The University of Texas at Austin
Internship report

Department of Mechanical Engineering
Operations Research and Industrial Engineering

Optimal scheduling for a multiclass queue with state dependent arrivals

Author:
H.C. Raaijmakers
idno. 0781195
DCno. DC 2017.032

Supervisor University of Texas at Austin:
Dr. J.J. Hasenbein
Supervisor Eindhoven University of Technology:
Prof. Dr. Ir. I.J.B.F. Adan

February 8, 2017

Acknowledgements

This report is the result of a three month internship at the University of Texas at Austin. Three months is relatively short, yet I have found some interesting conclusions and I am satisfied with the result. This project is in Operations Research and with a background in primarily Mechanical Engineering, this was sometimes difficult for me. Though, it was a great learning experience and there were always plenty of people around willing to help me.

Firstly, I would like to thank Dr. Hasenbein who supervised my research and was always available for questions and willing to help. Also, thanks to some of his students for helping me out and to The University of Texas at Austin in general, for making it possible for me to come to the United States. And, thanks to Prof. Dr. Ir. Adan for setting up this connection and the overseas support.

At last, I want to thank all the cool people I met during those three months, especially my roommates from the Macro House who made me feel at home straight away. It was my first time in the US and I did not think it was possible to do so many awesome things in just three months. Austin is a great city, and I am definitely planning on coming back once.

Table of Contents

1 Introduction

In this report, a special type of queueing network is studied. Namely, one in which the arrival rates depend on the type of product that is currently in service. This results in a matrix of arrival rates, instead of a vector. In the system, two queues will form and a decision maker has to decide which queue it is going to serve first. The goal of this report is to find an optimal allocation policy for the decision maker so that the average total cost rate over the infinite horizon is minimized.

For the ordinary case of this queueing network, in which the arrival rates are not dependent on the product in service, the optimal scheduling policy has already been determined. This was accomplished back in 1958 by W.E. Smith and is known as the $c\mu$ rule, or Smith's rule (Smith [8]). The $c\mu$ rule indicates that jobs should be prioritized according to the the value of the holding cost times the processing rate, i.e., $c \cdot \mu$, where larger values of this quantity receive higher priority. In this report it will be investigated whether this rule also applies to a queueing network with state dependent arrivals.

At first, the queueing network under study will be described in detail and it will be modeled as a Markov Decision Problem (MDP). The MDP model is solved using linear programming, but because of some numerical issues another solving method, modified policy iteration, is applied also. Subsequently, both methods and their results are compared and verified using a fluid model approximation and a simulation. Using the verified results, a simple rule of thumb that can be used in designing systems with state dependent arrivals is presented. At last, the study is summarized and suggestions for further research are proposed.

2 Model description

The system that is examined has two job classes $s = \{1, 2\}$ that are processed on a single flexible server with exponentially distributed service times μ_1 and μ_2. The jobs arrive according to a Poisson process, of which the arrival rates of the job classes depend on which job class is currently in service. For two job classes, this results in a 2x2 arrival rate matrix Λ. λ_{ij} is the arrival rate of type i when type j is in service. For jobs that are in a queue, holding costs of c_s per unit time for a job of class s are incurred. The queueing network is depicted in Figure 2.1.

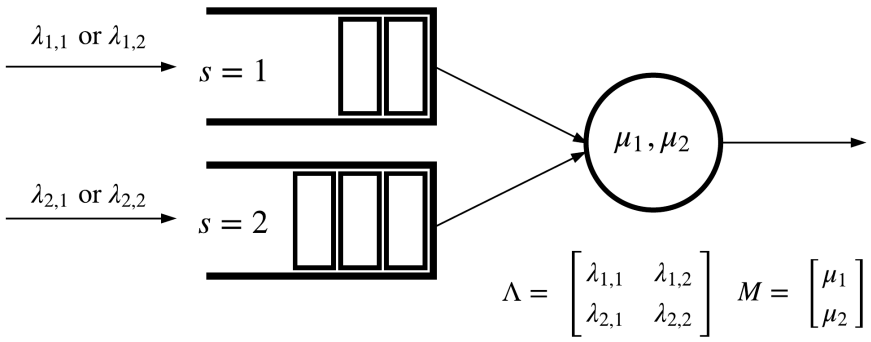


Figure 2.1: *Queueing network under study.*

In order for the system to be stable, the traffic intensity or utilization should be less than one. For the special case considered, this can be computed with the help of the matrix R that can be computed with

$$R_{ij} = \frac{\lambda_{ij}}{\mu_j}. \tag{2.1}$$

The spectral radius of R is denoted as $\rho(R)$, and for stability the following condition must be met: $\rho(R) \leq 1$ (Asmussen et al. [2]). Throughout this report it is assumed that the parameters are chosen in a way that results in a stable system. The traffic intensity of the system will be denoted by ρ.

Furthermore, it is assumed that the server is non-idling, meaning that there is always a product in service. Even when both queues are empty there is virtually a product 'in service'. However, the situation in which the decision maker can choose to service a certain product in order to decrease the arrival rate of one of the products, while in reality there are no products in the queue, should be prevented. Therefore the extra variables λ_{10} and λ_{20} are introduced as the arrival rates when both queues are empty. The obvious choice is to set these parameters equal to the maximum arrival rate of their corresponding product types, hence:

$$\begin{aligned} \lambda_{10} &= \max\{\lambda_{11}, \lambda_{12}\} \\ \lambda_{20} &= \max\{\lambda_{21}, \lambda_{22}\}. \end{aligned} \tag{2.2}$$

If a job is finished, the decision maker decides which job class is served next. Preemptive scheduling is allowed, which means that when a job with a higher priority than the job currently in service arrives, it gets selected immediately. All jobs have a higher priority than virtual jobs, so if a job arrives in an empty system it immediately enters service. Because all service times are exponential, it does not matter whether it is assumed that an interrupted job is continued later or simply starts over.

3 Markov Decision Process approach

The decisions maker's problem can be modeled via a Continuous Time Markov Decision Process (CTMDP). This is because the transition probabilities depend on the type of product in service. The fact that the Markov chain changes when the decision maker decides to service the other job type makes it a Markov Decision Process.

The reason for modeling the problem as a MDP is that with this method it is possible, under certain conditions, to find an optimal policy. A policy is a set of decision rules for each state, for each epoch of time. The solution of the MDP model is thus extremely complete and provides the decision maker with all information required.

In the continuous problem, the queue lengths at time t are depicted by $Q_s(t)$, with $s = \{1, 2\}$ corresponding to the type of products in the queue. $Q_1(t)$ and $Q_2(t)$ are non-negative and the state space consists of all combinations of the two. The decision maker's control action is denoted by $U(t) \in \{u^1, u^2\}$ corresponding to the allocation of a job to the server, of queue 1 or 2 respectively. Given an initial state (Q_0^1, Q_0^2) the decision maker uses a policy π that indicates the sequence of actions when there is a change of state. If the queue lengths depend on a particular kind of policy, this is denoted by $Q_s^\pi(t)$.

Since it is assumed that there is always a job in service, the initial state is at a moment another job has just finished. It does not matter which product this is because the past actions have no influence on the future actions in a Markov process.

Recall that the only costs are the holding costs per lot per unit of time c_1 and c_2 for job types 1 and 2. The objective is to minimize the total average cost rate over an infinite horizon. This is described by

$$\limsup_{t\to\infty} \frac{1}{t} \mathbb{E}_{(Q_0^1, Q_0^2)} \left[\int_0^t (c_1 Q_1^\pi(s) + c_2 Q_2^\pi(s)) ds \right] \tag{3.1}$$

for the initial state (Q_0^1, Q_0^2) (Sisbot and Hasenbein [7]). The discounted cost case involving the discount factor β is not of interest for the system considered here.

3.1 Uniformization

In a CTMDP the time interval between state transitions is exponentially distributed and differs per transition. To be able to analyze the problem using MDP theory and computational methods, the problem has to be converted to a Discrete Time Markov Decision Process (DTMDP), in which the transition times are discrete and constant. This can be achieved using a simple procedure called uniformization (Bertsekas [4]).

The state and control at any time t are denoted by $x(t)$ and $u(t)$ and will remain constant between state transitions. The state and control after k transitions will be denoted by x_k and u_k respectively. Correspondingly, the queue lengths will be denoted Q_k^s, with $s = \{1, 2\}$.

The system will thus be described in terms of states and state transitions. The state transitions will be described by probabilities. If the system is in state i and control u is applied, the next state will be j with probability $p_{ij}(u)$, according to:

$$p_{ij}(u) = P(x_{k+1} = j | x_k = i, u_k = u) \qquad i, j \in S,\ u \in U(i). \tag{3.2}$$

When a transition occurs in the continuous case, it means there has been an arrival in, or service completion of, either product class 1 or 2. The arrival can always be of both classes,

a service completion can naturally only occur if a product of that type is in service. The total state space is thus a combination of all possible combinations of Q^1 and Q^2, or queue lengths and is infinite if the queue buffers are unbounded.

When for example both queues are bounded at a maximum of holding $N = 4$ products, the queue lengths can be: $Q_k^s = \{0, 1, 2, 3, 4\}$ with $s = \{1, 2\}$. The number of states n is equal to the total amount of combinations and is thus $(N + 1)^2 = 25$. The matrix P that holds all transition probabilities has dimensions 25x25 in this case. See Figure 3.1 for a graphical representation of the state space for this example. Because of this structure, the size of the state space increases very rapidly as N increases and the effort that is needed to solve the MDP therefore also increases very rapidly. This effect is common for MDPs and it is known as the curse of dimensionality.

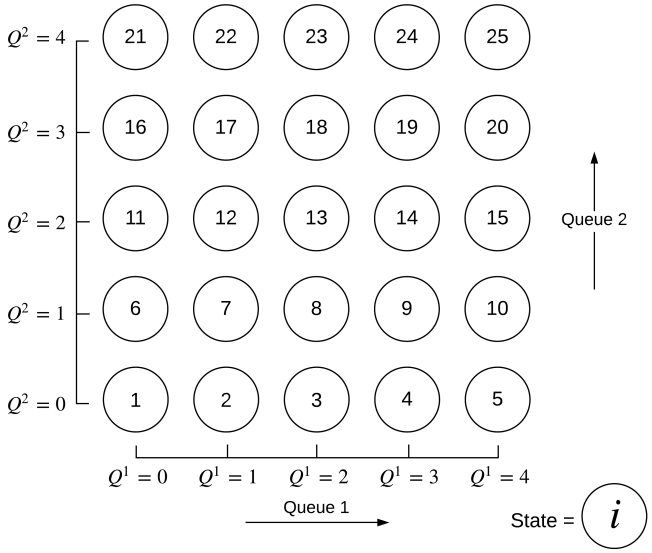


Figure 3.1: *State space with states i consisting of a combination of Q^1 and Q^2 for $N = 4$ and $n = 25$.*

The time interval between transitions is exponentially distributed with parameter $\nu_i(u)$. Using the system's properties described in Section 2, the transition rates can be determined. Their components are shown in Figure 3.2. The transition rate for each state is the sum of all outgoing arrows. Moreover, the transition probability of going from state i to j under a certain control is the transition rate component from i to j, divided by the total transition rate out of state i, for that specific control.

It is assumed that the state and control stay constant in between transitions. If all transition rates would be equal, the decision maker's problem would be identical to a discrete time MDP in which the transition times are fixed. This is because the length of the time interval between transitions does not matter, since both the control and state and thus the costs are constant in between transitions. In this case, only the average costs per state have to be scaled to compensate for the effect of randomness.

Nonetheless, in the system under study the transition rates are not constant. To convert these non-uniform transition rates to uniform transition rates a new uniform transition rate ν is introduced with $\nu_i(u) \leq \nu$ for all i and u. For the system considered, ν is therefore set

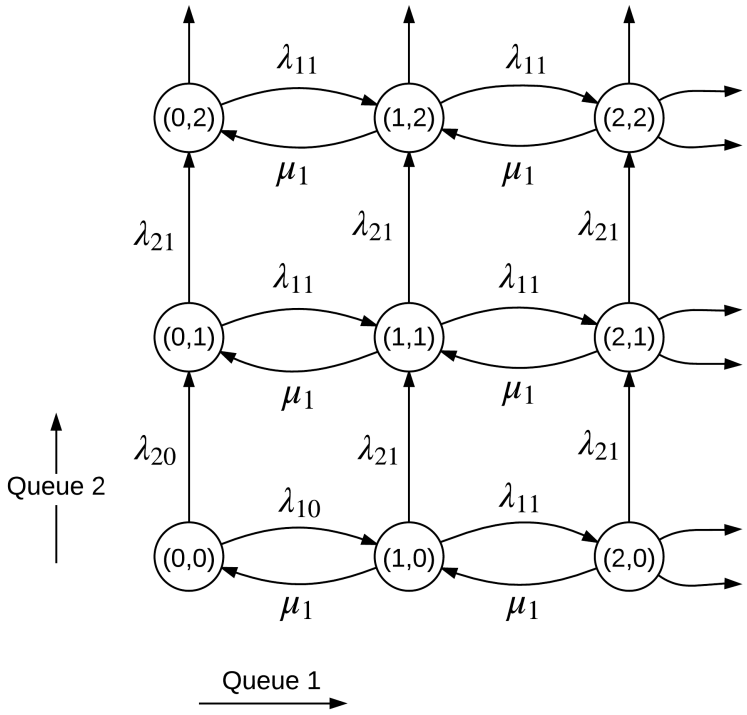


Figure 3.2: *Components of transition rates for control $u = u^1$.*

to

$$\nu = \max\{\lambda_{11} + \lambda_{21} + \mu_1, \lambda_{12} + \lambda_{22} + \mu_2\} \tag{3.3}$$

and the transition probabilities change according to

$$\tilde{p}_{ij}(u) = \begin{cases} \frac{\nu_i(u)}{\nu} p_{ij} & i \neq j \\ \frac{\nu_i(u)}{\nu} p_{ii} + 1 - \frac{\nu_i(u)}{\nu} & i = j. \end{cases} \tag{3.4}$$

This conversion basically creates the possibility of allowing fictitious transitions from a state to itself. Leaving state i at rate $\nu_i(u)$ in the original process is statistically identical to leaving state i at the faster rate ν, but with returning back to i with probability $1 - \nu_i(u)/v$.

Identical to the situation in the CTMDP, in the DTMDP the transition probabilities depend on the control that is applied. A graphical representation of the Markov chain which represents the system if control $u = u^1$ is chosen, and thus a product of type 1 is in service is shown in Figure 3.3. In this example the maximal transition rate $\nu = \lambda_{11} + \lambda_{21} + \mu_1$ and the arrival rates comply to $\lambda_{10} = \lambda_{11}$ and $\lambda_{20} = \lambda_{21}$. Because of this latter constraint, parts of the numerator and denominator cancel each other out resulting in the short expressions of Figure 3.3. It is used in this example to make the figure more readable.

At last the cost function needs to be scaled using the scaling factor $1/\nu$ so that it represents the cost rate per unit time. This results in the DTMDP cost function:

$$\tilde{g}(i) = \frac{1}{\nu}(c_1 Q^1 + c_2 Q^2). \tag{3.5}$$

The number of products Q^1 and Q^2 is only dependent on the state i, since no costs are connected to choosing a certain control. The specific relation between the state i and queue

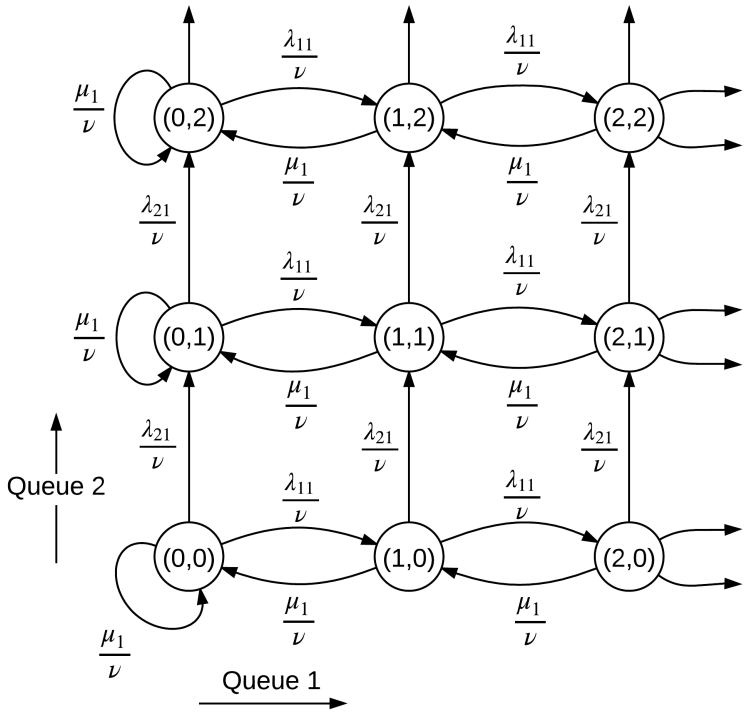


Figure 3.3: *Transition probabilities for control* $u = u^1$, $\nu = \lambda_{11} + \lambda_{21} + \mu_1$, $\lambda_{10} = \lambda_{11}$ *and* $\lambda_{20} = \lambda_{21}$.

lengths Q^1 and Q^2 depends on the size of the state space. This is due to the grid structure of the state space, which is depicted in Figure 3.1. In Section 3.4 an example of $\tilde{g}(i)$ will be given. For the remainder of this report $\tilde{p}$ and $\tilde{g}$ will be denoted by p and g.

3.2 Discrete Time Markov Decision Process model setup

Instead of at every time instant t the system can now be described at state x_k after k state transitions. The time index $[0, \infty)$ is now replaced by a countably infinite number of state transitions.

Using (3.5) and the new transition probabilities a new discrete objective function can be formulated. Each time the system is in state i and control u is applied, the expected cost is $g(i, u)$ and the system moves to state j with probability $p_{ij}(u)$.

The objective is to minimize the average cost rate over all policies $\pi = \{\phi_0, \phi_1, \ldots\}$ with $\phi_k(i) \in U(i)$ for all i and k, starting from an initial state x_0:

$$J_\pi(x_0) = \limsup_{N\to\infty} \frac{1}{N} \mathbb{E}\left[\sum_{k=0}^{N-1} \frac{1}{\nu}(c_1 Q(1)_k^\pi + c_2 Q(2)_k^\pi)\right], \tag{3.6}$$

in which the sub- and superscripts π indicate the dependence on a certain policy. $Q(1)_k^\pi$ and $Q_k^\pi(2)$ denote the queue lengths for job types 1 and 2, after transition k under policy π. Note that the initial state $x_0 = (Q_0^1, Q_0^2)$.

For most systems the optimal policy will be stationary, i.e., $\pi = \{\phi, \phi, \ldots\}$, meaning that the same function is used every time. For a stationary policy ϕ, the average cost of starting at

x_0 and applying ϕ is denoted by $J_\phi(x_0)$. The stage cost, probability and average costs under policy ϕ are denoted by

$$g_\phi = \begin{bmatrix} g(1,\phi(1)) \\ \vdots \\ g(n,\phi(n)) \end{bmatrix}, \quad P_\phi = \begin{bmatrix} p_{11}(\phi(1)) & \cdots & p_{1n}(\phi(1)) \\ & \vdots & \\ p_{n1}(\phi(n)) & \cdots & p_{nn}(\phi(n)) \end{bmatrix}, \quad J_\phi = \begin{bmatrix} J_\phi(1) \\ \vdots \\ J_\phi(n) \end{bmatrix}. \tag{3.7}$$

3.3 Conditions for equal average cost per stage

The objective is to find the policy that minimizes the average cost rate. It would be convenient if this optimal average cost would be the same for all initial states and that it is stationary. This is the case if the so-called Weak Accessibility (WA) condition holds for the system (Bertsekas [4]).

State i is accessible from state j if there exists a stationary policy ϕ and an integer k such that $P(x_k = j \,|\, x_0 = i, \phi) > 0$. The WA condition holds if the set of states can be partitioned into two subsets S_t and S_c such that all states in S_t are transient under every stationary policy and that for every two states i and j in S_c, j is accessible from i.

If for the system considered, the entire state space S is considered to be S_c the WA condition holds. This is true for every stationary policy, as long as control u^1 is applied when $Q^2 = 0$ and similarly u^2 is applied when $Q^1 = 0$. In this way state $(Q^1, Q^2) = (0, 0)$ can always be reached and because of the exponential arrival and processing times, all other states can also always be reached from $(0, 0)$. Thus the WA condition holds and it can be concluded that the average cost is the same for all initial states and there exists an optimal stationary policy that is unichain.

An unichain policy is a special type of policy for which the corresponding Markov chain has a single recurrent class. If the WA condition holds, it is always possible to convert a stationary policy into one that is unichain without affecting the average cost of any one chosen class of recurrent states. This is helpful since it will make it easier to find an optimal solution using numerical methods.

3.4 Computational solving methods

The MDP described has been solved by using linear programming (LP) and policy iteration (PI). Both are well-known computational methods. Initially, only linear programming was used to generate results. However, due to some inaccurate results as a consequence of numerical errors policy iteration was also applied as a means of verification.

To be able to use the numerical methods the state space has to be truncated. Nonetheless, if this number is chosen sufficiently high, useful results can be generated. Except for some boundary effects along the truncation border, the numerical results should be valid. The number of states after the truncation is denoted by n.

The input for both methods is the same and consists of the cost function $g(i)$ and the probability matrices $p_{ij}(u)$ for $u = \{u^1, u^2\}$. The way in which these parameters are computed will be described first. The problem setup is created using a user defined MATLAB function. In this way, it can be called by the scripts of the different solving methods. The script can be found in Appendix B.1.

The output of both methods is also the same and consists of two parts: the optimal average

costs per stage Ω and the optimal stationary policy ϕ, with Ω as a single value and ϕ as a list of length n with the controls u^1 or u^2 that should be applied at each state. At the end of each script, the optimal policy is converted back to the grid structure, so one can easily see which control should be applied for each combination of queue lengths (Q^1, Q^2).

All programs were run on a laptop with an Intel Pentium i7 2.2 GHz processor, with 8 GB RAM memory.

3.4.1 Probability matrices

As presented in Figure 3.1, the state i is a combination of the queue lengths Q^1 and Q^2. Because of this, the shape of the grid and thus the physical meaning of a state changes if the number of products N changes. Also, because the state space has to be truncated arrivals are lost when the queue is 'full' and the transition rates of the states on the border of the square grid have to be adjusted accordingly. In the program, all conversions happen automatically if the number of products N is adjusted.

An example of the possible outgoing transitions and corresponding transition rate components under a certain control for the state space of Figure 3.1 is depicted in Table 3.1. The transition rate components in Table 3.1 are equal to those that are visible in Figure 3.2. All states that are on the right and top border of the grid (States $5, 10, 15, 20, 21, 22, 23, 24$ and 25) have deviating transition rate components. This is a consequence of the truncation since there can be no λ_1 and λ_2 arrivals for the right and top border, respectively.

Table 3.1: *Transition rate components of possible outgoing transitions under controls $u = \{u^1, u^2\}$ for a state space truncated at $N = 4$ and $n = 25$.*

i	u^1	u^2
1	$\lambda_{10}, \lambda_{20}$	$\lambda_{10}, \lambda_{20}$
$2, 3, 4$	$\lambda_{11}, \lambda_{21}, \mu_1$	$\lambda_{12}, \lambda_{22}$
5	λ_{21}, μ_1	λ_{22}
$6, 11, 16$	$\lambda_{11}, \lambda_{21}$	$\lambda_{12}, \lambda_{22}, \mu_2$
$10, 15, 20$	λ_{21}, μ_1	λ_{22}, μ_2
21	λ_{11}	λ_{12}, μ_2
$22, 23, 24$	λ_{11}, μ_1	λ_{12}, μ_2
25	μ_1	μ_2
$7, 8, 9, 12, 13, 14, 17, 18, 19$	$\lambda_{11}, \lambda_{21}, \mu_1$	$\lambda_{12}, \lambda_{22}, \mu_2$

The total transition rate $v_i(u)$ for state i is the sum of the transition rate of all possible outgoing transitions. Recall that the probability of going from state i to j under control u is the transition rate component from state i to j under control u, divided by the total transition rate $v_i(u)$ out of state i. This results in two nxn probability matrices $P(u)$ for $u = \{u^1, u^2\}$. The non-uniform transition probabilities are converted to uniform ones using (3.4) of the uniformization procedure.

3.4.2 Cost function

The value of the uniform cost function $g(i)$ depends on the size of the state space again, for the same reason as in the section above. Its value at each state i is simply the sum of $c_1 Q_i^1$

and $c_2 Q_i^2$. The cost function is independent of the control action taken since there are no costs connected to applying a certain control. In the MATLAB script, the cost function is dependent on the control u and is therefore a two column matrix. Is was constructed this way in order to make it possible to potentially add a cost for control. For the case considered, the costs for both controls (the two columns) are simply equal. In Table 3.2 the non-uniform cost function for an example with $N = 4$, $c_1 = 0.5$ and $c_2 = 1$ is shown. This is computed according to (3.5), only without the scale conversion of $1/\nu$. Note that $g(i)$ is a column vector.

Table 3.2: *Cost function $g(i)$ for a state space truncated at $N = 4$ and $n = 25$, with $c_1 = 0.5$ and $c_2 = 1$.*

i	$g(i)$	Q_i^1	Q_i^2
1	0	0	0
2	0.5	1	0
3	1	2	0
4	1.5	3	0
5	2	4	0
6	1	0	1
7	1.5	1	1
8	2	2	1
⋮	⋮	⋮	⋮
25	6	4	4

4 Linear programming

One way to computationally find the optimal policy for a MDP is by solving a linear programming (LP) problem. This is very useful, since LP solvers are widely available. This means that only two steps have to be completed. The problem has to be set up so that it is suitable for the LP formulation and subsequently it has to be imported into an existing solver.

Nonetheless, this does not guarantee functional results. In this section the LP setup is described and its performance using multiple solvers is analyzed.

4.1 Problem formulation

The LP formulation that can be used to solve the problem setup of Section 3.4 is:

$$\begin{aligned}
\underset{y(i,u)}{\text{minimize}} \quad & \sum_{i=1}^{n} \sum_{u \in U(i)} y(i,u) g(i,u) \\
\text{subject to} \quad & \sum_{u \in U(i)} y(j,u) - \sum_{i=1}^{n} \sum_{u \in U(i)} y(i,u) p_{ij}(u) = 0 \quad j = 1, \ldots, n \\
& \sum_{i=1}^{n} \sum_{u \in U(i)} y(i,u) = 1 \\
& y(i,u) \geq 0 \qquad\qquad i = 1, \ldots, n, \;\; u \in U(i),
\end{aligned} \tag{4.1}$$

in which $y(i,u)$ is the long run fraction of time that the system is in state i and control u is chosen (Bello and Riano [3]). These are thus basically steady-state probabilities and they are independent of the initial state for the same reasons that the long run average costs are independent of the initial state, as described in Section 3.3. Note that $g(i,u)$ reduces to $g(i)$ for the system under study.

Solving the LP yields several results. Firstly, there is the value of the objective function that is minimized. This is the optimal average cost per stage Ω and it applies to each state because the conditions for equal average cost per stage are met.

These conditions also imply that the transition probability matrix of every stationary policy is irreducible, which means that there exists a deterministic decision rule instead of a randomized one that can be used to find the optimal policy. The decision rule $f(i,u)$ can be found using

$$f(i,u) = \frac{y(i,u)}{\Pi_i} \qquad i = 1, \ldots, n, \;\; u \in U(i) \tag{4.2}$$

where Π_i is the stationary distribution according to

$$\Pi_i = \sum_{u \in U(i)} y(i,u) \qquad i = 1, \ldots, n. \tag{4.3}$$

Because there exists an optimal (deterministic) solution, for each state i only one of the controls $u \in U(i)$ has a value and the others are zero. This value is thus equal to the stationary distribution and after dividing it by itself in (4.2), it equals 1. So, for every state i there is one control u which has value 1. Therefore the set of $f(i,u)$ for all states forms the optimal stationary policy ϕ, that prescribes either control u^1 or u^2 for each state i.

4.2 Implementations

To be able to be certain about the validity of the numerical results the problem was implemented and solved using several programs and solvers. The problem setup described in Section 3.4 was implemented in the following LP solvers:

- MATLAB Optimization Toolbox;
- CPLEX for MATLAB Toolbox;
- CBC and CLPEX via NEOS implemented using Pyomo (Hart et al. [6] and Dolan [5]).

The MATLAB Optimization Toolbox offers a variety of LP solvers, including a dual-simplex and a simplex method. The CPLEX for MATLAB Toolbox is a part of the IBM ILOG CPLEX Optimization Studio which offers high-performance mathematical programming for various optimization problems. Its integrated MATLAB function adapts the solver choice based on the input parameters. Pyomo is a Python based, open-source optimization software modeling program. It sets up the optimization problem and then uses an external solver to solve it. NEOS is an online server that offers various solvers that Pyomo can use, for LPs these are CPLEX and CBC. Only the CBC solver will be used since the CPLEX solver is already available via the CPLEX for MATLAB Toolbox.

All solvers are implemented in the MATLAB script of Appendix B.2 and can be selected by setting a parameter. If a Pyomo solver is chosen the problem is exported to a data file after the setup and subsequently a command prompt that calls the Python script is opened via MATLAB. When the solver is finished, the results are imported and processed by MATLAB again. The Python script which is called is given in Appendix B.3.

4.3 Issues with numerical results

The implementation as described above for the queueing network under study does not produce flawless results. The defects in the results have multiple forms and causes and their specifics are dependent on the type of solver and program that is selected. In this section, the different types of numerical issues leading to these defects are described.

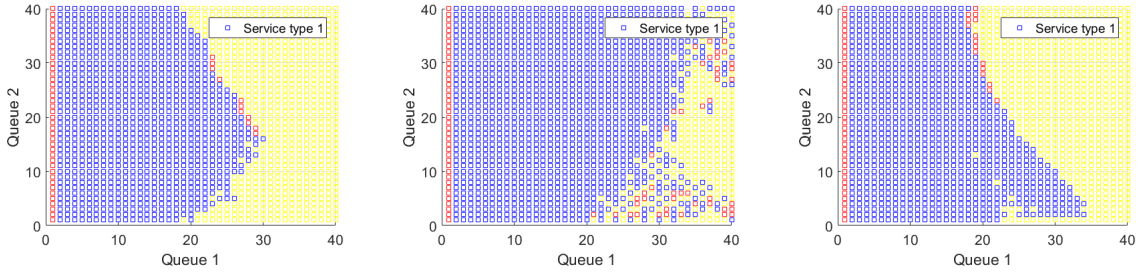
4.3.1 Infinitesimal numbers

In Figure 4.1 the result of a single MDP problem solved by different solvers for the parameters

$$\Lambda = \begin{bmatrix} 16 & 16 \\ 20 & 20 \end{bmatrix}, \quad M = \begin{bmatrix} 33 \\ 41 \end{bmatrix}, \quad C = \begin{bmatrix} 15 \\ 11 \end{bmatrix} \tag{4.4}$$

is shown. The parameters of this problem are set so that the arrivals are independent of the service process. If this is the case, scheduling according to the $c\mu$ rule is optimal, hence for this particular example product 1 should always be always be produced first.

Remarkable in all three figures is the yellow area and the several stray red squares. The program appoints the yellow color if the control value for a state is Not a Number (NaN). This happens to a state i, if $y(i, u) = 0$ for all $U(i)$. In this case, the stationary distribution is zero resulting in a division by zero in (4.2), which causes an error (or NaN). If the Markov chain is irreducible, this is impossible. The probability of being in any state under the stationary distribution can be small, but it is strictly positive.



(a) *MATLAB dual simplex* **(b)** *CPLEX via MATLAB* **(c)** *CBC via Pyomo*

Figure 4.1: *Optimal policy results using three different solvers for a standard problem with non-dependent arrival rates, according to (4.4.)*

The problem is a combination of the constraint tolerances, with the way in which the LP is formulated. Firstly, the values for $y(i,u)$ in certain states become exceedingly small, while they are used in the first constraint of (4.1). This constraint should only be satisfied if it is exactly zero, yet the solver thinks it is satisfied when it is within the constraint tolerances. Because of the fact that certain states have such very small values, their contribution to the constraint remains within the constraint tolerances and is thus practically nonexistent.

Moreover, the non-negativity constraint can also be violated to within the margins of the constraint tolerance. This results in some negative outcomes for $y(i,u)$, which are obviously incorrect. These erroneous values are again used in the constraints which results in more erroneous outcomes for other states.

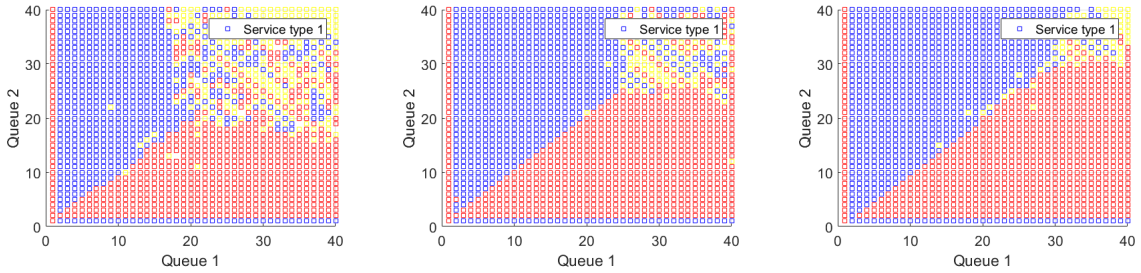
The constraint tolerances range from 1e-7 to 1e-12 for the different solvers, yet apparently the values for $y(i,u)$ are even smaller in certain states. There are several reasons for this, which will be explained with the use of an example. Suppose all problem parameters are equal, i.e., $\lambda_{11} = \lambda_{12} = \lambda_{21} = \lambda_{22}$, $\mu_1 = \mu_2$ and $c_1 = c_2$, then the system is somewhat similar to an M/M/1 queue. As mentioned in the beginning of this chapter, the sums of $y(i,u)$ for each state form the stationary distribution of the system. The stationary distribution of a (non-truncated) M/M/1 queue is $\Pi_i = (1-\rho)\rho^i$, with ρ as the utilization (Adan [1]).

Imagine a system where there is no truncation. Being in state (40,40) is equivalent to reaching state 80 in the M/M/1 queue. However, in this case, state $i = 80$ is divided over all 80 combinations of queue lengths that sum up to 80. For a utilization or traffic intensity of 0.9, a simple calculation of the stationary distribution, divided by 80 yields an average probability of 2.7e-7. This is already fairly small, yet it is only the average. The greatest problem is that it is far more likely that the system is in states close to the axis like (1,79), or (78,2) than in states in the middle like (40,40). The combination of all middle states will be referred to as the equal-queues-diagonal. The diagonals perpendicular to this diagonal, that cover all states that sum up to a single amount will be referred to as summed-queue-diagonals. These summed-queue-diagonals have peaks on their outer boundaries (close to the axes), rather than that they are uniform. As a consequence, the values for the stationary distribution in the center of these diagonals becomes overly small leading to untruthful results.

In Figure 4.2 this effect is illustrated. It displays the results of an implementation of the parameters

$$\Lambda_{ij} = \begin{cases} 35 \text{ if } \rho = 0.7, \ \forall\, i,j \\ 40 \text{ if } \rho = 0.8, \ \forall\, i,j \\ 45 \text{ if } \rho = 0.9, \ \forall\, i,j \end{cases} \qquad M = \begin{bmatrix} 100 \\ 100 \end{bmatrix}, \quad C = \begin{bmatrix} 10 \\ 10 \end{bmatrix}, \tag{4.5}$$

which correspond to the example described above, for three different traffic intensities. It is clearly visible that the erroneous yellow states originate along the equal-queues-diagonal. Furthermore, it is evident that the number of erroneous states is fewer if the traffic intensity of the system increases. A sound result, since the probability of reaching a high state is smaller for a lower traffic intensity resulting in overly small stationary distribution probabilities at an earlier stage.



(a) $\rho = 0.7$ **(b)** $\rho = 0.8$ **(c)** $\rho = 0.9$

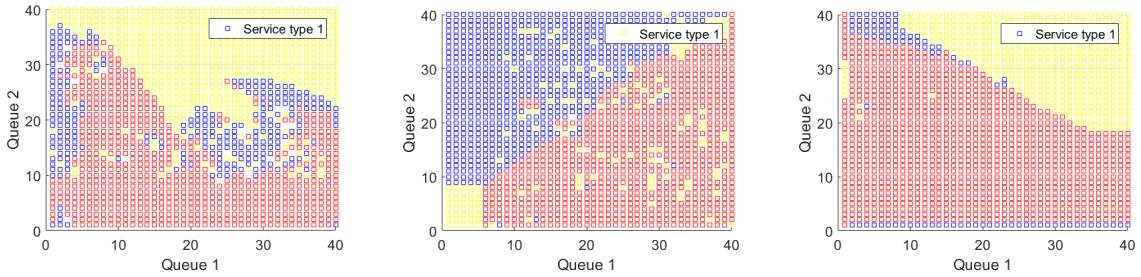
Figure 4.2: *Optimal policy results for a problem with symmetrical parameters according to (4.5) for three different traffic intensities using the CPLEX for MATLAB solver.*

Besides the traffic intensity, the decision maker's policy also has a major effect on the stationary distribution. In Figure 4.2 the distribution is reasonably symmetrical because the parameters are too. However, because the decision maker has to pick either one of the two products on the equal-queues-diagonal, the distribution is shifted to one of the axes. A graphical representation of the stationary distribution and some more elaboration on the example can be found in Appendix A. One can imagine that if the decision maker always chooses the same product, its effect on the stationary distribution becomes quite substantial. This explains the difference in the number of erroneous yellow squares between Figures 4.1 and 4.2. Because the decision maker always chooses product 1, reaching states with a high number of type 2 products is unlikely resulting in a large amount of erroneous squares on the right side of the queue grid.

4.3.2 Parameter choice

The performance of programs that use Pyomo depends on the system parameters that are selected. This is remarkable since the parameter choice should only have an effect on the result itself, not on the quality of the result. Nonetheless, for the solvers called via Pyomo this is the case. In Figure 4.3 the results of the Pyomo implementation for three slightly different parameter sets are depicted. Figure 4.3c gives the correct results, except for the erroneous upper part which is discussed above. Figures 4.3a and 4.3b on the other hand display a completely incorrect result.

Figures 4.3a and 4.3b are extreme cases where the results are completely useless, however it does prove the fact that certain solvers do not work for parameter sets that are 'complex'. Examples of complex parameter choices are sets in which for example the arrival rates are identical or state dependent. It thus also includes sets of which the optimal policy cannot be determined in a straightforward manner by using the $c\mu$ rule. Since complex cases like these are exactly the focus of this study, not all LP solvers are fit for use.



(a) $\Lambda = \begin{bmatrix} 20 & 20 \\ 10 & 20 \end{bmatrix}$ **(b)** $\Lambda = \begin{bmatrix} 20 & 20 \\ 20 & 20 \end{bmatrix}$ **(c)** $\Lambda = \begin{bmatrix} 20 & 20 \\ 19 & 19 \end{bmatrix}$

Figure 4.3: *Optimal policy results for a problem with* $\mu_1 = \mu_2 = 41$, $c_1 = 10$ *and* $c_2 = 12$ *and varying* Λ *for the CBC solver via Pyomo.*

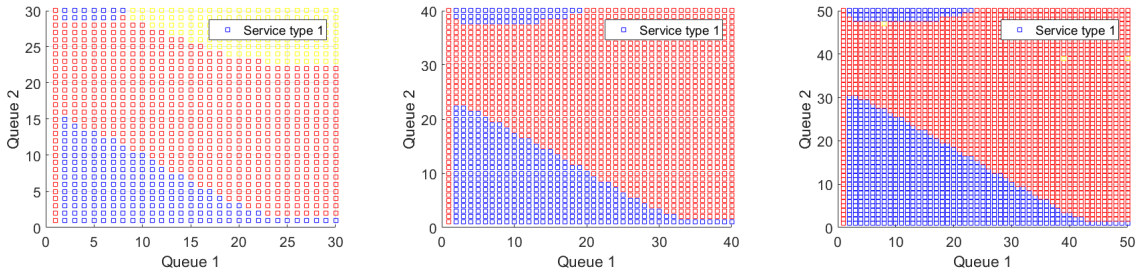
4.3.3 Boundary effects

In Section 3.4.1 the effect of the truncation on the probability matrices is displayed. Obviously, this also has an effect on the solution of the MDP. Because of these boundary effects, the results along the truncation border should be considered unreliable. Normally this is not a problem since with a sufficiently large state space, the boundary states are only a small fraction of the total and the results of the first couple of rows along the borders should simply be neglected.

However, in the MDP model of the queueing system that is examined the boundary effects appear to be extraordinary in size. In Figure 4.4 three MDP solutions of the queueing system with dependent arrival rates for different state space sizes with input parameters

$$\Lambda = \begin{bmatrix} 20 & 20 \\ 10 & 20 \end{bmatrix}, \quad M = \begin{bmatrix} 41 \\ 41 \end{bmatrix}, \quad C = \begin{bmatrix} 10 \\ 12 \end{bmatrix}, \tag{4.6}$$

are displayed. According to the $c\mu$ rule, product two should always be produced first which would result in a figure that is completely red except for the bottom row.



(a) $N = 29$ **(b)** $N = 39$ **(c)** $N = 49$

Figure 4.4: *Optimal policy results for a problem with the parameter set of (4.6) and varying state space size for the CPLEX for MATLAB solver.*

Nonetheless, the figures indicate that this is not the case. When inspecting just one of the three figures, one would expect that the optimal policy follows a certain threshold. However, when looking at all three figures it is clear that the triangle marking the threshold changes size as the state space changes side. The slope of the threshold is identical for every figure and its endpoint is repeatedly at exactly 7 lots away from the truncation border on the Queue

1 axis. Based on this observation, it is likely to assume that boundary effects play a role here. On the other hand, this would mean that the boundary effects cover the larger part of the state space which is rather unusual. In Sections 5 and 6 this is discussed further.

4.4 Evaluation of performance

Using (CPLEX) linear programming to solve MDPs is really fast; solving the $N = 39$ system of Figure 4.4c takes less than 4 seconds on the computer specified in Section 3.4. The method is also relatively easy to apply because of the availability of existing solvers, yet for the problem under study it is not flawless. The solver that shows the best overall performance is the CPLEX implementation for MATLAB. It is very fast and because it selects the solver algorithm that is best suitable for the problem's structure and size, it works reasonably well for all parameter choices and grid sizes. For this reason, the CPLEX for MATLAB solver is used for all results obtained via the LP method for the remainder of this report.

Nevertheless, using the LP method for the queueing system that is examined still leads to a number of problems. The errors due to the infinitesimal numbers can be reduced by pushing the traffic intensity to a value close to 1 and by truncating the state space at a lower number. Unfortunately, these measures have no discernible effect on the issues that are presumably caused by the boundary effects.

Overall, there are too many issues with the LP results to consider them inherently trustworthy. Therefore, alternative methods have to be used to verify the LP method's results.

5 Policy iteration

An efficient method to find the optimal policy of a MDP is using the policy iteration (PI) algorithm. This algorithm generates a stationary policy at every iteration, which always improves the objective function with respect to the previous policy. It is generally well-known for converging to an optimal policy very fast. Before stating the policy iteration algorithm itself, the principles on which it functions are explained (Bertsekas [4]).

5.1 Optimality equations

The objective function of (3.6) minimizes the total average cost rate, resulting in an optimal average cost per stage, starting from a certain initial state. Because of the conditions mentioned in Section 3.3 the average cost per stage is a common scalar, which is optimal, the same for all stages and independent of the initial state. It will be denoted by Ω and is defined as:

$$\Omega = \min_{\pi} J_{\pi}(i), \qquad i = 1, \ldots, n. \tag{5.1}$$

The optimal average cost can be found using a value iteration algorithm, that chooses the best control over all controls based on the one step costs and all expected future costs. This step is repeated multiple times. Under certain conditions, there exists a certain optimal value that is a fixed point for the algorithm, which means that repeating the iteration step will yield the same optimal value function. This is the case for the system considered and therefore

$$\Omega + h(i) = \min_{u \in U(i)} \left[g(i) + \sum_{j=1}^{n} p_{ij}(u) h(j) \right] \qquad i = 1, \ldots, n \tag{5.2}$$

can be written. This optimality equation is known as Bellman's equation. The scalar $h(i)$ is the minimum, over all policies, of the expected cost to reach state n from i for the first time and the cost that would be incurred if the cost per stage were equal to the average Ω. It can be interpreted as the relative value function for each state.

Bellman's equation states that $\Omega + h(i)$ remains the same if for all states i the control u that minimizes the expression right of the equals sign is applied. The set of all controls for each state that does this for step k, is ϕ_k and part of the policy $\pi = \{\phi_0, \phi_1, \ldots\}$.

Since for the system under study there exists a unichain and stationary optimal policy, ϕ_k will be the same at every step k, resulting in a single policy ϕ. The optimal average cost corresponding to this policy is denoted by Ω_{ϕ}. The same control is now applied to each state on every iteration step and Bellman's equation reduces to

$$\Omega_{\phi} + h(i) = g(i) + \sum_{j=1}^{n} p_{ij}(\phi(i)) h(j) \qquad i = 1, ..., n. \tag{5.3}$$

5.2 Policy iteration algorithm

The single-chain policy iteration algorithm is used, because every stationary policy encountered in the course of the algorithm is unichain. The algorithm has three steps, of which steps 2 and 3 are repeated until the optimal policy is found. As mentioned before, this usually happens rather fast and only takes a handful of iterations. A policy iteration algorithm written in MATLAB is attached in Appendix B.4.

Step 1: Initialization

An initial stationary policy ϕ^0 has to be guessed. The optimal policy for the regular case of the problem studied is the $c\mu$ rule and thus this policy chosen is for ϕ^0. Observe that in the states in which either one of the queues is empty, the non-empty product is always served. Otherwise, the Weak Accessibility condition is no longer met.

Step 2: Policy evaluation

For iteration step k, given the stationary policy ϕ^k, the corresponding average and differential costs Ω^k and $h^k(i)$ satisfying the system of equations

$$\Omega^k + h^k(i) = g(i) + \sum_{j=1}^{n} p_{ij}(\phi^k(i))h^k(j) \qquad i = 1, ..., n, \tag{5.4}$$

have to be computed. This is a system of n linear equations with $n+1$ unknowns, namely $\Omega_\phi, h(1), \ldots, h(n)$, which has an infinite number of solutions. However, if a single degree of freedom is fixed, the system has a unique solution. Therefore, a single component of h has to be set to an arbitrary value, which can be, for example, zero. In the scripts provided, state

$$h^k(1) = 0 \tag{5.5}$$

is taken as the reference. The solution of this system can also be obtained iteratively. This method is described in Section 5.3.

Step 3: Policy improvement

The next step is to find a new, improved policy ϕ^{k+1} by applying the right-hand side of

$$g(i) + \sum_{j=1}^{n} p_{ij}(\phi^{k+1}(i))h^k(j) = \min_{u \in U(i)} \left[g(i) + \sum_{j=1}^{n} p_{ij}(u)(j) \right] \qquad i = 1, \ldots, n. \tag{5.6}$$

If $\phi^{k+1} = \phi^k$, the algorithm terminates; otherwise, it returns to Step 2 with ϕ^{k+1} replacing ϕ^k. If all generated policies are unichain and the above procedure is applied, the policy iteration algorithm will terminate in a finite number of iterations and will produce an optimal stationary policy.

5.3 Modified policy iteration

An alternative way of completing the policy evaluation step is by using another method called relative value iteration (RVI), for the policy evaluation step. When the number of states is large, this method is often preferred because solving a system of equations of the size of the entire state space can be severely time-consuming.

Instead of solving the entire system for Ω and h, the value iteration algorithm is used. By simply applying the current policy ϕ_k of policy iteration step k, for a number of iterations l, the solution will converge to the h vector corresponding to the current policy. After every value iteration l, the h vector can be found using

$$h^{l+1} = T_\phi h^l - (T_\phi h^l(1))e, \tag{5.7}$$

in which e is a vector of ones of the same size as h and T_ϕ is the result of the right-hand term of (5.2) and is defined as

$$(T_{\phi^k} h)(i) = g(i) + \sum_{j=1}^{n} p_{ij}(\phi^k(i))h(j) \qquad i = 1, \ldots, n. \tag{5.8}$$

In this method, again $h(1)$ is chosen as the reference state.

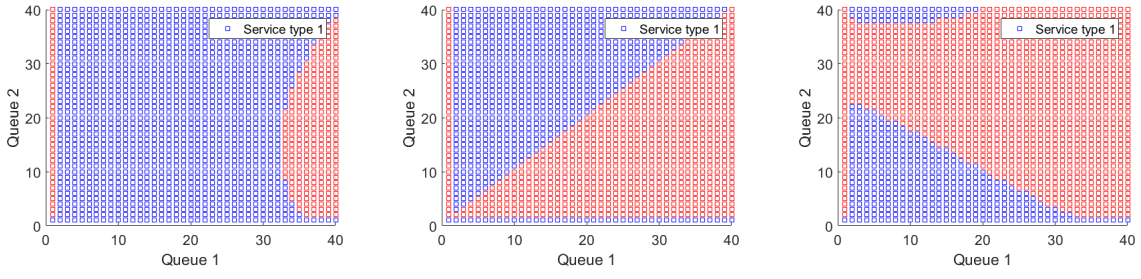
As $l \to \infty$, the relative value function h^l converges to h^k corresponding to ϕ^k for policy iteration step k. However, only a finite number of relative value iterations is required for the policy iteration algorithm to be able to function, so the PI algorithm can already continue to Step 3 before $h^l = h^k$. The number of iterations required depends on the size of the state space.

There are two parameters that have to be set as stopping criteria for the relative value iteration algorithm: the maximum number of RVI iterations, and the RVI tolerance level which stops the algorithm if the difference between the previous and current iteration is less than the tolerance. The two stopping criteria are always both active and they should be tuned in accordance with the size of the state space. The number of RVI iterations required increases as the state space increases. As the state space decreases, the RVI tolerance has to become smaller too.

The PI algorithm in which RVI, as described above, is applied is called modified policy iteration (MPI). In Appendix B.5 a MATLAB script that uses the MPI algorithm is given.

5.4 Implementation of algorithms

The results of three optimal policies obtained via the MPI algorithm are depicted in Figure 5.1. The input parameters used to generate Figures 5.1a, 5.1b and 5.1c are equal to those used to generate Figures 4.1(a,b,c), 4.2a and 4.4b, respectively. Since both the LP and MPI method are supposed to find the optimal policy, each MPI figure should be identical to its counterpart obtained via the LP method.



(a) *Normal case: (4.4)* **(b)** *Symmetrical: (4.5),* $\rho = 0.7$ **(c)** *Special case: (4.6)*

Figure 5.1: *Optimal policy results using three modified policy iteration for three different systems.*

A clear difference of the MPI figures with the respect to the LP figures is the absence of the erroneous yellow squares. As described in Section 4.3.1, these are caused by overly small numbers. This difficulty is a result of the setup of the optimization problem, wherein the value of the states (i.e., the squares) represent the steady state distribution, which becomes very small. In the MPI algorithm however, the states simply represent the policy corresponding to the iteration step and this problem is thus absent.

Another problem that one encounters while using particular solvers of the LP model is that it becomes unstable. The regular PI algorithm never has this problem. Apart from the boundary effects, it always displays the correct result. Unfortunately, it cannot be used for large state spaces ($N \geq \pm 40$ on the computer specified in Section 3.4) because it requires a lot of computing power. Naturally, it will still work on a high-performance computer, yet is sensible to use the MPI algorithm instead because it is more efficient and therefore significantly faster. The MPI algorithm also always works, as long as the parameters for the relative value iteration are set correctly.

MPI works well, yet solving systems with a large maximal queue length can take rather long. Solving the $N = 39$ system of Figure 5.1c takes approximately 7 minutes. The time it takes to solve systems with even larger maximal queue lengths increases rapidly, as the state space grows polynomially with respect to the maximal queue length. Also, more relative value iterations are required as the difference between Ω_k and Ω_{k+1} for adjacent policies becomes smaller.

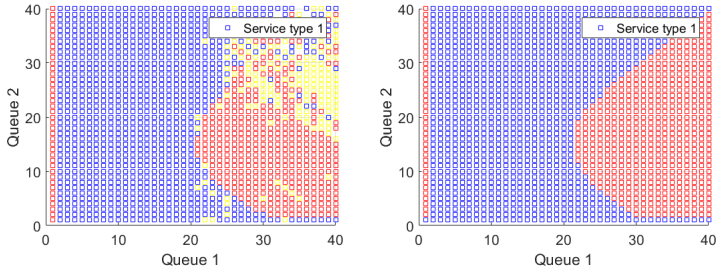
An advantage of regular PI is that unlike MPI, it gives the exact value for Ω. Nonetheless, the objective is to find the optimal policy and (with the correct stopping criteria) MPI does output this correctly. Considering the speed argument, MPI is thus a more suitable algorithm for the system under study.

Nevertheless, the overly strong boundary effects remain as can be seen in Figure 5.1c. This figure is exactly identical to Figure 4.4b of the LP method, which means that the boundary effects are independent of the method of solving the MDP and are thus linked to the problem setup.

6 Results I

The MDP model that is described in Section 3 has been solved using linear programming and modified policy iteration as depicted in Sections 4 and 5, respectively. By using the same input parameters for both methods and comparing the solutions it was verified that both solvers give the same results, as long as the erroneous values are neglected. Moreover, by applying the solvers on a regular queueing system and analytically computing the optimal policy using the $c\mu$ rule, it can also be concluded that the correct answer is found (if again the erroneous values are neglected). However, for certain parameter sets, including the set of the system under study, the boundary effects are abnormally large. The results clearly imply that the $c\mu$ rule is not optimal, yet it cannot be concluded with certainty because of the large amount of erroneous values.

An overview of the conclusions drawn in the paragraph above is provided in Figures 6.1 to 6.4. In all figures, the left figure is the solution obtained via the LP method and the right figure is the MPI solution. It is advisable to study the solution of both models because by comparing the figures it is possible to determine which part is likely to be correct and which part is definitely incorrect. For example, when examining both solutions of Figure 6.1 it can easily be concluded that the right half of the solutions should not be trusted because of the presence of erroneous values in the LP model. This would have been more difficult to determine if only the MPI solution would be available.

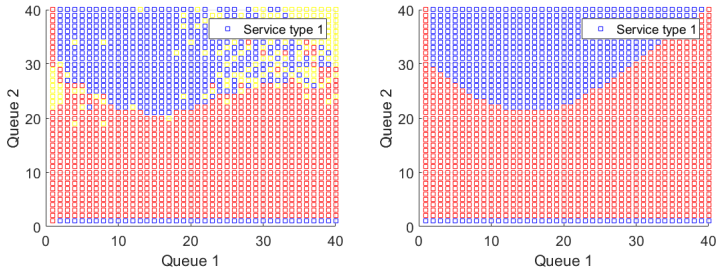


$$\Lambda = \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}, \; C = \begin{bmatrix} 10.1 \\ 10 \end{bmatrix}$$

$$\mu_1 = \mu_2 = 20$$

$$c \cdot \mu = \begin{cases} 202 & \text{for } \mathbf{product\ 1} \\ 200 & \text{for product 2} \end{cases}$$

Figure 6.1: *MDP solutions by LP (left) and MPI (right) for a regular queueing system. According to the cμ rule, product 1 should be produced first.*

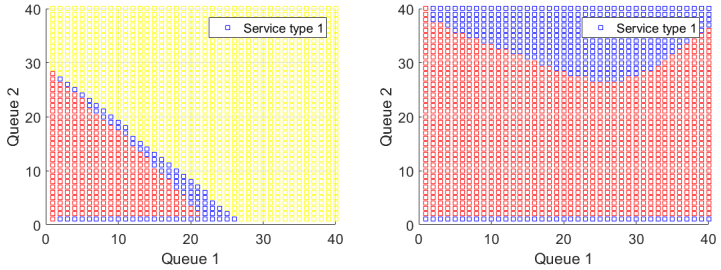


$$\Lambda = \begin{bmatrix} 8 & 8 \\ 8 & 8 \end{bmatrix}, \; C = \begin{bmatrix} 10 \\ 10.1 \end{bmatrix}$$

$$\mu_1 = \mu_2 = 20$$

$$c \cdot \mu = \begin{cases} 200 & \text{for product 1} \\ 202 & \text{for } \mathbf{product\ 2} \end{cases}$$

Figure 6.2: *MDP solutions by LP (left) and MPI (right) for a regular queueing system. According to the cμ rule, product 2 should be produced first.*

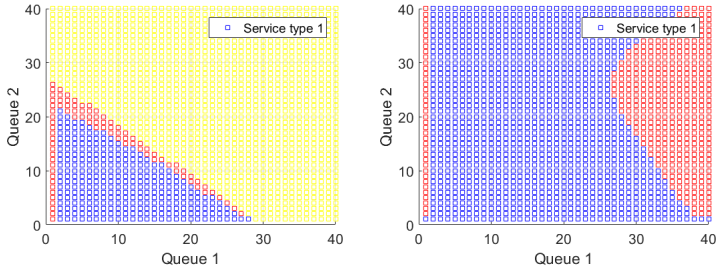


$$\Lambda = \begin{bmatrix} 8 & 6 \\ 4 & 8 \end{bmatrix}, \; C = \begin{bmatrix} 10.1 \\ 10 \end{bmatrix}$$

$$\mu_1 = \mu_2 = 20$$

$$c \cdot \mu = \begin{cases} 202 & \text{for } \mathbf{product\ 1} \\ 200 & \text{for product 2} \end{cases}$$

Figure 6.3: *MDP solutions by LP (left) and MPI (right) for a special queueing system. According to the cμ rule, product 1 should be produced first.*



$$\Lambda = \begin{bmatrix} 8 & 4 \\ 6 & 8 \end{bmatrix}, \; C = \begin{bmatrix} 10 \\ 10.1 \end{bmatrix}$$

$$\mu_1 = \mu_2 = 20$$

$$c \cdot \mu = \begin{cases} 200 & \text{for product 1} \\ 202 & \text{for } \mathbf{product\ 2} \end{cases}$$

Figure 6.4: *MDP solutions by LP (left) and MPI (right) for a special queueing system. According to the cμ rule, product 2 should be produced first.*

In Table 6.1 the computing times that the scripts need to obtain the results are depicted. It is clear that the (CPLEX) LP method is considerably faster than the MPI algorithm. Therefore, when experimenting with parameter sets is advisable to make use of the LP method. Once a useful result is obtained, it should be verified with the MPI algorithm.

Table 6.1: *MDP solution computing times for different solving methods on a system with an Intel Pentium i7 2.2 GHz processor, with 8 GB RAM memory.*

Figure	LP [s]	MPI [s]
6.1	3.8	414.3
6.2	3.6	411.7
6.3	3.1	426.9
6.4	2.8	433.1

Especially the results displayed in Figures 6.3 and 6.4 are interesting. The reason for this is that according to the MDP model, the $c\mu$ rule is not the optimal policy for these cases. This is an interesting and unexpected result since the $c\mu$ rule does not depend on the arrival rates. Because of this, and considering the difficulties with the MDP model it is sensible to perform extra means of verification. In the following two sections, two different approaches to verify the results that have been presented in this section are outlined.

7 Fluid model approach

This section functions as an extra means of verification of the results presented in Section 6. The queueing network under study will be approximated by formulating it as a fluid model. The fluid model is the deterministic equivalent of the queueing network described in Section 2, for which the arrival and service rates are now thus no longer exponential. The queueing network can be viewed as two buckets of fluid, representing the two queues. Both buckets are filled according to Λ and drained according to M. Naturally, only one bucket can be drained at the time and the decision maker should determine which one. Since only stable systems are considered, both buckets will eventually be empty and stay empty under any non-idling policy. The fluid model only provides information until this zero point is reached and it is therefore primarily useful for determining how to process large queue sizes.

7.1 Fluid dynamics

The fluid network will be described using a system of differential equations. For $t \geq 0$, and for job classes $s = \{1, 2\}$, the following quantities are defined: $A_s(t)$ is the number of jobs that have arrived in $[0, t]$, $T_s(t)$ is the cumulative time the server has spent on processing jobs of class s in $[0, t]$ and $D_s(t)$ indicates the number of the server completions of type s jobs in $[0, t]$. The queueing network process is defined by: $\mathbb{X}(t) = \{Q(t), A(t), D(t), U(t)\}$ (Sisbot and Hasenbein [7]). For $s = \{1, 2\}$ and $t \geq 0$, the dynamics of the fluid model are defined by

$$\begin{aligned} Q_s(t) &= Q_s(0) + A_s(t) - D_s(t) \\ A_s(t) &= \begin{bmatrix} \lambda_{s1} & \lambda_{s2} \end{bmatrix} \begin{bmatrix} T_1(t) \\ T_2(t) \end{bmatrix} \\ D_s(t) &= \mu_s T_s(t) \\ T(t) &= T_1(t) + T_2(t) = t. \end{aligned} \tag{7.1}$$

At each point in time the decision maker applies a control action $U(t) \in \{u^1, u^2\}$ corresponding to processing product 1 or 2, respectively. u^1 and u^2 can also be seen as the fraction of time the corresponding control was active. Therefore, $u^1(t) + u^2(t) \leq 1$ and they cannot be negative. The rate of change of the queue lengths can be written in terms of the derivatives:

$$\begin{aligned} \frac{d}{dt} Q_1(t) &= \lambda_{11}\, u^1(t) + \lambda_{12}\, u^2 - \mu_1\, u^1(t) \\ \frac{d}{dt} Q_2(t) &= \lambda_{21}\, u^1(t) + \lambda_{22}\, u^2 - \mu_2\, u^2(t). \end{aligned} \tag{7.2}$$

7.2 Optimization problem

By discretizing the fluid model into (small) intervals, the decision maker's task can be modeled as an optimization problem. The objective should obviously be to minimize the total holding costs and the constraints have to be based on the system dynamics. This results in the

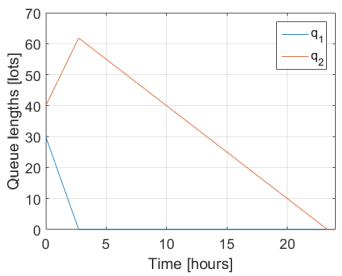
following linear programming problem:

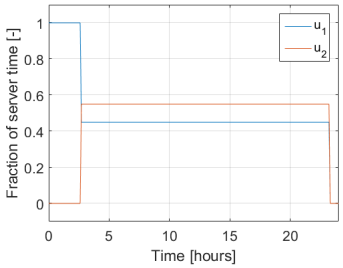
$$
\begin{aligned}
\text{find} \quad & \boldsymbol{d} = [u_0^1, u_1^1, \ldots, u_N^1, u_0^2, u_1^2 \ldots, u_N^2 \\
& \quad Q_1^1, Q_2^1, \ldots, Q_{N-1}^1, Q_1^2, Q_2^2, \ldots, Q_{N-1}^2] \\
\underset{d}{\text{minimize}} \quad & \Delta t \sum_{n=1}^{N-1} c_n^1 Q_n^1 + c_n^2 Q_n^2 \\
\text{subject to} \quad & Q_{n+1}^1 = Q_n^1 + (\lambda_{11} - \mu_1) u_n^1 \Delta t + \lambda_{12} u_n^2 \Delta t \quad n = 0, 1, \ldots, N \\
& Q_{n+1}^2 = Q_n^2 + (\lambda_{22} - \mu_2) u_n^2 \Delta t + \lambda_{21} u_n^1 \Delta t \quad n = 0, 1, \ldots, N \\
& u_n^1 + u_n^2 \leq 1 \quad n = 0, 1, \ldots, N \\
& u_n^1 \geq 0 \quad n = 0, 1, \ldots, N \\
& u_n^2 \geq 0 \quad n = 0, 1, \ldots, N,
\end{aligned}
\tag{7.3}
$$

in which Q_0^1 and Q_0^2 are the initial fluid levels which are non-negative, Q_N^1 and Q_N^2 are the final fluid levels and are equal to zero and c^1 and c^2 represent the holding costs for lots of type 1 and 2. The number of equations that needs to be solved depends on the number of increments N. The increment size depends on the total time T and is defined as $\Delta t = T/N$. T should be chosen sufficiently large, so that the fluid model has enough time to 'drain' and reach zero. Also, N should be sufficiently large to ensure that the intervals are small enough. In this model, λ_0 is not included and therefore the model only works until both queues are empty. The objective function and constraints are linear and the optimization problem can thus be solved using linear programming. In Appendix C, a MATLAB script is provided that sets up and solves the linear programming problem of (7.3).

7.3 Fluid model conclusions

In Figures 7.1 and 7.2 the results of the fluid model approximation are shown for two different parameter sets. Within each figure, the left diagram shows the dynamics of the fluid levels and the right figure shows the division of server time. Figure 7.1 is the fluid model of a regular queueing network without dependent arrival rates. As expected, the products are scheduled according to the $c\mu$ rule. However, in the second figure that represents a system that does have state dependent arrival rates, the optimal policy according to the fluid model is no longer in accordance with the $c\mu$ rule. It is clear that all server capacity is dedicated to product two until this queue is empty. The model thus schedules products according to the reversed $c\mu$ rule for this system.



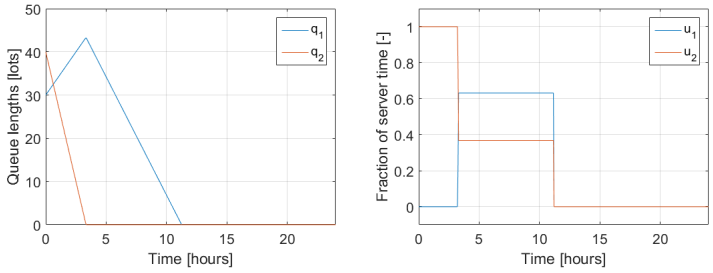


$$
\Lambda = \begin{bmatrix} 9 & 9 \\ 8 & 8 \end{bmatrix}, \; C = \begin{bmatrix} 12 \\ 10 \end{bmatrix}
$$

$$
\mu_1 = \mu_2 = 20
$$

$$
c \cdot \mu = \begin{cases} 240 & \text{for } \textbf{product 1} \\ 220 & \text{for product 2} \end{cases}
$$

Figure 7.1: *Fluid model solution for a regular queueing system. According to the cμ rule, product 1 should be produced first.*



$$\Lambda = \begin{bmatrix} 9 & 4 \\ 7 & 8 \end{bmatrix}, \; C = \begin{bmatrix} 12 \\ 10 \end{bmatrix}$$

$$\mu_1 = \mu_2 = 20$$

$$c \cdot \mu = \begin{cases} 240 & \text{for } \mathbf{product\ 1} \\ 220 & \text{for product 2} \end{cases}$$

Figure 7.2: *Fluid model solution for a special queueing system. According to the cμ rule, product 1 should be produced first.*

Even though the fluid model is only valid for long queues, the results do coincide with those of Section 6 in which was stated that the $c\mu$ rule is not always optimal. By comparing Figures 7.1 and 7.2 an explanation for this unexpected result can be found. The processing rates of both examples are equal, yet in Figure 7.2 the queues are both empty significantly faster. This is because, by clever allocating, the average arrival rate of the products is being reduced. As a consequence, fewer products arrive so the queues are empty faster, reducing the total holding costs. Even so, it should be noted that this also results in fewer products being serviced. Consequently, the reduction in holding costs might be neutralized or even surpassed by the losses in the throughput rate.

8 Verification by simulation

In this section, another means of verification is performed in order to be able to confirm the unexpected results of Section 6. In that section, it is stated that according to the MDP model, the $c\mu$ rule is not optimal for certain parameter sets of the queueing network with state dependent arrivals, as opposed to the regular model where the $c\mu$ rule is always optimal. A straightforward way of verifying that one policy outperforms another policy is by, firstly simulating the queueing network under both policies, and subsequently using a hypothesis test to affirm the results. In this section this procedure will be described.

8.1 Simulation setup

The queueing network was simulated for a sample size of $n_j = 50$ (i.e., 50 independent simulations) for both policies $j \in \{1, 2\}$, with policy 1 and 2 as the $c\mu$ and the reversed $c\mu$ rule, respectively. Each simulation has a duration of 2000 hours divided into time steps of 0.001 hours. This corresponds to a cumulative production of approximately 70000 lots per simulation for parameter set (8.1). To compensate for potential warm up effects, the first 100 hours (or 3500 lots) of the simulation data is not included in the processing. All simulation samples are completely independent and generate their own Poisson arrival process and exponential service times. The simulation was performed using the MATLAB script of Appendix D.1. In the script, two simulations for the two policies run in parallel and thus a sample for both policies is generated per simulation. However, they use random seeds and are therefore completely independent of each other.

The system parameters are

$$\Lambda = \begin{bmatrix} 20 & 20 \\ 10 & 20 \end{bmatrix}, \quad M = \begin{bmatrix} 41 \\ 41 \end{bmatrix}, \quad Q_0 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad C = \begin{bmatrix} 10 \\ 12 \end{bmatrix} \tag{8.1}$$

and are identical for all simulations. Λ, M, Q_0 and C indicate the arrival rates, processing rates, initial queue lengths and holding costs, respectively.

The sample mean $\bar{x}_j$ and sample variance s_j^2 for $j = \{1, 2\}$ for both policies are computed with

$$\bar{x}_j = \frac{1}{n_j} \sum_{i=1}^{n_j} x_i^j \qquad j = 1, 2 \tag{8.2}$$

$$s_j^2 = \frac{1}{n_j - 1} \sum_{i=1}^{n_j} (x_i^j - \bar{x}_j)^2 \qquad j = 1, 2, \tag{8.3}$$

using the average costs per hour of the samples.

The results are depicted in Table 8.1, with $\bar{x}_j$ as the mean of the average costs per hour. In this table, the confidence intervals are computed with Microsoft Excel using the Student's t-distribution based on a confidence level of 99%. Since the intervals are disjoint, the result implies that the reversed $c\mu$ policy outperforms the $c\mu$ policy. In Section 8.2 this result confirmed by a hypothesis test. The data set containing the average costs per hour of all samples $x_1^j, x_2^j, \ldots, x_{n_j}^j$ for $j = \{1, 2\}$ is given in Appendix D.2.

Table 8.1: *The mean, variance and 99% confidence intervals of average costs per hour of 50 samples for the $c\mu$ and reversed $c\mu$ policy.*

Policy	j	n_j	$\bar{x}_j$	s_j^2	$\bar{x}_j$ confidence interval
$c\mu$	1	50	79.64	32.40	$[77.49; 81.80]$
reversed $c\mu$	2	50	72.83	21.09	$[71.09; 74.57]$

8.2 Hypothesis test

To make sure that the simulation results are statistically significant and not due to random variation, a hypothesis test was carried out. The problem is one-tailed and the data consists of two samples of independent and identically distributed random variables with unequal variances. Therefore, the Welch's t-test is a suitable method to perform the hypothesis test (Watkins [9]). Because the goal is to find out if one policy is better than the other, it has to be a one-tailed test. The significance level is set to $\alpha = 1\%$. If the p-value that is obtained is smaller than α, the H_0 hypothesis should be rejected. The hypothesis applies to the parameter set of (8.1) and is set as follows:

H_0: The $c\mu$ policy is better than the reversed $c\mu$ policy.
H_1: The reversed $c\mu$ policy is better than the $c\mu$ policy.

Welch's t-test is an adaption of the Student's t-test and it defines the t-statistic according to:

$$t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}. \tag{8.4}$$

The effective degrees of freedom ν can be computed using the Welch-Satterhwaite equation, that is defined as:

$$\nu = \frac{\left(\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}\right)^2}{\frac{\left(\frac{s_1^2}{n_1}\right)^2}{n_1 - 1} + \frac{\left(\frac{s_2^2}{n_2}\right)^2}{n_2 - 1}}. \tag{8.5}$$

The outcome of (8.5) is not an integer, therefore the final p-value has to be obtained by interpolating the p-values of the adjacent integers. ν and t were computed using the data of Table 8.1 and implemented in the right-tailed Student's t-distribution of Microsoft Excel. This resulted in a p-value of $p = 1.28 \cdot 10^{-9}$. Since $p \ll 0.01$, the H_0 hypothesis is rejected and it is concluded that based on the simulation results the reversed $c\mu$ rule performs better than the $c\mu$ rule for the parameter set of (8.1). Accordingly, the $c\mu$ rule is not the optimal policy for this parameter set.

9 Results II

In Section 6 it is stated that the $c\mu$ rule might not always be optimal. With the help of Sections 7 and 8 this conjecture was established. The MDP and fluid models suggest that in some cases the reversed $c\mu$ policy is optimal instead, yet no decisive argument can be provided to validate this.

Nonetheless, the models can be used to determine when an alternative policy should be considered. After analyzing a large number of solutions produced with the MDP models discussed in this report, a very general rule of thumb could be defined as follows:

- If the ratios between state dependent arrival rates are asymmetric, i.e., $\frac{\lambda_{11}}{\lambda_{22}} \neq \frac{\lambda_{12}}{\lambda_{21}}$;
- And the ratio of $c \cdot \mu$ between products is larger than or approximately 0.75, i.e., $\frac{\min\{c_1\mu_1, c_2\mu_2\}}{\max\{c_1\mu_1, c_2\mu_2\}} \gtrapprox 0.75$;
- Then it is likely that the $c\mu$ is not the optimal policy and one of the MDP models should be used to verify this.

If one would have to design a decision maker for a queueing network with state dependent arrival rates, this rule of thumb could be consulted to examine whether an alternative to the $c\mu$ rule should be considered.

10 Conclusions and recommendations

10.1 Conclusions

In this report, the optimal scheduling of a special type of queueing network is discussed. The queueing network has been modeled as a Markov Decision Process. The major advantage of this method is that because the system fulfills certain requirements, the solution of the MDP model is an optimal policy. Moreover, the optimal policy is stationary, meaning that the optimal routing action only depends on the state of the system and is independent of time.

The MDP model has been solved using the linear programming method and using the modified policy iteration algorithm. The LP method is significantly faster, but the MPI algorithm has fewer errors in its result and it is therefore more reliable. It is remarkable that in both models the boundary effects are abnormally large. An explanation for this effect has not been found.

Nonetheless, it has been verified using both a fluid model approximation and a simulation that the MDP models do produce correct results, apart from some erroneous values. Because of the presence of errors, it is strongly advised to use both methods when solving a MDP. In this way, by comparing the two figures it is possible to identify which parts of the solution are incorrect and should be neglected.

With the help of the models, an exception on the $c\mu$ rule has been found. In a queueing system with state dependent arrivals, for certain parameter sets, the reversed $c\mu$ rule performs better than then the actual $c\mu$ rule. However, because of the numerical difficulties with the MDP models it cannot be concluded with certainty that this is also the optimal policy. To assist in the scheduling of this sort of special queueing networks a simple rule of thumb has been composed, which is presented in Section 9. It is based on system parameters and indicates whether it is likely or not that the $c\mu$ rule is not the optimal policy. The rule can be used to determine if further examination using the MDP models is necessary.

At last, it should be noted that all conclusions mentioned above are based on the situation in which the only costs involved are holding costs. By making clever use of the state dependent arrivals, the arrival rate of one of the products is suppressed for a part of the time. Therefore the total amount of products that is eventually produced is reduced, which negatively affects the total revenue. The benefits in terms of holding costs might not compensate for the losses in revenue. Based solely on this research, no conclusions can be drawn on the net benefit.

10.2 Recommendations for further research

A supplement to this report would be an explanation for the extraordinary size of the boundary effects, which are caused by the truncation of the state space. If they could be prevented, it would greatly improve the credibility and usability of the MDP models. In particular, the MPI algorithm's outcome would be improved, since the boundary effect is the only issue that is encountered while using this method.

If a completely correct MDP solution for a state dependent system were to be found, the uncertainty about the actual optimal policy will also be solved. This report suggests that for certain parameter sets, the reversed $c\mu$ rule is optimal, yet it cannot be validated. With a completely correct MDP solution, this can be validated and an alternative optimal policy can be established. Another option would be to find the optimal policy analytically. The numerical models could in this case serve as a means of verification for the results acquired

analytically.

Moreover, the scope of this research could be extended by including a reward for producing goods. As was mentioned in the conclusion, this study only focuses on the holding costs. By including a reward for the amount of produced goods, it could be established whether the reduction in holding costs compensates for the loss in revenue due to a decrease in production quantity.

References

[1] Adan, I.J.B.F. (2003). *Queuing Theory class, The M/M/1 system* [Course outline document]. Retrieved September 2016 from `http://www.win.tue.nl/~iadan/que/h4.pdf`.

[2] Asmussen, S., Ernst, P. and Hasenbein, J.J. (2016). *Stability and Tail Asymptotics in a Multiclass Queue with State Dependent Arrival Rates.* `arXiv:1609.03999v1 [math.PR]`

[3] Bello B., and Riano G., (n.d.) *Linear Programming solvers for Markov Decision Processes.* Retrieved September 2016 from `http://www.sys.virginia.edu/sieds06/papers/FMorningSession5.1.pdf`

[4] Bertsekas, D.P. (2005). *Dynamic Programming and Optimal Control* (Third ed., Vol. 2). Belmont, MA: Athena Scientific.

[5] Dolan, E. 2001. *The NEOS Server 4.0 Administrative Guide.* Technical Memorandum ANL/MCS-TM-250, Mathematics and Computer Science Division, Argonne National Laboratory.

[6] Hart, W. E., Laird, C., Watson J.P., and Woodruff, D.L.(2012). *Pyomo-Optimization Modeling in Python* (Vol. 67.) Springer Science & Business Media.

[7] Sisbot, E.A. and Hasenbein J.J. (2016). *Joint Routing and Scheduling Control in a Two-class Network with a Flexible Server* (Manuscript draft). Preprint, received August 2016.

[8] Smith, W.E. (1956). *Various optimizers for single-stage production*, Naval Research Logistics Quarterly 3 (pp. 59 - 66).

[9] Watkins, J.C.(n.d.). *An Introduction to the Science of Statistics: From Theory to Implementation* (Preliminary Edition). Retrieved October 2016 from `http://math.arizona.edu/~jwatkins/statbook.pdf`

A Stationary distribution example problem

This appendix focuses on the stationary distribution of a system with equal parameters. To be specific, the system from (4.5) with a traffic intensity of $\rho = 0.9$ is examined. In Figure A.1, the stationary distribution that was computed with (4.2) of this system is shown. The higher values are, the darker their shade of green is. It is clear, that as depicted in Section 4.3.1 the values of the distribution along the summed-queue-diagonals increase as the states approach the axes. Moreover, it is visible that the distribution is not completely symmetrical. For this particular example, the decision maker services product 1 along the equal-queues-diagonal (see Figure 4.2c) and therefore the distribution on the side of the Queue 2 axis is higher.

7	0.01487	0.00749	0.00368	0.00175	0.00079	0.00034	0.00014	0.00005
6	0.01699	0.00857	0.00417	0.00194	0.00085	0.00034	0.00013	0.00008
5	0.01967	0.00992	0.00475	0.00212	0.00087	0.00032	0.00019	0.00019
4	0.02323	0.01168	0.00541	0.00227	0.00085	0.00049	0.00048	0.00046
3	0.02841	0.01406	0.00611	0.00230	0.00130	0.00123	0.00113	0.00102
2	0.03710	0.01739	0.00664	0.00359	0.00319	0.00280	0.00246	0.00217
1	0.05485	0.02157	0.01063	0.00852	0.00700	0.00591	0.00508	0.00442
0	0.10121	0.03624	0.02331	0.01735	0.01396	0.01171	0.01006	0.00877
	0	1	2	3	4	5	6	7

Figure A.1: *Stationary distribution of the parameter set of (4.5) with* $\rho = 0.9$.

In Table A.1 the stationary distribution that is generated by the model is compared to one of an M/M/1 queue. The model's distribution is the sum of all values of the summed-queue-diagonal of the corresponding n. The analytical distribution is computed with

$$\Pi_n = (1 - \rho)\rho^n. \tag{A.1}$$

From Table A.1 can be read that the two distributions are nearly identical as expected.

Table A.1: *Comparison of stationary distributions obtained analytically and numerically by the MDP model.*

n	Analytical Π	MDP LP result Π
0	0.1	0.101209013
1	0.09	0.091088112
2	0.081	0.081979301
3	0.0729	0.073781371
4	0.06561	0.066403234
5	0.059049	0.05976291
6	0.0531441	0.053786619
7	0.04782969	0.048407957
8	0.043046721	0.043567162

B MDP related scripts

B.1 MATLAB script: MDP Problem setup

```
% MDP_problem_setup.m | Han Raaijmakers | Oct 2016
% Function that uses import parameters to compute transition probabilities,
% and perform uniformization.

% Inputs:
% N = Maximal queue length
% Lambda = Arrival rates matrix
% Lambda0 = Arrival rates when server is off vector
% Mu = Service rates vector
% C = Holding costs vector

% Outputs:
% p_d_u1 = probability matrix for control u = 1
% p_d_u2 = probability matrix for control u = 2
% g_d = stage cost matrix g

function [p_d_u1,p_d_u2,g_d,g] = MDP_problem_setup(N,Lambda0,Lambda,Mu,C)
    %% CREATE MODEL STRUCTURE

    Qcomb = (N+1)^2; % Number of states (queue length combinations)
    pos_a = (N+1)*N; % n.o. possible arrivals

    % DEFINE TRANSTION RATES

    % Define (continous) transition times
    v_u1 = zeros(Qcomb,1); % Column to store transition rates
    v_u2 = zeros(Qcomb,1); % Column to store transition rates

    % control: u = 1
    v_u1(:,1) = Lambda(1,1)+Lambda(2,1)+Mu(1);
    lb = 1;   % left border Q1 = 0;
    rb = N+1; % right border Q1 = max;
    tb = 1;   % top border Q2 = 0;
    bb = Qcomb-N; % bottom border Q2 = max;

    for i = 1:N+1
        v_u1(lb) = v_u1(lb)-Mu(1);
        lb = lb+N+1;
        v_u1(rb) = v_u1(rb)-Lambda(1,1);
        rb = rb+N+1;
        v_u1(bb) = v_u1(bb)-Lambda(2,1);
        bb = bb+1;
    end

    % control: u = 2
    v_u2(:,1) = Lambda(1,2)+Lambda(2,2)+Mu(2);
    lb = 1;   % left border Q1 = 0;
    rb = N+1; % right border Q1 = max;
    tb = 1;   % top border Q2 = 0;
    bb = Qcomb-N; % bottom border Q2 = max;

    for i = 1:N+1
        v_u2(tb) = v_u2(tb)-Mu(2);
        tb = tb+1;
        v_u2(rb) = v_u2(rb)-Lambda(1,2);
        rb = rb+N+1;
        v_u2(bb) = v_u2(bb)-Lambda(2,2);
        bb = bb+1;
    end

    % Correct for state (0,0)
    v_u1(1) = Lambda0(1) + Lambda0(2);
    v_u2(1) = Lambda0(1) + Lambda0(2);

```

```
65      % Define the largest possible transition time
66      v = max(max(v_u1),max(v_u2));
67
68      % DEFINE CTMDP PROBABILITIES
69      t_c_u1 = zeros(Qcomb,Qcomb); % Empty transition matrix u1
70      t_c_u2 = zeros(Qcomb,Qcomb); % Empty transition matrix u2
71      p_c_u1 = zeros(Qcomb,Qcomb); % Empty probability matrix u1
72      p_c_u2 = zeros(Qcomb,Qcomb); % Empty probability matrix u2
73
74      % Loop for Lambda1 arrivals
75      j = 2; %Counter
76      for i = 1:pos_a+(N+1)
77          if mod(i,N+1) == 0
78             j = j+1;
79             continue
80          end
81          t_c_u1(i,j) = Lambda(1,1);
82          t_c_u2(i,j) = Lambda(1,2);
83          j = j+1;
84      end
85
86      % Loop for Lambda2 arrivals
87      j = N+2; %Counter
88      for i = 1:pos_a
89          t_c_u1(i,j) = Lambda(2,1);
90          t_c_u2(i,j) = Lambda(2,2);
91          j = j+1;
92      end
93
94      % Correct for state (0,0)
95      t_c_u1(1,N+2) = Lambda0(2);
96      t_c_u1(1,2) = Lambda0(1);
97      t_c_u2(1,N+2) = Lambda0(2);
98      t_c_u2(1,2) = Lambda0(1);
99
100     % Loop for u1 service completion
101     j = 2; %Counter
102     for i = 1:pos_a+(N+1)
103         if mod(i,N+1) == 0
104            j = j+1;
105            continue
106         end
107         t_c_u1(j,i) = Mu(1);
108         j = j+1;
109     end
110
111     % Loop for u2 service completion
112     j = N+2; %Counter
113     for i = 1:pos_a
114         t_c_u2(j,i) = Mu(2);
115         j = j+1;
116     end
117
118     % Convert arrival/competion matrices to probability matrices.
119     for i = 1:Qcomb
120        p_c_u1(i,:) = t_c_u1(i,:)./v_u1(i);
121        p_c_u2(i,:) = t_c_u2(i,:)./v_u2(i);
122     end
123
124     % CTDMP COST FUNCTION
125
126     g = zeros(Qcomb,2); % Matrix to store costs
127     % Queue 1 costs
128     j = 1; % Counter
129     for i = 1:N+1
130         for k = 1:N
131             g(j+k,1) = C(1)*k;
132         end
133         j = j+N+1;
134     end
```

```
135
136     % Queue 2 costs
137     j = 1; % Counter
138     for i = 1:N+1
139         t = 1;
140         for k = N+1:N+1:pos_a
141             g(j+k,1) = g(j+k,1) + C(2)*t; % Include Q1 costs too
142             t = t+1;
143         end
144         j = j+1;
145     end
146     g(:,2) = g(:,1); % Costs are independent of control action
147
148     % APPLY UNIFORIMZATION TO CREATE DTMDP
149
150     % Cost function
151     g_d = g./v;
152
153     % Transition probabilities
154     p_d_u1 = zeros(size(p_c_u1));
155     p_d_u2 = zeros(size(p_c_u2));
156     for i = 1:Qcomb
157         for j = 1:Qcomb
158             if i == j
159                 p_d_u1(i,j) = (v_u1(i)/v) * p_c_u1(i,i) + 1 - (v_u1(i)/v);
160                 p_d_u2(i,j) = (v_u2(i)/v) * p_c_u2(i,i) + 1 - (v_u2(i)/v);
161             else
162                 p_d_u1(i,j) = (v_u1(i)/v) * p_c_u1(i,j);
163                 p_d_u2(i,j) = (v_u2(i)/v) * p_c_u2(i,j);
164             end
165         end
166     end
167
168 end
```

B.2 MATLAB script: MDP solver by Linear Programming

```
1  %% MDP_lp_solver.m | Han Raaijmakers | Oct 2016
2  % Sets up the problem as a Discrete time Markov Decision Process and uses
3  % Linear Programming to solve it. The LP solver can be called from the
4  % MATLAB  Optimzation Toolbox, from the IBM CPLEX Optimization toolbox,
5  % or from Pyomo.
6  % In the second case the problem is set up, processed in Python and
7  % imported again. This .m file and the pyomo_mdp_solver.py file must be
8  % in the same directory and this must be in the PATH for Python.
9
10 %clc;
11 clear all; close all;
12 disp('Running the MDP LP solver, with idling.');
13 fprintf('\n')
14
15 %% PARAMETERS
16 disp('Setting up model...');
17 fprintf('\n')
18
19 % Program parameters
20 % Solver choice:
21 % 1 = MATLAB simplex
22 % 2 = MATLAB dual-simplex
23 % 3 = IBM CPLEX via Matlab(warning signs are because of options definition)
24 % 4 = CBC via Pyomo
25 % 5 = IBM CPLEX via Pyomo
26 solver = 3;
27
28 % Maximal buffer size (state space truncation)
29 N = 39;
30
31 % System parameters
32 % Dependent arrival rates [lots/hour]
```

```
33 Lambda = [20   20;
34           10   20];
35
36 % Arrival rates when server is off
37 Lambda0(1) = max(Lambda(1,1),Lambda(1,2));
38 Lambda0(2) = max(Lambda(2,1),Lambda(2,2));
39
40 % Processing rates [lots/hour]
41 Mu(1) = 42;
42 Mu(2) = 42;
43
44 % Holding costs [dollars/lot/hour]
45 C(1) = 10;
46 C(2) = 11;
47
48 % Stability check
49 m = inv(diag(Mu));
50 M = Lambda*m;
51 EIG = abs(eig(M));
52 if max(EIG) >= 1
53     msg = ['This choice of parameters does not guarantee stability,'...
54         ' please choose different parameters.'];
55     error(msg)
56 end
57
58 % Print theoretical results
59 max_EIG = max(EIG);
60 disp(['The traffic intensity, or spectral density of the system is: ' ...
61       num2str(max_EIG)])
62 fprintf('\n')
63
64 muC_ratio = [C(1)*Mu(1);C(2)*Mu(2)];
65 Order = [1;2];
66 Order = [Order muC_ratio];
67 TO = flipud(sortrows(Order,2));
68 disp(['First product according to c*mu rule is:  ' num2str(TO(1,1)) ...
69      '     with c*mu = ' num2str(TO(1,2))])
70 disp(['Second product according to c*mu rule is: ' num2str(TO(2,1)) ...
71      '     with c*mu = ' num2str(TO(2,2)) ])
72 fprintf('\n')
73
74 ratio = TO(2,2) / TO(1,2);
75 disp(['Ratio between c*mu for number 2 and 1 is: ' num2str(ratio) ])
76 fprintf('\n')
77
78 %% CREATE MODEL STRUCTURE
79
80 Qcomb = (N+1)^2; % Number of states (queue length combinations)
81 pos_a = (N+1)*N; % n.o. possible arrivals
82
83 % Call to problem setup function
84 [p_d_u1,p_d_u2,g_d,g] = MDP_problem_setup(N,Lambda0,Lambda,Mu,C);
85
86 %% FORMULATE LINEAR PROGRAM
87
88 % Parameters
89 % g_d =  discrete time cost function
90 % p_d_u1 = discrete time probability matrix u1
91 % p_d_u1 = discrete time probability matrix u2
92
93 % Need to find:
94 % q(i,u): (N+1)^2x2 matrix with the optimal policy to chose for each state
95 % linprog output will be (N+1)^2*2x1 vector
96
97 % OBJECTIVE FUNCTION
98
99 f = [g_d(:,1)' g_d(:,2)'];
100
101 % EQUALITY CONSTRAINTS
102 Aeq = zeros(Qcomb-1,2*Qcomb); % Empty matrix, except for last constraint
```

```
103
104 % Everything on the right of equals sign (probabilities)
105 for i = 1:Qcomb
106    for j = 1:Qcomb
107       Aeq(i,j) = -p_d_u1(j,i);
108       Aeq(i,j+Qcomb) = -p_d_u2(j,i);
109    end
110       Aeq(i,i) = Aeq(i,i)+1;
111       Aeq(i,i+Qcomb) = Aeq(i,i+Qcomb)+1;
112 end
113 beq = zeros(Qcomb,1);
114
115 % Add final equality constraint
116 Aeq_f = zeros(1,Qcomb*2);
117 for i = 1:Qcomb
118     Aeq_f(1,i) = 1;
119     Aeq_f(1,i+Qcomb) = 1;
120 end
121 beq_f = 1;
122
123 Aeq = [Aeq;Aeq_f];
124 beq = [beq;beq_f];
125
126 % INEQUALITY CONSTRAINTS
127 A = [];
128 b = [];
129
130 % UPPER AND LOWER BOUNDS
131 % All larger than zero
132 lb = zeros(1,Qcomb*2);
133 ub = [];
134
135 %% EXECUTE LINEAR PROGRAM
136 t = cputime;
137 disp('Setting up solver started...')
138 fprintf('\n')
139 x0=[];
140
141 if solver <= 3 % MATLAB solver
142     if solver == 1 % simplex
143         options = optimoptions('linprog','Algorithm','simplex', ...
144         'Display', 'Iter','MaxIter',1e6,'MaxTime',3600, ...
145         'TolFun',1e-10,'TolCon',1e-9);
146         solver_name = char('MATLAB simplex');
147         [q,fval,exitflag,output] = linprog(f,A,b,Aeq,beq,lb,ub,x0,options);
148     elseif solver == 2 % dual-simplex
149         options = optimoptions('linprog','Algorithm','dual-simplex', ...
150             'Display', 'Iter','MaxIter',1e6,'MaxTime',3600,'TolFun', ...
151             1e-10,'TolCon',1e-9);
152         solver_name = char('MATLAB dual-simplex');
153         [q,fval,exitflag,output] = linprog(f,A,b,Aeq,beq,lb,ub,x0,options);
154     elseif solver == 3 % cplex
155         options = cplexoptimset('cplex');
156         options.display='iter';
157         solver_name = char('IBM CPLEX via MATLAB');
158         [q,fval,exitflag] = cplexlp(f,A,b,Aeq,beq,lb,ub,x0,options);
159     end
160
161     % Write results to usuable c_frac format
162     c_frac = zeros(Qcomb,2);
163     c_frac(:,1) = q(1:Qcomb);
164     c_frac(:,2) = q(Qcomb+1:2*Qcomb);
165
166 else % Pyomo solver
167
168     % Add empty rows and columns for headers
169     g_d = [zeros(size(g_d,1),1) g_d];
170     g_d = [zeros(1,size(g_d,2)); g_d];
171     p_d_u1 = [zeros(size(p_d_u1,1),1) p_d_u1];
172     p_d_u1 = [zeros(1,size(p_d_u1,2)); p_d_u1];
```

```
    p_d_u2 = [zeros(size(p_d_u2,1),1) p_d_u2];
    p_d_u2 = [zeros(1,size(p_d_u2,2)); p_d_u2];

    % Fill rows with row and column numbers
    g_d(2:end,1) = [1:Qcomb]';
    p_d_u1(2:end,1) = [1:Qcomb]';
    p_d_u1(1,2:end) = [1:Qcomb];
    p_d_u2(2:end,1) = [1:Qcomb]';
    p_d_u2(1,2:end) = [1:Qcomb];

    % Write and export matrices to csv file format
    csvwrite('g.csv',g_d);
    csvwrite('pu1.csv',p_d_u1);
    csvwrite('pu2.csv',p_d_u2);

    disp('Calling command prompt for Pyomo solver...')

    if solver == 4 % CBC
        % Call command prompt
        command = ['pyomo solve --solver-manager=neos --solver=cbc'...
            ' pyomo_mdp_solver.py --summary'];
        [status,cmdout] = system(command);
        disp(cmdout)
        solver_name = char('CBC via Pyomo');
    elseif solver == 5 % CPLEX
        % Call command prompt
        command = ['pyomo solve --solver-manager=neos --solver=cplex'...
            ' pyomo_mdp_solver.py'];
        [status,cmdout] = system(command);
        disp(cmdout)
        solver_name = char('IBM CPLEX via Pyomo');
    end

    % Import and transform data
    q_data = csvread('results.csv');
    c_frac_q1 = zeros(Qcomb,2);
    c_frac_q2 = zeros(Qcomb,2);
    c_frac_q1(:,1) = q_data(1,1:Qcomb);
    c_frac_q1(:,2) = q_data(2,1:Qcomb);
    c_frac_q2(:,1) = q_data(1,Qcomb+1:2*Qcomb);
    c_frac_q2(:,2) = q_data(2,Qcomb+1:2*Qcomb);
    % Sort indexes
    c_frac_q1 = sortrows(c_frac_q1);
    c_frac_q2 = sortrows(c_frac_q2);
    % Convert to workable format
    c_frac = [c_frac_q1(:,2) c_frac_q2(:,2)];

end

Solver_Time = cputime-t;
disp(['Solver process finished in ' num2str(Solver_Time) ' seconds.']);
disp('Setting up results...');
fprintf('\n')

%% ANALYZING RESUTLS

% Stationary Distribution
Y = zeros(Qcomb,1);
for i = 1:Qcomb
    Y(i) = c_frac(i,1)+c_frac(i,2);
end

% Deterministic rule
% Can be used because our MDP is irreducible
pi = zeros(Qcomb,1);
for i = 1:Qcomb
    pi(i,1) = c_frac(i,1)/Y(i);
    pi(i,2) = c_frac(i,2)/Y(i);
end

```

```
% Generate control matrix in queue form
C1 = zeros(N+1,N+1);
C2 = zeros(N+1,N+1);
C1_frac = zeros(N+1,N+1);
C2_frac = zeros(N+1,N+1);
Q_dist = zeros(N+1,N+1);
g_dist = zeros(N+1,N+1);

for i = 1:N+1
   % Optimal control
   C1(i,:) = pi(1+(i-1)*(N+1):i*(N+1),1);
   C2(i,:) = pi(1+(i-1)*(N+1):i*(N+1),2);
   % Fractions
   C1_frac(i,:) = c_frac(1+(i-1)*(N+1):i*(N+1),1);
   C2_frac(i,:) = c_frac(1+(i-1)*(N+1):i*(N+1),2);
   % Deterministic Distribution
   Q_dist(i,:) = Y(1+(i-1)*(N+1):i*(N+1));
   % Holding cost distribution
   g_dist(i,:) = g(1+(i-1)*(N+1):i*(N+1),1);
end

%% PLOTTING RESULTS

% Control routing rule
figure
axis([0 N+1 0 N+1]);
xlabel('Queue 1');
ylabel('Queue 2');
hold on
grid on
% Control 1
for i = 1:N+1
    for j = 1:N+1
        if C1(i,j) >= 0.99
            plot(j,i,'sb')
        elseif isnan(C1(i,j)) % Error value due to truncation
            plot(j,i,'sy')
        end
    end
end
% Control 2
for i = 1:N+1
    for j = 1:N+1
        if C2(i,j) >= 0.99
            plot(j,i,'sr')
        elseif isnan(C2(i,j))
            plot(j,i,'sy')
        end
    end
end
legend('Service type 1')
%title(solver_name)
shg

disp('Program finished.');
```

B.3 Python script: Pyomo model setup

```
# LP solver
# With modified constraints for faster building

# To execute, fun the following line in the command prompt:
# pyomo solve --solver-manager=neos --solver=cbc LPsolve_v4.py

from __future__ import division
from pyomo.environ import *
```

```
9
10 import pandas
11 import csv
12
13 # Import data
14 pu1 = pandas.read_csv('pu1.csv',sep = ';')
15 pu2 = pandas.read_csv('pu2.csv',sep = ';')
16 g = pandas.read_csv('g.csv',sep = ';')
17
18 # Add column headers
19 g.columns = ['state','u1','u2']
20 col_name1 = pu1.columns[0]
21 pu1=pu1.rename(columns = {col_name1:'state'})
22 col_name2 = pu2.columns[0]
23 pu2=pu2.rename(columns = {col_name2:'state'})
24
25 model = AbstractModel()
26
27 model.S = Param(within=NonNegativeIntegers, initialize = len(g.index)) # n.o. states
28
29 # SETS
30 model.I = RangeSet(1, model.S) # States
31
32 # Create special sets for borders and non-borders of grid
33 # Excluding point (0,0), and other corner points
34 # NOTE: different N than in MATLAB (queue length + 1)
35 N = int(sqrt(len(g.index)))
36
37 # Top border
38 tb = range(2,N)
39 # Left border
40 lb = []
41 for i in range(2,N**2-N):
42     if i%N == 1:
43         lb[len(lb):] = [i]
44 # Right border
45 rb = []
46 for i in range(N+1,N**2):
47     if i%N == 0:
48         rb[len(rb):] = [i]
49 # Bottom border
50 bb = range(N**2-N+2,N**2)
51 # Remaining states
52 mm = []
53 for i in range(N+1,N**2-N):
54     if i%N != 0 and i%N != 1:
55         mm[len(mm):] = [i]
56
57 # DECISION VARIABLE
```

```
model.q1 = Var(model.I, domain=NonNegativeReals) # decision variable q1(i)
model.q2 = Var(model.I, domain=NonNegativeReals) # decision variable q2(i)

# Define summation function
def obj_expression(model):
    return sum(
         g[g.state == i].u1.values[0] * model.q1[i]
         + g[g.state == i].u2.values[0] * model.q2[i]
         for i in model.I
    )

# Define objective function
model.OBJ = Objective(rule=obj_expression)

def qsum_constraint_rule(model):
    # return the expression for the constraint for i
    return sum((model.q1[i] + model.q2[i])
        for i in model.I
     ) == 1

model.qsumConstraint = Constraint(rule=qsum_constraint_rule)

# Sum constraint for state (0,0) top left
def p_00_sum_constraint_rule(model):
    return ((model.q1[1] + model.q2[1]) - (
       model.q1[1]*pu1[pu1.state==1][str(1)].values[0] +
       model.q2[1]*pu2[pu2.state==1][str(1)].values[0] +
       model.q1[2]*pu1[pu1.state==2][str(1)].values[0] +
       model.q2[N+1]*pu2[pu2.state==N+1][str(1)].values[0]) == 0)

model.p_00_sumConstraint = Constraint(rule=p_00_sum_constraint_rule)

# Sum contraint for top border states
def p_tb_sum_constraint_rule(model, j):
    return ((model.q1[j] + model.q2[j]) - (
       model.q1[j]*pu1[pu1.state==j][str(j)].values[0] +
       model.q2[j]*pu2[pu2.state==j][str(j)].values[0] +
       model.q1[j-1]*pu1[pu1.state==j-1][str(j)].values[0] +
       model.q2[j-1]*pu2[pu2.state==j-1][str(j)].values[0] +
       model.q1[j+1]*pu1[pu1.state==j+1][str(j)].values[0] +
       model.q2[j+N]*pu2[pu2.state==j+N][str(j)].values[0]) == 0)

model.p_tb_sumConstraint = Constraint(tb, rule=p_tb_sum_constraint_rule)

# Sum constraint for state (N,0) (top right)
def p_N0_sum_constraint_rule(model):
    return ((model.q1[N] + model.q2[N]) - (
       model.q1[N]*pu1[pu1.state==N][str(N)].values[0] +
       model.q2[N]*pu2[pu2.state==N][str(N)].values[0] +
```

```
107         model.q1[N-1]*pu1[pu1.state==N-1][str(N)].values[0] +
108         model.q2[N-1]*pu2[pu2.state==N-1][str(N)].values[0] +
109         model.q2[N+N]*pu2[pu2.state==N+N][str(N)].values[0]) == 0)
110
111 model.p_N0_sumConstraint = Constraint(rule=p_N0_sum_constraint_rule)
112
113 # Sum contraint for left border states
114 def p_lb_sum_constraint_rule(model, j):
115     return ((model.q1[j] + model.q2[j]) - (
116         model.q1[j]*pu1[pu1.state==j][str(j)].values[0] +
117         model.q2[j]*pu2[pu2.state==j][str(j)].values[0] +
118         model.q1[j-N]*pu1[pu1.state==j-N][str(j)].values[0] +
119         model.q2[j-N]*pu2[pu2.state==j-N][str(j)].values[0] +
120         model.q1[j+1]*pu1[pu1.state==j+1][str(j)].values[0] +
121         model.q2[j+N]*pu2[pu2.state==j+N][str(j)].values[0]) == 0)
122
123 model.p_lb_sumConstraint = Constraint(lb, rule=p_lb_sum_constraint_rule)
124
125 # Sum contraint for right border states
126 def p_rb_sum_constraint_rule(model, j):
127     return ((model.q1[j] + model.q2[j]) - (
128         model.q1[j]*pu1[pu1.state==j][str(j)].values[0] +
129         model.q2[j]*pu2[pu2.state==j][str(j)].values[0] +
130         model.q1[j-N]*pu1[pu1.state==j-N][str(j)].values[0] +
131         model.q2[j-N]*pu2[pu2.state==j-N][str(j)].values[0] +
132         model.q1[j-1]*pu1[pu1.state==j-1][str(j)].values[0] +
133         model.q2[j-1]*pu1[pu1.state==j-1][str(j)].values[0] +
134         model.q2[j+N]*pu2[pu2.state==j+N][str(j)].values[0]) == 0)
135
136 model.p_rb_sumConstraint = Constraint(rb, rule=p_rb_sum_constraint_rule)
137
138 # Sum contraint center states
139 def p_mm_sum_constraint_rule(model, j):
140     return ((model.q1[j] + model.q2[j]) - (
141         model.q1[j]*pu1[pu1.state==j][str(j)].values[0] +
142         model.q2[j]*pu2[pu2.state==j][str(j)].values[0] +
143         model.q1[j-1]*pu1[pu1.state==j-1][str(j)].values[0] +
144         model.q2[j-1]*pu1[pu1.state==j-1][str(j)].values[0] +
145         model.q1[j+1]*pu1[pu1.state==j+1][str(j)].values[0] +
146         model.q2[j+N]*pu2[pu2.state==j+N][str(j)].values[0] +
147         model.q1[j-N]*pu2[pu2.state==j-N][str(j)].values[0] +
148         model.q2[j-N]*pu2[pu2.state==j-N][str(j)].values[0]) == 0)
149
150 model.p_mm_sumConstraint = Constraint(mm, rule=p_mm_sum_constraint_rule)
151
152 # Sum constraint for state (0N) (bottom left)
153 def p_0N_sum_constraint_rule(model):
154     return ((model.q1[N**2-N+1] + model.q2[N**2-N+1]) - (
155         model.q1[N**2-N+1]*pu1[pu1.state==N**2-N+1][str(N**2-N+1)].values[0] +
```

```
156         model.q2[N**2-N+1]*pu2[pu2.state==N**2-N+1][str(N**2-N+1)].values[0] +
157     model.q1[N**2-N+1-N]*pu1[pu1.state==N**2-N+1-N][str(N**2-N+1)].values[0] +
158     model.q2[N**2-N+1-N]*pu2[pu2.state==N**2-N+1-N][str(N**2-N+1)].values[0] +
159     model.q1[N**2-N+1+1]*pu1[pu1.state==N**2-N+1+1][str(N**2-N+1)].values[0] )
160     == 0)
161
162 model.p_0N_sumConstraint = Constraint(rule=p_0N_sum_constraint_rule)
163
164 # Sum constraint for state (NN) (bottom right)
165 def p_NN_sum_constraint_rule(model):
166     return ((model.q1[N**2] + model.q2[N**2]) - (
167         model.q1[N**2]*pu1[pu1.state==N**2][str(N**2)].values[0] +
168         model.q2[N**2]*pu2[pu2.state==N**2][str(N**2)].values[0] +
169     model.q1[N**2-N]*pu1[pu1.state==N**2-N][str(N**2)].values[0] +
170     model.q2[N**2-N]*pu2[pu2.state==N**2-N][str(N**2)].values[0] +
171     model.q1[N**2-1]*pu1[pu1.state==N**2-1][str(N**2)].values[0] +
172     model.q2[N**2-1]*pu2[pu2.state==N**2-1][str(N**2)].values[0] )
173     == 0)
174
175 model.p_NN_sumConstraint = Constraint(rule=p_NN_sum_constraint_rule)
176
177 # Sum contraint for bottom border states
178 def p_bb_sum_constraint_rule(model, j):
179     return ((model.q1[j] + model.q2[j]) - (
180         model.q1[j]*pu1[pu1.state==j][str(j)].values[0] +
181         model.q2[j]*pu2[pu2.state==j][str(j)].values[0] +
182         model.q1[j-N]*pu1[pu1.state==j-N][str(j)].values[0] +
183         model.q2[j-N]*pu2[pu2.state==j-N][str(j)].values[0] +
184         model.q1[j-1]*pu1[pu1.state==j-1][str(j)].values[0] +
185         model.q2[j-1]*pu2[pu2.state==j-1][str(j)].values[0] +
186         model.q1[j+1]*pu1[pu1.state==j+1][str(j)].values[0]) == 0)
187
188 model.p_bb_sumConstraint = Constraint(bb, rule=p_bb_sum_constraint_rule)
189
190 # Storing results in a csv file
191 # SOURCE (adapted greatly though)
192 def pyomo_postprocess(options=None, instance=None,
193                       results=None):
194
195     # Collect data
196     vars = set()
197     data = {}
198     f = {}
199     for i in range(len(results.solution)):
200         data[i] = {}
201         for var in results.solution[i].variable:
202             vars.add(var)
203             data[i][var] = \
204                 results.solution[i].variable[var]['Value']
```

```
205         f[i] = results.solution[i].objective['OBJ']['Value']
206     #
207     # Write a CSV file, one row per solution.
208     # First column is function value, remaining columns
209     # are values of non-zero variables
210     #
211     rows = []
212     vars = list(vars)
213     vars.sort()
214     rows.append(['OBJ']+vars)
215     for i in range(len (results.solution)):
216         row = [f[i]]
217         for var in vars:
218             row.append( data[i].get(var,None)  )
219         rows.append(row)
220
221     # Rewrite rows to a usuable format for MATLAB
222     # Remove objective:
223     del rows[0][0]
224     del rows[1][0]
225
226     # Remove q's and brackets
227     rows_mat = rows
228     for i in range(0,2*N**2):
229         oldstr = rows[0][i]
230         end = len(oldstr)
231         newstr = oldstr[3:end-1]
232         rows_mat[0][i] = rows[0][i].replace(oldstr,newstr)
233
234     print "Creating results file results.csv"
235     with open("results.csv", "wb") as f:
236         writer = csv.writer(f, delimiter = ',' )
237         writer.writerows(rows_mat)
```

B.4 MATLAB script: MDP Policy Iteration Algorithm

```
1  %% MDP_policy_iteration.m | Han Raaijmakers | Oct 2016
2
3  clc; clear all; close all;
4  disp('Running the MDP policy iteration solver, with idling.');
5  fprintf('\n')
6
7  %% PARAMETERS
8  disp('Setting up model...');
9  fprintf('\n')
10
11 % Program parameters
12 % Maximal buffer size (state space truncation)
13 N = 19;
14 maxIter  = 20; % Maximum number of iterations
15
16 % System parameters
17 % Dependent arrival rates [lots/hour]
```

```
18 Lambda = [8   8;
19           8   8];
20
21 % Arrival rates when server is off
22 Lambda0(1) = max(Lambda(1,1),Lambda(1,2));
23 Lambda0(2) = max(Lambda(2,1),Lambda(2,2));
24
25 % Processing rates [lots/hour]
26 Mu(1) = 17;
27 Mu(2) = 17;
28
29 % Holding costs [dollars/lot/hour]
30 C(1) = 10;
31 C(2) = 12;
32
33 % Stability check
34 m = inv(diag(Mu));
35 M = Lambda*m;
36 EIG = abs(eig(M));
37 if max(EIG) >= 1
38     msg = ['This choice of parameters does not guarantee stability,'...
39         ' please choose different parameters.'];
40     error(msg)
41 end
42
43 % Print theoretical results
44 max_EIG = max(EIG);
45 disp(['The traffic intensity, or spectral density of the system is: ' ...
46       num2str(max_EIG)])
47 fprintf('\n')
48
49 muC_ratio = [C(1)*Mu(1);C(2)*Mu(2)];
50 Order = [1;2];
51 Order = [Order muC_ratio];
52 TO = flipud(sortrows(Order,2));
53 disp(['First product according to c*mu rule is:  ' num2str(TO(1,1)) ...
54      '     with c*mu = ' num2str(TO(1,2))])
55 disp(['Second product according to c*mu rule is: ' num2str(TO(2,1)) ...
56      '     with c*mu = ' num2str(TO(2,2)) ])
57 fprintf('\n')
58
59 ratio = TO(2,2) / TO(1,2);
60 disp(['Ratio between c*mu for number 2 and 1 is: ' num2str(ratio) ])
61 fprintf('\n')
62
63 %% CREATE MODEL STRUCTURE
64
65 Qcomb = (N+1)^2; % Number of states (queue length combinations)
66 pos_a = (N+1)*N; % n.o. possible arrivals
67
68 % Call to problem setup function
69 [p_d_u1,p_d_u2,g_d,g] = MDP_problem_setup(N,Lambda0,Lambda,Mu,C);
70
71 %% POLICY ITERATION ALGORITHM
72 fprintf('Starting policy iteration... \n');
73
74 % Initialize program variables
75 n = Qcomb; % Number of states
76 P = {p_d_u1, p_d_u2}; % Probability matrices
77
78 k = 1; % Iteration number. (0 = 1 in MATLAB)
79 mu = cell(1,maxIter); % Cell to store mu's.
80 h_mu = cell(1,maxIter);  % Cell to store J_mu's
81 O_mu = zeros(1,maxIter); % Array to store average costs
82
83 g_mu = zeros(n,1); % Cost vector g_mu (for current controls)
84 Th_muBE = zeros(2,2); % TJ_mu( ... , .... ) Matrix to store B.E. outcomes
85 mu_new = zeros(n,1); % Array to store new u
86 h_mu_new = zeros(n,1); % Array to store TJmu (new Jmu)
87 e = ones(n,1); % Vector with ones
```

```
88
89  h = sym('h',[n 1]); % Symbolic vector with h's
90  syms O;             % Symbolic variable for Omega
91
92  t = cputime;
93  % STEP 1: Initialization
94  fprintf('Step 1: Initialization... \n');
95
96  % Choose first policy according to cmu rule
97  mu0 = T0(1,1)*ones(n,1); % Initial control
98  % If one product is 0 it always optimal to service the other
99  bb = [1:N+1]; % State numbers of bottom border (Q2 = 0)
100 lb = []; % State numbers of left border (Q1 = 0)
101 for i = 2:Qcomb
102     if mod(i,N+1) == 1
103         lb = [lb i];
104     end
105 end
106 mu0(bb) = 1;
107 mu0(lb) = 2;
108 mu{1} = mu0; % Store in control list
109
110 % Take t=1 as a reference state and set h_mu_0(1) = 0;
111 h_mu1 = 0;
112 h(1) = h_mu1;
113
114 while true
115     % STEP 2: Policy Evaluation
116     fprintf('Step 2 (%d): Policy Evaluation... \n',k)
117
118     % Create transition probability matrix P_mu
119     P_mu = zeros(n,n);
120     for i = 1:n
121         for j = 1:n
122             P_mu(i,j) = P{mu{k}(i)}(i,j);
123         end
124     end
125
126     % Create cost vector g_mu
127     for i = 1:n
128         g_mu(i,1) = g_d(i,mu{k}(i));
129     end
130
131     eqn_sys = O*e + h == g_mu + P_mu*h; % System of equations
132     s = solve(eqn_sys); % Solve system of equaiotns
133     sol = struct2cell(s); % Change to usuable format.
134
135     Omu = sol{1};
136     O_mu(k) = Omu;  % Store average cost
137
138     hmu = zeros(n,1);
139     hmu(1) = h_mu1;
140     for i = 2:n
141         hmu(i) = sol{i};
142     end
143
144     h_mu{k} = hmu; % Store h vector
145
146     % STEP 3: Policy Improvement
147     fprintf('Step 3 (%d): Policy Improvement... \n',k)
148
149     for i = 1:n
150
151         % Compute summation parts of Bellman's Equation
152         sum_u1 = 0;
153         sum_u2 = 0;
154         for j = 1:n
155             sum_u1 = sum_u1 + p_d_u1(i,j)*hmu(j);
156             sum_u2 = sum_u2 + p_d_u2(i,j)*hmu(j);
157         end
```

```
158
159         % Compute Bellman's Equation
160         Th_muBE(i,1) = g_d(i,1) + sum_u1; %u1
161         Th_muBE(i,2) = g_d(i,2) + sum_u2; %u2
162
163         % Find minimizing controls
164         [m,id] = min(Th_muBE(i,:));
165         mu_new(i) = id;
166         h_mu_new(i) = m;
167     end
168     % Check if optimal policy is found: Jmu(k) = TJmu(k)
169     if mu{k} == mu_new
170         mu_star = mu_new;
171         h_star = hmu;
172         O_star = Omu;
173         break
174     end
175
176     % If not, return to step 2 and use current mu.
177     mu{k+1} = mu_new;
178
179     % Go to second iterion
180     k = k + 1;
181
182     % Show error if iteration limit is reached
183     if k == maxIter+1
184         msg = 'Iteration limit reached. Optimal policy is not found.';
185         error(msg);
186     end
187 end
188
189 disp('Policy Iteration succesfull.')
190 Stime = cputime-t;
191 fprintf('Optimal policy was found in %d iterations and %2.1f seconds \n'...
192          ,k,Stime)
193
194 %% ANALYZING RESULTS
195
196 fprintf('The average total cost Omega is: %5.4f \n',double(O_star))
197
198 % Create control matrices with NaN, to make errors visible
199 pi = NaN(Qcomb,2);
200
201 % Optimal control
202 for i = 1:Qcomb
203     if mu_star(i) == 1
204         pi(i,1) = 1;
205         pi(i,2) = 0;
206     elseif mu_star(i) == 2
207         pi(i,1) = 0;
208         pi(i,2) = 1;
209     end
210 end
211
212 C1 = zeros(N+1,N+1);
213 C2 = zeros(N+1,N+1);
214 % Transform to grid setup
215 for i = 1:N+1
216    C1(i,:) = pi(1+(i-1)*(N+1):i*(N+1),1);
217    C2(i,:) = pi(1+(i-1)*(N+1):i*(N+1),2);
218 end
219
220 %% PLOTTING RESULTS
221
222 % Control routing rule
223 figure
224 axis([0 N+1 0 N+1]);
225 xlabel('Queue 1');
226 ylabel('Queue 2');
227 hold on
```

```
grid on
% Control 1
for i = 1:N+1
    for j = 1:N+1
        if C1(i,j) >= 0.99
            plot(j,i,'sb')
        elseif isnan(C1(i,j)) % Error value due to truncation
            plot(j,i,'sy')
        end
    end
end
% Control 2
for i = 1:N+1
    for j = 1:N+1
        if C2(i,j) >= 0.99
            plot(j,i,'sr')
        elseif isnan(C2(i,j))
            plot(j,i,'sy')
        end
    end
end
legend('Service type 1')
title('Policy Iteration')
shg

disp('Program finished.');
```

B.5 MATLAB script: MDP Modified Policy Iteration

```
%% MDP_policy_iteration.m | Han Raaijmakers | Oct 2016

clear all; %clc; %close all;
disp('Running the MDP modified policy iteration solver, with idling');
fprintf('\n')

%% PARAMETERS
disp('Setting up model...');
fprintf('\n')

% Program parameters
% Maximal buffer size (state space truncation)
N = 39;
maxIter  = 20; % Maximum number of iterations
VI_max = 1500; % Maximum relative value iterations to find h_mu
VI_tol = 1e-15; % Tolerance for stopping value iterations to find h_mu

% System parameters
% Dependent arrival rates [lots/hour]
Lambda = [20   20;
          10   20];

% Arrival rates when server is off
Lambda0(1) = max(Lambda(1,1),Lambda(1,2));
Lambda0(2) = max(Lambda(2,1),Lambda(2,2));

% Processing rates [lots/hour]
Mu(1) = 41;
Mu(2) = 41;

% Holding costs [dollars/lot/hour]
C(1) = 10;
C(2) = 12;

% Stability check | SEE PAPER .....
m = inv(diag(Mu));
M = Lambda*m;
EIG = abs(eig(M));
if max(EIG) >= 1
    msg = ['This choice of parameters does not guarantee stability,'...
```

```
        ' please choose different parameters.'];
    error(msg)
end

% Print theoretical results
max_EIG = max(EIG);
disp(['The traffic intensity, or spectral density of the system is: ' ...
      num2str(max_EIG)])
fprintf('\n')

muC_ratio = [C(1)*Mu(1);C(2)*Mu(2)];
Order = [1;2];
Order = [Order muC_ratio];
TO = flipud(sortrows(Order,2));
disp(['First product according to c*mu rule is:  ' num2str(TO(1,1)) ...
     '     with c*mu = ' num2str(TO(1,2))])
disp(['Second product according to c*mu rule is: ' num2str(TO(2,1)) ...
     '     with c*mu = ' num2str(TO(2,2)) ])
fprintf('\n')

ratio = TO(2,2) / TO(1,2);
disp(['Ratio between c*mu for number 2 and 1 is: ' num2str(ratio) ])
fprintf('\n')

%% CREATE MODEL STRUCTURE

Qcomb = (N+1)^2; % Number of states (queue length combinations)
pos_a = (N+1)*N; % n.o. possible arrivals

% Call to problem setup function
[p_d_u1,p_d_u2,g_d] = MDP_problem_setup(N,Lambda0,Lambda,Mu,C);

%% POLICY ITERATION ALGORITHM
fprintf('Starting policy iteration... \n');

% Initialize program variables
n = Qcomb; % Number of states
P = {p_d_u1, p_d_u2}; % Probability matrices

k = 1; % Iteration number. (0 = 1 in MATLAB)
mu = cell(1,maxIter); % Cell to store mu's.
h_mu = cell(1,maxIter);  % Cell to store J_mu's
O_mu = zeros(1,maxIter); % Array to store average costs

g_mu = zeros(n,1); % Cost vector g_mu (for current controls)
Th_muBE = zeros(2,2); % TJ_mu( ... , .... ) Matrix to store B.E. outcomes
mu_new = zeros(n,1); % Array to store new u
h_mu_new = zeros(n,1); % Array to store TJmu (new Jmu)
e = ones(n,1); % Vector with ones

h_vi = zeros(Qcomb,VI_max);  % Matrix to store VI steps
h_vi_new = zeros(n,1); % Array h's for VI
Th_vi_old = zeros(n,1); % Array to store Th's
Th = zeros(2,2); % Matrix to store TH for VI

t = cputime;
% STEP 1: Initialization
fprintf('Step 1: Initialization... \n');

% Choose first policy according to cmu rule
mu0 = TO(1,1)*ones(n,1); % Initial control
% If one product is 0 it always optimal to service the other
bb = [1:N+1]; % State numbers of bottom border (Q2 = 0)
lb = []; % State numbers of left border (Q1 = 0)
for i = 2:Qcomb
    if mod(i,N+1) == 1
        lb = [lb i];
    end
end
mu0(bb) = 1;
```

```
mu0(lb) = 2;
mu{1} = mu0; % Store in control list

% Take t=1 as a reference state and set h_mu_0(1) = 0. Initialize the h
% vector for the relative value iteration to 1:Qcomb (except for state 1)
h_vi_int = [1:Qcomb]';
h_vi_int(1) = 0;

while true
    % STEP 2: Policy Evaluation
    fprintf('Step 2 (%d): Policy Evaluation... \n',k)

    % Create transition probability matrix P_mu
    P_mu = zeros(n,n);
    for i = 1:n
        for j = 1:n
            P_mu(i,j) = P{mu{k}(i)}(i,j);
        end
    end

    % Create cost vector g_mu
    for i = 1:n
        g_mu(i,1) = g_d(i,mu{k}(i));
    end

    % Use relative value iteration instead of solving the system
    % of equations
    h_vi(:,1) = h_vi_int;

    for l = 2:VI_max+1;
        for i = 1:n

            % Compute summation parts of Bellman's Equation
            sum_u1 = 0;
            sum_u2 = 0;
            for j = 1:n
                sum_u1 = sum_u1 + p_d_u1(i,j)*h_vi(j,l-1);
                sum_u2 = sum_u2 + p_d_u2(i,j)*h_vi(j,l-1);
            end

            % Compute Bellman's Equation
            Th(i,1) = g_d(i,1) + sum_u1; %u1
            Th(i,2) = g_d(i,2) + sum_u2; %u2

            % Find minimizing controls
            [m,id] = min(Th(i,:));
            Th_vi_old(i) = m;
        end
        % Use the minimum values to apply Th another time
        h_vi_new = Th_vi_old - Th_vi_old(1)*e;

        h_vi(:,l) = h_vi_new;

        % Stop iterating if tolerance is met
        VI_diff = max(abs(h_vi(:,l) - h_vi(:,l-1)));
        if VI_diff <= VI_tol
            break
        end
    end

    hmu = h_vi(:,end);
    h_mu{k} = hmu; % Store h vector

    % Compute Omega using hmu
    Omu = (g_mu + P_mu*hmu - hmu);
    Omu = Omu(1);
    O_mu(k) = Omu;

    % STEP 3: Policy Improvement
    fprintf('Step 3 (%d): Policy Improvement... \n',k)
```

```
181
182     for i = 1:n
183
184         % Compute summation parts of Bellman's Equation
185         sum_u1 = 0;
186         sum_u2 = 0;
187         for j = 1:n
188             sum_u1 = sum_u1 + p_d_u1(i,j)*hmu(j);
189             sum_u2 = sum_u2 + p_d_u2(i,j)*hmu(j);
190         end
191
192         % Compute Bellman's Equation
193         Th_muBE(i,1) = g_d(i,1) + sum_u1; %u1
194         Th_muBE(i,2) = g_d(i,2) + sum_u2; %u2
195
196         % Find minimizing controls
197         [m,id] = min(Th_muBE(i,:));
198         mu_new(i) = id;
199         h_mu_new(i) = m;
200     end
201     % Check if optimal policy is found: Jmu(k) = TJmu(k)
202     if mu{k} == mu_new
203         mu_star = mu_new;
204         h_star = hmu;
205         O_star = Omu;
206         break
207     end
208
209     % If not, return to step 2 and use current mu.
210     mu{k+1} = mu_new;
211
212     % Go to second iterion
213     k = k + 1;
214
215     % Show error if iteration limit is reached
216     if k == maxIter+1
217         msg = 'Iteration limit reached. Optimal policy is not found.';
218         error(msg);
219     end
220 end
221
222 disp('Policy Iteration succesfull.')
223 Stime = cputime-t;
224 fprintf('Optimal policy was found in %d iterations and %2.1f seconds \n'...
225          ,k,Stime)
226
227 %% ANALYZING RESULTS
228
229 fprintf('The average total cost Omega is: %5.4f \n',double(O_star))
230
231 % Create control matrices with NaN, to make errors visible
232 pi = NaN(Qcomb,2);
233
234 % Optimal control
235 for i = 1:Qcomb
236     if mu_star(i) == 1
237         pi(i,1) = 1;
238         pi(i,2) = 0;
239     elseif mu_star(i) == 2
240         pi(i,1) = 0;
241         pi(i,2) = 1;
242     end
243 end
244
245 C1 = zeros(N+1,N+1);
246 C2 = zeros(N+1,N+1);
247 % Transform to grid setup
248 for i = 1:N+1
249    C1(i,:) = pi(1+(i-1)*(N+1):i*(N+1),1);
250    C2(i,:) = pi(1+(i-1)*(N+1):i*(N+1),2);
```

```
251 end
252
253 %% PLOTTING RESULTS
254
255 % Control routing rule
256 figure
257 axis([0 N+1 0 N+1]);
258 xlabel('Queue 1');
259 ylabel('Queue 2');
260 hold on
261 grid on
262 % Control 1
263 for i = 1:N+1
264     for j = 1:N+1
265         if C1(i,j) >= 0.99
266             plot(j,i,'sb')
267         elseif isnan(C1(i,j)) % Error value due to truncation
268             plot(j,i,'sy')
269         end
270     end
271 end
272 % Control 2
273 for i = 1:N+1
274     for j = 1:N+1
275         if C2(i,j) >= 0.99
276             plot(j,i,'sr')
277         elseif isnan(C2(i,j))
278             plot(j,i,'sy')
279         end
280     end
281 end
282 legend('Service type 1')
283 shg
284
285 disp('Program finished.');
```

C MATLAB script: Fluid model optimization problem

```
% Fluid_LP.m | Han Raaijmakers | Sep 2016
% Approximates the problem as a fluid model and solves it using Linear
% Progamming

clear all; clc; close all;
disp('Running the fluid model approximation.');
fprintf('\n')

%% PARAMETERS

% Program
T = 25; % time interval
N = 500; % determines number of increments

% System parameters
% Initial queue lengths
Q_0(1) = 30;    % Product 1
Q_0(2) = 40;    % Product 2

% Dependent arrival rates [lots/hour]
Lambda = [45    45;
           45    45];

% Arrival rates when server is off
Lambda0(1) = max(Lambda(1,1),Lambda(1,2));
Lambda0(2) = max(Lambda(2,1),Lambda(2,2));

% Processing rates [lots/hour]
Mu(1) = 100;
Mu(2) = 100;

% Holding costs [dollars/lot/hour]
C(1) = 10;
C(2) = 12;

% Stability check | SEE PAPER .....
m = inv(diag(Mu));
M = Lambda*m;
EIG = abs(eig(M));
if max(EIG) >= 1
    msg = ['This choice of parameters does not guarantee stability,'...
        ' please choose different parameters.'];
    error(msg)
end

% Print theoretical results
max_EIG = max(EIG);
disp(['The traffic intensity, or spectral density of the system is: ' ...
      num2str(max_EIG)])
fprintf('\n')

muC_ratio = [C(1)*Mu(1);C(2)*Mu(2)];
Order = [1;2];
Order = [Order muC_ratio];
TO = flipud(sortrows(Order,2));
disp(['First product according to c*mu rule is:  ' num2str(TO(1,1)) ...
     '     with c*mu = ' num2str(TO(1,2))])
disp(['Second product according to c*mu rule is: ' num2str(TO(2,1)) ...
     '     with c*mu = ' num2str(TO(2,2)) ])
fprintf('\n')

ratio = TO(2,2) / TO(1,2);
disp(['Ratio between c*mu for number 2 and 1 is: ' num2str(ratio) ])
fprintf('\n')

% Final queue lengths
Q1f = 0;
```

```
68  Q2f = 0;
69
70  %% FORMULATE LINEAR PROGRAM
71
72  % Time increment
73  dt = T/N;
74
75  % Linear objective function
76  f = zeros(1,2*N + 2*(N-1));
77  for n = 1:N-1
78      f(2*N+n) = C(1)*dt;
79      f(2*N+(N-1)+n) = C(2)*dt;
80  end
81
82  % Linear equality constraints
83  Aeq = zeros(N*2,2*N + 2*(N-1));
84  beq = zeros(N*2,1);
85
86  % n = 0
87  % Equality 1:
88  Aeq(1,1) = (Mu(1)-Lambda(1,1))*dt;
89  Aeq(1,N+1) = -Lambda(1,2)*dt;
90  Aeq(1,2*N+1) = 1;
91  beq(1) = Q_0(1);
92
93  % Equality 2:
94  Aeq(2,1) = (-Lambda(2,1))*dt;
95  Aeq(2,N+1) = (Mu(2)-Lambda(2,2))*dt;
96  Aeq(2,2*N+(N-1)+1) = 1;
97  beq(2) = Q_0(2);
98
99  % n = N
100 % Equality 1
101 Aeq(3,N) = (Mu(1)-Lambda(1,1))*dt;
102 Aeq(3,2*N) = (-Lambda(1,2))*dt;
103 Aeq(3,2*N+(N-1)) = -1;
104 beq(3) = Q1f;
105
106 % Equality 2
107 Aeq(4,N) = (-Lambda(1,2))*dt;
108 Aeq(4,2*N) = (Mu(2)-Lambda(2,1))*dt;
109 Aeq(4,2*N+2*(N-1)) = -1;
110 beq(4) = Q2f;
111
112 % n = 1 ... N - 1
113 j = 4;
114 for n = 1:N-2
115     j = j+1;
116     % Equality 1:
117     Aeq(j,n) = (Mu(1)-Lambda(1,1))*dt;
118     Aeq(j,N+n) = (-Lambda(1,2))*dt;
119     Aeq(j,2*N+n) = -1;
120     Aeq(j,2*N+n+1) = 1;
121     beq(j) = 0;
122
123     j = j+1;
124     % Equality 2:
125     Aeq(j,n) = (-Lambda(2,1))*dt;
126     Aeq(j,N+n) = (Mu(2)-Lambda(2,2))*dt;
127     Aeq(j,2*N+(N-1)+n) = -1;
128     Aeq(j,2*N+(N-1)+n+1) = 1;
129     beq(j) = 0;
130 end
131
132 % Linear inequality constraints
133 A = zeros(N-1,2*N+2*(N-1));
134 b = zeros(N-1,1);
135
136 for n = 1:N-1
137    A(n,n) = 1; A(n,N+n) = 1; b(n) = 1;
```

```
138 end
139
140 % Lower and upper bounds
141 lb = zeros(1,2*N + 2*(N-1));
142 ub = [];
143
144 %% EXECUTE LINEAR PROGRAM
145
146 x0 = [];
147 options = []; optimset('Display','Iter');
148 [dqp, fval, exitflag, output, lambda] = ...
149     linprog(f,A,b,Aeq,beq,lb,ub,x0,options);
150
151 %% PLOTTING RESULTS
152 timeQ = 0:dt:T;
153 timeC = 0:dt:T-dt;
154
155 control(1,:) = dqp(1:N)';
156 control(2,:) = dqp(N+1:2*N)';
157
158 Q = zeros(2,N+1);
159 Q(1,1) = Q_0(1);
160 Q(1,2:N) = dqp(2*N+1:2*N+(N-1));
161 Q(1,N+1) = Q1f;
162 Q(2,1) = Q_0(2);
163 Q(2,2:N) = dqp(2*N+(N-1)+1:2*N+2*(N-1));
164 Q(2,N+1) = Q2f;
165
166 % % Plot controls
167 figure
168 plot(timeC,control(1,:));
169 hold on
170 grid on
171 plot(timeC,control(2,:));
172 xlabel('Time [hours]')
173 ylabel('Fraction of server time [-]')
174 legend('u_1','u_2')
175 axis([0 T-1 -0.1 1.1])
176
177 % Plot Queues
178 figure
179 plot(timeQ,Q(1,:));
180 hold on
181 grid on
182 plot(timeQ,Q(2,:));
183 xlabel('Time [hours]')
184 ylabel('Queue lengths [lots]')
185 legend('q_1','q_2')
186 xlim([0 T-1])
187 shg
```

D Simulation

D.1 MATLAB script: Simulation

```
% Simulation.m | Han Raaijmakers | Oct 2016
% Simulates arrivals according to Poisson process and exponential service
% times. Simultanously runs two paralles simulations, for a system
% following the cmu policy and a system following the reversed cmu policy.
% Both systems use the same exponential arrivals and service times.
% pol1 = cmu policy
% pol2 = reversed cmu policy

clear all; clc; close all
disp('Running the simulation.');
fprintf('\n')

%% PARAMETERS

% Simulation
simrep = 10; % Times to repeat simulation
samp = 0.001;     % Simulation time sample length [hour]
simlen = 1000;    % Simulation length [hours]
time = 0:samp:simlen; % Time vector
tlen = simlen/samp+1; % Time vector length

% System parameters
% Initial queue lengths
Q_0(1) = 1;    % Product 1
Q_0(2) = 1;    % Product 2

% Dependent arrival rates [lots/hour]
Lambda = [20    20;
          20    20];

% Arrival rates when server is off
Lambda0(1) = max(Lambda(1,1),Lambda(1,2));
Lambda0(2) = max(Lambda(2,1),Lambda(2,2));

% Processing rates [lots/hour]
Mu(1) = 50;
Mu(2) = 50;

% Holding costs [dollars/lot/hour]
C(1) = 10;
C(2) = 12;

% Stability check
m = inv(diag(Mu));
M = Lambda*m;
EIG = abs(eig(M));
if max(EIG) >= 1
    msg = ['This choice of parameters does not guarantee stability,'...
        ' please choose different parameters.'];
    error(msg);
end

% Print theoretical results
max_EIG = max(EIG);
disp(['The traffic intensity, or spectral density of the system is: ' ...
      num2str(max_EIG)])
fprintf('\n')

muC_ratio = [C(1)*Mu(1);C(2)*Mu(2)];
Order = [1;2];
Order = [Order muC_ratio];
TO = flipud(sortrows(Order,2));
disp(['First product according to c*mu rule is:  ' num2str(TO(1,1)) ...
     '     with c*mu = ' num2str(TO(1,2))])
```

```
65 disp(['Second product according to c*mu rule is: ' num2str(TO(2,1)) ...
66      '     with c*mu = ' num2str(TO(2,2)) ])
67 fprintf('\n')
68
69 ratio = TO(2,2) / TO(1,2);
70 disp(['Ratio between c*mu for number 2 and 1 is: ' num2str(ratio) ])
71 fprintf('\n')
72
73 %% SIMULATION
74
75 L = Lambda.*samp; % Scale lambda to simulation parameters
76 L0 = Lambda0*samp;
77 Mu_inv = 1./Mu; % Inverse of Mu
78
79 arr = zeros(2,tlen); % Matrix to store arrivals
80 Q_pol1 = zeros(2,tlen);   % Matrix representing queue length for cmu
81 Q_pol2 = zeros(2,tlen);   % Matrix representing queue length for cmu rev
82 Q_pol1(:,1) = Q_0(:);     % Initialize intial queue lengths
83 Q_pol2(:,1) = Q_0(:);     % Initialize intial queue lengths
84
85 Q_pol1_arr = zeros(2,tlen);
86 Q_pol2_arr = zeros(2,tlen);
87 Q_pol1_proc_times = [];
88 Q_pol2_proc_times = [];
89
90 % Product server statusses
91 server_pol1 = 3; % 1 = product #1, 2 = product #3, 3 = idle
92 server_pol2 = 3; % 1 = product #1, 2 = product #3, 3 = idle
93 To = TO(:,1);
94
95 TC_pol1 = zeros(1,simrep);
96 TC_pol2 = zeros(1,simrep);
97 AC_pol1 = zeros(1,simrep);
98 AC_pol2 = zeros(1,simrep);
99
100 cmu = To(1); % Priority product
101 rev = To(2); % Non Priority products
102 disp('Simulating...')
103 fprintf('\n')
104 for r = 1:simrep
105
106     fprintf('Simulation repetition number %d ... \n',r)
107
108     c = 2; % Counter
109     serv_t_pol1 = 0; % Current production time remaining pol1 simulation
110     serv_t_pol2 = 0; % Current production time remaining pol2 simulation
111
112     % Start simulation
113     for t = 0:samp:simlen-samp
114
115         if serv_t_pol1 <= 0  % No product being serviced in pol1 sim
116             server_pol1 = 3;
117             serv_t_pol1 = 0;
118         end
119
120         if serv_t_pol2 <= 0 % No product being serviced in pol2 sim
121             server_pol2 = 3;
122             serv_t_pol2 = 0;
123         end
124
125         % Simulate queue lengths pol1 simulation queue
126         if server_pol1 == 1
127             Q_pol1(1,c) = Q_pol1(1,c-1) + poissrnd(L(1,1));
128             Q_pol1(2,c) = Q_pol1(2,c-1) + poissrnd(L(2,1));
129         elseif server_pol1 == 2
130             Q_pol1(1,c) = Q_pol1(1,c-1) + poissrnd(L(1,2));
131             Q_pol1(2,c) = Q_pol1(2,c-1) + poissrnd(L(2,2));
132         else % server_cmu == 0
133             Q_pol1(1,c) = Q_pol1(1,c-1) + poissrnd(L0(1));
134             Q_pol1(2,c) = Q_pol1(2,c-1) + poissrnd(L0(2));
```

```
135         end
136         Q_pol1_arr(:,c) = Q_pol1(:,c) - Q_pol1(:,c-1);
137
138         % Simulate queue lengths pol2 simulation queue
139         if server_pol2 == 1
140             Q_pol2(1,c) = Q_pol2(1,c-1) + poissrnd(L(1,1));
141             Q_pol2(2,c) = Q_pol2(2,c-1) + poissrnd(L(2,1));
142         elseif server_pol2 == 2
143             Q_pol2(1,c) = Q_pol2(1,c-1) + poissrnd(L(1,2));
144             Q_pol2(2,c) = Q_pol2(2,c-1) + poissrnd(L(2,2));
145         else % server_rev == 0
146             Q_pol2(1,c) = Q_pol2(1,c-1) + poissrnd(L0(1));
147             Q_pol2(2,c) = Q_pol2(2,c-1) + poissrnd(L0(2));
148         end
149         Q_pol2_arr(:,c) = Q_pol2(:,c) - Q_pol2(:,c-1);
150
151         % Production process for pol1 simulation
152         if server_pol1 == 3 % Server is idle
153             num1 = size(Q_pol1_proc_times,2); % N.o. products so far
154             if Q_pol1(cmu,c) >= 1 % If queue of cmu product is non-empty
155                 % Simulate processing time cmu prod and adapt server time
156                 serv_t_pol1 = exprnd(Mu_inv(cmu));
157                 Q_pol1_proc_times(cmu,num1+1) = serv_t_pol1;
158                 % Decrease qeueu length for cmu product
159                 Q_pol1(cmu,c) = Q_pol1(cmu,c)-1;
160                 % Set server status to cmu producttype in service
161                 server_pol1 = cmu;
162             elseif Q_pol1(rev,c) >= 1 % Otherwise, non-preffered product
163                 % Simulate processing time rev prod and adapt server time
164                 serv_t_pol1 = exprnd(Mu_inv(rev));
165                 Q_pol1_proc_times(rev,num1+1) = serv_t_pol1;
166                 % Decrease qeueu length for rev product
167                 Q_pol1(rev,c) = Q_pol1(rev,c)-1;
168                 % Set server status to rev producttype in service
169                 server_pol1 = rev;
170             end
171         else % Server is busy
172             serv_t_pol1 = serv_t_pol1 - samp;
173         end
174         % Production process for pol1 simulation
175         if server_pol2 == 3 % Server is idle
176             num2 = size(Q_pol2_proc_times,2);
177             if Q_pol2(rev,c) >= 1 % If queue of rev product is non-empty
178                 % Simulate processing time cmu prod and adapt server time
179                 serv_t_pol2 = exprnd(Mu_inv(rev));
180                 Q_pol2_proc_times(rev,num2+1) = serv_t_pol2;
181                 % Decrease qeueu length for rev product
182                 Q_pol2(rev,c) = Q_pol2(rev,c)-1;
183                 % Set server status to rev producttype in service
184                 server_pol2 = rev;
185             elseif Q_pol2(cmu,c) >= 1 % Otherwise, non-preffered product
186                 % Simulate processing time cmu prod and adapt server time
187                 serv_t_pol2 = exprnd(Mu_inv(cmu));
188                 Q_pol2_proc_times(cmu,num2+1) = serv_t_pol2;
189                 % Decrease qeueu length for cmu product
190                 Q_pol2(cmu,c) = Q_pol2(cmu,c)-1;
191                 % Set server status to cmu producttype in service
192                 server_pol2 = cmu;
193             end
194         else % Server is busy
195             serv_t_pol2 = serv_t_pol2 - samp;
196         end
197
198         c = c+1;
199     end
200
201     % Compensate for potential warmup effects
202     wa = 0.1*simlen/samp; % Warmup part
203     Q_pol1(:,1:wa) = [];  % Remove data
204     Q_pol2(:,1:wa) = [];  % Remove data
```

```
205
206     % Compute total costs
207     TC_pol1(r) = sum(Q_pol1(1,:))*(C(1)*samp)+sum(Q_pol1(2,:))*(C(2)*samp);
208     TC_pol2(r) = sum(Q_pol2(1,:))*(C(1)*samp)+sum(Q_pol2(2,:))*(C(2)*samp);
209     % Compute average costs (with warmup compsensation)
210     AC_pol1(r) = TC_pol1(r) / (simlen*0.9);
211     AC_pol2(r) = TC_pol2(r) / (simlen*0.9);
212
213     AC_pol1(r)
214     AC_pol2(r)
215
216 end
217 %% ANALYSING RESULTS
218
219 TC_pol1_tot = mean(TC_pol1);
220 TC_pol2_tot = mean(TC_pol2);
221 AC_pol1_tot = mean(AC_pol1);
222 AC_pol2_tot = mean(AC_pol2);
223
224 disp('Simulation finished.')
```

D.2 Simulation data

Table D.1: *Data of the simulation of Section 8.*

$c\mu$ rule	reversed $c\mu$ rule	$c\mu$ rule	reversed $c\mu$ rule
83.4770	74.6631	70.7400	72.1367
81.6131	75.7797	85.9604	75.8423
69.4295	74.7288	70.9150	76.1848
86.8315	74.4690	82.7101	65.5119
78.5912	75.3627	81.8061	71.8850
76.2929	79.7004	72.8445	65.9974
73.7698	72.1215	72.1818	71.6089
76.5062	69.5098	80.8815	79.1500
79.1054	82.7101	73.6964	76.6413
76.5827	63.9633	84.7544	66.6709
85.8150	68.3516	82.0449	69.1907
78.4075	72.7985	74.3098	71.5728
85.5104	71.6426	83.4139	77.6172
86.1381	72.8920	74.9184	70.3047
81.7674	76.0661	93.1936	69.3288
77.0449	72.3111	74.2567	73.4967
82.1949	64.0299	81.2642	78.6775
83.2562	86.6245	81.1230	72.0988
88.8806	73.5087	91.3310	76.7744
87.6032	73.5429	73.0385	69.7879
76.4996	70.9625	71.6928	70.1336
78.4517	68.0835	79.6479	71.4596
75.9483	64.7376	83.2388	69.7172
81.1621	74.8025	76.9335	75.5910
83.6956	73.6976	70.6536	77.0532