{1cm} (2.15)

for finite horizon problems, or

\[ c(x_0 = i, m) = \left[ \sum_{k=0}^{\infty} \beta^k g(x_k, \mu(x_k), x_{k+1}) | x_0 = i \right] \]  \hspace{1cm} (2.16)

for infinite horizon problems. In order to obtain an estimate \( V(i) \), we average out \( c(x_0 = i, m) \) for all \( m \)’s:

\[ V(i) := \frac{1}{K} \sum_{m=1}^{K} c(x_0 = i, m) \]  \hspace{1cm} (2.17)
subsequent to the $K$th encounter with state $i$. We can update the cost-to-go estimates $V(i)$ after the end of each trajectory containing state $i$ in its path, using the formula

$$V(i) := V(i) + \gamma_m(c(x_0 = i, m) - V(i))$$

(2.18)

where

$$\gamma_m = \frac{1}{m}, \quad m = 1, 2, ...$$

(2.19)

and where the initial conditions are for all $i$

$$V(i) = 0.$$  

(2.20)

Temporal Differences

It can be proven that, making use of Eq. (2.16), Eq. (2.18) can be rewritten as

$$V(i) := V(i) + \gamma(d_0 + d_1 + ... + d_{N+1}),$$

(2.21)

where the quantities $d_k$, called temporal differences, are defined by

$$d_k = g(x_k, x_{k+1}) + \beta V(x_{k+1}) - V(x_k)$$

(2.22)

The temporal difference $d_k$ represents the difference between the current estimate $V(x_k)$ of the expected cost-to-go, and the predicted cost-to-go,

$$g(x_k, x_{k+1}) + \beta V(x_{k+1})$$

(2.23)

based on the simulated outcome of the current stage. The $l$th temporal difference $d_l$ can be calculated immediately after the transition from $x_l$ to $x_{l+1}$. This gives the opportunity of updating on-line, as soon as $d_l$ becomes available:

$$V(x_k) := V(x_k) + \gamma d_l, \quad l = k, ..., N - 1$$

(2.24)
instead of waiting until the end of the trajectory. This means that after every transition we update our estimates of all the states visited prior to that transition in the current trajectory:

\[ V(x_k) := V(x_k) + \gamma(g(x_l, x_{l+1}) + \beta V(x_{l+1}) - V(x_l)), \quad k = 1, \ldots, l, \quad (2.25) \]

Under the first-visit method, applying the update rule (2.25) is exactly equivalent to applying the original equation (2.18). Under the every-visit method though, there is a small discrepancy between the two update rules: In the on-line form of it (updating before reaching the end of the trajectory), the first-visit version causes an update in the estimate of \( V(x_k) \), affecting the value of the temporal differences used in subsequent updates. The effect of such a discrepancy is analyzed in [12], [13], and [5] Chap. 5.

### 2.2.3 Approximate Dynamic Programming Using Compact Representations

Methods presented here can be applied when a model of the system is not available, or when its size prohibits the use of a lookup table representation. As a result, these methods use a smaller set of parameters to approximate the cost-to-go function over the whole state space. The main idea is to calculate values for the cost-to-go (and/or the optimal policy) of a finite set (usually just a small subset) of states, and then make a least-squares fit of these values with a function of a given type, such as a polynomial function, a Fourier expansion, a wavelet expansion, or the output of a neural network.

A direct advantage of using simulation is the fact that one does not have to obtain values for every single state. In addition to that, one does not have to obtain very accurate values either, because the function approximator will do the necessary averaging ("smoothing"). As a result, one can choose to start a simulation run from every possible state, or just use a grid of states (start a simulation only from states where the number of customers in each queue is a multiple of some integer \( n \)). Also,
one can choose the starting states at random \((\text{random sampling})\). Another possibility is the use of a longer simulation (a trajectory through more than \(N\) states - where \(N\) the horizon) which will provide results for the cost-to-go of several states instead of only one.

**Approximate Value Iteration**

There exists an approximate version of Value Iteration, using compact representations, but it is not exactly related to the methods mentioned in this chapter, and we will not deal with it at all in the present thesis. More information about the algorithms and convergence issues concerning this method can be found in [4].

**Approximate Policy Iteration**

Approximate Policy Iteration generally works as follows: First, assuming an arbitrary policy \(\mu\), simulation is used to obtain enough information about the behavior of the system, and thus estimate \(V^\mu\). Then, the results are fitted with some kind of function approximator, and this approximation is in turn used to calculate a new, improved, policy \(\mu'\) in the next step of the -approximate now- policy iteration.

As outlined in ([5], pp. 271-272), one can see the method as “consisting of four modules:

1. The simulator, which given a state-decision pair \((i, u)\), generates the next state \(j\) according to the correct transition probabilities.

2. The decision generator, which generates the decision \(\tilde{\mu}(i)\) of the improved policy at the current state \(i\) for use in the simulator.

3. The cost-to-go approximator, function \(\tilde{V}^\mu(j, r)\) that is consulted by the decision generator for the approximate cost-to-go values to be used in the minimization.

4. The least squares solver, which accepts as input the sample trajectories produced by the simulator and solves the fitting problem.”
Under the title *approximate policy iteration* it is generally assumed that the first three steps are repeated for a large number of starting states. The results are then fed into the least squares solver (step 4) in batch mode.

**Approximate Iterative (Optimistic) Policy Iteration**

In another version of the above algorithm, the update of the fitting parameters occurs online. This means that all four steps mentioned above are executed consecutively.

For some of the above methods, there are no guarantees about reaching or even approaching an optimal solution. In some cases, there is even a possibility of divergence.

### 2.3 Other Methods

The Dynamic Programming methods are not problem-specific, they can be applied in a variety of contexts. They generally search for an optimal solution, under some assumptions and limitations of the specific problem. Besides these, there exist other methods which are specifically made to deal with queueing problems. They are not related to Dynamic Programming, so they could be seen as another approach and used as a basis for evaluating the quality and the success of the methods mentioned in the previous sections.

Traditionally, *heuristic* methods exist, that are easy to conceive, and easy to implement. Also, another method, using a fluid-model approximation, gives the possibility of more complicated policies than the heuristics under certain circumstances.

#### 2.3.1 Heuristic Methods

Several heuristic methods exist, proposing *static* priority rules. This means that the decision taken is independent of the number of customers waiting in the queues. They differ though, in the criterion which gives the static policy. More specifically, we mention 3 different rules.
• **Cμ-RULE**: Customers of class \( i \) are served at rate \( \mu_i \). The cost per unit time for a customer waiting in a queue of class \( i \) is \( c_i \). Then, the rule is to always choose, out of the non-empty queues, to serve the class with the highest \( c_i \mu_i \). If \( c_i \mu_i = c_j \mu_j \) for some \( i \neq j \) then the tie can be broken arbitrarily. This method is a result of examining each server independently. It is optimal for 1-server networks without feedback, but it may ignore important features of the optimal solution in more complicated cases ([1]).

• **FBFS**: This method can be applied in the case of reentrant lines. In such cases, a customer needs to be served by the same station more than once, during his stay in the network. Then, according to the *First-Buffer-First-Serve* rule, among all the buffers in a server, the buffer first reached by the customers should be served. When this buffer is empty, the buffer reached second (in the same server) should be served, etc.

• **LBFS**: The *Last-Buffer-First-Serve* rule, is the opposite of the above method. According to it, among all the buffers in a server, the buffer reached by the customers last should be served first. When this buffer is empty, then the buffer reached immediately before the last (in the same server) should be served, etc.

### 2.3.2 The Fluid Model Approximation

There are times where analytical solutions exist for problems with very similar structure to the one of interest. Then, a small assumption/approximation could make the problem a lot easier. Such is the case under the deterministic *Fluid Model* approximation of a queueing network, for which a *good enough* approximately optimal solution can be found. Under this assumption, instead of an integer number of customers passing through the system, there are continuous fluid flows; instead of exponential service rates, there are constant fluid velocities through the servers. That results in *good enough* approximately optimal solutions for large groups of previously intractable problems. Part of this thesis tries to evaluate the quality of the method, compared to the exact solution, on the relatively simple 3-queue problem. For a detailed analysis
of the method refer to [1].
Chapter 3

Queueing Systems: A 3-Queue Network

We will now examine how the theory of Dynamic Programming presented in the previous sections can be used in the case of queueing networks. We will present certain necessary assumptions and the general queueing model used hereafter.

3.1 Continuous Markov Chains and their Uniformization

In queueing systems state transitions correspond to arrivals or departures of customers. Both arrival times and service times are assumed random and independent of all earlier transition times, states and controls. We restrict ourselves to systems modelled by continuous-time Markov chains involving a finite or countable number of states. Then, it can be shown that the issues relating to the transition times can be worked out in a way that these models may be analyzed within the discrete-time, constant time-step framework discussed up to now, by means of a process called uniformization. Even most of the cases with service rates depending on the current state and the control chosen can be transformed into constant time-step problems. In order to do that, we allow for fictitious transitions from a state to itself. That way,
low-rate transitions are speeded up, including the probability of the state remaining unchanged after a transition. Thus a uniform transition rate can be established. ([3] pp. 275-280)

According to this model, simultaneous arrivals and/or departures are not allowed. Instead, at each time step, a single state transition has to occur, according to the determined probabilities (rates). This transition may be a true one, namely an arrival or a departure; it may also be a fictitious one, as mentioned above. Also, even if a server is empty, it may still incur a fictitious departure which will be ignored.

3.2 The 3-Queue Network Model

There exist two servers, $s_1$ and $s_2$, with exponential service rates $\mu_1$ and $\mu_2$ (Fig. 3-1). There exist two entry lines, two classes of customers. The first, arrive to $s_1$ according to a Poisson process of rate $\lambda_1$ and wait in queue 1, with $n_1$ customers. After they are served, they arrive to $s_2$ and enter queue 3, with $n_3$ customers. After they are served, they leave the system. The second class of customers arrive to the $s_1$ with a Poisson distribution of rate $\lambda_2$, and enter queue 2, of $n_2$ customers. After they are served, they too leave the system. We define (for now) as a state the number of customers in the three queues at any single time step, $(n_1, n_2, n_3)$. The system incurs a cost for each customer waiting in queue $(c_1, c_2, c_3)$. The goal is to minimize the long-term cost-to-go for each initial state.

3.2.1 Assumptions

There is a number of assumptions made, to make the problem tractable.

- Maximum number of customers in a queue
  
  In this problem it was necessary to have a low number of states in order to apply the methods that need a small-size system and a Lookup Table representation. For that reason, the queues have an upper limit in the number of customers they can hold at any time $(N_1, N_2, N_3)$. It was chosen that $N_1 = N_2 = N_3 = 20$ for
most of the cases examined. When a queue is full, it can still incur a customer arrival, which will be ignored. The state is then defined as \((n_1, n_2, n_3, overflow)\). The overflow variable takes a value \(a = 1, 2, 3\), depending on the queue where the overflow occurred, and a value \(a = 0\) otherwise. Hereafter, whenever mentioning a state \((n_1, n_2, n_3)\), no overflow is assumed, i.e. the complete definition of the state is \((n_1, n_2, n_3, 0)\).

In order to make the model resemble an infinite buffer problem, a high penalty is imposed each time a customer is thrown out of a queue due to overflow. It is not exactly clear what the form of the penalty should be, thus it was arbitrarily defined to be equal to a factor \(c_4\) times the cost of waiting in that queue, \(c_o\). The reasoning is that if this extra customer were kept, it would create a cost:

1. proportional to the cost of waiting in the specific queue; this effect is accounted for by \(c_o\).
2. equivalent to waiting for much more than one time step; this effect is accounted for by $c_4$.

It is not clear what the value of $c_4$ should be. The penalty ($= c_a \times c_4$) should be such that the optimal policy for the limited-buffer system is similar to the optimal policy for the unlimited-buffer one, although this will probably not be possible for the decisions in all states.

- Server is allowed to remain idle

There are cases where it seems plausible that a server remains idle even when there are customers waiting to be served. Such may be the case when the cost of waiting down the line is much higher than where the customer is currently waiting. Consider an example where our 3-queue system has costs ($c_1 = 1, c_2 = 2, c_3 = 10$), service rates ($\mu_1 = \mu_2 = 1$) and is currently in the state $(1, 0, 10)$. Then, intuitively, server $s_1$ should not serve the customer waiting in queue 1, but remain idle instead. The model used gives this option to the server.

- Preemptive service

Let us examine the following case. Given state $(n_1, n_2, n_3)$, it is decided that server $s_1$ will serve a class-2 customer, rather than a class-1 customer. In the next time step, a new customer arrives in queue 1, making the current state $(n_1 + 1, n_2, n_3)$. Then, the system has to decide on a new control for $s_1$. If now it is decided to serve a class-1 customer instead, then it switches. The customer previously being served remains in queue 2. This policy, called preemptive service, is the one adopted here. The alternative, the so-called non-preemptive service, would be to serve the class-2 customer until the end of its service, and only then decide on a new control.

- Discount factor

The performance measure we are actually interested in is the average cost per state. Nevertheless, most of the algorithms presented in the previous chapter converge faster when using a discounted cost-to-go value function. In addition to
that, when evaluating a specific policy, we are interested in obtaining a value for each possible initial state. This is possible, because the states are finitely many, and few. Thus, it was decided to use discounted cost-to-go as the performance measure instead. Values of $\beta = 0.9$, $0.95$ and $0.99$ are used during this study.

- Finite horizon approximation of an infinite horizon problem

Infinite horizon problems have stationary optimal policies as mentioned previously. By introducing a discount factor, we can end the simulations after a finite number of transitions, say $k$, neglecting the cost thereafter, because of the $\beta^k$ factor associated with it. Throughout this study, the lower the discount factor, the shorter the necessary horizon. For example, for $\beta = 0.95$, if the number of transitions is set to $k = 100$, $\beta^k = 0.00592$, and less than $0.6\%$ of the total cumulative cost-to-go of a state is ignored. If the number of transitions is set to $k = 200$, $\beta^k = 3.5 \times 10^{-5}$ ($0.0035\%$). For $\beta = 0.9$, if the number of transitions is set to $k = 100$, $\beta^k = 2.7 \times 10^{-5}$ ($0.0027\%$).

3.2.2 Uniformized DP Equation

Formalizing the above analysis into equations, it should be clear that the following hold:

Cost

The system incurs a cost

$$g(n_1, n_2, n_3, 0) = c_1n_1 + c_2n_2 + c_3n_3$$  \hspace{1cm} (3.1)

when there is no overflow, and a cost

$$g(n_1, n_2, n_3, a) = g(n_1, n_2, n_3, 0) + c_ac_4, \hspace{0.5cm} a = 1, 2, 3$$  \hspace{1cm} (3.2)

when there is overflow in queue $a$.  

37
Transition Rates

- $\lambda_1$: arrival rate of customers in queue 1.
- $\lambda_2$: arrival rate of customers in queue 2.
- $\mu_1$: service rate in server $s_1$.
- $\mu_2$: service rate in server $s_2$.

State Transitions

States with an overflow in queue $a$, necessarily have $n_a = N_a$ customers in that queue. They are equivalent to the states with the same number of customers, without an overflow, but only the equations for $n_a = N_a$ apply to them and their transition cost is given by Eq. (3.2) rather than Eq. (3.1). Given state $i = (n_1, n_2, n_3, a)$, the four possible state transitions, their probabilities and the new state $j$ are as follows:

1. A new arrival in queue 1 with probability

   $$ p_1 = \frac{\lambda_1}{\lambda_1 + \lambda_2 + \mu_1 + \mu_2} \quad (3.3) $$

   ending at state

   $$ j_1 = \begin{cases} 
   (n_1 + 1, n_2, n_3, 0), & n_1 < N_1, \\
   (n_1, n_2, n_3, 1), & n_1 = N_1. 
   \end{cases} \quad (3.4) $$

2. A new arrival in queue 2 with probability

   $$ p_2 = \frac{\lambda_2}{\lambda_1 + \lambda_2 + \mu_1 + \mu_2} \quad (3.5) $$

   ending at state

   $$ j_2 = \begin{cases} 
   (n_1, n_2 + 1, n_3, 0), & n_2 < N_2, \\
   (n_1, n_2, n_3, 2), & n_2 = N_2. 
   \end{cases} \quad (3.6) $$
3. End of service in server $s_1$ with probability

$$p_3 = \frac{\mu_1}{\lambda_1 + \lambda_2 + \mu_1 + \mu_2}.$$  \hspace{1cm} (3.7)

If a class-1 customer was being served, the transition will end at state

$$j_3 = \begin{cases} 
(n_1 - 1, n_2, n_3 + 1, 0), & n_1 > 0, n_3 < N_3, \\
(n_1 - 1, n_2, n_3, 3), & n_1 > 0, n_3 = N_3, \\
(n_1, n_2, n_3, 0), & n_1 = 0.
\end{cases}$$  \hspace{1cm} (3.8)

If a class-2 customer was being served, the transition will end at state

$$j_3 = \begin{cases} 
(n_1, n_2 - 1, n_3, 0), & n_2 > 0, \\
(n_1, n_2, n_3, 0), & n_2 = 0.
\end{cases}$$  \hspace{1cm} (3.9)

4. End of service in server $s_2$ with probability

$$p_4 = \frac{\mu_2}{\lambda_1 + \lambda_2 + \mu_1 + \mu_2}.$$  \hspace{1cm} (3.10)

ending at state

$$j_4 = \begin{cases} 
(n_1, n_2, n_3 - 1, 0), & n_3 > 0, \\
(n_1, n_2, n_3, 0), & n_3 = 0.
\end{cases}$$  \hspace{1cm} (3.11)

**Stable System**

When no limit exists for the number of customers waiting in a queue at any given time, we have an *infinite queue* system. Such a system is defined as stable when the expected number of customers in all its queues remains bounded. A necessary condition for this is the cumulative arrival rate at each server to be strictly smaller than its service rate, or in other words the utilization rate $\rho$ to be strictly less than one. Although in the current system limits have been imposed on the maximum number of customers ($N_a, N_b, N_c$), we would like to test the methods under the stable
system conditions. These conditions translate into

$$\lambda_1 + \lambda_2 < \mu_1$$  \hspace{1cm} (3.12)

and

$$\lambda_1 < \mu_2.$$  \hspace{1cm} (3.13)

or the equivalent

$$\rho_1 \equiv \frac{\lambda_1 + \lambda_2}{\mu_1} < 1$$  \hspace{1cm} (3.14)

and

$$\rho_2 \equiv \frac{\lambda_1}{\mu_2} < 1.$$  \hspace{1cm} (3.15)

The DP Equation

$$V^*(i) = g(i) + \beta \min_{u \in \{1,2\}} \sum_{k=1}^{4} p_k V^*(j_k)$$  \hspace{1cm} (3.16)

where $p_k$ and $j_k$ are defined earlier in this subsection.

### 3.3 The Fluid Model Approximation

When trying to optimize a queueing network, one should keep in mind three characteristics of the system: (a) Sequencing decisions often need to be made dynamically in time, i.e. static policies most of the times are not optima. (b) The problem is combinatorial, and (c) it is stochastic.

Consider a network with Poisson arrival rates and exponential service rates, just like our familiar 3-queue network. Now consider a similar system where, instead of individual customers arriving and being served, there are fluid inflows under constant speeds, equal to the arrival rates and the servers have a constant throughput equal to their service rate. Such a system is called a Fluid Model approximation of the original network. Under this approximation, the stochastic nature of the problem is removed, making it much easier. For relatively small problems, it even allows for analytical solutions. The 3-queue network is one of these problems. The quality of the policies
<table>
<thead>
<tr>
<th>Case</th>
<th>Conditions</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$c_1 \leq c_2$</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>$c_1 &gt; c_2$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$c_1 - c_2 \geq c_3$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$c_1 &gt; c_2$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$c_1 - c_2 &lt; c_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\mu_1 \geq \mu_2$</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>$c_1 &gt; c_2$</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>$c_1 - c_2 &lt; c_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\mu_1 &lt; \mu_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\frac{x_1(t)}{x_3(t)} \geq \frac{c_3}{c_1-c_2} \frac{\mu_1-\lambda_1}{\mu_2-\mu_1}$</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>$c_1 &gt; c_2$</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>$c_1 - c_2 &lt; c_3$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\mu_1 &lt; \mu_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\frac{x_1(t)}{x_3(t)} &lt; \frac{c_3}{c_1-c_2} \frac{\mu_1-\lambda_1}{\mu_2-\mu_1}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.1: Optimal policies (classes to be served) under the Fluid Model Approximation for $x_i > 0$

obtained via this model is yet to be determined.

We formulate the problem as follows: For $i = 1, 2, 3$, let $x_i(t)$ denote the amount of jobs of class $i$ at time $t$. The control variables are 1 if class 1 is to be served or 2 if class 2 is to be served. Then, it can be proven ([1]) that the optimal solution (class to be served) for the case where all $x_i > 0$ is given by Table 3.1.

The first three cases have static optimal policies, while cases 4 and 5 are part of a dynamic policy. The above hold given $x_i > 0$ for all $i$. Interesting situations occur when some $x_i = 0$:

- Under the conditions of case 1, if $x_2 = 0$, the problem becomes equivalent to a system of two queues in tandem. The analysis for this system also, exists in the same paper, [1]. The optimal decisions are given in the following Table 3.2.

- Under the conditions of case 2, if $x_1 = 0$, the system becomes decoupled, and the optimal decision is for server $s_1$ to serve class 2. We denote this as case 10.

- Under the conditions of case 3, if $x_3 = 0$, the optimal decision is for server $s_1$ to serve class 1. We denote this as case 11.
<table>
<thead>
<tr>
<th>Case</th>
<th>Conditions</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>$c_1 \geq c_3$</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>$c_1 &lt; c_3$</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>$\mu_1 \geq \mu_2$</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>$c_1 &lt; c_3$</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>$\mu_1 &lt; \mu_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$z_1(t) &gt; \frac{c_3 \mu_1 - \lambda_1}{c_1 \mu_2 - \mu_1}$</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>$c_1 &lt; c_3$</td>
<td>idle</td>
</tr>
<tr>
<td></td>
<td>$\mu_1 &lt; \mu_2$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$z_1(t) &lt; \frac{c_3 \mu_1 - \lambda_1}{c_1 \mu_2 - \mu_1}$</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.2: Optimal policies (classes to be served) under the Fluid Model Approximation for $x_2 = 0$

Cases 6 – 11 are the only deviations from the rules of cases 1 – 5. In the rest of situations with $x_i = 0$ for some $i$, the optimal decisions are the ones mentioned in cases 1 – 5 for the specific conditions. The following should be noted here: It is obvious that when $x_1 = x_2 = 0$, server $s_1$ remains idle. In addition to those, there are situations where it is optimal for server $s_1$ to remain idle when $x_2 = 0$, despite the fact that $x_1 > 0$. This happens for example when the cost $c_3$ is much higher of the cost of $c_1$ and the service rate in server $s_1$ is higher than or equal to the one in $s_2$. The different cases for $x_2 = 0$ are examined in Table 3.2.

We will try to assess the relation between the optimal solution for the fluid model and the one for the stochastic model. The authors of the paper express the belief that the following general principle holds:

1. The partition (thresholds) of the parameter space ($\lambda$’s, $c$’s, $\mu$’s) that characterizes the optimal policy for the fluid model is identical with the stochastic model.

2. Whenever the fluid model predicts that a static priority rule is optimal, the same policy is optimal for the stochastic model.

3. Whenever the fluid model predicts that a linear threshold curve is optimal, the optimal policy for the stochastic model is also of the threshold type (dynamic),
<table>
<thead>
<tr>
<th></th>
<th>costs</th>
<th>arrival rates</th>
<th>service rates</th>
<th>utilization factors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c_1, c_2, c_3$</td>
<td>$\lambda_1, \lambda_2$</td>
<td>$\mu_1, \mu_2$</td>
<td>$\rho_1, \rho_2$</td>
</tr>
<tr>
<td>case 1</td>
<td>1, 2, 2</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>0.875, 0.80</td>
</tr>
<tr>
<td>case 2</td>
<td>3, 1, 1</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>0.875, 0.80</td>
</tr>
<tr>
<td>case 3</td>
<td>2, 1, 3</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>0.875, 0.80</td>
</tr>
<tr>
<td>cases* 4 and 5</td>
<td>2, 1, 3</td>
<td>0.2, 0.15</td>
<td>0.4, 1.1</td>
<td>0.875, 0.18</td>
</tr>
<tr>
<td>case 6</td>
<td>1, 2, 2</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>0.875, 0.80</td>
</tr>
<tr>
<td>case 7</td>
<td>1, 2, 0.5</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>0.875, 0.80</td>
</tr>
<tr>
<td>cases* 8 and 9</td>
<td>1, 2, 0.5</td>
<td>0.2, 0.15</td>
<td>0.4, 1.1</td>
<td>0.875, 0.18</td>
</tr>
<tr>
<td>case 10</td>
<td>3, 1, 1</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>0.875, 0.80</td>
</tr>
<tr>
<td>case 11</td>
<td>2, 1, 3</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>0.875, 0.80</td>
</tr>
</tbody>
</table>

Table 3.3: Parameters resulting in different optimal policies. * threshold policies

but with a nonlinear threshold curve.

### 3.4 Interesting Instances (Choice of Parameters)

According to the Fluid Model approximation, the optimal policies may be static or dynamic, depending on the parameters ($\lambda$'s, $c$'s, $\mu$'s). We will obtain exact solutions with several sets of parameters, from different parts of the parameter space. We will check the optimality of the proposed static policies and the thresholds between them. More importantly, we will check the quality of the dynamic policy proposed. It is useful to know when the method deviates from the exact solution and the possible reasons for that deviation. The test cases chosen are shown in the Table 3.3.

One should keep in mind that what we call *exact solution* is itself an approximation; the queues have a limited size, and a penalty is imposed when a customer is thrown out of the system, trying to approximate an infinite-queue system. For that reason, we use the information obtained via the Fluid Model approximation, to understand more about the nature of the problem: the best values for the penalty (mentioned in the previous section), the most interesting cases (in the parameter space) to use during the simulations later on, etc.

43
3.5 Comparison of Exact Solution to Fluid Model Approximation

The exact solution is obtained by the direct application of Eq. (3.16). It results in one value for the optimal cost-to-go and another one for the optimal decision, for each state allowed by the system. Each queue $i$ can have $0 - N_i$ customers, making the total number of states equal to

$$N_{total} = (N_1 + 1)(N_2 + 1)(N_3 + 1)$$  \hspace{1cm} (3.17)$$

or 9261, for an upper limit of 20 customers per queue. States with overflow (for example $(n_1, n_2, n_3, 2)$) are ignored, as they are equivalent to the states with the same number of customers but without overflow, with just the penalty added to their cost-to-go (3.2).

We expect that the results concerning states where queues are almost full, are affected by the value of the penalty imposed $(c_4)$, and do not necessarily reflect the optimal costs-to-go and decisions of the infinite-buffer system. Thus, only the exact solution for states where all queues have at least 4 empty places, (at most 16 customers waiting, in the case of an upper limit of 20 customers per queue) will be considered when assessing the quality of the Fluid Model approximation. In turn, the results from the Fluid Model approximation for states where some queues have less than 4 empty places will be used to assess the appropriateness of various $c_4$ values. The $c_4$ values chosen as best approximating the infinite-buffer system will be used in the simulations.

3.5.1 Implementation & Results

Several test cases are examined and the results from the fluid model and the exact solution are compared. The parameters varied are:

- Discount factor: A discount factor of 0.9, 0.95, and 0.99 is tried. Its effect can be assessed by comparing the exact solutions for the different discount factors
with the fluid model (which solves the non-discounted problem).

- Number of steps: Depending on the discount factor, the number of steps taken into account can vary from 100 to 500 (See 3.2.1).

- Costs, arrival and service rates: The policies proposed by the fluid model approximation depend on the relative costs associated with each queue, as well as the customers' arrival rates and the machines' service rates (See 3.4). We are interested in examining the performance under these different cases.

- Penalty: We want to find the most appropriate value for the penalty, for each interesting case. Penalties of 0, 10, 20, 50, 100, and 200 are tried.

The parameter observed is the number of states at which the two policies (the exact solution and the fluid model approximation) are different. We distinguish between light-load and heavy-load states, as defined above. The number obtained for the light-load states is used to assess the quality of the fluid model approximation. On the other hand, the number obtained for the heavy-load states, under various penalties, is used to choose the appropriate value for the penalty. In addition, the total number of states (light- plus heavy-load) at which the two policies are different is also used to understand how varying the discount factor and number of steps may affect the quality of the results of the exact solution.

The most important results are shown in Table 3.4.

### 3.5.2 Conclusions

One thing to notice is the high dependence of the policies on the penalty imposed. Besides that, the results could be separated into three categories.

- Consider the case with costs $(3,1,1)$. It is expected that class 1 should always be served for all $n_1 > 0$. For such simple cases, the Fluid Model yields the optimal policy. This coincides with the policy from the exact solution with a penalty $c_4 = 10$, even with a discount factor of 0.9. Note that the results from the case of $(1,2,2)$ are exactly the same.
<table>
<thead>
<tr>
<th>Discount factor</th>
<th># of steps</th>
<th>arr. rates $\lambda_1, \lambda_2$</th>
<th>serv. rates $\mu_1, \mu_2$</th>
<th>costs $c_1, c_2, c_3$</th>
<th>penalty $c_4$</th>
<th># of different decisions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>1, 2, 2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>1, 2, 2</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>1, 2, 2</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>3, 1, 1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>3, 1, 1</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>3, 1, 1</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>2, 1, 3</td>
<td>0</td>
<td>35</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>2, 1, 3</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>2, 1, 3</td>
<td>20</td>
<td>16</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>2, 1, 3</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>2, 1, 3</td>
<td>10</td>
<td>31</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>2, 1, 3</td>
<td>20</td>
<td>31</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 0.25</td>
<td>2, 1, 3</td>
<td>50</td>
<td>31</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>0</td>
<td>2013</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>10</td>
<td>2018</td>
</tr>
<tr>
<td>0.9</td>
<td>100</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>20</td>
<td>2003</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>0</td>
<td>1573</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>10</td>
<td>1574</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>20</td>
<td>1575</td>
</tr>
<tr>
<td>0.9</td>
<td>200</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>50</td>
<td>1576</td>
</tr>
<tr>
<td>0.9</td>
<td>300</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>0</td>
<td>647</td>
</tr>
<tr>
<td>0.9</td>
<td>300</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>20</td>
<td>633</td>
</tr>
<tr>
<td>0.9</td>
<td>300</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>50</td>
<td>645</td>
</tr>
<tr>
<td>0.9</td>
<td>300</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>100</td>
<td>699</td>
</tr>
<tr>
<td>0.9</td>
<td>500</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>100</td>
<td>694</td>
</tr>
<tr>
<td>0.9</td>
<td>500</td>
<td>0.2, 0.15</td>
<td>0.4, 1.0</td>
<td>2, 1, 3</td>
<td>200</td>
<td>819</td>
</tr>
</tbody>
</table>

Table 3.4: Number of different decisions under light and under heavy load for different test cases (out of 9261 states).

- Consider the case with costs $(2, 1, 3)$ and service rate $\mu_2 = 0.25$. Here it is almost expected that class 2 should be served whenever there is a lot of customers waiting in queue 3. On the other hand, server $c_1$ should serve class 1 customers whenever queue 3 is almost empty. But the threshold is not clear. The Fluid Model results in serving class 1 only when $n_3 = 0$. But, as expected, the stochastic nature of the problem, ignored by the Fluid Model approximation, becomes more important when a queue is almost empty. That explains the small but nonzero number of different decisions between the Fluid Model and
the exact solution, even with the most appropriate choice of penalty; they all occur when \( n_2 = 0, n_3 = 1 \) or \( n_3 = 2 \) and \( n_1 = 1 \) to \( n_1 = 20 \).

Again in this case, the correct choice of penalty eliminates differences between the exact solution and the Fluid Model under heavy load. This penalty is \( c_4 = 10 \) when the discount factor is 0.9 (and our horizon is 100 steps) or \( c_4 = 20 \) when the discount factor is 0.95 (and our horizon is 200 steps).

- Finally, consider the case with costs \((2,1,3)\) and service rate \( \mu_2 = 1.0 \). The Fluid Model proposes a dynamic optimal policy, with a threshold which is independent of \( n_2 \), and proportional to the ratio \((n_1/n_3)\). It can be shown that although this proposed independence from \( n_2 \) is not exactly true, the dependence is insignificant. It also holds that the threshold is dependent almost solely on the ratio of \( n_1 \) to \( n_3 \). Nevertheless, the factor of proportionality does not totally agree with the one proposed by the Fluid Model, and this form of threshold collapses near the extremes (a queue totally empty or totally full).

It is the correlation of two parameters that creates these results: the penalty imposed, and the discount factor used.

If we have a low discount factor \((\beta = 0.9, 100 \text{ steps})\), it yields a policy highly dependent on both the penalty and \( n_2 \), and totally different from the model's.

If on the other hand we use a very high discount factor \((\beta = 0.99, 500 \text{ steps})\), and find an appropriate penalty, we get policies almost independent of \( n_2 \), i.e. a shape very close to the one of the model, though not exactly the same. The appropriate penalty mentioned above was found to be around the value of \( c_4 = 100 \). For values of \( c_4 < 50 \) and \( c_4 > 200 \), the high dependence on \( n_2 \) reappears.

The fact that a high discount factor \((\beta = 0.99)\) yields policies closer to the model's than a low discount factor \((\beta = 0.9)\), was expected because the model tackles a non-discounted problem.
3.6 Approximate Architectures for \( V^* \)

We are interested in investigating the effect of approximating \( V \) over the whole state space by a function of fewer parameters. Listed below are possible such functions, in order of increasing complexity. It is expected that increasing the complexity will improve the results (we will see later what improving the results may mean). Nevertheless, it will also increase the time needed to obtain these results, making a compromise necessary, especially in the harder problems. The number of parameters used will never be more than a few dozens, much less than the number of the values to be fitted (9261). Thus, there is no possibility of overfitting.

Possible Optimizers

In a state \((n_1, n_2, n_3)\), \( V \) may be approximated by:

- a first order polynomial (linear)

\[
\tilde{V}(n_1, n_2, n_3) = r_0 + \sum_{i=1}^{3} r_i n_i \tag{3.18}
\]

- a second order polynomial (quadratic)

\[
\tilde{V}(n_1, n_2, n_3) = r_0 + \sum_{i=1}^{3} r_i n_i + \sum_{i=1, j \geq i}^{3, j=3} r_{ij} n_i n_j \tag{3.19}
\]

- a third order polynomial (cubic)

\[
\tilde{V}(n_1, n_2, n_3) = r_0 + \sum_{i=1}^{3} r_i n_i + \sum_{i=1, j \geq i}^{3, j=3} r_{ij} n_i n_j + \sum_{i=1, j \geq i, k \geq j}^{3, j=3, k=3} r_{ijk} n_i n_j n_k \tag{3.20}
\]

- a neural network

\[
\tilde{V}(n_1, n_2, n_3) = f(n_1, n_2, n_3) \tag{3.21}
\]

where \( f \) is a non-linear function of the inputs.
<table>
<thead>
<tr>
<th>Input</th>
<th>Target Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>non-normalized</td>
<td>n1, n2, n3</td>
</tr>
<tr>
<td>normalized</td>
<td>(\frac{n_1}{N_1}, \frac{n_2}{N_2}, \frac{n_3}{N_3})</td>
</tr>
</tbody>
</table>

Table 3.5: Raw and normalized inputs and target outputs to be fitted

**Procedure**

The fitting criterion used is the minimization of the sum of the squared errors. For the polynomial function approximators (linear, quadratic, cubic) the parameters are evaluated using a standard package ([11]). The Neural Network used is a Multilayer Perceptron (MLP) with nonlinear units (sigmoid functions), trained using the backpropagation algorithm ([7]). No commercial software is used for it.

**Normalized Inputs**

Data normalization means shifting and/or dividing inputs and outputs by constants, so that they all lie in \([0, 1]\) or in \([-1, 1]\). In the specific queueing problem, this means that the normalized input to the function approximator is \((n_1/N_1, n_2/N_2, n_3/N_3)\) and the target output is \(V(n_1, n_2, n_3)/V(N_1, N_2, N_3)\) (Table 3.5).

It turns out that a system where the data to be fitted have been normalized gives smaller errors. Furthermore, many times normalization makes neural networks converge while the non-normalized data fitting algorithm diverged, and increases their speed because higher learning rates are possible.

### 3.7 Supervised Fitting of \(V^*\)

We are interested in comparing the various fitting functions without running long simulations. Thus, the following method is used. The exact solution is obtained, in the same way as before, giving one value for the cost-to-go from every state. Then, the results are fitted with the function of interest, yielding the reduced number of parameters (e.g. 10 for the quadratic function). Next, using the parameters just
obtained, we calculate the approximate value for the cost-to-go from each state, using the appropriate equation from equations (3.18-3.21). These approximate results are now compared with the original ones of the exact solution. The mean squared error can be calculated with one of the two following ways:

\[
MSE_1 = \frac{1}{N_{total}} \sum_{n_1=0}^{N_1} \sum_{n_2=0}^{N_2} \sum_{n_3=0}^{N_3} \left( \hat{V}(n_1, n_2, n_3) - V^*(n_1, n_2, n_3) \right)^2
\]  

(3.22)

or

\[
MSE_2 = \frac{1}{N_{total}} \sum_{n_1=0}^{N_1} \sum_{n_2=0}^{N_2} \sum_{n_3=0}^{N_3} \left( \frac{\hat{V}(n_1, n_2, n_3) - V^*(n_1, n_2, n_3)}{V^*(n_1, n_2, n_3)} \right)^2
\]  

(3.23)

3.7.1 Results

The results of this section are be based on the first definition of the MSE (3.22). The various approximation functions are tested to choose the most appropriate to use in the simulations of the next sections. All the tests presented here are from the test case with costs=(2, 1, 3), \( c_4 = 20 \), \( \lambda_1 = 0.2 \), \( \lambda_2 = 0.15 \), \( \mu_1 = 0.4 \), \( \mu_2 = 1.1 \), a discount factor \( \beta = 0.9 \), with a horizon of 100 steps.

Exact results for this test case range from \( V(0, 0, 0) = 15.20 \) and \( V(10, 10, 10) = 474.25 \) to \( V(20, 20, 20) = 1073.62 \).

- Using a Linear approximator we get an error

\[
MSE_1 = 309.14
\]  

(3.24)

which can be thought of as \( \hat{V}(10, 10, 10) \approx V(10, 10, 10) \pm 17.5 \)

- Using a Quadratic approximator we get an error

\[
MSE_1 = 30.92
\]  

(3.25)

which can be thought of as \( \hat{V}(10, 10, 10) \approx V(10, 10, 10) \pm 5.5 \), an error of about 1%.
• Using a Cubic approximator we get an error

\[ MSE_1 = 4.99 \]  \hspace{1cm} (3.26)

• Using a \((3 - 7 - 1)\) Neural Network as an approximator we get an error as low as \(MSE_1 = 10.46\). A long list of typical results can be seen in Table 3.6. We use a (usually 2-layer) Perceptron, with 3 normalized inputs \(\frac{n_1}{N_1}, \frac{n_2}{N_2}, \frac{n_3}{N_3}\), fully connected, a sigmoid function in the hidden layer(s) and a summation in the output layer. The initial weights are chosen randomly in the range \(w_i = [-0.5, 0.5]\). Several learning rates \((\eta)\) are tried, and a momentum term \((\alpha)\) is introduced, which seems to increase the speed dramatically (See Table 3.6).

### 3.7.2 Conclusions

• In every case studied, feeding the function approximator with normalized data yields better and/or faster results, as expected.

• The cubic performs better than the quadratic, as we expected. Nevertheless, the quadratic is already good enough.

• In order to confirm that the neural network and the back-propagation algorithm work correctly, we use a single-layer perceptron with the 10 terms of the quadratic function as inputs. It then rapidly converges to weights almost exactly equal to the ones given by the quadratic fitting algorithm of [11]. Nevertheless, when trying to find a nonlinear mapping of the inputs, by using a Multilayer Perceptron with a sigmoid function in its hidden layers, it becomes difficult for them to converge to satisfactory results. To reach convergence, a lot of computational time is needed. If that process is to be repeated for each step of the policy iteration, it makes neural networks totally impractical.

• As we saw, the results from the quadratic are very satisfactory, while using less memory and computer time than the cubic function and the neural nets. The
<table>
<thead>
<tr>
<th>run #</th>
<th>$\eta$</th>
<th>$\alpha$</th>
<th>performance/comments (1 epoch = 1000 iterations)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.01</td>
<td>0.0</td>
<td>saturates at $MSE_1 \approx 500$</td>
</tr>
</tbody>
</table>
| 2     | 0.05  | 0.0    | faster than run#1, $MSE_1 < 30$ after 1200 epochs  
$MSE_1 = 17.44$ at 2000 epochs |
| 3     | 0.05  | 0.3    | faster than run#2, $MSE_1 < 30$ after 500 epochs  
$MSE_1 = 14.47$ at 2000 epochs |
| 4     | 0.05  | 0.6    | faster than run#3, $MSE_1 < 30$ after 400 epochs  
$MSE_1 = 13.96$ at 2000 epochs |
| 5     | 0.1   | 0.0    | fast, but saturates early                       |
| 6     | 0.1   | 0.3    | faster than run#4, $MSE_1 = 12.97$ at 2000 epochs |
| 7     | 0.1   | 0.6    | faster than run#5, $MSE_1 = 10.54$ at 2000 epochs |
| 8     | 0.3   | 0.0    | $MSE_1 < 30$ after 400 epochs, almost saturates after 700 epochs  
$MSE_1 = 11.03$ at 2000 epochs |
| 9     | 0.3   | 0.3    | faster than run#7, $MSE_1 < 30$ after 120 epochs  
almost saturates after 300 epochs, $MSE_1 = 10.95$ at 2000 epochs |
| 10    | 0.3   | 0.6    | initially faster than run#7, $MSE_1 < 30$ after 86 epochs  
saturates after 300 epochs, $MSE_1 = 11.70$ at 2000 epochs |
| 11    | 0.5   | 0.0    | $MSE_1 < 30$ after 114 epochs, $MSE_1 = 10.98$ at 2000 epochs |
| 12    | 0.5   | 0.3    | $MSE_1 < 30$ after 122 epochs, $MSE_1 = 10.46$ at 2000 epochs  
close to instability |
| 13    | 0.5   | 0.6    | $MSE_1 < 30$ after 129 epochs, $MSE_1 = 13.48$ at 2000 epochs  
almost diverges, large fluctuations |
| 14    | 0.7   | 0.0    | $MSE_1 < 30$ after 83 epochs, $MSE_1 = 11.73$ at 2000 epochs  
large fluctuations, no improvement after 700 epochs |
| 15    | 0.9   | 0.0    | fluctuates around $MSE_1 \approx 300$ |
| 16    | 0.9   | 0.3    | fluctuates around $MSE_1 \approx 300$ |
| 17    | 0.7   | 0.0    | diminishing $\eta$, $MSE_1 = 18.91$ at 2000 epochs |
| 18    | 0.7   | 0.3    | diminishing $\eta$, faster than run#17,  
$MSE_1 < 30$ after 90 epochs, $MSE_1 = 12.60$ at 2000 epochs |
| 19    | 0.7   | 0.6    | diminishing $\eta$, $MSE_1 < 30$ after 68 epochs,  
$MSE_1 = 11.03$ at 2000 epochs, equally good as best constant $\eta$ |

Table 3.6: Results: 2-layer Perceptron approximation.
best compromise then seems to be the use of a quadratic function approximator for tackling the problem.

3.8 Approximate Policy Iteration

The 3-queue network is a relatively small and simple system. In more complicated problems though, explicit model of the system is not available making the use of simulation necessary. At the same time, a function of fewer parameters is used to approximate the cost-to-go function, due to memory constraints. These two factors (simulation and fitting) are creating inaccuracies and we want to investigate their individual and their combined effect on the quality of the results.

3.8.1 Implementation & Results

The extent of inaccuracies caused by simulation and fitting is studied by comparing results from three different methods (Table 3.7):

- **Exact Solution (ES):** This the original method, analyzed in 2.2.1) and (3.5. Lookup Table representation and the explicit model are used. This means that no approximation takes place.

- **Approximate Policy Iteration (API):** This method is using the technique analyzed in 3.7. Compact representation together with an explicit model are used. After each exact policy evaluation, the results are fitted and used in the decision-making for the next policy improvement step. Comparing API with ES, we can see the effect of using a smaller set of parameters to represent the cost function.

- **Simulation-Based Approximate Policy Iteration (SBAPI):** This method is analyzed in 2.2.3. Compact representation together with simulation are used. Comparing SBAPI with API, we can investigate the effect of using a simulation rather than an exact model where the cost of all the probabilistic transitions was taken into account. Moreover, comparing SBAPI with ES shows the combined effect of the two factors creating inaccuracies.
\[
\text{costs} = (5, 1, 1), \lambda_1 = 0.2, \lambda_2 = 0.15, \mu_1 = 0.4, \mu_2 = 25
\]

<table>
<thead>
<tr>
<th>different decisions</th>
<th>(MSE_1)</th>
<th>(MSE_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES vs. API</td>
<td>0</td>
<td>4.64</td>
</tr>
<tr>
<td>API vs. SBAPI</td>
<td>0</td>
<td>0.56</td>
</tr>
<tr>
<td>ES vs. SBAPI</td>
<td>0</td>
<td>4.65</td>
</tr>
</tbody>
</table>

\[
\text{costs} = (2, 1, 1), \lambda_1 = 0.2, \lambda_2 = 0.15, \mu_1 = 0.4, \mu_2 = 25
\]

<table>
<thead>
<tr>
<th>different decisions</th>
<th>(MSE_1)</th>
<th>(MSE_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES vs. API</td>
<td>1965</td>
<td>2.03</td>
</tr>
<tr>
<td>API vs. SBAPI</td>
<td>786</td>
<td>0.46</td>
</tr>
<tr>
<td>ES vs. SBAPI</td>
<td>2643</td>
<td>2.13</td>
</tr>
</tbody>
</table>

\[
\text{costs} = (2, 1, 3), \lambda_1 = 0.2, \lambda_2 = 0.15, \mu_1 = 0.4, \mu_2 = 25
\]

<table>
<thead>
<tr>
<th>different decisions</th>
<th>(MSE_1)</th>
<th>(MSE_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES vs. API</td>
<td>465</td>
<td>5.88</td>
</tr>
<tr>
<td>API vs. SBAPI</td>
<td>436</td>
<td>2.76</td>
</tr>
<tr>
<td>ES vs. SBAPI</td>
<td>885</td>
<td>6.60</td>
</tr>
</tbody>
</table>

\[
\text{costs} = (2, 1, 3), \lambda_1 = 0.2, \lambda_2 = 0.15, \mu_1 = 0.4, \mu_2 = 1.1
\]

<table>
<thead>
<tr>
<th>different decisions</th>
<th>(MSE_1)</th>
<th>(MSE_2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ES vs. API</td>
<td>785</td>
<td>5.39</td>
</tr>
<tr>
<td>API vs. SBAPI</td>
<td>423</td>
<td>1.15</td>
</tr>
<tr>
<td>ES vs. SBAPI</td>
<td>1208</td>
<td>5.60</td>
</tr>
</tbody>
</table>

Table 3.7: Exact Solution (ES), Approximate Policy Iteration (API), Simulation-Based Approximate Policy Iteration (SBAPI). Comparative results.

### 3.8.2 Conclusions

We see (Table 3.7) that discrepancies are produced both by the simulation and by the fitting. The total (SBAPI vs. ES) error is higher than any of the individual (SBAPI vs. ES, API vs. ES) errors, as expected. And it is (necessarily) less than, or at most equal to, their sum.

The total error is very low, of the order of 1%, which means that at least in the specific problem SBAPI with a quadratic fitting function is a very successful method. This encourages us to use it in more complex problems.
3.9 Further Discussion on Simulation Issues

A direct advantage of using simulation is the fact that one does not have to obtain values for every possible starting state ($N_{total} = 9261$). Several issues arise from that: How many states are needed? Which are the important ones? How accurate should the results obtained by the simulation be, or else, how many independent simulations are needed to get a satisfactory estimate for the cost-to-go function of a state? What is the effect of correlated results? How can we get the best results with the least number of simulations? The methods used to investigate the above issues are briefly analyzed below. Note that when mentioning states, we mean the ones for which an estimate of the cost-to-go function is obtained. It should be mentioned here that we are using the every-visit method (2.2.2). $M_{total}$ is the total number of state transitions simulated and $t$ is the number of transitions used to approximate the cost-to-go function given the discount factor $\beta$.

- Examination of all states
  
  The cost-to-go function is estimated for all possible starting states. If we have $r$ independent simulations for each state ($r$ independent estimates of its cost function), then
  
  $$M_{total} = N_{total} \times r \times t$$  
  
  (3.27)

- Random choice of states
  
  The cost-to-go function is estimated only for randomly chosen states. If we choose $N_{rand}$ states, with $r$ independent simulations for each state, then
  
  $$M_{total} = N_{rand} \times r \times t$$  
  
  (3.28)

- Deterministic choice of states
  
  The cost-to-go function is estimated only for states whose number of customers in each queue is a multiple of an integer step $m$. When $m = 10$ for example, simulations start from all states where the number of customers in each queue is 0, 10, or 20. For $m = 1$, this method is equivalent to the original one, of
simulating from all states.

\[ M_{\text{total}} = \left( \frac{N_1}{m} + 1 \right) \left( \frac{N_2}{m} + 1 \right) \left( \frac{N_3}{m} + 1 \right) \times r \times t \] (3.29)

- Use of longer trajectories

The method is briefly mentioned in the start of subsection 2.2.2. Given a trajectory with \( t_{\text{long}} > t \)

\[ M_{\text{total}} = r \times t_{\text{long}} \] (3.30)

where \( r \) here is the number of long trajectories simulated.

3.10 Results

The results presented in this section are obtained from the case with costs=(2,1,3), \( c_4 = 20, \lambda_1 = 0.2, \lambda_2 = 0.15, \mu_1 = 0.4, \mu_2 = 1.1 \), and a discount factor \( \beta = 0.9 \). The cost-to-go is the sum of the discounted costs from 100 transitions, as usual. The form of the results and the conclusions are similar for all the cases examined, and the amount of information makes it inappropriate to list all the data.

First we consider a random choice of starting states. The dependence of \( MSE_2 \) on the number of starting states can be seen in Table 3.8 and in Figure 3-5.

Let us then consider the deterministic choice of starting states, according to an integer step \( m \), special case \( (m = 1) \) of which is the examination of all states. Table 3.9 shows the \( MSE_2 \) obtained for values of \( m = 1, 2, \ldots, 10 \) and \( r = 1, 2, 3, 5 \), where \( r \) is the number of repetitions per state.

As mentioned above, it is possible to run longer trajectories and obtain estimates for the cost-to-go of multiple states. The results of this method relative to the total number of transitions, the number of starting states and the length of the individual trajectories is shown in Table 3.10.
<table>
<thead>
<tr>
<th>Initial states</th>
<th>Total transitions ($10^3$)</th>
<th>$MSE_2$ ($10^{-3}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>1000</td>
<td>6.919</td>
</tr>
<tr>
<td>5000</td>
<td>500</td>
<td>7.344</td>
</tr>
<tr>
<td>3000</td>
<td>300</td>
<td>6.787</td>
</tr>
<tr>
<td>2000</td>
<td>200</td>
<td>7.989</td>
</tr>
<tr>
<td>1000</td>
<td>100</td>
<td>9.707</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>9.555</td>
</tr>
<tr>
<td>300</td>
<td>30</td>
<td>1.431</td>
</tr>
<tr>
<td>200</td>
<td>20</td>
<td>1.437</td>
</tr>
<tr>
<td>100</td>
<td>10</td>
<td>3.595</td>
</tr>
<tr>
<td>50</td>
<td>5</td>
<td>4.923</td>
</tr>
<tr>
<td>25</td>
<td>2.5</td>
<td>1.334</td>
</tr>
</tbody>
</table>

Table 3.8: Results: Random choice of starting states.

3.11 Conclusions

Observing the results of the deterministic method, one can see that having very accurate values for the cost-to-go values of the starting states is not necessary when there are enough estimates. Only when there are very few estimates, is it useful to have accurate values for them. Otherwise, fitting does the necessary averaging of discrepancies (Figs. 3-2 and 3-3). Another way to see this is in Figure 3-4 where the improvement due to increased accuracy is significant only in large steps.

A similar effect can be seen in Figure 3-5. The effect of non-uniform distribution of initial states creates large errors only when their number is extremely small. There is no substantial improvement in the error by using more than 200 starting states (Fig. 3-6).

We can see that for any given number of estimates, uncorrelated estimates produce a much smaller error (Fig. 3-7).

As expected, both the use of too few estimates or too many correlated estimates have a detrimental effect on the error. When limited to a very small number of transitions, it is much better to use correlated results, in order to have more of them (Fig. 3-8). When allowed to have more transitions, it is not needed to have as many
<table>
<thead>
<tr>
<th>Step</th>
<th>Stepsize $m$</th>
<th>Repetitions</th>
<th>Total transitions ($10^3$)</th>
<th>$MSE_2$ ($10^{-3}$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>21</td>
<td>1</td>
<td>926.1</td>
<td>0.662</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>2</td>
<td>1852.2</td>
<td>0.639</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>3</td>
<td>2778.3</td>
<td>0.683</td>
</tr>
<tr>
<td>1</td>
<td>21</td>
<td>5</td>
<td>4630.5</td>
<td>0.664</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>1</td>
<td>133.1</td>
<td>0.692</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>2</td>
<td>266.2</td>
<td>0.714</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>3</td>
<td>399.3</td>
<td>0.763</td>
</tr>
<tr>
<td>2</td>
<td>11</td>
<td>5</td>
<td>665.5</td>
<td>0.682</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>1</td>
<td>34.3</td>
<td>0.856</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>2</td>
<td>68.6</td>
<td>0.726</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>3</td>
<td>102.9</td>
<td>0.729</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>5</td>
<td>171.5</td>
<td>0.672</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>1</td>
<td>21.6</td>
<td>1.006</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>2</td>
<td>43.2</td>
<td>0.856</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>3</td>
<td>64.8</td>
<td>1.203</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>5</td>
<td>108.0</td>
<td>0.920</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>1</td>
<td>12.5</td>
<td>2.009</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>2</td>
<td>25.0</td>
<td>1.254</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>37.5</td>
<td>1.124</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>5</td>
<td>62.5</td>
<td>1.095</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>6.4</td>
<td>2.684</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>2</td>
<td>12.8</td>
<td>1.586</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>3</td>
<td>19.2</td>
<td>1.429</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>5</td>
<td>32.0</td>
<td>1.154</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>1</td>
<td>2.7</td>
<td>3.750</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>2</td>
<td>5.4</td>
<td>2.338</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>3</td>
<td>8.1</td>
<td>2.174</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>5</td>
<td>13.5</td>
<td>1.713</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1</td>
<td>2.7</td>
<td>4.109</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>2</td>
<td>5.4</td>
<td>1.641</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>3</td>
<td>8.1</td>
<td>1.687</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>5</td>
<td>13.5</td>
<td>1.377</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1</td>
<td>2.7</td>
<td>3.481</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>2</td>
<td>5.4</td>
<td>2.118</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>3</td>
<td>8.1</td>
<td>2.463</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>5</td>
<td>13.5</td>
<td>1.508</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>1</td>
<td>2.7</td>
<td>4.297</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>2</td>
<td>5.4</td>
<td>3.635</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>3</td>
<td>8.1</td>
<td>1.877</td>
</tr>
<tr>
<td>10</td>
<td>3</td>
<td>5</td>
<td>13.5</td>
<td>2.190</td>
</tr>
</tbody>
</table>

Table 3.9: Results: Deterministic choice of starting states.
<table>
<thead>
<tr>
<th>Total transitions ((10^3))</th>
<th>Initial states</th>
<th>Estimates</th>
<th>Length of trajectories</th>
<th>(MSE_2) ((10^{-3}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>5000</td>
<td>5000</td>
<td>100</td>
<td>0.743</td>
</tr>
<tr>
<td>500</td>
<td>2000</td>
<td>302000</td>
<td>250</td>
<td>0.851</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>401000</td>
<td>500</td>
<td>0.761</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>450500</td>
<td>1000</td>
<td>0.981</td>
</tr>
<tr>
<td>500</td>
<td>300</td>
<td>470300</td>
<td>1667</td>
<td>1.245</td>
</tr>
<tr>
<td>500</td>
<td>200</td>
<td>480200</td>
<td>2500</td>
<td>1.473</td>
</tr>
<tr>
<td>500</td>
<td>100</td>
<td>490100</td>
<td>5000</td>
<td>1.853</td>
</tr>
<tr>
<td>500</td>
<td>50</td>
<td>495050</td>
<td>10000</td>
<td>2.212</td>
</tr>
<tr>
<td>500</td>
<td>20</td>
<td>498020</td>
<td>25000</td>
<td>2.662</td>
</tr>
<tr>
<td>100</td>
<td>1000</td>
<td>1000</td>
<td>100</td>
<td>0.837</td>
</tr>
<tr>
<td>100</td>
<td>800</td>
<td>20800</td>
<td>125</td>
<td>0.894</td>
</tr>
<tr>
<td>100</td>
<td>600</td>
<td>40600</td>
<td>167</td>
<td>0.936</td>
</tr>
<tr>
<td>100</td>
<td>500</td>
<td>50500</td>
<td>200</td>
<td>1.143</td>
</tr>
<tr>
<td>100</td>
<td>400</td>
<td>60400</td>
<td>250</td>
<td>0.873</td>
</tr>
<tr>
<td>100</td>
<td>300</td>
<td>70300</td>
<td>333</td>
<td>1.084</td>
</tr>
<tr>
<td>100</td>
<td>200</td>
<td>80200</td>
<td>500</td>
<td>0.962</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>90100</td>
<td>1000</td>
<td>1.173</td>
</tr>
<tr>
<td>100</td>
<td>50</td>
<td>95050</td>
<td>2000</td>
<td>1.636</td>
</tr>
<tr>
<td>100</td>
<td>20</td>
<td>98020</td>
<td>5000</td>
<td>2.607</td>
</tr>
<tr>
<td>50</td>
<td>500</td>
<td>500</td>
<td>100</td>
<td>1.166</td>
</tr>
<tr>
<td>50</td>
<td>400</td>
<td>10400</td>
<td>125</td>
<td>0.879</td>
</tr>
<tr>
<td>50</td>
<td>300</td>
<td>20300</td>
<td>167</td>
<td>1.173</td>
</tr>
<tr>
<td>50</td>
<td>200</td>
<td>30200</td>
<td>250</td>
<td>1.021</td>
</tr>
<tr>
<td>50</td>
<td>100</td>
<td>40100</td>
<td>500</td>
<td>0.944</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>45050</td>
<td>1000</td>
<td>1.035</td>
</tr>
<tr>
<td>50</td>
<td>20</td>
<td>48020</td>
<td>2500</td>
<td>2.122</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
<td>300</td>
<td>100</td>
<td>0.939</td>
</tr>
<tr>
<td>30</td>
<td>200</td>
<td>10200</td>
<td>150</td>
<td>1.220</td>
</tr>
<tr>
<td>30</td>
<td>100</td>
<td>20100</td>
<td>300</td>
<td>0.915</td>
</tr>
<tr>
<td>30</td>
<td>50</td>
<td>25050</td>
<td>600</td>
<td>1.158</td>
</tr>
<tr>
<td>30</td>
<td>20</td>
<td>28020</td>
<td>1500</td>
<td>1.504</td>
</tr>
<tr>
<td>20</td>
<td>200</td>
<td>200</td>
<td>100</td>
<td>2.711</td>
</tr>
<tr>
<td>20</td>
<td>100</td>
<td>10100</td>
<td>200</td>
<td>0.865</td>
</tr>
<tr>
<td>20</td>
<td>50</td>
<td>15050</td>
<td>400</td>
<td>1.353</td>
</tr>
<tr>
<td>20</td>
<td>20</td>
<td>18020</td>
<td>1000</td>
<td>1.355</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>2.188</td>
</tr>
<tr>
<td>10</td>
<td>50</td>
<td>50500</td>
<td>200</td>
<td>2.029</td>
</tr>
<tr>
<td>10</td>
<td>20</td>
<td>8020</td>
<td>500</td>
<td>2.327</td>
</tr>
<tr>
<td>5</td>
<td>50</td>
<td>50</td>
<td>100</td>
<td>8.326</td>
</tr>
<tr>
<td>5</td>
<td>20</td>
<td>3020</td>
<td>250</td>
<td>2.067</td>
</tr>
</tbody>
</table>

Table 3.10: Results: Longer trajectories
Figure 3-2: Deterministic choice of initial states: $MSE_2$ vs. number of sample points per queue

Figure 3-3: Deterministic choice of initial states: $MSE_2$ vs. step size
Figure 3-4: Deterministic choice of initial states: $MSE_2$ vs. number of repetitions

Figure 3-5: Random choice of initial states: $MSE_2$ vs. number of initial states
estimates as possible, but rather to have uncorrelated estimates.

In order to have the least amount of transitions, the best compromise is to have trajectories around 200 – 300 transitions long (100 – 200 correlated estimates per trajectory) and at least 100 independent trajectories (Fig. 3-9).

In Figure 3-10 the optimal method is summarized: When limited to very few transitions, it is better to have correlated estimates (shown as perpendicular lines on the left side of the graph, sets of runs with the same number of transitions and different number of starting states). On the other hand, when using a large number of transitions, uncorrelated estimates give lower error. This graph also shows that the error from all three methods is comparable.
Figure 3-7: Longer trajectories: $MSE_2$ vs. total number of estimates obtained

Figure 3-8: Longer trajectories: $MSE_2$ vs. length of trajectories
Figure 3-9: Longer trajectories: $MSE_2$ vs. length of trajectories (detail)

Figure 3-10: Combined Results: $MSE_2$ vs. total number of transitions
Chapter 4

Queueing Systems: A 10-Queue Network

The 3-queue system examined in the previous chapter is an interesting problem by itself. One can also draw important conclusions which can be useful when trying to tackle harder problems, like the one presented below.

4.1 The 10-Queue Network Model

There exists only one entry line and five servers, $s_{0,3,6}, s_{1,4,7}, s_{2,8}, s_{5},$ and $s_{9}$ (Fig. 4-1). New customers arrive at queue 0 according to a Poisson process of rate $\lambda$. Customers waiting in queue $i (i = 1, \ldots, 8)$ with $n_i$ customers, are served by server $s_i$ and then enter queue $i + 1$, with $n_{i+1}$ customers. Service times are exponentially distributed and equal to $\mu_i$. After the customers are served by server $s_9$, they leave the system. We define as a state the number of customers in the ten queues at any single time step, $(n_0, \ldots, n_9)$. The system incurs a cost for each customer waiting in queue $(c_1, \ldots, c_9)$. The goal is to derive a policy which minimizes the average cost-to-go. More information about the specific system and further references can be found in [10]. The paper finds an optimal solution for such kinds of systems under the assumptions that (a) costs are non-decreasing downstream, i.e. if $i \geq j$, then $c_i \geq c_j$ (which holds in our case), (b) the service times for all buffers in the same machine
are equal (which does not hold in our case), and (c) the last server is the bottleneck of the system (which again does not hold here). It also has a short analysis about reentrant lines, i.e. systems where customers are served multiple times from the same server during their stay in the network like the 10-queue system that is analyzed here (See also [9]).

4.1.1 Assumptions

The 10-queue model adopted here has many similarities, as well as some differences compared to the 3-queue that have to be mentioned. State transitions again correspond to arrivals or departures of customers. Generally, the Uniformized Continuous Markov Chain Model still holds as analyzed in the previous chapter (Section 3.1). Also, the system still provides preemptive service (Section 3.2.1 Preemptive service), as did the 3-queue model. Due to all the above, the possible state transitions and their probabilities follow exactly the same rules as in the 3-queue network (therefore they are omitted here). For the system to be stable, all of the following inequalities must hold (See below for unlimited buffer in queues):

\[
\frac{\lambda}{\mu_0} + \frac{\lambda}{\mu_3} + \frac{\lambda}{\mu_6} < 1 \quad (4.1)
\]

\[
\frac{\lambda}{\mu_1} + \frac{\lambda}{\mu_4} + \frac{\lambda}{\mu_7} < 1 \quad (4.2)
\]

\[
\frac{\lambda}{\mu_2} + \frac{\lambda}{\mu_8} < 1 \quad (4.3)
\]

\[
\frac{\lambda}{\mu_5} < 1 \quad (4.4)
\]

\[
\frac{\lambda}{\mu_9} < 1 \quad (4.5)
\]

Nevertheless, this problem is significantly harder than the one we encountered in the previous chapter. There are no analytical solutions. There are no exact solutions either. We have to approximate solutions and use assumptions which will make the problem computationally tractable. As a result:
Figure 4-1: The 10-queue network
• Servers are not allowed to remain idle

There are cases where it seems plausible for a server to remain idle even when there are customers waiting to be served. Such may be the case when the cost of waiting down the line is much higher than where the customer is currently waiting. Under the model used the server is allowed to remain idle only when at least one of its queues is empty. The optimizer can just decide to serve an empty queue, which is equivalent to remaining idle. On the contrary the server is not allowed to remain idle when there are customers waiting in all its queues. The reason for this is that convergence to an optimal policy becomes much harder when allowing for too many decisions, because of possible local minima. For example, in server $s_{0,3,6}$, when all its queues are non-empty, instead of the possible decisions to be four (serve queue 0, serve queue 3, serve queue 6, stay idle), we reduce them to three (serve queue 0, serve queue 3, serve queue 6). Similarly, in server $s_5$, instead of the possible decisions to be two (serve queue 5, stay idle), we reduce them to one, so we do not really have to deal with a minimization at all. At the same time, this constraint makes the system easier to simulate.

• Unlimited buffer in queues

Consider the case where a limit were imposed to the number of customers in a queue. Even if it were limited to 10 customers, instead of 20 used before, there would be 11 possible values for each queue (0 – 10 customers), which would give a total of $11^{10}$ or 26 billion states. They are far too many to store, to manipulate and more importantly to care individually for each one of them. It seems more logical to lift the constraint and allow for an unlimited buffer in the queues, thus bringing more changes to the model.

• Use of average cost; high-discount factor

An immediate necessity of the unlimited buffer queues is the use of the average cost as the measure of performance, as there is no way of comparing the cost-to-go or the decisions proposed for each state by different methods. A secondary
consequence of the size of the problem is that because we use average cost as a performance measure, we want the training to be as close as possible to the undiscounted case. Therefore a $\beta = 0.99$ is used.

4.1.2 Useful Conclusions from the 3-Queue Network Examined Previously (Chap. 3)

This 10-queue problem is a difficult one. Reasons for that are

- the large number of free parameters involved;
- the large number of different methods and variations of them;
- the lack of prior knowledge in terms of appropriateness of the various methods that exist. There is mostly intuition and belief that drives towards the use of one method over another.

Thus, some conclusions drawn from the study of the 3-queue network in the previous chapter could prove useful, in showing what might work, but also in showing what would probably fail.

Under this context, we have seen that neither the simulation nor the fitting ruined the optimization process and the results have been very satisfactory. Nevertheless, the fitting being successful in a value function spanning a 3-Dimensional space does not guarantee it will be equally successful in a value function spanning a 10-Dimensional one. Also, in the 10-queue network, there is a possibility of a policy making the simulation never visit an important subspace of this huge 10-D space as the theory behind some of the methods assumes.

The 3-queue network also showed that the use of neural networks is often problematic in terms of convergence. It is also unpredictable in terms of the quality of the results. At the same time, the conventional quadratic and cubic functions performed a satisfactory fitting. Therefore, these are the ones tried in the 10-queue network. The quadratic fitting function comprises the constant, 10 linear and 55 quadratic
terms, a total of 66 terms to be trained. Similarly, the cubic fitting function comprises the constant, 10 linear, 55 quadratic and 220 cubic terms, a total of 286 terms to be trained.

The Approximate Policy Iteration worked very well in the previous chapter. Nevertheless, here we do not have a finite number of states to perform full policy evaluations and full policy improvement steps on. Thus, a variation to this method will be used instead, called Approximate Iterative (Optimistic) Policy Iteration ([4]), which was mentioned briefly in 2.2.3 and is explained in more detail later (4.5).

4.2 Interesting Instance (Choice of Parameters)

A number of variables have to be set, in order to define a specific instance to be studied. These are the arrival rate $\lambda$, the service rates $\mu_0, \ldots, \mu_9$, and the costs $c_0, \ldots, c_9$ for waiting in the ten queues. In addition to those, the state(s) the trajectory will start from (initial queue sizes), has to be specified. This is not necessarily important because hopefully the algorithm should converge to a specific policy and to a specific set of parameters for the cost function, irrespective of the initial state(s). Also, the initial state needs to have enough customers to give an original momentum to the iteration but not so much that will cause divergence problems. Considering the above, the following parameters are chosen:

- Arrival rate $\lambda = 1$.
- Service rates:
  
  \begin{align*}
  s_{0,3,6}: & \quad \mu_0 = 2.5, \mu_3 = 3.0, \mu_6 = 6.0, \\
  s_{1,4,7}: & \quad \mu_1 = 8.0, \mu_4 = 4.0, \mu_7 = 2.0, \\
  s_{2,8}: & \quad \mu_2 = 2.0, \mu_8 = 2.5, \\
  s_5: & \quad \mu_5 = 1.5, \\
  s_9: & \quad \mu_9 = 1.5.
  \end{align*}

- Costs $c_0 = \ldots = c_9 = 1$.  

70
• Initial queue sizes \( n_0 = ... = n_9 = 2 \).

Having \( c_0 = ... = c_9 = 1 \) is equivalent to trying to minimize the total service time for the system.

4.3 Heuristic Methods

The 3 different methods applied here, have been explained roughly in 2.3.1. In the specific case of interest, they translate to the following static policies:

• FBFS policy
  In machine \( s_{0,3,6} \), customers waiting in queue 0 have priority. Queue 3 is served only when queue 0 is empty, while queue 6 is served when both queues 0 and 3 are empty. In machine \( s_{1,4,7} \), customers waiting in queue 1 have priority. Queue 4 is served only when queue 0 is empty, while queue 7 is served when both queues 1 and 4 are empty. In machine \( s_{2,8} \), customers waiting in queue 2 have priority. Queue 8 is served only when queue 2 is empty.

• LBFS policy
  In machine \( s_{0,3,6} \), customers waiting in queue 6 have priority. Queue 3 is served only when queue 6 is empty, while queue 0 is served when both queues 6 and 3 are empty. In machine \( s_{1,4,7} \), customers waiting in queue 7 have priority. Queue 4 is served only when queue 7 is empty, while queue 1 is served when both queues 7 and 4 are empty. In machine \( s_{2,8} \), customers waiting in queue 8 have priority. Queue 2 is served only when queue 8 is empty.

• \( C/\mu \)-RULE
  In machine \( s_{0,3,6} \), customers waiting in queue 6 have priority. Queue 3 is served only when queue 6 is empty, while queue 0 is served when both queues 6 and 3 are empty. In machine \( s_{1,4,7} \), customers waiting in queue 1 have priority. Queue 4 is served only when queue 1 is empty, while queue 7 is served when both queues 1 and 4 are empty. In machine \( s_{2,8} \), customers waiting in queue 8 have priority. Queue 2 is served only when queue 8 is empty.
4.4 Ordinal Optimization

The current problem can be tackled using yet another technique. This is the relatively new method of *Ordinal Optimization* ([8]). According to this method if one tries a large-enough sample of random parameters, one will get results close to the optimal solution. Bounds for the quality of the results with respect to the number of different parameter samples tried can be found in the paper. An example mentioned in the there is the following: If 12 numbers are picked at random out of a set of 200 numbers, at least one of them will be in the largest 12 with probability 0.5.

Generally the performance of the method highly depends on the nature of the problem to be solved, like the number of parameters and their possible range, as well as the shape of the cost function. The specific 10-queue system is suitable for reasons explained below:

- Using a quadratic fitting function, we need to randomly choose a set of 66 parameters, taking any real value. Nevertheless, observations as well as knowledge of the system excludes a huge range of values; for example negative linear terms would give an antioptimal policy; the range of the quadratic terms is much smaller than that of the linear ones; the constant term does not affect the policy produced.

- In addition to the above, a small change in the value of some terms usually causes only negligible change in the policies produced. That allows us to choose the random values from a discrete set rather than a continuous one, without severe loss of generality. That is done so that a larger part of the parameter space is explored.

- Most importantly, the cost function of interest is *flat*, in the sense that small changes in the policy cause even smaller changes in the cost function, a very helpful characteristic according to the aforementioned paper.

All the above suggest that with a reasonable amount of trials one could obtain parameters creating a policy with an average cost comparable to other heuristics. We
would like to examine this suggestion.

4.5 Approximate Iterative (Optimistic) Policy Iteration

In this version of Policy Iteration (See [4], and Section 2.2.3), each transition in a long trajectory is considered independently. This happens as follows: Let us consider a problem with a horizon of 100 steps, and a long trajectory of it. Then, after the $(k + 101)$th transition, we can calculate an estimate for the cost-to-go of the state which appeared in the $(k + 1)$th transition. The fitting is adjusted appropriately to incorporate this estimate, and so does the policy (as it depends on the fitting parameters). The adjustment occurs using the following update formula, also known as optimistic TD(0) ([12], [13], [5] Chap 6.3). It should be mentioned here that we are again using the every-visit method (2.2.2), as previously.

\[
\bar{\tau} = \bar{\tau} + \gamma \nabla \tilde{V}(i_k, \bar{\tau}) \left( g(i_k, \bar{\mu}(i_k), i_{k+1}) + \beta \tilde{V}(i_{k+1}, \bar{\tau}) - \tilde{V}(i_k, \bar{\tau}) \right)
\]  

(4.6)

In the case of a quadratic fitting function, there exist 66 parameters, as mentioned above. The policy $\bar{\mu}$ is never calculated explicitly. Given a state, the policy for it is found by applying Eq. (2.14).

4.6 Implementation & Results

Typical runs under a heuristic policy can be seen in Figure 4-2. The average cost for one time step is plotted against the number of transitions we average over. The average costs under the heuristic policies can be seen in Table 4.1, with the lowest being equal to 30.34 under the $C\mu$-Rule.

In order to evaluate the quality of the Ordinal Optimization method, more than 2000 sets of 66 parameters were generated at random. The distribution of results for all the trials can be seen in Figures 4-3, 4-4 and 4-5. The lowest cost that was
Figure 4-2: Average cost per time step vs. number of transitions

<table>
<thead>
<tr>
<th>Policy</th>
<th>Average cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C_\mu$-rule</td>
<td>30.34</td>
</tr>
<tr>
<td>FBFS</td>
<td>60.91</td>
</tr>
<tr>
<td>LBFS</td>
<td>32.71</td>
</tr>
<tr>
<td>Ordinal Optimization</td>
<td>26.95</td>
</tr>
<tr>
<td>Optimistic DP</td>
<td>27.10</td>
</tr>
</tbody>
</table>

Table 4.1: Average costs under different policies
obtained was equal to 26.95, which is better than the heuristics.

In order to evaluate the Approximate Iterative (Optimistic) Policy Iteration, we start from random weights and simulate a long trajectory, during which the weights are being trained. During the training, the system can be controlled in one of the following ways:

1. At each state, using the weights being trained, we calculate the approximate value for the cost-to-go of the possible next states depending on the control and decide accordingly. That way, the system turns out to be very unstable. As the weights are originally arbitrarily chosen and untrained, many decisions taken are anti-optimal, resulting to divergence of the algorithm. In order to avoid divergence, the learning rate has to be reduced dramatically, and the algorithm progresses extremely slowly (Even after 10 days of continuous running in a fast workstation, the parameters are not even close to converging). The alternative to this method is mentioned below.

2. In the start of the simulation, an imposed (heuristic) policy is used to control the system. After the weights reach or approach convergence to some values,
Figure 4-4: Histogram: The distribution of costs

Figure 4-5: Histogram: The distribution of costs (detail)
suggesting a cost function for the problem, we switch to the above method. It is very important to note here that while this cost function is dependent on the policy used, it does not itself create the same policy. For example, a static policy is represented by zero second-order terms. Nevertheless, if a static policy is used to train the parameters, it will give not-zero second-order terms. Then the policy produced by these parameters will be a dynamic policy, probably better than the original.

Another important issue is the choice of learning rate ($\gamma$). Besides the usual techniques of having a constant or diminishing learning rate, another technique is used. As mentioned in 2.2.2, the term in brackets in Eq. (4.6) represents the difference between the current estimate of the expected cost-to-go, and the predicted cost-to-go, and is called temporal difference. An abrupt increase in the absolute value of this temporal difference in subsequent iterations (say by a factor of 5) could be thought of as a sign of divergence. Then, if we decrease the learning rate $\gamma$ used in the next iterations by an order of magnitude, divergence may be avoided. In addition, if the temporal difference decreases, we can increase the $\gamma$ used in the next iterations by a factor of 1.01, while if it increases slightly, we can decrease $\gamma$ by a factor of 0.095. In addition to these, we can have minimum and maximum allowable values for $\gamma$. This technique seems to improve the first method mentioned above dramatically, both in speed and in quality, although the second method still seems much more promising.

The simulation can be seen as comprising two sessions, repeated consecutively. The first is the training session, where a heuristic method gives the policy, and the weights are trained. The second is the testing session, where the weights produce the policy and thus their quality is examined. Actually, the use of testing sessions is much more important. As mentioned in the previous chapter (3.11), the trajectories should not be too long because correlated results have detrimental effects on these methods. Thus, the interference of a testing trajectory helps eliminate the correlation and potentially drives the system to some other part of the state-space which the given policy never visited.

The convergence of weights when using the second method can be seen in Figure
4-6. All the weights were initially set to 0. On the left of the perpendicular line, weights are trained under the LBFS policy and on the right they are trained under the DP policy (i.e. by the weights themselves). The average cost is always obtained by using the DP policy in a short 1000-transitions’ simulation, and then averaging out the results of 100 simulations, as seen in Figure 4-7. After the cost seems not to improve further, we run a longer simulation, which produces a cost equal to 27.10, a very satisfactory result.

4.7 Conclusions

Observing the results, we can see that the Approximate Iterative (Optimistic) Policy Iteration gives satisfactory results. By satisfactory we mean that it gives better results than the heuristics. Nevertheless, one should keep in mind that:

- The values obtained are very close to the ones that Ordinal Optimization produced. They are also not so much better than the heuristics, for the use of such a complicated method and troublesome to be justified.
Figure 4-7: Average cost vs. number of iterations

- A lot more time is needed for the method to converge to such results, compared to the heuristics or even Ordinal Optimization.

- There are too many parameters that have to be chosen arbitrarily before starting the runs, that we cannot really say that the values mentioned above are the best one could get.

It is also very important to note here that Ordinal Optimization produced equally good results, with much less effort. Generally, the methods used are not fully understood and have been only partially exploited in this thesis. A more in-depth analysis might reveal sets of initial parameters that could yield more important results or find a way to accelerate the convergence of the policy iteration.
Bibliography


